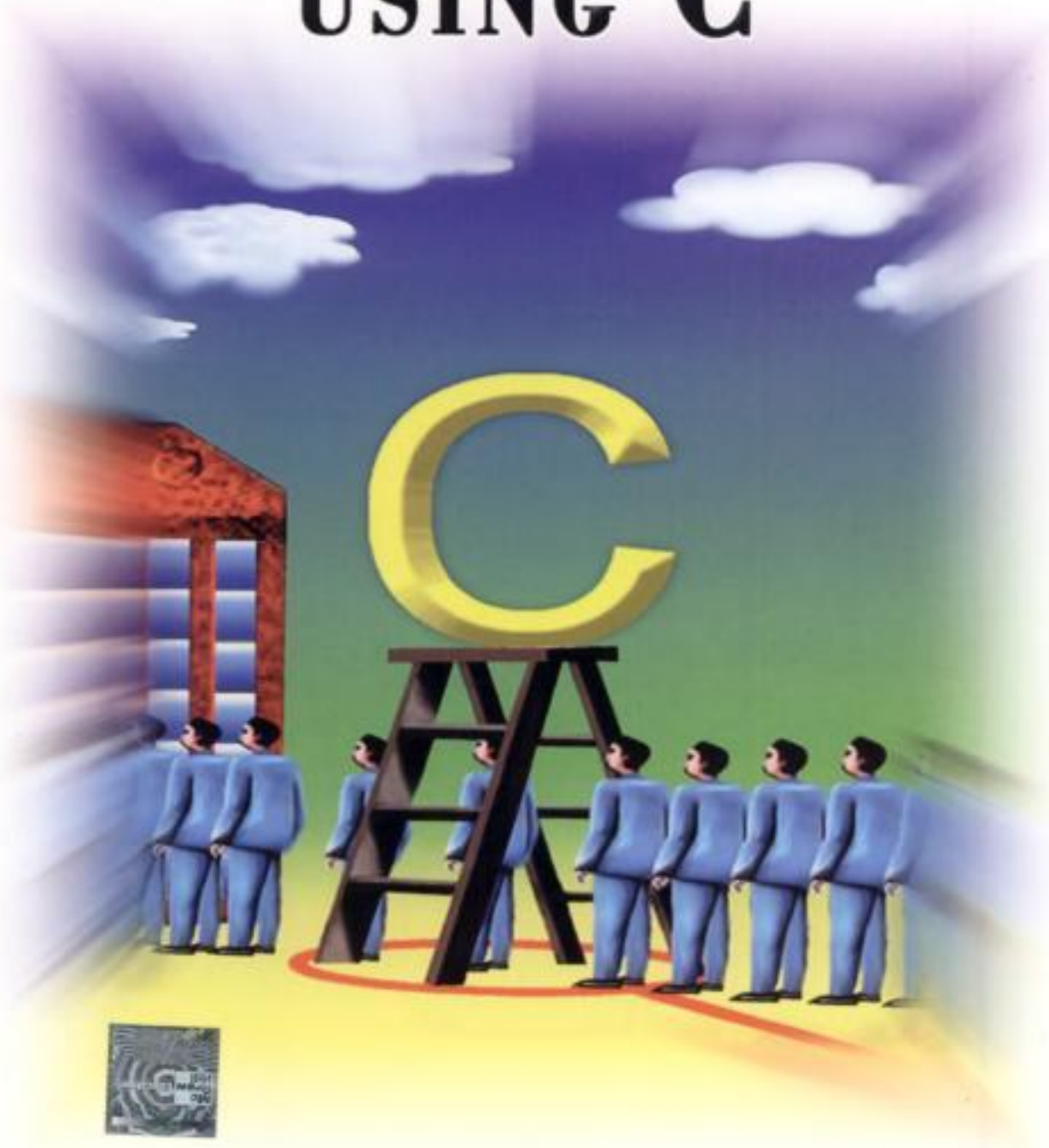
The McGraw-Hill Companies

ace series

DATA STRUCTURES USING C



ISRD Group

Data Structures Using C

Data Structures Using C

Instructional Software Research and Development (ISRD) Group

Lucknow

Tata McGraw-Hill Publishing Company Limited

NEW DELHI

McGraw-Hill Offices

New Delhi New York St Louis San Francisco Auckland Bogotá Caracas
Kuala Lumpur Lisbon London Madrid Mexico City Milan Montreal
San Juan Santiago Singapore Sydney Tokyo Toronto

Tata McGraw-Hill

First reprint 2007
RADYCRXYRZYLC

This edition can be exported from India only by the publishers,
Tata McGraw-Hill Publishing Company Limited.

ISBN 0-07-059102-4

Published by the Tata McGraw-Hill Publishing Company Limited,
7 West Patel Nagar, New Delhi 110 008, typeset in Times New Roman at
Tej Composers, WZ-391, Madipur Village, New Delhi 110 063 and
printed at Pashupati Printers (P) Ltd. 429/16 Gali No. 1, Friends Colony,
Industrial Area, G T Road, Shahdara Delhi 110 095

Cover Printer: SDR

The McGraw-Hill Companies

The ISRD Group

The Instructional Software Research and Development (ISRD) Group has been established by UPTEC Computer Consultancy Ltd. for content development. The Group is committed to develop and produce high quality learning and instructional material to facilitate all round professional development of students. The Group continuously strives to enrich their content repository by constantly researching and developing new learning and instructional material. The content thus developed is tailored to take various forms including text books, teaching aids, lecture notes, lab assignments and exercises, both in print as well as in electronic media (CDs etc.).

Many professionals have helped in creating UPTEC series of 'A' Level text books, and each of their contributions have been valuable. This particular book has taken its final shape with the noteworthy contributions of

Ms. Geetanjali Khanduri She is a Microsoft Certified Systems Engineer with specialization in Messaging. She also has to her credit the Microsoft Certified Trainer certification. Presently she is working as a faculty member with UPTEC Computer Consultancy Ltd.

Mr. Ashish Srivastava. Holds a Masters degree in Computer Science and is a Microsoft Certified Solutions Developer. He is presently working as Systems Analyst in a reputed multi-national company in Bangalore.

The work would not have been possible without the enriching guidance and support of the following key professionals:

Er. Alok Singhal An Electronics and Telecommunication Engineer from Roorkee University (now IIT Roorkee), with a Master's degree in Systems Engineering. Over 30 years of experience in the areas of entertainment electronics, microcomputers and microprocessor-based products. In his last assignment he was General Manager and Head of Communication Division at UPTRON India Ltd and was handling manufacturing and marketing of EPABX, RAX, radio communication, office communication, data communication, and computer products. He is presently holding the position of Vice President at UPTEC.

Prof.K.K. Bhutani Holds his Bachelor's, Master's and Doctoral degree from the University of Allahabad, he has more than 35 years of experience in academics and industry. He has been a professor of Electronics and Head of Computer Centre, University of Allahabad. He has also worked as General Manager (Computers), Computronics India. He is former Chairman, Computer Engineering Board, Institution of Engineers (India). At present, he is associated with UPTEC as a Director.

Er. K.N. Shukla An Electronics Engineer from IIT, Kanpur. He has more than 30 years of experience in electronics and computer industry. Started his career with J. K. Electronics, Kanpur. He has held several managerial positions at UPTRON, HILTRON and other similar companies. He was Director (Technical) at UPTRON India Limited before joining UPTEC as wholetime Director.

Dr. R.K. Jaiswal Holds a Doctoral degree from the University of Lucknow and has more than 16 years of experience in academics. He started his career with Lucknow University as a Lecturer, and has worked under College Science Improvement Programme (COSIP) a Government of India scheme. He has to his credit more than a dozen research papers in national and international journals. He is presently holding the position of Assistant General Manager at UPTEC.

Er. R.K. Lele Holds a bachelor's degree in Electrical Engineering from IIT Kanpur. He has Design and Development experience in embedded software/firmware, system software, computer & telecommunication/

data communication hardware and exposure to real-time software in the area of switching & communication protocols. At managerial position, he has handled various design projects with engineers during all the phases of the product design cycle. He is presently holding the position of General Manager (Systems) at UPTEC.

Prof. S.K. Singh With a bachelor's degree from IIT Kharagpur and master's degree from University of Missouri (USA) he has more than 35 years of professional experience with Bell Systems (USA), IBM, Rashtriya Chemicals & Fertilizers Ltd, National Institute for Training in Industrial Engineering (NITIE), 10 years as Professor of Computers and MIS at Indian Institute of Management (IIM), Lucknow, before joining UPTEC. Prof. Singh has been the Founder Chairman-Post Graduate Programme at IIM, Lucknow. He has also been a consultant to many national/international organizations. UPTEC series of 'A' Level text books have been developed under his active guidance.

Dr. Upendra Kumar An Industrial Engineer, obtained his Ph.D from IIT, Delhi. Has more than 25 years of experience in industry, academics and research. Dr. Kumar has worked with BHEL Corporate Systems Group, Eicher, UPTRON, National Institute for Training in Industrial Engineering (NITIE) and Indian Institute of Management (IIM), Lucknow. Dr. Kumar had conceived, created and established UPTRON-ACL and Computer Consultancy and Services Division at UPTRON earlier, as General Manager and Divisional Incharge. Has been consultant to many national/international organizations and is presently working as the Managing Director of UPTEC Computer Consultancy Ltd.

Ms. Baljeet Kaur She has more than 10 years of experience in academics and has been developing instructional material for UPTEC. At present, she is Manager, Instructional Software Research and Development (ISRD) Group at UPTEC.

We are also indebted to **Mr. Surya Prakash Sharma, Mr. Vikash Bajpayee** and **Mr. Rajan Awasthi** for their help with the DTP work — drawing figures, creating tables, and preparing table of contents and indexes.

Contents at a Glance

Contents

Preface

Data Structure is a core module in the curriculum of almost every computer science programme. This subject makes the students learn the art of analyzing algorithms as well as distinguishing between the specification of a data structure and its realization within an available programming language. It involves identifying the problem, analyzing different algorithms to solve the problem and choosing most appropriate data structure to represent the data.

The value of an implementation ultimately relies on its resource utilization : time and space, and this requires capability of analyzing different factors. The coverage of this book is in line with the syllabus of Data Structure course being taught in the BE/B.Tech. (Computer Science), BCA and MCA programmes of various universities.

The goal of this book is to help you understand different concepts of data structures. It specifically covers the entire syllabus of "Data Structures Through 'C' Language" course as prescribed by DOEACC for its 'A' and 'B' Level Programmes. It contains additional topics that be useful for the students of BE/ B.Tech. (Computer Science), BCA and MCA programmes of various universities.

Chapter 1 gives an introduction to the core topics of data structures. It unleashes the concepts of abstract data types (ADTs), data types, data structures and other useful tools that can be used to solve problems.

Chapter 2 provides an introduction to software engineering and also explores the principles of good program design, the approaches to algorithm design and the analysis of the algorithms.

Chapter 3 discusses one of the linear data structure — Arrays. The storage representation of arrays and various operations possible on arrays have also been explained in this chapter.

Chapter 4 gives an introduction to Linked List, which is again a linear data structure like array, but uses dynamic memory allocation rather than static memory allocation as in the case of arrays.

Chapter 5 presents the two most important applications of arrays and linked lists—Polynomials and Sparse Matrix.

Chapter 6 discusses one of the most important data structures—Stack. The representation of stacks through arrays and linked lists have also been explored in this chapter.

Chapter 7 deals with Queues. Various kind of queues—Circular queues, Dequeues, Priority queues—have been discussed. The chapter also presents the application of queues.

Chapter 8 explores one of the most important non-linear data structures i.e., Trees. The ways to represent binary trees through arrays and linked lists and the various operations and its traversal has also been discussed in this chapter.

Chapter 9 provides a detailed description of AVL trees. The other kinds of trees—Forests, Orchards and Expression trees—are also discussed in this chapter.

Chapter 10 explores the Multiway trees. The searching, insertion and deletion operations on a B-tree have also been discussed in this chapter.

Chapter 11 focuses on various searching and sorting methods. The implementation of various sorting methods—Bubble sort, Selection sort, Quick sort, Insertion sort, Merge sort, Radix sort, Heap sort—also form part of this chapter.

Chapter 12 discusses Graphs in detail. Various basic terms and applications of graphs have been explored in this chapter.

Chapter 13 is all about Hashing—a technique which provides a conceptually different mechanism to search a table for a given key value.

Chapter 14 gives an introduction to Strings. The representation of strings through arrays and linked lists and various string operations have been covered in this chapter.

Chapter 15 includes some of the techniques and algorithms that could be used to provide various levels of storage management and control.

Appendix A explains various mathematical concepts—Matrices, Polynomials, Logarithms, Factorials, etc.

Appendix B at the end of this book is a collection of question papers from July 2002 to July 2004 which will be useful for the students in preparation of the DOEACC exams.

We look forward to receiving feedback and suggestions from the users of this book. It will help us improve the future editions of this book.

ISRD Group

Syllabus

OBJECTIVE OF THE COURSE

The objective of the course is to introduce the fundamentals of Data Structures, Abstract concepts and how these concepts are useful in problem solving. After completion of this course student will be able to:

- Understand and use the process of abstraction using a programming language such as 'C'.
- Analyze step by step and develop algorithms to solve real problems.
- Implement various data structures viz. Stacks, Queues, Linked Lists, Trees and Graphs.
- Understand various searching and sorting techniques.

Given below is the outline of the course that an instructor can use for effective delivery of the course. The suggested time distribution for various topics are specifically in line with the syllabus prescribed for the subject by DOEACC for its 'A' and 'B' Level programmes.

Outline of Course

S.No	*Topic*	*Minimum Hours*
1.	Basic Concepts of data representation	03
2.	Introduction to Algorithm Design and Data Structure	06
3.	Arrays	05
4.	Stacks and Queues	08
5.	Linked lists	08
6.	Trees	10
7.	Searching, sorting and complexity	10
8.	Graphs	10
	Lectures =	**60**
	Practicals/Tutorials =	60
	Total =	**120**

Detailed Syllabus

1. Basic concepts of data representation **03 Hrs.**

Abstract data types: Fundamental and derived data types. Representation, primitive data structures.

2. Introduction to Algorithm Design and Data Structures **06 Hrs.**

Design and analysis of algorithm: Algorithm definition, comparison of algorithms. Top-down and bottom up approaches to Algorithm design. Analysis of Algorithm; Frequency count, Complexity measures in terms of time and space. Structured approach to programming.

3. Arrays **05 Hrs.**

Representation of arrays: single and multidimensional arrays. Address calculation using column and row major ordering. Various operations on Arrays. Vectors, Application of arrays: Matrix multiplication, Sparse polynomial representation and addition.

4. Stacks and Queues **08 Hrs.**

Representation of stacks and queues using arrays and linked-list. Circular queues, priority Queues and D-Queue. Applications of stacks: Conversion from infix to postfix and prefix expressions, Evaluation of postfix expression using stacks.

5. Linked Lists **08 Hrs.**

Singly linked list; operations on list. Linked stacks and queues. Polynominal representation and manipulation using linked lists. Circular linked lists, Doubly linked lists. Generalized list structure. Sparse Matrix representation using generalized list structure.

6. Trees **10 Hrs.**

Binary tree traversal methods: Preorder, In-order, Post-ordered traversal. Recursive and non-recursive Algorithm for above mentioned Traversal methods. Representation of trees and its applciations: Binary tree representation of a tree. Conversion of forest into tree. Threaded binary trees; Lexical binary trees. Decision and game trees. Binary search tree.: Height balanced (AVL) tree, B-trees.

7. Searching, Sorting and complexity **10 Hrs.**

Searching: Sequential and binary searches, indexed search, Hashing Schemes. Sorting: Insertion, selection, bubble, Quick, merge, radix, Shell, Heap sort. comparison of time complexity.

8. Graphs **10 Hrs.**

Graph representation: Adjacency matrix, Adjacency lists, Adjacency Multicasts. Traversal schemes: Depth first search, Breadth first search. Spanning tree: Definition, Minimal spanning tree algorithm. Shortest path algorithms (Prime's and Kruskal's).

The syallbus given above (specifically for DOEACC 'A' and 'B' Level programmes) has been covered in the first twelve chapters of the book. The last three chapters i.e., Chapter 13, 14 and 15 on Hashing, String Processing and Storage Management have been added to cover additional topics prescribed by other universities. The instructors can accordingly schedule these topics and cover the additional material given in the last three chapters according to the syllabus requirements of the concerned university/ institute.

CHAPTER

1 Introduction to Data Structures

Key Features

- Introduction to the Theory of Data Structures
- Data Representation
- Abstract Data Types
- Data Types
- Primitive Data Types
- Data Structure and Structured Type
- Atomic Type
- Difference between Abstract Data Types, Data Types and Data Structures
- Refinement Stages

The advancement in the study of data structures for analyzing algorithms has continued. The study of data structures has two objectives. The first is to identify and create useful mathematical entities and operations to determine what classes of problems can be solved by using these entities and operations. The second is to determine the representation of these abstract entities and to implement the abstract operations on these concrete representations.

This chapter introduces the core concepts of data structures. It explores the concepts of abstract data types (ADTs), data types, data structures and other useful tools that can be used to analyze and solve problems.

INTRODUCTION TO THE THEORY OF DATA STRUCTURES

The study of computer science encompasses the study of organization and flow of data in a computer. Data structure is the branch of computer science that unleashes the knowledge of how the data should be organized, how the flow of data should be controlled and how a data structure should be designed and implemented to reduce the complexity and increase the efficiency of the algorithm.

A course in data structures offers an excellent opportunity to introduce the concepts associated with object oriented programming. For example, the concept of classes in object-oriented programming language (like C++) is based on the key concept of Abstract Data Type (ADT). An ADT exhibits many of the concepts enshrined in the theory of data structures.

The theory of structures not only introduces you to the data structures, but also helps you to understand and use the concept of abstraction, analyse problems step by step and develop algorithms to solve real world problems. It enables you to design and implement various data structures, for example, the

stacks, queues, linked lists, trees and graphs. Effective use of principles of data structures increases efficiency of algorithms to solve problems like searching, sorting, populating and handling voluminous data.

Need of a Data Structure

A data structure helps you to understand the relationship of one data element with the other and organize it within the memory. Sometimes the organization might be simple and can be very clearly visioned, for example, list of names of months in a year. We can see that names of months have a linear relationship between them and thus can be stored in a sequential location or in a type of data structure in which each month points to the next month of the year and it is itself pointed by its preceding month. This principle is overruled in case of first and last month's names. Similarly, think of a set of data that represents location of historical places in a country (Fig. 1.1). For each historical place the location is given by country name followed by state name followed by city name, and then followed by the historical place name. We can see that such data form a hierarchical relationship between them and must be represented in the memory using a hierarchical type of data structure.

The above two examples clearly identify the usefulness of a data structure. A data structure helps you to analyze the data, store it and organize it in a logical or mathematical manner.

Fig. 1.1

DATA REPRESENTATION

Various methods are used to represent data in computers. Hierarchical layers of data structure are used to make the use of data structure easy and efficient. The basic unit of data representation is a **bit.** The value of a bit asserts one of the two mutually exclusive possibilities—0 or 1. Various combinations of two values of a bit are used to represent data in a different manner in different systems. Eight bits together form one **byte** which represents a **character** and one or more than one characters are used to form a **string**. A string can thus be seen as a data structure that emerges through several layers of data structures as shown in Fig. 1.2.

The representation of a string can be made easier (i.e. working with the strings without bothering about the NULL (\0) character at the end of the string) by wrapping it into another data structure which takes care of such intricacies and supports a set of operations that allows us to perform various string related operations like storing and fetching a string, joining two strings, finding the length of strings, etc.

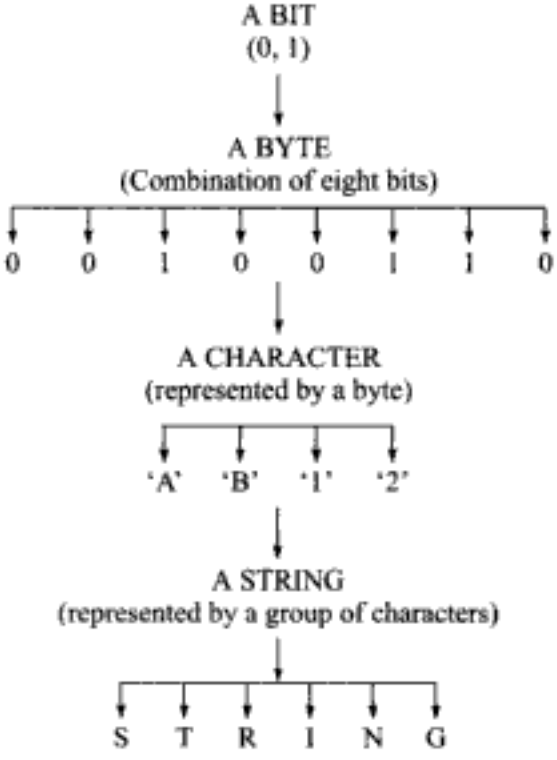


Fig. 1.2 *Data Representation*

The use of the concrete data structure during design creates lot of difficulties and requires much more efforts, such a problem can be avoided by using **abstract data type** in the design process. But before moving to the discussion of concepts of Abstract Data Types (ADTs), let us discuss how the primitive or basic data types of any language (i.e. integer, character, etc.) are internally represented in the memory.

Integer Representation

An integer is the basic data type which is commonly used for storing negative as well as non-negative integer numbers. The non-negative data is represented using **binary number system**. In this, each bit position represents the power of 2. The rightmost bit position represents 2^0 which is 1, the next represents 2^1 which is 2, then 2^2 which is 4 and so on. For example, 00100110 represents the integer as $2^1 + 2^2 + 2^5 = 2 + 4 + 32 = 38$.

For negative binary numbers the methods of representation used are **one's complement** and **two's complement**.

In **one's complement** method the number is represented by complementing each bit, i.e. changing each bit in its value to the opposite bit setting. For example, 0 0 1 0 0 1 10 represents 38, after complementing, it becomes 1 1 0 1 1 0 0 1 which is used to represent –38.

In **two's complement** method, 1 is added to one's complement representation of the negative number. For example, –38 is represented by 1 1 0 1 1 0 0 1 which on adding 1 to it will become

$$\begin{array}{r} 1\,1\,0\,1\,1\,0\,0\,1 \\ +\;1 \\ \hline 1\,1\,0\,1\,1\,0\,1\,0 \end{array}$$

, which represents –38 in two's complement notation.

Real Number Representation

The method used to represent real numbers in computers is **floating-point notation**. In this notation, the real number is represented by a number called a **mantissa,** times a base raised to an integer power, called an **exponent**. For example, if the base is fixed as 10, the number 209.52 could be represented as 20952×10^{-2}. The mantissa is 20952 and exponent is – 2. Both the mantissa and exponents are two's complement binary integers. For example, 20952 can be represented as 1 0 1 0 0 0 1 1 1 0 1 1 in binary form.

Therefore, the 24 bit representation of the number will be 0 0 0 0 0 0 0 0 1 0 1 0 0 0 1 1 1 0 1 1 and 8 bit two's complement binary representation of –2 is 1 1 1 1 1 1 1 0; thus, the number is represented as 0 0 0 0 0 0 0 0 1 0 1 00 0 1 1 1 0 1 1 . 1 1 1 1 1 1 1 0.

Character Representation

The information in computers is not always interpreted numerically. For example, to store the salary of an employee we use the integer representation of the data but with salary we also need to store the name of the employee which requires a different data representation. Such information is usually represented in character string form. There are different codes available to store data in character form such as BCD, EBCDIC and ASCII.

For example, if 8 bits are used to represent a character, then up to 2^8=256 different characters can be represented as bit patterns. 1 1 0 0 0 0 0 0 is used to represent the character 'A' and 1 1 0 0 0 0 0 1 is used to represent character 'B'. Then, finally AB can be represented as 1 1 0 0 0 0 0 0 1 1 0 0 0 0 0 1.

ABSTRACT DATA TYPES

An **Abstract Data Type** (ADT) is defined as a mathematical model of the data objects that make up a data type as well as the functions that operate on these objects. An abstract data type is the specification of logical and mathematical properties of a data type or structure. ADT acts as a useful guideline to implement a data type correctly. The specification of an ADT does not imply any implementation consideration. The implementation of an ADT involves the translation of the ADT's specification into syntax of a particular programming language. The important step is the definition of ADT that involves mainly two parts:

1. Description of the way in which components are related to each other
2. Statements of operations that can be performed on that data type.

For example, the **int** data type, available in the 'C' programming language provides an implementation of the mathematical concept of an integer number. The int data type in 'C' can be considered as an implementation of Abstract Data Type, INTEGER-ADT. INTEGER-ADT defines the set of numbers given by the union of the set {-1, -2, -3....∞} and the set of whole numbers {0, 1... + ∞}. INTEGER-ADT also specifies the operations that can be performed on an integer number, for example, addition, subtraction, multiplication, etc. In the specification of the INTEGER-ADT, following issues must be considered:

- Range of numbers that will be represented.
- Format for storing the integer numbers.

This determines the representation of a signed integer value.

Example of an ADT Specification for a Stack

A stack is a set of finite number of elements, where insertion or removal of an element is allowed from one end only. Any new element that joins the set is kept at the top and only the element at the topmost position can be taken out. In other words, a stack can be defined as a list where the only element that is accessible is the most recently inserted one.

There are a number of methods for specifying an ADT. The example below uses an informal notation to specify STACK abstract data type.

Specification of Abstract data type STACK is as follows:

```
abstract typedef <1, 2, 3 .....n, top> STACK;
condition top == NULL
abstract STACK PUSH (data)
Precondition  :      top ! = n
operation     :      insert (data)
                     top → data
                     /* top points to the most recently inserted elements */
abstract             Pop ( )
Preconditions top ! NULL
operation     :      Remove (top)
                     top → current data
```

DATA TYPES

Each programming language has its own set of data types. There are several definitions that define a data type. Few of these definitions are given below:

- A method of interpreting a bit pattern is often called a **data type**.
- The term **data type** refers to the implementation of the mathematical model specified by an ADT.
- A **data type** is the abstract concept defined by a set of logical properties.

In general, a data type can be defined as an abstract concept that defines an internal representation of data in the memory. Once the concept of data type is made independent of the hardware capabilities of the computer, a limitless number of data types can be considered.

A data type is an abstract concept defined by a set of logical properties. Once such abstract data type is defined and the legal operations involving that type are specified, a data type can then be implemented. An implementation may be a hardware implementation in which the circuitry, necessary to program the required operations, is designed as part of the computer, or otherwise there may be a software implementation, in which program consisting of already existing hardwired instructions is written to interpret bit strings in the desired fashion and to perform the required operations. A software implementation includes the following specifications:

- How an object of the new data type is represented by objects of existing data types?
- How such an object is implemented in conformance with the operations defined for it?

Why do We Need a Data Type?

We can think of a universal data type that may hold any value like character, integer, float or any complex number. Use of such data type has two disadvantages:

- Large volume of memory will be occupied by even a small size of data.
- Different types of data require different interpretation of bit strings while reading or writing. Different interpretations for different types of data would become very tedious. This can be explained in terms of negative integer and a float value. Most of the computers represent a negative value by storing 1 at the most significant bit (MSB). Similarly, a float value is specially handled by storing its characteristics and mantissa in a particular pattern. Both of these data types require special interpretation of bit strings which will become very complex while handling a universal data type.

Thus, we see that the data types facilitate the optimum use of memory as well as a defined way to interpret the bit strings for different types of data.

PRIMITIVE DATA TYPES

Every computer has a set of native data types. This means that it is constructed with a mechanism for manipulating bit pattern at a given location as binary numbers. **Primitive data types** are **basic** data types of any language that form the basic unit for the data structure defined by the user. A **primitive data type** defines how the data will be internally represented in, stored, and retrieved from the memory. In most programming languages, primitive data types are mapped to the native data types of the computer whereas in some languages new data types are offered using the software implementation in which a

program consisting of already existing hardware instructions is written to interpret bit strings in the desired fashion and to perform the required operations. For example, an integer type data can be directly mapped to the computer's native data type whereas 'Date' data type may be implemented using the software implementation which in turn again will use integer representation of Date type data. Few primitive data types which are commonly available with most programming languages are:

- Integer
- Character
- Real/Float Numbers

Integer

An integer data type is a primitive data type that may represent a range of numbers from $-2^{(n-1)} + 1$ to $2^{(n-1)} - 1$, where n depends upon the number of bits used to constitute one word in the computer.

Character

At a more general level, information can be represented in the form of characters. Any symbol from set 0–9, A–Z, a–z and other special symbols is a character. Most of the computers use eight bits to represent a character. Thus, 2^n (2^8), i.e. 256 characters can be represented using a string of eight bits. The number of bits necessary to represent a character in a particular computer is called the **byte size.**

Real/Float Numbers

A real number consists of two parts, mantissa and characteristic. A real number data type is generally denoted with the term **float**. This is because computers usually represent a real number using a floating-point notation. There are many varieties of floating-point notations and each has individual characteristic. A real number is represented using the following expression:

$$\mathbf{m} \times \mathbf{n}^{\mathbf{r}}$$

where **m** is the mantissa and **n** is the base (which is fixed 10) and **r** is the exponent. A floating point notation facilitates storage of numbers with extremely large and extremely small values. The range of values that can be represented in a float data type are from $-3.4*10^{38}$ to $+3.4*10^{38}$.

DATA STRUCTURE AND STRUCTURED TYPE

The term data structure refers to a set of computer variables that are connected in some logical or mathematical manner. More precisely, a data structure can be defined as the structural relationship present within the data set and thus should be viewed as 2 tuple. (N, R) where 'N' is the finite set of nodes representing the data structure and 'R' is the set of relationship among those nodes. For example, in a tree data structure each node is related to each other in a 'parent child' relationship. Thus, a large volume of data can be represented using a tree data structure and relationship between each data can also be shown.

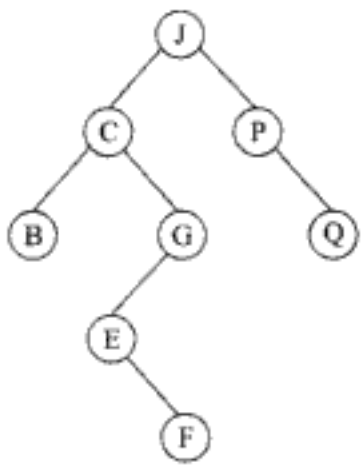


Fig. 1.3 *A Tree Data Structure*

A **structured type** refers to a data structure which is made up of one or more elements known as **components**. These elements

are simpler data structures that exist in the language. The components of structured data type are grouped together according to a set of rules, for example, the representation of **polynomials** requires at least two components:

- Coefficient
- Exponent

The two components together form a **composite type** structure to represent a polynomial. The study of these structured type data structures involves an analysis of how simple structures combine to form the composite and how to extract an element from the composite. In 'C' programming language, the structure type is implemented using **struct** keyword.

ATOMIC TYPE

Generally, a data structure is represented by a memory block, which has two parts:

- Data storage
- Address storage

This facilitates in storing the data and relating it to some other data by means of storing pointers in the address part.

An atomic type data is a data structure that contains only the data items and not the pointers. Thus, for a list of data items, several atomic type nodes may exist each with a single data item corresponding to one of the legal data types. The list is maintained using a list node which contains pointers to these atomic nodes and a type indicator indicating the type of atomic node to which it points. Whenever a test node is inserted in the list, its address is stored in the next free element of the list of pointers.

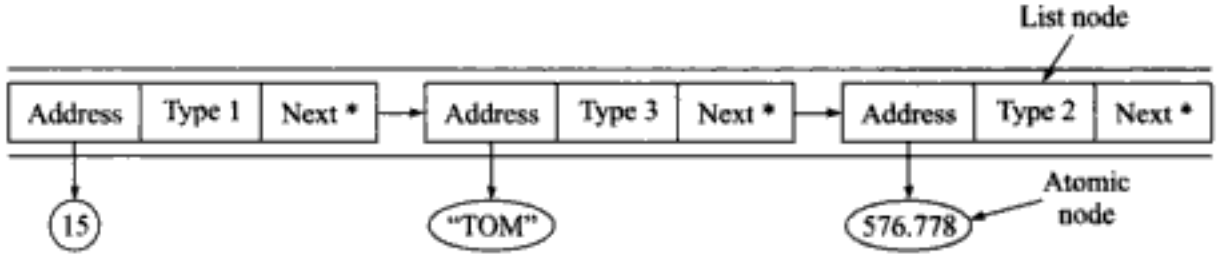


Fig. 1.4 *Atomic Nodes*

Figure 1.4 shows a list of atomic nodes maintained using list of nodes. In each node, type represents the type of data stored in the atomic node to which the list node points. 1 stands for integer type, 2 for real number and 3 for character type or any different assumption can be made at implementation level to indicate different data types.

DIFFERENCE BETWEEN ABSTRACT DATA TYPES, DATA TYPES AND DATA STRUCTURES

To avoid the confusion between abstract data types, data types, and data structures, it is relevant to understand the relationship between the three.

- An **abstract data type** is the specification of the data type which specifies the logical and mathematical model of the data type.
- A **data type** is the implementation of an abstract data type.
- **Data structure** refers to the collection of computer variables that are connected in some specific manner.

Thus, there seems to be an open relationship between the three, that is, a data type has its root in the abstract data type and a data structure comprises a set of computer variables of same or different data types.

REFINEMENT STAGES

The best approach to solve a complex problem is to divide it into smaller parts such that each part becomes an independent module which is easy to manage. An example of this approach is the System Development Life Cycle (SDLC) methodology. This helps in understanding the problem, analyzing solutions, and handling the problems efficiently.

The principle underlying writing large programs is the **top-down refinement**. While writing the main program, we decide **exactly** how the work will be divided into various functions and then, in the refinement process, it is further decided what will be the task of each function, and what inputs are to be given and results to be obtained. The data and actions of the functions are specified precisely.

Similarly, the purpose of studying Abstract data types is to find out some general principles that will help in designing efficient programs. There exists a similarity between the process of **top down refinement** of algorithms and the **top-down specification** of the data structures. In algorithm design, we begin with the problem and slowly specify more details until we develop a complete program. In data specification, we begin with the selection of mathematical concepts and abstract data types required for our problem, and slowly specify more details until finally we can describe our data structures in terms of programming language.

The application or the nature of problem determines the number of refinement stages required in the specification process. Different problems have different number of refinement stages, but in general, there are four levels of refinement processes:

- Conceptual or abstract level
- Algorithmic or data structures
- Programming or implementation
- Applications

Conceptual Level

At this level we decide how the data is related to each other, and what operations are needed. Details about how to store data and how various operations are performed on that data are not decided at this level.

Algorithmic or Data Structure Level

At data structure level we decide about the operations on the data as needed by our problem. For example, we decide what kind of data structure will be required to solve the problem—**contiguous list** will be preferred for finding the length of a list, or for retrieving any element, whereas for the evaluation of any expression into prefix or postfix, **stacks** will be used.

Programming or Implementation Level

At implementation level, we decide the details of how the data structures will be represented in the computer memory. For example, we decide whether the linked lists will be implemented with pointers or with the cursors in an array.

Application Level

This level settles all details required for particular application such as names for variables or special requirements for the operations imposed by applications.

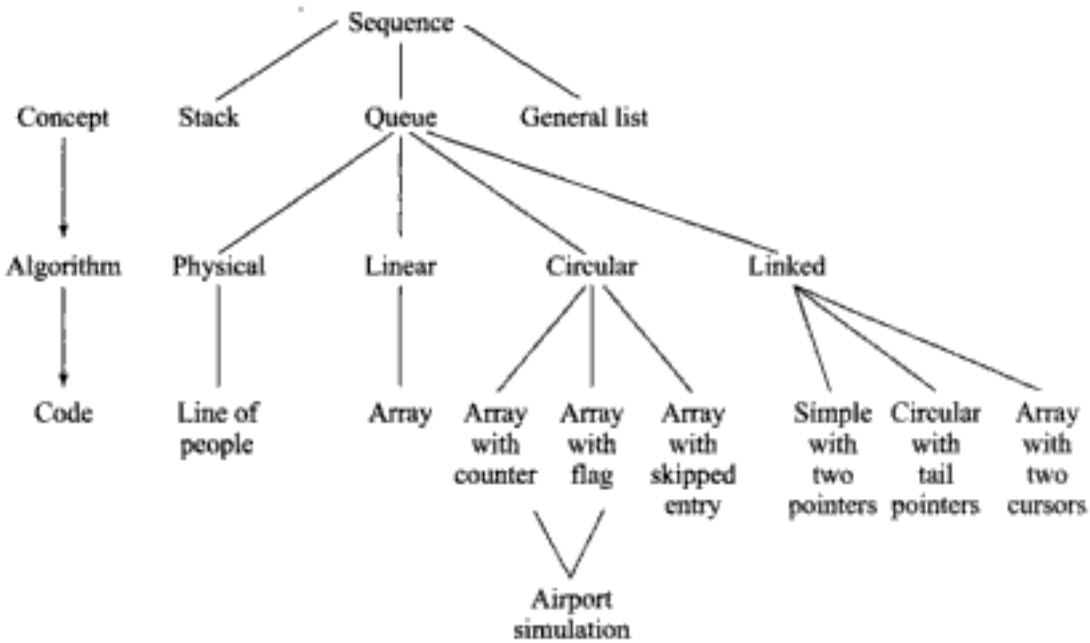


Fig. 1.5 *Refinement of a Queue*

The first two levels are often called **conceptual**. The middle two levels can be called **algorithmic** as they are concerned with representing data and the operations performed on the same. Last level is basically concerned with programming.

Figure 1.5 shows that at the conceptual level, the queue has been selected as an abstract data type and further at the next level circular queue has been selected, as this could provide the best solution. Last level shows the operations which can be performed on the data, for the Airport simulation.

Summary

- Data Structure is the particular organization of data either in a logical or mathematical manner.
- Data type is a concept that defines internal representation of data.
- An abstract data type is the specification of logical and mathematical properties of data types or structure. It acts as a guideline to implement a data structure.
- The relationship between ADT, data type and data structure is well defined. An abstract data type is the specification of a data type whereas data type is the implementation of ADT and data structure comprises computer variables of same or different data types.

Review Exercise

Multiple Choice Questions

1. Representation of an integer in the computer system can be done through
   a. 1's complement method
   b. 2's complement method
   c. Both a and b
   d. None of the above
2. Key concept for representing a real number is a mantissa times
   a. 10 raised to an exponent
   b. 2 raised to an exponent
   c. Base raised to an exponent
   d. None of the above
3. The set of native data types that a particular computer can support is determined by
   a. Type of hardware company
   b. What functions have been wired into hardware
   c. What software support is required
   d. None of the above
4. While considering data structure implementation, the factors under consideration is/are
   a. Time
   b. Space and time
   c. Time, space and processor
   d. None of the above

Fill in the Blanks

1. The method of interpreting a bit pattern is called a _____ .
2. One of the examples of a structured data type can be _____ .
3. _____ refers to the collection of computer variables that are connected in some specific manner.
4. Data structure defined at logical level is called _____ .

State Whether True or False

1. In 1's complement method, the positive '0' and negative '0' is separately represented.
2. The binary number system is the only means by which bits can be used to represent integers.
3. The limiting factor on the precision of numbers that can be represented on a particular machine is the number of significant digits in mantissa.
4. A data type is the collection of values and the set of operations on values.

Answer the Following Questions

1. Explain data type and why do we need data types?
2. Explain abstract data type with an example.
3. What is a data structure and what are the differences between data type, abstract data type and data structure?

4. Draw a diagram for showing levels of refinement for a stack and a list for any problem.
5. Develop an ADT specification for "Polynomials". Also include the operations associated with polynominals.
6. Suggest a suitable data structure for representation of imaginary numbers. An imaginary number is represented by a+ib where i is the iota for the number. Also give specification for the operations associated with them.

CHAPTER

2 Principles of Programming and Analysis of Algorithms

Key Features

- Software Engineering
- Program Design
- Algorithms
- Different Approaches to Designing an Algorithm
- Complexity
- Big 'O' Notation
- Algorithm Analysis
- Structured Approach to Programming
- Recursion
- Tips and Techniques for Writing Programs in C

This chapter starts with the introduction to software engineering and gives the fundamental knowledge required to solve various problems using computers. Algorithm design is one of the basic steps in solving problems. An algorithm is a set of steps and instructions required to solve a given problem. The chapter further explains the principles of good program design, the approaches to algorithm design and the analysis of the algorithms.

There are some mathematical tools like Big 'O' notation which are described for the purpose of analyzing algorithms. The basis and basics of structured programming have also been discussed. It also introduces various programming tools like recursion and looping which help in the proper implementation of algorithms.

SOFTWARE ENGINEERING

Software Engineering is the theory and practice of methods helpful for the construction and maintenance of large software systems. Development of a good software is a tedious process which continues for long time before the software or program takes the final shape and is put into use. There are many stages in the software development cycle. The process is often referred to as **Software Development Life Cycle (SDLC)**. In SDLC, the output from one stage becomes the input to the next stage.

In the **simplified version,** the software development life cycle may contain requirement analysis, design, implementation and maintenance phases which are implemented in sequence over a period of time. This simplified version can be depicted through Fig. 2.1.

The different steps in software development life cycle are as follows:

1. **Analyze** the problem precisely and completely.
2. **Build** a prototype and **experiment** with it until all specifications are finalized.

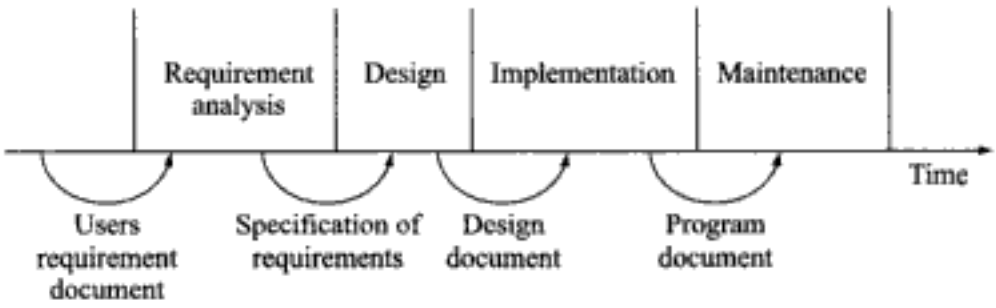


Fig. 2.1 *Software Development Life Cycle*

3. **Design** the algorithm using the tools of data structures.
4. **Verify** the algorithm, such that its correctness is self-evident.
5. **Analyze** the algorithm to determine its requirements.
6. **Code** the algorithm into an appropriate programming language.
7. **Test** and **evaluate** the program with carefully chosen data.
8. **Refine** and **Repeat** the foregoing steps until the software is complete.
9. **Optimize** the code to improve performance.
10. **Maintain** the program so that it meets the changing needs of its users.

PROGRAM DESIGN

Program design can be considered as an important phase of the software development life cycle. It is in this phase that the algorithms and data structures to solve a problem are proposed. Some of the various points that can help us evaluate the proposed program designs are as follows:

- As the design stage involves taking the specification and designing solutions to the problems, the designer needs to adopt a design strategy. The strategy adopted while designing should be according to the given specifications.
- Another important point which should be kept in mind while developing a solution strategy is that it should work correctly in all conditions.
- Generally, the people who use the system are not aware of the program design you have adopted. Thus, there is a system manual which is a detailed guide to how the design was achieved. In addition a user manual serves as a reference for the users who are not familiar with the system or machines.
- A large program should be divided into small modules and submodules by following one of the two decomposition approaches—top-down approach or bottom-up approach.
- Other important criteria by which a program can be judged are execution time and storage requirement.

ALGORITHMS

The term '**algorithm**' refers to the sequence of instructions that must be followed to solve a problem. In other words, an algorithm is a logical representation of the instructions which should be executed to perform a meaningful task.

An algorithm has certain characteristics. These are as follows:

- Each instruction should be unique and concise.
- Each instruction should be relative in nature and should not be repeated infinitely.
- Repetition of same task(s) should be avoided.
- The result should be available to the user after the algorithm terminates.

Thus, an algorithm is any well defined computational procedure, along with a specified set of allowable inputs, that produce some value or set of values as output.

After an algorithm has been designed, its **efficiency** must be analyzed. This involves determining whether the algorithm is economical in the use of computer resources, i.e. CPU time and memory. The term used to refer to the memory required by an algorithm is **memory space** and the term used to refer to the computational time is the **running time**.

The importance of efficiency of an algorithm is in the **correctness**—that is, does it always produce the correct result, and **program complexity** which considers both the difficulty of implementing an algorithm along with its efficiency.

DIFFERENT APPROACHES TO DESIGNING AN ALGORITHM

A complex system may be divided into smaller units called **modules**. The advantage of modularity is that it allows the principle of separation of concerns to be applied into two phases: when dealing with detail of each module in isolation (ignoring details of other modules) and when dealing with overall characteristics of all modules and their relationships in order to integrate them into a system. Modularity enhances design clarity, which in turn eases implementation, debugging, testing, documenting, and maintenance of the product.

A system consists of components, which have components of their own. Indeed a system is a hierarchy of components. The highest level component corresponds to the total system. To design such a hierarchy there are two possible approaches:

- Top-down approach
- Bottom-up approach

Top-Down Approach

A top-down design approach starts by identifying the major components of the system or program, decomposing them into their lower-level components and iterating until the desired level of module complexity is achieved. Top-down design method takes the form of **stepwise refinement**. In this, we start with the topmost module and incrementally add modules that it calls.

Thus, in top-down approach we start from an abstract design. In each step, design is refined into most concrete level until we reach the level where no more refinement is needed and the design can be implemented directly.

Bottom-Up Approach

A bottom-up design approach starts with designing the most basic or primitive components and proceeds to higher-level components. Bottom-up method works with **layers of abstraction**. Starting from the very bottom, the operations that provide a layer of abstraction are implemented. The operations of this

layer are then used to implement more powerful operations and still higher layer of abstraction, until the stage is reached where the operations supported by the layer are those desired by the system.

Top-Down versus Bottom-Up Approach

What strategy should be followed to design a system? Should we follow the top-down approach, breaking down the system into manageable components or should we adhere to the bottom up approach of defining a module and then grouping together several modules to form a new higher level module?

Stepwise refinement is "top-down" method for decomposing a system from high level specifications into more elementary levels. It has its criticisms that the submodules tend to be analysed in isolation, that no emphasis is given on the identification of communication or on reusability of components and little attention is paid to data and more generally, to information hiding.

Bottom-up approach follows information hiding. It suggests that we should first recognize what we wish to encapsulate within a module then provide an abstract interface to define the module's boundaries as seen from the clients. However, what is to be hidden depends on the result of some top-down design activity. Some information hiding has proven to be highly effective in supporting design for change, program requirements, or reusable components.

The top-down approach, however, is often useful way to better document a design. The design activity should not be constrained to proceed according to a fixed pattern but should be a blend of top-down and bottom-up approaches.

COMPLEXITY

When we talk of complexity in context of computers, we call it **computational complexity.** Computational complexity is a characterization of the time or space requirements for solving a problem by a particular algorithm. These requirements are expressed in terms of a single parameter that represents the size of the problem.

Given a particular problem, say one of the simplification problems. Let 'n' denote its size. The time required of a specific algorithm for solving this problem is expressed by a function:

$$f : R \rightarrow R$$

such that f(n) is the largest amount of time needed by the algorithm to solve the problem of size n. Function 'f' is usually called the time complexity function.

Thus, we conclude that the analysis of the program requires two main considerations:

- Time complexity
- Space complexity

The time complexity of a program/algorithm is the amount of computer time that it needs to run to completion. The space complexity of a program/algorithm is the amount of memory that it needs to run to completion.

Time Complexity

While measuring the time complexity of an algorithm, we concentrate on developing only the frequency count for all key statements (statements that are important and are the basic instructions of an algorithm). This is because, it is often difficult to get reliable timing figure because of clock limitations and the multiprogramming or the sharing environment.

Consider the algorithm given below:

ALGORITHM A	a = a + 1

ALGORITHM B	for x = 1 to n step 1 a = a + 1 Loop

ALGORITHM C	for x = 1 to n step 1 for y = 1 to n step 1 a = a + 1 Loop

In the algorithm A we may find that the statement a=a+1 is independent and is not contained within any loop. Therefore, the number of times this shall be executed is 1. We can say that the **frequency count** of algorithm A is **1**.

In the second algorithm, i.e. B, the key statement out of three statements is the assignment operation a = a+1. Because this statement is contained within a loop, the number of times it is executed is n, as the loop runs for n times. The frequency count for this algorithm is **n**.

According to the third algorithm, the frequency count for the statement a=a+1 is n^2 as the inner loop runs n times, each time the outer loop runs, the outer loop also runs for n times. n^2 is said to be different in increasing order of magnitude just like 1, 10, 100 depending upon the n. During the analysis of algorithm we shall be concerned with determining the order of magnitude of an algorithm. This means that we will determine only those statements which may have the greatest frequency count.

The following formulas are useful in counting the steps executed by an algorithm:

$$1 + 2 + \ldots + n = \frac{n(n+1)}{2}$$

$$1^1 + 2^2 + \ldots + n^2 = \frac{n(n+1)(2n+1)}{6}$$

If an algorithm performs f(n) basic operations when the size of its input is n, then its total running time will be cf(n), where c is a constant that depends upon the algorithm, on the way it is programmed, and on the way the computer is used, but c does not depend on the size of the input.

Space Complexity

The space needed by the program is the sum of the following components:

Fixed space requirement This includes the instruction space, for simple variables, fixed size structured variables, and constants.

Variable space requirement This consists of space needed by structured variables whose size depends on particular instance of variables. It also includes the additional space required when the function uses recursion.

BIG 'O' NOTATION

If f(n) represents the computing time of some algorithm and g(n) represents a known standard function like n, n^2, n log n, etc. then to write:

$$f(n) \text{ is } 0\ g(n)$$

means that f(n) of n is equal to biggest order of function g(n). This implies only when:

$|f(n)| \leq C|g(n)|$ for all sufficiently large integers n, where C is the constant whose value depends upon various factors discussed above.

From the above statements, we can say that the computing time of an algorithm is O(g(n)), we mean that its execution takes no more than a constant time g(n). n is the parameter which characterizes the input and/or outputs. For example, n might be the number of inputs or the number of outputs or their sum or the magnitude of one of them. If analysis leads to the result f(n)=0(g(n)), then it means that if the algorithm is run on the same computer on some input data for sufficiently large values of n, then the resulting computation time will be less than some constant time |g(n)|.

Why Big 'O' Notation

Big O notation helps to determine the time as well as space complexity of the algorithms. Using the Big O notation, the time taken by the algorithm and the space required to run the algorithm can be ascertained. This information is useful to set the prerequisites of algorithms and to develop and design efficient algorithms in terms of time and space complexity.

The Big 'O' Notation has been extremely useful to classify algorithms by their performances. Developers use this notation to reach to the best solution for the given problem. For example, for the Quick sort algorithm the worst case complexity is $O(n^2)$ whereas for Bubble sort the average case complexity is $O(n^2)$. Thus, Quick sort can be graded as the better algorithm for sorting by any developer who has choice between the two.

Most Common Computing Times of Algorithm

If the complexity of any algorithm is 0(1), it means that the computing time of the algorithm is constant 0(n) and it is called **linear time** which implies that it is directly proportional to n. $0(n^2)$ is called the quadratic time, $0(n^3)$ is the cubic time, $0(2^n)$ is exponential time, 0(log n) and 0(n log n) are the logarithmic times. Algorithms with exponential running time are not suitable for practical use.

The common computing times of algorithms in the order of performance are as follows:

- 0(1)
- 0(log n)
- 0(n)
- 0(nlog n)
- $0(n^2)$
- $0(n^3)$
- $0(2^n)$

Thus algorithms, according to their computational complexity, can be rated as per the aforementioned order of performance.

ALGORITHM ANALYSIS

There are different ways of solving a problem and there are different algorithms which can be designed to solve a problem. Therefore, there is a difference between a problem and an algorithm. A problem has a single problem statement that describes it in some general terms. However, there are many different ways to solve the problem, and some of the solutions may be more efficient than the others.

Consequently, analysis of algorithms focuses on computation of space and time complexity. **Space** can be defined in terms of space required to store the instructions and data whereas the **time** is the computer time an algorithm might require for its execution, which usually depends on the size of the algorithm and input.

There are different types of time complexities which can be analyzed for an algorithm:

- Best case time complexity
- Average case time complexity
- Worst case time complexity

Best Case Time Complexity

The best case time complexity of an algorithm is a measure of the minimum time that the algorithm will require for an input of size 'n'. The running time of many algorithms varies not only for the inputs of different sizes but also for the different inputs of same size. For example, in the running time of some sorting algorithms the sorting will depend on the ordering of the input data. Therefore, if an input data of 'n' items is presented in sorted order, the operations performed by the algorithm will take the least time, just checking the data in the sorted order which will correspond to the best case time complexity for an algorithm.

Worst Case Time Complexity

The worst case time complexity of an algorithm is a measure of the maximum time that the algorithm will require for an input of size 'n'. Therefore, if various algorithms for sorting are taken into account and, say, 'n' input data items are supplied in reverse order for any sorting algorithm, then the algorithm will require n^2 operations to perform the sort which will correspond to the worst-case time complexity of the algorithm.

The worst case time complexity is useful for a number of reasons. After knowing the worst case time complexity, we can guarantee that the algorithm will never take more than this time and, such a guarantee can be important in some time-critical software applications.

Average Case Time Complexity

The time that an algorithm will require to execute a typical input data of size 'n' is known as average case time complexity. We can say that the value that is obtained by averaging the running time of an algorithm for all possible inputs of size 'n' can determine average-case time complexity. This case of time complexity may not be considered good measure as in this we have to assume the underlying probability distribution for the inputs which if, in practice, is violated, then the determination of average case time complexity will be meaningless.

Therefore, the computation of exact time taken by the algorithm for its execution is very difficult. Thus, the work done by an algorithm for the execution of the input of size 'n' defines the time analysis as function f(n) of the input data items.

Another important step which can be considered in the analysis of an algorithm is identifying the **abstract operation** on which an algorithm is based. For example, to identify the largest element in an array, the primary operation of an algorithm is comparison of the array of 'n' elements to find out the element with maximum value, then the exchange operation becomes more important than comparison in the array.

STRUCTURED APPROACH TO PROGRAMMING

Structured programming is a subset of software engineering. It is a method for designing and coding programs in a systematic, organized manner. The emphasis of structured programming is mainly on the technical aspects of programming, whereas software engineering puts equal emphasis on technical, managerial, psychological and financial aspects of software development.

The term 'structured programming' was coined by Dijkstra in the article 'Structured Programming'. We deal with various tools required for building a structured program. **Wirth** has defined program as follows:

Program = algorithm + data structures

Some of the control structures which can be used for structured programming are given in Fig. 2.2.

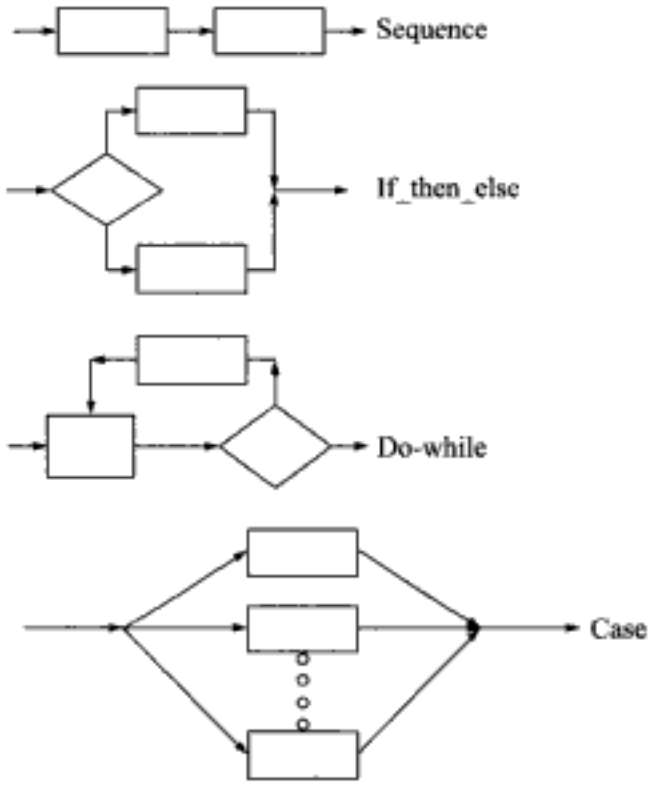


Fig. 2.2 *The Set of Basic Control Structures for Structured Programming*

Structured Programming emphasises functional specialization and tries to ensure that only one primary function is allocated to any module.

RECURSION

A recursive routine is one whose design includes a call to itself. In design phase of software development we use various problem solving methods in which recursion can be one of the powerful tools. For example, in a procedure to design a stack, we start at the top of the stack and move down the stack until the bottom is reached.

In an alternative perspective, while printing of the stack, it can be thought of as the top together with the remaining items of the stack — the 'substack', in order to emphasize the fact that items that comprise it is itself a stack. In this way, a stack of n items can be printed by printing the top element of the stack followed by printing of the substack consisting of n–1 items. Again, the printing is continued by printing the top element of the substack and then printing out the substack with n–2 elements or items. Likewise, at every stage the top item is printed and the remaining substack processed.

Finally, task is complete when the substack is empty and no printing is required. Therefore, it can be clearly seen that printing of substack requires exactly the same procedure of printing out the stack. In other words, the procedure of printing uses itself within the description. This is called **Recursion**.

Another common example which can explain the concept of recursion very easily is to find out the factorial of any number.

A 'C' function declaration for factorial is:

```
factorial(a)
int a;
{
        int fact = 1
        if(a>1)
        fact =a * factorial(a - 1);      /* recursive function call */
        return(fact);
}
```

In the function given above, when any number is passed to the function say:

number = 1
fact = a * factorial(a–1)
= 1 * factorial(a–1)
= 1

Similarly, if number = 2
fact = a * factorial(a–1)
= 2 * factorial(1)
= 2 * 1 = 2(from 1)

for number = 3
fact = a * factorial(a–1)
= 3 * factorial(2)
= 3 * 2 * factorial(from 2)
= 3 * 2 * 1(from 1)
= 6

for number = 4

```
fact        = a * factorial(a–2)
            = 4 * factorial(3)
            = 4 * 3 * factorial(2)(from 3)
            = 4 * 3 * 2 * factorial(1)(from 2)
            = 4 * 3 * 2 * 1(from 1)
            = 24
```

Looping and Recursion

Loops are used when we want to execute a part of the program or block of statements several times. There are many statements for the looping purpose. For example, while, do while, for, switch, etc.

A recursive function, as discussed above, is the function which calls itself (in function body) again and again.

For example, to print the sum of digits of any number using loops:

```
main()
{
        .
        .
        .
        do
        {
                rem = n %10;  /* taking last digits */
                sum + = rem;
                n / = 10      /* Skipping last digit */
        } while (n>0);
}
```

Principles of Recursion

Some basic principles which are used while designing algorithms with recursion are:

Find the Key Step When beginning with the design of algorithms through recursion one should try to find out the 'Key step' for the solution. Once you have executed the key step, find out whether the remainder of the problem can be solved in the same way, and modify the step, if necessary.

Find a Stopping Rule The stopping rule indicates that the problem or a substantial part of it is done and the execution of the algorithm can be stopped.

Outline your Algorithm After determining the key step for the problem and finding the cases which are to be handled, the next step is to combine these two using an **'if'** statement to select between them. The main program and the procedure for recursion are now to be written to carry the key step through until the stopping rule applies.

Check Termination Care should be taken to ensure that the recursion will always terminate after the finite number of steps and the stopping rule should also be satisfied. It should also handle the extreme cases correctly.

Draw a Recursion Tree The key tool for the analysis of recursive algorithms is the recursion tree as it helps in determining the amount of memory that the program will require and, the total size of tree reflects the number of times the key step will be performed and the total time needed for the program.

Some examples of recursion are given below:

```
sum(n)
int n;
{
        if(n > 0)
        {
                rem = n % 10;
                s + = rem;
                sum(n/10);    /* recursive function call */
        }
        return(s);
}
```

Similarly, another example of finding out the factorial of any number without recursion is

```
main()
{
                .
                .
                .
                if(n < 0)
        printf("No factorial of Negative number");
            else
                if(n==0)
        printf("Factorial of zero is 1n");
            else
                while(n > 1)
                {
                    fact * = n;
                    n - -;
                }
        }
```

Function for factorial using Recursion:

```
factorial(n)
int n;
{
        int fact = 1
        if(n > 1)
        fact = n * factorial(n - 1);      /* recursive function call */
```

```
        return(fact);
}
```

Comparison between Recursion and Iteration

For recursion, the **recursion tree** can help in providing useful information for deciding when recursion should and should not be used.

A recursion tree can be defined as a part of tree showing the recursive calls. Therefore, if a procedure or function makes only one recursive call to itself, its recursion tree is simple, rather it is a **chain,** each vertex having only one child.

Let us consider an example. In the problem of finding the factorial of any number, the main task is to calculate the factorial from (n–1) down to 1. The recursion tree for calculating factorial is given in Fig. 2.3.

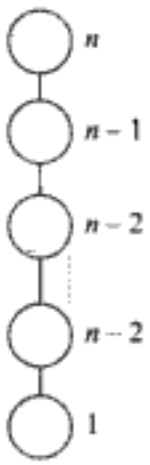


Fig. 2.3

After reading the tree from bottom to top we can obtain an iterative program. Thus, when the tree is reduced to a chain, the transformation of recursion to iteration is easy and saves both space and time.

Thus, in this example of finding factorial, after studying the recursion tree it was found that using iteration is easy and economical than recursion. But on the other hand, the recursion tree for calculating **Fibonacci series** is not a chain but contains many vertices signifying duplicate tasks. When a recursive program is run, it sets up a stack to use while traversing the tree, but if the results stored on the stacks are discarded rather than kept in some other data structure for further use, then a great deal of duplication of work may occur as in recursive calculation of Fibonacci series.

Therefore, for the Fibonacci numbers we need additional temporary variables to hold the information required for calculating the current number. Finally, by setting up or taking another data structure for such type of calculation, it is possible to change any recursive program into the non-recursive form.

TIPS AND TECHNIQUES FOR WRITING PROGRAMS IN 'C'

A **program** in any language is a collection of one or more functions. Every function is a collection of statements which performs a specific task. For example, a program written in 'C' can have the following format:

```
Comments
Preprocessor directives
Global_variables
main()
{
        local variables
        statements
        ------------
        ------------
func1()
{
```

```
        local variables
        statements
        ------------
        ------------
}
func2()
{
        local variables
        statements
        ------------
        ------------
}
```

A program starts with comments enclosed between /* and */. Comments can be given anywhere in the program.

The preprocessor directives are executed before C program code passes through the compiler. These preprocessor directives make programs more efficient. Most commonly used directives are: # include which includes files, # define which defines the macro name and macro expansion.

For example,

define TRUE 1
define FALSE 0

Here, the preprocessor searches for macro name TRUE and FALSE in the 'C' source code and substitutes 1 and 0 each time it is encountered.

Then there are declarations for global variables, which have same data type and same name throughout the function and are defined outside the main() function. But the declaration of too many global variables is not advisable.

To make the program more efficient **constants** are used. **Constants** are values that can be stored in the memory and can be changed during the execution of program. They can be defined for numeric, character and string data.

Another keyword which can be used is **typedef** which is used for defining new data types. The syntax for typedef is:

typedef type and dataname

Here, type is the data type and dataname is the user defined name.

'C' program contains the main() function but a program should be divided into functions. A function is a self-contained subprogram which performs some specific, well defined task.

Summary

- Software engineering is the study and practice of methods helpful for the construction and maintenance of large software systems.
- Program design involves taking the specifications and designing solutions to the problems accordingly.

- An algorithm is a precise specification of a sequence of instructions to be carried out in order to solve a given problem.
- Analysis of the algorithm is done after determining the running time of an algorithm based on the number of basic operations it performs. The running time varies depending upon the order in which input data is supplied to it.
- Analysis of an algorithm is done on the following basis:
  - Best case time complexity
  - Worst case time complexity
  - Average case time complexity
- Comparison of algorithms is done on the basis of the programming effort done for a program and on the basis of time and space requirements for the program.
- Big 'O' notation is extremely useful for classifying algorithms by their performances.
- The term structured programming was given by Prof. E. Dijkstra and it supports modular design approach. According to this, the program procedures can be built from a combination of three basic types of control structures:
  - Sequential
  - Conditional
  - Iterative
- Recursion is the name given to the phenomenon of defining a function in terms of itself.

Review Exercise

Multiple Choice Questions

1. Algorithm must be
   a. efficient
   b. concise and compact
   c. free of ambiguity
   d. None of these
2. In top-down approach
   a. A problem is subdivided into subproblems.
   b. A problem is tackled from beginning to end in one go.
   c. Subproblems are solved first, then all solutions to subproblems are put together to solve the main problem.
   d. None of the above
3. Modular programming uses
   a. only top-down method
   b. only bottom-up method
   c. both a & b
   d. None of the above

4. Which one of the following is better computing time (in analysis of algorithm)?
   a. 0 (N) b. $0(2^N)$
   c. 0 ($\log_2 N$) d. none of these
5. Which one is not true?
   a. Recursion involves entering an existing block.
   b. Recursion may maintain non-useful total variables and temporary variables through a stack.
   c. In a recursive process stacking and unstacking local variables is done by compiler to ensure that no problem arises.
   d. None of the above

Fill in the Blanks

1. In top-down approach, each subprogram should do _____ task(s) but do it well.
2. Top-down and bottom-up are two methods used for _____ programming.
3. A recursive function can be implemented using _____ data structure.
4. The key tool for the analysis of recursive algorithms is _____.

True or False

1. Recursive functions must be called directly.
2. In top-down approach detailed consideration is postponed.
3. 0(N) linear time is better than 0(1) constant time.
4. Top-down approach uses divide and conquer method.
5. The algorithms should always terminate and handle trivial cases correctly.

Descriptive Questions

1. Define software engineering.
2. Define algorithm and design an algorithm to find out the total number of even and odd numbers in a list of 100 numbers.
3. Explain different ways of analyzing algorithm.
4. Explain the structured approach to programming.
5. What is the need of stack in implementing a recursive function?
6. The greatest common divisor (GCD) of two positive integers is the largest integer that divides both of them. For example, the GCD of 8 and 12 is 4, the GCD of 9 and 18 is 9 and GCD of 16 and 25 is 1.
   a. Write a recursive function GCD (m, y : integer) : integer that implements the division algorithm. If y=0, then GCD of ´ and y is ´; otherwise GCD of Y and ´ **mod** Y.
   b. Rewrite the function in iterative form.
7. Determine what the following recursive 'C' function computes. Write an iterative function to accomplish the same purpose.

```
int func(int n)
{
        if(n==0)
        return 0;
        return(n + func (n-1));
}
```

CHAPTER

3 Arrays

Key Features

- Introduction to Linear and Non-Linear Data Structures
- Arrays in C
- Single Dimensional Arrays
- Array Operations
- Two Dimensional Arrays
- Multidimensional Arrays
- Pointers and Arrays
- An Overview of Pointers

This chapter discusses a class of linear data structures—arrays—and the various operations that can be performed on arrays. An array is an ordered collection of items, all of which are of the same type.

This chapter also explains the storage representation of arrays in row-major and column-major order.

INTRODUCTION TO LINEAR AND NON-LINEAR DATA STRUCTURES

Data structures are categorised into two classes: linear and non-linear.

In a linear data structure, member elements form a sequence. Such linear structures can be represented in memory by using one of the two basic strategies.

- By having the linear relationship between the elements represented by means of sequential memory locations. These linear structures are called **arrays**.
- By having relationship between the elements represented by pointers. These structures are called **linked lists**.

Arrays are useful when the number of elements to be stored is fixed. Operations like traversal, searching and sorting can easily be performed on arrays. On the other hand, linked lists are useful when the number of data items in the collection are likely to change.

There are various non-linear structures, such as, trees and graphs and various operations can be performed on these data structures such as:

Traversal One of the most important operations which involves processing each element in the list.

Searching Searching or finding any element with a given value or the record with a given key.

Insertion Adding a new element to the list.

Deletion Removing an element from the list.

Sorting Arranging the elements in some order.

Merging Combining two lists into a single list.

ARRAYS IN C

Arrays are preferred for situations which require similar type of data items to be stored together.

An array is a **finite** collection of **similar** elements stored in adjacent memory locations. By **finite** we mean that there are specific number of elements in an array and **similar** implies that all the elements in an array are of the same type. For example, an array may contain all integers or all characters.

Thus an array is a collection of variables of the same type that are referred by a common name. The elements of the array are referenced respectively by an **index set** containing **n** consecutive numbers. An array with **n** number of elements is referenced using an index that ranges from **0** to **n–1.** The lowest index of an array is called its **lower bound** and highest index is called the **upper bound**. The number of elements in an array is called its **range**. The elements of an array **arr[n]** containing **n** elements are referenced as arr[0], arr[1], arr[2]..., arr[n–1] where 0 is the lower bound and 'n–1' is the upper bound of the array.

Declaration of an Array in C

An array can be declared just as any other variable in C, i.e. data type followed by array name. The only addition in the declaration is the subscript in bracket which indicates the number of elements it will hold. By declaring an array, the specified number of memory locations (size of array) are allocated in the memory. For example,

```
int age[20];
float sal[10];
char grade[10];
```

The first example is of an integer type array where each element will hold an integer value, second one is the floating type array and the third is the character type array.

The elements of an array can be easily processed as they are stored in contiguous memory locations. For example,

```
int arr[5];
```

This is stored as

100	102	104	106	108
arr[0]	arr[1]	arr[2]	arr[3]	arr[4]

We can easily access and print the values from the array as:

```
scanf("%d", &arr[1]);
printf("%d", arr[1]);
```

An example of declaring and processing the elements of an array is given below:

```
/* To accept 10 numbers and print them. /*
    # include <stdio.h>
    main()
    {
          int arr[10], i;
          {
                for(i=0; i<10; i++)
                printf("Enter the %d number \n", i+1);
                scanf("%d", &arr[i]);
          }
          for(i=0; i<10; i++)
          printf("number %d is %d\n", i+1, arr[i]);
    }
```

Array Initialization

Arrays can be initialized at the time of declaration. For example,

```
int age[5] = [8, 10, 5, 15, 20];
float sal[3] = [2000, 2000.50, 1000];
```

The following values are assigned to each of the elements of the above-mentioned arrays:

Age	[0]	[1]	[2]	[3]	[4]
Value →	8	10	5	15	20
Address →	100	102	104	106	108

Sal	[0]	[1]	[2]
Value →	2000	2000.50	1000
Address →	1000	1004	1008

An array of characters is called a **string** and it is terminated by a null ('0') character. For example,

```
char name[5] = "Geeta";
```

The values assigned in the **name** array are as follows:

Name	[0]	[1]	[2]	[3]	[4]
Value →	G	e	e	t	a
Address →	100	101	102	103	104

The character type array can also be initialized as:

```
char name[ ] = "Geeta";
char name[ ] = {'G', 'e', 'e', 't', 'a'};
```

SINGLE DIMENSIONAL ARRAYS

A single dimensional array is the simplest form of an array. The array is given a name and its elements are referred to by their subscripts or indices. A one-dimensional array is used to store a large number of items in memory. It references all the items in a uniform manner.

Representation of Linear Arrays in Memory

The memory of computer is simply a sequence of addressed locations as shown in Fig. 3.1. Let **arr** be a linear array stored in the memory of computer.

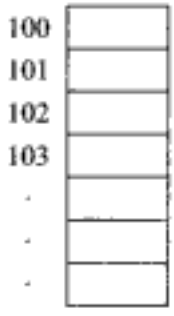


Fig. 3.1 *Computer Memory*

Add(arr[k]) = address of the arr[k] element of the array **arr**

As the elements of the array **arr** are stored in consecutive memory cells, the computer does not need to keep track of the address of every element of the array. It only needs to keep track of the address of the first element of the array which is denoted by:

Base(arr)

It is called the base address of the **arr**. Using this base address Base(arr), the computer calculates the address of any element of the array by using the following formula:

$$\text{Add(arr[k])} = \text{Base(arr)} + w(\text{k-lower bound})$$

where **w** is the size of the data type of the array **arr**. Observe that the time to calculate Add(arr[k]) is essentially the same for any value of k. Furthermore, given any subscript k, one can locate and access the contents of arr[k] by scanning any other element of the array.

ARRAY OPERATIONS

Traversal

Let **A** be a collection of data elements stored in the computer's memory. To print the contents of each element of A or count the number of elements of A with a given property, each element of A will have to be accessed or processed at least once. This is called **traversing**.

Algorithm for Array Traversal

Let **A** be a linear array with lower bound LB and upper bound UB. The following algorithm traverses **A** applying an operation **PROCESS** to each element of **A**.

```
1.    Initialize counter
            Set counter := LB
2.    Repeat steps 3 and 4 while counter ≤ UB
3.    Visit element
            Apply PROCESS to arr[counter]
4.    Increase counter
            Set counter = counter + 1
5.    Exit
```

Insertion and Deletion

Insertion refers to the operation of adding another element to the array and **deletion** refers to the operation of removing an element from the array.

If an element is to inserted at the end of the array, then the task is easily done, provided, there is enough space to accommodate additional elements in the array.

But if we need to insert an element in the middle of an array then half of the elements must be moved downwards to new locations to accommodate the new element and retain the order of other elements.

Similarly, deleting the element from the end of an array is not a problem, but deleting from the middle requires movement of each element upwards in order to fill up the void in the array.

Algorithm for Insertion

Let A be a linear array—the function used is INSERT(A, N, K, ITEM). N is the number of items, K is the positive integer such that $K \leq N$. The following algorithm inserts an element ITEM into the K^{th} position of array A.

```
1.  Initialize Counter
        Set J := N
2.  Repeat Steps 3 and 4 while J ≥ K
3.  Move J[th] element downward
        Set A[J+1] := A[J]
4.  Decrease Counter
        Set J := J-1
        End of step 2 loop
5.  Insert element
        Set A[K] := ITEM
6.  Reset N
        Set N := N+1
7.  Exit
```

Algorithm for Deletion

Let A be a linear array. The function used to delete from the array is

DELETE(A, N, K, ITEM)

where N is the number of elements, K is the positive integer such that $K \leq N$. The algorithm deletes K^{th} element from the array.

```
1.  Set ITEM := A[K]
2.  Repeat for J = K to N-1;
        [Move J+1 element upward]
            Set A[J] := A[J+1]
        End of loop
3.  Reset the number N of elements in A
        Set N := N-1
4.  Exit
```

/* Program to perform array operations. */

```
#include<stdio.h>
#include<conio.h>
#define MAX 5

void insert(int *, int pos, int num);
void del(int *, int pos);
void reverse(int *);
void display(int *);
void search(int *, int num);

void main( )
{
      int arr[5] ;

      clrscr( ) ;

          insert(arr, 1, 11);
          insert(arr, 2, 12);
          insert(arr, 3, 13);
          insert(arr, 4, 14);
          insert(arr, 5, 15);

          printf("\nElements of Array: ");
          display(arr);

          del(arr, 5);
          del(arr, 2);

          printf("\n\nAfter deletion: ");
          display(arr);

          insert(arr, 2, 222) ;
          insert(arr, 5, 555) ;
          printf("\n\nAfter insertion: ") ;
          display(arr) ;
          reverse(arr) ;
          printf("\n\nAfter reversing: ");
          display(arr);
          search(arr, 222) ;
          search(arr, 666) ;
```

```
        getch( ) ;
    }

    /* Inserts an element num at given position pos */
    void insert(int *arr, int pos, int num)
    {
        /* shift elements to right */
        int i ;
        for(i = MAX - 1 ; i >= pos ; i--)
              arr[i] = arr[i - 1] ;
        arr[i] = num ;
    }

    /* Deletes an element from the given position pos */
    void del(int *arr, int pos)
    {
        /* skip to the desired position */
        int i ;
        for(i = pos ; i < MAX ; i++)
        arr[i - 1] = arr[i] ;
        arr[i - 1] = 0 ;
    }

/* Reverses the entire array */
void reverse(int *arr)
{
    int i ;
    for(i = 0 ; i < MAX / 2 ; i++)
    {
        int temp = arr[i] ;
        arr[i] = arr[MAX - 1 - i] ;
        arr[MAX - 1 - i] = temp ;
    }
}

/* Searches array for a given element num */
void search(int *arr, int num)
{
    /* Traverse the array */
    int i ;
    for(i = 0 ; i < MAX ; i++)
    {
```

```
                if(arr[i] == num)
                {
                printf("\n\n The element %d is present at %dth position.",
                num, i + 1) ;

                return ;
                }
        }

        if(i == MAX)
        printf("\n\n The element %d is not present in the array.", num) ;
}

/* Displays the contents of array */
void display(int *arr)
{
        /* traverse the entire array */
        int i ;
            printf("\n") ;
        for(i = 0 ; i < MAX ; i++)
            printf("%d\t", arr[i]) ;
}
```

Sorting

Sorting refers to the operation of rearranging the elements of an array in some specified order. There are many different sorting algorithms. A very simple sorting algorithm known as **bubble sort** has been discussed here.

The algorithm for bubble sort is as follows:

Algorithm for Bubble Sort

Let **arr** be an array of N elements. The following algorithm sorts the elements of **arr**:

1. Repeat steps 2 and 3 for K=1 to N-1
2. Set PTR := 1 [Initializes pass pointer PTR]
3. Repeat while PTR ≤ N-K : [Execute Pass]
   a. If arr[PTR] > arr[PTR+1], then :
      Interchange arr[PTR] and arr[PTR+1]
      [End of if structure]
   b. Set PTR := PTR+1
      [End of inner loop]
      [End of step1 outer loop]
4. Exit.

/*Program to accept numbers and sort them in ascending order using bubble sort*/

```
# include<stdio.h>
main()
{
        int n,i,j,arr[10],temp;
        printf("Enter how many numbers :");
        scanf("%d",&n);
        for(i=0;i<n;i++)
        {
                printf(" enter the numbers%d :",i+1);
                scanf("%d",&arr[i]);
        }
        for(i=0;i<n;i++)
        for(j=0;j<n;j++)
        {
                if(arr[j]>arr[j+1])
                {
                        temp=arr[j];
                        arr[j]=arr[j+1];
                        arr[j+1]=temp;
                }
        }
        printf("Sorted numbers are: \n" );
        for(i=0;i<n;i++)
        printf("%d--",arr[i]);
        printf("\n");
}

/ If the entered numbers are

23      15      29      11      1

the program will show the following output:

Sorted numbers are:
1       11      15      23      29
```

Searching

Searching refers to the operation of finding the location of any item in the array. The search is said to be successful if the item is found; otherwise it is unsuccessful. There are many different searching algorithms. The algorithm one chooses generally depends on the way the input data is organized.

Linear Search Suppose **arr** is a linear array with **n** elements. Given no other information about **arr**, the simplest way to search for a given item in **arr** is to compare the item with each element in **arr** one by one. That is, first we test arr[1] = item, and then we test arr[2] = item, and so on. This method which traverses the data sequentially is called **linear** or **sequential search**.

The algorithm for linear search is as follows:

Algorithm for Linear Search

Suppose **arr** is a linear array with **N** elements, and **item** is the given item of information. This algorithm finds the location **LOC** of **item** in **arr** or sets **LOC=0** if search is unsuccessful.

1. [Insert ITEM at the end of arr]
   Set arr[N+1] := ITEM
2. [INITIALIZE COUNTER]
   SET LOC := 1
3. [Search for ITEM]
   Repeat while arr[LOC] ≠ ITEM
   Set LOC := LOC+1
   [End of loop]
4. [Successful ?]
   If LOC = N+1 then : Set LOC := 0
5. Exit.

/* Program for linear search in an array. */

```
# include<stdio.h>
# include<conio.h>
void main( )
{
   int arr[10] = {1. 2. 3. 9. 11. 13. 17. 25. 57. 90} ;
   int i. num ;

   clrscr() ;

   printf("Enter number to search: ") ;
   scanf("%d". &num) ;

   for(i = 0 ; i <= 9 ; i++)
   {
         if(arr[9] < num || arr[i] >= num)
         {
               if(arr[i] == num)
                     printf("The number is at position %d in the array.", i);
               else
                     printf("Number is not present in the array.") ;
               break ;
```

```
        }
    }
    getch( ) ;
}
```

TWO-DIMENSIONAL ARRAYS

A two-dimensional array is a collection of elements placed in **m** rows and **n** columns. There are two subscripts in the syntax of 2-D array in which one specifies the number of rows and the other the number of columns. In a two-dimensional array each element is itself an array.

The two-dimensional array is also called a **matrix**. Mathematically, a matrix A is a collection of elements 'a_{ij}' for all i and j's such that $0 \leq i<m$ and $0<j \leq n$. A matrix is said to be of order $m \times n$. A matrix can be conveniently represented by a two-dimensional array. The various operations that can be performed on a matrix are: addition, multiplication, transposition, and finding the determinant of the matrix.

An example of 2D array can be arr[2][3] containing 2 rows and 3 columns and arr[0][1] is an element placed at 0^{th} row and 1^{st} column in the array.

A two-dimensional array can thus be represented as given in Fig. 3.2.

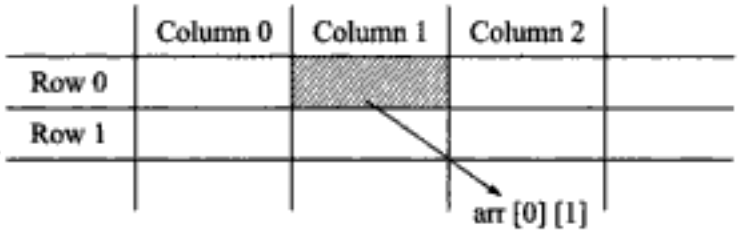


Fig. 3.2 *Representation of 2-D array in memory*

A two-dimensional array differentiates between the logical and physical view of data. A 2-D array is a logical data structure that is useful in programming and problem solving. For example, such an array is useful in describing an object that is physically two-dimensional such as a map or checkerboard.

Row Major and Column Major Order

All elements of a matrix get stored in the memory in a linear fashion. The two ways in which elements can be represented in computer's memory are—Row Major Order and Column Major Order. In **row-major representation** the first row of the array occupies the first set of memory locations, second occupies the next set, and so on.

The diagrammatic representation of two-dimensional array **arr[2][3]** in row major order is given in Fig. 3.3.

In Fig. 3.3, the memory locations are first occupied by the first row of the array. Now base(arr) has the address of first element of the array arr[0][0]. We also assume the size of each element in the array with **esize**.

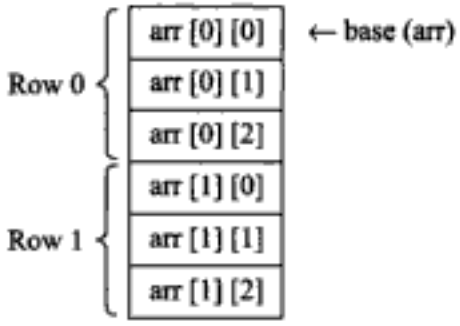


Fig. 3.3 *Representation of 2-D Array in Row Major Order*

Now let us calculate the address of an element in the following 2-D array:

int matrix [3][4] = { {1, 2, 3, 4}, {5, 6, 7, 8}, {9, 10, 11, 12} };

Formula to calculate the $(j, k)^{th}$ element of a 2-D array of m × n dimension is

$$A(j, k) = base(A) + W[N(j-1) + (k-1)]$$

where W = word size
N = number of columns

Let us assume that the base address of the array matrix is 100. Since W=2 (as array is of integer type whose size is 2), therefore, according to the formula, address of $(2, 3)^{th}$ element in the array matrix will be

$$\begin{aligned} LOC(2, 3) &= 100 + 2[4(2-1) + (3-1)] \\ &= 100 + (4 + 2) * 2 \\ &= 100 + 12 \\ &= 112 \end{aligned}$$

[Note: Lower bound of the array is assumed to be 1.]

Thus we see that in the above matrix, address of $(2, 3)^{th}$ element which is 7 is 112 (as depicted in the Fig. 3.4).

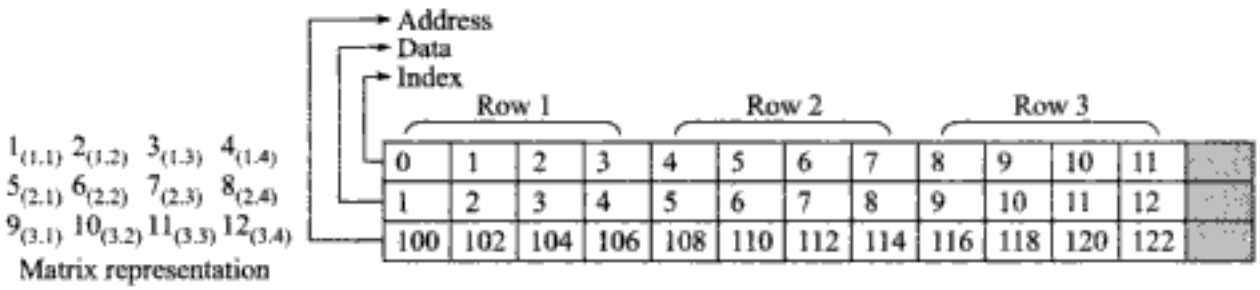


Fig. 3.4 *Row Major Representation*

Similarly, for the column major order representation, let us consider the same matrix.

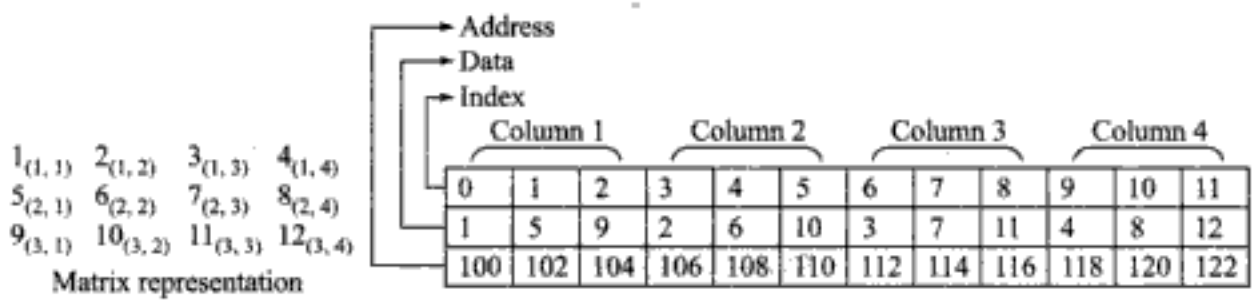


Fig. 3.5 *Column Major Representation*

To find the address of $(j, k)^{th}$ element in an array of m × n dimension and W word size of each element, the following formula can be used:

$$A(j, k) = Base(A) + W[M(k-1) + (j-1)]$$

$$\therefore \quad LOC(2, 3) = 100 + 2[3(3-1) + (2-1)]$$
$$= 100 + 2(7)$$
$$= 114$$

We see that the address of the same element ($(2, 3)^{th}$ i.e., 7) has changed to 114 instead of 112, in column major order representation of the array. This is due to the fact that the elements in the column major order representation are stored in a different order, i.e. all the elements of column 1 are stored first and then all the elements of column 2 are stored, followed by all the elements in column 3, and so on.

Matrix as a 2-D Array

A matrix can be represented by a two-dimensional array. A matrix 'A' is a collection of elements 'a_{ij}'.

The matrix is said to be of order m × n, i.e. it consists of 'm' number of rows and 'n' columns. The element a_{ij} is the element positioned at the intersection of i^{th} row and j^{th} column. When the number of rows and number of columns are equal, then the matrix 'A' is called **square matrix.**

The common matrix operations are:

- Addition of two matrices
- Multiplication of two matrices
- Transposition of a given matrix
- Evaluation of determinant of square matrix

/* Program for addition of two matrices*/

```
#include<stdio.h>
int mat1[3][3], mat2[3][3], mat3[3][3], i, j, row, col;
void main()
 {
   printf("Hello");
   printf("Enter the row of the matrix :");
   scanf("%d",&row);
   printf("Enter the column of the matrix :");
   scanf("%d",&col);
   printf("For first matrix :\n");
   for(i=0;i<row;i++)
   for(j=0;j<col;j++)
   {
      printf("Enter the number for row%d, column%d: ", i+1,j+1);
      scanf("%d",&mat1[i][j]);
   }
   printf("For second matrix :\n");
   for(i=0;i<row;i++)
   for(j=0;j<col;j++)
   {
      printf("Enter the number for row%d, column%d: " ,i+1,j+1);
      scanf("%d",&mat2[i][j]);
   }
```

```
    /*for addition*/
    for(i=0;i<row;i++)
    for(j=0;j<col;j++)
    mat3[i][j]=mat1[i][j]+mat2[i][j];

    /*for printing*/
    printf("Matrix 1 is:\n");
    for(i=0;i<row;i++)
    {
        for(j=0;j<col;j++)
        printf("%d\t",mat1[i][j]);
        printf("\n");
    }
    printf(" Matrix 2 is:\n");
    for(i=0;i<row;i++)
    {
        for(j=0;j<col;j++)
        printf("%d\t",mat2[i][j]);
        printf("\n");
    }
    printf("After addition matrix is:\n");
    for(i=0;i<row;i++)
    {
        for(j=0;j<col;j++)
        printf("%d\t", mat3[i][j]);
        printf("\n");
    }
 }
```

/*Program for multiplication of two matrices*/

```
# include<stdio.h>
main()
 {
    int mat1[3][3],mat2[3][3],mat3[3][3],i,j,row1,col1,row2,col2,k;
    goto begin;
    printf("\nEnter the row of first matrix ");
    scanf("%d",&row1);
    printf("\nEnter the column of first matrix ");
    scanf("%d",&col1);
    printf("\nEnter the row of second matrix ");
    scanf("%d",&row2);
    printf("\nEnter the column of second matrix ");
    scanf("%d",&col2);
```

```
    if(col1 != row2)
    goto begin:
    printf("\nCreation of first matrix: ");
    creatmat(mat1,row1,col1);
    printf("\nCreation of second matrix:");
    creatmat(mat2,row2,col2);
    for(i=0;i<row1;i++)
    for(j=0;j<col2;j++)
    {
        mat3[i][j]=0 ;
        for(k=0;k<col1;k++)
        mat3[i][j]+=mat1[i][j] * mat2[k][j];
    }
    printf("\nMatrix 1 is:\n");
    printmat(mat1,row1,col1);
    printf("\nMatrix 2 is:\n");
    printmat(mat2,row2,col2);
    printf("\nThe Resultant Matrix 3 is:\n");
    printmat(mat3,row1,col2);
 }

creatmat(mat,p,m)
int mat[3][3],p,m;
 {
    int i,j;
    for(i=0;i<p;i++)
    for(j=0;j<m;j++)
    {
        printf("\nEnter the number for row%d, column%d :", i+1,j+1);
        scanf("%d", &mat[i][j]);
    }
    return;
 }
void printmat(mat,p,m)
int mat[3][3],p,m;
 {
    int i,j;
    for(i=0;i<p;i++)
    {
        for(j=0;j<m;j++)
        printf("\n%d\t",mat[i][j]);
        printf("\n");
```

```
    }
  }
```

/*Program for transposing matrix*/

```
# include<stdio.h>
int mat1 [3][3],mat2[3][3],i,j,row,col;
void main()
 {
   printf(" Enter the row of the matrix :");
   scanf("%d",&row);
   printf("Enter the column of the matrix :");
   scanf("%d",&col);
   for(i=0;i<row;i++)
   for(j=0;j<col;j++)
   {
      printf("Enter the number for row%d, column%d: ",i+1,j+1 );
      scanf("%d",&mat1[i][j]);
   }
   printf(" Matrix is :\n");
   for(i=0;i<row;i++)
   {
      for(j=0;j<col;j++)
            printf("%d\t",mat1[i][j]);
            printf("\n");
   }
   for(i=0;i<row;i++)
   for(j=0;j<col;j++)
      mat2[i][j]=mat1[j][i];
      printf("Transpose of the matrix :\n");
   for(i=0;i<row;i++)
   {
      for(j=0;j<col;j++)
            printf("%d\t",mat2[i][j]);
            printf("\n");
   }
 }
```

MULTIDIMENSIONAL ARRAYS

The arrays can also have more than two dimensions. For example, a three-dimensional array may be declared as.

int arr[3][2][4];

An element of this array is referenced by three subscripts. The first specifies the plane number, the second specifies the row number and the third the column number. Such an array is useful when the value is determined by three inputs.

The number of elements in any array is the product of the ranges of all its dimensions. For example, **arr** contains 3 * 2 * 4 =24 elements. The arr[3][2][4] can be represented as shown in Fig. 3.6.

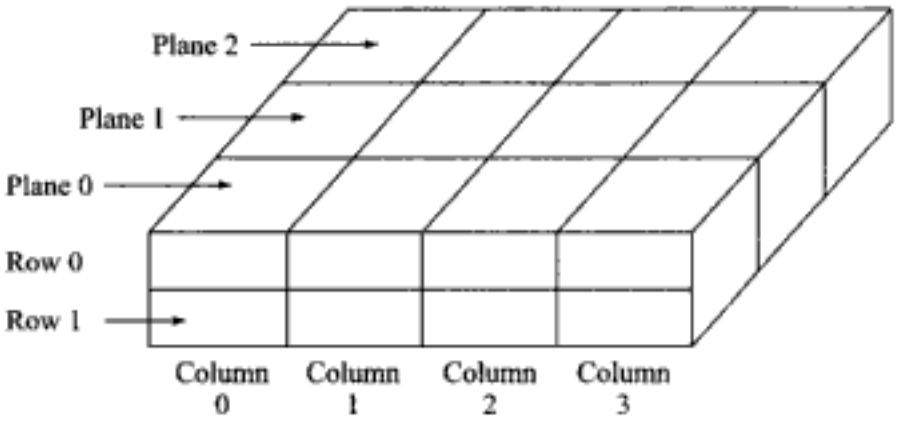
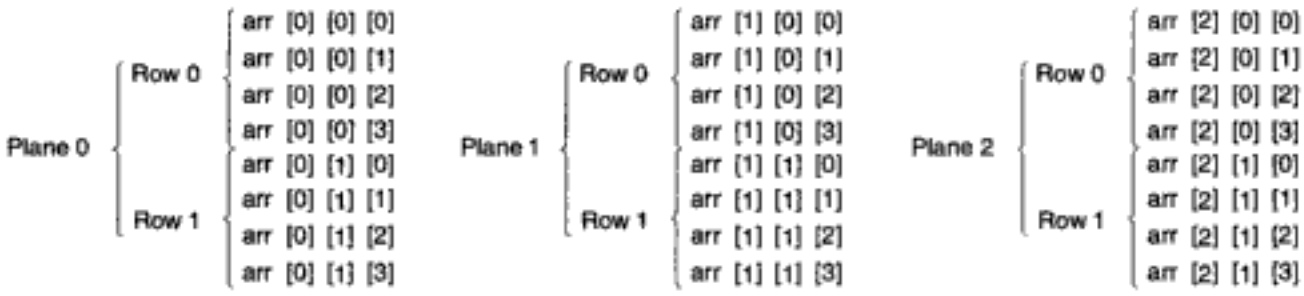


Fig. 3.6 *Representation of arr[3][2][4]*

Therefore, row major representation of the elements of arr[3][2][4] is as shown below:



It can be observed that, the last subscript varies most rapidly, and a subscript is not increased until all possible combinations of the subscript to its right have been exhausted.

POINTERS AND ARRAYS

Pointers are special variables which contain the address of another memory location. The name of the array is synonymous with a pointer to the memory location of the first element in the array. Pointers are useful in accessing any memory location directly. An address of a memory location is a long integer, which is stored in a pointer type variable. Pointers also allow arithmetic operations except subtraction, division, and multiplication between two operands of pointer type. This arithmetic property is useful while accessing a list of elements stored in a sequential location. We get nothing relevant by adding two pointers but subtraction of two pointers (if the result is positive) gives us the number of bytes available between two addresses.

On array declaration, sufficient amount of storage is allocated by the compiler to store all the elements of the array. The compiler also defines the name of the array as a pointer to the first element. For example, if an array **arr** is declared as follows:

```
int arr[4] = {2, 4, 6, 8};
```

The elements will be stored as follows:

arr[0]	arr[1]	arr[2]	arr[3]
2	4	6	8
100	102	104	106

The name **arr** acts as a constant pointer pointing to the first element, i.e. arr[0] of the array. Therefore, the value of **arr** is equal to the address where arr [0] is stored, i.e.

```
arr = &arr[0] = 100
```

An integer pointer can be made to point to the array **arr** by the following assignment:

```
ip = arr
ip = &arr[0]
```

When the pointer variable is incremented by 1 then it points to the next address which is equal to the base address +2 because pointer is of type int. Now, the value of each element of **arr** can be accessed using ip++ as shown below:

```
ip = &arr[0]     (=100)
ip+1 = &arr[1]   (=102)
ip+2 = &arr[2]   (=104)
```

So, the address of any element can be calculated by using its index and the size of the data type. For example,

```
address of arr[2] = base address + (2 * size of int)
                  = 100 + (2 * 2)
                  = 104
```

We can use pointers to access array elements. For getting the value of arr[2], *(ip+2) can be used.

AN OVERVIEW OF POINTERS

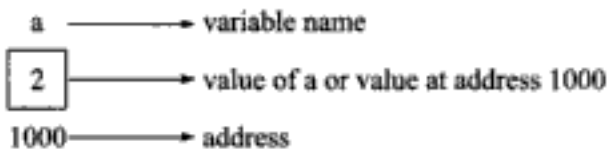
A **pointer** is a variable which contains the address of some memory location. Pointers are popular in programming because they

- provide the means through which the memory location of a variable can be directly accessed and manipulated as required;
- support dynamic allocation routines; and
- can improve the efficiency of certain routines.

Points to be Remembered while working with Pointers

- The '&' is the address operator, it represents the address of the variable.
- The **%u** is used for obtaining the address.
  For example,

```
main()
{
        int a=2;
        printf("value of a = %d\n", a);
        printf("address of a = %u\n", &a);
}
```



- The '*' operator is the value at address operator. It gives the value at specified address. Considering the above example we can use it as:

  printf("value at address %u = %d\n", &a, (*&a));

  So, we can say that the address of variable '**a**' preceded by * , gives the value at that address.
- When a pointer variable is declared, an '*' symbol should precede the variable name.
  For example,

  int *b; char *p; float *q;

- The address of a variable can be assigned to another variable.
  For example,

```
int *b;
b = &a;
```

  Here, **b** is the variable which contains the address of variable **a** as its value.
- Another pointer variable can store the address of a pointer variable.
  For example,

```
int a = 2;
int *b;
int **c;
b = &a;
c = &b;
```

  In the example given above, **c** has been declared as a pointer to pointer variable which contains the address of pointer variable **b**.
- Various arithmetic operations can be performed on pointers—postfix, prefix, increment, decrement. In pointer arithmetic, all pointers increase or decrease by the length of the data type they point to.

- A pointer holds the address of the very first byte of the memory location where it is pointing to. The address of the first byte is known as the base address. Therefore, adding 1 to a pointer actually adds the size of pointer's base type.
  For example,

```
int a = 2;
int *b;
b = &a
```

The variables of the example are represented as shown below

Therefore, b++ = 1000+2
The variable **b** contains the base address, i.e. the address of **a**, therefore, when increment or decrement operation is performed, the address of **a** is incremented or decremented, i.e. 1000+2 or 1000–2.
So, when a number is added to a pointer variable, the "bytes" that pointer data type holds are added number of times to the pointer variable.
For example,

```
int a = 2;
in *b;
b = &a;
b = b+1;
b = b+2
```

So, b = b+1
= 1000 + 1 * 2
= 1000 + 2 = 1002
Similarly, b = b+2 = 1000 + 2 * 2 = 1004
The same procedure is followed when a number is subtracted from a pointer variable.
b = b – 1 = 1000 – 2 = 998
b = b – 3 = 1000 – 3 * 2
= 1000 – 6 = 994

- Pointers can also be used in handling functions. The arguments are passed to the functions in two ways:
  – call by value
  – call by reference

Call by Value The process of passing the actual values of variables as arguments to a function is called **call by value.**

For example,

```
main()
{
    int a=2;
    int b=1;
    printf("Before calling the function, a and b are %d, %d\n", a, b);
    value(a, b);
    printf("After calling the function, a and b are %d, %d\n" , a, b);
}
value(p, q)
int p, q;
{
    p++;
    q++;
    printf("In function changes are %d%d\n", p, q);
}
```

Note the addresses of variables at different stages of the program:

Before calling function

a	b
2	1
1000	2000

In function

p	q
2	1
5000	8000

In incrementing

p	q
3	2
5000	8000

After calling function

a	b
2	1
1000	2000

The values of the variables **a** and **b** are not altered because when the function **value** is called, the values of the variables **a** and **b**, not their addresses, are passed into the function.

Call by Reference In call by reference, we pass the addresses of the variables as parameters to the function. The addresses passed to the variables are pointer variables, therefore, values at the addresses are incremented. Thus, the function which is called by 'reference' can change the value of the variable used in the call.

For example,

```
main()
{
        int a=2;
        int b=1;
        printf("Before calling the function, a and be are %d%d\n", a, b);
        ref(&a, &b);
        printf("After calling the function, a and b are %d%d\n", a, b);
}
ref(p, q)
int *p, *q;
{
        (*p)++;
        (*q)++;
        printf("In function changes are %d, %d\n", *p, *q);
}
```

The addresses of the variables at different stages of the program are given below:

Before calling function

a	b
2	1
1000	2000

In function

p	q
1000	2000
5000	8000

After (*p++) and (*q++)

p	q
1000	2000
5000	8000

After calling the function

p	q
3	2
1000	2000

Therefore, when addresses of a and b are passed to the parameters p and q, which are the pointer variables, the values at these addresses are incremented.

Summary

- Data structures are classified as linear and non-linear.
- An array is a finite collection of similar elements stored in adjacent memory locations.
- There are many operations which could be performed on arrays—insertion, deletion, searching, sorting, traversing.
- Arrays can be single-dimensional, two-dimensional, or multi-dimensional.
- There are two ways of representing two-dimensional arrays in memory—row-major order, and column-major order.
- Multidimensional arrays have more than two dimensions.
- Pointers are special variables which contain address of another memory location.
- There is a close relationship between pointers and arrays.
- The C compiler defines the name of the array as a pointer to the first element of the array.

Review Exercise

Multiple Choice Questions

1. Elements of an array are accessed by
   a. accessing function in built-in data structure
   b. mathematical function
   c. index
   d. none of the above
2. Array is a
   a. linear data structure
   b. non-linear data structure
   c. complex data structure
   d. none of the above
3. Row-major order in 2-Dimensional array refers to an arrangement where
   a. all elements of a row are stored in memory in sequence followed by next row in sequence and so on.
   b. all elements of a row are stored in memory in sequence followed by next column in sequence and so on.
   c. all elements of a column are stored in memory in sequence followed by next column in sequence.
   d. none of the above.
4. Array is
   a. data in physical order
   b. data in logical order
   c. both a & b
   d. none of the above

Fill in the Blanks

1. Array has elements of _____ data type.
2. Array may be accessed by _____.
3. If elements of a row are stored next to one another, the array is said to be stored in _____ order.
4. A data structure is said to be _____ if its elements form a sequence.
5. An array is a collection of _____ elements stored in _____ memory locations.
6. Index of an array containing 'n' elements varies from _____ to _____.
7. A 2-D array is also called _____.

Answer the Following Questions

1. Write a 'C' function to find out the maximum and second maximum number from an array of integers.
2. The median of an array of numbers is the element m of the array such that half the remaining numbers in the array are greater than or equal to 'm' and half are less than or equal to 'm', if the number of elements in the array are odd. If the number of elements is even, the median is the average of two elements m_1 and m_2 such that half of the remaining elements are greater than or equal to m_1 and m_2 and half of the elements are less than or equal to m_1 and m_2. Write a 'C' function that accepts an array of numbers and returns the median of the numbers in the array.
3. Write a 'C' function to find out whether there is an element 'a_{ij}' in an $m \times n$ matrix 'A' of numbers such that 'a_{ij}' is the smallest value in the i^{th} row and largest value in the j^{th} column. How many comparisons does your function make?
4. Suppose an array is declared as follows:

   $$\text{char } A[U_1][U_2] \ldots [U_m]$$

   If this array is stored in column major fashion then what is the addressing formula for the element $A[i_1][i_2] \ldots [i_m]$?
5. A square matrix is called symmetric if for all values of i and j a[i][j] = a[j][i]. Write a program which verifies whether a given 5×5 matrix is symmetric or not.
6. There are two arrays A and B. A contains 25 elements, whereas B contains 30 elements. Write a function to create an array C that contains only those elements that are common to A and B.

CHAPTER

4 Linked Lists

Key Features

- Introduction to Lists and Linked Lists
- Dynamic Memory Allocation
- Basic Linked List Operations
- Doubly Linked List
- Circular Linked List
- Atomic Node Linked List
- Linked List in Arrays
- Linked Lists Versus Arrays

A list can be defined as a collection of elements. We can add, search, or delete elements in a list. The list is maintained in two ways—through arrays and linked lists.

This chapter discusses the concepts of linked list which is a linear collection of data elements called nodes, each pointing to the next node by means of pointers. This chapter also explains different types of linked lists like doubly and circular linked lists and the various operations possible on these linked lists.

Linked lists overcome the drawbacks of arrays. In a linked list the number of elements need not be predetermined, more memory can be allocated and released during processing (Dynamic memory allocation). Insertions and deletions are easier to make.

INTRODUCTION TO LISTS AND LINKED LISTS

'List' is a term used to refer to a linear collection of data items. Data processing involves storing and processing data organized as a list. A list can be implemented either by using arrays or linked lists. In arrays there is a linear relationship between the data elements which is evident from the physical relationship of data in the memory. The address of any element in the array can easily be computed but, it is very difficult to insert and delete any element in an array. Usually, a large block of memory is occupied by an array which may not be in use and it is difficult to increase the size of an array, if required.

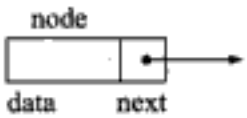
Another way of storing a list is to have each element in a list contain a field called a **link** or **pointer**, which contains the address of the next element in the list. The successive elements in the list need not occupy adjacent space in memory. This type of data structure is called a **linked list**.

Linked List

Linked list is the most commonly used data structure used to store similar type of data in memory. The elements of a linked list are not stored in adjacent memory locations as in arrays.

It is a linear collection of data elements, called **nodes**, where the linear order is implemented by means of **pointers**. A linked list allocates memory for storing list elements and connects elements together using pointers.

In a linear or single-linked list, a node is connected to the next node by a single link. A node in this type of linked list contains two types of fields—**data**, which holds a list element, and **next** which stores a link (i.e. pointer) to the next node in the list. A pointer to the head of the list is used to gain access to the list itself and the end of list is denoted by a **NULL** pointer.



The structure defined for a single linked list is implemented as follows:

```
struct node
{       int data;
        struct node * next;
}
```

The structure declared for linear linked list holds two members — an integer type variable '**data**' which holds the elements and another member of type '**node**', which has the variable **next**, which stores the address of the next node in the list.

DYNAMIC MEMORY ALLOCATION

C language requires us to specify the number of elements in an array at compile time. This may cause wastage of memory space. Such situations can be taken care of by using **dynamic data structures**.

Dynamic memory management techniques allow us to allocate additional memory space or to release unwanted space at run time, thus, optimizing the use of storage space.

The memory management functions that can be used for allocating and freeing memory during program execution are:

- malloc — Allocates requested size of bytes
- calloc — Allocates space for an array of elements
- free — Frees previously allocated space
- realloc — Modifies the size of previously allocated space.

Allocating a Block of Memory

The **malloc** function can be used to allocate a block of memory. It reserves a specified size of memory and returns a pointer of type **void** and takes the following form:

ptr = (cast-type *) malloc (byte-size)

The **malloc** returns a pointer (of **cast type**) to an area of memory with size **byte-size**. For example,

y=(int*) **malloc** (100 * sizeof(int));

In the statement given above, space '100 times the size of an **int**' bytes is reserved and the pointer **x** is assigned the address of the first byte of the allocated memory.

Allocating Multiple Blocks of Memory

The **calloc** function is used for seeking memory space for storing derived data types at run time. Multiple blocks of storage are allocated using the **calloc** function. The **calloc** takes the following form:

ptr = (cast-type *) **calloc** (n, elem-size)

Contiguous space for **n** blocks, each of size **elem-size** bytes is allocated.

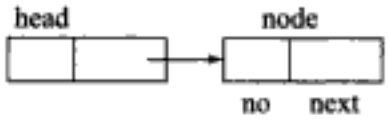
BASIC LINKED LIST OPERATIONS

Creating a Linked List

Linked list is used to avoid any reference to specific number of items in the list so that insertion and deletion is easily done. This can be achieved by using unnamed locations to store nodes. These can be created by using pointers and dynamic memory allocation functions such as **malloc**. The **head** pointer is used to create and access unnamed nodes. Notice the following code fragment:

```
struct linked_list
{
        int no;
        struct linked_list *next;
};
typedef struct linked_list node;
node *head;
head = (node*) malloc (size of (node));
```

The above statement obtains memory to store a node and assigns its address to **head** which is a pointer variable.



To store values in the member fields use the following statements:

head → no = 10;
head → next = NULL;

The second node can be added as follows:

head → next = (node*) malloc (size_of (node));
head → next → number = 20;
head → next → next = NULL;

Inserting an Element

Insertion in a linked list can be done in the following two ways:

Insertion at the Beginning of the List Suppose we already have 5 nodes in the list and wish to insert a new node in the beginning of the list.

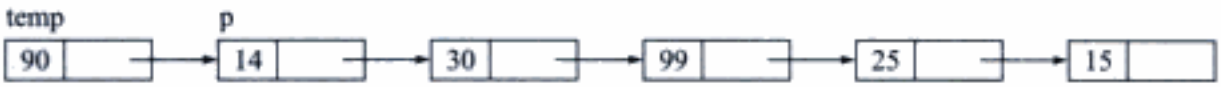


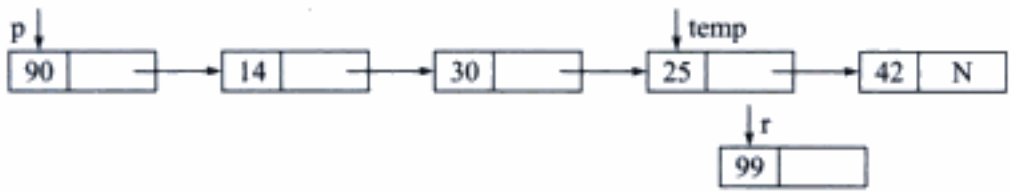
Fig. 4.1 *Insertion at the Beginning of the List*

/*Function to insert a new node at the beginning of the linked list*/

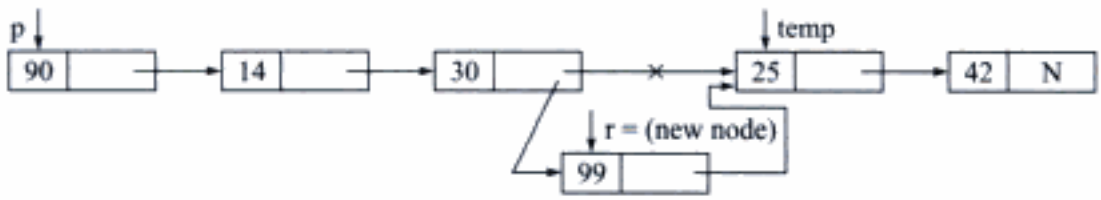
```
void add_beg(struct node **q, int no)
{
        struct node *temp ;

        /* add new node */
        temp = malloc(sizeof(struct node)) ;

        temp -> data = no ;
        temp -> next = *q ;t
        *q = temp ;
}
```

From the figure given above it can be seen that a **temp** variable of structure type is taken and space is allocated for this element using 'malloc' function and its data part contains the element or number and its link points to the existing first node.

Insertion After any Specified Node Suppose we have five nodes in the list and we wish to insert a node after the third node in the list. The process of element insertion is depicted in Fig. 4.2.



(a) Before insertion



(b) After insertion

Fig. 4.2 *Insertion After any Specified Node*

/*Function to insert a new node after the specified node*/

```
void add_after(struct node *q, int loc, int no)
{
        struct node *temp, *r ;
```

```
        int i ;

        temp = q ;
        /* Skip to desired portion */
        for(i = 0 ; i < loc ; i++)
        {
                temp = temp -> next ;
                /* If end of linked list is encountered */
                if(temp == NULL)
                {
                        printf("\nThere are less than %d elements in list", loc ) ;
                        return ;
                }
        }
        /*Insert new node */
        r = malloc(sizeof(struct node)) ;
        r -> data = no ;
        r -> next = temp -> next ;
        temp -> next = r ;
}
```

We begin with a loop and skip the desired number of nodes after which a new node is to be added. Suppose, we wish to add a new node containing data 99 after the third node in the list. The position of pointers, once the control reaches outside the loop, is shown in Fig. 4.2. Now space is allocated for the node to be inserted and 99 is stored in the data part of it.

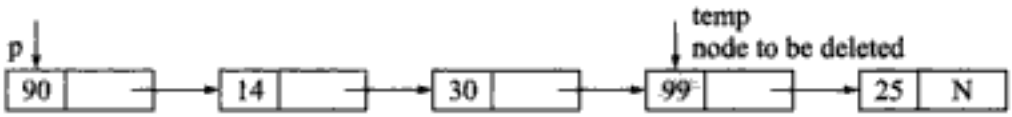
/* Function to insert a node at the end of a linked list */

```
void creat(struct node **q, int no)
{
        struct node *temp, *r ;

        if(*q == NULL)/* If the list is empty, create first node */
        {
                temp = malloc(sizeof(struct node));
                temp -> data = no;
                temp -> next = NULL;
                *q = temp ;
        }
        else
        {
                temp = *q ;

                /* go to last node */
```

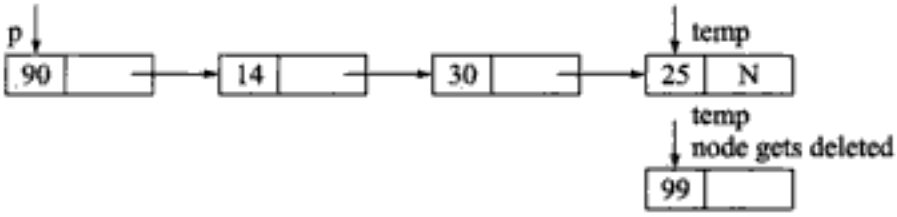
```
            while(temp -> next != NULL)
                  temp = temp -> next ;

            /* Add node at the end */
            r = malloc(sizeof(struct node)) ;
            r -> data = no;
            r -> next = NULL ;
            temp -> next = r ;
        }
    }
```

Deleting an Element

The process of deleting an element from a linked list is depicted in Fig. 4.3:



(a) Before deletion



(b) After deletion

Fig. 4.3 *Deletion in Linked List*

In the following function, we traverse through the entire linked list, checking at each node whether it has to be deleted. If the node being deleted is the first node then we shift the structure type pointer variable to the next node and then delete the earlier node.

If the node to be deleted is an intermediate node then the various pointers—the links before and after the deletion should be taken care of.

/* Function to Delete the specified node from the linked list */

```
void del(struct node **q, int no)
{
    struct node *old, *temp ;

    temp = *q ;

    while(temp != NULL)
    {
```

```
            if(temp -> data == no)
            {
                /* If node to be deleted is the first node in the linked list */
                    if(temp == *q)
                        *q = temp -> next;

                    /* Deletes the intermediate nodes from the linked list */
                    else
                        old -> next = temp -> next ;

                    /* Free the memory occupied by the node */
                    free(temp) ;
                    return ;
            }

            /* Traverse the linked list till the last node is reached */
            else
            {
                old = temp ;  /* old points to the previous node */
                temp = temp -> next ;  /* go to the next node */
            }
        }

        printf("\nElement %d not found", no) ;
    }
```

Displaying the Contents of the Linked List

It simply displays the elements of the linked list contained in the data part. For this purpose the list is traversed with the following statement till the NULL is encountered:

printf("%d", q → data); /* q points to the first node */
q = q → next;

/*Function to Display the contents of the linked list*/

```
    void display(struct node *start)
    {
        printf("\n") ;

        /* Traverse the entire linked list */
        while(start != NULL)
        {
            printf("%d ", start -> data) ;
```

```
        start = start -> next ;
    }
}
```

In the function, the variable **start** points to the first entry in the list. The address of the next node is taken by the statement:

start = start -> next

The end of the list is indicated by NULL value. The loop lists the current value of node to see if it is NULL. If node is not NULL, the loop displays the entry value and assigns node the address of the next list entry.

For example, Fig. 4.4, above we can see that the current node is 2 and data stored at this position is 11 therefore, 11 will be printed first then as the next is pointing to 5 the next current node will be 5 and the data in it (55) will be printed and the index will move to the number stored in the next of 5. The process is repeated till the NULL value is found in the next field of the number.

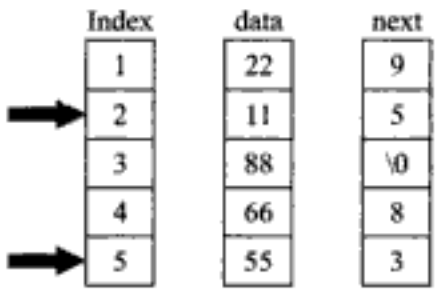


Fig. 4.4

Counting the Number of Nodes in a Linked List

The count function simply counts the number of nodes present in the linked list.

/*Function to Count the number of nodes present in the linked list*/

```
int count(struct node * q)
{
        int c = 0 ;

        /* traverse the entire linked list */
        while(q != NULL)
        {
                q = q -> next ;
                c++ ;
        }
        return c ;
}
```

Searching in Linked List

Searching means finding information in a given linked list.

/* Search a node with a given information */

```
void search(struct node *node1)
{
    int node_number = 0;
    int search_node;
    int flag = 0;
```

```
        node1 = &start ;

        printf("\n Input information of a node we want to search:");
        scanf("%d", &search_node);
        if(node1 == NULL)
        {
            printf("\n List is empty");
        }
        while(node1)
        {
            if(search_node == node->data)
            {
                printf("\n Search is successful");
                printf("\n Information which we want to search is: %d", search_node);
                printf("\n Position from beginning of the list: %d", node_number+1);
                node1 = node1->next;
                flag = 1;
            }
            else
            {
            node1 = node1->next;
        }
        node_number ++;
        }
    if(!flag)
    {
            printf("\n Search is unsuccessful");
            printf("\n Information does not exist in the list: %d", search_node);
    }
}
```

Reversing a Linked List

The reversing of the list means that last node becomes the first node and first becomes the last.

```
    void reverse(struct node **x)
    {
            struct node *q, *r, *s ;

            q = *x ;
            r = NULL ;

            /* Traverse the entire linked list */
            while(q != NULL)
```

```
        {
                s = r ;
                r = q ;
                q = q -> next ;
                r -> next = s ;
        }

        *x = r ;
}
```

The function reverse() receives the parameter struct node **x, which is the address of the pointer of the first node in the linked list. To traverse the linked list, a variable of type struct node * is required. **q** has been initialized the value of **x**, so **q** starts pointing to the first node.

The NULL value is stored in the link part of the first node, i.e.

s = r;
r = q;
r → next = s;

r which is of type struct node * is initialized to a NULL value. Since **r** contains NULL, **s** would also contain NULL. Now **r** is assigned to **q** so that **r** also starts pointing to the first node. Finally, **r → next** is assigned to **s** so that **r → next** becomes **NULL** which is nothing but the next part of the first node. Before storing a NULL value in the next part of the first node, **q** is made to point to the second node through

q = q → next;

When there is second iteration of while loop, **r** points to the first node and **q** points to the second node. Now the next part of the second node should point to first node. **q** is made to point to the second node.

q = q → next;

During the second iteration of while loop, **r** points to the first node and **q** points to the second node. Now the next part of the second node should point to first node which is done by:

s = r;
r = q;
r → next = s;

Since **r** points to the first node, **s** would also point to the first node. Now **r** is assigned the value of **q** so that **r** now points to the second node. Finally, **r → next** is assigned to **s** so that **r → next** starts pointing to the first node. But if we store the value of **s** in the second node then the address of third node would be lost. Hence, before storing the value of **s** in **r → next**, **q** is made to point to the third node by:

q = q → next;

While traversing the nodes through the while loop, **q** starts pointing to the next node in the list and **r** starts pointing to the previous node. By the end of while loop the first node becomes the last node and the last node becomes the first node.

When ***x=r** is executed, it ensures that pointer **p** now starts pointing to the node which is the last node in the original list.

Fig. 4.5 *Initial List*

Now, r=q, i.e. r also starts pointing to the first node (Fig. 4.6). q points to the second node as q = q → next.

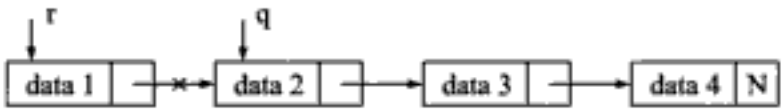


Fig. 4.6

r points to the first node and s also points to the first node as s = r (Fig. 4.7). Now, r is assigned the value of q, therefore, r now points to the second node (Fig. 4.7). From the above assignment as s was pointing to the first node so, r → next = s.

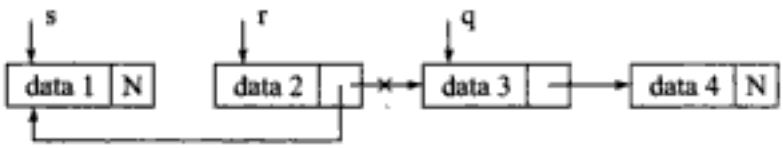


Fig. 4.7

The r → next starts pointing to the first node (Fig. 4.7)

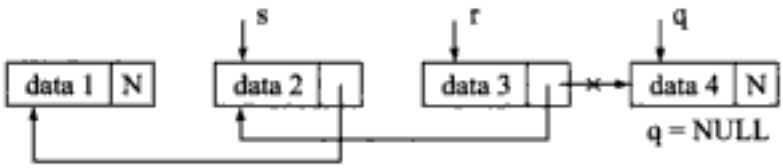


Fig. 4.8

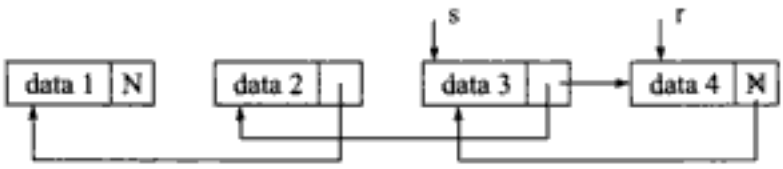


Fig. 4.9

Fig. 4.10 *Final Reversed List*

Sorting the List

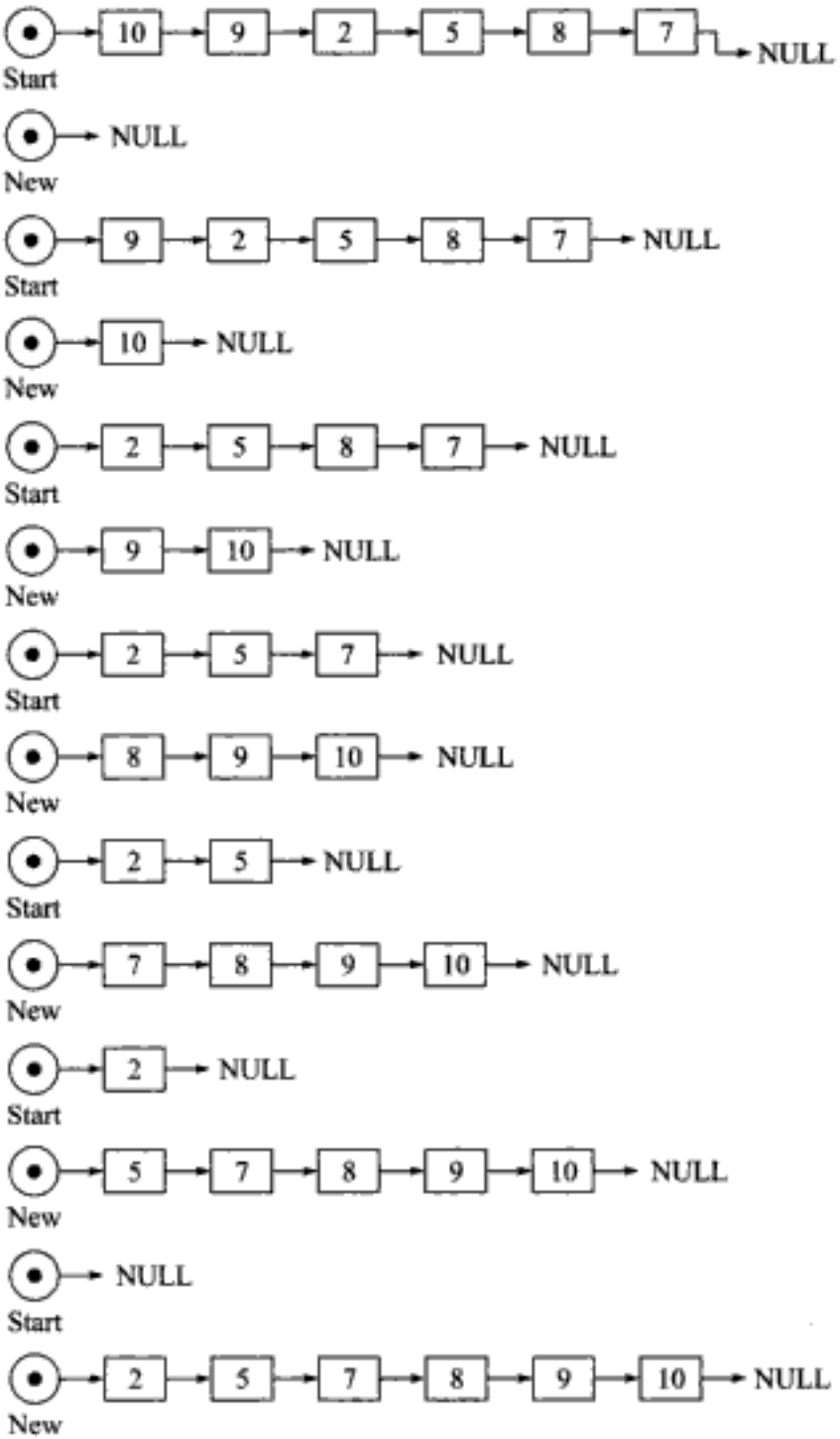
/*Function for sorting a linked list by address swapping */

```
void sort(node * first)
{
        node * temp=NULL,*i,*j,*dummy=NULL;
        int count=0;
        for(j=start;j;j=j->next,count++);
        while(count)
        {
                for(i=start;i;i=i->next)
                {
                        if(i->data>dummy->data)
                        dummy=i;
                }
                temp=start;

                //Remove the greatest data node at the end
                if(dummy==start)
                {
                        start=dummy->next;
                        getch();
                }
                else
                {
                        while(temp->next!=dummy)
                                temp=temp->next;
                        temp->next=dummy->next;
                        dummy->next=NULL;
                }
                        //Insert the nth greatest data node in the beginning
                        dummy->next=new;
                        new=dummy;
                        getch();
                        dummy=NULL;
                        count--;
        }
                start=new;
        }
```

According to the above program the function **sort** finds the largest element from the linked list and removes it. The node containing the largest element is removed from the linked list and is appended to the new list in the ascending order. When all the nodes from the main linked list are removed, the new list

contains all the nodes in ascending order and the address of the first node of the sorted list is assigned to the pointer pointing to the main list (the pointer to the main list contains NULL address pointer because all the elements are removed from the list).

The following illustrations explain the sorting method used in the above program:



Merging Two Linked Lists

Here we wish to merge two linked lists pointed by two pointers into a third list. While merging the list, care is taken to ensure that the elements common to the lists appear only once in the third list.

```
/* Merge the two linked lists, restricting the common elements to occur only
   once in the final list */
void merge(struct node *p, struct node *q, struct node **s)
{
        struct node *z ;

        z = NULL ;

        /* If both lists are empty */
        if(p == NULL && q == NULL)
                return ;

        /* Traverse both linked lists till the end. If end of any one list is
        reached loop is terminated */
        while(p != NULL && q != NULL)
        {
                /* If node being added in the first node */
                if(*s == NULL)
                {
                        *s = malloc(sizeof(struct node)) ;
                        z = *s ;
                }
                else
                {
                        z -> link = malloc(sizeof(struct node)) ;
                        z = z -> link ;
                }
                if(p -> data < q -> data)
                {
                        z -> data = p -> data ;
                        p = p -> link ;
                }
                else
                {
                        if(q -> data < p -> data)
                        {
                                z -> data = q -> data ;
                                q = q -> link ;
                        }
```

```
                else
                {
                        if(p -> data == q -> data)
                        {
                                z -> data = q -> data ;
                                p = p -> link ;
                                q = q -> link ;
                        }
                }
        }
    }

    /* If end of first list has not been reached */
    while(p != NULL)
    {
            z -> link = malloc(sizeof(struct node)) ;
            z = z -> link ;
            z -> data = p -> data ;
            p = p -> link ;
    }

    /* If end of second list has been reached */
    while(q != NULL)
    {
            z -> link = malloc(sizeof(struct node)) ;
            z = z -> link ;
            z -> data = q -> data ;
            q = q -> link ;
    }
    z -> link = NULL ;
}
```

The function **merge()** receives three parameters. The first two parameters are of the type struct node * which point to the two lists that are to be merged. The third parameter **s** is of type struct node ** which holds the address of pointer third which is a pointer to the resultant merged list.

The lists are checked before merging to find whether they are empty or not. If the lists are empty then the control simply returns from the function else a loop is executed to traverse the lists, that are pointed by p and q.

z is the pointer which points to the merged list and contains the NULL value. Inside the while loop we check the special case of adding the first node to the merged list pointed by **z**. If the node added is the first node then z is made to point to the first node by:

```
z = *s;
```

After this, data from both the lists are compared and smaller data is stored in the data part of the first node in the merged list. The pointers that point to the merged list and the lists from where we copied the data are incremented appropriately.

In the next iteration of while loop, the if condition for the first node fails and we reach to the else block. Here, we allocate the memory for the new node and its address is stored in z → next. The **z** is made to point to the node:

```
z = z → next;
```

If the data, while comparing in both the lists is equal, then it is added only once and pointers of all the three lists are incremented.

```
if(p → data == q → data)
{
        z → data = q → data;
        p = p → next;
        q = q → next;
}
```

DOUBLY LINKED LIST

The doubly or two-way linked list uses double set of pointers, one pointing to the **next** item and other pointing to the preceding item.

It can be traversed in two directions either from the beginning of the list to the end or in the backward direction from end of the list to the beginning.

A doubly linked list can be shown as follows:

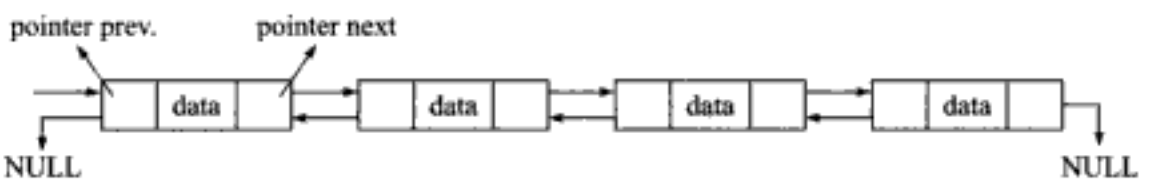


Fig. 4.11 *Doubly Linked List*

Each node contains three parts:

1. An information field which contains the data.
2. A pointer field **next** which contains the location of the next node in the list.
3. A pointer field **prev** which contains the location of the preceding node in the list.

The structure defined for doubly linked list is:

```
struct dnode
{
    int data;
    struct node * next;
    struct node * prev;
}
```

Creating a Doubly Linked List

The function d_create() adds a node at the end of the existing list. It also checks if the list is empty.

The function accepts two parameter—**s** of type struct dnode ** which contains the address of the pointer to the first node of the list or a NULL value in case of an empty list. The second parameter **num** is an integer which is to be added in the list. To begin with, we initialize **q** which is of type struct dnode * with the value stored at **s**. This is done because using **q** the entire list is traversed if it is non-empty.

If the list is empty, the condition gets satisfied and now the memory is allocated for the new node whose address is stored in **&s** (i.e., p). Using **s**, a NULL value is stored in **prev** and **next** and the value of **num** is assigned to its data part.

If the list is non-empty then through the statements

```
while(q → next ! = NULL)
    q = q → next;
```

q is made to point to the last node in the list. Then memory is allocated for the node whose address is stored in **r**. A NULL value is stored in the rest part of the list by:

```
r → prev = q;
q → next = r;
```

/*Function to add a new node at the end of the doubly linked list*/

```
void d_create( struct dnode **s, int num)
{
        struct dnode *r, *q = *s ;

        /* If the linked list is empty */
        if(*s == NULL)
        {
                /*Create a new node */
                *s = malloc(sizeof(struct dnode)) ;
                (*s) -> prev = NULL ;
                (*s) -> data = num ;
                (*s) -> next = NULL ;
        }
        else
        {
                /* Traverse the linked list till the last node is reached */
                while(q -> next != NULL)
                        q = q -> next ;
                /* Add a new node at the end */
                r = malloc(sizeof(struct dnode)) ;
                r -> data = num ;
                r -> next = NULL ;
                r -> prev = q ;
```

```
                q -> next = r ;
            }
        }
```

Adding a Node in the Beginning of Doubly Linked List

Function d_addatbeg() This function adds the node in the beginning of the list. The function takes two parameters **s** of type struct dounode ** which contains the address of the pointer to the first node and **num** is an integer to be added in the list.

The allocation of memory for the new node is done whose address is stored in **q**. The **num** is the data part of the node. A NULL value is stored in the **prev** part of new node as this is the first node in the list. This is done by:

$$q \rightarrow next = *s;$$

Now the address of this new node is stored in to the **prev** part of the first node. Make this new node the first node in the list.

$$(*s) \rightarrow prev = q$$
$$*s = q;$$

/*Function to add a new node at the beginning of the linked list*/

```
void d_addatbeg(struct dnode **s, int num)
{
        struct dnode *q ;
        /* Create a new node */
        q = malloc(sizeof(struct dnode)) ;

        /* Assign data and pointer to the new node */
        q -> prev = NULL ;
        q -> data = num ;
        q -> next = *s ;
        /* Make new node the head node */
        (*s) -> prev = q ;
        *s = q ;
}
```

Adding a Node After a Specified Position in Doubly Linked List

Function d_addafter() The d_addafter() function adds a node at the specified position of an existing list.

The function accepts three parameters. The first parameter **q** points to the first node of the list. The second parameter **loc** specifies the node number after which new node must be inserted. The third parameter is **num** which is to be added to the list.

To reach to the position where node is to be inserted, a loop is executed.

/*Function to add a new node after a specified number of nodes*/

```
void d_addafter(struct dnode *q, int loc, int num)
{
    struct dnode *temp ;
    int i ;

    /* Skip to desired portion */
    for(i = 0 ; i < loc ; i++)
    {
        q = q -> next ;
        /* If end of linked list is encountered */
        if(q == NULL)
            {
            printf ( "\nThere are less than %d elements", loc );
            return ;
            }
        }

            /* Insert new node */
            q = q -> prev ;
            temp = malloc(sizeof(struct dnode)) ;
            temp -> data = num ;
            temp -> prev = q ;
            temp -> next = q -> next ;
            temp -> next -> prev = temp ;
            q -> next = temp ;
        }
```

Deleting a Node from Doubly Linked List

This function deletes a node from the list if the data part matches with num. The function receives two parameters—the first is the address of pointer to the first node and second is the number to be deleted.

To traverse the list, a loop is run. The data part of each node is compared with num. If the num value matches the data part, then the position of the node to be deleted is checked.

If it happens to be the first node, then the first node is made to point to the next part of the first node.

$$*s = (*s) \rightarrow next;$$

The value NULL is stored in **prev** part of the second node, as it wiil now be the first node. This is done by:

$$*s = (*s) \rightarrow next = NULL;$$

If the node to be deleted is the last node then the NULL is stored in next part of the second last node to the position where the node is to be inserted. By this time **q** is pointing to the node before which the new node is to be added.

The statement q = q → prev; makes q point to the node after which the new node 0should be added. The memory is allocated for the new node and its address is stored in **temp**. The value of **num** is stored in the data part of the new node.

The statement temp → prev = q; makes the **prev** part of the new node to point to q. The address stored in the **next** part of the node pointed by **q** should be pointed by next part of the new node. This is done by:

```
temp → next = q → next;
```

Now the **prev** part of the next node is made to point to the new node:

```
temp → next → prev = temp;
```

Finally, the next part of **q** is made point to the new node.

```
q → next = temp;
if(q → next == NULL)
q → prev → next = NULL;
```

If the node to be deleted is the intermediate node, the address of next node is stored in next part of the previous node and the address of previous node is stored in the prev part of the next node.

```
q → prev → next = q → next;
q → next → prev = q → prev;
```

Finally, the memory occupied by the node being deleted is released by calling the function **free**.

/*Function to Delete the specified node from the doubly linked list*/

```
void d_delete(struct dnode **s, int num)
{
   struct dnode *q = *s ;

   /* Traverse the entire linked list */
   while(q != NULL)
   {
      /* If node to be deleted is found */
      if(q -> data == num)
      {
         /* If node to be deleted is the first node */
         if(q == *s)
         {
            *s = (*s) -> next ;
            (*s) -> prev = NULL ;
         }
            else
         {
            /* If node to be deleted is the last node */
            if(q -> next == NULL)
```

```
                q -> prev -> next = NULL ;
            else
            /* If node to be deleted is any intermediate node*/
            {
                q -> prev -> next = q -> next ;
                q -> next -> prev = q -> prev ;
            }
            free(q) ;
        }
        return ;  /* Return back after deletion */
    }
    q = q -> next ; /* Go to next node */
}
printf("\n%d not found.", num) ;
}
```

Displaying the Contents of Doubly Linked List

To display the contents of the doubly linked list, we follow the same algorithm that we had used in the singly linked list. Here **q** points to the first node in the list and the entire list is traversed using

```
q = q -> next ;
    void d_display(struct dnode *q)
{
    printf("\n") ;
    /* Traverse the entire linked list */
    while(q != NULL)
    {
        printf("%2d\t", q -> data) ;
        q = q -> next ;
    }
}
```

CIRCULAR LINKED LIST

A linked list in which last node points to the header node is called the **circular linked list**. The circular linked lists have neither a beginning nor an end. A small change in the structure of linear linked list is made, that is, the next field in the last node contains a pointer back to the first node rather than the NULL pointer. Therefore, the structure defined for circular linked list is same as for the linear linked list as given below:

```
struct node
{
        int data;
        struct node * next;
}
```

A shortcoming of the linear linked list is that with a given pointer to a node in linked list we cannot reach any of the nodes that precede the node which the given pointer variable is pointing to. This disadvantage is overcome by making a little change in the structure of linear linked list and thus making a circular linked list as shown in Fig. 4.12.

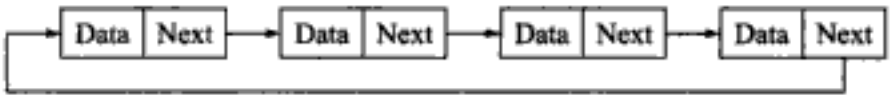


Fig. 4.12

A circular linked list can be used to represent a stack and a queue. The following program implements queue as a circular linked list. Pointers front and rear point to the first and last nodes of the list respectively.

/* Program to maintain a circular linked list */

```
void main( )
{  /* Here front points to the first node and rear points to the second node */
       struct node *front, *rear ;
       front = rear = NULL ;

       cir_add(&front, &rear, 10) ;
       cir_add(&front, &rear, 17) ;
       cir_add(&front, &rear, 18) ;
       cir_add(&front, &rear, 5) ;
       cir_add(&front, &rear, 30) ;
       cir_add(&front, &rear, 15) ;
       clrscr( ) ;

       printf("Before deletion:\n") ;
       cirq_disp(front) ;

       del_cirq(&front, &rear) ;
       del_cirq(&front, &rear) ;
       del_cirq(&front, &rear) ;

       printf("\n\nAfter deletion:\n") ;
       cirq_disp(front) ;
}
```

/*Adds a new element at the end of queue */

```
void cir_add(struct node **f, struct node **r, int item)
{
       struct node *q ;

       /* Create new node */
```

```
            q = malloc(sizeof(struct node)) ;
            q -> data = item ;
            /* If the queue is empty */
            if(*f == NULL)
                  *f = q ;
            else
                  (*r) -> next = q ;

      *r = q ;
      (*r) -> next = *f ;
}
```

/* Removes an element from front of queue */

```
int del_cirq ( struct node **f, struct node **r )
{
      struct node *q ;
      int item ;

      /* If queue is empty */
      if(*f == NULL)
            printf("queue is empty") ;
      else
      {
            if(*f == *r)
            {
                  item = (*f) -> data ;
                  free(*f) ;
                  *f = NULL ;
                  *r = NULL ;
            }
            else
            {
                  /* Delete the node */
                  q = *f ;
                  item = q -> data ;
                  *f = (*f) -> next ;
                  (*r) -> next = *f ;
                  free(q) ;
            }
            return(item) ;
      }
      return NULL ;
```

```
}

/* Displays whole of the queue */
void cirq_disp(struct node *f)
{
     struct node *q = f, *p = NULL ;

     /* Traverse the entire linked list */
     while(q != p)
     {
          printf("%d\t", q -> data) ;

          q = q -> next;
          p = f ;
     }
}
```

The front and rear are the two pointers which point to the first node of the list. They are initialized to NULL.

Creating a Circular Linked List using Queue

Function Ciradd() This function accepts three parameters. The first parameter receives the address of the pointer to the first node (i.e., address of first node—front), the second receives the address of the pointer to the last node (i.e., address of last node—rear). The third parameter holds the data items we need to add in the list.

The memory is allocated for the new node whose address is stored in pointer **q**. Then, the data which is present in **item** is stored in the data part of the new node. If the new node is added in the empty list then the address of the new node is stored in front *f = q;.

Then * r = q; is executed, which stores the address of the new node into **rear**. Thus, both **front** and **rear** point to the same node.

The statement (*r) → link = *f; is executed to store the address of the front node in the next part of the rear node (As the link part of the last node should contain the address of the first node).

Next, if the new node is not added in the first node then the address present in the next part of the last node is overwritten with the address of new node, (*r) → next = q;

Then the address of the new node is stored in the pointer rear *r = q; and the address of the first node is stored in the next part of the new node. This is done by:

(*r) → link = *f;

Deleting from Circular Linked List

Function Delcirq() This function receives two parameters. The first parameter is the pointer to the front and the second is the pointer to the rear. The condition is checked for the empty list.

If the list is not empty, then it is checked whether the front and rear point to the same node or not. If they point to the same node, then the memory occupied by the node is released and front and rear are both assigned a NULL value.

If the front and rear are pointing to different nodes then the address of the first node is stored in a →pointer q. Then the front pointer is made to point to the next node in the list, i.e. the node pointed by (*f) → link;. Now the address of front is stored in the next part of the last node. Then the memory occupied by the node being deleted is released.

Displaying the Circular Linked List

Function cirq-disp() This function receives the pointer to the first node in the list as a parameter. Then **q** is also made to point to the first node in the list. This is done because the entire list is traversed using **q**. Another pointer **p** is set to NULL initially. The circular list is traversed through a loop till the time we reach the first node again. We would reach the first node when **q** equals **p**.

ATOMIC NODE LINKED LIST

An atomic data type contains only the data items and not the pointers. Thus, for a list of data items several atomic type nodes may exist, each with a single data item corresponding to one of the legal data types. Their list is maintained using a list node which contains pointers to these atomic nodes and a type indicator indicating the type of atomic node to which it points. Whenever a list node is inserted in a list, its address is stored in the next free element of the list of pointers.

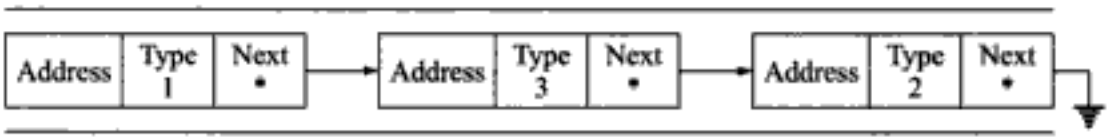


Fig. 4.13

/* Program to maintain a list of atomic nodes */

```
# include<stdio.h>
# include<conio.h>
typedef struct nodeList
{       int type;
        union AtomNode * data;
        struct nodeList * next;

} nodeList;
nodeList * start=NULL;
typedef union AtomNode
{
        char chardata;
        int intdata;
        long longdata;
```

```
            float floatdata;
            double doubledata;
      }AtomNode;
      void main()
      {
      int type,data;
      while(1)
            {
                  puts("1.  New data");
                  puts("2.  Display List");
                  puts("3.  Exit");
                  scanf("%d",&ch);
                  switch(ch)
                  {
                  case 1: scanf("%d",&type);
                        InsertNode(start,type);
                        break;
                  case 2: Display(start);
                        getch();
                        break;
                  case 3: exit(0);
                  }
            }
}
void InsertNode(nodeList * first,int type)
{
AtomNode * node;
nodeList * temp=NULL;
node=(AtomNode * ) malloc(sizeof(AtomNode));
switch(type)
{
     case 1:
            puts("Enter char data");
            scanf("%c",&node->chardata);
            break;
     case 2:
            puts("Enter int data");
            scanf("%d",&node->intdata);
            break;
     case 3:
            puts("Enter long data");
            scanf("%ld",&node->longdata);
```

```
            break;
      case 4:
            puts("Enter float data");
            scanf("%f",&node->floatdata);
            break;
      case 5:
            puts("Enter double data");
            scanf("%lf",&node->doubledata);
            break;
      default:
            puts("Invalid type");
}
      temp=(nodeList * ) malloc(sizeof(nodeList));
      temp->type=type;
      temp->data=node;
      temp->next=NULL;
      if(first)
      {
            while(first->next)
                  first=first->next;
            first->next=temp;
      }
      else
            start=temp;
      }
void Display(nodeList * first)
{
if (first)
      {
      while(first)
            {
switch(first->type)
      {
      case 1:
            printf("%c",first->data->chardata);
            break;
      case 2:
            puts("Enter int data");
            scanf("%d",&node->intdata);
            break;
      case 3:
            puts("Enter long data");
```

```
            scanf("%ld",&node->longdata);
            break;
      case 4:
            puts("Enter float data");
            scanf("%f",&node->floatdata);
            break;
      case 5:
            puts("Enter double data");
            scanf("%lf",&node->doubledata);
            break;
switch(type)
{
      case 1:
            puts("Enter char data");
            scanf("%c",&node->chardata);
            break;
      case 2:
            puts("Enter int data");
            scanf("%d",&node->intdata);
            break;
      case 3:
            puts("Enter long data");
            scanf("%ld",&node->longdata);
            break;
      case 4:
            puts("Enter float data");
            scanf("%f",&node->floatdata);
            break;
      case 5:
            puts("Enter double data");
            scanf("%lf",&node->doubledata);
            break;

            printf("\t%d",first->data);
            first=first->next;
            }
      }
else
      puts("List does not exist");
}
```

For the 'C' representation of atomic node list we require the following structure:

```
      struct nodelist
```

```
{
        int type;
        union atomicnode;
        struct nodelist * next;
}
and -   union atomicnode
                {
                        char chardata;
                        int intdata;
                }
```

In the nodelist structure the **type** member keeps the type of data according to the following convention—1 - char, 2 - int.

The member next is a pointer to nodelist which points to the next node and the member atomicnode is union type member.

The data structure for atomicnode has been chosen as a union instead of structure type because only one member of the data structure will be used for each data item. For example, a particular data can be of type char or int or long. Therefore, union is the best data structure for 'C' implementation of atomic node.

The program uses the data structure discussed above to maintain a list of atomic nodes.

The function receives two parameters—a pointer to the nodelist and the type of the data. According to the type of data, the data is scanned from the user and is stored in the new atomicnode. The address of this atomic node is stored in the atomic node pointer. New nodes, both for the nodelist and the atomic node, are allocated dynamically for the function. Similarly, logic for display () function selects the type of the data in the atomic node, reads and displays it accordingly.

LINKED LIST IN ARRAYS

Linked lists can be implemented without using pointers. For example, consider an ordered list of integers given by L = (10, –5, 0, 99). This list can be stored in an array, say "Ele". The concept of link can be implemented by using another array, "Next". The i^{th} element of "L" is guaranteed to be stored in the i^{th} index of an array "Ele".

The node in a linked list contains two parts—"**data**" and "**next**". These two parts of node are split and stored in two arrays "**Ele**" and "**Next**". If **Ele[i]** represents the data part of the node then **Next[i]** denotes the next part of that node. In this case the actual physical address is not denoted by next. Rather Next[i] is an integer and if Next[i] is j then the node next to the one represented by i^{th} index of "Ele" and the "Next" is the node represented by j^{th} index of "Ele" and "Next". If Next[i] = –1 then, the node under consideration is assumed to be the last node. Initially these arrays are unused and so a variable "free" is set to 0. The "free" function keeps track of the available parts in the arrays.

Figure 4.14 illustrates the initial situation of arrays:

When the first element of our list L is added to the array, **Ele[0]** contains 10, **Next[0]** remains –1, "**free**" becomes 1 and new variable "**start**" is required to remember which index in these arrays

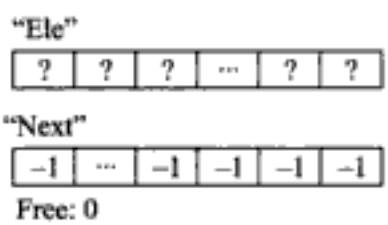


Fig. 4.14

represent first node in the list. Figure 4.15 illustrates the addition of first node in the list.

When an attempt is made to add the second element of the list, the existing element is traversed from "start". As in our case, the value is 0. Then the Next[0] is –1, this is the end of the list. So the new node is added after this node. The index where the new node is to be stored in "Ele" and "Next" is found by inspecting "free". Next [0] is updated by current value of "free" and "free' is also incremented.

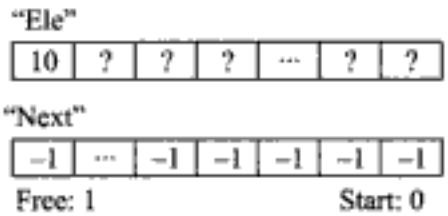


Fig. 4.15

Figure 4.16 illustrates adding second element in the list.

Note that Ele[1] is set to –5, Next[0] is set to 1 and Next[1] is set to –1. Another addition will need to traverse the list from "Start", start = 0. As Next[0] = 1 the next node can be found in index 1 of the array "Ele" and "Next". The Next[1] is found, the value is –1, i.e. the node is the last node. As before, Next[1] is set to the current value of "free". Ele[free] is set to –1, and "free" is incremented. Figure 4.17 and 4.18 illustrate the addition of third and fourth nodes to the list.

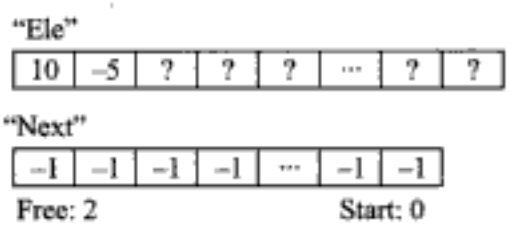


Fig. 4.16

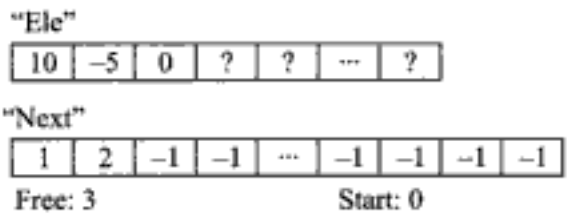


Fig. 4.17 *Adding Third Node to the List*

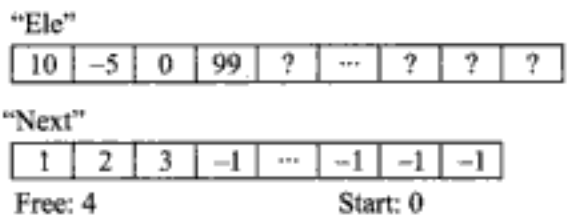


Fig. 4.18 *Adding Fourth Node to the List*

Now if we want to delete the second node in the linked list, the next field of the first node is to be changed, in respect of the next field of second node. The array will be shown as given in Fig. 4.19.

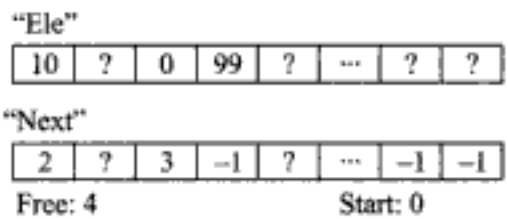


Fig. 4.19 *Deletion of the Second Node from the List*

Therefore, it can be seen that no shifting of elements is done. Only the "next" array is updated. Say, if we want to again **add** an element with value 56 to the list after deletion (Fig. 4.20).

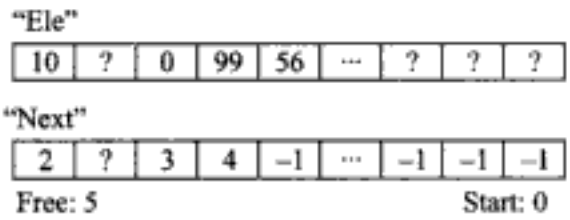


Fig. 4.20 *Adding Another Node to the List*

It can be observed that the unused node 2 (i.e. the element with index 1 in the arrays "Ele" and "Next") could not be reused. The variable "free" keeps on increasing without paying any attention to the unused nodes. Figure 4.21 illustrates the deletion of the first node in the list. The value of the "Next" array with index "start" has to be made the new value of "start". In this example start = 0 and Next[0] = 2 before deletion, so the new value of "start" become 2 and Next [0] becomes undefined after deletion.

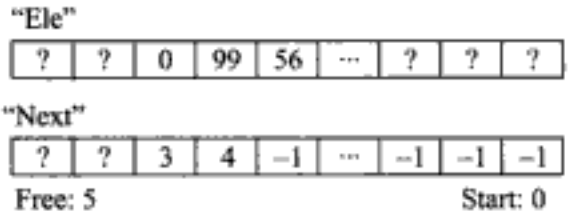


Fig. 4.21

LINKED LISTS VERSUS ARRAYS

We can store similar data in memory with the use of either an array or a linked list.

Arrays are very simple data structures that are easy to understand but they have the following disadvantages:

- The size of arrays cannot be increased or decreased during execution. They have a fixed dimension. For example, if we have allocated space for 10 elements and try to add more than 10 elements we are not able to do so, and on the other hand if we have allocated the space for 10 elements but are not using the whole space, the unused space goes waste.
- The elements in an array are stored in contiguous memory locations, but in many cases it may be possible that the contiguous memory space is not available.
- The operations like insertion of a new element in an array or deletion of an existing element after the specified position may be tedious as insertion or deletion requires each element after the specified position to be shifted one position to the right (insertion) or one position to the left (deletion).

Linked List can be used to overcome all these disadvantages.

- A linked list can grow or shrink during the execution of program.
- There is no problem of shortage of memory as the nodes are stored in different memory locations.
- In various operations like insertion and deletion no shifting of nodes is required.

The major disadvantage of dynamic implementation (linked lists) is that it may be more time consuming for the system to allocate and free storage to manipulate a variable list.

Summary

- Linked list is the most commonly used data structure to store similar type of data in memory.
- Self referential structures and pointer data types may represent the singly connected linked lists.
- To make the traversal operation easy, doubly connected linked lists are used, in which every node contains links to its left and right neighbours.
- The NULL value in the end of a single linked list denotes the end of the list. The NULL link when set to the beginning of the list, results in the list called circular linked list.
- The idea of dynamic memory allocation is to be able to allocate and deallocate memory at runtime in response to program requirements and thus manage that space efficiently.
- Arrays can also be used to implement the linked lists. For the implementation through arrays, we use two arrays, "data" and "next" to store the nodes of the lists.
- The atomic linked lists contain entities which are either atomic or list themselves.

Review Exercise

Multiple Choice Questions

1. In linked list, a node contains
   a. node, address field and data field
   b. node number, data field
   c. next address field, information field
   d. None of these
2. In linked list, the logical order of elements
   a. is same as their physical arrangement
   b. is not necessarily equivalent to their physical arrangement
   c. is determined by their physical arrangement
   d. None of the above
3. NULL pointer is used to tell
   a. end of linked list
   b. empty pointer field of a structure
   c. the linked list is empty
   d. All of the above
4. List pointer variable in linked list contains address of the
   a. following node in the list
   b. current node in the list
   c. first node in the list
   d. None of the above

5. Due to the linear structure of linked list having linear ordering, there is similarity between linked list and array in
   a. insertion of a node
   b. deletion of a node
   c. traversal of elements of list
   d. None of the above
6. Searching of linked list requires linked list to be created
   a. in stored order only
   b. in any order
   c. without underflow condition
   d. None of the above

Fill in the Blanks

1. The next address field is known as _____. (pointer/address)
2. In linked list, the identity of next element is _____ defined. (explicity/implicity)
3. Besides data field, each node of linked list contains at least _____ more fields. (one/two)
4. End of the linked list is marked by putting _____ in the next address field in the last node. (next/NULL pointer)
5. List pointer variable contains the address of _____ pointer. (first/last)
6. Attempting to delete a new node in _____ linked lists results in underflow. (empty/non-empty)

State whether True or False

1. List-null can be used to initialize list as empty list.
2. Pointer is used to provide the linear order in linked list.
3. In linked list, successive elements need not occupy adjacent space in memory.
4. List pointer variable contains the address of last node in linked list.
5. Attempting to create a new node, when free space pool has no space, results in overflow condition.

Descriptive Questions

1. Write a 'C' program to delete a node containing given information in singly linked list.
2. Write a 'C' function to combine two singly connected linked lists in the following manner. Suppose one list is "C" which can be given by $L = (l_0, l_1, l_2, \ldots l_m)$ and the other list is "M" where M can be expressed as $M = (m_0, m_1 \ldots m_m)$ where each l_1 and m_1 represents one node in respective lists. For simplicity you may assume that each node contains integer as data. After combining them the combined list should be $(l_0, m_0, l_1, m_1 \ldots)$. Do not use any additional node for writing the function.
3. Write a program that reads the name, age and salary of 10 persons and maintains them in linked list sorted by name.
4. There are two linked lists A and B containing the following data:
   A : 2, 5, 9, 14, 15, 7, 20, 17, 30
   B : 14, 2, 9, 13, 37, 8, 7, 28
   Write programs to create:
   (i) A linked list C that contains only those elements that are common in linked list A and B.
   (ii) A linked list D which contains all elements of A as well as B ensuring that there is no repetition of elements.

5. Assume a singly linked list containing integers. Write a function move() which would move a node forward by n positions in the linked list.
6. Write a 'C' program to create a doubly linked list in ascending sorted order of information.
7. Write a 'C' program to delete a node containing a given information in doubly linked list. Make necessary assumptions.
8. Consider a circular list with a pointer pointing to its tail. Write 'C' function to
   (i) insert a node in the front of the list
   (ii) insert a node at the rear of the list
   (iii) find the length of the list

CHAPTER

5 Polynomials and Sparse Matrix

Key Features

- Introduction to Polynomials
- Representation of Polynomials
- Introduction to Sparse Matrix
- Representation of Sparse Matrix through Linked Lists
- Representation of Complex Numbers

The two important applications of arrays and linked lists are—polynomials and sparse matrix.

A polynomial is made of different terms, each of which consists of a coefficient and an exponent. This chapter includes the representation of polynomials through linked lists and arrays and various operations performed on these polynomials.

Sparse matrix is a matrix most of whose elements are zero. A sparse matrix can also be represented by using arrays and linked lists. Various operations like addition and multiplication can be performed using different representations.

INTRODUCTION TO POLYNOMIALS

A polynomial, p(x), is an expression in variable x of the form $(ax^n + bx^{n-1} + ... + jx + k)$ where a, b, c, ...k are real numbers and n is a non-negative integer. The number n is called the degree of the polynomial.

An important characteristic of a polynomial is that each term in the polynomial expression consists of two parts—one is a **coefficient** and the other is an **exponent**.

Consider the following polynomial:

$$10x^5 + 15x^3 - 7x^2 - x$$

Here, (10, 15, –7, –1) are coefficients and (5, 3, 2, 1) are exponents.

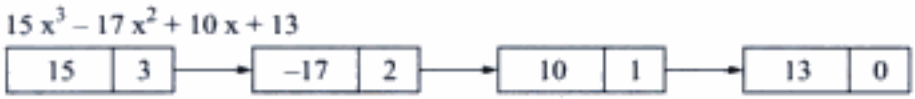
Exponents are the placeholders for any value that remains constant for each term in a single expression. In data structure, a polynomial can be represented as a list of nodes where each node consists of coefficient and an exponent.

Points to be considered when working with polynomials are:

- Sign of each coefficient and exponent is stored within the coefficient and exponent itself.
- Only addition of term with equal exponent is possible.

- Storage of each term in the polynomial must be done in ascending/descending order of their exponent.

For example,



REPRESENTATION OF POLYNOMIALS

Polynomials can be represented in the following two ways:

- Using arrays
- Using linked lists

Representation of Polynomials using Arrays

We often need to evaluate many polynomial expressions and perform basic arithmetic operations—addition, multiplication, etc.—on them. For this purpose, we need a way to represent a polynomial. The simplest way to represent a polynomial of degree "n" is to store the coefficient of (n+1) terms of a polynomial in an array. Thus, $x + 4x^2 - 6x^5$ is a polynomial of degree 5 and $x + 4x^4 - 9x^3 + 10x^6$ is a polynomial of degree 6.

Thus, each element of an array consists of two values, namely, coefficient and exponent. It is also assumed that exponent of each successive term is less than that of the previous term. After building an array for polynomials, we can use it to perform various operations.

A polynomial can be represented by using the structure given below. Here we also define a macro MAX to set the limit of exponent of any term in the polynomial. The following program inserts each term of the polynomial in an ascending order.

/* Program to display and add two polynomials using arrays */

```
# include <stdio.h>
# include <conio.h>

# define MAX 10

struct term
{
        int coeff ;
        int exp ;
} ;

struct poly
{
        struct term t[10] ;
        int totalterms ;
} ;
```

This program uses the following functions to manage a polynomial:

```
void initpoly(struct poly *) ;
void polycreate(struct poly *, int c, int e) ;
struct poly addpoly(struct poly, struct poly) ;
void display(struct poly) ;

void main( )
{
  struct poly p1, p2, p3 ;
  clrscr( ) ;
  initpoly(&p1) ;
  initpoly(&p2) ;
  initpoly(&p3) ;

  polycreate(&p1, 1, 7) ;
  polycreate(&p1, 2, 6) ;
  polycreate(&p1, 3, 5) ;
  polycreate(&p1, 4, 4) ;
  polycreate(&p1, 5, 2) ;

  polycreate(&p2, 1, 4) ;
  polycreate(&p2, 1, 3) ;
  polycreate(&p2, 1, 2) ;
  polycreate(&p2, 1, 1) ;
  polycreate(&p2, 2, 0) ;

  p3 = addpoly(p1, p2) ;

  printf("\nFirst polynomial:\n") ;
  display(p1) ;

  printf("\n\nSecond polynomial:\n") ;
  display(p2) ;

  printf("\n\nResultant polynomial:\n") ;
  display(p3) ;
  getch( ) ;
}

/* Initializes elements of struct poly */
void initpoly(struct poly *p)
{
  int i ;
```

```
{
  int i ;
  p -> totalterms = 0 ;
  for(i = 0 ; i < MAX ; i++)
  {
       p -> t[i].coeff = 0 ;
       p -> t[i].exp = 0 ;
  }
}

/* Adds the term of polynomial to the array t */
void polycreate(struct poly *p, int c, int e)
{
  p -> t[p -> totalterms].coeff = c ;
  p -> t[p -> totalterms].exp =  e ;
  (p -> totalterms) ++ ;
}

/* Displays the polynomial equation */
void display(struct poly p)
{
  int flag = 0, i ;
  for(i = 0 ; i < p.totalterms ; i++)
  {
       if(p.t[i].exp != 0)
              printf("%d x^%d + ", p.t[i].coeff, p.t[i].exp) ;
       else
       {
              printf("%d", p.t[i].coeff) ;
              flag = 1 ;
       }
  }
  if(!flag )
       printf("\b\b") ;

}
/* Adds two polynomials p1 and p2 */
struct poly addpoly(struct poly p1, struct poly p2)
{
  int i, j, c ;
  struct poly p3 ;
  initpoly(&p3) ;
```

```
  if(p1.totalterms > p2.totalterms)
      c = p1.totalterms ;
  else
      c = p2.totalterms ;

  for(i = 0, j = 0 ; i <= c ; p3.totalterms++)
  {
      if(p1.t[i].coeff == 0 && p2.t[j].coeff == 0)
            break ;
      if(p1.t[i].exp >= p2.t[j].exp)
      {
            if(p1.t[i].exp == p2.t[j].exp)
            {
                  p3.t[p3.totalterms].coeff = p1.t[i].coeff + p2.t[j].coeff ;
                  p3.t[p3.totalterms].exp = p1.t[i].exp ;
                  i++ ;
                  j++ ;
            }
            else
            {
                  p3.t[p3.totalterms].coeff = p1.t[i].coeff ;
                  p3.t[p3.totalterms].exp = p1.t[i].exp ;
                  i++ ;
            }
      }
      else
      {
            p3.t[p3.totalterms].coeff = p2.t[j].coeff ;
            p3.t[p3.totalterms].exp = p2.t[j].exp ;
            j++ ;
      }
  }
  return p3 ;
}

/* Multiplies two polynomials p1 and p2 */
struct poly mulpoly(struct poly p1, struct poly p2)
{
  int coeff, exp ;
  struct poly temp, p3 ;

  initpoly(&temp) ;
  initpoly(&p3) ;
```

```
    if(p1.totalterms != 0 && p2.totalterms != 0)
    {
        int i ;
        for(i = 0 ; i < p1.totalterms ; i++)
        {
            int j ;

            struct poly p ;
            initpoly(&p) ;

            for(j = 0 ; j < p2.totalterms ; j++)
            {
                coeff = p1.t[i].coeff * p2.t[j].coeff ;
                exp = p1.t[i].exp + p2.t[j].exp ;
                polycreate(&p, coeff, exp) ;
            }

            if(i != 0)
            {
                p3 = addpoly(temp, p) ;
                temp = p3  ;
            }
            else
                temp = p ;
        }
    }
    return p3 ;
}
```

In the program given above, the function **mulpoly()** is used for the multiplication of two polynomials. To build two polynomials we call the function **polycreate()**. The **mulpoly()** function takes two parameters p1 and p2 and returns the result in the third polynomial p3.

The **mulpoly()** function checks whether the two polynomials are non-empty. If they are not, then the control goes in a pair of for loops. Each term of first polynomial in p1 is multiplied with every term of second polynomial contained in p2. We also call **polycreate()** to add terms to P. The first resultant polynomial equation is stored in a temporary variable **temp** of the type struct poly. Then onwards **addpoly()** is called to add the resulting polynomial equation. The resulting term is displayed using the **display()** function.

Representation of Polynomials using Linked Lists

A polynomial can be thought of as an ordered list of non-zero terms. Each non-zero term is a 2-tuple containing two pieces of information:

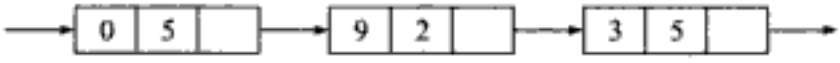
- the exponent part

- the coefficient part

For example, consider the following polynomial:

$$3x^5 - 9x^2 + 5$$

For this polynomial, the first tuple indicates the term $5.x^0$, i.e. 5, the second tuple indicates $-9.x^2$ and last denotes $3.x^5$. The tuples are arranged in increasing order of their exponent part, that means first tuple contains the non-zero with the least power of 'x' and the last tuple contains the non-zero term with highest power of 'x'.



The structure for a polynomial can be defined using a linked list:

```
struct poly
{
    float coeff;
    int exp;
    struct poly *next;
}
```

Once we build a linked list to represent a polynomial we can use such a list to perform common polynomial operations like addition and multiplication.

Program to Create, Display and Add Polynomials using Linked Lists The following structure represents a node of a linked list. The node can store one term of a polynomial. This program uses stdio.h, conio.h and alloc.h header files.

/* Program to create, display and add polynomials */

```
struct polynode
{
  float coeff ;
  int exp ;
  struct polynode *next ;
} ;

void create_poly(struct polynode **, float, int) ;
void display(struct polynode *) ;
void add_poly(struct polynode *, struct polynode *, struct polynode **) ;
void main( )
{
  struct polynode *first, *second, *total ;
  int i = 0 ;

  first = second = total = NULL ;  /* Empty linked lists */
```

```
  create_poly(&first, 1.4, 5) ;
  create_poly(&first, 1.5, 4) ;
  create_poly(&first, 1.7, 2) ;
  create_poly(&first, 1.8, 1) ;
  create_poly(&first, 1.9, 0) ;

  clrscr( ) ;
  display(first) ;

  create_poly(&second, 1.5, 6) ;
  create_poly(&second, 2.5, 5) ;
  create_poly(&second, -3.5, 4) ;
  create_poly(&second, 4.5, 3) ;
  create_poly(&second, 6.5, 1) ;

  printf("\n\n") ;
  display(second) ;

  /* Draws a dashed horizontal line */
  printf("\n") ;
  while(i++ < 79)
       printf("-") ;
  printf ("\n\n") ;
  add_poly(first, second, &total) ;
  display(total) ;  /* Displays the resultant polynomial */
}

/* Adds a term to a polynomial */
void create_poly(struct polynode **q, float x, int y)
{
  struct polynode *temp ;
  temp = *q ;

  /* Creates a new node if the list is empty */
  if(*q == NULL)
  {
       *q = malloc(sizeof(struct polynode)) ;
       temp = *q ;
  }
  else
  {
       /* Traverse the entire linked list */
```

```
        while(temp -> next != NULL)
                temp = temp -> next ;

                /* Create new nodes at intermediate stages */
                temp -> next = malloc ( sizeof ( struct polynode ) ) ;
                temp = temp -> next ;
    }

    /* Assign coefficient and exponent */
    temp -> coeff = x ;
    temp -> exp = y ;
    temp -> next = NULL ;
}

/* Displays the contents of linked list representing a polynomial */
void display(struct polynode *q)
{
    /* Traverse till the end of the linked list */
    while(q != NULL)
    {
        printf("%.1f x^%d  :  ", q -> coeff, q -> exp) ;
        q = q -> next ;
    }

    printf("\b\b\b") ;  /* Erases the last colon */
}

/* Adds two polynomials */
void add_poly(struct polynode *x, struct polynode *y, (struct  polynode **s)
{
    struct polynode *z ;

    /* If both linked lists are empty */
    if(x == NULL && y == NULL)
        return ;

    /* Traverse till one of the list ends */
    while(x != NULL && y != NULL)
    {
        /* Create a new node if the list is empty */
        if(*s == NULL)
        {
```

```
            *s = malloc(sizeof(struct polynode)) ;
            z = *s ;
        }
        /* Create new nodes at intermediate stages */
        else
        {
            z -> next = malloc(sizeof(struct polynode)) ;
            z = z -> next ;
        }

        /* Store a term of the larger degree polynomial */
        if(x -> exp < y -> exp)
        {
            z -> coeff = y -> coeff ;
            z -> exp = y -> exp ;
            y = y -> next ;  /* Go to the next node */
        }
        else
        {
            if(x -> exp > y -> exp)
            {
                z -> coeff = x -> coeff ;
                z -> exp = x -> exp ;
                x = x -> next ;  /* Go to the next node */
            }
            else
            {
               /* Add the coefficients, when exponents are equal */
                if(x -> exp == y -> exp)
                {
                    /* Assigning the added coefficient */
                    z -> coeff = x -> coeff + y -> coeff ;
                    z -> exp = x -> exp ;
                    /* Go to the next node */
                    x = x -> next ;
                    y = y -> next ;
                }
            }
        }
    }

    /* Assign remaining terms of the first polynomial to the result */
```

```
  while(x != NULL)
  {
       if(*s == NULL)
       {
              *s = malloc(sizeof(struct polynode)) ;
              z = *s ;
       }
       else
       {
              z -> next = malloc(sizeof(struct polynode)) ;
              z = z -> next ;
       }

       /* Assign coefficient and exponent */
       z -> coeff = x -> coeff ;
       z -> exp = x -> exp ;
       x = x -> next ;  /* Go to the next node */
  }
  /* Assign remaining terms of the second polynomial to the result */
  while(y != NULL)
  {
       if(*s == NULL)
       {
              *s = malloc(sizeof(struct polynode)) ;
              z = *s ;
       }
       else
       {
              z -> next = malloc(sizeof(struct polynode)) ;
              z = z -> next ;
       }

       /* Assign coefficient and exponent */
       z -> coeff = y -> coeff ;
       z -> exp = y -> exp ;
       y = y -> next ;  /* Go to the next node */
  }

  z -> next = NULL ;  /* Assign NULL at end of resulting linked list */
}
```

In the program given above, the function **create_poly()** is called several times to create two polynomials pointed to by the pointers p1 and p2.

The function **add_poly()** is called to add two polynomials. The function traverses the linked list till the end of any one of them is reached. While doing this traversal the polynomials are compared on term by term basis. If the exponents of two terms being compared are equal then their coefficients are added and their result is stored in p3. If the exponents of two terms are not equal the term with bigger exponent is added to the third polynomial.

While traversing, if the end of one of the lists is reached the control breaks out of the while loop. The remaining terms of other lists are added to the resulting polynomial.

The terms of the resulting polynomials are displayed using the function **display()**.

Multiplication of Two Polynomials using Linked List

/* Program to multiply two polynomials maintained as linked lists.*/

```
void mul_poly(struct polynode *, struct polynode *, struct polynode **) ;
void result_add_poly(float, int, struct polynode **) ;

/* Multiplies the two polynomials */
void mul_poly(struct polynode *x, struct polynode *y,
                        struct polynode **m)
{
  struct polynode *y1 ;
  float coeff1, exp1 ;

  y1 = y ;  /* Point to the starting of the second linked list */

  if(x == NULL && y == NULL)
       return ;

  /* If one of the list is empty */
  if(x == NULL)
       *m = y ;
  else
  {
       if(y == NULL)
              *m = x ;
       else  /* If both linked lists exist */
       {
              /* For each term of the first list */
              while(x != NULL)
              {
                     /* Multiply each term of the second linked list with a
                        term of the first linked list */
                     while(y != NULL)
```

```
                    {
                        coeff1 = x -> coeff * y -> coeff ;
                        exp1 = x -> exp + y -> exp ;
                        y = y -> next ;

                     /* Add the new term to the resultant polynomial */
                        padd(coeff1, exp1, m) ;
                    }

                    y = y1 ;  /* Reposition the pointer to the starting
                    of the second linked list */

                    x = x -> next ;  /* Go to the next node */
                }
            }
        }
}

/*Adds a term to the polynomial in the descending order of the exponent */
void result_add_poly(float c, int e, struct polynode **s)
{
    struct polynode *r, *temp = *s ;

    /* If list is empty or if the node is to be inserted before the first node */
    if(*s == NULL || e > (*s) -> exp)
    {
        *s = r = malloc(sizeof(struct polynode)) ;
        (*s) -> coeff = c ;
        (*s) -> exp = e ;
        (*s) -> next = temp ;
    }
    else
    {
        /* Traverse the entire linked list to search the position to insert a new
node */
        while(temp != NULL)
        {
            if(temp -> exp == e)
            {
                temp -> coeff += c ;
                return ;
            }
```

```
            if(temp -> exp > e && (temp -> next -> exp < e ||
            temp -> next == NULL))
            {
                  r = malloc(sizeof(struct polynode)) ;
                  r -> coeff = c;
                  r -> exp = e ;
                  r -> next = temp -> next ;
                  temp -> next = r ;
                  return ;
            }

            temp = temp -> next ;  /* Go to next node */
      }

      r -> next = NULL ;
      temp -> next = r ;
   }
}
```

The program given above uses **create_poly()** function to create two polynomials.

The **mul_poly()** function is used to multiply the polynomials. The function receives three parameters. The two parameters **x** and **y** point to the list that represents two polynomials. The third parameter is the address of the pointer of resultant list. The variables coeff1 and exp1 are used to hold the values of coefficient and exponent of current resultant node.

Another pointer is made to point to the second list such that after multiplication of all the terms of second list with the first term of first list, the pointer can be repositioned to the first node of the second list.

A condition is checked for the lists to be empty. If the lists are found to be empty the control returns back. If one of the two lists are found empty then the pointer of the resultant list is made to point to another list.

If the lists are not found empty, then a while loop runs till the end of the first list (x!=NULL). In this loop, for the first time, each term of second list is multiplied with the first term of the first list. Then again the pointer is repositioned to the first term of the second list (y=y1). The process continues till each term of the first polynomial is multiplied with the second polynomial.

The function **result_add_poly()** adds the node to the resultant list in the descending order of exponents of the polynomial.

We initialize a structure pointer temp with a value *s, where *s is pointer to the first node of the resultant list. When the function is called for the first time its value is NULL.

The condition here is checked again to find whether the resultant list is empty or not. If it is so then we add the first node. Hence, the memory is allocated for new node, and the value of coefficient and exponent is assigned to the coefficient and exponent part of new node. In the beginning **temp** is stored in the next part of the resultant new node.

When **result_add_poly()** is called to add the second node, we need to compare the exponent value of new node with that of the first node. If the exponent value of new node is greater than the exponent value of the first node in the resultant list, the new node is made the first node.

If both the above conditions are not satisfied, then the resultant list is traversed for searching the proper position where the new node is to be inserted. If the exponent of the same order already exists then simply the coefficient part is added. If it does not exist then the memory is allocated for new node.

INTRODUCTION TO SPARSE MATRIX

Sparse matrix can be defined as a matrix with maximum number of zero entries. A sparse matrix can be divided into two categories:

- N^2 sparse matrix
- Triangular sparse matrix

N^2 sparse matrix is a matrix with zero entries that form a square or a bar.

A sparse matrix with zero entries in its diagonal, either in the upper or lower side, is known as a triangular sparse matrix.

REPRESENTATION OF SPARSE MATRIX

The first method to represent a sparse matrix is called tuple method.

Tuple Method

A sparse matrix can be conveniently stored in the memory using 3-tuple method. Using this method, only the non-zero entries from the given matrix are stored in three tuples. The three tuples are—row, column, and value.

Consider the following sparse matrix with 3 rows and 4 columns.

	Col 1	Col 2	Col 3	Col 4
Row 1	15	0	0	21
Row 2	22	11	0	0
Row 3	0	19	35	16

The 3-tuple representation of above matrix will be:

	Row	Column	Value
a(a)	1	1	15
a(1)	1	4	21
a(2)	2	1	22
a(3)	2	2	11
a(4)	3	2	19
a(5)	3	3	35
a(6)	3	4	16

Index Function Sparse matrix is used to represent the matrix containing many zero elements. In this representation, only the non-zero elements are stored along with their row number and column number. Now consider the following matrix:

$$A = \begin{pmatrix} 4 & 0 & 0 & 0 \\ 3 & 11 & 0 & 0 \\ 1 & 22 & 33 & 0 \\ 7 & 45 & 41 & 22 \end{pmatrix}$$

Triangular Matrix

Row	Column	Value
0	0	0
1	0	3
1	1	11
2	0	1
2	1	22
2	2	33
3	0	7
3	1	45
3	2	41
3	3	22

3-Tuple Representation of matrix A

The above matrix is a triangular matrix having all the zero elements in the upper diagonal and non-zero elements in the lower diagonal.

Index function can be used where all the non-zero elements form a pattern such as a triangle.

Index function gives the sequential location of an array entry.

According to matrix A

Total number of elements in the first row = 1
Total number of elements in the second row = 2
Total number of elements in the third row = 3
" " " "
" " " "
Total number of elements in the $n-1^{th}$ row = n–1
Total number of elements in the n^{th} row = n

Therefore, total number of non-zero elements in matrix

$$A = 1 + 2 + 3 \ldots + n-1 + n = n * \frac{n+1}{2}$$

If we consider the dimension of matrix A as j × k, then total number of elements up to J–1 = $J-1 * \frac{j-1+1}{2}$ (as per the above formula)

$$= j * \frac{j-1}{2}$$

Therefore total number of elements in the j^{th} row and the k^{th} column.

$$= K + \left(j * \frac{j-1}{2} \right)$$

The above formula returns the total number of non-zero elements in matrix A. Since the number of last non-zero element i.e., j, k^{th} element is 10, now by substituting the dimensions of Matrix A in the above formula (i.e. j=4, k=4), we get : $4 + 4 * \frac{4-1}{2} = 10$.

This means that this formula is true to return the position of any j, k^{th} element lying in a sparse matrix. Since, the arrays in C are represented with lower bound 0, therefore, the value returned by the index function will be one more than the actual position of the element in a sparse matrix.

Similarly, Index function can be used to find the index number of $22_{(2, 1)}$ in the sparse matrix. Here, j = 3, k = 2.

$$= 2 + 3 * \frac{3-1}{0}$$

$$= 5$$

In array representation of sparse matrix using 2-tuple method, the index of $22_{(2, 1)}$ will be 5–1=4.

Dope Vector

In order to access each element (string in the above array) in the rectangular array (Fig. 5.1), most compilers of high-level languages simply write the index function into the machine-language. The calculation of the index function for accessing any element at run-time might be slow. Therefore, an auxiliary table is used to access elements in the rectangular array. This table keeps necessary information about each element to access it, e.g. example, address of the element, size, etc. This is known as **Dope vector**.

I	N	F	O	S	Y	S		
M	I	C	R	O	S	O	F	T
O	R	A	C	L	E			
C	I	S	C	O				
T	E	C	H	N	O			

Fig. 5.1 *Rectangular Array*

I	N	F	O	S	Y	S	\0	M	I	C	R	O	S	O	F	T	\0	O	R	A	C	L	E	\0	...

Fig. 5.2 *Row Major Order Representation*

The advantage of a dope vector is that it is much smaller than the rectangular array, and therefore it can be permanently kept into the memory without using too much space. Its entries need to be calculated only once. For all later references to the rectangular array, the compiler can find the position for i, j^{th} element by taking the entry in position i of the auxiliary table, adding **j**, and then moving to the resulting position.

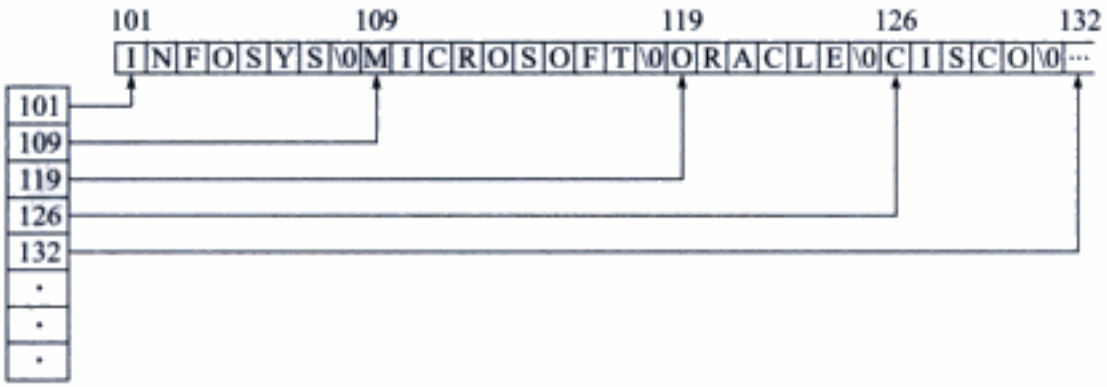


Fig. 5.3 *Dope Vector (Access Table)*

Representation of Sparse Matrix through Arrays

A sparse matrix is one where most of its elements are zero. Consider the following sparse matrix:

$$A = \begin{pmatrix} 15 & 0 & 0 & 21 \\ 22 & 11 & 0 & 0 \\ 0 & 19 & 35 & 16 \end{pmatrix}$$

The idea is to store information of non-zero elements only. If this is done then the matrix may be thought of as an ordered list of non-zero elements. Information about non-zero elements has three parts:

- An integer representing its row.
- An integer representing its column.
- The data associated with its elements.

The elements of the above sparse matrix are represented as follows using array:

1, 1, 15	1, 4, 21	2, 1, 22	2, 2, 11	3, 2, 19	3, 3, 35	3, 4, 16

/* Program to represent sparse matrix in an array */

```
#define MAX1 3
#define MAX2 3

struct sparse
{
  int *sp ;
  int row ;
} ;

void initsparse(struct sparse *) ;
void create_array(struct sparse *) ;
void display(struct sparse) ;
int count(struct sparse) ;
```

```
void create_tuple(struct sparse *, struct sparse) ;
void display_tuple(struct sparse) ;
void delsparse(struct sparse *) ;

void main( )
{
        struct sparse s1, s2 ;
    int c ;

        clrscr( );

    initsparse(&s1) ;
    initsparse(&s2 ) ;

    create_array(&s1) ;
    printf("\nElements in Sparse Matrix: ") ;
        display(s1) ;

        c = count(s1) ;
        printf("\n\nNumber of non-zero elements: %d", c) ;

        create_tuple(&s2, s1) ;
        printf("\n\nArray of non-zero elements: ") ;
    display_tuple(s2) ;

    delsparse(&s1) ;
    delsparse(&s2) ;

    getch( ) ;
}

/* Initialises element of structure */
void initsparse(struct sparse *p)
{
  p -> sp = NULL ;
}
/* Dynamically creates the matrix of size MAX1 x MAX2 */
void create_array(struct sparse *p)
{
  int n, i ;

  p -> sp =(int *)malloc(MAX1 * MAX2 * sizeof(int)) ;
```

```
    for(i = 0 ; i < MAX1 * MAX2 ; i++)
    {
                printf("Enter element no. %d: ", i) ;
                scanf("%d", &n) ;
                * (p -> sp + i) = n ;
    }
}

/* Displays the contents of the matrix */
void display(struct sparse p)
{
    int i ;

    /* Traverses the entire matrix */
    for(i = 0 ; i < MAX1 * MAX2 ; i++)
    {
        /* Positions the cursor to the new line for every new row */
        if(i % MAX2 == 0)
                printf("\n") ;
        printf("%d\t", * (p.sp + i)) ;
    }
}

/* Counts the number of non-zero elements */
int count(struct sparse p)
{
    int cnt = 0, i ;

    for(i = 0 ; i < MAX1 * MAX2 ; i++)
    {
        if(* (p.sp + i) != 0)
                cnt++ ;
    }
    return cnt ;
}

/* Creates an array that stores information about non-zero elements */
void create_tuple(struct sparse *p, struct sparse s)
{
    int r = 0 , c = -1, l = -1, i ;

    p -> row = count(s) + 1 ;
```

```
  p -> sp = (int *) malloc (p -> row * 3 * sizeof(int)) ;
    *(p -> sp + 0) = MAX1 ;
    *(p -> sp + 1) = MAX2 ;
    *(p -> sp + 2) = p -> row - 1 ;
    l = 2 ;
  for(i = 0 ; i < MAX1 * MAX2 ; i++)
  {
       c++ ;

       /* Sets the row and column values */
       if(((i % MAX2) == 0) && (i != 0))
       {
              r++ ;
              c = 0 ;
       }

       /* Checks for non-zero element
          row, column and non-zero element value
          is assigned to the matrix */
       if(* (s.sp + i) != 0)
       {
              l++ ;
              *(p -> sp + l) = r ;
              l++ ;
              *(p -> sp + l) = c ;
              l++ ;
              *(p -> sp + l) = *(s.sp + i) ;
       }
  }
}
/* Displays the contents of 3-tuple */
void display_tuple(struct sparse p)
{
    int i ;
  for(i = 0 ; i < p.row * 3 ; i++)
  {
       if(i % 3 == 0)
              printf("\n") ;
       printf("%d\t", * (p.sp + i)) ;
  }
}
```

```
/* Deallocates memory */
void delsparse(struct sparse *p)
{
  free(p -> sp) ;
}
In the program above a structure sparse is designed as:
struct sparse
{
  int *sp ;
  int row ;
} ;
```

The function **create_array()** dynamically creates a matrix of size max1 × max2. The values are taken for the matrix and the **display()** function displays the content of sparse matrix. The **count()** function counts the total number of non-zero elements.

The **create_tuple()** function creates a 2-dimensional array dynamically. The problem of how much space to be allocated is there. Since each row in the 3-tuple form represents non-zero elements in the original array the new array should contain as many rows as the number of non-zero elements in the original matrix. Hence, the first row of new array should contain number of rows, number of columns, and the number of non-zero elements in the original array. The size of the new array can be determined through the following statements:

```
p → row = count(s) +1;
p → sp = (int *) maloc (row * 3 * size(int));
```

The first statement given above can count the number of non-zero elements in the array. To count we have added 1. The 0^{th} row stores the information about number of rows the 1^{st} row stores, the row and column position of non-zero elements. The number of rows depends on the number of non-zero elements in the array, therefore, the array is created dynamically. There are 3 columns in the array. The 0^{th} column stores the rows number of non-zero element. The 1^{st} column stores the column number of the non-zero element and 2^{nd} column stores the value of non-zero element.

The display_tuple() function displays the contents of 3-tuple.

Sparse Matrix Operation Using Arrays

Addition of Two Sparse Matrices

/* Program to add two sparse matrices */

```
#define MAX1 3
#define MAX2 3
#define MAXSIZE 9
#define BIGNUM 100

struct sparse
{
```

```
  int *sp ;
  int row ;
  int *result ;
} ;

void initsparse(struct sparse *) ;
void create_array(struct sparse *) ;
int count(struct sparse) ;
void delsparse (struct sparse *) ;
/* Counts the number of non-zero elements */
int count(struct sparse s)
{
  int cnt = 0, i ;
  for(i = 0 ; i < MAX1 * MAX2 ; i++)
  {
      if(* (s.sp + i) != 0)
            cnt++ ;
  }
  return cnt ;
}

/* Carries out addition of two matrices */
void addmat(struct sparse *p, struct sparse s1, struct sparse s2)
{
  int i = 1, j = 1, k = 1 ;
  int elem = 1 ;
  int max, amax, bmax ;
  int rowa, rowb, cola, colb, vala, valb ;

  /* Get the total number of non-zero values
     from both the matrices */
  amax = *(s1.sp + 2) ;
  bmax = *(s2.sp + 2) ;
  max = amax + bmax ;

  /* Allocate memory for result */
  p -> result = (int *) malloc(MAXSIZE * 3 * sizeof(int)) ;

  while(elem <= max)
  {
      /* Check if i < max, non-zero values
         in first 3-tuple and get the values */
```

```
if(i <= amax)
{
        rowa =  *(s1.sp + i * 3 + 0) ;
        cola =  *(s1.sp + i * 3 + 1) ;
        vala =  *(s1.sp + i * 3 + 2) ;
}
else
        rowa = cola = BIGNUM ;

/* Check if j < max. non-zero values
   in second 3-tuple and get the values */
if(j <= bmax)
{
        rowb = *(s2.sp + j * 3 + 0) ;
        colb = *(s2.sp + j * 3 + 1) ;
        valb = *(s2.sp + j * 3 + 2) ;
}
else
    rowb = colb = BIGNUM ;

/* If row no. of both 3-tuple are same */
if(rowa == rowb)
{
        /* If col no. of both 3-tuple are same */
        if(cola == colb)
        {
                /* Add tow non-zero values
                   store in result */
                *(p -> result + k * 3 + 0) = rowa ;
                *(p -> result + k * 3 + 1) = cola ;
                *(p -> result + k * 3 + 2) = vala + valb ;
                i++ ;
                j++ ;
                max-- ;
        }

        /* If col no. of first 3-tuple is < col no. of
           second 3-tuple. then add info. as it is
           to result */
        if(cola < colb)
        {
                *(p -> result + k * 3 + 0) = rowa ;
```

```
                *(p -> result + k * 3 + 1) = cola ;
                *(p -> result + k * 3 + 2) = vala ;
                i++ ;
        }

        /* If col no. of first 3-tuple is > col no. of
           second 3-tuple, then add info. as it is
           to result */
        if(cola > colb)
        {
                *(p -> result + k * 3 + 0) = rowb ;
                *(p -> result + k * 3 + 1) = colb ;
                *(p -> result + k * 3 + 2) = valb ;
                j++ ;
        }
        k++ ;
}

/* If row no. of first 3-tuple is < row no. of
   second 3-tuple, then add info. as it is
  to result */
if(rowa < rowb)
{
        *(p -> result + k * 3 + 0) = rowa ;
        *(p -> result + k * 3 + 1) = cola ;
        *(p -> result + k * 3 + 2) = vala ;
        i++ ;
        k++ ;
 }

/* If row no. of first 3-tuple is > row no. of
   second 3-tuple, then add info. as it is
   to result */
if(rowa > rowb)
{
        *(p -> result + k * 3 + 0) = rowb ;
        *(p -> result + k * 3 + 1) = colb ;
        *(p -> result + k * 3 + 2) = valb ;
        j++ ;
        k++ ;
}
elem++ ;
```

```
    }

    /* Add info about the total no. of rows,
       cols, and non-zero values that the resultant array
       contains to the result */
    *(p -> result + 0) = MAX1 ;
    *(p -> result + 1) = MAX2 ;
    *(p -> result + 2) = max ;
}

/* Deallocates memory */
void delsparse(struct sparse *p)
{
    if(p -> sp != NULL)
        free(p -> sp) ;
    if(p -> result != NULL)
        free(p -> result) ;
}
```

The function **addmat()** carries out addition of two sparse matrices. The function firstly obtains the total number of non-zero elements that the 3-tuple would hold. The following statements help in the same:

```
amax = *(s1.sp+2);
bmax = *(s2.sp+2);
 max = amax + bmax;
```

The memory is then allocated for the target 3-tuple that stores the results of addition. The **while** loop helps in carrying out the operation of addition. The **i** and **j** variables are used as counters for first 3-tuple (pointed by s1.sp) and second 3-tuple (pointed by s2.sp). Then we retrieve the row number, column number and non-zero value of i^{th} and j^{th} non-zero elements respectively. The cases which are considered while addition are as follows:

- If the row numbers as well as column numbers of the non-zero values retrieved from first and second 3-tuple (pointed by s1.sp and s2.sp respectively) are same then two non-zero values vala and valb are added. The row number **rowa**, column number **cola** and **vala + valb** is then copied to the target 3-tuple pointed by the result.
- If the column number of first 3-tuple is less than the column number of second 3-tuple, then we have added the information about the i^{th} non-zero value of first 3-tuple to the target 3-tuple.
- If the column number of first 3-tuple is greater than the column number of second 3-tuple, then we have added the information about the j^{th} non-zero value of second 3-tuple to the target 3-tuple.
- If row number of first 3-tuple is less than the row number of second 3-tuple then we have added the information about the i^{th} non-zero value of first 3-tuple to the target 3-tuple.
- If row number of first 3-tuple is greater than the row number of second 3-tuple, then we have added the information about the j^{th} non-zero value of second 3-tuple to target 3-tuple.

Then the total number of rows, columns and non-zero values that the target 3-tuple holds are stored in the zeroth row of the target 3-tuple (pointed to by result).

The function **display()** displays the result of addition operation.

Multiplication of Two Sparse Matrices

/* Program to multiply two matrices */

```
#define MAX1 3
#define MAX2 3
#define MAXSIZE 20

#define TRUE 1
#define FALSE 2

struct sparse
{
  int *sp ;
  int row ;
  int *result ;
} ;

void sparseprod(struct sparse *, struct sparse, struct sparse) ;
void search_nonzero(int *sp, int ii, int*p, int*flag) ;
void searchinb(int *sp, int jj, int colofa, int*p, int*flag) ;
void main( )
{
  struct sparse s[5] ;
    int i ;

    clrscr( ) ;
  for(i = 0 ; i <= 3 ; i++)
  initsparse(&s[i]) ;

  create_array(&s[0]) ;

  create_tuple(&s[1], s[0]) ;
  display_tuple(s[1]) ;

  create_array(&s[2]) ;

  create_tuple(&s[3], s[2]) ;
  display_tuple(s[3]) ;
```

```
  sparseprod(&s[4], s[1], s[3]) ;

  printf("\nResult of multiplication of two matrices: ") ;
  display_result(s[4]) ;

    for(i = 0 ; i <= 3 ; i++)
  delsparse(&s[i]) ;

    getch( ) ;
}

/* Performs multiplication of sparse matrices */
void sparseprod(struct sparse *p, struct sparse a, struct sparse b)
{
  int sum, k, position, pose, flaga, flagb, i , j ;
    k = 1 ;

    p -> result = (int *) malloc(MAXSIZE * 3 * sizeof(int)) ;
  for ( i = 0 ; i < * ( a.sp + 0 * 3 + 0 ) ; i++ )
  {
      for ( j = 0 ; j < * ( b.sp + 0 * 3 + 1 ) ; j++  )
      {
            /* Search if an element is present at ith row */
            search_nonzero(a.sp, i, &position, &flaga) ;
            if(flaga == TRUE)
            {
                  sum = 0 ;

                  /* Run loop till there are elements at ith row
                     in first 3-tuple */
                  while(* (a.sp + position * 3 + 0) == i)
                  {
                        /* Search if an element is present at ith
                           column,
                           in second 3-tuple */
                  searchinb(b.sp, j, * (a.sp + position * 3 + 1),
                                    &posi, &flagb) ;

                        /* If found then multiply */
                        if(flagb == TRUE)
                  sum = sum + *(a.sp + position * 3 + 2) *
                                    *(b.sp + posi * 3 + 2) ;
```

```
                        position = position + 1 ;
                    }
                    /* Add result */
                    if(sum != 0)
                    {
                        * (p -> result + k * 3 + 0) = i ;

                        *(p -> result + k * 3 + 1) = j ;
                        *(p -> result + k * 3 + 2) = sum ;
                        k = k + 1 ;
                    }
                }
            }
        }

        /* Add total no. of rows, cols and non-zero values */
        *(p -> result + 0 * 3 + 0) = *(a.sp + 0 * 3 + 0) ;
        *(p -> result + 0 * 3 + 1) = *(b.sp + 0 * 3 + 1) ;
        *(p -> result + 0 * 3 + 2) = k - 1 ;
}

/* Searches if an element present at ith row */
void search_nonzero(int *sp, int ii, int *p, int *flag)
{
   int j ;
   *flag = FALSE ;
   for(j = 1 ; j <= * (sp + 0 * 3 + 2 ) ; j++)
   {
        if(* (sp + j * 3 + 0) == ii)
        {
                *p = j ;
                *flag = TRUE ;
                return ;
        }
    }
}

/* Searches if an element where col. of first 3-tuple
   is equal to row of second 3-tuple */
void searchinb(int *sp, int jj, int colofa, int *p, int *flag )
{
     int j ;
```

```
    *flag = FALSE ;
   for(j = 1 ; j <= *(sp + 0 * 3 + 2 ) ; j++)
    {
      if(* (sp + j * 3 + 1) == jj && *(sp + j * 3 + 0) == colofa)
      {
            *p = j ;
            *flag = TRUE ;
            return ;
      }
    }
}
```

The program for multiplication of sparse matrix holds three functions:

sparseprod() function. Using this we have allocated memory required to store the resultant tuple. The multiplication is done using the **for** loops. The outer **for** loop runs the number of times which is equal to the row dimension of first 3-tuple (pointed by a).

The inner **for** loop runs for the number of times, which is equal to column dimensions of second 3-tuple (pointed by b).

search_nonzero(). This function checks whether or not an element is present at i^{th} row. The flag is set to TRUE if a non-zero element is there and stores the position in **pos** using which a non-zero element in i^{th} row is retrieved. A **while** loop is run till there are elements at i^{th} row in a.

searchinb(). This function is called in while loop which searches for an element whose row number is equal to column number of an element of a currently being considered and column number is equal to j.

Transpose of Sparse Matrix

/* Program to transpose a sparse matrix */

```
#define MAX1 3
#define MAX2 3

struct sparse
{
  int *sp ;
  int row ;
} ;

int count(struct sparse) ;
void transpose(struct sparse *, struct sparse) ;
void display_transpose(struct sparse) ;
void main( )
{
  struct sparse s[3] ;
    int c, i ;
```

```
    for(i = 0 ; i <= 2 ; i++)
      initsparse(&s[i]) ;

    clrscr( ) ;
  create_array(&s[0]) ;
  printf("\nElements in Sparse Matrix:") ;
  display(s[0]) ;
  c = count(s[0]) ;
  printf("\n\nNumber of non-zero elements: %d", c) ;

  create_tuple(&s[1], s[0]) ;
  printf("\n\nArray of non-zero elements: ") ;
  display_tuple(s[1]) ;

  transpose(&s[2], s[1]) ;
  printf("\n\nTranspose of array:") ;
  display_transpose(s[2]) ;

  for(i = 0 ; i <= 2 ; i++)
      delsparse(&s[i]) ;
    getch( ) ;
}

/* Obtains transpose of an array */
void transpose(struct sparse *p, struct sparse s)
{
  int x, q, pos_1, pos_2, col, elem, c, y ;

    /* Allocate memory */
    p -> sp = (int *) malloc(s.row * 3 * sizeof(int)) ;
    p -> row = s.row ;

    /* Store total number of rows, cols
       and non-zero elements */
    *(p -> sp + 0) = *(s.sp + 1) ;
    *(p -> sp + 1) = *(s.sp + 0) ;
    *(p -> sp + 2) = *(s.sp + 2) ;

    col = *(p -> sp + 1) ;
    elem = *(p -> sp + 2) ;

    if(elem <= 0)
        return ;
```

```
    x = 1 ;

    for(c = 0 ; c < col ; c++)
    {
        for(y = 1 ; y <= elem ; y++)
        {
            q = y * 3 + 1 ;
            if(*(s.sp + q) == c)
            {
                pos_2 = x * 3 + 0 ;
                pos_1 = y * 3 + 1 ;
                *(p -> sp + pos_2) = *(s.sp + pos_1) ;
                pos_2 = x * 3 + 1 ;
                pos_1 = y * 3 + 0 ;
                *(p -> sp + pos_2) = *(s.sp + pos_1) ;

                pos_2 = x * 3 + 2 ;
                pos_1 = y * 3 + 2 ;
                *(p -> sp + pos_2) = *(s.sp + pos_1) ;

                x++ ;
            }
        }
    }
}
/* Displays 3-tuple after transpose operation */
void display_transpose(struct sparse p)
{
    int i ;

  for(i = 0 ; i < p.row * 3 ; i++)
  {
      if(i % 3 == 0)
            printf("\n") ;
      printf("%d\t", *(p.sp + i)) ;
  }
}
```

In the transpose() function we allocate memory to store the elements in 3-tuple. The total number of rows, columns and non-zero elements are also stored by using the following statements:

```
*(p → sp + 0) = *(s . sp + 1);
*(p → sp + 1) = *(s . sp + 0);
*(p → sp + 2) = *(s . sp + 2);
```

The statement **p → sp** is used to store the total number of columns, where the total number of rows should be stored. Similarly, the place where total number of columns should get stored, we have stored total number of rows. In the case of transpose, total number of rows becomes equal to the total number of columns.

The for loop is used to carry out the transpose operation. The outer for loop runs till the non-zero elements of **col** number of columns. In the inner for loop, we obtain the position at which column number of a non-zero element is stored. This is done through the statement:

```
q = y * 3 + 1;
```

Then it is checked that whether the column number of a non-zero element matches with the column number currently being considered, i.e. **C.** If the two values match, then the information is stored in target 3-tuple using the following statements:

```
pos_2 = x * 3 + 0;
pos_1 = y * 3 + 1;
*(p → sp + pos_2) = *(s.sp + pos_1);
```

The variable **pos_2** is used for the target 3-tuple, to store the position at which the data from source 3-tuple should get copied. The variable pos_1 is used for source 3-tuple to extract data from it. The third statement copies the column position of non-zero element from source 3-tuple to target 3-tuple. This column number gets stored at the row position in target 3-tuple.

Similarly, the row position of a non-zero element of source 3-tuple is copied at the column position of target 3-tuple.

```
pos_2 = x * 3 + 1;
pos_1 = y * 3 + 0;
*p → sp + pos_2 = *(s.sp + pos_1);
```

The non-zero value from source 3-tuple is copied to target 3-tuple using the following statements:

```
pos_2 = x * 3 + 2;
pos_1 = y * 3 + 2;
*(p → sp + pos_2) = *(s.sp + pos_1);
```

Thus, the target 3-tuple is transpose of an array the user entered for **create_array()** function. The elements are displayed using display-transpose function.

REPRESENTATION OF SPARSE MATRIX THROUGH LINKED LISTS

The sparse matrix can also be represented using a linked list. An element of sparse matrix consists of three integers

- its row number (i)
- its column number (j) and
- its value (val)

In linked representation, consider "head" nodes for each row and each column pointing to the elements in a particular row or column. A **head** node for a row consists of three parts (Fig. 5.4).

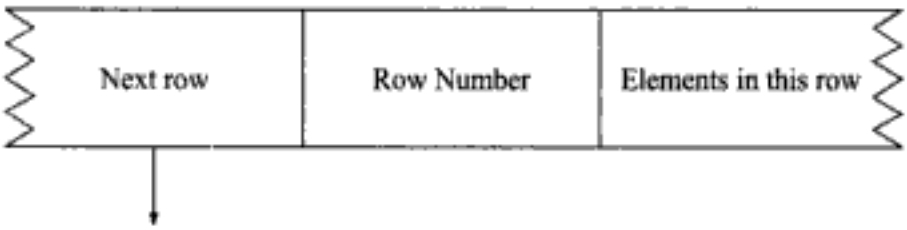


Fig. 5.4 *Parts of Head Node for a Row*

Row number indicates the row to which the "head" node is pointing to by component "Element". The head node also points to another "head" node for the next row.

The structure definition for Rowhead is

```
typedef struct Rowhead
{
    int rownum;
    Element * right;
    struct Rowhead * next;
} Row;
```

Similarly, a column consists of column number, a pointer pointing to next column and the pointer to elements in that column. The structure definition of colhead will be:

```
typedef struct columnhead
{
    int column;
    Element * down;
    struct columnhead * next;
    } Col;
```

The structure definition for Element will be:

```
typedef struct Element
{
    int i, j, val;
    struct Element * down;
    struct Element * right;
} Element;
```

The structure defined for one element can be shown as in Fig. 5.5:

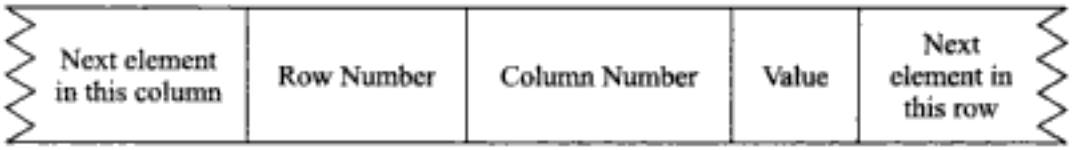


Fig. 5.5 *Structure for an Element*

A sparse matrix can be defined as a node having two pointers, one pointing to the list of rows and other pointing to the list of columns. The node also contains two integers specifying the number of rows and the number of columns. The node can be shown as in Fig. 5.6:

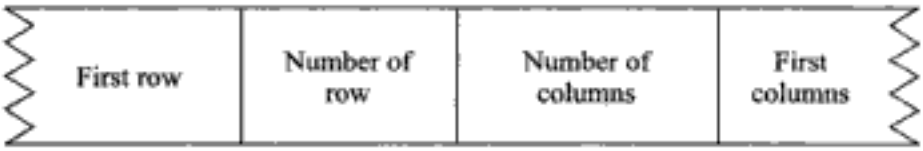


Fig. 5.6 *Node*

The structure definition of the node depicted above will be:

```
typedef struct sparsematrix
{
  Row * firstrow;
  Col * firstcol;
  int totrows;
  int totcols;
} sparsematrix;
```

/* Program to store sparse matrix as a linked list */

```
#define MAX1 3
#define MAX2 3

/* Structure for col headnode */
struct cheadnode
{
  int colno ;
  struct node *down ;
  struct cheadnode *next ;
} ;
/* Structure for row headnode */
struct rheadnode
{
  int rowno ;
  struct node * right ;
  struct rheadnode *next ;
} ;

/* Structure for node to store element */
struct node
{
  int row ;
  int col ;
  int val ;
  struct node *right ;
  struct node *down ;
```

```
} ;

/* Structure for special headnode */
struct spmat
{
  struct rheadnode *firstrow ;
  struct cheadnode *firstcol ;
  int noofrows ;
  int noofcols ;
} ;

struct sparse
{
  int *sp ;
  int row  ;
  struct spmat *smat ;
  struct cheadnode *chead[MAX2] ;
  struct rheadnode *rhead[MAX1] ;
  struct node *nd ;
} ;

void initsparse(struct sparse *) ;
void create_array(struct sparse *) ;
void display(struct sparse) ;
int count(struct sparse) ;
void create_triplet(struct sparse *, struct sparse) ;
void create_llist(struct sparse *) ;
void insert(struct sparse *, struct spmat *, int, int, int) ;
void show_llist(struct sparse) ;
void delsparse(struct sparse *) ;

void main( )
{
  struct sparse s1, s2 ;

    clrscr( ) ;

    initsparse(&s1) ;
    initsparse(&s2) ;

  create_array(&s1) ;

  printf("\nElements in sparse matrix: ") ;
```

```
  display(s1) ;
  create_triplet(&s2, s1) ;

  create_llist(&s2) ;
  printf( "\n\nInformation stored in linked list : ") ;
  show_llist(s2) ;

    delsparse(&s1) ;
    delsparse (&s2) ;

    getch( ) ;
}

/* Initializes structure elements */
void initsparse(struct sparse *p)
{
    int i ;
  /* Create row headnodes */
  for(i = 0 ; i < MAX1 ; i++)
       p -> rhead[i] = (struct rheadnode *) malloc(sizeof(struct rheadnode)) ;

  /* Initialize and link row headnodes together */
  for(i = 0 ; i < MAX1 - 1 ; i++)
  {
       p -> rhead[i] -> next = p -> rhead[i + 1] ;
       p -> rhead[i] -> right = NULL ;
       p -> rhead[i] -> rowno = i ;
  }
  p -> rhead[i] -> right = NULL ;
  p -> rhead[i] -> next = NULL ;
  /* Create col headnodes */
  for(i = 0 ; i < MAX1 ; i++)
       p -> chead[i] = (struct cheadnode *) malloc(sizeof(struct cheadnode)) ;

  /* Initialize and link col headnodes together */
  for(i = 0 ; i < MAX2 - 1 ; i++)
  {
       p -> chead[i] -> next = p -> chead[i + 1] ;
       p -> chead[i] -> down = NULL ;
       p -> chead[i] -> colno = i ;
  }
  p -> chead[i] -> down = NULL ;
```

```
    p -> chead[i] -> next = NULL ;

    /* Create and initialize special headnode */
    p -> smat = (struct spmat *) malloc(sizeof(struct spmat)) ;
    p -> smat -> firstcol = p -> chead[0] ;
    p -> smat -> firstrow = p -> rhead[0] ;
    p -> smat -> noofcols = MAX2 ;
    p -> smat -> noofrows = MAX1 ;
}

/* Creates, dynamically the matrix of size MAX1 x MAX2 */
void create_array(struct sparse *p)
{
    int n, i ;

    p -> sp = (int *) malloc(MAX1 * MAX2 * sizeof(int)) ;

    /* Get the element and store it */
    for(i = 0 ; i < MAX1 * MAX2 ; i++)
    {
        printf("Enter element no. %d:", i) ;
        scanf("%d", &n) ;
        *(p -> sp + i) = n ;
    }
}

/* Displays the contents of the matrix */
void display(struct sparse s)
{
    int i ;

    /* Traverses the entire matrix */
    for(i = 0 ; i < MAX1 * MAX2 ; i++)
    {
        /* Positions the cursor to the new line for every new row */
        if(i % MAX2 == 0)
            printf("\n") ;
        printf("%d\t", *(s.sp + i)) ;
    }
}

/* Counts the number of non-zero elements */
```

```
int count(struct sparse s)
{
  int cnt = 0, i ;

  for(i = 0 ; i < MAX1 * MAX2 ; i++)
  {
       if(*(s.sp + i) != 0)
              cnt++ ;
  }
  return cnt ;
}

/* Creates an array of triplet containing info. about non-zero elements */
void create_triplet(struct sparse *p, struct sparse s)
{
  int r = 0 , c = -1, l = -1, i ;

  p -> row = count ( s ) ;
  p -> sp = (int *) malloc(p -> row * 3 * sizeof(int)) ;

  for(i = 0 ; i < MAX1 * MAX2 ; i++)
  {
       c++ ;
       /* Sets the row and column values */
       if(((i % MAX2) == 0) &&(i != 0))
       {
              r++ ;
              c = 0 ;
       }

       /* Checks for non-zero element. Row, column and
          non-zero element value is assigned to the matrix */

       if(*(s.sp + i) != 0)
       {
              l++ ;
              *(p -> sp + l) = r ;
              l++ ;
              *(p -> sp + l) = c ;
              l++ ;
              *(p -> sp + l) = *(s.sp + i) ;
       }
```

```
    }
}

/* Stores information of triplet in a linked list form */
void create_llist(Struct sparse *p)
{
   int j = 0, i ;
   for(i = 0 ; i < p -> row ; i++, j+= 3)
        insert(p, p -> smat, *(p -> sp + j), *(p -> sp + j + 1),
               *(p -> sp + j + 2)) ;
}

/* Inserts element to the list */
void insert(struct sparse *p, struct spmat *smat , int r, int c, int v)
{
    struct node *temp1, *temp2 ;
    struct rheadnode *rh ;
    struct cheadnode *ch ;
    int i, j ;

   /*  Allocate and initialize memory for the node */
   p -> nd = (struct node *) malloc(sizeof(struct node)) ;
   p -> nd -> col = c ;
   p -> nd -> row = r ;
   p -> nd -> val = v ;

   /* Get the first row headnode */
   rh = smat -> firstrow ;

   /* Get the proper row headnode */
   for(i = 0 ; i < r ; i++)
        rh = rh -> next ;
   temp1 = rh -> right ;

   /* If no element added in a row */
   if(temp1 == NULL)
   {
        rh -> right = p -> nd ;
        p -> nd -> right = NULL ;
   }
   else
```

```
    {
        /* Add element at proper position */
        while((temp1 != NULL) && (temp1 -> col < c))
        {
                temp2 = temp1 ;
                temp1 = temp1 -> right ;
        }
        temp2 -> right = p -> nd ;
        p -> nd -> right = NULL ;
    }

    /* Link proper col headnode with the node */
    ch = p -> smat -> firstcol ;
    for(j = 0 ; j < c ; j++)
        ch = ch -> next ;
    temp1 = ch -> down ;

    /* If col not pointing to any node */
    if(temp1 == NULL)
    {
        ch -> down = p -> nd ;
        p -> nd -> down = NULL ;
    }
    else
    {
    /* Link previous node in column with next node in same column */
        while((temp1 != NULL) && (temp1 -> row < r))
        {
                temp2 = temp1 ;
                temp1 = temp1 -> down ;
        }
        temp2 -> down = p -> nd ;
        p -> nd -> down = NULL ;
    }
}

void show_llist(struct sparse s)
{
    struct node *temp ;
    /* Get the first row headnode */
    int r = s.smat -> noofrows ;
    int i ;
```

```
  printf("\n") ;

  for(i = 0 ; i < r ; i++)
  {
       temp = s.rhead[i] -> right ;
       if(temp != NULL)
       {
              while(temp -> right != NULL)
              {
                     printf("Row: %d Col: %d Val: %d\n", temp -> row,
                     temp -> col, temp -> val) ;
                     temp = temp -> right ;
              }
              if(temp -> row == i)
                     printf("Row: %d Col: %d Val: %d\n", temp -> row,
                     temp -> col, temp -> val) ;
       }
  }
}

/* Deallocates memory */
void delsparse(struct sparse *p)
{
  int r = p -> smat -> noofrows ;
  struct rheadnode *rh ;
  struct node *temp1, *temp2 ;
    int i, c ;

  /* Deallocate memory of nodes by traversing rowwise */
  for(i = r - 1 ; i >= 0 ; i--)
  {
       rh = p -> rhead[i] ;
       temp1 = rh -> right ;

       while(temp1 != NULL)
       {
              temp2 = temp1 -> right ;
              free(temp1) ;
              temp1 = temp2 ;
       }
  }

  /* Deallocate memory of row headnodes */
```

```
    for(i = r - 1 ; i >= 0 ; i--)
        free(p -> rhead[i]) ;

    /* Deallocate memory of col headnodes */
    c = p -> smat -> noofcols ;
    for(i = c - 1 ; i >= 0 ; i--)
        free(p -> chead[i]) ;
}
```

The program above first creates the matrix of size MAX1 × MAX2. The variable **s1** of type struct sparse holds this matrix. The function triplet contains the information about the non-zero elements of the matrix. The variable **s2** stores the triplet. The **create_list** () function stores the information in the form of linked list.

To add nodes in the linked list the **create_list** function calls the **insert()** function. The number of iterations of for loop depends on the number of rows in the triplet.

The **insert()** function accepts a pointer **smat** to the special node, the row number **r**, column number **c,** and the value **v** of the non-zero element. The new node is created by **nd**. Then, the **row**, **col** and **val** of this node has been initialized. To place this node in the rowlist at a proper position we need to know the first row of head node. This we have retrieved from the field **firstrow** of **smat**. To get the proper row head node for the node **nd** the following statements have been written:

```
for(i = 0; i < r; i++)
rh = rh → next;
```

The statement **temp1 = rh → right**; stores the address of first row. If the rowlist is empty then **temp1** would be NULL. If not, then the node **nd** is the first node in the rowlist and the field right would be made to point to the node **nd**. If **temp1** is not NULL then the last node in the rowlist is found out. The node in the rowlist whose right field stores NULL and column number stored in **col** is less than the column number of **nd,** would be the last node. The right field of that node would point to node **nd**.

This new node **nd** should also be linked with proper column head node. The first column head node is known by **smat**. The proper column head node **ch** is retrieved through a for loop. The **down** field of **ch** gives the address of the node, which is the first node in the column list. If column list is empty then **temp1** would be NULL. If this is the case then **nd** would be the first node in the column list and field **down** of **ch** would be made to point to the node **nd**.

Again if **temp1** is not NULL, the list is traversed to get the last node in the list. The node in the column list whose **down** field stores NULL value and row number stored in **row** is less than the row number of **nd** would be the last node.

The **show_list()** function reads and displays the data stored in linked list. First, the list is traversed rowwise, by doing so we know about the total number of row head nodes in the list. This is retrieved from the member number of rows of **smat**. A for loop runs for **r** times where r is number of rows. In every iteration of for loop, a rowlist is traversed with **rhead[i]** till we get a node whose right field is NULL and thus displays the data of each node in the row list.

REPRESENTATION OF COMPLEX NUMBERS

The ability to create a composite data structure is supported by C/C++ programming languages through the use of structures. A structure is a collection of one or more variables that are grouped together under

a single name. Thus, a structure allows a group of related variables to be treated as a unit instead of separate entities. A structure is declared as follows:

```
struct
{
    member list;
} structure name;
```

where member list is any valid list of variable declarations, and structure name is the variable given to the structure.

As an example, consider complex numbers. C/C++ do not provide a built-in data type for representing complex numbers. A complex number contains a real and imaginary part, it makes sense to group two parts of such a number together when using them in a program.

The structure defined for a complex number can be:

```
struct
{
    float real;
    float imaginary;
} complex_num;
```

To gain access to individual structure members, a structure membership operator must be used. For example:

```
complex_num.real = 3.2;
```

If we are given the address of structure variables, then the structure membership operator used will be:

```
(&complex_num) → real = 3.2;
```

Both the statements given above will assign the real part of the complex_num to the value of 3.2.

Summary

- Each term in the polynomial expression consists of two parts—a coefficient and an exponent.
- The simplest way to represent a polynomial of degree "n" is to store the coefficient of (n+1) terms of a polynomial in an array.
- Sparse matrix can be defined as a matrix with maximum number of zero entries.
- A sparse matrix can be stored in the memory using 3-stuple method. Using this method, only the non-zero entries from the given matrix are stored in three tuples—row, column, and value.
- The ability to create a composite data structure is supported by C/C++ programming languages through the use of structures. A structure is a collection of one or more variables that are grouped together under a single name.
- A structure allows a group of related variables to be treated as a unit instead of separate entities.

Review Exercise

Multiple Choice Questions

1. Sparse matrices are represented using
   a. singly linked list where each node contains non-zero elements of the matrix
   b. circular linked list for each row and column with each row corresponding to non-zero element.
   c. a binary tree
   d. a doubly linked list
2. A complex number contains
   a. a tangible and an intangible part
   b. a real and an imaginary part
   c. a decimal and a fractional part
   d. None of the above
3. An element of sparse matrix consists of integers_____.
   a. two
   b. three
   c. six
   d. ten
4. A sparse matrix is one where most of its elements are
   a. even
   b. prime
   c. zero
   d. odd

Fill in the Blanks

1. Polynominals in memory can be represented by _____ lists.
2. For representing polynomial in memory using linked list each node must have _____ fields.
3. A polynomial is made of different terms each of which consists of a _____ and _____.
4. A sparse matrix with zero entries in its diagonal either in the upper or lower side is known as _____.

State whether True or False

1. A polynomial in two variables can be represented as a linked list of structure.
2. The index function can be used to represent polynomials in computers.
3. Index function gives the sequential location of an array entry.
4. A sparse matrix cannot be stored in the memory using 3-tuple method.

Descriptive Questions

1. Define polynomial as an Abstract Data Type. Write a 'C' function to multiply two polynomials and to return the product.
2. Write 'C' functions to add and multiply two large integers which cannot be represented by built-in data types.

3. Implement the following complex ADT using the 'C' programming language. The complex ADT is used to represent complex numbers of the form $z = \alpha + i\beta$ where α and β are real numbers and i is the imaginary number. The operations supported by this ADT are:
   i. Addition (z1, z2).

   [Note: $(\alpha_1 + i\beta_1) + (\alpha_2 + i\beta_2) = (\alpha_1 + \alpha_2) + i\,(\beta_1 + \beta_2)$**]**

   ii. Subtraction (21, 22)

   [Note: $(\alpha_1 - i\beta_1) - (\alpha_2 - i\beta_2) = (\alpha_1 - \alpha_2) - i\,(\beta_1 - \beta_2)$**]**
4. Write a 'C' function to compute the product 'C' of two sparse matrices 'A' and 'B' represented as ordered lists instead of two-dimensional arrays. To compute one element in 'C' dot product of one row of 'A' and one column of 'B' is to be evaluated, for this purpose, compute the transpose of B first. Now each row in transpose of 'B' corresponds to a column in B. B' now can be used for efficient multiplication.
5. Develop an ADT specification for "Polynominals". Also include the operations associated with polynominals.
6. Suggest a suitable data structure for representation of imaginary numbers. An imaginary number is represented by a+ib where i is the iota for the number. Also give specification for the operation associated with them.

CHAPTER

6 Stacks

Key Features

- Introduction to Stacks
- Stack as an Abstract Data Type
- Representation of Stacks through Arrays
- Representation of Stacks through Linked Lists
- Applications of Stacks
- Stacks and Recursion

This chapter explores one of the most important data structures—Stack. The stack has been defined as an abstract data type which serves as a concrete and valuable tool for problem solving.

A stack is a linear data structure where all the insertions and deletions are done at end rather than in the middle. A stack can be implemented by using both arrays and linked lists.

The main application of stack is in the conversion and evaluation of expressions in Polish Notation—Infix, Prefix and Postfix. This chapter will also highlight the relation of stacks with recursion.

INTRODUCTION TO STACKS

The linear data structures—linked lists and arrays—allow us to insert and delete elements at any place in the list, at the beginning, at the end, or in the middle. In computer science we may come across situations, where insertion or deletion is required only at one end, either at the beginning or end of the list. The suitable data structures to fulfil such requirements are **stacks** and **queues.**

A **stack** is a linear structure in which addition or deletion of elements takes place at the same end. This end is often called the **top** of stack. For example, a stack of plates, a stack of coins, etc. As the items in this type of data structure can be added or removed from the top, it means the last item to be added to the stack is the first item to be removed. Therefore stacks are also called Last In First Out (LIFO) lists.

STACK AS AN ABSTRACT DATA TYPE

Stacks can also be defined as Abstract Data Types (ADT). A stack of elements of any particular type is a finite sequence of elements of that type together with the following operations:

1. **Initialize** the stack to be empty.
2. Determine whether stack is **empty** or not.

3. Determine if stack is **full** or not.
4. If stack is not full, then add or insert a new node at one end of the stack called top. This operation is called **push.**
5. If the stack is not empty, then retrieve the node at its top.
6. If the stack is not empty, then delete the node at its top. This is called **pop.**

The above definition of stack produces the concept of stack as an abstract data type. The basic operations that can be performed on stacks are—**push** and **pop**.

REPRESENTATION OF STACKS THROUGH ARRAYS

Stacks can be represented in the memory through arrays. To do this job we maintain a linear array STACK, a pointer variable TOP which contains the location of top element. The variable MAXSTACK gives the maximum number of elements held by the stack. The TOP = 0 or TOP = NULL will indicate that the stack is empty.

Figure 6.1 shows array representation of a stack. The TOP is pointing to 3 which says that stack has three items and as the MAXSTACK=8, there is still space for accommodating five items.

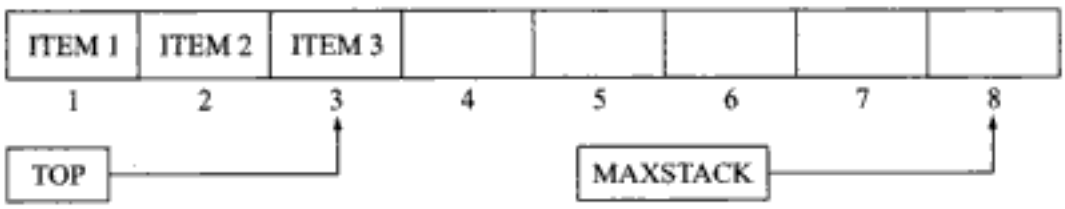


Fig. 6.1 *Array Representation of a Stack*

The operation of adding (pushing) an item onto a stack and the operation of removing (popping) an item can be implemented using the **PUSH** and **POP** functions.

[Note: Sometimes when a new data is to be inserted into a data structure but there is no available space; the situation is called OVERFLOW. On the contrary, if one wants to delete data from the data structure that is empty then the situation is called UNDERFLOW.**]**

/*Program implements stack as an array */

```
# include <stdio.h>
# include <conio.h>

# define ARR 10

struct stack
{
          int a[ARR] ;
          int top ;
} ;
```

```
void init_stack(struct stack *) ;
void push(struct stack *, int item) ;
int pop(struct stack *) ;

void main( )
{
     struct stack s ;
     int i ;

     clrscr( ) ;

     init_stack(&s) ;

     push(&s, 8) ;
     push(&s, 20) ;
     push(&s, -4) ;
     push(&s, 15) ;
     push(&s, 18) ;
     push(&s, 12) ;
     push(&s, 16) ;
     push(&s, 25) ;
     push(&s, 0) ;
     push(&s, 10) ;
     push(&s,  5) ;

     i = pop(&s) ;
     printf("\n\nItem popped: %d", i) ;

     i = pop(&s) ;
     printf("\nItem popped: %d", i) ;

     i = pop(&s) ;
     printf("\nItem popped: %d", i) ;

     i = pop(&s) ;
     printf("\nItem popped: %d", i) ;

     i = pop(&s) ;
     printf("\nItem popped: %d", i) ;

     getch( ) ;
}
```

```
/* Initializes the stack */
void init_stack(struct stack *s)
{
    s -> top = -1 ;
}

/* Adds an element to the stack */
void push(struct stack *s, int item)
{
    if(s -> top == ARR - 1)
    {
          printf("\nStack is full.") ;
          return ;
    }
    s -> top++ ;
    s -> a[s ->top] = item ;
}

/* Removes an element from the stack */
int pop(struct stack *s)
{
    int data ;
    if(s -> top == -1)
    {
          printf("\nStack is empty.") ;
          return NULL ;
    }
    data = s -> a[s -> top] ;
    s -> top-- ;
    return data ;
}
```

A structure stack has been defined as follows:

```
struct stack
{
    int a[ARR];
    int top;
}
```

The **push()** and **pop()** functions are used to insert or delete elements from the top of the stack. The elements are actually stored in the array. The variable **top** is an index into that array. The insertions and deletions are done with the value of top in the array. To begin with, the stack is empty, the top is set to a value –1, by initializing the stack with init_stack function.

In our case, as the array holds 10 elements, therefore, stack would be considered full if the value of top becomes 9 and if in such a case an element is added, the message '**Stack is full**' is given to the user.

The **push()** function in main() has been called to add 11 elements to the stack. The value of top would become 9 after adding tenth element, as a result of which, the eleventh element would not get added to the stack.

The pictorial depiction of pushing elements in stack is given in Fig. 6.2.

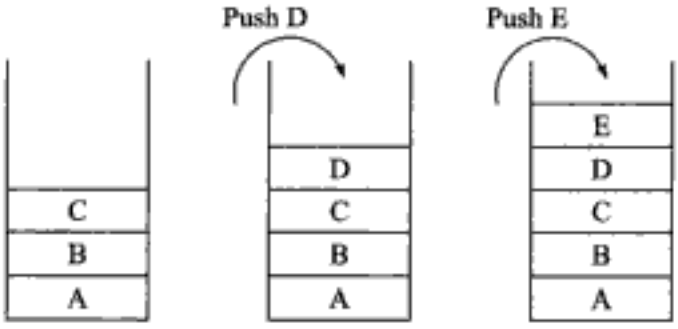


Fig. 6.2 *Pushing Elements in Stack*

Figure 6.3 gives the pictorial depiction of popping elements from stack.

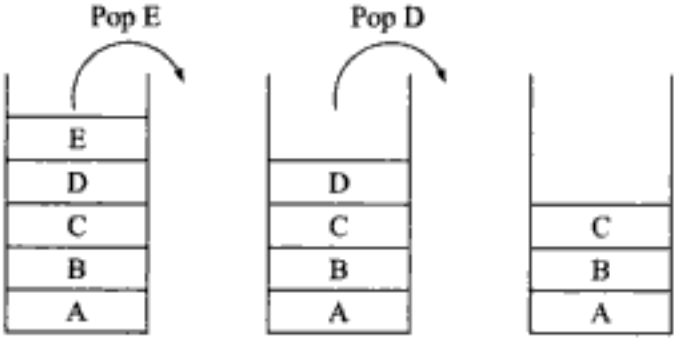


Fig. 6.3 *Popping Elements from Stack*

REPRESENTATION OF STACKS THROUGH LINKED LISTS

The stack can also be implemented using linked lists. The array representation of stack suffers from the drawbacks of the array's size, that cannot be increased or decreased once it is declared. Therefore, either much of the space is wasted, if not used, or, there is shortage of space if needed.

The stack as linked list is represented as a single linked list. Each node in the list contains data and a pointer to the next node. The structure defined to represent stack is as follows:

```
struct node
{
    int data;
    node *next;
};
```

The pictorial depiction of stack in linked list is given in Fig. 6.4

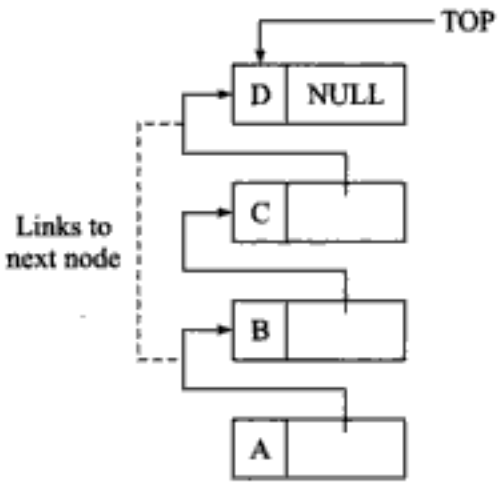


Fig. 6.4 *Stack as a Linked List*

/* Program implements linked list as a stack. */

```
# include <stdio.h>
# include <conio.h>
# include <alloc.h>

/* Structure containing data part and link part */
struct node
{
     int data ;
     struct node *link ;
} ;
void push(struct node **, int) ;
int pop(struct node **) ;
void delstack(struct node **) ;

void main( )
{
     struct node *s = NULL ;
     int i ;

     clrscr( ) ;

     push(&s, 10) ;
     push(&s, 19) ;
     push(&s, 1) ;
     push(&s, 20) ;
```

```
    push(&s, 32) ;
    push(&s, 15) ;

    i = pop(&s) ;
    printf("\nItem popped: %d", i) ;

    i = pop(&s) ;
     printf("\nItem popped: %d", i) ;

    i = pop(&s) ;
    printf("\nItem popped: %d", i) ;

    delstack(&s) ;


    getch( ) ;
}

/* Adds a new node to the stack as linked list */
void push(struct node **top, int item)
{
    struct node *temp ;
    temp = (struct node *) malloc(sizeof(struct node)) ;

    if(temp == NULL)
           printf("\nStack is full.") ;

    temp -> data = item ;
    temp -> link = *top ;
    *top = temp ;
}

/* Pops an element from the stack */
int pop(struct node **top)
{
    struct node *temp ;
    int item ;

    if(*top == NULL)
{
           printf("\nStack is empty.") ;
           return NULL ;
```

```
}

            temp = *top ;
            item = temp -> data ;
            *top = (*top) -> link ;

            free(temp) ;
            return item ;
}

/* Deallocates memory */
void delstack(struct node **top)
{
     struct node *temp ;

     if(*top == NULL)
            return ;

     while(*top != NULL)
     {
            temp = *top ;
            *top = (*top) -> link ;
            free (temp) ;
     }
}
```

In the program given above, a structure node has been designed. The variable **s** is used to point to the structure node. To make the stack empty NULL is assigned to **s**. The **push()** creates a new node dynamically.

After creating the node the pointer **s** should point to the newly created item of the list. Hence, the address of this new node is assigned to **s** using the pointer **top**.

To remove the node from the stack **pop()** function has been used. If the stack is not empty the topmost item gets removed.

The pictorial depiction of pushing into stack using linked list is given in Fig. 6.5.

Figure 6.6 gives the pictorial depiction for pop operation from the stack using linked list.

APPLICATIONS OF STACKS

Stacks are frequently used in evaluation of arithmetic expressions.

Reversing a List

A simple example of stack application is reversal of a given list. We can accomplish this task by pushing each character onto the stack as it is read. When the line is finished, characters are then popped off the stack—they come off in reverse order as shown in Fig. 6.7.

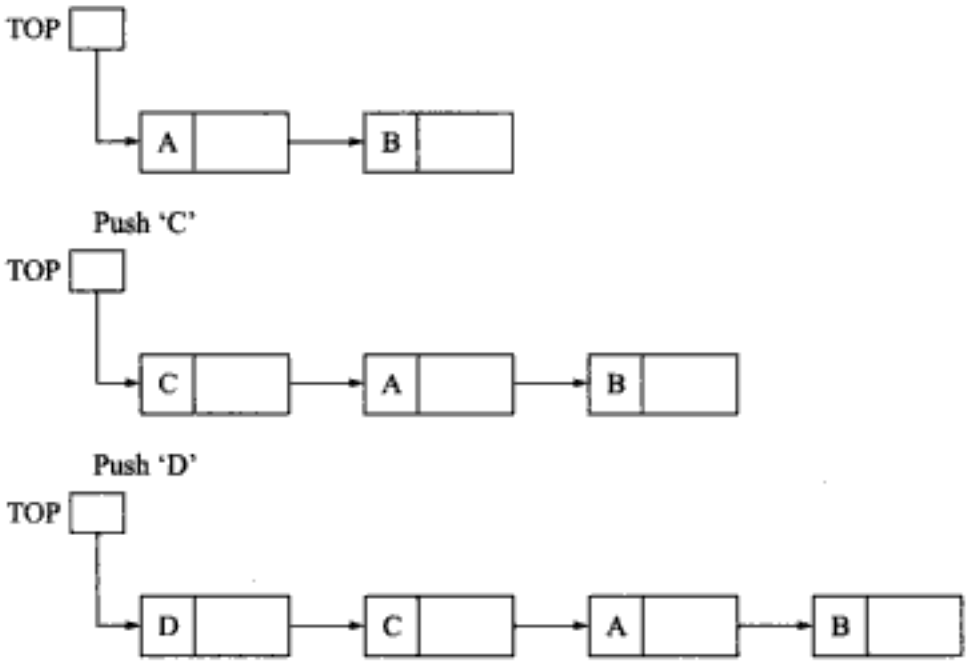


Fig. 6.5 *Pushing an Element into Stack using Linked List*

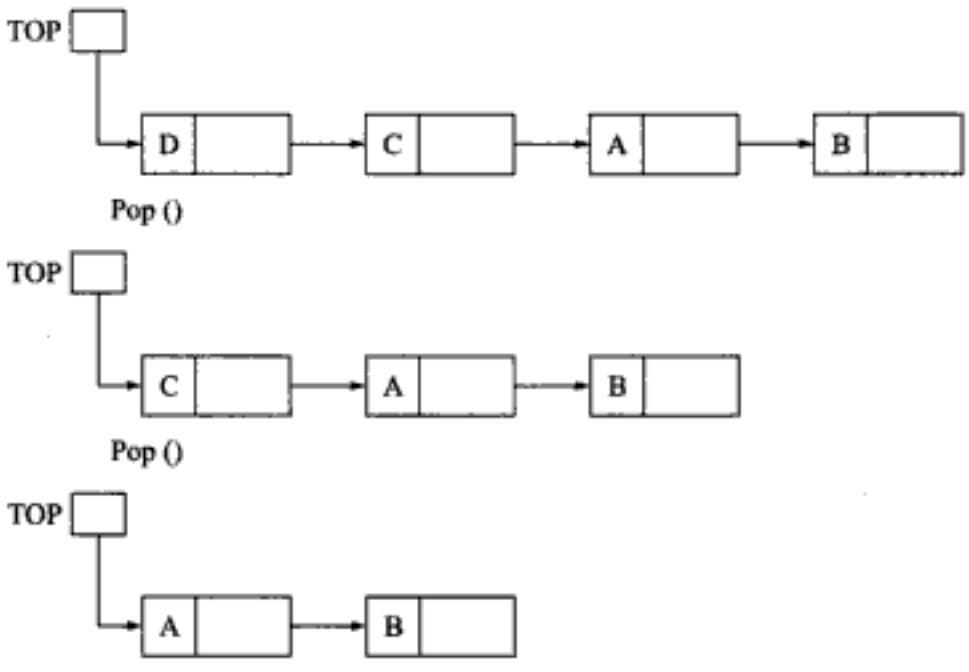


Fig. 6.6 *Popping an Element from the Stack using Linked List*

Polish Notations

The process of writing the operators of an expression either before their operands or after them is called the Polish Notation. This notation was introduced by Jan Lukasiewicz. The main property of Polish Notation is that the order in which operations are to be performed is ascertained by the position of the operators and operands in the expression.

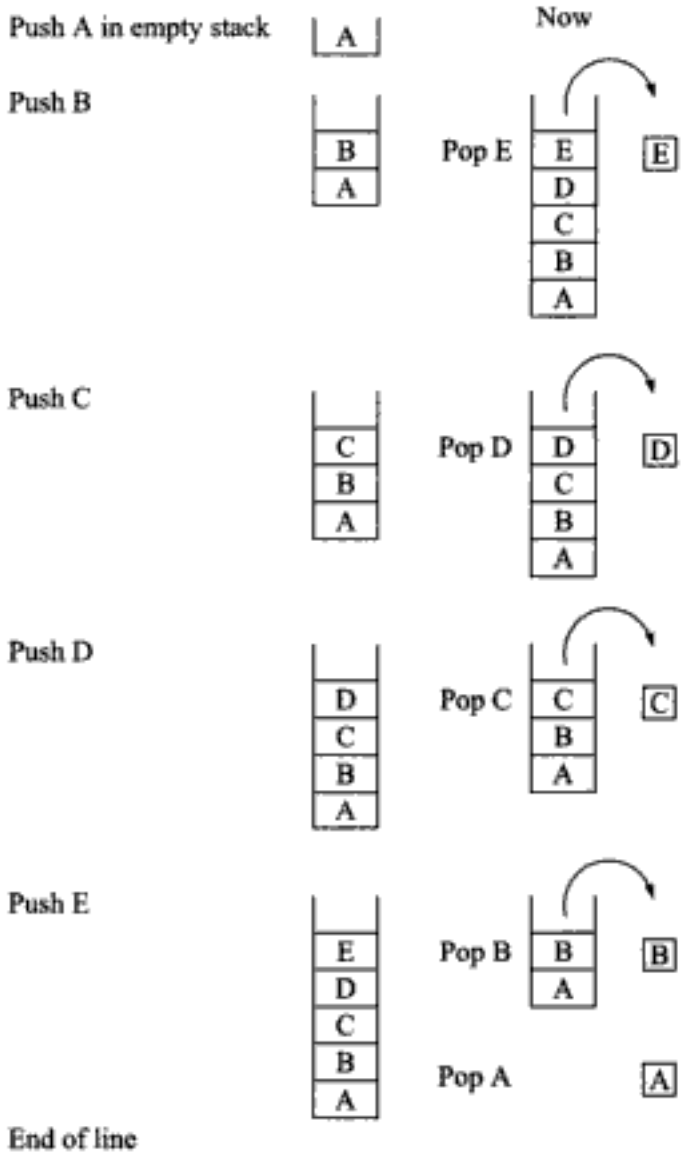


Fig. 6.7 *Reversing a List*

The computer system can understand and work only on binary paradigm, it assumes that an arithmetic operation can take place between two operands only. For example, A+B, C×D, D/A, etc. Usually an arithmetic expression may consist of more than one operator and two operands, for example, (A+B)×C(D/(J+D)). These complex arithmetic expressions can be converted into polish strings using stacks which can then be executed in two operands and an operator form.

The notation refers to these complex arithmetic expressions in three forms:

- If the operator symbols are placed before its operands, then the expression is in **prefix notation**.
- If the operator symbols are placed after its operands, then the expression is in **postfix notation**.
- If the operator symbols are placed between the operands then the expression is in **infix notation**.

For example,

Table 6.1 Notations

Infix Notation	*Prefix Notation*	*Postfix Notation*
A + B	+ AB	AB +
(A – C) * B	* – ACB	AC – B *
A + (B * C)	+ A * BC	ABC * +
(A + B)/(C – D)	/ + AB – CD	AB + CD – /
(A + (B * C))/(C – (D * B))	/ + A * BC – C * DB	ABC * + CDB * –/

Conversion of Infix to Postfix Expression

While evaluating an infix expression, there is an evaluation order according to which the operations are executed:

- Brackets or Parentheses
- Exponentiation
- Multiplication or Division
- Addition or Subtraction

The operators with the same priority (e.g. * and /) are evaluated from left to right.
The steps to convert infix expression to postfix expression are as follows:

(i) The actual evaluation is determined by inserting braces.
(ii) Convert the expression in the innermost braces into postfix notation by putting the operator after the operands.
(iii) Repeat the above step (ii) until the entire expression is converted into postfix notation.

Algorithm to Convert Infix Expression to Postfix Expression

The algorithm transforms the infix expression **A** into equivalent postfix form **B**. The algorithm uses stack to temporarily hold operators and left parentheses. The postfix expression **B** will be constructed left to right using operands from **A** and operators which are removed from STACK. We begin by pushing left parenthesis onto STACK and adding a right parenthesis at the end of **A**. The algorithm is finished when STACK is empty.

The algorithm is as follows:

1. Push left parenthesis "(" onto STACK and add right parenthesis ")" to the end of A.
2. Scan A from left to right and repeat steps 3 to 6 for each element of A until the stack is empty.
3. If an operand is encountered, add it to B.
4. If a left parenthesis is encountered push it onto the stack.
5. If an operator is encountered then
   a. Repeatedly pop from the STACK and add to B each operator (on the top of stack) which has the same precedence as or higher precendence than operator.
   b. Add operator to STACK.

6. If a right parenthesis is encountered, then:
   a) Repeatedly pop from the STACK and add to B each operator (on the top of STACK) until a left parenthesis is encountered.
   b) Remove the left parenthesis. (Do not add left parenthesis to B)
7. Exit

For example, consider the following arithmetic infix expression.

```
A: A + ( B * C – ( D / E ↑ F ) * G ) * H )
   1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
A:A+(B*C–(D/E↑F)*G)*H)
```

The elements of A have now been labelled from left to right for easy reference.
Table 6.2 shows the status of STACK and of the Postfix string **B** as each element of **A** is scanned.

1. Each element is simply added to B and does not change STACK.
2. The subtraction operator (–) in row 7 sends * from STACK to P before it (–) is pushed onto the STACK.
3. The right parenthesis in row 14 sends ↑ and then / from STACK to B, and then removes the left parenthesis from the STACK.
4. The right parenthesis in row 20 sends * then + from STACK to B and then removes the left parenthesis from the top of STACK.
   After step 20 is executed, the stack is empty

Table 6.2 *Evaluation of an Expression from Infix to Postfix using Stack*

Symbol Scanned		*Stack*	*Expression*
1.	A	(	A
2.	+	(+	A
3.	(	(+ (	A
4.	B	(+ (	A B
5.	*	(+ (*	A B
6.	C	(+ (*	A BC
7.	–	(+ (–	A B C *
8.	(	(+ (– (	A B C *
9.	D	(+ (– (	A B C * D
10.	/	(+ (– (/	A B C * D
11.	E	(+ (– (/	A B C * D E
12.	↑	(+ (– (/ ↑	A B C * D E
13.	F	(+ (– (/ ↑	A B C * D E F
14.	)	(+ (–	A B C * D E F ↑ /
15.	*	(+ (– *	A B C * D E F ↑ /
16.	G	(+ (– *	A B C * D E F ↑ / G
17.	)	(+	A B C * D E F ↑ / G * –
18.	*	(+ *	A B C * D E F ↑ / G * –
19.	H	(+ *	A B C * D E F ↑ / G * – H
20.	)	–	A B C * D E F ↑ / G * – H * +

/*Program to convert infix & postfix expression*/

```
#include<stdio.h>
#include<conio.h>
#include<string.h>
#include<ctype.h>
#include<stdlib.h>
#include<graphics.h>
#include<alloc.h>

//MACROS

#define ENTER '\0'
#define BLANK ' '
#define TAB '\t'
#define MAXLENGTH 64
#define empty (-1)
#define operator (-10)
#define operand (-20)
#define leftparen (-30)
#define rightparen (-40)

/* SYMBOL PRECEDENCE */

#define leftparenprec 0
#define addprec 1
#define subprec 1
#define multprec 2
#define divprec 2
#define remprec 2
#define none 999

void read_input(void);
void infix_to_postfix(void);
void write_output(void);
void push(char symbol);
char pop(void);
int get_type(char );
int get_prec(char );
int white_space(char );
void full_stack();
void empty_stack();
```

```
char infix[MAXLENGTH+1],stack[MAXLENGTH],postfix[MAXLENGTH+1];
static int top;
static char *symbols="()+-%*/";

void main()
{
     int gdriver=DETECT,gmode;
     char choice;
     initgraph( &gdriver, &gmode,"w:\\software\\bgi");
     normvideo();
     printf("\n\n\t\t\t <<<<<<<< NOTATIONS >>>>>>>>\n\n");
     do
     {
            top=empty;
           //clrscr();
            read_input();
            infix_to_postfix();
            write_output();
            printf("\n\n Do You Wish To Continue [y/n] : ");
            choice=getch();
            printf("\n");
     }while(choice=='Y' || choice=='y');
     getch();
     }
     void infix_to_postfix(void)
     {
       int i,p,len,type,precedence;
       char next;
       i=p=0;
       len=strlen(infix);
       while(i<len)
       {
            if(!white_space(infix[i]))
            {
                  type=get_type(infix[i]);
                  switch(type)
                  {
                  case leftparen:
                         push(infix[i]);
                         break;
                  case rightparen:
                         while((next=pop())!='(')
```

```
                                postfix[p++]=next;
                        break;
                case operand:
                        postfix[p++]=infix[i];
                        break;
                case operator:
                        precedence=get_prec(infix[i]);
                        while(top>empty && precedence<=get_prec(stack[top]))
                        postfix[p++]=pop();
                        push(infix[i]);
                        break;
                }
        }
        i++;
    }
    while(top>empty)
        postfix[p++]=pop();
    postfix[p]=ENTER;
}

int get_type(char symbol)
{
    switch(symbol)
    {
        case '(' : return(leftparen);
        case ')' : return(rightparen);
        case '+':
        case '-':
        case '%':
        case '*':
        case '/':
                return(operator);
        default : return(operand);
    }
}

int get_prec(char symbol)
{
        switch(symbol)
        {
    case '+' : return(addprec);
    case '-' : return(subprec);
```

```
        case '*' : return(multprec);
        case '/' : return(divprec);
        case '%' : return(remprec);
        case '(' : return(leftparenprec);
        default : return(none);
        }
}

void push(char symbol)
{
        if(top>MAXLENGTH)
                full_stack();
        else
                stack[++top]=symbol;
}

char pop(void)
{
        if(top<=empty)
                {
                        printf("\n Sorry, Stack Is Empty !\n");
                        exit(2);
                }
        else
                return(stack[top--]);
}

void full_stack(void)
{
        printf("\n Sorry, Stack Is Full !\n");
        exit(1);
}

void empty_stack(void)
{
        printf("\n Sorry, Stack Is Empty !\n");
        exit(2);
}
//READ

void read_input(void)
{
```

```
    printf("\n Enter The Infix Upto %d Characters Only : ",MAXLENGTH);
    gets(infix);
}

//WRITE

void write_output(void)
{
    printf("\n Infix : %s",infix);
    printf("\n Postfix : %s",postfix);
}
//WHITE SPACE

int white_space(char symbol)
{
    return(symbol==BLANK || symbol==TAB || symbol==ENTER);
}
```

Evaluation of Postfix Expression

In an infix expression, it is difficult for the machine to keep track of precedence of operators. On the other hand, a postfix expression itself determines the precendence of operators (as the placement of operators in a postfix expression itself determines the precedence of operators). It is easier for a machine to execute a postfix expression than an infix expression.

As postfix expression is without parentheses and can be evaluated as two operands and operator at a time, this becomes easier for the compiler and the computer to handle. Evaluation rule of a postfix expression states:

"While reading the expression from left to right, push the element into the stack if it is an operand; pop the two operands from the stack, if the element is an operator (except NOT operator). In case it is NOT an operator, pop one operand from the stack and then evaluate it (two operators and an operand). Push back the results of the evaluation. Repeat it till the end of stack".

Algorithm for Evaluation of Postfix Expression

/* Reading the expression takes place from **left to right** */

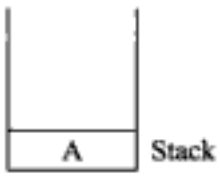
1. Read the next element /* first element for first time */
2. If element is operand then:
   i. Push the element in the stack
3. If element is operator then
   i. Pop two operands from the stack /* POP one operand in case of NOT operator */
   ii. Evaluate the expression formed by the two operands and the operator.
   iii. Push the results of the expression in the stack end.
4. If no more elements then

i. POP the result
else
goto step 1

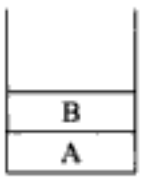
For example, evaluate the postfix expression

AB + C × D / if A = 2, C = 4, D = 5, starting from left to right.

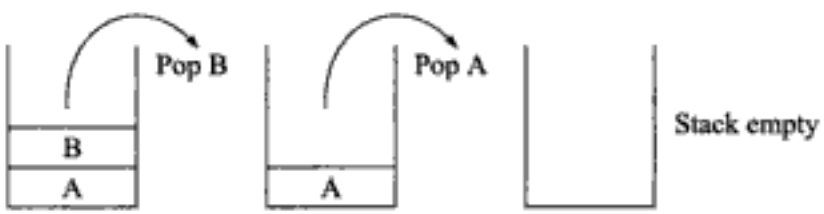
1. First element is operand A, push A into the stack.



2. Second element is also operand B, push B also into the stack.

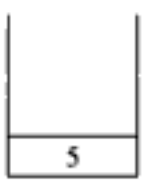


3. The third element '+' is an operator, pop 2 elements from the stack, i.e. A and B and evaluate the expression.

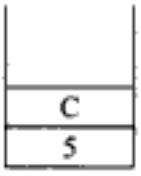


Evaluating = A + B
= 2 + 3
Result = 5

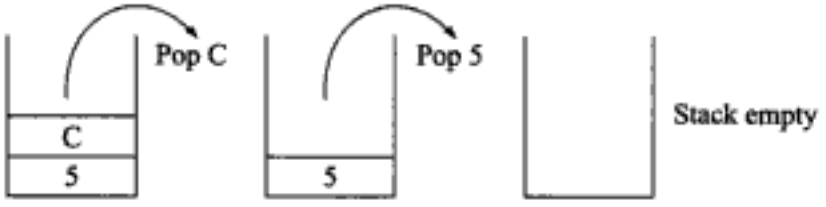
4. Push the result, i.e. 5 into the stack.



5. Next element C is operand; push it into the stack.



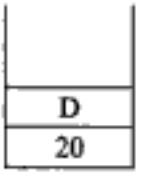
6. Next * is an operator, pop 2 operands from the stack, i.e. 5 and C and evaluate



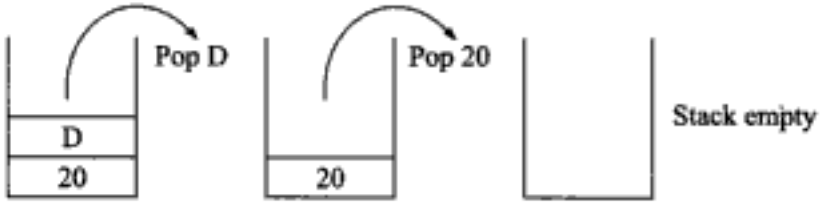
Evaluating $= 5 \times C$
$= 5 \times 4$
$= 20$

7. Push the result 20 into the stack.

8. Next 'D' is operand; push it into the stack.



9. Next '/' is an operator.



10. Pop 2 operands and evaluate.
    Evaluating $= 20/D = 20/5 = 4$.
11. End of expression, thus, the result is 4.

/* Program to evaluate postfix expression */

```
#include<math.h>
```

```
#include<string.h>
#include<stdio.h>
#include<conio.h>

#define MAX 30
#define OPERAND 10
#define OPERATOR 20

typedef struct prexp
{
int top;
int stack[MAX];
}stck;

void init(stck *);
void push(stck *,int);
int pop(stck *);
void eval(stck *,char,int,int);
void main()
{
 char pos[MAX];
 int num1,num2,item,l,i,pr;
 stck stk;

 fflush(stdin);
 clrscr();

 init(&stk);
 printf("Enter the postfix expression : ");
 gets(pos);

           for(i=0;pos[i]!='\0';i++)
           {

                 if(pos[i]==' '||pos[i]=='\t')
                 continue;

                  switch(gettype(pos[i]))
                 {
                   case OPERAND: item=pos[i]-'0';
                    push(&stk,item);
                    break;
```

```
                case OPERATOR:num1=pop(&stk);
                 num2=pop(&stk);
                 eval(&stk,pos[i].num2,num1);
                }
}

 printf("%d",stk.stack[0]);
 getch();
}

void init(stck *st)
{
  st->top=-1;
}

void push(stck *st,int num)
{
  st->top++;
  st->stack[st->top]=num;
}

int pop(stck *st)
{
  int num;
  num=st->stack[st->top];
  st->top--;
  return num;
}

void eval(stck *st,char op,int num1,int num2)
{
  int res;
       switch(op)
       {
        case '+':res=num1+num2;
          break;
 case '-':res=num1-num2;
  break;
 case '*':res=num1*num2;
   break;
 case '/':res=num1/num2;
   break;
```

```
        case '%':res=num1%num2;
          break;
        case '$':res=pow(num1,num2);
          break;
       }
  push(st,res);
}

int gettype(char c)
{
            switch(c)
            {
            case '+':
            case '-':
            case '*':
            case '/':
            case '$':
            case '%':return OPERATOR;
            default:return OPERAND;
            }
}
```

Converting an Infix into Prefix Expression

Algorithm to Convert Infix to Prefix Form

Suppose A is an arithmetic expression written in infix form. The algorithm finds equivalent prefix expression B.

1. Push ')' onto STACK, and add '(' to the end of A.
2. Scan A from right to left and repeat steps 3 to 6 for each element of A until the STACK is empty.
3. If an operand is encountered add it to B.
4. If a right parenthesis is encountered, push it onto stack.
5. If an operator is encountered then:
   a. Repeatedly pop from STACK and add to B each operator (on the top of STACK) which has same or higher precedence than the operator.
   b. Add operator to STACK.
6. If a left parenthesis is encountered then
   a. Repeatedly pop from the STACK and add to B (each operator on top of stack until a left parenthesis is encountered.
   b. Remove the left parenthesis.
7. Exit

   For example, A $ B * C – D + E / F / (G + G)

Table 6.3 *Evaluation of an Expression from Infix to Prefix using Stack*

Symbol Scanned	*Stack*	*Prefix Expression*
)	)	
G	)	G
+	) +	G
G	) +	GG
(	Empty	+ GG
/	/	+ GG
F	/	F + GG
/	//	F + GG
E	//	EF + GG
+	+	// EF + GG
D	+	D // EF + GG
–	+ –	D // EF + GG
C	+ –	CD // EF + GG
*	+ – *	CD // EF + GG
B	+ – *	BCD // EF + GG
$	+ – * $	BCD // EF + GG
A	+ – * $	ABCD // EF + GG
	Empty	+ – * $ ABCD // EF + GG

/* Program to convert infix expression to prefix expression */

```
#include<stdio.h>
#include<conio.h>

#define MAX 30
#define OPERAND 10
#define OPERATOR 20
#define LEFTPARA 30
#define RIGHTPARA 40

typedef struct prestk
{
            int top;
            char stack[MAX];
}stack;

void init(stack *);
void push(stack *,char);
char pop(stack *);
int getprec(char);
int gettype(char);
void main()
```

```
{
    stack stk;
    char inf[MAX],ch,pre[MAX];
    int l,i,k=0,pr;
    fflush(stdin);
    clrscr();

    init(&stk);
    printf(" Enter the infix expression   :");
    gets(inf);
    l=strlen(inf);
    for(i=l-1;i>=0;i--)
    {
                switch(gettype(inf[i]))
                {
                        case OPERAND:pre[k++]=inf[i];
                        break;
                        case  OPERATOR:pr=getprec(inf[i]);
                        while(pr<getprec(stk.stack[stk.top])&&stk.top!=-1)
                        pre[k++]=pop(&stk);
                        push(&stk,inf[i]);
                        break;
                        case RIGHTPARA:push(&stk,inf[i]);
                        break;
                        case LEFTPARA:while((ch=pop(&stk))!=')')
                        pre[k++]=ch;
                }
          }

    while(stk.top!=-1)
    pre[k++]=pop(&stk);
    pre[k]='\0';
    strrev(pre);
    puts(pre);
    getch();
}

void init(stack *st)
{
    st->top=-1;
}

void push(stack *st,char c)
```

```
{
     st->top++;
     st->stack[st->top]=c;
}

char pop(stack *st)
{
     char c;
     c=st->stack[st->top];
     st->top--;
     return c;
}

int getprec(char c)
{
            switch(c)
            {
             case ')':return 0;
             case '+':
             case '-':return 1;
             case '*':
             case '/':
             case '%':return 2;
             case '$':return 3;
            }
}

int gettype(char c)
{
            switch(c)
            {
            case '+':
            case '-':
            case '*':
            case '/':
            case '$':
            case '%':return OPERATOR;
            case '(':return LEFTPARA;
            case ')':return RIGHTPARA;
            default:return OPERAND;
            }
}
```

Evaluation of Prefix Expression

Algorithm for Evaluation of Prefix Expression

/* Reading the expression takes place from **right to left** */

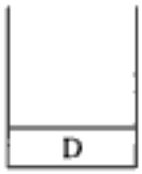
1. Read the next element
2. If element is operand then
   i. Push the element in the stack
3. If element is operator then
   i. Pop two operands from the stack
   ii. Evaluate the expression formed by two operands and the operator
   iii. Push the results of the expression in the stack end
4. If no more elements then
   i. Pop the result
   else
   goto step 1.

For example, evaluate the prefix expression:

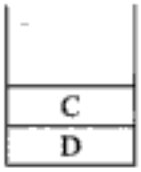
+ A * B + CD if A=2 B=3 C=4 D=5

Starting from right to left: +A * B + CD

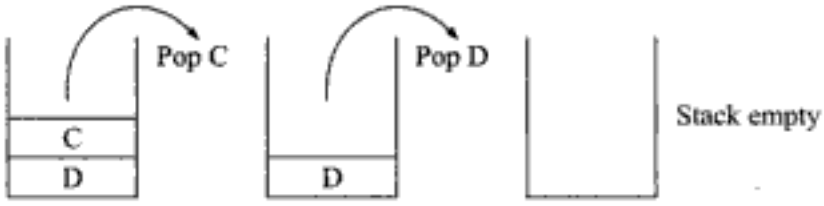
1. First element is operand D, push D into the stack.



2. Second element is also C, push C also into the stack.

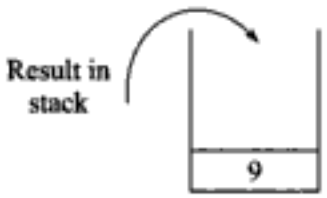


3. The third element '+' is an operator, pop 2 elements from the stack, i.e. C and D and evaluate the expression.

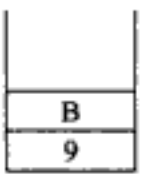


Evaluating = C + D
= 5 + 4
= 9

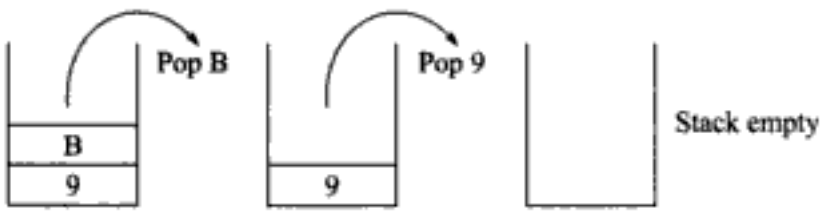
4. Push the result, i.e. 9 into the stack



5. Next element B is operand; push it into the stack.

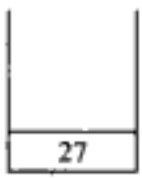


6. Next * is an operator, pop 2 operands from the stack, i.e. B and 9 and evaluate.

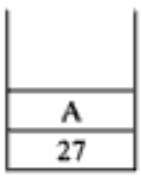


Evaluating = B * 9
= 3 * 9
= 27

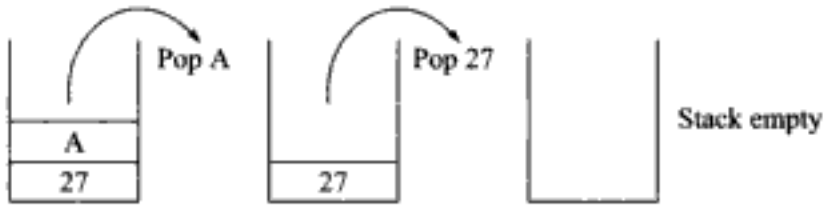
7. Push the result 27 into the stack.



8. Next A is an operand, push it into stack.



9. Next + is an operator, pop 2 elements and evaluate.



Evaluating = A + 27
= 2 + 27
= 29

10. End of expression, thus, the result is 29.

/* Program to evaluate prefix expression */

```
#include<math.h>
#include<string.h>
#include<stdio.h>
#include<conio.h>

#define MAX 30
#define OPERAND 10
#define OPERATOR 20

typedef struct prexp
{
int top;
int stack[MAX];
}stck;

void init(stck *);
void push(stck *,int);
int pop(stck *);
void eval(stck *,char,int,int);

void main()
{
char pre[MAX];
int num1,num2,item,l,i,pr;
stck stk;

fflush(stdin);
clrscr();

init(&stk);
```

```
printf(" ENTER THE PREFIX EXPRESSION ");
gets(pre);
l=strlen(pre);

     for(i=l-1;i>=0;i--)
{

            if(pre[i]==' '||pre[i]=='\t')
            continue;

            switch(gettype(pre[i]))
            {
            case OPERAND: item=pre[i]-'0';
             push(&stk,item);
             break;
            case OPERATOR:num1=pop(&stk);
             num2=pop(&stk);
             eval(&stk,pre[i],num1,num2);
            }
     }
printf("%d",stk.stack[0]);
getch();
}

void init(stck *st)
{
     st->top=-1;
}

void push(stck *st,int num)
{
     st->top++;
     st->stack[st->top]=num;
}

int pop(stck *st)
{
     int num;
     num=st->stack[st->top];
     st->top--;
     return num;
}
```

```
void eval(stck *st,char op,int num1,int num2)
{
     int res;
            switch(op)
            {
            case '+':res=num1+num2;
             break;
            case '-':res=num1-num2;
             break;
            case '*':res=num1*num2;
             break;
            case '/':res=num1/num2;
             break;
            case '%':res=num1%num2;
             break;
            case '$':res=pow(num1,num2);
             break;
            }
     push(st,res);
}

int gettype(char c)
{
            switch(c)
            {
            case '+':
            case '-':
            case '*':
            case '/':
            case '$':
            case '%':return OPERATOR;
            default:return OPERAND;
            }
}
```

STACKS AND RECURSION

Suppose a procedure contains either a call statement to itself or a call statement to a second procedure that may eventually result in a call statement back to the original procedure. Then such a procedure is called a **recursive procedure**. For example, the problem of factorial can be solved using a recursive procedure.

Recursion may be useful in developing algorithms for specific problems. The stacks may be used to implement recursive procedures.

Before going to stack implementation of recursive procedures, let us discuss about subprograms. A subprogram can contain both parameters and local variables. The parameters are the variables which receive values from objects in the calling program, called arguments and which transmit values back to the calling program. The subprogram, besides the parameters and local variables, also keeps track of return address in the calling program. This return address is essential, since control must be transferred back to its proper place in the calling program. Once the subprogram is finished executing and control is transferred back to its calling program, the values of local variables and return address are no longer needed.

Suppose the subprogram is a recursive program. Then each level of execution of the subprogram may contain different values for parameters and local variables and for the return address. Now, if the recursive subprogram call itself, then these current values must be saved, since they will be used again when program is reactivated.

The translation of recursive procedure into a non-recursive procedure using stack is as follows:

1. Declare a stack that will hold the active records consisting of all local variables, parameters called by value and labels to specify where the function is recursively called (if it calls itself from several places).

   The non-recursive function for a recursive function starts with an initialization block which initializes the stack to NULL. The stack and the stack top pointer are defined as global variables.
2. To enable each recursive call to start at the beginning of the original function its first executable statement after the stack initialization block is associated with a label.

   Now, inside the function the following steps should be considered, where it is recursively called, while working with stack.
3. Push all local variables and parameters called by value into the stack.
4. Push an integer 'i' into the stack if this is the i^{th} place from where the function is called recursively.
5. Set the formal parameters called by value to the values given in the new call to the recursive function.
6. Replace the call to recursive function with a 'goto' statement which is the first statement after the stack initialization block. A label is associated to this statement in step 2.
7. Make a new statement label L (if this is the i^{th} place where the function is called recursively) and attach the label to the first statement after the call to the same function (so that a return can be made to this label).

   At the end of recursive function or whenever the function returns to the calling program, the following steps should be performed:
8. If the stack is empty, then the recursion has finished; make a normal return.
9. Otherwise, pop the stack to restore the values of all local variables and parameters called by value.
10. Pop the integer 'i' from the stack and use this to go to the statement labelled L.

Summary

- A stack is a linear structure implemented in LIFO (Last In First Out) manner where insertions and deletions take place at the same end.
- An insertion in a stack is called pushing and deletion from a stack is called popping.

- When a stack, implemented as an array, is full and no new element can be accommodated, it is called an OVERFLOW.
- When a stack is empty and an attempt is made to delete an element from the STACK, it is called UNDERFLOW.
- The main application of stack can be implementation of Polish notation which refers to a notation in which operator symbol is placed either before its operands (prefix notation) or after its operands (postfix notation). The usual form, in which operator is placed in between the operands, is called infix notation.
- The other application of stack can be reversing a list and avoiding recursions in various programs.

Review Exercise

Multiple Choice Questions

1. A data structure in which elements are added and removed only at one end is known as
   a. queue
   b. stack
   c. array
   d. None of these
2. Stack is
   a. Static data structure
   b. Dynamic data structure
   c. In built data structure
   d. None of these
3. Underflow is a condition where you
   a. Insert a new node when there is no free space for it
   b. Delete a non-existent node in the list
   c. Delete a node in empty list
   d. None of the above
4. Get the value of most recently inserted node and delete the node
   a. POP
   b. PUSH
   c. EMPTY
   d. None of the above

Fill in the Blanks

1. A stack may be represented by a _____ linked list. (linear / non-linear)
2. Attempting to create a new node in empty _____ in any data structure, results in overflow. (freespace / nospace)
3. Push operation in stack may result in _____ (overflow / underflow)
4. If TOP points to the top of stack, then TOP is _____ (increased / decreased)

5. If A [n] denotes a stack where bottom of the stack is denoted by A[1] then stack overflow condition occurs when an element is _____ with TOP equal to n. (pushed / popped)
6. A programming language that supports nested procedure call is implemented using _____. (stacks / queues)

State whether True or False

1. Push operation in stack is performed at the rear end.
2. PUSH operation in stack may result in underflow.
3. A stack may be represented by a linear linked list.
4. For a stack, arbitrary amount of memory can be allocated.

Descriptive Questions

1. Consider the following stack, where STACK is allocated N = 6 memory cells.
   STACK : AAA, DDD, EEE, FFF, GGG ----------
   Describe the stack as the following operations take place:
   a. PUSH (STACK, KKK)
   b. POP (STACK, ITEM)
   c. PUSH (STACK, LLL)
   d. PUSH (STACK, SSS)
   e. POP (STACK, ITEM)
   f. PUSH (STACK, TTT)

   [Note Consider the overflow condition.**]**
2. Write an algorithm which, upon user's choice, either pushes or pops an element from the stack implemented as an array (The elements are not shifted after a push or pop).
3. Write a program to convert an infix arithmetic expression into a postfix arithmetic expression. The algorithm for your program should use the following expression:

   $$Q : (A–B) * (C/D) + E$$

   Show in tabular form the changing status of stack.
4. Convert the expression (A+B) / (C–D) into postfix expression and then evaluate it for A = 10, B = 20, C = 15, D = 5. Display the stack status after each operation.
5. Write non-recursive versions of the exponential algorithm Power (real x, positive integer n) where n is a positive integer. The function x^n is defined as the product of n copies of x

   $$\underbrace{x.x.x.x}_{n \text{ time}}$$

   a. using stack.
   b. without using stack.

CHAPTER

7 Queues

Key Features

- Introduction
- Queue as an Abstract Data Type
- Representation of Queues
- Circular Queues
- Double Ended Queues—Deques
- Priority Queues
- Application of Queues

Queue is a linear data structure used to represent a linear list. It allows insertion of an element to be made at one end and deletion of an element to be performed at the other end.

This chapter starts by giving an introduction to the basic concepts of queues. It further discusses various kinds of queues which are often used to simulate real world situations.

INTRODUCTION

Queue is a linear list of elements in which deletion of an element can take place only at one end, called the **front** and insertion can take place only at the other end, called the **rear**. The terms '**front**' and '**rear**' are used while describing queues in a linked list.

The first element in a queue will be the first one to be removed from the list. Therefore, queues are also called FIFO (First In First Out) lists. The common examples for queues can be a queue of people waiting to purchase tickets, where the first person in the queue is the first one to be served.

Many examples of queues can be noticed within a computer system—there may be queues of tasks waiting for the line printer, for access to disk storage or even, in time sharing system, for use of the CPU. Within a single program there may be multiple requests to be kept in queue, or one task may create other tasks which must be executed in turn by keeping them in a queue.

QUEUE AS AN ABSTRACT DATA TYPE

The definition of an abstract data type clearly states that for a data structure to be abstract, it should have the following two characteristics—Firstly, there should be a particular way in which components are related to each other and secondly, a statement of the operations that can be performed on elements of the abstract data type should be specified.

Thus, a queue, as an abstract data type, can be defined as follows:

A queue of elements of type A is a finite sequence of elements of A together with the following operations:

1. Initialize a queue to be empty.
2. Determine if a queue is empty or not.
3. Determine if a queue is full or not.
4. Insert a new element after the last element in a queue, if it is not full.
5. Retrieve the first element of a queue, if it is not empty.
6. Delete the first element in a queue, if it is not empty.

REPRESENTATION OF QUEUES

Queues, being the linear data structure, can be represented by using both arrays and linked lists.

Representation of a Queue as an Array

Array is a data structure that stores a fixed number of elements. One of the major limitations of an array is that its size should be fixed prior to using it. But the size of the Queue keeps on changing as the elements are either removed from the front end or added at the rear end. One of the solutions to this problem would be to declare an array with a maximum size. Figure 7.1 shows the representation of a queue as an array:

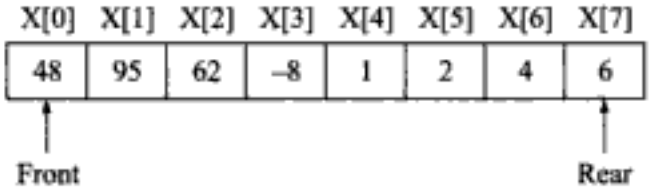


Fig. 7.1 *Representation of a Queue as an Array*

/* Program that implements queue as an array. */

```
#include <stdio.h>
#include <conio.h>
#define MAX 10
void insertque(int *, int, int *, int *) ;
int deleteque(int *, int *, int *) ;
void main( )
{
  int a[MAX] ;
  int front = -1, rear = -1, i ;
  clrscr( ) ;
  insertque(a, 20, &front, &rear) ;
  insertque(a, 5,  &front, &rear) ;
  insertque(a, 9,  &front, &rear) ;
  insertque(a, 7,  &front, &rear) ;
  insertque(a, 22, &front, &rear) ;
```

```
    insertque(a, 12, &front, &rear) ;
    insertque(a, 15, &front, &rear) ;
    insertque(a, 18, &front, &rear) ;
    insertque(a, 10, &front, &rear) ;
    insertque(a, 30, &front, &rear) ;
    insertque(a, 35, &front, &rear) ;
    i = deleteque(a, &front, &rear) ;
    printf("\nItem deleted: %d", i) ;
    i = deleteque(a, &front, &rear) ;
    printf("\nItem deleted: %d", i) ;
    i = deleteque(a, &front, &rear) ;
    printf("\nItem deleted: %d", i) ;
    getch( ) ;
}
/* Adds an element to the queue */
void insertque(int *a, int item, int *pfront, int *prear)
{
    if(*prear == MAX - 1)
    {
        printf("\nQueue is full.") ;
        return ;
    }
    (*prear)++ ;
    a[*prear] = item ;
    if(*pfront == -1)
        *pfront = 0 ;
}
/* Removes an element from the queue */
int deleteque(int *a, int *pfront, int *prear)
{
    int data ;
    if(*pfront == -1)
    {
        printf("\nQueue is Empty.") ;
        return NULL ;
    }
    data = a[*pfront] ;
    a[*pfront] = 0 ;
    if(*pfront == *prear)
        *pfront = *prear = -1 ;
    else
        (*pfront)++ ;
```

```
    return  data ;
}
```

In the program we have taken an array **a** to maintain the queue. The two variables **front** and **rear** have been declared to point to both the ends of the queue.

The two functions **insertque()** and **deleteque()** are used to perform the insertion and deletion operations.

Before adding the elements to the queue, a condition is checked to find whether insertion is possible or not. The array begins with 0 index therefore, the maximum number of elements that can be stored are **MAX-1**. If these number of elements are already present in the queue then the queue is reported to be full. If the elements are added then the rear is incremented using the pointer **prear** and new item is stored in the array.

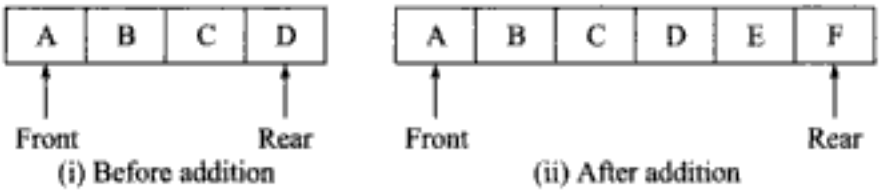


Fig. 7.2 *Adding Elements*

The **front** and **rear** variables are initially set to –1, which denotes that the queue is empty. If the item being added is the first element, (i.e. if the variable **front** has a value –1) then as the item is added, the queue **front** is set to 0 indicating that the queue is now no longer empty.

Now, the function **deleteque()** deletes an element from the queue. The function first checks if there are any elements for deletion. If not, the queue is said to be empty otherwise an element is deleted from a[*pfront].

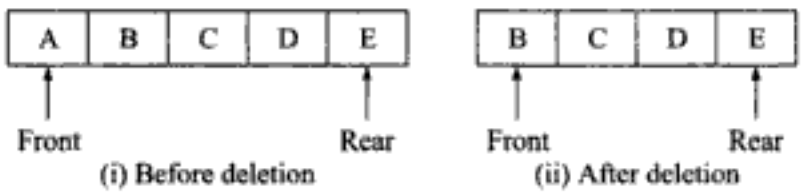


Fig. 7.3 *Deleting Elements*

Say, we have added 5 elements to the queue. The value of **rear** would be 4 and the value of **front** would be 0. If we go on deleting elements from the queue and when the fifth element is deleted, the queue would be empty. To ensure that an 'empty queue' message appears when another attempt is made to delete, the **front** and **rear** are reset to –1.

Representation of a Queue as a Linked List

Queue can also be represented using a linked list. The linked list representation of a queue does not have any restrictions on the number of elements it can hold. The elements in a linked list are allocated dynamically, hence it can grow as long as there is sufficient memory available for dynamic allocation.

/* Program that implements queue as a linked list. */

```
#include <stdio.h>
#include <conio.h>
struct node
{
  int data ;
  struct node *next ;
} ;

struct queue
{
  struct node *front ;
  struct node *rear ;
} ;
void initqueue(struct queue *) ;
void addqueue(struct queue *, int) ;
int delqueue(struct queue *) ;
void deallqueue(struct queue *) ;
void main( )
{
  struct queue a ;
  int i ;
  clrscr( ) ;
  initqueue(&a) ;
  addqueue(&a, 15) ;
  addqueue(&a, 5) ;
  addqueue(&a, 16) ;
  addqueue(&a, 20) ;
  addqueue(&a, 12) ;
  addqueue(&a, 18) ;
  addqueue(&a, 25) ;
  i = delqueue(&a) ;
  printf("\nItem removed: %d", i) ;
  i = delqueue(&a) ;
  printf("\nItem removed: %d", i) ;
  i = delqueue(&a) ;
  printf("\nItem removed: %d", i) ;
  delqueue(&a) ;
  getch( ) ;
}
/* Initialises data member */
void initqueue(struct queue *q)
```

```
{
  q -> front = q -> rear = NULL ;
}
/* Adds an element to the queue */
void addqueue(struct queue *q, int item)
{
  struct node *temp ;
  temp = (struct node *) malloc(sizeof(struct node)) ;
  if(temp == NULL)
       printf("\nQueue is full.") ;
  temp -> data = item ;
  temp -> next = NULL ;
  if(q -> front == NULL)
  {
       q -> rear = q -> front = temp ;
       return ;
  }
  q -> rear -> next = temp ;
  q -> rear = q -> rear -> next ;
}
/* Removes an element from the queue */
int delqueue(struct queue * q)
{
  struct node *temp ;
  int item ;
  if(q -> front == NULL)
  {
       printf("\nQueue is empty.") ;
       return NULL ;
  }
  item = q -> front  -> data ;
  temp = q -> front ;
  q -> front = q -> front -> next ;
  free(temp) ;
  return item ;
}
/* Deallocates memory */
void deallqueue(struct queue *q)
{
  struct node *temp ;
  if(q -> front == NULL)
       return ;
```

```
    while(q -> front != NULL)
    {
        temp = q -> front ;
        q -> front = q -> front -> next ;
        free(temp) ;
    }
}
```

The program above has a structure **queue** which contains two data members—**front** and **rear**, both are pointers to the structure **node**. The queue is empty in the beginning therefore, both **front** and **rear** are set to **NULL**.

The **addqueue()** function adds a new element at the rear end of the list. If the element added is the first element, then both **front** and **rear** are made to point to the new node. If the element added is not the first element, then only **rear** is made to point to the new node and **front** continues to point to the first node in the list.

The **delqueue()** function removes an element which is at the front end of the list. Removal of an element from the list actually deletes the node to which **front** is pointing. After deletion of a node, **front** is made to point to the next node that comes in the list and **rear** points to the last node.

To deallocate the memory for the existing nodes the function **deallqueue()** is called before **main()** comes to an end.

CIRCULAR QUEUES

Circular queues are the queues implemented in circular form rather than in a straight line. Circular queues overcome the problem of unutilised space in linear queue implemented as an array. In the array implementation there is a possibility that the queue is reported full even though slots of the queue are empty (since **rear** has reached the end of array).

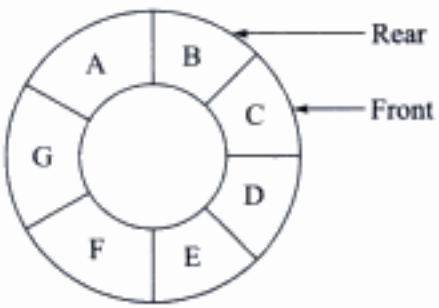


Fig. 7.4 *Circular Queue*

Suppose an array **x** of **n** elements is used to implement a circular queue. If we go on adding elements to the queue we may reach **x[n–1]**. We cannot add any more elements to the queue since the end of the array has been reached. Instead of reporting the queue is full, if some elements in the queue have been deleted then there might be empty slots at the beginning of the queue. In such case these slots would be filled by new elements added to the queue. In short, just because we have reached the end of the array, the queue would not be reported as full. The queue would be reported full only when all the slots in the array are occupied.

/* Program that implements circular queue as an array. */

```
#include <stdio.h>
#include <conio.h>
#define MAX 10
void insertque(int *, int, int *, int *) ;
int deleteque(int *, int *, int *) ;
void display(int *) ;
```

```
void main( )
{
  int a[MAX] ;
  int i, front, rear ;
  clrscr( ) ;
  /* Initialise data member */
  front = rear = -1 ;
  for(i = 0 ; i < MAX ; i++)
       a[i] = 0 ;
  insertque(a, 10, &front, &rear) ;
  insertque(a, 25, &front, &rear) ;
  insertque(a, 11, &front, &rear) ;
  insertque(a, 8, &front, &rear) ;
  insertque(a, 20, &front, &rear) ;
  printf("\nElements in the circular queue: ") ;
  display(a) ;
  i = deleteque(a, &front, &rear) ;
  printf("Item deleted: %d", i) ;
  i = deleteque(a, &front, &rear) ;
  printf("\nItem deleted: %d", i) ;
  printf("\nElements in the circular queue after deletion: ") ;
  display(a) ;
  insertque(a, 2, &front, &rear) ;
  insertque(a, 14, &front, &rear) ;
  insertque(a, 12, &front, &rear) ;
  insertque(a, 5, &front, &rear) ;
  insertque(a, 0, &front, &rear) ;
  printf("Elements in the circular queue after addition: ") ;
  display(a) ;
  insertque(a, 30, &front, &rear) ;
  printf("Elements in the circular queue after addition: ") ;
  display(a) ;
  getch( ) ;
}
/* Adds an element to the queue */
void insertque(int *a, int item, int *pfront, int *prear)
{
  if((*prear == MAX - 1 && *pfront == 0 ) || ( *prear + 1 == *pfront))
  {
       printf("\nQueue is full.") ;
       return ;
  }
```

```
  if(*prear == MAX - 1)
       *prear = 0 ;
  else
       (*prear)++ ;
  a[*prear] = item ;
  if(*pfront == -1)
       *pfront = 0 ;
}
/* Removes an element from the queue */
int deleteque(int *a, int *pfront, int *prear)
{
  int data ;
  if(*pfront == -1)
  {
       printf("\nQueue is empty.") ;
       return NULL ;
  }
  data = a[*pfront] ;
  a[*pfront] = 0 ;
  if(*pfront == *prear)
  {
       *pfront = -1 ;
       *prear = -1 ;
  }
  else
  {
       if(*pfront == MAX - 1)
              *pfront = 0 ;
       else
              (*pfront)++ ;
  }
  return data ;
}
/* Displays element in a queue */
void display(int * arr)
{
  int i ;
  printf("\n") ;
  for(i = 0 ; i < MAX ; i++)
       printf("%d\t", arr[i]) ;
  printf("\n") ;
}
```

In the program given above, the array **a** is used to store elements of the circular queue. The two functions—**insertque()** and **deleteque()**—are used to add and remove the elements from the queue. The function **display()** displays the existing elements of the queue. Initially, the values of **front** and **rear** are set to –1, which shows the queue is empty.

The **main()** function calls the **insertque()** function 5 times to insert elements in the circular queue. The conditions that are checked before inserting the elements are outlined below:

- If the **front** and **rear** are in adjacent locations (i.e. **rear** following **front**) the message 'Queue is Full' is displayed.
- If the value of **front** is –1 then it denotes that the queue is empty and that the element to be added would be the first element in the queue. The values of **front** and **rear** in such a case are set to 0 and new element gets placed at 0^{th} position.
- Some of the positions at the front end of the array might be empty. This happens if we have deleted some elements from the queue, when the value of **rear** is MAX–1 and the value of **front** is greater than 0. In such a case value of **rear** is set to 0 and the element to be added is added to this position.
- The element is added at the **rear** position in case the value of **front** is either equal to or greater than 0 and the value of **rear** is less than MAX–1.

Thus, if we add 5 elements, the value of **front** and **rear** becomes 0 and 4 respectively. The function **display()** displays the elements in the queue.

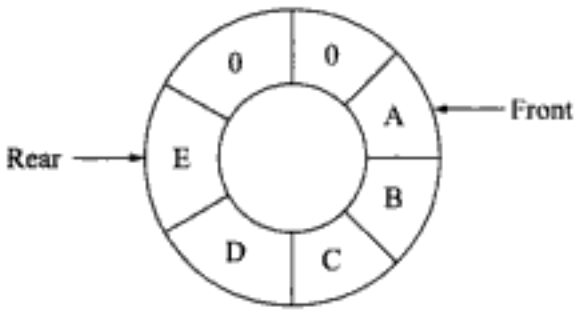


Fig. 7.5 *Circular Queue after Adding 5 Elements*

The **deleteque()** function is called twice to remove 2 elements from the queue. The following conditions are checked before deleting:

- First it is checked whether the queue is empty or not. The value of **front** in our case is 4 hence, the element at the **front** position will be deleted.
- Now, it is checked if the value of **front** is equal to **rear**. If it is, then the element which will be deleted is the only element in the queue. If this element is removed, the queue will become empty and **front** and **rear** are set to –1.

On deleting an element from the queue the value of **front** is set to 0 if it is equal to MAX–1 otherwise **front** is simply incremented by 1.

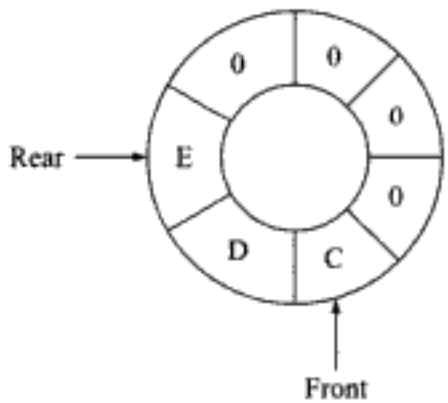


Fig. 7.6 *Circular Queue after Deleting 2 Elements*

DOUBLE ENDED QUEUES—DEQUES

A **deque** is a linear list in which elements can be added or removed at either end but not in the middle. The items can be added or deleted from the **front** or **rear** end, but no changes can be made elsewhere in the list.

There are two variations of a deque—an **input restricted deque** and an **output restricted deque**—which are intermediate between deque and a regular queue. Specifically, an input restricted deque is a deque which allows insertions at only one end of the list, but allows deletions at both ends of the list. An output restricted deque is a deque which allows deletions at only one end of the list but allows insertions at both ends of the list.

The two possibilities that must be considered while inserting or deleting elements into the queue are:

- When an attempt is made to insert an element into a deque which is already full, an **overflow** occurs.
- When an attempt is made to delete an element from a deque which is empty, **underflow** occurs.

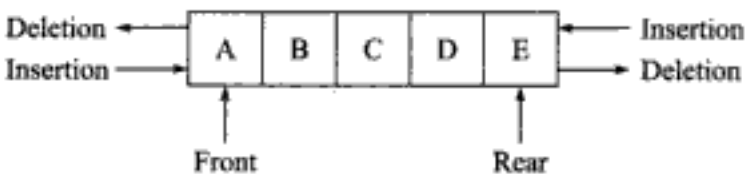


Fig. 7.7 *Representation of a Deque*

Representation of a Deque in an Array

/* Program that implements a deque using an array. */

```
#include <stdio.h>
#include <conio.h>
#define MAX 10
void dequeaddatbeg(int *, int, int *, int *) ;
void dequeaddatend(int *, int, int *, int *) ;
int dequedelatbeg(int *, int *, int *) ;
int dequedelatend(int *, int *, int *) ;
void display(int *) ;
int count(int *) ;
void main( )
{
  int arr[MAX] ;
  int front, rear, i, n ;
  clrscr( ) ;
  /* Initialises data members */
  front = rear = -1 ;
  for(i = 0 ; i < MAX ; i++)
      arr[i] = 0 ;
  dequeaddatend(arr, 17, &front, &rear) ;
```

```
  dequeaddatbeg(arr, 10, &front, &rear) ;
  dequeaddatend(arr, 8, &front, &rear) ;
  dequeaddatbeg(arr, -9, &front, &rear) ;
  dequeaddatend(arr, 13, &front, &rear) ;
  dequeaddatbeg(arr, 28, &front, &rear) ;
  dequeaddatend(arr, 14, &front, &rear) ;
  dequeaddatbeg(arr, 5, &front, &rear) ;
  dequeaddatend(arr, 25, &front, &rear) ;
  dequeaddatbeg(arr, 6, &front, &rear) ;
  dequeaddatend(arr, 21, &front, &rear) ;
  dequeaddatbeg(arr, 11, &front, &rear) ;
  printf("\nElements in a deque: ") ;
  display(arr) ;
  n = count(arr) ;
  printf("\nTotal number of elements in deque: %d", n) ;
  i = dequedelatbeg(arr, &front, &rear) ;
  printf("\nItem extracted: %d", i) ;
  i = dequedelatbeg(arr, &front, &rear) ;
  printf("\nItem extracted:%d", i) ;
  i = dequedelatbeg(arr, &front, &rear) ;
  printf("\nItem extracted:%d", i) ;
  i = dequedelatbeg(arr, &front, &rear) ;
  printf("\nItem extracted: %d", i) ;
  printf("\nElements in a deque after deletion: ") ;
  display(arr) ;
  dequeaddatend(arr, 16, &front, &rear) ;
  dequeaddatend(arr, 7, &front, &rear) ;
  printf("\nElements in a deque after addition: ") ;
  display(arr) ;
  i = dequedelatend(arr, &front, &rear) ;
  printf("\nItem extracted: %d", i) ;
  i = dequedelatend(arr, &front, &rear) ;
  printf("\nItem extracted: %d", i) ;
  printf("\nElements in a deque after deletion: ") ;
  display(arr) ;
  n = count(arr) ;
  printf("\nTotal number of elements in deque: %d", n) ;
  getch( ) ;
}
/* Adds an element at the beginning of a deque */
void dequeaddatbeg(int *arr, int item, int *pfront, int *prear)
{
```

```
  int i, k, c ;
  if(*pfront == 0 && *prear == MAX - 1)
  {
       printf("\nDeque is full.\n") ;
       return ;
  }
  if(*pfront == -1)
  {
       *pfront = *prear = 0 ;
       arr[*pfront] = item ;
       return ;
  }
  if(*prear != MAX - 1)
  {
       c = count(arr) ;
       k = *prear + 1 ;
       for(i = 1 ; i <= c ; i++)
       {
              arr[k] = arr[k - 1] ;
              k-- ;
       }
       arr[k] = item ;
       *pfront = k ;
       (*prear)++ ;
  }
  else
  {
       (*pfront)-- ;
       arr[*pfront] = item ;
  }
}
/* Adds an element at the end of a deque */
void dequeaddatend(int *arr, int item, int *pfront, int *prear)
{
  int i, k ;
  if(*pfront == 0 && *prear == MAX - 1)
  {
       printf("\nDeque is full.\n") ;
       return ;
  }
  if(*pfront == -1)
  {
```

```
        *prear = *pfront = 0 ;
        arr[*prear] = item ;
        return ;
    }
    if(*prear == MAX - 1)
    {
        k = *pfront - 1 ;
        for(i = *pfront - 1 ; i < *prear ; i++)
        {
                k = i ;
                if(k == MAX - 1)
                        arr[k] = 0 ;
                else
                        arr[k] = arr[i + 1] ;
        }
        (*prear)-- ;
        (*pfront)-- ;
    }
    (*prear)++ ;
    arr[*prear] = item ;
}
/* Removes an element from the *pfront end of deque */
int dequedelatbeg(int *arr, int *pfront, int *prear)
{
    int item ;
    if(*pfront == -1)
    {
        printf("\nDeque is empty.\n") ;
        return 0 ;
    }
    item = arr[*pfront] ;
    arr[*pfront] = 0 ;
    if(*pfront == *prear)
        *pfront = *prear = -1 ;
    else
        ( *pfront )++ ;
    return item ;
}
/* Removes an element from the *prear end of the deque */
int dequedelatend(int *arr, int *pfront, int *prear)
{
    int item ;
```

```
    if(*pfront == -1)
    {
        printf("\nDeque is empty.\n") ;
        return 0 ;
    }
    item = arr[*prear] ;
    arr[*prear] = 0 ;
    (*prear)-- ;
    if(*prear == -1)
        *pfront = -1 ;
    return item ;
}
/* Displays elements of a deque */
void display(int *arr)
{
    int i ;
    printf("\n front->") ;
    for(i = 0 ; i < MAX ; i++)
        printf("\t%d", arr[i]) ;
    printf(" <-rear") ;
}
/* Counts the total number of elements in deque */
int count(int *arr)
{
    int c = 0, i ;
    for(i = 0 ; i < MAX ; i++)
    {
        if(arr[i] != 0)
            c++ ;
    }
    return c ;
}
```

In the program given above we have two functions—**dequeaddatbeg()** and **dequeaddatend()**—to add elements at the beginning and at the end of the deque respectively.

The function **dequeaddatend()** checks whether the deque is full or not. A message is displayed if the dequeue is full. The next condition is to check whether the element to be added is the first element. If it is, then the value of **front** and **rear** are set to 0 and the first element is placed in the deque.

Fig. 7.8 *Deque after Addition of First Element*

Now the **dequeaddatbeg()** function is called to add an element in front end of the deque. This function, before adding an element, checks for two conditions, first whether deque is full or not and second whether the element to be added is the first element of the deque.

Considering our deque which contains one element **A**, both the conditions will evaluate to false. So, we add another element **B** at 0^{th} position. As the 0^{th} position is occupied by the previous element, we need to shift the element **A** to the right. But the shifting is possible if there is vacant place available to the right of the element. In other words, this would be possible if the value of **rear** is less than **MAX**. Since we have only one element in the deque it would be shifted to the right and the second element **B** would be added at 0^{th} position.

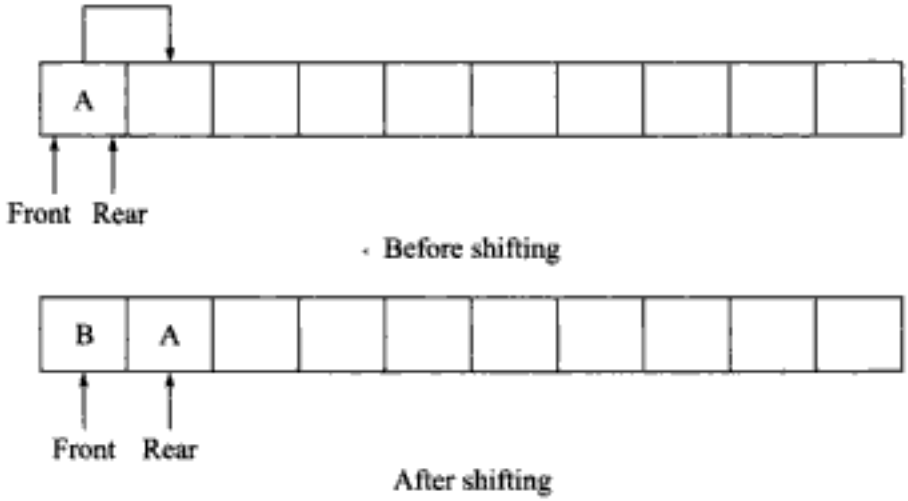


Fig. 7.9 *Adding an Element in Deque*

Now, the value of **front** and **rear** would be 0 and 1 respectively. Similarly, other elements would be added to the deque.

The function **dequedelatbeg()** removes an element from the **front** position. After an element is removed, **front** stores the index of next element in the deque. Hence, the value of **front** is incremented by 1. In the function **dequedelatend()**, on removing element at the end, **rear** should store the index of the element that occupies the position to the left of the element being deleted. Hence, the value of **rear** is decremented by 1.

Think of a situation when the value of rear has reached MAX–1 and the value of front is greater than 0 say, the value of front is 4, after deleting first 4 elements. This would be the stage when deque is not full. But at this stage it would not be possible to add an element at the **rear** end of the deque. To do this, elements would be required to be shifted one position to the left. This situation is handled in the function **dequeaddatend()** by the following code statements:

```
int i, k;
if(*prear == MAX-1)
{
  k=*pfront-1;
  for(i = *pfront-1; i < *prear; i++)
  {
```

```
        k=i;
        if(k==MAX-1)
                arr[k]=0;
        else
                arr[k]=arr[i+1];
    }
    (*prear)- -;
    (*pfront)- -;
}
```

The value of temporary variable **k** is set to **front**-1 if the above code evaluates to true. If the value of front is 4 the value of k would be 3. Using the for loop the elements are shifted one position to the left. The new element is added at the end of deque.

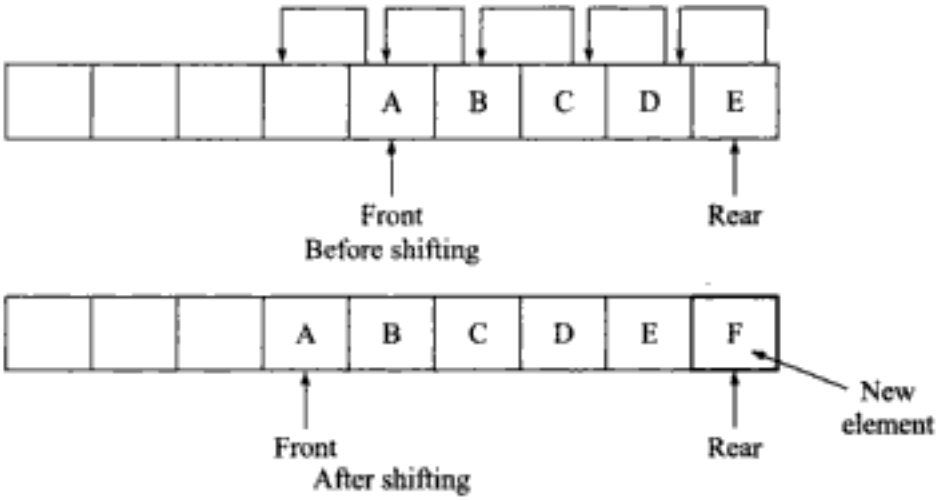


Fig. 7.10 *Adding of an Element at the End of Deque*

Representation of a Deque in a Linked List

/* Program on deque that implements a linked list. */

```
#include <stdio.h>
#include <conio.h>
#include <alloc.h>
struct node
{
   int data ;
   struct node *link ;
} ;
struct dqueue
{
   struct node *front ;
   struct node *rear ;
```

```
} ;
void initdqueue(struct dqueue *) ;
void dequeaddatend(struct dqueue *, int item) ;
void dequeaddatbeg(struct dqueue *, int item) ;
int dequedelatbeg(struct dqueue *) ;
int dequedelatend(struct dqueue *) ;
void display(struct dqueue) ;
int count(struct dqueue) ;
void deletedqueue ( struct dqueue * ) ;
void main( )
{
  struct dqueue dq ;
    int i, n ;
    clrscr( ) ;
    initdqueue(&dq) ;
  dequeaddatend(&dq, 10) ;
  dequeaddatbeg(&dq, 8);
  dequeaddatend(&dq, 12) ;
  dequeaddatbeg(&dq, 2);
  dequeaddatend(&dq, 14) ;
  dequeaddatbeg(&dq, 5) ;
  dequeaddatend(&dq, 19) ;
  dequeaddatbeg(&dq, 25) ;
  display(dq) ;
  n = count(dq) ;
  printf("\nTotal elements: %d", n) ;
  i = dequedelatbeg(&dq);
  printf("\nItem extracted = %d", i) ;
  i = dequedelatbeg(&dq) ;
    printf("\nItem extracted = %d", i) ;
  i = dequedelatbeg(&dq) ;
    printf("\nItem extracted = %d", i) ;
  i = dequedelatend(&dq) ;
    printf("\nItem extracted = %d", i) ;
  display(dq) ;
  n = count(dq) ;
  printf(" \nElements Left: %d", n) ;
    deletedqueue(&dq) ;
    getch( ) ;
}
/* Initializes elements of structure */
void initdqueue(struct dqueue *p)
{
```

```
  p -> front = p -> rear = NULL ;
}
/* Adds item at the end of dqueue */
void dequeaddatend(struct dqueue *p, int item)
{
  struct node *temp ;
  temp = (struct node *)malloc(sizeof(struct node));
  temp -> data = item ;
  temp -> link = NULL ;
  if(p -> front == NULL)
       p -> front = temp ;
  else
       p -> rear -> link = temp ;
  p -> rear = temp ;
}
/* Adds item at beginning of dqueue */
void dequeaddatbeg(struct dqueue *p, int item)
{
  struct node *temp ;
  int *q ;
  temp = (struct node *) malloc(sizeof(struct node));
  temp -> data = item ;
  temp -> link = NULL ;
  if(p -> front == NULL)
       p -> front = p -> rear = temp ;
  else
  {
       temp -> link = p -> front ;
       p -> front = temp ;
  }
}
/* Deletes item from beginning of dqueue */
int dequedelatbeg(struct dqueue *p)
{
  struct node *temp = p -> front ;
  int item ;
  if(temp == NULL)
  {
       printf("\nQueue is empty.") ;
       return 0 ;
  }
  else
  {
```

```
        temp = p -> front ;
        item = temp -> data ;
        p -> front = temp -> link ;
        free(temp) ;
        if(temp == NULL)
                p -> rear = NULL ;
        return(item) ;
    }
}
/* Deletes item from end of dqueue */
int dequedelatend(struct dqueue *p)
{
    struct node *temp , *rleft, *q ;
    int item ;
    temp = p -> front ;
    if(p -> rear == NULL)
    {
        printf("\nQueue is empty.") ;
        return 0 ;
    }
    else
    {
        while(temp != p -> rear)
        {
                rleft = temp ;
                temp = temp -> link ;
        }
        q = p -> rear ;
        item = q -> data ;
        free(q) ;
        p -> rear = rleft ;
        p -> rear -> link = NULL ;
        if(p -> rear == NULL)
                p -> front = NULL ;
        return(item) ;
    }
}
/* Displays the queue */
void display(struct dqueue dq)
{
    struct node *temp = dq.front ;
    printf("\nfront -> ") ;
    while(temp != NULL)
```

```
  {
       if(temp -> link == NULL)
       {
            printf("\t%d", temp -> data) ;
            printf(" <- rear") ;
       }
       else
            printf("\t%d", temp -> data) ;
       temp = temp -> link ;
  }
  printf("\n") ;
}
/* Counts the number of items in dqueue */
int count(struct dqueue dq)
{
  int c = 0 ;
  struct node *temp = dq.front ;

  while(temp != NULL)
  {
       temp = temp -> link ;
       c++ ;
  }
  return c ;
}
/* Deletes the queue */
void deletedqueue(struct dqueue *p)
{
    struct node *temp ;
  if(p -> front == NULL)
       return ;
  while(p -> front != NULL)
  {
       temp = p -> front ;
       p -> front = p -> front -> link ;
       free(temp) ;
  }
}
```

In the program above, for the linked representation of deque, two structures have been maintained:

```
struct node
{
     int data;
```

```
        struct node *next;
    };
```

The above structure maintains the linked list and the other structure **dqueue** contains the structure node type two variables ***front** and ***rear** to maintain the queue.

```
    struct dqueue
    {
        struct node *front;
        struct node *rear;
    };
```

The **initdqueue()** initializes elements of structure. The **dequeaddatbeg()** and the **dequeaddatend()** functions, again do the same work in array—the functions add the elements at the beginning and end of the deque.

Similarly, the functions **dequedelatbeg()** and **dequedelatend()** delete the elements from the beginning and end of the deque.

The linked list representation is probably used when we want to allocate the memory for elements dynamically.

Input-Restricted and Output-Restricted Deques

In an input restricted deque, the insertion of elements is restricted to one end only, but the deletion of elements can be done at both the ends.

/*Input-restricted deque program using an array*/

```
#include <stdio.h>
#include <conio.h>
#include <alloc.h>
#define MAX 10
struct dqueue
{
  int arr[MAX] ;
  int front, rear ;
} ;
void initdqueue(struct dqueue *) ;
void dequeaddatend(struct dqueue *, int item) ;
int dequedelatbeg(struct dqueue *) ;
int dequedelatend(struct dqueue *) ;
void display(struct dqueue) ;
int count(struct dqueue) ;
void main( )
{
  struct dqueue dq ;
    int i, n ;
    clrscr( ) ;
```

```
    initdqueue(&dq) ;
  dequeaddatend(&dq, 1) ;
  dequeaddatend(&dq, 2) ;
  dequeaddatend(&dq, 3) ;
  dequeaddatend(&dq, 4) ;
  dequeaddatend(&dq, 5) ;
  dequeaddatend(&dq, 6) ;
  dequeaddatend(&dq, 7) ;
  dequeaddatend(&dq, 8) ;
  dequeaddatend(&dq, 9) ;
  dequeaddatend(&dq, 10) ;
    display(dq) ;
  n = count(dq) ;
  printf("\nTotal eTements: %d", n) ;
  i = dequedelatbeg(&dq) ;
  printf("\nItem removed = %d", i) ;
  i = dequedelatbeg(&dq) ;
  printf("\nItem removed = %d", i) ;
  i = dequedelatend(&dq) ;
  printf("\nItem removed = %d", i) ;
  i = dequedelatend(&dq) ;
  printf("\nItem removed = %d", i) ;
  n = count(dq) ;
  printf("\nElements Left: %d", n) ;
  display(dq) ;
  dequeaddatend(&dq, 19) ;
  dequeaddatend(&dq, 16) ;
  dequeaddatend(&dq, 11) ;
  dequeaddatend(&dq, 15) ;
  dequeaddatend(&dq, 24) ;
  dequeaddatend(&dq, 25) ;
  display(dq) ;
    getch( ) ;
}
/* Initializes elements of structure */
void initdqueue(struct dqueue *p)
{
    int i ;
  p -> front = p -> rear = -1 ;
  for(i = 0 ; i < MAX ; i++)
      p -> arr[i] = 0 ;
}
```

```
/* Adds item at the end of dqueue */
void dequeaddatend(struct dqueue *p, int item)
{
    int i, k ;
  if(p -> front == 0 && p -> rear == MAX)
  {
     printf("\nQueue is full.\n") ;
     return ;
  }
  if(p -> rear == -1 && p -> front == -1)
  {
      p -> rear = p -> front = 0 ;
      p -> arr[p -> rear] = item ;
      (p -> rear)++ ;
      return ;
  }
  if(p -> rear == MAX)
  {
      for(i = k = p -> front - 1 ; i < p -> rear ; i++)
      {
            k = i ;
            if(k == MAX - 1)
                  p -> arr[k] = 0 ;
            else
                  p -> arr[k] = p -> arr[i + 1] ;
      }
      (p -> rear)-- ;
      (p -> front)-- ;
  }
  p -> arr[p -> rear] = item ;
  (p -> rear)++ ;
}
/* Deletes item from beginning of dqueue */
int dequedelatbeg(struct dqueue *p)
{
    int item ;
  if(p -> front == -1 && p -> rear == -1)
  {
      printf("\nQueue is empty.\n") ;
      return 0 ;
  }
  item = p -> arr[p -> front] ;
```

```
  p -> arr[p -> front] = 0 ;
  (p -> front)++ ;
  if(p -> front == MAX)
       p -> front = -1 ;
  return item ;
}
/* Deletes item from end of dqueue */
int dequedelatend(struct dqueue *p)
{
    int item ;
  if(p -> front == -1 && p -> rear == -1)
  {
       printf("\nQueue is empty.\n") ;
       return 0 ;
  }
  (p -> rear)-- ;
  item = p -> arr[p -> rear] ;
  p -> arr[p -> rear] = 0 ;
  if ( p -> rear == 0 )
       p -> rear = -1 ;
  return item ;
}
/* Displays the queue */
void display(struct dqueue dq)
{
    int i ;
    printf("\n front -> ") ;
  for(i = 0 ; i < MAX ; i++)
       printf("%d\t", dq.arr[i]) ;
  printf(" <- rear") ;
}
/* Counts the number of items in dqueue */
int count(struct dqueue dq)
{
  int c, i ;
  for(i = c = 0 ; i < MAX ; i++)
  {
       if(dq.arr[i] != 0)
            c++ ;
  }
  return c ;
}
```

The program for input-restricted deque would not contain the function **dequeaddatbeg()** as the input-restricted deque restricts the insertion of elements at one end only.

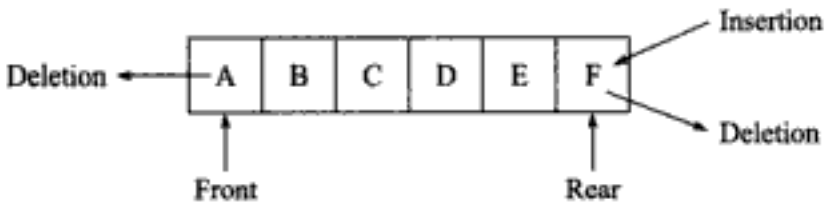


Fig. 7.11 *Representation of an Input-Restricted Deque*

Similarly, the program for output-restricted deque would not contain the function **dequedelatbeg()**.

/*Output-restricted deque using an array*/

```
#include <stdio.h>
#include <conio.h>
#include <alloc.h>
#define MAX 10
struct dqueue
{
  int arr[MAX] ;
  int front, rear ;
} ;
void initdqueue(struct dqueue *) ;
void dequeaddatbeg(struct dqueue *, int) ;
void dequeaddatend(struct dqueue *, int) ;
int dequedelatend(struct dqueue *) ;
void display(struct dqueue) ;
int count(struct dqueue) ;
void main( )
{
  struct dqueue dq ;
    int i, n ;
    clrscr( ) ;
    initdqueue(&dq) ;
  dequeaddatend(&dq, 11) ;
  dequeaddatbeg(&dq, 1) ;
  dequeaddatend(&dq, 17) ;
  dequeaddatbeg(&dq, 10) ;
  dequeaddatend(&dq, 11) ;
  dequeaddatbeg(&dq, 8) ;
  dequeaddatend(&dq, 18) ;
  dequeaddatbeg(&dq, 5) ;
```

```
  dequeaddatend(&dq, 19) ;
  dequeaddatbeg(&dq, 6) ;
  dequeaddatend(&dq, 15) ;
  dequeaddatbeg(&dq, 3) ;
  display(dq) ;
  n = count(dq) ;
  printf("\nTotal elements: %d", n) ;
  i = dequedelatend(&dq) ;
  printf("\nItem removed = %d", i) ;
  i = dequedelatend(&dq) ;
  printf("\nItem removed = %d", i) ;
  i = dequedelatend(&dq) ;
  printf("\nItem removed = %d", i) ;
  i = dequedelatend(&dq) ;
  printf("\nItem removed = %d", i) ;
  n = count(dq) ;
  printf("\nElements Left: %d", n) ;
  display(dq) ;
  dequeaddatbeg(&dq, 4) ;
  dequeaddatbeg(&dq, 3) ;
  dequeaddatbeg(&dq, 2) ;
  dequeaddatbeg(&dq, 1) ;
  display(dq) ;
    getch( ) ;
}
/* Initializes elements of structure */
void initdqueue(struct dqueue *p)
{
    int i ;
  p -> front = p -> rear = -1 ;
  for(i = 0 ; i < MAX ; i++)
      p -> arr[i] = 0 ;
}
/* Adds item at beginning of dqueue */
void dequeaddatbeg(struct dqueue *p, int item)
{
  int c, i, k ;
  if(p -> front == 0 && p -> rear == MAX)
  {
     printf("\nQueue is full.\n") ;
     return ;
  }
  if(p -> front == -1 && p -> rear == -1)
```

```
  {
       p -> front = p -> rear = 0 ;
       p -> arr[p -> front] = item ;
       return ;
  }
  if(p -> rear != MAX)
  {
       c = count(*p) ;
       k = p -> rear ;
       for(i = 1 ; i <= c ; i++)
       {
              p -> arr[k] = p -> arr[k - 1] ;
              k-- ;
       }
       p -> arr[k] = item ;
       p -> front = k ;
       (p -> rear)++ ;
  }
  else
  {
       (p -> front)-- ;
       p -> arr[p -> front] = item ;
  }
}
/* Adds item at the end of dqueue */
void dequeaddatend(struct dqueue *p, int item)
{
  int i, k ;
  if(p -> front == 0 && p -> rear == MAX)
  {
     printf("\nQueue is full.\n") ;
     return ;
  }
  if(p -> rear == -1 && p -> front == -1)
  {
       p -> rear = p -> front = 0 ;
       p -> arr[p -> rear] = item ;
       (p -> rear)++ ;
  return ;
  }
  if(p -> rear == MAX)
  {
```

```
        k = p -> front - 1 ;
        for(i = p -> front - 1 ; i < p -> rear ; i++)
        {
                k = i ;
                if(k == MAX - 1)
                        p -> arr[k] = 0 ;
                else
                        p -> arr[k] = p -> arr[i + 1] ;
        }
        (p -> rear)-- ;
        (p -> front)-- ;
    }
    p -> arr[p -> rear] = item ;
    (p -> rear)++ ;
}
/* Deletes item from end of dqueue */
int dequedelatend(struct dqueue *p)
{
     int item ;
    if(p -> front == -1 && p -> rear == -1)
    {
        printf("\nQueue is empty.\n") ;
        return 0 ;
    }
    (p -> rear)-- ;
    item = p -> arr[p -> rear] ;
    p -> arr[p -> rear] = 0 ;
    if(p -> rear == 0)
        p -> rear = -1 ;
    return item ;
}
/* Displays the queue */
void display(struct dqueue dq)
{
    int i ;
    printf("\n front -> ") ;
    for(i = 0 ; i < MAX ; i++)
        printf("\t%d", dq.arr[i]) ;
    printf(" <- rear ") ;
}
/* Counts the number of items in dqueue */
int count(struct dqueue dq)
{
```

```
    int c, i ;
    for(i = c = 0 ; i < MAX ; i++)
    {
        if(dq.arr[i] != 0)
            c++ ;
    }
    return c ;
}
```

The output-restricted dequeue, restricts the deletion of elements at one end only.

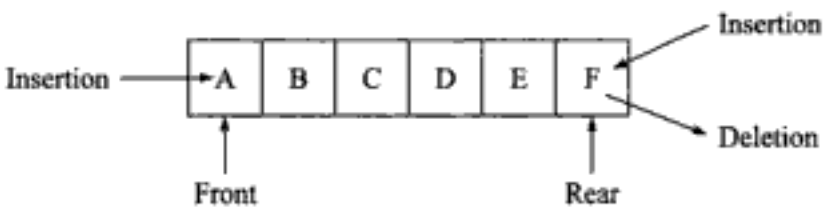


Fig. 7.12 *Representation of Output-restricted Dequeue*

PRIORITY QUEUES

A **priority queue** is a collection of elements where each element is assigned a priority and the order in which elements are deleted and processed is determined from the following rules:

- An element of higher priority is processed before any element of lower priority.
- Two elements with the same priority are processed according to the order in which they are added to the queue.

An example of a priority queue can be a timesharing system: programs of high priority are processed first, and programs with the same priority form a standard queue.

One way to represent a priority queue in memory is by means of one-way list:

- Each node in the list contains three items of information—an information field **INFO**, a priority number **PRNO** and the link **NEXT**.
- Node **A** will precede Node **B** in the list when **A** has higher priority than **B** or when both the nodes have same priority but **A** was added to the list before **B**. This means that the order in one-way list corresponds to the order of priority queue.

Figure 7.13 shows priority queue with 7 elements where element **B** and C have same priority numbers.

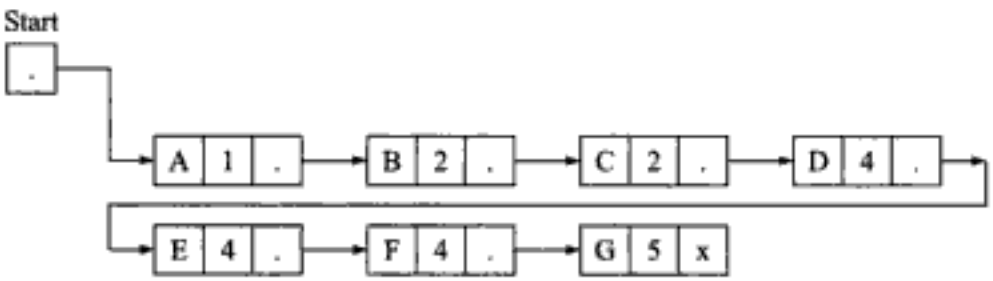


Fig. 7.13 Priority queue

Array Implementation of Priority Queue

Another way to represent priority queue is through arrays.

If an array is used to store the elements of priority queue then insertion is easy but deletion of elements would be difficult. This is because while inserting elements in the priority queue they are not inserted in an order. As a result, deleting an element with the highest priority would require examining the entire array to search for such an element. Moreover, an element in a queue can be deleted from the front end only.

The array element of a priority queue can have the following structure:

```
struct data
{
    int item;
    int priority;
    int order;
};
```

The structure holds the type of data item, priority of the element and the order in which the element has been added.

Another structure **pque** has been defined which contains an array **d** to hold the elements of priority queue. The elements in the array are of struct data type holding information about the job to be processed, priority of the job, and the order in which the elements will be added.

The function **add()** adds the element to the priority queue whereas **delete()** removes an element from the priority queue.

In **main()** function, 5 elements are added using the **for** loop. In **add()** function, the element **dt** gets added to the queue at the **rear** end. Suppose the first element to be added holds data as {A, 4, 1}. After adding this element the value of **front** and **rear** would be 0. The elements in this function are arranged in ascending order of their priorities.

Suppose the second element to be added to queue is {B, 3, 2}. The priority of second element is lower than the element at the 0^{th} position in the queue. Hence, the second element would get placed at 0^{th} position and the element at 0^{th} position would occupy the first position. Figure 7.14 illustrates the same.

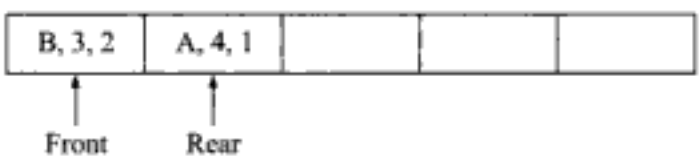


Fig. 7.14 Priority of elements

Now if there are two elements with the same priority, then the elements are arranged according to their order number in which they are entered. Thus, the elements in the priority queue are arranged prioritywise and within the same priority as per the order of entry. Figure 7.15 shows a priority queue with 5 elements.

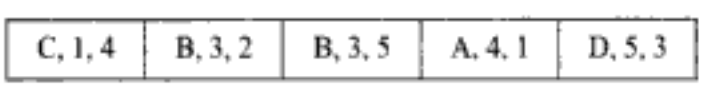


Fig. 7.15 Priority queue with 5 elements

Next, in order to process the information with the highest priority the **remove()** function is called. In this function the element at the **front** is removed.

/* Program that implements a priority queue using an array. */

```
#include <stdio.h>
#include <conio.h>
#define MAX 5
struct data
{
  char job[MAX] ;
  int prno ;
  int ord ;
} ;
struct pque
{
  struct data d[MAX] ;
  int front ;
  int rear ;
} ;
void initpque(struct pque *) ;
void add(struct pque *, struct data) ;
struct data delete(struct pque *) ;
void main( )
{
  struct pque q ;
  struct data dt, temp ;
  int i, j = 0 ;
  clrscr( ) ;
  initpque(&q) ;
  printf("Enter Job description(max 4 chars)and its priority\n") ;
  printf("Lower the priority number, higher the priority\n") ;
  printf("Job Priority\n") ;
  for(i = 0 ; i < MAX ; i++)
  {
       scanf("%s %d", &dt.job, &dt.prno) ;
       dt.ord = j++ ;
       add(&q, dt) ;
  }
  printf("\n") ;
  printf("Process jobs prioritywise\n") ;
  printf("Job\tPriority\n") ;
  for(i = 0 ; i < MAX ; i++)
  {
```

```
        temp  = delete(&q) ;
        printf("%s\t%d\n", temp.job, temp.prno) ;
    }
    printf("\n") ;
    getch( ) ;
}
/* Initialises data members */
void initpque(struct pque *pq)
{
    int i ;
    pq -> front = pq -> rear = -1 ;
    for(i = 0 ; i < MAX ; i++)
    {
        strcpy(pq -> d[i].job, '\0') ;
        pq -> d[i].prno = pq -> d[i].ord = 0 ;
    }
}
/* Adds item to the priority queue */
void add(struct pque *pq, struct data dt)
{
    struct data temp ;
    int i, j ;
    if(pq -> rear == MAX - 1)
    {
        printf("\nQueue is full.") ;
        return ;
    }
    pq -> rear++ ;
    pq -> d[pq -> rear] = dt ;
    if(pq -> front == -1)
        pq -> front = 0 ;
    for(i = pq -> front ; i <= pq -> rear ; i++)
    {
        for(j = i + 1 ; j <= pq -> rear ; j++)
        {
            if(pq -> d[i].prno > pq -> d[j].prno)
            {
                temp = pq -> d[i] ;
                pq -> d[i] = pq -> d[j] ;
                pq -> d[j] = temp ;
            }
```

```
            {
                if(pq -> d[i].prno == pq -> d[j].prno)
                {
                    if(pq -> d[i].ord > pq -> d[j].ord)
                    {
                        temp = pq -> d[i] ;
                        pq -> d[i] = pq -> d[j] ;
                        pq -> d[j] = temp ;
                    }
                }
            }
        }
    }
}
/* Removes item from priority queue */
struct data  delete(struct pque *pq)
{
    struct data  t ;
    strcpy(t.job, '') ;
    t.prno = 0 ;
    t.ord = 0 ;
    if(pq -> front == -1)
    {
        printf("\nQueue is Empty.\n") ;
        return t ;
    }
    t = pq -> d[pq -> front] ;
    pq -> d[pq -> front] = t ;
    if(pq -> front == pq -> rear)
        pq -> front = pq -> rear = -1 ;
    else
        pq -> front++ ;
    return  t ;
}
```

APPLICATION OF QUEUES

Simulation is the use of one system to initiate the behaviour of another system. Simulations are often used when it would be too expensive or dangerous to experiment with real systems. There are physical simulations such as wind tunnels used to experiment with designs for car bodies and flight simulators used to train airline pilots. Mathematical simulations are system of equations used to describe some system and computer simulation uses the steps of a program to initiate the behaviour of system under study.

In computer simulation, the objects being studied are usually represented as data, often as data structures, like lists or arrays whose entries describe the properties of the objects. Actions in the system being studied are represented as operations on the data, and rules describing these actions are translated into computer algorithms. By changing the value of data or by modifying these algorithms, we can observe the changes in the computer simulation, and then we can draw inferences about the actual system.

While one object in a system is involved in some action, other objects and actions will often need to wait. Hence queues are important data structures for use in computer simulations.

Airport Simulation

As an example, let us consider a small but busy airport with only one runway. In each unit of time, one plane can land or take off, so, at any given unit of time, the runway may be idle or a plane may be landing or taking off and there may be several planes waiting either to land or take off. We, therefore, need two queues which are used for landing and takeoff to hold these planes. It is better to keep a plane waiting on land than in the air, so, a small airport allows a plane to take off only if there are no planes waiting to land. Hence, after receiving request from new planes to land and take off, our simulation will first service the head of the queue of the plane waiting to land, and only if the landing queue is empty will it allow a plane to take off. We shall wish to run the simulation through many units of time and therefore we embed the main action of the program in a loop that runs for current time (curtime) from 1 to a variable end time.

The main() functon in simulation program will make use of queues to keep track of all the relevant statistics concerning the problem, such as the number of planes processed, the average time spent waiting, and number of planes (if any) refused service. These details are reflected in the declaration of constants, types and variables to be inserted into the main program. We shall then need to write the subprograms to specify how this information is processed.

The steps for simulation are as follows:

1. Initializing the parameters and printing messages.
2. Accepting a new plane.
3. Handling a full queue.
4. Processing an arriving plane.
5. Processing a departing plane.
6. Marking an idle time unit
7. Finishing the simulation.

Random Numbers

A key step in this simulation is to decide, at each unit, how many new planes are ready to land or take off. Although there are many ways in which these decisions can be made, one of the most interesting and useful is to make random decisions. When the program is run repeatedly with random decisions, the results will differ from run to run, and with sufficient experimentation, the simulation may display a range of behaviour not unlike that of the actual system being studied.

Many computer systems include random number generators and, if one is available on your system, it can be used in place of one developed here.

The idea is to start with one number and apply a series of arithmetic operations that will produce another number with no obvious connection to the first. Hence, the numbers we produce are not truly random, as each one depends in a definite way on its predecessor, and we should more properly speak of **pseudorandom** numbers. If we begin the simulation with the same value each time the program is run then whole sequence of pseudorandom numbers will be exactly the same, so we normally begin by setting the starting point for the pseudorandom integers to some random value, for example, the time of day.

The function time returns the number of seconds elapsed since 00 : 00 : 00 GMT, January 1970. The expression time (NULL) % 10000 produces an integer between 0 and 9999 the number of seconds elapsed modules 10000. This number provides different starting point for the function **Randomize()** each time it is run.

We can then use the standard system function rand for producing each pseudo random number from its predecessors. The function rand produces as its result an integer number between 0 and INT-MAX. For this simulation we wish to obtain an integer giving the number of planes arriving ready to land (or take off) in a given time unit. We can assume that time when plane enters the system is independent of that of any other plane.

The number of planes arriving in one unit of time then follows what is called a **Poisson distribution** in statistics. To calculate the number we need to know **expected value** that is average number of planes, arriving in one unit of time, e.g. if on an average one plane arrives in each four time units, then the expected value is 0.25. Sometimes several planes may arrive in same time unit, but often no planes arrive for a long time, so taking the average over many units gives 0.25.

Program for Airport Simulation

/* Airport simulation */

```
#include <stdio.h>
#include <conio.h>
#include <stdlib.h>
#include <ctype.h>
#include <math.h>
#include <time.h>
#include <limits.h>
#define MAX 3
#define ARRIVE 0
#define DEPART 1
struct plane
{
  int id ;     /*Identification number of airplane*/
  int tm ;     /*Time arrival in queue*/
} ;
struct queue
{
  int count ;
  int front ;
```

```
  int rear ;
  struct plane p[MAX] ;
} ;
void initqueue(struct queue *) ;
void addqueue(struct queue *, struct plane) ;
struct plane delqueue(struct queue *) ;
int size(struct queue) ;
int empty(struct queue) ;
int full(struct queue) ;
void initqueue(struct queue *pq)
{
  pq -> count = 0 ;
  pq -> front = 0 ;
  pq -> rear = -1 ;
}
void addqueue(struct queue *pq, struct plane item)
{
  if(pq -> count >= MAX)
  {
       printf("\nQueue is full.\n") ;
       return ;
  }
  (pq -> count)++ ;
  pq -> rear = (pq -> rear + 1) % MAX ;
  pq -> p[pq -> rear] = item ;
}
struct plane delqueue(struct queue *pq)
{
  struct plane pl ;
  if(pq -> count <= 0)
  {
       printf("\nQueue is empty.\n") ;
       pl.id = 0 ;
       pl.tm = 0 ;
  }
  else
  {
       (pq -> count)-- ;
       pl = pq -> p[pq -> front] ;
       pq -> front = (pq -> front + 1) % MAX ;
  }
  return pl ;
}
```

```
int size(struct queue q)
{
  return q.count ;
}
int empty(struct queue q)
{
  return(q.count <= 0) ;
}
int full(struct queue q)
{
  return(q.count >= MAX) ;
}
struct airport
{
  struct queue landing ;
  struct queue takeoff ;
  struct queue *pl ;
  struct queue *pt ;
  int idletime ;              /*Number of units when runway is idle*/
  int landwait;               /*Total waiting time for planes landed*/
  int takeoffwait ;           /*Total waiting time for take off*/
  int nland;                  /*Number of planes landed*/
  int nplanes;                /*Number of planes processed so far*/
  int nrefuse;                /*Number of planes refused of airport*/
  int ntakeoff ;              /*Number of planes taken off*/
  struct plane pln ;
} ;
void initairport(struct airport *) ;
void start(int *, double *, double *) ;
void newplane(struct airport *, int, int) ;
void refuse(struct airport *, int) ;
void land( struct airport *, struct plane, int) ;
void fly(struct airport *, struct plane, int) ;
void idle(struct airport *, int) ;
void conclude(struct airport *, int) ;
int randomnumber(double) ;
void apaddqueue(struct airport *, char) ;
struct plane apdelqueue(struct airport *, char) ;
int apsize(struct airport, char) ;
int apfull(struct airport, char) ;
int apempty(struct airport, char) ;
void myrandomize( ) ;
void initairport(struct airport *ap)
```

```
{
    initqueue(&(ap-> landing)) ;
    initqueue(&(ap -> takeoff)) ;
  ap -> pl = &(ap -> landing) ;
  ap -> pt = &(ap -> takeoff) ;
  ap -> nplanes = ap -> nland = ap -> ntakeoff = ap -> nrefuse = 0 ;
  ap -> landwait = ap -> takeoffwait = ap -> idletime = 0 ;
}
void start(int *endtime, double *expectarrive, double *expectdepart)
{
  int flag = 0 ;
  char wish :=
  printf("\nProgram that simulates an airport with only one runway.\n") ;
  /*Instruct user*/
  printf("One plane can land or depart in each unit of time.\n") ;
  printf("Up to %d planes can be waiting to land or take off at any time.\n", MAX) ;
  printf("How many units of time will the simulation run?") ;
  /*Input parameter*/
  scanf( "%d", endtime) ;
  myrandomize( ) ;
  do
  {
      printf("\nExpected number of arrivals per unit time? ") ;
      /*Error checking*/
      scanf("%lf", expectarrive) ;
      printf("\nExpected number of departures per unit time? ") ;
      scanf("%lf", expectdepart) ;
      if(*expectarrive < 0.0 || *expectdepart < 0.0)
      {
            printf("These numbers must be nonnegative.\n") ;
            flag = 0 ;
      }
      else
      {
            if(*expectarrive + *expectdepart > 1.0)
            {
                  printf("The airport will become saturated. Read new numbers? ")

              fflush(stdin) ;
                  scanf("%c", &wish) ;
                  if(tolower ( wish ) == 'y')
                        flag = 0 ;
                  else
```

```
                        flag = 1 ;
                }
                else
                        flag = 1 ;
        }
  } while(flag == 0) ;
}
void newplane(struct airport *ap, int curtime, int action)
/*Newplane :make a new record for a plane.update nplanes*/
{
    (ap -> nplanes)++ ;
  ap -> pln.id = ap -> nplanes ;
  ap -> pln.tm = curtime ;
  switch(action)
  {
       case ARRIVE:
              printf("\n") ;
              printf("Plane %d ready to land.\n", ap -> nplanes) ;
              break ;
       case DEPART:
              printf("\nPlane %d ready to take off.\n", ap -> nplanes) ;
              break ;
  }
}
void refuse(struct airport *ap, int action)
/*Refuses: processes the plane when queue is full*/
{
  switch(action)
  {
       case ARRIVE:
              printf("\tplane  %d directed to another airport.\n", ap -> pln.id) ;
              break ;
       case DEPART:
              printf("\tplane %d told to try later.\n", ap -> pln.id) ;
              break ;
  }
  (ap -> nrefuse)++ ;
}

void land(struct airport *ap, struct plane pl, int curtime)
/* Land: process a plane p that is actually landing */
  {
  int wait ;
```

```
  wait = curtime - pl.tm ;
  printf("%d: Plane %d landed ", curtime, pl.id) ;
  printf("in queue %d units \n", wait) ;
  (ap -> nland) ++ ;
  (ap -> landwait) += wait ;
}
void fly(struct airport *ap, struct plane pl, int curtime)
/* Fly: process a plane p that is actually landing*/
{
  int wait ;
  wait = curtime - pl.tm ;
  printf("%d: Plane %d took off ", curtime, pl.id) ;
  printf("in queue %d units \n", wait) ;
  (ap -> ntakeoff)++ ;
  (ap -> takeoffwait) += wait ;
}
void idle(struct airport *ap, int curtime)
/*Idle:updates variables for idle runaway*/
{
  printf("%d: Runway is idle.\n", curtime) ;
  ap -> idletime++ ;
}
void conclude(struct airport *ap, int endtime)
/* Conclude: write out statistics and conclude simulation*/
{
  printf("\tSimulation has concluded after %d units.\n", endtime) ;
  printf("\tTotal number of planes processed: %d\n", ap -> nplanes) ;
  printf("\tNumber of planes landed: %d\n", ap -> nland) ;
  printf("\tNumber of planes taken off: %d\n", ap -> ntakeoff) ;
  printf("\tNumber of planes refused use: %d\n", ap -> nrefuse) ;
  printf("\tNumber left ready to land: %d\n", apsize ( *ap, 'l')) ;
  printf("\tNumber left ready to take off: %d\n", apsize(*ap, 't')) ;
  if(endtime > 0)
       printf("\tPercentage of time runway idle: %lf \n", ((double) ap -> idletime /
endtime) * 100.0) ;
  if(ap -> nland > 0)
       printf("\tAverage wait time to land: %lf \n", ((double) ap -> landwait / ap -
> nland)) ;
  if(ap -> ntakeoff > 0)
       printf("\tAverage wait time to take off: %lf \n", ((double) ap -> takeoffwait
/ ap -> ntakeoff)) ;
}
int randomnumber(double expectedvalue)
```

```
/*Randomnumber: generates a pseudorandom integer according to the Poisson distribution*/
{
  int n = 0 ;     /*Counter of iterations*/
  double em ;
  double x ;      /*Pseudorandom number */
  em = exp(-expectedvalue) ;
  x = rand( ) / (double) INT_MAX ;
  while(x > em)
  {
       n++ ;
       x *= rand( ) / (double) INT_MAX ;
  }
  return n ;
}
void apaddqueue(struct airport *ap, char type)
{
  switch(tolower(type))
  {
       case 'l' :
                addqueue(ap -> pl, ap -> pln) ;
                break ;
       case 't' :
                addqueue(ap -> pt, ap -> pln) ;
                break ;
  }
}
struct plane apdelqueue(struct airport *ap, char type)
{
  struct plane pl ;
  switch(tolower(type))
  {
       case 'l' :
                pl = delqueue(ap -> pl) ;
                break ;
       case 't':
                pl = delqueue(ap -> pl) ;
                break ;
  }
  return pl ;
}
int apsize(struct airport ap, char type)
{
  switch(tolower(type))
```

```
  {
       case 'l':
                return(size(*(ap.pl))) ;
       case 't':
                return(size(*(ap.pt))) ;
  }
  return 0 ;
}
int apfull(struct airport ap, char type)
{
  switch(tolower(type))
  {
       case 'l':
                return(full(*(ap.pl))) ;
       case 't':
                return(full(*(ap.pt))) ;
  }
  return 0 ;
}
int apempty(struct airport ap, char type)
{
  switch(tolower(type))
  {
       case 'l':
                return(empty(*(ap.pl))) ;
       case 't':
                return(empty (*(ap.pt))) ;
  }
  return 0 ;
}
void myrandomize( )
/*Sets starting point for pseudorandam numbers*/
{
  srand((unsigned int) (time(NULL) % 10000)) ;
}
void main( )
{
  struct airport a ;
  int i;
  int pri;              /* Pseudo random integer*/
  int curtime;         /*Current time : one unit = time for take off and landing*/
  int endtime ;    /*Total number of time units to run */
```

```
double expectarrive;            /*Number of planes arriving in one unit*/
double expectdepart ;           /*Number of planes newly ready to take off*/
struct plane temp ;
  clrscr( ) ;
  initairport(&a);
start(&endtime, &expectarrive, &expectdepart) ;
for(curtime = 1 ; curtime <= endtime ; curtime++)
{
     pri = randomnumber(expectarrive) ;
     for(i = 1 ; i <= pri ; i++)          /*Add to landing quue*/
     {
            newplane(&a, curtime, ARRIVE) ;
            if(apfull(a, 'l'))
                    refuse(&a, ARRIVE) ;
            else
                   apaddqueue(&a, 'l') ;
     }
     pri = randomnumber(expectdepart) ;
     for(i = 1 ; i <= pri ; i++)        /*Add to takeoff queue*/
     {
            newplane(&a, curtime, DEPART) ;
            if(apfull(a, 't'))
               refuse(&a, DEPART) ;
            else
               apaddqueue(&a, 't') ;
     }
     if(!(apempty(a, 'l')))  /*Bring plane to land */
     {
            temp = apdelqueue(&a, 'l') ;
            land(&a, temp, curtime) ;
     }
     else
     {
            if(!(apempty(a, 't'))) /*Allow plane to take off */
            {
                   temp = apdelqueue(&a, 't') ;
                   fly(&a, temp, curtime) ;
            }
            else
                   idle(&a, curtime) ;      /*Runway idle*/
     }
}
```

```
  conclude(&a, endtime) ;          /*Finish simulation*/
    getch( ) ;
}
```

Summary

- Queue is a linear data structure that permits insertion of new element at one end and deletion of an element at the other end. Queue is also referred to as first-in-first-out (FIFO) list.
- Circular queues are the queues implemented in circle rather than a straight line.
- Deques are the queues in which elements can be added or removed at either end but not in the middle.
- An input-restricted deque is a deque which allows insertions at only one end but does not allow deletions at both the ends of the list.
- An output-restricted deque is a deque which allows deletions at only one end of the list but allows insertions at both the ends of the list.
- A queue in which it is possible to insert an element or remove an element at any position depending on some priority is called priority queue.
- Queue is a data structure used in many applications like event simulation, job scheduling, etc.

Review Exercise

Multiple Choice Questions

1. "FRONT = REAR" pointer refers to empty
   a. stack
   b. queue
   c. array
   d. None of the above
2. A data structure in which insertion and deletion can take place at both the ends is called
   a. deque
   b. stack
   c. circular queue
   d. None of the above
3. Using arrays, most efficient implementation of queue is on
   a. Linear queue
   b. Priority queue
   c. Circular queue
   d. None of the above
4. Let P be the queue of integers defined as follows:

```
# define MAXQ 500
struct queue
```

```
{
        int  items[MAXQ];
        int  front, rear;
} q;
```

To insert an element in the queue we can use:

a. ++q.items[q.rear] = x;
b. q.items[+q.rear] = x;
c. q.items[q.rear]++ = x;
d. None of the above.

Fill in the Blanks

1. A queue can be defined as a _____. (data type / data structure)
2. The term head of the queue is same as the term _____. (front / rear)
3. FRONT = REAR pointer refers to _____ queue. (empty / full)
4. A/An _____ is a queue in which insertion of an element takes place at both the ends but deletion occurs at one end only. (input-restricted / output restricted)

State whether True or False

1. A queue can be implemented using a circular array with front and rear indices and one position left vacant.
2. Queue is a useful data structure for any simulation application.
3. A priority queue is implemented using an array of stacks.
4. Queues are often referred to as Last In First Out (LIFO) data structure.
5. A deque is a generalization of both a stack and a queue.

Descriptive Questions

1. Show how a sequence of insertions and removals from a queue represented by a linear array can cause overflow to occur upon an attempt to insert an element into an empty queue.
2. How would you implement a queue of stacks? A stack of queues? A queue of queues? Write routines to implement the appropriate operations of each of these data structures.
3. A deque is an ordered set of items from which items may be deleted at either end and into which items may be inserted at either end. Call the two ends of the deque left and right. How can a deque be represented as a C array? Write four routines: remvleft, remvright, insertleft, insertright to remove and insert elements at the left and right ends of the deque such that they detect overflow and underflow.
4. What is a circular queue? Write a C program to insert an item in the circular queue. Write another C function for printing elements of the queue in reverse order.
5. Can a queue be represented by a circular linked list with only pointer pointing to the tail of the queue? Write 'C' functions for "add" and "delete" operations on such a queue.
6. Define an input-restricted deque as a deque for which only the operation remvleft, remvright, insertleft are valid and an output-restricted deque as a deque for which only the operations remvleft, insertright, insertleft are valid. Show how each of these can be used to represent both a stack and a queue.
7. Show how to sort a set of input numbers using priority queue and the operation pqinsert, pqmindelete and empty.
8. Represent N queues in a single one-dimensional array. Write functions for 'add' and 'delete' operations on the i^{th} queue.

CHAPTER

8 Binary Trees

Key Features

- Introduction to Non-Linear Data Structures
- Introduction to Binary Trees
- Types of Trees
- Basic Definition of Binary Trees
- Properties of Binary Tree
- Representation of Binary Trees
- Operations on a Binary Search Tree
- Binary Tree Traversal
- Reconstruction of Binary Tree
- Counting Number of Trees
- Applications of Binary Trees

This chapter explores one of the most important non linear data structures, i.e. trees. Various kinds of trees have been discussed.

We begin with the most important tree structure—Binary tree—which is a finite set of elements that is either empty or further divided into subtrees. The two ways to represent binary trees are through arrays and linked lists. This chapter gives a detailed account of the various operations that can be performed on a binary search tree.

INTRODUCTION TO NON-LINEAR DATA STRUCTURES

The data structures we have discussed so far were mainly linear—strings, arrays, lists, stacks and queues.

This chapter discusses a non-linear data structure called **tree**. Trees are mainly used to represent data containing a hierarchical relationship between elements, for example, records, family trees and table of contents. Consider the following parent–child relationship:

In Fig. 8.1, each node represents a person whose name is written in that node. Each line connecting two nodes in the figure denotes a relation, namely '**parent–child**' relation between the connected nodes.

INTRODUCTION TO BINARY TREES

A tree may be defined as a finite set '**T**' of one or more nodes such that there is a node designated as the root of the tree and the other nodes (excluding the root) are divided into $n \geq 0$ disjoint sets $T_1, T_2, ..., T_n$ and each of these sets is a tree in turn. The trees $T_1, T_2, ..., T_n$ are called the sub-trees or children of the root.

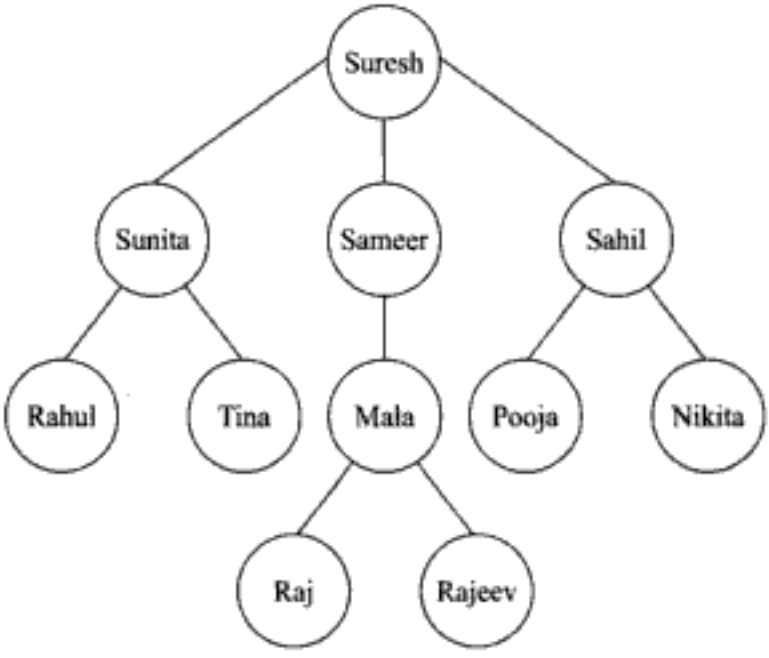


Fig. 8.1 *A Hypothetical Family Tree*

Generally, it is a convention to draw root node at the top and let the tree grow downwards.

Binary tree is a special type of tree in which every node or vertex has either no children, one child or two children. A binary tree is an important class of tree data structure in which a node can have at most two children (which are subtrees). Child of a node in a binary tree on the left is called the "left child" and the node in the right is called the "right child". An example of binary tree can be seen in Fig. 8.2.

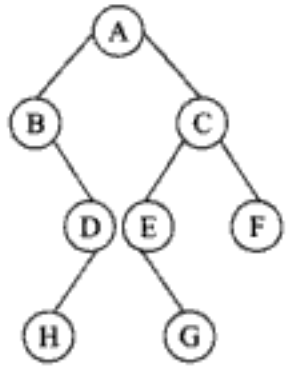


Fig. 8.2 *Binary Tree*

In Fig. 8.2, **A** is the root node which has two children **B** and **C**. The nodes **B**, **D** and **E** have only one child—**D**, **H** and **G** respectively. Every node in the tree is root of some other subtree. **B** and **C** are the roots of the subtrees of node **A**. In Fig. 8.2, node **E** has a right subtree rooted at **G** and node **B** has the right subtree rooted at **D**. The nodes **H**, **G** and **F** have no subtrees and they are the leaf nodes in the tree given above.

Similarly, a binary tree may also be defined as follows:

- A binary tree is an empty tree.
- A binary tree consists of a node called root, a left subtree and a right subtree both of which are binary trees once again.

Some of the examples of binary tree are given in Fig. 8.3:

In Fig. 8.3, (ii) and (iii) are distinct since the root of a binary tree of (ii) has an empty right subtree while the root of the binary tree of (iii) has an empty left subtree. But both the trees in (ii) and (iii) are same as the ordering of the subtrees of the root is not important for trees. The two trees are said to be identical if they have same structure or, in other words, if they have the same shape. The trees are said to be copies if they are similar and if they have the same contents at corresponding nodes.

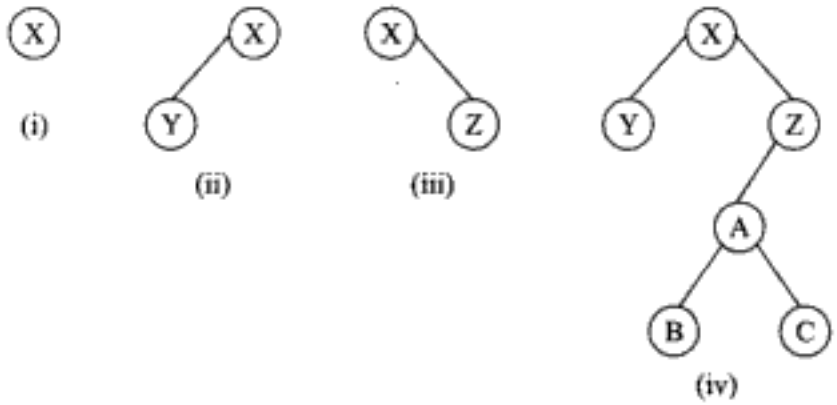


Fig. 8.3 *Examples of Binary Trees*

Applications of Binary Trees

Binary trees are used to represent a non-linear data structure. There are various forms of Binary trees. Binary trees play a vital role in software applications. One of the most important applications of Binary trees is in the searching algorithms. Most efficient and commonly used search known as Binary search uses a special form of binary tree known as Binary Search tree. A binary search tree always has two children and the left child node is always lighter than the root node, right child node is always heavier than the root node. Binary search tree brings down the time complexity of algorithm to less than 50%.

Another application of binary trees may be seen in populating a voluminous, relational and hierarchical data, into the memory. This increases the efficiency of the algorithm which manages this data. This is because Binary trees allow the algorithms to access a particular node at low cost and Binary trees also help to insert a new node easily.

Similarly, Binary trees are used in decision making, artificial intelligence, compilers, expression evaluation, etc.

TYPES OF TREES

General Trees

A general tree (sometimes called a tree) is defined as a non-empty finite set **T** of elements, called nodes, such that:

- The tree contains the root element.
- The remaining elements of the tree form an ordered collection of zero or more disjoint trees $T_1, T_2, \ldots T_m$.

The trees $T_1, T_2, \ldots T_m$ are called **subtrees** of root and the roots of $T_1, T_2 \ldots T_m$ are called *successors* of the root. For example, we assume that the general tree T is rooted, that is, the tree has a distinguished node **R** called the root of **T** and that **T** is ordered, i.e. children of each node **N** of Tree **T** has a specific order.

Figure 8.4 depicts a general tree—A, B, C, D, E, F, G, H, I, J, K, L, M, N.

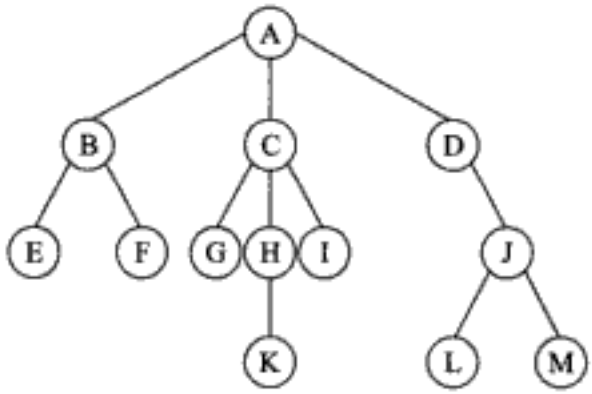


Fig. 8.4 *General Tree Representation*

Binary Search Tree

When we place constraints on how data elements can be stored in the tree, the items must be stored in such a way that the key values in left subtree of the root are less than the key value of the root, and the key values of all the nodes in the right subtree of the root are greater than the key value of the root. When this relationship holds in all the nodes in the tree then the tree is called a **binary search tree**.

The binary search tree is one of the most important data structures in computer science. This structure enables one to search for and find an element with an average running time = $\mathbf{O(\log_2 n)}$. Figure 8.5 depicts a binary search tree.

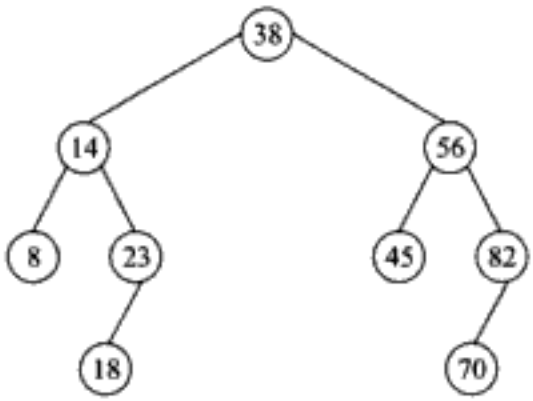


Fig. 8.5 *Binary Search Tree*

Extended Binary Tree or 2-Trees

A binary tree can be converted to an extended binary tree by adding new nodes to its leaf nodes, and to the nodes that have only one child. These new nodes are added in such a way that all the nodes in the resultant tree have either zero or two children. The extended tree is also known as a **2-tree**. The nodes of the original tree are called **internal nodes** and the new nodes that are added to binary tree, to make it extended binary tree, are called **external nodes**.

For example, consider the binary tree given in Fig. 8.6.

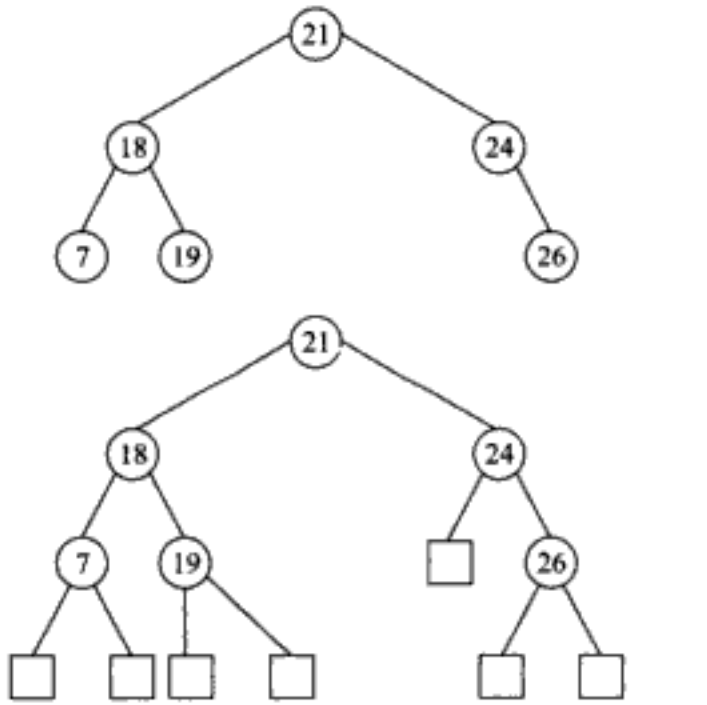


Fig. 8.6 *Binary Tree Converted to an Extended Binary Tree*

In Fig. 8.6, the nodes with circular shape are internal nodes and the nodes with square shape are called external nodes.

Few points to be remembered about extended binary trees are:

- If a tree has n nodes then the number of branches it has is n–1.
- Every node in a tree has exactly one parent except the root node.
- A single path connects any two nodes of a tree.
- For a binary tree of height **h** the maximum number of nodes can be $2^{h+1} - 1$.
- Any binary tree with n internal nodes has (n+1) external nodes.

Threaded Binary Tree

When a binary tree is represented using pointers the empty subtrees are set to NULL, i.e., 'left' pointer of a node whose left child is an empty subtree is normally set to NULL. Similarly, the 'right' pointer of a node whose right child is empty subtree is also set to NULL. Thus, a large number of pointers are set to NULL. These null pointers can be used in different ways. Assume that the 'left' pointer of a node '**n**' is set to NULL as the left child of '**n**' is an empty subtree, then the 'left' pointer of '**n**' can be set to point to the **inorder predecessor** of '**n**'. Similarly, if the 'right' child of a node '**m**' is empty the 'right' pointer of '**m**' can be set to point to the **inorder successor** of '**m**'. In Fig. 8.7 links with arrow heads indicate links leading to inorder predecessors or inorder successors.

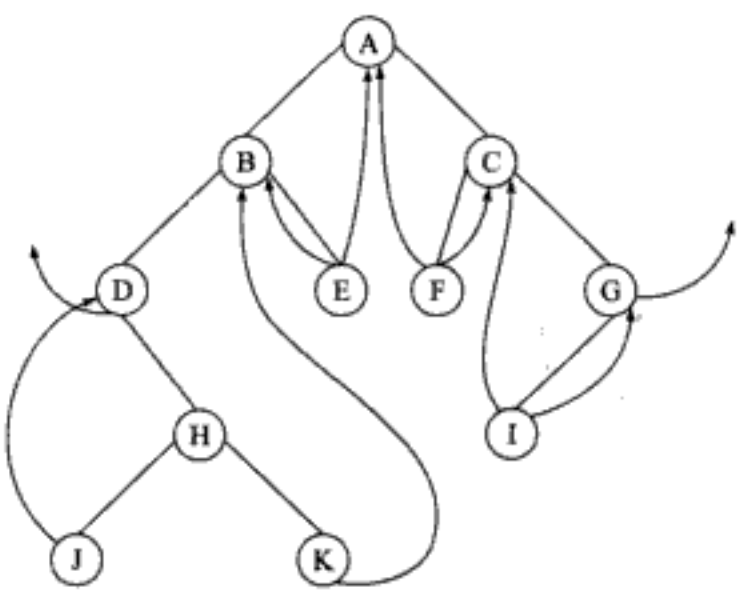


Fig. 8.7 *Threaded Binary Tree*

AVL Trees

If the heights of both left and right subtrees are given then the searching in binary tree is efficient. When we frequently make insertions and deletions in a binary search tree, it is likely to get unbalanced . The efficiency of searching is ideal if the difference between the heights of left and right subtrees of all the nodes in a binary search tree is at the most **one.** Such a binary tree is called an **AVL tree** or **Height Balanced Tree**. Figure 8.8 shows some of the AVL trees.

```
    printf("\n") ;

    for(i = 0 ; i < r ; i++)
    {
        temp = s.rhead[i] -> right ;
        if(temp != NULL)
        {
            while(temp -> right != NULL)
            {
                printf("Row: %d Col: %d Val: %d\n", temp -> row,
                temp -> col, temp -> val) ;
                temp = temp -> right ;
            }
            if(temp -> row == i)
                printf("Row: %d Col: %d Val: %d\n", temp -> row,
                temp -> col, temp -> val) ;
        }
    }
}

/* Deallocates memory */
void delsparse(struct sparse *p)
{
    int r = p -> smat -> noofrows ;
    struct rheadnode *rh ;
    struct node *temp1, *temp2 ;
      int i, c ;

    /* Deallocate memory of nodes by traversing rowwise */
    for(i = r - 1 ; i >= 0 ; i--)
    {
        rh = p -> rhead[i] ;
        temp1 = rh -> right ;

        while(temp1 != NULL)
        {
            temp2 = temp1 -> right ;
            free(temp1) ;
            temp1 = temp2 ;
        }
    }

    /* Deallocate memory of row headnodes */
```

```
    {
        /* Add element at proper position */
        while((temp1 != NULL) && (temp1 -> col < c))
        {
                temp2 = temp1 ;
                temp1 = temp1 -> right ;
        }
        temp2 -> right = p -> nd ;
        p -> nd -> right = NULL ;
    }

    /* Link proper col headnode with the node */
    ch = p -> smat -> firstcol ;
    for(j = 0 ; j < c ; j++)
        ch = ch -> next ;
    temp1 = ch -> down ;

    /* If col not pointing to any node */
    if(temp1 == NULL)
    {
        ch -> down = p -> nd ;
        p -> nd -> down = NULL ;
    }
    else
    {
    /* Link previous node in column with next node in same column */
        while((temp1 != NULL) && (temp1 -> row < r))
        {
                temp2 = temp1 ;
                temp1 = temp1 -> down ;
        }
        temp2 -> down = p -> nd ;
        p -> nd -> down = NULL ;
    }
}

void show_llist(struct sparse s)
{
    struct node *temp ;
    /* Get the first row headnode */
    int r = s.smat -> noofrows ;
    int i ;
```

3. Implement the following complex ADT using the 'C' programming language. The complex ADT is used to represent complex numbers of the form $z = \alpha + i\beta$ where α and β are real numbers and i is the imaginary number. The operations supported by this ADT are:
   i. Addition (z1, z2).

      [Note: $(\alpha_1 + i\beta_1) + (\alpha_2 + i\beta_2) = (\alpha_1 + \alpha_2) + i(\beta_1 + \beta_2)$**]**

   ii. Subtraction (21, 22)

      [Note: $(\alpha_1 - i\beta_1) - (\alpha_2 - i\beta_2) = (\alpha_1 - \alpha_2) - i(\beta_1 - \beta_2)$**]**

4. Write a 'C' function to compute the product 'C' of two sparse matrices 'A' and 'B' represented as ordered lists instead of two-dimensional arrays. To compute one element in 'C' dot product of one row of 'A' and one column of 'B' is to be evaluated, for this purpose, compute the transpose of B first. Now each row in transpose of 'B' corresponds to a column in B. B' now can be used for efficient multiplication.
5. Develop an ADT specification for "Polynominals". Also include the operations associated with polynominals.
6. Suggest a suitable data structure for representation of imaginary numbers. An imaginary number is represented by a+ib where i is the iota for the number. Also give specification for the operation associated with them.

Hidden page

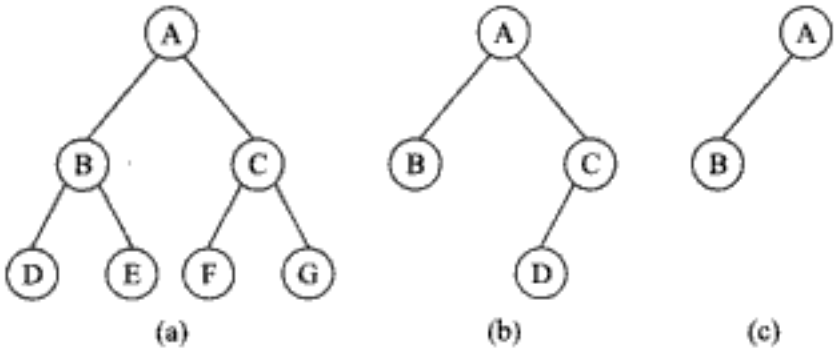


Fig. 8.8 *AVL Trees*

2-3 Trees

The insertion and deletion in an AVL tree involves many rotations to make it a balanced tree. This makes the entire operation complicated. To eliminate this complication, a data structure called 2–3 tree can be used. Figure 8.9 shows a 2–3 tree.

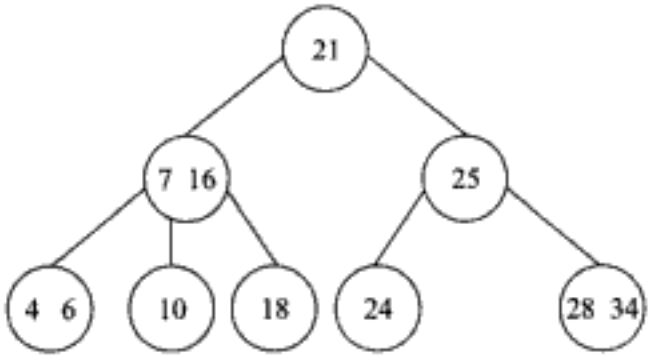


Fig. 8.9 *2–3 Tree*

B-Trees

A node of a binary search tree or an AVL tree can hold only one value; on the other hand, 2–3 tree can have at most two values per node. To improve the efficiency of operations performed on a tree we need to reduce the height of the tree. Therefore, B-tree is a balanced search tree data structure designed for use with large data sets in secondary storage. Figure 8.10 shows a B-tree of degree 4:

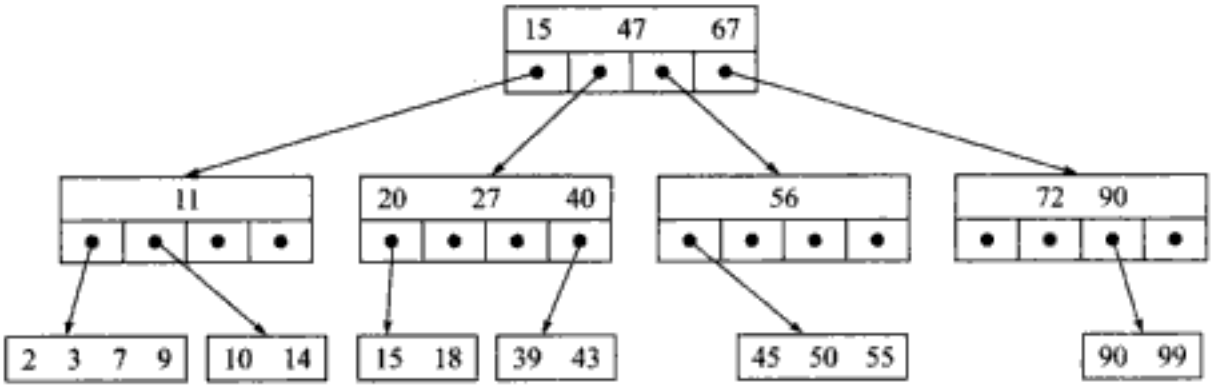


Fig. 8.10 *B-tree of Degree 4*

Heap

Heap is a complete binary tree. There are two types of heaps. If the value present at any node is greater than all its children, then the tree is called the **max-heap** or **descending heap**. In the case of **min-heap** or **ascending heap** the value present at any node is smaller than all its children.

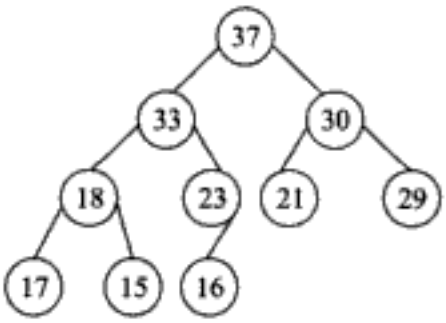


Fig. 8.11 *Max-heap*

Forest

Forest is a set of several trees that are not linked to each other. Forest can be represented as a binary tree. The Fig. 8.12 shows a binary tree built from a forest.

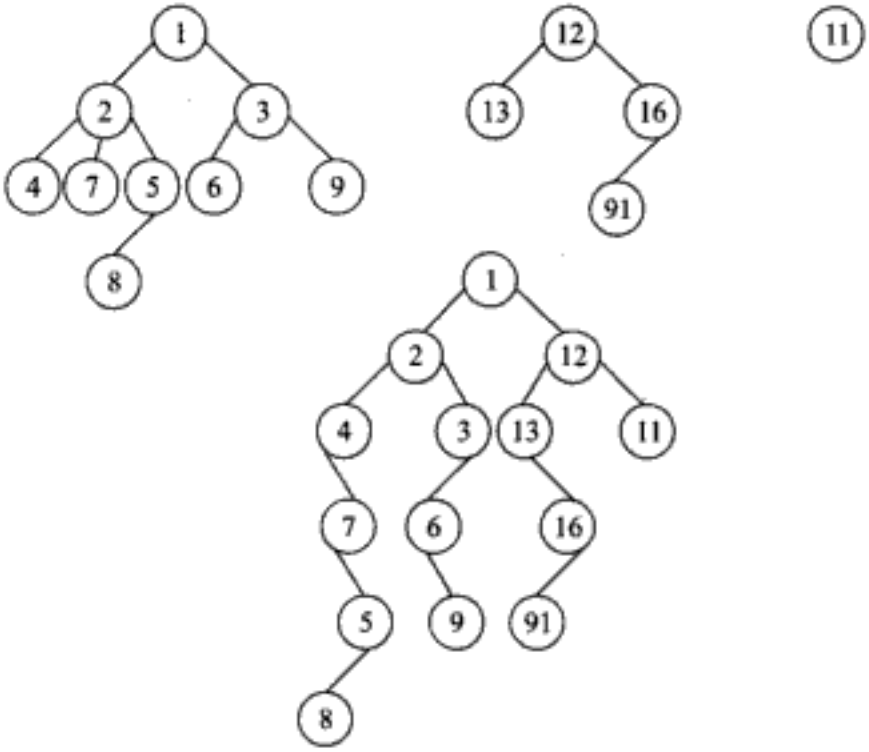


Fig. 8.12 *Forest*

Red Black Trees

In a red-black tree data structure, we adopt a colouring convention for the vertices in a binary search tree. Specifically, each vertex in a red-black tree is coloured with either red or black.

A red-black tree is an augmented binary search tree in which the arrangement of vertices obeys the following constraints:

- (Black rule) : Every leaf is coloured black.
- (Red rule) : If a vertex is red, then both of its children are black.
- (Path rule) : Every path from the root to a leaf contains the same number of black vertices.

Figure 8.13 shows a red black tree in which **black** vertices are shown **darkened,** the **red** vertices are **not darkened**, and the external vertices are drawn as boxes.

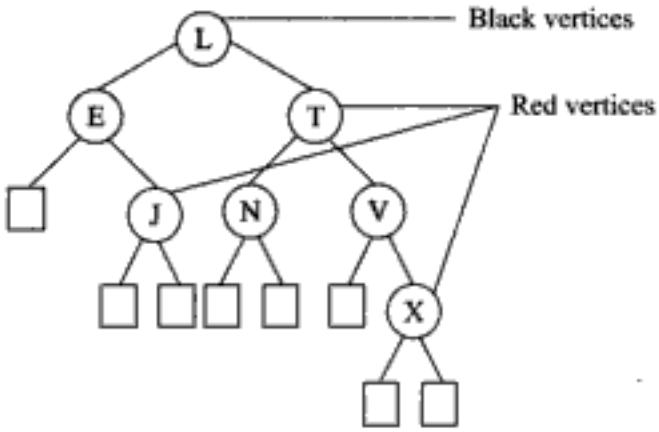


Fig. 8.13 *Red Black Tree*

BASIC DEFINITION OF BINARY TREES

Consider the tree given in Fig. 8.14 where **A** is the **root** of a binary tree.

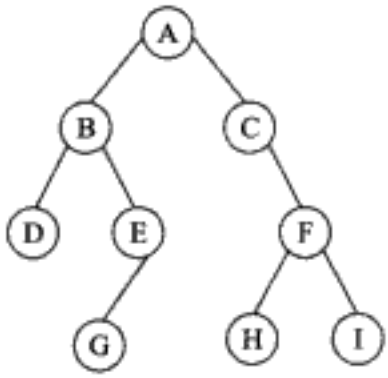


Fig. 8.14 *Binary Tree*

- **Father** As in Fig. 8.14, tree **A** is the root and **B** is the root of its left and right subtree, then **A** is said to be the **father** of **B** and **B** is said to be the **left** or **right** son.
- **Leaf Nodes** A node that has no sons such as (D, G, H or I) is called a leaf node.
- **Ancestor and Descendant** A node **n** is an ancestor of node $\mathbf{n}_2$ (and $\mathbf{n}_2$ is descendant of $\mathbf{n}_1$) if $\mathbf{n}_1$ is either the father of $\mathbf{n}_2$ or the father of some ancestor of $\mathbf{n}_2$, eg. in the tree shown in Fig. 8.14, **A** is an ancestor of **C.**
- **Left descendant and right descendant** A node $\mathbf{n}_2$ is a left descendant of node $\mathbf{n}_1$ if $\mathbf{n}_2$ is either a left son of $\mathbf{n}_1$ or a descendant of the left son of $\mathbf{n}_1$. A right descendant can be defined similarly as left descendant.
- **Siblings** All the children of a given vertex or node are known as **siblings.**
- **Strictly binary tree** If every non-leaf node in a binary tree has a non-empty left and right subtrees, the tree is known as **strictly binary tree**. The binary tree shown in Fig. 8.15 is a strictly binary tree.
- **Degree** of a node is the number of nodes connected to a particular node. For example, in Fig. 8.15 the node containing the data **D** has a degree 3. The degree of a leaf node is always one.

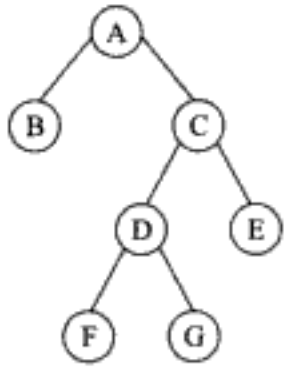


Fig. 8.15 *Strictly Binary Tree*

- **Level of a node** The level of a node in a binary tree is defined as follows: the root of the tree has level 0, and the level of any other node in the tree is one more than the level of its father. For example, consider Fig. 8.16. Node that **E** is at level 2 and node **H** is at level 3.

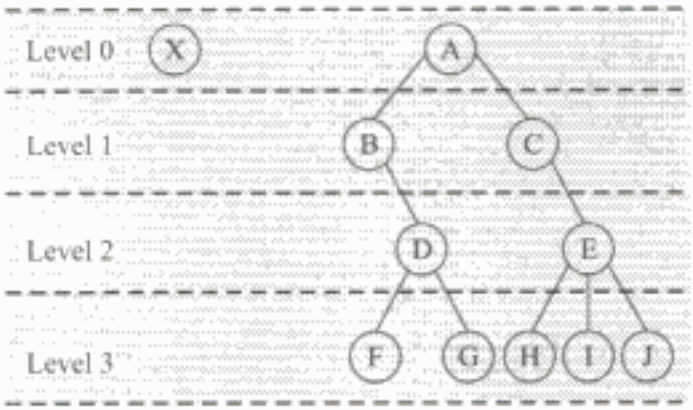


Fig. 8.16 *Level of a Node*

- **Depth/Height of a Tree** The height of a binary tree '**T**', denoted by **Height (T)**, is defined as follows:

  Height(T) = maximum level of any node of a binary tree T.

  Consider the binary tree in Fig. 8.17. The maximum level of any node in this tree is 4. Therefore, height of this tree is 4.

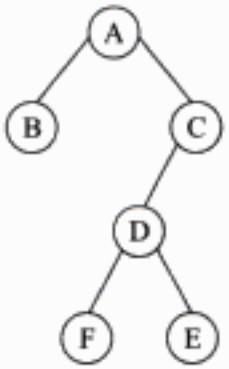


Fig. 8.17 *Binary Tree*

If all non-leaf nodes of a binary tree have exactly two non-empty children and levels of all leaf nodes of a binary tree is same then the tree is called **complete** or **full binary tree**. The tree shown above is not a full binary tree as the levels of leaf nodes '**B**' and '**D**' are not the same and there is a node '**C**' which has only one non-empty child.

An example of complete binary tree can be seen in Fig. 8.18.

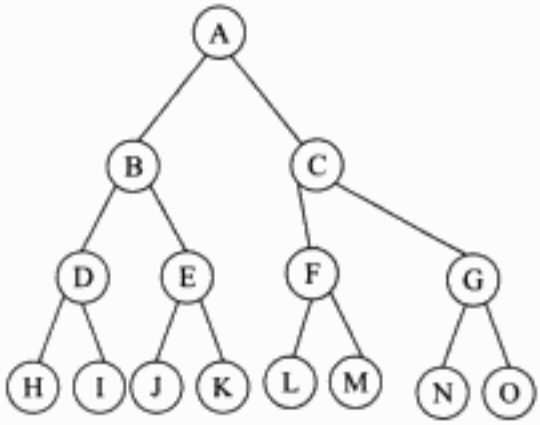


Fig. 8.18 *A Complete Binary Tree of Height 3*

PROPERTIES OF BINARY TREE

- A tree with "**n**" nodes has exactly **(n–1)** edges or branches.

  This property can be proved by induction on the number of nodes in a tree. If **n=0** then the tree contains only one node and the number of edges or branches is **0**. Therefore, the induction base is proved.

  Consider any tree with "**n**" nodes. It must have a root and a root must have **K** children **(K>0)**. Let n_i depict the number of nodes in the i^{th} child for **i=0** to **K–1**. Naturally, $n=1+\Sigma_{i=0,\ K-1}(n_i)$. Each child is itself a tree with fewer than **n** nodes. Hence, the induction hypothesis can be assumed to be valid for these trees. Therefore, the number of branches

Hidden page

Hidden page

Hidden page

```
        {
                if(tree[i-1].data>num)
                {
                        i=2*tree[i-1].k;
                }
                else
                {
                        i=2*tree[i-1].k+1;
                }
        }
        tree[i-1].k=i;
        tree[i-1].data=num;
    }
}
void display(void)
{
    int row=0,col=0,root = 0;
    int i=0;
    for(i=0;i<=MAX;i++)
    {
        printf("\n %d",tree[i].data);
    }
    //getch();
}
void main()
{
    int length=10;
    char ch='y';
    int choice = 0;
    int key = 0;
    clrscr();
    do
    {
        clrscr();
        printf("\n\t <<< ARRAY REPRESENTATION OF BINARY TREE >>>\n");
        printf("\n 1. Initialize.");
        printf("\n 2. Insert.");
        printf("\n 3. Delete.");
        printf("\n 4. Display.");
        printf("\n 5. Exit.\n");
        printf("\n Enter Your Choice : ");
        scanf("%d",&choice);
```

```
        switch(choice)
        {
        case 1:
                //INITIALIZE
                initialize(tree,length);
                printf("\n Binary Tree Initialized.\n");
                break;
        case 2:
                //INSERT
                printf("\n Enter The Key : ");
                scanf("%d",&key);
                create(tree,key);
                break;
        case 3:
                //DELETE
                break;
        case 4:
                //DISPLAY
                display();
                break;
        case 5:
                //EXIT
                exit(1);
        default:
                printf("\n Invalid Choice, Try Again...");
        }
        fflush(stdin);
        printf("\n Do You Wish to Continue [y/n] : ");
        ch=getch();
    }while(ch=='y');
    getch();
}
```

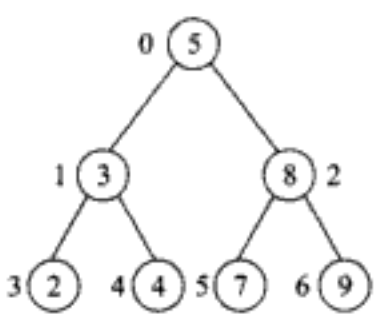


Fig. 8.19 *Binary Tree*

— Consider the binary tree in Fig. 8.19, if we index each node in the tree in the order of their levels and their position at the particular level then each node will occupy its unique index number. Starting from root having index 0 then the left child of the root that is node having key value 3 will have index 1 followed by the right child of root with index 2 and so on.

In a binary tree let **K** to be the index of any node then left child of the node is at position **2K** and the right child of the node is at the position **2K+1**. Thus, we conclude that any node with index **K** has a left child at position **2K** or with index **2K** and the right child at position **2K+1** or with index **2K+1**. Similarly, in the array representation of binary trees, the root of the tree is stored in the first element of the array

Hidden page

For example, consider the following binary tree.

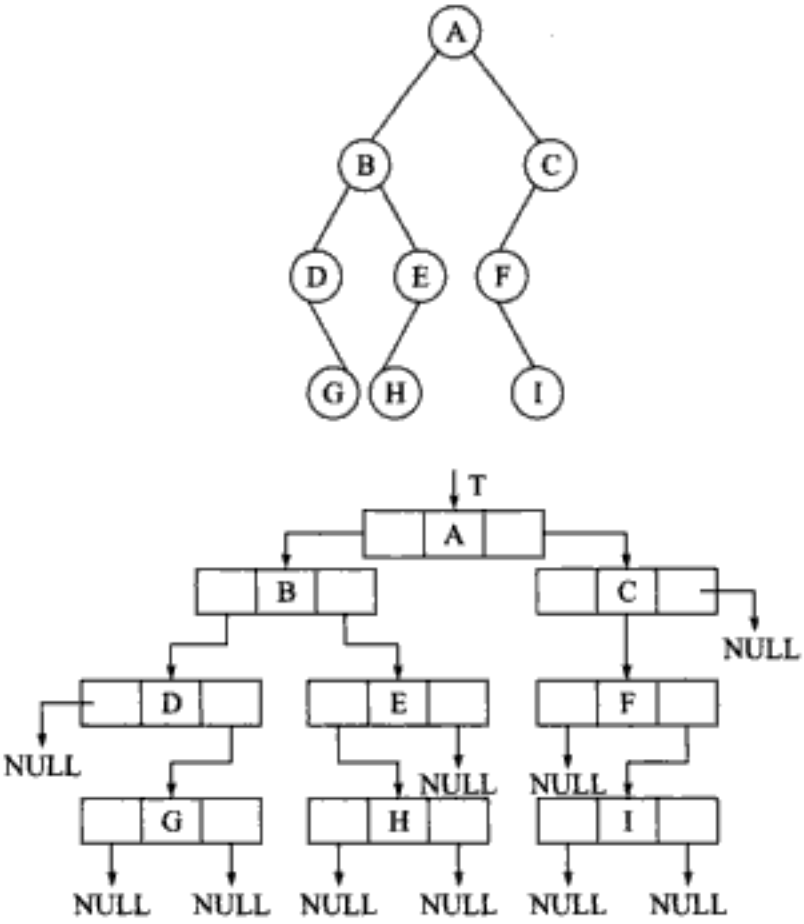


Fig. 8.20 *Linked List Representation of Binary Tree*

OPERATIONS ON A BINARY SEARCH TREE

Searching, insertion and deletion of a node are the most basic operations that are required to maintain a tree.

Searching a Node

To search any node in a binary tree, first of all the data to be searched is compared with the data of the root node. If the data is equal to the data of the root node then the searching is successful. If the data is found to be greater than the data of the root node then the searching proceeds in the right subtree of the root node, otherwise, searching proceeds in the left subtree of the root node.

Same procedure is repeated for left and right subtrees until the data is found. While searching the data if the leaf node of tree is reached and data is not found then it is concluded that the data is not present in the tree.

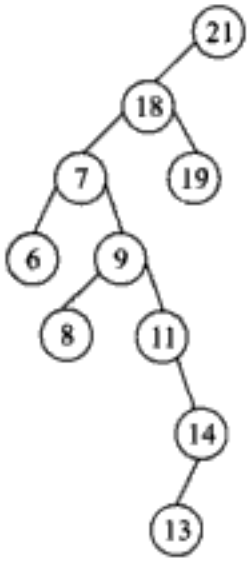


Fig. 8.21 *Searching Node*

For example, consider the binary search tree **T**. Suppose item to be searched is 9.

1. Compare ITEM = 9 with the root 21 of the tree **T**, since 9 < 21, proceed to the left child of the tree.
2. Compare ITEM = 9 with 18 since 9 < 18 proceed to the left subtree of 18, which is 7.
3. Compare ITEM = 9 with 7 since 9 > 7 proceed in the right subtree of the node which holds a value 7.
4. Now 9 is compared with the node which holds the value 9 and since 9 is found the searching process ends here.

Insertion of a Node in a Binary Tree

For node insertion in a binary search tree, initially the data that is to be inserted is compared with the data of the root node. If data is found to be greater than or equal to the data of the root node then the new node is inserted in the right subtree of the root node, otherwise, the new node is inserted in the left subtree of the root node.

The root node of the right or left subtree is considered and its data is compared with the data that is to be inserted and the same procedure is repeated. This is done till the left or right subtree where the new node is to be inserted is found to be empty. Finally, the new node is made the appropriate child of this current node.

For example, consider the binary search tree **T** in Fig. 8.22. Suppose ITEM = 20 is given to insert.

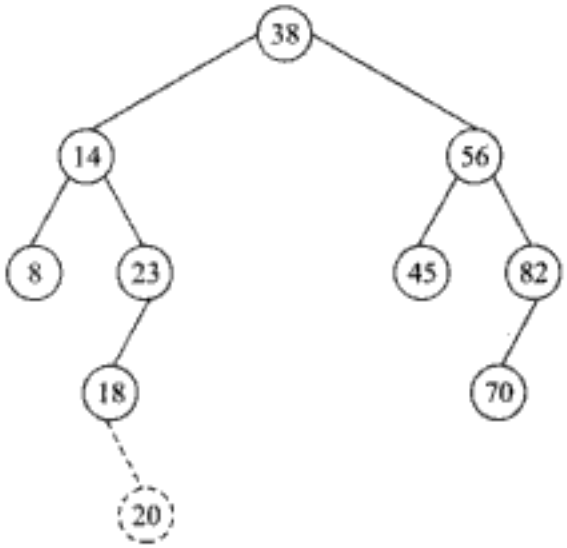


Fig. 8.22 Newly inserted node

1. Compare ITEM = 20 with the root 38 of the tree **T**. Since 20 < 38 proceed to the left child of 38, which is 14.
2. Compare ITEM = 20 with 14, since 20 > 14 proceed to the right child of 14, which is 23.
3. Compare ITEM = 20 with 23 since 20 < 23 proceed to the left child of 23 which is 18.
4. Compare item = 20 with 18 since 20 > 18 and 18 does not have right child, 20 is inserted as the right child of 18.

Deletion from a Binary Tree

In deletion process there are four possible conditions we need to take into account:

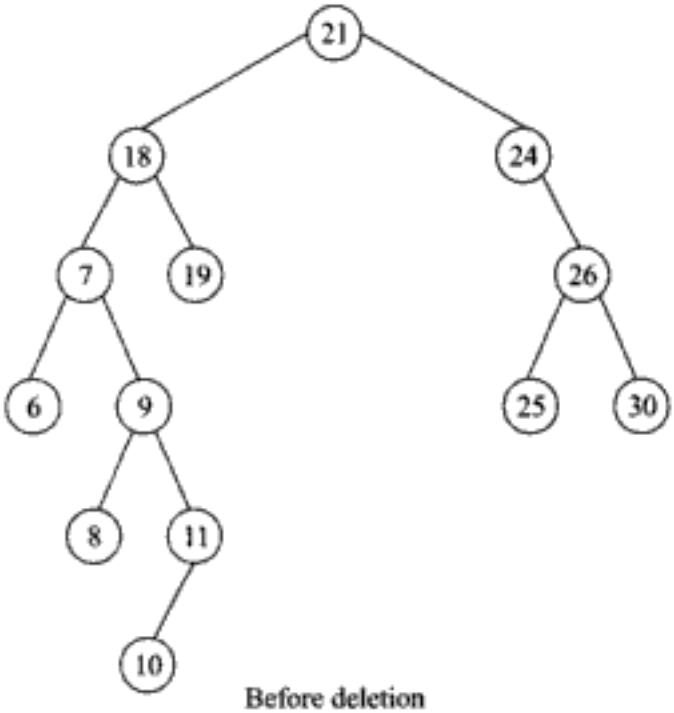
(i) No node in the tree holds the specified data.
(ii) The node containing the data has no children.
(iii) The node containing the data has exactly one child.
(iv) The node containing data has two children.

Condition (i) In this case we simply print the message that the data item is not present in the tree.

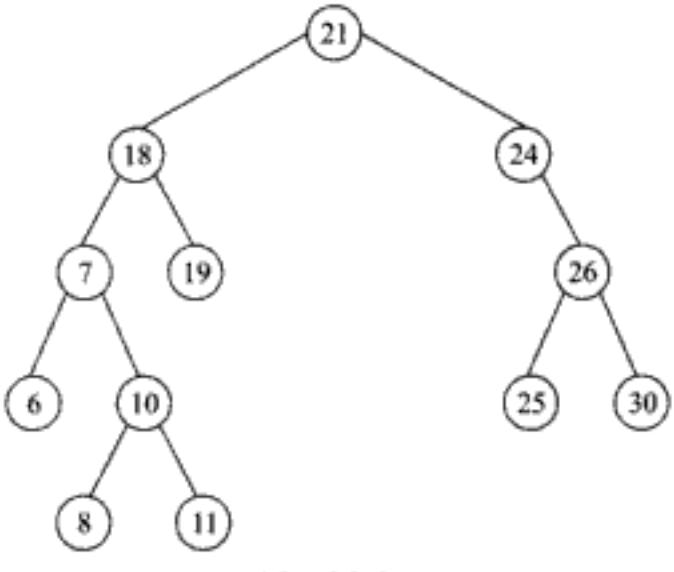
Condition (ii) In this case since the node to be deleted has no children the memory occupied by this should be freed and either the left link or the right link of the parent of this node should be set to NULL. Which of these is to be set to NULL depends upon whether the node being deleted is a left child or right child of its parent.

Hidden page

copied into the node to be deleted and a pointer should be set up pointing to the inorder successor (node 10). This inorder successor would always have one or zero child. It should then be deleted using the same procedure as for deleting one child or a zero child node. Thus, the whole logic of deleting a node with two children is to locate the inorder successor, copy its data and reduce the problem to a simple deletion of a node with one or zero child. This is shown in Fig. 8.24.



Before deletion



After deletion

Fig. 8.24 *Deletion of Node that has Both Left and Right Child (i.e. Node 9)*

Hidden page

```
  inorder(bt) ;
  getch();
}
/* inserts a new node in a binary search tree */
void insert(struct btreenode **sr, int num)
{
  if(*sr == NULL)
  {
       *sr = malloc(sizeof (struct btreenode)) ;
       (*sr) -> leftchild = NULL ;
       (*sr) -> data = num ;
       (*sr) -> rightchild = NULL ;
  }
  else  /* Search the node to which new node will be attached */
  {
       /* If new data is less, traverse to left */
       if(num < (*sr) -> data)
              insert(&((*sr) -> leftchild), num) ;
       else
              /* Else traverse to right */
              insert(&((*sr) -> rightchild), num) ;
  }
}
/* Deletes a node from the binary search tree */
void delete(struct btreenode **root, int num)
{
  int found ;
  struct btreenode *parent, *x, *xsucc ;
  /* If tree is empty */
  if(*root == NULL)
  {
       printf("\nTree is empty") ;
       return ;
  }
  parent = x = NULL ;
  /* Call to search function to find the node to be deleted */
  search(root, num, &parent, &x, &found) ;
  /* If the node to be deleted is not found */
  if(found == FALSE)
  {
       printf("\nData to be deleted, not found") ;
       return ;
```

```
}
/* If the node to be deleted has two children */
if(x -> leftchild != NULL && x -> rightchild != NULL)
{
     parent = x ;
     xsucc = x -> rightchild ;
     while(xsucc -> leftchild != NULL)
     {
            parent = xsucc ;
            xsucc = xsucc -> leftchild ;
     }
     x -> data = xsucc -> data ;
     x = xsucc ;
}
/* If the node to be deleted has no child */
if(x -> leftchild == NULL && x -> rightchild == NULL)
{
     if(parent -> rightchild == x)
            parent -> rightchild = NULL ;
     else
            parent -> leftchild = NULL ;
     free ( x ) ;
     return ;
}
/* If the node to be deleted has only right child */
if(x -> leftchild == NULL && x -> rightchild != NULL)
{
     if(parent -> leftchild == x)
            parent -> leftchild = x -> rightchild ;
     else
            parent -> rightchild = x -> rightchild ;
     free(x) ;
     return ;
}
/* If the node to be deleted has only left child */
if(x -> leftchild != NULL && x -> rightchild == NULL)
{
     if(parent -> leftchild == x)
            parent -> leftchild = x -> leftchild ;
     else
            parent -> rightchild = x -> leftchild ;
     free(x) ;
```

```
        return ;
    }
}
/*Returns the address of the node to be deleted, address of its parent and
   whether the node is found or not */
void search(struct btreenode **root, int num, struct btreenode **par, struct
        btreenode **x, int *found)
{
    struct btreenode *q ;
    q = *root ;
    *found = FALSE ;
    *par = NULL ;
    while(q != NULL)
    {
        /* If the node to be deleted is found */
        if(q -> data == num)
        {
                *found = TRUE ;
                *x = q ;
                return ;
        }
        *par = q ;
        if(q -> data > num)
                q = q -> leftchild ;
        else
                q = q -> rightchild ;
    }
}
/* Traverse a binary search tree in an LDR (Left-Data-Right) fashion */
void inorder(struct btreenode *sr)
{
    if(sr != NULL)
    {
        inorder(sr -> leftchild) ;
    /* Print the data of the node whose left child is NULL or the path has
                already been traversed */
        printf("%d\t", sr -> data) ;
        inorder (sr -> rightchild) ;
    }
}
```

In the above program the working of the function **insert()** and **inorder()** is exactly the same as what we saw in other program. The function **search()** searches for the given number in the tree and returns

the address of its parent and an integer value which holds either TRUE or FALSE (0 or 1) depending upon whether the number is found or not.

The function **search()** receives five parameters. The first parameter **sr** is the address of the root node of the tree, second parameter **num** is the number that is to be searched and the third and fourth parameters (par and x) are the pointers to the addresses of the parent of the node where data is found and the address of the node itself, respectively, and last parameter **found** is the integer value indicating whether the element is found or not.

In the **insert()** function, initially the variable pointed to by **found** is set to **FALSE** value and the value of **par** is set to NULL, because if the node is not found then the pointer pointed by **par** should hold a **NULL** value. Then a while loop is executed with a condition **q!=NULL** where **q** holds the address of the root node. Inside the while loop a condition is checked for the data to be searched. If the data is found then a **TRUE** value is stored in the variable printed by **found** and the address of current node is stored in **x**. If the data is found in the first iteration then the value of **par** is NULL, since it has no parent.

If the data is not found in the first iteration then inside the while the address of current node is stored in **par** and then the data is compared with the data present in the current node. If the data of the current node is greater than the data which is to be searched then **q** will hold the address of the left subtree of the current node otherwise it will hold the address of the right subtree of the current node. This way any function that calls **search()** gets the address of the node where the node is present and the address of its parent.

The function **removes()** calls the function **rem()** to delete the node that contains the given number. In the function **rem()** two parameters are received. The first is the address of the root node and other is the number that is to be deleted. Initially, a condition is checked whether a new node is empty. If it is, then the message is displayed and the control is returned, otherwise the function **search()** is called with NULL values stored in the two pointer variables **parent** and **x**.

If the data that is to be deleted is not found then the last argument passed to function **search()** holds a value **FALSE**. Hence, after the call to function **search()** a condition is checked whether **found = FALSE**. If it is equal to FALSE then it indicates that the data is not present in the tree. So far appropriate message will be displayed and control will be returned back.

If the data that is to be deleted is found then the following four conditions would arise:

Condition (i) Node to be deleted has two children.

In this case, initially the address of the node that is to be deleted is stored in the pointer **parent**. But by doing so the address of the parent node that is to be deleted is lost. We do not mind doing so, because we are not interested in storing any or the NULL value in the left and right child of the parent node. What we need to do is to find out the in-order successor of the node that is to be deleted. For this the address of the right child of the node that is to be deleted is stored in pointer variable **xsucc**. Then a while loop is executed where a condition is checked whether the left child of **xsucc,** is NULL. If it is not then the address of **xsucc** is stored in the parent and the left child of the **xsucc** is stored in **xsucc**. As a result at the end of while loop the value present in **xsucc** is the address of the inorder successor of the node that is to be deleted and will always either have one child or no child.

After the while loop the value present in the inorder successor is copied into the data that is to be deleted. Finally the address of inorder successor is stored in **x** which is the node that is to be deleted. So the logic of deleting the records which have two children is converted to the case of deleting the node which has only one child or no child as discussed below.

Condition (ii) Node to be deleted has no child.
Since both the children of the node that is to be deleted hold the NULL value, is stored a NULL value in the respective child of the parent. This is done by checking the condition **parent → rightchild==x**.

If the right child of the parent node is equal to the value in the node to be deleted, then a NULL value is stored in the rightchild of the parent, otherwise it is stored in the leftchild. Finally, the memory occupied by the node to be deleted is released and control is returned.

Condition (iii) Node to be deleted has only right child.
Here a condition is checked whether the node that is to be deleted is the left child of its parent, this is done through the following statement:

If (parent → leftchild = = x)

If this condition is true then the address of the right child of the node that is to be deleted is stored in the left child of the parent node, otherwise in the right child of the parent node. Then the memory 0 occupied by the node to be deleted is released and the control is returned.

Condition (iv) Node to be deleted has only left child.
The function for this works exactly as for the node which has only right child.

BINARY TREE TRAVERSAL

Recursive Traversal of Binary Tree

Traversal of a binary tree is one of the most important operations required to be performed on a binary tree. Traversing is the process of visiting every node in the tree exactly once. Therefore, a complete traversal of binary tree implies visiting the nodes of the tree in some linear sequence.

There are three standard ways of traversing a binary tree **T** with root **R**:

- Preorder or depth-first order
- Inorder or symmetric order
- Postorder

Inorder Traversal In the inorder traversal we follow the following three steps:

- Traverse the left subtree in inorder.
- Visit the root.
- Traverse the right subtree in inorder.

For example, consider the binary tree given below:

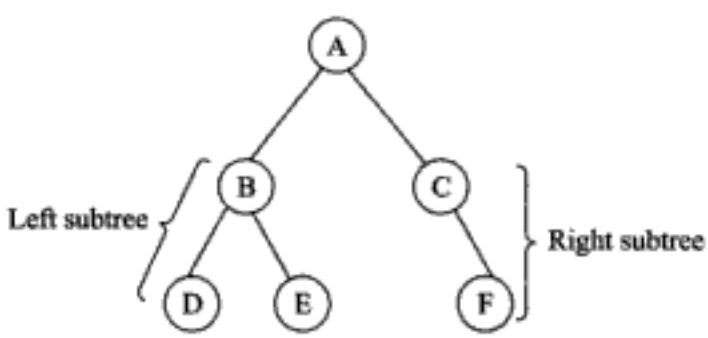


Fig. 8.25 *Inorder Traversal*

Hidden page

```
#include<alloc.h>
struct NODE
{
  int data;
  struct NODE *Left_Child;
  struct NODE *Right_Child;
};
struct NODE *Binary_Tree(char *, int, int);
void display(struct NODE *, int);
void Pre_order(struct NODE *);
void In_order(struct NODE *);
void Post_order(struct NODE *);
/* Function to create a binary tree */
struct NODE *  Binary_Tree (char *List, int Lower, int Upper)
{
  struct NODE *Node;
  int Mid = (Lower + Upper)/2;
  Node = (struct NODE*) malloc(sizeof(struct NODE));
  Node->data = List [Mid];
  if(Lower>= Upper)
  {
       Node->Left_Child = NULL;
       Node->Right_Child = NULL;
       return(Node);
  }
  if(Lower <= Mid - 1)
       Node->Left_Child = Binary_Tree(List, Lower, Mid - 1);
  else
       Node->Left_Child = NULL;
  if(Mid + 1 <= Upper)
       Node->Right_Child = Binary_Tree(List, Mid + 1, Upper);
  else
       Node->Right_Child = NULL;
  return(Node);
}
/* Output function */
void display(struct NODE *T, int Level)
{
  int i;
  if(T)
  {
       display(T->Right_Child, Level+1);
       printf("\n");
```

Hidden page

Hidden page

The description given above can be explained by the function given below:

```
void inorder(struct btree *root)
{
  while(1)
  {
       while(root! = NULL)
       {
              push(&stk, root → left);
              root = root → left;
       }
       if(empty(&stk))
              return;
              root = pop(&stk);
       printf ("%d", root → data);
       root = root → right;
  }
}}
```

For preorder and postorder traversals change the position of printf function accordingly.

/* Program to non-recursively traverse a binary tree */

```
# include<stdio.h>
# include<conio.h>
# include<stdlib.h>
struct btree
{
  int d;
  struct btree *l;
  struct btree *r;
} *p,*stk[50];
int top=-1,x;
 void insert(struct btree *,int);
 void delete(int);
 void inorder(struct btree *);
 void preorder(struct btree *);
 void postorder(struct btree *);
 void showtree(struct btree *,int,int);
 void disptree(struct btree *);
 void main(void)
 {
  int n,ch,val,i=1,x,nd;
  p=NULL;
  clrscr();
```

```
while(1)
{
      clrscr();
      printf("B i n a r y   T r e e   S t r u c t u r e\n");
      printf("1.enter the data\n");
      printf("2.want inorder traversing\n");
      printf("3.want preorder traversing\n");
      printf("4.want postorder traversing\n");
      printf("5.delete\n");
      printf("6.show tree\n");
      printf("7.exit\n\n");
     printf("\n Enter your choice: \n\n" );
  switch(ch)
     {
            case 1:printf(" Insert the data :");
             scanf("%d",&val);
             insert(p,val);
             showtree(p,38,9);
             getch();
             break;
            case 2:printf("Inorder Traversal:\n");
             inorder(p);
             break;
            case 3:printf("Preorder Traversal:\n");
             preorder(p);
             break;
            case 4:printf("Postorder Traversal:\n");
             postorder(p);
             break;
            case 5:printf("Enter the value you want to delete:");
             scanf("%d",&nd);
             delete(nd);
             for(i=9;i<=23;i++)
                   printf("                                    \n");
             showtree(p,38,9);
             getch();
             break;
            case 6:showtree(p,38,9);
             getch();
             break;
            case 7:exit(1);
      break;
```

```
        default:
        printf("enter your choice again");
       continue;
        };
       }
 }
void insert(struct btree *q,int v)
 { struct btree *temp,*pr;
        temp=malloc(sizeof(struct btree));
        temp->l=temp->r=NULL;
        temp->d=v;
       if (p==NULL)
       p=temp;
       else
    {
     while(q!=NULL)
     {
     pr=q;
     if(v <q->d)
      q=q->l;
     else
      q=q->r;
     }
     q=temp;
     if(v <pr->d)
      pr->l=temp;
    else
     pr->r=temp;
     }
    }
void inorder(struct btree *q)
{
 do
  { while(q!=NULL)
    { stk[++top]=q;
      q=q->l;
    }
    if (top!=-1)
     { q=stk[top--];
       printf("\t%d",q->d);
       q=q->r;
     }
```

Hidden page

```
    }
   } while(q!=NULL || top!=-1);
   getch();
 }
void delete(int dd)
{
 struct btree *q,*cur,*par;
 q=p;
 while(q!=NULL)
 {
  if(q->d>dd)
  {
   par=q;
   q=q->l;
   continue;
  }
  if(q->d<dd)
  {
   par=q;
   q=q->r;
   continue;
  }
  if(q->d==dd)
   break;
  else
  {
   printf("Value not found, press any key...");
   getch();
   return;
  }
 }
 cur=q;
 //Case when current node is leaf node
 if(cur->l==NULL && cur->r==NULL)
 {
  if(par->d<dd)
   par->r=NULL;
  else
   par->l=NULL;
  free(cur);
  return;
 }
```

Hidden page

```
  gotoxy((col-2),(row+1));
  printf("/");
  if((row%2)==0)
   showtree(q->l,(col-3),(row+3));
  else
   showtree(q->l,(col-4),(row+3));
 }
 if(q->r!=NULL)
 {
  gotoxy((col+2),(row+1));
  printf("\\");
  if((row%2)==0)
   showtree(q->r,(col+3),(row+3));
  else
   showtree(q->r,(col+4),(row+3));
 }
return;
}
```

RECONSTRUCTION OF BINARY TREE

The postorder, preorder and inorder traversal of binary trees may result in same sequence of nodes. As a result original tree cannot be reconstructed, given its inorder or preorder or postorder traversal alone. However, if sequence of nodes produced by inorder and postorder traversal of a binary tree are provided then a unique binary tree can be reconstructed. Consider the following example which illustrates the reconstruction of a binary tree given its inorder and postorder traversal.

Inorder : H D I J E K B A L F M C N G O
Postorder : H I D J K E B L M F N O G C A

Since the first node visited in postorder traversal of a binary tree is the left node, the root of the binary tree becomes **'A'**. In reconstruction of binary tree from inorder and postorder, the first node is taken from the right hand side of the postorder sequence, i.e. **A**.

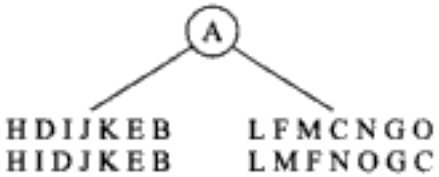


Fig. 8.27

Therefore the nodes to the left of **'A'** in the given **inorder sequence** belong to the left subtree and nodes to the right of **'A'** belong to the right subtree of the tree. Moreover, the order in which the nodes to the left of 'A' occur in the given inorder sequence is the same as the inorder sequence of the left subtree.

Now the same scheme outlined earlier can be applied to both the left and the right subtree once again. Now, from the figure above consider the left subtree, i.e. I.S. **HDIJKEB** and P.S. **HIDJKEB**. From the postorder sequence the root of this subtree is **'B'**. The inorder sequence of the left and right subtree of the subtree rooted at B are HDIJKE and HIDJKE.

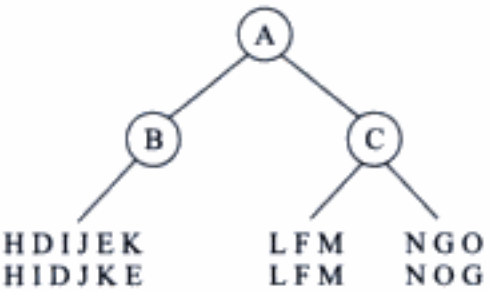


Fig. 8.28

Continuing the same set of operations in each subtree, the tree can be reconstructed. Further again from the postorder sequence, the root of this subtree is **E**. The position of **E** in inorder sequence determines its position on the left subtree rooted at B whereas again, looking in the postorder sequence we find K as the next root and its order in inorder sequence determines its position in the right subtree rooted at E. Similarly, the steps are repeated for all the remaining left and right subtrees. Therefore, the reconstruction of the tree can be seen in Fig. 8.29.

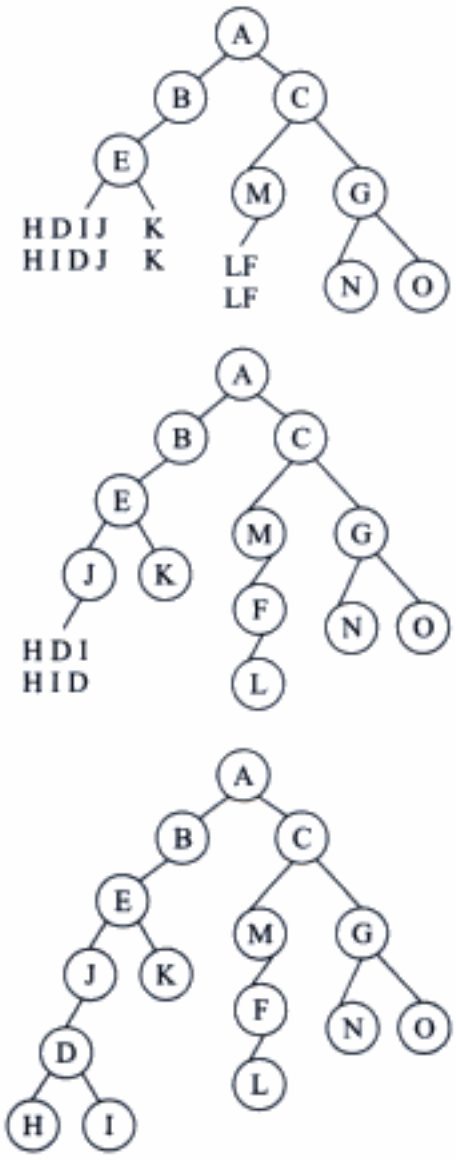


Fig. 8.29

/* Program to reconstruct a binary search tree. */

```
#include <stdio.h>
#include <conio.h>
#include <alloc.h>
#define MAX 101
struct node
{
  struct node *left ;
  int data ;
  struct node *right ;
} ;
void insert(struct node **, int) ;
void preorder(struct node *) ;
void postorder(struct node *) ;
void inorder(struct node *) ;
struct node * recons(int *, int *, int) ;
void deltree(struct node *) ;
```

```
int in[MAX], pre[MAX], x ;
void main( )
{
  struct node *t, *p, *q ;
  int req, i, num ;
  t = NULL ;  /* Empty tree */
  clrscr( ) ;
  printf("Specify the number of items to be inserted:") ;
  while(1)
  {
       scanf("%d", &req) ;
       if (req >= MAX || req <= 0)
              printf ("\nEnter number between 1 to 100.\n") ;
       else
              break ;
  }
  for(i = 0 ; i < req ; i++)
  {
       printf("Enter the data:") ;
       scanf("%d", &num) ;
       insert(&t, num) ;
  }
  printf("\nIn-order   Traversal:\n") ;
  x = 0 ;
  inorder(t) ;
  printf("\nPre-order  Traversal:\n") ;
  x = 0 ;
  preorder(t) ;
  printf("\nPost-order Traversal:\n") ;
  x = 0 ;
  postorder(t) ;
  deltree(t) ;
  t = NULL ;
  t = recons(in, pre, req) ;
  printf("\n\nAfter reconstruction of the binary tree.\n") ;
  x = 0 ;
  printf("\nIn-order   Traversal:\n") ;
  inorder(t) ;
  x = 0 ;
  printf("\nPre-order  Traversal:\n") ;
  preorder(t) ;
  x = 0 ;
```

```
  printf("\nPost-order Traversal:\n") ;
  postorder(t) ;
  deltree(t) ;
  getch( ) ;
}
/* Inserts a new node in a binary search tree */
void insert(struct node **sr, int num)
{
  if(*sr == NULL)
  {
       *sr = (struct node *) malloc(sizeof (struct node)) ;
       (*sr) -> left = NULL ;
       (*sr) -> data = num ;
       (*sr) -> right = NULL ;
       return ;
  }
  else  /* Search the node to which new node will be attached */
  {
       /* If new data is less, traverse to left */
       if(num < (*sr) -> data)
              insert(&((*sr) -> left), num) ;
       else
              /* Else traverse to right */
              insert(&((*sr) -> right), num) ;
  }
}
void preorder(struct node *t)
{
  if(t != NULL)
  {
       printf("%d\t", pre[x++]= t -> data) ;
       preorder(t -> left) ;
       preorder(t -> right) ;
  }
}
void postorder(struct node *t)
{
  if(t != NULL)
  {
       postorder(t -> left) ;
       postorder(t -> right) ;
       printf("%d\t", t -> data) ;
  }
```

```
}
void inorder(struct node *t)
{
  if (t != NULL)
  {
       inorder(t -> left) ;
       printf("%d\t", in[x++]= t -> data) ;
       inorder(t -> right) ;
  }
}
struct node * recons(int *inorder, int *preorder, int noofnodes)
{
  struct node *temp, *left, *right ;
  int tempin[100], temppre[100], i, j ;
  if(noofnodes == 0)
       return NULL ;
  temp = (struct node *) malloc(sizeof(struct node)) ;
  temp -> data = preorder[0] ;
  temp -> left = NULL ;
  temp -> right = NULL ;
  if(noofnodes == 1)
       return temp ;
  for(i = 0 ; inorder[i] != preorder[0] ;)
       i++ ;
  if(i > 0)
  {
       for(j = 0 ; j <= i  ; j++)
              tempin[j] = inorder[j] ;
       for(j = 0 ; j < i  ; j++)
              temppre[j] = preorder[j + 1] ;
  }
  left = recons(tempin, temppre, i) ;
  temp -> left = left ;
  if(i < noofnodes - 1)
  {
       for(j = i ; j < noofnodes - 1 ; j++)
       {
              tempin[j - i] = inorder[j + 1] ;
              temppre[j - i] = preorder[j + 1] ;
       }
  }
  right = recons(tempin, temppre, noofnodes - i - 1) ;
```

Hidden page

Hidden page

Summary

- Tree is the most important linear data structure.
- A tree consists of a root node and a number of subtrees.
- There are different types of trees showing different types of properties.
- There are many terms associated with a binary tree like its right and left child, subtrees, brothers, etc.
- A binary tree can be implemented as a self-referential structure using pointers or arrays.
- The operations that can be performed on a binary tree are searching, insertions and deletions.
- A fundamental operation on a binary tree is traversal. Traversal of a binary tree implies visiting every node of that tree exactly once in some sequence. There are three standard types of tree traversal—preorder, inorder and postorder.
- We can reconstruct a binary tree from its inorder and preorder traversal.

Review Exercise

Multiple Choice Questions

1. The inorder traversal of some binary tree produced sequence DBEAFC, and the postorder traversal of the same tree produced the sequence DEBFCA, which of the following is a correct preorder traversal sequence:
   a. DBAECF
   b. ABEDFC
   c. ABDECF
   d. None of these
2. A balanced binary tree is a binary tree in which the heights of two subtrees of every node never differ by more than
   a. 2
   b. 1
   c. 3
   d. None of these
3. Which of the following statements is TRUE in view of a complete binary tree?
   a. The number of nodes at each level is 1 less than some power of 2.
   b. The outdegree of every node is exactly equal to 2 or 0.
   c. The total number of nodes in the tree is always some power of 2.
   d. None of these
4. Level of any node of a tree is
   a. Its distance from the root.
   b. Height of its left subtree minus height of its right subtree.
   c. Height of its right subtree minus height of its left subtree.
   d. None of the above

Hidden page

CHAPTER

9 Advanced Trees, Forests and Orchards

Key Features

- AVL Trees or Height-Balanced Trees
- Representation of AVL Trees
- Operations on AVL Trees
- Threaded Binary Trees
- Forests and Orchards
- Expression Trees

"Tree" is one of the most important non-linear data structures which is used for efficiently performing operations like insertion, deletion and searching.

However, while working with large volumes of data, construction of a well balanced tree for storing all data is not feasible. Thus, only useful data is stored as a tree and actual volume of data being used constantly changes through insertion of new data and deletion of existing data. Therefore, the advanced trees like AVL trees are used.

Due to the importance of threaded binary trees, it is worthwhile to develop non-recursive algorithms to manipulate them and to study the time and space requirements of these algorithms. We will find that, by changing the NULL link in a binary tree to special links called threads, it is possible to perform traversals, insertions, deletions without using either stacks or recursions.

The other kinds of trees which have been discussed in this chapter are forests, orchards and expression trees.

AVL TREES OR HEIGHT-BALANCED TREES

Definition of AVL Tree

An AVL tree is a binary search tree in which the difference of heights of left and right subtrees of any node is less than or equal to one. The technique of balancing the height of binary trees was developed by Adelson-Velskii and Landi. Therefore, these trees are known as **AVL Trees** or **Balanced Binary Trees**.

In a full binary search tree, the height of the left and right subtrees of any node are equal. Such trees are ideal for efficient searching because the height of such a tree with "n" nodes is $\mathbf{0(\log_2 n)}$. But when insertions and deletions are performed randomly on a binary search tree, the binary tree often becomes unbalanced.

An AVL tree can be defined as follows—Let **T** be a non-empty binary tree with $\mathbf{T_L}$ and $\mathbf{T_R}$ as its left and right subtrees. **T** is height balanced if:

- T_L and T_R are height balanced.
- $h_L - h_R \leq 1$ where h_L and h_R are the heights of T_L and T_R respectively.

Let us define more precisely the notation of a "balanced" tree. The height of a binary tree is the maximum level of its leaves (this is also sometimes known as the depth of tree). The height of null tree is defined as –1. Thus, a balanced binary tree or AVL tree is a binary tree in which the heights of two subtrees of every node never differ by more than 1.

The **balance factor** of a node in a binary tree can have value 1, –1 or 0 depending on whether the height of its left subtree is greater than, less than or equal to the height of its right subtree. The balance factor of each node is indicated in Fig. 9.1.

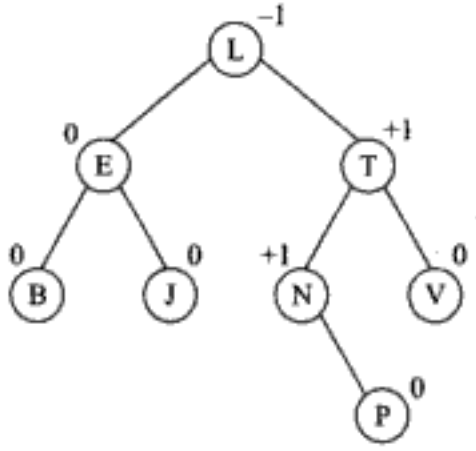
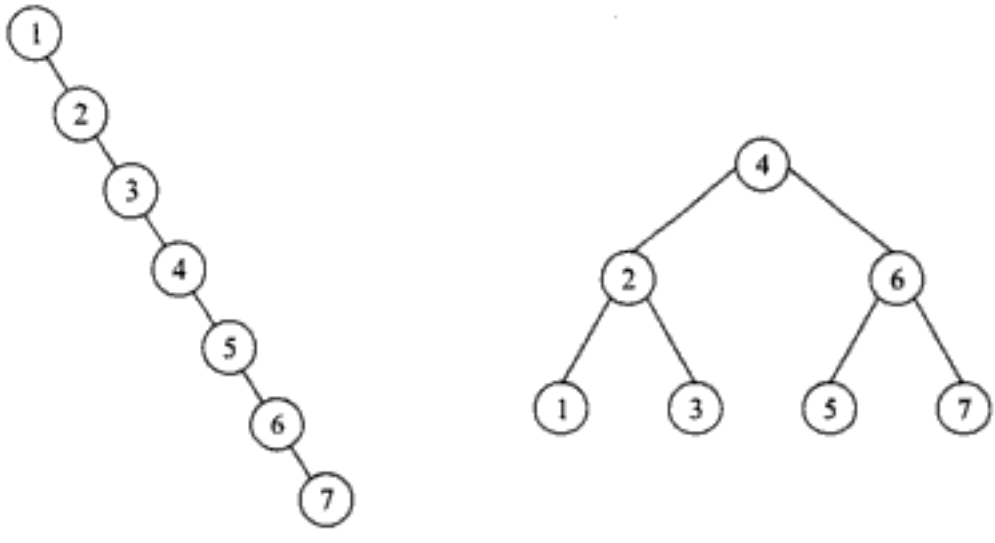


Fig. 9.1 *Balance Factors in Binary Tree*

Advantages of an AVL Tree

Since AVL trees are height balanced trees, operations like insertion and deletion have low time complexity. Let us consider an example. If we have the following tree with keys 1, 2, 3, 4, 5, 6, 7 then a binary tree would look like this:



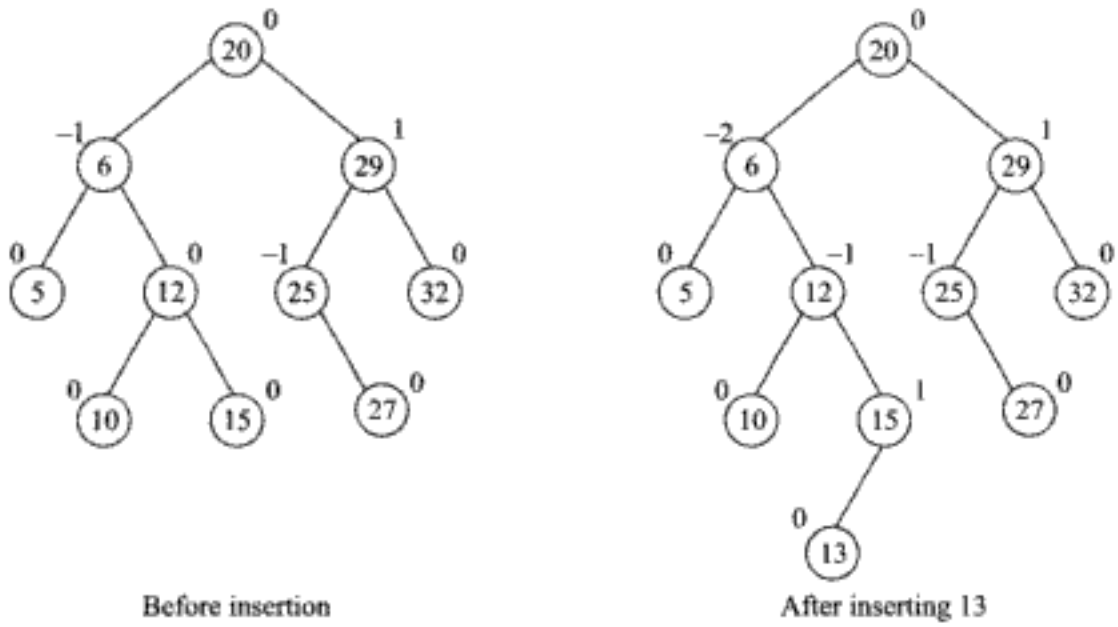
(a) Binary tree (b) AVL tree

Fig. 9.2

In order to insert a node with a key **Q** in the binary tree (a), the algorithm requires **7** comparisons, but if we insert the same key in AVL tree (b), the algorithm will just require **3** comparisons, which is less than half of the binary tree. Thus, we see that use of AVL trees will increase the efficiency of the programs.

Hidden page

Hidden page



Before insertion — After inserting 13

Fig. 9.6 *Addition of a New Node to a Leaf Node of Taller Subtree*

Consider the tree given in Fig. 9.7 with balance factors:

Here, the balance factor of **P** is **–1** and that of **Q** is **0**. $\mathbf{Lc_1}$ is the left child of **P** and $\mathbf{Lc_2}$ and $\mathbf{Rc_2}$ are the left and right children of the node **Q**. After insertion there are two cases that can make the tree an unbalanced tree. These are as follows:

(a) The new node is inserted as a child (left or right) of the leaf node of subtree Rc_2 (Fig. 9.8).

(b) The new node is inserted as a child (left or right) of the leaf node of structure Lc_2 (Fig. 9.9).

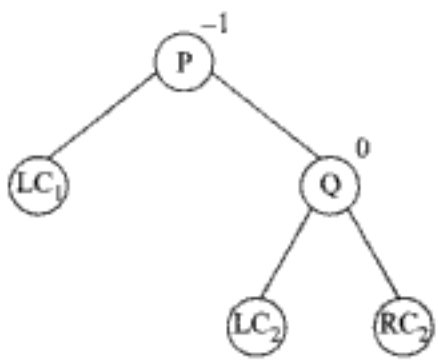


Fig. 9.7 *Tree with Balance Factor*

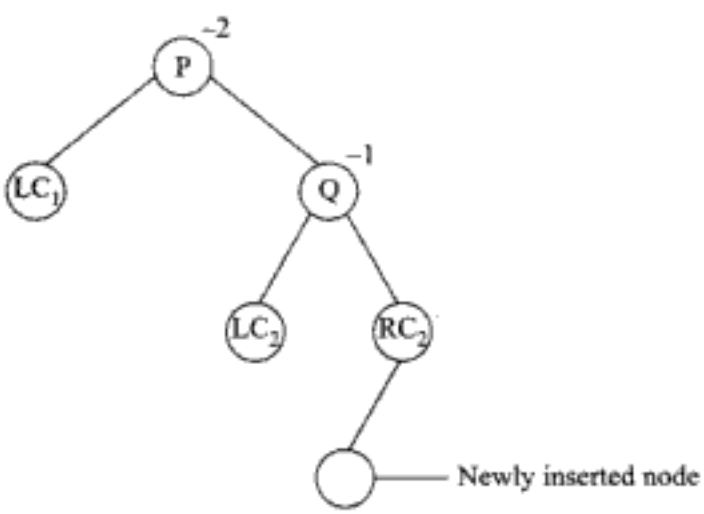


Fig. 9.8

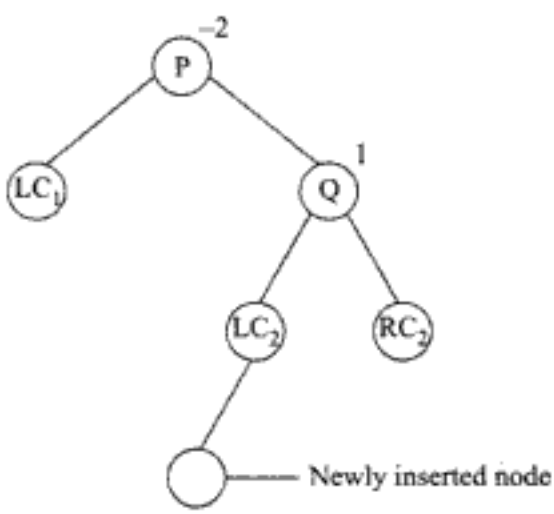


Fig. 9.9

To accomplish the balance in both these cases consider the following:

Case (a) Consider the tree given below again (Fig. 9.10).

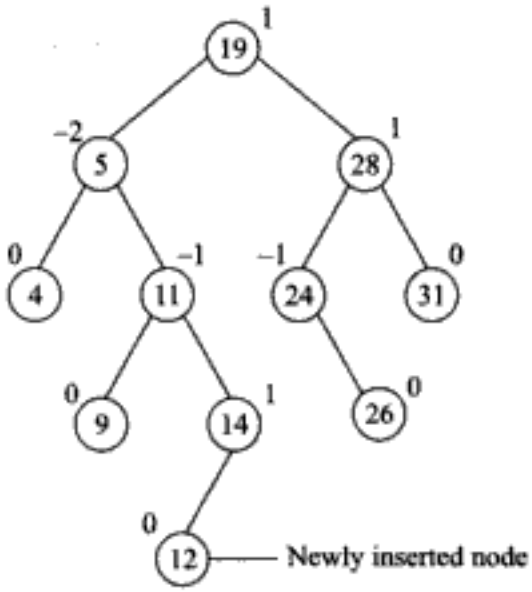


Fig. 9.10 *Insertion of a New Node*

As seen from the figure, on insertion of the new node, the balance factor of the node containing the data **5** violates the condition of an AVL tree. To rebalance the tree, we are required to make a **left-rotation** of a tree along the node containing the data **5** as the left child containing the data **11** and the node containing the data **9** as a right child of the node containing the data **5**. This is shown below considering the same tree taken for Case (a).

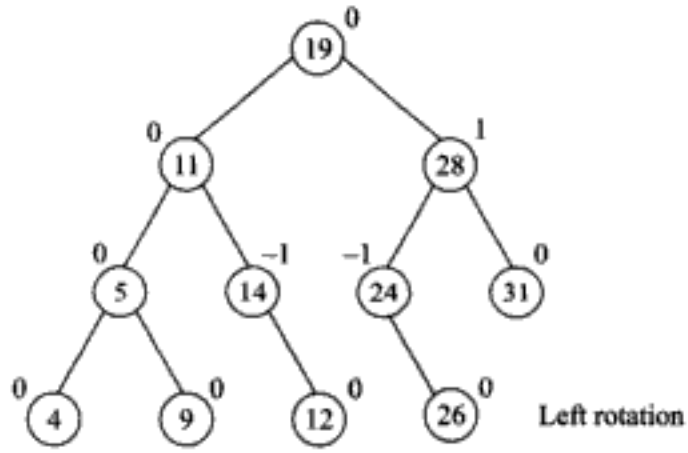


Fig. 9.11 *Rebalancing the Tree Using Left Rotation*

Case (b) Now suppose instead of **12,** we insert a node with value **10**. This node would get inserted as the right child of the node containing value **9**. After this the tree no longer remains a balanced tree as the balance factor of node containing value **5** breaks the rules of AVL tree as shown in Fig. 9.12:

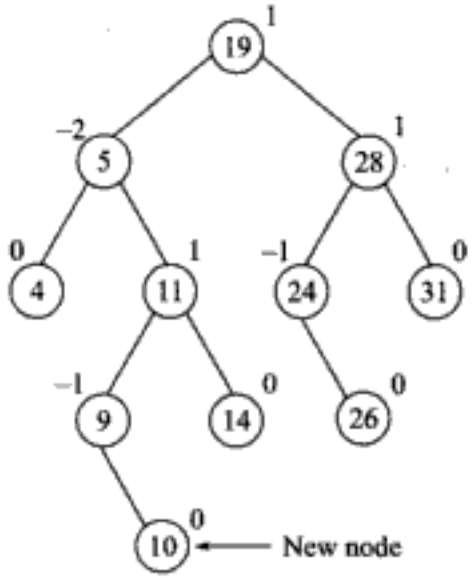


Fig. 9.12 *Unbalanced Tree after Addition of New Node*

In this case, to rebalance the tree, we are initially required to make a **right rotation** of the tree along the node containing **11**. Right rotation makes node **9** the right child of node **5**. Node **11** becomes the right child of node **9** and node **10** becomes left child of node **11**. This is shown in Fig. 9.13.

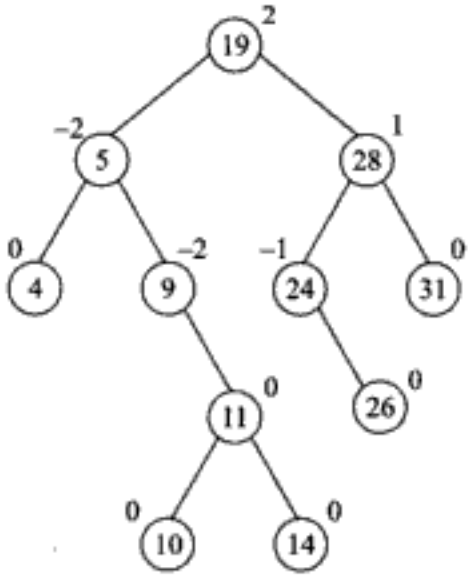


Fig. 9.13 *Rebalancing the Tree Again using Right Rotation*

But even after right rotation, the tree remains unbalanced and hence, it is rotated to left along the node **5**. As a result node **9** becomes the left child of node **19**. Node **5** becomes the left child of node **9**. Since there is no left child for node **9** the right child of node **5** is empty. Thus finally, tree becomes a balanced binary tree or an AVL tree. This procedure of rotating the tree, first to the right and then to the left is known as **double rotation**. The resultant tree is shown in Fig. 9.14.

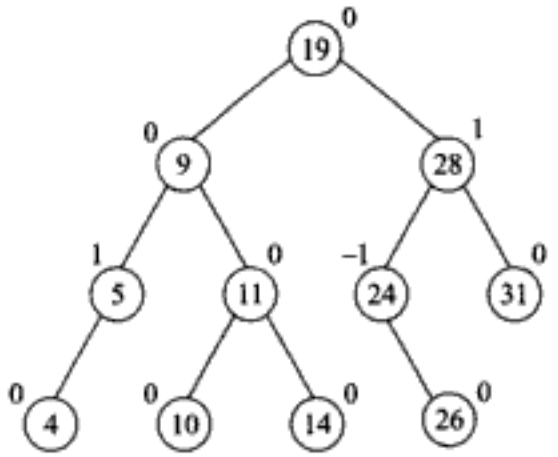


Fig. 9.14 *Balanced Tree after Double Rotation*

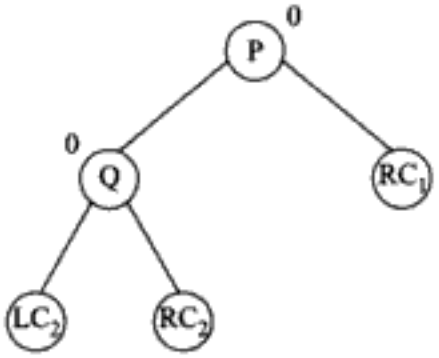


Fig. 9.15 *Unbalanced AVL Tree*

There are other two possibilities where an AVL tree becomes unbalanced due to insertion of new node. Consider Fig. 9.15.

Now again consider the insertion of new nodes.

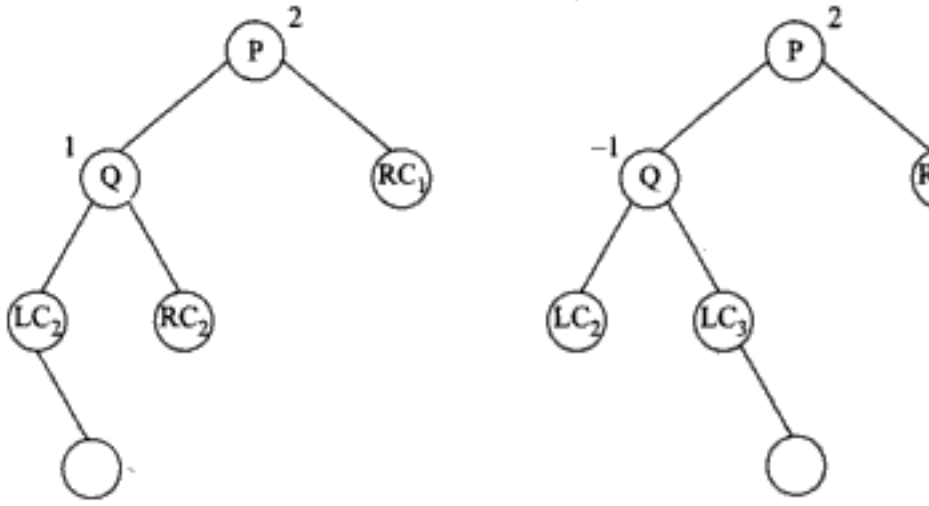


Fig. 9.16 *Insertion of New Nodes*

For the tree (i), to balance the tree only a right rotation is required whereas for the tree (ii) a double rotation is required—initially a left rotation and then a right rotation.

Deletion of a Node

The deletion of a node is exactly identical to the deletion of node in a binary search tree. Initially, the node to be deleted is searched. The conditions for deletion are same as a node to be deleted could be a leaf node, a node with one child or a node with two children.

Hidden page

Hidden page

Hidden page

```
                    case -1:
                            node1 = root -> right ;
                            if(node1 -> balfact == -1)
                            {
                                    printf("\nLeft rotation along %d.",
                                         root -> data) ;
                                    root -> right = node1 -> left ;
                                    node1 -> left = root ;
                                    root -> balfact = 0 ;
                                    root = node1 ;
                            }
                            else
                            {
                                    printf("\nDouble rotation. right along %d", node1
-> data) ;
                                    node2 = node1 -> left ;
                                    node1 -> left = node2 -> right ;
                                    node2 -> right = node1 ;
                                    printf(" then left along %d.\n",
root -> data) ;
                                    root -> right = node2 -> left ;
                                    node2 -> left = root ;
                                    if(node2 -> balfact == -1)
                                            root -> balfact = 1 ;
                                    else
                                            root -> balfact = 0 ;
                                    if(node2 -> balfact == 1)
                                            node1 -> balfact = -1 ;
                                    else
                                            node1 -> balfact = 0 ;
                                    root = node2 ;
                            }
                            root -> balfact = 0 ;
                            *h = FALSE ;
                }
            }
        }
        return(root ) ;
}
/* Deletes an item from the tree */
struct AVLNode * deldata(struct AVLNode *root, int data, int *h)
{
```

```
struct AVLNode *node ;
if(!root)
{
     printf("\nNo such data.") ;
     return(root) ;
}
else
{
     if(data < root -> data)
     {
            root -> left = deldata(root -> left, data, h) ;
            if(*h)
                   root = balr(root, h) ;
     }
     else
     {
            if(data > root -> data)
            {
                   root -> right = deldata(root -> right, data, h) ;
                   if(*h )
                          root = ball(root, h) ;
            }
            else
            {
                   node = root ;
                   if(node -> right == NULL)
                   {
                          root = node -> left ;
                          *h = TRUE ;
                          free(node) ;
                   }
                   else
                   {
                          if(node -> left == NULL)
                          {
                                 root = node -> right ;
                                 *h = TRUE ;
                                 free(node) ;
                          }
                          else
                          {
                                 node -> right = del(node -> right, node, h) ;
```

```
                                    if(*h)
                                        root = ball(root, h) ;
                                }
                            }
                        }
                    }
                }
            return(root) ;
        }
        struct AVLNode * del(struct AVLNode *succ, struct AVLNode *node, int *h)
        {
            struct AVLNode *temp = succ ;
            if(succ -> left != NULL)
            {
                succ -> left = del(succ -> left, node, h) ;
                if(*h)
                    succ = balr(succ, h) ;
            }
            else
            {
                temp = succ ;
                node -> data = succ -> data ;
                succ = succ -> right ;
                free(temp) ;
                *h = TRUE ;
            }
            return(succ) ;
        }
        /* Balances the tree, if right sub-tree is higher */
        struct AVLNode * balr(struct AVLNode *root, int *h)
        {
            struct AVLNode *node1, *node2 ;
            switch(root -> balfact)
            {
                case 1:
                    root -> balfact = 0 ;
                    break;

                case 0:
                    root -> balfact = -1 ;
                    *h  = FALSE ;
                    break;
```

Hidden page

```
  return(root) ;
}
/* Balances the tree, if left sub-tree is higher */
struct AVLNode * ball(struct AVLNode *root, int *h)
{
  struct AVLNode *node1, *node2 ;
  switch(root -> balfact)
  {
      case -1:
            root -> balfact = 0 ;
            break ;
      case 0:
            root -> balfact = 1 ;
            *h = FALSE ;
            break ;
      case 1:
            node1 = root -> left ;
            if(node1 -> balfact >= 0)
            {
                  printf("\nRight rotation along %d.", root -> data) ;
                  root -> left = node1 -> right ;
                  node1 -> right = root ;
                  if(node1 -> balfact == 0)
                  {
                        root -> balfact = 1 ;
                        node1 -> balfact = -1 ;
                        *h = FALSE ;
                  }
                  else
                  {
                        root -> balfact = node1 -> balfact = 0 ;
                  }
                  root = node1 ;
            }
            else
            {
                  printf("\nDouble rotation, left along %d", node1
                                    -> data) ;
                  node2 = node1 -> right ;
                  node1 -> right = node2 -> left ;
                  node2 -> left = node1 ;
                  printf(" then right along %d.\n", root -> data) ;
```

```
                root -> left = node2 -> right ;
                node2 -> right = root ;
                if(node2 -> balfact == 1)
                        root -> balfact = -1 ;
                else
                        root -> balfact = 0 ;
                if(node2-> balfact == -1)
                        node1 -> balfact = 1 ;
                else
                        node1 -> balfact = 0 ;
                root = node2 ;
                node2 -> balfact = 0 ;
            }
    }
    return(root) ;
}
/* Displays the tree in-order */
void display(struct AVLNode *root)
{
    if(root != NULL)
    {
        display(root -> left) ;
        printf("%d\t", root -> data) ;
        display(root -> right) ;
    }
}
/* Deletes the tree */
void deltree(struct AVLNode *root)
{
    if(root != NULL)
    {
        deltree(root -> left) ;
        deltree(root -> right) ;
    }
    free(root) ;
}
```

In the program, initially, eleven nodes are created and then two nodes are deleted. After deletion of the node, since the tree becomes unbalanced, it is balanced by performing appropriate rotations. The function like **insert()**, **deldata()** and **display()** are called from the **main()** to add, delete and display the nodes. The function **insert()** adds node to the tree. The functions like **ball()** and **balr()** are called by function **deldata()** to balance the tree.

The function **insert()** is used to add a new node to the tree. It receives three parameters—the first is the address of the root node of the tree or subtree to which the new node is to be added. The second is an integer that holds the data of the node that is to be added and third is the address of a variable that is used as a flag to check whether there is a need for balancing the tree after addition of a new node.

In the function **insert()**, it is checked whether root is NULL. If it is, then the tree is empty and the new node is going to be the first node. Now memory is allocated for the new node. Next data is stored in the data part, NULL in the left and right part of the root and a value 0 is assigned to its **balfact** field as at this point new node is going to be the leaf node.

If a tree is non-empty then the new node is added as a child of the leaf node. To determine whether the new node should be a left child or right child again it is checked whether the data of the new node is less than the data of the current node. If it is, then a recursive call is made to the function **insert()** by passing the address of the left subtree. If the left subtree is empty then the new node is made the left child of the current node. Then using **if(*h)** it is checked if there is a need for balancing.

If balancing needs to be done then a switch case is applied on the **balfact** of the current node. If **balfact** of the current node is **1** (left subtree of current node is higher) then it is checked whether the **balfact** of the leftchild of current node is **1**. If it is, then simply a right rotation is required along the current node, otherwise a double rotation is required. After rotation, **balfact** of current node is assigned a value **0** and a **FALSE** value is stored in the **flag** variable pointed to by **h**. If **balfact** of current node is 0, **balfact** is simply assigned a value 1. If **balfact** of current node is **–1**, then **balfact** is assigned a value **0** and a **FALSE** value is stored in the flag variable pointed to by **h**.

There is one more possibility—the data of the new node is greater than the data of the current node. If it is, then a recursive call is made to function **insert()** by passing the address of the right child of current node. Here too the switch-case is applied, if needed. Finally, the address of the current node is returned.

The function **deldata()** works like **insert()**. Here also the recursive call is made for either the left or right child depending upon the data to be deleted. If the data is found then its inorder successor is searched and a call is made to function **del()** which deletes the node. The functions **ball()** and **balr()** are called to balance the tree after deletion of the node.

The function **display()** is nothing but inorder traversal of the tree.

THREADED BINARY TREES

When a binary tree is represented using pointers, then pointers to empty subtrees are set to **NULL**, that is, 'left' pointer of the node whose left child is an empty subtree is normally set to **NULL**. Similarly, the 'right' pointer of a node whose right child is an empty subtree is set to **NULL**. Thus, a large number of pointers are set to **NULL**.

Assume that the 'left' pointer of a node '**n**' is set to **NULL** as the left child of '**n**' is an empty subtree. Then the 'left' pointer of '**n**' can be said to point to the inorder **predecessor** of **n**. Similarly, if the 'right' child of a node '**m**' is empty then the right pointer of **m** can be said to point to the inorder **successor** of '**n**'.

In Fig. 9.17, links with arrow heads indicate links leading to inorder predecessor or inorder successor while lines denote the usual links in a binary tree.

The links with arrows and the normal links indicate different relationship between nodes and the links. It must be understood whether "left" or "right" link of node "**n**" is leading to its children or to the inorder predecessor or inorder successor of "**n**".

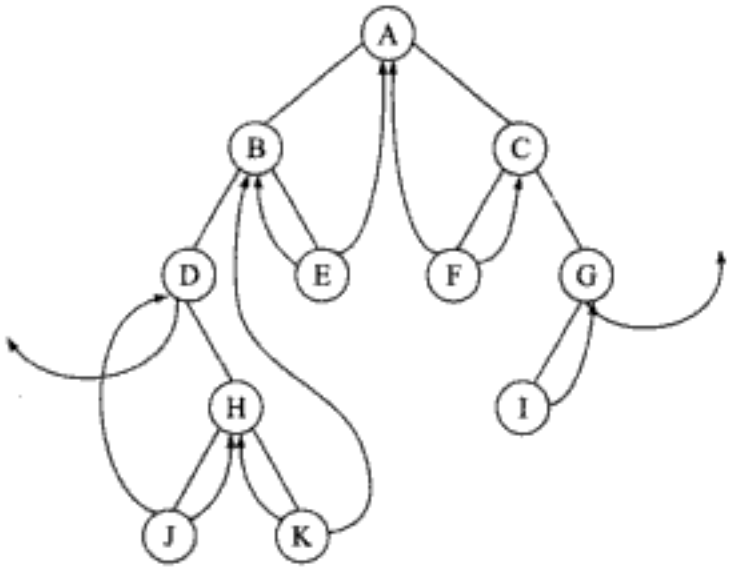


Fig. 9.17 *Threaded Binary Tree*

Two flags '**leftflag**' and '**rightflag**' are used per node to indicate the type of its '**left**' and '**right**' links.

If '**leftflag**' of a node is **0** then its '**left**' link leads to the left subtree of **n** otherwise to the inorder predecessor of '**n**'.

Similarly, if '**rightflag**' of a node '**n**' is **0** then right link leads to the right subtree of **n** otherwise the right link leads to inorder successor of '**n**'.

The links are used as threads only when they would have pointed to empty subtree in a non-threaded binary tree.

Hence the structure of a node in such a tree contains two more fields:

```
struct bthreadNode
{
    int rightflag, leftflag;
    int data;
    struct bthreadNode * left, right;
};
```

Now, consider the tree in Fig. 9.17. The lines with arrows denote threads. In a non-threaded binary trees replace these lines with arrows with links to NULL. The inorder traversal of this tree is:

D, J, H, K, B, E, A, F, C, I, G.

In the tree given in Fig. 9.17, '**J**' does not have a child, the left and right pointers of this node are used as threads. The left is used to point **D** which is inorder predecessor of this node **J** and the right is used to point **H** which is inorder successor. Consider **D** and **G**—the left and right links keep dangling because they do not have any inorder predecessor or successor.

Another approach to get rid of such problems is to introduce a header node. The left and right pointers of the header node are treated as normal links and are initialized to point to header node itself.

Hidden page

Hidden page

```
        /*SHOW TREE*/
        showtree(p,38,1);
        gotoxy(12,23);printf("\n Press any key...");
        getch();
        break;
    case 6:
        /*EXIT*/
        exit(1);
    default:
        printf("\n Wrong choice! Press any key...");
        getch();
    }
    nosound();
    }while(ch!=7);
}
//INSERT NODE
void addnode(int v)
{
    struct btree *r,*q;
    q=p;
    r=malloc(sizeof(struct btree));
    r->info=v;
    r->rt=1;
    r->lt=1;
    if(p==NULL)
    {
        p=r;
        r->left=head;
        r->right=head;
        return;
    }
    while(q!=head)
    {
        if(q->info>v && q->lt==0)
        q=q->left;
        else if(q->info<v && q->rt==0)
        q=q->right;
        else break;
    }
    if(q->info>v)
    {
        q->lt=0;
```

Hidden page

```
        showtree(q->left,(col-4),(row+3));
        }
    }
    if(q->rt==0)
    {
        gotoxy((col+2),(row+1));
        textcolor(14);
        delay(100);
        printf("\\");
        if((row%2)==0)
        {
        gotoxy((col+3),(row+2));
        textcolor(14);
        delay(100);
        printf("|");
        showtree(q->right,(col+3),(row+3));
        }
        else
        {
        gotoxy((col+4),(row+2));
        delay(100);
        textcolor(14);
        printf("\\");
        showtree(q->right,(col+4),(row+3));
        }
    }
    return;
}
//INORDER
void inorder(struct btree *q)
{
    int flag=0;
    while(q!=head)
    {
        if(flag==0)
        {
        while(q->lt==0)
        q=q->left;
        }
        delay(350);
        printf(" %d\t",q->info);
        flag=q->rt;
        q=q->right;
```

```
    }
    getch();
}
//PREORDER
void preorder(struct btree *q)
{
    int flag=0;
    while(q!=head)
    {
        while(flag==0)
        {
        delay(350);
        printf(" %d\t",q->info);
        if(q->lt==0)
        q=q->left;
        else
        break;
        }
        flag=q->rt;
        q=q->right;
    }
    getch();
}
//DELETE NODE
void delnode(int dd)
{
    struct btree *q,*cur,*par;
    int flag=0;
    par=q=p;
    while(q!=head)
    {
        if(q->info>dd)
        {
        par=q;
        q=q->left;
        continue;
        }
        if(q->info<dd)
        {
        par=q;
        q=q->right;
        continue;
```

```
        }
        if(q->info==dd)
        {
        flag=1;
        break;
        }
}
if(flag==0)
{
printf(" Value not found, press any key...");
getch();
return;
}
cur=q;
    //Case when current node is leaf node
if(cur->lt==1 && cur->rt==1)
{
        if(par->info<dd)
        {
        par->right=cur->right;
        par->rt=cur->rt;
        }
        else
        {
        par->left=cur->left;
        par->lt=cur->lt;
        }
        free(cur);
        return;
}
    //Case when current node has no left child
if(cur->lt==1 && cur->rt==0)
{
        if(cur==p)
        {
        p=cur->right;
        if(p->lt==1)
        p->left=cur->left;
        free(cur);
        return;
        }
        if(par->info<dd)
        {
```

```
        par->right=cur->right;
        par=par->right;
        }
        else
        {
        par->left=cur->right;
        par=par->left;
        }
while(par->lt==0)
par=par->left;
par->left=cur->left;
free(cur);
return;
}
    //Case when current node has no right child
if(cur->rt==1 && cur->lt==0)
{
        if(cur==p)
        {
        p=cur->left;
        if(p->rt==1)
        p->right=cur->right;
        free(cur);
        return;
        }
        if(par->info<dd)
        {
        par->right=cur->left;
        par=par->right;
        }
        else
        {
        par->left=cur->left;
        par=par->left;
        }
while(par->rt==0)
par=par->right;
par->right=cur->right;
free(cur);
return;
}
    //Case when current node has both child
```

Hidden page

In our work so far with binary trees we have benefited from using recursion, and for other classes of trees we shall continue to do so. Employing recursion means reducing a problem to a smaller one. Hence, we should see what happens if we take a rooted tree, an ordered tree and strip off the root. What is then left is (if not empty) a set of rooted trees or an ordered set of ordered trees respectively.

The standard term for an arbitrary set of trees is **forest**, but when we use this term we generally assume that the trees are rooted. The phrase **ordered forest** is sometimes used for an ordered set of ordered trees, but we shall adopt the equally descriptive term **orchard**.

Note that we can only obtain a forest or an orchard by removing the root from a rooted tree, by starting with a forest or an orchard, attaching a new vertex at the top, and adding branches from the new vertex (which will be the root) to the roots of all trees in forest or the orchard.

It is possible that a binary tree be empty; that is, it may have no vertices. It is also possible that a forest or an orchard be empty; that is, it contains no trees. It is however, not possible that a rooted or an ordered tree be empty, since it is guaranteed to contain a root at least. If we wish to start building trees and forests, we can note that the tree with only one vertex is obtained by attaching a new root to an empty forest. Once we have this tree, we can make a forest consisting of as many one-vertex trees as we wish and attach a new root to build all rooted trees of height 1. In this way we can continue to construct all the rooted trees in turn in accordance with the following mutually recursive definitions:

- A rooted tree consists of a single vertex **v**, called the root of the tree, together with a forest **F**, whose those trees are called the subtrees of the root.
- A forest **F** is (possibly empty) set of rooted trees.
- An ordered tree **T** consists of a single vertex **V** called the root of the tree, together with an orchard **O** whose trees are called subtrees of root **V**. We may denote the ordered tree with the ordered pair

$$\mathbf{T = \{V, O\}}$$

- An orchard **O** is either empty set **f** or consists of an ordered tree **T**, called the first tree of the orchard, together with another orchard **O′** (which contains the remaining trees of the orchard). We may denote orchard with the ordered pair:

$$\mathbf{O = (T, O')}$$

- The theorem between orchard and binary trees is as follows:
  Let **S** be any finite set of vertices. There is a one-to-one correspondence **f** from the set of orchards whose set of vertices is **S** to be the set of binary trees whose set of vertices is **S**.

Rotations

Rotation is the transformation from orchard to binary tree. In a binary tree $\mathbf{[v, f(O_1), f(O_2)]}$ of the left link from **V** goes to the root of the binary tree $\mathbf{f(O_1)}$, which in fact was the first child of **V** in the ordered tree $\mathbf{\{V, O_1\}}$. The right link from **V** goes to the vertex that was formerly the root of the next ordered tree to the right, that is, '**Left link**' in the binary tree corresponds to '**first child**' in an ordered tree and '**right link**' corresponds to '**next sibling**'. In geometrical terms, the transformation reduces to the following rules:

1. Draw the orchard so that the first child of each vertex is immediately below the vertex, rather than centering the children below the vertex.

2. Draw a vertical link from each vertex to its first child and draw a horizontal link from each vertex to its next sibling.
3. Remove the remaining original links.
4. Rotate the diagram 45 degrees clockwise so that the vertical links appear as left links and the horizontal links as right links.

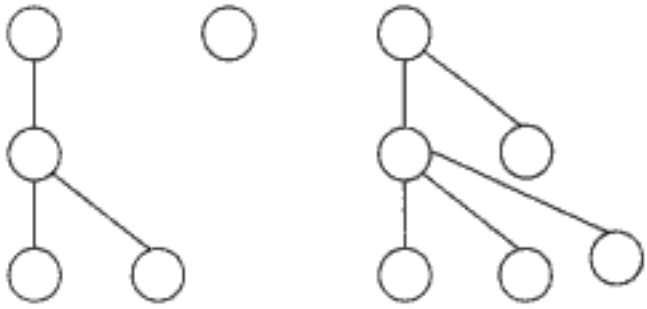


Fig. 9.18 *Orchard*

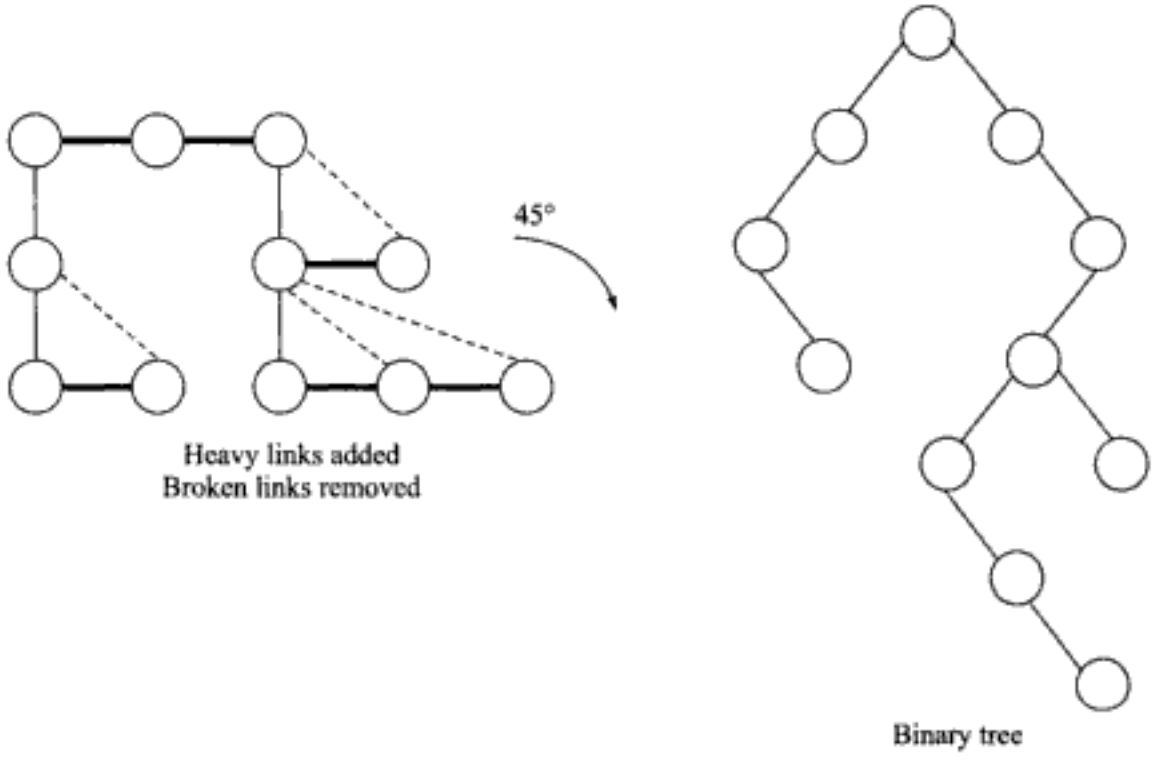


Fig. 9.19 *Conversion from Orchard to Binary Tree*

EXPRESSION TREES

An ordered tree may be used to represent a general expression in much the same way that a binary tree may be used to represent a binary expression. Since a node may have any number of sons, non leaf nodes need not represent only binary operators but can represent operators with any number of operands. Figure 9.20 illustrates expressions and their tree representations;

$$(((x - y) + 2) / ((u - v) * w) + t)$$

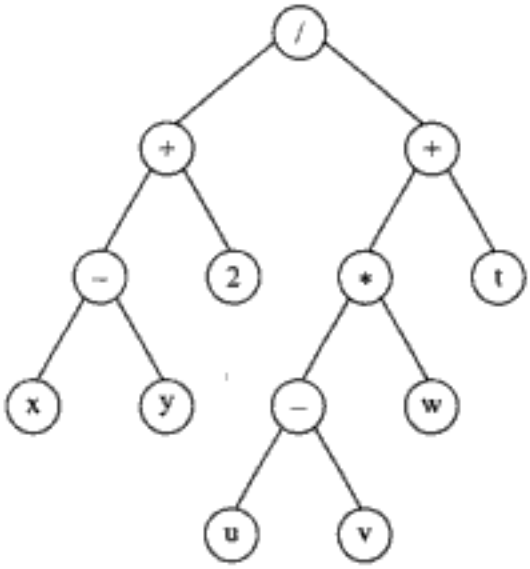


Fig. 9.20 *Expression Tree*

Now consider the following expression:

$$E = v + u / x * y - z$$

Before constructing the binary tree for expression **E**, first we consider the precedence of the operators that lie in the expression. The precedence of the operators that lie in the expression **E** is as follows:

(i) / and * have highest precedence.
(ii) + and – have lowest precedence

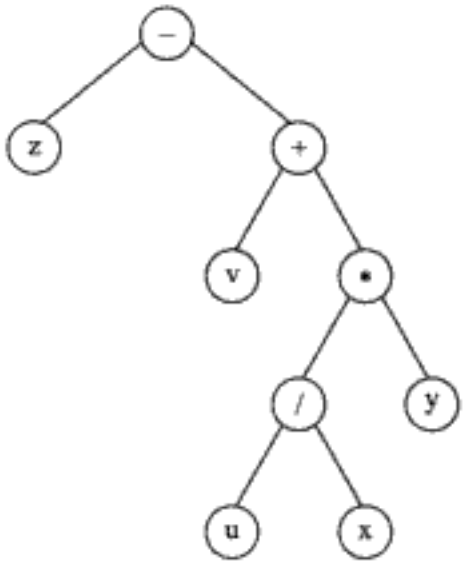


Fig. 9.21 *Operator Precedence*

Now the construction of the binary tree may be done in following steps:

(i) In the expression the operands of / are **u** and **x**.

Hidden page

Traversing the binary tree of Fig. 9.22(b) is the same as traversing the ordered tree of Fig. 9.22(a). However, a tree as such in Fig. 9.22(a) may be considered as a binary tree in its own right, rather than as an ordered tree. Thus, it is possible to perform a binary traversal (rather than a general traversal) directly on the tree in Fig. 9.22(a). Opposite Fig. 9.22(a) and (b) are the binary traversals of the trees.

Note that the preorder traversals of both the binary trees are the same. Thus, if a preorder traversal on a binary tree representing a binary expression yields the prefix of the expression, that traversal on an ordered tree representing a general expression that happens to have only binary operators yields prefix as well. However, postorder traversal of two binary trees are not the same. Instead inorder binary traversal of the second (which is the same as the inorder general traversal of the first if it is considered as an ordered tree) is same as the postorder binary traversal of the first. Thus, an inorder general traversal of an ordered tree representing a binary expression is equivalent to the postorder binary traversal of the binary tree representing the expression which yields postfix.

Summary

- AVL trees are height balanced trees where the difference between heights of left and right subtrees rooted at any node cannot be larger than one.
- The fundamental operations, namely searching, insertion and deletion can be performed more efficiently on AVL trees. The complexities of these operations on an AVL tree having '**n**' node is $\mathbf{0(\log_n)}$.
- In an AVL tree the only case that causes difficulty when the new node is added to a subtree of the root that is strictly taller than other subtree then the height is increased. This would cause one subtree to have height 2 more than other, whereas the AVL condition is that the height difference is never more than 1.
- When an AVL tree is right high the action needed in this case is called left rotation. On the other hand when the tree is left high, the right rotation is performed. In some cases, the tree is needed to rotate twice, then the condition is called double rotation.
- In a binary tree, many nodes have one child or no children. The pointers for the empty children for these nodes are set to NULL. A more effective utilization of these pointers is possible if 'NULL' left pointers are set to point to the inorder predecessor of that node and 'NULL' right pointers are set to point to the inorder successor of that node. These pointers, so introduced, are called threads. Threads help in writing non-recursive versions of inorder, preorder and postorder traversal.
- An ordered tree may be used to represent a general expression.
- A forest F is a set of rooted trees.
- An orchard O is either the empty set ϕ, or consists of an ordered tree T, called the first tree of the orchard, together with another orchard O.

Hidden page

Hidden page

CHAPTER

10 Multiway Trees

Key Features

- 2-3 Trees
- Multiway Search Trees
- B^+ Trees
- Heaps
- Construction of a Heap

In a binary search tree each node contains a single key and points to two subtrees. One of these subtrees contains all the keys in the tree rooted at node that are less than the key in node and other subtree contains all the keys in the tree rooted at node greater than (or equal to) the key in node.

This concept is extended to general search tree in which each node contains one or more keys. A balanced order n-multiway search tree in which each non-root node contains at least (n–1)/2 keys is called **B-tree**. A **2-3 tree** is the most elementary B tree, with two or three elements per node. **Heap** is another data structure used for implementing a priority queue. This chapter gives a detailed description of Multiway trees.

2-3 TREES

The objective of various search trees is to organize data in such a way, that any sequence of search, insertion and deletion operations can be performed in an efficient manner. It is already seen that AVL trees serve this purpose by ensuring that the trees are height-balanced. But the algorithms for performing these operations are very difficult to understand and implement.

A different technique is employed to balance trees in another data structure known as **2-3** trees. A **2-3 tree** is a general tree which satisfies the following criteria:

- Each non-leaf node in a 2-3 tree must have exactly two or three non-empty children, each of which is a 2-3 tree in turn.
- All leaf nodes must have the same level.
- The nodes which have two children contain exactly one key value '**K**'. Key values of all nodes in the left child of these nodes are less than '**K**' and the key values of all nodes in the right child of these nodes are greater than '**K**'.

- Nodes having three children contain two key values '**K**' and '$\mathbf{K_1}$', $\mathbf{K < K_1}$. The children are ordered—left, middle and right. Key values of all nodes in the middle subtree are more than '**K**' but less than '$\mathbf{K_1}$' and key values of all nodes in the right subtree are more than '$\mathbf{K_1}$'.

An example of a 2-3 tree is shown in Fig. 10.1. Each non-leaf internal node in a 2-3 tree has 2 or 3 children. If all non-leaf nodes in a 2-3 tree of height '**h**' have three children then the total number of elements would be $(\mathbf{3^{h+1} - 1})$. Similarly, if each node of 2-3 tree has two children then the tree has $(\mathbf{2^{h+1} - 1})$ elements. Therefore, a 2-3 tree with "**n**" elements having height "**h**" always satisfies the following condition:

$$\mathbf{2^{h+1} - 1 \leq n \leq 3^{h+1} - 1.}$$

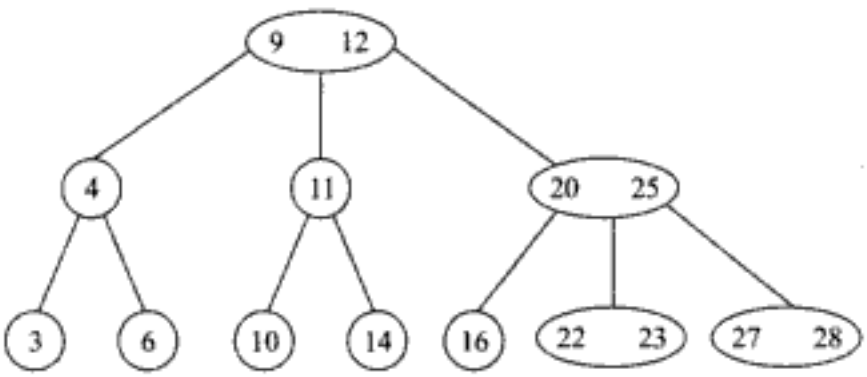


Fig. 10.1 *A 2-3 Tree*

Searching

To search for a key value '**K**' in a 2-3 tree represented by **T**, start searching from the root node of '**T**'. To search in a node with two children, compare '**K**' with **key[0]**. If '**K**' is less than **key[0]**, continue the search in subtree rooted at **child[0]**; otherwise continue search in subtree rooted at **child[1]**.

While searching in a node with three children, if **K** is less than **key[0]**, search in the tree rooted at **child[0]**. Continue the search in tree rooted at **child[1]** if **key[0] < K < key [1]**; otherwise search the tree rooted at **child[2]**.

If '**K**' is equal to **key[0]** in a node having two children then the search terminates successfully and a pointer to a node is returned. Similarly, if '**K**' is equal to **key[0]** or **key[1]** in a node with three children search process terminates successfully. If '**K**' does not exist in **T** already, then search process will terminate unsuccessfully at a leaf node (with two or three children).

Insertion and Deletion

The process of inserting a key '**K**' in a 2-3 tree begins by searching for '**K**' in '**T**'. The search will become unsuccessful at a leaf node '**x**' where '**K**' is said to be added. If '**x**' has two children add '**K**' in the array '**key**' of **x** and increase the number of children of '**x**'. The insertion procedure is over. If '**x**' is a leaf node with two key values **key[0]** and **key[1]** then '**x**' has to be **split** into two nodes '**y**' and '**z**'. '**y**' will contain the key with the smallest value among '**K**', **key[0]** and **key[1]**; **z** will contain the largest value among **K**, **key[0]** and **key[1]**.

Let the middle value among **K**, **k[0]** and **k[1]** is **m**. '**m**' is added to the parent node '**p**' of **x**. If **p** has two children then its number of children would increase with the addition of one key value '**m**'. If '**x**'

is **child[i]** of '**p**' before insertion then '**y**' will become **child[i]** of '**p**'; '**m**' becomes **key[i]** of **p** and **z** would becomes **child[i+1]** of '**p**'. These manipulations would complete the insertion procedure in this case. If '**p**' has three children then **p** will be split (in the same way x was split) and the process continues upwards along the path from '**x**' to the root. In case '**x**' is the root then a new node with two children is created which becomes the new root with '**y**' as its left child and '**z**' as its right child.

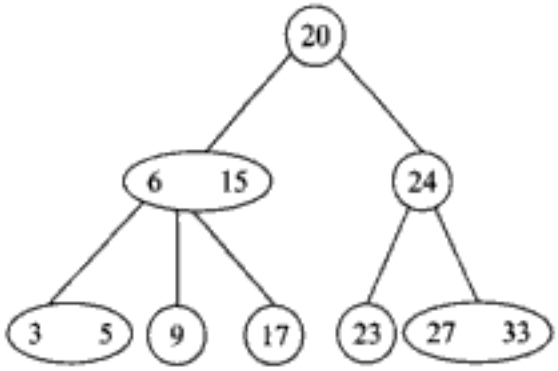


Fig. 10.2 *Original Tree*

Consider the tree shown in Fig. 10.2 and suppose we want to insert the value 2. To insert the value, first we need to search the appropriate position for the value. The searching process will terminate at the leaf node that contains the data 3 and 5. The value that is to be inserted, i.e. 2 is added to this node (Fig. 10.3).

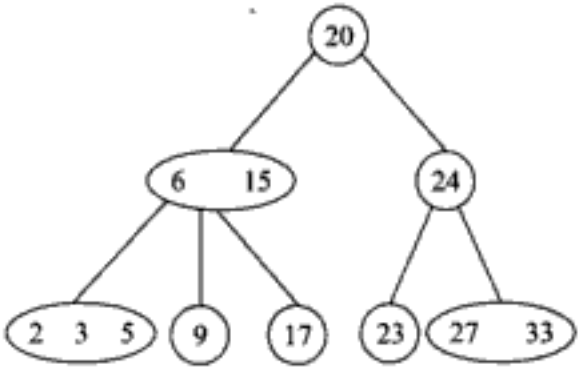


Fig. 10.3 *Tree after Adding 2*

Adding 2 to the node that already contains 3 and 5 violates the definition of 2-3 trees. To again make it a 2-3 tree, the **median** of the three values 2, 3 and 5 is taken and that value is shifted to the parent of this node. In our case median value happens to be 3 which is moved up to the parent node. As a result the parent node now contains 3, 6 and 15. Also the node containing 2 and 5 is now split into two different nodes containing values 2 and 5. These two nodes are then attached as first and second child of parent node containing the data 3, 6 and 15 as shown in Fig. 10.4.

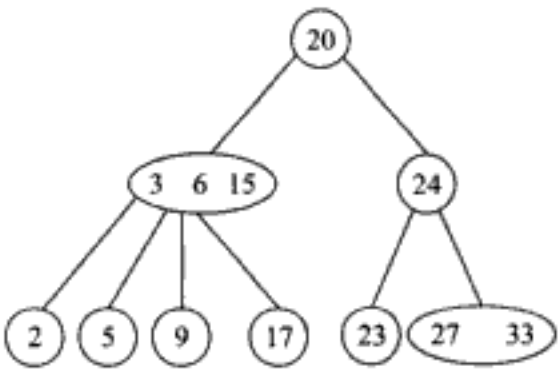


Fig. 10.4 *Nodes Attached as First and Second Child of Parent Node*

Now the node containing 3, 6 and 15 violates the rule of 2-3 tree. So again the same process of finding the median of the three values and shifting that value to parent node is repeated. In our case, 6 is shifted to its parent node that contains a value 20. This time the node that contains value 3 and 15 is split in such a way that 3 forms the left subtree and 15 forms the middle subtree of the root node that contains the values 6 and 20.

Also, the old child that contains a value 5 and 9 now becomes the right and left child of the nodes that contain the values 3 and 15 respectively. The final 2-3 tree is shown in Fig. 10.5.

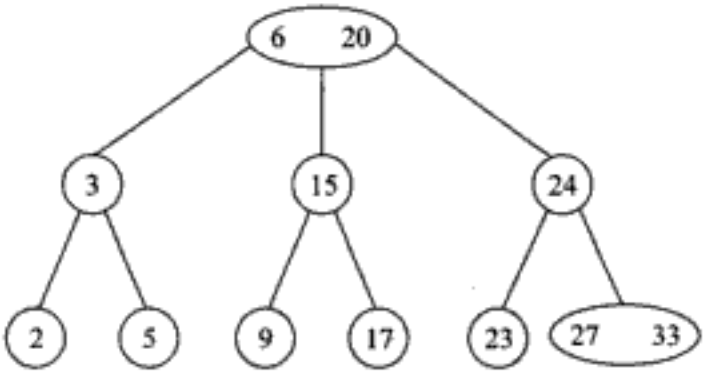


Fig. 10.5 *The Final 2-3 Tree*

The process of **Deletion** of a value from a 2-3 tree is exactly reverse of insertion. In case of insertion the node where the data is to be inserted is split if it contains maximum number of values (i.e. two values). But in case of deletion, two nodes are merged if the node of the value to be deleted contains minimum number of values (i.e. only one value).

Consider the tree shown below.

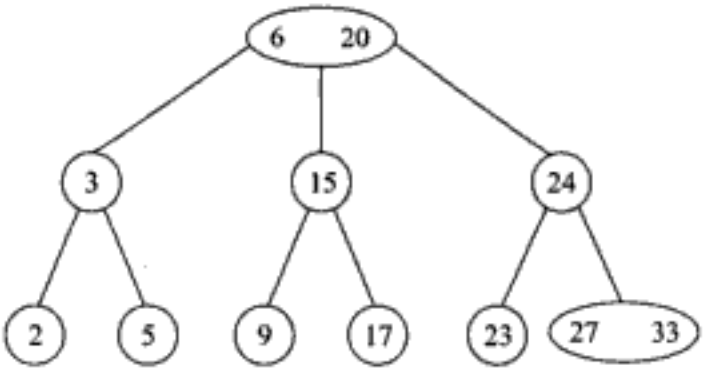


Fig. 10.6 *Tree before Deletion of a Value*

Suppose the node that contains the value 17 is to be deleted. The parent holds the value 6 and 15 and 15 is predecessor of 17. Hence, 17 is replaced by 15 and then is merged with its sibling, i.e. with 9. Then the node that contains 9 and 15 is made the right child of the node that initially contained the values 6 and 15 and now contains only a single value 6 (as 15 is shifted to the place of deleted node) as shown in Fig. 10.7.

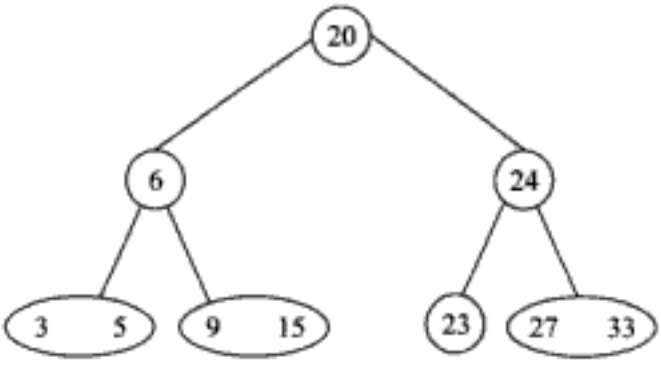


Fig. 10.7 *Tree after Deletion of a Value*

MULTIWAY SEARCH TREES

Binary search trees generalize directly to multiway search trees in which, for some integer **m**, called the order of the tree, each node has at most **m** children. If **K ≤ m** is the number of children, then nodes contain exactly **K-1** keys which partition all the keys into key subsets. If some of these subsets are empty, then the corresponding children in the tree are empty.

Balanced Multiway Trees

Our goal is to devise multiway search tree that will minimize file accesses. Hence, we wish to make the height of the tree as small as possible. We can accomplish this by ensuring first that no empty subtrees appear above the leaves (so the division of keys into subsets is as efficient as possible); second that all leaves be on the same level (so that searches will be guaranteed to terminate with about the same number of accesses) and third, that every node (except the leaves) has at least some minimal number of children. We shall require that each node (except the leaves) has at least as many children as the maximum possible. The condition leads to the following definition.

A **B-tree** of order **m** is an **m-way** tree in which

- All leaves are on same level.
- All internal nodes except the root have at most **m** (non-empty) children, and at least **[m/2]** non-empty children.
- The number of keys in each internal node is one less than the number of its children, and these keys partition the keys in the children in the fashion of a search tree.
- The root has at most **m** children, but may have as few as 2 if it is not a leaf, or none if the tree consists of the root alone.

Insertion in a B-Tree

The B-trees are not allowed to grow at their leaves; instead they are forced to grow at the root. The general method of insertion is as follows:

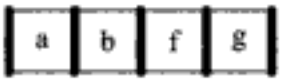
- First a search is made to see if the new key is in the tree. This search will terminate in failure at a leaf.
- The new key is added to the leaf node. If the node was not previously full, then insertion is finished.
- When a key is added to a full node, then the nodes split into two nodes on the same level except that the median key is not put into either of the new nodes but instead sent up the tree to be inserted into the parent node.
- When a search is later made through a tree, a comparison with the median key will serve to direct the search into proper subtree.
- When a key is added to full root, then the root splits into two and the median key sent upward becomes a new root.

The process can be understood with the example given below. The following keys are inserted in a B-tree of order of 5.

a g f b k d h m j e s i r x c l n t u p

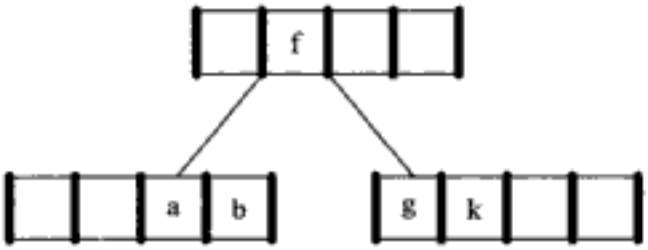
The first four keys will be inserted into one node, as shown below:

1. **a g f b :**



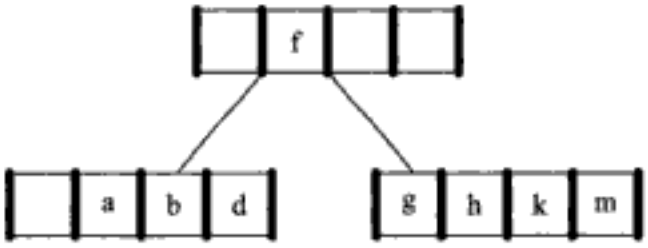
They are sorted into proper order as they are inserted.

2. **k :**



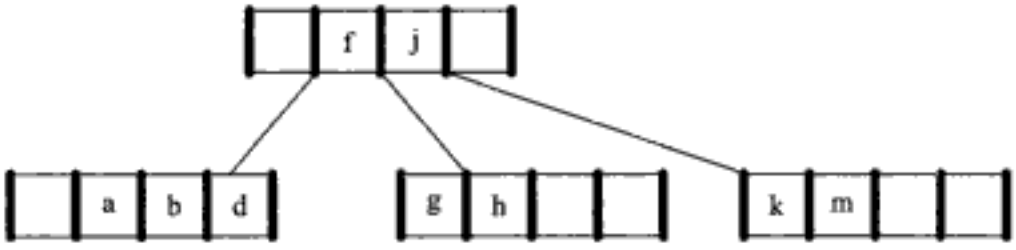
Now the insertion of **k** causes the node to split into two and the median key **f** moves up to enter a new node.

3. **d, h m :**



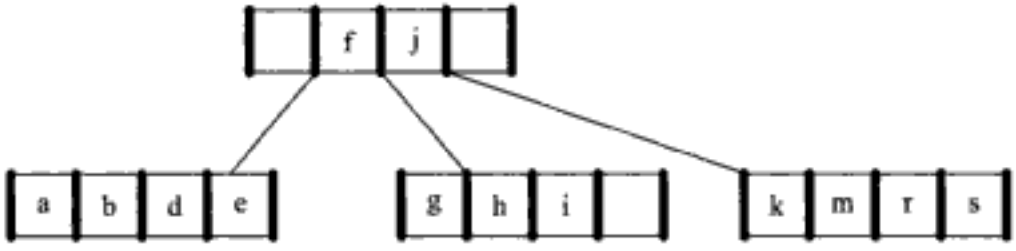
Now since the split nodes are now only half full, the next three keys can be inserted without difficulty. However, these simple insertions can require rearrangement of keys within a node.

4. **j :**

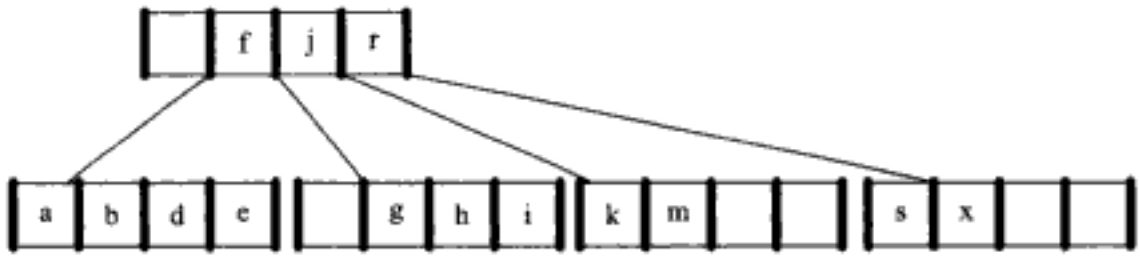


The next insertion **j** again splits a node and this time it is **j** itself that is the median key and therefore moves up to join **f** in the root.

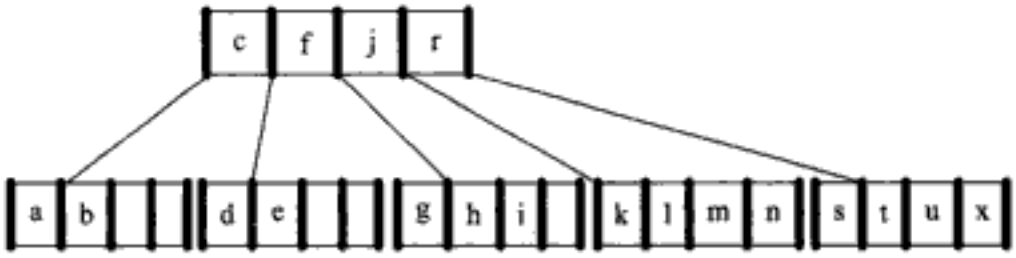
5. **e s i r :**



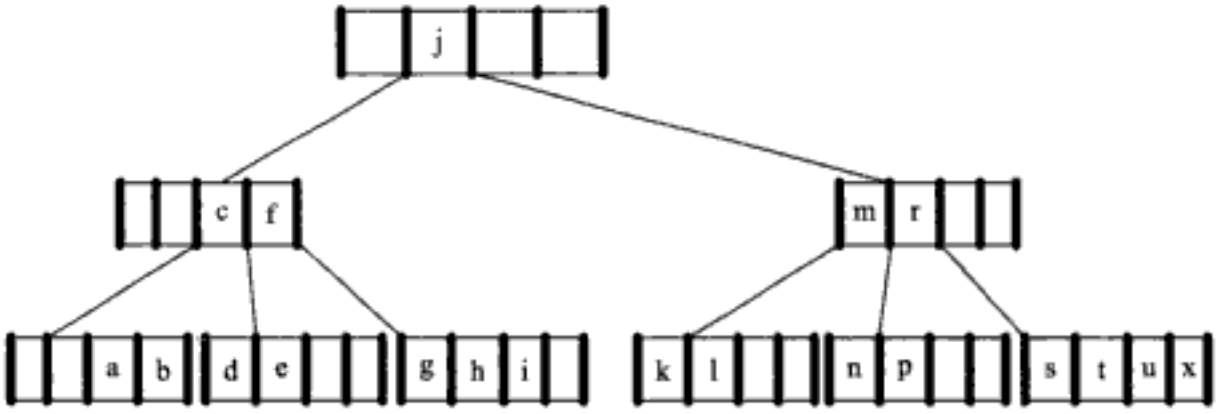
6. **x :**



7. **c l n t u :**



8. **p :**



The insertion **p** splits the node originally containing **k, l, m, n** sending median key **m** upward into the node containing **e, f, j, r** which is however already full. Hence, this node in turn splits and a new node containing **j** is created.

Deletion in a B-Tree

During insertion, the new key always goes first into leaf. If the key that is to be deleted is not the leaf then its immediate **predecessor** or **successor** is promoted into the position of the deleted key. The natural order of keys are guaranteed to be in the leaf nodes. Hence, we can promote the immediate predecessor or successor into the position occupied by the deleted key and delete the key from leaf.

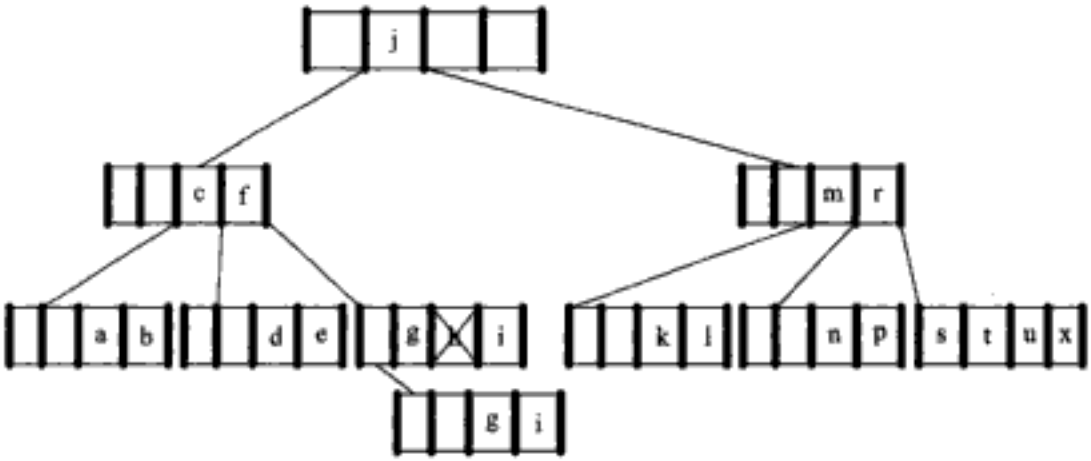
If the leaf contains more than minimum number of keys, then one can be deleted with no further action. If the leaf contains the minimum number, then we first look at the two leaves (or in case of a node on the outside, one leaf) that are immediately adjacent and children of same node. If one of these has more than the minimum number of keys, then one of them can be moved into the parent node and the key from the parent is moved into the leaf where the deletion is occurring.

If finally the adjacent leaf has only the minimum number of keys, then the two leaves and the median key from the parent can all be combined as one new leaf which will contain no more than the maximum

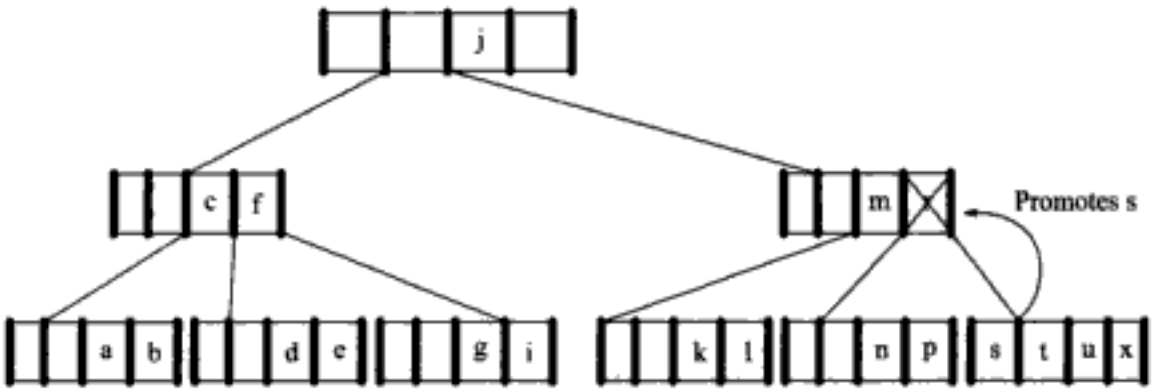
number of keys allowed. If this step leaves parent node with too few keys, then the process propagates upwards. In this case, the last key is removed from the root, and then the height of the tree decreases.

The process of deletion in the B-tree of order 5 is shown in the figure below.

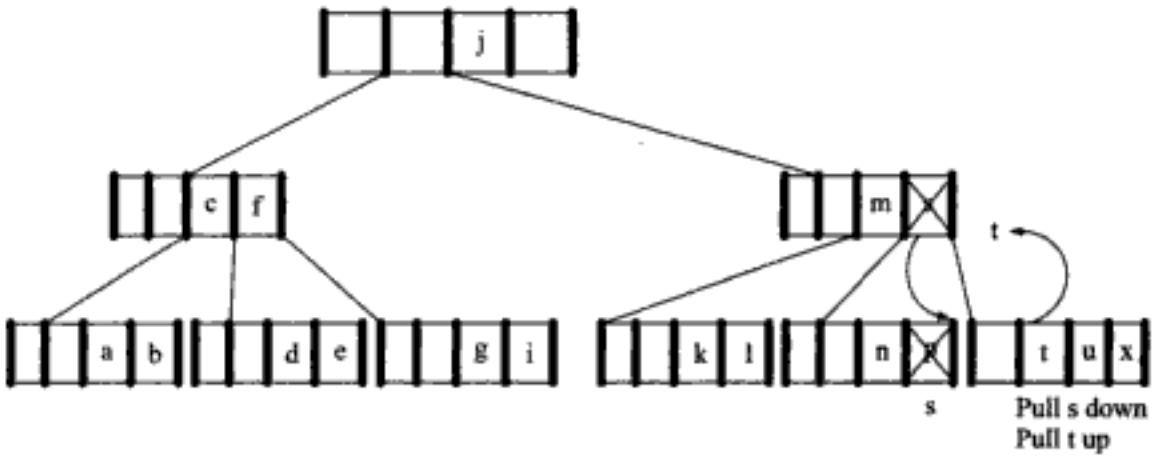
1. The first deletion **h**, is from a leaf with more than the minimum number of keys and causes no problem.



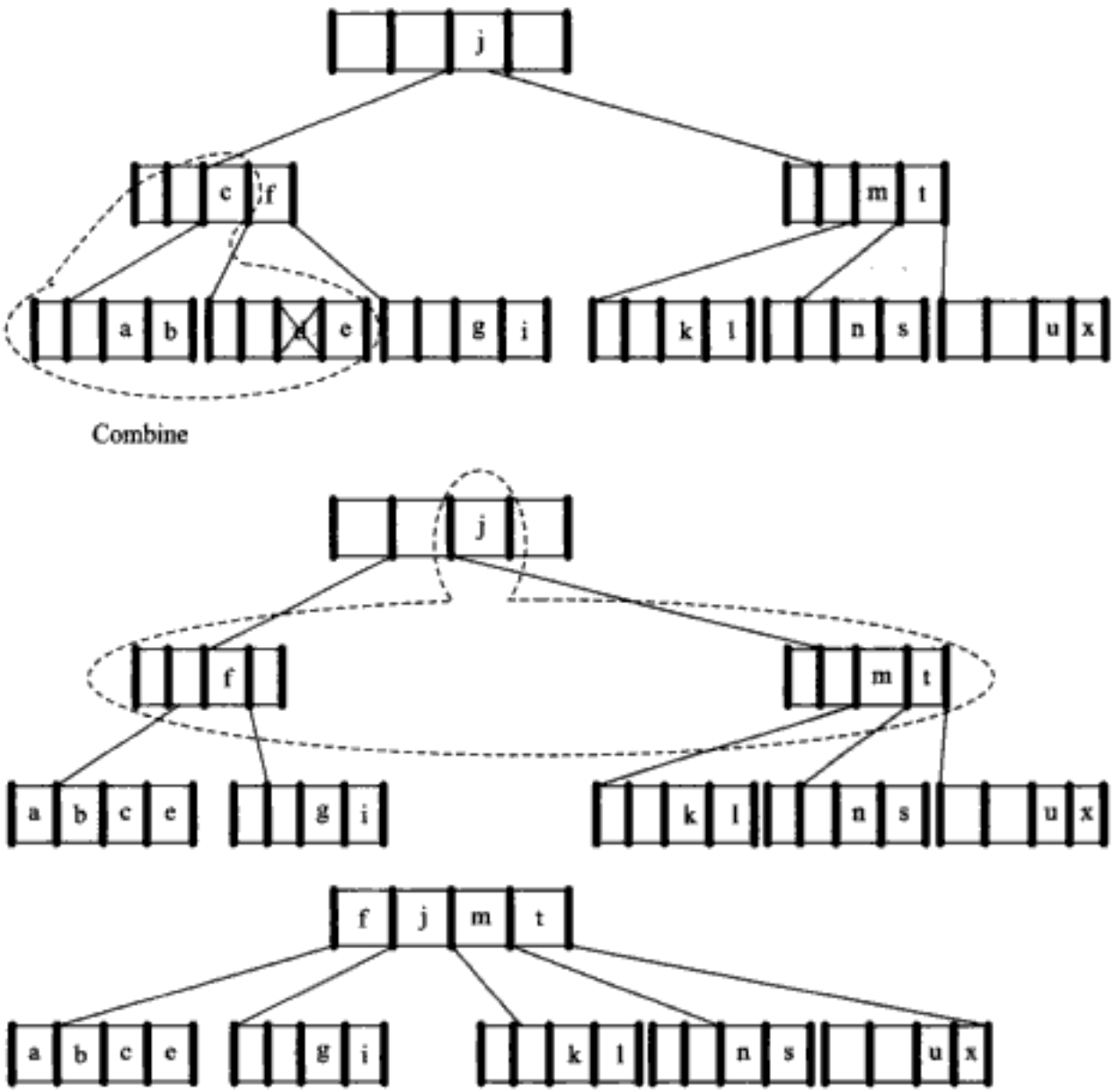
2. The second deletion **r** is not from a leaf, and therefore, the immediate successor of **r**, which is **s**, is promoted into the position of **r**, and then **s** is deleted from the leaf.



3. The third deletion **p** leaves its nodes with too few keys. The key **s** from the parent node is therefore brought down and replaced by the key **t**.



The next is deletion of **d.** This deletion again leaves the node with too few keys and neither of its sibling nodes can spare a key. The node is therefore combined with one of the siblings and with the median key from the parent node, as shown by the dotted line in the diagram and the combined node **a, b, c, e** in the other diagram. The process however leaves the parent node with only the key **f**. The top three nodes of the tree must therefore be combined.



B⁺ TREES

One of the major drawbacks of B tree is the difficulty of traversing the tree sequentially. An improvement over B-tree is B^+ tree which retains the rapid random access property of the B-tree while also allowing rapid sequential access. In B^+ trees all the keys are maintained in leaves and all non-leaf nodes contain the replication of keys to define path for locating individual record. The leaves are linked together to provide a sequential path for traversing the key.

The linked list of the leaves is called a "**sequence set**". The B^+ tree may be considered to be a natural extension of an indexed sequential file. Each level of the tree is the index to the succeeding level and the lowest level of the sequence set is the index to the file itself.

Insertion into B^+ trees proceeds much like B-trees except that the node is split, the middle key is retained in the left half node as well as being promoted to the father.

When a key is deleted from a leaf it can be retained in non-leaf node. Time complexity in a B^+ tree for searching algo is **0(1)**.

Additional advantage of a B^+ tree is that no record pointers are required to be maintained in a non-leaf node which increases the potential order of tree.

HEAPS

A **binary heap** is a simple data structure that efficiently supports the priority queue operations.

A **binary heap** (or simply heap) is a complete binary tree in which the key value stored at any vertex is less than or equal to the key values of its children.

There are two types of heaps—If the value present at any node is greater than all its children then such a tree is called **max-heap** or **descending heap**. In case of a **min-heap** or **ascending heap** the value present at any node is smaller than all its children. Figure 10.8 shows the descending heap.

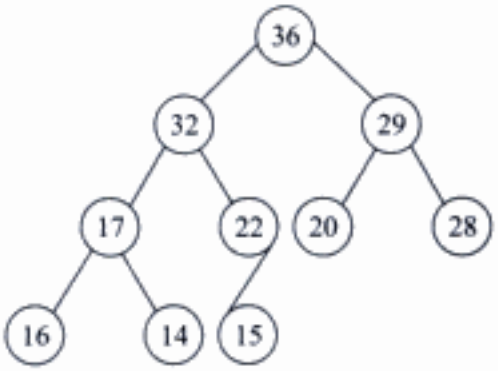


Fig. 10.8 *Descending Heap*

Insertion of a Node in a Heap

To insert an element to the heap, the node is inserted after the last element in the array and is compared with its parent that is present at $(i/2)^{th}$ position. If it is found to be greater than its parent then they are exchanged. This procedure is repeated till the element is placed at its appropriate place.

Consider the example in which a node containing a value 24 is inserted in the tree shown in Fig. 10.9.

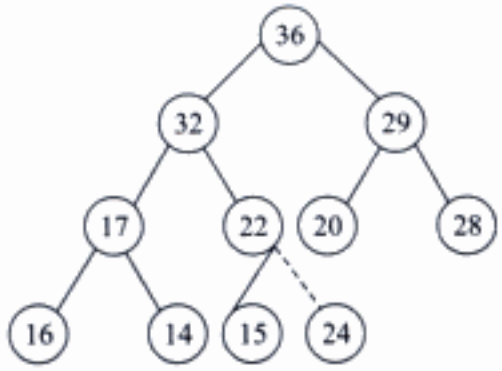


Fig. 10.9 *Insertion of a Node in a Heap*

For the tree given above consider the following array:

	a[0]	a[1]	a[2]	a[3]	a[4]	a[5]	a[6]	a[7]	a[8]	a[9]
a	1000	36	32	29	17	22	20	28	16	14

a[10]	a[11]	a[12]	a[13]	a[14]
15	\0	\0	\0	\0

From the array given above we can see that the root node of the tree starts from the index **1** of the array. The 0th element is called the **senitel value** that is the maximum value and is not the node of the tree. It can be any number, say 1000. This value is required because while adding new node, certain operations are performed in a loop and to terminate the loop senitel value is used.

Hidden page

Adjustment of nodes starts from the level one less than the maximum level of the tree (as the leaf nodes are always heap). Each subtree of that particular level is made a heap. Then all the subtrees at the level two less than maximum level of trees are made heaps. This procedure is repeated till the root node gets the maximum or the minimum element in the list. As a result final tree becomes heap.

For example, consider an array that contains 15 elements given below:

6, 8, 24, 17, 22, 25, 15, 18, 35, 40, 2, 32, 1, 4, 10

The tree can be constructed from the array as shown in Fig. 10.12.

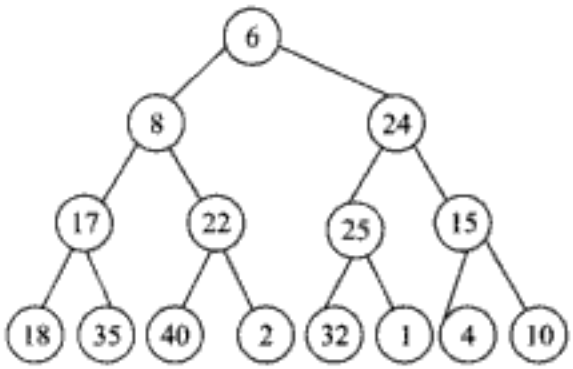


Fig. 10.12 *Tree Construction from the Array*

Steps to Construct Heap

1. The first step is to consider the elements present at a level one less than the maximum level. From the tree above these are 17, 22, 25 and 15. They are converted to heaps in the same way as replacing the node of a heap. This is shown in Fig. 10.13.

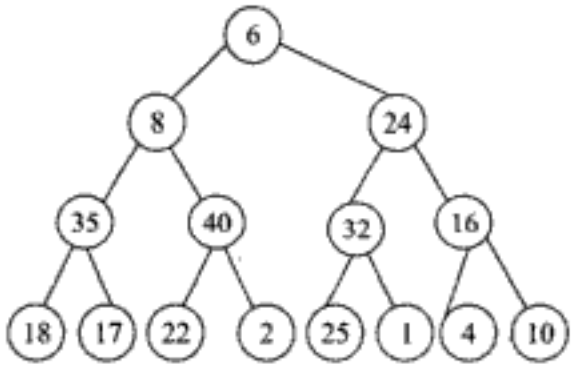


Fig. 10.13 *Step 1*

2. In the next step the elements that are present at a level two less than the maximum level of the tree are considered. The elements in our case are 24 and 8.
3. Likewise, in each step one level is decremented and all the subtrees at that level are converted to heaps.

As a result, the entire tree gets converted to a heap.

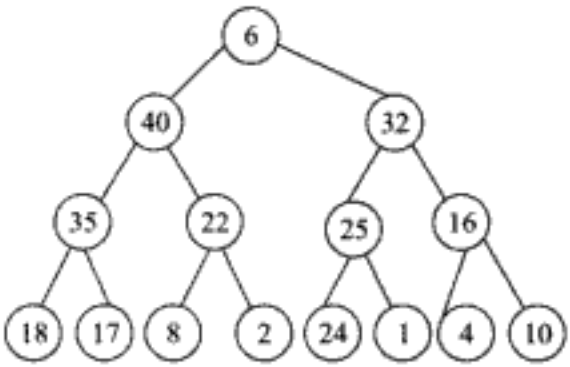


Fig. 10.14 *Step 2*

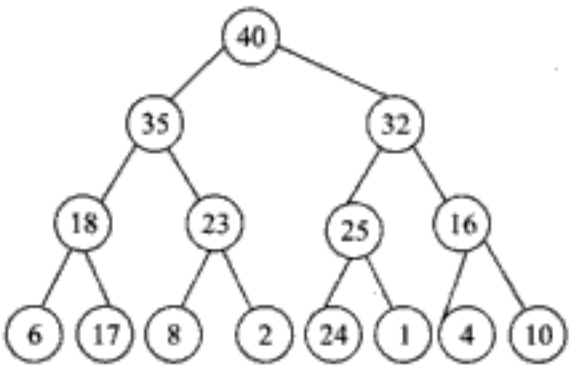


Fig. 10.15 *Step 3*

Summary

- A different way to construct a height balanced tree is possible if some restrictions are applied on the non-leaf nodes. This is done in a 2-3 tree where all non-leaf nodes must have either two or three children.
- The process of insertion and deletion operations in 2-3 tree involves splitting a node or merging two nodes.
- A B-tree of order "m" is an m-way search tree where each node except the root must have [m/2] children at most "m" children. The root node is allowed to have two children at the minimum.
- The searching, insertion and deletion operations on a B-tree are same as that of 2-3 tree. It is interesting to note that B-tree grows or shrinks upwards during insertion and deletion of key values.
- Heap is another data structure for implementing a priority queue. A heap is a complete binary tree with a special property. This property states that the key value at any node is greater than or equal to the key values of both its children.

Review Exercise

Multiple Choice Questions

1. Which of the following statements is True in view of multiway search trees? If a node has
   a. 4 subtrees, it contains 3 keys
   b. 5 keys, it has 7 subtrees
   c. 6 subtrees, it contains 6 keys
   d. None of the above
2. The element at the root of the heap is
   a. largest
   b. smallest
   c. depending on type of heap it may be smallest or largest
   d. None of the above

Fill in the Blanks

1. In a B-tree of order n, each non root node contains at least _____ keys. (n/n/2)
2. The minimum number of keys contained in each non-root node of a B-tree of order 15 is _____. (7/9)
3. A B-tree of order n is also called _____. (n – (n–1) / n + (n–1)
4. A _____ is a complete binary tree where value at each node is at least as much as values at children node. (heap/B-tree)
5. In a B-tree of order m, no node has more than _____ subtrees. (m/n)

Descriptive Questions

1. Insert the following keys into a B-tree of order (i) 3 (ii) 4 and (iii) 5.

   10, 24, 23, 31, 16, 26, 35, 29, 20, 46.

Hidden page

CHAPTER

11 Searching and Sorting

Key Features

- Sorting—An Introduction
- Efficiency of Sorting Algorithms
- Bubble Sort
- Selection Sort
- Quick Sort
- Insertion Sort
- Merge Sort
- Binary Tree Sort
- Radix Sort
- Shell Sort
- Heap Sort
- Searching—An Introduction
- Linear or Sequential Search
- Binary Search
- Indexed Sequential Search

This chapter explores various searching and sorting techniques. The process of identifying or finding a particular record is called "Searching" whereas "Sorting" implies arranging a set of data in some logical order.

The sorting methods can be implemented in different ways—by selection, insertion or merging. The mechanics of various sorting and searching methods have been explored in this chapter to enable a quick comparison between them.

SORTING—AN INTRODUCTION

Sorting refers to the operation of arranging a set of data in some given order. A collection of records is called a **list** where each record contains one or more fields. The field, which contains unique value for each record is known as the **key** field. For example, a telephone directory can be considered as a list where each record has three fields—name, address and phone number. Being unique, phone number can work as a key to locate any record in the list.

Sorting is the operation of arranging the records of a table in some order according to an ordering criterion. Sorting is performed according to the key value of each record. The records are sorted either numerically or alphanumerically. The records are arranged in ascending or descending order depending on the numerical value of the key. For example, sorting of a list of marks obtained by a student in the class.

The methods of sorting can be divided into two categories:

- Internal Sorting
- External Sorting

Internal Sorting

If all the data that is to be sorted can be adjusted at a time in main memory, then internal sorting methods are used.

External Sorting

When the data to be sorted can't be accommodated in the memory at the same time and some has to be kept in auxiliary memory (hard disk, floppy, tape, etc.), then external sorting methods are used.

EFFICIENCY OF SORTING ALGORITHMS

The complexity of a sorting algorithm measures the running time of a function in which **n** number of items are to be sorted. The choice of which sorting method is suitable for a problem depends on various efficiency considerations for different problems. Three most important of these considerations are:

- The length of time spent by programmer in coding a particular sorting program.
- Amount of machine time necessary for running the program.
- The amount of memory necessary for running the program.

To get an idea of amount of time required to sort an array of "**n**" elements by a particular method, the normal approach is to analyze the method to find the number of comparisons (or exchanges) required by it. Most of the sorting methods are **data sensitive** and therefore, the metrics for them depend on the order in which they appear in an input array. Various sorting methods are analyzed in the cases like—**best case**, **worst case** or **average case**. Therefore, the results of these cases or analysis is often a formula giving the average time (or number of operations) required for a particular sort of size **n**.

Most of the sort methods we consider have time requirements that range from **0(nlog n)** to $\mathbf{0(n^2)}$. A sort should not be selected only because its sorting time is **0(nlogn)**; the relation of the file size **n** and the other factors affecting the actual sorting time must be considered.

A second method of determining the time requirements of a sorting technique is to actually run the program and measure its efficiency (either by measuring the absolute time units or the number of operations performed).

The time needed by a sort depends on the original sequence of data. In some sorts, where input data is almost sorted can be completely sorted in time **0(n)**, whereas input data that is in reverse order needs time that is $\mathbf{0(n^2)}$. For other sorts, the time required is **0(n log n)**, regardless of the original order of data. The original sequence of data can help us in deciding the sorting method to select.

Once a particular sorting technique is selected the need is to make the program as efficient as possible. Once a sorting program has been written and tested, the next major goal is to improve its speed. Any improvement in sorting time significantly affects the overall efficiency, and a small improvement in the execution of the method saves a great deal of computer time.

Space constraints are usually less important than time considerations. The reason for this can be, as for most sorting programs, the amount of space needed is closer to **0(n)** than to $\mathbf{0(n^2)}$. The second reason is that, if more space is required, it can almost always be found in auxiliary storage. An ideal sort is an **in-place sort** where additional space requirements are **0(1)**.

The in-place sort manipulates the elements to be sorted within the array or list space that contained original unsorted input.

There is a relationship between time and space for sorting algorithms—those programs that require less time usually require more space and vice-versa. There are some algorithms that use both minimum time and minimum space, that is, they are **0(n log n)** in-place sorts.

BUBBLE SORT

In this sorting method, to arrange elements in ascending order, we begin with the 0^{th} element and compare it with the 1^{st} element. If it is found to be greater than the 1^{st} element, then they are interchanged. In this way all the elements are compared (excluding last) with their next element and are interchanged if required. On completing the first iteration, the largest element gets placed at the last position. Similarly, in the second iteration second largest element gets placed at the second last position and so on. As a result, after all the iterations, the list becomes a sorted list.

Algorithm for Bubble Sort

1. In the first iteration, 0^{th} element 23 is compared with 1^{st} element 15 and since 23 is greater than 15, they are interchanged.
2. Now the 1^{st} element 23 is compared with 2^{nd} element 29 but 23 being less than 29 they are not interchanged.
3. Repeat the process until $(n-2)^{nd}$ element is compared with $(n-1)^{th}$ element. If during comparison, $(n-2)^{nd}$ element is found to be greater than the $(n-1)^{th}$, then they are interchanged else not.
4. When the first iteration is over $(n-1)^{th}$ element holds the largest number.
5. Now the second iteration begins with 0^{th} element 15. The above process of comparing and interchanging is repeated but this time the last comparison is made between $(n-3)^{rd}$ and $(n-2)^{nd}$ elements.
6. If there are **n** number of elements in the input list, then **(n–1)** iterations need to be performed.

Program for Bubble Sort

The following program implements the bubble sort algorithm:

```
/* Bubble sort. */
#include <stdio.h>
#include <conio.h>
void main( )
{
     int arr[5] = {23, 15, 29, 11, 1} ;
     int i, j, temp ;
     clrscr( ) ;
     printf("Bubble sort.\n") ;
     printf("\nArray before sorting:\n") ;
     for(i = 0 ; i <= 4 ; i++)
          printf("%d\t", arr[i]) ;
     for(i = 0 ; i <= 3 ; i++)
     {
```

```
            for(j = 0 ; j <= 3 - i ; j++)
            {
                  if(arr[j] > arr[j + 1])
                  {
                        temp = arr[j] ;
                        arr[j] = arr[j + 1] ;
                        arr[j + 1] = temp ;
                  }
            }
      }
      printf("\n\nArray after sorting:\n") ;
      for(i = 0 ; i <= 4 ; i++)
            printf("%d\t", arr[i]) ;
      getch( ) ;
}
```

Output:

```
  Bubble sort
Array before sorting:
23  15  29  11  1
Array after sorting:
  1  11  15  23  29
```

Hence, in the above program the elements are compared—**arr[j]** and **arr[j+1]**. If the element **arr[j]** is found greater than **arr[j+1]** then they are interchanged.
On the other hand to sort the list in descending order the comparisons are made as follows:

if(arr[j] < arr [j+1])

Complexity of Bubble Sort

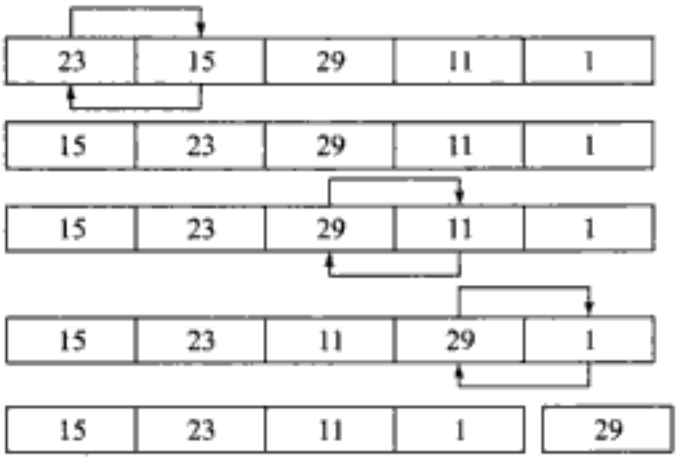
Bubble sort is data sensitive. The number of iterations required may be between '**1**' and "**(n–1)**". The best case for bubble sort is when only one iteration is required. The number of comparisons required is **(n–1)**. This case arises when input array is already sorted.

The worst case arises when the given array is sorted in reverse order. In this case all the iterations required will be:

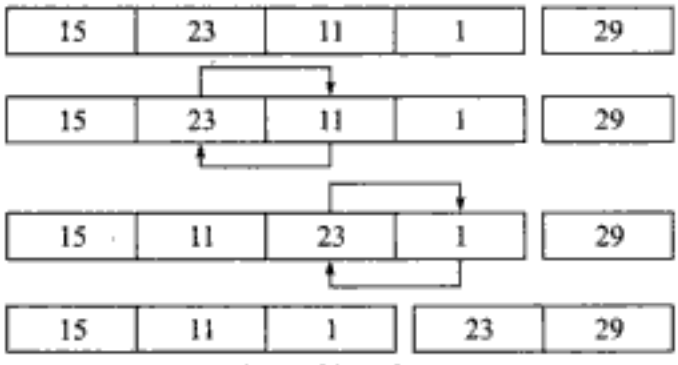
$$(n-1) + (n-2) + \ldots + 2 + 1 = n(n-1)/2$$

In an average case to find the number of comparisons, one must find the expected number of iterations first. An extra iteration is required to ensure that the sorting has been completed.

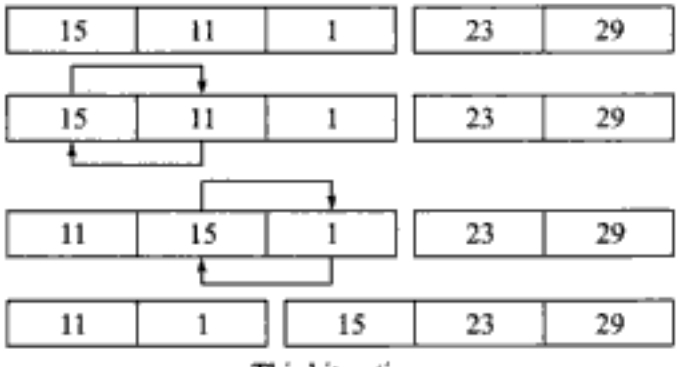
Worst case complexity	$= 0(n^2)$
Average case complexity	$= 0(n^2)$
Best case complexity	$= 0(n^2)$



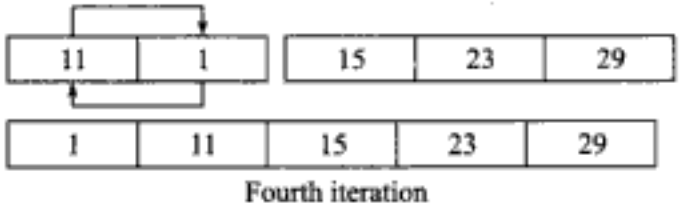
First iteration



Second iteration



Third iteration



Fourth iteration

Fig. 11.1 *Bubble Sort*

SELECTION SORT

The selection sort is the easiest method of sorting. In this, to sort the data in ascending order, the 0^{th} element is compared with all other elements. If the 0^{th} element is found to be greater than the compared

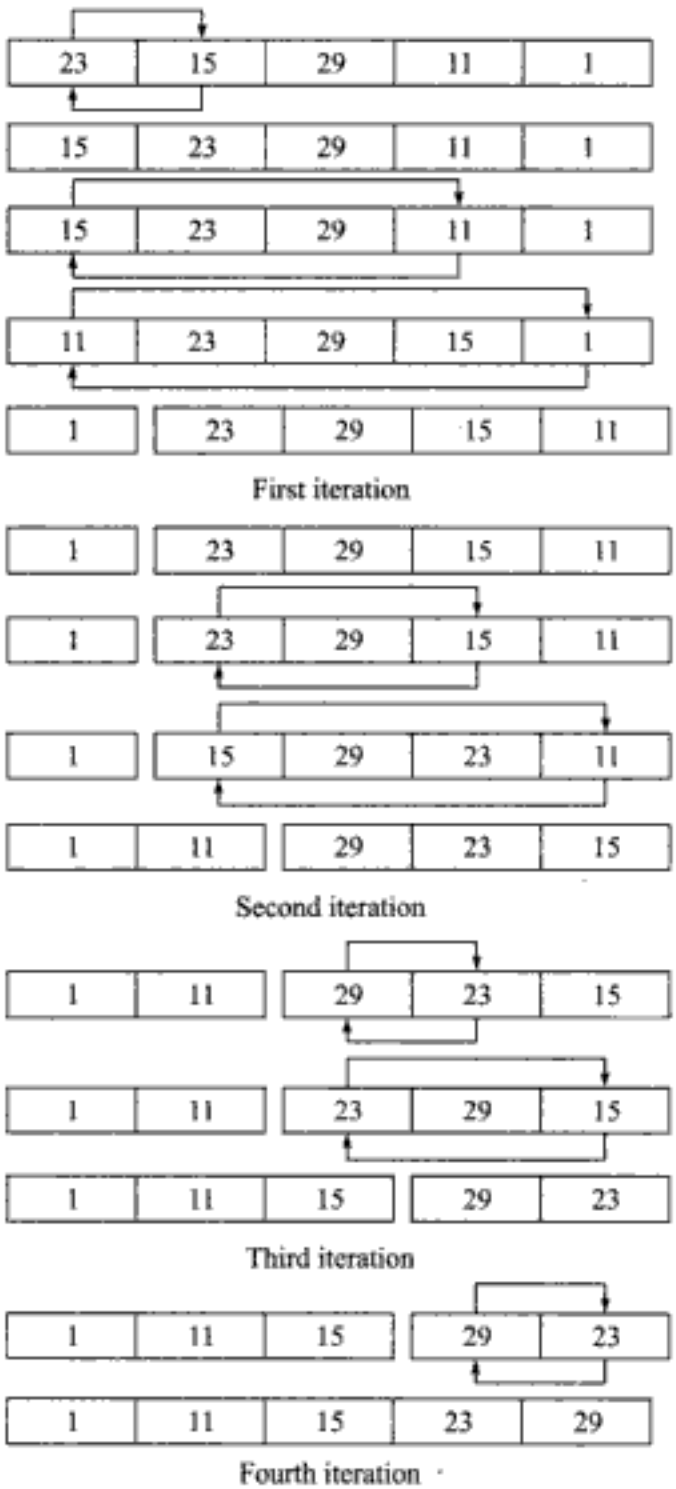


Fig. 11.2 *Selection Sort*

element then they are interchanged. In this way, after the first iteration, the smallest element is placed at 0^{th} position. The procedure is repeated for 1^{st} element and so on.

Algorithm for Selection Sort

Say an array **arr** contains 5 elements. The algorithm for selection sort is as follows:

1. In the first iteration 0^{th} element 23 is compared with 1^{st} element 15 and since 23 is greater than 15, they are interchanged.
2. Now the 0^{th} element 15 is compared with the 2^{nd} element 29. But 15 is less than 29, hence they are not interchanged.
3. The process is repeated till the 0^{th} element is compared with rest of the elements. During the comparison if 0^{th} element is found to be greater than the compared element, then they are interchanged, else not.
4. At the end of first iteration 0^{th} elements holds the smallest number.
5. The second iteration starts with the 1^{st} element 23. The process of comparison and swapping is repeated.
6. If there are **n** number of elements then after **(n–1)** iterations the array is sorted.

Program for Selection Sort

```
/*  Selection sort. */
#include <stdio.h>
#include <conio.h>
void main( )
{
     int arr[5] = {23. 15. 29. 11. 1} ;
     int i. j. temp ;
     clrscr( ) ;
     printf("Selection sort.\n") ;
     printf("\nArray before sorting:\n") ;
     for(i = 0 ; i <= 4 ; i++)
            printf("%d\t". arr[i]) ;
     for(i = 0 ; i <= 3 ; i++)
     {
            for(j = i + 1 ; j <= 4 : j++)
            {
                   if(arr[i] > arr[j])
                   {
                          temp = arr[i] ;
                          arr[i] = arr[j] ;
                          arr[j] = temp ;
                   }
            }
     }
     printf("\n\nArray after sorting:\n") ;
```

```
    for(i = 0 ; i <= 4 ; i++)
          printf("%d\t", arr[i]) ;
    getch( ) ;
}
```

```
Output:
    Selection sort
    Array before sorting:
    23 15 29 11 1
    Array after sorting
    1 11 15 23 29
```

In the above program, **arr[i]** is compared with **arr[j]**, then they are interchanged. The value of **j** is starting from **i+1** as we need to compare any element with its next element till the last element of the array.

Complexity of Selection Sort

Average Case Complexity	$= 0(n^2)$
Worst Case Complexity	$= 0(n^2)$
Best Case Complexity	$= 0(n^2)$

QUICK SORT

Quick sort is a very popular sorting method. It was introduced by Haore in 1962. It is a very important method of internal sorting. According to this algorithm it is faster and easier to sort two small arrays than one larger one. The strategy that quick sort follows is of **divide and conquer**. In this approach, numbers are divided and again subdivided and division goes on until it is not possible to divide them further. The procedure it follows is of recursion. It is also known as **partition exchange sort**. The procedure of quick sort can be understood from Fig. 11.3 where '*' indicates the pivot element and '●' indicates the element whose position is finalized.

Algorithm for Quick Sort

1. In the first iteration, the $\mathbf{0^{th}}$ element 10 is placed at its final position and the array is divided. Two index variables **a** and **b** are taken to divide the array. The indexes are initialized in a way such that **a** refers to the $\mathbf{1^{st}}$ element 1 and **b** refers to the $\mathbf{(n-1)^{th}}$ element 2.
2. The job of index variable **a** is to search an element that is greater than the value at $\mathbf{0^{th}}$ location. So **a** is incremented by one till the value stored at **a** is greater than $\mathbf{0^{th}}$ element. In our case, it is incremented till 11, as 11 is greater than 10.
3. Similarly, **b** needs to search an element that is smaller than the $\mathbf{0^{th}}$ element so **b** is decremented by one till the value stored at **b** is smaller than the value at $\mathbf{0^{th}}$ location. In our case, **b** is not decremented because 2 is less than 10.
4. When these elements are found they are interchanged. Again from the current positions **a** and **b** are incremented and decremented respectively and exchanges are made appropriately if desired.
5. The process ends whenever the index pointers meet. In our case, they are crossed at the value 0 and 20 for the indexes **a** and **b** respectively. Finally, the $\mathbf{0^{th}}$ element 10 is interchanged with the value at index **b**, i.e. 0. The position **b** is now the final position of the pivot element 10.

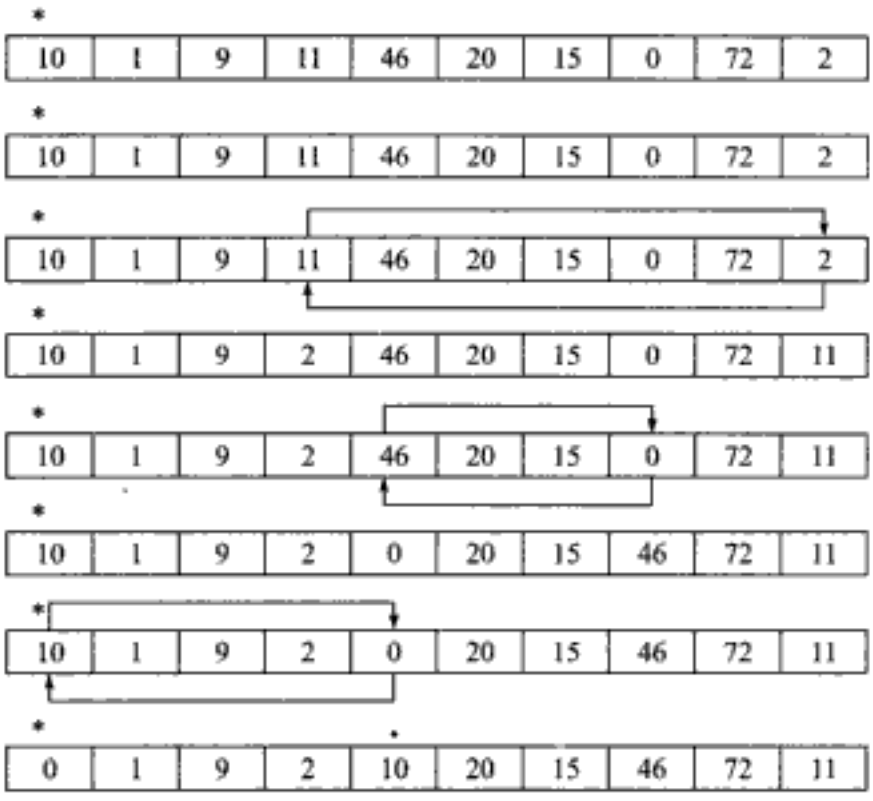


Fig. 11.3 *Quick Sort*

6. As a result, the whole array is divided into two parts where all the elements before 10 are less than 10 and all the elements after 10 are greater than 10.
7. The same procedure is applied for the two sub-arrays. As a result, at the end when all sub-arrays are left with one element, the original array becomes sorted.

Program for Quick Sort

```
/* Quick sort. */
#include <stdio.h>
#include <conio.h>
int split(int*, int, int) ;
void main( )
{
     int arr[10] = {10, 1, 9, 11, 46, 20, 15, 0, 72, 2} ;
     int i ;
     void quicksort(int *, int, int) ;
     clrscr( ) ;
     printf("Quick sort.\n") ;
     printf("\nArray before sorting:\n") ;
     for(i = 0 ; i <= 9 ; i++)
            printf("%d\t", arr[i]) ;
     quicksort(arr, 0, 9) ;
```

```
    printf("\nArray after sorting:\n") ;
    for( i = 0 ; i <= 9 ; i++)
        printf("%d\t", arr[i]) ;
    getch( ) ;
}
void quicksort(int z[ ], int lower, int upper)
{
    int i ;
    if(upper > lower)
    {
        i = split(z, lower, upper) ;
        quicksort(z, lower, i - 1) ;
        quicksort(z, i + 1, upper) ;
    }
}
int split(int z[ ], int lower, int upper)
{
    int i, a, b, t ;
    a = lower + 1 ;
    b = upper ;
    i = z[lower] ;
    while(b >= a)
    {
        while(z[a] < i)
            a++ ;
        while(z[b] > i)
            b-- ;
        if(b > a)
        {
            t = z[a] ;
            z[a] = z[b] ;
            z[b] = t ;
        }
    }
    t = z[lower] ;
    z[lower] = z[b] ;
    z[b] = t ;
    return b ;
}
```

The arguments given to the function **quicksort()** indicate the part of the array that is being currently operated on. The first and the last indexes are also passed to define the part of the array to be processed. The initial call to **quicksort()** would contain the arguments 0 and 9. Since, there are 10 integers in the array.

In the function **quicksort()** the condition is checked whether **upper** is greater than **lower**. If the condition is satisfied then only the array is split in two parts, otherwise the control will simply be returned. To split the array in two parts the function **split()** is called.

In the function **split()**, to start with, two variables **a** and **b** are taken which are assigned to the values **lower+1** and **upper**. Then the while loop is executed to check whether the indexes **a** and **b** cross each other. If they are not crossed then inside the while loop two more nested **while** loops are executed to increase the index **a** and decrease the index **b** to their appropriate places. Then it is checked whether **b** is greater than **a**. If so, the elements at $\mathbf{a}^{th}$ and $\mathbf{b}^{th}$ positions are interchanged.

Then, finally when the control returns to the **quicksort()** two recursive calls are made to the function **quicksort()**. This is done to sort the split sub-arrays. As a result, after all recursive calls, when the control reaches the function **main()** the arrays become sorted.

Complexity of Quick Sort

Assuming that the size of the file n is a power of 2, say $n=2^m$ so that $m=\log_2 n$. Assume that the proper position for the pivot always turns out to be the exact middle of the subarray. In that case, there will be approximately **(n–1)** comparisons in the first iteration after which the file is split into two subfiles each of size **n/2** approximately. For each of these two files there are approximately **n/2** comparisons and a total of four files each of size **n/4** comparisons yield a total **n/8** subfiles. After halving the subfiles **n** times, there are **n** files of size **1**. Thus, the total number of comparisons for the entire sort is

n + 2 * (n/2) + 4 * (n/4) + 8 * (n/8) + ... + n *(n/n)

or

n + n + n + n + ... + n (m terms). There are **m** terms because the file is subdivided **m** times.

The total number of comparisons is **0(n*m)** or **0(n log n)** (as $\mathbf{m=\log_2 n}$). Thus, the best case complexity can be **0(n log n)**.

Suppose now, the array is sorted, and say the pivot value is in its correct position, then original file is split into subfiles of sizes **0** and **n–1**. If this process continues, a total of **n–1** subfiles are sorted, the first of size **n**, the second of size **n–1**, the third of size **n–2**, and so on. Assuming, **K** comparisons to rearrange a file of size **K**, the total number of comparisons to sort the entire list is

n + (n–1) + (n–2) + ... + 2 which is worst case complexity that is $\mathbf{0(n^2)}$.

The efficiency of the quick sort method can be enhanced:

- Choosing a better pivot element
- Using better algorithms for sub-lists
- Eliminating recursions

INSERTION SORT

Insertion sort is performed by inserting a particular element at the appropriate position. In insertion sort, the first iteration starts with comparison of 1^{st} element with the 0^{th} element. In the second iteration 2^{nd} element is compared with the 0^{th} and 1^{st} element. In general in every iteration an element is compared with all elements. If at same point it is found that the element can be inserted at a position then space is created for it by shifting the other elements one position to the right and inserting the element at the suitable position. This procedure is repeated for all the elements in the array.

Hidden page

Hidden page

Hidden page

Hidden page

Hidden page

BINARY TREE SORT

Binary tree sort makes use of a binary search tree (BST). In this method, each element in the input list is examined and placed in its proper position in a binary tree. In binary tree each element is known as a **node**.

To place an element in an appropriate position, the element is compared with the node element. If this element is less than the element in the node, then it is moved to the right branch. Now, if we access the elements according to inorder traversal (left, root, right), we would get the elements in ascending order.

Let us understand this by taking an array, say **xarr**, of 10 elements. The elements are

13, 4, 11, 17, 59, 27, 19, 3, 92, 5.

The tree built from these elements will be:

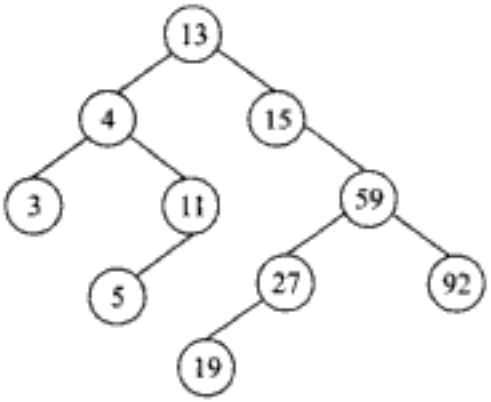


Fig. 11.6 *Binary Search Tree*

Algorithm for Binary Tree Sort

1. To create the binary tree we start with the **0th** element 13. It is the root of the tree.
2. While inserting the **1st** element, i.e. 4, it is compared with its root element 13. Since 4 is less than 13 it is made the left child of the root node 13.
3. While inserting the **2nd** element of the list, i.e., 15, it is compared with the root element 13. Since 15 is greater than 13, it is made the right child of the root node 13.
4. Using the above procedure, all the elements are placed in their proper positions in the binary search tree.
5. To get the elements in the sorted order the tree is traversed in inorder. The inorder traversal of the binary search tree lists the elements in ascending order.

Program for Binary Tree Sort

```
/* Binary Tree Sorting. */
#include <stdio.h>
#include <conio.h>
#include <alloc.h>
struct BinaryTree
```

```
{
     struct BinaryTree *LeftChild ;
     int data ;
     struct BinaryTree *RightChild ;
} ;
void insert(struct BinaryTree **, int) ;
void inorder(struct BinaryTree *) ;
void main( )
{
     struct BinaryTree *BTree ;
     int Data[10] = {11, 2, 9, 13, 57, 25, 17, 1, 90, 3} ;
     int i ;
     BTree = NULL ;
     clrscr( ) ;
     printf("Binary tree sort.\n") ;
     printf("\nArray:\n") ;
     for(i = 0 ; i <= 9 ; i++)
     printf("%d\t", Data[i]) ;
     for(i = 0 ; i <= 9 ; i++)
     insert(&BTree, Data[i]) ;
     printf("\nIn-order traversal of Binary Tree:\n") ;
     inorder(BTree) ;
     getch( ) ;
}
void insert(struct BinaryTree **sr, int num)
{
     if(*sr == NULL)
     {
            *sr = malloc(sizeof(struct BinaryTree)) ;

            (*sr) -> LeftChild = NULL ;
            (*sr) -> data = num ;
            (*sr) -> RightChild = NULL ;
     }
     else
     {
            if(num < (*sr) -> data)
                   insert(&((*sr) -> LeftChild), num) ;
            else
                   insert(&((*sr) -> RightChild), num) ;
     }
```

```
}
void inorder(struct BinaryTree *sr)
{
     if(sr != NULL)
     {
            inorder(sr -> LeftChild) ;
            printf("%d\t", sr -> data) ;
            inorder(sr -> RightChild) ;
     }
}
```

In the **insert()** function two parameters are passed—one is the pointer to the node of the tree and other is the data that is to be inserted. At the start the pointer to the node contains the NULL value, which indicates an empty tree. To find whether the pointer is NULL condition is checked. If it is found to be NULL a new node is created and the data that is to be inserted is stored in its data part. The left and the right child of this new node are set a NULL value, as the node being inserted is always going to be the leaf node.

The data of the current node is compared with the data that is to be inserted if the tree or sub-tree is not empty. If the data that is to be inserted is found to be smaller than the current node then a recursive call is made to the **insert()** function by passing the address of the node of the left sub-tree, otherwise the address of the node to the right subtree is passed. The new node is inserted at a place when in the recursive cell.

The function **inorder()** traverses the tree as per the inorder traversal. This function receives the address of the root node as parameter. To find whether the pointer is NULL a condition is checked whether the pointer is NULL. If the pointer is found to be NULL then the recursive call is made first for the left child and then for the right child. The values passed are the addresses of the left and right children that are present in the pointers left and right respectively. In between these two calls the data of the current node is printed.

The limitation of the binary tree sort is that extra space is required for constructing the tree.

Complexity of Binary Tree Sort

The worst case complexity is $\mathbf{0(n^2)}$ whereas best case complexity and average case complexity are **0(n log n)**.

RADIX SORT

The radix sort is used generally when we intend to sort a large list of names alphabetically. Radix in this case can be 26, as there are 26 alphabets. Therefore, if we want to arrange a list of names we can classify the names into 26 classes, where the first class consists of those names starting with '**A**'. The second class consists of names starting with '**B**' and so on.

All this can be done in first iteration. In the second iteration each class is alphabetized according to the second letter of the name, and so on. The radix sort method is used by a card sorter—A card sorter contains 13 receiving pockets labelled as follows:

9, 8, 7, 6, 5, 4, 3, 2, 1, 0, 11, 12, R (Reject)

The radix sort is based on the values of the actual digits in the positional representation. For example, the number 235 in decimal notation is written with a 2 in a hundred's position, 3 in ten's position and 5 in the unit's position.

The sorter used above uses decimal numbers where the radix is 10 and hence uses only the first 10 pockets of the sorter. Suppose we want to sort the following set of three digits numbers:

242, 986, 504, 428, 321

Pockets

Inputs	0	1	2	3	4	5	6	7	8	9
242			242							
986							986			
504					504					
428									428	
321		321								

First iteration

Pockets

Inputs	0	1	2	3	4	5	6	7	8	9
428			428							
986									986	
504	504									
242					242					
321			321							

Second iteration

Pockets

Inputs	0	1	2	3	4	5	6	7	8	9
986										986
242			242							
428					428					
321				321						
504						504				

Third iteration

Fig. 11.7 *Radix Sort*

The sorted list is as follows

986	504	428	321	242

In the example given above in the first iteration, the **unit's** digits are sorted into pockets and are then collected from pocket 9 to 0. The digits sorted and collected are send as input to second iteration where **ten's** digits are sorted and again collected and send to the third iteration where **hundred's** digits are sorted and collected. Finally, it gives the sorted list.

Algorithm for Radix Sort

1. In the first iteration the elements are picked up and kept in various pockets checking their unit's digit.
2. The cards are collected from pocket 9 to pocket 0 and again they are given as input to sorter.
3. In second iteration, the ten's digits are sorted.
4. Repeat through step 2.
5. In the final iteration (in the above example) the digits at hundred's position are sorted.
6. Repeat through step 2.

Complexity of Radix Sort

The time requirement for radix sort method depends on the number of digits and the number of elements in the file. The loop is traversed for each digit (**m** times) and the inner loop is traversed (**n** times) for each element in the file the sort is approximately **0(m*n)**. The **m** approximates **logn** so that **0(m*n)** approximates to **0(n log n)**.

Program for Radix Sort

```
/* Radix Sort */
# include<stdio.h>
# include<malloc.h>
# include<stdlib.h>
struct node
{
     int data ;
     struct node *next;
};
typedef struct node node1;
node1 *first;
node1 *pocket[100], *pocket1[100];
void create_node(node1 *, int);
void display(node1 *);
node1 *radix_sort(node1 *);
int large(node1 * );
int numdig(int );
int digit(int , int);
void update(int, node1 *);
node1 *Make_link(int, node1 *);
/* This function creates nodes and takes input data */
void  create_node(node1 *rec, int n)
```

Hidden page

```
            }
            r = rec ;
            while(r != NULL)
            {
                  int dig = digit(r->data, j);
                  next = r->next ;
                  update(dig,r);
                  r = next;
            }
            if(r!= NULL)
            {
                  int dig = digit(r->data,j);
                  update(dig,r);
            }
            while(pocket1[poc] == NULL)
                  poc ++;
            rec = Make_link(poc, rec);
      }
      return(rec);
}
/* This function finds largest number in the list */
int large(node1 *rec)
{
      node1 *save ;
      int p = 0;
      save = rec ;
      while(save != NULL)
      {
            if(save ->data > p)
            {
                  p = save->data;
            }
            save = save->next ;
      }
      printf("\n Largest element: %d", p);
      return(p);
}
/* This function finds number digits in a number */
int numdig(int large)
{
      int temp = large ;
```

```
    int num = 0 ;
    while(temp != 0)
    {
        ++num ;
        temp = temp/10 ;
    }
    printf("\n  Number of digits of the number %d is %d", large, num);
    return(num);
}
int digit(int num, int j)
{
    int dig, i, k;
    int temp = num ;
    for(i = 0 ; i < j ; i++)
    {
        dig = temp % 10 ;
        temp = temp / 10 ;
    }
    printf("\n  %d digit of number  %d is %d", j, num, dig);
    return(dig);
}
/* This function updates the pockets value */
void  update(int dig, node1 *r)
{
    if(pocket[dig] == NULL)
    {
        pocket[dig] = r ;
        pocket1[dig] = r ;
    }
    else
    {
        pocket[dig]->next = r ;
        pocket[dig] = r ;
    }
    r->next = NULL;
}
/* This function creates links between the nodes */
node1* Make_link(int poc , node1 *rec)
{
    int i, j, k;
    node1 *pointer;
```

```
        rec = pocket1[poc];
        for(i = poc +1 ; i< 10 ; i++)
        {
                pointer = pocket[i-1];
                if(pocket[i] != NULL)
                pointer->next= pocket1[i];
                else
                pocket[i] = pointer ;
        }
        return(rec);
}
/* Main function */
void  main()
{
        node1 *start, *pointer;
        int number;
        printf("\n Input the number of elements in the list:");
        scanf("%d", &number);
        start = (node1 *)malloc(sizeof(node1));
        create_node(start, number);
        printf("\n Given list is as follows \n");
        display(start);
        start = radix_sort(start);
        printf("\n Sorted list is as follows:\n");
        display (start);
}
```

SHELL SORT

As we have seen, insertion and selection sort behave differently. Selection sort moves the entries very efficiently but does not make redundant comparisons. In its best case, insertion sort does the minimum number of comparisons but is inefficient in moving entries only one place at a time, as it compares only adjacent keys.

If we were to modify it, first comparing the keys far apart, then it could sort the entries. Afterwards the entries closer together would be sorted and finally the increment between the keys would be reduced to 1 to ensure that the list is completely in order. This is the idea implemented in 1959 by Donald. L. Shell in the sorting method bearing his name. The method is also called **diminishing–increment sort**. The shell sort can be explained with the following example. Suppose we want to sort:

9, 1, 4, 6, 3, 5, 7, 11, 14, 10, 12, 0, 2, 8

The given numbers are first sorted at **distance 5** from each other and then resorted with **distance 3** and finally insertion sort has been performed:

Hidden page

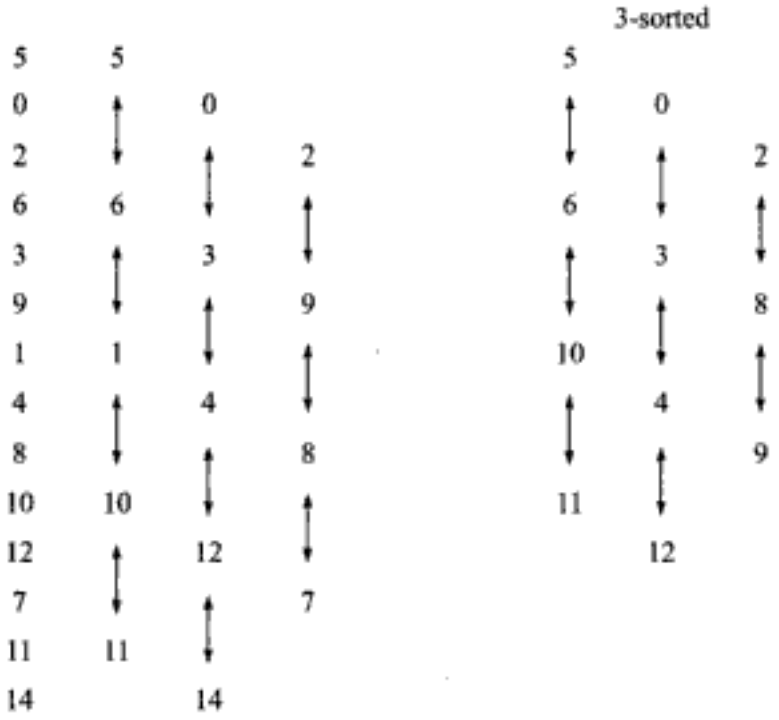


Fig. 11.8 *Shell Sort*

After getting the sorted list with distance 3 from each, other simple **insertion sort** is performed to get the final sorted list.

It is not mandatory to make 5, 3 and 1 as increments. Many other choices can also be made, but it should be considered that the choices like powers of 2 such as 8, 4, 2, 1 are not fruitful as the same keys compared in one pass would be compared again at the next pass. Therefore, making other choices may give better chance of obtaining new information from more of the comparisons.

The final iteration of shell sort has increment 1, shell sort really is insertion sort optimized by the preprocessing stage of first sorting sublists using larger increments. Although the preprocessing stage will speed up the sorting considerably by eliminating many moves of entries by only one position.

Algorithm for Shell Sort

1. In the first iteration, the elements are splitted with a gap of say, 5, 3, etc. and are sublisted.
2. The sublist obtained after splitting is again sorted.
3. The sorted sublist is recombined.
4. The sorted list, in the second iteration, is again sublisted with a gap (here, it is 3).
5. Repeat through steps 2 and 3.

Program for Shell Sort

```
/* Shell sort */
#include <stdio.h>
#include <stdlib.h>
void shell_sort(int array[], int size)
```

Hidden page

Complexity of Shell Sort

The analysis of shell sort may be quite difficult as the actual time requirement for a specific sort depends on the number of elements in the array and on their actual values. It is better if the choice of increments is made of **prime** numbers as this guarantees that successive iterations intermingle subfiles so that the entire file is indeed sorted and the span equals 1 on the last iteration.

Therefore, the shell sort can be approximated by $\mathbf{0n(\log n^2)}$.

HEAP SORT

Heap sort is based on tree structure that reflects a particular order of a corporate hierarchy. As in an organization structure of corporate management the President is at the top, when President retires Vice-President competes for the job and wins promotion and creates a vacancy. Hence vacancy continually appears at the top, the employees are competing for the promotion, as each employee reaches the "top of the heap" that position again becomes vacant. This example illustrates the idea underlying the heap sort method.

Heap sort thus proceeds in two phases:

- The entries are arranged in the list satisfying the requirements of a heap.
- We repeatedly remove the top of the heap and promote another entry to take its place.

In the second phase, we recall that the root of the tree (which is the first entry of the list as well as top of the heap) has the largest key. This key belongs to the end of the list.

The Heap sort can be understood from the following example—Assume that the array **x** that is to be sorted contains the following elements:

13, 4, 11, 15, 59, 27, 19, 3, 92, 5

The first step now is to create a heap from the array elements. For doing this imagine a binary tree that can be built from the array elements. The zeroth element would be the root element and left and right child of any element x[i] would be at **x[2*i+1]** and **x[2*i+2]** respectively.

The binary tree of the given array will be:

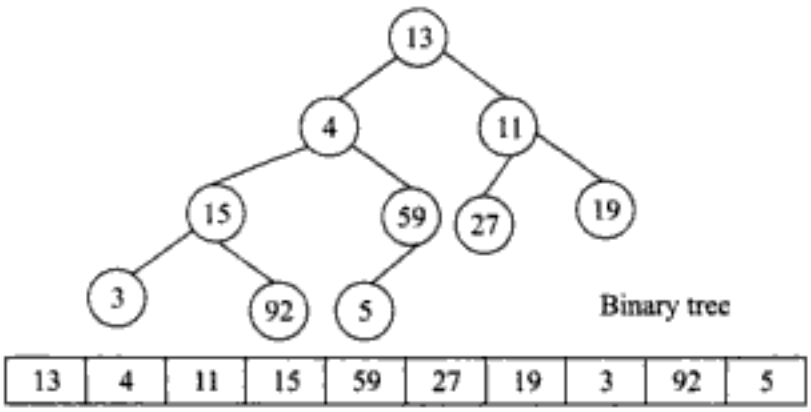


Fig. 11.9 *Binary Tree*

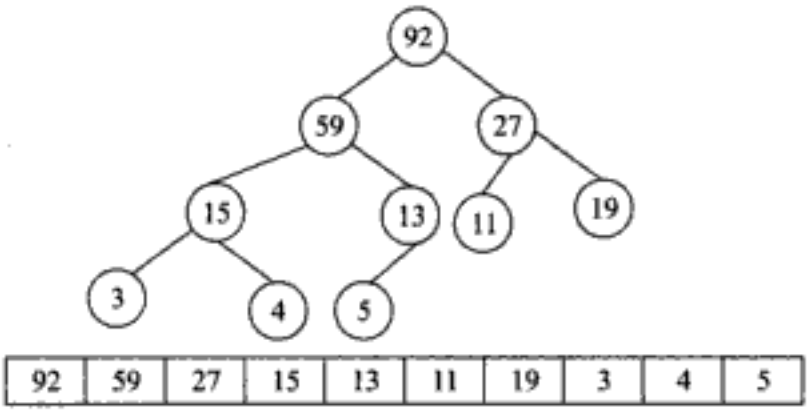
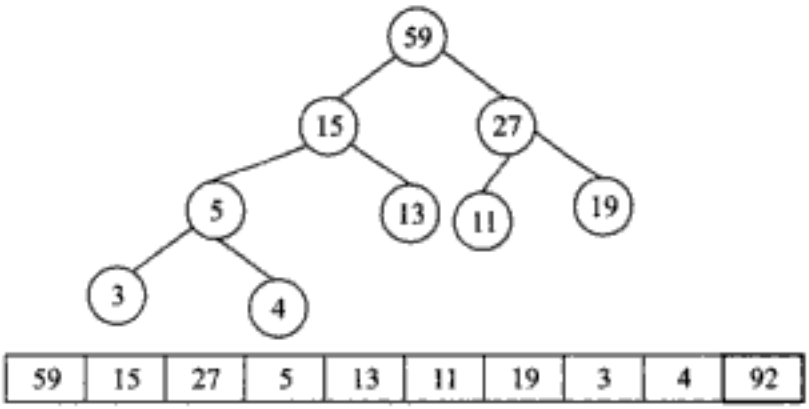
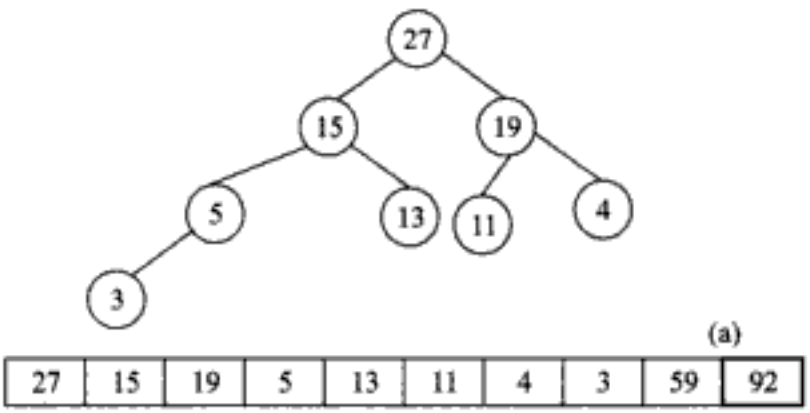


Fig. 11.10 *Heap Built from Binary Tree*

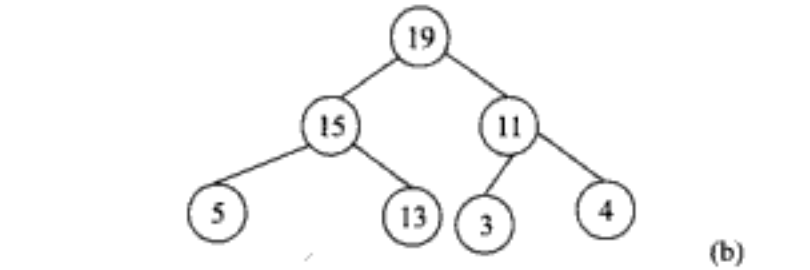
Now the root element 92 is moved to the last location by exchanging it with 5. Finally, 92 is eliminated from the heap by reducing the maximum index value of array by 3. The left elements are rearranged into heap.



Similarly, one by one root element of the tree is removed:

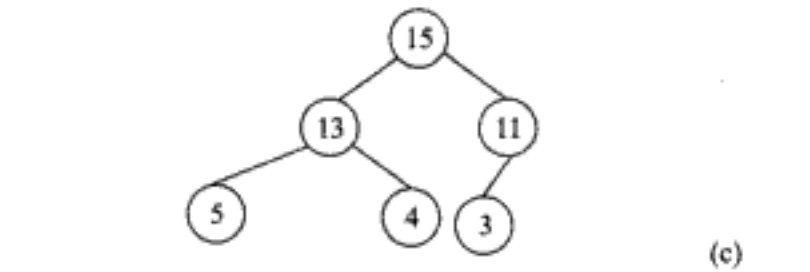


Heap after eliminating 59



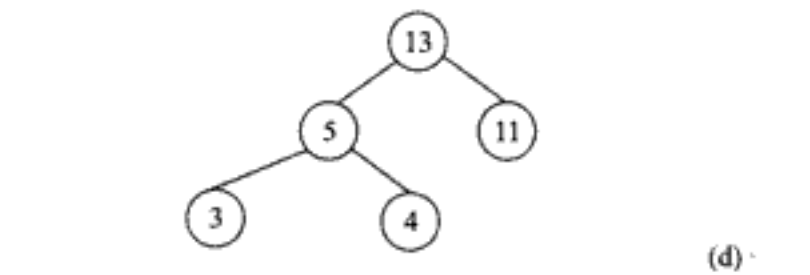
(b)

19	15	11	5	13	3	4	27	59	92



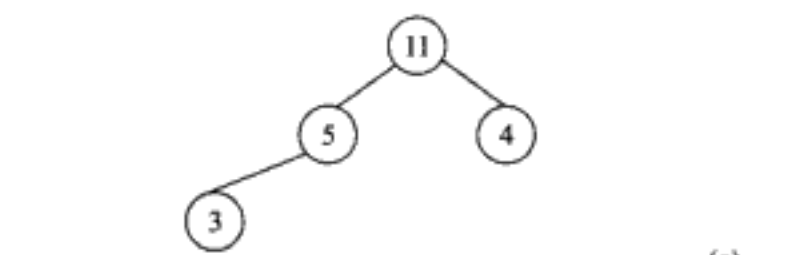
(c)

15	13	11	5	4	3	19	27	59	92



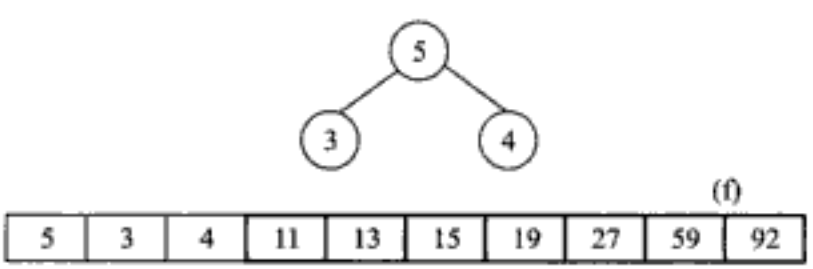
(d)

13	5	11	3	4	15	19	27	59	92



(e)

11	5	4	3	13	15	19	27	59	92



(f)

5	3	4	11	13	15	19	27	59	92

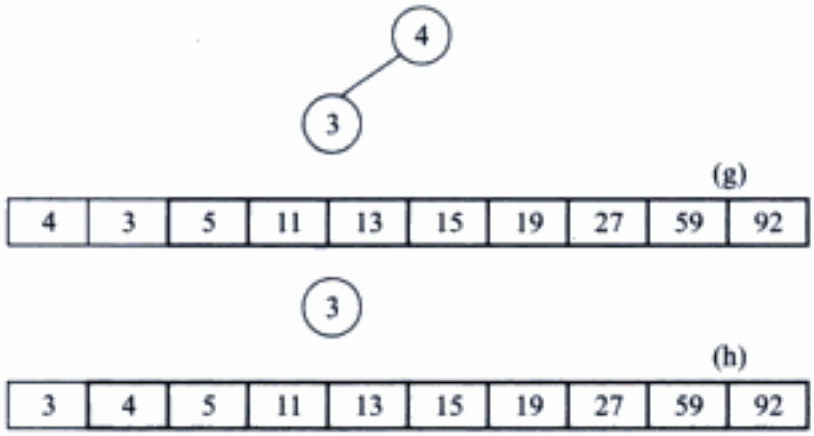


Fig. 11.11 *Heap*

Program for Heap Sort

```
/*  Heap Sort. */
#include <stdio.h>
#include <conio.h>
void creatheap(int [ ], int) ;
void sort(int [ ], int) ;
void main( )
{
     int xarr[10] = {11, 2, 9, 13, 57, 25, 17, 1, 90, 3} ;
     int i ;
     clrscr( ) ;
     printf("Heap Sort.\n") ;
     creatheap(xarr, 10) ;
     printf("\nBefore Sorting:\n") ;
     for(i = 0 ; i <= 9 ; i++)
            printf("%d\t", xarr[i]) ;
     sort(xarr, 10) ;
     printf("\nAfter Sorting:\n") ;
     for(i = 0 ; i <= 9 ; i++)
            printf("%d\t", xarr[i]) ;
     getch( );
}
void creatheap(int x[ ], int n)
{
     int i, val, s, f ;
     for(i = 1 ; i < n ; i++)
     {
            val = x[i] ;
            s = i ;
```

```
            f = (s - 1) / 2 ;
            while(s > 0 && x[f] < val)
            {
                  x[s] = x[f] ;
                  s = f ;
                  f = (s - 1) / 2 ;
            }
            x[s] = val ;
      }
}
void sort(int x[ ], int n)
{
      int i, s, f, ivalue ;
      for(i = n - 1 ; i > 0 ; i--)
      {
            ivalue = x[i] ;
            x[i] = x[0] ;
            f = 0 ;
            if(i == 1)
                  s = -1 ;
            else
            s = 1 ;
            if(i > 2 && x[2] > x[1])
                  s = 2 ;
            while(s >= 0 && ivalue < x[s])
            {
                  x[f] = x[s] ;
                  f = s ;
                  s = 2 * f + 1 ;
                  if(s + 1 <= i - 1 && x[s] < x[s + 1])
                        s++ ;
                  if (s > i - 1)
                        s = -1 ;
            }
            x[f] = ivalue ;
      }
}
```

In the program given above, two functions have been called from the **main()—creatheap()** and **sort()**. As the name suggests, **creatheap()** function is used to create the heap from the tree that can be built from the array. It receives two parameters—the base address of the array and the number of elements present in the array. Here, data of each element is compared with its child data and parent and child data is swapped if required. The function **sort** is called by passing two arguments—the base

address of the array and the number of elements in the heap. The function sorts the heap by eliminating the root elements one by one and moving them to the end of the array.

Complexity of Heap Sort

To analyze the heap sort, note that a complete binary tree with **n** nodes (where **n** is one less than the power of two) has **log(n+1)** levels. Thus, if each element in the array were a leaf, requiring it to be filtered through the entire tree, both while creating and adjusting the heap, the sort would be **0(n log n)**.

In the average case, the heap sort is not as efficient as the quick sort. However, heap sort is far superior to quick sort in the worst case. In fact, heap sort complexity remains **0(n log n)** in the worst case.

SEARCHING—AN INTRODUCTION

We often spend time in searching for the desired item. If the data is kept properly in sorted order, then searching becomes very easy and efficient.

Searching is an operation which finds the place of a given element in the list. The search is said to be successful or unsuccessful depending upon whether the element that is being searched is found or not. Some of the standard searching methods are discussed below.

Linear or Sequential Search This is the simplest method for searching. In this method, the element to be found is searched sequentially in the list. This method can be used on a sorted or an unsorted list. In case of a sorted list searching starts from **0th** element and continues until the element is found or the element whose value is greater than (assuming the list is sorted in ascending order) the value being searched is reached. As against this, searching in case of unsorted list starts from the **0th** element and continues until the element is found or the end of list is reached.

For example,

10	1	9	11	46	20	15	0	72	2

The list given above is the list of elements in an unsorted array. The array contains 10 elements. Suppose the element to be searched is 46. So 46 is compared with all the elements starting from the **0th** element and searching process ends where 46 is found or the list ends.

The performance of the linear search can be measured by counting the comparisons done to find out an element. The number of comparisons is **0(n)**.

Program for Linear Search in Sorted Array

```
/*  Linear search in a sorted array. */
#include <stdio.h>
#include <conio.h>
void main( )
{
     int list[10] = {0, 1, 2, 9, 10, 11, 15, 20, 46, 72} ;
     int i, no ;
     printf("Enter number to search: ") ;
```

```
    scanf("%d", &no) ;
    for(i = 0 ; i <= 9 ; i++)
    {
        if(list[9] < no || list[i] >= no)
        {
            if(list[i] == no)
            printf("The number is at position %d in the array.", i) ;
            else
                printf("Number is not present in the array.") ;
            break ;
        }
    }
    getch( ) ;
}
```

Here, inside the **for** loop it is checked whether **list[9]** is less than **no** or **list[i]** is greater than or equal to **no**. If the condition is satisfied then again the condition is checked whether **list[i]** is equal to **no**. Depending upon the condition, the desired message will be printed. In either case the **for** loop is terminated because there is no point in searching the element further (as the **list** is in sorted order).

The number of comparisons in case of sorted list might be less as compared to the unsorted list because the search need not always continue till the end of the list.

Program for Linear Search in Unsorted Array

```
/* Linear search in an unsorted array. */
#include <stdio.h>
#include <conio.h>
void main( )
{
    int list[10] = {10, 1, 9, 11, 46, 20, 15, 0, 72, 2} ;
    int i, no ;
    printf("Enter number to search: ") ;
    scanf("%d", &no) ;
    for(i = 0 ; i <= 9 ; i++)
    {
        if(list[i] == no)
            break ;
    }
    if(i == 10)
        printf("Number is not present in the array.") ;
    else
        printf("The number is at position %d in the array.", i) ;
    getch( ) ;
}
```

In the program no is the number that is to be searched in the array **list**. Inside the **for** loop each time **list[i]** is compared with **no**. If any element is equal to **no** then that is the position of element where the number being searched is found. Hence, break is applied to the **for** loop.

In case of a **sorted list**, searching of element starts from 0^{th} element. Searching ends when the element is found or any element of the list is found to be greater than the element to be searched.

For example,

0	1	2	9	10	11	15	20	46	72

Sorted Array

BINARY SEARCH

Binary search technique is very fast and efficient. It requires the list of elements to be in sorted order.

In this method, to search an element we compare it with the element present at the centre of the list. If it matches then the search is successful otherwise, the list is divided into two halves—one from the 0^{th} element to the centre element (first half), another from the centre element to the last element (second half). As a result all the elements in first half are smaller than centre element whereas all the elements in second half are greater than the centre element.

The searching will now proceed in either of the two halves depending upon whether the target element is greater or smaller than the centre element. If the element is smaller than the centre element then the searching is done in the first half, otherwise it is done in the second half.

The process of comparing the required element with the centre element and if not found then dividing the elements into two halves is repeated till the element is found or the division of half parts gives one element.

0	1	2	9	10	11	15	20	46	72

Sorted List for Binary Search

Let us take an array **arr** that consists of 10 sorted numbers and 46 is the element that is to be searched. The binary search when applied to this array works as follows:

(a) 46 is compared with the element present at the centre of list (i.e. 10) since 46 is greater than 10, the sorting is done in the second half of the array.
(b) Now, 46 is compared with the centre element of the second half of the array (i.e. 20). Again, 46 is greater than 20, the searching will be done between 20 and the last element 72.
(c) The process is repeated till 46 is found or no further subdivision of array is possible.

Program for Binary Search

```
/*  Binary search in a sorted array. */
#include <stdio.h>
#include <conio.h>
void main( )
{
```

```
    int Data[10] = {0, 1, 2, 9, 10, 11, 15, 20, 46, 72} ;
    int Mid, Lower = 0 , Upper = 9, Num, Flag = 1 ;
    clrscr( ) ;
    printf("Enter number to search: ") ;
    scanf("%d", &Num) ;
    printf("The List of Data is :\n\n");
    printf("%-20s","\nData");
    for(Num=0;Num<=Upper;Num++)
          printf("%2d ",Data[Num]);
          printf("%-20s","\nIndexNo.");
    for(Num=0;Num<=Upper;Num++)
          printf("%3d",Num);
    for(Mid = (Lower + Upper) / 2 ; Lower <= Upper) ;
    Mid = (Lower + Upper) / 2)
    {
          if(Data[Mid] == Num)
          {
                printf("\n\n\n\tThe number is at position %d in the array.",
          Mid) ;
                Flag = 0 ;
                break ;
          }
          if(Data[Mid] > Num)
                Upper = Mid - 1 ;
          else
                Lower = Mid + 1 ;
    }
    if(Flag)
    printf("\nElement is not present in the array.") ;
    getch( ) ;
}
```

In the program each time through the loop **arr[mid]** is compared with **num** as **mid** holds the index of the middle element of the array. If the **num** is found then the search ends. If not then in further searching it is checked whether **num** is present in lower half or upper half of the array. If **num** is found to be smaller than the middle element then **mid–1** is made the upper limit, keeping the lower limit as it is otherwise **mid+1** is made the lower limit of searching, keeping the upper limit as it is. During each iteration the value of mid is calculated as **mid = (lower+upper)/2**.

For binary search, maximum number of comparisons for successful and unsuccessful search is given by $\mathbf{0(\log_2 n)}$.

In comparison with sequential search it is found that the binary search is more efficient when any element is likely to be searched. As in binary search, in each iteration the number of elements to be searched reduces from **n** to **n/2**. On the other hand, sequential search checks sequentially for every element in the list.

INDEXED SEQUENTIAL SEARCH

Another technique to improve search efficiency for a sorted file is indexed sequential search, but it involves an increase in the amount of memory space required. An auxiliary table called an **index**, is set aside in addition to a sorted file. Each element in the index table consists of a **key kindex** and pointer to the record in the file that corresponds to kindex. The elements in the index, as well as elements in the file, must be sorted on the file.

The algorithm used for searching an indexed sequential file is simple and straight. Let **r**, **k** and **key** be defined as before. Let **kindex** be an array of the keys in the index, and let **pindex** be the array of pointers within the index to the actual records in the file, and the size of index is also taken in a variable. The indexed sequential search can be understood from Fig. 11.12.

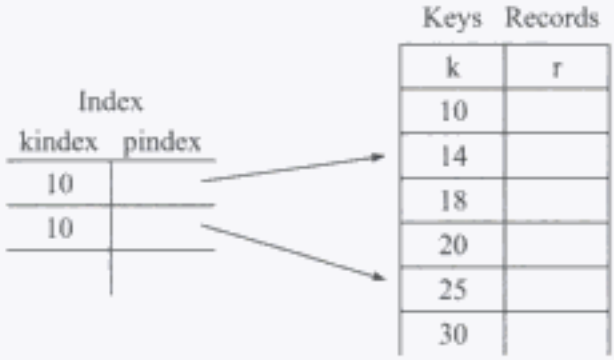


Fig. 11.12 *Indexed Sequential Search*

The advantage of indexed sequential method is that items in the table can be examined sequentially if all records in the file have to be accessed, yet the search time for particular item is reduced. A sequential search is performed on smaller index rather than on the larger table. Once the index position is found, search is made on the record table itself.

Deletion from an indexed sequential table can be made most easily by flagging deleted entries. When sequential searching is done deleted items are ignored. The item is deleted from the original table.

Insertion into an indexed sequential table may be difficult as there may not be any place between two table entries which may lead to a shift to a large number of elements.

However, the deleted items can be overwritten.

Summary

- Sorting and searching are fundamental operations which are frequently used in many applications.
- Sorting refers to the process of arranging a list of elements in a particular order. The elements are arranged in increasing or decreasing order of their key values.
- In choosing the sorting method, take into account the ways in which the keys will usually be arranged before sorting, the size of application, the amount of time available for programming, the need to save computer time and space, the way in which data structures are implemented, the cost of moving data and cost of comparing keys.

- There are many metrics available for the purpose of comparison. The significant among them are average and worst case complexities in terms of number of comparisons between two records, the number of exchanges, transfer of records, etc.
- The insertion of records in the sorted list defines a class of sorting methods. These methods are used to sort lists when records are obtained dynamically. Some of these methods include shuttle sort, shell sort and insertion sort.
- The sorting methods can also be classified on the basis of selecting the smallest, second smallest and third smallest elements and putting these elements at their proper positions. Selection sort and Bubble sort may come in these sorting methods.
- Merge sort algorithm works on the concept of "divide and conquer", which is one of the most widely applicable and most powerful methods for designing algorithms. It divides the input lists into two equal halves, sorts them and finally merges these sorted lists. The merge sort operations can be done in linear time. The major shortcoming of this approach is that it requires a 0(n) additional space during merge operation.
- The comparison between quick sort and merge sort is normal. The average number of comparisons in merge sort is n log n, the same for quick sort is 1.39 n log n.

Review Exercise

Multiple Choice Questions

1. Average case time complexity of the heap-sort algorithm is:
   a. $0(n \log_2 n)$
   b. 0(n ln n)
   c. $0(n^2)$
   d. $0(n^3)$
2. Worst case time complexity of heap sort algorithm is:
   a. $0(n \log_2 n)$
   b. 0(n ln n)
   c. $0(n^2)$
   d. $0(n^3)$
3. For sorting contiguous lists of records quick sort may be preferred over merge sort because:
   a. it requires less time always.
   b. it does not require extra space for auxiliary memory.
   c. it requires more programming effort.
   d. some programming languages do not support recursion.
4. In quick sort, the desirable choice for the partitioning element will be
   a. first element of the list
   b. last element of the list
   c. median of the list
   d. a randomly chosen element of the list

Hidden page

6. Explain why it is desirable to choose all the increments of the shell sort so that they are relatively prime.
7. Find the expected number of passes, comparisons and exchanges for Bubble sort when the number of elements is equal to '10'. Compare this result with the actual number of the operations when the given sequence is as follows:

   7, 1, 3, 4, 10, 9, 8, 6, 5, 2

8. Write a C function that sorts a file by first applying the radix-sort to the most significant r digits. It then uses insertion sort to sort the entire field.
9. Trace sequential search as it searches for each of the keys present in a list containing three items. Determine how many comparisons are made and thereby check the formula for the average number of comparisons for a successful search.
10. It is required to search an arbitrary array of 'n' elements to find the element closer to a given number. Derive an algorithm based on the principle of linear search. Assuming that given number is not equal to any of the array elements, find the expected number of comparisons.

CHAPTER

12 Graphs

Key Features

- Introduction to Graphs
- Terms Associated with Graphs
- Sequential Representation of Graphs
- Linked Representation of Graphs
- Traversal of Graphs
- Spanning Trees
- Shortest Path
- Application of Graphs

This chapter discusses graphs in detail. Graphs can be regarded as data structures that embody relationships among the data more complicated than those represented by a list or a tree.

Many problems can be described by many to many relation among a set of objects. These problems are best solved by using graphs. Various basic terms and applications of graphs have also been explained in this chapter.

INTRODUCTION TO GRAPHS

A graph is a data structure that is used to represent a relational data, e.g. a set of terminals in a network or a roadmap of all cities in a country. Such complex relationship can only be represented using a graph data structure.

Mathematically, a graph can be defined as a set of two tuples such that G=(V, E) where G is graph, V represents vertices and E indicates the set of edges of graph G.

Consider the graph given below:

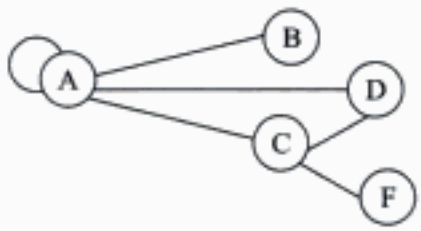


Fig. 12.1 *Graph*

The set of vertices in the above graph is {A, B, C, D, E} and the set of edges will be {(A, B), (A, D), (A, C), (C, D), (C, E)}.

TERMS ASSOCIATED WITH GRAPHS

Directed Graph A directed graph **G** is also called digraph which is the same as multigraph except that each edge **e** in **G** is assigned a direction or in other words each edge in **G** is identified with an ordered pair **(U, V)** of nodes in **G** rather than an unordered pair. Figure 12.2 illustrates three directed graphs:

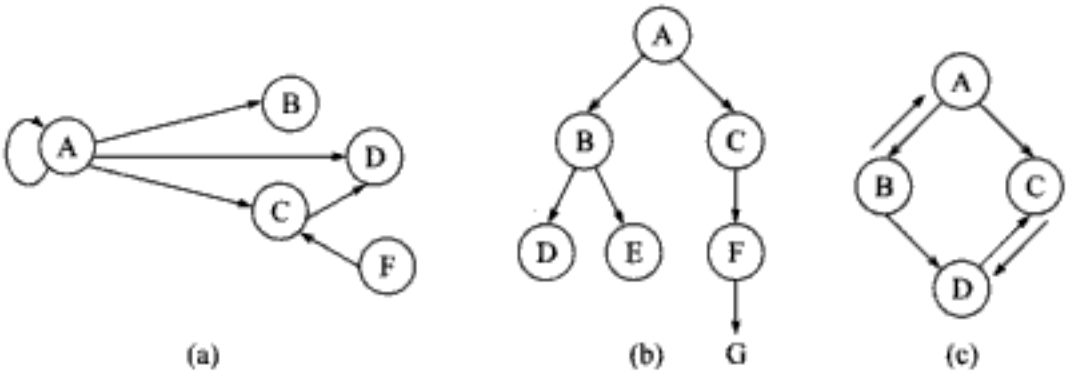


Fig. 12.2 *Directed Graphs*

The set of edges for the graph in Figure12.2(b) is **{<A, B>, <A, C>, <A, D>, <C, D>, <F, C>, <E, G>, <A, A>}**. We use angle brackets to indicate an ordered pair.

A directed graph **G** is said to be **connected** or **strongly connected** if for each pair **(U, V)** of nodes in **G** there is a path from **U** to **V** and there is also a path from **V** to **U**.

A directed graph **G** is said to be **simple** if **G** has no parallel edges. A simple graph **G** may have loops but it cannot have more than one loop at a given node.

Undirected Graph An undirected graph **G** is a graph in which each edge **e** is not assigned a direction. Examples of undirected graphs can be seen in Fig. 12.3 given below:

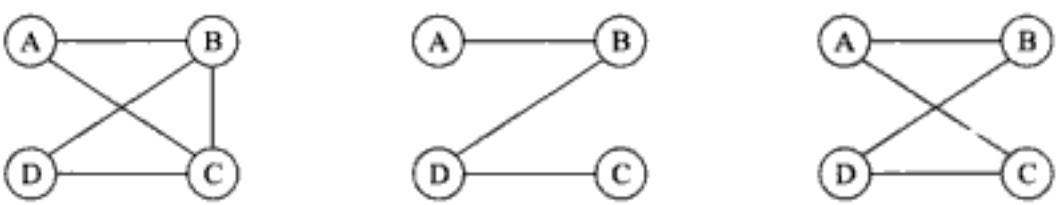


Fig. 12.3 *Undirected Graph*

Connected Graph A graph is called connected if there is a path from any vertex to any other vertex. An example of connected graph can be seen in Fig. 12.4.

A graph **G** is connected if and only if there is a simple path between any two nodes in **G**. A graph **G** is said to be complete if every node **U** in **G** is adjacent to every other node **V** in **G**. A complete graph with **n** nodes has:

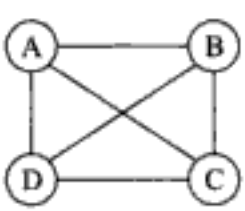


Fig. 12.4 *Connected Graph*

$$n * \frac{(n-1)}{2} \text{edges.}$$

Multiple edges Distinct edges **e** and **e′** are called multiple edges if they connect the same end points, i.e. if **e = (U, V)** then **e′ = (U, V)**.

Loops An edge e is called a **loop** if it has identical end points, i.e. **e=(U, U)**. The definition of a graph generally does not allow multiple edges or loops.

Path A path is a sequence of distinct vertices, each adjacent to the next. Figure 12.5 shows a path.

The **length** of such a path is number of **edges** on that path.

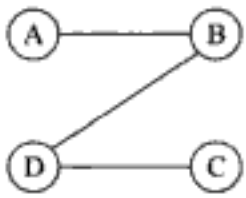


Fig. 12.5 *Path*

Cycle A path from a node to itself is called a **cycle**. Thus, a cycle is a path in which the initial and final vertices are same. For example, consider the graph given in Fig. 12.6.

The path (A, B, C, A) or (A, C, D, B, A) are cycles of different lengths in the graph. If a graph contains a cycle, it is **cyclic**, otherwise it is **acyclic**. A directed graph is also referred to as DAG.

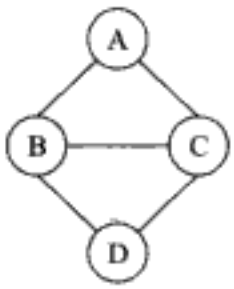


Fig. 12.6 *Cycle*

[Note A graph need not be a tree but a tree must be a graph.]

Therefore, a free tree is defined as a connected undirected graph with no cycles as shown in Fig. 12.7.

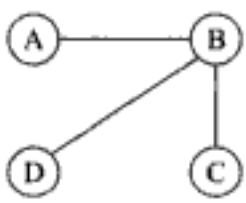


Fig. 12.7 *Free Tree*

Degree, incidence, adjacency A vertex V is **incident** to an edge **e** if **V** is one of the two vertices in the ordered pair of vertices that constitute **e**.

The **degree** of a vertex is the number of edges incident to it.

The **indegree** of a vertex **V** is the number of edges that have **V** as the head and the **outdegree** of vertex **V** is the number of edges that have **V** as the tail [Fig. 12.8].

Figure 12.8 has a vertex **V** which has indegree **1**, outdegree **2** and degree **3**.

A vertex **V** is adjacent to vertex **U** if there is an edge from **U** to **V**. If **V** is adjacent to **U**, **V** is called a **successor** of **U** and **U** a **predecessor** of **V**.

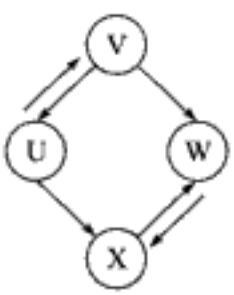


Fig. 12.8 *Indegree of a Vertex*

Weighted Graph A weighted graph is a graph in which edges are assigned weights. This is often required to model certain physical situations by means of a graph. Consider the graph below which has weights. Weights in the graph denote the distance between the vertex connected by the corresponding edges.

Weights of an edge is also called its **cost**. In case of a weighted graph, an edge is a 3-tuple **(U, V, W)**, where **U** and **V** denote vertices connected by an edge and **w** denotes the weight of the edge.

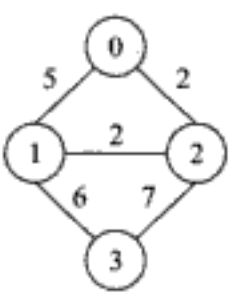


Fig. 12.9 *Weighted Graph*

Sub-Graph and Relation A graph **G′** is called a subgraph of **G=(V, E)** if **V′** is a subset of **V** and **E′** is a subset of **E**. For **G′** to be a subgraph of **G** all the edges and vertices of **G′** should be in **G**.

Hidden page

Adjacency matrices of undirected graphs have a number of interesting properties. Since self loops are not allowed in undirected graphs, the main diagonal contains zeros only. Moreover since **(i, j)** and **(j, i)** represent the same edge in an undirected graph, the adjacency matrix of an undirected graph is symmetric about its main diagonal. Consider an undirected graph given below:

The adjacency matrix for the graph in Fig. 12.12 is as follows:

$$A = \begin{pmatrix} 0 & 1 & 0 & 1 & 0 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{pmatrix}$$

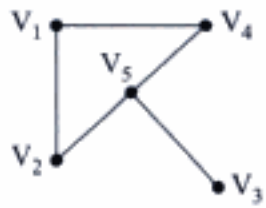


Fig. 12.12 *Undirected Graph*

Path Matrix

Let **G** be a simple directed graph with **m** nodes $V_1, V_2 \ldots V_m$. The path matrix or the reach ability matrix of **G** is the m-square matrix $P = (P_{ij})$ defined as follows:

$$P_{ij} = \begin{cases} 1 \text{ if there is path from } V_i \text{ to } V_j) \\ 0 \text{ otherwise.} \end{cases}$$

Suppose there is a path from V_i to V_j, then there must be a simple path from V_i to V_j or there must be a cycle from V_i to V_j when $V_i = V_j$. Since **G** has only **m** nodes, such a simple path must have length **m–1** or less, or such a cycle must have length **m** or less. This means that there is a non-zero **ij** entry in the matrix B_m.

Incidence Matrix

Let '**G**' be a graph with **n** vertices, '**e**' edges and no self loops. Consider a **(n*e)** matrix $A = (a_{ij})$ whose rows are equivalent to the '**n**' vertices and the columns to '**e**' edges as follows:

$$a_{ij} = \begin{cases} 1 \text{ if } e_i \text{ is incident to Vertex} \\ 0 \text{ otherwise} \end{cases}$$

This kind of a matrix is known as the **incidence matrix** of a graph. Consider the graph given in Fig. 12.13.

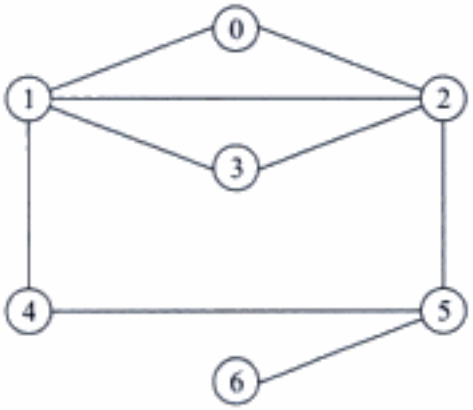


Fig. 12.13 *Graph*

The incidence matrix of above graph will be

	(0, 1)	(0, 2)	(1, 2)	(1, 3)	(1, 4)	(2, 3)	(2, 5)	(4, 5)	(5, 6)
0	1	1	0	0	0	0	0	0	0
1	–1	0	–1	1	1	0	0	0	0
2	0	–1	1	0	0	1	1	0	0
3	0	0	0	–1	0	–1	0	0	0
4	0	0	0	0	–1	0	0	1	0
5	0	–1	0	0	0	0	–1	–1	–1
6	0	0	0	0	0	0	0	0	1

Fig. 12.14 *Incidence Matrix*

The elements 'a_{ij}' of an incidence matrix of a directed graph are defined as follows:

a_{ij} = 1 if 'e_j' is incident out of 'V_i'
= –1 if 'e_j' is incident into 'V_i'
= 0 if 'e_j' is not incident to 'V_j'

Since every edge is incident to exactly two vertices, each column of **A** has two **1s** for undirected graphs. The number of **1s** in a row gives the outdegree and the number of **(–1)** in a row gives the indegree of the vertex in a directed graph.

Edges that are parallel in a graph produce similar columns in its incidence matrix. It may be recalled that an adjacency matrix cannot be used to represent a graph with parallel edges. However, incidence matrix requires more spaces than adjacency matrix as this is a **(n*e)** matrix and usually, **e>n**.

LINKED REPRESENTATION OF GRAPHS

Let **G** be a directed graph with **m** vertices. The sequential representation of **G** in memory, i.e. the representation of **G** by its adjacency matrix **A** has a number of major disadvantages. First of all it may be difficult to insert and delete nodes in **G**. This is because the size of **A** may need to be reordered so, there may be many changes in matrix **A**. Thus, a graph **G** is usually represented in memory by a linked representation.

Consider the graph **G**. Table 12.1 shows each vertex in **G** followed by its adjacency list, which is its list of nodes also called its successors or neighbours. Figure 12.15 shows a diagram of linked representation of **G** in memory. Specifically the linked representation will contain two lists (or files), a node vertex list and an edge list, as follows:

Table 12.1 *Vertex and adjacency list*

Vertex	*Adjacency List*
A	B, C, D
B	C
C	-
D	C, E
E	C

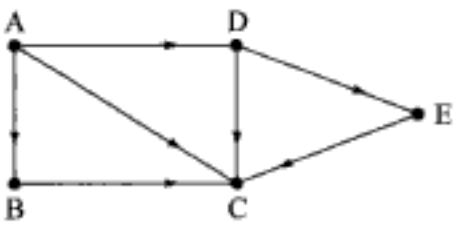


Fig. 12.15 *Linked Representation of a Graph*

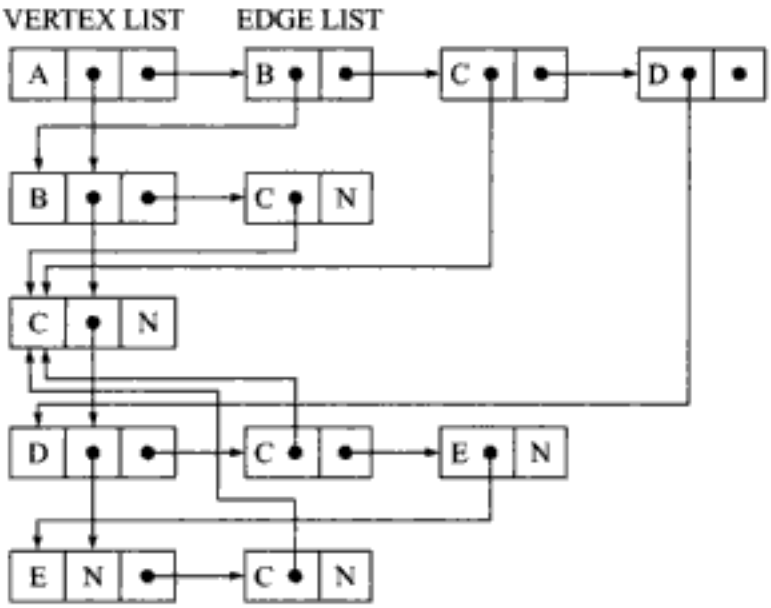
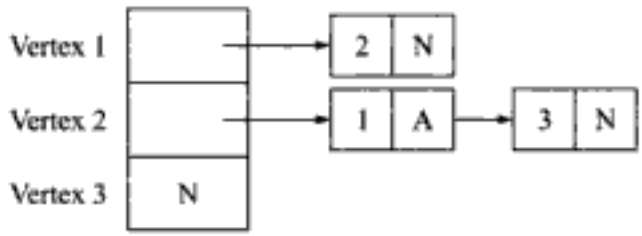


Fig. 12.16 *Linked Representation*

Adjacency List Representation

In this representation, the **n** rows of the adjacency matrix are represented as **n** linked lists. There is one list for each vertex in the graph. The nodes in list **i** represent the vertices that are adjacent from vertex **i**. Each list has a head node. The head nodes are sequential providing easy random access to the adjacency list for any particular vertex. The adjacency list for graphs **G1** and **G2** are shown in Fig. 12.17.

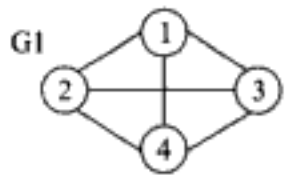


Given a vertex in a directed or undirected graph we may wish to visit all vertices in the graph that are reachable from this vertex. This can be done in two ways—**Depth first search** and **Breadth first search**.

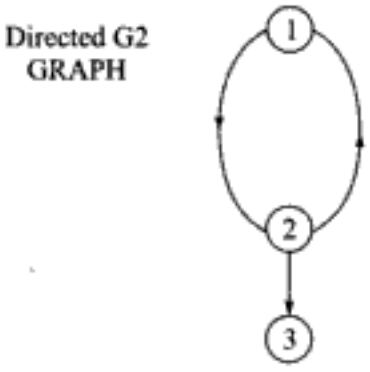
TRAVERSAL OF GRAPHS

In many situations we may intend to examine all the vertices in a graph in some systematic order. Traversal of a graph implies visiting each of its vertices exactly once. The two commonly used techniques are:

- Depth First Search (DFS)
- Breadth First Search (BFS)



Set of vertices = {1, 2, 3, 4}
Set of edges = {(1, 2), (1, 3), (1, 4), (2, 3), (2, 4), (3, 4)}



Set of vertices = {1, 2, 3}
Set of edges = {<1, 2>, <2, 1>, <2, 3>}

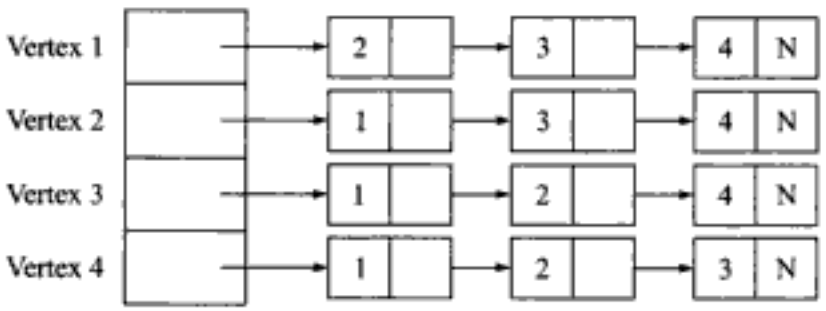


Fig. 12.17 *Adjacency List for G1*

Depth First Search (DFS)

This is a popular technique for systematically traversing the vertices of a graph. This method starts traversing the graph from a given vertex '$\mathbf{V_0}$', that is '$\mathbf{V_0}$' is the first vertex to be visited. The next vertex to be visited is an unvisited vertex adjacent to '$\mathbf{V_0}$'. If '$\mathbf{V_0}$' has a number of unvisited adjacent vertices then any one of them may be selected for visiting next. Once a vertex is visited, it is marked as 'visited' to make the task of finding unvisited adjacent vertices easier. Thus, from any vertex '$\mathbf{V_i}$', a new adjacent unvisited vertex '$\mathbf{V_j}$' is explored. Next, another new vertex adjacent to '$\mathbf{V_j}$' is explored without traversing along other edges incident to '$\mathbf{V_i}$'. When a vertex '$\mathbf{V_p}$' does not have any unvisted adjacent vertex '$\mathbf{V_m}$' then depth first search is initiated from the vertex '$\mathbf{V_m}$'. The process continues until all adjacent vertices of the starting vertex have been visited.

Algorithm for Depth First Search This algorithm executes a depth first search on a graph **G**, beginning at a starting node **A**.

1. Initialize all nodes to the ready state (**STATUS = 1**).
2. Push the starting node **A** onto **STACK** and change its status to the waiting state (**STATUS=2**).
3. Repeat steps 4 and 5 until **STACK** is empty.
4. Pop the top node **N** of **STACK**. Process **N** and change its status to the processed state (**STATUS=3**).

5. Push onto **STACK** all the neighbours of **N** that are still in ready state **(STATUS=1)**, and change their status to the waiting state **(STATUS=2)**.
6. Exit.

Consider the graph **G** given in Fig. 12.18. Suppose we want to find and print all the nodes reachable from the node **J** (including **J** itself). The steps for the Depth—first search will be as follows:

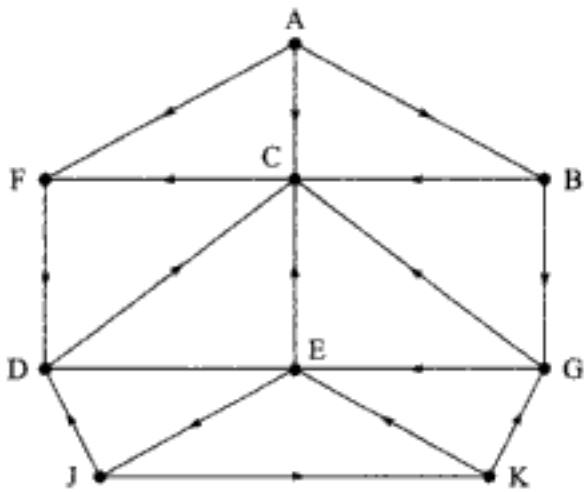


Fig. 12.18

(a) Initially, push **J** onto stack as follows:

STACK : J

(b) Pop and print the top element **J**, and then push onto the stack all the neighbours of **J** (those that are in ready state) as follows:

Print J STACK D, K

(c) Pop and print top element **K**, and then push onto stack all the neighbours of **K** (those that are in ready state) as follows :

Print K STACK D, E, G

(d) Pop and priat the top element **G**, and then push onto stack all the neighbours of **G** (those that are in ready state).

Print G : STACK D, E, C

Note that only **C** is pushed onto the stack, since the other neighbour, **E**, is not in the ready state (because **E** has already been pushed onto the stack).

(e) Pop and print the top element **C** and then push onto the stack all the neighbours of **C** (those that are in ready state) as follows:

Print C STACK D, E, F

(f) Pop and print the top element **F** (those in ready state) as follows:

Print F: STACK D, E

Note that only neighbour **D** of **F** is not pushed onto the stack, since **D** is not in the ready state (because **D** has a already been pushed onto the stack).

(g) Pop and print the top element **E** and push onto the stack all the neighbours of **D** (those in the ready state) as follows:

Print E: STACK: D

Note that none of the three neighbours of **E** is in the ready state.

(h) Pop and print the top element **D**, and push onto the stack all the neighbours of **D** (those in the ready state) as follows:

Print D: STACK: empty

The stack is now empty, so the Depth first search of **G** starting at **J** is now complete. Accordingly, the nodes which were printed,

J, K, G, C, F, E, D

are precisely the nodes which are reachable from **J**.

Analysis of Depth First Algorithm If **G** is represented by its adjacency list then the vertices **W** adjacent to **V** can be determined by following a chain of links.

Since the algorithm DFS would examine each node in the adjacency list at most once and there are **2e** list nodes, the time to complete the search is **0(e)**.

If **G** is represented by its adjacency matrix, then the time to determine all vertices adjacent to **V** is **0(n)**. Since at most **n** vertices are visited, the total time is $\mathbf{0(n^2)}$.

/* Program that implements depth first search algorithm. */

```
#include <stdio.h>
#include <conio.h>
#include <alloc.h>
#define TRUE 1
#define FALSE 0
#define MAX 8
struct node
{
  int data ;
  struct node *next ;
} ;
int visited[MAX] ;
void dfs( int, struct node **) ;
struct node * getnode_write(int) ;
void del(struct node *) ;
void main( )
{
  struct node *arr[MAX] ;
  struct node *v1, *v2, *v3, *v4 ;
```

```
  int i ;
clrscr( ) ;
v1 = getnode_write(2) ;
arr[0] = v1 ;
v1 -> next = v2 = getnode_write ( 3 ) ;
v2 -> next = NULL ;
v1 = getnode_write(1) ;
arr[1] = v1 ;
v1 -> next = v2 = getnode_write(4) ;
v2 -> next = v3 = getnode_write(5) ;
v3 -> next = NULL ;
v1 = getnode_write(1) ;
arr[2] = v1 ;
v1 -> next = v2 = getnode_write(6) ;
v2 -> next = v3 = getnode_write(7) ;
v3 -> next = NULL ;
v1 = getnode_write(2) ;
arr[3] = v1 ;
v1 -> next = v2 = getnode_write(8) ;
v2 -> next = NULL ;
v1 = getnode_write(2) ;
arr[4] = v1 ;
v1 -> next = v2 = getnode_write(8) ;
v2 -> next = NULL ;
v1 = getnode_write(3) ;
arr[5] = v1 ;
v1 -> next = v2 = getnode_write(8) ;
v2 -> next = NULL ;
v1 = getnode_write(3) ;
arr[6] = v1 ;
v1 -> next = v2 = getnode_write(8) ;
v2 -> next = NULL ;
v1 = getnode_write(4) ;
arr[7] = v1 ;
v1 -> next = v2 = getnode_write(5) ;
v2 -> next = v3 = getnode_write(6) ;
v3 -> next = v4 = getnode_write(7) ;
v4 -> next = NULL ;
dfs(1, arr) ;
  for(i = 0 ; i < MAX ; i++)
    del(arr[i]) ;
  getch( ) ;
```

```
}
void dfs(int v, struct node **p)
{
  struct node *q ;
  visited[v - 1] = TRUE ;
  printf("%d\t", v) ;
  q = *(p + v - 1) ;
  while(q != NULL)
  {
       if(visited[q -> data - 1] == FALSE)
              dfs(q -> data, p) ;
       else
              q = q -> next ;
  }
}
struct node * getnode_write(int val)
{
    struct node *newnode ;
  newnode = (struct node *) malloc(sizeof(struct node)) ;
  newnode -> data = val ;
  return newnode ;
}
void del(struct node *n)
{
  struct node *temp ;
    while(n != NULL)
    {
        temp = n -> next ;
        free ( n ) ;
        n = temp ;
    }
}
```

In **main()** the function **getnode_write()** is called several times to create the lists. After creation of each list the address of first node in the list is stored in an element of array **arr** which is array of pointers.

When all the lists stand created the function **dfs()** is called that visits each vertex and marks it as visited by storing a value in an array **visited** which is defined globally.

To deallocate the memory that is dynamically created, a **for** loop is executed by passing the elements of array **arr**.

The graph **G** in figure below is represented by its adjacency list shown in Fig. 12.19. If a depth first search is initiated from vertex V_1, then the vertices of **G** are visible in the order : V_1, V_2, V_4, V_8, V_5, V_6, V_3, V_7.

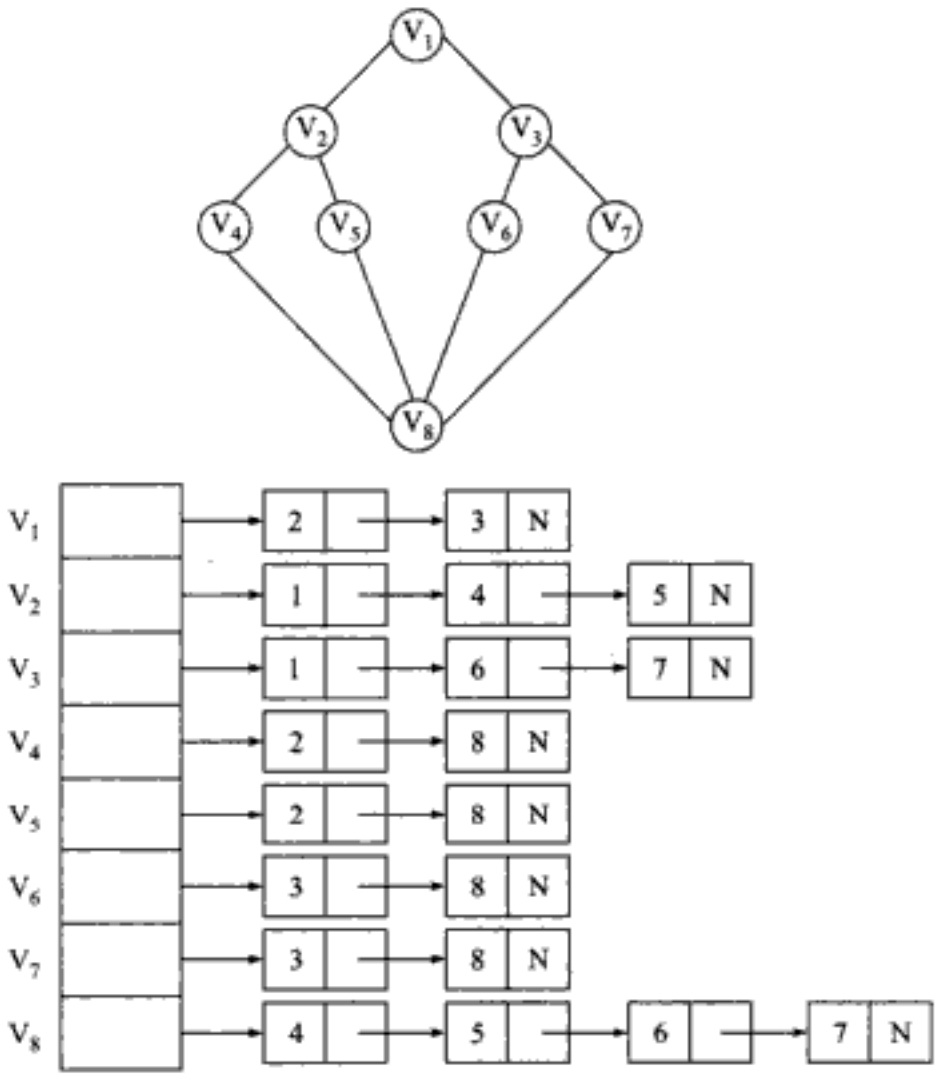


Fig. 12.19 *Graph Represented by Its Adjacency List*

Breadth First Search

This is another popular method to visit the vertices of graph systematically. This method starts from a given vertex 'V_0'. The vertex 'V_0' is marked visited. All adjacent vertices of 'V_0' are visited next. Then one of the adjacent vertices of 'V_p' is taken up and its unvisited adjacent vertices are visited next. The process continues until all vertices reachable from 'V_0' are visited. A breadth first search (BFS) initiated from 'V_i' visits all vertices in 'V_P' where 'V_P' denotes the set of all unvisited adjacent vertices of 'V_i'. Next the process continues from any vertex in 'V_P'. This method continues until all the vertices adjacent to 'V_i' are fully explored. The algorithm for breadth first traversal has to maintain a list of vertices which have already been visited but whose adjacent vertices have not yet been explored. The vertices whose neighbours are yet to be visited can be stored in a queue.

Algorithm for Breadth First Search This algorithm executes a breadth first search on a graph **G** beginning at a starting node **A**.

1. Initialize all nodes to the ready state **(STATUS=1)**.

2. Put the starting node **A** in **QUEUE** and change its status to waiting state **(STATUS=2)**.
3. Repeat steps 4 and 5 until **QUEUE** is empty.
4. Remove the front node **N** of **QUEUE**. Process **N** and change its status to the processed status **(STATUS=3)**.
5. Add to the rear of **QUEUE** all the neighbours of **N** that are in the ready state **(STATUS=1)** and change their status to the waiting state **(STATUS=2)**.
6. Exit.

Consider the graph **G** given in Fig. 12.20. Suppose we want to find all the nodes reachable from **A** to **J**.

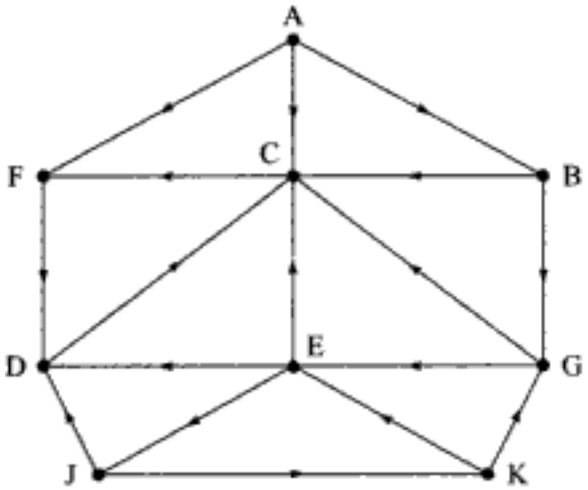


Fig. 12.20

During the execution of search we have kept the track of origin of each edge by using an array **ORIG** together with array **QUEUE**.

a) Initially, add **A** to **QUEUE** and add **NULL** to **ORIG** as follows:

FRONT = 1 QUEUE = A
REAR = 1 ORIG = 0

b) Remove the front element **A** from **QUEUE** by setting FRONT = FRONT+1 and add to **QUEUE** the neighbours of **A** as follows:

FRONT = 2 QUEUE : A, F, C, B
REAR = 4 ORIG : O A A A

c) Remove the front element **F** from **QUEUE** by setting FRONT = FRONT+1 and add to **QUEUE** the neighbours of **F** as follows:

FRONT =3 QUEUE : A, F, C, B, D
REAR = 5 ORIG : O A A A F

d) Remove the front element **C** from **QUEUE** and add to **QUEUE** the neighbours of **C** (which are in ready state) as follows:

FRONT = 4 QUEUE : A, F, C, B, D

REAR = 5 ORIG : O A A A F

Note that the neighbour **F** of **C** is not added to **QUEUE**, since **F** is not in the ready state (because **F** has already been added to **QUEUE**).

e) Remove the front element **B** from **QUEUE**, and add to **QUEUE** the neighbour of **B** (the one in the ready state) as follows:

FRONT = 5 QUEUE : A F C B D G
REAR = 6 ORIG : O, A, A, A, A, F, B

Note that only G is added to QUEUE since the other neighbour, C is not in the ready state.

f) Remove the front element **D** from **QUEUE** and add to **QUEUE** the neighbour of **D** (the one in the ready state) as follows:

FRONT = 6 QUEUE : A, F, C, B, D, G
REAR = 6 ORIG : O, A, A, A, A, F, B

g) Remove the front element **G** from **QUEUE** and add to **QUEUE** the neighbour of **G** (the one in ready state) as follows:

FRONT = 7 QUEUE : A, F, C, B, D, G, E
REAR = 7 ORIG : O, A, A, A, F, B, G

h) Remove the front element **E** from **QUEUE** and add to **QUEUE** the neighbour of **E** (the one in ready state) as follows:

FRONT = 8 QUEUE : A, F, C, B, D, G, E, J
REAR = 8 ORIG : O, A, A, A, F, B, G, E

We stop as soon as **J** is added to **QUEUE**, since **J** is our final destination. We now backtrack from **J**, using the array **ORIG** to find path. Thus,

J ← E ← G ← B ← A

is the required path.

Analysis of Breadth First Algorithm Each vertex visited gets into the queue exactly once, so the loop for ever is integrated at most **n** times.

If an adjacency matrix is used then the loop takes **0(n)** times for each vertex visited.

The total time is therefore $\mathbf{0(n^2)}$.

/* Program that implements breadth first search algorithm. */

```
#include <stdio.h>
#include <conio.h>
#include <alloc.h>
#define TRUE 1
#define FALSE 0
#define MAX 8
struct node
{
```

```
  int data ;
  struct node *next ;
} ;
int visited[MAX] ;
int q[8] ;
int front, rear ;
void bfs(int, struct node **) ;
struct node * getnode_write(int) ;
void addqueue(int) ;
int deletequeue( ) ;int isempty( ) ;
void del(struct node *) ;
void main( )
{
  struct node *arr[MAX] ;
  struct node *v1, *v2, *v3, *v4 ;
    int i ;
    clrscr( ) ;
  v1 = getnode_write(2) ;
  arr[0] = v1 ;
  v1 -> next = v2 = getnode_write(3) ;
  v2 -> next = NULL ;
  v1 = getnode_write(1) ;
  arr[1] = v1 ;
  v1 -> next = v2 = getnode_write(4) ;
  v2 -> next = v3 = getnode_write(5) ;
  v3 -> next = NULL ;
  v1 = getnode_write(1) ;
  arr[2] = v1 ;
  v1 -> next = v2 = getnode_write(6) ;
  v2 -> next = v3 = getnode_write(7) ;
  v3 -> next = NULL ;
   v1 = getnode_write(2) ;
  arr[3] = v1 ;
  v1 -> next = v2 = getnode_write(8) ;
  v2 -> next = NULL ;
  v1 = getnode_write(2) ;
  arr[4] = v1 ;
  v1 -> next = v2 = getnode_write(8) ;
  v2 -> next = NULL ;
  v1 = getnode_write(3) ;
  arr[5] = v1 ;
  v1 -> next = v2 = getnode_write(8) ;
```

Hidden page

Hidden page

```
        n = temp ;
    }
}
```

The working of functions **getnode_write()** and **del()** and arrays **arr[]** and **visited[]** is exactly the same as of program in depth first search.

The function **bfs()** visits each vertex and marks it visited. The function **isempty()**, **addqueue()** and **deletequeue()** are called while maintaining the queue of vertices.

SPANNING TREES

If '**G**' is a weighted graph and '$\mathbf{T_1}$', '$\mathbf{T_2}$' are two spanning trees of '**G**', then the sum of weights of all edges in '$\mathbf{T_1}$' may be different from that of '$\mathbf{T_2}$'. A spanning tree '**T**' of '**G**' where the sum of weights of all edges in '**T**' is minimum is called the **minimal cost spanning tree** or **minimal spanning tree** of **G**.

Trees can be defined as special cases of graphs. A tree may be defined as a connected graph without any cycle. Some examples of graphs as trees are:

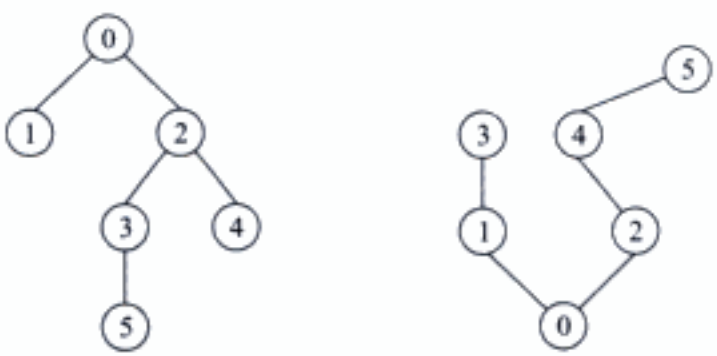


Fig. 12.21 Examples of graphs as trees

The only difference between general trees and trees defined here is that a tree defined as a special case of graph does not have special vertex called root. In fact, any vertex can be chosen as root.

A sub-graph of a graph **G=(V, E)** which is tree containing all vertices of '**V**' is called a spanning tree of '**G**'. If '**G**' is not connected then there is no spanning tree of '**G**'. Now consider the graph **G** given in Fig. 12.22.

Two spanning trees of the above graph are shown in Fig. 12.23.

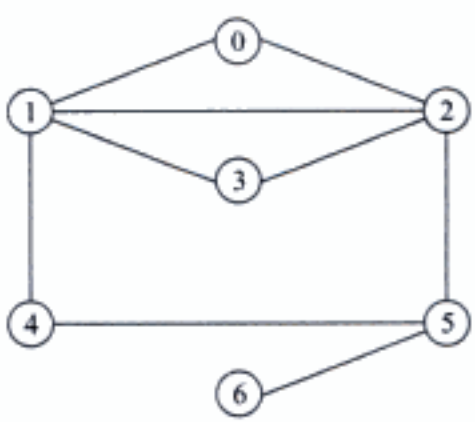


Fig. 12.22

Algorithms for Computing Minimal Spanning Tree

There are two popular algorithms to calculate minimal cost spanning tree of a weighted undirected graph:

Kruskal's Algorithm Kruskal's algorithm functions on a list of edges of a graph where the list is arranged in order of weight of the edges. It begins with a forest of '**m**' trees if the graph has '**m**'

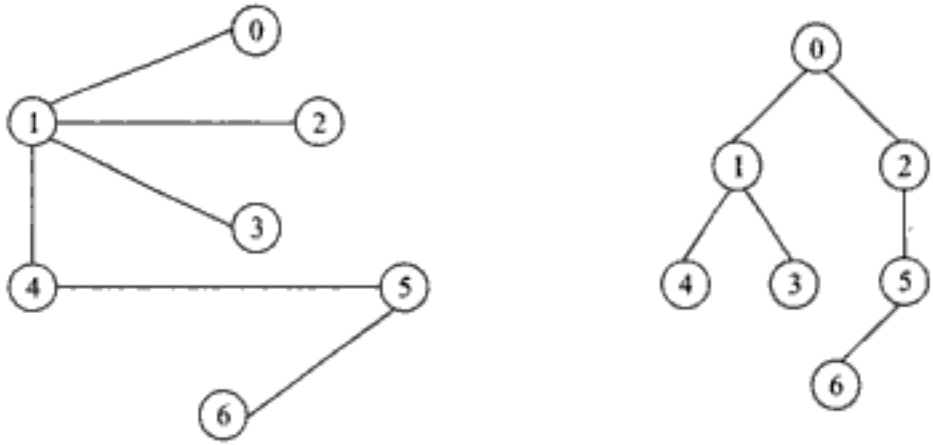


Fig. 12.23 *Spanning Trees*

vertices. The trees in the forest contain a different vertex of the graph and no edges. One edge from the sorted list is taken for connecting two forests in each step of the algorithm. The edge is added if the incorporation does not form a cycle. Once an edge is selected, it is deleted from the list of edges. the algorithm continues until **(n–1)** edges are added to the list of edges exhausted. When the algorithm ends after adding **(n–1)** edges, a minimum spanning tree is produced.

Consider the graph given in Fig. 12.24. There are eleven edges. An edge connecting the vertices '**i**' and '**j**' may be represented by a **tuple(i, j)**.

The list of edges of the graph sorted in non-descending order may be given by {(0, 2), (1, 2), (0, 1), (1, 3), (2, 3), (2, 5), (4, 6), (4, 5), (3, 6), (5, 6), (1, 4)}. The initial forest can be shown by Fig. 12.25.

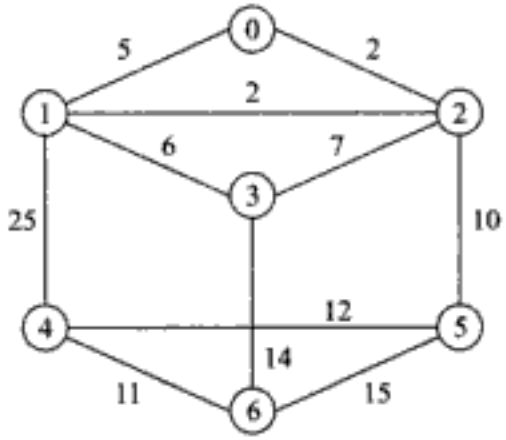


Fig. 12.24

The first edge (0, 2) from the list of edges is taken into account. Since the addition of this edge does not form a cycle, it can be added to change the partial minimum spanning tree as shown in Fig. 12.26.

(1,2) is the next edge to be considered. Inclusion of this edge does not result in a cycle and hence it will be added to the partially formed minimum spanning tree as shown in Fig. 12.27.

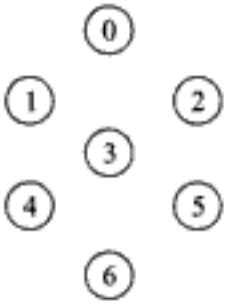


Fig. 12.25

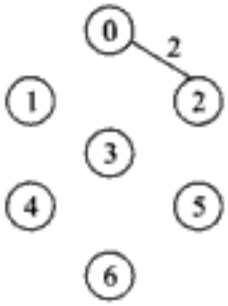


Fig. 12.26

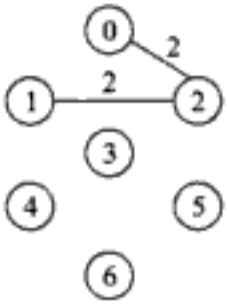


Fig. 12.27

The next edge (0, 1) is not added to the partially formed tree as it forms a cycle. The next edge (1, 3) is now taken into account. Since inclusion of this edge does not form a cycle, this edge can be included and the partial minimum spanning tree is created as in Fig. 12.28.

If the next edge (2, 3) is added then a cycle 2-1-3-2 is formed. Therefore, this edge is not added to the partially formed tree. The next edge (2, 5) is selected. Since inclusion of this edge does not form a cycle, this edge can be added and the partial minimal spanning tree is formed as shown in Fig. 12.29.

Next, the edge (4, 6) is chosen. This edge can be added. Note that the result of each step of the algorithm is not a tree but a forest. In fact, after a successful addition of an edge the number of forests decreases. The number of trees in the forest, after addition of edge (4, 6), is 2. [Fig. 12.30]

The next edge in the list (4, 5) can be added in the forest by merging the two trees creating a minimal cost spanning tree as shown in Fig. 12.31.

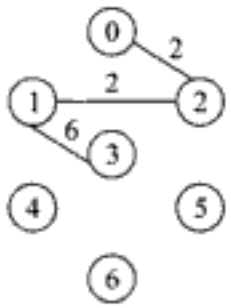


Fig. 12.28

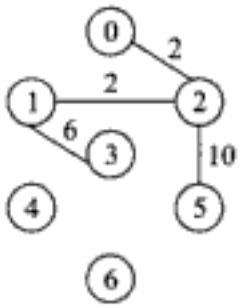


Fig. 12.29

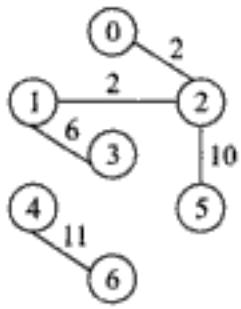


Fig. 12.30

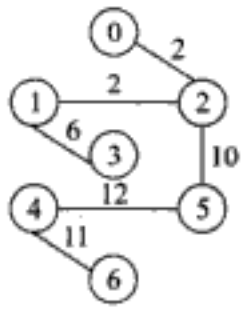


Fig. 12.31

Prim's Algorithm Prim's algorithm starts with any arbitrary vertex as the partial minimal spanning tree '**T**'. In each iteration of algorithm one edge **(U, V)** is added to the partial tree '**T**' so that exactly one end of this edge belongs to the set of vertices in '**T**'. Of all such possible edges, the edge having the least cost is selected. The algorithm continues to add **(n–1)** edges. Consider the graph given in Fig. 12.32(a) in which we start with vertex **0**.

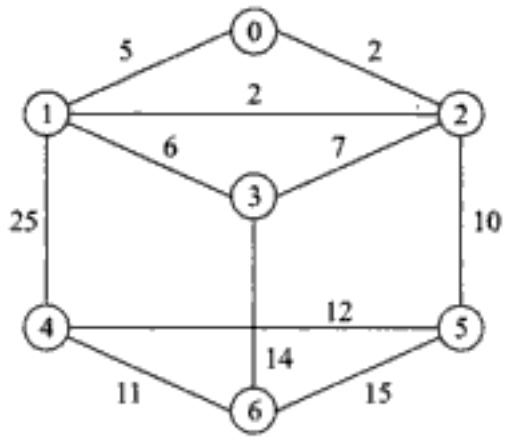


Fig. 12.32(a)

Hidden page

SHORTEST PATH

As a final application of graphs, one requiring somewhat more sophisticated reasoning, we consider the following problem: We are given a directed graph **G** in which every edge has a weight attached, and our problem is to find a path from one vertex **V** to another **W** such that the sum of the weights on the path is as small as possible. We call such a path a **shortest path**.

Length of a path in a weighted graph is defined to be the sum of costs or weights of all edges in that path. In general there could be more than one path between a pair of specified vertices, say "$\mathbf{V_i}$" and "$\mathbf{V_j}$" and a path with the minimum cost or weight is called the **shortest** path from "$\mathbf{V_i}$" to "$\mathbf{V_j}$". Note that the shortest path between two vertices may not be unique.

Consider the weighted undirected graph given in Fig. 12.33.

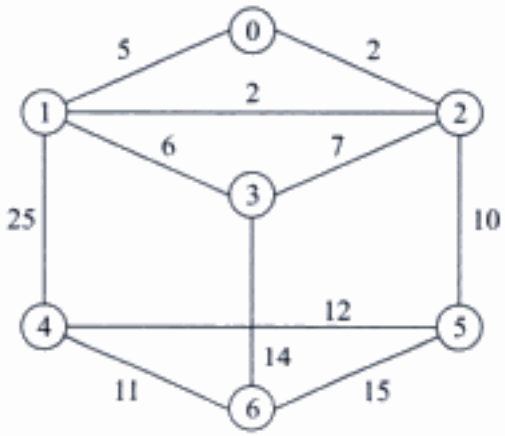


Fig. 12.33

It can be easily seen that there are more than three paths 0–1, 0–2–1, 0–2–3–1, from the vertex '**0**' to the vertex '**1**'. The path with the shortest path length is 0–2–1. The length of the shortest path is 4.

There are many different variations of the shortest path problem. They vary with respect to the specification of the start vertex (referred to as source) and end vertex (referred to as destination). Some of the commonly known variants are listed below:

- The shortest path from a specified source vertex to a specified destination vertex.
- The shortest path from one specified vertex to all other vertices. This problem is also known as single source shortest path problem.
- The shortest path between all possible source and destination vertices. This problem is also known as all pairs shortest path problem.
- The shortest path can be determined using Kruskal's on Prim's algorithm.

APPLICATION OF GRAPHS

Let us assume one input line containing four integers followed by any number of input lines with two integers each. The first integer on the first line, n, represents number of cities which, for simplicity, are numbered from 0 to n – 1. The second and third integers on that line are between 0 to n-1 and represent two cities. It is desired to travel from the first city to second using exactly nr roads, where nr is the fourth integer on the first input line. Each subsequent input line contains two integers representing two

cities, indicating that there is a road from the first city to the second. The problem is to determine whether there is a path of required length by which one can travel from the first of the given cities to the second.

Following is the plan for solution:

Create a graph with the cities as nodes and the roads as arcs. To find the path of length nr from node A to node B, look for a node C such that an arc exists from A to C and a path of length nr – 1 exists from C to B. If these conditions are satisfied for some node C, the desired path exists. If the conditions are not satisfied for any node C, the desired path does not exist.

The traversal of a graph like Depth first traversal has many important applications such as finding the components of a graph, detecting cycles in an undirected graph, determining whether the graph is bi-connected, etc.

Summary

- Graphs provide an excellent way to model the essential features of many applications, thereby facilitating specification of the underlying problems and formulation of algorithms for their solution.
- Graphs may be implemented in many ways—by the use of different kinds of data structures.
- In many applications, edges are to be assigned costs or weights. Such graphs are known as weighted graphs.
- There are various ways of representing a graph. Adjacency matrix of a graph is a matrix representation. Adjacency list of a graph is a linked-list representation.
- There are two popular techniques for graph traversal—breadth first and Depth first.
- Two famous algorithms to complete a minimal spanning tree of a weighted graph are Kruskal's and Prim's algorithm.

Review Exercise

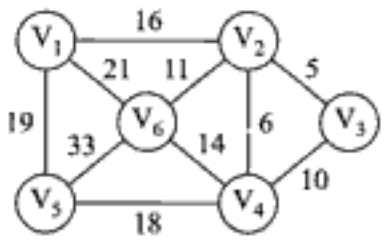
Multiple Choice Questions

1. A vertex with degree one in a graph is called
   a. A leaf
   b. Pendant vertex
   c. Adjacency list
   d. None of the above
2. In an adjacency matrix parallel edges are given by
   a. Similar columns
   b. Similar rows
   c. Not representable
   d. None of the above
3. Breadth-first search
   a. Scans all incident edges before moving to other vertex.
   b. Scans adjacent unvisited vertex as soon as possible.

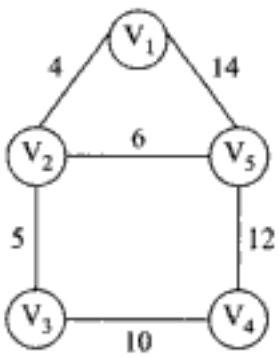
Hidden page

7. Explain Adjacency list representation of the Graphs. Write Adjacency list representation of the graphs of previous question.
8. Write a program that converts Adjacency matrix representation into its adjacency list representation.
9. Explain Adjacency multi list representation of a Graph.
10. What is a path matrix?
11. Explain Warshall's minimal algorithm for finding the path matrix of a graph given its adjacency matrix.
12. What do you mean by traversal of any Graph?
13. Write depth first search algorithm for the traversal of any graph. Write a 'C' program for the same. Explain your algorithm's time complexity with the help of an example.
14. Explain breadth first search algorithm for the traversal of any graph with suitable examples. Define time complexity of the algorithm. Write a "C" program for the same.
15. Another way to represent a graph is by its incidence matrix. There is one row for each vertex and one column for each edge. Then INC(i,j)=1 if edge j is incident to vertex i. Write a program that converts adjacency matrix representation of any graph to its incidence matrix representation.
16. Define spanning tree and minimal spanning tree. Write Kruskal's algorithm for finding minimal spanning tree of any graph. Find the minimal spanning tree of the following graphs by Kruskal's algorithm.

a.



b.



17. Write Prim's algorithm for finding minimal spanning tree of any graph. Find the minimal spanning trees of the graph of previous questions by Prim's algorithm.
18. By considering the complete graph with n vertices, show that the number of spanning trees is at least $2^{n-1}-1$.

CHAPTER

13 Hashing

Key Features

- Hashing—An Introduction
- Hash Functions
- Collision in Hashing
- Collision or Conflict Resolution Techniques
- Open Addressing
- Analysis of Open Addressing
- Chaining
- Analysis of Chaining
- Theoretical Comparison of Hashing Methods
- Empirical Comparison of Hashing Methods

Many comparison based search operations cannot be made better than $\mathbf{0(\log_2 n)}$ time complexity even on the average. For achieving better performance, more efficient search methods are required. Hashing is one such method.

Hashing tries to compute the address where an element is to be inserted or found in a table by applying a hash function to a given key value.

HASHING—AN INTRODUCTION

In various sorting techniques we assumed that the record being sorted is stored in a table and it is necessary to pass through some number of keys before finding the desired one. The search time of each algorithm depends on the number of '**n**' elements in the collection of data. **Hashing** or **hash function** is a technique used to perform a search which is essentially independent of the number of elements, i.e. **n**. Hashing provides a conceptually different mechanism to search a table for a given key value. In hashing, the record corresponding to a given key value is directly referred by calculating its address from key value.

The hashing function has one flaw. Suppose two keys **K1** and **K2** are **h(K1)** and **h(K2)**. Then when a record with key **K1** is entered into the table, it is inserted at position **h(K1)**. But when **K2** is hashed, because it can hash to the same value as that of **K1**, an attempt may be made to insert the record into the same position where the record with key **K1** is already stored. However, the two records cannot occupy the same position. Such a situation is called a **hash collision**. Thus, the objective of a collision resolution strategy is to locate an empty position as efficiently as possible, once the collision occurs.

HASH FUNCTIONS

The two principle criteria used in selecting a hash function are as follows: First, the function should be very easy and quick to compute. Second, the function should, as far as possible, uniformly divide the hash addresses throughout the available memory address range so that there are minimum number of collisions. To compute various hashing functions we assume that there is a file **F** of **n** records with a set **k** of keys which uniquely determine the records in **F**. Secondly, we assume that **F** is maintained in memory by a table **T** of **m** memory locations, and that **L** is a set of memory addresses of the locations in **T**.

Some of the popular hash functions are described below.

Division Method

In this method, we choose a number **m** (i.e. memory locations) larger than the number of **n** (i.e. number of records) of keys in **K** (i.e. keys which uniquely determine records). The number of **m** is usually chosen to be a prime number or a number without small divisors, since this frequently minimizes the number of collisions. The hash function **H** is defined by:

$$\mathbf{H(K) = K(mod\ m)}$$

or

$$\mathbf{H(K) = K(mod\ m) + 1}$$

Here **K (mod m)** denotes the remainder when **K** is divided by **m**. The second formula is used when we want the hash addresses to range from **1** to **m** rather than from **0** to **m–1**.

For example, a company has 68 employees and they have been assigned 4-digit employee number. Assume **L** (memory addresses in the table) of 100 two digit addresses: 00, 01, 02...99. Applying the above hash function, say, for employee numbers:

3205, 7148, 2345

Here, we choose a prime number m close to **99**, such as **97** then —

H(3205) = 4
H(7148) = 67
H(2345) = 17

That is, dividing 3205 by 97 gives a remainder of 4, dividing 7148 by 97 gives a remainder of 67, and dividing 2345 by 97 gives a remainder of 17.

In the other case where the memory addresses begin with 01 rather than 00, we choose the hash function:

H(3205) = 4+1 = 5
H(7148) = 67+1 = 68
H(2345) = 17+1=18

MID Square Method

When the key **K** is squared then the hash function H is defined by

$$\mathbf{H(K) = l}$$

Hidden page

people are independent, the probabilities multiply, and we obtain that the probability that **m** people have different birthday is

$$\frac{364}{365} \times \frac{363}{365} \times \frac{362}{365} \times \ldots \times \frac{365 - m\text{-}1}{365}$$

This expression becomes less than 0.5 whenever $m \geq 24$.

With reference to hashing, the example above tells us that with any problem of reasonable size we are almost certain to have some collisions. Our approach, therefore, should not only be to try to minimize the number of collisions but also to handle those that occur as expeditiously as possible.

COLLISION OR CONFLICT RESOLUTION TECHNIQUES

Every hash function is likely to generate the same address for some set different key values. Suppose we want to add new records **R** with key **K** to our file **F** but suppose the memory location **H(K)** is already occupied. This situation is called **collision**. Therefore, one must know how to deal with such situations. The collision resolution techniques fall in two broad classes:

- Open addressing
- Chaining

OPEN ADDRESSING

Linear Probing

Suppose, that a new record **R** with key **K** is to be added into memory table **T**, but that the memory location with hash addresses **H(K)=h** is already occupied. One natural way to resolve the collision is to assign **R** to the first available location. This collision resolution is called **linear probing**.

Linear probing is easy to implement but it suffers from "primary clustering". A collision at any particular address in the memory indicates that many keys are mapped to the same address. Therefore, all keys mapped at that particular address will be clustered around the slot building up of long run of occupied slots. This would increase the search and insertion time values.

Clustering

One problem with linear probing is that it results in a situation called **clustering**. A good hash function results in a uniform distribution of indexes throughout the array's index range. Therefore, initially records are inserted throughout the arrays, each room equally likely to be filled. Over time, however after a number of collisions have been resolved, the distribution of records in the array becomes less and less uniform. The records tend to cluster together, as multiple keys begin to contend for a single hash function.

Thus, the main disadvantage of linear probing is that records tend to cluster—appear next to one another.

Quadratic Probing

In this method, suppose a record **R** with key **K** has the address **H(K)=h**. Then instead of searching the locations with addresses **h, h+1, h+2...**, we linearly search the locations with addresses -

h, h+1, h+4, h+9, h+16...h+i^2...

Hidden page

For example, consider the diagram given below:

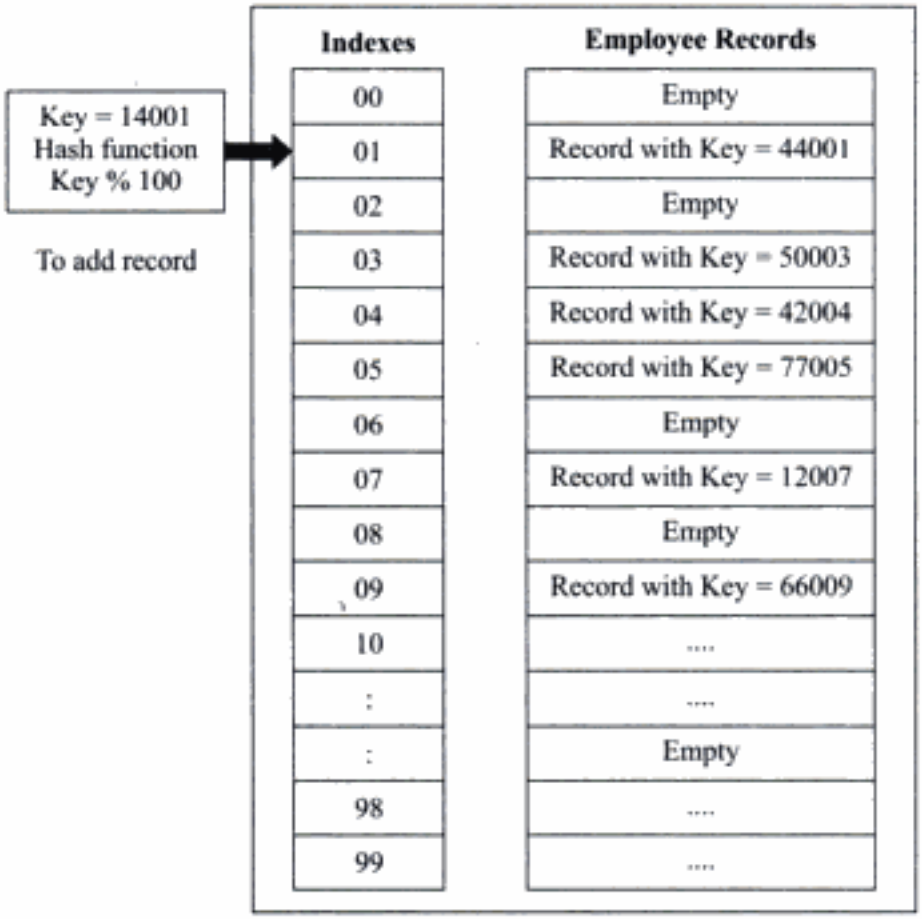


Fig. 13.1 *Rehashing*

Counting Probes

As with other methods of information retrieval, we would like to know how many comparison of keys occur on average during both successful and unsuccessful attempts to locate a record with given target key. We shall use the word **probe** for looking at one item and comparing its key with the target.

The number of probes we need clearly depends on how full the table is. Therefore, (as for searching methods) we let **n** be the number of items in the table, and **t** (which is same as HASHSIZE) be the number of positions in the array. The **load factor** of the table = Load Factor **(λ) = n/t**. Thus, **λ = 0** signifies an empty table; **λ = 0.5** a table is half full. For open addressing λ can never exceed 1, but for chaining there is no limit on the value of **λ**. Hence, we consider chaining and open addressing separately.

ANALYSIS OF OPEN ADDRESSING

For our analysis of number of probes performed in open addressing, let us first ignore the problem of clustering by assuming that not only are the first probes random, but after a collision, the next probe will be random over all remaining positions of the table. In fact, let us assume that the table is so large that all probes can be regarded as independent events.

Let us first study an unsuccessful search. The probability that the first probe hits an occupied cell is λ, the load factor. The probability the probe hits the empty cell is $(1-\lambda)$. The probability that the unsuccessful search terminates in exactly two probes is therefore $\lambda(1-\lambda)$ and similarly the probability that exactly K probes are made in a unsuccessful search is $\lambda^{K-1}(1-\lambda)$. The expected number $U(\lambda)$ of probes in an unsuccessful search is therefore:

$$U(\lambda) = \sum_{K=1}^{\infty} K\lambda^{K-1}(1-\lambda)$$

On further evaluation we obtain:

$$U(\lambda) = \frac{1}{(1-\lambda)^2}(1-\lambda) = \frac{1}{1-\lambda}$$

To count the probes needed for successful search, we note that the number required will be exactly one more than the number of probes in the unsuccessful search made before inserting the item. Now let us consider the table as being empty in the beginning, with each item inserted one at a time. As these items are inserted, the load factor grows slowly from 0 to its final value λ. It is reasonable for us to approximate this step by step growth by continuous growth and replace a sum with integral—we conclude average number of probes in a successful search is approximately

$$S(\lambda) = \frac{1}{\lambda}\int_0^{\lambda} U(\mu)d\mu = \frac{1}{\lambda}\,\text{tn}\,\frac{1}{1-\lambda}$$

Similar calculations may be done for open addressing with linear probing where it is no longer reasonable to assume that successive probes are independent. For linear probing the average number of probes for an unsuccessful search increases to:

$$\frac{1}{2}\left(1+\frac{1}{(1-\lambda)^2}\right)$$

and for successful search number becomes

$$\frac{1}{2}\left(1+\frac{1}{(1-\lambda)}\right)$$

CHAINING

Chaining involves maintaining two tables in the memory. First of all, as before, there is a table **T** in memory which contains the records in **F**. Now the table has an additional field **LINK** which is used so that all records in **T** with the same hash address **h** may be linked together to form a linked list. If a new record, say **R,** has to be added in the file **F** we place **R** in the first available location in the table and a reference or a pointer of **R** is added to the linked list. When the linked list is sorted, then the record is inserted in the beginning. This is also called **Separate Chaining**.

Coalesced Chaining

This technique is identical to linear probe method except that all the keys that hash to the same address are linked together. Each node in the hash table has an additional field that will either hold the address of

Hidden page

the expected number of items—the one being searched is **λ=n/t**. Hence, the average number of probes for an unsuccessful search is **λ**.

Now suppose that the search is successful. From the analysis of sequential search, we know that the average number of comparisons is **1/2(K+1)**, where **K** is the length of chain containing the target record. But the expected length of this chain is no longer **λ**, since we know in advance that it must contain at least one node (the target). The **(n–1)** nodes other than the target are distributed uniformly over all chains; hence the expected number of chains with the target is **1+(n–1)/t**. Except for the tables of trivially small size, we may approximate **(n–1)/t** by **n/t=λ**. Hence, the average number of probes for a successful search is nearly

$$\frac{1}{2}(K+1) \approx \frac{1}{2}(1+\lambda+1) = 1+\frac{1}{2}\lambda$$

THEORETICAL COMPARISON OF HASHING METHODS

Table 13.1 gives the value of the above expressions for different values of load factor.

Table 13.1 *Theoretical Comparison of Hashing Methods*

Load factor	0.10	0.50	0.80	0.–90	0.99	2.00
Successful search, expected number of probes.						
Chaining	1.05	1.25	1.40	1.45	1.50	2.00
Open, Random Probes	1.05	1.4	2.0	2.6	4.6	–
Open, Linear Probes	1.06	1.5	3.0	5.5	50.5	–
Unsuccesful search, expected number of probes.						
Chaining	0.10	0.50	0.80	0.90	0.99	2.00
Open, Random Probes	1.1	2.0	5.0	10.0	100	–
Open, Linear Probes	1.12	2.5	13	50	5000	–

Several conclusions can be drawn from Table 13.1. First, it is clear that chaining requires fewer probes than addressing. On the other hand, traversal of linked list is usually slower than array access, which can reduce the advantage specially if key comparison can be done quickly. Chaining comes into own when the records are large and comparisons of keys takes significant time. Chaining is especially advantageous when unsuccessful searches are common since with chaining an empty list or very short list may be found, so that often no key comparisons need to be done at all to show that a search is unsuccessful.

With open addressing and successful searches the simpler method of linear probing is not significantly slower than more sophisticated methods, at least until the table is almost completely full. For unsuccessful searches, however, clustering quickly causes linear probing to degenerate into long sequential search. We might conclude therefore, that if searches are quite likely to be successful and the load factor is moderate then linear probing is quite satisfactory, but in another circumstances other methods must be used.

EMPIRICAL COMPARISON OF HASHING METHODS

It is important to remember that the computations given in earlier table are only approximate and also that in practice nothing is completely random so that we can always expect some difference between the theoretical computations and the actual results. For the sake of comparisons, therefore, the table below gives the results of one empirical study, using 900 keys that are pseudorandom numbers between 0 and 1.

Table 13.2 *Results of Empirical Study*

Load Factor	0.1	0.5	0.8	0.9	0.99	2.0
Successful search	average number of probes					
Chaining	1.04	1.2	1.4	1.4	1.5	2.0
Open, Quadratic Probes	1.04	1.5	2.1	2.7	5.2	–
Open, Linear Probes	1.05	1.6	3.4	6.2	21.3	–
Load Factor	0.1	0.5	0.8	0.9	0.99	2.0
Unsuccessful search	average number of probes					
Chaining	0.10	0.50	0.80	0.90	0.99	2.00
Open, Quadratic Probes	1.13	2.2	5.2	11.9	126	–
Open, Linear Probes	1.13	2.7	15.4	59.8	430	–

In comparison with other methods of information retrieval, the important thing to note about all these numbers is that they depend only on the load factor, not on the absolute number of items in the table. Retrieval from a hash table with 20,000 items in 40,000 possible positions is no slower, on average, than is retrieval from a table with 20 items in 40 possible positions. With sequential search a list 1000 times the size will take 1000 times as long to search. With binary search this ratio is reduced to 10 (more precisely to lg 1000) but still the time needed increases with the size, which it does not with hashing.

Finally, we should emphasize the importance of devising a good hash function—one that executes quickly and maximizes the speed of keys. If the hash function is poor, the performance of hashing can degenerate to that of sequential search.

Summary

- A hash table is a data structure that is often used to implement many abstract data types or, DICTIONARY ADT.
- Hashing is a process of computing the address where an element is to be inserted or found in a table by applying a hash function to a given key value.
- Many different hash functions and collision resolution techniques are available like division method, folding, etc., which distribute the keys uniformly to the slots of the hash table.
- There are two broad categories of collision resolution techniques—open addressing and chaining. Some of the important open addressing methods include linear and quadratic probing, double hashing, etc. Important chaining methods include separate chaining, coalesced chaining, and bucket chaining.

Review Exercise

Multiple Choice Questions

1. What is not true about hashing?
   a. Also called KAT
   b. It is an algorithm producing address given a key
   c. ISAM is a technique for hashing
   d. All are true
2. What is not true about collision?
   a. It occurs when two or more keys produce same address
   b. Every hashing algorithm may produce it
   c. Another hashing function may be used to resolve collision
   d. All are true
3. Folding is a method of generating
   a. index function for a triangular matrix
   b. header node for a circularly linked list
   c. a hash function
   d. None of the above
4. The technique of linear probing for collision resolution can lead to
   a. clustering
   b. radix sort
   c. efficient storage utilization
   d. overflow
5. In which collision processing method, it is not required to detect a given list position if it is occupied or not.
   a. Quadratic
   b. Linked
   c. Rehashing
   d. None of the above

Fill in the Blanks

1. Application of a hash function, to a record key, results in a value known as a _____ (hash key / hash index).
2. Hashing takes key as input and gives _____ of key as output (address / value).
3. Division remainder method is a _____ technique (collision / hashing).
4. _____ occurs when hashing produces same address for two different keys (collision / probing)
5. In _____ collision processing method the key causing collision is kept at first vacant position (linear / quadratic).

State whether True or False

1. Hashing is also known as KAT.
2. Division remainder method of hashing uses a prime number to arrive at address.
3. Folding method of hashing rules out the possibility of collision.
4. There is no hashing method without the possibility of collision.
5. Multiple hashing uses more than one hashing function to resolve collision.

CHAPTER

14 String Processing

Key Features

- Introduction to Strings
- Representation of Strings through Arrays
- Representation of Strings through Linked Lists
- String as an ADT
- String Operations
- Pattern Matching Algorithms
- Improvements on the Pattern Matching Algorithms

A wide variety of applications from different domains such as word processing, computer graphics, etc. involve manipulation of text. Text can be treated as a stream of symbols and is often referred to as a string.

Therefore efficient representation and handling of strings play an important role in data structure.

INTRODUCTION TO STRINGS

A **string** is defined as a list whose entries are characters. Examples of strings are—**'This is a string'** or **'Name'**. Thus a finite sequence S of zero or more characters is called a string. An empty string is denoted by' '. A string may be represented by an array or linked list.

REPRESENTATION OF STRINGS THROUGH ARRAYS

A string of characters having length 'n' can be implemented by a one dimensional array of 'n' elements where the i^{th} element of the array stores the i^{th} element of the string. Type declaration for such a string would be:

```
typedef struct string
{
     int length;
     char str[1000];
} string;
```

In such a representation there must be a defined maximum length of the strings to be manipulated. The actual length of a string varies and can only be known during execution. But in this representation the maximum length has to be specified. If the maximum length is too big, then substantial wastage of

memory occurs. On the other hand if the maximum length is small, then it will not be possible to store strings with lengths more than the maximum specified.

The advantage of such a representation is that the array elements may be directly accessed and this could speed up several operations on strings.

REPRESENTATION OF STRINGS THROUGH LINKED LISTS

An alternative mode of representation of strings is through linked lists. A string represented by a linked list is identified by a pointer to the linked list. Each node in the list contains symbol and a pointer to the next symbol in the string. The pointer of the node containing the last symbol in the string is set to null. Thus, the string points to the first symbol and the list is to be traversed to get all of its symbols. In this case a string may be declared as

```
typedef struct string
{
    char symbol;
    struct string *next;
} string;
```

Consider the string "abbacabca". The array representation of the string is shown in Fig. 14.1.

Fig. 14.1 *Array Representation of the String*

The linked list representation of the string is shown in Fig. 14.2.

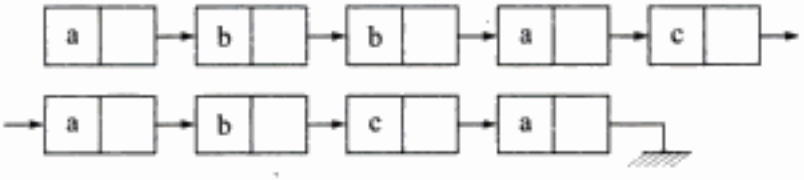


Fig. 14.2 *Linked List Representation of the String*

STRING AS AN ADT

The string with zero characters is called an empty string or the null string. Specific strings will be denoted by enclosing their characters in single quotation marks. The quotation marks will also serve as string delimiters. Hence,

'THE END' 'TO BE OR NOT TO BE'

are strings of 7 and 18 lengths respectively.

There are several operations which typically arise in the context of string manipulation.

STRING OPERATIONS

Although a string may be viewed simply as a sequential or linear array of characters, there is a fundamental difference in use between strings and other types of arrays. Specifically, groups of consecutive elements in a string (such as words, phrases and sentences), called **substrings**, may be units in themselves.

Hidden page

Hidden page

For example,

LENGTH('COMPUTER') = 8

/* Program to find out the length of a given string */

```
main()
{
  char str[20];
  int i, length=0;
  printf("Enter the string :");
  scanf("%s, str);
        for(i=0; str[i] != '\0'; i++)
              length++;
  printf("Length of string is %d\n", length);
}
```

PATTERN MATCHING ALGORITHMS

There are many pattern matching algorithms to decide whether or not a given string pattern **P** appears in a string text **T**. We assume that the length of **P** does not exceed the length of **T**.

Brute Force Algorithm

This is one of the most simple algorithms for string searching. It starts by scanning the text from its $\mathbf{0^{th}}$ index and checks whether pattern matches the text from that position. To check whether match occurs or not, all the first "**m**" characters in the string from its 0^{th} position are compared one by one to all "**m**" characters of the pattern. During these comparisons if $\mathbf{(0 + k)^{th}}$ character of the text is not the same as the $\mathbf{k^{th}}$ character of the pattern $(\mathbf{0 \leq k \leq m})$ then a match cannot occur from the 0^{th} position of the string. It then becomes necessary to find the match starting with 1^{st} position or index of the string. This process continues until a match is found starting from the $\mathbf{i^{th}}$ position of the text or till the text is exhausted. The latter case indicates that the pattern is non-existent in the text.

The following 'C' function is based on the brute force algorithm. The parameters "**s**" and "**p**" are declared as array of characters—"**s**" denotes the text and "**p**" denotes the pattern.

/* Pattern searching using Brute Force algorithm */

```
int bruteStringSearch(char *s, char *p)
{
     int i, j, k, m, n;
           n = strlen(s);
           m = strlen(p);
     for(i=0; i<=n-m; i++)
     {
           j=0;
           k=i;
           while((s[k] == p[j]) && (j<m))
```

```
            {
                k++;
                j++
            }
            if(j==m)
            return i;
        }
        return(-1)
    }
```

Consider that **s** = "xxxxxx" and '**p**' be "xxxy". It is clear that starting from any position of the text one has to make all four character comparisons to detect a mismatch. This is because the mismatch is detected only after the last character of the pattern is compared to the corresponding character of the string. In this example, exactly **(n–m+1) * m** number of comparisons are required where '**n**' and '**m**' are lengths of text and pattern respectively.

In practice however, a mismatch will be detected much earlier than all "**m**" comparisons which will reduce the average case complexity of this algorithm.

Knuth-Morris Pratt Algorithm

Knuth and Pratt developed another algorithm for string searching. The algorithm runs in **0(n+m)** worst case time which is much better than the worst case complexity of the Brute-Force algorithm.

Knuth-Morris algorithm avoids many redundant comparisons, which brute force algorithm would have performed. Consider a string "xyyyyx" and the pattern "xyyxy". At first, the brute force method tries to match the pattern "xyyxy" with the text beginning from the 0^{th} index. The comparison between s[3] and p[3] results in a mismatch. Once this mismatch occurs the Brute Force algorithm would blindly proceed to check for a possible match starting from the index one to the text. So the next comparison to be performed by the Brute Force method is between the 0^{th} character of the pattern (i.e. 'X') and the 1^{st} character of the string (i.e., 'y').

Considering the available information at the end of the 0^{th} trial (starting from the 0^{th} index of the text), this comparison becomes unnecessary. After the comparisons made in the initial trial, it is already known that the first three characters of the pattern are the same as the first three characters of the string (as the fourth comparison resulted in a mismatch).

Moreover, since the pattern is known, it is also known that the first two characters in it ('x' and 'y' respectively) are distinct. Using this knowledge about the pattern and the knowledge that the first three characters of the text have matched with those of the pattern, it can be safely concluded that the first two characters in the string must be distinct (as they are 'x' & 'y' respectively). So the 0^{th} character in the pattern must be distinct from the 1^{st} character of the text.

Therefore, without making any more character comparison, it can be concluded that the trial starting from the 1^{st} position of the text is bound to result in a mismatch.

The Brute Force algorithm does not attempt to make use of such acquired knowledge. It would blindly go for trial from the index one of the text and will detect a mismatch only after making at least one comparison (between the 1^{st} character of the string and the 0^{th} character of the pattern).

/* Function declared for the Knuth-Morris Pratt algorithm */

```
int KMP(char *s, char *p)
```

```
{
    int i, j, m, n;
    n = strlen(s)
    m = strlen(p);
    for(i=0; j=0; i<n && j<m; i++; j++)
    while((j>=0) && (s[i]!=p[j]))
          j = next[j];
    if(j==m)
          return i-m;
    else
    return -1
}
```

Analysis of Brute Force and Knuth-Morris Pratt Algorithm

Searching for a pattern in a text is an important and significant problem. The simple Brute Force algorithm which accomplishes this task is computationally expensive. If '**m**' and '**n**' are lengths of the pattern and the text respectively then the time complexity of Brute Force string searching algorithm is **0(mn)**.

Knuth-Morris Pratt algorithm examines the pattern to exploit periodically the substrings in the pattern. This algorithm first computes a "next" table and then uses this table to search the pattern in a text. The time complexity of KMP algorithm is **0(m+n)**.

IMPROVEMENTS ON THE PATTERN MATCHING ALGORITHMS

There are several improvements that can be effected:

- To avoid the situation where length of the pattern is greater than the remaining length of the string but the algorithm is still searching for a match.
- To check whether first and last characters of pattern match with those in string before checking the remaining characters of pattern.

A significant improvement of performance may be achieved if the pattern is matched against the string from right to left. On a mismatch the pattern may be shifted right possibly skipping few trials.

This idea leads to a right to left Brute Force string searching algorithm. Combining these algorithms with a right to left version of KMP algorithm yields a much better algorithm known as **Boyer-Moore algorithm**. Although the worst case time complexity of Boyer-Moore algorithm 0(m+n), its average complexity is 0(n/m).

Boyer-Moore Algorithm

Let '**s**' be "abaabcabbcbabbababb" and '**p**' be "bababb". Let '**x**' and '**y**' store the lengths of '**p**' and '**s**' respectively. So, x=6 and y=19.

In this algorithm, the occurrence of a pattern is searched from right to left. This, the 0^{th} match starts by comparing **p(x–1)** with **s(x–1)** and continues to the left until the comparison fails or **p[0]** becomes equal to **s[0]**.

In our example, the 0^{th} match starts by comparing **s[5]** with **p[5]** as x=5. As s[5]="c" and p[5] = 'b', the comparison fails. Not only this, all matches starting from index '0' to index '**5**' of '**s**' will fail. This

Hidden page

```
while(j!= 0)
  {
      p = PAT;
      r = save;
      if(q → data == j → data)
            {
                  while(p → data == r → data && p != q)
                        {
                              p = p → link;
                              r = r → link;
                        }
                        If(p == q)
                        {
                              i = save;
                              return;
                        }
                              save = save → link;
                              j = j → link;
            }
  }
```

Analysis of Algorithm

If we apply NFIND to the strings **S** = 'aa ... a'; and **PAT** = 'a ... ab';, then the computing time for these inputs is **O(m)** where **m** = length(s) which is far better than **FIND** which required **O(mn)**. However, the worst case computing time is **O(mn)**.

Fig. 14.4 *Action of NFIND on S and PAT*

Hidden page

CHAPTER

15 Storage Management

Key Features

- Dynamic Storage Management—An Introduction
- Compaction of Blocks of Storage
- First-Fit Method
- Best-Fit Method
- Worst-Fit Method
- Comparison between First-Fit and Best-Fit Methods
- Boundary Tag Method
- Buddy System
- Garbage Collection

In the earlier chapters we made use of data structures that required some form of memory management in order to handle requests for allocation and release of memory. When the lists and tree structures were discussed, we saw that nodes were simply created but it was not considered how these nodes will be released when they were no longer needed.

This chapter discusses some of the techniques and algorithms that can be used to provide various levels of storage management and control compatibility.

DYNAMIC STORAGE MANAGEMENT—AN INTRODUCTION

In many cases, we assumed that storage is allocated and freed one node at a time. There are two characteristics of nodes that make the above methods suitable. The first is that each node of a given type is of fixed size and the second is that the size of each node is fairly small. But in some cases a program might require a large amount of contiguous storage which would be difficult to obtain one block at a time, or a program may require storage blocks in a large variety of sizes. In such cases, memory management system must be able to process requests for variable length blocks. Thus processing certain types of data structures efficiently requires dynamic storage management capability.

Consider a small memory of 1024 bytes. Suppose a request is made for three blocks of storage of 348, 100 and 212 bytes. Let us assume that these blocks are allocated sequentially.

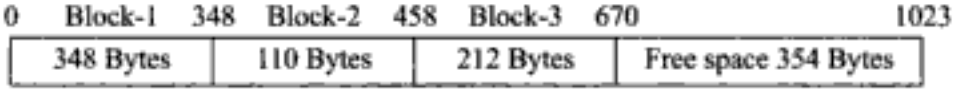


Fig. 15.1

Now let us suppose that the second block of 110 bytes is set free. Although there are now 464 bytes of free space, a request of 400 bytes could not be satisfied since the free space is divided into non-contiguous blocks.

If block three is set to be free, it is not desirable to retain three free blocks of 110, 212 and 354 bytes. Rather the block can be combined into a single large block of 676 bytes so that large memory requests can be further satisfied. After combination they appear in two fragments of that of size 348 bytes and 676 bytes respectively.

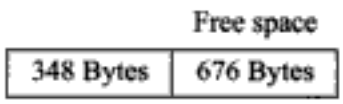


Fig. 15.2

COMPACTION OF BLOCKS OF STORAGE

Initially, memory is a large block of available storage. As requests for storage come, blocks of memory are allocated sequentially, starting from the first location of the memory. A variable **freepoint** is used to store the address of the first location following the last allocated block.

When a block of size **n** is allocated, freepoints are increased by '**n**'. This continues until a block of size **n** is requested and thus **freepoint + (n–1)** is the highest address of the memory.

For compaction, a block of storage routine or system compaction routine is brought into action and all other routines are terminated. Such a routine copies all allocated blocks into sequential memory locations starting from lowest address in the memory. Thus, all free blocks which were interspersed are eliminated and a freepoint is set to the sum of size of all allocated blocks. When such a technique is used, special care must be taken to ensure that the pointer value is correct. Therefore, in order to have successful compaction there must be a method to determine if contents in a given location are addresses.

A compaction routine requires a method that can compute address as offset from the base address. Compaction routine also requires a method by which size of the block and its status could be determined.

There are two drawbacks of compaction routine:

- This technique stops all other user processing while the compaction of blocks takes place.
- There is problem of pointer maintenance.

Consider a memory as fragmented below:

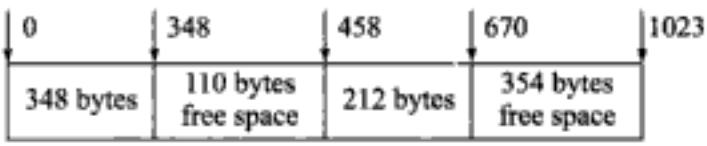


Fig. 15.3

Now, if a request is made for a block of 250 bytes the free space of 354 bytes or the location 670 through 920 would be used.

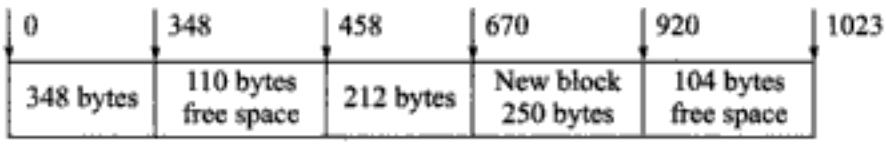


Fig. 15.4

Again, if we consider the memory before allocating space for new block and now if a request for a block of 50 bytes comes, the request could be satisfied by either the block of 348 bytes or the block of size 110 bytes or the block of size 250 bytes. In each, a part of a free block becomes allocated, leaving the remaining portion free.

Every time that a request is made for storage, a free area large enough to accommodate the size requested must be allocated. The most obvious method for keeping track of the free blocks is to use a **linear linked list**. Each free block contains a field containing the size of the block and a field containing a pointer to the next free block. These fields are in some uniform location (say, the first two words) in the block. If **p** is the address of free block, the expression **size(p)** and **next(p)** are used to refer to these two quantities. A global pointer to **freeblock** points to the first free block on this list. Let us see how blocks are added on to this list when they are freed.

Consider the situation in the figure given below:

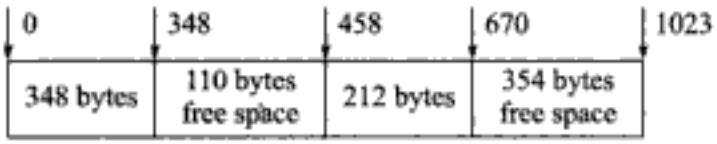


Fig. 15.5

Figure 15.5, when reproduced to show the free list can be seen from Fig. 15.6.

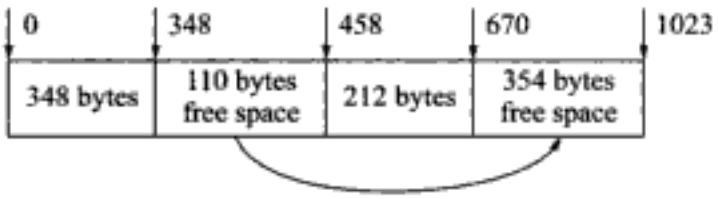


Fig. 15.6

There are several methods of selecting the free block to use when processing a request for storage. Important among these methods are:

- First Fit Method
- Best Fit Method
- Worst Fit Method

FIRST-FIT METHOD

In the first fit method, the free list is traversed sequentially to find the first free block whose size is larger than or equal to the amount of memory requested. Once the block is found, it is allocated to the requesting application (if it is equal in size to the amount requested) or is split into two portions (if it is greater than the amount requested). The first of these portions remains on the list and the second is allocated. The reason for allocating the second portion rather than first is that the free list **next** pointer is at the beginning of each free block. By leaving the first portion of the block on the free list, this pointer need not be copied to some other location and the **next** field of the previous block in the list need not be changed.

Hidden page

Hidden page

Whereas under best-fit method the block of size 54 is split as shown in Fig. 15.10.

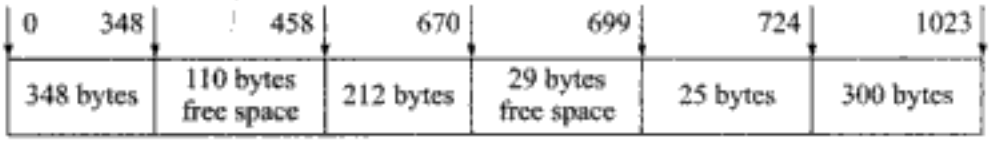


Fig. 15.10

Now again, if a block of size 100 is requested, the request can be fulfilled under best-fit, since a block of size 110 is available, but it cannot be fulfilled under first-fit. This illustrates an advantage of the best-fit method, in that large free blocks remain unsplit so that requests for large blocks can be satisfied. In the first-fit method, a very large block of free storage at the beginning of free list is nibbled away by small requests so that it is severely shrunken by the time large requests arrive.

However, there may be some conditions where the first-fit method succeeds and best-fit fails. For example, consider a case in which system begins with free block of size 110 and 54 and then makes successive requests for 25, 70 and 50 words.

Figure 15.11 illustrates that first-fit method succeeds in fulfilling these requests whereas the best-fit does not. The reason is that remaining unallocated portions of blocks are smaller under best-fit than under first-fit.

	Blocks remaining	
Request	First-fit	Best-fit
Initially	110, 54	110, 54
25	85, 54	110, 29
70	15, 54	40, 29
50	15, 4	Cannot be fulfilled

Fig. 15.11

WORST-FIT METHOD

Another method of allocating block of storage is the **worst-fit method**. In this method, the system always allocates a portion of largest free block in memory. The basic idea behind this method is that by using a small number of very large blocks repeatedly to satisfy the majority of requests, many moderately sized blocks will be left unfragmented. Thus, this method is likely to satisfy a larger number of requests as compared to the other methods. For example, if memory initially consists of blocks of sizes 200, 300 and 100 the sequence of requests 150, 100, 125, 100 and can be satisfied by worst-fit method but neither by the first-fit nor by the best-fit method.

The major reason for choosing one method over the other is efficiency. In each of the methods the search can be made more efficient. For example, a true first-fit which allocates the block of lowest memory address first, will be most efficient if the available list is maintained in the order of increasing memory address. On the other hand, if the available list is maintained in the order of increasing size, a best-fit search for a block becomes more efficient. And finally, if the list is maintained in order of

decreasing size, a worst-fit search requires no searching as the largest block is always first on the list.

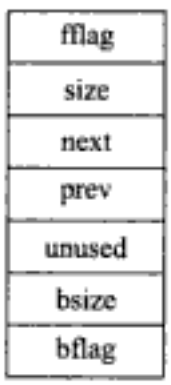
BOUNDARY TAG METHOD

It is desirable to eliminate all searching during liberation to make the process more efficient. One method of doing this comes at the expense of keeping extra information in all blocks (both free and allocated).

A search is necessary during liberation to determine if the newly freed block may be combined with some existing free block. There is no way of detecting whether such a block exists or which block it is, without a search. However, if such a block exists, it must immediately proceed or succeed the block being freed. The first address of block that follows the block of size **n** at **alloc** is **alloc+n**. Suppose that every block contains a field **flag** that is true if the block is allocated and **free** if the block is free. Then by examining **flag (alloc+n)**, it can be determined whether or not the block immediately following the block at **alloc** is free.

It is more difficult to determine the status of the block immediately preceding the block at **alloc**. The address of last location of that preceding block is of course **alloc-1**. But there is no way of finding the first location without knowing its size. Suppose each block contains two flags **fflag** and **bflag**. Both of these are **true** if a block is allocated and **false** otherwise. The **fflag** is at a specific offset from the front of the block and **bflag** is at a specific negative offset from the back of the block. To access both **fflag** and **bflag,** the first and last location of the block must be known. The status of the block must be known. The status of the block following the block at **alloc** can be determined from the value of **fflag (alloc+n)** and the status of block preceding the block at **alloc** can be determined from the value **bflag (alloc-1)**.

A list of free blocks is still needed for the allocation process. When a block is freed, its neighbours are examined. If both blocks are allocated, the block can simply be appended to the front of the free list. If one (or both) of its neighbours is free, the neighbours can be removed from the free list combined with the newly freed block, and the newly created large block can be placed at the head of the free-list. This could reduce the search time under first-fit allocation as well, since a previously allocated block is likely to be large enough to satisfy the next allocation request.



FREE BLOCK

To remove an arbitrary block from the free list (to combine it with a newly freed block) without traversing the entire list, the free list must be doubly linked. Thus, each free block must contain two pointers **next** and **prev** to the next and previous free blocks on the free list. It is also necessary to be able to access these two pointers from the last location of a free block (This is needed when combining a newly freed block with a free block that immediately precedes it in memory). Thus, the front of free block must be accessible from its rear. One way to do this is to introduce a **bsize** field at a given negative offset from the last location of each free block. The figure below illustrates the structure of free and allocated blocks under this method which is called, **Boundary Tag Method**. Each of the control fields **(fflag, size, next, prev, bsize and bflag)** are shown as occupying a complete word.

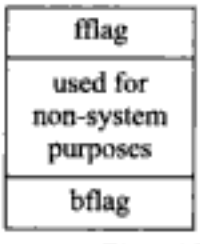


Fig. 15.12

Hidden page

Garbage collection is usually done in two phases. The first phase called the **marking phase** involves marking of all nodes that are accessible from the external pointer. The second phase called the **collection phase** which involves proceeding sequentially through memory and freeing all nodes that have not been marked.

Therefore, it can be said that first, the computer runs through all lists, tagging those cells that are currently in use and then the computer runs through the memory, collecting all untagged space on to the free-storage list. The garbage collection may take place when there is only some minimum amount of space left in the free storage list and when the CPU is idle and has time to do the collection.

Summary

- Storage is a precious computer resource and its effective management is critical for enhancing the operational efficiency. There are several strategies employed for dynamic storage management.
- Memory freeing refers to the release of an allocated block of storage and its return to the free list.
- In the first-fit allocation algorithm the block that is allocated is the first block that is found to be larger than or equal to the requested amount.
- The best-fit method does not use the first suitable block found but instead continues searching the list until the smallest suitable block is found.
- Boundary tag method is a procedure to free a block which uses doubly linked list structure for the available list.
- Garbage collection is the process of collecting all unused nodes and returning them to the available space.

Review Exercise

Multiple Choice Questions

1. Compaction is
   a. Compression of information fields of a list
   b. Moving all memory locations currently in use into single contiguous region of memory
   c. Conversion of a link list to an array
   d. None of the above

State whether True or False

1. First-fit technique is used in memory management.
2. Garbage collection is about freeing dynamically allocated memory when not in use.
3. Internal fragmentation is a big disadvantage of buddy system.
4. The term liberation is the process of again allocating an allocated block of storage.

Descriptive Questions

1. Implement the first-fit, best-fit and worst-fit methods of storage allocation as follows: Write a function **getblock(n)** that returns the address of block of size **n** that is available for allocation and modifies the free

list appropriately. The function should utilize the following variables memsize, the number of locations in memory [memsize], an array of integers representing the memory, freeblock, a pointer to the first location of the first freeblock on the list.

The value of **size(p)** may be obtained by the expression **memory(p)** and the value of **next(p)** by the expression **memory[p+1]**.

2. Implement the boundary tag method of liberation as a C function as in the above exercise. The values of size(p) and bsize(p) should be obtained by the expression abs(memory [p]), fflag(p), and bflag(p) by (memory[p] > 0) next (p) by memory[P+1] and prev(p) by memory[P+2].
3. How could the freelist be organized to reduce the search time in best-fit method? What liberation algorithm would be used for such a freelist?
4. Prove formally (using mathematical induction) that in binary buddy system
   a. there are 2^{m-i} possible i blocks.
   b. the starting address of i-block is an integer multiple of 2^i.
5. Implement a binary buddy system as a set of C program.

APPENDICES

APPENDIX

A Mathematical Concepts for Data Structures

Key Features

- Matrix
- Polynomials
- Sum of Powers of Integers
- Logarithms
- Harmonic Numbers
- Permutations, Combinations, Factorials
- Fibonacci Numbers
- Catalan Numbers

This appendix discusses important mathematical concepts which have been used in many applications. There are many mathematical results which are used in algorithm analysis.

The concept of logarithms is used to provide a convenient way to handle very large numbers whereas the concepts of Fibonacci and Catalan numbers are additional topics included in this chapter.

MATRIX

A matrix is a rectangular arrangement of numbers, arranged in rows and columns. For example,

$$\begin{pmatrix}5\\1\end{pmatrix}, \begin{pmatrix}5 & 3\\1 & 2\end{pmatrix}, [5 \quad 3 \quad 2], \text{ etc.}$$

Each number or entity in a matrix is called its **element**.
Plural of matrix is **matrices**.

Order of a Matrix The order of a matrix = Number of rows in it × Number of columns in it; For example, if a matrix has **m** rows and **n** columns, its order is written as **m × n** (read as m by n). Consider the matrix

$$\begin{pmatrix}2 & 1 & 5\\3 & -2 & 7\end{pmatrix} \leftarrow 1^{st} \text{ row}, \quad \leftarrow 2^{nd} \text{ row}$$

1st column, 2nd column, 3rd column

It has 2 rows and 3 columns; hence its order = 2 × 3 (read as 2 by 3)

[Note While stating the order of a matrix, the number of rows is given first followed by the number of columns.]

Notation: Matrices, in general, are denoted by capital letters. For example, if A is a matrix with **m** rows and **n** columns, then it is written as $\mathbf{A_{m \times n}}$.

Similarly, $\mathbf{B_{2 \times 3}}$ denotes a matrix B with 2 rows and 3 columns.

Types of Matrices

Row Matrix A matrix, which has only one row, is called a **row matrix**. For example,

[a b] ← Single row

1st column, 2nd column

Since, this matrix has 1 row and 2 columns, its order = **1 × 2** (1 by 2).

Similarly, [a b c] is a row matrix of order 1 x 3.

A row matrix is also called a **row vector**.

Column Matrix A matrix which has only one column is called a **column matrix**. For example,

$\begin{pmatrix} a \\ b \end{pmatrix}$ ← 1st row, ← 2nd row

Single column

Since, this matrix has 2 rows and 1 column, its order = **2 × 1** (2 by 1).

Similarly, $\begin{pmatrix} a \\ b \\ c \end{pmatrix}$ is a column matrix of order 3 × 1.

A column matrix is also called a **column vector**.

Square Matrix A matrix which has equal number of rows and columns is called a **square matrix**. For example,

$\begin{pmatrix} a & b \\ c & d \end{pmatrix}$ ← 1st row, ← 2nd row

1st column, 2nd column

Since, this matrix has 2 rows and 2 columns, its order = **2 × 2** (2 by 2).

Similarly, $\begin{pmatrix} 5 & 7 & 4 \\ 2 & -1 & 0 \\ 0 & 3 & 4 \end{pmatrix}$ is a square matrix of order 3×3.

Rectangular Matrix A matrix in which the number of rows are not equal to the number of columns (i.e., $m \neq n$), is called a **rectangular matrix**. For example,

$$\begin{pmatrix} 2 & 4 & 7 \\ 1 & 0 & 5 \end{pmatrix} \qquad \begin{pmatrix} 3 & 1 \\ 6 & 2 \\ 1 & 7 \end{pmatrix}$$

order is 2×3 order is 3×2

Zero or Null Matrix If each element of a matrix is zero, it is called a **zero matrix** or a **null matrix**. For example,

$$[0 \quad 0], \begin{pmatrix} 0 \\ 0 \end{pmatrix}, \begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}, \text{ etc.}$$

Diagonal Matrix A square matrix, which has all its elements zero except those on the leading (or principal) diagonal, is called a **diagonal matrix**. For example,

$$\begin{pmatrix} 2 & 0 \\ 0 & 3 \end{pmatrix}, \begin{pmatrix} 5 & 0 & 0 \\ 0 & -2 & 0 \\ 0 & 0 & 3 \end{pmatrix}, \text{ etc.}$$

[Leading (principal) diagonal means; the diagonal from top left to bottom right].

Unit or Identity Matrix A diagonal matrix, in which each element of its leading diagonal is unity (i.e., 1) and all other elements are zeros, is called a **unit** or **identity matrix**. It is denoted by I. In other words, it is a square matrix, in which each element of its leading diagonal is equal to 1 and all other remaining elements are zero each. For example,

$$\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}, \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}, \text{ etc.}$$

Transpose of a Matrix

Transpose of a matrix is the matrix obtained by interchanging its rows and columns. If **A** is a matrix, then its transpose is denoted by $\mathbf{A}^t$. For example,

$$\text{If } A = \begin{pmatrix} 2 & 3 & 1 \\ 0 & 4 & 7 \end{pmatrix}, \text{ then its transpose } A^t = \begin{pmatrix} 2 & 0 \\ 3 & 4 \\ 1 & 7 \end{pmatrix}$$

Hidden page

MULTIPLICATION OF MATRICES

Two matrices **A** and **B** can be multiplied together to get the product matrix **AB**, if and only if, the number of columns in **A** (the left hand matrix) is equal to the number of rows in **B** (the right hand matrix).

Step 1: Multiply every element of 1st row of matrix **A** with corresponding element of 1st column of **B** and add them to get the first element of the 1st row of the product matrix **AB**.

Step 2: Multiply every element of 1st row of matrix **A** with corresponding element of 2nd column of **B** and add them to get the second element of the 1st row of the product matrix **AB**.

Step 3: Similarly, multiply the elements of 2nd row of **A** with corresponding elements of the columns of **B** and get elements of the second row of **AB**.

$\therefore$ Product of matrices **A** and **B** = **AB**

$$= \begin{pmatrix} 1^{st} \text{ row of A} \times 1^{st} \text{ column of B} & 1^{st} \text{ row of A} \times 2^{nd} \text{ column of B} \\ 2^{nd} \text{ row of A} \times 1^{st} \text{ column of B} & 2^{nd} \text{ row of A} \times 2^{nd} \text{ column of B} \end{pmatrix}$$

i.e. If $A = \begin{pmatrix} -2 & 3 \\ 4 & 1 \end{pmatrix}$ and $B = \begin{pmatrix} 1 & 2 \\ 3 & 5 \end{pmatrix}$

then $$AB = \begin{pmatrix} -2 & 3 \\ 4 & 1 \end{pmatrix}\begin{pmatrix} 1 & 2 \\ 3 & 5 \end{pmatrix}$$

$$= \begin{pmatrix} -2\times1+3\times3 & -2\times2+3\times5 \\ 4\times1+1\times3 & 4\times2+1\times5 \end{pmatrix}$$

$$= \begin{pmatrix} 7 & 11 \\ 7 & 13 \end{pmatrix}$$

Similarly, $$BA = \begin{pmatrix} 1 & 2 \\ 3 & 5 \end{pmatrix}\begin{pmatrix} -2 & 3 \\ 4 & 1 \end{pmatrix}$$

$$= \begin{pmatrix} 1^{st} \text{ row of B} \times 1^{st} \text{ column of A} & 1^{st} \text{ row of B} \times 2^{nd} \text{ column of A} \\ 2^{nd} \text{ row of B} \times 1^{st} \text{ column of A} & 2^{nd} \text{ row of B} \times 2^{nd} \text{ column of A} \end{pmatrix}$$

$$= \begin{pmatrix} 1\times-2+2\times4 & 1\times3+2\times1 \\ 3\times-2+5\times4 & 3\times3+5\times1 \end{pmatrix}$$

$$= \begin{pmatrix} 6 & 5 \\ 14 & 14 \end{pmatrix}$$

Identity Matrix for Multiplication In matrix multiplication, the unit matrix **I** is known as the **identity matrix** for multiplication; since, on multiplying any matrix with identity matrix of the same order; the matrix remains unaltered.

If I is the unit matrix and **A** is any matrix of the same order as that of I then

$$\mathbf{A \times I = A = I \times A}$$

For example,

Let $A = \begin{pmatrix} 2 & 3 \\ 4 & 6 \end{pmatrix}$

then $A \times I = \begin{pmatrix} 2 & 3 \\ 4 & 6 \end{pmatrix}\begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix} = \begin{pmatrix} 2\times1+3\times0 & 2\times0+3\times1 \\ 4\times1+6\times0 & 4\times0+6\times1 \end{pmatrix} = \begin{pmatrix} 2 & 3 \\ 4 & 6 \end{pmatrix} = A$

and $I \times A = \begin{pmatrix} 1 & 0 \\ 0 & 1 \end{pmatrix}\begin{pmatrix} 2 & 3 \\ 4 & 6 \end{pmatrix} = \begin{pmatrix} 1\times2+0\times4 & 1\times3+0\times6 \\ 0\times2+1\times4 & 0\times3+1\times6 \end{pmatrix} = \begin{pmatrix} 2 & 3 \\ 4 & 6 \end{pmatrix} = A$

$\therefore \quad A \times I = A = I \times A$

If the order of matrix **A** is **m×n** and order of matrix **B** is **n×p** then the product **AB** is possible (i.e. matrices are compatible for multiplication) as number of columns of first matrix **A** is equal to the number of rows of second matrix **B**. And the order of product matrix, **C**, obtained is **m×p** (i.e. number of rows of first matrix × number of columns of second matrix).

i.e. $\mathbf{A_{m\times n} \times B_{n\times p} = C_{m\times p}}$

For example:

Let $A = \begin{pmatrix} 2 & 0 \\ 5 & 3 \\ 4 & 1 \end{pmatrix}$ and $B = \begin{pmatrix} 1 \\ 3 \end{pmatrix}$

Here, **A** is 3×2 matrix and **B** is 2×1 matrix, therefore, product **AB** is possible; and the other of product matrix will be 3×1.

$$AB = \begin{pmatrix} 2 & 0 \\ 5 & 3 \\ 4 & 1 \end{pmatrix}\begin{pmatrix} 1 \\ 3 \end{pmatrix}$$

$$= \begin{pmatrix} 2\times1+0\times3 \\ 5\times1+3\times3 \\ 4\times1+1\times3 \end{pmatrix} = \begin{pmatrix} 2 \\ 14 \\ 7 \end{pmatrix}$$ which is a 3×1 matrix.

[Note

1. In matrix multiplication, multiply from left to right, i.e. for any three matrices A, B and C which are compatible for multiplication; to get ABC: first of all multiply B with C and then A with the product of B and C.
2. In general $AB \neq BA$ i.e. product of matrices is not commutative.
3. $(AB)C = A(BC)$ i.e. product of matrices is associative
4. If $A \neq 0$ and $AB = AC$, then it is not necessary that B=C.
5. If AB=0, then it is not necessary that A=0 or B=0.
6. If A=0 or B=0, then AB=0=BA.
7. $A(B+C) = AB + AC$ i.e. multiplication of matrices is distributive with respect to addition.]

Hidden page

Hidden page

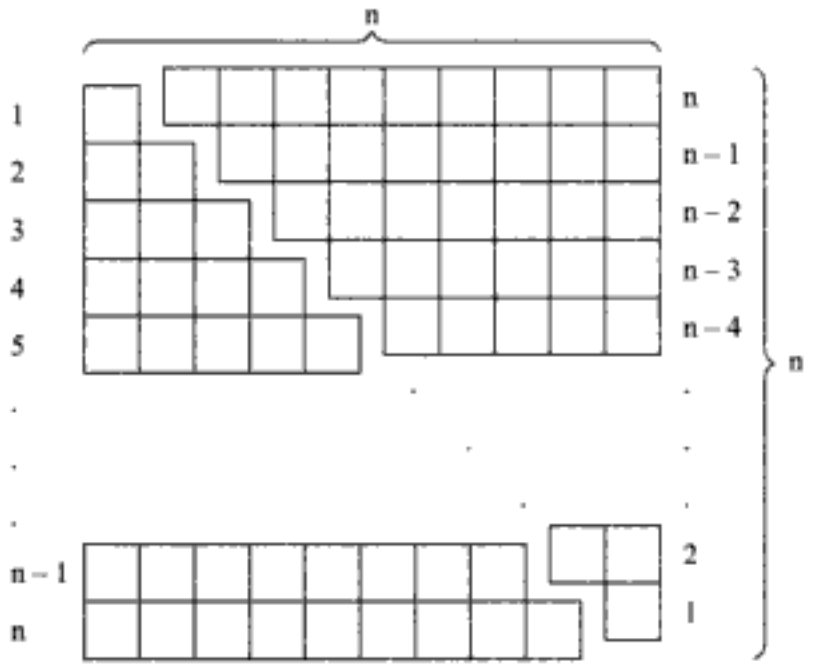


Fig. A.1 *Geometrical Proof for Sum of Integers*

It follows that

$$1^2 + 2^2 + \ldots + (n-1)^2 + n^2 = \frac{(n-1)n(2(n-1)+1)}{6} + n^2$$

$$= \frac{2n^3 - 3n^2 + n + 6n^2}{6} = \frac{n(n+1)(2n+1)}{6}$$

which is the desired result, and the proof by induction is complete.

A convenient shorthand for a sum of the sort appearing in these identities is to use the capital Greek letter sigma, Σ, in front of the typical summand, with the initial value of the index controlling the summation written below the sign, and the final value above. Thus the identities discussed above can be written as

$$\sum_{K=1}^{n} k = \frac{n(n+1)}{2} \quad \text{and}$$

$$\sum_{k=1}^{n} k^2 = \frac{n(n+1)(2n+1)}{6}$$

Two other formulas are also useful, particularly in working with trees.

$$1 + 2 + 4 + \ldots + 2^{m-1} = 2^{m-1}$$
$$1 \times 1 + 2 \times 2 + 3 \times 4 + \ldots + m \times 2^{m-1} = (m-1) \times 2^{m+1}$$

In summation notation these equations can be written as follows:

$$\sum_{k=0}^{m-1} k = 2^{m-1}$$

$$\sum_{k=1}^{m} k \times 2^{k-1} = (m-1) \times 2^{m+1}$$

Proof: The first formula will be proved in a more general form. We start with the following identity, which, for any value of $x \neq 1$, can be verified simply by multiplying both sides by $x-1$:

$$\frac{x^{m-1}}{x-1} = 1 + x + x^2 + \ldots + x^{m-1}$$

for any $x \neq 1$. With $x=2$ this expression becomes the first formula.

To establish the second formula we take the same expression in the case of $m+1$ instead of m:

$$\frac{x^{m-1}-1}{x-1} = 1 + x + x^2 + \ldots + x^{m}$$

for any $x \neq 1$, and differentiate with respect to x:

$$\frac{(x-1)(m+1)x^m - (x^{m+1}-1)}{(x-1)^2} = 1 + 2x + 3x^2 + \ldots + mx^{m-1}$$

for any $x \neq 1$. Setting $x=2$ now gives the second formula.

Suppose that $|x|<1$ in the above formulas. As m becomes large, it follows that x^m becomes small. In the ultimate analysis

$$\lim_{m \to \infty} x^m = 0$$

Taking the limit as $m \to \infty$ in the preceding equations gives

If $|x| < 1$ then

$$\sum_{k=0}^{\infty} x^k = \frac{1}{1-x}$$

infinite series

$$\sum_{k=1}^{\infty} kx^{k-1} = \frac{1}{(1-x)^2}$$

LOGARITHMS

The main reason for using logarithms is to convert multiplication and division into addition and subtraction, and exponentiation into multiplication. Before the advent of pocket calculators, logarithms were an essential tool for hand calculation: witness the large table of logarithms and the once ubiquitous slide rule. Even though we now have other methods for numerical calculation, the fundamental properties of logarithms give them importance that extends far beyond their use as computational tools.

The behaviour of many phenomena, in nature, reflects an intrinsically logarithmic structure; that is, by using logarithms we find important relationships that are not otherwise obvious. Measuring the loudness of sound, for example, is logarithmic: if one sound is 10 db (decibels) louder than another, then the actual acoustic energy is 10 times as much. If the sound level in one room is 40 db and it is 60 db in

another, then the human perception may be that the second room is one and half times as noisy as the first, but there is actually 100 times more sound energy in the second room. This phenomenon is why a single violin soloist can be heard above a full orchestra (when playing a different line), and yet the orchestra requires so many violins to maintain a proper balance of sound.

Secondly, logarithms provide a convenient way to handle very large numbers. The scientific notation, where a number is written as a small real number (often in the range from 1 to 10) times a power of 10, is really based on the concept of logarithms, since the power of 10 is essentially the logarithm of the number. Scientists who need to use very large numbers (like astronomers, nuclear physicists, and geologists) frequently speak of orders of magnitude, and thereby concentrate on the logarithm of the number.

Thirdly, a logarithmic graph is a very useful device for displaying the properties of a function over a much broader range than a linear graph. With a logarithmic graph, we can arrange to display detailed information on the function for small values of the argument and at the same time give an overall view for much larger values. Logarithmic graphs are especially appropriate when we wish to show percentage changes in a function.

Definition of Logarithms

Logarithms are defined in terms of a real number $a > 1$, which is called the base of the logarithms. For any number $x > 0$, we define $\log_a x = y$ where y is the real number such that $a^y = x$. The logarithm of a negative number, and the logarithm of 0, are not defined.

Simple Properties

From the definition and from the properties of exponents we obtain

$$\log_a 1 = 0,$$
$$\log_a a = 1,$$
$$\log_a x < 0 \text{ for all } x \text{ such that } 0 < x < 1.$$
$$0 < \log_a x < 1 \text{ for all } x \text{ such that } 1 < x < a.$$
$$\log_a x < 1 \text{ for all } x \text{ such that } a < x.$$

The logarithm function has a graph like the one given in Fig. A.2.
We also obtain the identities

$$\log_a (xy) = (\log_a x) + (\log_a y)$$
$$\log_a (x/y) = (\log_a x) - (\log_a y)$$
$$\log_a x^z = z \log_a x$$
$$\log_a x^z = z$$
$$a^{\log_a x} = x$$

that hold for any positive real numbers x and y, and for any real number z.

HARMONIC NUMBERS

As a final application of logarithms, we obtain an approximation to a sum that appears frequently in the analysis of algorithms, especially that of sorting methods. The n^{th} **harmonic number** is defined to be the sum of the reciprocals of the integers from 1 to n.

Hidden page

By refining this method of approximation by an integral, it is possible to obtain a very much closer approximation to Hn, if such is desired. Specifically,

The harmonic number H_n, $n \geq 1$, satisfies

$$H_n = \ln n + \gamma + \frac{1}{2n} - \frac{1}{12n^2} + \frac{1}{120n^4} - e$$

where $0 < \varepsilon < 1/(252n^6)$, and $\gamma \approx 0.577215665$ is known as **Euler's constant**.

PERMUTATIONS, COMBINATIONS, FACTORIALS

Permutations

A **permutation** of objects is an ordering or arrangement of the objects in a row. If we begin with **n** different objects, then we can choose any of the **n** objects to be the first one in the arrangement. There are then **n–1** choices for the second object, and since these choices can be combined in all possible ways, the number of choices multiples. Hence the first two objects may be chosen in **n(n–1)** ways. There are left **n–2** objects, any one of which may be chosen as the third in the arrangement. Continuing in this way, we see that the number of permutations of **n** distinct object is

$$n! = n \times (n-2) \times \ldots \times 2 \times 1$$

Object to permute	a b c d					
Choose a first:	a b c d	a b d c	a c b d	a c d b	a d b c	a d c b
Choose b first:	b a c d	b a d c	b c a d	b c d a	b d a c	b d c a
Choose c first:	c a b d	c a d b	c b a d	c b d a	c d a b	c d b a
Choose d first:	d a b c	d a c b	d b a c	d b c a	d c a b	d c b a

Fig. A.4 *Constructing Permutations*

Note that we have assumed that the objects are all distinct, that is, we can distinguish each object from every other one. It is often easier to count configurations of distinct objects than when some are indistinguishable. The latter problem can sometimes be solved by temporarily labelling the objects so they are all distinct, then counting the configurations, and finally dividing by the number of ways in which the labelling could have been done. The special case discussed in the next section is especially important.

Combinations

A **combination** of **n** objects taken **k** at a time is a choice of **k** objects out of the **n**, without regard for the order of selection. The number of such combinations is denoted either by

$$C(n, k) \text{ or by } \left(\frac{n}{k}\right)$$

We can calculate **C(n, k)** by starting with the **n!** permutations of **n** objects and form a combination simply by selecting the first **k** objects in permutation. The order, however, in which these **k** objects appear is ignored in determining a combination, so we must divide by the number **k!** of ways to order **k** objects chosen. The order of the **n–k** objects not chosen is also ignored, so we must also divide by **(n–k)!**. Hence,

$$C(n,k) = \frac{n!}{k!(n-k)!}$$

Objects from which to choose: a b c d e f				
a b c	a c d	a d f	b c f	c d e
a b d	a c e	a e f	b d e	c d f
a b e	a c f	b c d	b d f	c e f
a b f	a d e	b c e	b e f	d e f

Fig. A.5 *Combinations of 6 objects, taken 3 at a time*

The number of combinations **C(n, k)** is called a **binomial coefficient**, since it appears as the coefficient of $x^k y^{n-k}$ in the expansion of $(x+y)^n$. There are hundreds of different relationships and identities about various sums and products of binomial coefficients.

Factorials

We frequently use permutations and combinations in analyzing algorithms, and for these applications we must estimate the size of n! for various values of n. An excellent approximation to n! for various values of n was obtained by JAMES STIRLING in the eighteenth century:

Theorem

$$n! = \sqrt{2\pi n}\left(\frac{n}{3}\right)^n\left[1 + \frac{1}{12n} + O(n^{\frac{1}{2}})\right]$$

We usually use this approximation in logarithmic form instead:

Corollary

$$\ln n! = \left(n + \frac{1}{2}\right)\ln n - n + \frac{1}{2}\ln(2\pi) + \frac{1}{12n}O(n^{\frac{1}{2}})$$

Note that, as n increases, the approximation to the logarithm becomes more and more accurate; that is, the difference between the approximation and actual value approaches 0. The difference between direct approximation to the factorial and n! itself will not necessarily become small (that is, the difference need not go to 0), but the percentage error becomes arbitrarily small (the ratio goes to 1). Knuth gives refinements of Stirling's approximation that are even closer.

The complete proof of Stirling's approximation requires techniques from advanced calculus that would take us too far afield here. We can, however, use a bit of elementary calculus to illustrate the first step of the approximation. First, we take the natural logarithm of a factorial, noting that the logarithm of a product is the sum of the logarithms:

$$\begin{aligned}\ln n! &= \ln(n \times (n-1) \times \ldots \times 1) \\ &= \ln n + \ln(n-1) + \ldots + \ln 1 \\ &= \sum_{x=1}^{n} \ln x\end{aligned}$$

Next, we approximate the sum by an integral, as shown in Fig. A.6.

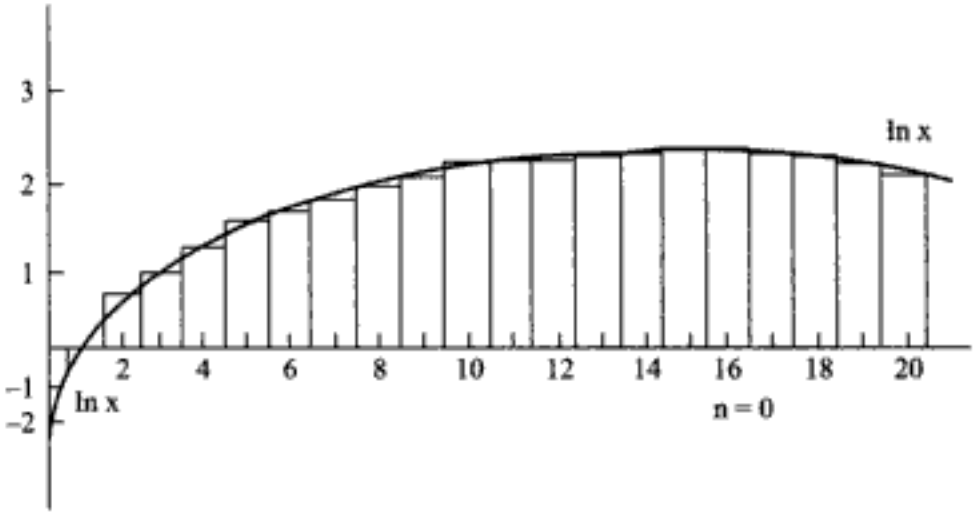


Fig. A.6 *Approximation of ln n! by an Integral*

It is clear from the diagram that the area under the step function, which is exactly ln n!, is approximately the same as the area under the curve, which is

$$\int_{\frac{1}{2}}^{n+\frac{1}{2}} \ln \times dx = (x \ln - x)\Big|_{\frac{1}{2}}^{n+\frac{1}{2}} = \left(n+\frac{1}{2}\right)\ln\left(n+\frac{1}{2}\right) - n + \frac{1}{2}\ln 2$$

For large values of n, the difference between In n and In $\left(n+\frac{1}{2}\right)$ is insignificant and hence this approximation differs from Stirling's only by the constant difference between $\frac{1}{2}$ 1n 2 (about 0.35) and $\frac{1}{2}$ In(2π) (about 0.919).

FIBONACCI NUMBERS

The Fibonacci numbers originated as an exercise in arithmetic proposed by Leonardo Fibonacci in 1202:

How many pairs of rabbits can be produced from a single pair in a year? We start with a single newly born pair; it takes one month for a pair to mature, after which they produce a new pair each month, and the rabbits never die.

In month 1, we have only one pair. In month 2, we still have only one pair, but they are now mature. In month 3, they have reproduced, so we now have two pairs. And so it goes. The number F_n of pairs of rabbits that we have in month n satisfies.

$$F_0 = 0,\ F_1 = 1, \text{ and } F_n = F_n - 1 + F_{n-2} \text{ for } n \geq 2.$$

This same sequence of numbers, called the **Fibonacci sequence**, appears in many problems.

We shall use the method of **generating functions**, which is important for many other applications. The generating function is a formal infinite series in a symbol x, with the Fibonacci numbers as coefficients:

$$F(x) = F_0 + F_1x + F_2x^2 + \ldots + F_nx^n + \ldots$$

We do not worry about whether this series converges, or what the value of x might be, since we are not going to set x to any particular value. Instead, we shall only perform formal algebraic manipulations on the generating function.

Next, we multiply by powers of x:

$$F(x) = F_0 + F_1x + F_2x^2 + \ldots + F_nx^n + \ldots$$
$$xF(x) = F_0x + F_1x^2 + \ldots + F_{n-1}x^n + \ldots$$
$$x^2F(x) = F_0x^2 + \ldots + F_{n-2}x^n + \ldots$$

and subtract the second two equations from the first:

$$(1 - x - x^2)\, F(x) = F_0 + (F_1 - F_0)\, x = x,$$

since $F_0 = 0$, $F_1 = 1$, and $F_n = F_{n-1} + F_{n-2}$ for all $n \geq 2$. We, therefore, obtain

$$F(x) = \frac{x}{1 - x - x^2}$$

The roots $1 - x - x^2$ are $\frac{1}{2}(-1 \pm \sqrt{5})$. By the method of partial fraction we can thus rearrange the formula for F(x) as:

$$F(x) = \frac{1}{\sqrt{5}}\left(\frac{1}{1 - \varnothing x} - \frac{1}{1 - \psi x}\right)$$

where $$\varnothing = \frac{1}{\sqrt{5}}(1 + \sqrt{5}) \quad \text{and} \quad \psi = 1 - \varnothing = \frac{1}{\sqrt{5}}(1 + \sqrt{5}).$$

(Check this equation for F(x) by putting the two fractions on the right over a common denominator.) The next step is to expand the fractions on the right side dividing three denominators into 1:

$$F(x) = \frac{1}{\sqrt{5}}\,(1 + \varnothing x + \varnothing^2x^2 + \ldots - 1 - \psi x - \psi^2x^2 - \ldots).$$

The final step is to recall that the coefficients of F(x) are the Fibonacci numbers, and therefore, to equate the coefficients of each power of x on both sides of this equation. We thus obtain

$$F_n = \frac{1}{\sqrt{5}}(\varnothing^n - \psi^n).$$

Approximate values for $\varnothing$ and ψ are

$$\varnothing \approx 1.618034 \text{ and } \psi \approx -0.618034.$$

This surprisingly simple answer to the values of the Fibonacci numbers is interesting in several ways. It is, first, not even immediately obvious why the right side should always be an integer. Second, ψ is a negative number of sufficiently small absolute value that we always have $F_n = \varnothing^n/\sqrt{5}$ rounded to the nearest integer. Third, the number $\varnothing$ is itself interesting. It has been studied since the times of the ancient Greeks—it is often called the **golden mean**—and the ratio of $\varnothing$ to 1 is said to give the most pleasing shape of a rectangle.

CATALAN NUMBERS

The purpose of this section is to count the binary trees with n vertices. We shall accomplish this result via a slightly circuitous route, discovering along the way several other problems that have the same

answer. The resulting numbers, called the Catalan numbers, are of considerable interest in that they appear in the answers to many apparently unrelated problems.

The Main Result

For $n \geq 0$, the n^{th} **Catalan number** is defined as

$$Cat(n) = \frac{C(2n, n)}{n+1} = \frac{(2n)!}{(n+1)!n!}$$

The number of distinct binary trees with n vertices, $n \geq 0$, is the n^{th} catalan number Cat(n).

The Proof by One-to-One Correspondences

Let us first recall the one-to-one correspondence between the binary trees with **n** vertices and the orchards with **n** vertices. Hence to count binary trees, we may just as well count orchards.

Well-Formed Sequences of Parentheses Second, let us consider the set of all well-formed sequences of n left parentheses '(' and n right parentheses ')'. A sequence is **well formed** means that, when scanned from left to right, the number of right parentheses encountered never exceeds the number of left parentheses. Thus '((()))' and '()()()' are well formed, but '( )) (( )' is not, nor is '(()', since the total numbers of left and right parentheses in the expression must be equal.

There is a one-to-one correspondence between the orchards with n vertices and the well-formed sequences of **n** left parentheses and **n** right parentheses, $n \geq 0$.

To define this correspondence, we first recall that an orchard is either empty or is an ordered sequence of ordered trees. We define the bracketed form of an orchard to be the sequence of bracketed form of its trees, written one after the next in the same order as the trees in the orchard. The bracketed form of the empty orchard is empty. We recall also that an ordered tree is defined to consist of its root vertex, together with an orchard of subtrees. We thus define the bracketed form of an ordered tree to consist of a left parenthesis '(' followed by the (name of the) root, followed by the bracketed form of the orchard of subtrees, and finally a right parenthesis ')'.

The bracketed forms of several ordered trees and orchards appear in Fig. A.7. It should be clear that the mutually recursive definitions above produce a unique bracketed form for any orchard and that the resulting sequence of parentheses is well formed. If, on the other hand, we begin with a well-formed sequence of parentheses, then the outermost pair(s) of parentheses correspond to the tree(s) of an orchard, and within such a pair of parentheses is the description of the corresponding tree in terms of its root and its orchard of subtrees. In this way, we have now obtained a one-to-one correspondence between the orchards with **n** vertices and the well-formed sequences of **n** left and **n** right parentheses.

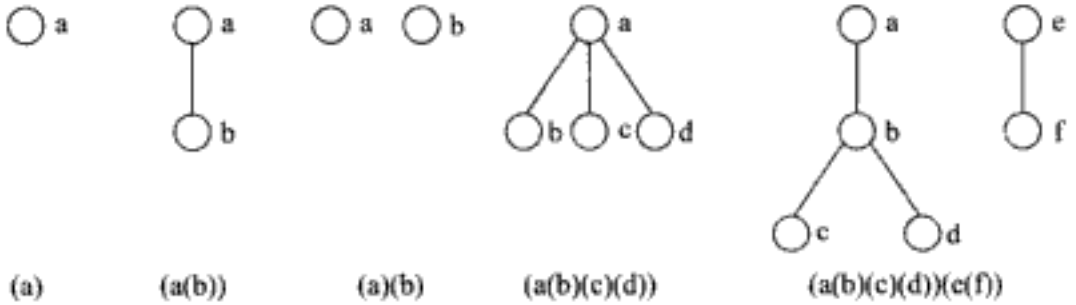


Fig. A.7 *Bracketed Form of Orchards*

In counting orchards we are not concerned with the labels attached to the vertices, and hence we shall omit the labels and, with the correspondence outlined above, shall now count well-formed sequences of n left and n right parentheses, with nothing else inside the parentheses.

Stack Permutations Let us note that, by replacing each left parenthesis by –1 and each right parenthesis by –1, the well-formed sequences of parentheses correspond to sequences of +1 and –1 such that the partial sums count the items on the stack at a given time. From this it can be shown that the number of stack permutations of n objects is yet another problem for which the Catalan numbers provide the answer. Even more, if we start with an orchard and perform a complete traversal (walking around each branch and vertex in the orchard as though it were a decorative wall), counting +1 each time we go down a branch, and –1 each time we go up a branch (with +1 –1 for each leaf), then we thereby essentially obtain the correspondence with well-formed sequences over again.

Arbitrary Sequence of Parentheses Out final step is to count well-formed sequences of parentheses, but to do this we shall instead count the sequences that are not well formed, and subtract from the number of all possible sequences. We need a final one-to-one correspondence:

[Note The sequences of n left and n right parentheses that are not well formed correspond exactly to all sequences of n–1 left parentheses and n+1 right parentheses (in all possible orders).**]**

To prove this correspondence, let us start with a sequence of **n** left and **n** right parentheses that is not well formed. Let **k** be the first position in which the sequence goes wrong, so the entry at position **k** is a right parenthesis, and there is one more right parenthesis than left up through this position. Hence strictly to the right of position **k** there is one fewer right parenthesis than left. Strictly to the right of position **k**, then, let us replace all left parentheses by right and all right parentheses by left. The resulting sequence will have n–1 left parentheses and n+1 right parentheses altogether.

Conversely, let us start with a sequence of n–1 left parentheses and n+1 right parentheses, and let k be the first position where the number of right parentheses exceeds the number of left (such a position must exist, since there are more right than left parentheses altogether). Again let us exchange left for right and right for left parentheses in the remainder of the sequence (position after k). We thereby obtain a sequence of n left and n right parentheses that is not well formed and have constructed the one-to-one correspondence as desired.

With all these preliminary correspondences, our counting problem reduce to simple combinations. The number of sequences of n–1 left and n+1 right parentheses is the number of ways to choose the n–1 positions occupied by left parentheses from the 2n positions in the sequence that is the number is C(2n.n–1). This number is also the number of sequences of n left and n right parentheses that are not well formed. The number of all sequences of n left and n right parentheses is similarly C(2n.n), so the number of well-formed sequence is

$$C(2n.n) - C(2n.n-1)$$

which is precisely the n^{th} catalan number.

Due to all the one-to-one correspondences, we also have:

The number of well-formed sequences of **n** left and **n** right parentheses, the number of permutations of **n** objects obtainable by a stack, the number of orchards with **n** vertices, and the number of binary trees with **n** vertices are equal to the $\mathbf{n}^{th}$ Catalan number Cat(n).

APPENDIX

B Sample Question Papers

A6-R3: Data Structures Through 'C' Language (July 2004)

Note:

- There are TWO PARTS in this Module/Paper. PART ONE contains FOUR questions and PART TWO contains FIVE questions.
- PART ONE is to be answered in the TEAR-OFF ANSWER SHEET only, attached to the question paper, as per the instructions contained therein. PART ONE is NOT to be answered in the answer book.
- Maximum time allotted for PART ONE is ONE HOUR. Answer book for PART TWO will be supplied at the table when the answer sheet for PART ONE is returned. However, candidates, who complete PART ONE earlier than one hour, can collect the answer book for PART TWO immediately after handing over the answer sheet for PART ONE.

TOTAL TIME: 3 HOURS **TOTAL. MARKS : 100**
(PART ONE — 40; PART TWO — 60)

PART ONE

(Answer all the questions)

1. Each question below gives a multiple choice of answers. Choose the most appropriate one and enter in the "tear-off" answer sheet attached to the question paper, following instructions therein. (1×10)

1.1 The lower bound in 'C' Language is

(a) always 0.
(b) defined by programmer in the program.
(c) dependent on C compiler.
(d) None of the above.

1.2 Given: an array declared as
int item [n1][m1],
base (item) is address of item [0][0],
esize is the size of each element in the array.
Then, the address of item [i1][i2] is at
(a) base(item) + (i1 * i2 + m1) * esize
(b) base(item) + (i1 * m1 + i2) * esize
(c) base(item) + (i1 * n1 + i2) * esize
(d) None of the above.

1.3 In 'C' Language variables and parameters declared within a function are known as
(a) functional variables.
(b) floating-point variables.
(c) automatic variables.
(d) None of the above.

1.4 The total number of comparisons in bubble sort is
(a) O $(n)^2$
(b) O (n^2)
(c) O (2n)
(d) None of the above.

1.5 Which statement is true for indexed search method?
(a) an auxiliary table is set aside in addition to the file itself.
(b) the index is maintained in the separated indexed file.
(c) the indexed is stored on separate storage media for fast retrieval.
(d) None of the above.

1.6 A balanced binary tree is a binary tree in which the heights of the two subtrees of every node never differ by more than
(a) 2
(b) 1
(c) 0
(d) None of the above.

1.7 A node n is adjacent to a node m if there is an arc from
(a) m to n
(b) n to m
(c) either m to anynode or n to anynode
(d) None of the above.

1.8 Which of the following representation of graphs is more adequate?
(a) Stack representation of graphs.
(b) Adjacency matrix representation of graphs.
(c) Linked representation of graphs.
(d) None of the above.

1.9 An ordered forest is one whose _____
(a) component trees are ordered.
(b) component trees are unordered.

Hidden page

Hidden page

```
            low = mid - 1;
        } /* end while */
        return (-1);
```
(5+5+5)

7. (a) Discuss basic methods of dealing with a hash clash.
   (b) The greatest common divisor (GCD) of two positive integers is the largest integer that divides both of them. For example, GCD of 18 and 24 is 6, 16 and 72 is 8. Write a recursive function using the recursive algorithm given below:

```
GCD (x,y) = x if y = 0
  GC(x,y) = GCD(y, x%y)
```

   (c) Write a program to read a string (one line of characters) and push any vowels in the string to a stack called vowels. Then pop your stack repeatedly and count the number of vowels in the string. (4+5+6)
8. (a) How does the data structure tree differ from the other data structures?
   (b) Write a 'C' function to traverse a binary tree level by level. In each level the tree is traversed from left to right.
   (c) What is the idea behind using threaded binary trees? (4+5+6)
9. (a) What objective should be sought in the designing of a hash function?
   (b) Write a 'C' program to store the roll number and names of the students in a binary search tree. Write a 'C' function which searches for the roll number of a student in the binary tree. (8+7)

A6-R3: Data Structures Through 'C' Language (January 2004)

Note:

- There are TWO PARTS in this Module/Paper. PART ONE contains FOUR questions and PART TWO contains FIVE questions.
- PART ONE is to be answered in the TEAR-OFF ANSWER SHEET only, attached to the question paper, as per the instructions contained therein. PART ONE is NOT to be answered in the answer book.
- Maximum time allotted for PART ONE is ONE HOUR. Answer book for PART TWO will be supplied at the table when the answer sheet for PART ONE is returned. However, candidates, who complete PART ONE earlier than one hour, can collect the answer book for PART TWO immediately after handing over the answer sheet for PART ONE.

TOTAL TIME: 3 HOURS

TOTAL. MARKS : 100
(PART ONE — 40; PART TWO — 60)

PART ONE

(Answer all the questions)

1. Each question below gives a multiple choice of answers. Choose the most appropriate one and enter in the "tear-off" answer sheet attached to the question paper, following instructions therein. (1×10)

1.1 A link field in a structure in 'C' is a place holder for _____
- (a) Memory Address
- (b) Old value of a variable
- (c) A static component
- (d) None of the above

1.2 Inserting a node in a doubly linked list after a given node requires _____
- (a) One pointer changes
- (b) Four pointer change
- (c) Two pointer change
- (d) None of the above

1.3 In 'C', malloc() function returns a pointer to _____
- (a) integer
- (b) character
- (c) structure
- (d) string

1.4 If a binary tree is threaded for an in-order traversal order, a NULL right link of any node is replaced by the address of its _____
- (a) successor

Hidden page

2.5 Storage for an array of structures is statically allocated.
2.6 Insertion sort is also called bubble sort.
2.7 In binary search, keys must be ordered.
2.8 Depth first search uses a stack for storing nodes.
2.9 Minimal spanning tree of a graph represented by an NxN adjacency matrix has N-nodes.
2.10 The C statement

```
int *p;
```

allocates storage for a pointer to an integer.

3. Match words and phrases in Column X with the closest related meaning/word(s)/phrases in Column Y. Enter your selection in the "tear-off" answer sheet attached to the question paper, following instructions therein. (1×10)

X		Y	
3.1	A node with no sub tree	A.	Symbol table
3.2	To return memory at run time	B.	One-dimensional array
3.3	Heap	C.	Leaf node
3.4	Quicksort	D.	Generalized list
3.5	List of lists	E.	A kind of binary tree
3.6	Call by value	F.	Average complexity 0(nlogn)
3.7	Compile time memory allocation	G.	Value copied into local storage of the called function
3.8	Binary Search tree	H.	Buffer management
3.9	Circular queue	I.	Static allocation
3.10	String	J.	Free( )
		K.	High level data type
		L.	Abstract data type

4. Each statement below has a blank space to fit one of the word(s) or phrases in the list below. Enter your choice in the "tear-off" answer sheet attached to the question paper, following instructions therein. (1×10)

a. stack	b. memory	c. increasing
d. FIFO	e. linked	f. depth
g. one	h. two	i. structure
j. malloc( )	k. random	l. queue
m. time	n. sequential	

4.1 In a 'C' program, we can define a composite data type called _____, which may contain fields of type integer, character, float, double, etc.
4.2 The maximum level of any leaf in the tree is also known as _____ of the tree.
4.3 In an AVL tree the heights of the two sub trees of every node never differ by more than _____.
4.4 In a 'C' program, _____ function is used to create new nodes in a linked list dynamically.
4.5 Stack can be implemented using _____ list.
4.6 A priority queue does not follow _____ rule strictly.

4.7 If we store elements in an array, a program can have _____ access to the elements of the array.

4.8 Kruskal's algorithm for building minimal cost spanning tree of a graph considers edges for inclusion in the tree in the _____ order of the cost.

4.9 Recursion uses _____ as a data structure.

4.10 Space complexity of an algorithm indicates its _____ requirement.

PART TWO

(Answer any FOUR questions)

5. (a) A stack contains a set of 10 integers. It is required to assign the integers, from the stack, to variables X, Y, Z in the following order

X = 3rd integer in the stack from bottom

Y = 3rd integer from the top

Z = 5th integer from the top.

Write a 'C' program to perform these operations without changing content or configuration of the stack. You cannot have random access to the elements in stack.

(b) Discuss how quick sort behaves when it is applied to a sorted array? (12+3)

6. (a) Consider the following code:

```
struct address{
char name [60];
char road [100];
struct address *next;
} info;
void Example (struct address *j)
    { static struct address *I=0;
        if (I==0) I=j;
            else I->next=j;
        j->next=0;
        I = j ;
    }
```

Explain how the function Example can be used to construct a linked list. Provide relevant 'C' program.

(b) Explain following concepts:

(i) Garbage Collection

(ii) Merge Sort

(iii) Adjacency list of a graph (9+[2×3])

7. A library uses a computer based information system for keeping track of its collection. With each book, the information system stores the following information:

(i) Accession number of the book, which is an integer

(ii) Issue code which I, B, L, A indicating issued to a member, sent for binding, issued to another library and available respectively.

(iii) Identity of the member, which is an alphanumeric quantity.

The services extended by the library are:

(a) provides status of a requested book.

(b) provides list of all issued books.

(c) provides list of all books issued to a member.

Design a data structure for storing the above information and providing the services efficiently. Provide relevant C program. (15)

8. (a) What is a B-tree? Discuss the algorithm used for insertion of a node into a B-tree.

   (b) Write a 'C' program to evaluate the post-fix expression and show the step for:

   4 2 8 + * 12 4 / - (7+8)

9. a) Describe the Kruskal's algorithm for constructing minimal spanning tree of a graph. Can you comment about time complexity of the algorithm?

   b) Design a data structure for representing a polynomial involving two variables X and Y. Provide an algorithm for addition of two such polynomials. (8+7)

A6-R3: Data Structure through 'C' Language (July 2003)

Note:

- There are TWO PARTS in this Module/Paper.
- PART ONE has to be answered in the "tear-off" answer sheet attached to this question paper, following the instructions therein and NOT in the answer book.
- Maximum time allotted for PART ONE is ONE HOUR. Answer book for PART TWO will be supplied at the table when the answer sheet for PART ONE is collected. However, candidates, who complete PART ONE earlier than one hour, can collect the answer book for PART TWO immediately on handing over the answer sheet for PART ONE.

TOTAL TIME: 3 HOURS **MAX. MARKS : 100**
(PART ONE — 40; PART TWO — 60)

PART ONE

(Answer ALL questions, each question carries one mark)

1. Each question below gives a multiple choice of answers. Choose the most appropriate one and record in the "tear-off" sheet attached to the question paper, following instructions therein. (1×10)

1.1 An array m is declared as
double m[2][4];
Array m has
(a) 2 elements
(b) 4 elements
(c) 8 elements
(d) None of the above

1.2 If A, B and C are inserted into a stack in the lexicographic order, the order of removal will be
(a) A, B, C
(b) C, B, A
(c) B, C, A
(d) None of the above

1.3 If P, Q and R are inserted into a queue in this order, the order or removal will be
(a) P, Q, R
(b) R, Q, P
(c) Q, P, R
(d) None of the above

1.4 If a binary search tree has an inorder traversal of 1, 2, 3, 4, 5, 6 and the root node contains 3 and has 5 as the root of its right subtree, the order in which numbers were inserted in this tree is
(a) 3 was inserted first and 5 was inserted before 4 and 6
(b) 3 was inserted first and 4 was inserted before 5 and 6

(c) 3 was inserted first and 6 was inserted before 4 and 5
(d) None of the above

1.5 The time complexity for MergeSort is:
(a) O (log n)
(b) O (n)
(c) O (nlog n)
(d) O (n*n)

1.6 Prim's algorithm is a method available for finding out the minimum cost of a spanning tree. Its time complexity is given by
(a) O (n*n)
(b) O (n log n)
(c) O (n)
(d) O (1)

1.7 Which of the following algorithms gives best guaranteed performance
(a) heap sort
(b) quick sort
(c) insertion sort
(d) bubble sort

1.8 What is the minimum of nodes required to arrive at a B-tree of order 3 and depth 2? Assume that the root is at depth 0.
(a) 4
(b) 5
(c) 12
(d) 13

1.9 The minimum height of a binary tree having 1000 nodes would be
(a) 10
(b) 11
(c) 100
(d) None of the above

1.10 The number of trees possible with 10 nodes is:
(a) 1000
(b) 1200
(c) 1014
(d) None of the above

2. Each statement below is either TRUE or FALSE. Choose the most appropriate one and record in the "tear-off" sheet attached to the question paper, following instructions therein. (1×10)

2.1 Developing a program from its documentation means that every statement in the program has a comment.

2.2 Each function is executed in the order in which it is defined in the source file.

2.3 When an element is deleted from a linked list, it is automatically returned to the heap.

2.4 All pointers to a node returned to the heap are automatically reset NULL so they cannot reference the node returned to the heap.

2.5 Two elements of the same array cannot be of different data types.
2.6 Strings are always null terminated sequence of characters.
2.7 The minimal spanning tree is the shortest path in a given weighted graph.
2.8 The name of an array with no subscript always refers to the address of the initial array element.
2.9 Recursion is generally more efficient than iteration.
2.10 Function calloc from stdlib statically allocates an array.

3. Match words and phrases in Column X with the nearest in meaning of words and phrases in Column Y. Enter your choice in the tear-off sheet attached to the question paper following instructions therein. (1×10)

	X	Y
3.1	Insertion/Deletion may be carried out at either end	a. Breadth first search algorithm
3.2	Usefully exploits the space consumed by null pointer in a binary tree	b. D-Queue
3.3	All leaves are at the same level	c. B-tree
3.4	Levelwise traversal of the nodes of a binary tree can be achieved by	d. Threaded binary tree
3.5	Connected components of a graph can be found by making repeated use of	e. Inorder traversal
3.6	A vertex in a connected graph which by its deletion disconnected the graph in two or more non-empty components	f. AVL tree
3.7	An example of 'divide and conquer'	g. 0
3.8	The pop operation	h. Removes node from stack
3.9	Number of edges incidents on the vertex	i. Articulation point
3.10	Level of a root node of a tree	j. Binary search algorithm
		k. Degree of vertex
		l. undefined
		m. Recursion
		n. AVL

4. Each statement below has a BLANK space. Find the most appropriate one from the list below to fill up the blank. Enter your choice in the tear-off sheet attached to the question paper following instructions therein. (1×10)

a. exhausted	b. linked list	c. left
d. procedural	e. parameter	f. right
g. Stack	h. tree	i. queue
j. leaf	k. Row	l. major
m. adjacency list	n. Kruskal's	o. Merge sort

4.1 A system designer who is solving a complex problem using _____ abstraction will focus first on what a function is to do leaving the details on how this is accomplished for later.

The McGraw-Hill Companies

ace series

Designed at an introductory level, this book conforms to the syllabus of the DOEACC's 'A' level certificate course on *Data Structures through 'C' Language*. The book will also be useful to the engineering and diploma level students of computer science who take a course on Data Structures.

Salient Features:

- Queues, Lists, Searching and Sorting methods covered comprehensively.
- Use of numerical terminal notations instead of alphabetical ones makes it simpler for the student to work on sequences and traversals.
- Highly illustrative with over 265 diagrams.
- Previous years question papers of DOEACC's 'A' level examinations included as an appendix.

visit us at www.tatamcgrawhill.com
ISBN 0-07-059102-4
9 780070 591028

Tata McGraw-Hill