GLOBAL INSTITUTE OF TECHNOLOGY, JAIPUR

Notes File

Subject: Data Structures and Algorithms

Subject Code: 3CS4-05

Prepared By: Sohan Gupta

Assistant Professor, CSE

RAJASTHAN TECHNICAL UNIVERSITY, KOTA

Syllabus II Year-III Semester: B.Tech. Computer Science and Engineering 3CS4-05: Data Structures and Algorithms

Credit-3 Max. Marks: 150 (IA:30, ETE:120)

Syllabus

1. Stacks: Basic Stack Operations, Representation of a Stack using Static Array and Dynamic Array, Multiple stack implementation using single array, Stack Applications: Reversing list, Factorial Calculation, Infix to postfix Transformation, Evaluating Arithmetic Expressions and Towers of Hanoi.

2. Queues: Basic Queue Operations, Representation of a Queue using array, Implementation of Queue Operations using Stack, Applications of Queues- Round Robin Algorithm. Circular Queues, DeQueue Priority Queues.
   Linked Lists:Introduction, single linked list, representation of a linked list in memory, Different Operations on a Single linked list, Reversing a single linked list, Advantages and disadvantages of single linked list, circular linked list, double linked list and Header linked list.

3. Searching Techniques: Sequential and binary search.Sorting Techniques: Basic concepts, Sorting by: bubble sort, Insertion sort, selection sort, quick sort, heap sort, merge sort, radix sort and counting sorting algorithms.
4. Trees: Definition of tree, Properties of tree, Binary Tree, Representation of Binary trees using arrays and linked lists, Operations on a Binary Tree, Binary Tree Traversals (recursive), Binary search tree, B-tree , B+ tree, AVL tree, Threaded binary tree.
5. Graphs: Basic concepts, Different representations of Graphs, Graph Traversals (BFS & DFS), Minimum Spanning Tree(Prims &Kruskal), Dijkstra’s shortest path algorithms.Hashing: Hash function, Address calculation techniques, Common hashing functions, Collision resolution: Linear and Quadratic probing, Double hashing.

Objective: Why Are Data Structures So Important?

You may wonder why we pay so much attention to data structures and why we review them in such a great details. The reason is we aim to make out of you thinking software engineers. Without knowing the basic data structures and computer algorithms in programming well, you cannot be good developers and risk to stay an amateur. Whoever knows data structures and algorithms well and starts thinking about their correct use has big chance to become a professional – one that analyzes the problems in depth and proposes efficient solutions.

Data Structures and Algorithms are fundamental to programming and to understanding computation. The purpose of this module is to provide students with a coherent introduction to techniques for using data structures and some basic algorithms, with the tools for applying these techniques to computational problems. Teaching and learning methods include lectures and reading material which describe techniques for analysing algorithms and applications of data structures, and a problem sheet which gives students an opportunity to practice problem solving.

Course Outcomes

- Able to understand the concepts of data structure, data type and array data structure.
- Able to analyze algorithms and determine their time complexity.
- Able to implement linked list data structure to solve various problems.
- Able to understand and apply various data structure such as stacks, queues, trees and graphs to solve various computing problems using C programming language.
- Able to implement and know when to apply standard algorithms for searching and sorting.
- Able to effectively choose the data structure that efficiently model the information in a problem

On completion of the module the student should be able to:

- Understand a variety of techniques for designing algorithms.
- Understand a wide variety of data structures and should be able to use them appropriately to solve problems
- Understand some fundamental algorithms.

Content

- Analysis of running time of algorithms: asymptotic notation, analysis of recursive algorithms.
- Efficient algorithms for sorting and selection: selection sort, merge sort, insertion sort, quick sort, binary search.
- Efficient data structures: sets, lists, queues and stacks.
- Dictionary data structures: hash tables, binary search trees.
- Elementary tree and graph algorithms: depth first and breadth first search.

Introduction

There are hundreds of books written on this subject. In the four volumes, named "The Art of Computer Programming", Donald Knuth explains data structures and algorithms in more than 2500 pages. Another author, Niklaus Wirth, has named his book after the answer to the question "why are data structures so important", which is "Algorithms + Data Structures = Programs". The main theme of the book is again the fundamental algorithms and data structures in programming.

If this book is about data structures and algorithms, then perhaps we should start defining these terms.. We begin with a definition for “algorithm..”

Algorithm: A finite sequence of steps for accomplishing some computational task.. An algorithm must Have steps that are simple and definite enough to be done by a computer, and Terminate after finitely many steps.

This definition of an algorithm is similar to others you may have seen in prior computer science courses.. Notice that an algorithm is a sequence of steps, not a program.. You might use the same algorithm in different programs, or express the same algorithm in different languages, because an algorithm is an entity that is abstracted from implementation details.. Part of the point of this course is to introduce you to algorithms that you can use no matter what language you program in.. We will write programs in a particular language, but what we are really studying is the algorithms, not their implementations..

The definition of a data structure is a bit more involved.. We begin with the notion of an abstract data type..

Abstract data type (ADT): A set of values (the carrier set), and operations on those values..

Here are some examples of ADTs:

Boolean—The carrier set of the Boolean ADT is the set { true, false }.. The operations on these values are negation, conjunction, disjunction, conditional, is equal to, and perhaps some others..

Integer—The carrier set of the Integer ADT is the set { .... ., -2, -1, 0, 1, 2, }, and the
operations on these values are addition, subtraction, multiplication, division, remainder, is equal to, is less than, is greater than, and so on.. Note that although some of these operations yield other Integer values, some yield values from other ADTs (like true and false), but all have at least one Integer value argument..

String—The carrier set of the String ADT is the set of all finite sequences of characters from some alphabet, including the empty sequence (the empty string).. Operations on string values include concatenation, length of, substring, index of, and so forth..

Bit String—The carrier set of the Bit String ADT is the set of all finite sequences of bits, including the empty strings of bits, which we denote λ: { λ, 0, 1, 00, 01, 10, 11, 000, ... }.. Operations on bit strings include complement (which reverses all the
bits), shifts (which rotates a bit string left or right), conjunction and disjunction (which combine bits at corresponding locations in the strings, and concatenation and truncation..

The thing that makes an abstract data type abstract is that its carrier set and its operations are mathematical entities, like numbers or geometric objects; all details of implementation on a computer are ignored.. This makes it easier to reason about them and to understand what they are.. For example, we can decide how div and mod should work for negative numbers in the Integer ADT without having to worry about how to make this work on real computers.. Then we can deal with implementation of our decisions as a separate problem.. Once an abstract data type is implemented on a computer, we call it a data type..

Data type: An implementation of an abstract data type on a computer..

Thus, for example, the Boolean ADT is implemented as the boolean type in Java, and the bool type in C++; the Integer ADT is realized as the int and long types in Java, and the Integer class in Ruby; the String ADT is implemented as the String class in Java and Ruby..

Abstract data types are very useful for helping us understand the mathematical objects that we use in our computations, but, of course, we cannot use them directly in our programs.. To use ADTs in programming, we must figure out how to implement them on a computer.. Implementing an ADT requires two things:

Representing the values in the carrier set of the ADT by data stored in computer memory, and

Realizing computational mechanisms for the operations of the ADT..

Finding ways to represent carrier set values in a computer's memory requires that we determine how to arrange data (ultimately bits) in memory locations so that each value of the carrier set has a unique representation.. Such things are data structures..

Data structure: An arrangement of data in memory locations to represent values of the carrier set of an abstract data type..

Realizing computational mechanisms for performing operations of the type really means finding algorithms that use the data structures for the carrier set to implement the operations of the ADT.. And now it should be clear why we study data structures and algorithms together: to implement an ADT, we must find data structures to represent the values of its carrier set and algorithms to work with these data structures to implement its operations..

A course in data structures and algorithms is thus a course in implementing abstract data types.. It may seem that we are paying a lot of attention to a minor topic, but abstract data types are really the foundation of everything we do in computing.. Our computations work on data.. This data must represent things and be manipulated according to rules.. These things and the rules for their manipulation amount to abstract data types..

Usually there are many ways to implement an ADT.. A large part of the study of data structures and algorithms is learning about alternative ways to implement an ADT and evaluating the alternatives to determine their advantages and disadvantages.. Typically some alternatives will be better for certain applications and other alternatives will be better for other applications.. Knowing how to do such evaluations to make good design decisions is an essential part of becoming an expert programmer..

DATA STRUCTURE: -Structural representation of data items in primary memory to do storage& retrieval operations efficiently.

FILE STRUCTURE: Representation of items in secondary memory.

While designing data structure following perspectives to be looked after.

i. Application(user) level: Way of modeling real-life data in specific context.

ii. Abstract(logical) level: Abstract collection of elements & operations.

iii. Implementation level: Representation of structure in programming language.

Data structures are needed to solve real-world problems. But while choosing implementations for it, its necessary to recognize the efficiency in terms of TIME and SPACE.

TYPES:

Simple: built from primitive data types like int, char & Boolean. eg: Array & Structure
Compound: Combined in various ways to form complex structures.
1:Linear: Elements share adjacency relationship& form a sequence. Eg: Stack, Queue , Linked List
2: Non-Linear: Are multi-level data structure. eg: Tree, Graph.

ABSTRACT DATA TYPE :

Specifies the logical properties of data type or data structure. Refers to the mathematical concept that governs them.
They are not concerned with the implementation details like space and time efficiency.
They are defined by 3 components called Triple =(D,F,A) D=Set of domain
F=Set of function A=Set of axioms / rules

Algorithm Complexity

We cannot talk about efficiency of algorithms and data structures without explaining the term "algorithm complexity", which we have already mentioned several times in one form or another. We will avoid the mathematical definitions and we are going to give a simple explanation of what the term means.
Algorithm complexity is a measure which evaluates the order of the count of operations, performed by a given or algorithm as a function of the size of the input data. To put this simpler, complexity is a rough approximation of the number of steps necessary to execute an algorithm. When we evaluate complexity we speak of order of operation count, not of their exact count. For example if we have an order of N^2 operations to process N elements, then $N^2/2$ and $3*N^2$ are of one and the same quadratic order.
Algorithm complexity is commonly represented with the O(f) notation, also known as asymptotic notation or “Big O notation”, where f is the function of the size of the input data. The asymptotic computational complexity O(f) measures the order of the consumed resources (CPU time, memory, etc.) by certain algorithm expressed as function of the input data size.
Complexity can be constant, logarithmic, linear, n*log(n), quadratic, cubic, exponential, etc. This is respectively the order of constant, logarithmic, linear and so on, number of steps, are executed to solve a given problem. For simplicity, sometime instead of “algorithms complexity” or just “complexity” we use the term “running time”.

Complexity and Execution Time

The **execution speed** of a program depends on the complexity of the algorithm, which is executed. If this complexity is low, the program will execute fast even for a big number of elements. If the complexity is high, the program will execute slowly or will not even work (it will hang) for a big number of elements.

If we take an average computer from 2008, we can assume that it can perform about **50,000,000 elementary operations per second**. This number is a rough approximation, of course. The different processors work with a different speed and the different elementary operations are performed with a different speed, and also the computer technology constantly evolves. Still, if we accept we use an average home computer from 2008, we can make the following conclusions about the **speed of execution** of a given program depending on the algorithm complexity and size of the input data.

Best, Worst and Average Case

Complexity of algorithms is usually evaluated in the **worst case** (most unfavorable scenario). This means in the average case they can work faster, but in the worst case they work with the evaluated complexity and not slower.

Let’s take an example: **searching in array**. To find the searched key in the **worst case**, we have to check all the elements in the array. In the **best case** we will have luck and we will find the element at first position. In the average case we can expect to check half the elements in the array until we find the one we are looking for. Hence in the **worst case** the complexity is O(N) – **linear**. In the average case the complexity is O(N/2) = O(N) – linear, because when evaluating complexity one does not take into account the constants. In the best case we have a constant complexity O(1), because we make only one step and directly find the element.

Time complexity, space complexity, and the O-notation

Time complexity

How long does this sorting program run? It possibly takes a very long time on large inputs (that is many strings) until the program has completed its work and gives a sign of life again. Sometimes it makes sense to be able to estimate the running time *before* starting a program. Nobody wants to wait for a sorted phone book for years! Obviously, the running time depends on the number n of the strings to be sorted. Can we find a formula for the running time which depends on n?

Having a close look at the program we notice that it consists of two nested for-loops. In both loops the variables run from 0 to n, but the inner variable starts right from where the outer one just stands. An if with a comparison and some assignments not necessarily executed reside inside the two loops. A good measure for the running time is the number of executed comparisons.[11]

In the first iteration n comparisons take place, in the second n-1, then n-2, then n-3 etc. So 1+2+...+n comparisons are performed altogether. According to the well known Gaussian sum formula these are exactly $1/2\cdot(n-1)\cdot n$ comparisons. Figure 2.8 illustrates this. The screened area corresponds to the number of comparisons executed. It apparently corresponds approx. to half of the area of a square with a side length of n. So it amounts to approx. $1/2\cdot n^2$.

he O-notation

In other words: c is not really important for the description of the running time! To take this circumstance into account, running time complexities are always specified in the so-called *O-notation* in computer science. One says: The sorting method has running time *$O(n^2)$*. The expression O is also called *Landau's symbol*.

space complexity

The better the time complexity of an algorithm is, the faster the algorithm will carry out his work in practice. Apart from time complexity, its *space complexity* is also important: This is essentially the number of memory cells which an algorithm needs. A good algorithm keeps this number as small as possible, too.

There is often a *time-space-tradeoff* involved in a problem, that is, it cannot be solved with few computing time *and* low memory consumption. One then has to make a compromise and to exchange computing time for memory consumption or vice versa, depending on which algorithm one chooses and how one parameterizes it.

Mathematically speaking, $O(n^2)$ stands for a set of functions, exactly for all those functions which, "in the long run", do not grow faster than the function n^2, that is for those functions for which the function n^2 is an upper bound (apart from a constant factor.) To be precise, the following holds true: A function f is an element of the set $O(n^2)$ if there are a factor c and an integer number n_0 such that for all n equal to or greater than this n_0 the following holds:
$f(n) \leq c \cdot n^2$.

The Big O-notation

Time complexities are always specified in the so-called **O-notation**in computer science. One way to describe complexity is by saying that the sorting method has running time **$O(n^2)$**. The expression O is also called **Landau's symbol.**

Mathematically speaking, $O(n^2)$ stands for a set of functions, exactly for all those functions which, "in the long run", do now grow faster than the function n^2, that is for those functions for which the function n^2 is an upper bound (apart from a constant factor). To be precise, the following holds true: A function f is an element of the set $O(n^2)$ if there are a factor c and an integer number n_0 such that for all n equal to or greater than this n_0 the following holds:
$f(n) <= c \cdot n^2$

A function f from $O(n^2)$ may grow considerably more slowly than n^2 so that, mathematically speaking, the quotient f/n^2 converges to 0 with growing n. An example of this is the function $f(n) = n$.

Ways to calculate frequency/efficiency of an algorithm:

Eg. (i):

```
for i = 1 to n
for j = 1 to n
```

i	j	No. of times
1	1 to n	n
2	2 to n	n-1
3	3 to n	n-2
-	–	-
-	–	-
-	–	-
n-1	n-1 to n	2
n	n to n	1

Therefore,

1+2+3+……………+(n-1)+n

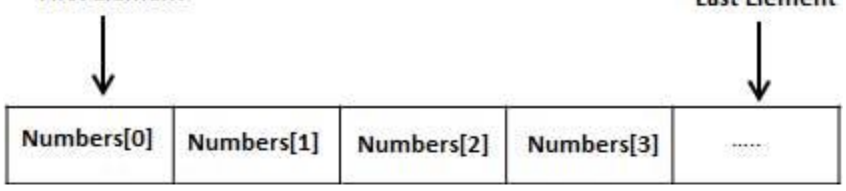
=n(n+1)/2 (Sum of n terms)

Frequency, f (n) = n (n+1)/2 = $n^2/2 + n/2 = 0.5\ n^2 + 0.5\ n$
A higher value of n^2 will be most effective. 0.5 is a negligible value.
Therefore in Big (O) notation, f (n) = O (n^2).

Eg. (ii):
for i = 1 to n
sum = sum + I;
Big (O) Notation, f(n) = O(n)

Linear Search:
1. **Best Case:** When search value is found at first position
f(n) = 1
2. **Worst Case:** Value not in array or in the end
f(n) = n (n number of comparisons are made).
3. **Average Case:** 1, 2, 3…n
(1 + 2 + 3 + ……………………… + n) / n
=n(n+1)/2.n = (n+1)/2
=n/2 + 1/2
=0.5 n + 0.5
f(n) = O(n)
Thus Big-O Notation is a useful measurement tool for measuring time complexity.

Array
C programming language provides a data structure called **the array**, which can store a fixed-size sequential collection of elements of the same type. An array is used to store a collection of data, but it is often more useful to think of an array as a collection of variables of the same type.
Instead of declaring individual variables, such as number0, number1, ..., and number99, you declare one array variable such as numbers and use numbers[0], numbers[1], and ..., numbers[99] to represent individual variables. A specific element in an array is accessed by an index.
All arrays consist of contiguous memory locations. The lowest address corresponds to the first element and the highest address to the last element.



Declaring Arrays
To declare an array in C, a programmer specifies the type of the elements and the number of elements required by an array as follows:
type arrayName [arraySize];
This is called a *single-dimensional* array. The **array Size** must be an integer constant greater than zero and **type** can be any valid C data type. For example, to declare a 10-element array called **balance** of type double, use this statement: double balance [10];

Now *balance* is variable array which is sufficient to hold up to 10 double numbers.

Initializing Arrays

You can initialize array in C either one by one or using a single statement as follows:

double balance[5] = {1000.0, 2.0, 3.4, 7.0, 50.0};

The number of values between braces { } can not be larger than the number of elements that we declare for the array between square brackets [ ].

If you omit the size of the array, an array just big enough to hold the initialization is created. Therefore, if you write:

double balance[] = {1000.0, 2.0, 3.4, 7.0, 50.0};

You will create exactly the same array as you did in the previous example. Following is an example to assign a single element of the array:

balance[4] = 50.0;

The above statement assigns element number 5th in the array with a value of 50.0. All arrays have 0 as the index of their first element which is also called base index and last index of an array will be total size of the array minus 1. Following is the pictorial representation of the same array we discussed above:

Accessing Array Elements

An element is accessed by indexing the array name. This is done by placing the index of the element within square brackets after the name of the array. For example:

double salary = balance[9];

The above statement will take 10th element from the array and assign the value to salary variable. Following is an example which will use all the above mentioned three concepts viz. declaration, assignment and accessing arrays:

```
#include <stdio.h>

int main ()
{
  int n[ 10 ]; /* n is an array of 10 integers */
  int i,j;

  /* initialize elements of array n to 0 */
  for ( i = 0; i < 10; i++ )
  {
    n[ i ] = i + 100; /* set element at location i to i + 100 */
  }

  /* output each array element's value */
  for (j = 0; j < 10; j++ )
  {
    printf("Element[%d] = %d\n", j, n[j] );
  }

  return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
Element[0] = 100
Element[1] = 101
Element[2] = 102
Element[3] = 103
Element[4] = 104
Element[5] = 105
Element[6] = 106
Element[7] = 107
Element[8] = 108
Element[9] = 109
```

C programming language allows multidimensional arrays. Here is the general form of a multidimensional array declaration:

```
type name[size1][size2]...[sizeN];
```

For example, the following declaration creates a three dimensional 5 . 10 . 4 integer array:

```
int threedim[5][10][4];
```

Two-Dimensional Arrays:

The simplest form of the multidimensional array is the two-dimensional array. A two-dimensional array is, in essence, a list of one-dimensional arrays. To declare a two-dimensional integer array of size x,y you would write something as follows:

```
type arrayName [ x ][ y ];
```

Where **type** can be any valid C data type and **arrayName** will be a valid C identifier. A two-dimensional array can be think as a table which will have x number of rows and y number of columns. A 2-dimensional array **a**, which contains three rows and four columns can be shown as below.

Thus, every element in array a is identified by an element name of the form **a[i][j]**, where a is the name of the array, and i and j are the subscripts that uniquely identify each element in a.

Initializing Two-Dimensional Arrays:

Multidimensional arrays may be initialized by specifying bracketed values for each row. Following is an array with 3 rows and each row has 4 columns.

```
int a[3][4] = {
 {0, 1, 2, 3} ,   /*  initializers for row indexed by 0 */
 {4, 5, 6, 7} ,   /*  initializers for row indexed by 1 */
 {8, 9, 10, 11}   /*  initializers for row indexed by 2 */
};
```

The nested braces, which indicate the intended row, are optional. The following initialization is equivalent to previous example:

```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

Accessing Two-Dimensional Array Elements:

An element in 2-dimensional array is accessed by using the subscripts, i.e., row index and column index of the array. For example:

```
int val = a[2][3];
```

The above statement will take 4th element from the 3rd row of the array. You can verify it in the above diagram. Let us check below program where we have used nested loop to handle a two dimensional array:

```
#include <stdio.h>

int main ()
{
  /* an array with 5 rows and 2 columns*/
  int a[5][2] = { {0,0}, {1,2}, {2,4}, {3,6},{4,8}};
  int i, j;

  /* output each array element's value */
  for ( i = 0; i < 5; i++ )
  {
    for ( j = 0; j < 2; j++ )
    {
      printf("a[%d][%d] = %d\n", i,j, a[i][j] );
    }
  }
  return 0;
}
```

When the above code is compiled and executed, it produces the following result:

```
a[0][0]: 0
a[0][1]: 0
a[1][0]: 1
a[1][1]: 2
a[2][0]: 2
a[2][1]: 4
a[3][0]: 3
a[3][1]: 6
a[4][0]: 4
a[4][1]: 8
```

As explained above, you can have arrays with any number of dimensions, although it is likely that most of the arrays you create will be of one or two dimensions.

Array Representation:

We can represrnt an array in types Column-major and Row-major

Arrays may be represented in Row-major form or Column-major form. In Row-major form, all the elements of the first row are printed, then the elements of the second row and so on upto the last row. In Column-major form, all the elements of the first column are printed, then the elements of the second column and so on upto the last column. The ‘C’ program to input an array of order m x n and print the array contents in row major and column major is given below. The following array elements may be entered during run time to test this program:

Input: Rows: 3, Columns: 3;

```
1   2   3
4   5   6
```

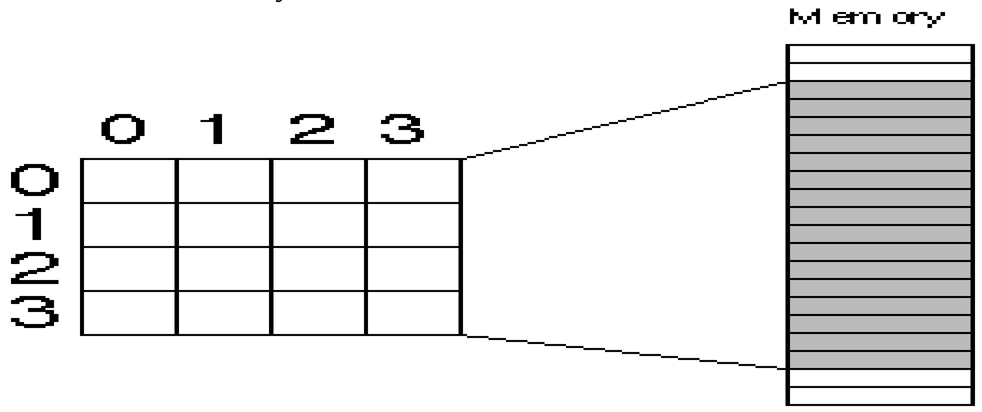
```
7     8     9
```
Output:

Row Major:
```
1     2     3
4     5     6
7     8     9
```
Column Major:
```
1     4     7
2     5     8
3     6     9
```

/* Program to input an array and display in row-major and column major form */
```
#include <stdio.h>
#define MAX 10
Intmain(){
int arr[MAX][MAX],m,n,i,j;
printf("nEnter total number of rows?");
scanf("%d",&m);
printf("nEnter total number of columns?");
scanf("%d",&n);
for(i=0;i<m;i++){
for(j=0;j<n;j++){
printf("nEnter number?");
scanf("%d",&arr[i][j]);
}
}
printf("nnRow-Major Order:n");
for(i=0;i<m;i++){
for(j=0;j<n;j++){
printf("%dt",arr[i][j]);
}
printf("n");
}
printf("nnColumn-Major Order:n");
for(i=0;i<m;i++){
for(j=0;j<n;j++){
printf("%dt",arr[j][i]);
}
printf("n");
}
}
```
1.3.8 Multidimensional Arrays
The 80x86 hardware can easily handle single dimension arrays. Unfortunately, there is no magic addressing mode that lets you easily access elements of multidimensional arrays. That's going to take some work and lots of instructions.

Before discussing how to declare or access multidimensional arrays, it would be a good idea to figure out how to implement them in memory. The first problem is to figure out how to store a multi-dimensional object into a one-dimensional memory space.

Consider for a moment a Pascal array of the form `A: array [0...3, 0...3] of char`. This array contains 16 bytes organized as four rows of four characters. Somehow you've got to draw a correspondence with each of the 16 bytes in this array and 16 contiguous bytes in main memory. The figure below shows one way to do this.



Array as storage elements

An *array element* is one of the scalar data items that make up an array. A subscript list (appended to the array or array component) determines which element is being referred to. A reference to an array element takes the following form:

array(*subscript-list*)

array

Is the name of the array.

subscript-list

Is a list of one or more subscripts separated by commas. The number of subscripts must equal the rank of the array.

Each subscript must be a scalar integer (or other numeric) expression with a value that is within the bounds of its dimension.

Representation of Multidimensional Array:

For example, in two-dimensional array BAN, element BAN (1,2) has a subscript order value of 4; in three-dimensional array BOS, element BOS(1,1,1) has a subscript order value of 1.

In an array section, the subscript order of the elements is their order within the section itself. For example, if an array is declared as B(20), the section B(4:19:4) consists of elements B(4), B(8), B(12), and B(16). The subscript order value of B(4) in the array section is 1; the subscript order value of B(12) in the section is 3.

1.1 Basics of Stacks

Introduction:

A stack is an Abstract Data Type (ADT), commonly used in most programming languages. It is named stack as it behaves like a real-world stack, for example – a deck of cards or a pile of plates, etc.

Example

A real-world stack allows operations at one end only. For example, we can place or remove a card or plate from the top of the stack only. Likewise, Stack ADT allows all data operations at one end only. At any given time, we can only access the top element of a stack.

This feature makes it LIFO data structure. LIFO stands for Last-in-first-out. Here, the element which is placed (inserted or added) last, is accessed first. In stack terminology, insertion operation is called **PUSH** operation and removal operation is called **POP** operation.

1.2 Basic Operations on Stack:

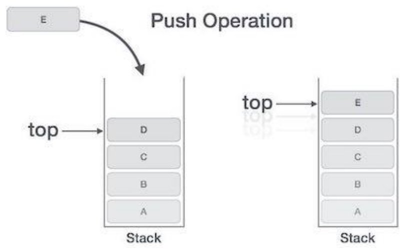
A stack is a container of objects that are inserted and removed according to the last-in first-out (LIFO) principle. In the pushdown stacks only two operations are allowed: **push** the item into the stack, and **pop** the item out of the stack. A stack is a limited access data structure - elements can be added and removed from the stack only at the top. **push** adds an item to the top of the stack, **pop** removes the item from the top. A helpful analogy is to think of a stack of books; you can remove only the top book, also you can add a new book on the top.

A stack is a **recursive** data structure. Here is a structural definition of a Stack: a stack is either empty or it consists of a top and the rest which is a stack;

1.2.1 Push Operation

The process of putting a new data element onto stack is known as a Push Operation. Push operation involves a series of steps −

- **Step 1** − Checks if the stack is full.
- **Step 2** − If the stack is full, produces an error and exit.
- **Step 3** − If the stack is not full, increments **top** to point next empty space.
- **Step 4** − Adds data element to the stack location, where top is pointing.
- **Step 5** − Returns success.



If the linked list is used to implement the stack, then in step 3, we need to allocate space dynamically.

Algorithm for PUSH Operation

A simple algorithm for Push operation can be derived as follows −

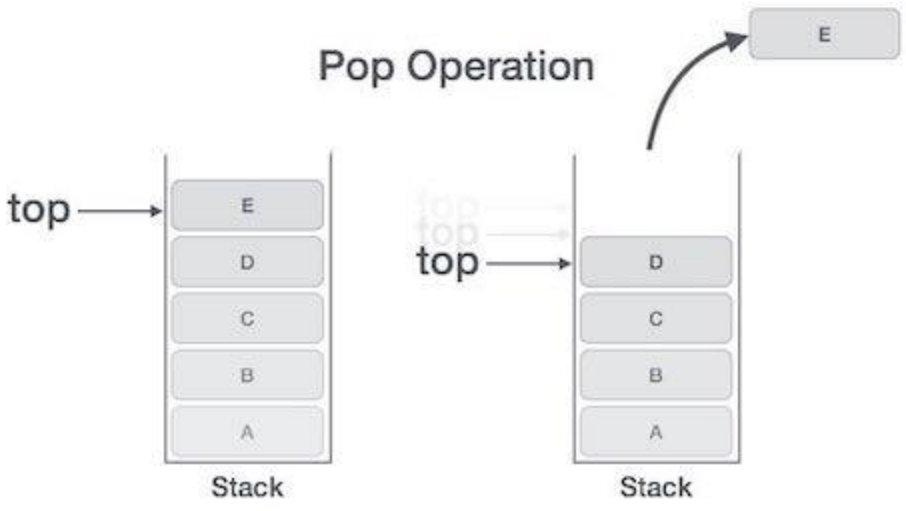
```
begin procedure push: stack, data

   if stack is full
      return null
   endif

   top ← top + 1
   stack[top] ← data

end procedure
```

1.2.2 Pop Operation

Accessing the content while removing it from the stack, is known as a Pop Operation. In an array implementation of pop() operation, the data element is not actually removed, instead **top** is decremented to a lower position in the stack to point to the next value. But in linked-list implementation, pop() actually removes data element and deallocates memory space.

A Pop operation may involve the following steps −

- **Step 1** − Checks if the stack is empty.
- **Step 2** − If the stack is empty, produces an error and exit.
- **Step 3** − If the stack is not empty, accesses the data element at which **top** is pointing.
- **Step 4** − Decreases the value of top by 1.
- **Step 5** − Returns success.



Algorithm for Pop Operation

A simple algorithm for Pop operation can be derived as follows −

```
begin procedure pop: stack

   if stack is empty
      return null
   endif

   data ← stack[top]
   top ← top - 1
   return data

end procedure
```

To use a stack efficiently, we need to check the status of stack as well. For the same purpose, the following functionality is added to stacks −
peek() − get the top data element of the stack, without removing it.
isFull() − check if stack is full.
isEmpty() − check if stack is empty.
At all times, we maintain a pointer to the last PUSHed data on the stack. As this pointer always represents the top of the stack, hence named top. The top pointer provides top value of the stack without actually removing it.
First we should learn about procedures to support stack functions −

peek()

Algorithm of peek() function −

```
begin procedure peek
   return stack[top]
end procedure
```

Implementation of peek() function in C programming language −

Example

```
int peek() {
   return stack[top];
}
```

isfull()

Algorithm of isfull() function −

```
begin procedure isfull

   if top equals to MAXSIZE
      return true
   else
      return false
   endif

end procedure
```

Implementation of isfull() function in C programming language −

Example

```
bool isfull() {
   if(top == MAXSIZE)
      return true;
   else
      return false;
}
```

isempty()

Algorithm of isempty() function −

```
begin procedure isempty

   if top less than 1
      return true
   else
      return false
   endif

end procedure
```

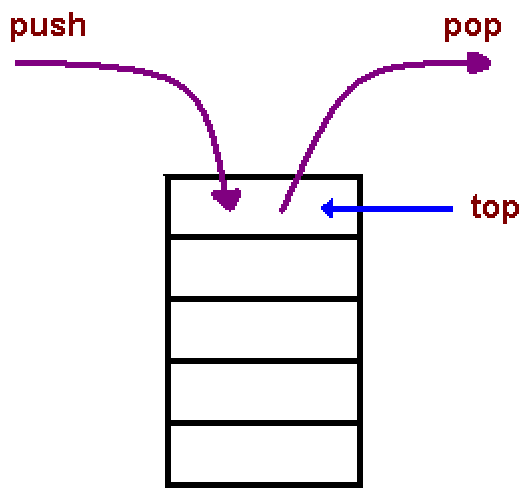
Implementation of isempty() function in C programming language is slightly different. We initialize top at -1, as the index in array starts from 0. So we check if the top is below zero or -1 to determine if the stack is empty. Here's the code −

Example

```
bool isempty() {
   if(top == -1)
      return true;
   else
      return false;
}
```

1.3 Stack Representation

The following diagram depicts a stack and its operations −

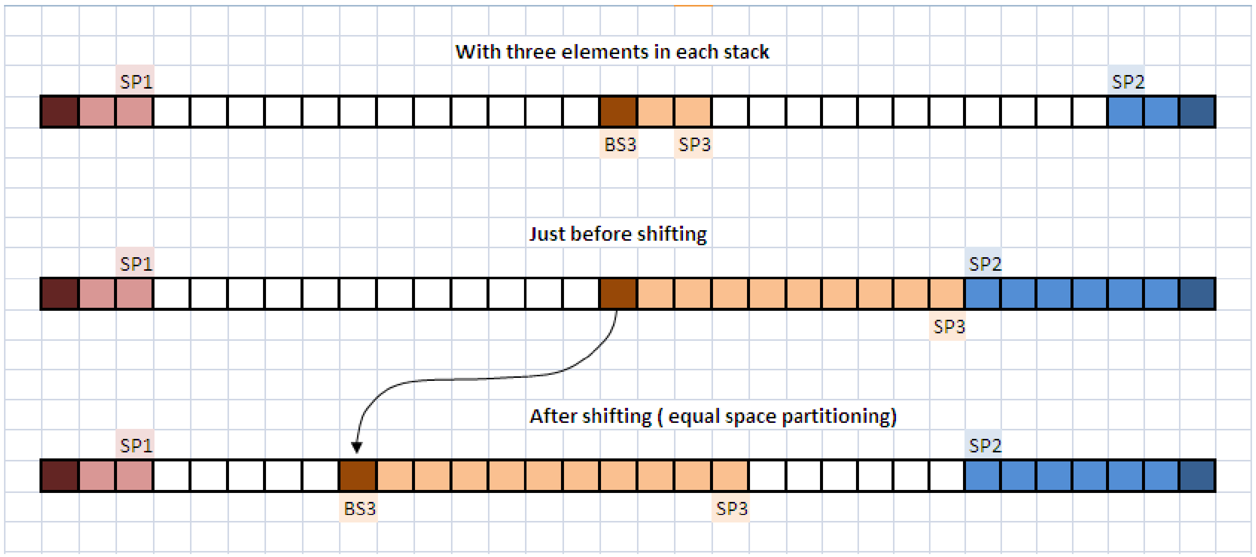


A stack can be implemented by means of Array, Structure, Pointer, and Linked List. Stack can either be a fixed size one or it may have a sense of dynamic resizing. Here, we are going to implement stack using arrays, which makes it a fixed size stack implementation.

- Dynamic memory = memory allocated at run-time.
- Static memory = memory allocated at compile-time.
- Programming languages like C and Pascal provide libraries for managing dynamic memory. This memory is allocated from a special area called heap. The heap is managed by a special language component that implements functions like new and dispose in Pascal or malloc(), calloc() and free() in C.
- In languages that do not provide facilities for dynamic memory allocation it is possible to simulate it using static memory. Also, even if the pointer data type is not supported , it is possible to simulate it using integer indices.
- The idea is to define the heap as a static block of memory and define a set of operations for managing objects allocated in this block. To simplify things, we shall assume that all the objects have the same size (i.e. they are homogenous).

1.4 **Multiple stack implementation using single array :**

To implement multiple stacks in a single array, one approach is to divide the array in k slots of size n/k each, and fix the slots for different stacks, we can use arr[0] to arr[n/k-1] for first stack, and arr[n/k] to arr[2n/k-1] for stack2 and so on where arr[] is the array of size n.


With three elements in each stack
SP1
SP2
BS3
SP3
Just before shifting
SP1
SP2
SP3
After shifting (equal space partitioning)
SP1
SP2
BS3
SP3

Although this method is easy to understand, but the problem with this method is inefficient use of array space.A stack push operation may result in stack overflow even if there is space available in arr[].

Examples:

Input :Enter the Number of stacks

Output :5 Input : 1.Push 2.pop 3.Display 4.exit

Input :Enter your choice

Output :1Input :Enter the stack number

Output :2

Input :Enter element to push

Output :15Input :Enter your choice

Output :2

Input :Enter the stack no to pop

Output :2

Output :15 from stack 2 is deletedInput: Enter your choice

output: 3

output: Stack is emptyInput :Enter your choice

Output :4

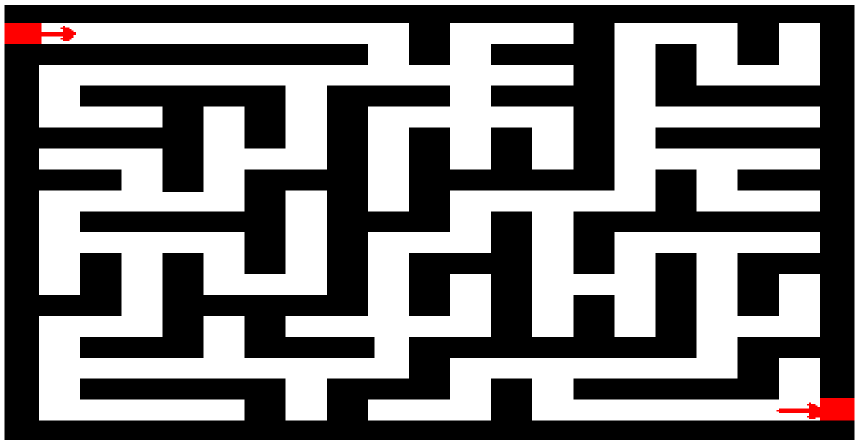
Output :Program Terminated

Algorithm:

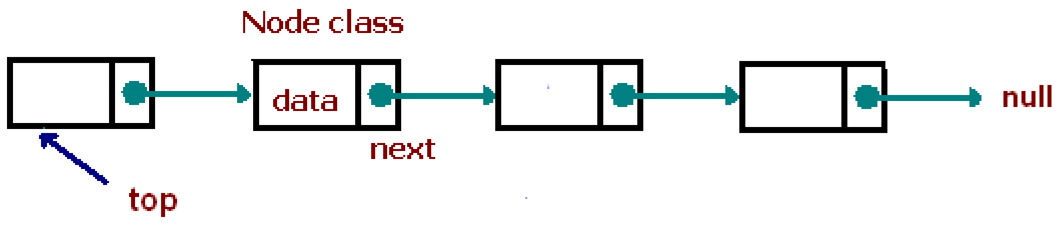
1. Here we use 2 arrays *min[]* and *max[]* to represent the lower and upper bounds for a stack
2. Array *s[]* stores the elements of the stack
3. Array *top[]* is used to store the top index for each stack
4. Variable *ns* represents the stack number
5. Variable **size** represents the size for each stack in an array
6. First we build a function *init()* to initialize the starting values
7. Then we have a function *createstack()* to create the stack
8. Function *Push()* & *Pop()* are used to push and pop an element to and from the stack
9. Function *Display()* is used to display the elements in a particular stack

1.5 Applications of Stack

- The simplest application of a stack is to reverse a word. You push a given word to stack - letter by letter - and then pop letters from the stack.
- Another application is an "undo" mechanism in text editors; this operation is accomplished by keeping all text changes in a stack.
- **Backtracking**. This is a process when you need to access the most recent data element in a series of elements. Think of a labyrinth or maze - how do you find a way from an entrance to an exit?



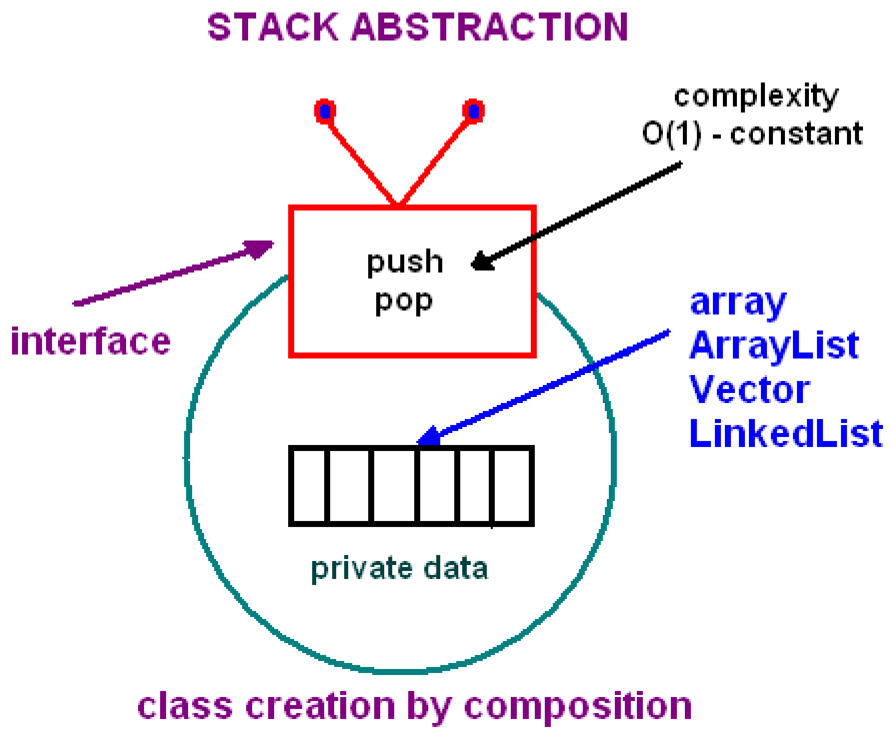
- Once you reach a dead end, you must backtrack. But backtrack to where? to the previous choice point. Therefore, at each choice point you store on a stack all possible choices. Then backtracking simply means popping a next choice from the stack.



Implementation of Stack

In the standard library of classes, the data type stack is an *adapter* class, meaning that a stack is built on top of other data structures. The underlying structure for a stack could be an array, a vector, an Array List, a linked list, or any other collection. Regardless of the type of the underlying data structure, a Stack must implement the same functionality. This is achieved by providing a unique interface:

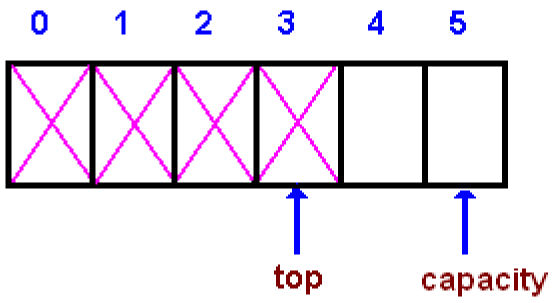
The following picture demonstrates the idea of implementation *by composition*.



Another implementation requirement (in addition to the above interface) is that all stack operations must run in **constant time O(1)**. Constant time means that there is some constant k such that an operation takes k nanoseconds of computational time regardless of the stack size.

Array-based implementation

In an array-based implementation we maintain the following fields: an array A of a default size (≥ 1), the variable *top* that refers to the top element in the stack and the *capacity* that refers to the array size. The variable *top* changes from -1 to capacity - 1. We say that a stack is empty when top = -1, and the stack is full when top = capacity-1.



In a fixed-size stack abstraction, the capacity stays unchanged, therefore when *top* reaches *capacity*, the stack object throws an exception. See ArrayStack.java for a complete implementation of the stack class.
In a dynamic stack abstraction when *top* reaches *capacity*, we double up the stack size.

1.6 Recursion

Recursion in computer science is a method where the solution to a problem depends on solutions to smaller instances of the same problem (as opposed to iteration).[1] The approach can be applied to many types of problems, and recursion is one of the central ideas of computer science.
"The power of recursion evidently lies in the possibility of defining an infinite set of objects by a finite statement. In the same manner, an infinite number of computations can be described by a finite recursive program, even if this program contains no explicit repetitions."

Example of recursion in C programming
Write a C program to find sum of first n natural numbers using recursion. Note: Positive integers are known as natural number i.e. 1, 2, 3....n

```
#include <stdio.h>
int sum(int n);
int main(){
   int num,add;
   printf("Enter a positive integer:\n");
   scanf("%d",&num);
   add=sum(num);
   printf("sum=%d",add);
}
int sum(int n){
   if(n==0)
     return n;
   else
     return n+sum(n-1);    /*self call  to function sum() */
}
```

Output
```
Enter a positive integer:
```

```
5
15
```

In, this simple C program, `sum()` function is invoked from the same function. If *n* is not equal to 0 then, the function calls itself passing argument 1 less than the previous argument it was called with. Suppose, *n* is 5 initially. Then, during next function calls, 4 is passed to function and the value of argument decreases by 1 in each recursive call. When, n becomes equal to 0, the value of n is returned which is the sum numbers from 5 to 1.

For better visualization of recursion in this example:

```
sum(5)
=5+sum(4)
=5+4+sum(3)
=5+4+3+sum(2)
=5+4+3+2+sum(1)
=5+4+3+2+1+sum(0)
=5+4+3+2+1+0
=5+4+3+2+1
=5+4+3+3
=5+4+6
=5+10
=15
```

Every recursive function must be provided with a way to end the recursion. In this example when, *n* is equal to 0, there is no recursive call and recursion ends.

1.6.1 **Reverse a Stack using Recursion :**

Given a stack of integers, we have to reverse the stack elements using recursion. We cannot any loop like for, while etc and we can only use push, pop, isEmpty and isFull functions of given stack.

```
                Input Stack
                  2  <--- Top
                  4
                  8
                  9
Output Stack
                9 <--- Top
                8
               4
               2
```

Here we are going to use recursion to reverse the stack elements. We will store the top elements of stack on function stack one by one until stack becomes empty. When stack becomes empty, we will insert an element at the bottom of stack and then insert all the elements stores in function stack back in same sequence. Here we will use two user defined functions "insertAtBottom" and "reverse".

void insertAtBottom(int num) : This function inserts a number "num" at the bottom of stack using recursion.
void reverse() : This function pop's top element and store it in function stack. Then it recursively calls itself to reverse remaining stack. Once remaining stack elements are reversed, it calls insertAtBottom function to insert top element at the bottom of stack.

1.6.2 Factorial Program in C: Factorial of n is the product of all positive descending integers. Factorial of n is denoted by n!. For example:
5! = 5*4*3*2*1 = 120
3! = 3*2*1 = 6
Program:

```c
#include<stdio.h>
int main()
{
 int i,fact=1,number;
 printf("Enter a number: ");
  scanf("%d",&number);
    for(i=1;i<=number;i++){
      fact=fact*i;
  }
  printf("Factorial of %d is: %d",number,fact);
return 0;
}
```

Output:

```
Enter a number: 5
Factorial of 5 is: 120
```

Types of recursion
Single recursion and multiple recursion
Recursion that only contains a single self-reference is known as **single recursion**, while recursion that contains multiple self-references is known as **multiple recursion**. Standard examples of single recursion include list traversal, such as in a linear search, or computing the factorial function, while standard examples of multiple recursions include tree traversal, such as in a depth-first search, or computing the Fibonacci sequence.
Indirect recursion
Most basic examples of recursion, and most of the examples presented here, demonstrate ***direct* recursion**, in which a function calls itself. *Indirect* recursion occurs when a function is called not by itself but by another function that it called (either directly or indirectly). For example, if *f* calls *f*, that is direct recursion, but if *f* calls *g* which calls *f*, then that is indirect recursion of *f*. Chains of three or more functions are possible; for example, function 1 calls function 2, function 2 calls function 3, and function 3 calls function 1 again.

Advantages and Disadvantages of Recursion

Recursion is more elegant and requires few variables which make program clean. Recursion can be used to replace complex nesting code by dividing the problem into same problem of its sub-type.

In other hand, it is hard to think the logic of a recursive function. It is also difficult to debug the code containing recursion.

1.7 Evaluating Arithmetic Expressions Using Stack

An important application of stacks is in parsing. For example, a compiler must parse arithmetic expressions written using infix notation:

1 + ((2 + 3) * 4 + 5)*6

We break the problem of parsing infix expressions into two stages. First, we convert from infix to a different representation called postfix. Then we parse the postfix expression, which is a somewhat easier problem than directly parsing infix.

The stack organization is very effective in evaluating arithmetic expressions. Expressions are usually represented in what is known as Infix notation, in which each operator is written between two operands (i.e., A + B). With this notation, we must distinguish between (A + B)*C and A + (B * C) by using either parentheses or some operator-precedence convention. Thus, the order of operators and operands in an arithmetic expression does not uniquely determine the order in which the operations are to be performed.

1. **Polish notation (prefix notation) –**
   It refers to the notation in which the operator is placed before its two operands . Here no parentheses are required, i.e.,
   ```
   +AB
   ```
2. **Reverse Polish notation(postfix notation) –**
   It refers to the analogous notation in which the operator is placed after its two operands. Again, no parentheses is required in Reverse Polish notation, i.e.,
   ```
   AB+
   ```

1.8 Transformation of Arithmetic Expressions

In the stack, we can convert the arithmetic expression in infix to prefix or postfix as per your requirement .they are basically two types

1. **Converting from Infix to Postfix.**
2. **Converting from Infix to Prefix**

1.8.1 Converting from Infix to Postfix.

Typically, we deal with expressions in infix notation

2 + 5

where the operators (e.g. +, *) are written between the operands (e.q, 2 and 5). Writing the operators after the operands gives a postfix expression 2 and 5 are called operands, and the '+' is operator. The above arithmetic expression is called infix, since the operator is in between operands. The expression

2 5 +

Writing the operators before the operands gives a prefix expression

+2 5

Suppose you want to compute the cost of your shopping trip. To do so, you add a list of numbers and multiply them by the local sales tax (7.25%):

70 + 150 * 1.0725

Depending on the calculator, the answer would be either 235.95 or 230.875. To avoid this confusion we shall use a postfix notation

70 150 + 1.0725 *

Postfix has the nice property that parentheses are unnecessary.

Now, we describe how to convert from infix to postfix.

1. Read in the tokens one at a time
2. If a token is an integer, write it into the output
3. If a token is an operator, push it to the stack, if the stack is empty. If the stack is not empty, you pop entries with higher or equal priority and only then you push that token to the stack.
4. If a token is a left parentheses '(', push it to the stack
5. If a token is a right parentheses ')', you pop entries until you meet '('.
6. When you finish reading the string, you pop up all tokens which are left there.
7. Arithmetic precedence is in increasing order: '+', '-', '*', '/';

Example. Suppose we have an infix expression:2+(4+3*2+1)/3. We read the string by characters.

'2' - send to the output.
'+' - push on the stack.
'(' - push on the stack.
'4' - send to the output.
'+' - push on the stack.
'3' - send to the output.
'*' - push on the stack.
'2' - send to the output.

1.8.2 Evaluating a infix to Postfix Expression. We describe how to parse and evaluate a postfix expression.

1. We read the tokens in one at a time.
2. If it is an integer, push it on the stack
3. If it is a binary operator, pop the top two elements from the stack, apply the operator, and push the result back on the stack.

Consider the following postfix expression

5 9 3 + 4 2 * * 7 + *

Here is a chain of operations

Stack Operations	Output
push(5);	5
push(9);	5 9
push(3);	5 9 3
push(pop() + pop())	5 12
push(4);	5 12 4
push(2);	5 12 4 2
push(pop() * pop())	5 12 8

```
push(pop() * pop())        5 96
push(7)                     5 96 7
push(pop() + pop())         5 103
push(pop() * pop())        515
```

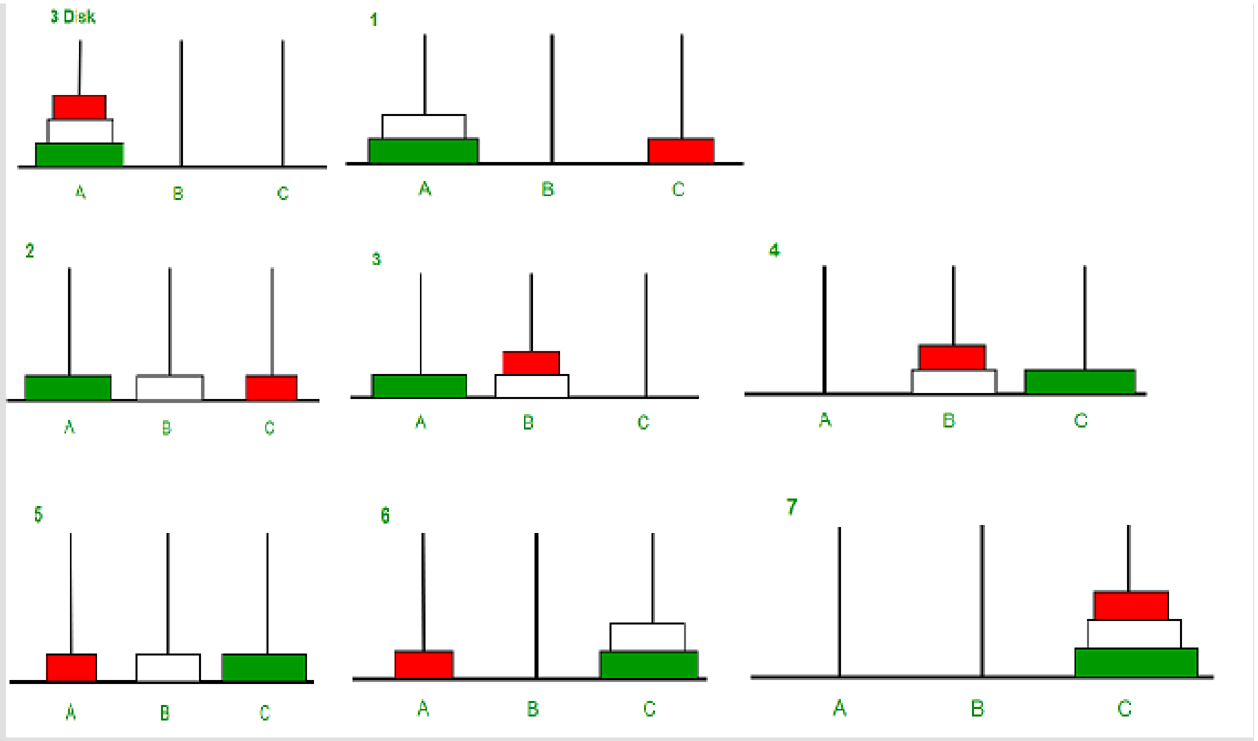
Note, that division is not a commutative operation, so 2/3 is not the same as 3/2.

1.9 Tower of Hanoi :

Tower of Hanoi is a mathematical puzzle where we have three rods and n disks. The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:
1) Only one disk can be moved at a time.
2) Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
3) No disk may be placed on top of a smaller disk.

Example :

```
Take an example for 2 disks :

Let rod 1 = 'A', rod 2 = 'B', rod 3 = 'C'.


Step 1 : Shift first disk from 'A' to 'B'.

Step 2 : Shift second disk from 'A' to 'C'.

Step 3 : Shift first disk from 'B' to 'C'.


The pattern here is :

Shift 'n-1' disks from 'A' to 'B'.

Shift last disk from 'A' to 'C'.

Shift 'n-1' disks from 'B' to 'C'.


Image illustration for 3 disks :
```



Program:

```
#include<stdio.h>

void TOH(int n,char x,char y,char z) {

   if(n>0) {

      TOH(n-1,x,z,y);

      printf("\n%c to %c",x,y);

      TOH(n-1,z,y,x);

   }

}

int main() {

   int n=3;

   TOH(n,'A','B','C');

}
```

Important Questions:

1. Write a program to input an array of five elements and print that array?
2. Write a program which stores the marks of 5 students in an array and then print all the marks. The marks of students are 75,85,80,95,90.
3. Write a program to demonstrate the working of array.
4. Program to read and write array and find the sum of Even and odd.
5. What is an array ?
6. Differentiate between one-dimensional and two -dimensional arrays.
7. What are limitations of arrays?
8. Give an example to show the usefulness of an arrays?
9. Name some areas to application of two-dimensional arrays.
10. How a two-dimensional array is represented in Memory?
11. Give postfix form for   A+ [(B+C) + (D+E)*F]/G
12. What is a stack? List the application of the stack in computers.
13. What are the basic operations performed on a stack?
14. How do you represent a stack in C ?
15. Why stack is called a LIFO data structure ?
16. What is the significance of the top in a stack ?
17. When do you say stack is full and stack is empty ?
18. Write algorithm in C to perform push and pop operations.
19. Differentiate between prefix and postfix expressions.
20. Evaluate the following postfix expressions

    4589+-+

    4025 + 2922-24+

    24159^/10+
21. Devise an algorithm which translates a postfix expression to a prefix expression.