

PREFACE

1
Chapter

INTRODUCTION TO DATA STRUCTURE

Introduction.....	2
What is data structure?	2
Need of Data Structures.....	2
Data Structure Classification	3
Primitive Data Structure	3
Non-primitive Data Structure.....	4
Linear Data Structures	4
Non Linear Data Structures.....	4
Representation of Data Structures.....	5
Sequential Representation	5
Drawback of Sequential Representation	5
Linked Representation.....	5
Abstract Data type (ADT)	6
Advantages of Data Structures.....	7
Algorithm analysis.....	8
Computational and Asymptotic Complexity	9
Random Access Machine Model (RAM).....	11
Detailed analysis.....	12
The Best, Average, and Worst Cases.....	14
<input checked="" type="checkbox"/> Multiple Choice Questions.....	15
<input type="checkbox"/> Discussion Exercise	16

2
Chapter

THE STACKS AND ITS APPLICATIONS

What is stack?	18
Basic features of Stack	18
Applications of Stack.....	18
The Stack ADT.....	19
Implementation of Stack	19

Array (static) implementation of a stack	19
Algorithm for Push Operation of Stack.....	20
Deletion of an element from a stack (Pop operation)	21
Visiting each element of the stack (Peek or Display operation).....	22
Analysis of Stack Operations	24
Stack Applications	24
Infix, Prefix and Postfix Notation.....	24
Infix expression	24
Prefix notation.....	24
Algorithm to convert infix to postfix notation	28
Trace of Conversion of infix to postfix Algorithm.....	28
Converting an Infix expression to prefix expression.....	31
Converting an Infix expression to prefix expression by using stack.....	31
Arithmetic Expression Evaluation	41
<input checked="" type="checkbox"/> Multiple Choice questions.....	50
<input type="checkbox"/> Discussion Exercise	51

3 Chapter

QUEUE

Introduction.....	54
What is a queue?.....	54
Basic features of Queue.....	55
Applications of Queue	55
Basic Operations of queue	55
The Queue as ADT	56
Implementation of queue.....	56
Linear queue	56
Circular queue.....	61
Initialization of Circular queue	62
Declaration of a Circular Queue	62
Priority Queue.....	69
Types of priority queues.....	71
The priority queue ADT	71
<input checked="" type="checkbox"/> Multiple Choice questions.....	75
<input type="checkbox"/> Discussion Exercise	66

4

Chapter

LIST

Introduction.....	78
Static Data Structure vs. Dynamic Data Structure.....	78
What is a Static Data structure?	78
What is Dynamic Data Structure?	78
Array Implementation of Lists.....	78
Deleting of an existing element from given list.....	80
Modification of an existing element of given list.....	81
Traversing of an existing list.....	81
Merging of any two existing list.....	82
Queues as a list	86
<input checked="" type="checkbox"/> Multiple Choice questions.....	87
<input type="checkbox"/> Discussion Exercise	88

5

Chapter

LINKED LISTS

Introduction.....	90
Linked List.....	90
Uses of Linked List	91
Self Referential Structure	91
Header Nodes.....	91
Singly Linked list	93
Circular Linked list.....	104
Doubly Linked List (DLL)	110
Circular Doubly Linked List	117
Linked list implementation of stack (or dynamic implementation of stack)	124
Linked List Implementation of Circular Queue.....	133
<input checked="" type="checkbox"/> Multiple Choice questions.....	138
<input type="checkbox"/> Discussion Exercise	139

6

Chapter

RECUSION

Introduction.....	142
Divide-and-conquer algorithms	142
Key differences between recursion and iteration.....	143
Recursion Examples.....	143
Tower of Hanoi problem	148
<input checked="" type="checkbox"/> Multiple Choice questions.....	157
<input type="checkbox"/> Discussion Exercise	158

7

Chapter

TREE

Introduction.....	160
Definition	160
Terminology	161
Binary Trees.....	162
Properties of Binary tree	162
Advantages of Binary Tree:.....	163
Types of binary tree.....	163
Strictly binary tree.....	163
Complete binary tree.....	163
Almost complete binary tree.....	164
Operations on Binary Tree	165
Binary Tree Representation.....	165
Binary Tree Traversal Techniques.....	167
Pre-order traversal.....	167
In-order traversal	168
Post-order traversal	169
Expression Tree	169
Binary Search Tree (BST).....	170
Advantages of using binary search tree	172
Operations on Binary search tree (BST).....	173
Algorithm for BST searching.....	174
Insertion of a node in BST.....	175
Deleting a node from the BST	177
Huffman algorithm.....	184
AVL Tree.....	186
Left Rotation	187
Deleting data element from AVL tree.....	193
Multi-way tree.....	199
B Tree.....	199
Operations on a B-Tree	200
Deleting from a B-tree	203

<input checked="" type="checkbox"/> Multiple Choice questions.....	211
<input type="checkbox"/> Discussion Exercise	212

8

Chapter

SORTING

Introduction.....	214
In-place	214
Stable.....	214
Bubble Sort	215
Characteristics of Bubble Sort	215
Selection Sort.....	217
Insertion Sort.....	220
Quick Sort.....	223
Merge Sort.....	228
Shell Sort.....	232
Radix Sort.....	235
Heaps	238
Array Representation of Heaps	238
Inserting element to an existing heap	239
Heaps as Priority Queues.....	241
Operations on Heaps.....	241
<input checked="" type="checkbox"/> Multiple Choice questions.....	249
<input type="checkbox"/> Discussion Exercise	250

9

Chapter

SEARCHING

Introduction.....	254
Sequential Search.....	254
Binary Search.....	255
Efficiency	257
Tree Search.....	258
Hashing	259
Hash Function	259
Types of Hash functions	260
Extraction	261
Hash Collision.....	261
Collision Resolution	261
Quadratic Probing	266
Double Hashing.....	267
• Chaining	268
• Bucket Addressing.....	270
<input checked="" type="checkbox"/> Multiple Choice questions.....	271
<input type="checkbox"/> Discussion Exercise	272

10

Chapter

GRAPHS

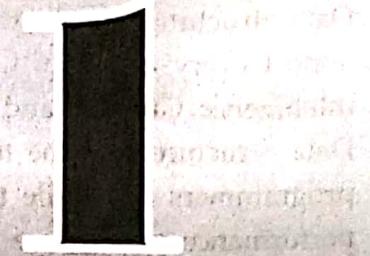
Introduction	274
Directed and Undirected Graph	274
Applications of graph.....	275
Directed multi-graph.....	276
Simple and multi-graphs	276
Pseudo-graphs.....	277
Mixed graph	277
Graph Terminology	278
Graph as an ADT	278
Graph Representation	279
Adjacency List	279
Adjacency matrix representation of graph.....	280
Incidence Matrices	280
Graph Traversals	280
Depth First Traversals (DFS)	281
Complete C program for DFS	288
Shortest Paths.....	292
Dijkstra's Algorithm.....	294
Steps in Dijkstra's algorithm	295
Floyd's Warshall Algorithm.....	295
Spanning tree.....	298
Minimum Spanning Tree	301
Kruskal's Algorithm	301
Prim's Algorithm	302
Network Flow Problems	307
Saturated and Unsaturated edges	310
Multiple choice questions	311
	316

11

Chapter

ALGORITHMS

Deterministic and Non-deterministic algorithm	320
Divide and Conquer Algorithm	321
Advantages of divide and conquer algorithm	321
Series and parallel algorithm	321
Parallel algorithm	322
Serial vs. Parallel Processing	323
Heuristic and Approximate Algorithms	324
Approximate algorithms	324
<input checked="" type="checkbox"/> Multiple Choice questions	325
<input type="checkbox"/> Discussion Exercise	325
<input type="checkbox"/> Bibliography	326
<input type="checkbox"/> TU Questions	327
	328



Соударств сюбілії в таблі

Соударств сюбілії в таблі є більш поглиблений відповідь на питання про те, які засоби зберігання даних використовуються в комп'ютерах. Важливим є те, що сюбілії використовують групи змінних, які зазвичай називають об'єктами. Це означає, що змінні, які зберігають інформацію, є зв'язані між собою. Наприклад, якщо ми маємо змінну, яка зберігає ім'я користувача, то можемо зберігати із нею інформацію про його пароль, адресу та інші дані.

INTRODUCTION TO DATA STRUCTURE

також відомі як структури даних, є центральними поняттями в комп'ютерній науці та інженерії.

Need for Data Structures

могуть виникнути від потреб інформації про складні системи та процеси, які вимагають докладного аналізу та обробки великої кількості даних.

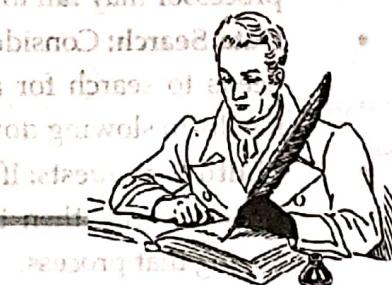
Важливий аспект цих систем є те, що вони повинні бути здатні обробляти великі обсяги даних та виконувати складні обчислювальні операції. Для цього використовуються спеціальні структури даних, які дозволяють ефективно зберігати та обробляти великі обсяги даних. Ці структури даних можуть бути реалізовані в різних формах, залежно від конкретної проблеми та вимог до системи.

Важливий аспект цих систем є те, що вони повинні бути здатні обробляти великі обсяги даних та виконувати складні обчислювальні операції. Для цього використовуються спеціальні структури даних, які дозволяють ефективно зберігати та обробляти великі обсяги даних. Ці структури даних можуть бути реалізовані в різних формах, залежно від конкретної проблеми та вимог до системи.

CHAPTER OUTLINE

After studying this chapter, the reader will be able to understand the

- Definition, Abstract data types, Importance of data structure.



INTRODUCTION

Data Structure can be defined as the group of data elements which provides an efficient way of storing and organizing data in the computer so that it can be used efficiently. Some examples of Data Structures are arrays, Linked List, Stack, Queue, etc. Data Structures are widely used in almost every aspect of Computer Science i.e. operating System, Compiler Design, Artificial intelligence, Graphics and many more.

Data Structures are the main part of many computer science algorithms as they enable the programmers to handle the data in an efficient way. It plays a vital role in enhancing the performance of software or a program as the main function of the software is to store and retrieve the user's data as fast as possible.

What is data structure?

The data structure is basically a technique of organizing and storing of different types of data items in computer memory. It is considered as not only the storing of data elements but also the maintaining of the logical relationship existing between individual data elements. It can also be defined as a mathematical or logical model, which relates to a particular organization of different data elements.

Thus, a data structure is the portion of memory allotted for a model, in which the required data can be arranged in a proper fashion. Data structure mainly specifies the following four things:

- Organization of data.
- Accessing methods
- Degree of associativity
- Processing alternatives for information

To develop a program of an algorithm, we should select an appropriate data structure for that algorithm. Therefore, algorithm and its associated data structures form a program.

$$\text{Algorithm} + \text{Data structure} = \text{Program}$$

Need of Data Structures

As applications are getting complex and amount of data is increasing day by day, there may arise the following problems:

- **Processor speed:** To handle very large amount of data, high speed processing is required; but as the data is growing day by day to the billions of files per entity, processor may fail to deal with that much amount of data.
- **Data Search:** Consider an inventory size of 500 items in a store; if our application needs to search for a particular item, it needs to traverse 500 items every time, results in slowing down the search process.
- **Multiple requests:** If thousands of users are searching the data simultaneously on a web server, then there are the chances that a very large server can be failed during that process.

In order to solve the above problems, data structures are used. Data is organized to form a data structure in such a way that all items are not required to be searched and required data can be searched instantly.

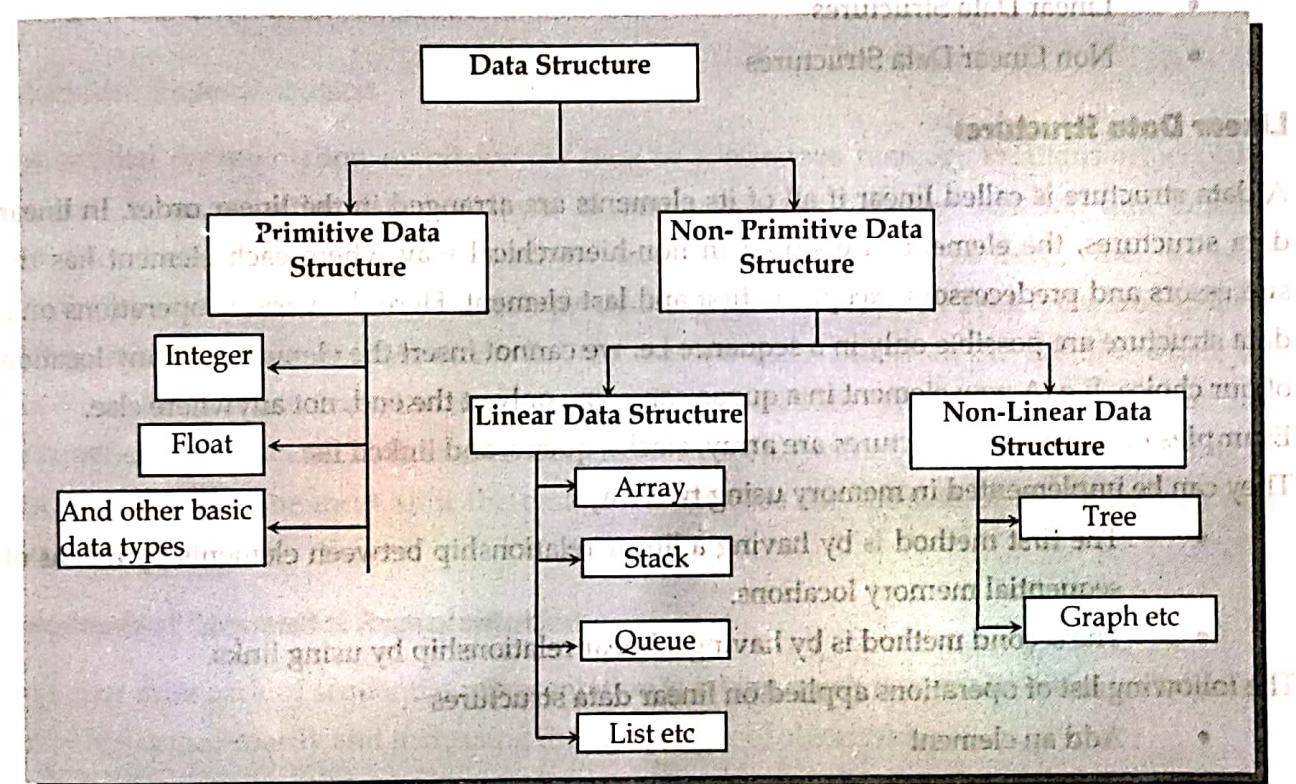
DATA STRUCTURE CLASSIFICATION

As we have discussed above, anything that can store data can be called as a data structure, hence Integer, Float, Boolean, Char etc, all are data structures. They are known as **Primitive Data Structures**.

We also have some complex Data Structures, which are used to store large and connected data. Some examples of Abstract Data Structure are:

- Linked List
- Tree
- Graph
- Stack, Queue etc.

All these data structures allow us to perform different operations on data. We select these data structures based on which type of operation is required. We will look into these data structures in more details in our later lessons. The classification of data structure mainly consists of:



Primitive Data Structure

The primitive data structures are known as basic data structures. These data structures are directly operated upon by the machine instructions. Normally, primitive data structures have different representation on different computers.

Example of primitive data structure:

- Integer
- Float
- Character
- Pointer

The data structures, typically those data structure that are directly operated upon by machine level instructions i.e. the fundamental data types such as int, float, double, etc in case of programming are known as primitive data structures.

Non-primitive Data Structure

The non-primitive data structures are highly developed complex data structures. Basically, these are developed from the primitive data structure. The non-primitive data structure is responsible for organizing the group of homogeneous and heterogeneous data elements.

Example of Non-primitive data structure:

- Arrays
- Lists
- Files

The data structures, which are not primitive, are called non-primitive data structures.

There are two types of non-primitive data structures:

- Linear Data Structures
- Non Linear Data Structures

Linear Data Structures

A data structure is called linear if all of its elements are arranged in the linear order. In linear data structures, the elements are stored in non-hierarchical way where each element has the successors and predecessors except the first and last element. Here the various operations on a data structure are possible only in a sequence i.e. we cannot insert the element into any location of our choice. E.g. A new element in a queue can come only at the end, not anywhere else.

Examples of linear data structures are array, stacks, queue, and linked list.

They can be implemented in memory using two ways:

- The first method is by having a linear relationship between elements by means of sequential memory locations.
- The second method is by having a linear relationship by using links.

The following list of operations applied on linear data structures

- Add an element
- Delete an element
- Traverse
- Sort the list of elements
- Search for a data element

Non Linear Data Structures

This data structure does not form a sequence i.e. each item or element is connected with two or more other items in a non-linear arrangement. The data elements are not arranged in sequential structure.

When the data elements are organized in some arbitrary function without any sequence, such data structures are called non-linear data structures. Examples of such type are trees, graphs. The relationship of adjacency is not maintained between elements of a non-linear data structure. The following list of operations applied on non-linear data structures.

- Add elements
- Delete elements
- Display the elements
- Sort the list of elements
- Search for a data element

Representation of Data Structures

Any data structure can be represented in two ways. They are:

- Sequential representation
- Linked representation

Sequential Representation

A sequential representation maintains the data in continuous memory locations which takes less time to retrieve the data but leads to time complexity during insertion and deletion operations. Because of sequential nature, the elements of the list must be freed, when we want to insert a new element or new data at a particular position of the list. To acquire free space in the list, one must shift the data of the list towards the right side from the position where the data has to be inserted. Thus, the time taken by CPU to shift the data will be much higher than the insertion operation and will lead to complexity in the algorithm. Similarly, while deleting an item from the list, one must shift the data items towards the left side of the list, which may waste CPU time.

Drawback of Sequential Representation

The major drawback of sequential representation is taking much time for insertion and deletion operations unnecessarily and increasing the complexity of algorithm.

Linked Representation

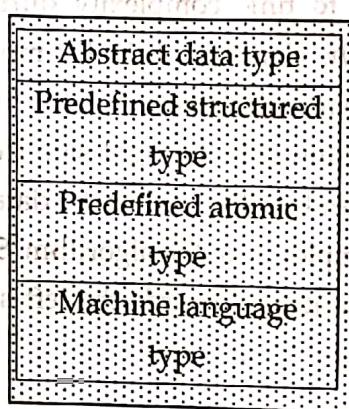
Linked representation maintains the list by means of a link between the adjacent elements which need not be stored in continuous memory locations. During insertion and deletion operations, links will be created or removed between which takes less time when compared to the corresponding operations of sequential representation. Because of the advantages mentioned above, generally, linked representation is preferred for any data structure.

The advantages and disadvantages of the various data structures described in this book are tabulated below:

Data Structure	Advantages	Disadvantages
Array	Quick insertion, very fast access if index known.	Slow search, slow deletion, and fixed size.
Ordered array	Quicker search than unsorted array.	Slow insertion and deletion, fixed size.
Stack	Provides last-in, first-out access.	Slow access to other items.
Queue	Provides first-in, first-out access.	Slow access to other items.
Linked list	Quick insertion, quick deletion.	Slow search.
Binary tree	Quick search, insertion, deletion (if tree remains balanced)	Deletion algorithm is complex.
Hash table	Very fast access if key known. Fast insertion.	Slow deletion, access slow if key not known, inefficient memory usage.
Heap	Fast insertion, deletion.	Slow access to other items.
Graph	Models real-world situations.	Some algorithms are slow and complex.

ABSTRACT DATA TYPE (ADT)

An abstract data type (ADT) consists of a data type together with a set of operations, which define how the type may be manipulated. This specification is stated in an implementation independent manner.



Abstract data types exist conceptually and concentrate on the mathematical properties of the data type ignoring implementation constraints and details.

An abstract data type (ADT) is a specification of only the behavior of instances of that type. Such a specification may be all that is needed to design a module that uses the type. Primitive types are like ADTs. We know what the `int` type can do (add, subtract, multiply, etc.). But we need not know how it does these operations. And we need not even know how an `int` is actually stored. As clients, we can use the `int` operations without having to know how they are implemented. In fact, if we had to think about how they are implemented, it would probably be a distraction from designing the software that will use them.

The ADT uses generic terms for types: Integer instead of `int`, and Real instead of `double`. That is because it is supposed to be independent of any specific programming language. In general, a complete ADT would also include documentation that explains exactly how each operation should behave.

The advantages offered by abstract data types include:

- Modularity
- Precise specifications
- Information hiding
- Simplicity
- Integrity
- Implementation independence

Advantages of Data Structures

- **Efficiency:** Efficiency of a program depends upon the choice of data structures. For example: suppose, we have some data and we need to perform the search for a particular record. In that case, if we organize our data in an array, we will have to search sequentially element by element. Hence, using array may not be very efficient here. There are better data structures which can make the search process efficient like ordered array, binary search tree or hash tables.
- **Reusability:** Data structures are reusable, i.e. once we have implemented a particular data structure, we can use it at any other place. Implementation of data structures can be compiled into libraries which can be used by different clients.
- **Abstraction:** Data structure is specified by the ADT which provides a level of abstraction. The client program uses the data structure through interface only, without getting into the implementation details.

Algorithm

An algorithm is a finite sequence of instructions for solving a stated problem. A stated problem is a well-defined task that has to be performed and must be solvable by an algorithm. The algorithm must eventually terminate, producing the required result.

Characteristics of an Algorithm

- **Input:** An algorithm should have zero or more inputs
- **Output:** An algorithm should have one or more outputs
- **Finiteness:** Every step in an algorithm should end in finite amount of time
- **Unambiguous:** Each step in an algorithm should clearly stated
- **Effectiveness:** Each step in an algorithm should be effective

To judge an algorithm the most important factors is to have a direct relationship to the performance of the algorithm. These have to do with their computing time & storage requirements (referred as Time complexity & Space complexity).

Example 1: Algorithm to test whether given entered number is odd or even.

Step 1: Start

Step 2: Read a .

Step 3: Find modules of a by 2 ($r = a \% 2$)

Step 4: If $r = 0$ then Print a is even else Print a is odd

Print a is even

Else

Print a is odd

Step 5: stop

Example 2: Algorithm to find roots of quadratic equation.

```

Step 1: Start
Step 2: Input a, b, c
Step 3: calculate discriminant  $d = b^2 - 4 \times a \times c$ 
Step 4: if  $d=0$  then
        Print "Roots are equal"
         $\text{root}_1 = \text{root}_2 = \frac{-b}{2a}$ 
Step 5: else if  $d>0$  then
         $\text{root}_1 = \frac{-b + \sqrt{b^2 - 4ac}}{2a}$ 
         $\text{root}_2 = \frac{-b - \sqrt{b^2 - 4ac}}{2a}$ 
Step 6: Else Print "Roots are imaginary"
Step 7: Print Root1 and Root2
Step 8: Stop
    
```

Example 3: Algorithm to find largest number among given three numbers.

```

Step 1: Start
Step 2: Declare variables a, b, c
Step 3: Read values of a, b, c
Step 4: if  $a > b$  and  $a > c$  then
        Print a is the greatest number
    else
        if  $b > c$ 
            Print b is the greatest number
        else
            Print c is the greatest number
Step 5: Stop
    
```

ALGORITHM ANALYSIS

An algorithm is a clearly specified set of instructions that a computer will follow to solve the problem. Once an algorithm is given for a problem and determined to be correct, the next step is to determine the amount of resources, such as time and space that the algorithm will require. This step is called algorithm analysis. An algorithm that requires several hundred gigabytes of main memory is not useful for most current machines, even if it is completely correct. The amount of time that any algorithm takes to run almost always depends on the amount of input that it must process. We expect, for instance, that sorting 10,000 elements requires more time than sorting 10 elements. The running time of an algorithm is thus a function of the input size. The exact value of the function depends on many factors, such as the speed of the host machine, the quality of the compiler, and in some cases, the quality of the program.

Computational and Asymptotic Complexity

Whenever we want to perform analysis of an algorithm, we need to calculate the complexity of that algorithm. But when we calculate complexity of an algorithm it does not provide exact amount of resource required. So instead of taking exact amount of resource we represent that complexity in a general form which produces the basic nature of that algorithm. We use that general form for analysis process.

Complexity analysis of an algorithm is very hard if we try to analyze exact. We know that the complexity (worst, best, or average) of an algorithm is the mathematical function of the size of the input. So if we analyze the algorithm in terms of bound (upper and lower) then it would be easier. For this purpose, we need the concept of asymptotic notations. The figure below gives upper and lower bound concept.

Why asymptotic notations important?

- They give a simple characterization of an algorithm's efficiency.
- They allow the comparison of the performance of various algorithms.

Big Oh (O) notation

When we have only asymptotic upper bound then we use O notation. If f and g are any two functions from set of integers to set of integers then function $f(x)$ is said to be big oh of $g(x)$ i.e. $f(x)=O(g(x))$ if and only if there exists two positive constants c and x_0 such that for all $x \geq x_0$, $f(x) \leq c \cdot g(x)$

The above relation says that $g(x)$ is an upper bound of $f(x)$

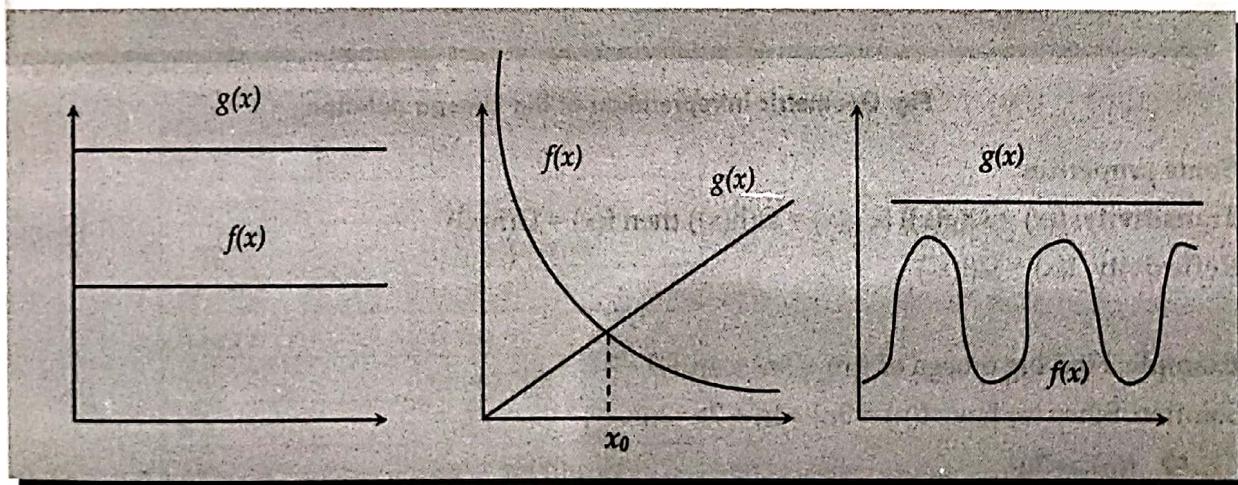


Fig: Geometric interpretation of Big-Oh notation

Some properties

Transitivity: $f(x) = O(g(x))$ & $g(x) = O(h(x))$ then $f(x) = O(h(x))$

Reflexivity: $f(x) = O(f(x))$

Transpose symmetry: $f(x) = O(g(x))$ if and only if $g(x) = \Omega(f(x))$

$O(1)$ is used to denote constants.

For all values of $n \geq n_0$, plot shows clearly that $f(n)$ lies below or on the curve of $c \cdot g(n)$

Example: Find big oh of given function $f(n) = 3n^2 + 4n + 7$

Solution:

$$\text{we have } f(n) = 3n^2 + 4n + 7 \leq 3n^2 + 4n^2 + 7n^2 \leq 14n^2$$

$$f(n) \leq 14n^2$$

$$\text{where, } c=14 \text{ and } g(n) = n^2, \text{ thus } f(n) = O(g(n)) = O(n^2)$$

Big Omega (Ω) notation

Big omega notation gives asymptotic lower bound. If f and g are any two functions from integers to set of integers, then function $f(x)$ is said to be big omega of $g(x)$ i.e. $f(x) = \Omega(g(x))$ if and only if there exists two positive constants c and x_0 such that

$$\text{For all } x \geq x_0, f(x) \geq c \cdot g(x)$$

The above relation says that $g(x)$ is a lower bound of $f(x)$.

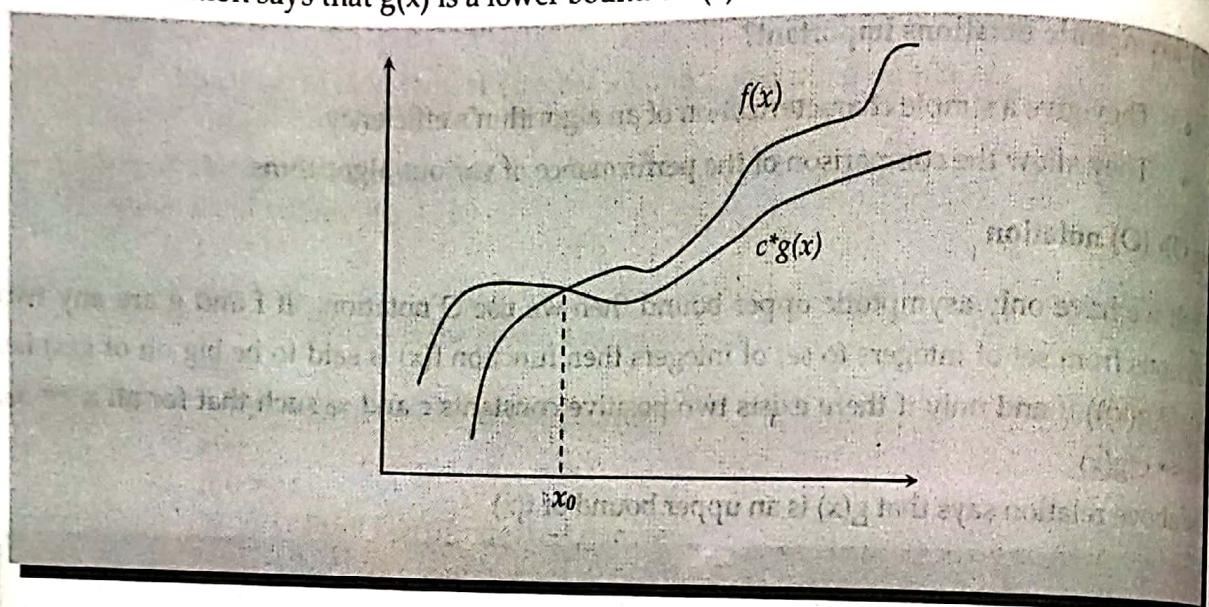


Fig: Geometric interpretation of Big Omega notation

Some properties

Transitivity: $f(x) = \Omega(g(x))$ & $g(x) = \Omega(h(x))$ then $f(x) = \Omega(h(x))$

Reflexivity: $f(x) = \Omega(f(x))$

Example: Find big omega of $f(n) = 3n^2 + 4n + 7$

Solution: Since we have $f(n) = 3n^2 + 4n + 7 \geq 3n^2$

$$\Rightarrow f(n) \geq 3n^2$$

$$\text{where, } c=3 \text{ and } g(n) = n^2, \text{ thus } f(n) = \Omega(g(n)) = \Omega(n^2)$$

Big Theta (Θ) notation

When we need asymptotically tight bound then we use this notation. If f and g are any two functions from set of integers to set of integers then function $f(x)$ is said to be big theta of $g(x)$ i.e. $f(x) = \Theta(g(x))$ if and only if there exists three positive constants c_1, c_2 and x_0 such that for all $x \geq x_0$, $c_1 \cdot g(x) \leq f(x) \leq c_2 \cdot g(x)$

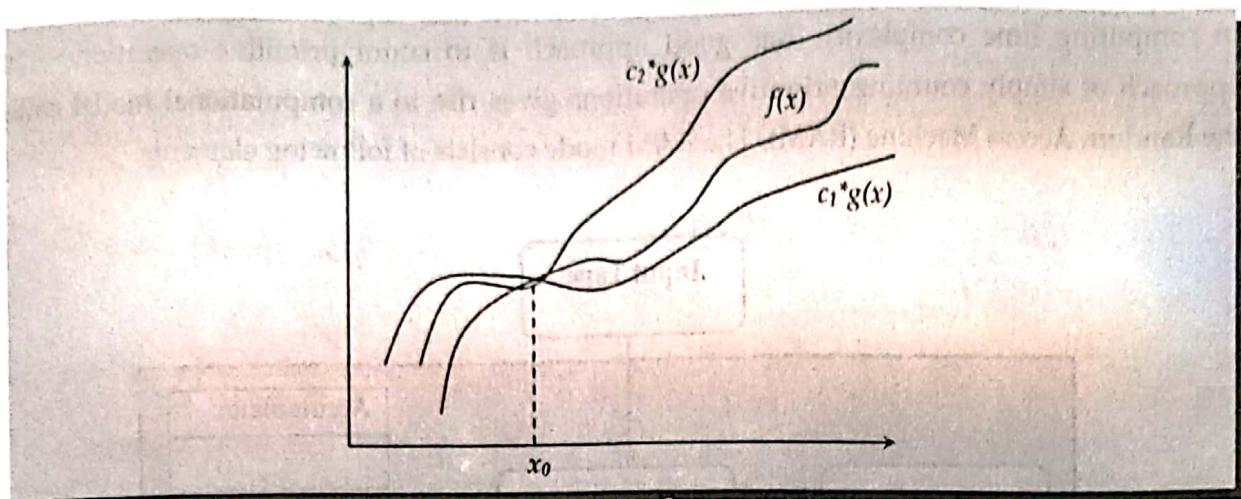


Fig: Geometric interpretation of Big Theta notation

Some properties

Transitivity: $f(x) = \Theta(g(x))$ & $g(x) = \Theta(h(x))$ then $f(x) = \Theta(h(x))$

Reflexivity: $f(x) = \Theta(f(x))$

Symmetry: $f(x) = \Theta(g(x))$ if and only if $g(x) = \Theta(f(x))$

Example: If $f(n) = 3n^2 + 4n + 7$ $g(n) = n^2$, then prove that $f(n) = \Theta(g(n))$.

Proof: let us choose c_1, c_2 and n_0 values as 14, 1 and 1 respectively then we can have,

$f(n) \leq c_1 * g(n)$, $n \geq n_0$ as $3n^2 + 4n + 7 \leq 14 * n^2$, and

$f(n) \geq c_2 * g(n)$, $n \geq n_0$ as $3n^2 + 4n + 7 \geq 1 * n^2$

For all $n \geq 1$ (in both cases).

So $c_2 * g(n) \leq f(n) \leq c_1 * g(n)$ is trivial.

Hence $f(n) = \Theta(g(n))$.

Functions in order of increasing growth rate

Function	Name
C	Constant
$\log n$	Logarithmic
$\log_2 n$	Log-squared
N	Linear
$n \log n$	$n \log n$
n^2	Quadratic
n^3	Cubic
2^n	Exponential

RANDOM ACCESS MACHINE MODEL (RAM)

This RAM model is the base model for our study of design and analysis of algorithms to have design and analysis in machine independent scenario. In this model each basic operation (+, -) takes one step, loops and subroutines are not basic operations. Each memory reference takes one step. We measure run time of algorithm by counting the steps.

In computing time complexity, one good approach is to count primitive operations. The approach of simply counting primitive operations gives rise to a computational model called the Random Access Machine (RAM). The RAM mode consists of following elements.

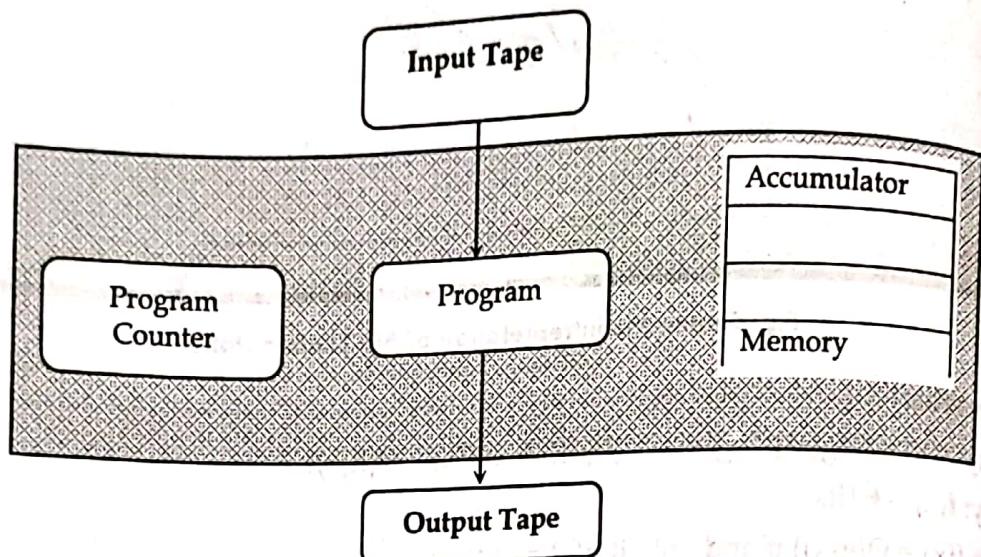


Fig: RAM Model

Input tape/output tape: input tape consists of a sequence of squares, each of which can store integer. Whenever one square is read from the tape head moves one square to the right. The output tape is also a sequence of squares, in each square an integer can be written. Output is written on the square under the tape head and after the writing, the tape head moves one square to the right over writing on the same square is not permitted.

Memory: The memory consists of a sequence of registers, each of which is capable of holding an integer.

Program: Program for RAM contains a sequence of labeled instructions resembling those found in assembly language programs. All computations take place in the first register called accumulator. A RAM program defines a mapping from input tape to the output tape.

Program counter: The program counter determines the next instruction to be executed.

Some examples of primitive operations are:

- Assigning a value to a variable
- Performing an arithmetic operation
- Indexing into an array
- Returning from method
- Calling a method
- Comparing two numbers etc

Detailed analysis

We can analyze iterative algorithms by counting steps by using RAM model as below.

Example 1: Algorithm for sequential search

Sequential_Search(A, n, key)

```

int i, flag=0;
for(i=0; i<n; i++)
{
    if(A[i]==key)
    {
        flag=1;
    }
}
if(flag==1)
    Print "Search successful"
else
    Print "Search un-successful"

```

Space complexity

By counting number of memory references gives the space complexity of given algorithm.

The variable 'i' takes 1 unit space

The variable 'n' takes 1 unit space

The variable 'key' takes 1 unit space

The variable 'flag' takes 1 unit space

The variable 'A' takes n unit space

Now total space complexity is;

$$T(n)=1+1+1+1+n$$

$$= 4 + n$$

$$= O(n)$$

Time complexity

By counting number of statements takes the total time complexity of given algorithm.

Declaration statement takes 1 step time

In for loop;

i=0 takes 1 step time

i<n takes (n+1) step time

i++ takes n step time

Within for loop if condition takes n step time

Within if statement flag=0 takes at most n step time

If statement outside the for loop takes 1 step time

Print statement takes 1 step time

Now total time complexity is given by;

$$T(n)=1+1+(n+1)+n+n+n+1+1$$

$$=5+4n \leq 5n+4n$$

$$=9n$$

$$\Rightarrow T(n)=O(n)$$

In brief, meanings of the various growth functions

Mathematical Expression	Relative Rates of Growth	Name
$T(n) = O(F(n))$	Growth of $T(n)$ is \leq growth of $F(n)$	Big oh
$T(n) = \Omega(F(n))$	Growth of $T(n)$ is \geq growth of $F(n)$	Big omega
$T(n) = \Theta(F(n))$	Growth of $T(n)$ is $=$ growth of $F(n)$	Big theta
$T(n) = o(F(n))$	Growth of $T(n)$ is $<$ growth of $F(n)$	Little oh

THE BEST, AVERAGE, AND WORST CASES

The least possible execution time taken by an algorithm for a particular input is known as best case. Best case complexity gives lower bound on the running time of the algorithm for any instance of input. This indicates that the algorithm can never have lower running time than best case for particular class of problems.

Worst case complexity: The maximum possible execution time taken by an algorithm for a particular input is known as worst case. It gives upper bound on the running time of the algorithm for all the instances of the input. This insures that no input can overcome the running time limit posed by worst case complexity.

Average case complexity: It gives average number of steps required on any instance of the input.

Example: - let's take an algorithm for Quick sort

QuickSort(A,l,r)

```
{
    if(l < r)
    {

```

```
        p = Partition(A, l, r);
        QuickSort(A, l, p-1);
        QuickSort(A, p+1, r);
    }
}
```

Partition(A, l, r)

```
{
    x = l;
    y = r;
    p = A[l];
    while(x < y)
    {

```

```
        while(A[x] <= p)
        {

```

```
            x++;
        }
    }

```

```
    while(A[y] >= p)
    {

```

```
        y--;
    }

```

```
    if(x < y)

```

```
        swap(A[x], A[y]);
    }
}
```

A[l] = A[y];

A[y] = p;

Return y; //return position of pivot

}

Best Case Time Complexity

It gives best case when whole problem is divided into two partitions of equal size, therefore, $T(n) = 2T(n/2) + O(n)$, Solving this recurrence we get,

$$\Rightarrow \text{Time Complexity} = O(n \log n)$$

Worst Case Time Complexity

When array is already sorted or sorted in reverse order, one partition contains $n-1$ items and another contains zero items, therefore its recurrence relation is, $T(n) = T(n-1) + O(1)$, Solving this recurrence we get

$$\Rightarrow \text{Time Complexity} = O(n^2)$$

Average Case Time Complexity

All permutations of the input numbers are equally likely. On a random input array, we will have a mix of well balanced and unbalanced splits. Good and bad splits are randomly distributed across throughout the tree suppose we are alternate: Balanced, Unbalanced, Balanced.

$$B(n) = 2UB(n/2) + \Theta(n) \text{ Balanced}$$

$$UB(n) = B(n-1) + \Theta(n) \text{ Unbalanced}$$

$$\begin{aligned} \text{Solving: } B(n) &= 2(B(n/2 - 1) + \Theta(n/2)) + \Theta(n) \\ &= 2B(n/2 - 1) + \Theta(n) \\ &= \Theta(n \log n) \end{aligned}$$

MULTIPLE CHOICE QUESTIONS

1. Which of the following is/are the levels of implementation of data structure?
 - (a) Abstract level
 - (b) Application level
 - (c) Implementation level
 - (d) All of the above
2. Which of the following is not the part of ADT description?
 - (a) Data
 - (b) Operations
 - (c) Both of the above
 - (d) None of the above
3. Which of the following data structure can't store the non-homogeneous data elements?
 - (a) Arrays
 - (b) Records
 - (c) Pointers
 - (d) Stacks
4. Which of the following is non-liner data structure?
 - (a) Stacks
 - (b) List
 - (c) Strings
 - (d) Trees
5. Which of the following data structure is non linear type?
 - (a) Strings
 - (b) Lists
 - (c) Stacks
 - (d) Graph
6. Which of the following data structure is linear type?
 - (a) Graph
 - (b) Trees
 - (c) Binary tree
 - (d) Stack
7. Operations on a data structure may be
 - (a) creation
 - (b) destruction
 - (c) selection
 - (d) all of the above
8. Which of the following are the operations applicable a primitive data structures?
 - (a) create
 - (b) destroy
 - (c) update
 - (d) all of the above
9. Arrays are best data structures
 - (a) for relatively permanent collections of data
 - (b) for the size of the structure and the data in the structure are constantly changing
 - (c) for both of above situation
 - (d) for none of above situation
10. Which of the following data structures are indexed structures?
 - (a) Linear arrays
 - (b) Linked lists
 - (c) Graphs
 - (d) Trees



DISCUSSION EXERCISE

1. What is main concept behind the asymptotic notations? Explain big oh in detail with suitable example.
2. What do you mean by ADT? Show that data type int as ADT.
3. What is main drawback of big oh notation?
4. What are the various operations that can be performed on different Data Structures?
5. What do you mean by linear data structure and non-linear data structure? Differentiate them with suitable example.
6. Solving a problem requires running an $O(N)$ algorithm and then afterwards a second $O(N)$ algorithm. What is the total cost of solving the problem?
7. Solving a problem requires running an $O(N^2)$ algorithm and then afterwards an $O(N)$ algorithm. What is the total cost of solving the problem?
8. How can you convert a satisfiability problem into a three-satisfiability problem for an instance when an alternative in a Boolean expression has two variables? One variable?
9. Is it true that
 - a. if $f(n)$ is $\Theta(g(n))$, then $2f(n)$ is $\Theta(2g(n))$?
 - b. $f(n) + g(n)$ is $\Theta(\min(f(n), g(n)))$?
 - c. $2na$ is $O(2n)$?
10. Determine the complexity of the following implementations of the algorithms for adding, multiplying, and transposing $n \times n$ matrices:


```

for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        a[i][j] = b[i][j] + c[i][j];
for (i = 0; i < n; i++)
    for (j = 0; j < n; j++)
        for (k = 0; k < n; k++)
            a[i][j] += b[i][k] * c[k][j];
for (i = 0; i < n - 1; i++)
    for (j = i+1; j < n; j++)
    {
        tmp = a[i][j]; a[i][j] = a[j][i];
        a[j][i] = tmp;
    }
      
```
11. Group the following into equivalent Big-Oh functions:
 - a. $x^2, x, x^2 + x, x^2 - x$, and $(x^3 / (x - 1))$.
12. What is RAM model? Describe their functionalities.
13. How can you determine space and time complexity of the algorithm? Describe with suitable example.
14. Define worst, best and average case complexity of the algorithm. Explain with suitable example.
15. What is algorithm? Write down characteristics of algorithm.
16. Find big oh, big omega and big theta of following function,
 - a. $F(x) = 3x^4 + 9x^2 + 8x + 6$
17. List some well-known problems that are NP-complete when expressed as decision problems.
18. Why we need asymptotic notation? Explain.
19. Show that array as an ADT with suitable example.
20. In what areas do data structures are applied?

2

THE STACKS AND ITS APPLICATIONS



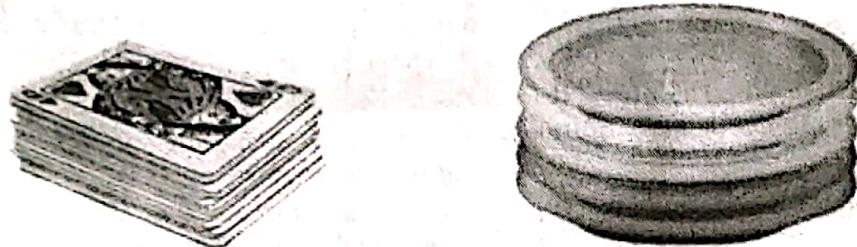
CHAPTER OUTLINE

After studying this chapter, the reader will be able to understand the

- Introduction, Stack as an ADT, POP and PUSH operation, Stack applications; evaluation of infix, postfix and prefix Expressions, conversion of expression.

WHAT IS STACK?

A stack is a linear data structure in which an element may be inserted or deleted only at one end called the top end of the stack i.e. the elements are removed from a stack in the reverse order of that in which they were inserted into the stack. Stack uses a variable called top which points to the topmost element in the stack. top is incremented while pushing (inserting) an element in to the stack and decremented while popping (deleting) an element from the stack. It is named stack as it behaves like a real-world stack, for example - a deck of cards or a pile of plates, etc.



A stack follows the principle of last-in-first-out (LIFO) system. According to the stack terminology, PUSH and POP are two terms used for insert and delete operations.

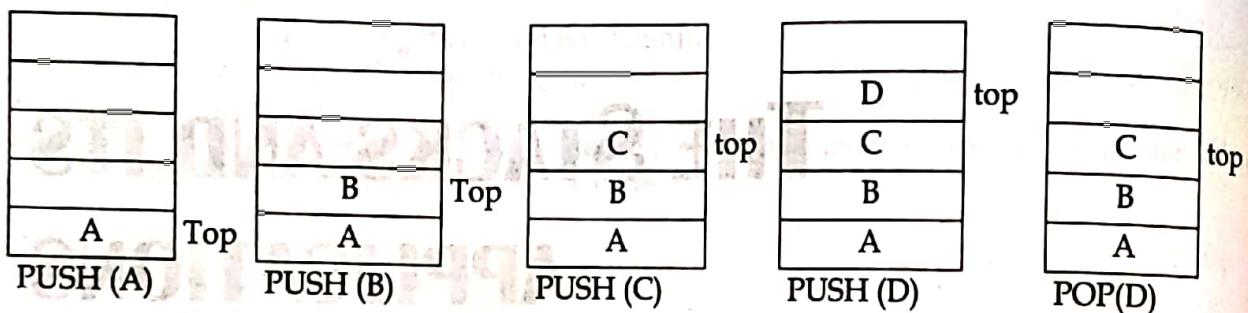


Fig: push, pop operations in Stack

Basic features of Stack

- Stack is an ordered list of similar data type.
- Stack is a LIFO (Last in First out) structure or we can say FILO (First in Last out).
- push() function is used to insert new elements into the Stack and pop() function is used to remove an element from the stack. Both insertion and removal are allowed at only one end of Stack called Top.
- Stack is said to be in Overflow state when it is completely full and is said to be in Underflow state if it is completely empty.

Applications of Stack

Stack is used directly and indirectly in the following fields:

- To evaluate the expressions (postfix, prefix)
- To keep the page-visited history in a Web browser
- To perform the undo sequence in a text editor
- Used in recursion
- To check the correctness of parentheses sequence

DATA STRUCTURE

- To pass the parameters between the functions in a C program
- Can be used as an auxiliary data structure for implementing algorithms
- Can be used as a component of other data structures
- Tree Traversals

The Stack ADT

A stack of elements of type T is a finite sequence of elements of T together with the operations:

- Create Empty Stack (S):** Create or make stack S be an empty stack.
- Push(S, x):** Insert x at one end of the stack, called its **top**.
- Top(S):** If stack S is not empty; then retrieve the element at its **top**.
- Pop(S):** If stack S is not empty; then delete the element at its **top**.
- IsFull(S):** Determine whether stack S is full or not. Return **true** if S is full; return **false** otherwise.
- IsEmpty(S):** Determine whether stack S is empty or not. Return **true** if S is an empty stack; return **false** otherwise.

Implementation of Stack

Stack can be implemented in following two ways:

1. Array Implementation of stack (or static implementation).
2. Linked list implementation of stack (or dynamic).

Array (static) implementation of a stack

It is one of two ways to implement a stack that uses a one dimensional array to store the data. In this implementation top is an integer value (an index of an array) that indicates off the top position of a stack. Each time data is added or removed, top is incremented or decremented accordingly, to keep track of current top of the stack. By convention, in C implementation the empty stack is indicated by setting the value of top to -1 (top=-1).

Structure for stack

```
#define MAXSIZE 10
```

```
struct stack
```

```
{
    int items[MAXSIZE]; //Declaring an array to store items
    int top; //Top of a stack
};
```

```
typedef struct stack st;
```

Creating Empty stack

The value of top=-1 indicates the empty stack in C implementation.

```
void create_empty_stack(struct stack e) /*Function to create an empty stack*/
{
    e.top=-1;
}
```

Stack Empty or Underflow

This is the situation when the stack contains no element. At this point the top of stack is present at the bottom of the stack. In array implementation of stack, conventionally top=-1 indicates empty.

The following function return 1 if the stack is empty, 0 otherwise.

```
int IsEmpty()
```

```
{
```

```
    if (top == -1)
```

```
        return 1;
```

```
    else
```

```
        return 0;
```

```
}
```

Stack Full or Overflow

This is the situation when the stack becomes full, and no more elements can be pushed onto stack. At this point the stack top is present at the highest location (MAXSIZE-1) of the stack.

The following function returns true (1) if stack is full, false (0) otherwise.

```
int IsFull()
```

```
{
```

```
    if (top == MAXSIZE-1)
```

```
        return 1;
```

```
    else
```

```
        return 0;
```

```
}
```

Adding an element onto the stack (push operation)

Adding an element into the top of the stack is referred to as push operation. Push operation involves following two steps.

- Increment the variable Top so that it can now refer to the next memory location.
- Add element at the position of incremented top. This is referred to as adding new element at the top of the stack.

Stack is over flown when we try to insert an element into a completely filled stack therefore, our main function must always avoid stack overflow condition.

Algorithm for Push Operation of Stack

Let Stack [MAXSIZE] is an array to implement the stack. The variable top denotes the top of the stack. This algorithm adds or inserts an item at the top of the stack

1. Start

2. Check for stack overflow as

if top==MAXSIZE-1 then

 print "Stack Overflow" and Exit the program

else

 Increase top by 1 as

 Set, top=top+1

3. Read elements to be inserted say element
4. Set, Stack[top] = element // Inserts item in new top position
5. Stop

Implementation of push algorithm in C language

void push (int val) // n is size of the stack

```

{
    if (top == MAXSIZE)
        printf("\n Stack Overflow");
    else
        {
            top = top +1;
            stack[top] = val;
        }
}

```

Deletion of an element from a stack (Pop operation)

Deletion of an element from the top of the stack is called pop operation. The value of the variable top will be decremented by 1 whenever an item is deleted from the stack. The top most element of the stack is stored in another variable and then the top is decremented by 1. The operation returns the deleted value that was stored in another variable as the result. The underflow condition occurs when we try to delete an element from an already empty stack.

Algorithm for POP operation

This algorithm deletes the top element of the stack and assigns it to a variable element

1. Start
2. Check for the stack Underflow as
If top<0 then
Print "Stack Underflow" and Exit the program
else
Remove the top element and set this element to the variable as
Set, element=Stack [top]
Decrement top by 1 as
Set, top=top-1
3. Print 'element' as a deleted item from the stack
4. Stop

The C function for POP operation

Alternatively, we can define the POP function as give below:

```

void pop()
{
    int item;
    if (top < 0) //Checking Stack Underflow
        printf("The stack is Empty");
}

```

```

else
{
    item = stack[top]; //Storing top element to item variable
    top = top-1;           //Decrease top by 1
    printf("The popped item is=%d", item); //Displaying the deleted items
}
}

```

Visiting each element of the stack (Peek or Display operation)

Display operation involves returning the element which is present at the top of the stack without deleting it. Underflow condition can occur if we try to return the top element in already empty stack.

Display algorithm

- Start
- Check for the stack Underflow as

If $\text{top} < 0$ then

Print "Stack Underflow" and Exit the program

Else

Set, $i = \text{top}$

While ($i >= 0$)

Display $\text{stack}[i]$

Decrement i by 1 as

Set, $i = i - 1$

End while

- Stop

Complete menu driven program in C to array implementation of stack

```

#include<stdio.h>
#include<conio.h>
int stack[100], i, j, choice=0, n, top=-1;
void push();
void pop();
void Display();
void main ()
{
    printf("Enter the number of elements in the stack ");
    scanf("%d", &n);
    printf("*****Stack operations using array*****");
    printf("\n-----\n");
    while(choice != 4)
    {
        printf("Chose one from the below options...\n");
        printf("\n1.Push\n2.Pop\n3.Show\n4.Exit");
        printf("\n Enter your choice \n");
        scanf("%d", &choice);
    }
}

```

```

switch(choice)
{
    case 1:
        push();
        break;
    case 2:
        pop();
        break;
    case 3:
        Display();
        break;
    case 4:
        printf("Exiting....");
        break;
    default:
        printf("Please Enter valid choice ");
}
}

void push()
{
    int val;
    if (top == n )
        printf("\n Overflow");
    else
    {
        printf("Enter the value?");
        scanf("%d",&val);
        top = top +1;
        stack[top] = val;
    }
}

void pop ()
{
    if(top == -1)
        printf("Underflow");
    else
        top = top -1;
}

```

```

void Display()
{
    for (i=top; i>=0; i--)
    {
        printf("%d\n", stack[i]);
    }
    if (top == -1)
    {
        printf("Stack is empty");
    }
}

```

Analysis of Stack Operations

Below mentioned are the time complexities for various operations that can be performed on the Stack data structure.

- Push Operation : O(1)
- Pop Operation : O(1)
- Top Operation : O(1)
- Search Operation : O(n)

The time complexities for push() and pop() functions are O(1) because we always have to insert or remove the data from the top of the stack, which is a one step process.

Stack Applications

There are a lot of computer applications of stack as described above but here we need to study following applications:

- Conversion of infix expression to postfix expression
- Conversion of infix expression to prefix expression
- Conversion of prefix expression to infix expression
- Conversion of postfix expression to infix expression
- Evaluation of postfix expression
- Evaluation of infix expression
- Evaluation of prefix expression

Infix, Prefix and Postfix Notation

One of the applications of the stack is to evaluate the expression. We can represent the expression following three types of notation:

- Infix expression
- Prefix expression
- Postfix expression

Infix expression

It is an ordinary mathematical notation of expression where operator is written in between the operands. Example: A+B. Here + is an operator and 'A' and 'B' are called operands.

Prefix notation

In prefix notation the operator precedes the two operands. That is the operator is written before the operands. It is also called polish notation. For example the equivalent prefix expressions of given infix expression A+B is +AB.

Postfix notation

In postfix notation the operators are written after the operands so it is called the postfix notation (post mean after). In this notation the operator follows the two operands. For example the equivalent postfix expressions of given infix expression $A+B$ is $AB+$.

Note: Both prefix and postfix is parenthesis free expressions. For example

Infix Expression	Prefix Expression	Postfix Expression
$A + B * C + D$	$+ + A * B C D$	$A B C * + D +$
$(A + B) * (C + D)$	$* + A B + C D$	$A B + C D + *$
$A * B + C * D$	$+ * A B * C D$	$A B * C D * +$
$A + B + C + D$	$++ + A B C D$	$A B + C + D +$

Precedence rule

While converting infix to postfix or prefix expression we have to consider the precedence rule and the precedence rules are as follows. To study further, we must also know the operator precedence and their associativity:

Token	Operator	Precedence	Associativity
$()$	function call	17	left-to-right
$[]$	array element		
$\cdot .$	struct or union member		
$- ++$	increment, decrement	16	left-to-right
$!$	logical NOT	15	right-to-left
\sim	one's complement		
$- +$	unary minus or plus		
$\& *$	address or indirection		
sizeof	size (in bytes)		
(type)	type cast	14	right-to-left
$* / \%$	Multiplicative	13	left-to-right
$+ -$	binary add or subtract	12	left-to-right
$<<>>$	Shift	11	left-to-right
$> >=$	Relational	10	left-to-right
$< <=$			
$== !=$	Equality	9	left-to-right
$\&$	bitwise AND	8	left-to-right
$^$	bitwise XOR	7	left-to-right
$ $	bitwise OR	6	left-to-right
$\&\&$	logical AND	5	left-to-right
$\ \ $	logical OR	4	left-to-right
$? :$	Conditional	3	right-to-left
$= += /=$	Assignment	2	right-to-left
$* *= \%=$			
$\&= ^=$			
,	Comma	1	left-to-right

Exponentiation (the expression A^B is A raised to the B power, so that $3^2=9$). When parenthesized operators of the same precedence are scanned, the order is assumed to be left-right except in the case of exponentiation, where the order is assumed to be from right to left.

- $A+B+C$ means $(A+B)+C$
- A^B^C means A^B^C

Consider an example that illustrate the converting of infix to postfix expression, $A + (B * C)$. Use the following rule to convert it in postfix:

- Parenthesis for emphasis
- Convert the multiplication
- Convert the addition
- Postfix form

Associativity

Associativity describes the rule where operators with the same precedence appear in a expression. For example, in expression $a + b - c$, both $+$ and $-$ have the same precedence, the which part of the expression will be evaluated first, is determined by associativity of those operators. Here, both $+$ and $-$ are left associative, so the expression will be evaluated as $(a + b) - c$.

Precedence and associativity determines the order of evaluation of an expression. Following is an operator precedence and associativity table (highest to lowest) –

Operator	Precedence	Associativity
Exponentiation $^$	Highest	Right Associative
Multiplication ($*$) & Division ($/$)	Second Highest	Left Associative
Addition ($+$) & Subtraction ($-$)	Lowest	Left Associative

The above table shows the default behavior of operators. At any point of time in expression evaluation, the order can be altered by using parenthesis. For example: In $a + b * c$, the expression part $b*c$ will be evaluated first, with multiplication as precedence over addition. We here use parenthesis for $a + b$ to be evaluated first, like $(a + b)*c$.

Illustration

- A + (B * C) Infix form
- A + (B * C) Parenthesis for emphasis
- A + (BC*) Convert the multiplication
- A (BC*) + Convert the addition
- ABC*+ Post-fix form

Consider an example

- (A + B) * ((C - D) + E) / F Infix form
 (AB+) * ((C - D) + E) / F
 (AB+) * ((CD-) + E) / F
 (AB+) * (CD-E+) / F
 (AB+CD-E+*) / F
 AB+CD-E-*F/ Postfix form

Algorithm to convert infix to postfix notation

Here we use a single stack.

1. Start
2. Scan the Infix string from left to right.
3. Initialize an empty stack.
4. If the scanned character is an operand, add it to the Postfix string.
5. If the scanned character is an operator and if the stack is empty push the character to stack.
6. If the scanned character is an Operator and the stack is not empty, compare the precedence of the character with the element on top of the stack.
7. If top Stack symbol has higher precedence over the scanned character then pop the top element of stack else push the scanned character to stack.
8. Repeat step 7 until the stack is not empty and top Stack has precedence over the character.
9. Repeat 4 and 8 steps till all the characters are scanned.
10. After all characters are scanned, we have to add any character that the stack may have to the Postfix string.
11. If stack is not empty add top Stack to Postfix string and Pop the stack.
12. Repeat step 11 as long as stack is not empty.
13. Stop

Tracing

Convert following infix expression to their equivalent postfix expression

$A+(B*C-(D/E-F)*G)*H$

Stack	Input	Output
Empty	$A+(B*C-(D/E-F)*G)*H$	-
Empty	$+(B*C-(D/E-F)*G)*H$	A
+	$(B*C-(D/E-F)*G)*H$	A
+()	$B*C-(D/E-F)*G)*H$	A
+()	$*C-(D/E-F)*G)*H$	AB
+(*)	$C-(D/E-F)*G)*H$	AB
+(*)	$-(D/E-F)*G)*H$	ABC
+(-)	$(D/E-F)*G)*H$	ABC*
+(-)	$D/E-F)*G)*H$	ABC*
+(-)	$/E-F)*G)*H$	ABC*D
+(-/)	$E-F)*G)*H$	ABC*D
+(-/)	$-F)*G)*H$	ABC*DE
+(-/)	$F)*G)*H$	ABC*DE/
+(-/)	$F)*G)*H$	ABC*DE/
+(-/)	$)*G)*H$	ABC*DE/F
+(-)	$*G)*H$	ABC*DE/F
+(-*)	$G)*H$	ABC*DE/F-
+(-*)	$)*H$	ABC*DE/F-G
+	$*H$	ABC*DE/F-G*-
+	H	ABC*DE/F-G*-
+	End	ABC*DE/F-G*-H
Empty	End	ABC*DE/F-G*-H*

Also we can convert given infix to their equivalent postfix expression by using two separate stacks as below,

Algorithm to convert infix to postfix notation

Let here two stacks opstack and poststack are used and otos & ptos represents the opstack and poststack top respectively. The opstack is used to store operators of given expression, poststack is used to store converted corresponding postfix expression.

1. Start

2. Scan one character at a time of an infix expression from left to right

3. Opstack = the empty stack

4. Repeat till there is data in infix expression

 4.1 If scanned character is '(' then push it to opstack

 4.2 If scanned character is operand then push it to poststack

 4.3 If scanned character is operator then

 if (opstack != -1)

 While (precedence (opstack [otos]) > precedence (scan character)) then

 pop and push it into poststack

 Otherwise

 Push scanned character into opstack

 4.4 If scanned character is ')' then

 Pop and push into poststack until '(' is not found and ignore both symbols

5. Pop and push into poststack until opstack is not empty.

6. Stop

Trace of Conversion of infix to postfix Algorithm

The following tracing of the algorithm illustrates convert infix to postfix the algorithm Consider an infix expression

$((A-(B+C))*D)$(E+F)$

Scan character	Poststack	Opstack
(.....	(
(.....	((
A	A	((A
-	A	((A-
(A	((A-()
B	AB	((A-B
+	AB	((A-B)
C	ABC	((A-B)+
)	ABC+	((A-B)+
)	ABC+-	((A-B)+
*	ABC+-	((A-B)+*
D	ABC+-D	((A-B)+* D
)	ABC+-D*	((A-B)+* D)
\$	ABC+-D*
(ABC+-D*	\$
E	ABC+-D*E	\$(E
+	ABC+-D*E	\$(E
F	ABC+-D*EF	\$(E F
)	ABC+-D*EF+	\$(E F)
.....	ABC+-D*EF+\$ (postfix)

Tracing example 2: Trace the algorithm to convert given infix to postfix expression
 $(A+B^*C\$D)/((E+F-G)^*H)\I/J

Scan character	Poststack	Opstack
(.....	(
A	A	(
+	A	(+)
B	AB	(+)
*	AB	(+*)
C	ABC	(+*)
\$	ABC	(+* \$)
D	ABCD	(+* \$)
)	ABCD \$*+
/	ABCD\$*+	/
(ABCD\$*+	/(
(ABCD\$*+	/(()
E	ABCD\$*+ E	/(()
+	ABCD\$*+E	/(() +
F	ABCD\$*+EF	/(() +
-	ABCD\$*+EF+	/(() -
G	ABCD\$*+EF+G	/(() -
)	ABCD\$*+EF+G -	/(
*	ABCD\$*+EF+G -	/(*)
H	ABCD\$*+EF+G -H	/(*
)	ABCD\$*+EF+G -H*	/
\$	ABCD\$*+EF+G -H*	/\$
I	ABCD\$*+EF+G -H* I	/\$
/	ABCD\$*+EF+G -H* I\$/	/
J	ABCD\$*+EF+G -H* I\$/J	/
.....	ABCD\$*+EF+G -H* I\$/J/

Complete program in C to convert an expression from infix to postfix

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
#include<string.h>
int precedence(char);
void main()
{
    int i, otos=-1, ptos=-1, len, length;
    char infix[100], poststack[100], opstack[100];
```

```

printf("Enter a valid infix\n");
gets (infix);
length=strlen(infix);
len=length;
for(i=0;i<=length-1;i++)
{
    if(infix[i]=='(')
    {
        opstack[++otos]=infix[i];
        len++;
    }
    else if(isalpha(infix[i]))
    {
        poststack[++ptos]=infix[i];
    }
    else if (infix[i]==')')
    {
        len++;
        while(opstack[otos]!='(')
        {
            poststack[++ptos]=opstack[otos];
            otos--;
        }
        otos--;
    }
    else //operators
    {
        if(precedency(opstack[otos])>=precedency(infix[i]))
        {
            poststack[++ptos]=opstack[otos-];
            opstack[++otos]=infix[i];
        }
        opstack[++otos]=infix[i];
    }
}
while(otos!= -1)
{
    poststack[++ptos]=opstack[otos];
    otos--;
}
for(i=0; i<len; i++) /*for displaying*/
{
    printf("%c", poststack[i]);
}
getch();
}

int precedence(char ch) /*precedence function*/

```

```

switch(ch)
{
    case '$':
        return(4);
    case '*':
        return(3);
    case '/':
        return(2);
    case '+':
    case '-':
        return(1);
    default:
}

```

Output

Enter valid infix expression

(A+B*C/D)*(E-F/G)

ABCD/*+EFG/-*

Converting an Infix expression to prefix expression

The precedence rule for converting an expression from infix to prefix is identical to that of infix to postfix. Only changes from postfix conversion are that the operator is placed before the operands rather than after them. The prefix of A+B-C is -+ABC.

Example: Consider an infix expression:

$$\begin{aligned}
 & A \$ B * C - D + E / F / (G + H) \text{ infix form} \\
 & = A \$ B * C - D + E / F / (+GH) \\
 & = \$AB* C - D + E / F / (+GH) \\
 & = *\$ABC-D+E/F/ (+GH) \\
 & = *\$ABC-D+ / EF/ (+GH) \\
 & = *\$ABC-D+ / EF+GH \\
 & = (-\$ABCD) + (/EF+GH) \\
 & = +-*\$ABCD//EF+GH \text{ which is in prefix form.}
 \end{aligned}$$

Converting an Infix expression to prefix expression by using stack

Let here two stacks opstack and prestack are used and otos & ptos represents the opstack top and prestack top respectively. The opstack is used to store operators of given expression and prestack is used to store converted corresponding postfix expression.

1. Start
2. Scan one character at a time of an infix expression from right to left
3. Opstack=the empty stack

4. Repeat till there is data in infix expression
- 4.1 If scanned character is ')' then push it to opstack
 - 4.2 If scanned character is operand then push it to prestack
 - 4.3 If scanned character is operator then
 - if (opstack != -1) While (precedence (opstack [otos]) > precedence (scan character)) then pop and push it into prestack
 - Otherwise Push scanned character into opstack
- 4.4 If scanned character is '(' then Pop and push into prestack until ')' is not found and ignore both symbols
5. Pop and push into prestack until opstack is not empty.
6. Pop and display prestack which gives required prefix expression
7. Stop
- Trace of Conversion of infix expression to prefix algorithm**
- The following tracing of the algorithm illustrates convert infix to prefix the algorithm. Consider an infix expression
- $((A-(B+C))*D)\$(E+F)$

Scan character	Prestack	Opstack
))
F	F)
+	F)+
E	FE)+
(FE+
\$	FE+	\$
)	FE+	\$)
D	FE+D	\$)
*	FE+D	\$)*
)	FE+D	\$)*)
)	FE+D	\$)*()
C	FE+DC	\$)*()
+	FE+DC	\$)*()+
B	FE+DCB	\$)*()+
(FE+DCB+	\$)*()
-	FE+DCB+	\$)*()-
A	FE+DCB+A	\$)*()-
(FE+DCB+A-	\$)*(-
(FE+DCB+A-*	\$)*(-
.....	FE+DCB+A-* \$

Finally pop and display contain of given prestack we get \$*-A+BCD+EF. This is required prefix expression of given infix expression

Tracing example 2: Trace the algorithm to convert given infix to prefix expression

$(A+B*C*D)/((E+F-G)*H)$I/J$

Scan character	Prestack	Opstack
J	J
/	J	/
I	JI	/
\$	JI	/\$
)	JI	/\$)
H	JIH	/\$)
*	JIH	/\$)*
)	JIH	/\$)*)
G	JIHG	/\$)*)
-	JIHG	/\$)*) -
F	JIHGF	/\$)*) -
+	JIHGF	/\$)*) - +
E	JIHGFE	/\$)*) - +
(JIHGFE+-	/\$)*
(JIHGFE+- *	/\$
/	JIHGFE+- * \$	//
)	JIHGFE+- * \$	//)
D	JIHGFE+- * \$D	//)
\$	JIHGFE+- * \$D	//)\$
C	JIHGFE+- * \$DC	//)\$
*	JIHGFE+- * \$DC\$	//)*
B	JIHGFE+- * \$DC\$B	//)*
+	JIHGFE+- * \$DC\$B*	//)+
A	JIHGFE+- * \$DC\$B*A	//)+
(JIHGFE+- * \$DC\$B*A+	//)
.....	JIHGFE+- * \$DC\$B*A+ /	/)
.....	JIHGFE+- * \$DC\$B*A+ //

Finally pop and display contain of given prestack we get // + A * B \$ C D \$ * - + E F G H I J

This is required prefix expression of given infix expression

Complete program in C to convert an expression from infix to prefix expression

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
#include<string.h>
int precedence(char);
void main()
```

```

int i, otos=-1, ptos=-1, len, length;
char infix[100], prestack[100], opstack[100];
printf("Enter a valid infix\n");
gets (infix);
length=strlen(infix);
len=length;
for(i=length-1;i>=0;i--)
{
    if(infix[i]=='')
    {
        opstack[++otos]=infix[i];
        len++;
    }
    else if(isalpha(infix[i]))
    {
        prestack[++ptos]=infix[i];
    }
    else if (infix[i]=='(')
    {
        len++;
        while(opstack[otos]!=')')
        {
            prestack[++ptos]=opstack[otos];
            otos--;
        }
        otos--;
    }
    else //operators
    {
        if(precedency(opstack[otos])>precedency(infix[i]))
        {
            prestack[++ptos]=opstack[otos--];
            opstack[++otos]=infix[i];
        }
        opstack[++otos]=infix[i];
    }
}
while(otos!=-1)
{
    prestack[++ptos]=opstack[otos];
    otos--;
}

```

```

for(i=len-1; i>=0; i--) /*for displaying*/
{
    printf("%c", prestack[i]);
}
getch();

int precedence(char ch) /*precedence function*/
{
    switch(ch)
    {
        case '$':
            return(4);
        case '*':
        case '/':
            return(3);
        case '+':
        case '-':
            return(2);
        default:
            return(1);
    }
}

```

Output

Enter valid infix expression

 $(A+B*C/D)*(E-F/G)$ $+A^*/BCD-E/FG$ **Conversion of postfix expression to infix expression**

Here we use a single stack that contains the operands and converted sub strings and finally contain converted infix expression. Here we scan one character at a time from left to right of given postfix expression.

Conversion steps

1. While there are input symbol left in given postfix expression
2. Read the next symbol from input i.e. scan one character at a time from left to right of given postfix expression
3. If the symbol is an operand
Push it onto the stack.
4. Otherwise,
The symbol is an operator.
5. If there are fewer than 2 values on the stack
Show Error /* input not sufficient values in the expression */

6. Else
 - Pop the top 2 values from the stack.
 - Put the operator, with the values as arguments and form a string.
 - Encapsulate the resulted string with parenthesis.
 - Push the resulted string back to stack.
7. If there is only one value in the stack
 - That value in the stack is the desired infix string.
8. If there are more values in the stack
 - Show Error /* the user input has too many values */

Example: Convert following postfix expression to their equivalent infix expression,
ABC-+DE-FG-H+/*

Expression	Stack
ABC-+DE-FG-H+/*	Null
BC-+DE-FG-H+/*	A
C-+DE-FG-H+/*	AB
-+DE-FG-H+/*	ABC
+DE-FG-H+/*	A (B-C)
DE-FG-H+/*	(A+B-C)
E-FG-H+/*	(A+B-C) D
-FG-H+/*	(A+B-C) D E
FG-H+/*	(A+B-C) (D-E)
G-H+/*	(A+B-C) (D-E) F
-H+/*	(A+B-C) (D-E) FG
H+/*	(A+B-C) (D-E) (F-G)
+/*	(A+B-C) (D-E) (F-G) H
/*	(A+B-C) (D-E) (F-G+H)
*	(A+B-C) ((D-E)/ (F-G+H))
Null	(A+B-C) * (D-E)/ (F-G+H)

Postfix to Infix implementation in C Programming

Here we use linked list so it is better to use this program after the study of linked list unit.

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

```
#include <stdlib.h>
```

```
int top = 10;
```

```
struct node
```

```
{
```

```
    char ch;
```

```
    struct node *next;
```

```
    struct node *prev;
```

```
} *stack[11];
```

```

typedef struct node node;
void push(node *str)
{
    if (top <= 0)
        printf("Stack is Full ");
    else
    {
        stack[top] = str;
        top++;
    }
}

node *pop()
{
    node *exp;
    if (top >= 10)
        printf("Stack is Empty ");
    else
        exp = stack[+top];
    return exp;
}

void convert(char exp[])
{
    node *op1, *op2;
    node *temp;
    int i;
    for (i=0;exp[i]!='\0';i++)
        if (exp[i] >= 'a' && exp[i] <= 'z' || exp[i] >= 'A' && exp[i] <= 'Z')
        {
            temp = (node*)malloc(sizeof(node));
            temp->ch = exp[i];
            temp->next = NULL;
            temp->prev = NULL;
            push(temp);
        }
        else if (exp[i] == '+' || exp[i] == '-' || exp[i] == '*' || exp[i] == '/' || exp[i] == '$')
        {
            op1 = pop();
            op2 = pop();
            temp = (node*)malloc(sizeof(node));
            temp->ch = exp[i];
            temp->next = NULL;
            temp->prev = op2;
            op2->next = temp;
            push(op1);
        }
}

```

```

        temp->ch = exp[i];
        temp->next = op1;
        temp->prev = op2;
        push(temp);
    }

void display(node *temp)
{
    if (temp != NULL)
    {
        display(temp->prev);
        printf("%c", temp->ch);
        display(temp->next);
    }
}

void main()
{
    char exp[50];
    clrscr();
    printf("Enter the postfix expression :");
    scanf("%s", exp);
    convert(exp);
    printf("\n The Equivalent Infix expression is:");
    display(pop());
    printf("\n\n");
    getch();
}

```

Conversion of prefix expression to infix expression

Here we use a single stack that contains the operands and converted sub strings and finally contain converted infix expression. Here we scan one character at a time from right to left of given prefix expression.

Conversion steps

1. While there are input symbol left
2. Read the next symbol from input i.e. read one character at a time from right to left of given prefix expression
3. If the symbol is an operand
Push it onto the stack.
4. Otherwise,
The symbol is an operator.
5. If there are fewer than 2 values on the stack
Show Error /* input not sufficient values in the expression */

6. Else
Pop the top 2 values from the stack.

Put the operator, with the values as arguments and form a string.

Encapsulate the resulted string with parenthesis.

Push the resulted string back to stack.

7. If there is only one value in the stack

That value in the stack is the desired infix string.

8. If there are more values in the stack

Show Error /* the user input has too many values */

Example: Convert following prefix expression to their equivalent infix expression.

*+A-BC/-DE+-FGH

Expression	Stack
*+A-BC/-DE+-FGH	Null
*+A-BC/-DE+-FG	H
*+A-BC/-DE+-F	HG
*+A-BC/-DE+-	HGF
*+A-BC/-DE+	H (F-G)
*+A-BC/-DE	(F-G+H)
*+A-BC/-D	(F-G+H) E
*+A-BC/-	(F-G+H) ED
*+A-BC/	(F-G+H) (D-E)
*+A-BC	(D-E)/ (F-G+H)
*+A-B	(D-E)/ (F-G+H) C
*+A-	(D-E)/ (F-G+H) CB
*+A	(D-E)/ (F-G+H) (B-C)
*+	(D-E)/ (F-G+H) (B-C) A
*	(D-E)/ (F-G+H) (A+B-C)
Null	(A+B-C)* (D-E)/ (F-G+H)

Postfix to Infix implementation in C-Programming

```
#include <string.h>
#include <ctype.h>
#include <conio.h>
char opnds[50][80], oprs[50];
int topr=-1,topd=-1;
pushd(char *opnd)
{
    strcpy(opnds[++topd],opnd);
}
char *popd()
{
    return(opnds[topd--]);
```

```

pushr(char opr)
{
    oprs[++topr]=opr;
}
char popr()
{
    return(oprs[topr--]);
}
int empty(int t)
{
    if( t == 0) return(1);
    return(0);
}

void main()
{
    char prfx[50],ch,str[50],opnd1[50],opnd2[50],opr[2];
    int i=0,k=0,opndcnt=0;
    clrscr();
    printf("Give an Expression = ");
    gets(prfx);
    printf(" Given Prefix Expression : %s\n", prfx);
    while( (ch=prfx[i++]) != '\0')
    {
        if(isalnum(ch))
        {
            str[0]=ch; str[1]='\0';
            pushd(str); opndcnt++;
            if(opndcnt >= 2)
            {
                strcpy(opnd2,popd());
                strcpy(opnd1,popd());
                strcpy(str,"(");
                strcat(str,opnd1);
                ch=popr();
                opr[0]=ch;opr[1]='\0';
                strcat(str,opr);
                strcat(str,opnd2);
                strcat(str,")");
            }
        }
    }
}

```

```

        pushd(str);
        opndcnt=1;
    }
}
else
{
    pushr(ch);
    if(opndcnt==1)
        opndcnt=0; /* operator followed by single operand*/
}
if(!empty(topd))
{
    strcpy(opnd2,topd());
    strcpy(opnd1,topd());
    strcpy(str,"(");
    strcat(str,opnd1);
    ch=popr();
    opr[0]=ch;opr[1]='\0';
    strcat(str,opr);
    strcat(str,opnd2);
    strcat(str,")");
    pushd(str);
}
printf(" Infix Expression: ");
puts(opnds[topd()]);
getch();
}

```

Arithmetic Expression Evaluation

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.

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.,

$+AB$

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.,

$AB+$

Stack organized computers are better suited for post-fix notation then the traditional infix notation. Thus the infix notation must be converted to the post-fix notation. The conversion from infix notation to post-fix notation must take into consideration the operational hierarchy.

Evaluating the Postfix expression

Prefix and Postfix expressions can be evaluated faster than an infix expression. This is because we don't need to process any brackets or follow operator precedence rule. In postfix and prefix expressions whichever operator comes before will be evaluated first, irrespective of its priority. Also, there are no brackets in these expressions. As long as we can guarantee that a valid prefix or postfix expression is used, it can be evaluated with correctness.

Each operator in a postfix expression refers to the previous two operands in the expression. To evaluate the postfix expression, we use the following procedure:

- Each time we read an operand we push it onto a stack.
- When we reach an operator, its operands will be the top two elements on the stack.
- We can then pop these two elements perform the indicated operation on them and push the result on the stack so that it will be available for use as an operand of the next operator.

Consider an example

$$\begin{aligned} & 3 \ 4 \ 5 \ * \ + \\ & = 3 \ 20 \ + \\ & = 23 \ (\text{answer}) \end{aligned}$$

Evaluating the given postfix expression

$$\begin{aligned} & 6 \ 2 \ 3 \ + \ - \ 3 \ 8 \ 2 \ / \ + \ * \ 2 \ \$ \ 3 \ + \\ & = 6 \ 5 \ - \ 3 \ 8 \ 2 \ / \ + \ * \ 2 \ \$ \ 3 \ + \\ & = 1 \ 3 \ 8 \ 2 \ / \ + \ * \ 2 \ \$ \ 3 \ + \\ & = 1 \ 3 \ 4 \ + \ * \ 2 \ \$ \ 3 \ + \\ & = 1 \ 7 \ * \ 2 \ \$ \ 3 \ + \\ & = 7 \ 2 \ \$ \ 3 \ + \\ & = 49 \ 3 \ + \\ & = 52 \end{aligned}$$

Algorithm to evaluate the postfix expression

Here we use only one stack called vstack (value stack).

1. Scan one character at a time from left to right of given postfix expression
 - 1.1 if scanned symbol is operand then
 - read its corresponding value and push it into vstack

- 1.2 if scanned symbol is operator then
 - pop and place into op2
 - pop and place into op1
 - Compute result according to given operator and push result into vstack
2. Pop and display which is required value of the given postfix expression
3. Stop

Trace of Evaluation of postfix expression

Consider an example to evaluate the postfix expression tracing the algorithm

$ABC+*CBA- +*$

$123+*321- +*$

Scanned character	Value	Op2	Op1	Result	vstack
A	1	1
B	2	1 2
C	3	1 2 3
+	3	2	5	15
*	5	1	5	5
C	3	5 3
B	2	5 3 2
A	1	5 3 2 1
-	1	2	1	5 3 1
+	1	3	4	5 4
*	4	5	20	20

Its final value is 20.

Complete program in C for evaluating postfix expression

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
#include<string.h>
void push(int);
int pop();
int vstack[100];
int tos=-1;
void main()
{
    int i, res, len, op1, op2, value[100];
    char postfix[100], ch;
    clrscr();
    printf("Enter a valid postfix\n");
    gets(postfix);
```

```

len=strlen(postfix);
for(i=0;i<=len-1;i++)
{
    if(isalpha(postfix[i]))
    {
        printf("Enter value of %c",postfix[i]);
        scanf("%d",&value[i]);
        push(value[i]);
    }
    else
    {
        ch=postfix[i];
        op2=pop();
        op1=pop();
        switch(ch)
        {
            case '+':
                push(op1+op2);
                break;
            case '-':
                push(op1-op2);
                break;
            case '*':
                push(op1*op2);
                break;
            case '/':
                push(op1/op2);
                break;
            case '$':
                push(pow(op1,op2));
                break;
            case '%':
                push(op1%op2);
                break;
        }
    }
}
printf("The result is:");
res=pop();
printf("%d", res);
getch();
}

void push(int val) /*insertion function*/
{
    vstack[++tos]=val;
}

```

```

int pop() /*deletion function*/
{
    int n;
    n=vstack[tos-1];
    return(n);
}

```

Output

Enter valid postfix expression

ab*c

Enter value of a

10

Enter value of b

2

Enter value of c

5

The result is: 15

Evaluating the Prefix Expression

To evaluate the given prefix expression is same as that of evaluation of postfix expression only different is that here we scan one character from right to left and set first popped element to first operand instead of second operand unlike in postfix expression evaluation. Consider an example

$$\begin{aligned}
 & + 5 * 3 2 \text{ prefix expression} \\
 & = + 5 6 \\
 & = 11
 \end{aligned}$$

Example 2: Evaluate the given prefix expression

$$\begin{aligned}
 & / + 5 3 - 4 2 \quad \text{prefix equivalent to } (5+3)/(4-2) \text{ infix notation} \\
 & = / 8 - 4 2 \\
 & = / 8 2 \\
 & = 4
 \end{aligned}$$

Example 3: Evaluate following prefix expression,

$$+ A - / B \$ C D * E + F - G H I \text{ where } A=1, B=2, C=3, D=2, E=1, F=5, G=9, H=3, I=2$$

At first set numeric values of given operands as,

$$\begin{aligned}
 & + 1 - / 2 \$ 3 2 * 1 + 5 - / 9 3 2 \\
 & = + 1 - / 2 9 * 1 + 5 - / 9 3 2 \\
 & = + 1 - 0 * 1 + 5 - / 9 3 2 \\
 & = + 1 - 0 * 1 + 5 - 3 2 \\
 & = + 1 - 0 * 1 + 5 1 \\
 & = + 1 - 0 * 1 6 \\
 & = + 1 - 0 6 \\
 & = + 1 (-6) \\
 & = -5 \text{ Ans.}
 \end{aligned}$$

Algorithm to evaluate the prefix expression

Here we use only one stack called vstack (value stack).

1. Scan one character at a time from right to left of given prefix expression

- 1.1 if scanned symbol is operand then

- read its corresponding value and push it into vstack

- 1.2 if scanned symbol is operator then

- pop and place into op1

- pop and place into op2

- Compute result according to given operator and push result into vstack

2. Pop and display which is required value of the given prefix expression

3. Stop

Trace of Evaluation of prefix expression

Consider an example to evaluate the prefix expression tracing the algorithm

$+A-/B$CD^*E+F-/GHI$ where A=1, B=2, C=3, D=2, E=1, F=5, G=9, H=3, I=2

Scanned character	Value	Op1	Op2	Result	vstack
I	2	2
H	3	23
G	9	239
/	9	3	3	23
B	3	2	1	1
F	5	15
+	5	1	6	6
E	1	61
*	1	6	6	6
D	2	62
C	3	623
\$	3	2	9	69
B	2	692
/	2	9	0	60
-	0	6	-6	-6
A	1	-61
+	1	-6	-5	-5

Its final value is -5.

Complete program in C for evaluating prefix expression

```
#include<stdio.h>
#include<conio.h>
#include<math.h>
#include<string.h>
void push(int);
```

```

int pop();
int vstack[100];
int tos=-1;
void main()
{
    int i, res, len, op1, op2, value[100];
    char prefix[100], ch;
    clrscr();
    printf("Enter a valid postfix\n");
    gets(postfix);
    len=strlen(postfix);
    for(i=len-1;i>=0;i--)
    {
        if(isalpha(postfix[i]))
        {
            printf("Enter value of %c", prefix[i]);
            scanf("%d", &value[i]);
            push(value[i]);
        }
        else
        {
            ch=prefix[i];
            op1=pop();
            op2=pop();
            switch(ch)
            {
                case '+':
                    push(op1+op2);
                    break;
                case '-':
                    push(op1-op2);
                    break;
                case '*':
                    push(op1*op2);
                    break;
                case '/':
                    push(op1/op2);
                    break;
                case '$':
                    push(pow(op1,op2));
                    break;
                case '%':
                    push(op1%op2);
                    break;
            }
        }
    }
}

```

```

    printf("The result is:");
    res=pop();
    printf("%d", res);
    getch();
}

void push(int val) /*insertion function*/
{
    vstack[++tos]=val;
}
int pop() /*deletion function*/
{
    int n;
    n=vstack[tos--];
    return(n);
}

```

Evaluation of infix expression

An infix expression is evaluated using two stacks, one for operator and another for operand. The infix string is read in an array, from which symbols/characters are fetched one by one and the following checks are performed:

1. If the character is an operand, push it to the operand stack.
2. If the character is an operator then do following:
 - If the operator stack is empty then push it to the operator stack.
 - Else If the operator stack is not empty,
 - If the character's precedence is greater than or equal to the precedence of the stack top of the operator stack, then push the character to the operator stack.
 - If the character's precedence is less than the precedence of the stack top of the operator stack then pop two operands say opnd2 and opnd1 and perform operation specified by popped operation.
 - Push the result in operand stack
 - Repeat above steps until the symbol becomes less than the popped operator.
3. If the character is "(", then push it onto the operator stack.
4. If the character is ")", then do Process (as explained above) until the corresponding ")" encountered in operator stack. Now just pop out the "(".
5. Repeat above steps until the operator stack is not empty.
6. Finally pop the value from the operand stack and return it.
7. Stop

Example: Evaluate following infix expression without using stack data structure
 $A+B-C*(D+E-F/G)$ where value of $A=1, B=4, C=2, D=8, E=3, F=9, G=3$

Solution:

$$\begin{aligned}
 & A+B-C*(D+E-F/G) \\
 & = 1+4-2*(8+3-9/3)
 \end{aligned}$$

$$= 1 + 4 - 2^2(8 + 3 - 3)$$

$$= 1 + 4 - 2^2(8 + 0)$$

$$= 1 + 4 - 2^2 \cdot 8$$

$$= 1 + 4 - 16$$

$$= 5 - 16$$

$$= -11$$

Trace of algorithm

Example: Evaluate following infix expression by using stack data structure

$A + B - C * (D + E - F / G)$ where value of A=1, B=4, C=2, D=8, E=3, F=9, G=3

Scan symbol	Value	Operator Stack	Operand Stack	Operand 2	Operand 1	Result
A	1		1			
+		+				
B	4	+	14			
-		+-	14			
C	2	+-	142			
*		+-*	142			
(+-*(142			
D	8	+-*(1428			
+		+-*(+	1428			
E	3	+-*(+	14283			
-		+-*(+-	14283			
F	9	+-*(+-	142839			
/		+-*(+-/	142839			
G	3	+-*(+-/	1428393			
)		+-*(+-	142833	3	9	3
		+-*(+	14280	3	3	0
		+-*(+-	1428	0	8	8
		+-*	1428			
		+-	1416	8	2	16
		+	1-12	16	4	-12
			-11	-12	1	-11

Example 2: Evaluate following infix expression by using stack data structure
 $(A+B-C)*D+E-(F/G)$ where value of A=1, B=4, C=2, D=8, E=3, F=9, G=3

Scan symbol	Value	Operator Stack	Operand Stack	Operand 2	Operand 1	Result
((
A	1	(1			
+		(+				
B	4	(+)	14			
-		(+-)	14			
C	2	(+-)	142			
)		(+)	12	2	4	2
*		(3	2	1	3
D	8	*	38			
+		+	24	8	3	24
E	3	+	243			
-		+-	243			
(+-()	243			
F	9	+-()	2439			
/		+-(/)	2439			
G	3	+-(/)	24393			
)		+-()	2433	3	9	3
.....		+-	2433			
		+	240	3	3	0
		24	0	24	24

MULTIPLE CHOICE QUESTIONS

- Which of the following is an application of stack?
 - Finding factorial
 - Infix to postfix conversion
 - Tower of Hanoi
 - All of the above
- The data structure which is one ended is
 - queue
 - stack
 - tree
 - graph
- When does top value of the stack changes?
 - Before deletion
 - While checking underflow
 - At the time of deletion
 - After deletion

4. is very useful in situation when data have to stored and then retrieved in reverse order.
- Stack
 - Queue
 - List
 - Link list
5. Inserting an item into the stack when stack is not full is called Operation and deletion of item from the stack, when stack is not empty is called operation.
- push, pop
 - pop, push
 - insert, delete
 - delete, insert
6. Process of inserting an element in stack is called _____
- Create
 - Push
 - Evaluation
 - Pop
7. In a stack, if a user tries to remove an element from empty stack it is called _____
- Underflow
 - Empty collection
 - Overflow
 - Garbage Collection
8. Which of the following applications may use a stack?
- A parentheses balancing program
 - Tracking of local variables at run time
 - Compiler Syntax Analyzer
 - Data Transfer between two asynchronous process
9. What is the value of the postfix expression $6\ 3\ 2\ 4\ +\ -\ *:$
- 1
 - 40
 - 74
 - 18
10. Entries in a stack are "ordered". What is the meaning of this statement?
- A collection of stacks is sortable
 - Stack entries may be compared with the ' $<$ ' operation
 - The entries are stored in a linked list
 - There is a Sequential entry that is one by one



DISCUSSION EXERCISE

- What is stack? How it is differ from queue? Show that stack and queue as ADT.
- How you can convert given infix expression into their equivalent postfix expression?
- Evaluate following postfix expressions by using stack:

- $A\ B\ -\ C\ D\ E\ F\ / +\ G\ -\ *\ -\ H\ -$
- $A\ B\ C\ D\ * -\ E\ / +\ F +\ G\ H\ / -$
- $A\ B +\ C\ * D\ E\ F\ G\ * +\ / +\ H\ -$

Where, A=4, B=8, C=2, D=5, E=6, F=9, G=1, H=3

- Convert following infix expressions into prefix and postfix expressions:

$$\begin{aligned} &A * (B + C) / \\ &(A * (B + (C / D))) \end{aligned}$$

5. Write a program to add any number of large integers. The problem can be approached at least two ways.
- First, add two numbers and then repeatedly add the next number with the result of previous addition.
 - Create a vector of stacks and then use a generalized version of addingLargeNumbers(all stacks at the same time).
6. Write a program that determines whether an input string is a palindrome; that whether it can be read the same way forward and backward. At each point, you can read only one character of the input string; do not use an array to first store this string and then analyze it (except, possibly, in a stack implementation). Consider using multiple stacks.
7. Write down basic operations of stack. And also describe push and pop function of stack.
8. What is main concept behind recursion? Write a program in java to find reverse of given string by using recursion.
9. Put the elements on the stack S in ascending order using one additional stack and some additional non-array variables.
10. What are the basic operations of stack? Describe them with suitable java program.
11. Show that stack as an ADT.
12. Trace the algorithm to convert given infix to postfix expressions:
 - $(A + B) * C + D / (E + F * G) - H$
 - $A + ((B - C * D) / E) + F - G / H$
 - $(A * B + C) / D - E / (F + G)$
 - $A - B - C * (D + E / F - G) - H$
 - $A * B / C + (D + E - (F * (G / H)))$
 - $((H * (((A + ((B + C) * D)) * F) * G) * E)) + J$
13. What are the application areas of stack in computer application fields?
14. What is dynamic programming? How it is differ from greedy algorithm?
15. What is Stack and where it can be used?
16. What are Infix, prefix, Postfix notations?
17. What are the various operations that can be performed on different Data Structures?
18. Write down an algorithm to convert given infix to postfix expression.
19. Write down an algorithm to evaluate given postfix expression.
20. Transfer elements from stack S_1 to stack S_2 so that the elements from S_2 are in the same order as on S_1
 - Using one additional stack
 - Using no additional stack but only some additional non-array variables



3

Queues are abstract data structures that store and manage groups of elements in sequence. Commonly used in computer science, they are often used in conjunction with stacks and linked lists.

Queues are linear data structures that follow the First In First Out (FIFO) principle. This means that the first element added to the queue is the first one to be removed from it.

Queues are commonly used in computer science for tasks such as printing, file processing, and network communication. They are also used in everyday life, such as in a bank or post office, where people wait in line to be served.

QUEUE



Queues are a type of abstract data structure (ADT) that follows the First In First Out (FIFO) principle. This means that the first element added to the queue is the first one to be removed from it. Queues are commonly used in computer science for tasks such as printing, file processing, and network communication. They are also used in everyday life, such as in a bank or post office, where people wait in line to be served.

CHAPTER OUTLINE

After studying this chapter, the reader will be able to understand the

- Introduction, Queue as an ADT, Primitive operations in Queue, Linear and circular queue and their applications, Enqueue and Dequeue, Priority queue

INTRODUCTION

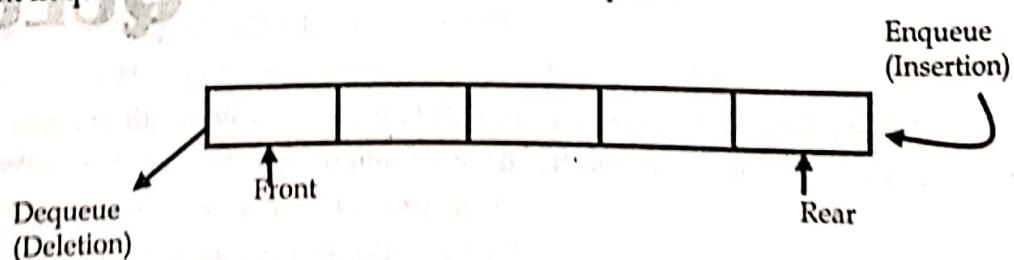
Queue is an abstract data structure, somewhat similar to Stacks. Unlike stacks, a queue is open at both its ends. One end is always used to insert data (enqueue) and the other is used to remove data (dequeue). Queue follows First-In-First-Out methodology, i.e., the data item stored first will be accessed first.



A real-world example of queue can be a single-lane one-way road, where the vehicle enters first and exits first. More real-world examples can be seen as queues at the ticket windows and bus-stops.

WHAT IS A QUEUE?

Queue is a Linear Data Structure which follows First in First out mechanism. It means: the first element inserted is the first one to be removed. Queue uses two variables rear and front. Rear is incremented while inserting an element into the queue and front is incremented while deleting an element from the queue. Queue is also called First in First out (FIFO) system since the first element in queue will be the first element out of the queue.



Like stacks, queues may be represented in various ways, usually by means of one-way lists or linear arrays. Generally, they are maintained in linear array QUEUE. Two pointers FRONT and REAR are used to represent front and last element respectively. N may be the size of the linear array. The condition when FRONT is NULL indicating that the queue is empty. The condition when REAR is N indicated overflow.

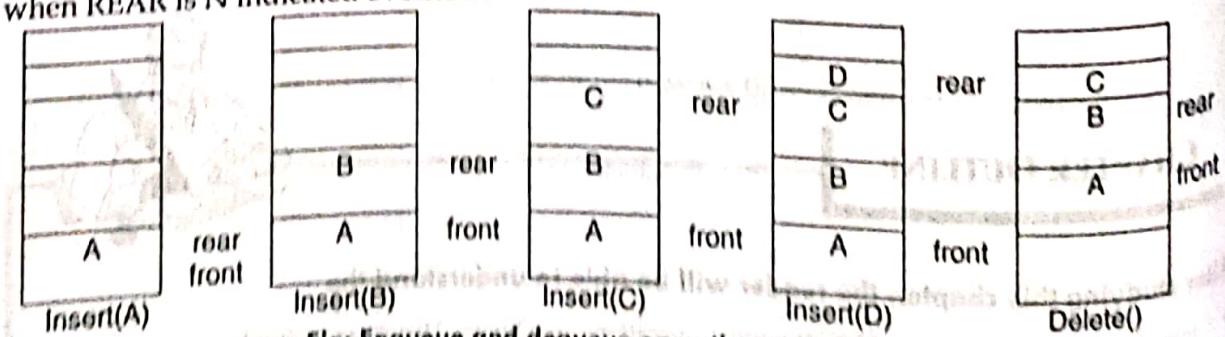


Fig: Enqueue and dequeue operations of linear queue

Basic features of Queue

- Like stack, queue is also an ordered list of elements of similar data types.
- Queue is a FIFO (First in First Out) structure.
- Once a new element is inserted into the Queue, all the elements inserted before the new element in the queue must be removed, to remove the new element.
- peek() function is often used to return the value of first element without dequeuing it.

Applications of Queue

Due to the fact that queue performs actions on first in first out basis which is quite fair for the ordering of actions. There are various applications of queues discussed as below.

1. In real life scenario, Call Center phone systems uses Queues to hold people calling them in an order, until a service representative is free.
2. Queues are widely used as waiting lists for a single shared resource like printer, disk, CPU.
3. Queues are used in asynchronous transfer of data (where data is not being transferred at the same rate between two processes) for e.g. pipes, file IO, sockets.
4. Queues are used as buffers in most of the applications like MP3 media player, CD player, etc.
5. Queue is used to maintain the play list in media players in order to add and remove the songs from the play-list.
6. Queues are used in operating systems for handling interrupts.
7. Task waiting for the printing

Basic Operations of queue

Queue operations may involve initializing or defining the queue, utilizing it, and then completely erasing it from the memory. Here we shall try to understand the basic operations associated with queues:

- enqueue() – add (store) an item to the queue.
- dequeue() – remove (access) an item from the queue.
- Display() – Retrieve all elements of queue.

Few more functions are required to make the above-mentioned queue operation efficient. These are –

- peek() – Gets the element at the front of the queue without removing it.
- isfull() – Checks if the queue is full.
- isempty() – Checks if the queue is empty.

In queue, we always dequeue (or access) data, pointed by front pointer and while enqueueing (or storing) data in the queue we take help of rear pointer.

The Queue as ADT

A queue q of type T is a finite sequence of elements with the operations

- **MakeEmpty(q):** To make q as an empty queue
- **IsEmpty(q):** To check whether the queue q is empty or not. Return true if q is empty, return false otherwise.
- **IsFull(q):** To check whether the queue q is full or not. Return true if q is full, return false otherwise.
- **Enqueue(q, x):** To insert an item x at the rear of the queue, if and only if q is not full;
- **Dequeue(q):** To delete an item from the front of the queue q, if and only if q is not empty.
- **Traverse (q):** To read entire queue that is display the content of the queue.

Implementation of queue

There are two techniques for implementing the queue:

- **Array implementation of queue (static memory allocation)**
- **Linked list implementation of queue (dynamic memory allocation)**

Array implementation of queue

In array implementation of queue, an array is used to store the data elements. Array implementation is also further classified into three types

- **Linear array implementation (linear queue)**
- **Circular array implementation (Circular queue)**
- **Priority queue**

Linear queue

By default the queue is linear. Up to now we describe queue as linear queue. So the description of linear queue is same as described above.

Structure of linear queue

```
#define SIZE 10
Struct LQueue
{
    int rear;
    int front;
    int item[SIZE];
}; typedef Struct LQueue qt;
qt rear, front;
rear=-1;
front=0;
```

Algorithm for insertion an item in queue (Enqueue)

1. Start
2. Initialize front=0 and rear=-1
If rear>=MAXSIZE-1
Print "queue overflow" and return
Else
Set, rear=rear+1
queue[rear]=item
3. Stop

Algorithm to delete an element from the queue (Dequeue)

1. Start
2. If rear<front
Print "queue is empty" and return
Else
Item=queue [front++]
Print "item" as a deleted element
3. Stop

Defining the operations of linear queue

The MakeEmpty function

```
void makeEmpty(qt *q)
```

```
{
    q->rear=-1;
    q->front=0;
}
```

The IsEmpty function

```
int IsEmpty(qt *q)
```

```
{
    if(q->rear<q->front)
        return 1;
    else
        return 0;
}
```

The Isfull function

```
int IsFull(qt *q)
```

```
{
    if(q->rear==MAXSIZE-1)
        return 1;
    else
        return 0;
}
```

The Enqueue function

```
void Enqueue(qt *q, int newItem)
```

```
{
    if(IsFull(q))
    {
        // handle overflow
    }
    else
    {
        // handle enqueue
    }
}
```

```

        printf("queue is full");
        exit(1);
    }
    else
    {
        q->rear++;
        q->items[q->rear]=newitem;
    }
}

```

The Dequeue function

```

int Dequeue(qt *q)
{
    if(IsEmpty(q))
    {
        printf("queue is Empty");
        exit(1);
    }
    else
    {
        return(q->items[q->front]);
        q->front++;
    }
}

```

Menu driven program for array implementation of linear queue in C

```

#include<stdio.h>
#include<conio.h>
#define SIZE 20
struct queue
{
    int item[SIZE];
    int rear;
    int front;
};
typedef struct queue qu;
void insert(qu* );
void delet(qu* );
void display(qu* );
void main()
{
    int ch;
    qu *q;
    q->rear=-1;
    q->front=0;
    clrscr();
    printf("Menu for program:\n");
}

```

```

printf("1:insert\n2:delete\n3:display\n4:exit\n");
do
{
    printf("Enter your choice\n");
    scanf("%d", &ch);
    switch(ch)
    {
        case 1:
            insert(q);
            break;
        case 2:
            delet(q);
            break;
        case 3:
            display( q );
            break;
        case 4:
            exit(1);
            break;
        default:
            printf("Your choice is wrong\n");
    }
}while(ch<5);
getch();
}

/*insert function*/
void insert(qu *q)
{
    int d;
    printf("Enter data to be inserted\n");
    scanf("%d",&d);
    if(q->rear==SIZE-1)
    {
        printf("Queue is full\n");
    }
    else
    {
        q->rear++;
        q->item[q->rear]=d;
    }
}
void delet(qu *q) /*delete function*/
{
    int d;
    if(q->rear<q->front)
    {
        printf("Queue is empty\n");
    }
}

```

```

else
{
    d=q->item[q->front];
    q->front++;
    printf("Deleted item is:");
    printf("%d\n",d);
}
void display(qu *q) /*display function*/
{
    int i;
    if(q->rear<q->front)
    {
        printf("Queue is empty\n");
    }
    else
    {
        for(i=q->front;i<=q->rear;i++)
        {
            printf("%d\t", q->item[i]);
        }
    }
}

```

Output

1: Enqueue

2: Dequeue

3: Display

Enter your choice

1

Enter data item to be inserted

99

Enter your choice

1

Enter data item to be inserted

55

Enter your choice

1

Enter data item to be inserted

45

Enter your choice

3

Queue elements are:

99 55 45 Enter your choice

2

Deleted element=99

Enter your choice

2

Deleted element=55

Enter your choice

Complexity Analysis of Queue Operations

Just like Stack, in case of a Queue too, we know exactly, on which position new element will be added and from where an element will be removed, hence both these operations requires a single step.

- Enqueue: $O(1)$
- Dequeue: $O(1)$
- Display: $O(n)$
- Search: $O(n)$

Problems in linear queue

Although, the technique of creating a queue is easy, but there are some drawbacks of using this technique to implement a queue.

Memory wastage: The space of the array, which is used to store queue elements, can never be reused to store the elements of that queue because the elements can only be inserted at front end and the value of front might be so high so that, all the space before that, can never be filled.

Deleted	Deleted	Deleted	Deleted	Deleted	10	20	30		
0	1	2	3	4	5	6	7	8	9

Front Rear

The above figure shows how the memory space is wasted in the array representation of queue. In the above figure, a queue of size 10 having 3 elements is shown. The value of the front variable is 5; therefore, we cannot reinsert the values in the place of already deleted element before the position of front. That much space of the array is wasted and cannot be used in the future (for this queue).

CIRCULAR QUEUE

A circular queue is one in which the insertion of a new element is done at very first location of the queue if the last location of the queue is full. When the rear part reaches at the last index of linear queue but front part is not at first index of the queue then in this case if we try to insert some elements, then according to the logic when rear is $N-1$ then it encounters an overflow situation. But there are some elements are left blank at the beginning part of the array. To utilize those left over spaces more efficiently, a circular fashion is implemented in queue representation. The circular fashion of queue reassigned the rear pointer with 0 if it reaches $N-1$ and beginning elements are free and the process is continued for deletion also. Such queues are called Circular Queue.

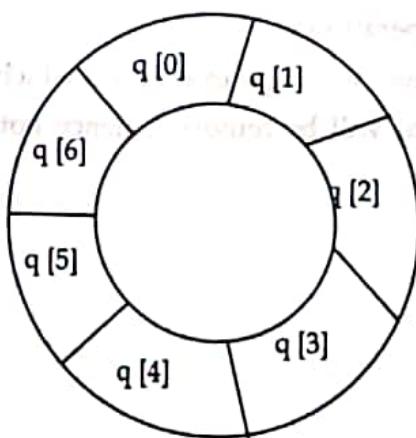


Fig: Circular queue with 7 data items

A circular queue overcomes the problem of unutilized space in linear queue implementation as array. In circular queue we sacrifice one element of the array thus to insert n elements in a circular queue we need an array of size $n+1$ (or we can insert one less than the size of the array in circular queue).

Initialization of Circular queue

```
rear = front=MAXSIZE-1
```

Algorithms for inserting an element in a circular queue

This algorithm is assumed that rear and front are initially set to MAXSIZE-1.

1. Check queue full condition as,
if (front===(rear+1)%MAXSIZE)
 print Queue is full and exit
else

2. cqueue[rear]=item;
3. Stop

Algorithms for deleting an element from a circular queue

This algorithm is assumed that rear and front are initially set to MAXSIZE-1.

1. [Checking empty condition]
if (rear==front)
 Print Queue is empty and exit
else

2. front=(front+1)%MAXSIZE; [increment front by 1]
3. item=cqueue[front];
4. return item;
5. Stop

Declaration of a Circular Queue

```
# define MAXSIZE 50 /* size of the circular queue items*/
```

```
struct cqueue
```

```
{  
    int front;  
    int rear;
```

```
int items[MAXSIZE];
};

typedef struct cqueue cq;
```

The IsEmpty function

```
int IsEmpty()
{
    if(rear==front)
        return 1;
    else
        return 0;
}
```

The Isfull function

```
int IsFull()
{
    if(front==(rear+1)%SIZE)
        return 1;
    else
        return 0;
}
```

The Enqueue function

```
void Enqueue(cq *q, int newitem)
{
    if(q->front==(q->rear+1)%MAXSIZE)
    {
        printf("queue is full");
        exit(1);
    }
    else
    {
        q->rear=(q->rear+1)%MAXSIZE;
        q->items[q->rear]=newitem;
    }
}
```

The Dequeue function

```
int Dequeue(cq *q)
{
    if(q->rear<q->front)
    {
        printf("queue is Empty");
        exit(1);
    }
    else
    {
        printf("Front choice is wrong");
    }
}
```

```

    q->front=(q->front+1)%MAXSIZE;
    return(q->items[q->front]);
}
}
}

```

Array implementation of circular queue with sacrificing one cell in C

```

#include<stdio.h>
#include<conio.h>
#define SIZE 5
struct cqueue
{
    int item[SIZE];
    int rear;
    int front;
};
typedef struct cqueue qu;
void insert(qu *);
void delet(qu *);
void display(qu *);
void main()
{
    int ch;
    qu *q;
    q->rear=SIZE-1;
    q->front=SIZE-1;
    clrscr();
    printf("Menu for program:\n");
    printf("1.Enqueue\n2.Dequeue\n3.Display\n4:exit\n");
    do
    {
        printf("Enter your choice\n");
        scanf("%d",&ch);
        switch(ch)
        {
            case 1:
                insert(q);
                break;
            case 2:
                delet(q);
                break;
            case 3:
                display(q);
                break;
            case 4:
                exit(1);
                break;
            default:
                printf("Your choice is wrong\n");
        }
    }
}

```

```

        }
    }while(ch<5);
    getch();

}

void insert(qu *q) /*insert function*/
{
    int d;
    if((q->rear+1)%SIZE==q->front)
        printf("Queue is full\n");
    else
    {
        q->rear=(q->rear+1)%SIZE;
        printf ("Enter data to be inserted\n");
        scanf("%d",&d);
        q->item[q->rear]=d;
    }
}

void delet(qu *q) /*delete function*/
{
    if(q->rear==q->front)
        printf("Queue is empty\n");
    else
    {
        q->front=(q->front+1)%SIZE;
        printf("Deleted item is:");
        printf("%d\n",q->item[q->front]);
    }
}

void display(qu *q) /*display function*/
{
    int i;
    if(q->rear==q->front)
        printf("Queue is empty\n");
    else
    {
        printf("Items of queue are:\n");
        for(i=(q->front+1)%SIZE;i!=q->rear;i=(i+1)%SIZE)
        {
            printf("%d\t",q->item[i]);
        }
        printf("%d\t",q->item[q->rear]);
    }
}

```

Output

- 1: Enqueue
- 2: Dequeue
- 3: Display
- 4: Exit

Enter your choice
1
Enter data item to be inserted
22
Enter your choice
1
Enter data item to be inserted
54
Enter your choice
1
Enter data item to be inserted
22
Enter your choice
1
Enter data item to be inserted
77
Enter your choice
1
Enter data item to be inserted
99
Queue full
Enter your choice
3
Queue elements are:
22 54 22 77 Enter your choice
2
Deleted element=22
Enter your choice
2
Deleted element=54
Enter your choice
2
Deleted element=22
Enter your choice
1
Enter data item to be inserted
43
Enter your choice
3
Queue elements are:
77 43 Enter your choice

Array Implementation of circular queue without sacrificing one cell by using a count variable

```
#include<stdio.h>
#include<conio.h>
#define SIZE 5
struct cqueue
```

```

int item[SIZE];
int rear;
int front;
};

int count=0;
typedef struct cqueue qu;
void insert(qu *);
void delet(qu *);
void display(qu *);
void main()
{
    int ch;
    qu *q;
    q->rear=SIZE-1;
    q->front=SIZE-1;
    clrscr();
    printf("Menu for program:\n");
    printf("1:Enqueue\n2:Dequeue\n3:Display\n4:exit\n");
    do
    {
        printf("Enter your choice\n");
        scanf("%d",&ch);
        switch(ch)
        {
            case 1:
                insert(q);
                break;
            case 2:
                delet(q);
                break;
            case 3:
                display(q);
                break;
            case 4:
                exit(1);
                break;
            default:
                printf("Your choice is wrong\n");
        }
    }while(ch<5);
    getch();
}

void insert(qu *q) /*insert function*/
{
    int d;
    if(count==SIZE)
        printf("Queue is full\n");
}

```

```

    else
    {
        q->rear=(q->rear+1)%SIZE;
        printf("Enter data to be inserted\n");
        scanf("%d",&d);
        q->item[q->rear]=d;
        count++;
    }
}

void delet(qu *q) /*delete function*/
{
    if(count==0)
        printf("Queue is empty\n");
    else
    {
        q->front=(q->front+1)%SIZE;
        printf("Deleted item is:");
        printf("%d\n",q->item[q->front]);
        count--;
    }
}

void display(qu *q) /*display function*/
{
    int i;
    if(q->rear==q->front)
        printf("Queue is empty\n");
    else
    {
        printf("Items of queue are:\n");
        for(i=(q->front+1)%SIZE; i!=q->rear; i=(i+1)%SIZE)
        {
            printf("%d\t",q->item[i]);
        }
        printf("%d\t",q->item[q->rear]);
    }
}

```

Output

1: Enqueue

2: Dequeue

3: Display

4: Exit

Enter your choice

1

Enter data item to be inserted

22

Enter your choice

1 Enter data item to be inserted
 44 Enter your choice
 1 Enter data item to be inserted
 23 Enter your choice
 1 Enter data item to be inserted
 76 Enter your choice
 1 Enter data item to be inserted
 16 Enter your choice
 1 Enter data item to be inserted
 .22 Queue full
 Enter your choice
 3 Queue elements are:
 22 44 23 76 16 Enter your choice
 Deleted element=22
 Enter your choice
 2 Deleted element=44
 Enter your choice
 2 Deleted element=23
 Enter your choice
 3 Queue elements are:
 76 16 Enter your choice

1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26	27	28	29	30	31	32	33	34	35	36	37	38	39	40	41	42	43	44	45	46	47	48	49	50	51	52	53	54	55	56	57	58	59	60	61	62	63	64	65	66	67	68	69	70	71	72	73	74	75	76	77	78	79	80	81	82	83	84	85	86	87	88	89	90	91	92	93	94	95	96	97	98	99	100	101	102	103	104	105	106	107	108	109	110	111	112	113	114	115	116	117	118	119	120	121	122	123	124	125	126	127	128	129	130	131	132	133	134	135	136	137	138	139	140	141	142	143	144	145	146	147	148	149	150	151	152	153	154	155	156	157	158	159	160	161	162	163	164	165	166	167	168	169	170	171	172	173	174	175	176	177	178	179	180	181	182	183	184	185	186	187	188	189	190	191	192	193	194	195	196	197	198	199	200	201	202	203	204	205	206	207	208	209	210	211	212	213	214	215	216	217	218	219	220	221	222	223	224	225	226	227	228	229	230	231	232	233	234	235	236	237	238	239	240	241	242	243	244	245	246	247	248	249	250	251	252	253	254	255	256	257	258	259	260	261	262	263	264	265	266	267	268	269	270	271	272	273	274	275	276	277	278	279	280	281	282	283	284	285	286	287	288	289	290	291	292	293	294	295	296	297	298	299	300	301	302	303	304	305	306	307	308	309	310	311	312	313	314	315	316	317	318	319	320	321	322	323	324	325	326	327	328	329	330	331	332	333	334	335	336	337	338	339	340	341	342	343	344	345	346	347	348	349	350	351	352	353	354	355	356	357	358	359	360	361	362	363	364	365	366	367	368	369	370	371	372	373	374	375	376	377	378	379	380	381	382	383	384	385	386	387	388	389	390	391	392	393	394	395	396	397	398	399	400	401	402	403	404	405	406	407	408	409	410	411	412	413	414	415	416	417	418	419	420	421	422	423	424	425	426	427	428	429	430	431	432	433	434	435	436	437	438	439	440	441	442	443	444	445	446	447	448	449	450	451	452	453	454	455	456	457	458	459	460	461	462	463	464	465	466	467	468	469	470	471	472	473	474	475	476	477	478	479	480	481	482	483	484	485	486	487	488	489	490	491	492	493	494	495	496	497	498	499	500
---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----	-----

PRIORITY QUEUE

A priority queue is a collection of elements such that each element has been assigned a priority value such that the order in which elements are deleted and processed comes from the following rules. Priority Queue is an extension of queue with following properties.

- Every item has a priority associated with it.
- An element with high priority is dequeued before an element with low priority.
- If two elements have the same priority, they are served according to their order in the queue.

Array Implementation of ascending priority queue in C

Here a one dimensional array is used to store the elements and elements are stored in given sequence of data input but they are deleted in ascending order i.e. delete smallest element first. This means giving priority to shortest job first.

Example: Implementation of ascending priority queue

```
#include<stdio.h>
#include<conio.h>
#define SIZE 20
struct cqueue
{
    int item[SIZE];
    int rear;
    int front;
};
typedef struct queue pq;
void insert(pq* );
void delet(pq* );
void display(pq* );
void main()
{
    int ch;
    pq *q;
    q->rear=-1;
    q->front=0;
    clrscr();
    printf("Menu for program:\n");
    printf("1:Enqueue\n2:Dequeue\n3:Display\n4:Exit\n");
    do
    {
        printf("Enter your choice\n");
        scanf("%d",&ch);
        switch(ch)
        {
            case 1:
                insert(q);
                break;
            case 2:
                delet(q);
                break;
            case 3:
                display(q);
                break;
            case 4:
                exit(1);
                break;
        }
    } while(ch!=4);
}
```

```

default:
    printf("Your choice is wrong\n");
}
}while(ch<5);
getch();
}

void insert(pq *q) /*insert function*/
{
    int d;
    if(q->rear==SIZE-1)
        printf("Queue is full\n");
    else
    {
        printf ("Enter data to be inserted\n");
        scanf("%d",&d);
        q->rear++;
        q->item[q->rear]=d;
    }
}

void delet(pq *q) /*delete function*/
{
    int i, temp=0, x;
    x=q->item[q->front];
    if(q->rear<q->front)
    {
        printf("Queue is empty\n");
        return 0;
    }
    else
    {
        for(i=q->front+1; i<q->rear; i++)
        {
            if(x>q->item[i])
            {
                temp=i;
                x=q->item[i];
            }
        }
        for(i=temp; i< q->rear-1; i++)
        {
            q->item[i]=q->item[i+1];
        }
        q->rear--;
        return x;
    }
}

void display(pq *q) /*display function*/
{
}

```

```

int i;
if(q->rear < q->front)
    printf("Queue is empty\n");
else
{
    printf("Items of queue are:\n");
    for(i=q->front ; i<=q->rear; i++)
    {
        printf("%d\t", q->item[i]);
    }
}

```

Output

1: Enqueue

2: Dequeue

3: Display

4: Exit

Enter your choice

1

Enter data item to be inserted

99

Enter your choice

1

Enter data item to be inserted

76

Enter your choice

1

Enter data item to be inserted

34

Enter your choice

3

Queue elements are:

99 76 34 Enter your choice

2

Deleted element is= 34

Enter your choice

3

Queue elements are:

99 76 Enter your choice

1

Enter data item to be inserted

23

Enter your choice

3

Queue elements are:

99 76 23 Enter your choice

1

Enter data item to be inserted

232

Enter your choice

3

Queue elements are:

99 76 23 232 Enter your choice

2

Deleted element is= 23

Enter your choice

2

Deleted element is= 76

Enter your choice

2

Deleted element is= 99

Enter your choice

2

Deleted element is= 232

Enter your choice

2

Queue Empty

Enter your choice

Discussion Exercise

MULTIPLE CHOICE QUESTIONS

1. Which one of the following is an application of Queue Data Structure?
 - When a resource is shared among multiple consumers.
 - When data is transferred asynchronously (data not necessarily received at same rate as sent) between two processes
 - Load Balancing
 - All of the above
2. A linear list of elements in which deletion can be done from one end (front) and insertion can take place only at the other end (rear) is known as a ?
 - Queue
 - Stack
 - Tree
 - Linked list
3. A queue is a?
 - FIFO (First In First Out) list
 - LIFO (Last In First Out) list.
 - Ordered array
 - Linear tree
4. If the elements "A", "B", "C" and "D" are placed in a queue and are deleted one at a time, in what order will they be removed?
 - ABCD
 - DCBA
 - DCAB
 - ABCD
5. In the array implementation of circular queue, which of the following operation take worst case linear time?
 - Insertion
 - Deletion
 - To empty a queue
 - None
6. If the MAX_SIZE is the size of the array used in the implementation of circular queue. How is rear manipulated while inserting an element in the queue?
 - $\text{rear} = (\text{rear} \% 1) + \text{MAX_SIZE}$
 - $\text{rear} = \text{rear} \% (\text{MAX_SIZE} + 1)$
 - $\text{rear} = (\text{rear} + 1) \% \text{MAX_SIZE}$
 - $\text{rear} = \text{rear} + (1 \% \text{MAX_SIZE})$

7. A circular queue is implemented using an array of size 10. The array index starts with 0, front is 6, and rear is 9. The insertion of next element takes place at the array index.
 (a) 0 (b) 7 (c) 9 (d) 10
8. A normal queue, if implemented using an array of size MAX_SIZE, gets full when
 (a) Rear=MAX_SIZE-1 (b) Front=(rear+1)mod MAX_SIZE
 (c) Front=rear+1 (d) Rear=front
9. Queues serve major role in
 (a) Simulation of recursion (b) Simulation of arbitrary linked list
 (c) Simulation of limited resource allocation
 (d) D. All of the mentioned
10. If queue is implemented using arrays, what would be the worst run time complexity of queue and dequeue operations?
 (a) $O(n), O(n)$ (b) $O(1), O(n)$ (c) $O(n), O(1)$ (d) $O(1), O(1)$



DISCUSSION EXERCISE

1. What is queue? What are main drawbacks of linear queue over circular queue?
2. What is concept behind priority queue? Write complete java program to implement ascending priority queue.
3. What are the application areas of queue? Show that queue as ADT.
4. How can you implement circular queue in C by using linked list?
5. What are the application areas of priority queue? Show that priority queue act as an ADT.
6. What are the basic operations of queue? Describe them with suitable practical example.
7. What is queue? Differentiate between simple queue and circular queue.
8. What is FIFO? How it is differ from LIFO?
9. What is a Dequeue? Explain structure of queue with suitable example.
10. Which Data Structure should be used for implementing LRU cache?
11. What are the full and empty conditions for circular queue?
12. Show that priority queue act as an ADT.
13. In which case circular queue is more superior than linear queue?
14. Priority queue violates the queue property. Is this argument is true? Justify.
15. Write complete program in C to implement various operations of linear queue.
16. How circular queue differ from linear queue? Explain.
17. Define ascending and descending priority queue with suitable example.
18. What are the main operations of priority queue? Explain.
19. Describe advantages and disadvantages of circular queue.
20. What are the application areas of queue in real life? Explain.



4

4. List - A list is a collection of elements. It is a sequence of elements. It consists of data and some operations defined on it. It can be implemented using arrays or linked lists.

4.1 Static List

A static list is a list which has a fixed size and cannot be modified after creation.

For example, if we want to store the names of students in a class, then we can use a static list. We can define a fixed size for the list and then add the names of the students one by one. Once the list is full, we cannot add any more names to it. This is called a static list. It is also known as an array.

LIST

An array is a collection of elements of the same type. It is a static list. It is used to store data in contiguous memory locations. All the elements in an array have the same size and type. They are stored in a row-major order. An array is a collection of elements of the same type. It is a static list. It is used to store data in contiguous memory locations. All the elements in an array have the same size and type. They are stored in a row-major order.

Dynamic List

A dynamic list is a list which can be modified. It can be resized. It is a collection of elements which can be added or removed at any time. It is also known as a linked list. It is used to store data in a non-contiguous memory locations. All the elements in a dynamic list have the same size and type. They are stored in a row-major order.



CHAPTER OUTLINE

After studying this chapter, the reader will be able to understand the

- Introduction, Static and Dynamic List structure, Array implementation of list, Queues as a list

C- Function to insert new element at specified valid position of an existing list

```
void insert(int a[100], int n)
```

{

```
    int pos, nel, i;
```

```
    printf("Enter position at which you want to insert new element");
```

```
    scanf("%d", &pos);
```

```
    printf("Enter element to be inserted");
```

```
    scanf("%d", &nel);
```

```
    for(i=n-1; i>=pos; i--)
```

{

```
        a[i+1] = a[i];
```

}

```
    a[pos]=nel;
```

```
    n=n+1;
```

}

Deleting of an existing element from given list

This operation is used to delete any element from given list within valid position from 0 to $(n-1)^{\text{th}}$ index of given list. Where 0^{th} index is the first index of given list and $(n-1)^{\text{th}}$ index is the index of last element of given predefined list of size n. Here we need to shift all elements one position left from index of deleted element to the index of last element and finally decrease size of given list by one.

3	4	5	6	7	11	23	44	55	6				
0	1	2	3	4	5	6	7	8	9	99

Delete an element 11 from given list

3	4	5	6	7	23	44	55	6					
0	1	2	3	4	5	6	7	8	9	10	99

Algorithm

1. Start
2. Read position of element to be deleted 'pos'
3. Swap all elements one position left from specified position to last element.
4. decrement size of an list by one as,
 $n=n-1;$
5. Stop

C- Function to delete element at specified valid positions from an existing list

```
void delet(int a[100], int n)
```

{

```
    int pos, i;
```

```
    printf("Enter position at which you want to delete an element");
```

```
    scanf("%d", &pos);
```

```
    for(i=pos; i<n; i++)
```

```
        a[i] = a[i+1];
```

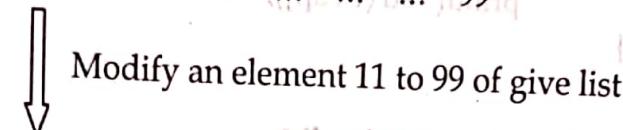
```
    n=n-1;
```

}

Modification of an existing element of given list

This operation is used to modify any element from given list within valid position from 0 to $(n-1)^{\text{th}}$ index of given list. Where 0^{th} index is the first index of given list and $(n-1)^{\text{th}}$ index is the index of last element of given predefined list of size n. Here we do not need to shift of elements.

3	4	5	6	7	11	23	44	55	6				
0	1	2	3	4	5	6	7	8	9	10	11	12	13



3	4	5	6	7	99	23	44	55	6				
0	1	2	3	4	5	6	7	8	9	10	11	12	13

Algorithm

- Start
- Read position of element to be update 'pos'
- Read element to be updated say it be 'el'
- Set element at given position as,
list[pos]=el;
- Stop

C- Function to update element at specified valid positions of an existing list

void update(int a[100], int n)

```
{
    int pos, num;
    printf("Enter position at which you want to update an element");
    scanf("%d", &pos);
    printf("Enter new element\n");
    scanf("%d", &num);
    a[pos]=num;
}
```

Traversing of an existing list

This operation is used to display all elements from first index to last index of given list. Here we do not need to shift of elements.

Algorithm

- Start
- Loop for $i=0$ to $(\text{list_size}-1)$
 - Display list[i]
 - Increment i by one as $i=i+1$
- Stop

C- Function to Traverse of an existing list

```
void traverse(int a[100], int n)
```

```
{
```

```
    int i;
```

```
    printf("Current elements of list are:\n");
```

```
    for(i=0; i<n; i++)
```

```
{
```

```
    printf("%d\t", a[i]);
```

```
}
```

```
}
```

Merging of any two existing list

This operation is used to append all the elements of one list to the end of another list. Here do not need to shift of elements.

List A	<table border="1" style="border-collapse: collapse; width: 100px;"> <tr><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>11</td><td>23</td><td>44</td><td>55</td><td>6</td><td>...</td><td>99</td></tr> <tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>...</td><td>99</td></tr> </table>	3	4	5	6	7	11	23	44	55	6	...	99	1	2	3	4	5	6	7	8	9	10	...	99
3	4	5	6	7	11	23	44	55	6	...	99														
1	2	3	4	5	6	7	8	9	10	...	99														

List B	<table border="1" style="border-collapse: collapse; width: 100px;"> <tr><td>87</td><td>67</td><td>40</td><td>...</td><td>99</td></tr> <tr><td>0</td><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>...</td><td>99</td></tr> </table>	87	67	40	...	99	0	1	2	3	4	5	6	7	8	9	10	...	99
87	67	40	...	99															
0	1	2	3	4	5	6	7	8	9	10	...	99							

Merging of given two lists into single list

List A	<table border="1" style="border-collapse: collapse; width: 100px;"> <tr><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>11</td><td>23</td><td>44</td><td>55</td><td>6</td><td>87</td><td>67</td><td>40</td><td>...</td><td>99</td></tr> <tr><td>1</td><td>2</td><td>3</td><td>4</td><td>5</td><td>6</td><td>7</td><td>8</td><td>9</td><td>10</td><td>11</td><td>12</td><td>13</td><td>...</td><td>99</td></tr> </table>	3	4	5	6	7	11	23	44	55	6	87	67	40	...	99	1	2	3	4	5	6	7	8	9	10	11	12	13	...	99
3	4	5	6	7	11	23	44	55	6	87	67	40	...	99																	
1	2	3	4	5	6	7	8	9	10	11	12	13	...	99																	

Algorithm

1. Start

2. For $i=0$ to size of first list minus one

Set all elements of first list to new list as

New_list[i]=first_list[i];

3. Set $j=\text{size of first list}$

4. Loop for $i=0$ to size of second list minus one

Set all elements of second list to new list as,

New_list[j]=second_list[i]

Increment j by one as, $j=j+1$

5. Increment size of new list by size of first list plus size of second list

6. Stop

C- Function to merge any two existing list into single common list

```
void merging(int a[100], int b[100], int n, int m)
```

```
{
    int i, j=n;
    printf("Enter element of second list");
    for(i=0; i<m; i++)
    {
        scanf("%d", &b[i]);
        a[j]=b[i];
        j++;
        n=n+1;
    }
}
```

Complete menu driven program in C to show the operations in list

```
#include<stdio.h>
#include<conio.h>
#include<process.h>
void create(int [100], int* );
void insert(int [100], int* );
void delet(int [100], int* );
void update(int [100], int * );
void traverse(int [100], int* );
void searching(int [100], int* );
void merging(int [100], int [100], int *, int *);
int main()
{
    int a[100], b[100], m, nel, pos, i;
    int n;
    int choice;
    printf("\n Manu for program:\n");
    printf("1>Create\n2:insert\n3:delete\n4:Update\n5:Traverse\n6:searching\n7:merging\n\n");
    do
    {
        printf("\n Enter your choice\n");
        scanf("%d", &choice);
        switch(choice)
        {
            case 1:
                printf("Enter no of elements of first list\n");
                scanf("%d", &n);
                create(a, &n);
                break;
            case 2:
                insert(a, &n);
                break;
        }
    }
}
```

```

        break;
    case 3:    let memory suffice out from given set of options
        delete(a, &n);
        break;
    case 4:    if all process to increase profit than
        update(a, &n);
        break;
    case 5:    if all profit < 0 then
        traverse(a, &n);
        break;
    case 6:    if all profit > 0 then
        searching(a, &n);
        break;
    case 7:    if all profit = 0 then
        printf("Enter size of second list");
        scanf("%d", &m);
        merging(a, b, &n, &m);
        break;
    default:
        printf("Invalid choice");
        while(choice<8):
            void create(int a[100], int *n)
            {
                int i;
                for(i=0;i<n;i++)
                    printf("Enter %d elements", *n);
                scanf("%d", &a[i]);
            }
            void insert(int a[100], int *n)
            {
                int pos, nele, i;
                printf("Enter position at which you want to insert new element");
                scanf("%d", &pos);
                printf("Enter new element");
                scanf("%d", &nele);
                for(i=*n-1; i>=pos; i--)
                    a[i+1] = a[i];
                a[pos] = nele;
                *n = *n + 1;
            }
}

```

```

void delete(int a[100], int *n)
{
    int pos, i;
    printf("Enter position at which you want to delete an element");
    scanf("%d", &pos);
    for(i=pos; i<*n; i++)
    {
        a[i] = a[i+1];
    }
    *n=*n-1;
}

void update(int a[100], int *n)
{
    int pos, num;
    printf("Enter position at which you want to update an element");
    scanf("%d", &pos);
    printf("Enter new element\n");
    scanf("%d", &num);
    a[pos]=num;
}

void traverse(int a[100], int *n)
{
    int i;
    printf("Current Elements of list are:\n");
    for(i=0;i<*n;i++)
    {
        printf("%d\t", a[i]);
    }
}

void searching(int a[100], int *n)
{
    int k,i;
    printf("Enter searched item");
    scanf("%d", &k);
    for(i=0;i<*n;i++)
    {
        if(k==a[i])
        {
            printf("Search Successful");
            break;
        }
    }
    if(i==*n)
    printf("Search Unsuccessful");
}

```

```

void merging(int a[100], int b[100], int *n, int *m)
{
    int i, j=*n;
    printf("Enter element of second list");
    for(i=0;i<*m;i++)
    {
        scanf("%d",&b[i]);
        a[j]=b[i];
        j++;
        *n=*n+1;
    }
}

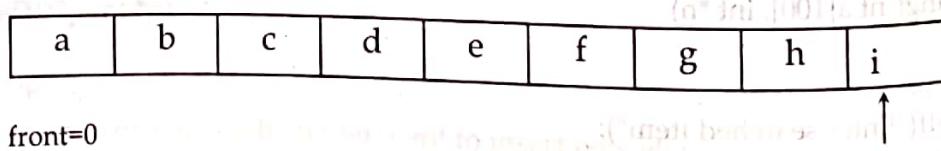
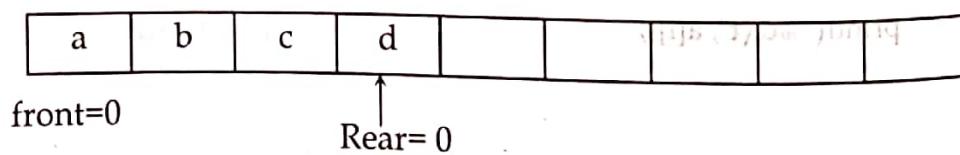
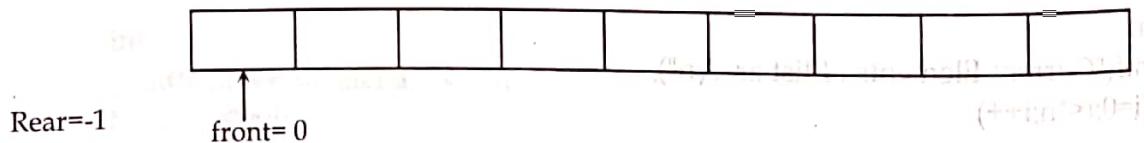
```

(n* int [001] is m) is also true

QUEUES AS A LIST

A Queue is a linear structure which follows a particular order in which the operations are performed. The order is First in first out (FIFO). A good example of a queue is any queue of consumers for a resource where the consumer that came first is served first. The difference between stacks and queues is in removing. In a stack we remove the item the most recently added; in a queue, we remove the item the least recently added.

In list representation of queue we insert element from rear part of list and delete element from front part and shift all elements one position left.



Deleting element from queue

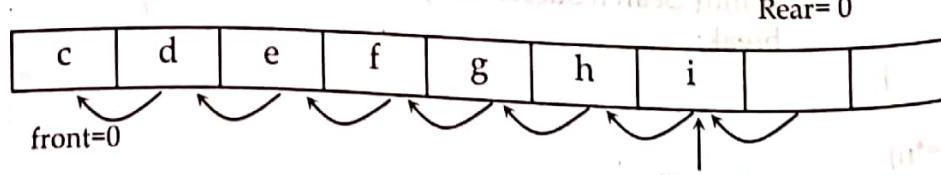
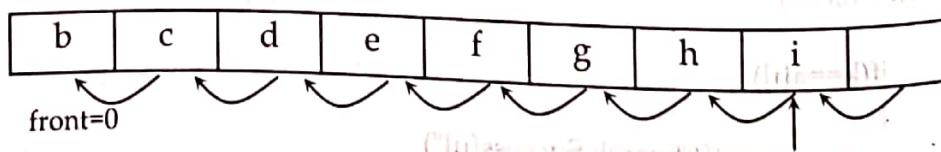


Fig: Inserting and deleting elements from queue by using list

MULTIPLE CHOICE QUESTIONS

1. A linear collection of data elements where the linear node is given by means of pointer is called?
a. Linked list b. Node list
c. Primitive list d. None of the above
2. The operation of processing each element in the list is known as
a. sorting b. merging
c. inserting d. traversal
3. The insertion of elements in list treated as
a. like inserting element in one dimensional array
b. like inserting element in stack
c. like inserting element in queue
d. all of the above
4. Deleting element from list treated as
a. shifting all elements one position left of list
b. shifting all elements one position right of list
c. do not needed shifting of elements
d. None of the above
5. Operation of list is.....
a. inserting element at specific position b. delete from one end
c. search element from list d. all of the above
6. Merging two list means....
a. appending one list to end of another list
b. inserting elements of one list to another list in any position
c. displaying elements of two lists d. None of the above
7. Traversing of a list means....
a. displaying largest element of list b. displaying all elements of list in order
c. sorting elements of given list d. finding sum of all elements of given list
8. Static list means....
a. allocate memory at compile time b. allocate memory statically
c. allocate memory in contiguous memory location
d. all of the above
9. Array implementation of list means.....
a. use of list in one dimensional array
b. need shifting of elements after deleting element from list
c. all of the above d. none of the above

10. After deleting element from list we need one of the flowing.....
- shifting of element one position right
 - shifting of element one position left
 - increasing size of list by one
 - searching largest element from list



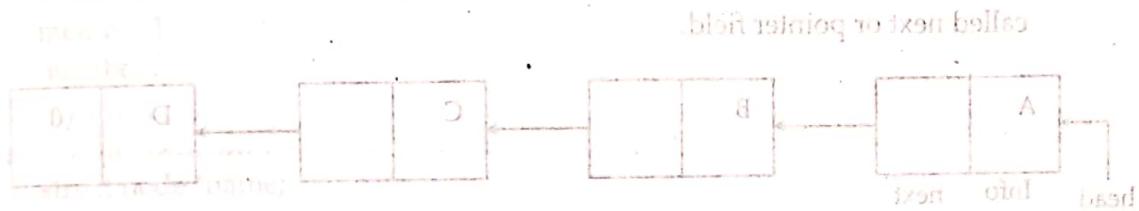
DISCUSSION EXERCISE

- What is list? How it is differ from array? Explain.
- What is static list? Explain.
- Explain dynamic list with suitable example.
- Write down advantages and disadvantages of static list.
- Write down advantages and disadvantages of dynamic data structure.
- How to insert new element to given list? Explain
- How to append one list to another list? Explain with suitable example.
- What do you mean by traversing of a list? Explain
- Write down the operations of list and explain any two of them
- Write complete program in C to simulate the insertion, deletion and traversing operation of list.
- What do you mean by queue as a list? Explain with suitable example.
- Show that list as an ADT.
- Differentiate between static and dynamic structure.
- Write down the algorithm for updating existing element of given list.
- What do you mean by update operation of list? Explain with suitable example.
- What are the requirements for deleting element from list? Explain
- How can you implement stack by using list? Explain.
- What is list? How it is differ from linked list? Explain.
- How do you find duplicate numbers in a list if it contains multiple duplicates?
- How do you find the largest and smallest number in an unsorted integer list?

5

A doubly linked list is a linked list where each node contains two pointers to the next and previous nodes.

The first pointer is called the forward pointer or next pointer and the second pointer is called the backward pointer or previous pointer. This allows for efficient insertion and deletion of nodes at any position in the list.



Example: Self-referential structure

struct node

LINKED LISTS

Quick Definition

A linked list is a linear data structure that consists of a sequence of elements, each containing data and a reference to the next element in the sequence. This reference is called a link. Unlike arrays, which store data in contiguous memory locations, linked lists store data in separate memory locations. Each node in a linked list contains a data field and a pointer to the next node. The last node's pointer is usually set to null. Insertion and deletion operations are faster than arrays because they do not require shifting elements.

Implementation: A linked list can be implemented using either static or dynamic memory allocation. In static memory allocation, the size of the list is fixed at compile time, and all nodes are allocated in contiguous memory blocks. In dynamic memory allocation, the size of the list is determined at runtime, and nodes are allocated individually as needed. Dynamic memory allocation is more flexible than static memory allocation, but it requires more overhead for managing memory.

CHAPTER OUTLINE

After studying this chapter, the reader will be able to understand the

- Introduction, Linked List as an ADT, Dynamic implementation, insertion and deletion of note to and from a list, insertion and deletion after and before nodes, linked stacks and queues, doubly linked list and its advantages.



INTRODUCTION

Arrays work well for unordered sequences and even for ordered sequences if they don't change much. But if we want to maintain an ordered list that allows quick insertions and deletions, we should use a linked data structure. The basic linked list consists of a collection of connected, dynamically allocated nodes.

LINKED LIST

To overcome the disadvantage of fixed size arrays linked list were introduced. A linked list consists of nodes of data which are connected with each other. Every node consists of two fields' data (info) and the link (next). The nodes are created dynamically.

- **Info:** the actual element to be stored in the list. It is also called data field.
- **Link:** one or two links that point to next and previous node in the list. It is also called next or pointer field.

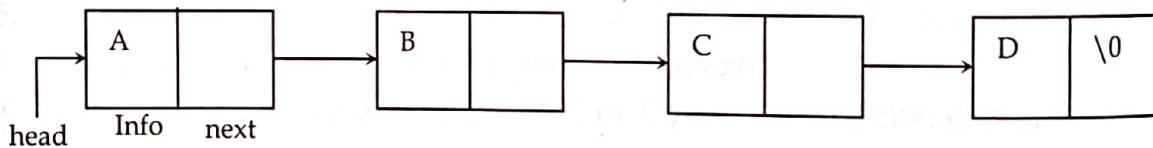


Fig: - Singly linked list with four nodes

What is the difference between Linked List and Linear Array?

S. No.	Array	Linked List
1.	Insertions and deletions are difficult.	Insertions and deletions can be done easily.
2.	It needs movements of elements for insertion and deletion	It does not need movement of nodes for insertion and deletion.
3.	In it space is wasted	In it space is not wasted.
4.	It is more expensive	It is less expensive.
5.	It requires less space as only information is stored	It requires more space as pointers are also stored along with information.
6.	Its size is fixed	Its size is not fixed.
7.	It cannot be extended or reduced according to requirements	It can be extended or reduced according to requirements.
8.	Same amount of time is required to access each element	Different amount of time is required to access each element.
9.	Elements are stored in consecutive memory locations.	Elements may or may not be stored in consecutive memory locations.
10.	If have to go to a particular element then we can reach their directly.	If we have to go to a particular node then we have to go through all those nodes that come before that node.

Uses of Linked List

- The list is not required to be contiguously present in the memory. The node can reside anywhere in the memory and linked together to make a list. This achieves optimized utilization of space.
- List size is limited to the memory size and doesn't need to be declared in advance.
- Empty node cannot be present in the linked list.
- We can store values of primitive types or objects in the singly linked list.

Self Referential Structure

It is sometimes desirable to include within a structure one member that is a pointer to the parent structure type. Hence, a structure which contains a reference to itself is called self-referential structure. In general terms, this can be expressed as:

```
struct node {  
    member 1;  
    member 2;  
    .....  
    struct node *name;
```

Example: Self referential structure

```
struct node {  
    int info;  
    struct node *next;
```

This is a structure of type node. The structure contains two members: an info integer member, and a pointer to a structure of the same type (i.e., a pointer to a structure of type node), called next. Therefore this is a **self-referential structure**.

Header Nodes

A header node is an extra node in a linked list that holds no data but serves to satisfy the requirement that every node containing an item have a previous node in the list. By using the header node, we greatly simplify the code—with a negligible space penalty. In more complex applications, header nodes not only simplify the code but also improve speed because, after all, fewer tests mean less time.

The use of a header node is somewhat controversial. Some argue that avoiding special cases is not sufficient justification for adding fictitious cells; they view the use of header nodes as little more than old-style hacking. Even so, we use them here precisely because they allow us to demonstrate the basic link manipulations without obscuring the code with special cases.

Whether a header should be used is a matter of personal preference. Furthermore, in a class implementation, its use would be completely transparent to the user. However, we must be careful: The printing routine must skip over the header node, as must all searching routines. Moving to the front now means setting the current position to `header.next`, and so on.

Points to be noted for linked list

- The nodes in a linked list are not stored contiguously in the memory
- You don't have to shift any element in the list.
- Memory for each node can be allocated dynamically whenever the need arises.
- The size of a linked list can grow or shrink dynamically

Advantage of Link list

- Link list is an example of dynamic data structure. They can grow and shrink during the execution of program.
- Efficient memory utilization. Memory is not pre allocated like static data structure. The allocation of memory depends upon the user ,i.e. no need to pre-allocate memory
- Insertion and deletion easily performed.
- Linear Data Structures such as Stack, Queue can be easily implemented using Linked list
- Faster Access time, can be expanded in constant time without memory overhead

Types of Linked Lists

There are mainly four common types of Linked List:

1. Single linked list
2. Double linked list
3. Circular linked list
4. Circular doubly linked list

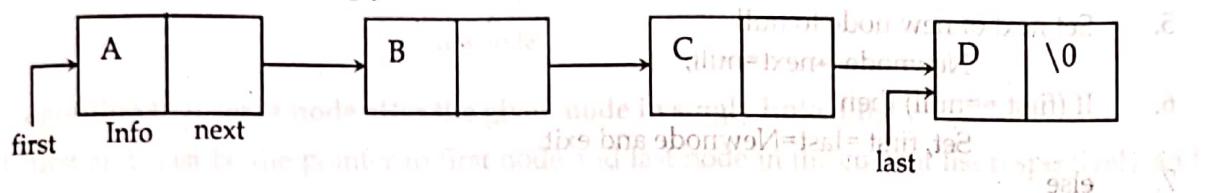
Valid operations on linked list

1. Inserting elements to linked list
 - Inserting an element at first position
 - Inserting an element at end
 - Inserting an element at specified position
 - Inserting an element after given node
2. Deleting elements from linked list
 - Deleting an element at first position
 - Deleting an element at end
 - Deleting a node after given node
 - Deleting specific order node
3. Searching a node
4. Merging linked lists

Singly Linked list

A singly linked list is a dynamic data structure which may grow or shrink, and growing and shrinking depends on the operation made. In this type of linked list each node contains two fields one is info field which is used to store the data items and another is link field that is used to point the next node in the list. The last node has a NULL pointer. The address of first node can be contained by an external pointer called 'first'.

The following example is a singly linked list that contains four elements A, B, C and D.



Structure of a node of singly linked list

We can define a node as follows:

```
struct Node
{
    int info;
    struct Node *next;
};
```

typedef struct Node NodeType;

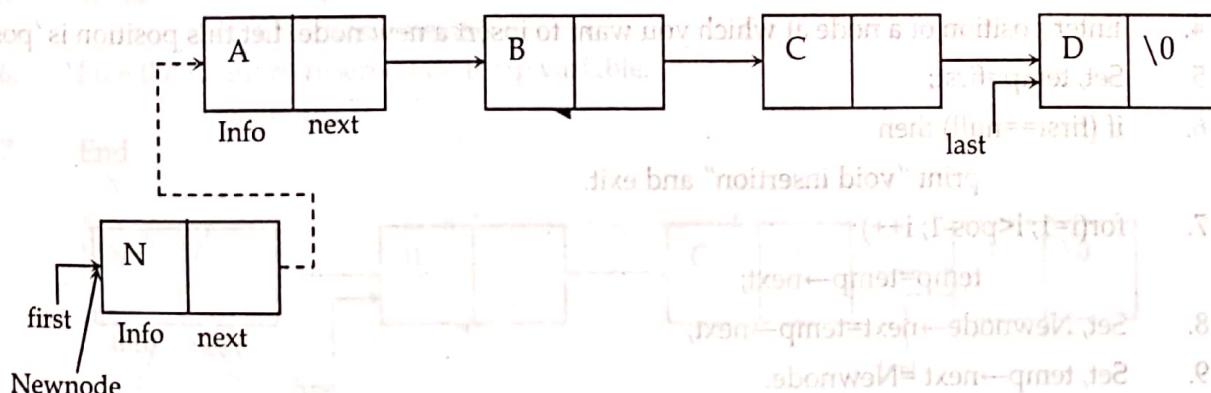
NodeType *first; //first is a pointer type structure variable that points to first node

NodeType *last; //last is a pointer type structure variable that points to last node

An algorithm to insert a node at the beginning of the singly linked list

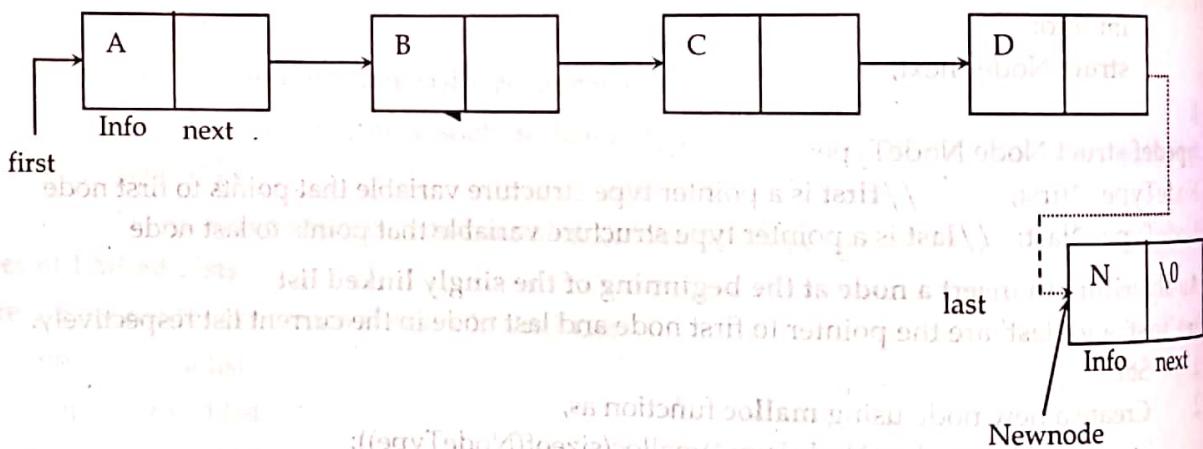
Let 'first' and 'last' are the pointer to first node and last node in the current list respectively.

1. Start
2. Create a new node using malloc function as,
 Newnode=(NodeType*)malloc(sizeof(NodeType));
3. Read data item to be inserted say 'el'
4. Assign data to the info field of new node
 Newnode.info=el;
5. Set next of new node to first
 Newnode.next=first;
6. Set the first pointer to the new node
 first = Newnode;
7. End



An algorithm to insert a node at the end of the linked list:

1. Start
2. Create a new node using malloc function as,
 $\text{Newnode} = (\text{NodeType}^*) \text{malloc}(\text{sizeof}(\text{NodeType}))$;
3. Read data item to be inserted say 'el'
4. Assign data item to the info field of new node
 $\text{Newnode} \rightarrow \text{info} = \text{el}$;
5. Set next of new node to null
 $\text{Newnode} \rightarrow \text{next} = \text{null}$;
6. If ($\text{first} == \text{null}$) then
Set, $\text{first} = \text{last} = \text{Newnode}$ and exit.
7. else
Set, $\text{last} \rightarrow \text{next} = \text{newnode}$;
Set, $\text{last} = \text{newnode}$;
8. End

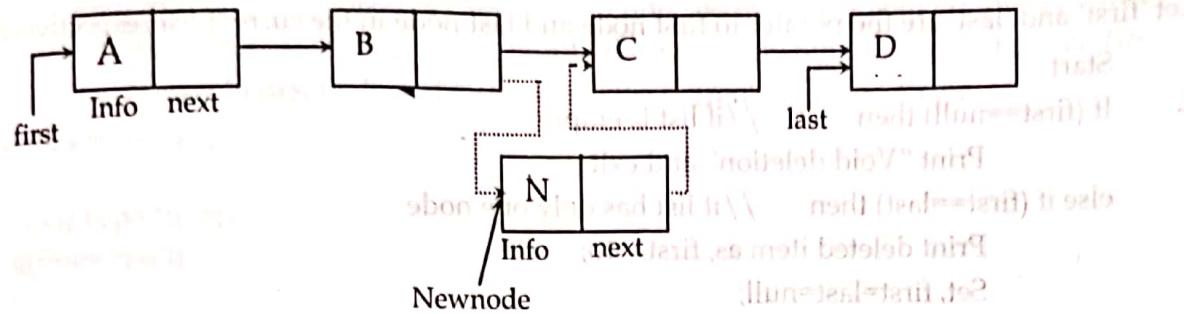


An algorithm to insert a node at the specified position in a singly linked list

Let 'first' and 'last' are the pointer to first node and last node in the current list respectively.

1. Start
2. Create a new node using malloc function as,
 $\text{Newnode} = (\text{NodeType}^*) \text{malloc}(\text{sizeof}(\text{NodeType}))$;
3. Assign data to the info field of new node
 $\text{Newnode} \rightarrow \text{info} = \text{el}$;
4. Enter position of a node at which you want to insert a new node. Let this position is 'pos'.
5. Set, $\text{temp} = \text{first}$;
6. if ($\text{first} == \text{null}$) then
print "void insertion" and exit.
7. for($i=1; i < \text{pos}-1; i++$)
 $\text{temp} = \text{temp} \rightarrow \text{next}$;
8. Set, $\text{Newnode} \rightarrow \text{next} = \text{temp} \rightarrow \text{next}$;
9. Set, $\text{temp} \rightarrow \text{next} = \text{Newnode}$.

10. End

**An algorithm to insert a node after the given node in singly linked list**

Let *first and *last be the pointer to first node and last node in the current list respectively and *p be the pointer to the node after which we want to insert a new node.

1. Start
2. Create a new node using malloc function
 - a. Newnode=(NodeType*)malloc(sizeof(NodeType));
3. Read data item to be inserted say 'item'
4. Assign data to the info field of new node as,
 - a. Newnode->info=item;
5. Set next of new node to next of p as,
 - a. Newnode->next=p->next;
6. Set next of p to Newnode
 - a. p->next=Newnode
7. End

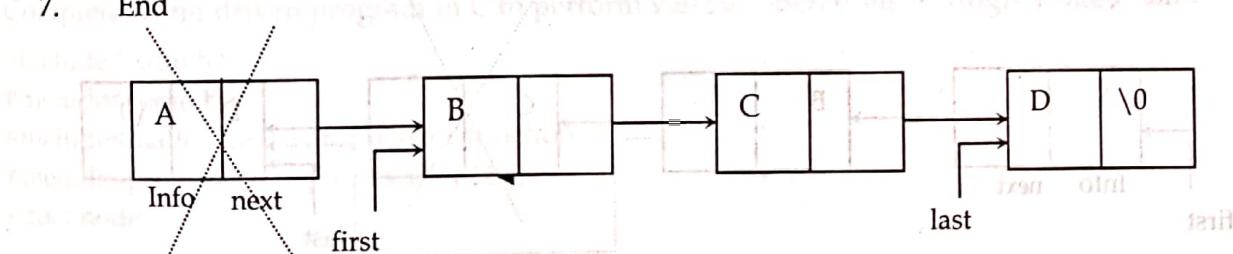
An algorithm to deleting the first node of the singly linked list

Let 'first' and 'last' are the pointer to first node and last node in the current list respectively.

1. Start
2. If(first==null) then

Print "Void deletion" and exit
3. Else if (first==last)

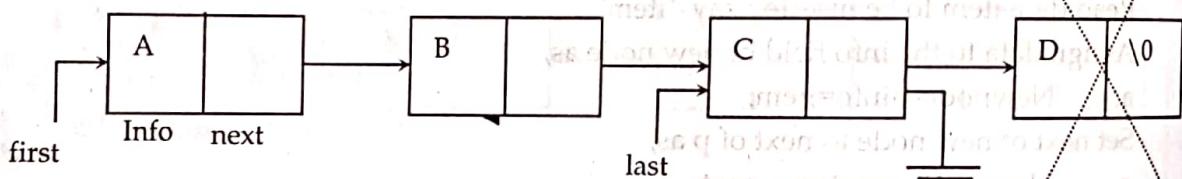
Print deleted item as, first.info;
first=last=null
4. Store the address of first node in a temporary variable temp.
Set, temp=first;
5. Set first to next of first.
Set, first=first->next;
6. Free the memory reserved by temp variable.
free(temp);
7. End



An algorithm to deleting the last node of the singly linked list:

Let 'first' and 'last' are the pointer to first node and last node in the current list respectively.

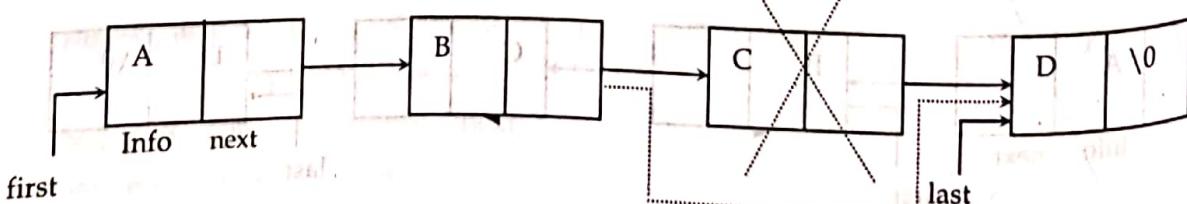
1. Start
2. If (first==null) then //if list is empty
Print "Void deletion" and exit
3. else if (first==last) then //if list has only one node
Print deleted item as, first.info;
Set, first=last=null;
4. else
temp=first;
while(temp->next!=last)
Set, temp=temp->next;
Set, temp->next=null;
Set, last=temp;
5. End



An algorithm to delete a node at the specified position in a singly linked list

Let 'first' and 'last' are the pointer to first node and last node in the current list respectively.

1. Start
2. Read position of a node which to be deleted let it be 'pos'.
3. if first==null then
Print "void deletion" and exit
Otherwise,
4. Enter position of a node at which you want to delete a new node. Let this position is 'pos'.
5. Set, temp=first
6. for(i=1; i<pos-1; i++)
Set, temp=temp->next;
7. Print deleted item is temp.next.info
8. Set, loc=temp->next;
9. Set, temp->next = loc->next;
10. End



Searching an item in a linked list

To search an item from a given linked list we need to find the node that contain this data item. If we find such a node, then searching is successful otherwise searching unsuccessful. Let 'first' and 'last' are the pointer to first node and last node in the current list respectively.

```
void searchItem(int key)
{
    NodeType *temp;
    if(first==NULL)
    {
        printf("empty linked list");
        exit(0);
    }
    else
    {
        temp=first;
        while(temp!=NULL)
        {
            if(temp->info==key)
            {
                printf("Search successful");
                break;
            }
            temp=temp->next;
        }
        if(temp==NULL)
            printf("Unsuccessful search");
    }
}
```

An algorithm to delete a node after the given node in singly linked list

Let *first and *last be the pointer to first node and last node in the current list and *p be the pointer to the node after which we want to delete a new node.

1. Start
2. If ($p==NULL$ or $p->next==NULL$) then
 - print "deletion not possible and exit"
3. Set loc= $p->next$
4. Set $p->next=loc->next;$
5. Free (loc)
6. End

Complete menu driven program in C to perform various operations in singly linked list

```
#include<stdio.h>
#include<conio.h>
#include<malloc.h> //for malloc function
#include<process.h> //for exit function
struct node
```

```

{
    int info;
    struct node *next;
};

typedef struct node NodeType;
NodeType *first, *last;
first=last=NULL;
void insert_atfirst(int);
void insert_givenposition(int);
void insert_atend(int);
void delet_first();
void delet_last();
void delet_nthnode();
void info_sum();
void count_nodes();
void display();
void main()
{
    int choice;
    int item;
    clrscr();
    do
    {
        printf("\n manu for program:\n");
        printf("1. insert first \n2.insert at given position \n3 insert at last \n4: Delete
first node\n 5: delete last node\n6: delete nth node\n7: info sum\n8: count
nodes\n9: Display items\n10:exit\n");
        printf("enter your choice\n");
        scanf("%d", &choice);
        switch(choice)
        {
            case 1:
                printf("Enter item to be inserted");
                scanf("%d", &item);
                insert_atfirst(item);
                break;
            case 2:
                printf("Enter item to be inserted");
                scanf("%d", &item);
                insert_givenposition(item);
                break;
            case 3:
                printf("Enter item to be inserted");
                scanf("%d", &item);
                insert_atend();
                break;
        }
    } while(choice!=10);
}

```

```

        case 4: delete_first();
        break;
    case 5: delete_last();
        break;
    case 6: delete_nthnode();
        break;
    case 7: info_sum();
        break;
    case 8: count_nodes();
        break;
    case 9: display();
        break;
    case 10: exit(1);
        break;
    default: printf("invalid choice\n");
    }
}
while(choice<10);
getch();
}
void insert_atfirst(int item)
{
    NodeType *Newnode;
    Newnode=(NodeType*)malloc(sizeof(NodeType));
    if(first==null)
    {
        Newnode->next=null;
        first=Newnode;
        last=Newnode;
    }
    else
    {
        Newnode->next=first;
        first=Newnode;
    }
}
void insert_givenposition(int item)
{
    int pos, i;
    NodeType *Newnode, *temp;
    newnode->info=item;
}

```

```

printf(" Enter position of a node at which you want to insert a new node");
scanf("%d", &pos);
if(first==NULL)
{
    first=newnode;
    last=newnode;
}
else
{
    temp=first;
    for(i=1; i<pos-1; i++)
    {
        temp=temp->next;
    }
    Newnode->next=temp->next;
    temp->next =Newnode;
}
void insert_atend(int item)
{
    NodeType *NewNode;
    Newnode=(NodeType*)malloc(sizeof(NodeType));
    Newnode->info=item;
    Newnode->next=NULL;
    if(first==NULL)
    {
        first=newnode;
        last=newnode;
    }
    else
    {
        last->next=newnode;
        last=newnode;
    }
}
void delet_first()
{
    NodeType *temp;
    if(head==NULL)
    {
        printf("Void deletion | n");
        return;
    }
    else
    {
        temp=head;
        head=head->next;
    }
}

```

```

        free(temp);
    }

    void delet_last()
    {
        NodeType *hold,*temp;
        if(head==NULL)
        {
            printf("Void deletion\n");
            return;
        }
        else if(head->next==NULL)
        {
            hold=head;
            head=NULL;
            free(hold);
        }
        else
        {
            temp=head;
            while(temp->next->next!=NULL)
                temp=temp->next;
            hold=temp->next;
            temp->next=NULL;
            free(hold);
        }
    }

    void delet_nthnode()
    {
        NodeType *hold,*temp;
        int pos, i;
        if(first==NULL)
        {
            printf("Void deletion\n");
            return;
        }
        else
        {
            temp=first;
            printf("Enter position of node which node is to be deleted\n");
            scanf("%d",&pos);
            for(i=1;i<pos-1;i++)
                temp=temp->next;
            hold=temp->next;
            temp->next=hold->next;
            free(hold);
        }
    }
}

```

```

void info_sum()
{
    NodeType *temp;
    temp=first;
    while(temp!=NULL)
    {
        printf("%d\t", temp->info);
        temp=temp->next;
    }
}

void count_nodes()
{
    int cnt=0;
    NodeType *temp;
    temp=first;
    while(temp!=NULL)
    {
        cnt++;
        temp=temp->next;
    }
    printf("total nodes=%d", cnt);
}

void display()
{
    NodeType *temp;
    temp=first;
    if(first==null)
    {
        printf("Empty linked list");
        exit(1);
    }
    else
    {
        while(temp!=NULL)
        {
            printf("%d\t", temp->info);
            temp=temp->next;
        }
    }
}

```

Output

Menu for program

1. insert first
2. insert at given position
3. insert at last
4. Delete first node

5: delete last node
 6: delete nth node
 7: info sum
 8: count nodes
 9: Display items
 10: exit
 Enter your choice

1
 Enter data item to be inserted

43
 Enter your choice

1
 Enter data item to be inserted

12
 Enter your choice

1
 Enter data item to be inserted

88
 Enter your choice

9
 88 12 43 Enter your choice

4
 12 43 Enter your choice

10

Advantages of singly linked list

- Insertions and Deletions can be done easily.
- It does not need movement of elements for insertion and deletion.
- Its space is not wasted as we can get space according to our requirements.
- Its size is not fixed.
- It can be extended or reduced according to requirements.
- Elements may or may not be stored in consecutive memory available; even then we can store the data in computer.
- It is less expensive.

Disadvantages of singly linked list

- It requires more space as pointers are also stored with information.
- Different amount of time is required to access each element.
- If we have to go to a particular element then we have to go through all those elements that come before that element.
- We cannot traverse it from last & only from the beginning.
- It is not easy to sort the elements stored in the linear linked list.

The Linked list act as ADT

A linked list of n-nodes with n-elements of type T is a sequence of elements of T together with the operations:

- **Create()**: Create or make a node
- **Insert (x)**: Insert x to linked list
- **Delete ()**: If linked list is not empty then delete given node.

- **Traverse ()**: Display all of the nodes of given linked list.
- **IsEmpty()**: Determine whether linked list is empty or not. Return **true** if it is empty; return **false** otherwise.
- **Find () or search()**: Find out given node from linked list
- **Count()**: Count number of nodes of given linked list
- **Free ()**: release memory space of given node of linked list.

Circular Linked list

A circular linked list is a list where the link field of last node points to the first node of the list. Circular linked lists can be used to help the traverse the same list again and again if needed. A circular list is very similar to the linear list where in the circular list the pointer of the last node points not NULL but the first node. The main advantage of circular linked list is that it requires minimum time to traverse the nodes which are already traversed, without moving to starting node.

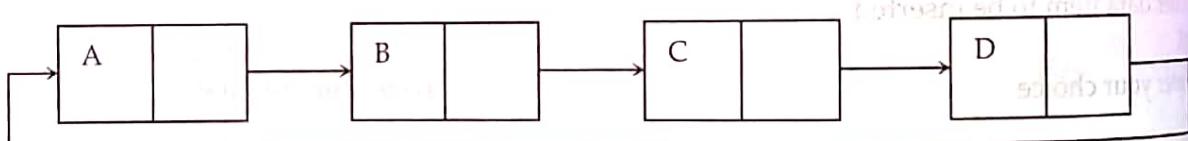


Fig: Circular linked list with four nodes

Advantages and disadvantages of circular linked list

Advantages

1. If we are at a node, then we can go to any node. But in linear linked list it is not possible to go to previous node
2. It saves time when we have to go to the first node from the last node. It can be done in single step because there is no need to traverse the in-between nodes. But in double linked list, we will have to go through in between nodes

Disadvantages

1. It is not easy to reverse the linked list
2. If proper care is not taken, then the problem of infinite loop can occur
3. If we are at a node and go back to the previous node, then we cannot do it in single step.

Representation of circular linked list

We declare the structure for the circular linked list in the same way as declared it for the singly linked list.

```
struct Node
{
    int info;
    struct Node *next;
};

typedef struct Node NodeType;
NodeType *first;
NodeType *last;
```

Algorithm to insert a node at the beginning of a circular linked list

1. Create a new node by using malloc function as,

```
Newnode=(NodeType*)malloc(sizeof(NodeType));
```

2. Read data item to be inserted say it be 'el'

3. Set Newnode→info=el

4. if first==null then

```
Set, Newnode→next=Newnode
```

```
Set, first=Newnode
```

```
Set, last =Newnode
```

5. else

```
Set, Newnode→next=start
```

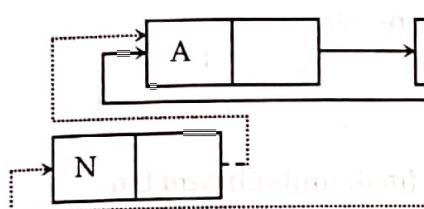
```
Set, first=Newnode
```

```
Set, last→next=Newnode
```

6. End



(last=last→next=Newnode) (last=first=Newnode) (last=last→next=Newnode) (last=last→next=Newnode)

**Algorithm to insert a node at the end of a circular linked list**

1. Create a new node by using malloc function as,

```
Newnode=(NodeType*)malloc(sizeof(NodeType));
```

2. Read data item to be inserted say it be 'el'

3. Set Newnode→info=el

4. if start==null then

```
Set, Newnode→next=Newnode
```

```
Set, start=Newnode
```

```
Set, last=Newnode
```

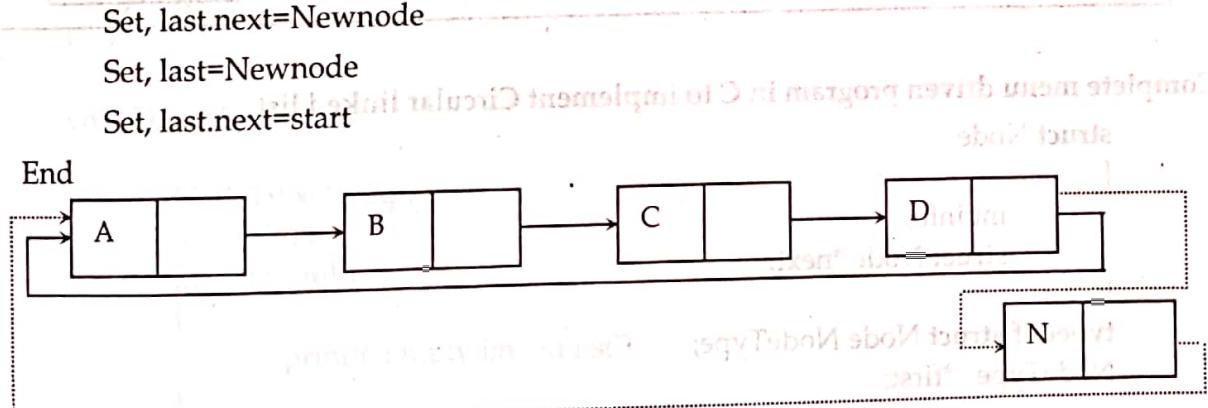
5. else

```
Set, last.next=Newnode
```

```
Set, last=Newnode
```

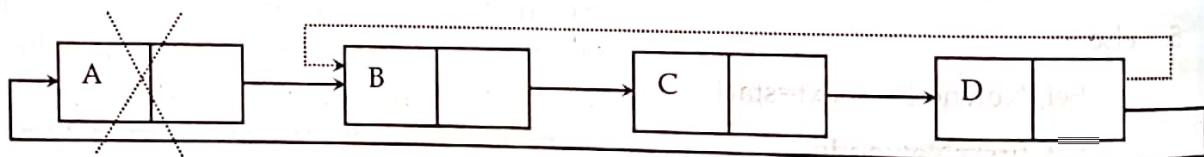
```
Set, last.next=start
```

6. End



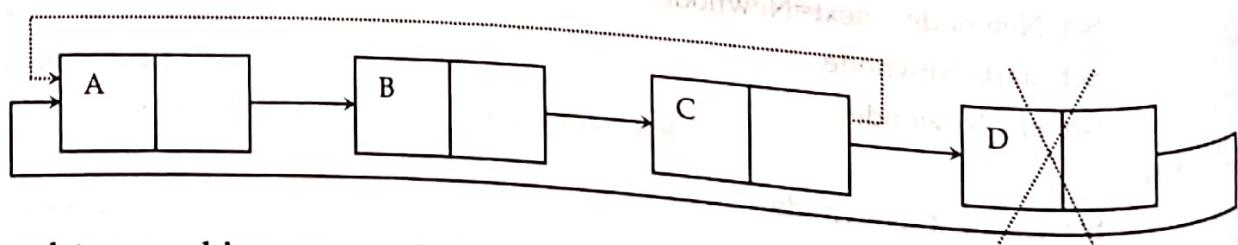
Algorithm to delete a node from the beginning of a circular linked list

1. Start
2. if first==null then
Print "Empty list" and exit
3. else
Print the deleted element=first→info
Set, first=first→next
Set, last→next=first;
4. Stop



Algorithm to delete a node from the end of a circular linked list

1. Start
2. if first==null then
Print "empty list" and exit
3. else if first==last
 - Print deleted element=first.info
 - Set, first=last=null
4. else
 - Set, temp=first
 - while(temp→next!=last)
Set, temp=temp→next
 - End while
 - Print the deleted element=last→info
 - Set, last=temp
 - Set, last→next=first
5. Stop



Complete menu driven program in C to implement Circular linked list

```

struct Node
{
    int info;
    struct Node *next;
};

typedef struct Node NodeType;
NodeType *first;
  
```

```

NodeType *last;
first=null;
last=null;

void insertbeg(int item)
{
    Newnode=(NodeType*)malloc(sizeof(NodeType));
    Newnode->info=item;
    if(first==null)
    {
        Newnode->next=Newnode;
        first=Newnode;
        last=Newnode;
    }
    else
    {
        Newnode->next=first;
        first=Newnode;
        last->next=Newnode;
    }
    switch(choice)
}

void insertEnd(int item)
{
    Newnode=(NodeType*)malloc(sizeof(NodeType));
    Newnode->info=item;
    if(first==null)
    {
        first=Newnode;
        last=Newnode;
        Newnode->next=Newnode;
    }
    else
    {
        last->next=Newnode;
        last=Newnode;
        Newnode->next=first;
    }
}

void DeleteFirst()
{
    NodeType *temp;
    temp=first;
    if(first==null)
    {
        printf("Empty linked list");
    }
    else
    {
        if(first==last)
        {
            free(first);
            first=null;
            last=null;
        }
        else
        {
            temp=first;
            first=first->next;
            free(temp);
        }
    }
}

```

```

        else if(first==last)
        {
            first=null;
            last=null;
            free(temp);
        }
        else
        {
            first=first->next;
            last->next=first;
            free(temp);
        }
    }
void DeleteLast()
{
    NodeType *temp;
    temp=last;
    if(last==null)
    {
        printf("Empty linked list");
    }
    else if(first==last)
    {
        first=null;
        last=null;
        free(temp);
    }
    else
    {
        while(temp->next!=last)
        {
            temp=temp->next;
        }
        temp->next=first;
        last=temp;
        temp=first;
        free(temp);
    }
}
void Display()
{
    NodeType *temp;
    if(first==null)
    {
        printf("Empty linked list");
    }
    else
    {
        ("list contains elements")
    }
}

```

```

temp=first;
while(temp!=last)
{
    printf(temp->info);
    temp=temp->next;
}
printf(last->info);
}

void main()
{
    int choice;
    int item;
    printf("1:Insert at beginning");
    printf("2:Insert at last");
    printf("3:delete first node");
    printf("4:delete last node");
    printf("5:Display");
    do
    {
        printf("Enter your choice");
        scanf("%d", &choice);
        switch(choice)
        {
            case 1:
                printf("Enter data item to be inserted");
                scanf("%d", &item);
                insertbeg(item);
                break;
            case 2:
                printf("Enter data item to be inserted");
                scanf("%d", &item);
                insertEnd(item);
                break;
            case 3:
                DeleteFirst();
                break;
            case 4:
                DeleteLast();
                break;
            case 5:
                Display();
                break;
            default:
                printf("Invalid choice Please enter correct choice");
        }
    }while(choice<6);
}

```

Fig: Double linked list deletion

Output

```

1: Insert at beginning
2: Insert at last
3: delete first node
4: delete last node
5: Display
Enter your choice
1
Enter data item to be inserted
22
Enter your choice
1
Enter data item to be inserted
87
Enter your choice
1
Enter data item to be inserted
32
Enter your choice
5
32 87 22 Enter your choice
2
Enter data item to be inserted
908
Enter your choice
5
32 87 22 908 Enter your choice
3
Enter your choice
5
87 22 908 Enter your choice
3
Enter your choice

```

Doubly Linked List (DLL)

A linked list in which all nodes are linked together by multiple numbers of links i.e. each node contains three fields (two pointer fields and one data field) rather is called doubly linked list. It provides bidirectional traversal. Doubly circular linked list can be conceptualized as two singly linked lists formed from the same data items, but in opposite sequential orders.

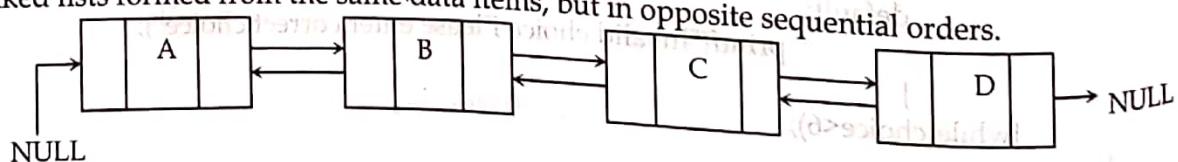


Fig: Doubly linked list with three nodes

The above diagram represents the basic structure of Doubly Circular Linked List. In doubly circular linked list, the previous link of the first node points to the last node and the next link of the last node points to the first node. In doubly circular linked list, each node contains two fields called links used to represent references to the previous and the next node in the sequence of nodes.

Advantages and disadvantages of doubly linked list

Advantages

1. We can traverse in both directions i.e. from starting to end and as well as from end to starting.
2. It is easy to reverse the linked list.
3. If we are at a node, then we can go to any node. But in linear linked list, it is not possible to reach the previous node.

Disadvantages

1. It requires more space per node because one extra field is required for pointer to previous node.
2. Insertion and deletion take more time than linear linked list because more pointer operations are required than linear linked list.

Representation of doubly linked list

```
struct node
{
    int info;
    struct node *prev;
    struct node *next;
};

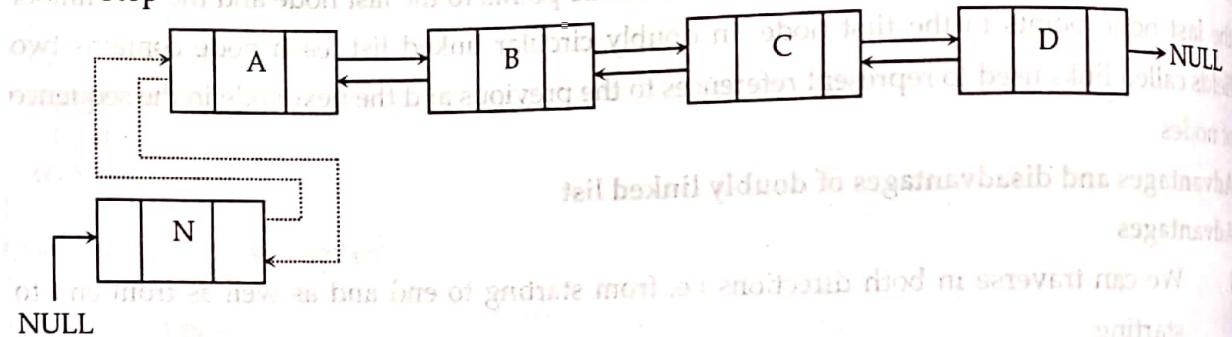
typedef struct node NodeType;
NodeType *first=NULL;
NodeType *last=NULL;
```

Algorithm to insert a node at the beginning of a doubly linked list

1. Start
 2. Create a new node by using malloc function as,
- ```
Newnode=(NodeType*)malloc(sizeof(NodeType))
```
3. Read data item to be inserted say it be 'el'
  4. Set Newnode→info=el
  5. Set Newnode→prev=Newnode→next=null
  6. If first==null then
    - Set first=last=Newnode
    - Otherwise,
  7. Set Newnode→next=first
  8. Set first→prev=Newnode

9. Set first=Newnode

10. Stop



### Algorithm to insert a node at the end of a doubly linked list

1. Start

2. Create a new node by using malloc function as,

Newnode=(NodeType\*)malloc(sizeof(NodeType))

3. Read data item to be inserted say it be 'el'

4. Set Newnode→info=el

5. Set Newnode→next=NULL

6. If first==NULL then

Set, first=last=Newnode

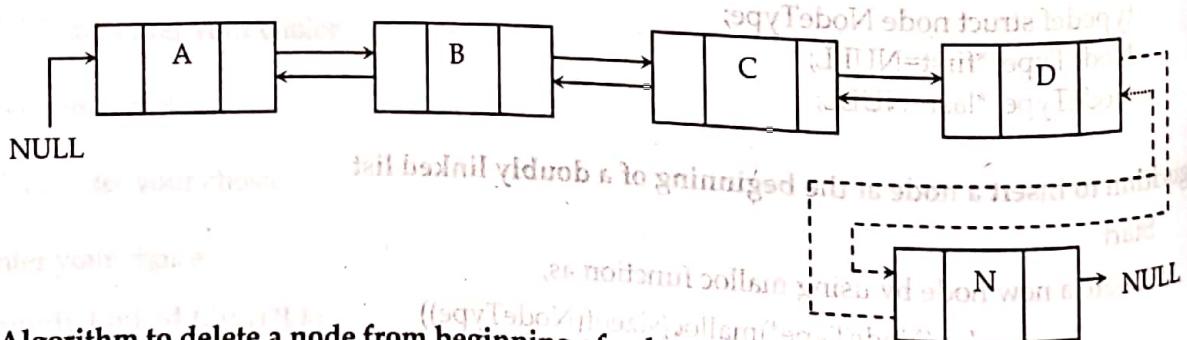
Otherwise,

7. Set last→next=Newnode

8. Set Newnode→prev=last

9. Set last=Newnode

10. stop



### Algorithm to delete a node from beginning of a doubly linked list

1. Start

2. if first==NULL then

Print "empty list" and exit

3. else

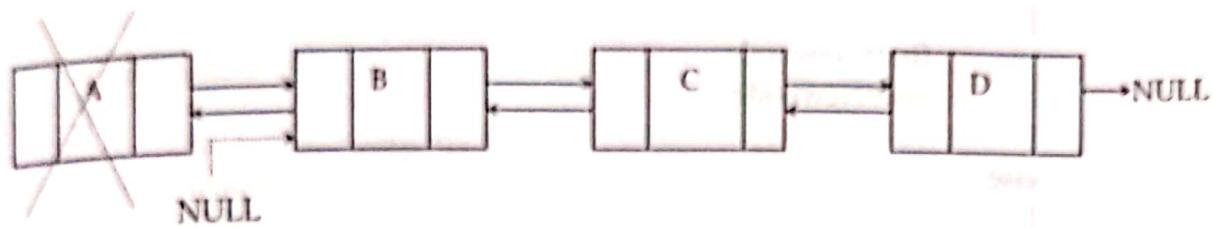
Set, temp=first

Set, first=first→next

Set, first→prev=null

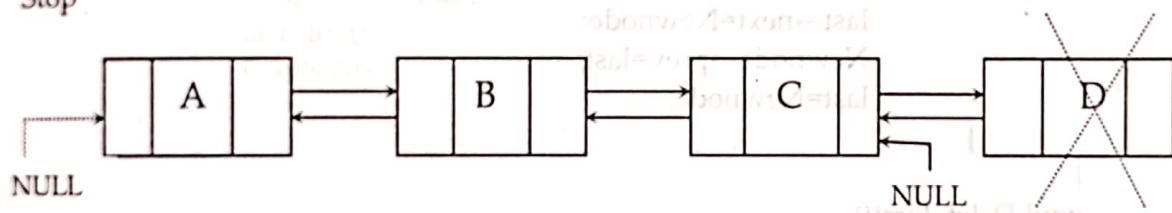
Free(temp)

4. Stop



### Algorithm to delete a node from end of a doubly linked list

1. Start
2. if first==NULL then  
    Print "empty list" and exit
3. else if(first==last) then  
    Set, first=last=NULL
4. else  
    Set, temp=first;  
    while(temp→next!=last)  
        temp=temp→next  
    end while  
    Set temp→next=null  
    Set, last=temp
5. Stop



### Complete menu driven program in C to implement doubly linked list

```

struct node
{
 int info;
 struct node *prev;
 struct node *next;
};

typedef struct node NodeType;
NodeType *first=NULL;
NodeType *last=NULL;
void insertbeg(int el)
{
 NodeType *Newnode;
 Newnode=(NodeType*)malloc(sizeof(NodeType));
 Newnode→info=el;
}

```

```

 }
}

void main()
{
 int choice;
 int item;
 printf("1:Insert at beginning");
 printf("2:Insert at last");
 printf("3:delete first node");
 printf("4:delete last node");
 printf("5:Display");
 do
 {
 printf("Enter your choice");
 scanf("%d", &choice);
 switch(choice)
 {
 case 1:
 printf("Enter data item to be inserted");
 scanf("%d", &item);
 insertbeg(item);
 break;
 case 2:
 printf("Enter data item to be inserted");
 scanf("%d", &item);
 insertEnd(item);
 break;
 case 3:
 DeleteFirst();
 break;
 case 4:
 DeleteLast();
 break;
 case 5:
 Display();
 break;
 default:
 printf("Invalid choice Plz enter correct choice");
 }
 }while(choice<6);
}

```

**Output**

- 1: Insert at beginning
- 2: Insert at last
- 3: delete first node
- 4: delete last node
- 5: Display

Enter your choice

1 Enter data item to be inserted

33 Enter your choice

2 Enter data item to be inserted

77 Enter your choice

1 Enter data item to be inserted

98 Enter your choice

5 33 77 Enter your choice

2 Enter data item to be inserted

54 Enter your choice

98 33 77 54 Enter your choice

2 Enter data item to be inserted

5 Enter your choice

98 33 77 54 Enter your choice

2 Enter data item to be inserted

5 Enter your choice

98 33 77 54 5 Enter your choice

3 Enter your choice

Enter your choice

### Circular Doubly Linked List

A circular doubly linked list is one which has the successor and predecessor pointer in circular manner. It is a doubly linked list where the next link of last node points to the first node and previous link of first node points to last node of the list. The main objective of considering circular doubly linked list is to simplify the insertion and deletion operations performed on doubly linked list.

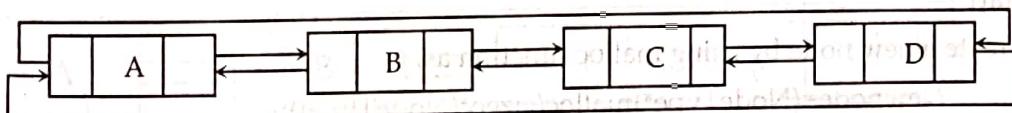


Fig: Circular doubly linked list with four nodes

Representation of circular doubly linked list

```
struct node
{
 int info;
 struct node *prev;
 struct node *next;
};

typedef struct node NodeType;
NodeType *first=NULL;
NodeType *last=NULL;
```

**Algorithm to insert a node at the beginning of a circular doubly linked list**

1. Start
2. Create a new node by using malloc function as,

Newnode=(NodeType\*)malloc(sizeof(NodeType));

3. Read data item to be inserted say it be 'el'
4. Set Newnode→info=el
5. If first==null then

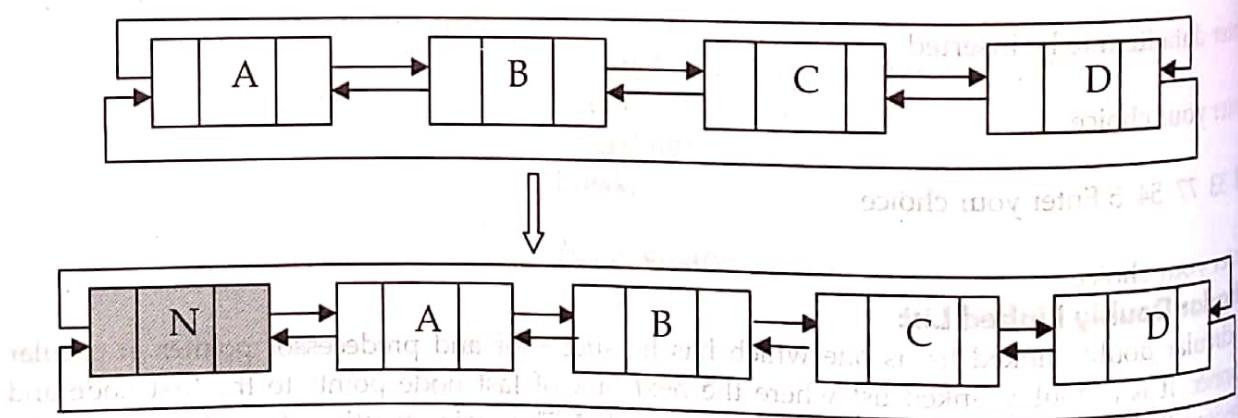
Set, first=last=Newnode

Set, Newnode→next=Newnode

Set, Newnode→prev=Newnode

Otherwise,

6. Set Newnode→next=first
7. Set, first→prev=Newnode
8. Set first=Newnode
9. Set last→next=first
10. Set first→prev=last
11. Stop

**Algorithm to insert a node at the end of a circular doubly linked list**

1. Start
2. Create a new node by using malloc function as,

Newnode=(NodeType\*)malloc(sizeof(NodeType));

3. Read data item to be inserted say it be 'el'
4. Set Newnode→info=el
5. If first==null then

Set, first=last=Newnode

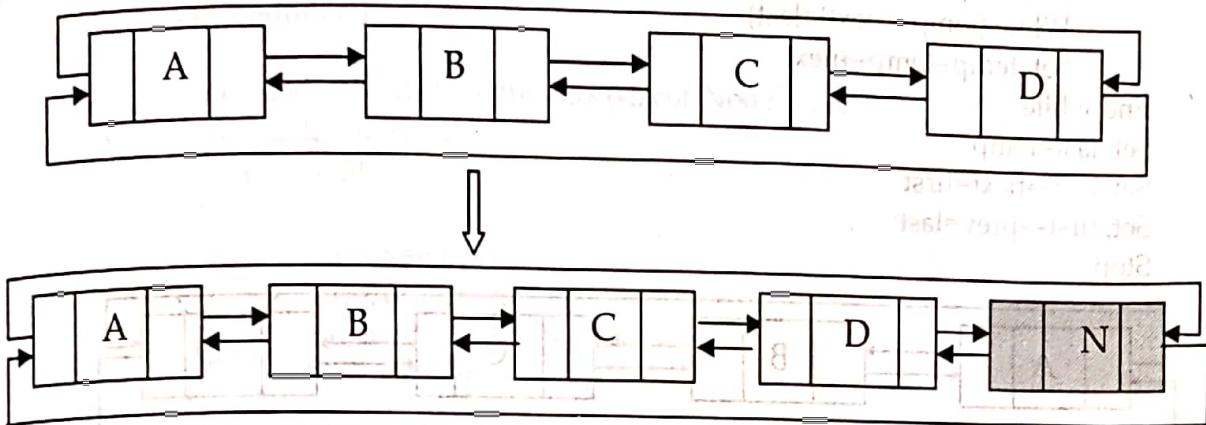
Set, Newnode→next=Newnode

Set, Newnode→prev=Newnode

Otherwise,

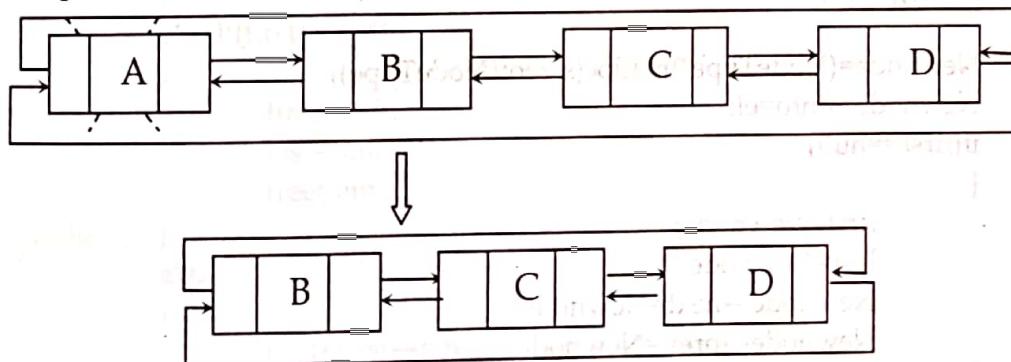
6. Newnode→next=first
7. Set, first→prev=Newnode

8. Set  $\text{last} \rightarrow \text{next} = \text{Newnode}$
9. Set  $\text{Newnode} \rightarrow \text{prev} = \text{last}$
10. Set  $\text{last} = \text{Newnode}$
11. Stop



#### Algorithm to delete a node from the beginning of a circular doubly linked list

1. Start
2. If  $\text{first} == \text{null}$  then
  - Print "Empty linked list" and exit
3. Else if  $\text{first} == \text{last}$  then
  - Set,  $\text{temp} = \text{first}$
  - Set,  $\text{first} = \text{last} = \text{null}$
4. Otherwise,
  - Set,  $\text{temp} = \text{first}$
  - Set,  $\text{first} = \text{first} \rightarrow \text{next}$
  - Set,  $\text{first} \rightarrow \text{prev} = \text{last}$
  - Set,  $\text{last} \rightarrow \text{next} = \text{first}$
5. Free( $\text{temp}$ )
6. Stop



#### Algorithm to delete a node from the end of a circular doubly linked list

1. Start
2. If  $\text{first} == \text{null}$  then
  - Print "Empty linked list" and exit

```

void DeleteLast()
{
 NodeType *temp;
 temp=first;
 if(last==null)
 {
 printf("Empty linked list");
 }
 else if(first==last)
 {
 first=null;
 last=null;
 free(temp);
 }
 else
 {
 while(temp->next!=last)
 {
 temp=temp->next;
 }
 last=temp;
 last->next=first;
 first->prev=last;
 free(temp->next);
 }
}

struct NodeType
void Display()
{
 NodeType *temp;
 temp=first;
 if(first==null)
 {
 printf("Empty linked list");
 }
 else
 {
 while(temp!=last)
 {
 printf("%d\t", temp->info);
 temp=temp->next;
 }
 printf("%d", last->info);
 }
}

void main()
{
}

```

```

int choice;
int item;
printf("1:Insert at beginning");
printf("2:Insert at last");
printf("3:delete first node");
printf("4:delete last node");
printf("5:Display");
do
{
 printf("Enter your choice");
 scanf("%d", &choice);
 switch(choice)
 {

```

**no. 2 Operation**

```

 case 1:
 printf("Enter data item to be inserted");
 scanf("%d", &item);
 insertbeg(item);
 break;
 case 2:
 printf("Enter data item to be inserted");
 scanf("%d", &item);
 insertEnd(item);
 break;
 case 3:
 DeleteFirst();
 break;
 case 4:
 DeleteLast();
 break;
 case 5:
 Display();
 break;
 default:
 printf("Invalid choice Plz enter correct choice");
 }
} while(choice<6);
}

```

**Output**

1: Insert at beginning

2: Insert at last

3: delete first node

4: delete last node

5: Display

Enter your choice

1

Enter data item to be inserted

22

5. Display 'temp → data' and move it to the next node. Repeat the same until temp reaches to the first node in the stack (temp → next! = NULL).
6. Stop

### A Complete menu driven program in C for linked list implementation of stack

```
#include<stdio.h>
#include<conio.h>
#include<malloc.h>
#include<process.h>
struct node
{
 int info;
 struct node *next;
};
typedef struct node NodeType;
NodeType *top;
top=0;
void push(int);
void pop();
void display();
void main()
{
 int choice, item;
 clrscr();
 do
 {
 printf("\n1.Push \n2.Pop \n3.Display\n4:Exit\n");
 printf("enter ur choice\n");
 scanf("%d",&choice);
 switch(choice)
 {
 case 1:
 printf("\nEnter the data:\n");
 scanf("%d",&item);
 push(item);
 break;
 case 2:
 pop();
 break;
 case 3:
 display();
 break;
 case 4:
 exit(1);
 break;
 default:
 printf("invalid choice\n");
 }
 }while(choice<5);
```

```

getch();
}

void push(int item) /*PUSH function*/
{
 NodeType *nnode;
 int data;
 nnode=(NodeType *)malloc(sizeof(NodeType));
 if(top==0)
 {
 nnode->info=item;
 nnode->next=NULL;
 top=nnode;
 }
 else
 {
 NodeType *temp;
 nnode->info=item;
 nnode->next=(NodeType *)malloc(sizeof(NodeType));
 temp=top;
 top=nnode;
 }
}
}

void pop() /*POP function*/
{
 NodeType *temp;
 if(top==0)
 {
 printf("Stack contain no elements:\n");
 return;
 }
 else
 {
 temp=top;
 top=top->next;
 printf("\ndeleted item is %d\t",temp->info);
 free(temp);
 }
}
}

void display() /*display function*/
{
 NodeType *temp;
 if(top==0)
 {
 printf("Stack is empty\n");
 return;
 }
 else
 {
 temp=top;
 }
}

```

```

5. void display()
6. {
7. printf("Stack items are:\n");
8. temp = top;
9. while(temp!=0)
10. {
11. printf("%d\t",temp->info);
12. temp = temp->next;
13. }
14. }

```

**Output**

1: Push

2: Pop

3: Display

Enter your choice

1

Enter data item to be inserted

22

Enter your choice

1

Enter data item to be inserted

44

Enter your choice

1

Enter data item to be inserted

55

Enter your choice

3

55 44 22 Enter your choice

2

Popped item=55

Enter your choice

**Linked list implementation of linear queue**

Similar to stack, the queue can also be implemented using both arrays and linked lists. But it also has the same drawback of limited size. Hence, we will be using a linked list to implement the queue. In linked list implementation of queue, we need to create a node for each data items and arrange nodes in first in first out manner.

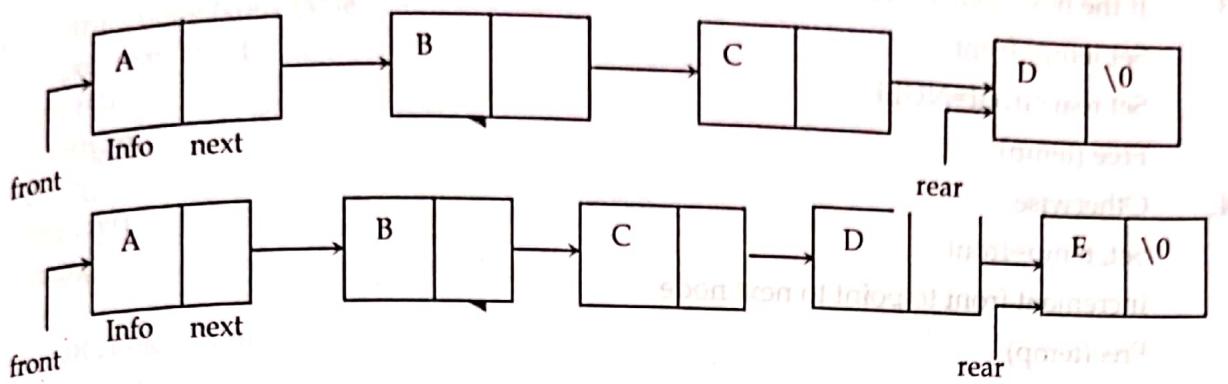
**Algorithm**

1. Start
2. Create a new node say newNode
3. Set the value to be inserted into info field of newNode.
4. If the Queue is empty, then set both front and rear to point to newNode and set next field of newNode to NULL.
5. If the Queue is not empty, then set next of rear to the newNode and the rear to point to the new node.
6. Stop

The time complexity for Enqueue operation is O(1). The Method for Enqueue will be like the following.

**Insert function**

Let \*rear and \*front are pointers to the first node of the list initially and insertion of node in linked list done at the rear part and deletion of node from the linked list done from front part.

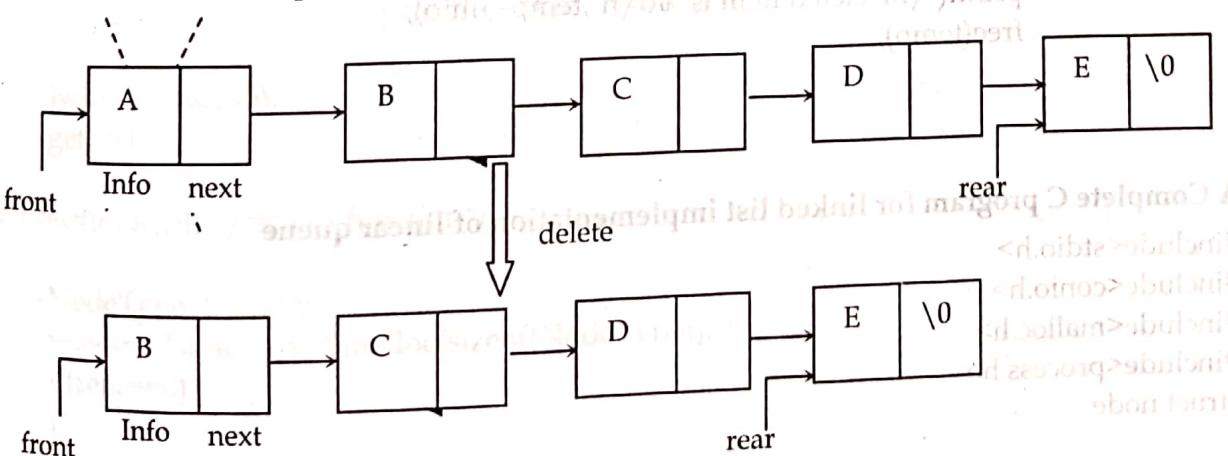


**void insert(int item)**

```
{
 NodeType *nnode;
 nnode=(NodeType *)malloc(sizeof(NodeType));
 if(rear==0)
 {
 nnode->info=item;
 nnode->next=NULL;
 rear=front=nnode;
 }
 else
 {
 nnode->info=item;
 nnode->next=NULL;
 rear->next=nnode;
 rear=nnode;
 }
}
```

**Delete function**

Let \*rear and \*front are pointers to the first node of the list initially and insertion of node in linked list done at the rear part and deletion of node from the linked list done from front part.



```

 nnode->next=NULL;
 rear=front=nnode;
 }
 else
 {
 nnode->info=item;
 nnode->next=NULL;
 rear->next=nnode;
 rear=nnode;
 }
}
void delet() /*delete function*/
{
 NodeType *temp;
 if(front==0)
 {
 printf("Queue contain no elements:\n");
 return;
 }
 else if(front->next==NULL)
 {
 temp=front;
 rear=front=NULL;
 printf("\nDeleted item is %d\n",temp->info);
 free(temp);
 }
 else
 {
 temp=front;
 front=front->next;
 printf("\nDeleted item is %d\n",temp->info);
 free(temp);
 }
}
void display()/*display function*/
{
 NodeType *temp;
 temp=front;
 printf("\nqueue items are:\t");
 while(temp!=NULL)
 {
 printf("%d\t", temp->info);
 temp=temp->next;
 }
}

```

**Output**

1: Enqueue  
2: Dequeue  
3: Display  
4. Exit  
Enter your choice

1  
Enter data item to be inserted

22  
Enter your choice

1  
Enter data item to be inserted

87  
Enter your choice

1  
Enter data item to be inserted

55  
Enter your choice

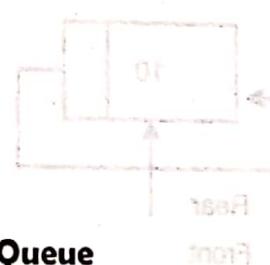
22 87 55 Enter your choice

2  
Deleted item=22

Enter your choice

2  
Deleted item=87

Enter your choice



### **Linked List Implementation of Circular Queue**

By using circular linked list we can easily simulate circular queue. Inserting node at beginning of circular linked list act as enqueue operation of circular queue. Similarly deleting last node of circular linked list act as dequeue operation. In circular queue we can insert element from rear end and delete element from front end. Like this linked list implementation of circular queue we can insert new node at beginning of circular linked list i.e. from rear parts and deleted from end of circular linked list i.e. from front part of circular linked list.

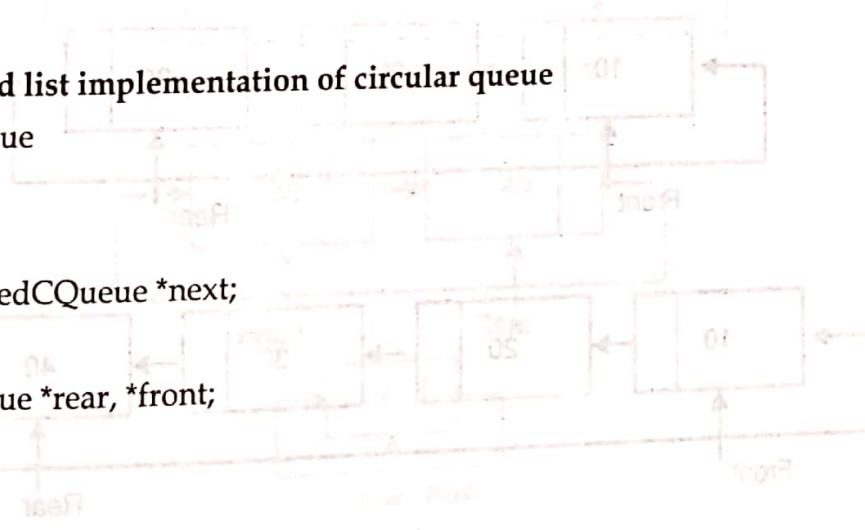
#### **Structure for linked list implementation of circular queue**

Struct LinkedCQueue

```
{
 int info;
 Struct LinkedCQueue *next;
};
```

Struct LinkedCQueue \*rear, \*front;

rear=front=NULL;



### Insertion Algorithm

Inserting node at beginning of circular linked list act as enqueue operation

1. Start
2. Create a new node by using malloc function as,

```
Newnode=(Struct LinkedCQueue *)malloc(sizeof(Struct LinkedCQueue));
```

3. Read data item to be inserted say it be 'el'

4. Set Newnode→info=el

5. if first==null then

```
Set, Newnode→next=Newnode
```

```
Set, first=Newnode
```

```
Set, last =Newnode
```

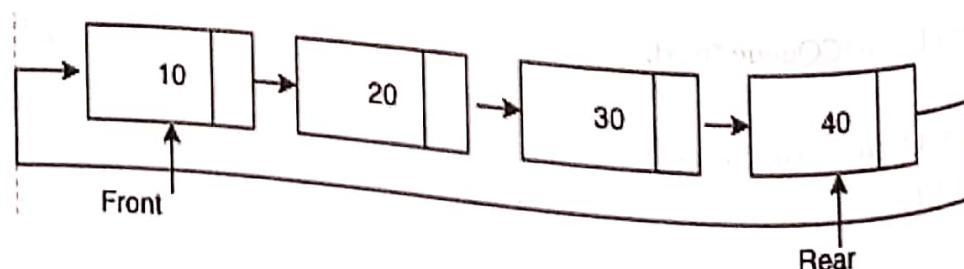
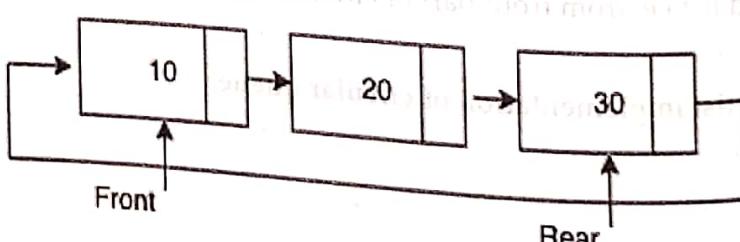
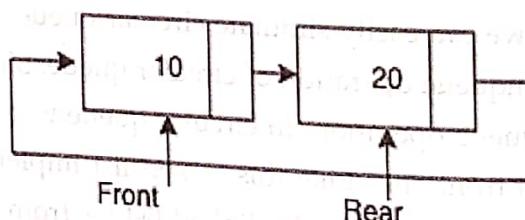
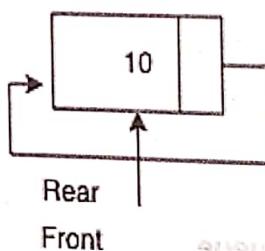
6. else

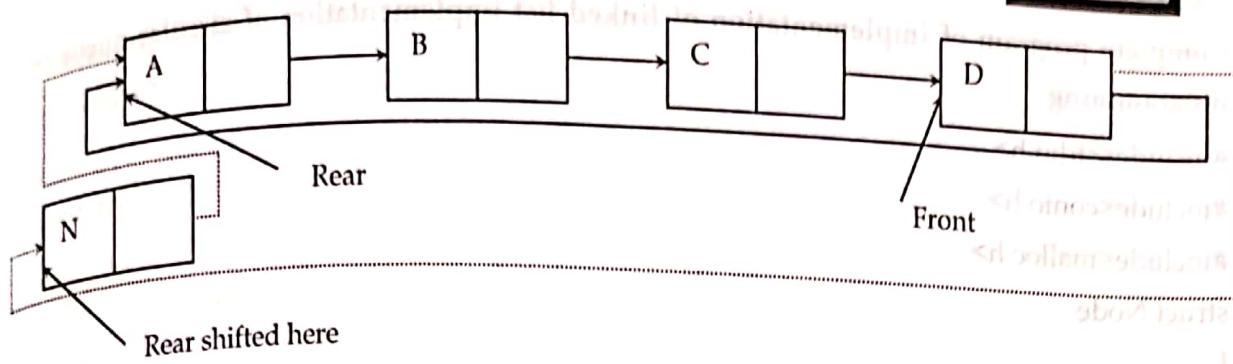
```
Set, Newnode→next=start
```

```
Set, first=Newnode
```

```
Set, last→next=Newnode
```

7. End

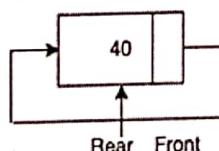
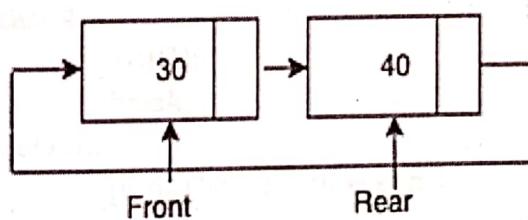
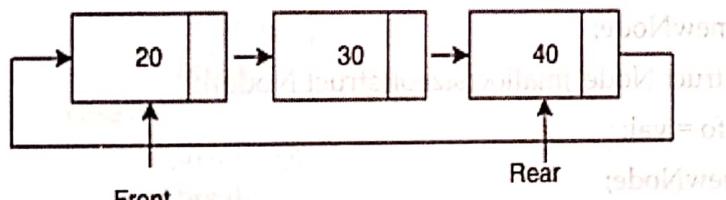
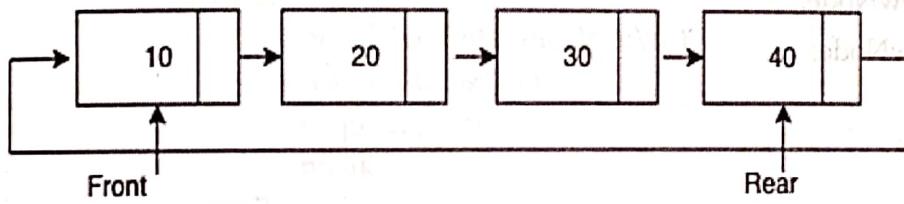




### Dequeue Algorithm

Deleting last node of circular linked list act as dequeue operation of linked list implementation of circular queue

1. Start
2. if first==null then  
Print "empty list" and exit
3. else if first==last  
• Print "first.info" as deleted element  
• Set, first=last=null
4. else  
Set, temp=first  
while(temp→next!=last)  
Set, temp=temp→next  
End while  
Print the deleted element=last→info  
Set, last=temp  
Set, last→next=first
5. Stop



**Complete program of implementation of linked list implementation of circular queue in C programming**

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

struct Node
{
 int info;
 struct Node *next;
};

Node *front = NULL;
Node *rear = NULL;

void enqueue(int val)
{
 if(front==NULL || rear==NULL)
 {
 //create a new node
 struct Node *newNode;
 newNode = (struct Node*)malloc(sizeof(struct Node));
 newNode->info = val;
 newNode->next = newNode;
 front = newNode;
 rear = newNode;
 }
 else
 {
 struct Node *newNode;
 newNode = (struct Node*)malloc(sizeof(struct Node));
 newNode->info = val;
 rear->next = newNode;
 newNode->next = front;
 rear = newNode;
 }
}

void dequeue()
{
 struct Node *n;
 n = front;
 front = front->next;
 free(n);
}
```

```

front = front->next;
delete(n);
}

void display()
{
 struct Node *ptr;
 ptr = front;
 do
 {
 printf("%d\t", ptr->info);
 ptr = ptr->next;
 }while(ptr != rear->next);
}

void main()
{
 int choice, item;
 clrscr();
 do
 {
 printf("\n1.Enqueue \n2.Dequeue \n3.Display\n4:Exit\n");
 printf("enter your choice\n");
 scanf("%d", &choice);
 switch(choice)
 {
 case 1:
 printf("\nEnter the data:\n");
 scanf("%d", &item);
 enqueue(item);
 break;
 case 2:
 dequeue();
 break;
 case 3:
 display();
 break;
 case 4:
 exit(1);
 break;
 default:
 printf("invalid choice\n");
 }
 }while(choice<5);
 getch();
}

```

**DISCUSSION EXERCISE**

## MULTIPLE CHOICE QUESTIONS

1. A linear collection of data elements where the linear node is given by means of pointer is called?
  - a) Linked list
  - b) Node list
  - c) Primitive list
  - d) None of the mentioned
2. What would be the asymptotic time complexity to add a node at the end of singly linked list, if the pointer is initially pointing to the head of the list?
  - a)  $O(1)$
  - b)  $O(n)$
  - c)  $\Theta(n)$
  - d)  $\Theta(1)$
3. What would be the asymptotic time complexity to add an element in the linked list?
  - a)  $O(1)$
  - b)  $O(n)$
  - c)  $O(n^2)$
  - d) None of the mentioned
4. The concatenation of two list can performed in  $O(1)$  time. Which of the following variation of linked list can be used?
  - a) Singly linked list
  - b) Doubly linked list
  - c) Circular doubly linked list
  - d) Array implementation of list
5. In linked list each node contain minimum of two fields. One field is data field to store the data second field is?
  - a) Pointer to character
  - b) Pointer to integer
  - c) Pointer to node
  - d) Node
6. Consider the following definition in c programming language
 

```
struct node
{
 int data;
 struct node * next;
}
typedef struct node NODE;
NODE *ptr;
```

 Which of the following c code is used to create new node?
  - a) `ptr = (NODE*)malloc(sizeof(NODE));`
  - b) `ptr = (NODE*)malloc(NODE);`
  - c) `ptr = (NODE*)malloc(sizeof(NODE*));`
  - d) `ptr = (NODE)malloc(sizeof(NODE));`
7. Generally collection of Nodes is called as \_\_\_\_\_.
  - a) Stack
  - b) Heap
  - c) Linked List
  - d) Pointer

8. Which of the following is not a type of Linked List?
- Doubly Linked List
  - Circular Linked List
  - Singly Linked List
  - Hybrid Linked List
9. Linked list is generally considered as an example of \_\_\_\_\_ type of memory allocation.
- Static
  - Dynamic
  - None of these
  - Compile Time
10. In doubly linked lists, traversal can be performed?
- Only in forward direction
  - Only in reverse direction
  - In both directions
  - None of the above



### DISCUSSION EXERCISE

- What is list? Differentiate between list and linked list.
- What is main advantage and disadvantage of using doubly linked list over singly linked list?
- Write an algorithm for printing a singly linked list in reverse, using only constant extra space. This instruction implies that you cannot use recursion but you may assume that your algorithm is a list method.
- Suggest an array implementation of linked lists.
- What is main concept behind skip list?
- Write a method to check whether two singly linked lists have the same contents.
- Write a method to reverse a singly linked list using only one pass through the list.
- Put numbers in a singly linked list in ascending order. Use this operation to find the median in the list of numbers.
- How can a singly linked list be implemented so that insertion requires no test for whether head is null?
- One way to implement a queue is to use a circularly linked list. Assume that the list does not contain a header and that you can maintain one iterator for the list. For which of the following representations can all basic queue operations be performed in constant worst-case time? Justify your answers.
  - Maintain an iterator that corresponds to the first item in the list.
  - Maintain an iterator that corresponds to the last item in the list.

11. What does the following function do for a given Linked List?

```
void fun1(struct Node* head)
{
 if(head == NULL)
 return;
 fun1(head->next);
 printf("%d ", head->data);
}
```

12. What type of memory allocation is referred for Linked lists?
13. Describe what is Node in link list? And name the types of Linked Lists?
14. Mention what are the applications of Linked Lists?
15. What does the dummy header in linked list contain?
16. Mention what is the difference between singly and doubly linked lists?
17. Mention how to insert a new node in linked list where free node will be available?
18. Mention for which header list, you will found the last node contains the null pointer?
19. Mention how to delete first node from singly linked list?
20. Mention the steps to insert data at the starting of a singly linked list?



Answers:

11. The function `fun1` traverses a singly linked list starting from the head. It prints the data of each node and then recursively calls itself to process the next node. This results in printing the data of nodes in reverse order of their insertion.

12. The type of memory allocation referred for Linked lists is dynamic memory allocation.

13. A Node in link list is a structure that contains data and a pointer to the next node. The types of Linked Lists are Singly Linked List, Doubly Linked List, and Circular Linked List.

14. Applications of Linked Lists include file processing, disk management, and memory management.

15. A dummy header in linked list contains a null pointer in its next field, indicating that there is no more data in the list.

16. The difference between singly and doubly linked lists is that singly linked lists have a single pointer to the next node, while doubly linked lists have two pointers: one to the previous node and one to the next node.

17. To insert a new node in linked list where free node will be available, we need to update the next pointer of the previous node to point to the new node, and set the next pointer of the new node to the original next pointer of the previous node.

18. For which header list, you will found the last node contains the null pointer? In a singly linked list, the last node contains a null pointer in its next field.

19. To delete first node from singly linked list, we need to update the next pointer of the second node to point to the original next pointer of the first node, and then free the memory of the first node.

20. To insert data at the starting of a singly linked list, we need to allocate memory for a new node, set its data to the required value, and then update the next pointer of the new node to point to the original head of the list.

# 6

Recursion  
• A recursive function is a function that calls itself.  
• It is used to solve problems that can be broken down into smaller subproblems.

• Example: Factorial function:  $n! = n \cdot (n-1) \cdot (n-2) \cdots 1$ .  
• If  $n=5$ , then  $5! = 5 \cdot 4 \cdot 3 \cdot 2 \cdot 1 = 120$ .

• Recursive functions are often used to solve problems that can be broken down into smaller subproblems.

• Example: Fibonacci series:  $F_n = F_{n-1} + F_{n-2}$ , where  $F_0 = 0$  and  $F_1 = 1$ .  
• If  $n=6$ , then  $F_6 = F_5 + F_4 = 5 + 3 = 8$ .

• Recursion is always slower than iteration because it involves more memory overhead.

• Iteration is faster than recursion because it involves less memory overhead.

• Recursion makes code easier to understand and maintain.

• Recursion is useful for solving problems that involve trees or graphs.

• Recursion is useful for solving problems that involve lists or arrays.

• Recursion is useful for solving problems that involve sets or maps.

• Recursion is useful for solving problems that involve strings or characters.

• Recursion is useful for solving problems that involve matrices or tensors.

• Recursion is useful for solving problems that involve graphs or trees.

• Recursion is useful for solving problems that involve sets or maps.

## RECURSION

(C) Introduction to recursive functions and their applications

(C) Iteration vs. recursion

(C) Fibonacci series

(C) Factorial function

(C) Recursion vs. iteration

### CHAPTER OUTLINE

After studying this chapter, the reader will be able to understand the

➤ Introduction, Principle of recursion, recursion vs. iteration, recursion example: TIH and Fibonacci series, Applications of Recursion, search tree



## INTRODUCTION

Recursion is a process by which a function calls itself repeatedly, until some specified condition has been satisfied. The process is used for repetitive computations in which each action is stated in terms of a previous result. In order to solve a problem recursively, two conditions must be satisfied. First, the problem must be written in a recursive form, and second, the problem statement must include a stopping condition.

For the problems solve recursively following two conditions must be satisfied:

- Each time a function calls itself and it must be closer to a solution.
- The problem statement must include a stopping condition.

**Example:** Finding factorial of a given number

Let us consider the factorial of a number and its algorithm described recursively:

We know that  $n! = n * (n-1)!$

$(n-1)! = n-1 * (n-2)!$  And so on up to 1.

### Algorithm

Factorial (n)

{

    if  $n==1$

        return 1

    else

        return  $n * \text{Factorial}(n-1)$

}

**Let's trace the evaluation of factorial (5)**

Factorial (5) =

5\*Factorial (4) =

5\*(4\*Factorial (3)) =

5\*(4\*(3\*Factorial (2))) =

5\*(4\*(3\*(2\*Factorial (1)))) =

5\*(4\*(3\*(2\*(1\*Factorial (0)))))) =

5\*(4\*(3\*(2\*(1\*1)))) =

5\*(4\*(3\*2)) =

5\*(4\*6) =

5\*24 =

120

### Divide-and-conquer algorithms

An important problem-solving technique that makes use of recursion is dividing and conquering. A divide-and-conquer algorithm is an efficient recursive algorithm that consists of two parts:

- Divide, in which smaller problems are solved recursively (except, of course, base cases)
- Conquer, in which the solution to the original problem is then formed from the solutions to the sub problems

Traditionally, routines in which the algorithm contains at least two recursive calls are called divide-and-conquer algorithms, whereas routines whose text contains only one recursive call are not. Consequently, the recursive routines presented so far in this chapter are not divide-and-conquer algorithms. Also, the sub problems usually must be disjoint (i.e., essentially none overlapping), so as to avoid the excessive costs seen in the sample recursive computation of the Fibonacci numbers.

### Key differences between recursion and iteration

1. Recursion is when a method in a program repeatedly calls itself whereas, iteration is when a set of instructions in a program are repeatedly executed.
2. A recursive method contains set of instructions, statement calling itself, and a termination condition whereas iteration statements contain initialization, increment, condition, set of instruction within a loop and a control variable.
3. A conditional statement decides the termination of recursion and control variable's value decides the termination of the iteration statement.
4. If the method does not lead to the termination condition it enters to infinite recursion. On the other hand, if the control variable never leads to the termination value the iteration statement iterates infinitely.
5. Infinite recursion can lead to system crash whereas, infinite iteration consumes CPU cycles.
6. Recursion is always applied to method whereas, iteration is applied to set of instruction.
7. Variables created during recursion are stored on stack whereas, iteration doesn't require a stack.
8. Recursion causes the overhead of repeated function calling whereas; iteration does not have a function calling overhead.
9. Due to function calling overhead execution of recursion is slower whereas, execution of iteration is faster.
10. Recursion reduces the size of code whereas, iterations make a code longer.

### RECURSION EXAMPLES

#### Example 1: Calculation of the factorial of an integer number using recursive function

```
#include<stdio.h>
#include<conio.h>
void main()
{
 clrscr();
 int n;
 long int facto;
 long int factorial(int n);
 printf("Enter value of n:");
 scanf("%d", &n);
 facto=factorial(n);
 printf("%d! = %ld", n, facto);
 getch();
}
```

```

} called and recursive function will pass the value of n to the next call of factorial function.
long int factorial(int n)
{
 if(n == 0)
 return 1;
 else
 return n * factorial(n-1);
}

```

**Output**

Enter value of n:  
6  
Factorial of 6=720

**Example 2: Program to find factorial of an integer number without using recursive function**

```

#include<stdio.h>
#include<conio.h>
void main()
{
 int n;
 long int facto;
 long int factorial(int n);
 printf("Enter value of n:");
 scanf("%d", &n);
 facto=factorial(n);
 printf("%d! = %ld", n, facto);
 getch();
}

long int factorial(int n)
{
 long int facto=1;
 int i;
 if(n==0)
 return 1;
 else
 {
 for(i=1;i<=n;i++)
 facto=facto*i;
 return facto;
 }
}

```

**Output**

Enter value of n:  
6  
Factorial of 6=720

**Example 3: Program to generate Fibonacci series up to n terms using recursive function.**  
 (Hint Fibonacci sequence=0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55...)

```
#include<stdio.h>
#include<conio.h>
void main()
{
 int n,i;
 int fibo(int);
 printf("Enter n:");
 scanf("%d", &n);
 printf("Fibonacci numbers up to %d terms:\n", n);
 for(i=1;i<=n;i++)
 printf("%d\n", fibo(i));
 getch();
}
```

```
int fibo(int k)
{
 if(k == 1 || k == 2)
 return 1;
 else
 return fibo(k-1)+fibo(k-2);
}
```

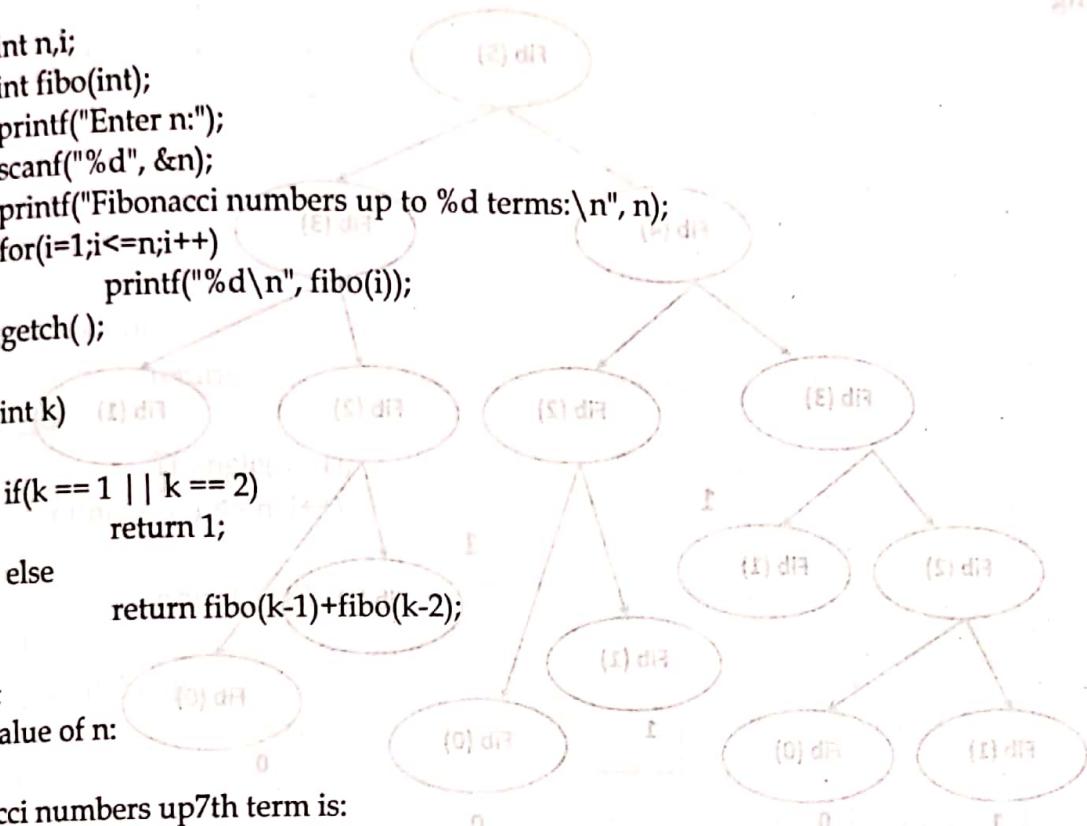
**Output**

Enter value of n:

7

Fibonacci numbers up to 7th term is:

1 1 2 3 5 8 13

**Example 4: Program to find nth term of Fibonacci series using recursion**

```
#include<stdio.h>
#include<conio.h>
void main()
{
 int n,i;
 int fibo(int);
 printf("Enter value of n:");
 scanf("%d", &n);
 printf("nth Fibonacci term is:\n");
 printf("%d", fibo(n));
 getch();
}

int fibo(int k)
{
 if(k == 1 || k == 2)
 return 1;
 else
 return fibo(k-1)+fibo(k-2);
}
```

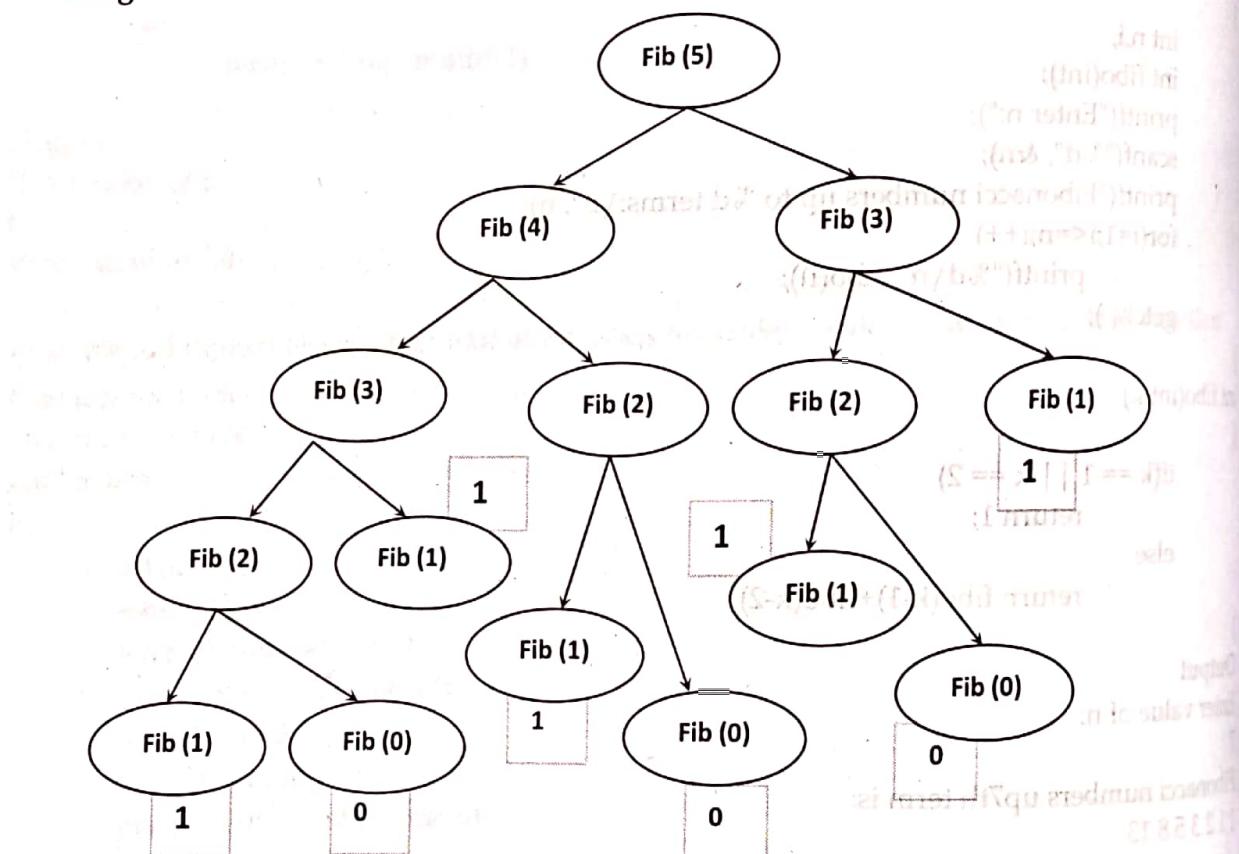
**Output**

Enter value of n:

10

10th term of Fibonacci series is: 55

**Tracing**



Fifth term of Fibonacci series =  $1+0+1+1+0+1+0+1=5$

**Example 5: Program to find sum of first n natural numbers using recursion**

```

#include<stdio.h>
#include<conio.h>
void main()
{
 int n;
 int sum_natural(int);
 printf("Enter value of n");
 scanf("%d", &n);
 printf("Sum of first %d natural numbers = %d", n, sum_natural(n));
 getch();
}
int sum_natural(int n)
{
 if(n == 1)
 return 1;
 else
 return n + sum_natural(n-1);
}

```

**Example 6: Recursive java program to display following pattern**

```

* * * * *
* * * *
* * *
* *
* *
#include<stdio.h>
#include<conio.h>
void Triangle(int n)
{
 if (n <= 0)
 return;
 else
 Triangle(n - 1);
 for (int i = 1; i <= n; i++)
 {
 printf("*\t");
 }
 printf("\n");
}
int main()
{
 Triangle(7);
 return 0;
}

```

The trace of the program will be as follows:

```

Triangle (7)
Triangle (6)
Triangle (5)
Triangle (4)
Triangle (3)
Triangle (2)
Triangle (1)
Triangle (0) ← base case
Triangle (1) ← prints 1 star & new line
Triangle (2) ← prints 2 stars & new line
Triangle (3) ← prints 3 stars & new line
Triangle (4) ← prints 4 stars & new line
Triangle (5) ← prints 5 stars & new line
Triangle (6) ← prints 6 stars & new line
Triangle (7) ← prints 7 stars & new line

```



## Tower of Hanoi problem

### Initial state

- There are three poles named as origin, intermediate and destination.
- $n$  number of different-sized disks having hole at the center is stacked around the origin pole in decreasing order.
- The disks are numbered as 1, 2, 3, 4... n.

### Objective

- Transfer all disks from origin pole to destination pole using intermediate pole for temporary storage.

### Conditions

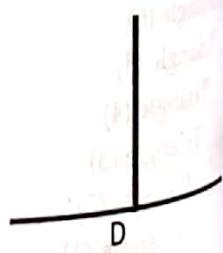
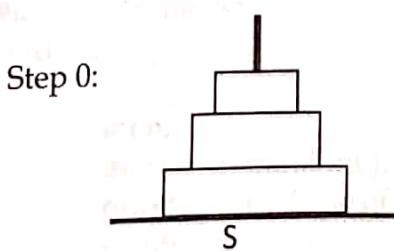
- Move only one disk at a time.
- Each disk must always be placed around one of the pole.
- Never place larger disk on top of smaller disk.

### Algorithm

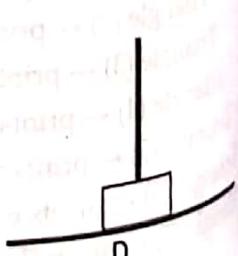
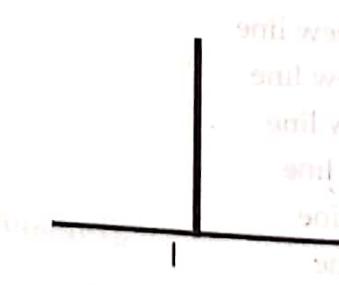
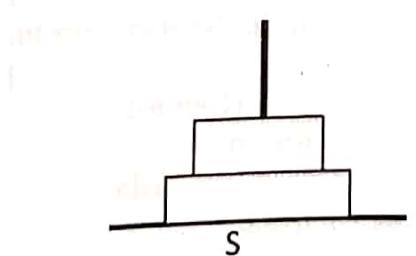
To move a tower of  $n$  disks from **source** to **destination** (where  $n$  is positive integer)

1. If  $n==1$ :  
1.1. Move a single disk from **source** to **destination**
2. If  $n > 1$ :  
2.1. Let **temp** be the remaining pole other than **source** and **destination**  
2.2. Move a tower of  $(n-1)$  disks form **source** to **temp**  
2.3. Move a single disk from **source** to **destination**  
2.4. Move a tower of  $(n-1)$  disks form **temp** to **destination**
3. Stop

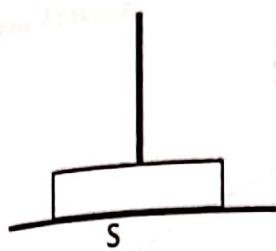
### Tracing (Tracing for $n=3$ )



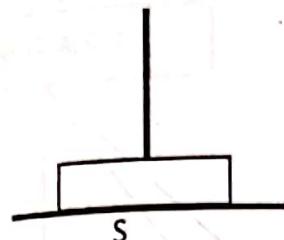
### Step 1:



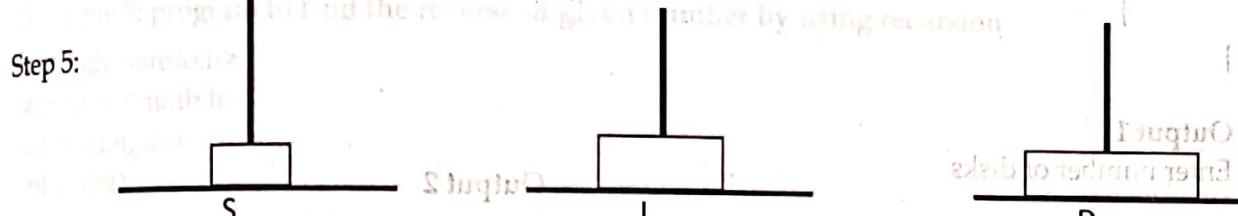
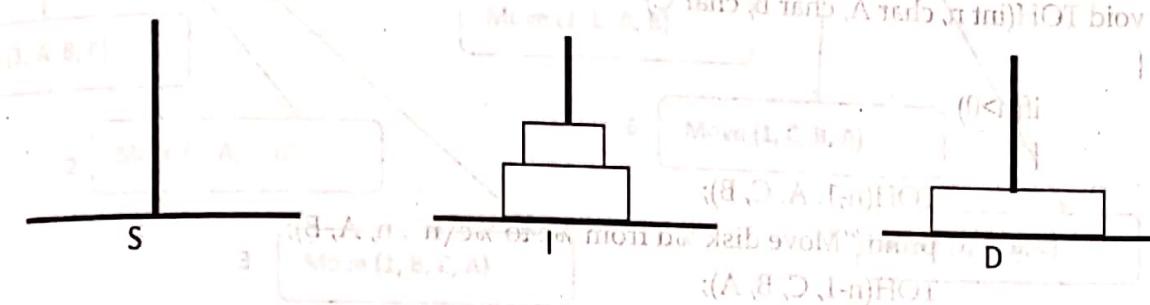
Step 2:



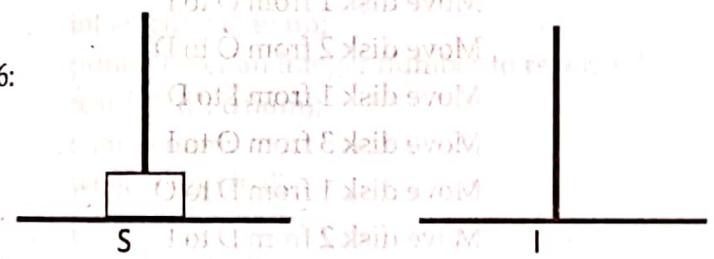
Step 3:



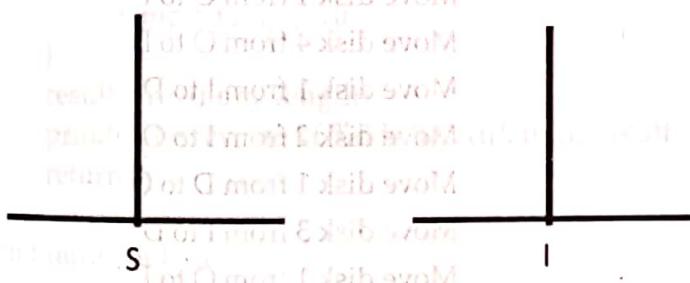
Step 4:



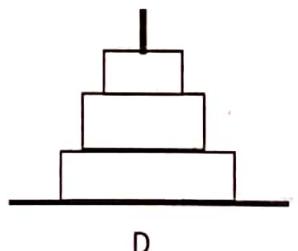
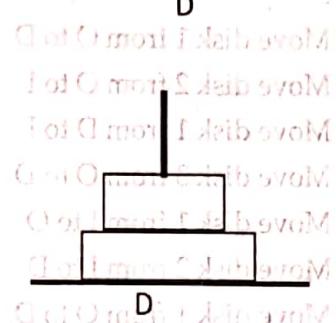
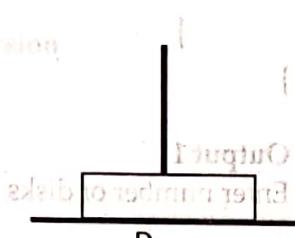
Step 6:



Step 7:

Number of steps required in TOH problem =  $(2^n - 1)$ 

For n=3,

Number of steps required in TOH problem =  $(2^n - 1) = (2^3 - 1) = 8 - 1 = 7$ 

**Example 7: Recursive solution of tower of Hanoi**

```
#include <stdio.h>
#include <conio.h>
void TOH(int, char, char, char); // Function prototype
void main()
{
 int n;
 printf("Enter number of disks");
 scanf("%d", &n);
 TOH(n, 'O', 'D', 'I');
 getch();
}
void TOH(int n, char A, char B, char C)
{
 if(n>0)
 {
 TOH(n-1, A, C, B);
 printf("Move disk %d from %c to %c\n", n, A, B);
 TOH(n-1, C, B, A);
 }
}
```

**Output 1**

Enter number of disks

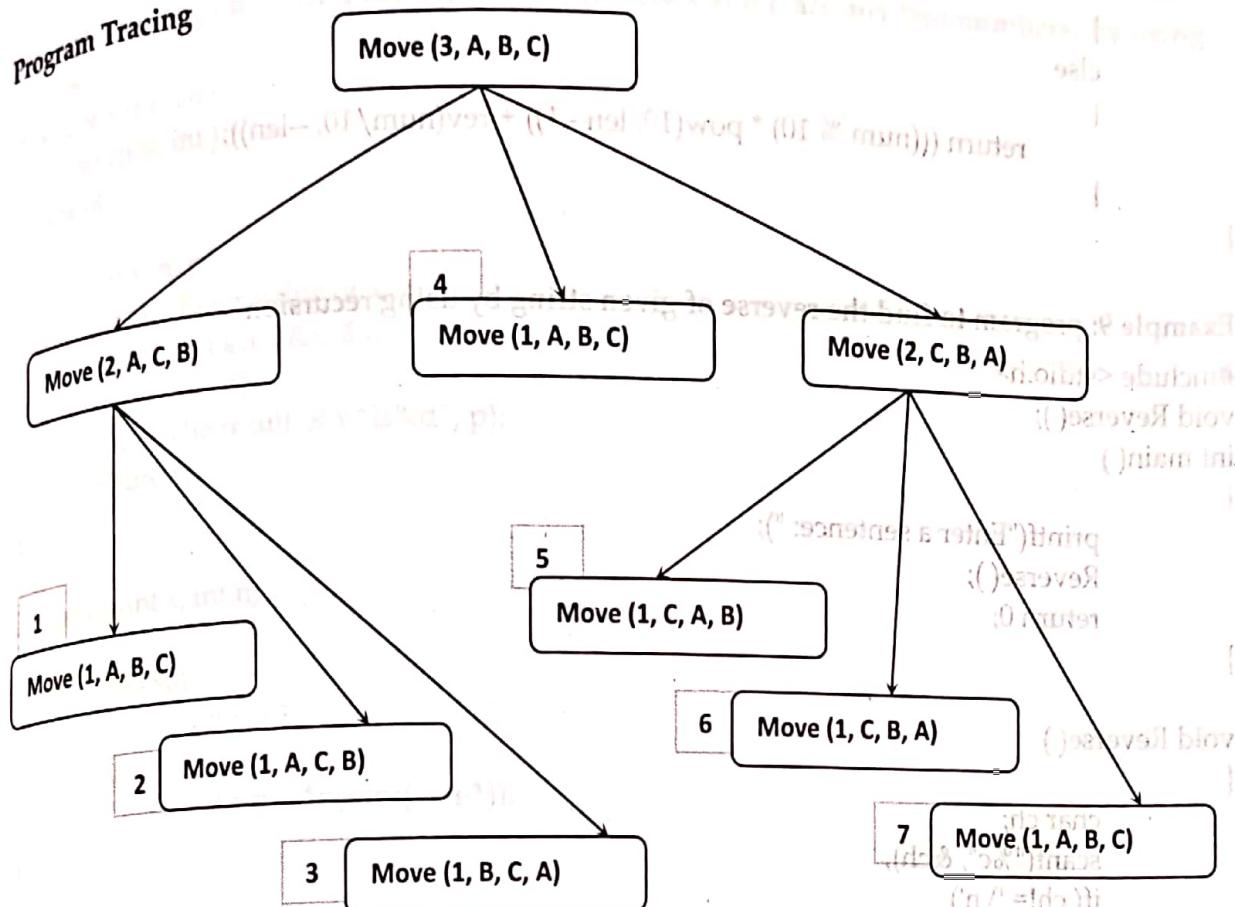
3  
Move disk 1 from O to D  
Move disk 2 from O to I  
Move disk 1 from D to I  
Move disk 3 from O to D  
Move disk 1 from I to O  
Move disk 2 from I to D  
Move disk 1 from O to D

**Output 2**

Enter number of disks: 4

Move disk 1 from O to I  
Move disk 2 from O to D  
Move disk 1 from I to D  
Move disk 3 from O to I  
Move disk 1 from D to O  
Move disk 2 from D to I  
Move disk 1 from O to I  
Move disk 4 from O to D  
Move disk 1 from I to D  
Move disk 2 from I to O  
Move disk 1 from D to O  
Move disk 3 from I to D  
Move disk 1 from O to I  
Move disk 2 from O to D  
Move disk 1 from I to D

## Program Tracing



## Example 8: program to find the reverse of given number by using recursion

```

#include <stdio.h>
#include <math.h>
int rev(int, int);
int main()
{
 int num, result;
 int length = 0, temp;
 printf("Enter an integer number to reverse: ");
 scanf("%d", &num);
 temp = num;
 while (temp != 0)
 {
 length++;
 temp = temp / 10;
 }
 result = rev(num, length);
 printf("The reverse of %d is %d.\n", num, result);
 return 0;
}
int rev(int num, int len)
{
 if (len == 1)
 {
 return num;
 }

```

```

 }
 else
 {
 return (((num % 10) * pow(10, len - 1)) + rev(num/10, --len));
 }
}
}

```

### Example 9: program to find the reverse of given string by using recursion

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

```
void Reverse();
```

```
int main()
```

```
{
```

```
 printf("Enter a sentence: ");
```

```
 Reverse();
```

```
 return 0;
```

```
}
```

```
void Reverse()
```

```
{
```

```
 char ch;
```

```
 scanf("%c", &ch);
```

```
 if(ch!= '\n')
```

```
 Reverse();
```

```
 printf("%c", ch);
```

```
}
```

*Output:*

### Example 10: program to find the Highest common factor (greatest common divisor) of any two numbers by using recursion

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

```
int GCD(int, int);
```

```
int main()
```

```
{
```

```
 int a, b, g;
```

```
 printf("Enter any two numbers");
```

```
 scanf("%d%d", &a, &b);
```

```
 g=GCD(a, b);
```

```
 printf("GCD of given two numbers=%d", g);
```

```
 return 0;
```

```
}
```

```
int GCD(int x, int y)
```

```
{
```

```
 if(y==0)
```

```
 return x;
```

```
 else
```

```
 return (GCD(y, x%y));
```

```
}
```

**Example 11:** program to find the value of  $x^n$ , (where x and n are any two numbers) by using recursion.

```
#include <stdio.h>
int power(int, int);
int main()
{
 int x, n, p;
 printf("Enter value of x and n");
 scanf("%d%d", &x, &n);
 p=power(x, n);
 printf("The result of x^n is %d", p);
 return 0;
}

int power(int x, int n)
{
 if(n==0)
 return 1;
 else
 return (x*power(x, n-1));
}
```

**Example 12:** program to display string "MY name is Khan" 10 times by using recursion.

```
#include <stdio.h>
int main()
{
 static int i=10;
 if(i>0)
 {
 printf("My name is Khan\n");
 i--;
 main();
 }
}
```

#### Output

```
My name is Khan
```

**Example 13: program to check whether given string is palindrome or not by using recursion.**

(Note: string "madam" is palindrome but string "Bhupi" is not palindrome)

```
#include <stdio.h>
#include <string.h>
void checkpalindrome(char [], int);
int main()
{
 char word[15];
 printf("Enter a word to check if it is a palindrome\n");
 scanf("%s", word);
 check(word, 0);
 return 0;
}
void checkpalindrome(char word[], int index)
{
 int len;
 len = strlen(word) - (index + 1);
 if (word[index] == word[len])
 {
 void Recursion()
 {
 if (index + 1 == len || index == len)
 {
 printf("The entered word is a palindrome\n");
 return;
 }
 checkpalindrome (word, index + 1);
 }
 }
 else
 printf("The entered word is not a palindrome\n");
}
```

**Example 14: program to check whether given number is palindrome or not by using recursion.** (Note: number 121 is palindrome but 5678 is not palindrome)

#include<stdio.h>

```
int main()
{
 int num,rev;
 printf("\n Enter a number :");
 scanf("%d", &num);
 rev=reverse(num);
 printf("\n After reverse the no is :%d", rev);
 return 0;
}
int reverse(int num)
{
 static int r, sum=0;
 if(num!=0)
 {
 r=num%10;
 sum=sum*10+r;
 reverse(num/10);
 }
 else
 return sum;
}
```

## Advantages of Recursion

- The code may be much easier to write.
- To solve some problems which are naturally recursive such as tower of Hanoi.

## Disadvantages of Recursion

- Recursive functions are generally slower than non-recursive functions.
- May require a lot of memory to hold intermediate results on the system stack.
- It is difficult to think recursively so one must be very careful when writing recursive functions.

## Types of recursion

Recursion can be categorized into following 5 types:

- Direct recursion
- Indirect recursion
- Tail recursion
- Linear recursion
- Tree recursion

### Direct recursion

A function is called direct recursive if it calls directly to itself until some base condition is not satisfied.

**Example:** Function for finding factorial of given number by using direct recursion method.

```
int fact(int num)
{
 if(num==0)
 return 1;
 else
 return num*fact(num-1);
}
```

### Indirect recursion

Up to now we discussed only direct recursion, where a method f() called itself. However, f() can call itself indirectly via a chain of other calls. For example, f() can call g(), and g() can call f(). This is the simplest case of indirect recursion. The chain of intermediate calls can be of an arbitrary length, as in:

$$f() \rightarrow f1() \rightarrow f2() \rightarrow \dots \rightarrow fn() \rightarrow f()$$

Simply, a function is said to be indirect recursive if it calls to another function and again this new function calls to its calling function.

**Example:** Program for Indirect recursion in C

```
int fun1(int x)
{
 if(x<=0)
 return 1;
 else
 return fun2(x);
}

int fun2(int y)
{
 return fun1(y-1);
}
```

**What is the difference between direct and indirect recursion?**

A function fun is called direct recursive if it calls the same function fun. A function fun is called indirect recursive if it calls another function say fun\_new and fun\_new calls fun directly or indirectly. Difference between direct and indirect recursion has been illustrated in below.

**An example of direct recursion**

```

void directRecFun()
{
 // some code...
 directRecFun();
}

// some code... /> Recursion starts here
void indirectRecFun()
{
 // some code...
 indirectRecFun2();
}

void indirectRecFun2()
{
 // some code...
 indirectRecFun();
}

```

An example of indirect recursion

**What is the difference between factorial and iterative factorial?**

A function fact(n, accumulator) is said to be tail recursive if its recursive call is said to be tail recursive. Simply a function that returns the value of its recursive call is said to be tail recursive. Simply a function is said to be tail recursive if there are no any calculations occur in the recursive stage and it only returns the value.

**Example: Complete program in C to showing the use of tail recursion**

```

tail recursion
Fact(n, accumulator)
{
 if (n == 0)
 return Fact(0, 1);
 else
 return Fact(n - 1, n * accumulator);
}

Fact(n)
{
 return Fact(n, 1);
}

void main()
{
 int num;
 cout << "Enter any number";
 cin >> num;
 cout << "Factorial of given number = ";
 cout << Fact(num);
}

```

**Factorial(n)**

```

{
 if (n == 0)
 return 1;
 else
 return (n * Factorial(n - 1));
}

```

**Factorial of given number = 3628800**

**MULTIPLE CHOICE QUESTIONS**

1. Recursion is a method in which the solution of a problem depends on \_\_\_\_\_
  - Larger instances of different problems
  - Larger instances of the same problem
  - Smaller instances of the same problem
  - Smaller instances of different problems
2. Which of the following problems can be solved using recursion?
  - Factorial of a number
  - Nth Fibonacci number
  - Length of a string
  - All of the mentioned
3. Recursion is similar to which of the following?
  - Switch Case
  - Loop
  - If-else
  - None of the mentioned
4. In recursion, the condition for which the function will stop calling itself is.....
  - Best case
  - Worst case
  - Base case
  - There is no such condition
5. Which of the following statements is true?
  - Recursion is always better than iteration
  - Recursion uses more memory compared to iteration
  - Recursion uses less memory compared to iteration
  - Iteration is always better and simpler than recursion
6. How many times is the recursive function called, when the following code is executed?  

```
void my_recursive_function(int n)
{
 if(n == 0)
 return;
 printf("%d ",n);
 my_recursive_function(n-1);
}
```
7. Which of the following recursive formula can be used to find the factorial of a number?
  - $\text{fact}(n) = n * \text{fact}(n)$
  - $\text{fact}(n) = n * \text{fact}(n+1)$
  - $\text{fact}(n) = n * \text{fact}(n-1)$
  - $\text{fact}(n) = n * \text{fact}(1)$
8. What is the space complexity of the above recursive implementation to find the factorial of a number?
  - $O(1)$
  - $O(n)$
  - $O(n^2)$
  - $O(n^3)$
9. Suppose the first fibonacci number is 0 and the second is 1. What is the sixth fibonacci number?
  - 5
  - 6
  - 7
  - 8

10. Which of the following methods can be used to find the nth Fibonacci number?
- Dynamic programming
  - Recursion
  - Iteration
  - All of the mentioned



## DISCUSSION EXERCISE

- What is Recursion? What is base condition in recursion?
  - How a particular problem is solved using recursion?
  - Why Stack Overflow error occurs in recursion?
  - What is the difference between direct and indirect recursion?
  - What is difference between tailed and non-tailed recursion?
  - How memory is allocated to different function calls in recursion?
  - What are the disadvantages and disadvantages of recursive programming over iterative programming?
  - Explain the functionality of following functions.
- ```
int fun1(int n)
{
    if(n == 1)
        return 0;
    else
        return 1 + fun1(n/2);
}
```
- Write a program in C to copy one string to another using recursion.
 - Write a program in C to find the first capital letter in a string using recursion.
 - Write a program in C to find the Hailstone Sequence of a given number.
- Note:** The hailstone sequence starting at 13 is: [13 40 20 10 5 16 8 4 2 1]
The length of the sequence is 10.
- Write a program in C to calculate the power of any number using recursion.
 - Write a program in C to Check whether a given String is Palindrome or not.
 - Write a program in C to multiply two matrix using recursion.
 - Write a program in C to find the LCM of two numbers using recursion.
 - Write a program in C to convert a decimal number to binary using recursion.
 - Write a program in C to find the Factorial of a number using recursion.
 - Write a program in C to reverse a string using recursion.
 - Write a program in C to get the largest element of an array using recursion.
 - Write a program in C to find GCD of two numbers using recursion.

T

TREE

-
-
-

Definition

A tree is a non-linear data structure that consists of nodes connected by edges. A tree has a root node which is the top-most node. All other nodes are children of the root node. A tree can have multiple children.

The height of a tree is the maximum number of edges between the root node and a leaf node. The depth of a tree is the maximum number of edges between any two leaf nodes. The width of a tree is the maximum number of children of a single node.



CHAPTER OUTLINE

After studying this chapter, the reader will be able to understand the

- Introduction, Basic operation in Binary tree, tree search and insertion/Deletion, Binary tree traversals (Pre-order, post-order and in-order), Tree Height, level, and Depth, Balanced trees: AVL Balanced trees, Balancing algorithm, the Huffman algorithm, Game tree, B-tree.

INTRODUCTION

A tree is a nonlinear data structure that models a hierarchical organization. The characteristic features are that each element may have several successors called its children and every element except one called the root has a unique predecessor called its parent. A tree is a data structure which consists of a finite set of elements, called nodes and a finite set of directed lines, called branches that connect the nodes. The number of branches associated with a node is the degree of the node. The branch coming towards the node is the in-degree of the node. Similarly the outgoing branches from the node mean the out-degree of that node. The first node of the tree is known as root. When the tree is empty, then root is equals to NULL. In another case the in-degree of the root is always 0.

Trees can be defined in two ways: non-recursively and recursively. The non-recursive definition is the more direct technique, so we begin with it. The recursive formulation allows us to write simple algorithms to manipulate trees. Trees fall into the category of non-primitive non-linear data structures in the classification of data structure. They contain a finite set of data items referred as nodes. We can represent a hierarchical relationship between the data elements using trees.

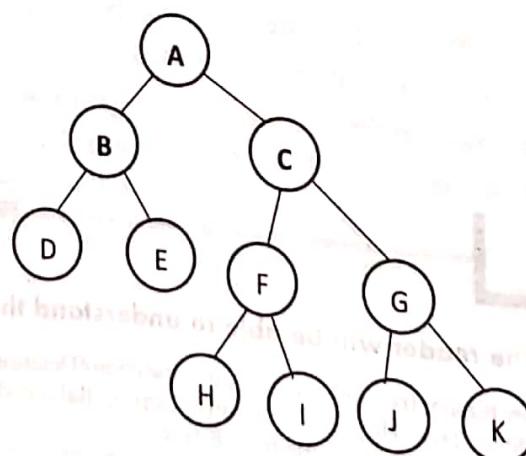
A Tree has the following characteristics:

- A Tree is a recursive data structure containing the set of one or more data nodes where one node is designated as the root of the tree while the remaining nodes are called as the children of the root.
- The nodes other than the root node are partitioned into the non empty sets where each one of them is to be called sub-tree.
- Nodes of a tree either maintain a parent-child relationship between them or they are sister nodes.
- In a general tree, a node can have any number of children nodes but it can have only a single parent.
- Unlike natural trees, trees in the data structure always grow in length towards the bottom.

Definition

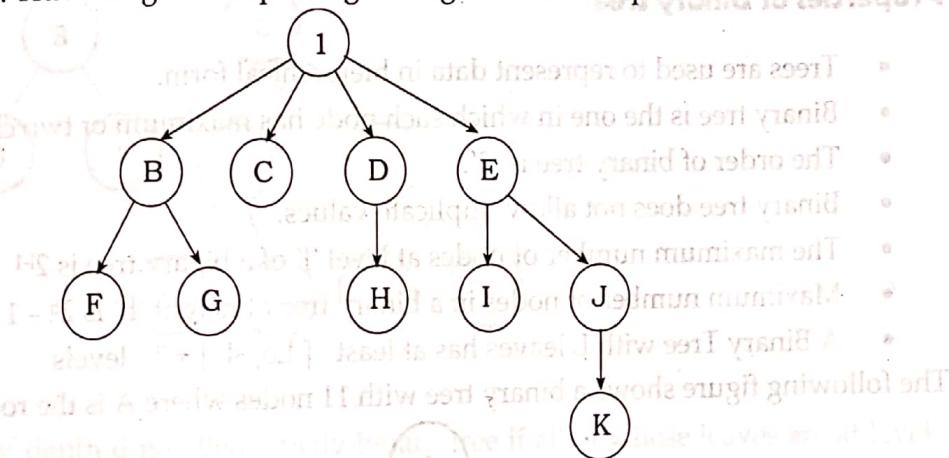
A tree is a nonlinear data structure and is generally defined as a nonempty finite set of elements which consists of set of nodes called vertices and set of edges which links vertices such that:

- T contains a distinguished node called root of the tree.
- The remaining elements of tree form an ordered collection of zero or more disjoint subsets called sub tree.



Terminology

- **Root Node:** The starting node of a tree is called Root node of that tree
- **Terminal Nodes:** The node which has no children is said to be terminal node or leaf.
- **Non-Terminal Node:** The nodes which have children is said to be Non-Terminal Nodes
- **Degree:** The degree of a node is number of sub trees of that node
- **Levels:** Level of a node represents the generation of a node. If the root node is at level 0, then its next child node is at level 1, its grandchild is at level 2, and so on.
- **Depth:** The length of largest path from root to terminals is said to be depth of tree.
- **Depth of a node:** The depth of a node is the length of the path from the root to the node.
- **Height of a node:** The height of a node is the length of the path from the node to the deepest leaf.
- **Siblings:** The children of same parent are said to be siblings
- **Ancestors:** The ancestors of a node are all the nodes along the path from the root to the node
- **Keys:** Key represents a value of a node based on which a search operation is to be carried out for a node.
- **Sub-tree:** Sub-tree represents the descendants of a node.
- **Traversing:** Traversing means passing through nodes in a specific order.



The root node is A; A's children are B, C, D, and E. Because A is the root, it has no parent; all other nodes have parents. For instance, B's parent is A. A node that has no children is called a leaf. The leaves in this tree are C, F, G, H, I, and K. The length of the path from A to K is 3 (edges); the length of the path from A to A is 0 (edges).

A tree with N nodes must have $N - 1$ edges because every node except the parent has an incoming edge. The depth of a node in a tree is the length of the path from the root to the node. Thus the depth of the root is always 0, and the depth of any node is 1 more than the depth of its parent. The height of a node in a tree is the length of the path from the node to the deepest leaf. Thus the height of E is 2. The height of any node is 1 more than the height of its maximum-height child. Thus the height of a tree is the height of the root. Nodes with the same parent are called siblings; thus B, C, D, and E are all siblings. If there is a path from node u to node v, then

u is an ancestor of v and v is a descendant of u . If $u \neq v$, then u is a proper ancestor of v and v is a proper descendant of u .

Node	Height	Depth
A	3	0
B	1	1
C	0	1
D	1	1
E	2	1
F	0	2
G	0	2
H	0	2
I	0	2
J	1	2
K	0	3

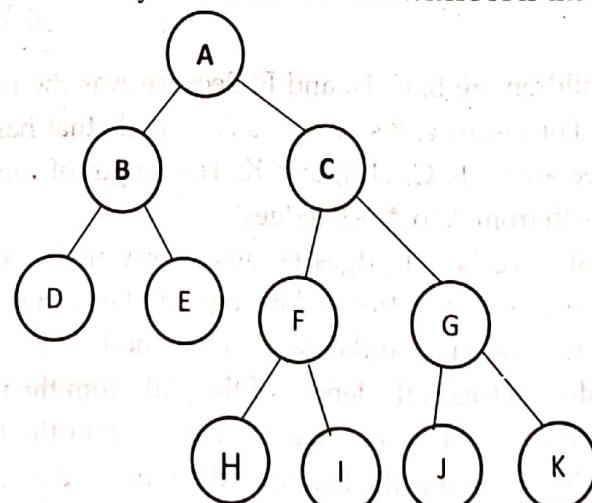
BINARY TREES

A binary tree is a finite set of elements that are either empty or is partitioned into three disjoint subsets. The first subset contains a single element called the **root** of the tree. The other two subsets are themselves binary trees called the **left** and **right sub-trees** of the original tree. A left or right sub tree can be empty. Each element of a binary tree is called a **node** of the tree.

Properties of Binary tree

- Trees are used to represent data in hierarchical form.
- Binary tree is the one in which each node has maximum of two child- node.
- The order of binary tree is '2'.
- Binary tree does not allow duplicate values.
- The maximum number of nodes at level 'l' of a binary tree is 2^{l-1}
- Maximum number of nodes in a binary tree of height 'h' is $2^h - 1$
- A Binary Tree with L leaves has at least $\lceil \log_2 L \rceil + 1$ levels

The following figure shows a binary tree with 11 nodes where A is the root.



Simply, a tree with at most two children for every internal node is called binary tree.

Advantages of Binary Tree:

- Searching in Binary tree become faster
- Maximum and minimum elements can be directly picked up
- It is used for graph traversal and to convert an expression to postfix and prefix forms

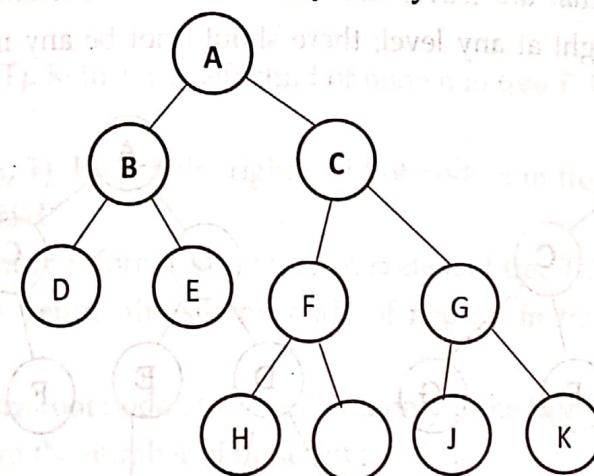
Types of binary tree

Binary tree can be categorized into following 3 types

- Strictly binary tree
- Complete binary tree
- Almost complete binary tree

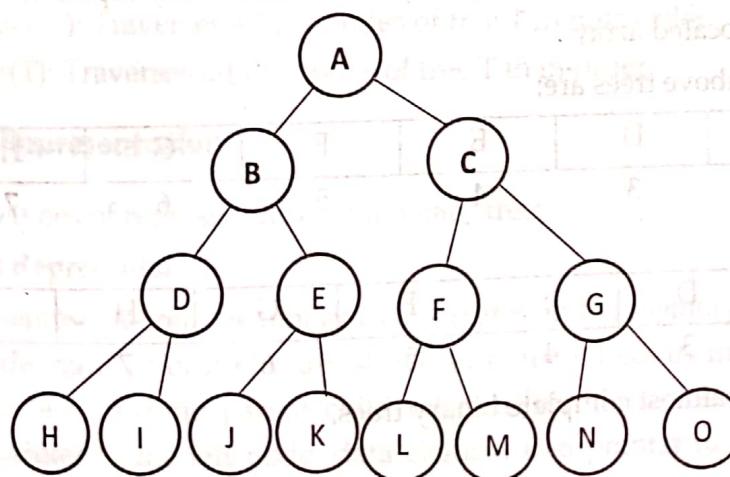
Strictly binary tree

If every non-leaf node in a **binary tree** has nonempty left and right sub-trees, the tree is termed a **strictly binary tree**. Or, to put it another way, all of the nodes in a strictly binary tree are of degree zero or two, never degree one. A **strictly binary tree** with n leaves always contains $(2n - 1)$ nodes. Simply, if every non-leaf (internal) node in a binary tree has nonempty left and right sub-trees, then such a tree is called a **strictly binary tree**.



Complete binary tree

A complete binary tree of depth d is called strictly binary tree if all of whose leaves are at level d . A complete binary tree with depth d has 2^d leaves and $(2^d - 1)$ non-leaf (internal) nodes.



Properties of Full Binary Tree

1. A binary tree of height h with no missing node.
2. All leaves are at level h and all other nodes have two children.
3. All the nodes that are at a level less than h have two children each.
4. If a full binary tree has i internal nodes:
 - Number of leaves $l = i + 1$
 - Total number of nodes $n = 2 * i + 1$
5. If a full binary tree has n nodes:
 - Number of internal nodes $i = (n - 1) / 2$
 - Number of leaves $l = (n + 1) / 2$
6. If a full binary tree has l leaves:
 - Total Number of nodes $n = 2 * l - 1$

Almost complete binary tree

A binary tree of depth d is an almost complete binary tree if:

- Each leaf in the tree is either at level d or at level $d - 1$.
- For any node nd in the tree with a right descendant at level d , all the left descendants of nd that are leaves are also at level d . It means nodes should be present in left to right at any level; there should not be any missing node in the traversal.

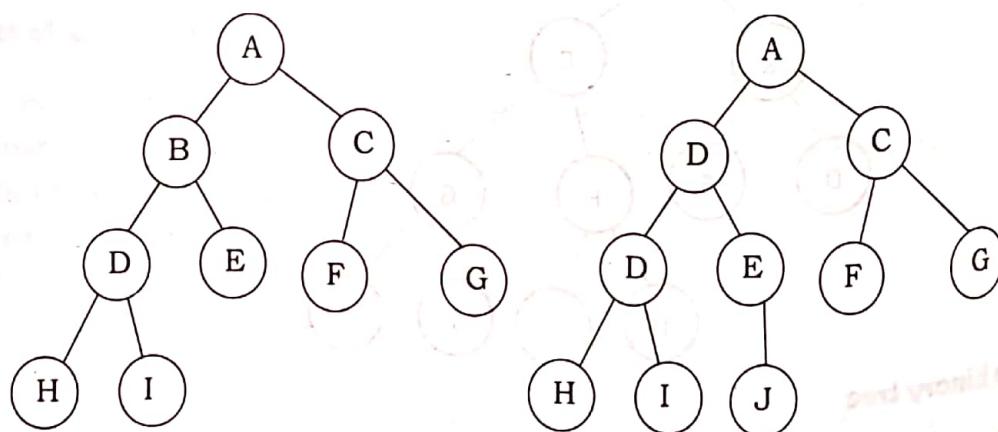


Fig: Almost complete binary trees

Simply, a binary tree is called almost complete binary tree if their array representation has no any null values within the allocated array..

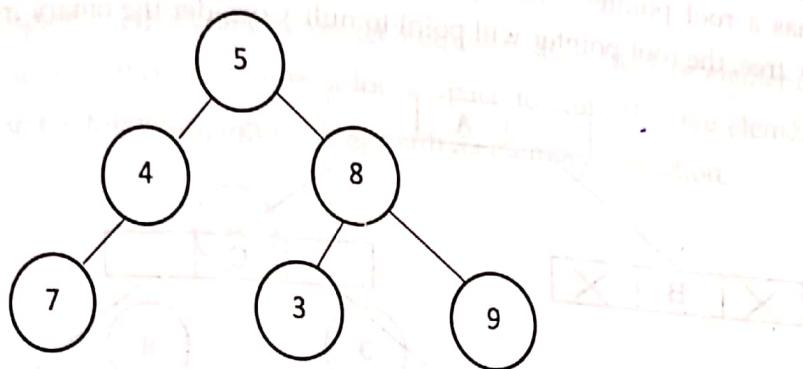
The array representations of above trees are:

A	B	C	D	E	F	G	H	I	J
0	1	2	3	4	5	6	7	8	9

A	B	C	D	E	F	G	H	I	J
0	1	2	3	4	5	6	7	8	9

Hence both of above trees are almost complete binary trees.

Let's take a binary tree as,



Its array representation is:

5	4	8	7	Null	3	9
0	1	2	3	4	5	6

Since array representation of above binary tree contains null value within the array, hence the above binary tree is not almost complete binary tree.

OPERATIONS ON BINARY TREE

- **Father (n, T):** Return the parent node of the node n in tree T. If n is the root, NULL is returned.
- **LeftChild (n, T):** Return the left child of node n in tree T. Return NULL if n does not have a left child.
- **RightChild (n, T):** Return the right child of node n in tree T. Return NULL if n does not have a right child.
- **Info (n, T):** Return information stored in node n of tree T (i.e. Content of a node).
- **Sibling (n, T):** return the sibling node of node n in tree T. Return NULL if n has no sibling.
- **Root (T):** Return root node of a tree if and only if the tree is nonempty.
- **Size (T):** Return the number of nodes in tree T
- **MakeEmpty (T):** Create an empty tree T
- **SetLeft (S, T):** Attach the tree S as the left sub-tree of tree T
- **SetRight (S, T):** Attach the tree S as the right sub-tree of tree T.
- **Preorder (T):** Traverses all the nodes of tree T in pre-order.
- **postorder(T):** Traverses all the nodes of tree T in post-order
- **Inorder(T):** Traverses all the nodes of tree T in in-order.

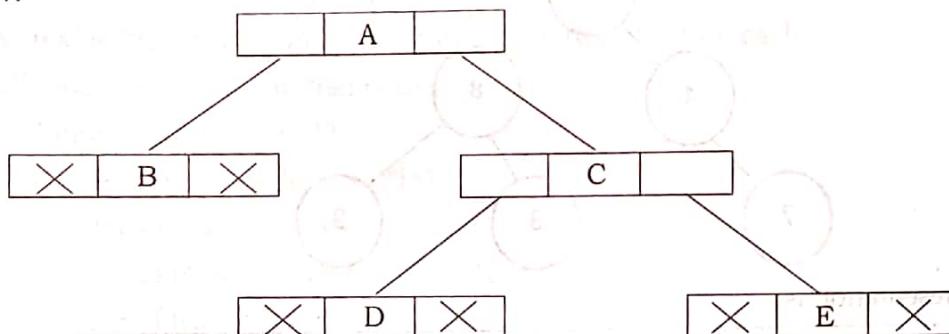
Binary Tree Representation

There are two types of representation of a binary tree:

1. Linked Representation

In this representation, the binary tree is stored in the memory, in the form of a linked list where the number of nodes are stored at non-contiguous memory locations and linked together by inheriting parent child relationship like a tree. Every node contains three parts: pointer to the left node, data element and pointer to the right node. Each binary

tree has a root pointer which points to the root node of the binary tree. In an empty binary tree, the root pointer will point to null. Consider the binary tree given in the figure below.



In the above figure, a tree is seen as the collection of nodes where each node contains three parts: left pointer, data element and right pointer. Left pointer stores the address of the left child while the right pointer stores the address of the right child. The leaf node contains NULL in its left and right pointers.

The following image shows about how the memory will be allocated for the binary tree by using linked representation. There is a special pointer maintained in the memory which points to the root node of the tree. Every node in the tree contains the address of its left and right child. Leaf node contains NULL in its left and right pointers.

Structure of linked list implementation of binary tree

The structure of binary tree is as below:

root

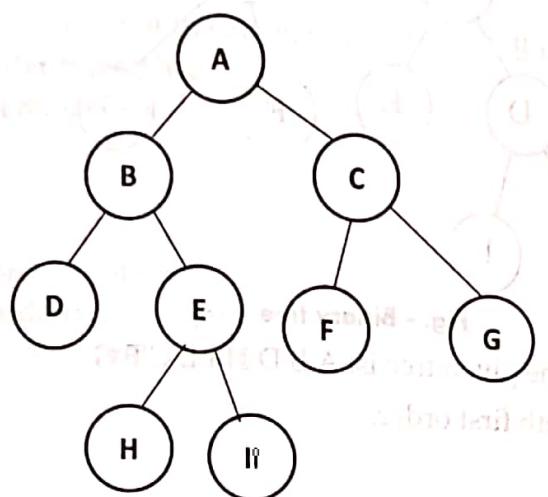
	Left	Data	Right
3	-1	D	-1
1			
2			
3	9	A	7
4			
5	-1	E	-1
6			
7	1	C	5
8			
9	-1	B	-1
10			

```

struct bnode
{
    int info;
    struct bnode *left;
    struct bnode *right;
};
struct bnode *root=NULL;
  
```

2. Sequential Representation

This is the simplest memory allocation technique to store the tree elements but it is an inefficient technique since it requires a lot of space to store the tree elements. A binary tree is shown in the following figure along with its memory allocation.



A	B	C	D	E	F	G			H	I
1	2	3	4	5	6	7	8	9	10	11

In this representation, an array is used to store the tree elements. Size of the array will be equal to the number of nodes present in the tree. The root node of the tree will be present at the 1st index of the array. If a node is stored at ith index then its left and right children will be stored at 2i and 2i+1 location. If the 1st index of the array i.e. tree[1] is 0, it means that the tree is empty.

Binary Tree Traversal Techniques

Traversing a tree means visiting each node in a specified order. This process is not as commonly used as finding, inserting, and deleting nodes. One reason for this is that traversal is not particularly fast. But traversing a tree is useful in some circumstances and the algorithm is interesting. The tree traversal is a way in which each node in the tree is visited exactly once in a symmetric manner. There are three popular methods of traversal:

- Pre-order traversal
- In-order traversal
- Post-order traversal

The order most commonly used for binary search trees is in-order.

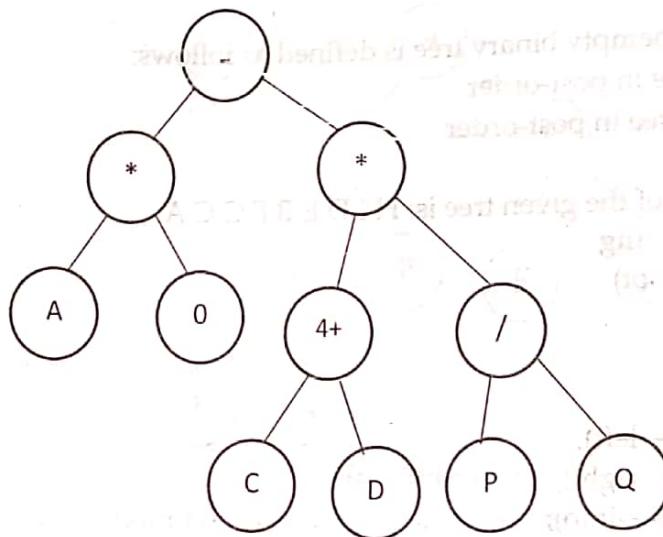
Pre-order traversal

In pre-order traversing first **root** is visited followed by **left** sub-tree and **right** sub-tree.

The pre-order traversal of a nonempty binary tree is defined as follows:

- Visit the root node
- Traverse the left sub-tree in pre-order

Step 3:



The expression tree for a given expression can be built recursively from the following rules:

- The expression tree for a single operand is a single root node that contains it.
- If E_1 and E_2 are expressions represented by expression trees T_1 and T_2 and if op is an operator, then the expression tree for the expression $E_1 \ op \ E_2$ is the tree with root node containing op and sub-trees T_1 and T_2 .

An expression has three representations, depending upon which traversal algorithm is used to traverse its tree. The preorder traversal produces the prefix representation, the in-order traversal produces the infix representation, and the post-order traversal produces the postfix representation of the expression. The postfix representation is also called reverse Polish notation or RPN.

BINARY SEARCH TREE (BST)

A binary search tree (BST) is a binary tree that is either empty or in which every node contains a key (value) and satisfies the following conditions:

- All keys in the left sub-tree of the root are smaller than the key in the root node
- All keys in the right sub-tree of the root are greater than the key in the root node
- The left and right sub-trees of the root are again binary search trees

For a binary tree to be a binary search tree, the data of all the nodes in the left sub-tree of the root node should be less than the data of the root. The data of all the nodes in the right sub-tree of the root node should be greater than equal to the data of the root. As a result, the leaves on the farthest left of the tree have the lowest values, whereas the leaves on the right of the tree have the greatest values.

Example 1: Construct BST from following data items

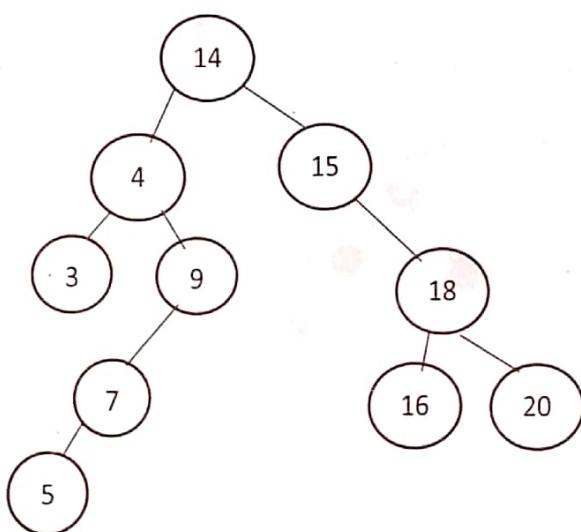
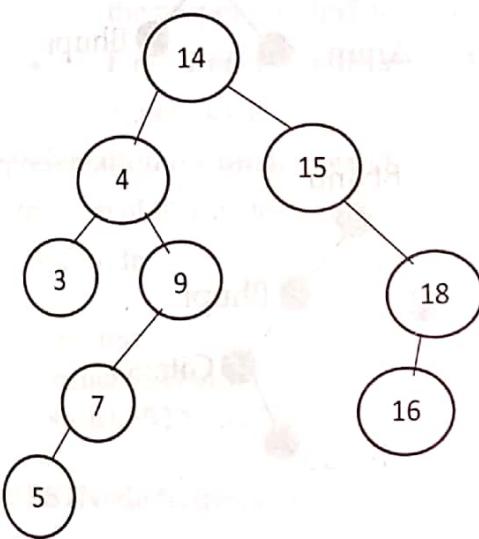
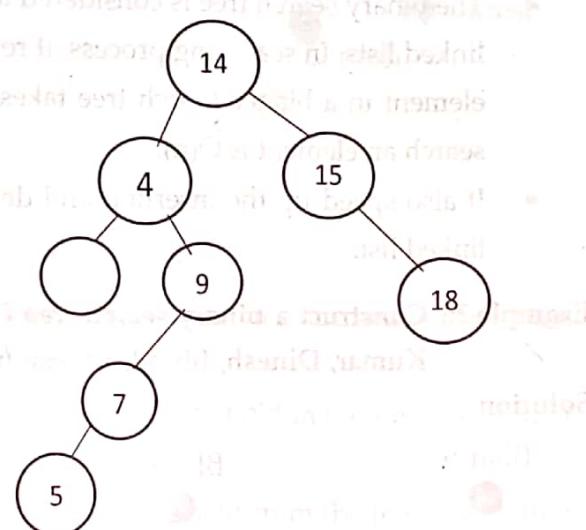
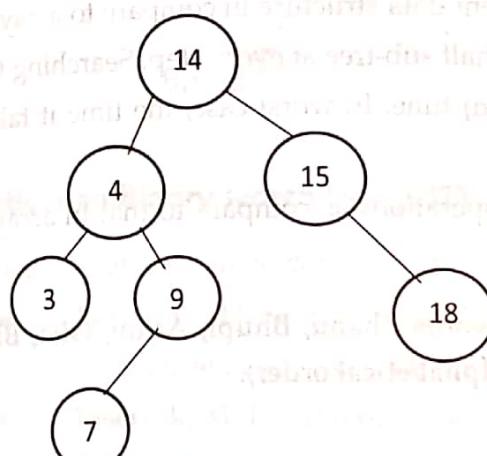
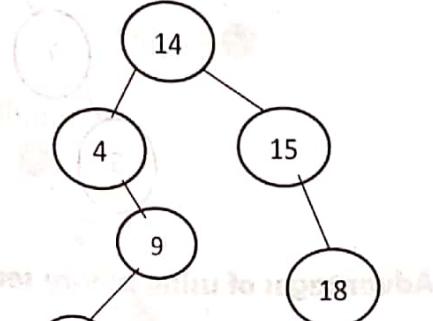
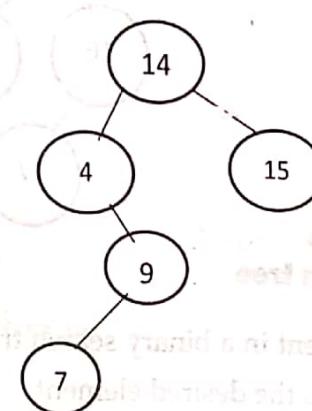
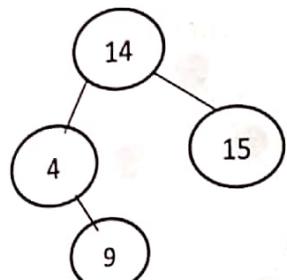
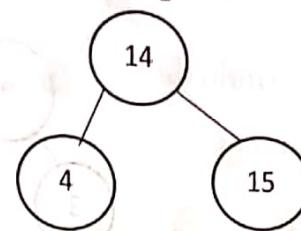
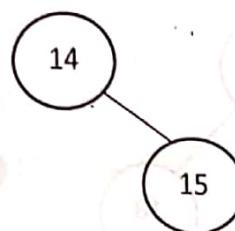
14, 15, 4, 9, 7, 18, 3, 5, 16, 4, 20, 17, 9

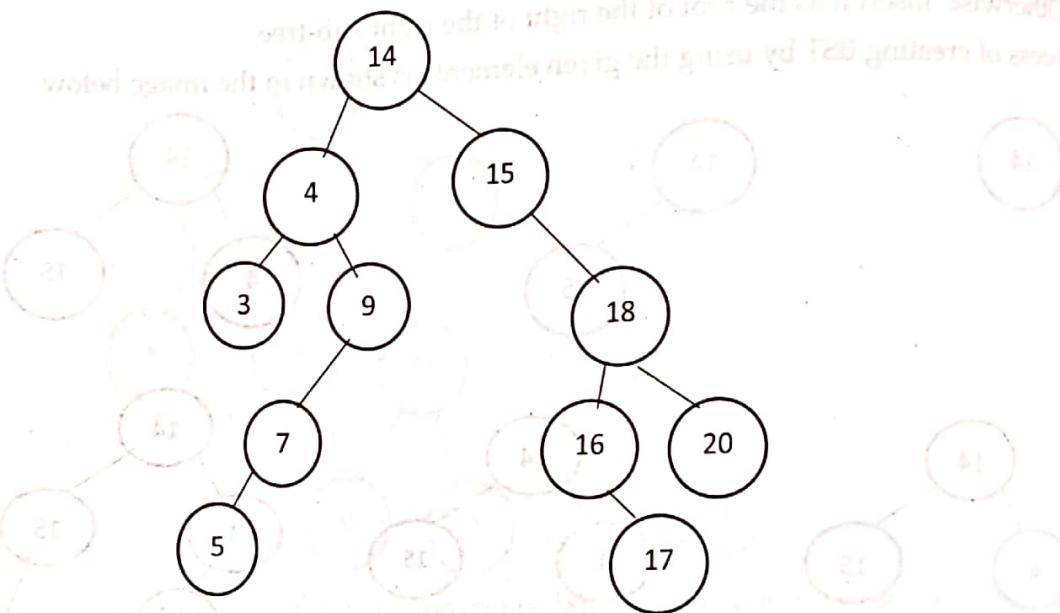
Solution:

1. Insert 14 into the tree as the root of the tree.
2. Read the next element, if it is lesser than the root node element; insert it as the root of the left sub-tree.

3. Otherwise, insert it as the root of the right of the right sub-tree.

The process of creating BST by using the given elements is shown in the image below.





Advantages of using binary search tree

- Searching become very efficient in a binary search tree since, we get a hint at each step, about which sub-tree contains the desired element.
- The binary search tree is considered as efficient data structure in compare to arrays and linked lists. In searching process, it removes half sub-tree at every step. Searching for an element in a binary search tree takes $O(\log_2 n)$ time. In worst case, the time it takes to search an element is $O(n)$.
- It also speed up the insertion and deletion operations as compare to that in array and linked list.

Example 2: Construct a binary search tree for the words Bhanu, Bhupi, Arjun, Gita, Bindu, Kumar, Dinesh, Binod, Umesh (using alphabetical order).

Solution:

Bhanu

Bhanu

Bhanu

Bhupi

Bhupi

Bhupi

Arjun

Bhanu

Arjun

Bhupi

Bhanu

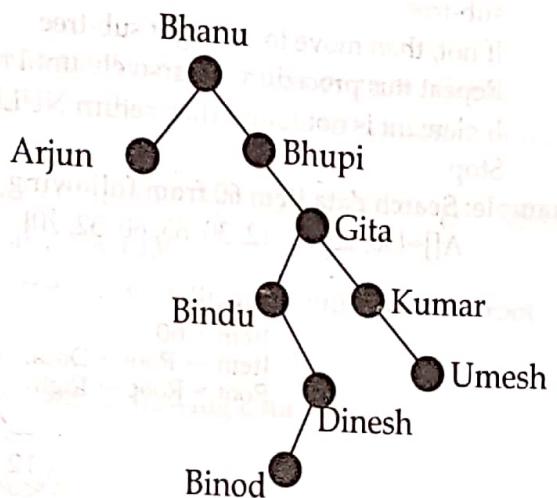
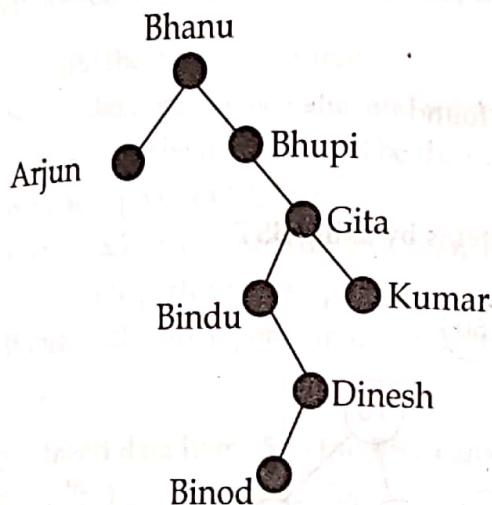
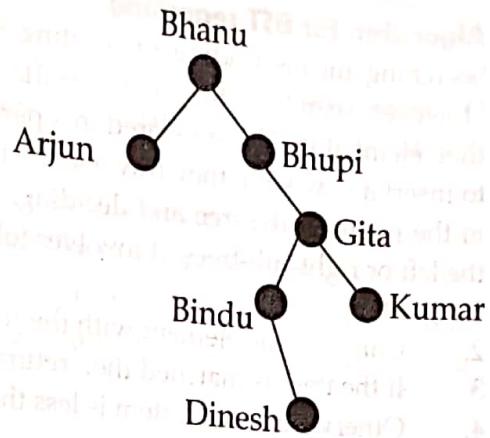
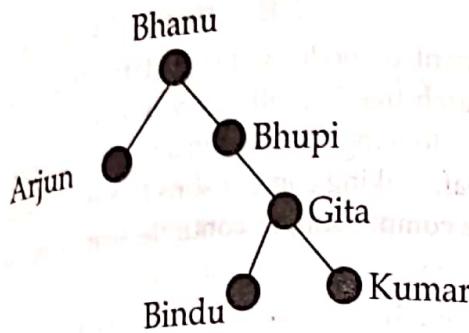
Gita

Bhupi

Arjun

Bhupi

Bindu



Operations on Binary search tree (BST)

Following operations can be done in BST:

- **Search (k, T):** Search for key k in the tree T. If k is found in some node of tree then return true otherwise return false.
- **Insert (k, T):** Insert a new node with value k in the info field in the tree T such that the property of BST is maintained.
- **Delete (k, T):** Delete a node with value k in the info field from the tree T such that the property of BST is maintained.
- **FindMin(T), FindMax(T):** Find minimum and maximum element from the given nonempty BST.

Representation for Binary search tree

The structure of BST is described as below:

```

struct BSTNode
{
    int info;
    struct BSTNode *left;
    struct BSTNode *right;
};
struct BSTNode *root=NULL;
  
```

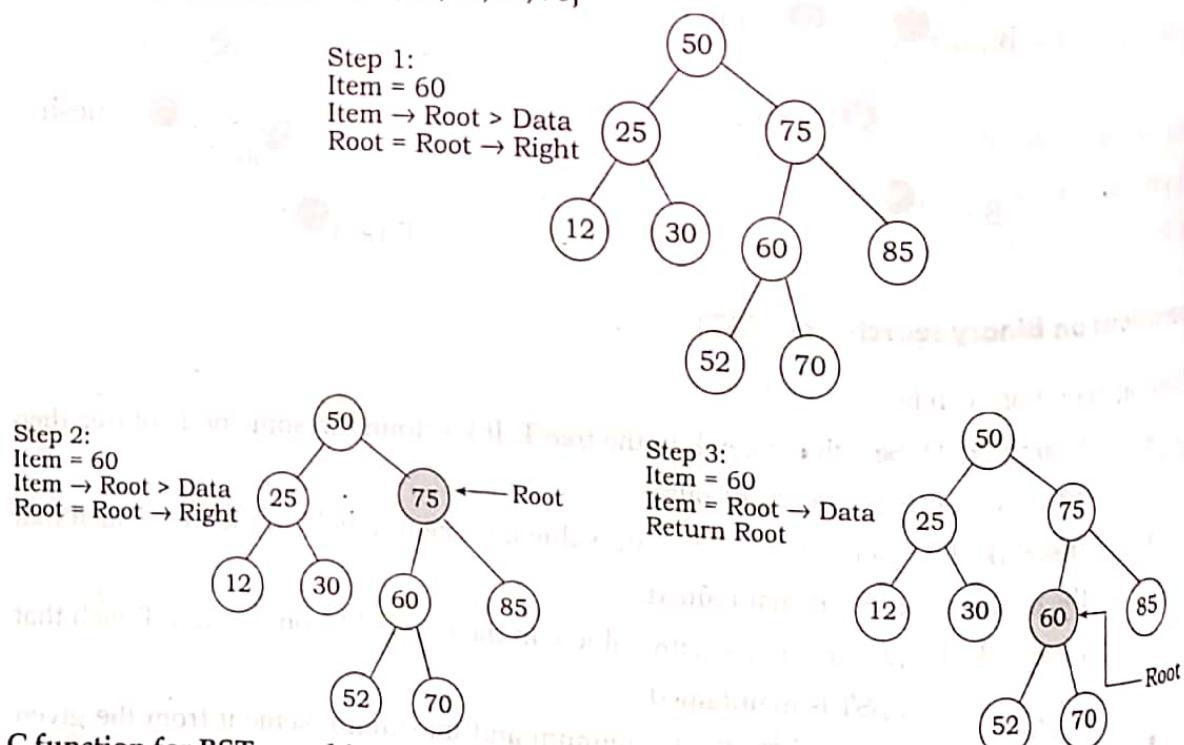
Algorithm for BST searching

Searching means finding or locating some specific element or node within a data structure. However, searching for some specific node in binary search tree is pretty easy due to the fact that, element in BST are stored in a particular order. When looking for a key in a tree (or a place to insert a new key), they traverse the tree from root to leaf, making comparisons to keys stored in the nodes of the tree and deciding, on the basis of the comparison, to continue searching in the left or right sub-trees. It involves following steps:

1. Start
2. Compare the element with the root of the tree.
3. If the item is matched then return the location of the node.
4. Otherwise check if item is less than the element present on root, if so then move to the left sub-tree.
5. If not, then move to the right sub-tree.
6. Repeat this procedure recursively until match found.
7. If element is not found then return NULL
8. Stop

Example: Search data item 60 from following data items by using BST

A[] = {50, 25, 75, 12, 30, 85, 60, 52, 70}



C function for BST searching

```
BSTNode * BinSearch(struct BSTNode *root, int key)
{
    if(root == NULL)
        return NULL;
    else if (key == root->info)
        return root;
    else if(key < root->info)
        return BinSearch(root->left, key);
    else
        return BinSearch(root->right, key);
}
```

Insertion of a node in BST

Insert function is used to add a new element in a binary search tree at appropriate location. Insert function is to be designed in such a way that, it must not violate the property of binary search tree at each value.

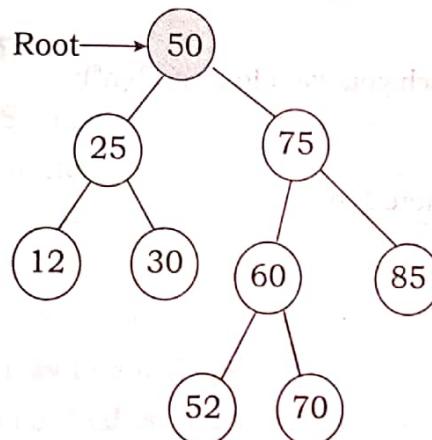
In a binary search tree, the insertion operation is performed with $O(\log n)$ time complexity. In binary search tree, new node is always inserted as a leaf node. To insert a new item in a tree, we must first verify that its key is different from those of existing elements. To do this a search is carried out. If the search is unsuccessful, then item is inserted. The insertion operation is performed as follows:

1. Start
2. Allocate the memory for tree.
3. Set the data part to the value and set the left and right pointer of tree, point to NULL.
4. If the item to be inserted, will be the first element of the tree, then the left and right of this node will point to NULL.
5. Else, check if the item is less than the root element of the tree, if this is true, then recursively perform this operation with the left of the root.
6. If this is false, then perform this operation recursively with the right sub-tree of the root.
7. Stop

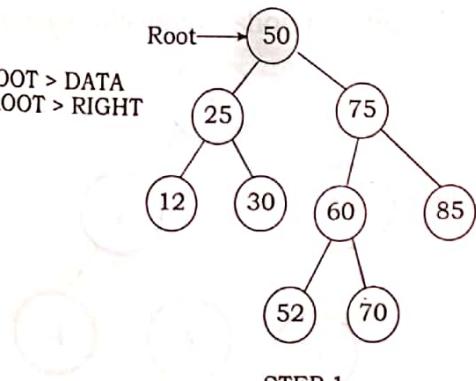
Example: Insert data item 95 to the BST constructed from following data

$A[] = \{50, 25, 75, 12, 30, 85, 60, 52, 70\}$

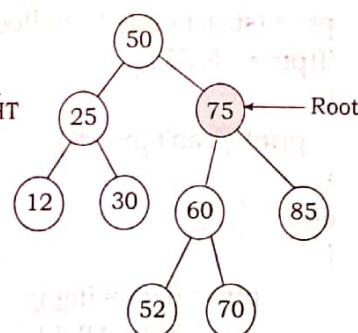
Solution: At first construct BST from above data items as,

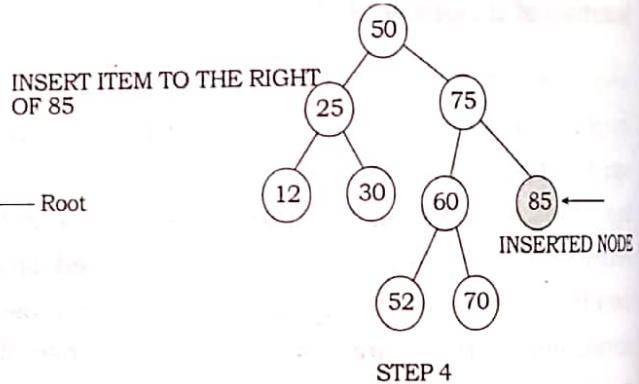
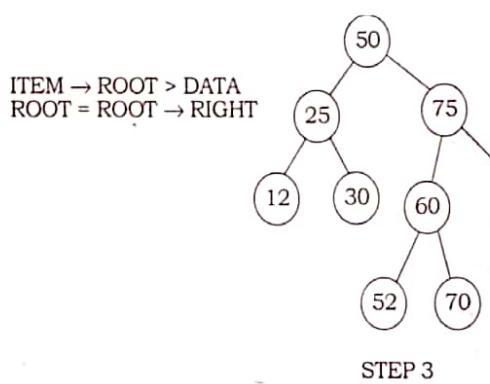


ITEM > ROOT > DATA
ROOT = ROOT > RIGHT



ITEM → ROOT → DATA
ROOT = ROOT → RIGHT



**BST insertion function**

```
#include<stdio.h>
#include<stdlib.h>
void insert(int);
struct node
{
    int data;
    struct node *left;
    struct node *right;
};
struct node *root;
void main ()
{
    int choice, item;
    do
    {
        printf("\nEnter the item which you want to insert?\n");
        scanf("%d", &item);
        insert(item);
        printf("\n Press 1 to insert more ?\n");
        scanf("%d", &choice);
    }while(choice == 1);
}

void insert(int item)
{
    struct node *ptr, *parentptr, *nodeptr;
    ptr = (struct node *) malloc(sizeof (struct node));
    if(ptr == NULL)
    {
        printf("can't insert");
    }
    else
    {
        ptr -> data = item;
        ptr -> left = NULL;
        ptr -> right = NULL;
        if(root == NULL)
```

```

root = ptr;
root->left = NULL;
root->right = NULL;

```

} else

```

parentptr = NULL;
nodeptr = root;
while(nodeptr != NULL)
{
    parentptr = nodeptr;
    if(item < nodeptr->data)
        nodeptr = nodeptr->left;
    else
        nodeptr = nodeptr->right;
    if(item < parentptr->data)
        parentptr->left = ptr;
    else
        parentptr->right = ptr;
}

```

printf("Node Inserted");

}

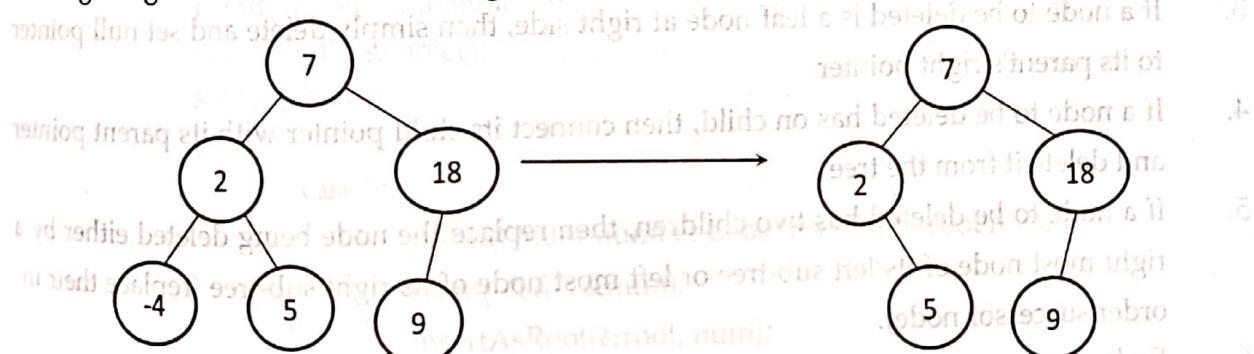
}

Deleting a node from the BST

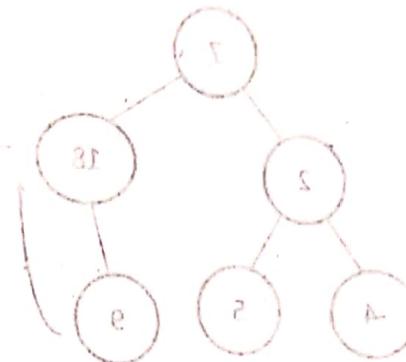
While deleting a node from BST, first we need to perform a search for deleting element. Once we have found the node to be deleted, there may be three cases shown in given below. The level of complexity in performing the operation depends on the position of the node to be deleted in the tree. It is by far more difficult to delete a node having two sub-trees than to delete a leaf; the complexity of the deletion algorithm is proportional to the number of children the node has.

1. The node to be deleted may be a leaf node

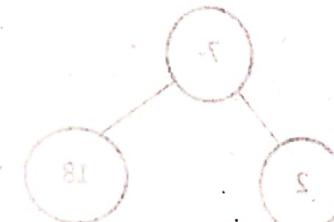
The node is a leaf; it has no children. This is the easiest case. The appropriate reference of its parent is set to null and the space occupied by the deleted node is later claimed by the garbage collector as shown in fig below;



Suppose node to be deleted is -4



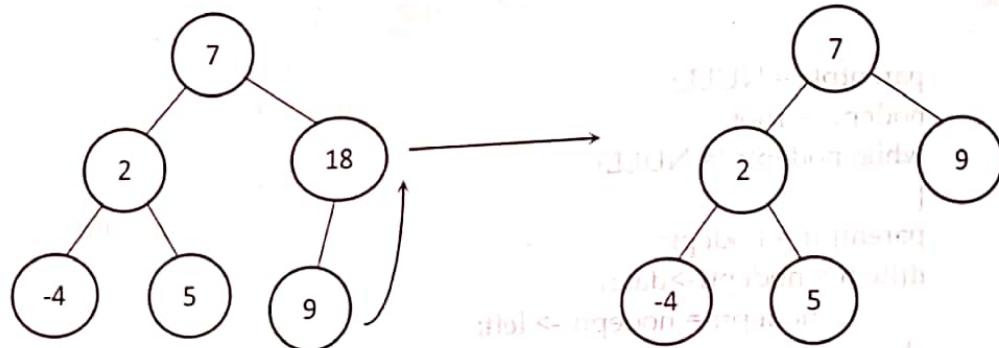
OR



2. The node to be deleted has one child

In this case the child of the node to be deleted is appended to its parent node. In this way, the node's children are lifted up by one level as shown in fig below:

Suppose node to be deleted is 18

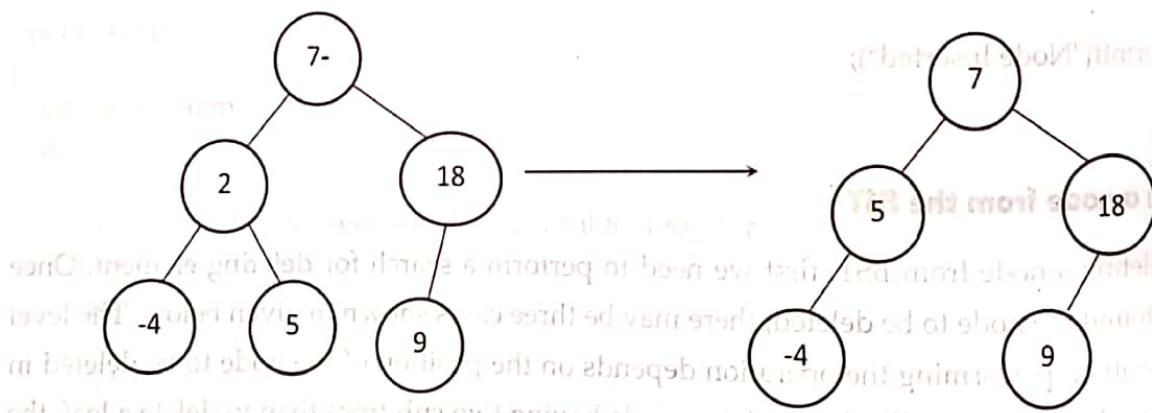


3. The node to be deleted has two children

In this case node to be deleted is replaced by its in-order successor node.

OR

The node to be deleted is either replaced by its right sub-tree's leftmost node or its left sub-tree's rightmost node.



Suppose node to be deleted is 2

General algorithm to delete a node from a BST

1. Start
2. If a node to be deleted is a leaf node at left side, then simply delete and set null pointer to its parent's left pointer.
3. If a node to be deleted is a leaf node at right side, then simply delete and set null pointer to its parent's right pointer
4. If a node to be deleted has one child, then connect its child pointer with its parent pointer and delete it from the tree
5. If a node to be deleted has two children, then replace the node being deleted either by right most node of its left sub-tree or left most node of its right sub-tree (replace their in-order successor node).
6. End

Complete menu driven program in C to perform various operations in Binary tree (in BST)

```

#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
struct BTreenode
{
    int info;
    struct BTreenode *left;
    struct BTreenode *right;
};
typedef struct BTreenode BinaryTree;
void insertAsRoot(BinaryTree **, int);
void insertElement(BinaryTree *, int);
void PreorderTraversal(BinaryTree *);
void InorderTraversal(BinaryTree *);
void PostorderTraversal(BinaryTree *);
void PrintLeafNodes(BinaryTree *);
void PrintOneChildNode(BinaryTree *);
void PrintTwoChildNode(BinaryTree *);
int CountNodes(BinaryTree *);
int main()
{
    BinaryTree *root=NULL;
    int choice;
    int num;
    do{
        printf("-----Menu for Binary Tree Operations-----\n");
        printf("1: Insert as root node in BST\n 2: Insert in non-empty BST\n 3: Pre-order Traversal\n 4: In-order traversal\n 5: Post-order Traversal\n 6: Print leaf nodes of BST\n 7: Print nodes with one child of BST\n 8: Print nodes with two child nodes\n 9: Count nodes of BST\n 10: Exit\n");
        printf("Enter your choice");
        scanf("%d", &choice);
        switch(choice)
        {
            case 1:
                printf("Enter number to be inserted as root node");
                scanf("%d", &num);
                insertAsRoot(&root, num);
                break;
        }
    }
    while(choice!=10);
}
void insertAsRoot(BinaryTree **root, int num)
{
    if(*root==NULL)
    {
        *root=(BinaryTree *)malloc(sizeof(BinaryTree));
        (*root)->info=num;
        (*root)->left=NULL;
        (*root)->right=NULL;
    }
    else
    {
        if(num<(*root)->info)
        {
            insertAsRoot(&(*root)->left, num);
        }
        else
        {
            insertAsRoot(&(*root)->right, num);
        }
    }
}
void insertElement(BinaryTree *root, int num)
{
    if(root==NULL)
    {
        root=(BinaryTree *)malloc(sizeof(BinaryTree));
        root->info=num;
        root->left=NULL;
        root->right=NULL;
    }
    else
    {
        if(num<root->info)
        {
            insertElement(&root->left, num);
        }
        else
        {
            insertElement(&root->right, num);
        }
    }
}
void PreorderTraversal(BinaryTree *root)
{
    if(root!=NULL)
    {
        printf("%d ", root->info);
        PreorderTraversal(root->left);
        PreorderTraversal(root->right);
    }
}
void InorderTraversal(BinaryTree *root)
{
    if(root!=NULL)
    {
        InorderTraversal(root->left);
        printf("%d ", root->info);
        InorderTraversal(root->right);
    }
}
void PostorderTraversal(BinaryTree *root)
{
    if(root!=NULL)
    {
        PostorderTraversal(root->left);
        PostorderTraversal(root->right);
        printf("%d ", root->info);
    }
}
void PrintLeafNodes(BinaryTree *root)
{
    if(root!=NULL)
    {
        if((root->left==NULL)&&(root->right==NULL))
        {
            printf("%d ", root->info);
        }
        PrintLeafNodes(root->left);
        PrintLeafNodes(root->right);
    }
}
void PrintOneChildNode(BinaryTree *root)
{
    if(root!=NULL)
    {
        if((root->left==NULL)&&(root->right!=NULL))
        {
            printf("%d ", root->info);
        }
        if((root->left!=NULL)&&(root->right==NULL))
        {
            printf("%d ", root->info);
        }
        PrintOneChildNode(root->left);
        PrintOneChildNode(root->right);
    }
}
void PrintTwoChildNode(BinaryTree *root)
{
    if(root!=NULL)
    {
        PrintTwoChildNode(root->left);
        PrintTwoChildNode(root->right);
        printf("%d ", root->info);
    }
}
int CountNodes(BinaryTree *root)
{
    if(root==NULL)
    {
        return 0;
    }
    else
    {
        return 1+CountNodes(root->left)+CountNodes(root->right);
    }
}

```

```

case 2: printf("Enter number to be inserted:");
scanf("%d", &num);
insertElement(root, num);

case 3:
    printf("Enter number to be inserted:");
    scanf("%d", &num);
    insertElement(root, num);

case 4:
    printf("Elements of BST in Pre-order are\n");
    PreorderTraversal(root);
    break;

case 5:
    printf("Elements of BST in In-order are\n");
    InorderTraversal(root);
    break;

case 6:
    printf("Elements of BST with no children are\n");
    PrintNoneChildNode(root);
    break;

case 7:
    printf("Elements of BST with one child are\n");
    PrintOneChildNode(root);
    break;

case 8:
    printf("Elements of BST with two children are\n");
    PrintTwoChildNode(root);
    break;

case 9:
    printf("Total nodes of BST are (%d)", CountNodes(root));
    break;

case 10:
    if (root == NULL)
        break;
    else
        printf("Root node is %d", root->data);
        default:
            exit(1);
}

while(choice<11):
    printf("Invalid choice");
    break;
return 0;
}

```

```

void insertAsRoot(BinaryTree **head, int n)
{
    BinaryTree *newnode;
    newnode=(BinaryTree*)malloc(sizeof(BinaryTree));
    newnode->info=n;
    newnode->left= newnode->right=NULL;
    *head=newnode;
}

void insertElement(BinaryTree *head, int n)
{
    BinaryTree *newnode;
    if(head->info>n)
    {
        if(head->left==NULL)
        {
            newnode=(BinaryTree*)malloc(sizeof(BinaryTree));
            newnode->info=n;
            newnode->left= newnode->right=NULL;
            head->left=newnode;
        }
        else
            insertElement(head->left, n);
    }
    else
    {
        if(head->right==NULL)
        {
            newnode=(BinaryTree*)malloc(sizeof(BinaryTree));
            newnode->info=n;
            newnode->left= newnode->right=NULL;
            head->right=newnode;
        }
        else
            insertElement(head->right, n);
    }
}

void PreorderTraversal(BinaryTree *head)

```

```

{
    if(head!=NULL)
    {
        printf("%d\t", head->info);
        PreorderTraversal(head->left);
        PreorderTraversal(head->right);
    }
}

void InorderTraversal(BinaryTree *head)
{
    if(head!=NULL)
    {
        InorderTraversal(head->left);
        printf("%d\t", head->info);
        InorderTraversal(head->right);
    }
}

void PostorderTraversal(BinaryTree *head)
{
    if(head!=NULL)
    {
        PostorderTraversal(head->left);
        PostorderTraversal(head->right);
        printf("%d\t", head->info);
    }
}

void PrintLeafNodes(BinaryTree *head)
{
    if(head!=NULL)
    {
        if (head->left==NULL && head->right==NULL)
        {
            printf("%d\t", head->info);
        }
        PrintLeafNodes(head->left);
    }
}

```

PrintLeafNodes(head→right);

```
void PrintOneChildNode(BinaryTree *head)
{
    if(head!=NULL)
    {
        if (head→left==NULL && head→right!=NULL) || | head→left!=NULL &&
head→right==NULL)
            printf("%d\t", head→info);
        PrintOneChildNode(head→left);
        PrintOneChildNode(head→right);
    }
}
```

void PrintTwoChildNode(BinaryTree *head)

```
{ if(head!=NULL)
{
    if (head→right!=NULL && head→left!=NULL)
    {
        printf("%d\t", head→info);
        PrintTwoChildNode(head→left);
        PrintTwoChildNode(head→right);
    }
}
```

int CountNodes(BinaryTree *head)

```
{ if(head==NULL)
    return 0;
else
    return (1+ CountNodes(head→left) + CountNodes(head→right)); }
```

HUFFMAN ALGORITHM

"Huffman algorithm is a method for building an extended binary tree with a minimum weighted path length from a set of given weights."

This method is mainly applicable to many forms of data transmission. In 1951, David Huffman found the "most efficient method of representing numbers, letters, and other symbols using binary code". Now standard method used for data compression. In Huffman Algorithm, a set of nodes assigned with values if fed to the algorithm.

Initially 2 nodes are considered and their sum forms their parent node. When a new element is considered, it can be added to the tree. Its value and the previously calculated sum of the tree are used to form the new node which in turn becomes their parent.

Huffman coding is a lossless data compression algorithm. In this algorithm, a variable-length code is assigned to input different characters. The code length is related to how frequently characters are used. Most frequent characters have the smallest codes and longer codes for least frequent characters. There are mainly two parts. First one to create a Huffman tree, and another one to traverse the tree to find codes. Let us take any four characters and their frequencies, and sort this list by increasing frequency.

Since to represent 4 characters the 2 bit is sufficient thus take initially two bits for each character this is called fixed length character.

Character	frequencies
A	03
T	07
E	10
O	05

Now sort these characters according to their frequencies in non-decreasing order.

Character	frequencies	Code
A	03	00
O	05	01
T	07	10
E	10	11

Here before using Huffman algorithm the total number of bits required is $nb = 3*2 + 5*2 + 7*2 + 10*2 = 06 + 10 + 14 + 20 = 50$ bits

Now using Huffman man algorithm as,

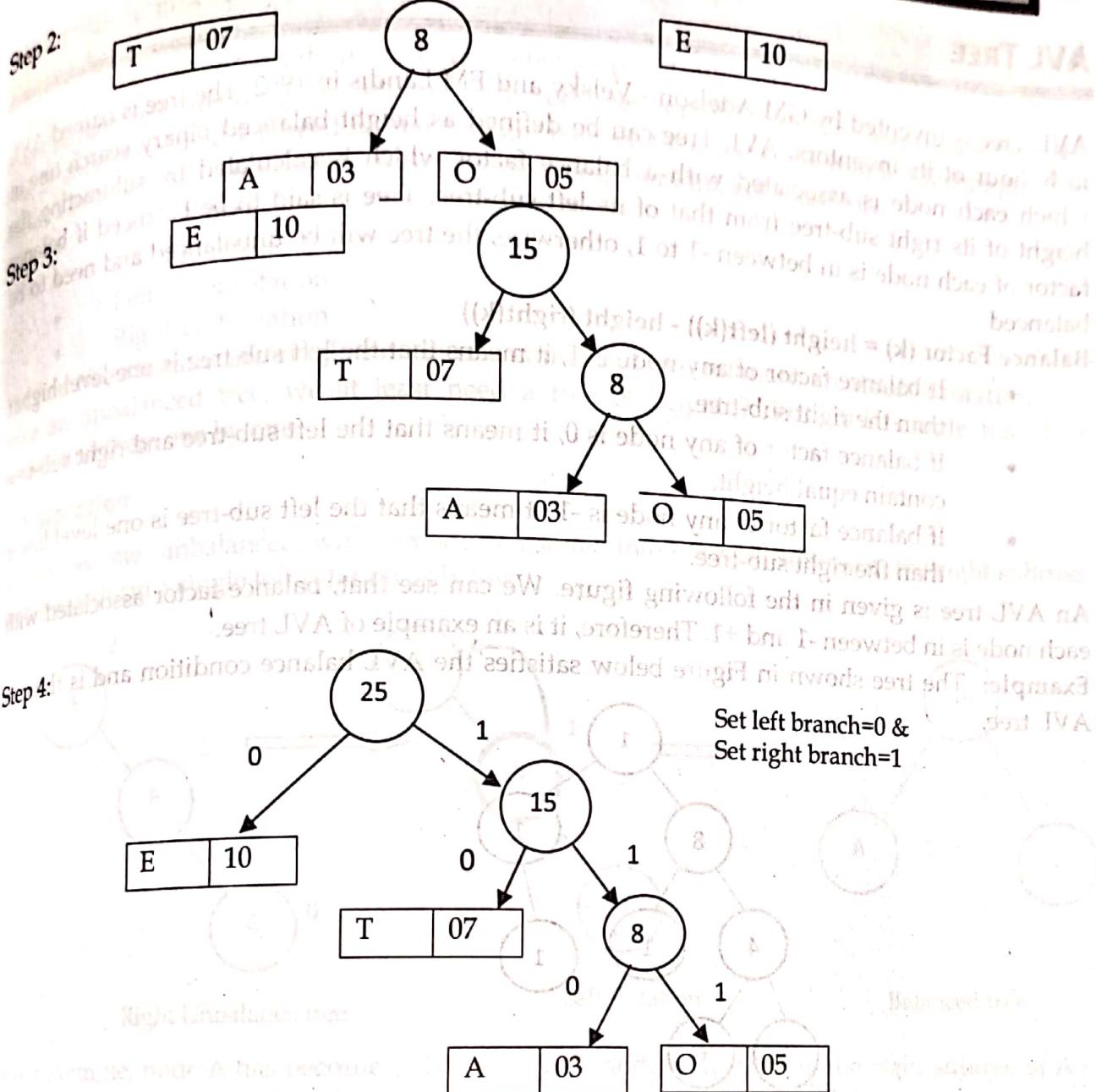
Step 1:

A	3
---	---

O	5
---	---

T	7
---	---

E	10
---	----



Now from variable length code we get following code sequence.

Character	frequencies	Code
A	03	110
O	05	111
T	07	10
E	10	0

Thus after using Huffman algorithm the total number of bits required is:

$$\begin{aligned}
 \text{Number of bits} &= 3*3 + 5*3 + 7*2 + 10*1 \\
 &= 09 + 15 + 14 + 10 \\
 &= 48 \text{ bits} \\
 &= (50 - 48) / 50 * 100\% \\
 &= 4\%
 \end{aligned}$$

Since in this small example, we can save about 4% of space by using Huffman algorithm. If we take large example with a lot of characters and their frequencies, we can save a lot of space.

AVL TREE

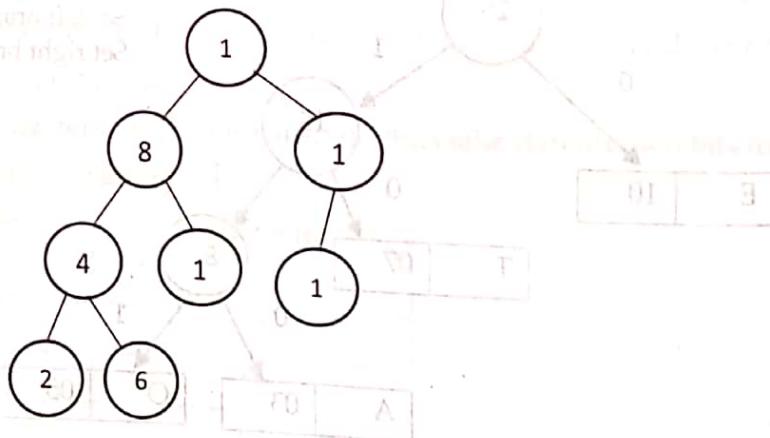
AVL Tree is invented by GM Adelson - Velsky and EM Landis in 1962. The tree is named AVL in honour of its inventors. AVL Tree can be defined as height balanced binary search tree in which each node is associated with a balance factor which is calculated by subtracting the height of its right sub-tree from that of its left sub-tree. Tree is said to be balanced if balance factor of each node is in between -1 to 1, otherwise, the tree will be unbalanced and need to be balanced.

Balance Factor (k) = height (left(k)) - height (right(k))

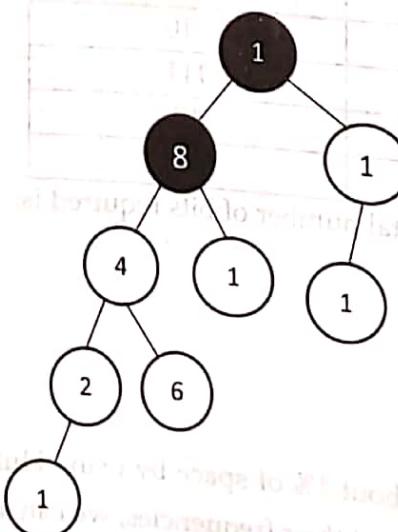
- If balance factor of any node is 1, it means that the left sub-tree is one level higher than the right sub-tree.
- If balance factor of any node is 0, it means that the left sub-tree and right sub-tree contain equal height.
- If balance factor of any node is -1, it means that the left sub-tree is one level lower than the right sub-tree.

An AVL tree is given in the following figure. We can see that, balance factor associated with each node is in between -1 and +1. Therefore, it is an example of AVL tree.

Example: The tree shown in Figure below satisfies the AVL balance condition and is thus an AVL tree.



The tree shown in Figure below, which results from inserting 1, using the usual algorithm, is not an AVL tree because the darkened nodes have left sub trees whose heights are 2 larger than their right sub trees.



How to make non AVL tree to AVL tree?

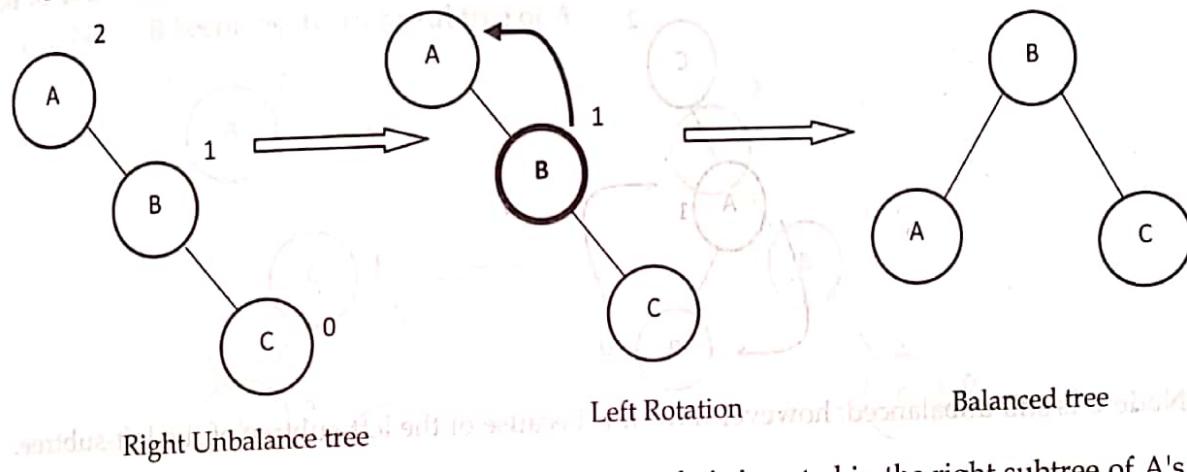
We now that, Balance Factor = height (left subtree) - height (right subtree). If the difference in the height of left and right sub-trees is more than 1, or less than 0 then the tree is unbalanced and we need to make height balance by using some rotation techniques. To balance itself, an AVL tree may perform the following four kinds of rotations:

- Left rotation
- Right rotation
- Left-Right rotation
- Right-Left rotation

The first two rotations are **single rotations** and the next two rotations are **double rotations**. To have an unbalanced tree, we at least need a tree of height 2. With this simple tree, let's understand them one by one.

Left Rotation

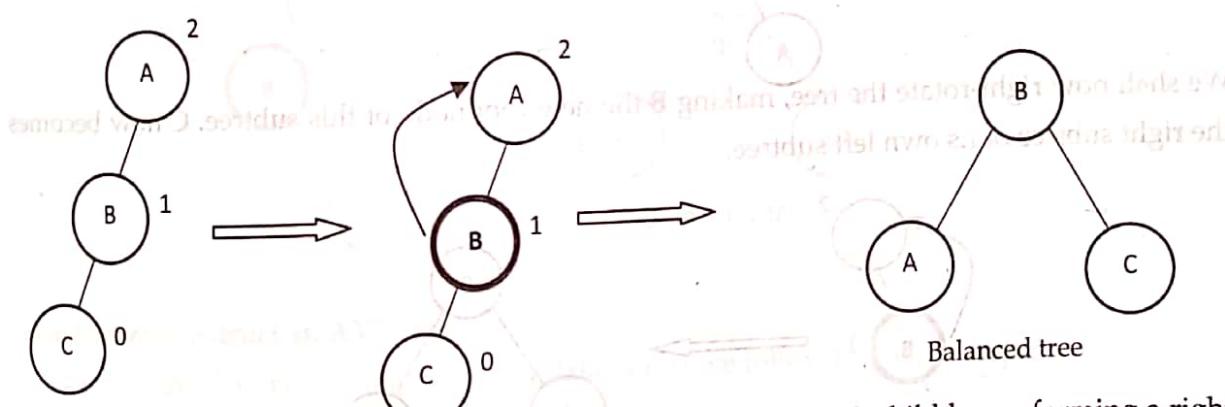
If a tree becomes unbalanced, when a node is inserted into the right subtree of the right subtree, then we perform a single left rotation as below,



In our example, node A has become unbalanced as a node is inserted in the right subtree of A's right subtree. We perform the left rotation by making A the left-subtree of B.

Right Rotation

AVL tree may become unbalanced, if a node is inserted in the left subtree of the left subtree. The tree then needs a right rotation.

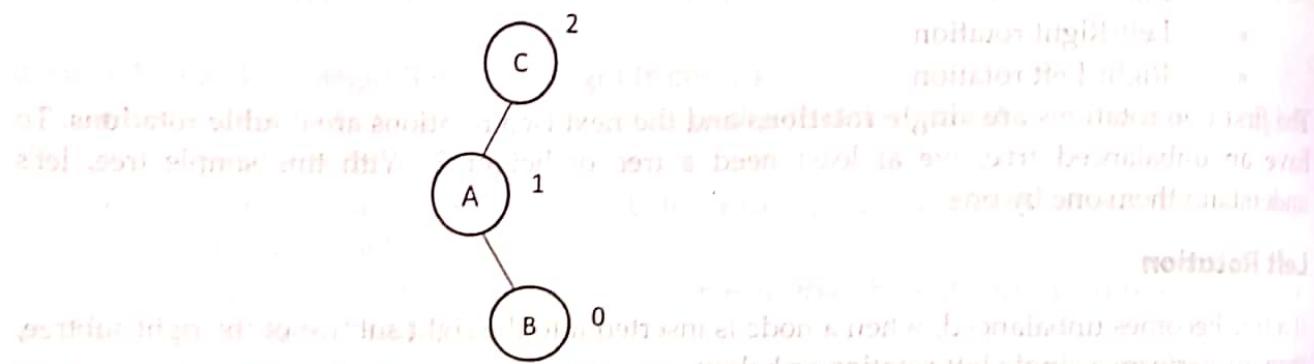


As depicted, the unbalanced node becomes the right child of its left child by performing a right rotation.

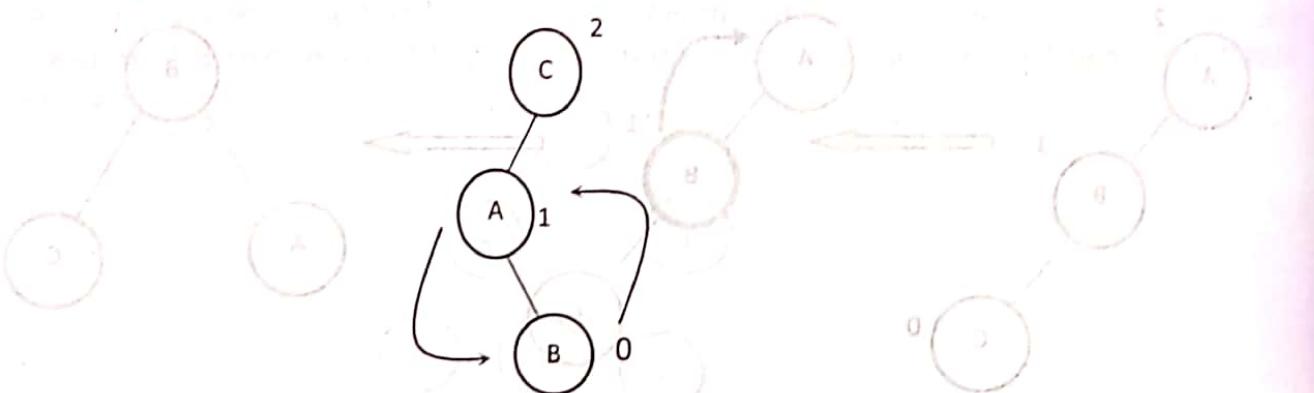
Left-Right Rotation

Double rotations are slightly complex version of already explained versions of rotations. To understand them better, we should take note of each action performed while rotation. Let's first check how to perform Left-Right rotation. A left-right rotation is a combination of left rotation followed by right rotation.

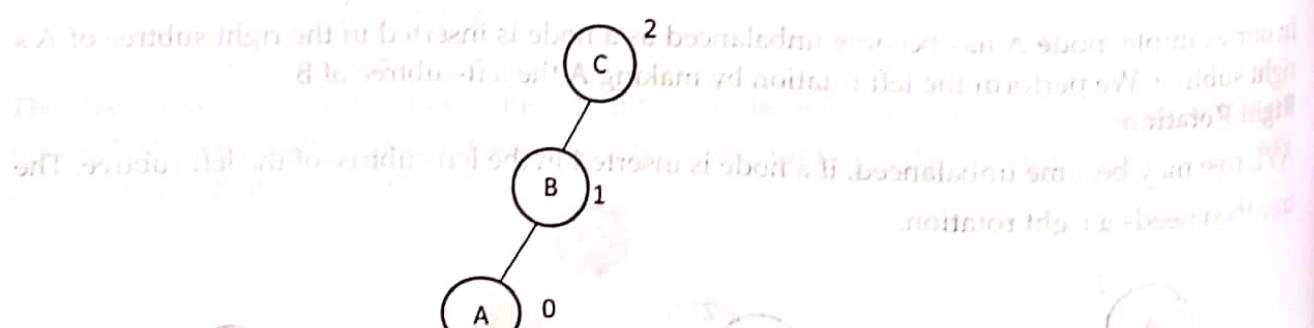
A node has been inserted into the right subtree of the left subtree. This makes **C** an unbalanced node. These scenarios cause AVL tree to perform left-right rotation.



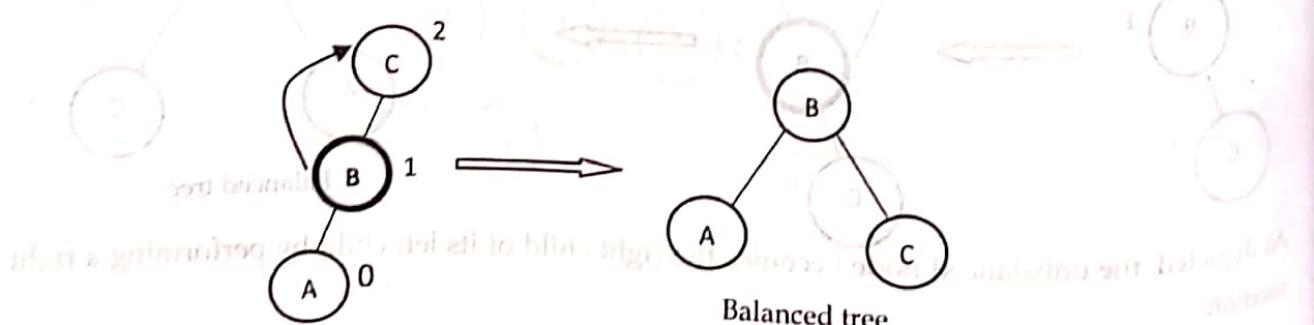
We first perform the left rotation on the left subtree of **C**. This makes **A**, the left subtree of **B**.



Node **C** is still unbalanced; however now, it is because of the left-subtree of the left-subtree.

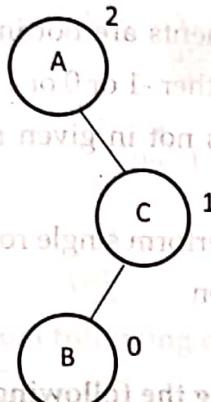


We shall now right-rotate the tree, making **B** the new root node of this subtree. **C** now becomes the right subtree of its own left subtree.

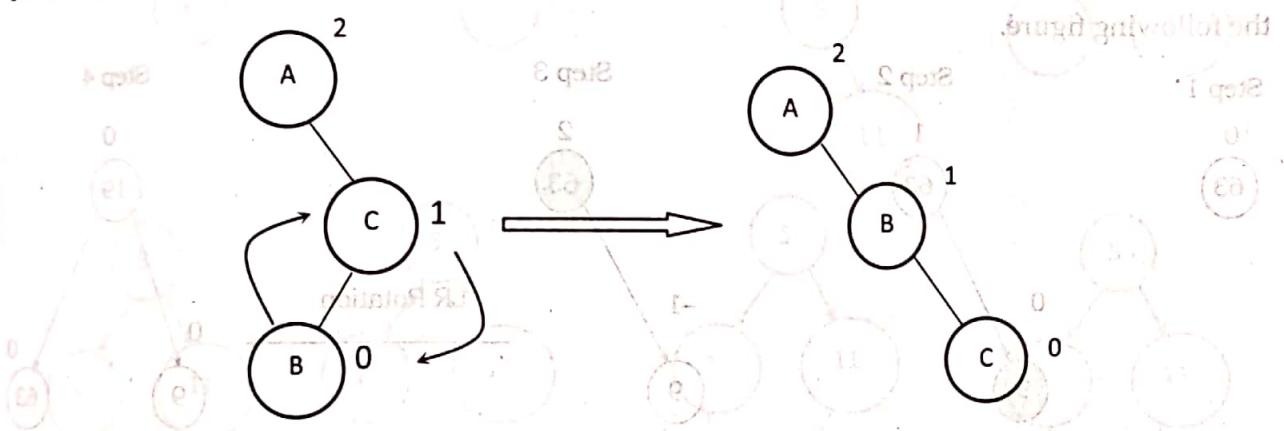


Right-Left Rotation

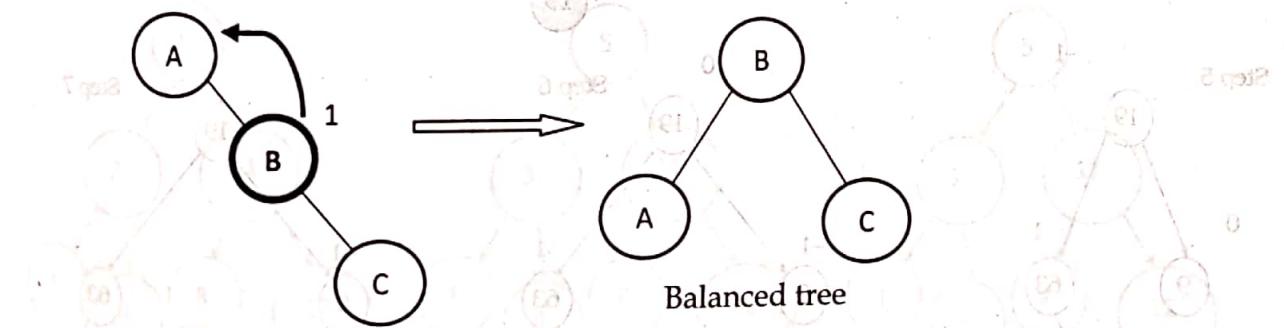
The second type of double rotation is Right-Left Rotation. It is a combination of right rotation followed by left rotation. A node has been inserted into the left subtree of the right subtree. This makes A, an unbalanced node with balance factor 2.



First, we perform the right rotation along C node, making C the right subtree of its own left subtree B. Now, B becomes the right subtree of A.

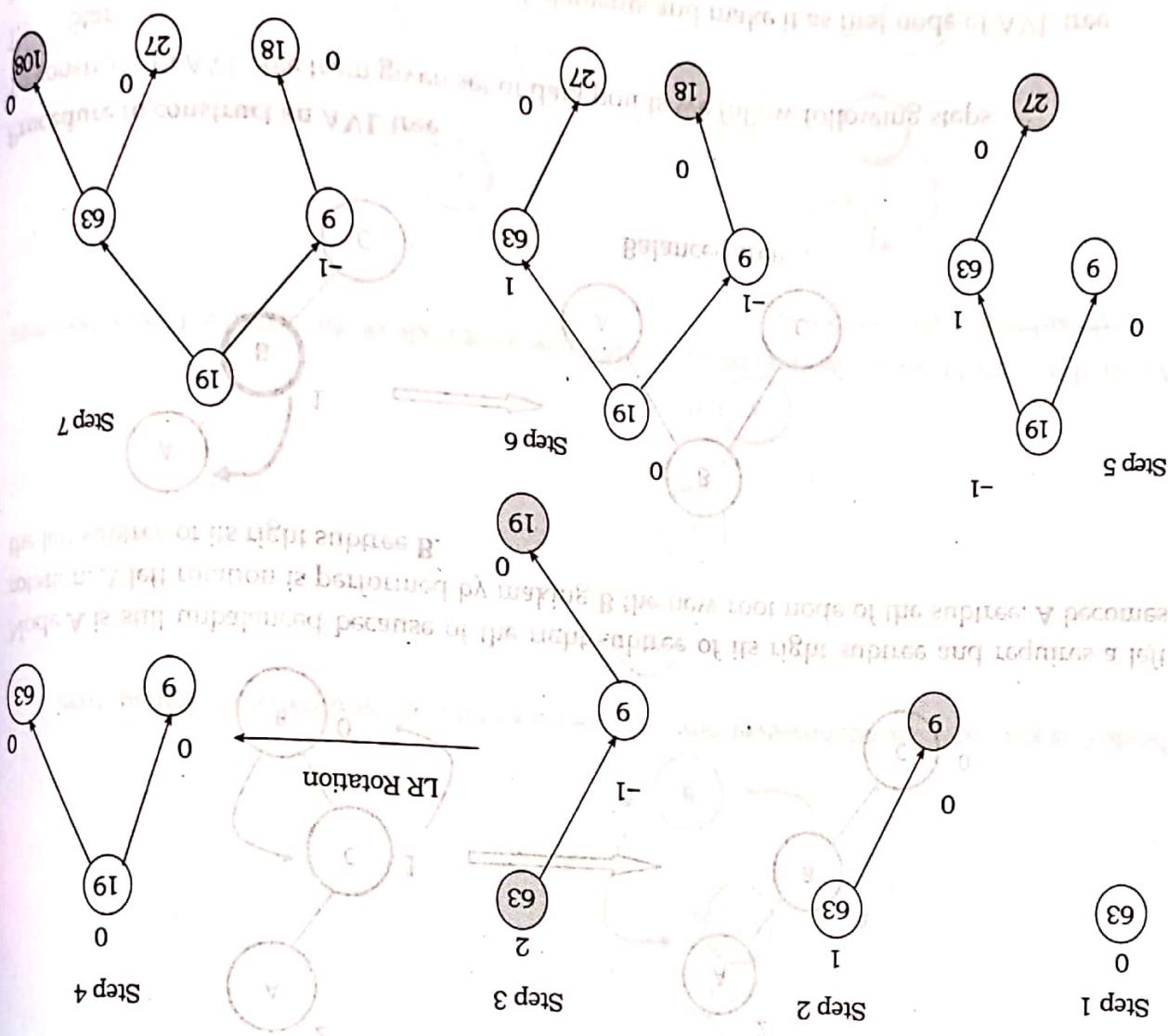


Node A is still unbalanced because of the right subtree of its right subtree and requires a left rotation. A left rotation is performed by making B the new root node of the subtree. A becomes the left subtree of its right subtree B.

**Procedure to construct an AVL tree**

To construct an AVL tree from given set of data points we follow following steps:

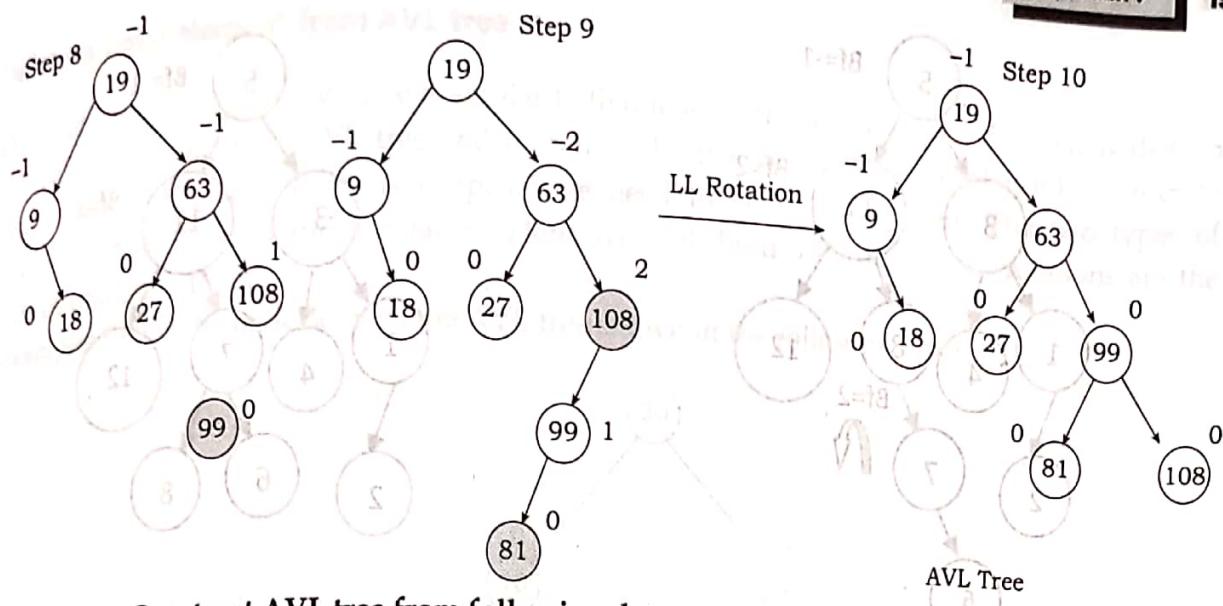
1. Start
2. Take a first element of given array of elements and make it as first node of AVL tree



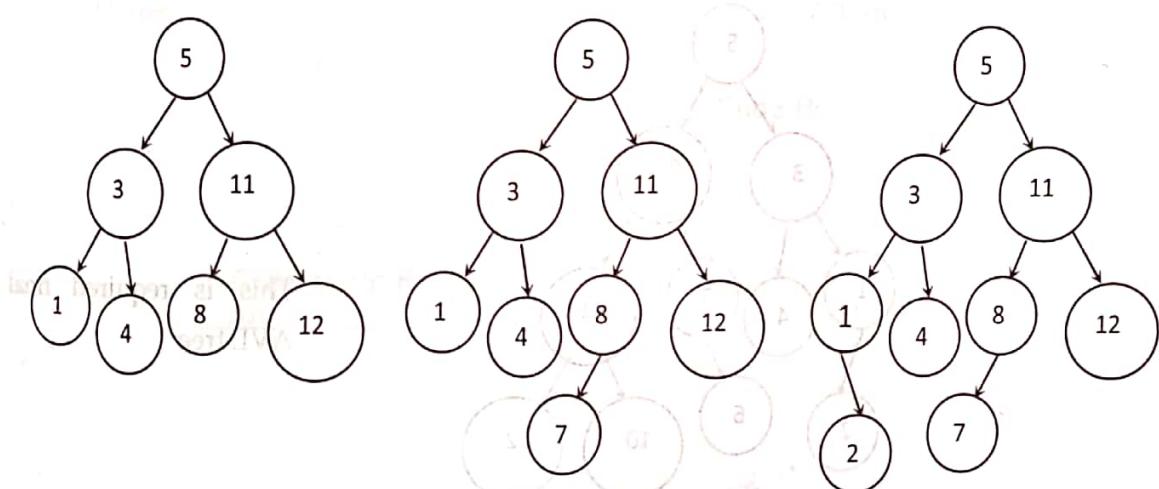
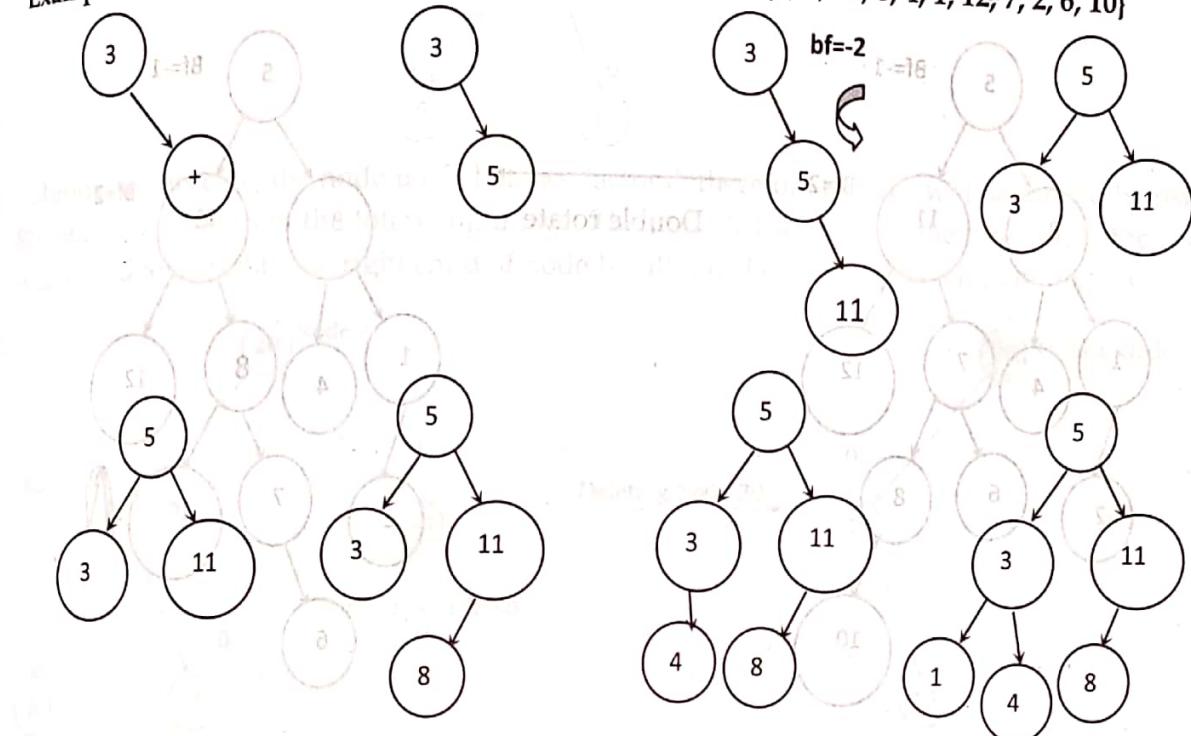
Solution: The process of constructing an AVL tree from the given set of elements is shown in the following figure.

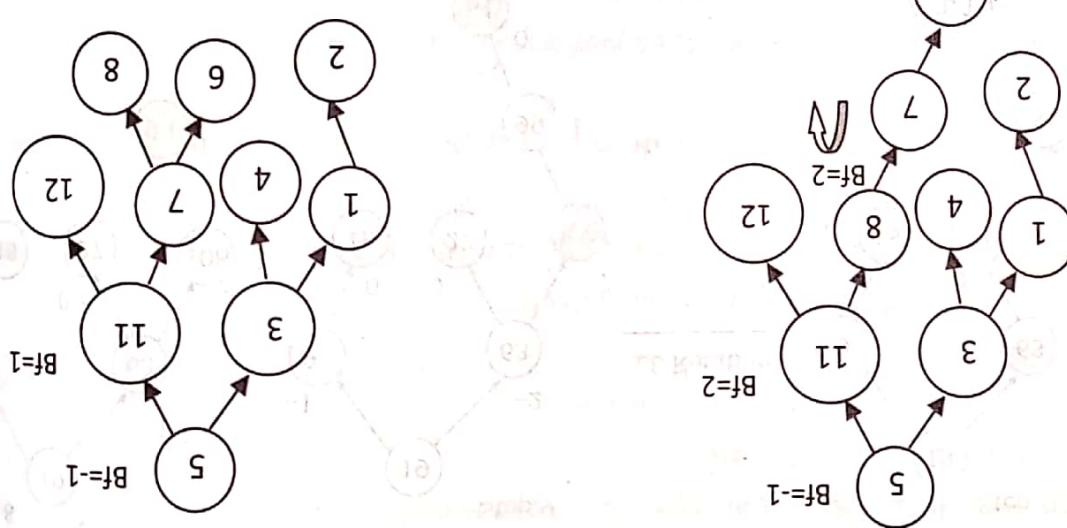
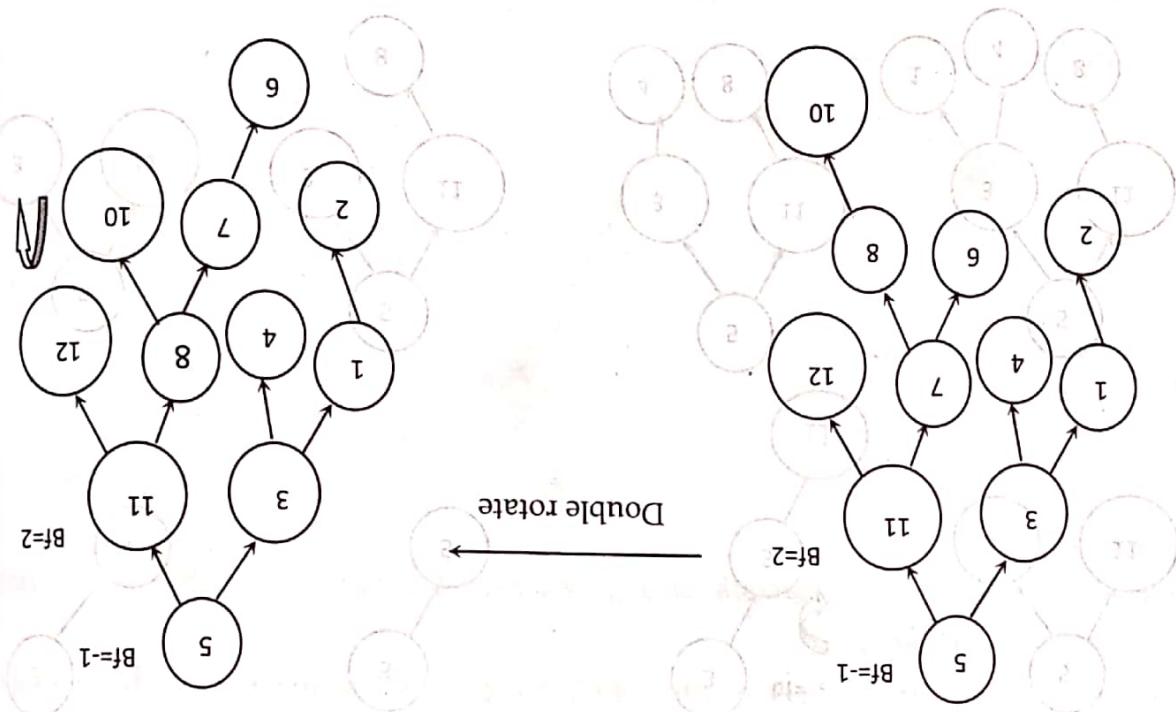
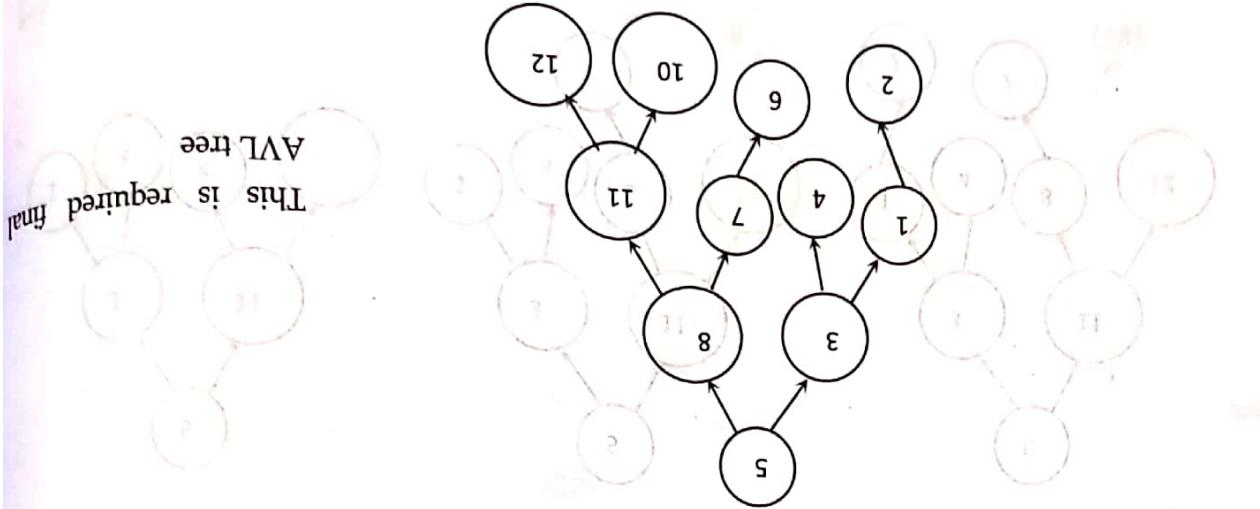
Example 1: Construct an AVL tree by inserting the following elements in the given order.

1. Set node(1) in right side of node(0)
2. Set next node either left side or right side of given AVL tree
3. a. If node(1) > node(2) then
 - i. Set node(2) in left side of node(1)
 - ii. Set node(2) in right side of node(1)
 - iii. Otherwise
 - a. If balance factor for each node either -1 or 1, then rotate either single or double
 - b. If straight path take place, then perform single rotation
 - c. Otherwise perform double rotation
4. Continue this process until all the elements are not included in resulting AVL tree and maintain balance factor for each node either -1 or 0 or 1.
5. If balance factor of a particular node is not in given range (-1 to +1) then rotate either single or double
6. Stop



Example 2: Construct AVL tree from following data sets: {3, 5, 11, 8, 4, 1, 12, 7, 2, 6, 10}

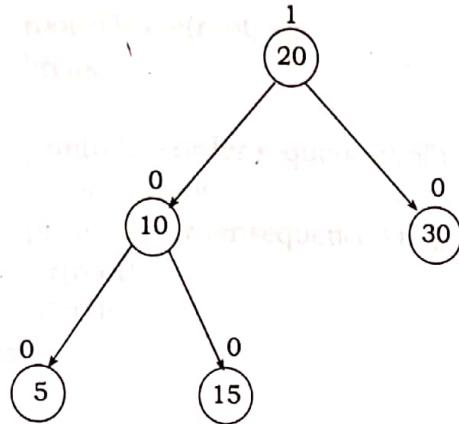




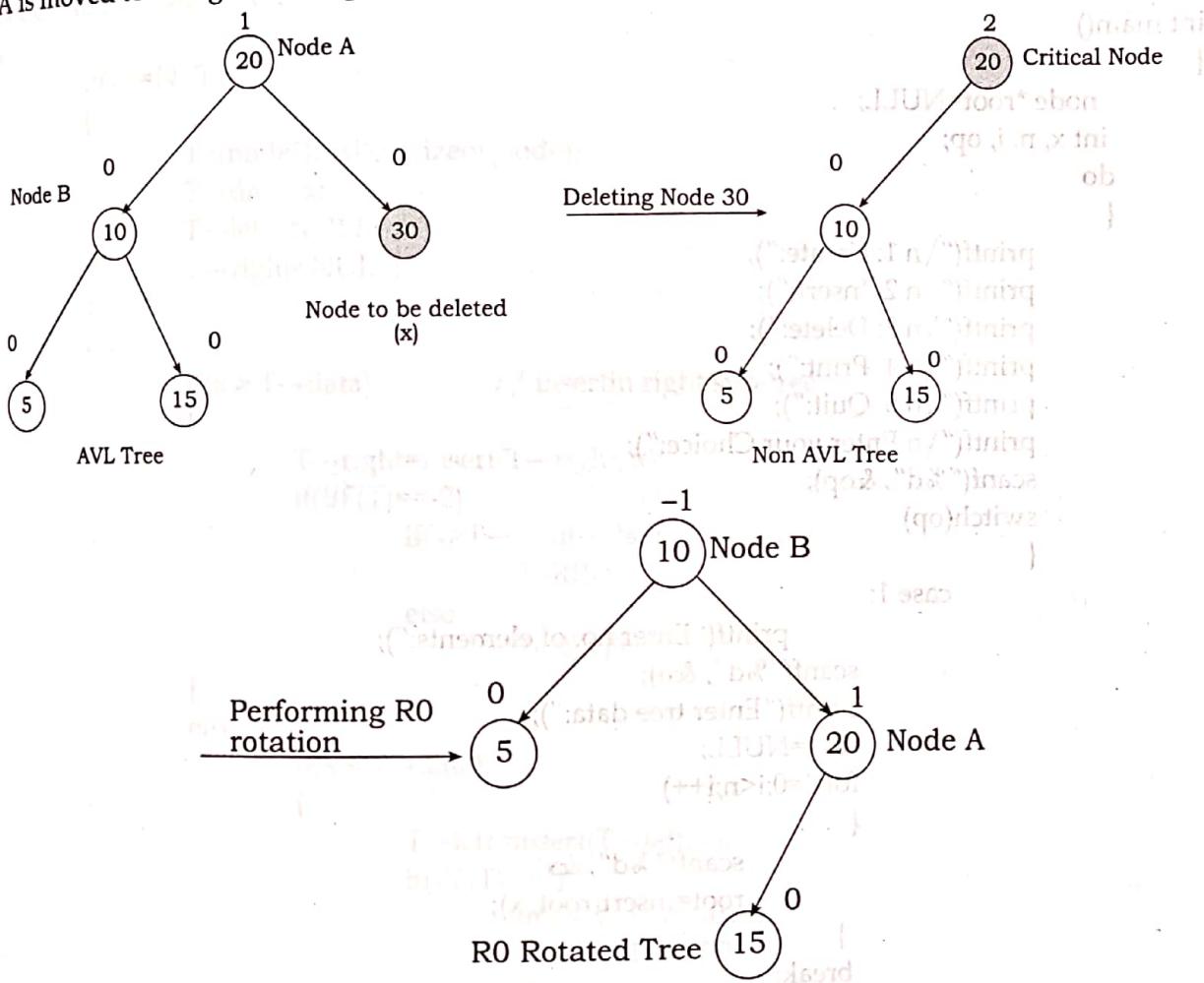
Deleting data element from AVL tree

Deleting a node from an AVL tree is similar to that in a binary search tree. Deletion may disturb the balance factor of an AVL tree and therefore the tree needs to be rebalanced in order to maintain the AVLness. For this purpose, we need to perform rotations. The two types of rotations are L rotation and R rotation. Here, we will discuss R rotations. L rotations are the mirror images of them.

Example: Delete the node 30 from the AVL tree shown in the following image.



Solution: In this case, the node B has balance factor 0, therefore the tree will be rotated by using R0 rotation as shown in the following image. The node B(10) becomes the root, while the node A is moved to its right. The right child of node B will now become the left child of node A.



Program in C for inserting an element in AVL Tree

```
#include<conio.h>
#include<stdio.h>
#include<stdlib.h>
typedef struct node
{
    int data;
    struct node *left,*right;
    int ht;
}node;
node *insert(node *,int);
node *Delete(node *,int);
void preorder(node *);
void inorder(node *);
int height( node *);
node *rotateright(node *);
node *rotateleft(node *);
node *RR(node *);
node *LL(node *);
node *LR(node *);
node *RL(node *);
int BF(node *);
int main()
{
    node *root=NULL;
    int x, n, i, op;
    do
    {
        printf("\n 1: Create:");
        printf("\n 2: Insert:");
        printf("\n 3: Delete:");
        printf("\n 4: Print:");
        printf("\n 5: Quit:");
        printf("\n Enter your Choice:");
        scanf("%d", &op);
        switch(op)
        {
            case 1:
                printf("Enter no. of elements:");
                scanf("%d", &n);
                printf("Enter tree data:");
                root=NULL;
                for(i=0;i<n;i++)
                {
                    scanf("%d", &x);
                    root=insert(root, x);
                }
                break;
        }
    }
}
```

case 2:

```
printf("Enter a data:");
scanf("%d",&x);
root=insert(root,x);
break;
```

case 3:

```
printf("Enter a data:");
scanf("%d",&x);
root=Delete(root,x);
break;
```

case 4:

```
printf("Preorder sequence:\n");
preorder(root);
printf("In-order sequence:\n");
inorder(root);
printf("\n");
break;
```

} while(op!=5);

return 0;

node * insert(node *T, int x)

{ if(T==NULL)

```
T=(node*)malloc(sizeof(node));
T->data=x;
T->left=NULL;
T->right=NULL;
```

}

else

if(x > T->data) // insert in right sub-tree

{

T->right=insert(T->right, x);

if(BF(T)==-2)

if(x > T->right->data)

T=RR(T);

else T=RL(T);

else q=T->right->right;

}

else

if(x < T->data)

{ T->left=insert(T->left, x);

if(BF(T)==2)

if(x < T->left->data)

T=LL(T);

else

```

    T=LR(T);           // left child is taller than right child

    }                   // if (T->right!=NULL) then
    T->ht=height(T); // calculate height of tree
    return(T);
}

node * Delete(node *T, int x)
{
    node *p;
    if(T==NULL)
    {
        return NULL;
    }
    else
    {
        if(x > T->data) // insert in right sub-tree
        {
            T->right=Delete(T->right, x);
            if(BF(T)==2)
                if(BF(T->left)>=0)
                    T=LL(T);
                else
                    T=LR(T);
        }
        else
        {
            if(x<T->data)
            {
                T->left=Delete(T->left, x);
                if(BF(T)==-2) //Rebalance during windup
                    if(BF(T->right)<=0)
                        T=RR(T);
                    else
                        T=RL(T);
            }
            else
            {
                //data to be deleted is found
                if(T->right!=NULL)
                {
                    p=T->right;
                    while(p->left!=NULL)
                        p=p->left;
                    T->data=p->data;
                    T->right=Delete(T->right, p->data);
                    if(BF(T)==2)//Rebalance during windup
                        if(BF(T->left)>=0)
                            T=LL(T);
                        else
                            T=LR(T);
                }
            }
        }
    }
}

```

```

    else
        return(T->left);
    }
    T->ht=height(T);
    return(T);
}

int height(node *T)
{
    int lh, rh;
    if(T==NULL)
        return(0);
    if(T->left==NULL)
        lh=0;
    else
        lh=1+T->left->ht;
    if(T->right==NULL)
        rh=0;
    else
        rh=1+T->right->ht;
    if(lh>rh)
        return(lh);
    return(rh);
}

node * rotateright(node *x)
{
    node *y;
    y=x->left;
    x->left=y->right;
    y->right=x;
    x->ht=height(x);
    y->ht=height(y);
    return(y);
}

node * rotateleft(node *x)
{
    node *y;
    y=x->right;
    x->right=y->left;
    y->left=x;
    x->ht=height(x);
    y->ht=height(y);
    return(y);
}

node * RR(node *T)
{
    T=rotateleft(T);
    return(T);
}

```

```

}

node * LL(node *T)
{
    T=rotateright(T);
    return(T);
}

node * LR(node *T)
{
    T->left=rotateleft(T->left);
    T=rotateright(T);
    return(T);
}

node * RL(node *T)
{
    T->right=rotateright(T->right);
    T=rotateleft(T);
    return(T);
}

int BF(node *T)
{
    int lh, rh;
    if(T==NULL)
        return(0);
    if(T->left==NULL)
        lh=0;
    else
        lh=1+T->left->ht;
    if(T->right==NULL)
        rh=0;
    else
        rh=1+T->right->ht;
    return(lh-rh);
}

void preorder(node *T)
{
    if(T!=NULL)
    {
        printf("%d(Bf=%d)", T->data, BF(T));
        preorder(T->left);
        preorder(T->right);
    }
}

void inorder(node *T)
{
    if(T!=NULL)
    {
        inorder(T->left);
        printf("%d(Bf=%d)", T->data, BF(T));
        inorder(T->right);
    }
}

```

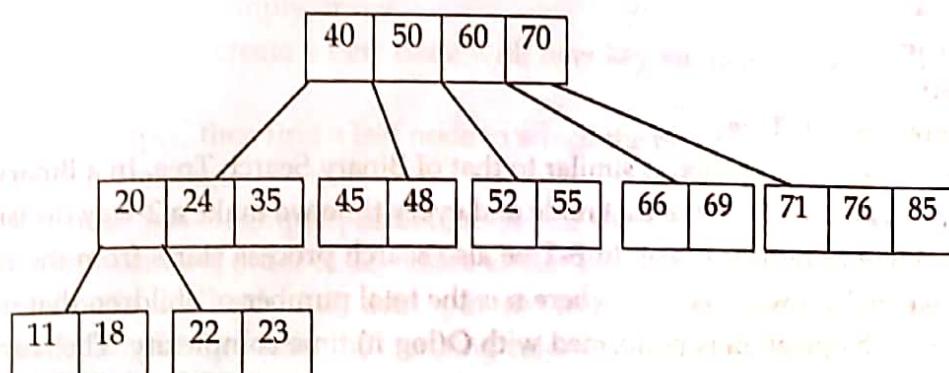
Time Complexity

The rotation operations (left and right rotate) take constant time as only few pointers are being changed there. Updating the height and getting the balance factor also take constant time. So the time complexity of AVL insert remains same as BST insert which is $O(h)$ where h is height of the tree. Since AVL tree is balanced, the height is $O(\log n)$. So time complexity of AVL insert is $O(\log n)$.

Multi-way tree

A multi-way tree is a tree that can have more than two children. A **multi-way tree of order m** or an **m-way tree** is one in which a tree can have m children. As with the other trees that have been studied, the nodes in an m -way tree will be made up of key fields, in this case $m-1$ key fields and pointers to children. If $k \leq m$ is the number of children, then the node contains exactly $k - 1$ keys, which partition all the keys into k subsets consisting of all the keys less than the first key in the node, all the keys between a pair of keys in the node, and all keys greater than the largest key in the node.

Example: multi way tree of order 5



To make the processing of m -way trees easier some type of order will be imposed on the keys within each node, resulting in a **multi way search tree of order m** (or an **m -way search tree**). By definition an m -way search tree is a m -way tree in which:

- Each node has m children and $m-1$ key fields
- The keys in each node are in ascending order.
- The keys in the first i children are smaller than the i^{th} key
- The keys in the last $m-i$ children are larger than the i^{th} key

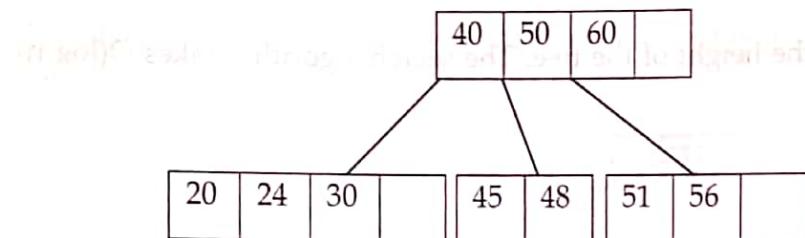
B Tree

B Tree is a specialized m -way tree that can be widely used for disk access. A B-Tree of order m can have at most $m-1$ keys and m children. One of the main reasons of using B tree is its capability to store large number of keys in a single node and large key values by keeping the height of the tree relatively small.

A B-tree of order m contains all the properties of an M way tree. In addition, it contains the following properties.

1. Every node in a B-Tree contains at most m children.
2. Every node in a B-Tree except the root node and the leaf node contain at least $m/2$ children.

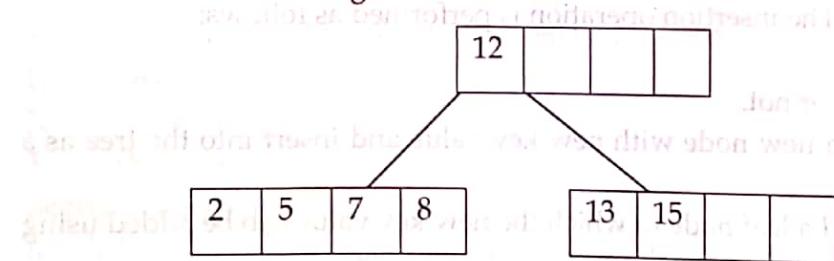
Inserting the number 30 results in:



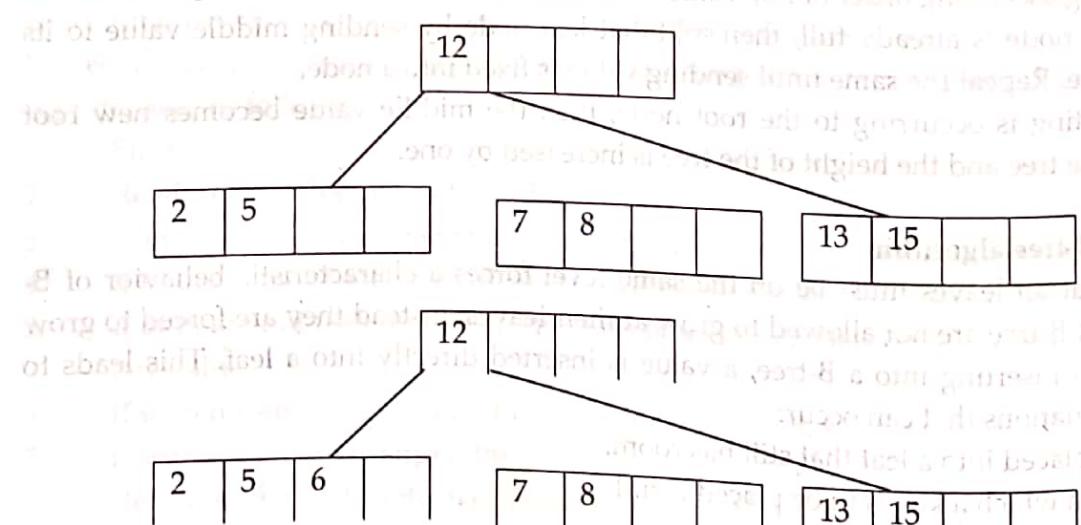
Case 2: The leaf in which a key is to be placed is full

- In this case, the leaf node where the value should be inserted is split in two, resulting in a new leaf node. Half of the keys will be moved from the full leaf to the new leaf. The new leaf is then incorporated into the B-tree. The new leaf is incorporated by moving the middle value to the parent and a pointer to the new leaf is also added to the parent. This process continues up the tree until all of the values have "found" a location.

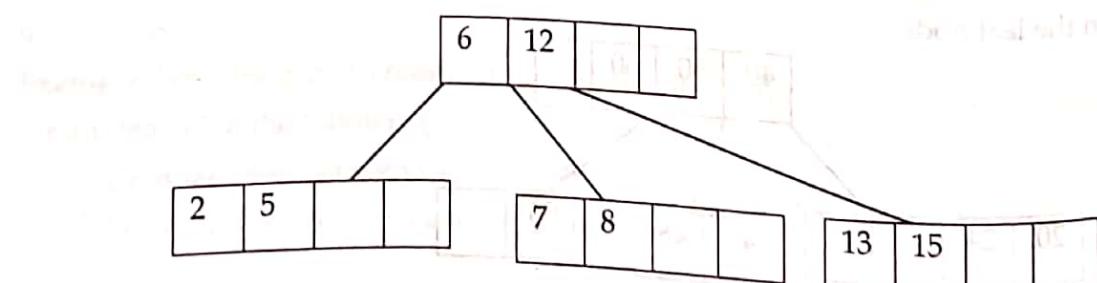
Insert 6 into the following B-tree:



Results in a split of the first leaf node:



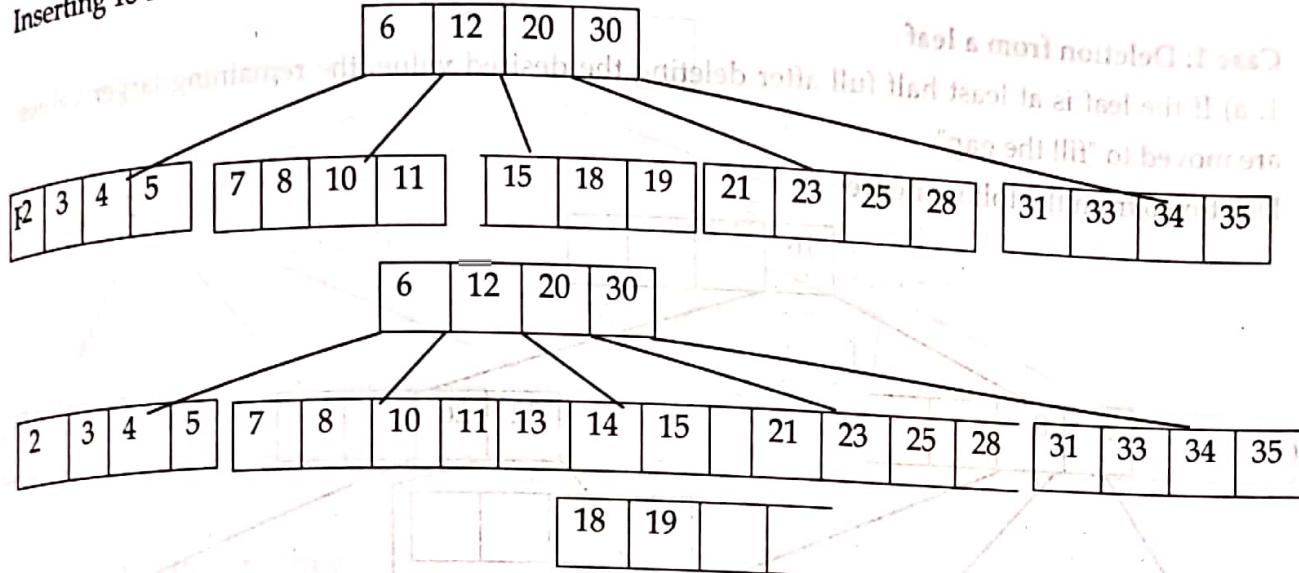
The new node needs to be incorporated into the tree - this is accomplished by taking the middle value and inserting it in the parent:



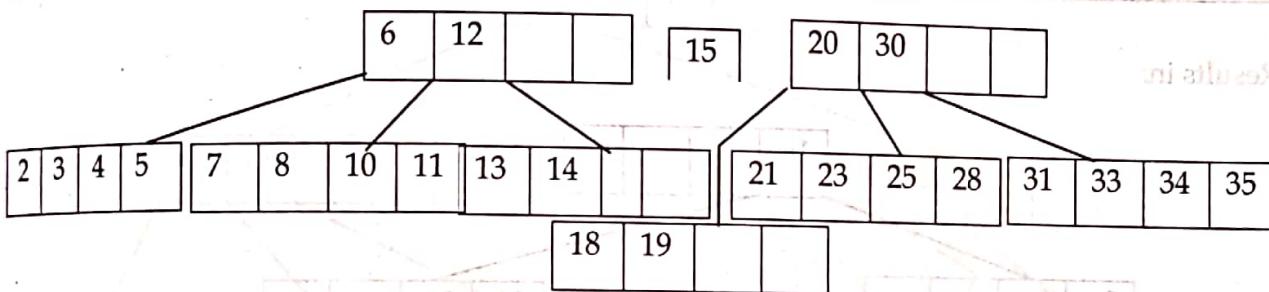
Case 3: The root of the B-tree is full

The upward movement of values from case 2 means that it's possible that a value could move up to the root of the B-tree. If the root is full, the same basic process from case 2 will be applied and a new root will be created. This type of split results in 2 new nodes being added to the B-tree.

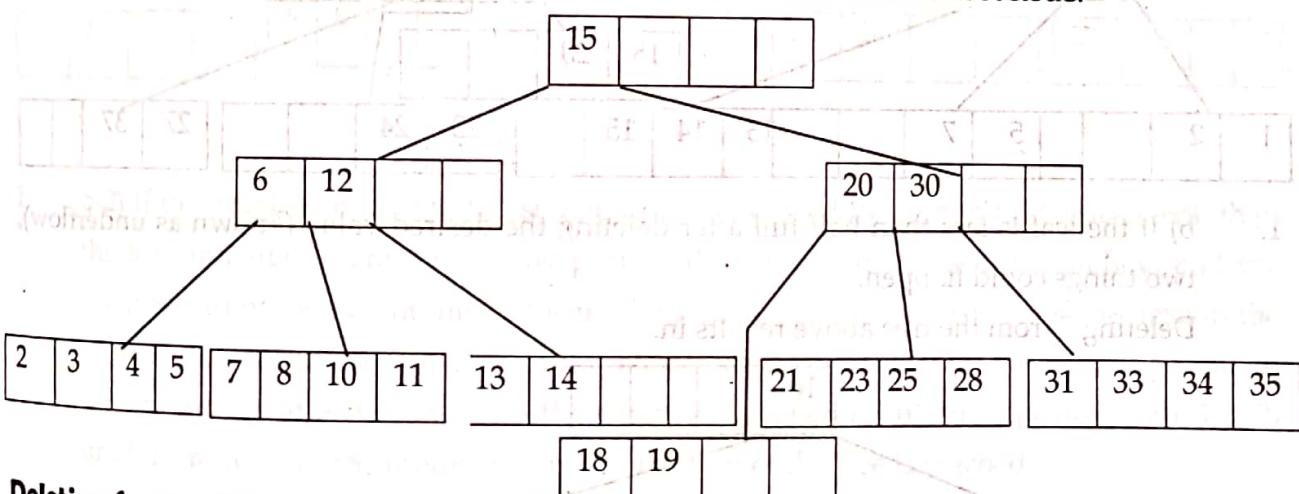
Inserting 13 into the following tree:



The 15 needs to be moved to the root node but it is full. This means that the root needs to be divided:



The 15 is inserted into the parent, which means that it becomes the new root node:

**Deleting from a B-tree**

As usual, this is the hardest of the processes to apply. The deletion process will basically be a reversal of the insertion process - rather than splitting nodes, it's possible that nodes will be merged so that B-tree properties, namely the requirement that a node must be at least half full, can be maintained.

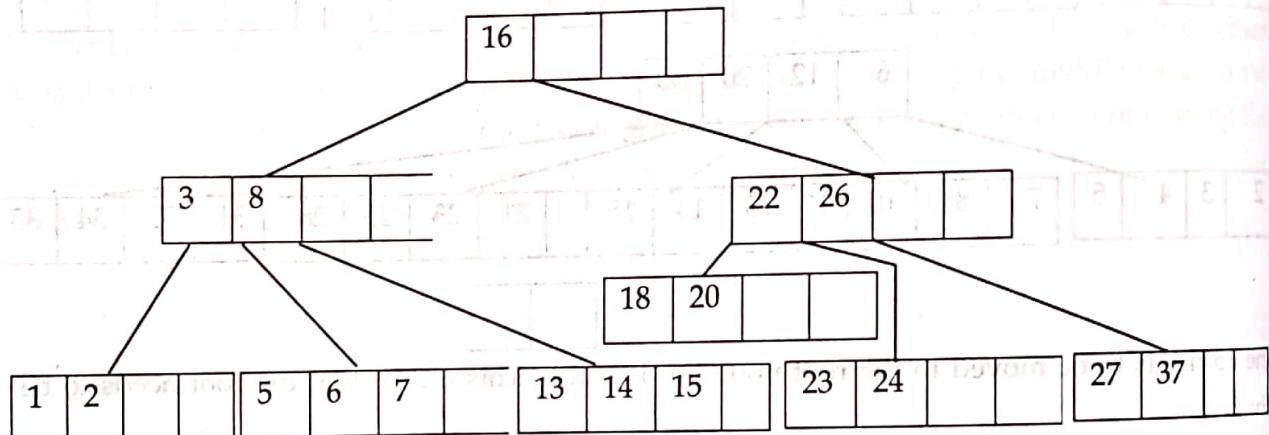
There are two main cases to be considered:

1. Deletion from a leaf
2. Deletion from a non-leaf

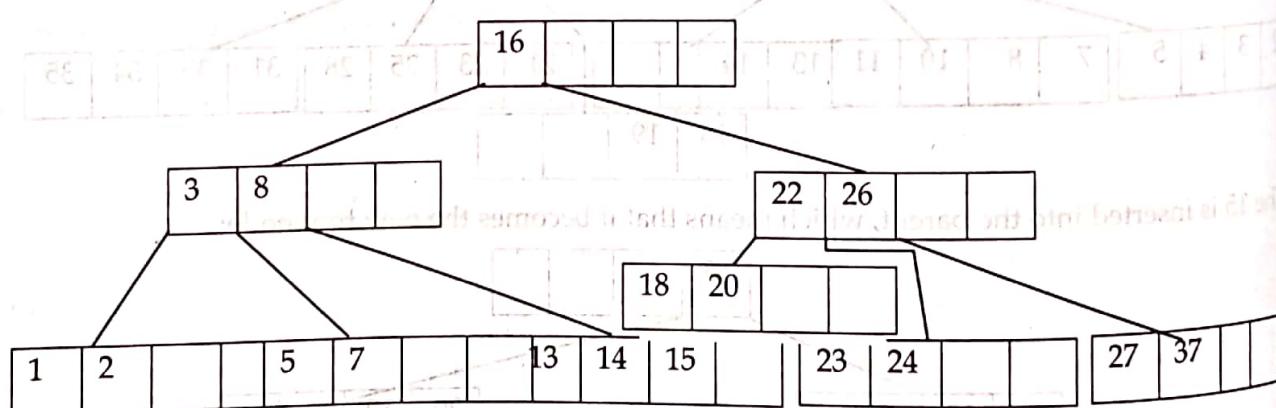
Case 1: Deletion from a leaf

1. a) If the leaf is at least half full after deleting the desired value, the remaining larger values are moved to "fill the gap".

Deleting 6 from the following tree:

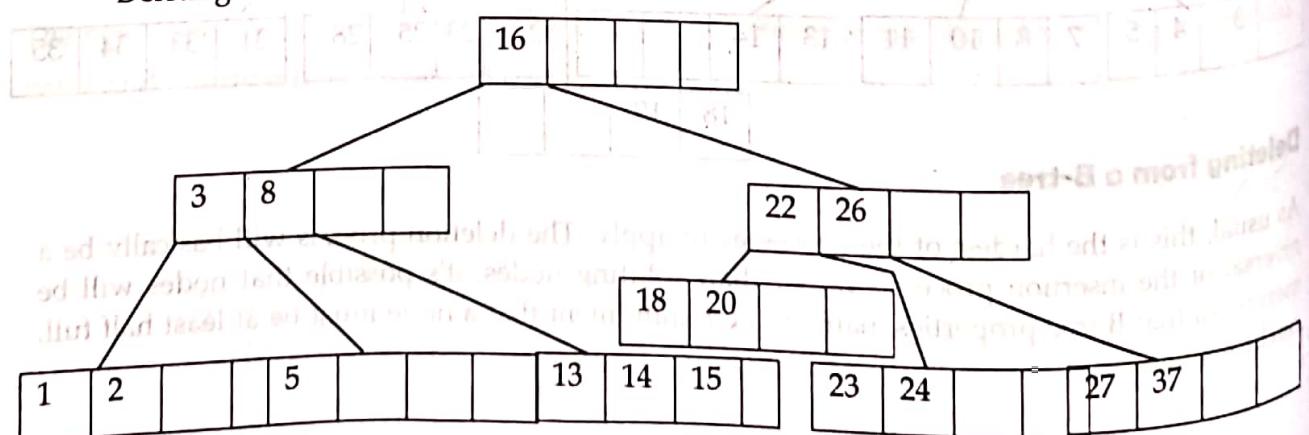


Results in:

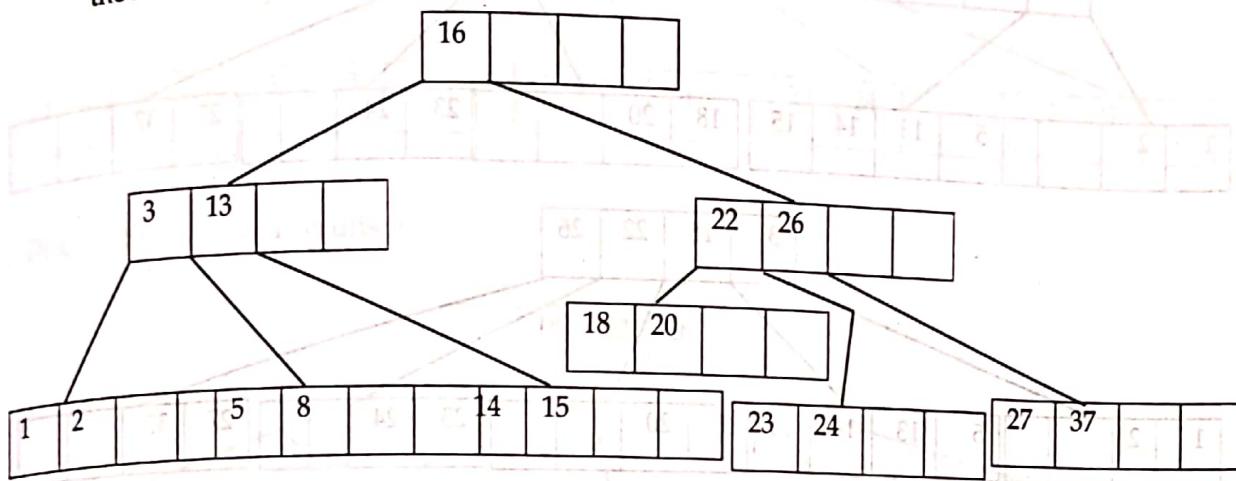


1. b) If the leaf is less than half full after deleting the desired value (known as underflow), two things could happen:

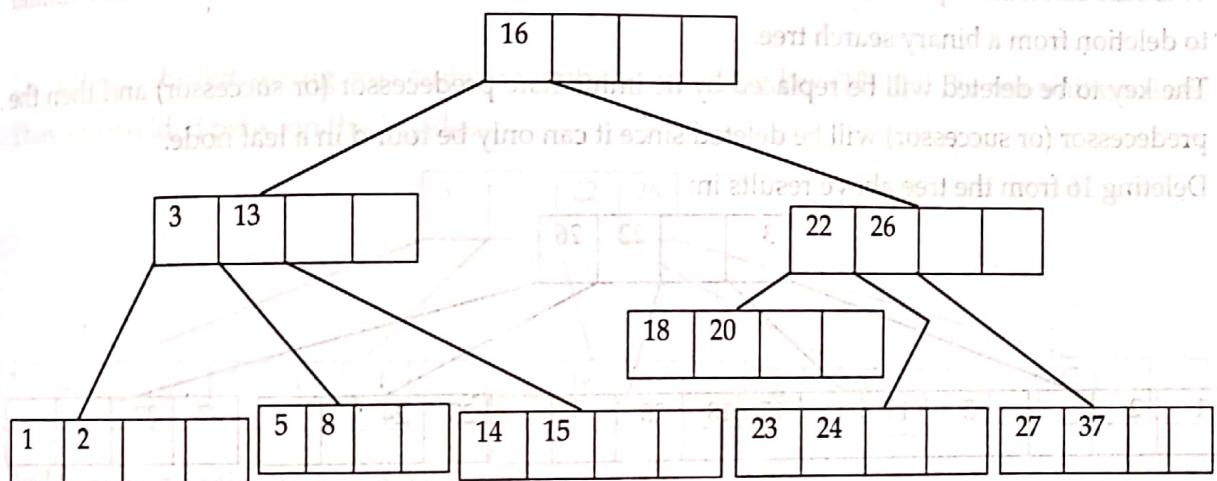
Deleting 7 from the tree above results in:



1.b-1) If there is a left or right sibling with the number of keys exceeding the minimum requirement, all of the keys from the leaf and sibling will be redistributed between them by moving the separator key from the parent to the leaf and moving the middle key from the node and the sibling combined to the parent.

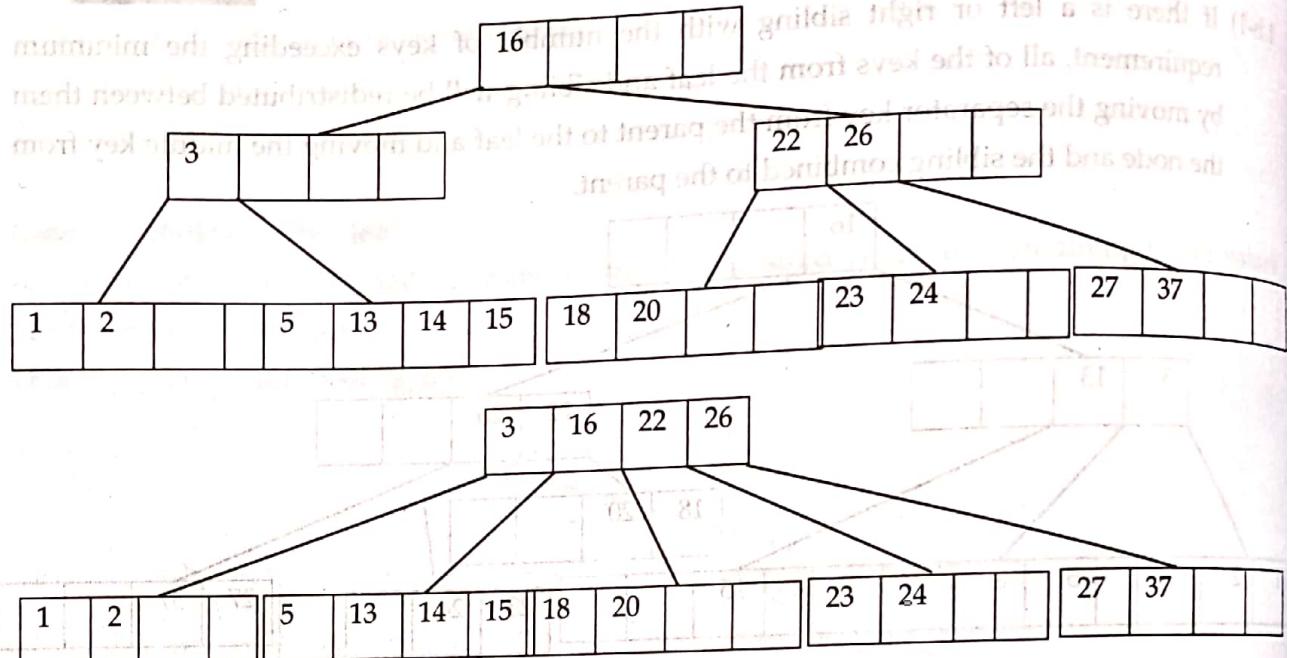


Now delete 8 from the tree:



1. b-2) If the number of keys in the sibling does not exceed the minimum requirement, then the leaf and sibling are merged by putting the keys from the leaf, the sibling, and the separator from the parent into the leaf. The sibling node is discarded and the keys in the parent are moved to "fill the gap". It's possible that this will cause the parent to underflow. If that is the case, treat the parent as a leaf and continue repeating step (1. b-2) until the minimum requirement is met or the root of the tree is reached.

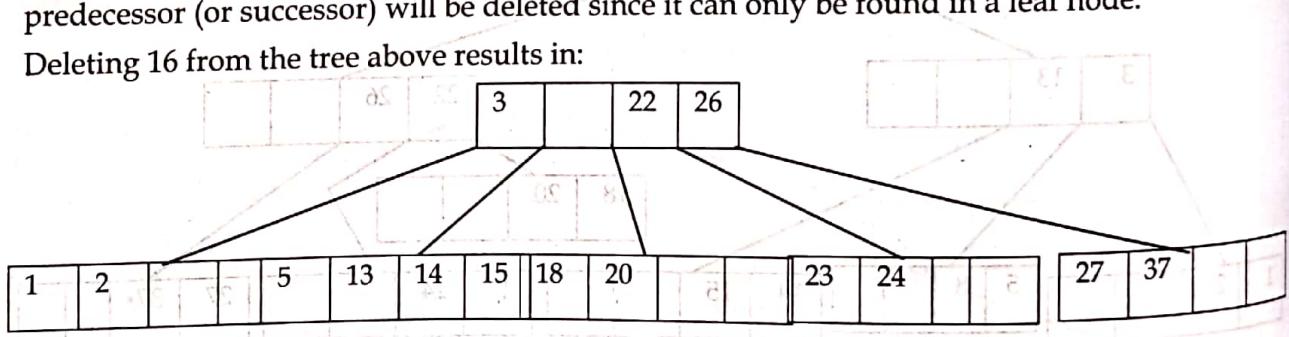
Special Case for 1.b-2: When merging nodes, if the parent is the root with only one key, the keys from the node, the sibling, and the only key of the root are placed into a node and this will become the new root for the B-tree. Both the sibling and the old root will be discarded.

**Case 2: Deletion from a non-leaf**

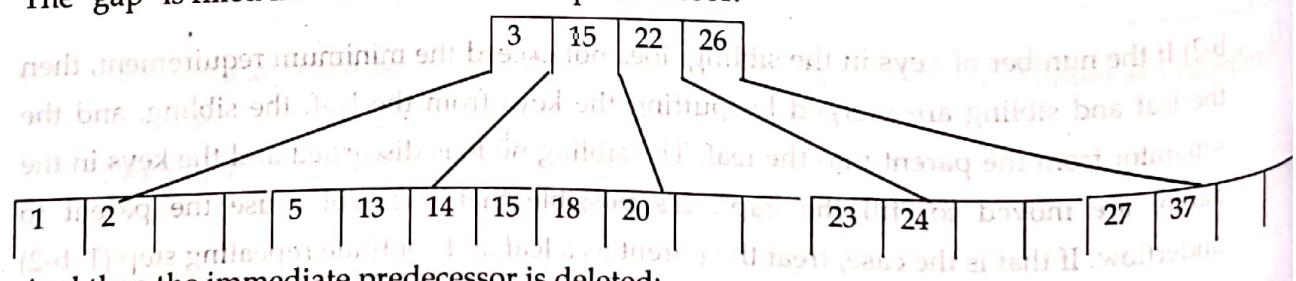
This case can lead to problems with tree reorganization but it will be solved in a manner similar to deletion from a binary search tree.

The key to be deleted will be replaced by its immediate predecessor (or successor) and then the predecessor (or successor) will be deleted since it can only be found in a leaf node.

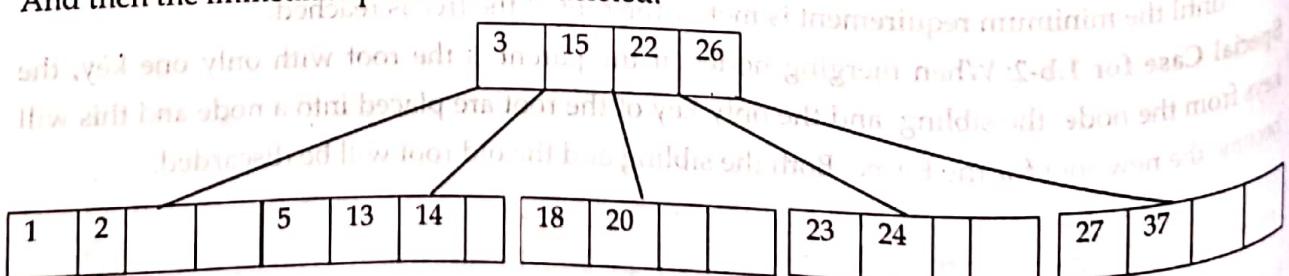
Deleting 16 from the tree above results in:



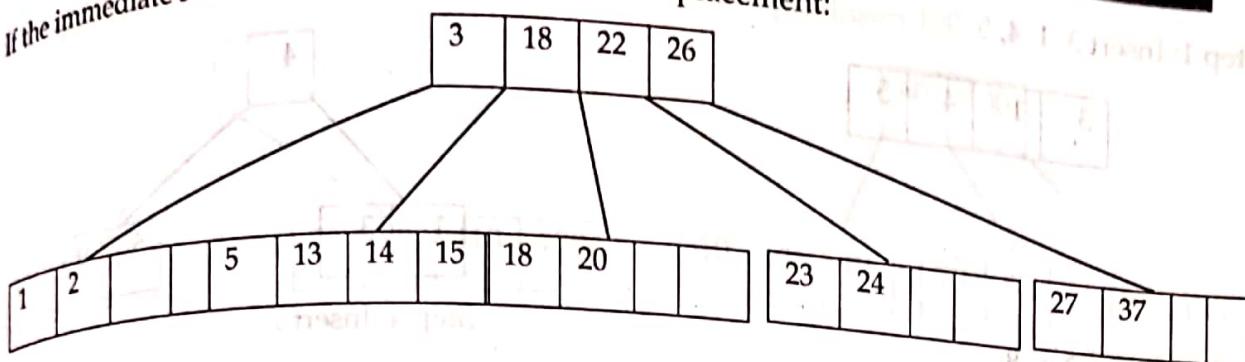
The "gap" is filled in with the immediate predecessor:



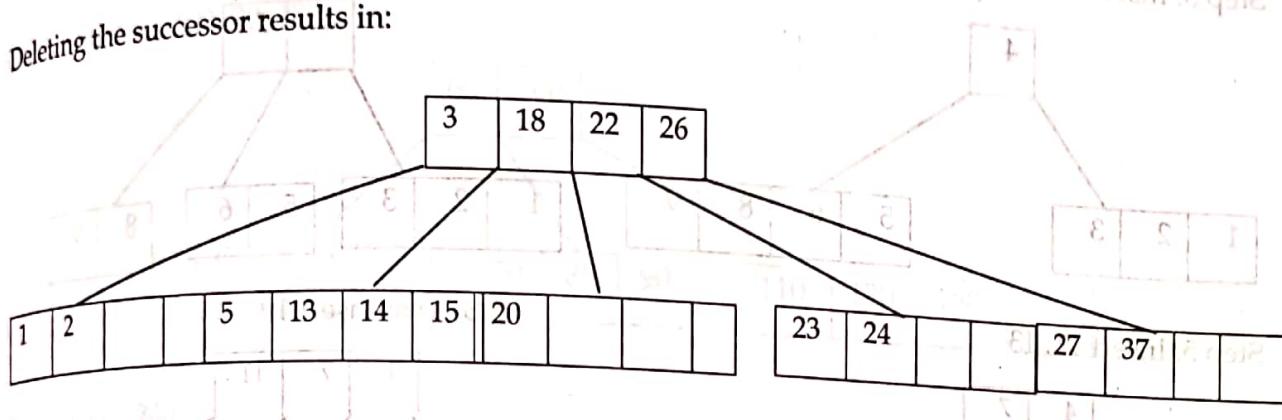
And then the immediate predecessor is deleted:



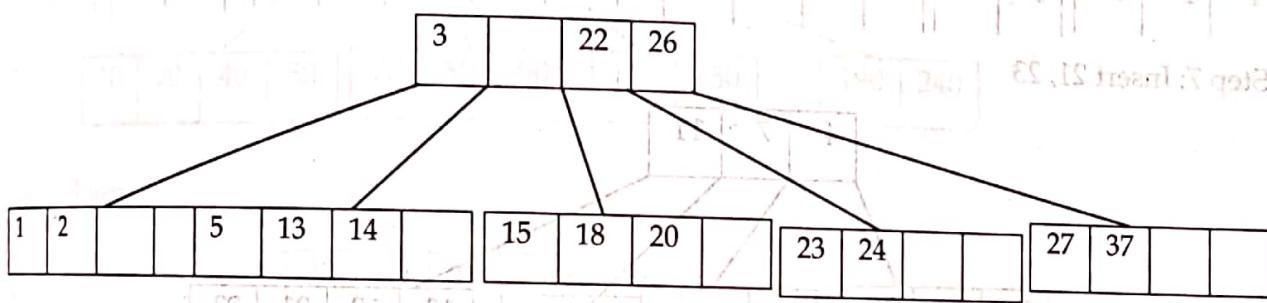
If the immediate successor had been chosen as the replacement:



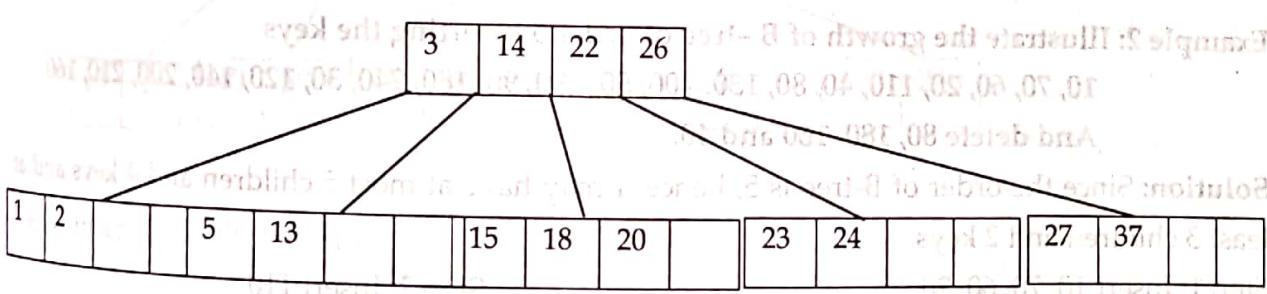
Deleting the successor results in:



The values in the left sibling are combined with the separator key (18) and the remaining values. They are divided between the 2 nodes:



And then the middle value is moved to the parent:

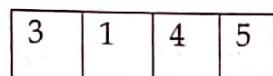


Example 1: Illustrate the growth of B -tree of order 5 inserting the keys

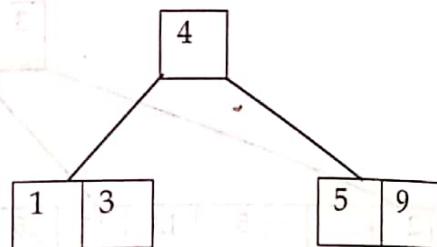
3, 1, 4, 5, 9, 2, 6, 8, 7, 11, 13, 19, 2, 23

Solution: Since the order of B-tree is 5, hence it may have at most 5 children and 4 keys and at least 3 children and 2 keys.

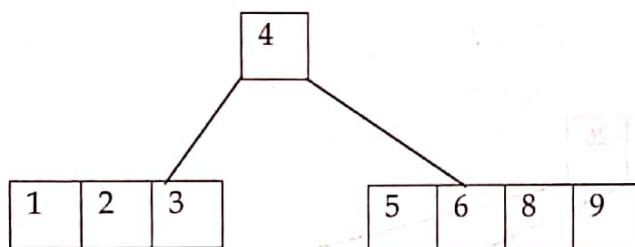
Step 1: Insert 3, 1, 4, 5



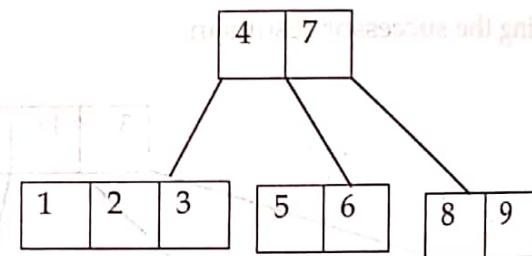
Step 2: Insert 9



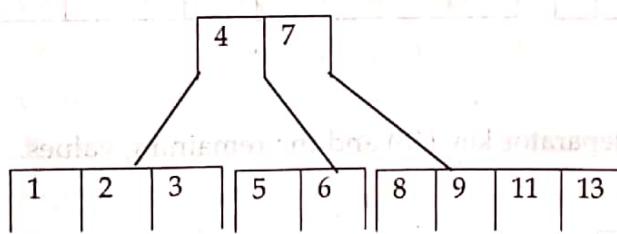
Step 3: Insert 2, 6, 8



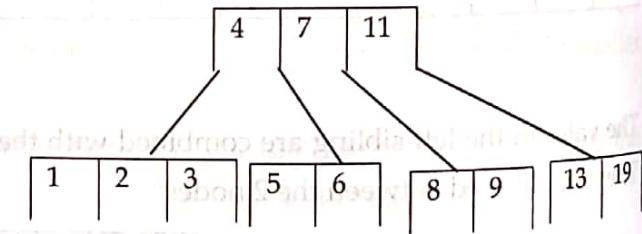
Step 4: Insert 7



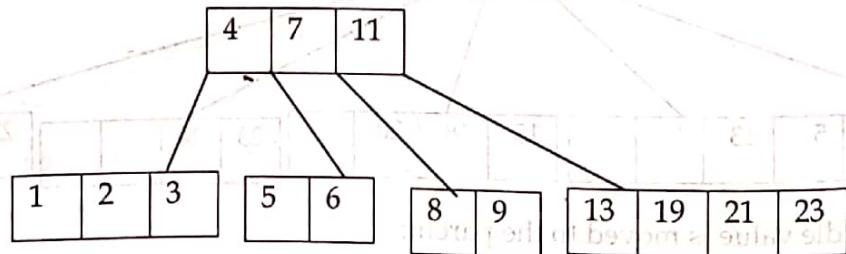
Step 5: Insert 11, 13



Step 6: Insert 19



Step 7: Insert 21, 23



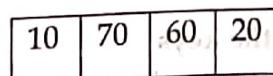
Example 2: Illustrate the growth of B-tree of order 5 inserting the keys

10, 70, 60, 20, 110, 40, 80, 130, 100, 50, 190, 90, 180, 240, 30, 120, 140, 200, 210, 160

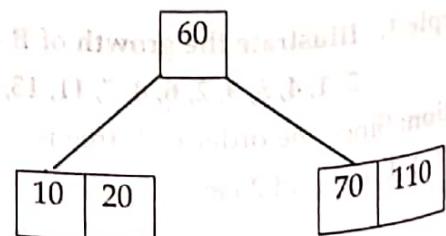
And delete 80, 180, 160 and 40.

Solution: Since the order of B-tree is 5, hence it may have at most 5 children and 4 keys and at least 3 children and 2 keys.

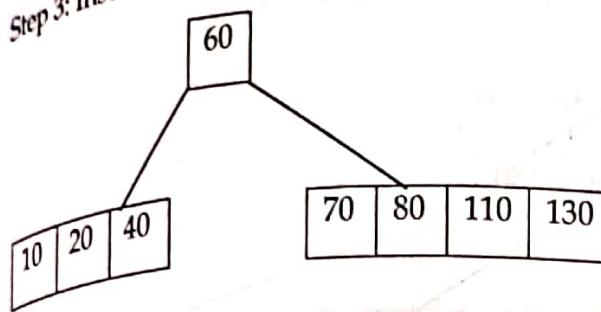
Step 1: Insert 10, 70, 60, 20



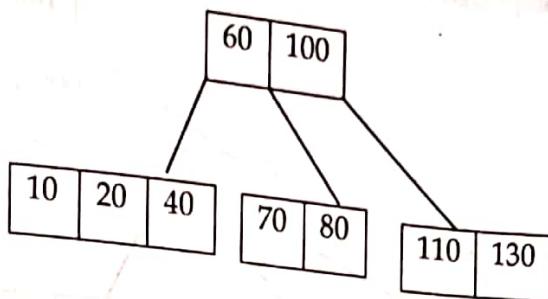
Step 2: Insert 110



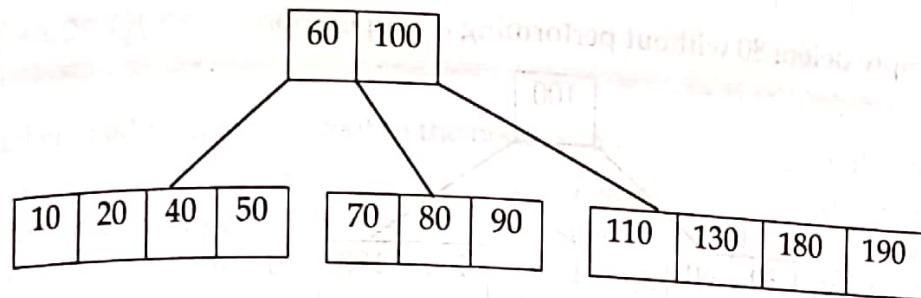
Step 3: Insert 40, 80, 130



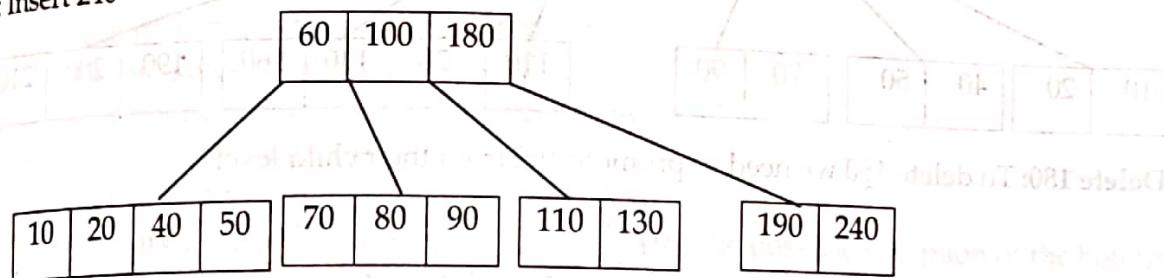
Step 4: Insert 100



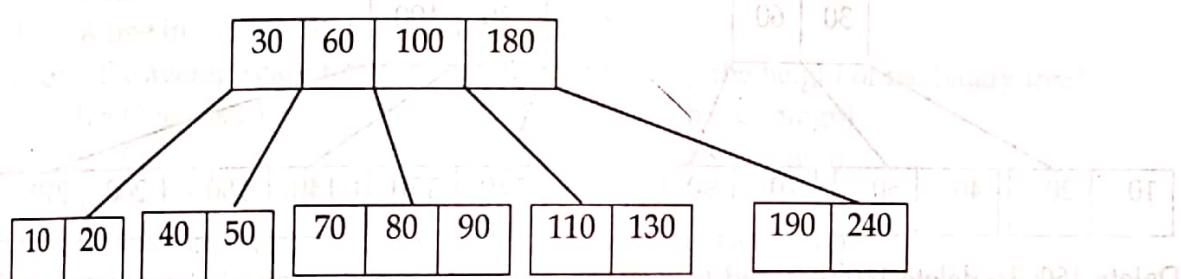
Step 5: Insert 50, 190, 90, 180



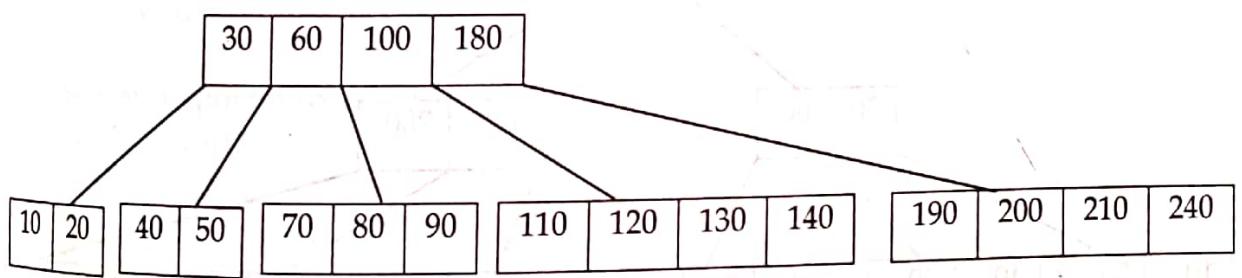
Step 6: Insert 240



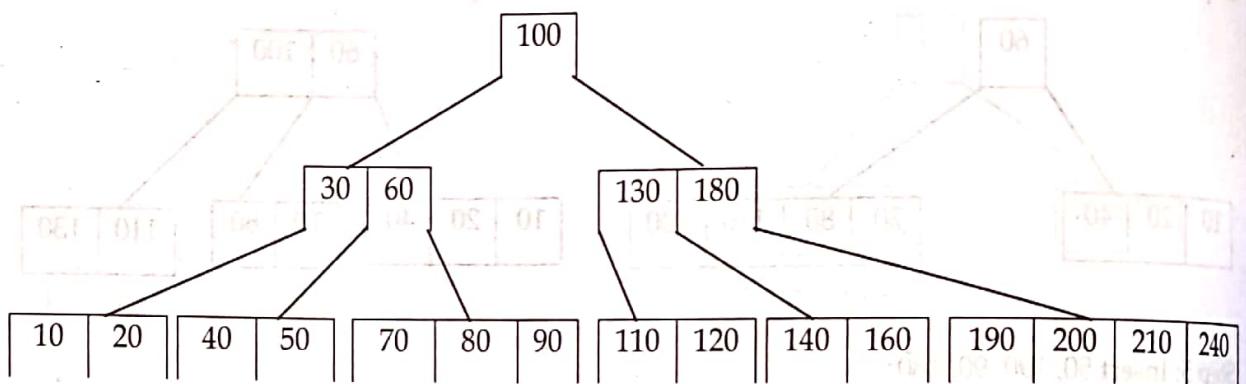
Step 7: Insert 30



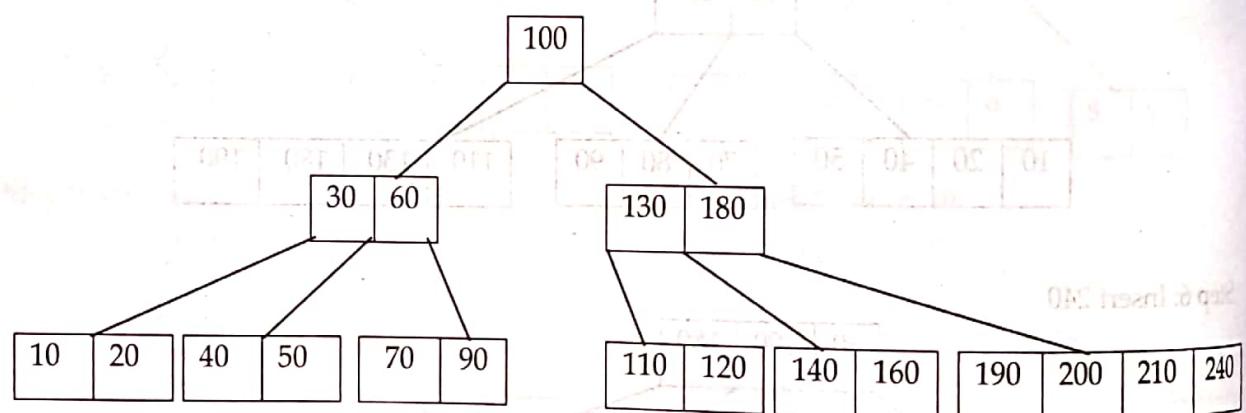
Step 8: Insert 120, 140, 200, 210



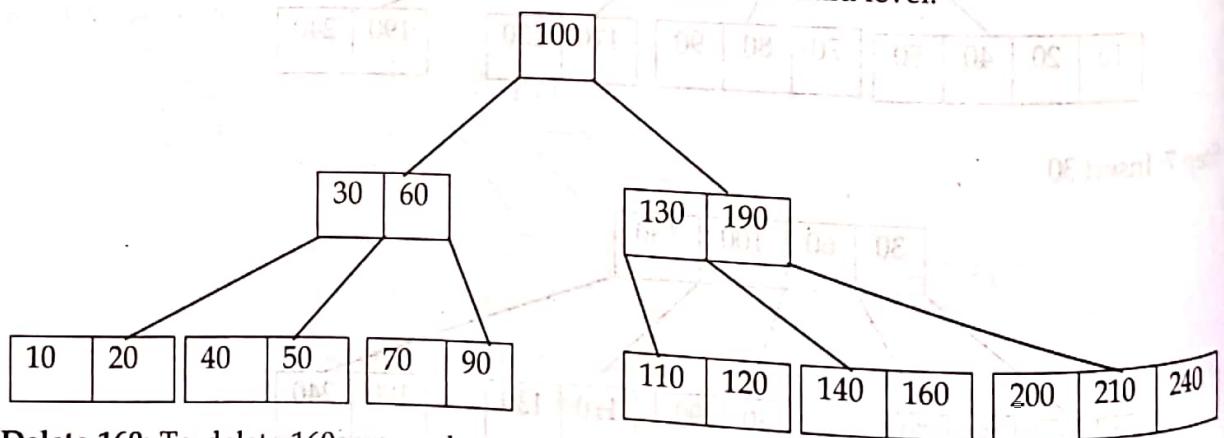
Step 9: Insert 160



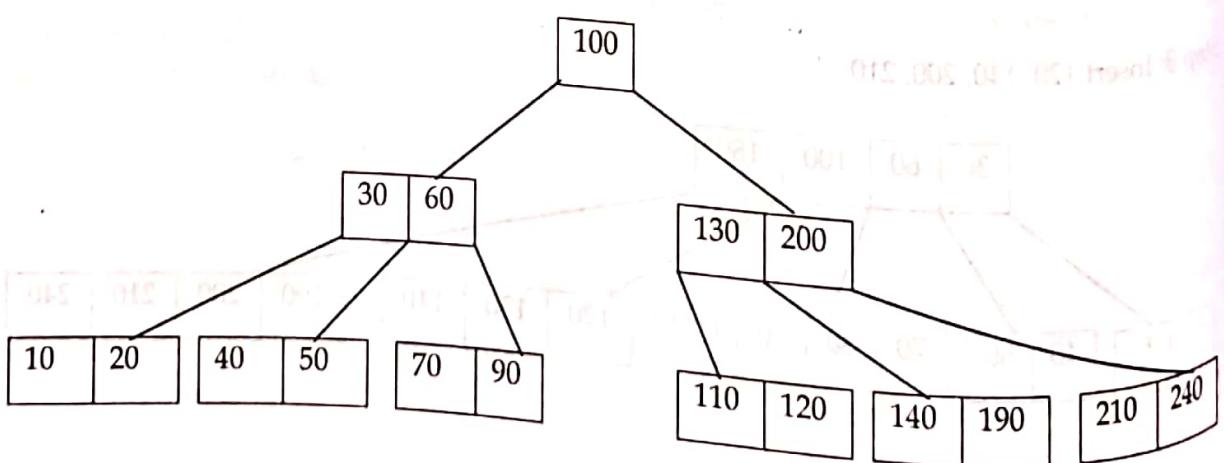
Delete 80: Simply delete 80 without performing any operation.



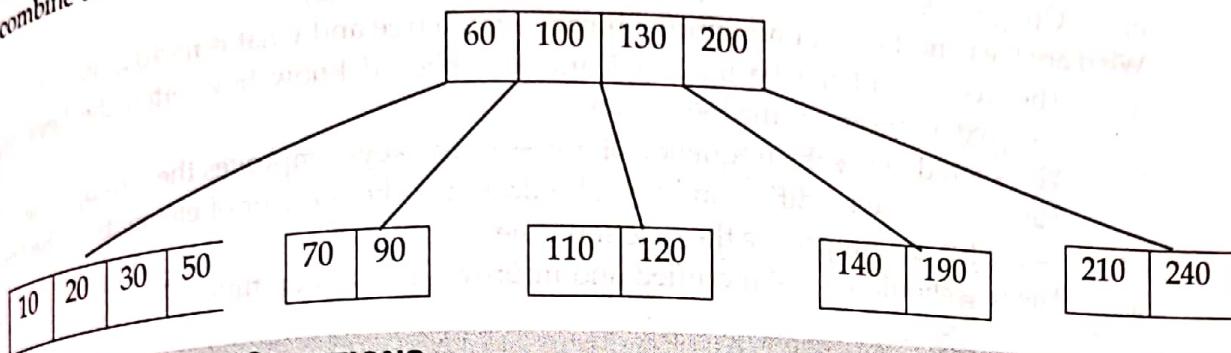
Delete 180: To delete 180 we need to promote 190 from their child level.



Delete 160: To delete 160 we need to demote 190 to their child note and promote 200 from their child level.



Delete 40: To delete 40 we need to combine 10, 20, 30, 40 and 50 as a left child of 60 and also combine 60, 100, 130 and 200 to be root level.



MULTIPLE CHOICE QUESTIONS

1. The number of edges from the root to the node is called _____ of the tree.
 - Height
 - Depth
 - Length
 - Width
2. The number of edges from the node to the deepest leaf is called _____ of the tree.
 - Height
 - Depth
 - Length
 - Width
3. What is a full binary tree?
 - Each node has exactly zero or two children
 - Each node has exactly two children
 - All the leaves are at the same level
 - Each node has exactly one or two children
4. What is a complete binary tree?
 - Each node has exactly zero or two children
 - A binary tree, which is completely filled, with the possible exception of the bottom level, which is filled from right to left
 - A binary tree, which is completely filled, with the possible exception of the bottom level, which is filled from left to right
 - A tree In which all nodes have degree 2
5. What is the average case time complexity for finding the height of the binary tree?
 - $h = O(\log \log n)$
 - $h = O(n \log n)$
 - $h = O(n)$
 - $h = O(\log n)$
6. Which of the following is not an advantage of trees?
 - Hierarchical structure
 - Faster search
 - Router algorithms
 - Undo/Redo operations in a notepad
7. In a full binary tree if there are L leaves, then total number of nodes N is?
 - $N = 2*L$
 - $N = L + 1$
 - $N = L - 1$
 - $N = 2*L - 1$
8. What does the following piece of code do?


```
public void func(Tree root)
{
    System.out.println(root.data());
    func(root.left());
    func(root.right());
}
```

 - Preorder traversal
 - Inorder traversal
 - Postorder traversal
 - Level order traversal

1. What is tree? Explain types of binary tree with suitable example.
2. How many internal nodes do the full binary tree of height $h = 3$ have?
3. What is binary tree? How it is different from BST?
4. How many internal nodes do a full binary tree of height $h = 9$ have?
5. How many nodes do a full binary tree of height $h = 9$ have?
6. What is the range of possible heights of a binary tree with $n = 100$ nodes?
7. What are the two main applications of heaps? How it is different from BST?
8. How efficient are insertions into and removals from a heap?
9. What is B-tree? Construct a B-tree of order 3 from given 10 elements.
10. Write down deletion process of an element from B-tree.
11. What do you mean by rotation operation in AVL tree?
12. What is complete tree? How it is different from AVL tree?
13. What is use of Huffman algorithm? Explain Huffman algorithm with suitable example.
14. If the nodes of a binary tree are numbered according to their natural mapping, and the visit operation prints the node's number, which traversal algorithm will print the numbers in order?
15. Draw the expression tree for $a * (b + c) * (d * e + f)$.
16. What are the bounds on the number of nodes in a binary tree of height 4?
17. What are the bounds on the height of a binary tree with 7 nodes?
18. What form does the highest child of a binary tree have?
19. What form does the lowest binary tree (i.e., the least height) have for a given number of nodes?
20. The order of visitation in the binary tree have for a given number of nodes?



DISCUSSION EXERCISE

- a. The tree should not be modified and improves the lookup time accessed, it improves the lookup cost
- b. You should know the frequency of access of the keys, improves the lookup time
- c. The tree can be modified and you should know the number of elements in the tree before hand, it improves the deletion time
- d. The tree should be just modified and improves the lookup time
9. What are the worst case and average case complexities of a binary search tree?
10. What are the conditions for an optimal binary search tree and what is its advantage?
- a) $O(n)$, $O(n)$
b) $O(n)$, $O(\log n)$
c) $O(\log n)$, $O(n)$
d) $O(n)$, $O(\log n)$
10. What are the conditions for an optimal binary search tree and what is its advantage?
- a. The tree should not be modified and you should know how often the keys are accessed, it improves the lookup cost
- b. You should know the frequency of access of the keys, improves the lookup time
- c. The tree can be modified and you should know the number of elements in the tree before hand, it improves the deletion time
- d. The tree should be just modified and improves the lookup time

8

Sort algorithms are used to sort data in ascending or descending order. There are many sorting algorithms. Some of them are quick sort, bubble sort, merge sort, insertion sort, selection sort, shell sort, heap sort, etc. In this chapter, we will study about quick sort algorithm. Quick sort is a divide and conquer algorithm. It divides the array into two halves and sorts them separately. The main idea behind quick sort is to select a pivot element from the array and partition the array around it such that all elements less than the pivot are on its left and all elements greater than the pivot are on its right. This process is repeated until the entire array is sorted.

SORTING

Sorting is the process of arranging data in a particular order. It is an important part of computer science. Sorting is used in many applications such as databases, file systems, and search engines. It is also used in many other fields such as chemistry, physics, and biology.

There are many different types of sorting algorithms. Some of them are quick sort, bubble sort, merge sort, insertion sort, selection sort, shell sort, heap sort, etc. Quick sort is a divide and conquer algorithm. It divides the array into two halves and sorts them separately. The main idea behind quick sort is to select a pivot element from the array and partition the array around it such that all elements less than the pivot are on its left and all elements greater than the pivot are on its right. This process is repeated until the entire array is sorted.



CHAPTER OUTLINE

After studying this chapter, the reader will be able to understand the

- Introduction, internal and external Sort, insertion and selection sort, exchange sort, Bubble and Quick sort, Merge and Radix sort, Shell Sort, Binary sort, Heap sort as priority queue, efficiency of sorting, Big O notation.

INTRODUCTION

Sorting is the process of ordering elements in an array in specific order e.g. ascending or descending on the basis of value, chronological ordering a records, priority order.

Sorting is categorized as internal sorting and external sorting.

1. **Internal sorting:** By internal sorting means we are arranging the numbers within the array only which is in computer primary memory.
2. **External sorting:** External sorting is the sorting of numbers from the external file by reading it from secondary memory.

Let P be a list of n elements $P_1, P_2, P_3, \dots, P_n$ in memory. Sorting P means arranging the contents of P in either increasing or decreasing order i.e.

$$P_1 \leq P_2 \leq P_3 \leq P_4 \leq P_5 \leq \dots \leq P_n$$

There are n elements in the list, therefore there are $n!$ Ways to arrange them.

Consider a list of values: 2, 4, 6, 8, 9, 1, 22, 4, 77, 8, 9

After sorting the values: 1, 2, 4, 4, 6, 8, 8, 9, 9, 22, 77

In-place

An in-place sorting algorithm uses constant extra space for producing the output (modifies the given array only). It sorts the list only by modifying the order of the elements within the list. For example, Insertion Sort and Selection Sorts are in-place sorting algorithms as they do not use any additional space for sorting the list and a typical implementation of Merge Sort is not in-place, also the implementation for counting sort is not in-place sorting algorithm.

Stable

A sorting algorithm is stable if two elements that are equal remain in the same relative position after sorting is completed. This is of interest, since in some sorting applications you sort first on one key and then on another. It is nice to know that two items that are equal on the second key remain sorted on the first key.

Why we using sorting?

We know that searching a sorted array is much easier than searching an unsorted array. This is especially true for people. That is, finding a person's name in a phone book is easy, but finding a phone number without knowing the person's name is virtually impossible. As a result, any significant amount of computer output is generally arranged in some sorted order so that it can be interpreted. The following are some more examples.

- Words in a dictionary are sorted
- Files in a directory are often listed in sorted order.
- The index of a book is sorted
- The card catalog in a library is sorted by both author and title.
- A listing of course offerings at a university is sorted, first by department and then by course number.
- Many banks provide statements that list checks in increasing order by check number.

- In a news paper, the calendar of events in a schedule is generally sorted by date.
- Musical compact disks in a record store are generally sorted by recording artist.
- In the programs printed for graduation ceremonies, departments are listed in sorted order and then students in those departments are listed in sorted order.

BUBBLE SORT

Bubble Sort is the simplest sorting algorithm that works by repeatedly swapping the adjacent elements if they are in wrong order. In Bubble sort, each element of the array is compared with its adjacent element. The algorithm processes the list in passes. A list with n elements requires $n-1$ passes for sorting. Consider an array A of n elements whose elements are to be sorted by using Bubble sort. The algorithm processes like following.

1. In Pass 1, $A[0]$ is compared with $A[1]$, $A[1]$ is compared with $A[2]$, $A[2]$ is compared with $A[3]$ and so on. At the end of pass 1, the largest element of the list is placed at the highest index of the list.
2. In Pass 2, $A[0]$ is compared with $A[1]$, $A[1]$ is compared with $A[2]$ and so on. At the end of Pass 2 the second largest element of the list is placed at the second highest index of the list.
3. In pass $n-1$, $A[0]$ is compared with $A[1]$, $A[1]$ is compared with $A[2]$ and so on. At the ends of this pass. The smallest element of the list is placed at the first index of the list.

Characteristics of Bubble Sort

- Large values are always sorted first.
- It only takes one iteration to detect that a collection is already sorted.
- The best time complexity for Bubble Sort is $O(n)$. The average and worst time complexity is $O(n^2)$.
- The space complexity for Bubble Sort is $O(1)$, because only single additional memory space is required.

Tracing: Sort the following data items by using Bubble sort

$A[] = \{25, 57, 48, 37, 12, 92, 86, 33\}$

Array position	0	1	2	3	4	5	6	7
Initial state	25	57	48	37	12	92	86	33
Pass 1	25	48	37	12	57	86	33	92
Pass 2	24	37	12	48	57	33	86	92
Pass 3	24	12	37	48	33	57	86	92
Pass 4	12	24	37	33	48	57	86	92
Pass 5	12	24	33	37	48	57	86	92
Pass 6	12	24	33	37	48	57	86	92
Pass 7	12	24	33	37	48	57	86	92
Pass 8	12	24	33	37	48	57	86	92

Algorithm

Let's look at implementing the optimized version of bubble sort:

1. Start
2. For the first iteration, compare all the elements (n). For the subsequent runs, compare ($n-1$) ($n-2$) and so on.
3. Compare each element with its right side neighbor.
4. Swap the smallest element to the left.
5. Keep repeating steps 1-3 until the whole list is covered.
6. Stop

Pseudo code

BubbleSort (A, n)

```
{
    for(i = 0; i < n-1; i++)
    {
        for(j = 0; j < n-i-1; j++)
        {
            if(A[j] > A[j+1])
                temp = A[j];
                A[j] = A[j+1];
                A[j+1] = temp;
        }
    }
}
```

Time Complexity

Inner loop executes ($n-1$) times when $i=0$, ($n-2$) times when $i=1$ and so on:

$$\begin{aligned} \text{Time complexity} &= (n-1) + (n-2) + (n-3) + \dots + 2 + 1 \\ &= n(n-1)/2 \\ &= O(n^2) \end{aligned}$$

There is no best-case linear time complexity for this algorithm.

Complete program in C for Bubble sort

```
#include<stdio.h>
#include<conio.h>
void bubble(int[], int);
void main()
{
    int n;
    int a[100], i;
    printf("Enter no of data items:\n");
    scanf("%d", &n);
    printf("Enter %d data items:\n", n);
    for(i=0; i < n; i++)
        scanf("%d", &a[i]);
```

```

printf("The data items before sorting:\n");
for(i=0; i<n; i++)
    printf("%d\t", a[i]);
bubble(a, n);
printf("The data items after sorting:\n");
for(i=0; i<n; i++)
    printf("%d\t", a[i]);
}

void bubble(int a[], int n) /*bubble function*/
{
    int i, j, temp;
    for(i=0; i<n-1; i++)
    {
        for(j=0; j<n-i-1; j++)
        {
            temp=a[j];
            if(a[j]>a[j+1])
            {
                a[j]=a[j+1];
                a[j+1]=temp;
            }
        }
    }
}

```

Input/output

Enter number of data points

10

Enter 10 numbers

66 5 44 3 55 44 2 1 7 9

The data items before sorting:

66 5 44 3 55 44 2 1 7 9

The data items after sorting:

1 2 3 5 7 9 44 44 55 66

SELECTION SORT

In selection sort, the smallest value among the unsorted elements of the array is selected in every pass and inserted to its appropriate position into the array. First, find the smallest element of the array and place it on the first position. Then, find the second smallest element of the array and place it on the second position. The process continues until we get the sorted array. The array with n elements is sorted by using $n-1$ pass of selection sort algorithm.

1. In 1st pass, smallest element of the array is to be found along with its index pos. then, swap $A[0]$ and $A[pos]$. Thus $A[0]$ is sorted, we now have $n-1$ elements which are to be sorted.
2. In 2nd pass, position pos of the smallest element present in the sub-array $A[n-1]$ is found. Then, swap, $A[1]$ and $A[pos]$. Thus $A[0]$ and $A[1]$ are sorted, we now left with $n-2$ unsorted elements.

3. In $n-1$ th pass, position pos of the smaller element between $A[n-1]$ and $A[n-2]$ is to be found. Then, swap, $A[pos]$ and $A[n-1]$and so on.

Tracing: Sort the following data items by using Selection sort

$$A[] = \{25, 57, 48, 37, 12, 92, 86, 33\}$$

Solution:

Array position	0	1	2	3	4	5	6	7
Initial state	25	57	48	37	12	92	86	33
Pass 1	12	57	48	37	25	92	86	33
Pass 2	12	25	48	37	57	92	86	33
Pass 3	12	25	33	37	57	92	86	48
Pass 4	12	25	33	37	57	92	86	48
Pass 5	12	25	33	37	48	92	86	57
Pass 6	12	25	33	37	48	57	86	92
Pass 7	12	25	33	37	48	57	86	92
Pass 8	12	25	33	37	48	57	86	92

Algorithm

- Start
- Consider the first element to be sorted and the rest to be unsorted
- Assume the first element to be the smallest element.
- Check if the first element is smaller than each of the other elements:
 - If yes, do nothing
 - If no, choose the other smaller element as minimum and repeat step 3
- After completion of one iteration through the list, swap the smallest element with the first element of the list.
- Now consider the second element in the list to be the smallest and so on till all the elements in the list are covered.
- Stop

Pseudo code

SelectionSort(A)

```
{
  for( i = 0; i < n ; i++)
  {
    least = A[i];
    p=i;
    for ( j = i + 1; j < n; j++)
    {
      if (A[j] < A[i])
      {
        least = A[j];
        p=j;
      }
    }
    swap (A[i], A[p]);
  }
}
```

Time Complexity

Inner loop executes for $(n-1)$ times when $i=0$, $(n-2)$ times when $i=1$ and so on: $+2+1$
 Time complexity = $(n-1) + (n-2) + (n-3) + \dots$
 $= O(n^2)$

There is no best-case linear time complexity for this algorithm, but number of swap operations is reduced greatly.

Complete program in C for Selection sort

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

```
void selection(int[ ], int);
```

```
void main()
```

```
{
    int n;
    int a[100], i;
    printf("Enter no of data items:\n");
    scanf("%d", &n);
    printf("Enter %d data items:\n", n);
    for(i=0; i<n; i++)
        scanf("%d", &a[i]);
    printf("The data items before sorting:\n");
    for(i=0; i<n; i++)
        printf("%d\t", a[i]);
    selection(a, n);
    printf("The data items after sorting:\n");
    for(i=0; i<n; i++)
        printf("%d\t", a[i]);
    getch();
}
```

```
/*selection Function*/
```

```
void selection(int a[ ], int n)
```

```
{
    int i, j, temp, index, least;
    for(i=0; i<n; i++)
    {
        least=a[i];
        index=i;
        for(j=i+1; j<n; j++)
        {
            if(a[j]<least)
            {
                least=a[j];
                index=j;
            }
        }
        if(i!=index)
        {
            temp=a[i];
            a[i]=a[index];
            a[index]=temp;
        }
    }
}
```

Input/output

Enter number of data points

10

Enter 10 numbers

66 5 44 3 55 44 2 1 7 9

The data items before sorting:

66 5 44 3 55 44 2 1 7 9

The data items after sorting:

1 2 3 5 7 9 44 44 55 66

INSERTION SORT

Insertion sort is the simple sorting algorithm which is commonly used in the daily lives while ordering a deck of cards. In this algorithm, we insert each element onto its proper place in the sorted array.

Consider an array A whose elements are to be sorted. Initially, A[0] is the only element on the sorted set. In pass 1, A[1] is placed at its proper index in the array.

In pass 2, A[2] is placed at its proper index in the array. Likewise, in pass n-1, A[n-1] is placed at its proper index into the array.

To insert an element A[k] to its proper index, we must compare it with all other elements i.e. A[k-1], A[k-2], and so on until we find an element A[j] such that, A[j] <= A[k].

All the elements from A[k-1] to A[j] need to be shifted and A[k] will be moved to A[j+1].

Tracing: Sort the following data items by using Insertion sort

A[]={25, 57, 48, 37, 12, 92, 86, 33}

Solution:

Array position	0	1	2	3	4	5	6	7
Initial state	25	57	48	37	12	92	86	33
After a[0..1] is sorted (pass 1)	25	57	48	37	12	92	86	33
After a[0..2] is sorted (pass 2)	25	57	48	37	12	92	86	33
After a[0..3] is sorted (pass 3)	25	48	57	37	12	92	86	33
After a[0..4] is sorted (pass 4)	25	37	48	57	12	92	86	33
After a[0..5] is sorted (pass 5)	12	25	37	48	57	92	86	33
After a[0..6] is sorted (pass 6)	12	25	37	48	57	92	86	33
After a[0..7] is sorted (pass 7)	12	25	37	48	57	86	92	33
After a[0..8] is sorted (pass 8)	12	25	33	37	48	57	86	92

Algorithm

1. Start
2. Consider the first element to be sorted and the rest to be unsorted

- Compare with the second element:
3. i. If the second element < the first element, insert the element in the correct position of the sorted portion
 - ii. Else, leave it as it is
 - Repeat 1 and 2 until all elements are sorted
 4. Stop
 - 5.

Pseudo code

Insertion(A, n)

Input: An array A and its size n

```

| for i = 1 to n
|   temp = A[i]
|   j = i - 1
|   while(j >= 0 && A[j] > temp)
|     {
|       A[j + 1] = A[j]
|       j = j - 1
|     }
|   A[j + 1] = temp
}

```

Complete program in C for Insertion sort

```

#include<stdio.h>
#include<conio.h>
void insertion(int[], int);
void main()
{
    int n;
    int a[100], i;
    clrscr();
    printf("Enter no of data items:\n");
    scanf("%d", &n);
    printf("Enter %d data items:\n", n);
    for(i=0; i<n; i++)
    {
        scanf("%d", &a[i]);
    }
    printf("The data items before sorting:\n");
    for(i=0; i<n; i++)
    {
        printf("%d\t", a[i]);
    }
    insertion(a, n);
    printf("The data items after sorting:\n");
    for(i=0; i<n; i++)

```

```

Input: {
    printf("%d\t", a[i]);
}
getch();
}

/*insertion Function*/
void insertion(int a[ ], int n)
{
    int i, j, temp;
    for(i=0; i<n; i++)
    {
        temp = a[i];
        j=i-1;
        while((temp<a[j])&&j>=0)
        {
            a[j+1]=a[j];
            j--;
        }
        a[j+1]=temp;
    }
}

```

Input/output

Enter number of data points

10

Enter 10 numbers

69 5 44 3 55 45 2 1 7 9

The data items before sorting:

69 5 44 3 55 45 2 1 7 9

The data items after sorting:

1 2 3 5 7 9 44 45 55 69

Divide-and-conquer algorithms

An important problem-solving technique that makes use of recursion is divide and conquer. A divide-and-conquer algorithm is an efficient recursive algorithm that consists of two parts:

- **Divide**, in which smaller problems are solved recursively (except, of course, base cases)
- **Conquer**, in which the solution to the original problem is then formed from the solutions to the sub-problems

Traditionally, routines in which the algorithm contains at least two recursive calls are called divide-and-conquer algorithms, whereas routines whose text contains only one recursive call are not. Consequently, the recursive routines presented so far in this section are not divide-and-conquer algorithms. Also, the sub-problems usually must be disjoint (i.e., essentially no overlapping), so as to avoid the excessive costs seen in the sample recursive computation of the Fibonacci numbers.

QUICK SORT

As its name implies, **quick sort** is a fast divide-and-conquer algorithm. Its average running time is $O(n \log n)$. Its speed is mainly due to a very tight and highly optimized inner loop. It has quadratic worst-case performance, which can be made statistically unlikely to occur with a little effort. On the one hand, the quicksort algorithm is relatively simple to understand and prove correct because it relies on recursion. On the other hand, it is a tricky algorithm to implement because minute changes in the code can make significant differences in running time. It has two phases:

- The partition phase and
- The sort phase

As we will see, most of the work is done in the partition phase - it works out where to divide the work. The sort phase simply sorts the two smaller problems that are generated in the partition phase. This makes Quick sort a good example of the **divide and conquers** strategy for solving problems.

- **Divide**

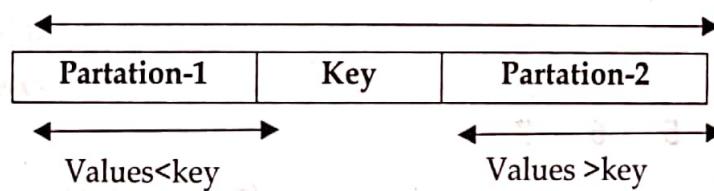
Partition the array $A[l \dots r]$ into two sub-arrays $A[l \dots \text{pivot}]$ and $A[\text{pivot}+1 \dots r]$, such that each element of $A[l \dots \text{pivot}]$ is smaller than or equal to each element in $A[\text{pivot}+1 \dots r]$.

- **Conquer**

Recursively sort $A[l \dots \text{pivot}]$ and $A[\text{pivot}+1 \dots r]$ using Quick sort

- **Combine**

Trivial: the arrays are sorted in place. No additional work is required to combine them.



Tracing: Sort the following data items by using Quick sort

$$A[] = \{5, 3, 2, 6, 4, 1, 3, 7\}$$

Pass 1:

5	3	2	6	4	1	3	7
L	P	R					

5	3	2	6	4	1	3	7
P	L						R

5	3	2	3	4	1	6	7
P	L						R

5	3	2	3	4	1	6	7
P	R						L

1	3	2	3	4	5	6	7
P		R	L				

1	3	2	3	4	5	6	7
---	---	---	---	---	---	---	---

Pass 2:

1	3	2	3	4	5	6	7
L, P		R		L	R		
P,	L		R	L			

Pass 3:

1	3	2	3	4	5	6	7
L, P		R					
P,	R	L					

Pass 4:

1	3	2	3	4	5	6	7
L	R						

Pass 5:

1	3	2	3	4	5	6	7
L, P	R						

1	3	2	3	4	5	6	7
P	L, R						

1	2	3	3	4	5	6	7
P	L, R						

1	2	3	3	4	5	6	7

Algorithm

1. Start
2. Choose a pivot
3. Set a left pointer and right pointer

4. Compare the left pointer element (lelement) with the pivot and the right pointer element (relement) with the pivot.
5. Check if lelement < pivot and relement > pivot:

 - If yes, increment the left pointer and decrement the right pointer
 - If not, swap the lelement and relement

- When left >= right, swap the pivot with either left or right pointer.
- Repeat steps 1 - 5 on the left half and the right half of the list till the entire list is sorted.
- Stop

Pseudo code

QuickSort(A, l, r)

```

{
    if(l < r)
    {
        p = Partition(A, l, r);
        QuickSort(A, l, p-1);
        QuickSort(A, p+1, r);
    }
}

Partition(A, l, r)
{
    x = l;
    y = r;
    p = A[l];
    while(x < y)
    {
        while(A[x] <= p)
            x++;
        while(A[y] >= p)
            y--;
        if(x < y)
            swap(A[x], A[y]);
    }
    A[l] = A[y];
    A[y] = p;
    return y; /*return position of pivot*/
}

```

Time Complexity

Best Case

Quick sort gives best time complexity when elements are divided into two partitions of equal size; therefore recurrence relation for this case is;

$$T(n) = 2T(n/2) + O(n)$$

By solving this recurrence, we get,

$$T(n) = O(n \log n)$$

Worst case

Quick sort gives worst case when elements are already sorted. In this case one partition contains the $n-1$ elements and another partition contains no element. Therefore, its recurrence relation is;

$$T(n) = T(n-1) + O(n)$$

By solving this recurrence relation, we get

$$T(n) = O(n^2)$$

Average case

It is the case between best case and worst case. All permutations of the input numbers are equally likely. On a random input array, we will have a mix of well balanced and unbalanced splits. Good and bad splits are randomly distributed across throughout the tree. Suppose we are alternate: Balanced, Unbalanced, Balanced.

$$B(n) = 2UB(n/2) + \Theta(n) \text{ Balanced}$$

$$UB(n) = B(n-1) + \Theta(n) \text{ Unbalanced}$$

$$\begin{aligned} \text{Solving: } B(n) &= 2(B(n/2 - 1) + \Theta(n/2)) + \Theta(n) \\ &= 2B(n/2 - 1) + \Theta(n) \\ &= \Theta(n \log n) \end{aligned}$$

Complete program in C for Quick sort

```
#include<stdio.h>
#include<conio.h>
int partition(int a[10], int l, int r)
{
    int x=l;
    int y=r;
    int p=a[l], temp;
    while(x<y)
    {
        while(a[x]<=p)
            x++;
        while(a[y]>p)
            y--;
        if(x<y)
    }
```

```

    {
        temp=a[x];
        a[x]=a[y];
        a[y]=temp;
    }
}

```

```

a[l]=a[y];
a[y]=p;
return y;
}

```

```
void quick(int a[10], int l, int r)
```

```

int p;
if(l<r)
{
    p=partition(a, l, r);
    quick(a, l, p-1);
    quick(a, p+1, r);
}
}

```

```
void main()
```

```

int a[100], n, i, l, r;
printf("Enter no of elements\n");
scanf("%d", &n);
printf("Enter %d elements", n);
l=0;
r=n-1;
for(i=0; i<n; i++)
{
    scanf("%d", &a[i]);
}

```

```

printf("elements before sort:\n");
for(i=0; i<n; i++)
{

```

```
    printf("%d\t", a[i]);
}

```

```
quick(a, l, r);

```

```
printf("elements after sort:\n");
for(i=0; i<n; i++)
{

```

```
    printf("%d\t", a[i]);
}

```

```
getch();
}

```

Input/output

Enter number of data points

10

Enter 10 numbers

69 5 44 3 55 45 2 1 7 9

The data items before sorting:

69 5 44 3 55 45 2 1 7 9

The data items after sorting:

1 2 3 5 7 9 44 45 55 69

MERGE SORT

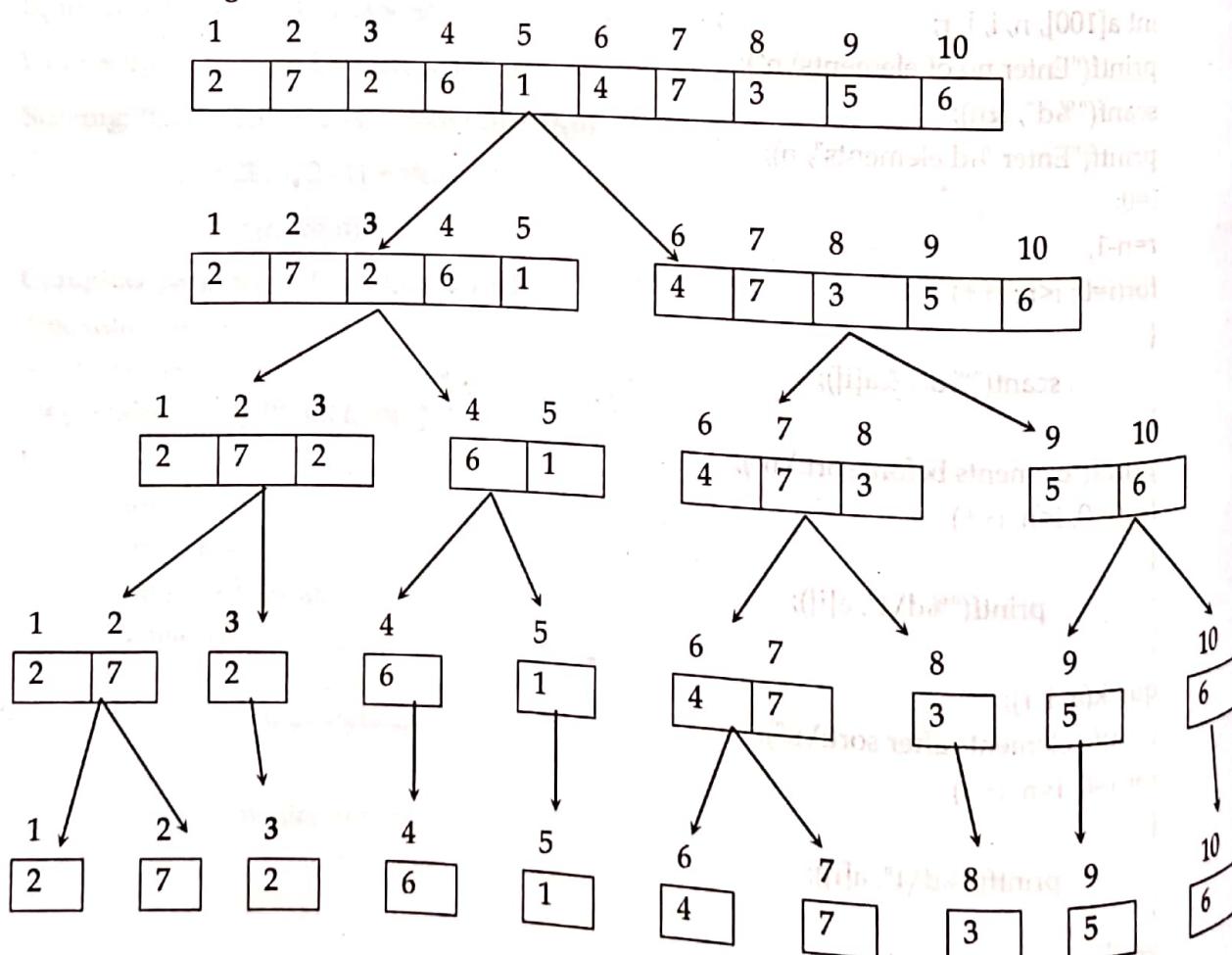
Merge sort is an efficient sorting algorithm which involves merging two or more sorted files into a third sorted file. Merging is the process of combining two or more sorted files into a third sorted file. The merge sort algorithm is based on divide and conquer method.

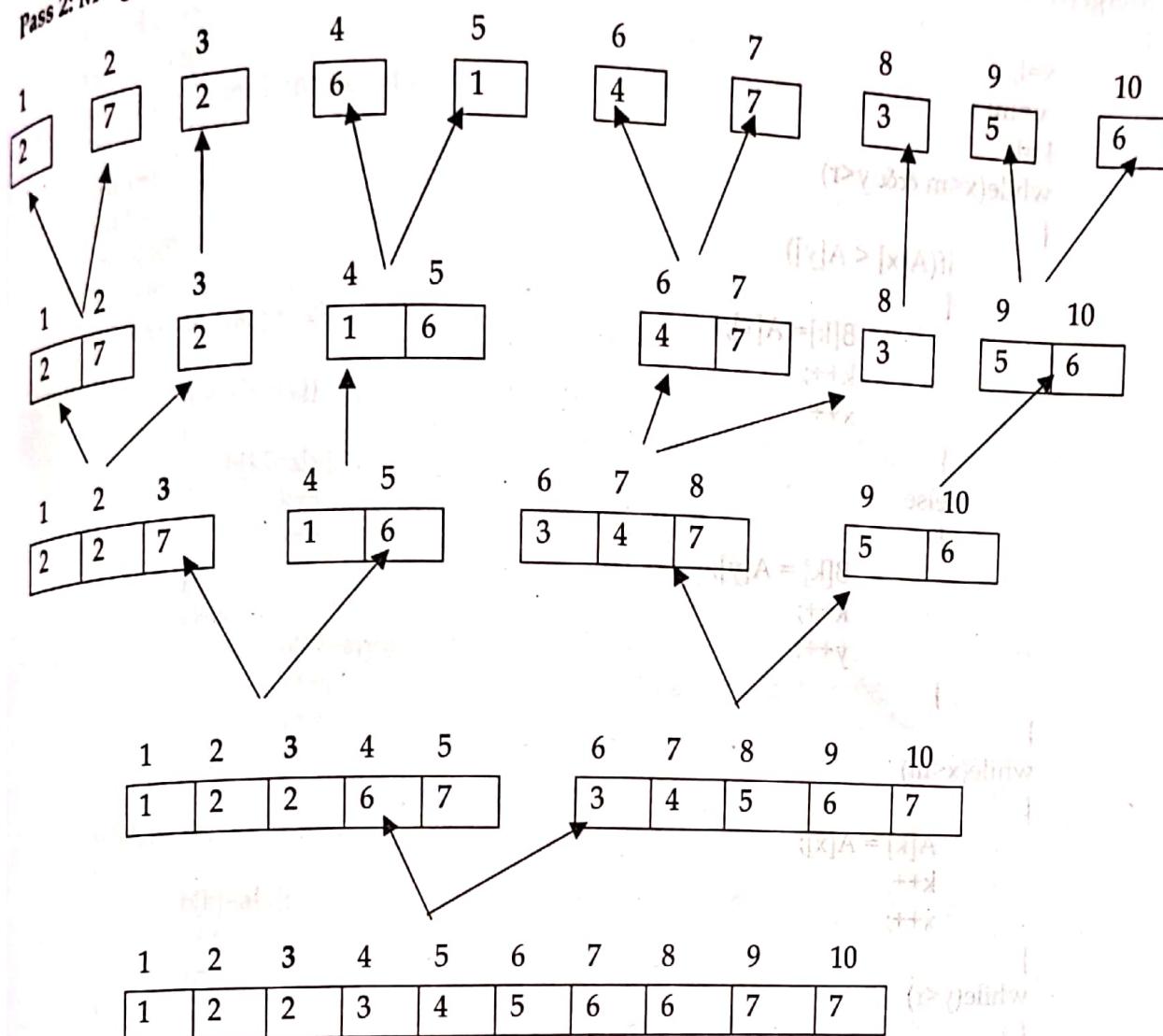
The process of merge sort can be formalized into three basic operations.

1. Divide the array into two sub arrays
2. Recursively sort the two sub arrays
3. Merge the newly sorted sub arrays

Tracing: $A[] = [4, 7, 2, 6, 1, 4, 7, 3, 5, 6]$

Pass 1: Dividing



Pass 2: Merging**Algorithm**

1. Start
2. Split the unsorted list into two groups recursively until there is one element per group
3. Compare each of the elements and then sort and group them
4. Repeat step 2 until the whole list is merged and sorted in the process
5. Stop

Pseudo code

```
MergeSort(A, l, r)
{
```

```
    If(l < r)
    {
        m = ⌊(l + r)/2⌋          //Divide
        MergeSort(A, l, m)        //Conquer
        MergeSort(A, m + 1, r)    //Conquer
        Merge(A, l, m+1, r)       //Combine
    }
```

Merge(A, B, l, m, r)

```

{
    x=l;
    y=m;
    k=l;
    while(x<m && y<r)
    {
        if(A[x] < A[y])
        {
            B[k]=A[x];
            k++;
            x++;
        }
        else
        {
            B[k]=A[y];
            k++;
            y++;
        }
    }
    while(x<m)
    {
        A[k]=A[x];
        k++;
        x++;
    }
    while(y<r)
    {
        A[k]=A[y];
        k++;
        y++;
    }
}
for(i=l; i<=r; i++)
    A[i] = B[i]
}

```

Time Complexity

No of sub-problems=2

Size of each subproblem=n/2

Dividing cost=constant

Merging cost=n

Thus recurrence relation for Merge sort is;

$$T(n) = 1 \quad \text{if } n=1$$

$$T(n) = 2 T(n/2) + O(n) \quad \text{if } n>1$$

By solving this recurrence relation, we get,

$$\text{Time Complexity} = T(n) = O(n \log n)$$

Complete program in C for Merge sort

```

#include<stdio.h>
#include<conio.h>
void merge(int a[ ], int l, int m, int r)
{
    int x=l;
    int k=l, i;
    int b[10];
    int y=m;
    while(x<m && y<=r)
    {
        if(a[x]<a[y])
        {
            b[k]=a[x];
            k++;
            x++;
        }
        else{
            b[k]=a[y];
            k++;
            y++;
        }
    }
    while(x<m)
    {
        b[k]=a[x];
        k++;
        x++;
    }
    while(y<=r)
    {
        b[k]=a[y];
        y++;
        k++;
    }
    for(i=l; i<=r; i++)
    {
        a[i]=b[i];
    }
}
void merge_sort(int a[ ], int l, int r)
{
    int mid;
    if(l<r)
    {
        mid=(l+r)/2;
        merge_sort(a, l, mid);
        merge_sort(a, mid+1, r);
        merge(a, l, mid+1, r);
    }
}

```

```

void main()
{
    int a[100], n, i, l, r;
    printf("Enter no of elements\n");
    scanf("%d", &n);
    printf("Enter %d elements", n);
    l=0;
    r=n-1;
    for(i=0; i<n; i++)
    {
        scanf("%d", &a[i]);
    }
    printf("elements before sort:\n");
    for(i=0; i<n; i++)
    {
        printf("%d\t", a[i]);
    }
    merge_sort(a, l, r);
    printf("\n elements after sort:\n");
    for(i=0; i<n; i++)
    {
        printf("%d\t", a[i]);
    }
    getch();
}

```

Input/output

Enter number of data points

10

Enter 10 numbers

69 5 44 3 55 45 2 1 7 9

The data items before sorting:

69 5 44 3 55 45 2 1 7 9

The data items after sorting:

1 2 3 5 7 9 44 45 55 69

SHELL SORT

Shell sort is the generalization of insertion sort which overcomes the drawbacks of insertion sort by comparing elements separated by a gap of several positions. In general, Shell sort performs the following steps.

- **Step 1:** Arrange the elements in the tabular form and sort the columns by using insertion sort.
- **Step 2:** Repeat Step 1; each time with smaller number of longer columns in such a way that at the end, there is only one column of data to be sorted.

Though it is not the fastest algorithm known, Shell sort is a sub quadratic algorithm whose code is only slightly longer than the insertion sort, making it the simplest of the faster

algorithms. Shells idea was to avoid the large amount of data movement, first by comparing elements that were far apart and then by comparing elements that were less far apart, and so on, gradually shrinking toward the basic insertion sort. The shell sort is a diminishing increment sort. This sort divides the original file into separate sub-files. These sub-files contain every k^{th} element of the original file. The value of k is called increment.

For example, if there are n elements to be sorted and value of k is five then,

Sub-file 1 $\rightarrow a[0], a[5], a[10], a[15]$

Sub-file 2 $\rightarrow a[1], a[6], a[11], a[16]$

Sub-file 3 $\rightarrow a[2], a[7], a[12], a[17]$

Sub-file 4 $\rightarrow a[3], a[8], a[13], a[18]$

Sub-file 5 $\rightarrow a[4], a[9], a[14], a[19]$

Tracing: $A[] = \{4, 7, 2, 8, 1, 4, 7, 3, 5\}$

Pass 1:

4	7	2	8	1	4	7	3	5

Increment=6(say):

4	3	2	8	1	4	7	7	5

Pass 2:

4	3	2	8	1	4	7	7	5

Increment=6/2=3:

4	1	2	8	3	4	7	7	5

4	1	2	7	3	4	8	7	5

Pass 3:

4	1	2	7	3	4	8	7	5

Increment=3/2=1:

1	2	3	4	4	5	7	7	8

Pass 3:

4	1	2	7	3	4	8	7	5

Increment=3/2=1:

1	2	3	4	4	5	7	7	8

Complete program in C for Shell sort

```

void main()
{
    int array[100], n;
    int k, i, j, increment, temp;
    printf("Enter number of elements\n");
    scanf("%d", &n);
    for (increment = n/2; increment > 0; increment /= 2)
    {
        for (i = increment; i < n; i++)
        {
            temp = array[i];
            for (j = i; j >= increment; j -= increment)
            {
                if (temp < array[j - increment])
                {
                    array[j] = array[j - increment];
                }
                else
                {
                    break;
                }
            }
            array[j] = temp;
        }
    }
    printf("After Sorting:");
    for (k = 0; k < 5; k++)
    {
        printf("\t%d", array[k]);
    }
}

```

Input/output

Enter number of data points

10

Enter 10 numbers

69 5 44 3 55 45 2 1 7 9

The data items before sorting:

69 5 44 3 55 45 2 1 7 9

The data items after sorting:

1 2 3 5 7 9 44 45 55 69

RADIX SORT

Radix sort is a small method that many people intuitively use when alphabetizing a large list of names. Specifically, the list of names is first sorted according to the first letter of each name, that is, the names are arranged in 26 classes. Intuitively, one might want to sort numbers on their most significant digit. However, Radix sort works counter-intuitively by sorting on the least significant digits first. On the first pass, all the numbers are sorted on the least significant digit and combined in an array. Then on the second pass, the entire numbers are sorted again on the second least significant digits and combined in an array and so on.

Key points of radix sort algorithm

1. Radix Sort is a linear sorting algorithm.
2. Time complexity of Radix Sort is $O(n^d)$, where n is the size of array and d is the number of digits in the largest number.
3. It is not an **in-place sorting** algorithm as it requires extra additional space.
4. Radix Sort is **stable sort** as relative order of elements with equal values is maintained.
5. Radix sort can be slower than other sorting algorithms like merge sort and quick sort, if the operations are not efficient enough. These operations include inset and delete functions of the sub-list and the process of isolating the digits we want.
6. Radix sort is less flexible than other sorts as it depends on the digits or letter.

Algorithm

Radix-Sort (list, n)

```
{
    shift = 1
    for loop = 1 to keyszie do
        for entry = 1 to n do
            bucketnumber = (list[entry].key / shift) mod 10
            append (bucket[bucketnumber], list[entry])
        list = combinebuckets()
        shift = shift * 10
}
```

1	2	3	4	5	6	7	8	9	10
10	2	901	803	1024					
10	2	901	803	1024					
10	2	901	803	1024					
10	2	901	803	1024					
10	2	901	803	1024					
10	2	901	803	1024					
10	2	901	803	1024					
10	2	901	803	1024					
10	2	901	803	1024					

Analysis

Each key is looked at once for each digit or letter if the keys are alphabetic of the longest key. Hence, if the longest key has m digits and there are n keys, radix sort has order $O(m \cdot n)$. However, if we look at these two values, the size of the keys will be relatively small when compared to the number of keys. For example, if we have six-digit keys, we could have a million different records. Here, we see that the size of the keys is not significant, and this algorithm is of linear complexity $O(n)$.

Example 1: Consider the array of length 6 given below. Sort the array by using Radix sort.

$$A = \{10, 2, 901, 803, 1024\}$$

Pass 1: (Sort the list according to the digits at 0's place)

$$10, 901, 2, 803, 1024$$

Pass 2: (Sort the list according to the digits at 10's place)

$$02, 10, 901, 803, 1024$$

Pass 3: (Sort the list according to the digits at 100's place)

$$02, 10, 1024, 803, 901$$

Pass 4: (Sort the list according to the digits at 1000's place)

$$02, 10, 803, 901, 1024$$

Therefore, the list generated in the step 4 is the sorted list, arranged from radix sort.

Example 2: Assume the input array is:
 [10, 21, 17, 34, 44, 11, 654, 123]

Based on the algorithm, we will sort the input array according to the one's digit (least significant digit).

0	10
1	21, 11
2	
3	123
4	34, 44, 654
5	
6	
7	17
8	
9	

So, the array becomes 10, 21, 11, 123, 34, 44, 654, 17 every value fall in the cell.

Now, we'll sort according to the ten's digit:

0	
1	10, 11, 17
2	21, 123
3	34
4	44
5	654
6	
7	
8	
9	

Now, the array becomes: 10, 11, 17, 21, 123, 34, 44, 654

Finally, we sort according to the hundred's digit (most significant digit):

0	010, 011, 017, 021, 034, 044
1	123
2	
3	
4	
5	
6	654
7	
8	
9	

The array becomes: [10, 11, 17, 21, 34, 44, 123, 654] which is sorted.

Complete program in C for simulation of Radix sort

```

#include <stdio.h>
int largest(int a[]);
void radix_sort(int a[]);
void main()
{
    int i;
    int a[10]={90,23,101,45,65,23,67,89,34,23};
    radix_sort(a);
    printf("\n The sorted array is: \n");
    for(i=0;i<10;i++)
        printf(" %d\t", a[i]);
}

int largest(int a[])
{
    int larger=a[0], i;
    for(i=1;i<10;i++)
    {
        if(a[i]>larger)
            larger = a[i];
    }
    return larger;
}

void radix_sort(int a[])
{
    int bucket[10][10], bucket_count[10];
    int i, j, k, remainder, NOP=0, divisor=1, larger, pass;
    larger = largest(a);
    while(larger>0)
    {
        NOP++;
        larger/=10;
    }
    for(pass=0;pass<NOP;pass++) // Initialize the buckets
    {
        for(i=0;i<10;i++)
            bucket_count[i]=0;
        for(i=0;i<10;i++)
        {
            remainder = (a[i]/divisor)%10;
            bucket[remainder][bucket_count[remainder]] = a[i];
            bucket_count[remainder] += 1;
        }
        i=0;
        for(k=0;k<10;k++)
        {
            for(j=0;j<bucket_count[k];j++)

```

```

    {
        a[i] = bucket[k][j];
        i++;
    }
}
divisor *= 10;
}
}

```

HEAPS

A heap is an almost complete binary tree whose elements have keys that satisfy the following heap property: the keys along any path from root to leaf are descending (i.e. non-increasing). Heaps could represent family descendant trees because the heap property means that every parent is older than its children.

In brief, a heap is an almost complete binary tree of n nodes such that the value of each node is less than or equal to the value in parent node. This type of heap is called max heap. By default, the heap is max heap. There are two types of heap:

1. Max heap and
2. Min heap

Heaps are used to implement priority queues and to the heap sort algorithm.

Example: Construct a heap from a set of 6 elements {15, 19, 10, 7, 17, 16}

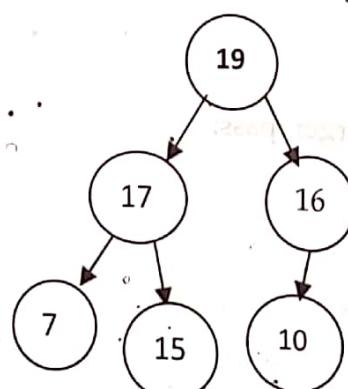


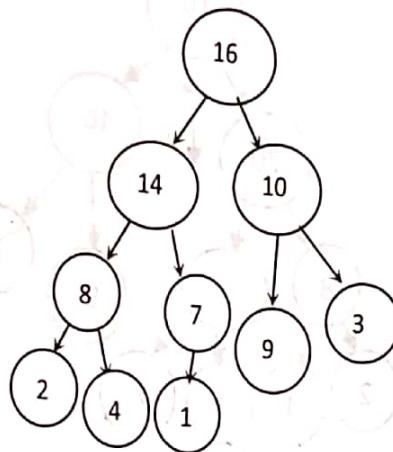
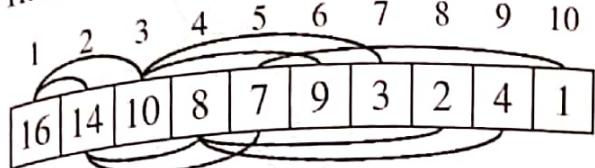
Fig Heap of given set of elements

Array Representation of Heaps

Every complete binary tree has a natural mapping into an array. The mapping is obtained from a level-order traversal of the tree. In the resulting array, the parent of the element at index i is at index $i/2$, and the children are at indexes $2i$ and $2i+1$.

- A heap can be stored as an array A .
- Root of tree is $A[1]$
- Left child of $A[i] = A[2i]$
- Right child of $A[i] = A[2i + 1]$
- Parent of $A[i] = A[\lfloor i/2 \rfloor]$
- $\text{Heapsize}[A] \leq \text{length}[A]$

The elements in the sub-array $A[(\lfloor n/2 \rfloor + 1) \dots n]$ are leaves

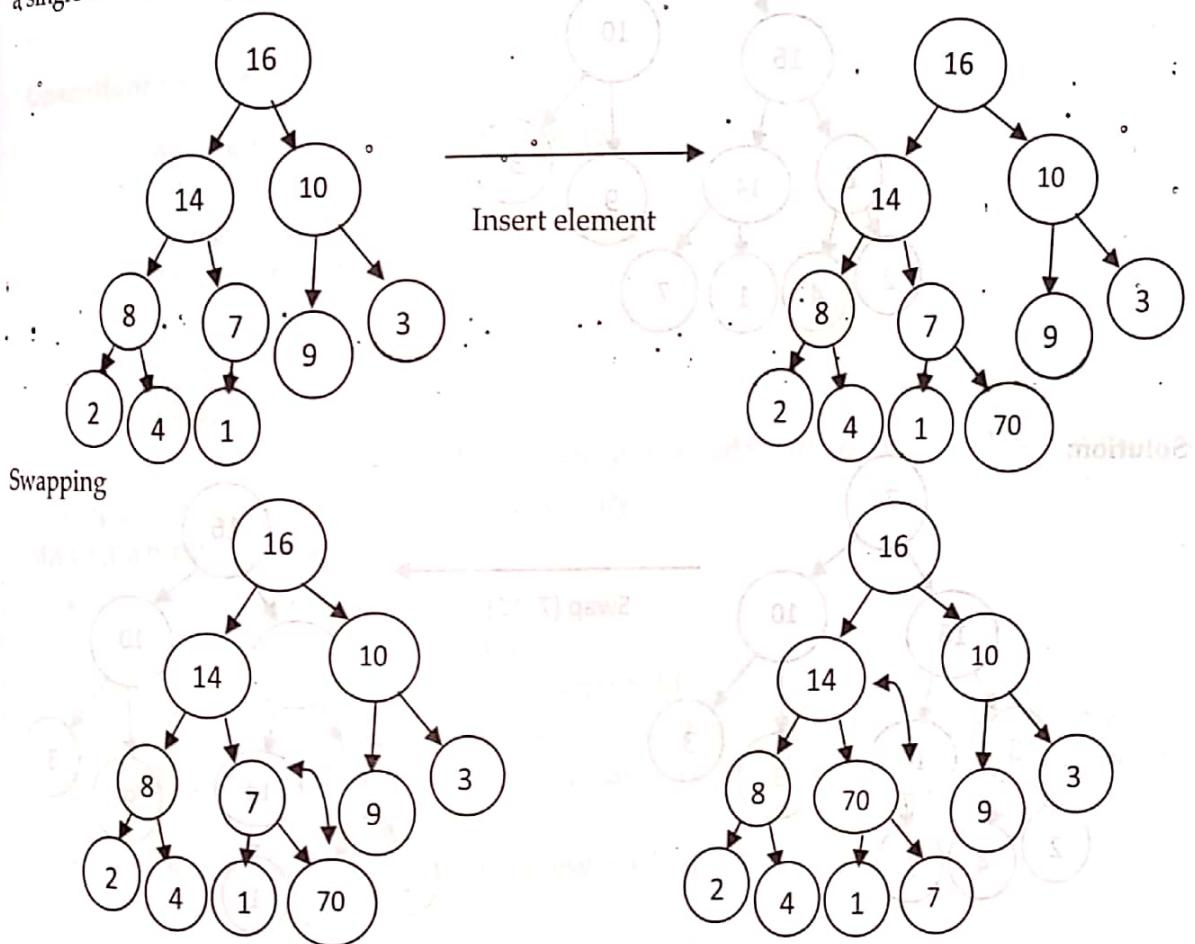


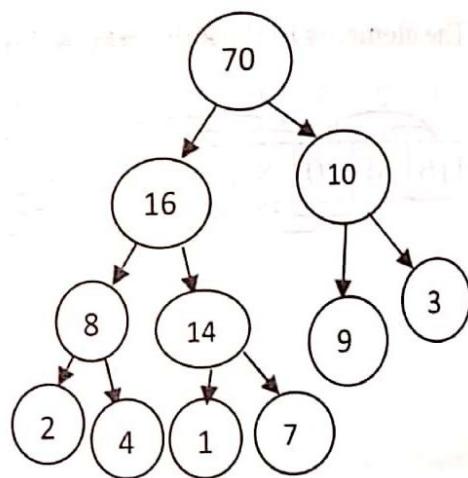
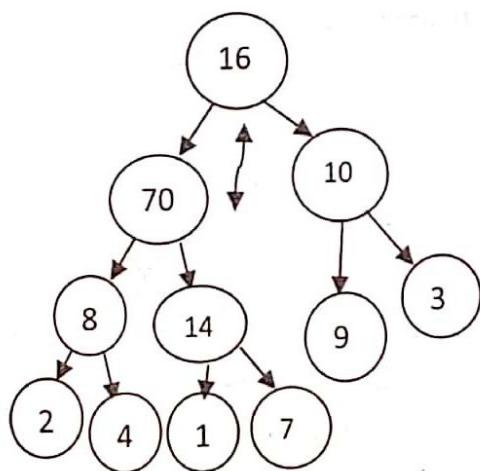
Inserting element to an existing heap

Elements are inserted into a heap next to its right-most leaf at the bottom level. Then the heap property is restored by percolating the new element up the tree until it is no longer "older" (i.e., its key is greater) than its parent. On each iteration, the child is swapped with its parent.

Example: Insert element 70 to given heap

Figure below shows how the key 70 would be inserted into the heap. The element 70 is added to the tree as a new last leaf. Then it is swapped with its parent element 7 because $70 > 7$. Then it is swapped with its parent element 14 because $70 > 14$, again swap it with its parent element 16 because $70 > 16$. Now the heap property has been restored because the new element 70 is less than its parent and greater than its children. Note that the insertion affects only the nodes along a single root-to-leaf path.

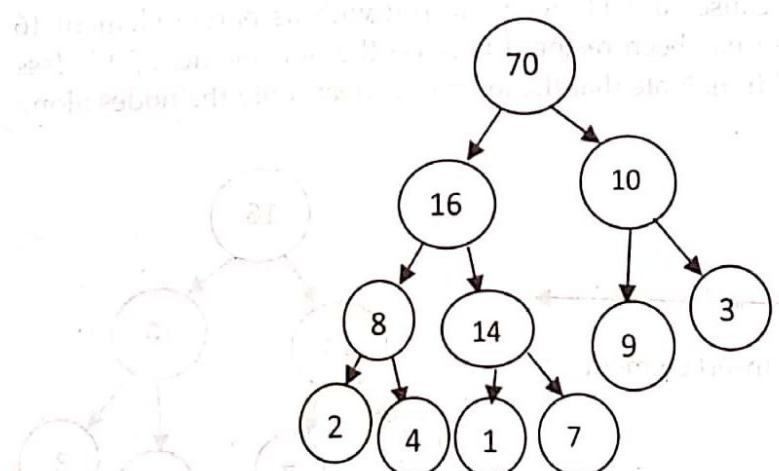




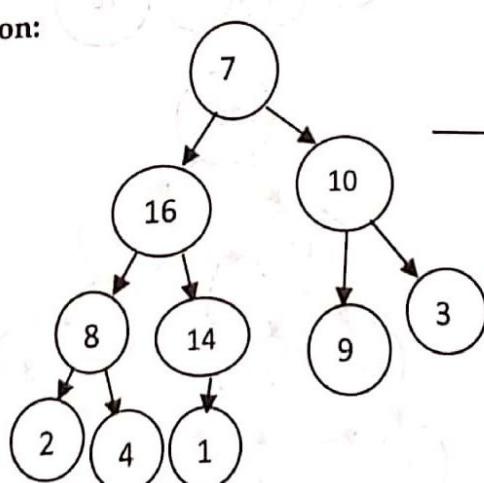
Deleting element from an existing heap

The heap removal algorithm always removes the root element from the tree. This is done by moving the last leaf element into the root element and then restoring the heap property by collating the new root element down the tree until it is no longer "younger" (i.e., its key is less) than its children. On each iteration, the parent is swapped with the older of its two children.

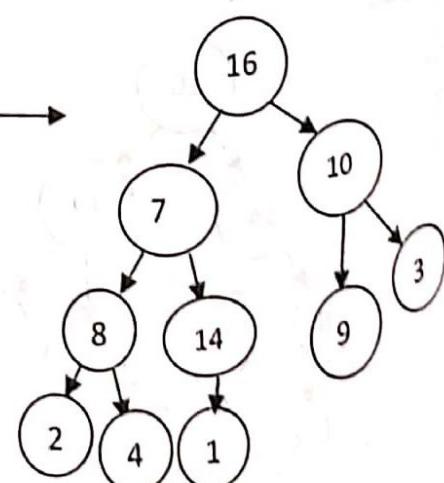
Example: Removing root element (70) from given heap shown below

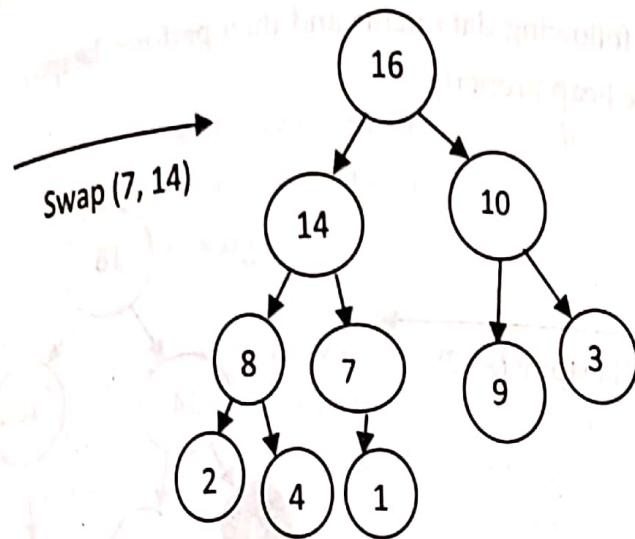


Solution:



Swap (7, 16)





HEAPS AS PRIORITY QUEUES

A stack is a LIFO container: The last one in comes out first. A queue is a "FIFO container": The first one in comes out first. A priority queue is a "BIFO container": The best one in comes out first. That means that each element is assigned a priority number, and the element with the highest priority comes out first. Priority queues are widely used in computer systems. For example, if a printer is shared by several computers on a local area network, the print jobs that are queued to it would normally be held temporarily in a priority queue wherein smaller jobs are given higher priority over larger jobs.

Priority queues are usually implemented as heaps since the heap data structure always keeps the element with the largest key at the root and its insertion and removal operations are so efficient.

Operations on Heaps

1. Maintain/Restore the max-heap property
 - MAX-HEAPIFY
2. Create a max-heap from an unordered array
 - BUILD-MAX-HEAP
3. Sort an array in place
 - HEAPSORT

Heapify Property

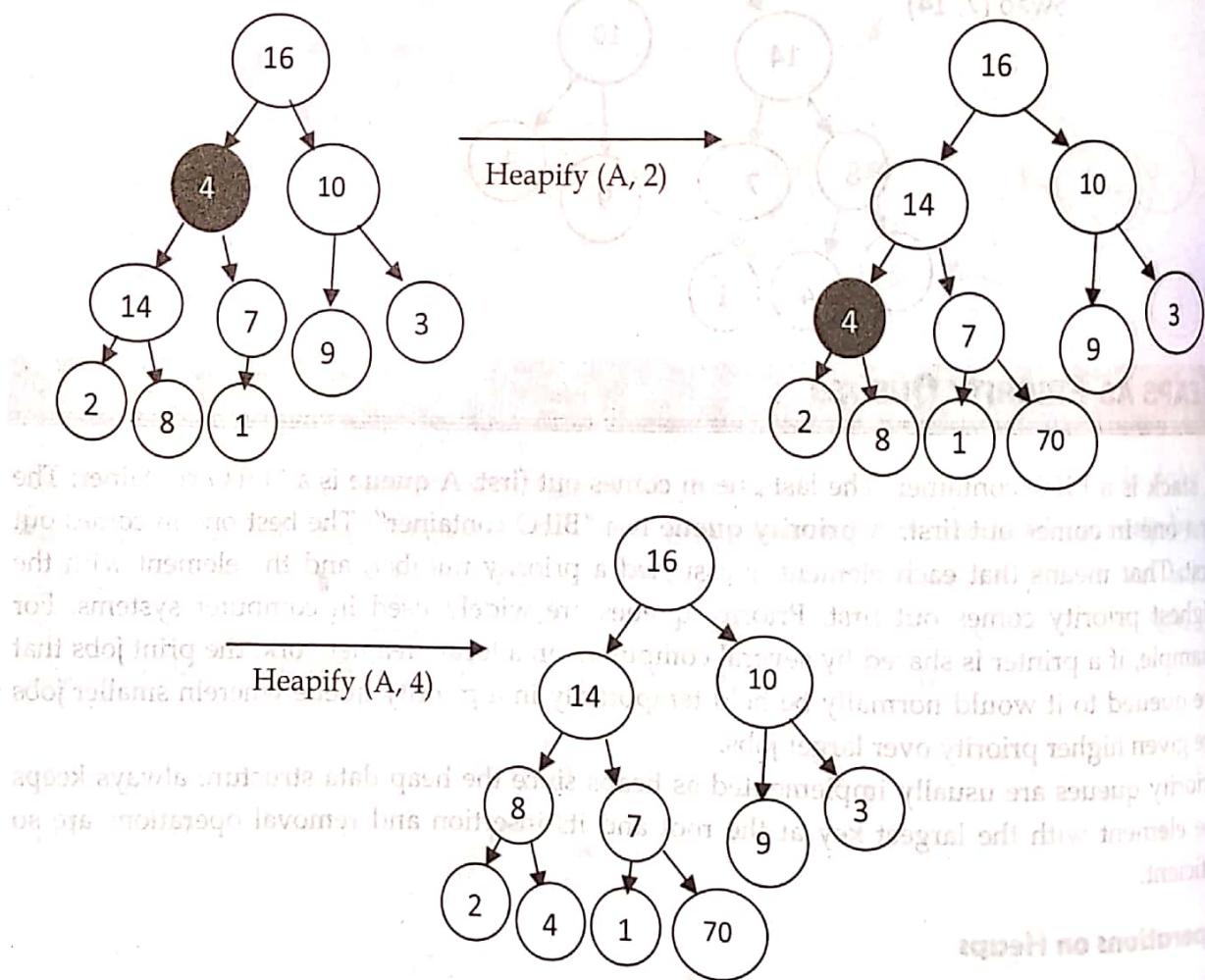
If any node violates the heap property then swap this node with its larger children to maintain the heap property, this operation is called heapify.

MAX-HEAPIFY operation

- Find location of largest value of:
 $A[i]$, $A[Left(i)]$ and $A[Right(i)]$
- If not $A[i]$, max-heap property does not hold.
- Exchange $A[i]$ with the larger of the two children to preserve max-heap property.
- Continue this process of compare/exchange down the heap until sub-tree rooted at i is a max-heap.
- At a leaf, the sub-tree rooted at the leaf is trivially a max-heap.

Example: construct binary tree of following data items and then perform heapify operation on the particular nodes that violates the heap property.

$A[] = \{16, 4, 10, 14, 7, 9, 3, 2, 8, 1\}$



Algorithm

Max-Heapify(A, i, n)

{

$i = \text{Left}(i)$

$r = \text{Right}(i)$

 largest = i

 if $i \leq n$ and $A[i] > A[\text{largest}]$

 largest = i

 if $r \leq n$ and $A[r] > A[\text{largest}]$

 largest = r

 if largest $\neq i$

 exchange ($A[i], A[\text{largest}]$)

 Max-Heapify($A, \text{largest}, n$)

}

Analysis

In the worst case Max-Heapify is called recursively h times, where ' h ' is height of the heap and since each call to the heapify takes constant time Time complexity = $O(h) = O(\log n)$

Building a Heap

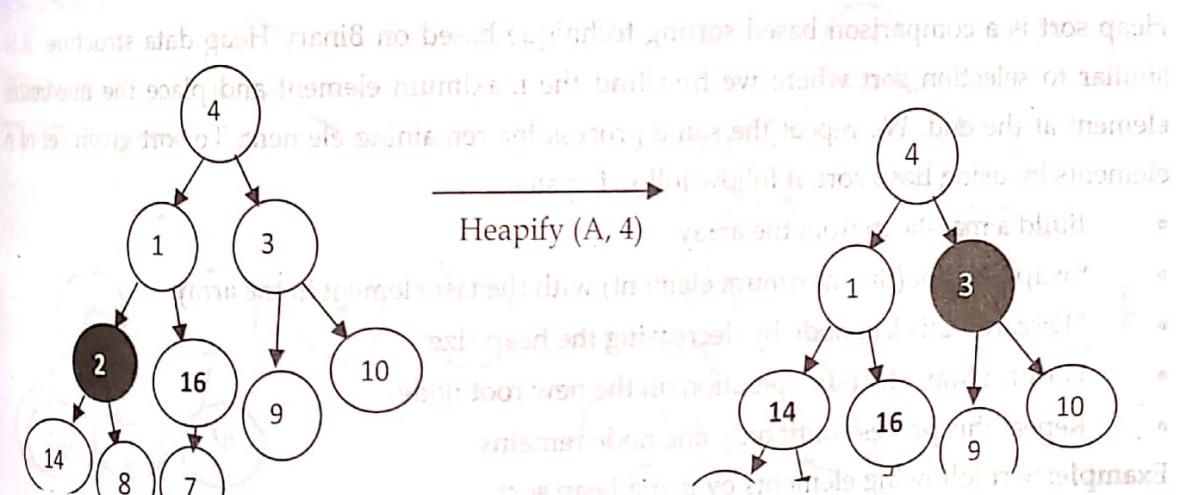
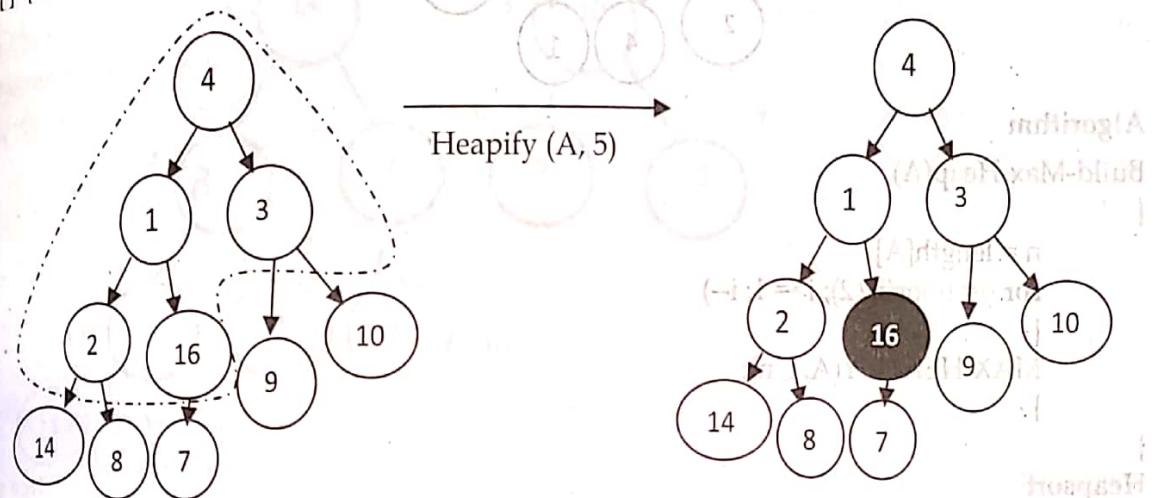
To build a max-heap from any tree, we can thus start heapifying each sub-tree from the bottom up and end up with a max-heap after the function is applied on all the elements including the root element. In the case of complete tree, the first index of non-leaf node is given by $n/2 - 1$. All other nodes after that are leaf-nodes and thus don't need to be heapified.

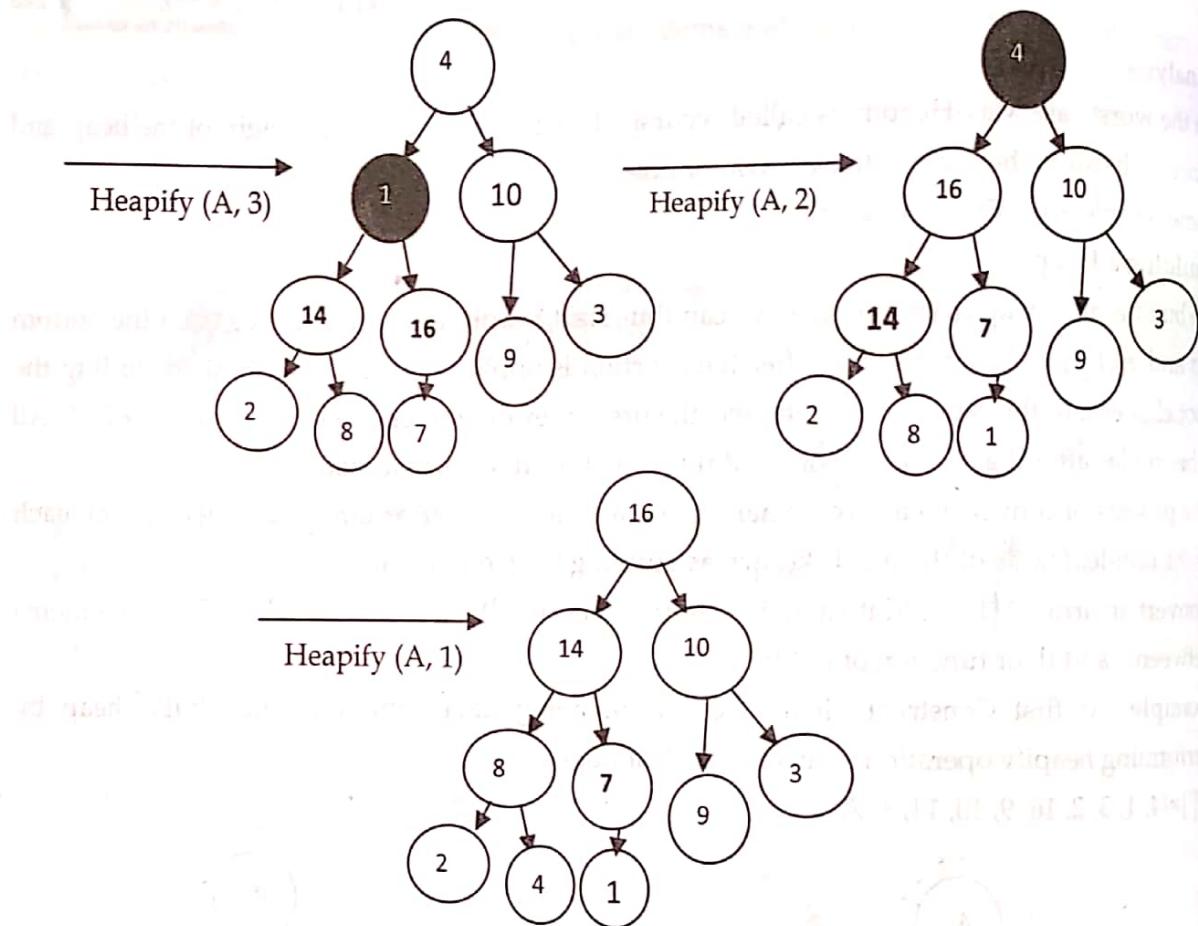
The process of converting a given binary tree into a heap by performing heap operation on each of the non-leaf node of the tree is known as building heap operation.

Convert an array $A[1 \dots n]$ into a max-heap ($n = \text{length}[A]$). Apply MAX-HEAPIFY on elements between 1 and floor function of $(n/2)$.

Example: At first Construct binary tree of following data items and then build heap by performing heapify operation on every non-leaf nodes.

$$A[] = \{4, 1, 3, 2, 16, 9, 10, 14, 8, 7\}$$





Algorithm

Build-Max-Heap(A)

```
{
    n = length[A]
    for (i = floor(n/2); i >= 1; i--)
    {
        MAX-HEAPIFY(A, i, n);
    }
}
```

Heapsort

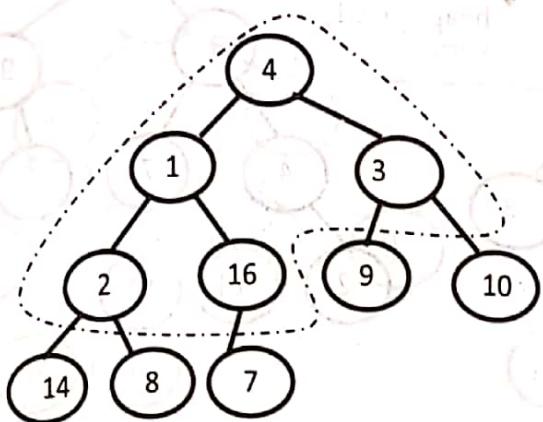
Heap sort is a comparison based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for remaining element. To sort given set of n elements by using heap sort, it follows following steps

- Build a max-heap from the array
- Swap the root (the maximum element) with the last element in the array
- “Discard” this last node by decreasing the heap size
- Perform Max-Heapify operation on the new root node
- Repeat this process until only one node remains

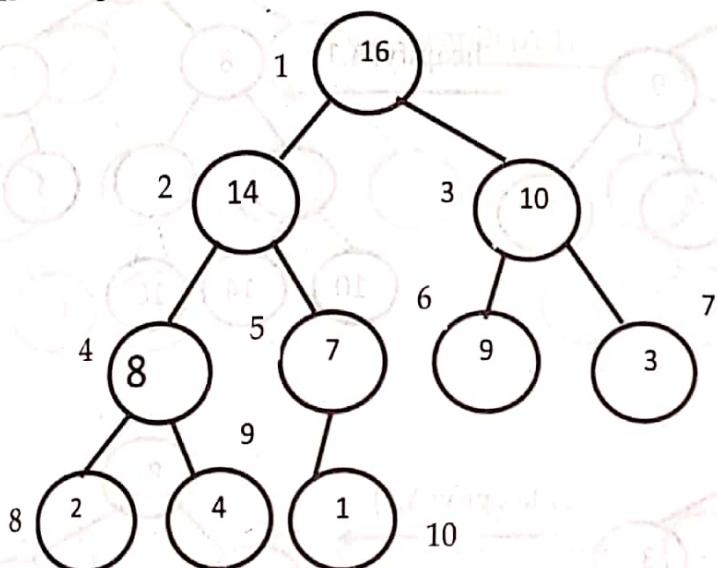
Example: Sort following elements by using heap sort

$A[] = \{4, 1, 3, 2, 16, 9, 10, 14, 8, 7\}$

Solution: At first construct a binary tree of given array,

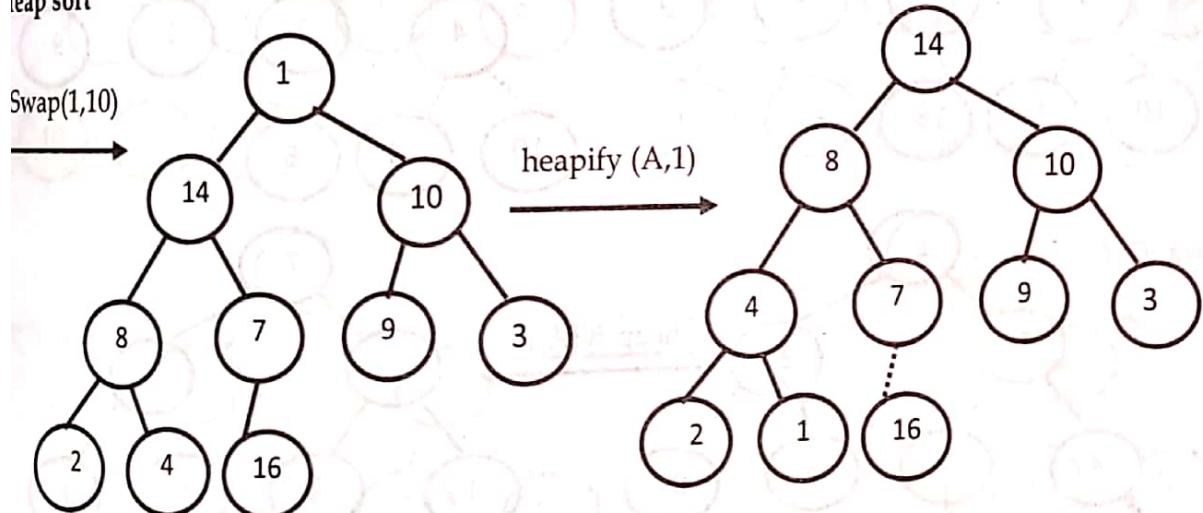


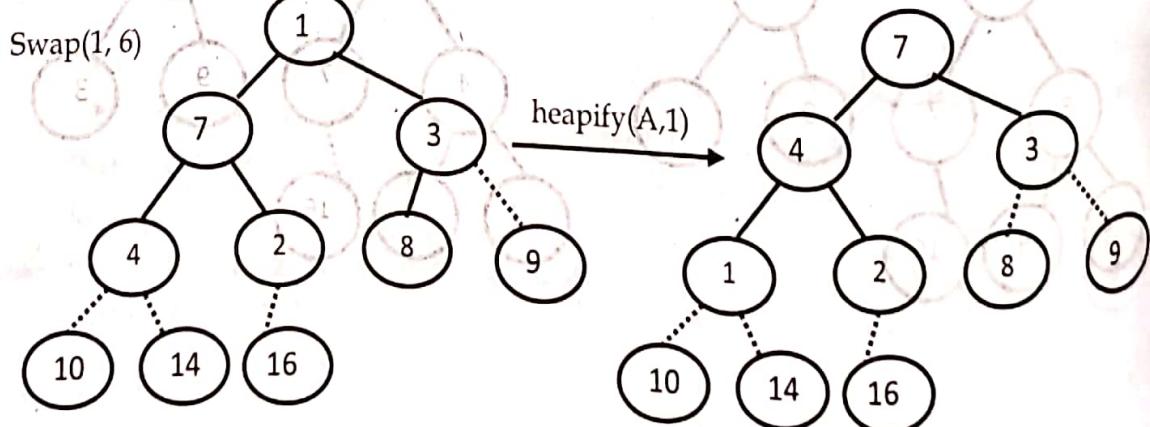
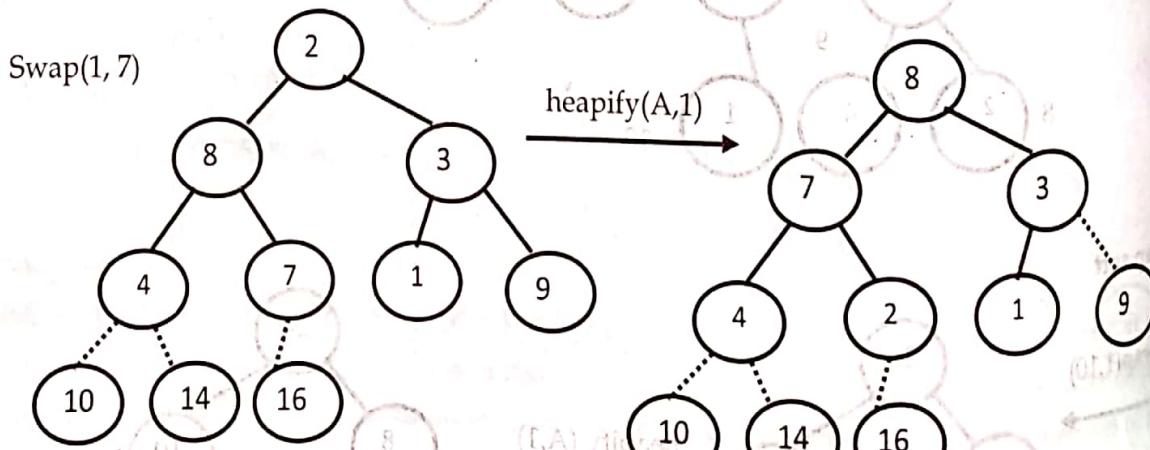
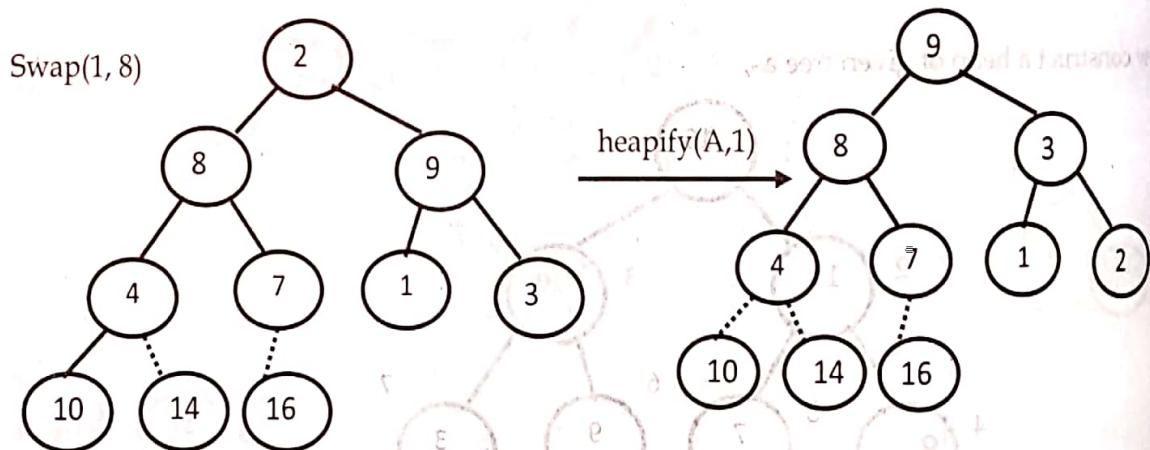
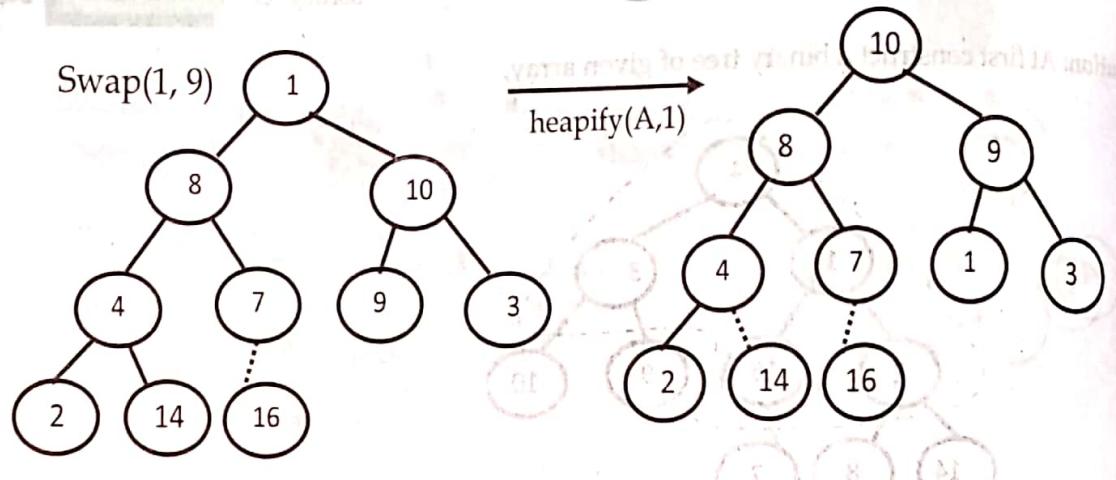
Now construct a heap of given tree as,

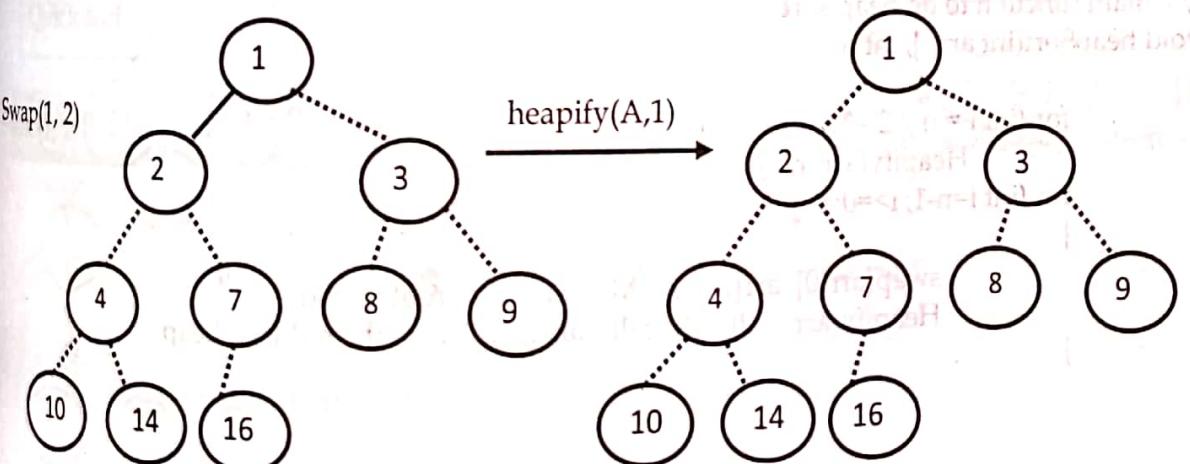
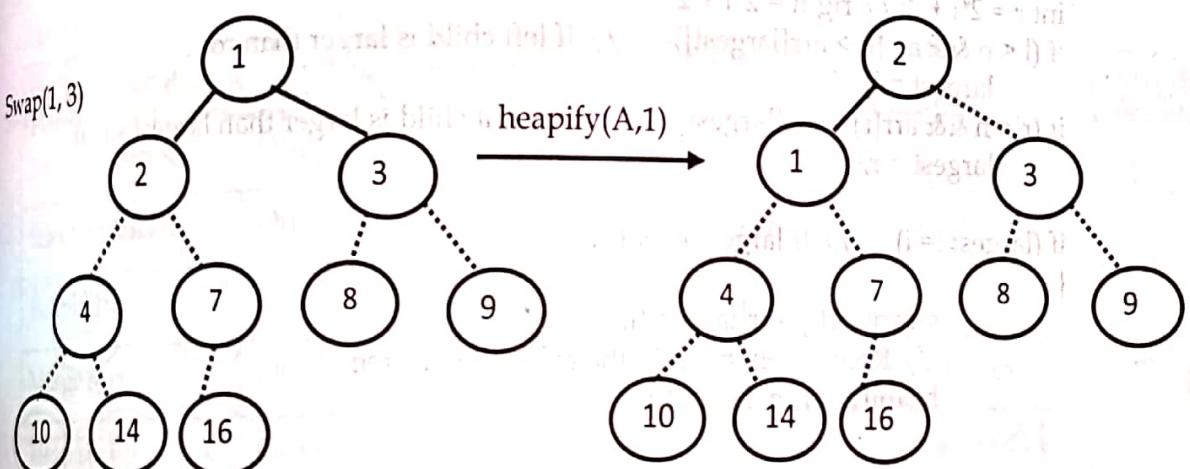
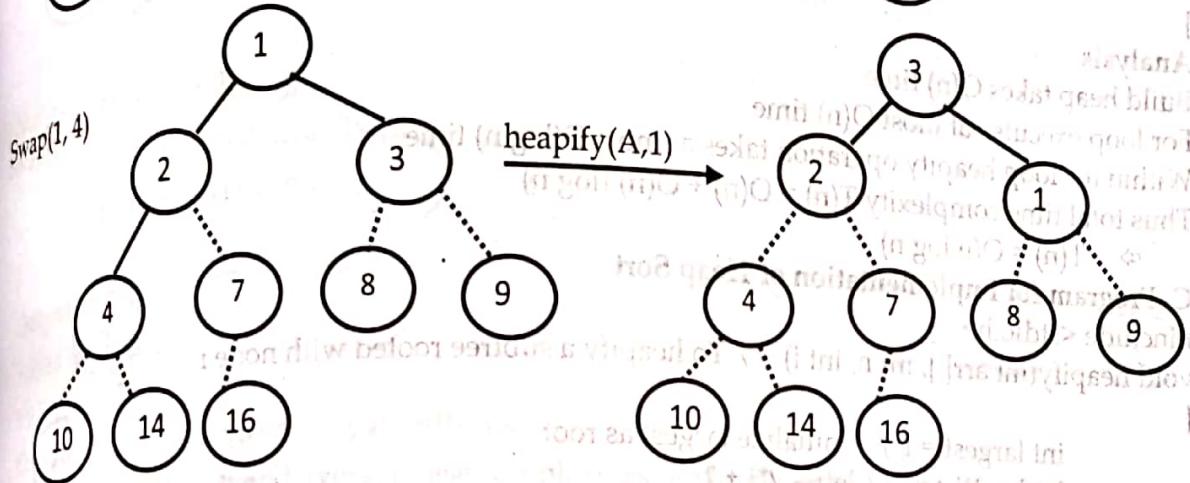
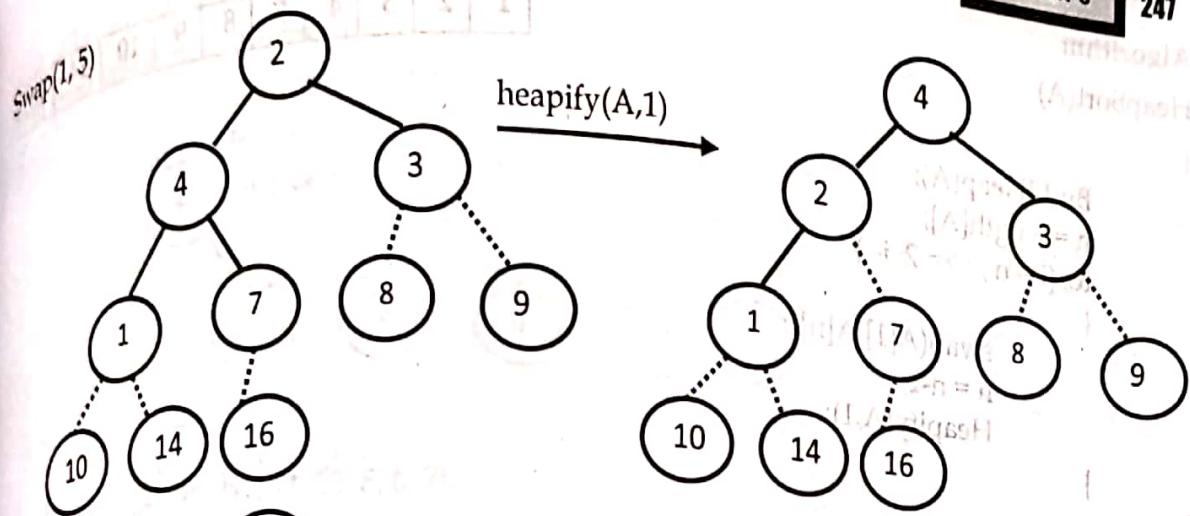


Heap sort

Swap(1,10)







Algorithm

HeapSort(A)

{

```

    BuildHeap(A);
    n = length[A];
    for(i = n ; i >= 2; i--)
    {
        swap(A[1],A[n]);
        n = n-1;
        Heapify(A,1);
    }
}

```

AnalysisBuild heap takes $O(n)$ timeFor loop executes at most $O(n)$ timeWithin for loop heapify operation takes at most $O(\log n)$ timeThus total time complexity $T(n) = O(n) + O(n)(\log n)$

$$\Rightarrow T(n) = O(n \log n)$$

C- Program for implementation of Heap Sort

#include <stdio.h>

void heapify(int arr[], int n, int i) // To heapify a subtree rooted with node i

```

{
    int largest = i; // Initialize largest as root
    int l = 2*i + 1; // left = 2*i + 1
    int r = 2*i + 2; // right = 2*i + 2
    if (l < n && arr[l] > arr[largest]) // If left child is larger than root
        largest = l;
    if (r < n && arr[r] > arr[largest]) // If right child is larger than largest so far
        largest = r;

    if (largest != i) // If largest is not root
    {
        swap(arr[i], arr[largest]);
        // Recursively heapify the affected sub-tree
        heapify(arr, n, largest);
    }
}

```

// main function to do heap sort

void heapSort(int arr[], int n)

{

```

        for (int i = n / 2 - 1; i >= 0; i--)
            Heapify(arr, n, i);
        for (int i=n-1; i>=0; i--)
    {

```

```

            swap(arr[0], arr[i]); // Move current root to end
            Heapify(arr, i, 0); // call max heapify on the reduced heap
    }
}

```

1	2	3	4	7	8	9	10	14	16
---	---	---	---	---	---	---	----	----	----

* A utility function to print array of size n */

```
void printArray(int arr[], int n)
```

```
    for (int i=0; i<n; ++i)
        cout << arr[i] << " ";
    cout << "\n";
```

```
int main()
```

```
    int arr[] = {12, 11, 13, 5, 6, 7};
```

```
    int n = 6;
```

```
    HeapSort(arr, n);
```

```
    printf("Sorted array is:\n");
```

```
    printArray(arr, n);
```

```
}
```

Output

Sorted array is

5 6 7 11 12 13

Comparison of various sorting algorithms

In brief the worst, best and average case complexities of various sorting algorithms are tabulated below;

Sorting technique	Worst case	Average case	Best case	Comment
Insertion sort	$O(n^2)$	$O(n^2)$	$O(n)$	
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	Unstable
Bubble sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Require extra memory
Heap sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Large constant
Quick sort	$O(n^2)$	$O(n \log n)$	$O(n \log n)$	Small constant

MULTIPLE CHOICE QUESTIONS

- What is an external sorting algorithm?
 - Algorithm that uses tape or disk during the sort
 - Algorithm that uses main memory during the sort
 - Algorithm that involves swapping
 - Algorithm that are considered 'in place'

2. What is an internal sorting algorithm?
 - a) Algorithm that uses tape or disk during the sort
 - b) Algorithm that uses main memory during the sort
 - c) Algorithm that involves swapping
 - d) Algorithm that are considered 'in place'
3. What is the worst case complexity of bubble sort?
 - a) $O(n\log n)$
 - b) $O(\log n)$
 - c) $O(n)$
 - d) $O(n^2)$
4. What is the average case complexity of bubble sort?
 - a) $O(n\log n)$
 - b) $O(\log n)$
 - c) $O(n)$
 - d) $O(n^2)$
5. What is the advantage of bubble sort over other sorting techniques?
 - a) It is faster
 - b) Consumes less memory
 - c) Detects whether the input is already sorted
 - d) All of the mentioned
6. Merge sort uses which of the following technique to implement sorting?
 - a) backtracking
 - b) greedy algorithm
 - c) divide and conquer
 - d) dynamic programming
7. Which of the following method is used for sorting in merge sort?
 - a) merging
 - b) partitioning
 - c) selection
 - d) exchanging
8. Which of the following sorting algorithms is the fastest?
 - a) Merge sort
 - b) Quick sort
 - c) Insertion sort
 - d) Shell sort
9. Which of the following methods is the most effective for picking the pivot element?
 - a) first element
 - b) last element
 - c) median-of-three partitioning
 - d) random element
10. Which of the following sorting algorithms is used along with quick sort to sort the sub arrays?
 - a) Merge sort
 - b) Shell sort
 - c) Insertion sort
 - d) Bubble sort



DISCUSSION EXERCISE

1. Why sorting is important in computer science? Describe any one of the best sorting technique with suitable example.
2. What is stable sort? List out any two stable sorting techniques with example.
3. What is sorting? How it is differ from searching?
4. Which sorting technique gives worst case when elements are in already sorted form?

5. Here is an array of ten integers:

3 8 9 1 7 0 2 6 4

Draw this array after the first iteration of the large loop in a selection sort (sorting from smallest to largest).

6. Here is an array of ten integers:

5 3 8 9 1 7 0 2 6 4

Draw this array after the first iteration of the large loop in an insertion sort (sorting from smallest to largest). This iteration has shifted at least one item in the array!

What is heap? Describe heap sort with suitable example.

7. Here is an array of ten integers:

5 3 8 9 1 7 0 2 6 4

Suppose we partition this array using quick sort's partition function and using 5 for the pivot. Draw the resulting array after the partition finishes.

8. Here is an array of ten integers:

5 3 8 9 1 7 0 2 6 4

Draw this array after the two recursive calls of merge sort are completed, and before the final merge step has occurred.

10. Some sorting methods, like heap sort and array-based quick sort, are not naturally stable. Suggest a way to make any sorting algorithm stable by extending the keys (making them longer and adding extra information).

11. Suppose we have a $O(n)$ time algorithm that finds median of an unsorted array. Now consider a Quick Sort implementation where we first find median using the above algorithm, then use median as pivot. What will be the worst case time complexity of this modified Quick Sort?

12. Given an unsorted array. The array has this property that every element in array is at most k distance from its position in sorted array where k is a positive integer smaller than size of array. Which sorting algorithm can be easily modified for sorting this array and what is the obtainable time complexity?

13. Here is an array of ten integers:

[5 3 8 9 1 7 0 2 6 4]

Draw this array after the FIRST iteration of the large loop in an insertion sort (sorting from smallest to largest). This iteration has shifted at least one item in the array!

14. Describe a case where quick sort will result in quadratic behavior.

15. Here is an array which has just been partitioned by the first step of quick sort:

[3, 0, 2, 4, 5, 8, 7, 6, 9]

Which of these elements could be the pivot?

16. Write two or three clear sentences to describe how a heap sort works.

17. How does a selection sort work for an array? Explain with suitable example.
 18. What is the maximum number of comparisons needed to sort 7 items using radix sort? (assume each item is 4 digit decimal number)
 19. Which of the sorting methods will be the best if number of swapping done, is the only measure of efficiently?
 20. Sort following data items by using a). Heap sort b). Quick sort c). Merge sort.
- a) [4, 5, 6, 88, 1, 22, 43, 56, 53, 2, 11, 57, 9]

9

Search is one of the most important techniques used in computer programming. It is used in many applications such as search engines, file systems, databases, etc. In this chapter, we will learn about different search techniques and their implementation. We will also discuss the time complexity of various search algorithms and how they can be optimized. By the end of this chapter, you will have a good understanding of the basic concepts of search and how they can be applied in real-life situations.

SEARCHING

Searching is a fundamental operation in computer science. It is used in many applications such as search engines, file systems, databases, etc. In this chapter, we will learn about different search techniques and their implementation. We will also discuss the time complexity of various search algorithms and how they can be optimized. By the end of this chapter, you will have a good understanding of the basic concepts of search and how they can be applied in real-life situations.

CHAPTER OUTLINE

After studying this chapter, the reader will be able to understand the

- Introduction to search technique: essential of search, sequential search, binary search, tree search, general search tree, Hashing; Hash function and hash tables, collision resolution techniques, efficiency comparison of different search technique



INTRODUCTION

Searching is an operation or a technique that helps finds the place of a given element or value in the list. Any 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 technique that is being followed in the data structure is listed below:

- Linear Search or Sequential Search
- Binary Search

SEQUENTIAL SEARCH

Linear search algorithm finds given element in a list of elements with $O(n)$ time complexity where n is total number of elements in the list. This search process starts comparing of search element with the first element in the list. If both are matching then results with element found otherwise search element is compared with next element in the list. If both are matched, then the result is "element found". Otherwise, repeat the same with the next element in the list until search element is compared with last element in the list, if that last element also doesn't match, then the result is "Element not found in the list". That means, the search element is compared with element by element in the list.

Algorithm

1. Start
2. Read the search element from the user
3. Compare, the search element with the first element in the list.
4. If both are matching, then display "Given element found!!!" and terminate the function
5. If both are not matching, then compare search element with the next element in the list.
6. Repeat steps 4 and 5 until the search element is compared with the last element in the list.
7. If the last element in the list is also doesn't match, then display "Element not found!!!" and terminate the function
8. Stop

Pseudo code

```
LinearSearch(A, n, key)
```

```
{
    flag=0;
    for(i=0; i<n; i++)
    {
        if(A[i] == key)
            flag=1;
    }
    if(flag==1)
        Print "Search successful"
    else
        Print "Search un successful"
```

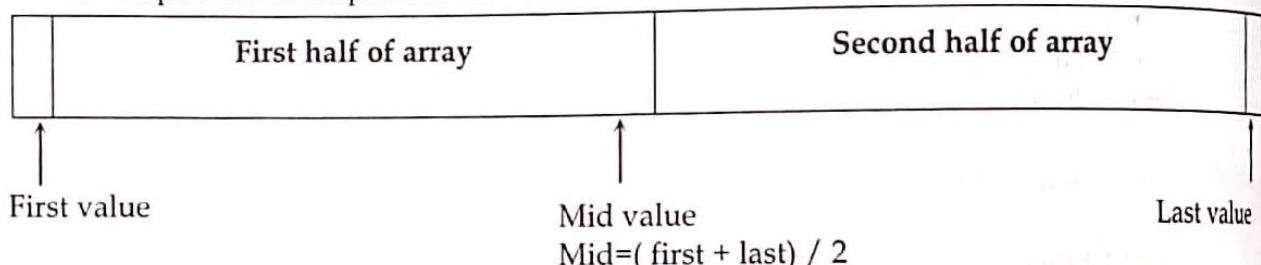
AnalysisTime complexity = $O(n)$ **Complete C program for Sequential search**

```
#include <stdio.h>
int main()
{
    int a[100], key, i, n;
    printf("Enter the number of elements in array\n");
    scanf("%d", &n);
    printf("Enter %d numbers \n", n);
    for (i = 0; i < n; i++)
    {
        scanf("%d", &a[i]);
    }
    printf("Enter the number to search\n");
    scanf("%d", &key);
    for (i = 0; i < n; i++)
    {
        if (a[i] == key)
        {
            printf("%d is present at location %d.\n", key, i+1);
            break;
        }
    }
    if (i == n)
        printf("%d is not present in array.\n", key);
    return 0;
}
```

BINARY SEARCH

Binary search algorithm finds given element in a list of elements with $O(\log n)$ time complexity where n is total number of elements in the list. The binary search algorithm can be used with only sorted list of element. That means, binary search can be used only with list of element which are already arranged in an order. The binary search cannot be used for list of element which is in random order. This search process starts comparing of the search element with the middle element in the list. If both are matched, then the result is "element found". Otherwise, we check whether the search element is smaller or larger than the middle element in the list. If the search element is smaller, then we repeat the same process for left sub list of the middle element. If the search element is larger, then we repeat the same process for right sub list of the middle element. We repeat this process until we find the search element in the list or until we left with a sub list of only one element. And if that element also doesn't match with the search element, then the result is "Element not found in the list". The logic behind this technique is

1. First find the middle element of the array
2. Compare the middle element with an item.
3. There are three cases:
 - a. If it is a desired element then search is successful
 - b. If it is less than desired item then search only the first half of the array.
 - c. If it is greater than the desired element, search in the second half of the array.
4. Repeat the same process until element is found or exhausts in the search area.



Tracing

Search for key= 6 in {-1, 5, 6, 18, 19, 25, 46, 78, 102, 114}

Initially

-1	5	6	18	19	25	46	78	102	114
0	1	2	3	4	5	6	7	8	9

Step 1: Since [middle element (19) > key (6)]

-1	5	6	18	19	25	46	78	102	114
	1	2	3	4	5	6	7	8	9

Step 2: Since [middle element (5) < key (6)]

-1	5	6	18	19	25	46	78	102	114
			3	4	5	6	7	8	9

Step 1: Since [middle element (6) > key (6)]

-1	5	6	18	19	25	46	78	102	114
0	1	2	3	4	5	6	7	8	9

Algorithm

1. Start
2. Read the search element from the user
3. Find the middle element in the sorted list
4. Compare, the search element with the middle element in the sorted list.
5. If both are matching, then display "Given element found!!!" and terminate the function
6. If both are not matching, then check whether the search element is smaller or larger than middle element.
7. If the search element is smaller than middle element, then repeat steps 2, 3, 4 and 5 for the left sub list of the middle element.

8. If the search element is larger than middle element, then repeat steps 2, 3, 4 and 5 for the right sub list of the middle element.
9. Repeat the same process until we find the search element in the list or until sub list contains only one element.
10. If that element also doesn't match with the search element, then display "Element not found in the list!!!" and terminate the function.
11. Stop

Pseudo code

```
BinarySearch(a, l, r, key)
```

```
{
    int m;
    int flag=0;
    if(l<=r)
    {
        m = (l + r)/2;
        if(key==a[m])
            flag=m;
        else if (key<a[m])
            return BinarySearch(a, l, m-1, key);
        else
            return BinarySearch(a, m+1, r, key);
    }
    else
        return flag;
}
```

Efficiency

From the above algorithm we can say that the running time of the algorithm is

$$T(n) = T(n/2) + O(1)$$

By solving this we get $O(\log n)$

In the best case output is obtained at one run i.e. $O(1)$ time if the key is at middle.

In the worst case the output is at the end of the array so running time is $O(\log n)$ time.

In the average case also running time is $O(\log n)$.

Complete C program for Binary search

```
#include<stdio.h>
#include<conio.h>
int BinarySearch(int a[100], int l, int r, int key)
{
    int m;
    int flag=0;
    if(l<=r)
```

```

m = (l + r) / 2;
if(key == a[m])
    flag = m;
else if (key < a[m])
    return BinarySearch(a, l, m-1, key);
else
    return BinarySearch(a, m+1, r, key);
}
else
return flag;
}

void main()
{
    int n, a[100], i, key, flag;
    printf("Enter no of data items:\n");
    scanf("%d", &n);
    printf("Enter %d data items in sorted form:\n", n);
    for(i=0; i<n; i++)
        scanf("%d", &a[i]);
    printf("Enter searched item");
    scanf("%d", &key);
    flag = BinarySearch(a, 0, n-1, key);
    if(flag == 0)
        printf("Search Un-Successful");
    else
        printf("Search successful and found at location %d", flag+1);
}

```

TREE SEARCH

A tree structure is a hierarchy of linked nodes where each node represents a particular state. Nodes have none, one or more child nodes. A solution is a path from the "root" node (representing the initial state) to a "goal" node (representing the desired state). Tree search algorithms attempt to find a solution by traversing the tree structure - starting at the root node and examining (expanding) the child nodes in a systematic way. Tree search algorithms differ by the order in which nodes are traversed and can be classified into two main groups:

- **Blind search algorithms** (e.g. "Breadth-first" and "Depth-first") use a fixed strategy to methodically traverse the search tree. Blind search is not suitable for complex problems as the large search space (number of different possible states to search) makes them impractical given time and memory constraints.
- **Best-first search algorithms** (e.g. "Greedy" and "A*") use a heuristic function to determine the order in which nodes are traversed, giving preference to states that are judged to be most likely to reach the required goal. Using a "heuristic" search strategy reduces the search space to a more manageable size.

A search strategy is complete if it is guaranteed to find a solution if one exists. A search strategy is optimal if it is guaranteed to find the best solution when several solutions exist.

Comparison of linear search and binary search

S. no	Base of comparison	Linear search	Binary search
1	Time complexity	$O(N)$	$O(\log_2 N)$
2	Best case time	$O(1)$ first element	$O(1)$ center element
3	Prerequisite of an array	No prerequisite	Array must be sorted in order
4	Input data	No need to be sorted	Need to be sorted
5	Access	Sequential	Random
6	Comparison	equality	ordering

HASHING

It is an efficient searching technique in which key is placed in direct accessible address for rapid search. Hashing provides the direct access of records from the file no matter where the record is in the file. Due to which it reduces the unnecessary comparisons. This technique uses a hashing function say h which maps the key with the corresponding key address or location.

Hashing is a different approach to searching which calculates the position of the key in the table based on the value of the key. The value of the key is the only indication of the position. When the key is known, the position in the table can be accessed directly, without making any other preliminary tests, as required in a binary search or when searching a tree. This means that the search time is reduced from $O(n)$, as in a sequential search, or from $O(\log n)$, as in a binary search, to at least $O(1)$; regardless of the number of elements being searched, the run time is always the same. But this is just an ideal, and in real applications, this ideal can only be approximated.

Hashing is a method and useful technique to implement dictionaries. This method is used to perform searching, insertion and deletion at a faster rate. A function called Hash Function is used to compute and return position of the record instead of searching with comparisons. The data is stored in array called as Hash table. The Hash Function maps keys into positions in a hash table. The mapping of keys to indices of a hash table is known as Hash Function. The major requirement of hash function is to map equal keys to equal indices.

Given a key, the algorithm computes an index that suggests where the entry can be found:

$$\text{Index} = f(\text{key}, \text{array_size})$$

The value of index is determined by 2 steps

- $\text{hash} = \text{hash_func(key)}$
- $\text{index} = \text{hash \% array_size}$

HASH FUNCTION

A function that transforms a key into a table index is called a hash function. A hash function is any function that can be used to map data of arbitrary size to data of fixed size. The values returned by a hash function are called hash values, hash codes, hash sums, or simply hashes. The hash function will take any item in the collection and return an integer in the range of slot

names, between 0 and m-1. Assume that we have the set of integer items 54, 26, 93, 17, 77, and 31. Our first hash function, sometimes referred to as the "remainder method," simply takes an item and divides it by the table size, returning the remainder as its hash value $h(item) = item \% 11$. Table below gives all of the hash values for our example items. Note that this remainder method (modulo arithmetic) will typically be present in some form in all hash functions, since the result must be in the range of slot names.

Item	Hash Value
54	10
26	4
93	5
17	6
77	0
31	9

Table: Simple Hash Function Using Remainders

Once the hash values have been computed, we can insert each item into the hash table at the designated position as shown in Figure below.

0	1	2	3	4	5	6	7	8	9	10
77	None	None	None	26	93	17	None	None	31	54

Figure: Hash Table with Six Items

Now when we want to search for an item, we simply use the hash function to compute the slot name for the item and then check the hash table to see if it is present. This searching operation is $O(1)$, since a constant amount of time is required to compute the hash value and then index the hash table at that location. If everything is where it should be, we have found a constant time search algorithm.

Types of Hash functions

There are different types of hash functions are used some of them are described below:

Division

A hash function must guarantee that the number it returns is a valid index to one of the table cells. The simplest way to accomplish this is to use division modulo TSize = `sizeof(table)`, as in $h(K) = K \bmod TSize$, if K is a number. It is best if TSize is a prime number; otherwise, $h(K) = (K \bmod p) \bmod TSize$ for some prime $p > TSize$ can be used. However, nonprime divisors may work equally well as prime divisors provided they do not have prime factors less than 20. The division method is usually the preferred choice for the hash function if very little is known about the keys.

Folding
The folding method for constructing hash functions begins by dividing the item into equal-size pieces (the last piece may not be of equal size). These pieces are then added together to give the resulting hash value. For example, if our item was the phone number 436-555-4601, we would take the digits and divide them into groups of 2 (43, 65, 55, 46, 01). After the addition, $43+65+55+46+01$, we get 210. If we assume our hash table has 11 slots, then we need to perform the extra step of dividing by 11 and keeping the remainder. In this case $210 \% 11$ is 1, so the phone number 436-555-4601 hashes to slot 1. Some folding methods go one step further and reverse every other piece before the addition. For the above example, we get $43+56+55+64+01=219$ which gives $219 \% 11=10$.

Mid-Square Function

In the mid-square method, the key is squared and the middle or mid part of the result is used as the address. If the key is a string, it has to be preprocessed to produce a number by using, for instance, folding. In a mid-square hash function, the entire key participates in generating the address so that there is a better chance that different addresses are generated for different keys. For example, if the key is 3,121 then $(3,121)^2 = 9,740,641$, and for the 1,000-cell table, $h(3,121) = 406$, which is the middle part of $(3,121)^2$. In practice, it is more efficient to choose a power of 2 for the size of the table and extract the middle part of the bit representation of the square of a key. If we assume that the size of the table is 1,024, then, in this example, the binary representation of $(3,121)^2$ is the bit string 100101001010000101100001, with the middle part shown in italics. This middle part, the binary number 0101000010, is equal to 322. This part can easily be extracted by using a mask and a shift operation.

Extraction

In the extraction method, only a part of the key is used to compute the address. For the social security number 123-45-6789, this method might use the first four digits, 1,234; the last four, 6,789; the first two combined with the last two, 1,289; or some other combination. Each time, only a portion of the key is used. If this portion is carefully chosen, it can be sufficient for hashing, provided the omitted portion distinguishes the keys only in an insignificant way. For example, in some university settings, all international students' ID numbers start with 999. Therefore, the first three digits can be safely omitted in a hash function that uses student IDs for computing table positions. Similarly, the starting digits of the ISBN code are the same for all books published by the same publisher (e.g., 0534 for Brooks/Cole Publishing Company). Therefore, they should be excluded from the computation of addresses if a data table contains only books from one publisher.

HASH COLLISION

In computer science, a **collision** or **clash** is a situation that occurs when two distinct pieces of data have the same hash value, checksum, fingerprint, or cryptographic digest. Collisions are unavoidable whenever members of a very large set (such as all possible person names, or all possible computer files) are mapped to a relatively short bit string. This is merely an instance of the pigeonhole principle. The impact of collisions depends on the application. When hash functions and fingerprints are used to identify similar data, such as homologous DNA sequences or similar audio files, the functions are designed so as to maximize the probability of collision between distinct but similar data.

Collision Resolution

When two items hash to the same slot, we must have a systematic method for placing the second item in the hash table. This process is called collision resolution. If the hash function is perfect, collisions will never occur. However, since this is often not possible, collision resolution becomes a very important part of hashing. Some popular methods for minimizing collision are:

- Open addressing
 - ✓ Linear probing

- ✓ Quadratic probing
- ✓ Double hashing
- Rehashing
- Chaining
- Hashing using buckets etc

Open Addressing

In open addressing, when a data item can't be placed at the index calculated by the hash function, another location in the array is sought. We'll explore three methods of open addressing, which vary in the method used to find the next vacant cell. These methods are **linear probing**, **quadratic probing**, and **double hashing**.

Linear probing

A hash table in which a collision is resolved by putting the item in the next empty place within the occupied array space is called linear probing. It starts with a location where the collision occurred and does a sequential search through a hash table for the desired empty location. Hence this method searches in straight line, and it is therefore called linear probing. The main disadvantage of this process is clustering problem.

Example: Insert keys {89, 18, 49, 58, 69} with the hash function $h(x) = x \bmod 10$ using linear probing.

Solution: when $x=89$

$$h(89)=89\%10=9$$

Insert key 89 in hash-table in location 9

When $x=18$

$$h(18)=18\%10=8$$

Insert key 18 in hash-table in location 8

When $x=49$

$$h(49)=49\%10=9 \text{ (Collision occur)}$$

So insert key 49 in hash-table in next possible vacant location of 9 is 0

When $x=58$

$$h(58)=58\%10=8 \text{ (Collision occur)}$$

Insert key 58 in hash-table in next possible vacant location of 8 is 1
(since 9, 0 already contains values).

When $x=69$

$$h(69)=69\%10=9 \text{ (Collision occur)}$$

Insert key 69 in hash-table in next possible vacant location of 9 is 2 (since 0, 1 already contains values).

0	1	2	3	4	5	6	7	8	9
49	58	69	None	None	None	None	None	18	89

Complete C program for linear probing

```

#include<stdio.h>
#include<stdlib.h>
struct item
{
    int key;
    int value;
};
struct hashtable_item
{
    struct item *data;
    int flag;
    /* flag = 0 : data does not exist
     * flag = 1 : data exists
     * flag = 2 : data existed at least once
    */
};
struct hashtable_item *array;
int size = 0;
int max = 10;
void init_array()
{
    int i;
    for (i = 0; i < max; i++)
    {
        array[i].flag = 0;
        array[i].data = NULL;
    }
}
int hashcode(int key)
{
    return (key % max);
    //return (key % max);
}
void insert(int key, int value)
{
    int index = hashcode(key);
    int i = index;
    struct item *new_item = (struct item*) malloc(sizeof(struct item));
    new_item->key = key;
    new_item->value = value;
    while (array[i].flag == 1)
    {
        if (array[i].data->key == key)
        {
            printf("\n Key already exists, hence updating its value \n");
            array[i].data->value = value;
        }
        else
        {
            printf("Index %d (%d) is empty\n", i, index);
            array[i].data = new_item;
            array[i].flag = 2;
        }
        i = (i + 1) % max;
    }
}

```

```

        return;
    }
    i = (i + 1) % max;
    if (i == index)
    {
        printf("\n Hash table is full, cannot insert any more item \n");
        return;
    }
    array[i].flag = 1;
    array[i].data = new_item;
    size++;
    printf("\n Key (%d) has been inserted \n", key);
}
/* to remove an element from the hash table */
void remove_element(int key)
{
    int index = hashcode(key);
    int i = index;
    while (array[i].flag != 0)
    {
        if (array[i].flag == 1 && array[i].data->key == key)
        {
            array[i].flag = 2;
            array[i].data = NULL;
            size--;
            printf("\n Key (%d) has been removed \n", key);
            return;
        }
        i = (i + 1) % max;
        if (i == index)
        {
            break;
        }
    }
    printf("\n This key does not exist \n");
}

/* to display all the elements of hash table */
void display()
{
    int i;
    for (i = 0; i < max; i++)
    {
        struct item *current = (struct item*) array[i].data;
        if (current == NULL)
        {
            printf("\n Array[%d] has no elements \n", i);
        }
    }
}

```

```

    }
    else
    {
        printf("\n Array[%d] has elements:-\n %d(key) and %d(value)", i, current→key,
               current→value);
    }
}

int size_of_hashtable()
{
    return size;
}

void main()
{
    int choice, key, value, n, c;
    clrscr();
    array = (struct hashtable_item*) malloc(max * sizeof(struct hashtable_item*));
    init_array();
    do {
        printf("Implementation of Hash Table in C with Linear Probing \n\n");
        printf(" MENU-: \n1.Inserting item in the Hash table"
               "\n2.Removing item from the Hash table"
               "\n3.Check the size of Hash table"
               "\n4.Display Hash table"
               "\n\n Please enter your choice-:");
        scanf("%d", &choice);
        switch(choice)
        {
            case 1:
                printf("Inserting element in Hash table\n");
                printf("Enter key and value-:\t");
                scanf("%d %d", &key, &value);
                insert(key, value);
                break;
            case 2:
                printf("Deleting in Hash table \n Enter the key to delete-:");
                scanf("%d", &key);
                remove_element(key);
                break;
            case 3:
                n = size_of_hashtable();
                printf("Size of Hashtable is:-%d\n", n);
                break;
            case 4:
                display();
                break;
            default:
                printf("Wrong Input\n");
        }
    }
}

```

```

    }
    printf("\n Do you want to continue:-(press 1 for yes)\t");
    scanf("%d", &c);
}while(c == 1);
getch();
}

```

Quadratic Probing

Quadratic probing is a collision resolution method that eliminates the primary clustering problem take place in a linear probing. When collision occur then the quadratic probing works as follows:

$$(\text{Hash value} + 1^2) \% \text{table size}$$

If there is again collision occurs then there exist rehashing.

$$(\text{Hash value} + 2^2) \% \text{table size}$$

If there is again collision occurs then there exist rehashing.

$$(\text{Hash value} = 3^2) \% \text{table size}$$

In general in i^{th} collision

$$h_i(x) = (\text{hash value} + i^2) \% \text{table size}$$

Example: Insert keys {89, 18, 49, 58, and 69} with the hash-table size 10 using quadratic probing.

Solution: when $x=89$

$$h(89)=89 \% 10=9$$

Insert key 89 in hash-table in location 9

When $x=18$

$$h(18)=18 \% 10=8$$

Insert key 18 in hash-table in location 8

When $x=49$

$$h(49)=49 \% 10=9 \quad (\text{Collision occur})$$

So use following hash function,

$$h1(49)=(49+1) \% 10=0$$

Hence insert key 49 in hash-table in location 0

When $x=58$

$$h(58)=58 \% 10=8 \quad (\text{Collision occur})$$

So use following hash function,

$$h1(58)=(58+1) \% 10=9$$

Again collision occur use again the following hash function,

$$h2(58)=(58+22) \% 10=2$$

Insert key 58 in hash-table in location 2

When $x=69$

$$h(69)=69 \% 10=9 \quad (\text{Collision occur})$$

So use following hash function,

$$h1(69)=(69+1) \% 10=0$$

Again collision occur use again the following hash function,

$$h_2(69) = (69 + 22) \% 10 = 3$$

Insert key 69 in hash-table in location 3

0	1	2	3	4	5	6	7	8	9
49	None	58	69	None	None	None	None	18	89

DOUBLE HASHING

Double hashing is a collision resolution method that eliminates the primary clustering problem take place in a linear probing. When collision occur then the double hashing define new hash function as follows:

$$h_2(x) = R - (x \bmod R)$$

Where, R is a prime number smaller than hash table size.

The new element can be inserted in the position by using following function,

$$\text{Position} = (\text{original hash value} + i \cdot h_2(x)) \% \text{Table size}$$

Example: Insert keys {89, 18, 49, 58, and 69} with the hash-table size 10 using double hashing.

Solution: when $x=89$

$$h(89) = 89 \% 10 = 9$$

Insert key 89 in hash-table in location 9

When $x=18$

$$h(18) = 18 \% 10 = 8$$

Insert key 18 in hash-table in location 8

When $x=49$

$$h(49) = 49 \% 10 = 9 \text{ (collision occur)}$$

Now define new hash function as,

$$h_2(x) = R - (x \bmod R)$$

$$\text{Or, } h_2(x) = 7 - 49 \% 7 = 7 - 0 = 7 \text{ and}$$

$$\text{Position} = (\text{original hash value} + i \cdot h_2(x)) \% \text{Table size}$$

Insert key 49 in hash-table in location 6

When $x=58$ Insert key 49 in hash-table in location 6

$$h(58) = 58 \% 10 = 8 \text{ (collision occur)}$$

Now define new hash function as,

$$h_2(x) = R - (x \bmod R)$$

$$\text{Or, } h_2(x) = 7 - 58 \% 7 = 7 - 2 = 5 \text{ and}$$

$$\text{Position} = (\text{original hash value} + i \cdot h_2(x)) \% \text{Table size}$$

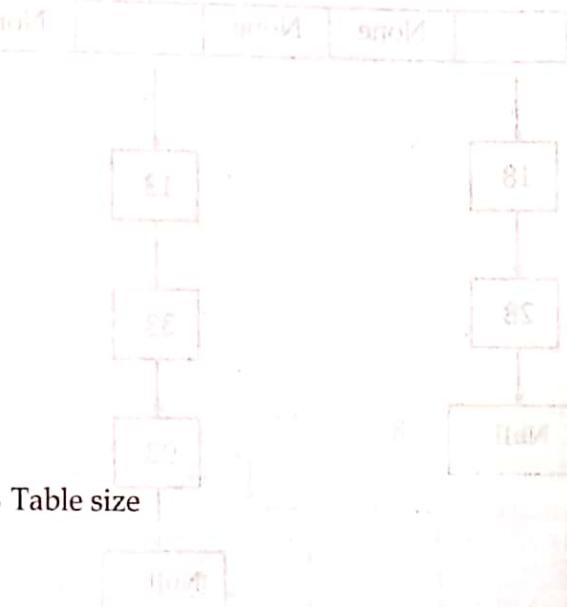
$$= (58 + 1 \cdot 5) \% 10 = 63 \% 10 = 3$$

Insert key 58 in hash-table in location 3

When $x=69$

$$h(69) = 69 \% 10 = 9 \text{ (collision occur)}$$

Now define new hash function as,



$$h_2(x) = R - (x \bmod R)$$

$$\text{Or, } h_2(x) = 7 - 69\%7 = 7-6 = 1 \text{ and}$$

$$\begin{aligned} \text{Position} &= (\text{original hash value} + i \cdot h_2(x)) \% \text{Table size} \\ &= (69 + 1 \cdot 1) \% 10 = 70 \% 10 = 0 \end{aligned}$$

Insert key 69 in hash-table in location 0

0	1	2	3	4	5	6	7	8	9
69	None	None	58	None	None	49	None	18	89

Chaining

An alternative method for handling the collision problem is to allow each slot to hold a reference to a collection (or chain) of items. Chaining allows many items to exist at the same location in the hash table. When collisions happen, the item is still placed in the proper slot of the hash table. As more and more items hash to the same location, the difficulty of searching for the item in the collection increases. Figure below shows the items as they are added to a hash table that uses chaining to resolve collisions.

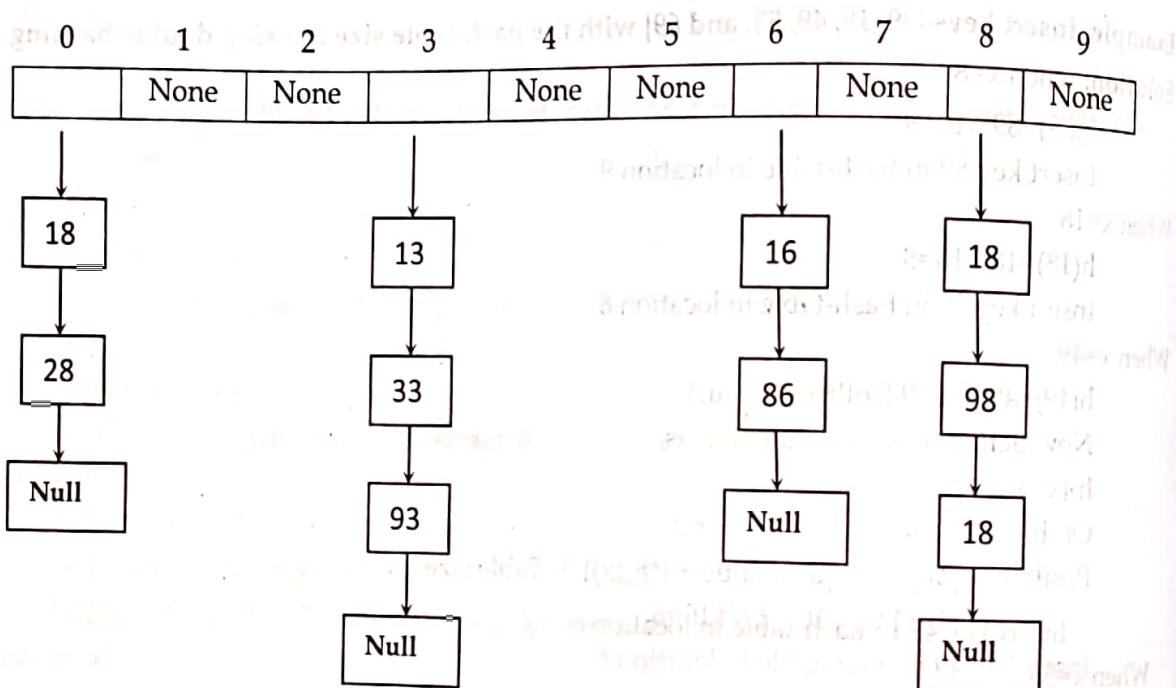


Figure: Collision Resolution with Chaining

When we want to search for an item, we use the hash function to generate the slot where it should reside. Since each slot holds a collection, we use a searching technique to decide whether the item is present. The advantage is that on the average there are likely to be many fewer items in each slot, so the search is perhaps more efficient.

Example: Insert keys {102, 18, 49, 58, 69, 87, 88, 77, 83, and 120} with the hash-table size 10 using chaining method.

Solution: when $x=102$

$$h(102)=102 \% 10=2$$

Insert key 102 in hash-table in location 2

When $x=18$
 $h(18)=18\%10=8$

Insert key 18 in hash-table in location 8

When $x=49$
 $h(49)=49\%10=9$

Insert key 49 in hash-table in location 9

When $x=58$
 $h(58)=58\%10=8$

Insert key 58 in hash-table in location 8

When $x=69$
 $h(69)=69\%10=9$

Insert key 69 in hash-table in location 9

When $x=87$
 $h(87)=87\%10=7$

Insert key 87 in hash-table in location 7

When $x=88$
 $h(88)=88\%10=8$

Insert key 88 in hash-table in location 8

When $x=77$
 $h(77)=77\%10=7$

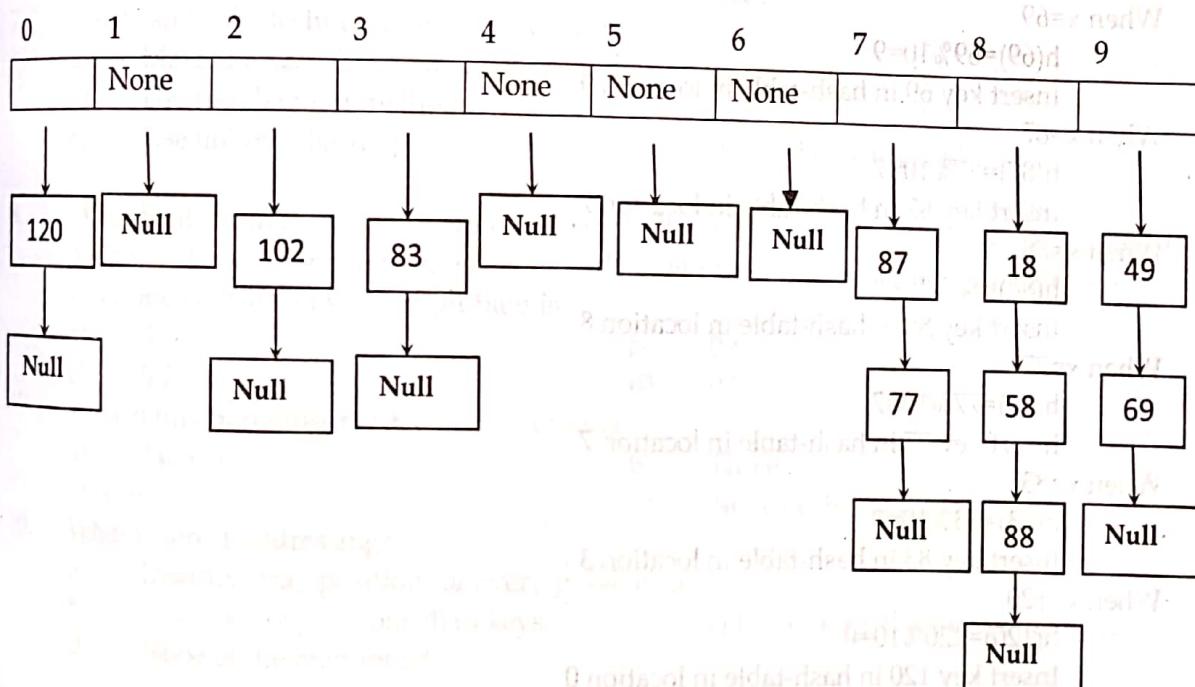
Insert key 77 in hash-table in location 7

When $x=83$
 $h(83)=83\%10=3$

Insert key 83 in hash-table in location 3

When $x=120$
 $h(120)=120\%10=0$

Insert key 120 in hash-table in location 0



Bucket Addressing

Another solution to the collision problem is to store colliding elements in the same position in the table. This can be achieved by associating a bucket with each address. A bucket is a block in space large enough to store multiple items. By using buckets, the problem of collisions is not totally avoided. If a bucket is already full, then an item hashed to it has to be stored somewhere else. By incorporating the open addressing approach, the colliding item can be stored in the next bucket if it has an available slot when using linear probing, or it can be stored in some other bucket when, say, quadratic probing is used.

The colliding items can also be stored in an overflow area. In this case, each bucket includes a field that indicates whether the search should be continued in this area or not. It can be simply a yes/no marker. In conjunction with chaining, this marker can be the number indicating the position in which the beginning of the linked list associated with this bucket can be found in the overflow area.

Example: Insert keys {102, 18, 49, 58, 69, 87, 88, 77, 83, 120} with the hash-table size 10 using bucket hashing.

Solution: when $x=102$

$$h(102)=102\%10=2$$

Insert key 102 in hash-table in location 2

When $x=18$

$$h(18)=18\%10=8$$

Insert key 18 in hash-table in location 8

When $x=49$

$$h(49)=49\%10=9$$

Insert key 49 in hash-table in location 9

When $x=58$

$$h(58)=58\%10=8$$

Insert key 58 in hash-table in location 8

When $x=69$

$$h(69)=69\%10=9$$

Insert key 69 in hash-table in location 9

When $x=87$

$$h(87)=87\%10=7$$

Insert key 87 in hash-table in location 7

When $x=88$

$$h(88)=88\%10=8$$

Insert key 88 in hash-table in location 8

When $x=77$

$$h(77)=77\%10=7$$

Insert key 77 in hash-table in location 7

When $x=83$

$$h(83)=83\%10=3$$

Insert key 83 in hash-table in location 3

When $x=120$

$$h(120)=120\%10=0$$

Insert key 120 in hash-table in location 0

0	1	2	3	4	5	6	7	8	9
120	Null	102	83	Null	Null	Null	87	18	49
Null	77	58	69						
Null	88	Null							

MULTIPLE CHOICE QUESTIONS

- A technique for direct search is
 - Binary Search
 - Linear Search
 - Tree Search
 - Hashing
- The searching technique that takes $O(1)$ time to find a data is
 - Linear Search
 - Binary Search
 - Hashing
 - Tree Search
- The goal of hashing is to produce a search that takes
 - $O(1)$ time
 - $O(n^2)$ time
 - $O(\log n)$ time
 - $O(n \log n)$ time
- Key value pairs is usually seen in
 - Hash tables
 - Heaps
 - Both Hash tables and Heaps
 - Skip list
- What is a hash function?
 - A function has allocated memory to keys
 - A function that computes the location of the key in the array
 - A function that creates an array
 - None of the mentioned
- Which of the following scenarios leads to linear running time for a random search hit in a linear-probing hash table?
 - All keys hash to same index
 - All keys hash to different indices
 - All keys hash to an even-numbered index
 - All keys hash to different even-numbered indices
- What can be the techniques to avoid collision?
 - Make the hash function appear random
 - Use the chaining method
 - Use uniform hashing
 - All of the mentioned
- A hash table can store a maximum of 10 records, currently there are records in location 1, 3, 4, 7, 8, 9, 10. The probability of a new record going into location 2, with hash functions resolving collisions by linear probing is
 - 0.1
 - 0.6
 - 0.2
 - 0.5
- A hash function must meet _____ criteria.
 - Two
 - Three
 - Four
 - None of the mentioned
- What is direct addressing?
 - Distinct array position for every possible key
 - Fewer array positions than keys
 - Fewer keys than array positions
 - None of the mentioned



DISCUSSION EXERCISE

1. What do you mean by hashing? Describe hash function with suitable example.
2. What is the minimum number of keys that are hashed to their home positions using the linear probing technique?
3. Which one of the following choices gives a possible order in which the key values could have been inserted in the table?

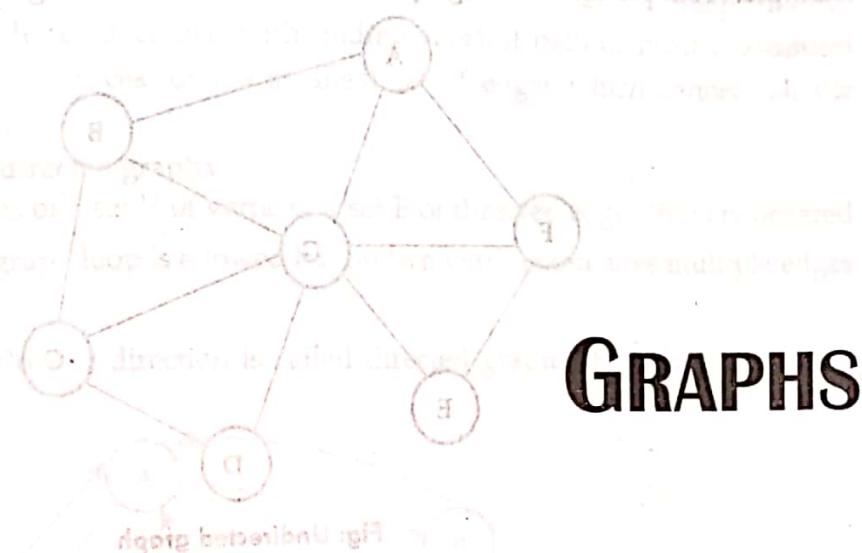
(A) 46, 42, 34, 52, 23, 33	(B) 34, 42, 23, 52, 33, 46
(C) 46, 34, 42, 23, 52, 33	(D) 42, 46, 33, 23, 34, 52
4. Strictly speaking, the hash function used in extendible hashing also dynamically changes. In what sense is this true?
5. Apply the linear hashing method to hash numbers 12, 24, 36, 48, 60, 72, and 84 to an initially empty table with three buckets and with three cells in the overflow area. What problem can you observe? Can this problem bring the algorithm to a halt?
6. Outline an algorithm to delete a key from a table when the linear hashing method is used for inserting keys.
7. Describe double hashing with suitable example.
8. What is the drawback of using linear probing over quadratic probing? Explain.
9. What do you mean by hash collision? Explain hash collision resolving techniques with suitable example.
10. What is main drawback of using double hashing? Explain
11. Explain similarities and dissimilarities between searching techniques and hashing.
12. Here is an array with exactly 15 elements:
[1 2 3 4 5 6 7 8 9 10 11 12 13 14 15]
Suppose that we are doing a binary search for an element. Circle any elements that will be found by examining two or fewer numbers from the array.
13. Draw a hash table with open addressing and a size of 9. Use the hash function " $k \% 9$ ". Insert the keys: 5, 29, 20, 0, 27 and 18 into your table (in that order).
14. Draw a hash table with chaining and a size of 9. Use the hash function " $k \% 9$ " to insert the keys 5, 29, 20, 0, and 18 into your table.
15. Suppose that an open-address hash table has a capacity of 811 and it contains 81 elements. What is the table's load factor?
16. I plan to put 1000 items in a hash table, and I want the average number of accesses in a successful search to be about 2.0.
 - About how big should the array be if I use open addressing with linear probing?
 - About how big should the array be if I use chained hashing?
17. Here is an array with exactly 15 elements:
[1 2 3 4 5 6 7 8 9 10 11 12 13 14 15]
Suppose that we are doing a serial search for an element. Circle any elements that will be found by examining two or fewer numbers from the array.
18. Suppose you place m items in a hash table with an array size of s . What is the correct formula for the load factor?
19. A chained hash table has an array size of 512. What is the maximum number of entries that can be placed in the table?
20. What kind of initialization needs to be done for a chained hash table? Also what is the worst-case time for binary search finding a single item in an array?



INTRODUCTION

10

Graphs If there is a directed edge from vertex V_i to vertex V_j , then we say V_i is adjacent to V_j .



GRAPHS

CHAPTER OUTLINE

After studying this chapter, the reader will be able to understand the

- Introduction, Graph as an ADT, transitive closure, Warshall's Algorithm, Types of graph, graph traversal and spanning forests, Kruskal's and Round robin algorithms, Shortest path algorithm, Greedy algorithm, Dijkstra's Algorithm



INTRODUCTION

Graph is a non linear data structure; it contains a set of points known as nodes (or vertices) and set of links known as edges (or Arcs) which connects the vertices. A graph is an ordered pair $G = (V, E)$ comprising a set V of vertices or nodes together with a set E of edges or links. Each edge has either one or two vertices associated with it, called its end points. Each edge is a pair (v, w) , where $v, w \in V$. Vertices are sometimes called nodes, and edges are sometimes called arcs. If the edge pair is ordered, the graph is called a **directed graph**. Directed graphs are sometimes called digraphs. In a digraph, vertex w is adjacent to vertex v if and only if $(v, w) \in E$. Sometimes an edge has a third component, called the edge cost (or weight) that measures the cost of traversing the edge.

Example: A simple undirected graph with 7 vertices and 11 edges

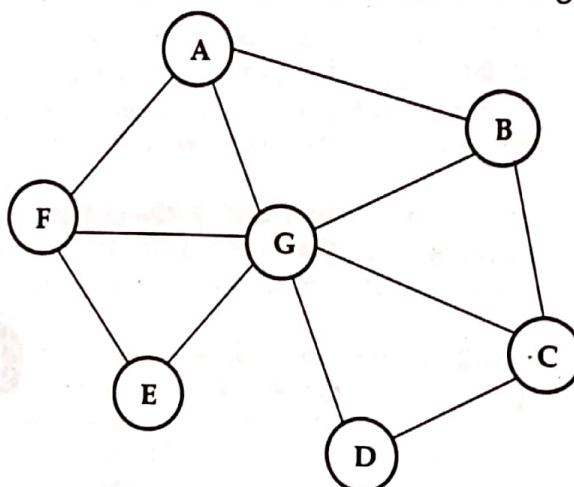
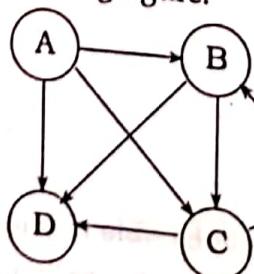


Fig: Undirected graph

DIRECTED AND UNDIRECTED GRAPH

A graph can be directed or undirected. However, in an undirected graph, edges are not associated with the directions with them. An undirected graph is shown in the above figure since its edges are not attached with any of the directions. If an edge exists between vertex A and B then the vertices can be traversed from B to A as well as A to B.

In a directed graph, edges form an ordered pair. Edges represent a specific path from some vertex A to another vertex B. Node A is called initial node while node B is called terminal node. A directed graph is shown in the following figure.



Applications of graph

There are large numbers of applications of graph. We may have seen maps of a country in which airlines routes are shown as lines (arcs) connecting pairs of cities. Each line represents the edge and each city represents the node for the graph. This is a simple famous example of graph in practice.

- Graphs are used to model the geographic maps of cities in which each place in city can be represented by the node and the road connecting such places are represented by an arc (edge).
- Graphs are used to model the computer network in which each node is a machine (computer, hub, router, switch etc.) and the link between them represents the edge.
- They are used to analyze the electrical circuits, project planning, genetics etc.
- So any structured problem can be modeled by graphs. Then can help to solve typical problems those concerned with finding shortest path or most economical route between two vertices, or the smallest set of edges which connect all the vertices in a graph.

Definition of directed and undirected graphs

A directed graph (V, E) consists of a set V of vertices, a set E of directed edges that are ordered pairs of elements of V . In this graph loop is allowed but no two vertices can have multiple edges in same direction.

Simply a simple graph with flow of direction is called directed graph. The below figure is a directed graph.

Example:

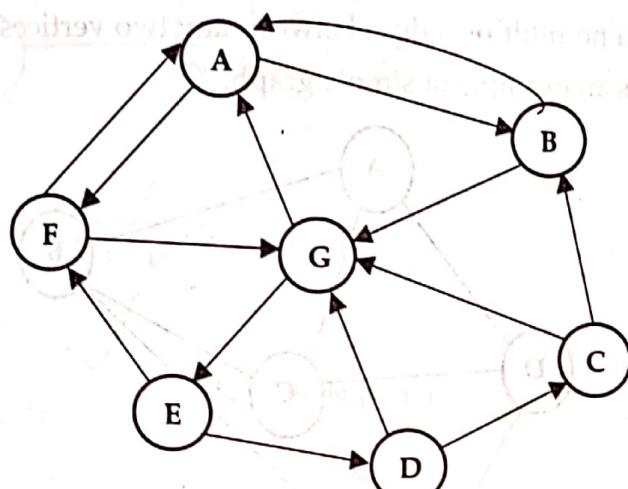


Fig: A directed graph

An undirected graph (V, E) consists of a set V of vertices, a set E of edges that are ordered pairs of elements of V . In this graph each edge has not any direction.

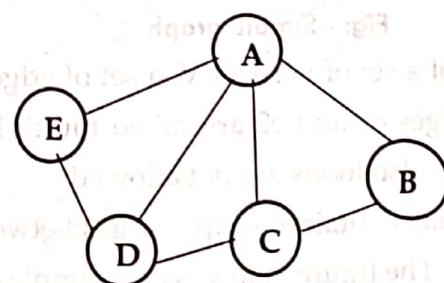


Fig: An undirected graph

Directed multi-graph

A directed multi-graph $G = (V, E)$ consists of a set of vertices V , a set of edges E , and a function f from E to $\{(u, v) \mid u, v \in V\}$. The edges e_1 and e_2 are called multiple edges if $f(e_1) = f(e_2)$. Simply a directed graph in which two vertices may also have multiple edges in the same direction is called directed multi-graph. The figure below is an example of a directed multi-graph.

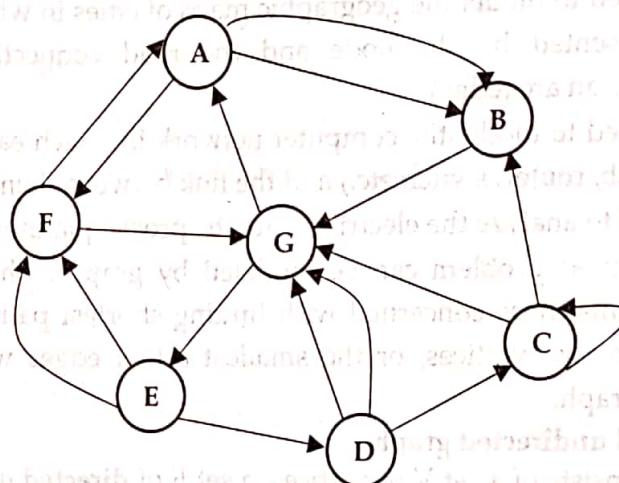


Fig: Directed multi-graph

Simple and multi-graphs

An undirected graph with no multiple edges between any two vertices or loops is called simple graph. The figure below is an example of simple graph.

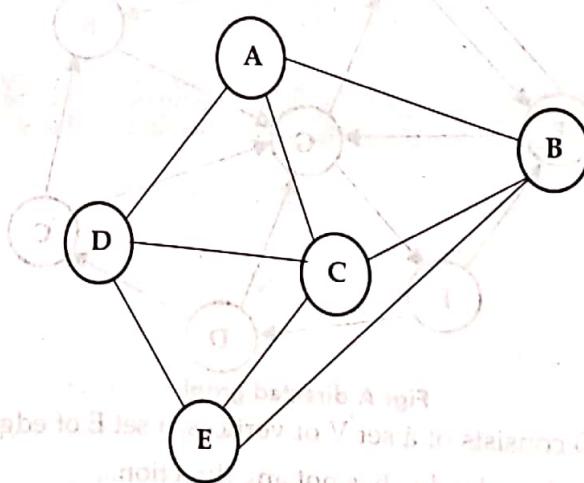
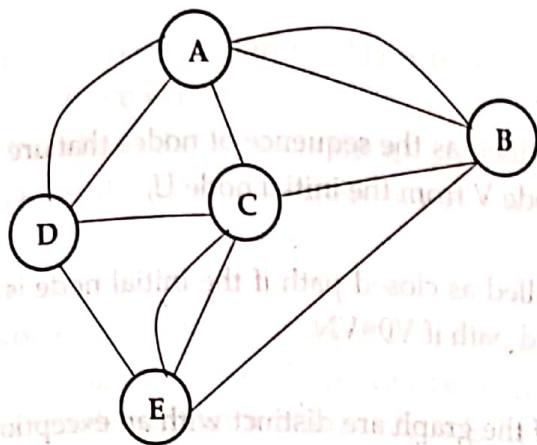


Fig: - Simple graph

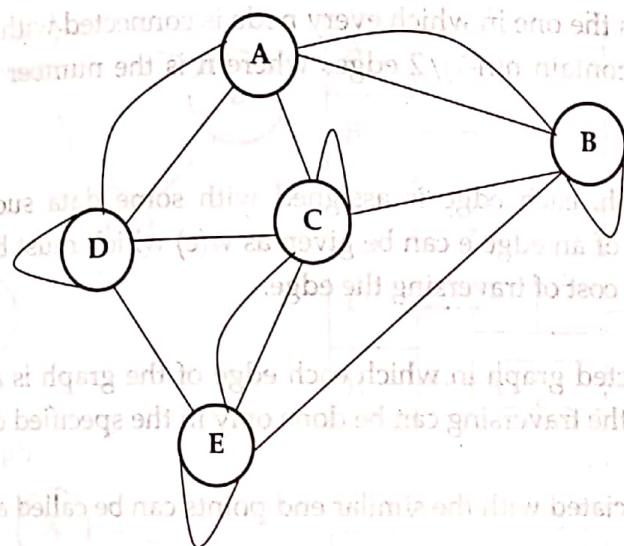
A multi-graph $G = (V, E)$ consists of a set of vertices V , a set of edges E , and a function f from E to $\{\{u, v\} \mid u, v \in V, u \neq v\}$. The edges e_1 and e_2 are called multiple or parallel edges if $f(e_1) = f(e_2)$. In this representation of graph also loops are not allowed.

Simply an undirected graph that may contain multiple edges between any two vertices but not contains loop is called multi-graph. The figure below is an example of a multi-graph.

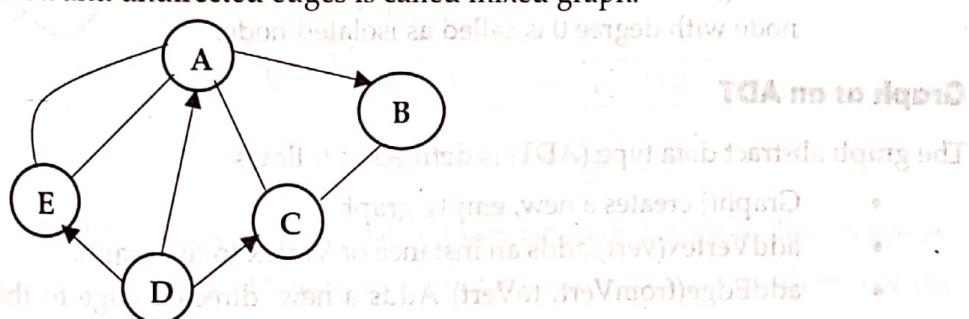
**Fig: Multi-graph****Pseudo-graphs**

A pseudo-graph $G = (V, E)$ consists of a set of vertices V , a set of edges E , and a function f from E to $\{(u, v) \mid u, v \in V\}$. An edge is a loop if $f(e) = \{u, u\} = \{u\}$ for some $u \in V$.

Simply an undirected graph with loops and multiple edges between any two vertices is called pseudo graph. The figure below is an example of a multi-graph.

**Fig: Pseudo graph****Mixed graph**

A graph with both directed and undirected edges is called mixed graph.

**Fig: Mixed graph**

Graph Terminology

- **Path**
A path can be defined as the sequence of nodes that are followed in order to reach some terminal node V from the initial node U.
- **Closed Path**
A path will be called as closed path if the initial node is same as terminal node. A path will be closed path if $V_0 = V_N$.
- **Simple Path**
If all the nodes of the graph are distinct with an exception $V_0 = V_N$, then such path P is called as closed simple path.
- **Cycle**
A cycle can be defined as the path which has no repeated edges or vertices except the first and last vertices.
- **Connected Graph**
A connected graph is the one in which some path exists between every two vertices (u, v) in V. There are no isolated nodes in connected graph.
- **Complete Graph**
A complete graph is the one in which every node is connected with all other nodes. A complete graph contain $n(n-1)/2$ edges where n is the number of nodes in the graph.
- **Weighted Graph**
In a weighted graph, each edge is assigned with some data such as length or weight. The weight of an edge e can be given as $w(e)$ which must be a positive (+) value indicating the cost of traversing the edge.
- **Digraph**
A digraph is a directed graph in which each edge of the graph is associated with some direction and the traversing can be done only in the specified direction.
- **Loop**
An edge that is associated with the similar end points can be called as Loop.
- **Adjacent Nodes**
If two nodes u and v are connected via an edge e, then the nodes u and v are called as neighbors or adjacent nodes.
- **Degree of the Node**
A degree of a node is the number of edges that are connected with that node. A node with degree 0 is called as isolated node.

Graph as an ADT

The graph abstract data type (ADT) is defined as follows:

- **Graph()** creates a new, empty graph.
- **addVertex(vert)** adds an instance of Vertex to the graph.
- **addEdge(fromVert, toVert)** Adds a new, directed edge to the graph that connects two vertices.

- `addEdge(fromVert, toVert, weight)` Adds a new, weighted, directed edge to the graph that connects two vertices.
- `getVertex(vertKey)` finds the vertex in the graph named `vertKey`.
- `getVertices()` returns the list of all vertices in the graph.

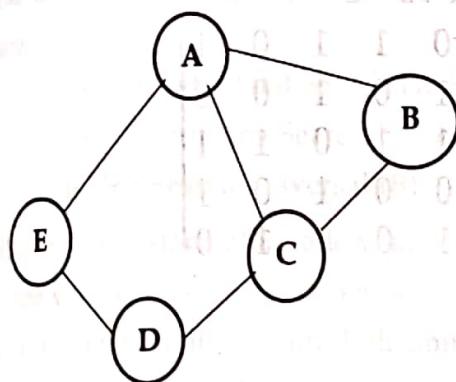
Graph Representation

Graph is a mathematical structure and finds its applications in many areas of interest in which problems need to be solved using computer. Graph can be represented in many ways; one of the ways of representing a graph without multiple edges is by listing its edges. Some other ways are described below:

Adjacency List

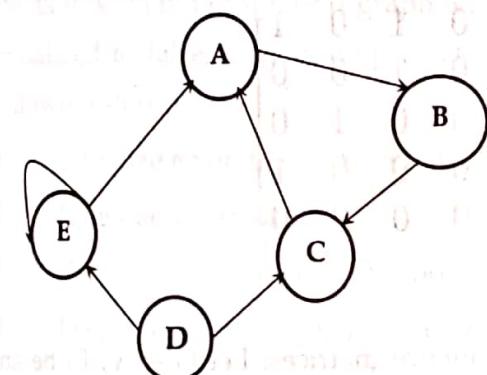
This representation of graph specifies the vertices that are adjacent to each vertex of the graph. This type of representation is suitable for the undirected graphs without multiple edges, and directed graphs. This representation looks as in the tables below.

Example: Let's take an undirected graph



A	B	C	E	\0
B	A	C	\0	
C	A	B	D	\0
D	C	E	\0	
E	A	D	\0	

For directed graph



A	B	\0
B	C	\0
C	A	\0
D	C	E
E	A	E

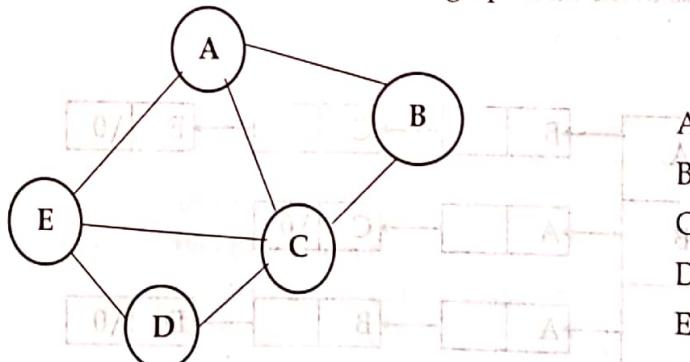
If we try to apply the algorithms of graph using the representation of graphs by lists of edges, or adjacency lists it can be tedious and time taking if there are high numbers of edges. For the sake of the computation, the graphs with many edges can be represented in other ways. In this class we discuss two ways of representing graphs in form of matrix.

Adjacency matrix representation of graph

The adjacency lists, can be cumbersome if there are many edges in the graph. To simplify computation, graphs can be represented using matrices. Two types of matrices commonly used to represent graphs will be presented here. One is based on the adjacency of vertices, and the other is based on incidence of vertices and edges. Suppose that $G = (V, E)$ is a simple graph where $|V| = n$. Suppose that the vertices of G are listed arbitrarily as v_1, v_2, \dots, v_n . The adjacency matrix A of G , with respect to this listing of the vertices, is the $n \times n$ zero-one matrix with 1 as its $(i, j)^{\text{th}}$ entry when v_i and v_j are adjacent, and 0 as its $(i, j)^{\text{th}}$ entry when they are not adjacent. In other words, if its adjacency matrix is $A = [a_{ij}]$, then

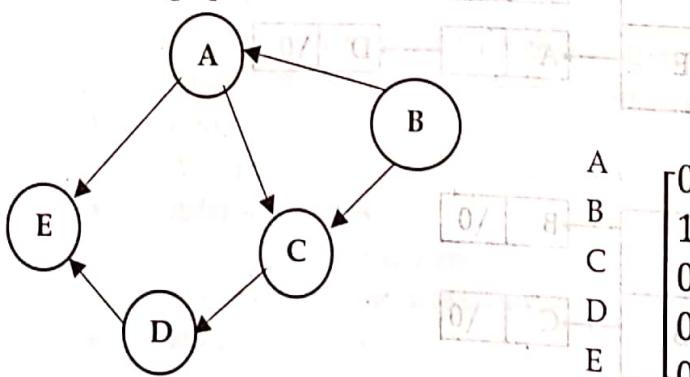
$$a_{ij} = \begin{cases} 1 & \text{if } \{v_i, v_j\} \text{ is an edge of } G, \\ 0 & \text{otherwise} \end{cases}$$

Example: Let's take an undirected graph



	A	B	C	D	E
A	0	1	1	0	1
B	1	0	1	0	0
C	1	1	0	1	1
D	0	0	1	0	1
E	1	0	1	1	0

For directed graph



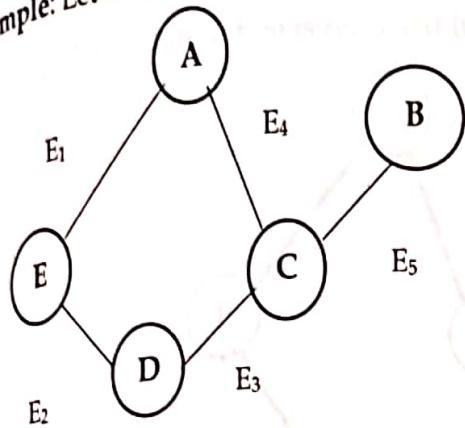
	A	B	C	D	E
A	0	0	1	0	1
B	1	0	1	0	0
C	0	0	0	1	0
D	0	0	0	0	1
E	0	0	0	0	0

Incidence Matrices

Another common way to represent graphs is to use incidence matrices. Let $G = (V, E)$ be an undirected graph. Suppose that v_1, v_2, \dots, v_n are the vertices and e_1, e_2, \dots, e_m are the edges of G . Then the incidence matrix with respect to this ordering of V and E is the $n \times m$ matrix $M = [m_{ij}]$, where $m_{ij} = 1$ when edge e_j is incident with v_i , 0 otherwise.

$$m_{ij} = \begin{cases} 1 & \text{when edge } e_j \text{ is incident with } v_i, \\ 0 & \text{otherwise} \end{cases}$$

Example: Let's take an undirected graph



	E1	E2	E3	E4	E5
A	1	0	0	1	0
B	0	0	0	0	1
C	0	0	1	1	1
D	0	1	1	0	0
E	1	1	0	0	0

GRAPH TRAVERSALS

Traversing a graph means visiting all the vertices in a graph exactly one. In tree traversal, there is a special node root, from which the traversal starts. But in graph all nodes are treated equally. So the starting node in graph can be chosen arbitrarily. In general, for diagrammatic illustration, the left-up node is taken as starting vertex for traversal. Two common approaches for traversal are:

- Breadth First Search Traversal (BFS)
- Depth First Search Traversal (DFS)

Breadth First Search Traversal (BFS)

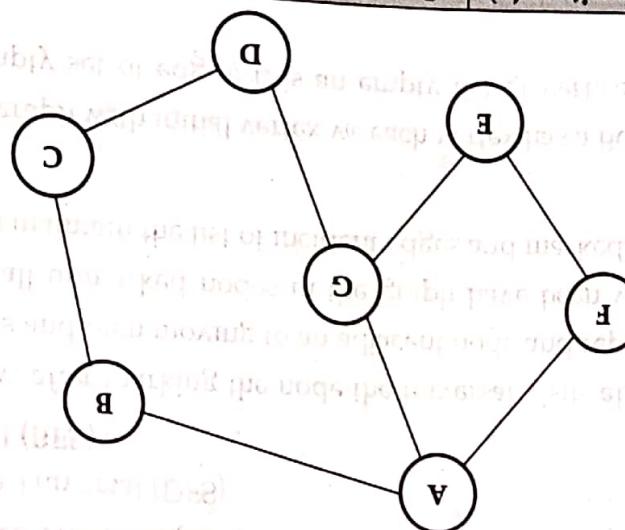
The traversal starts at a node v_0 , after marking the node the traversal visits all incident edges to node v_0 after marking the nodes and then moving to an adjacent node and repeating the process. The traversal continues until all unmarked nodes in the graph have been visited. A queue is maintained in the technique to maintain the list of incident edges and marked nodes.

Steps in BFS

Let, $G = (V, E)$ is a graph or digraph with initial vertex v_0 ; each vertex has a Boolean visited field initialized to false; T is an empty set of edges; L is an empty list of vertices. Then it follows following steps:

1. Initialize an empty queue Q for temporary storage of vertices.
2. Enqueue v_0 into Q .
3. Repeat steps 4–6 while Q is not empty.
4. Dequeue Q and set such item into v .
5. Add v to vertex list, L .
6. Do step 7 for each vertex w that is adjacent to v .
7. If w has not been visited, do steps 8–9.
8. Add the edge vw to edge list, T .
9. Enqueue w into queue, Q .

Queue (Q)	Dequeue item (v)	Set of vertices (L)	Adjacent vertex(w)	Edge list (T)
A	
	A
	AB
B	
	AB
BF	
BFG	B
BFGC	B	A, B	F, G	AB, AF, AG, BC
BFGC	F	A, B, F	G	AB, AF, AG, BC
BFGC	G	A, B, F, G	C	AB, AF, AG, BC
CED	C	A, B, F, G, C	D	AB, AF, AG, BC, EF
CEDE	E	A, B, F, G, C, E	D	AB, AF, AG, BC, EF, CD
DE	D
.....



Let's take start vertex is $v_0 = A$

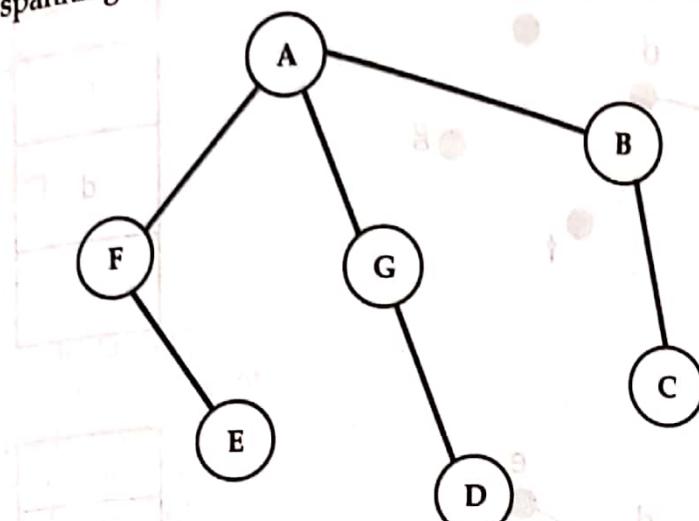
Tracing of BFS algorithm

```

Algorithm
BFS (G, s) // s is start vertex
{
    L = Φ; // an empty queue
    T = {s}; // enqueue (L, s);
    while (L ≠ Φ)
    {
        v = Dequeue (L); // For each neighbor w to v
        if (w ∈ L & w ∉ T)
        {
            T = T ∪ {w}; // put edge {v, w}
            Enqueue (L, w);
        }
    }
}

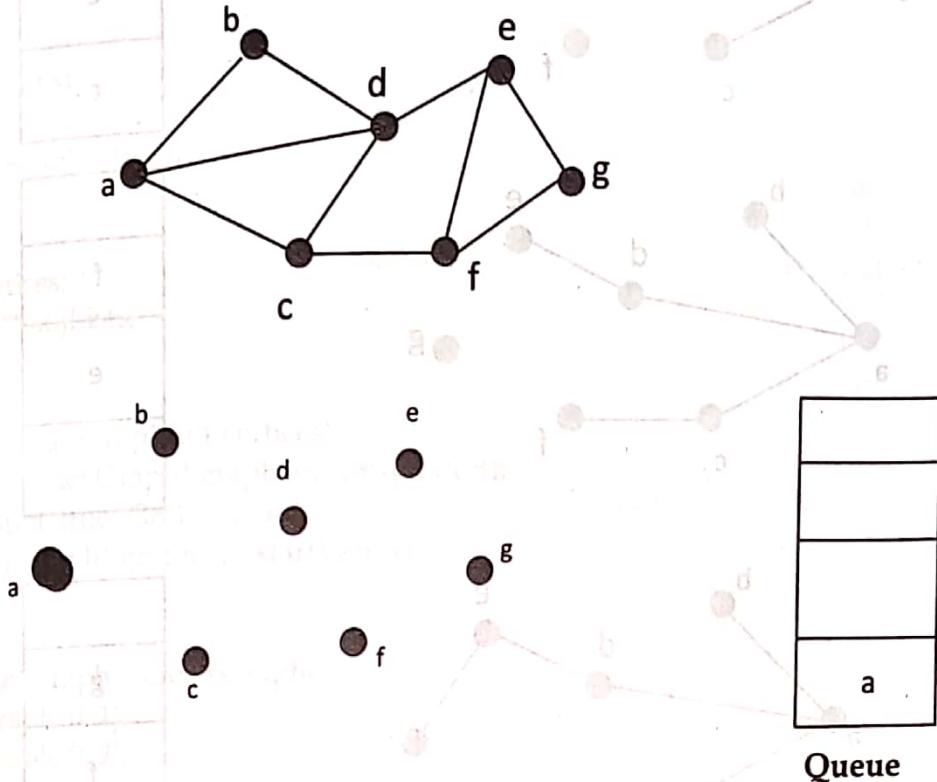
```

The resulting BFS order of visitation is returned in the list $L = (A, B, F, G, C, E, D)$, and the resulting BFS spanning tree is returned in the set $T = \{AB, AF, AG, BC, EF, GD\}$.

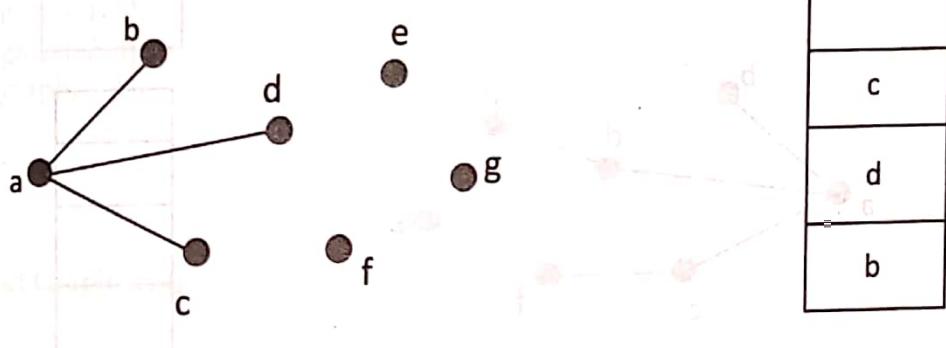


Tracing method 2 for BFS

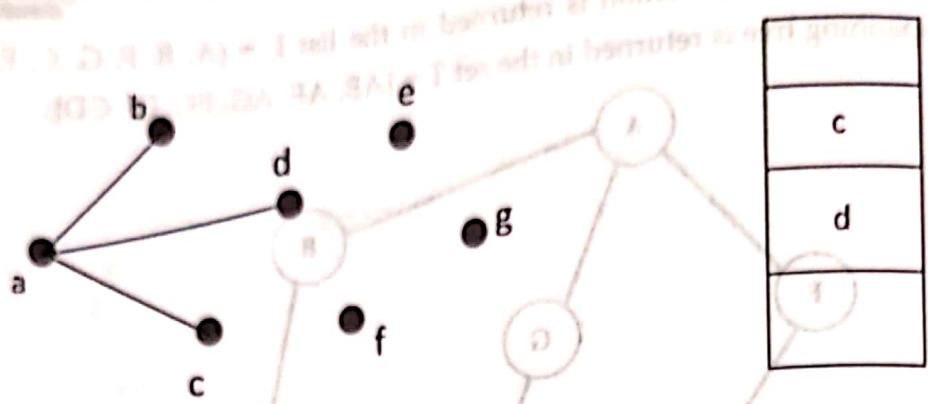
Step 1:



Step 2:



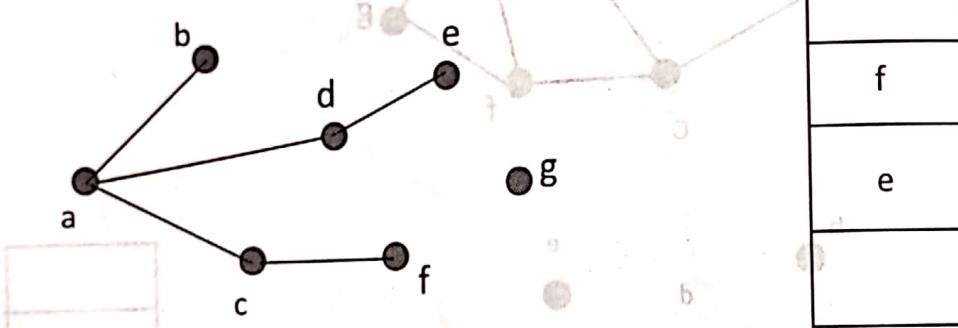
Step 3:



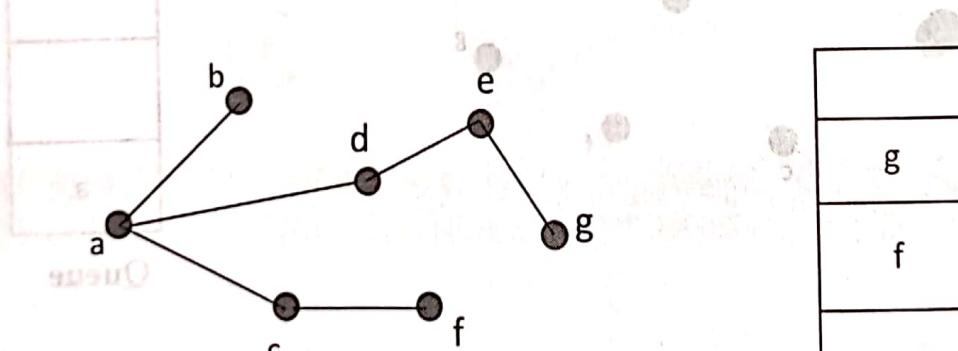
Step 4:



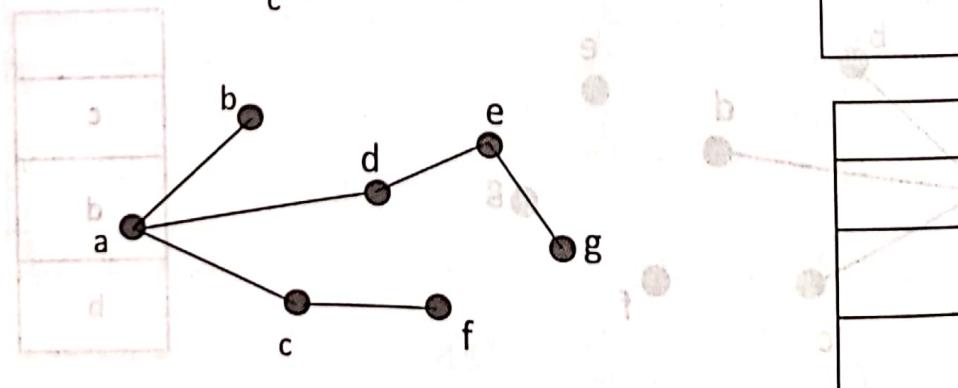
Step 4:



Step 5:



Step 6:



Complete C program for BFS

```

include <stdio.h>
#include <stdlib.h>
#define SIZE 40
struct queue
{
    int items[SIZE];
    int front;
    int rear;
};
struct queue* createQueue();
void enqueue(struct queue* q, int);
int dequeue(struct queue* q);
void display(struct queue* q);
int isEmpty(struct queue* q);
void printQueue(struct queue* q);
struct node
{
    int vertex;
    struct node* next;
};
struct node* createNode(int);
struct Graph
{
    int numVertices;
    struct node** adjLists;
    int* visited;
};
struct Graph* createGraph(int vertices);
void addEdge(struct Graph* graph, int src, int dest);
void printGraph(struct Graph* graph);
void bfs(struct Graph* graph, int startVertex);
int main()
{
    struct Graph* graph = createGraph(6);
    addEdge(graph, 0, 1);
    addEdge(graph, 0, 2);
    addEdge(graph, 1, 2);
    addEdge(graph, 1, 4);
    addEdge(graph, 1, 3);
    addEdge(graph, 2, 4);
    addEdge(graph, 3, 4);
    bfs(graph, 0);
    return 0;
}
void bfs(struct Graph* graph, int startVertex)
{

```

```

struct queue* q = createQueue();
graph->visited[startVertex] = 1;
enqueue(q, startVertex);
while(!isEmpty(q))
{
    printQueue(q);
    int currentVertex = dequeue(q);
    printf("Visited %d\n", currentVertex);
    struct node* temp = graph->adjLists[currentVertex];
    while(temp)
    {
        int adjVertex = temp->vertex;
        if(graph->visited[adjVertex] == 0)
        {
            graph->visited[adjVertex] = 1;
            enqueue(q, adjVertex);
        }
        temp = temp->next;
    }
}

struct node* createNode(int v)
{
    struct node* newNode = malloc(sizeof(struct node));
    newNode->vertex = v;
    newNode->next = NULL;
    return newNode;
}

struct Graph* createGraph(int vertices)
{
    struct Graph* graph = malloc(sizeof(struct Graph));
    graph->numVertices = vertices;
    graph->adjLists = malloc(vertices * sizeof(struct node*));
    graph->visited = malloc(vertices * sizeof(int));

    int i;
    for (i = 0; i < vertices; i++)
    {
        graph->adjLists[i] = NULL;
        graph->visited[i] = 0;
    }
    return graph;
}

void addEdge(struct Graph* graph, int src, int dest)

```

```

| struct node* newNode = createNode(dest); // Add edge from src to dest
| newNode->next = graph->adjLists[src];
| graph->adjLists[src] = newNode;
| newNode = createNode(src); // Add edge from dest to src
| newNode->next = graph->adjLists[dest];
| graph->adjLists[dest] = newNode;
| }

| struct queue* createQueue()
| {
|     struct queue* q = malloc(sizeof(struct queue));
|     q->front = -1;
|     q->rear = -1;
|     return q;
| }

| int isEmpty(struct queue* q)
| {
|     if(q->rear == -1) // If queue is empty
|         return 1;
|     else
|         return 0;
| }

| void enqueue(struct queue* q, int value)
| {
|     if(q->rear == SIZE-1) // If queue is full
|         printf("\n Queue is Full!!!");
|     else
|     {
|         if(q->front == -1) // If queue is empty
|             q->front = 0;
|         q->rear++;
|         q->items[q->rear] = value;
|     }
| }

| int dequeue(struct queue* q)
| {
|     int item;
|     if(isEmpty(q))
|     {
|         printf("Queue is empty");
|         item = -1;
|     }
|     else
|     {
|         item = q->items[q->front];
|     }
|     return item;
| }

```

```

    q→front++;
    if(q→front > q→rear)
    {
        printf("Resetting queue");
        q→front = q→rear = -1;
    }
}
return item;
}
void printQueue(struct queue *q)
{
    int i = q→front;
    if(isEmpty(q))
    {
        printf("Queue is empty");
    }
    else
    {
        printf("\n Queue contains \n");
        for(i = q→front; i < q→rear + 1; i++)
            printf("%d ", q→items[i]);
    }
}

```

DEPTH FIRST TRAVERSAL (DFS)

DFS traversal of a graph produces a **spanning tree** as final result. **Spanning Tree** is a graph without any loops. We use **Stack data structure** with maximum size of total number of vertices in the graph to implement DFS traversal of a graph.

A depth first search of an arbitrary graph can be used to perform a traversal of a general graph. The technique picks up a node and marks it. An unmarked adjacent node to previous node is then selected and marked, becomes the new start node, possibly leaving the previous node with unexplored edges for the present. The traversal continued recursively, until all unmarked nodes of the current path are visited. The process is continued for all the paths of the graph. If this cannot be done, move back to another vertex and repeat the process. The whole process is continued until all the vertices are met. This method of search is also called backtracking. A Stack data structure is maintained in the technique to maintain the list of incident edges and marked nodes.

Algorithm

1. Start
2. Define a Stack of size total number of vertices in the graph.
3. Select any vertex as starting point for traversal. Visit that vertex and push it on to the Stack.
4. Visit any one of the adjacent vertex of the vertex which is at top of the stack which is not visited and push it on to the stack.

5. Repeat step 4 until there are no new vertex to be visit from the vertex on top of the stack.
 6. When there is no new vertex to be visit then use back tracking and pop one vertex from the stack.
 7. Repeat steps 4, 5 and 6 until stack becomes Empty
 8. When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph
 9. Stop

Pseudo code

Recursive steps for DFS are shown in below:
 $\text{DFS}(G, s)$

```
| T={s};  
| Traverse (s);
```

```
| Traverse (v)
```

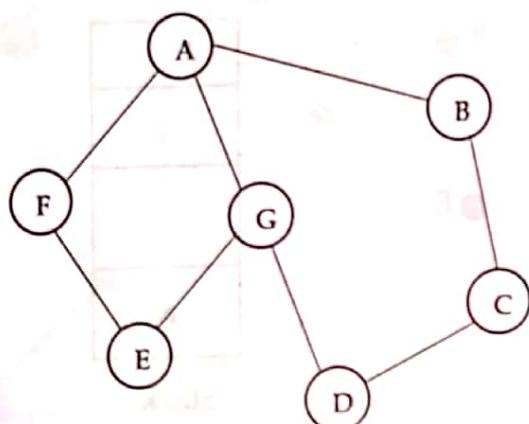
```
| for each w adjacency to v and not yet in T
```

```
| {  
|   T=T U {w}; // put edge {v, tw} also  
|   Traverse (w);
```

```
| }
```

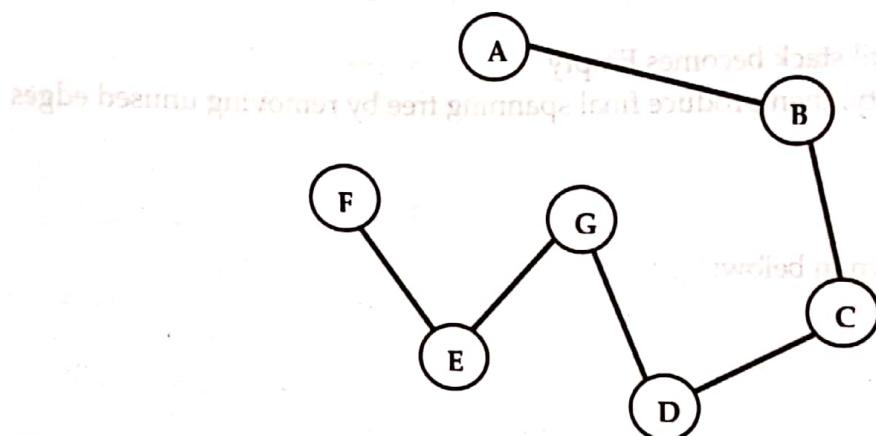
Tracing

The start vertex is $v_0 = A$

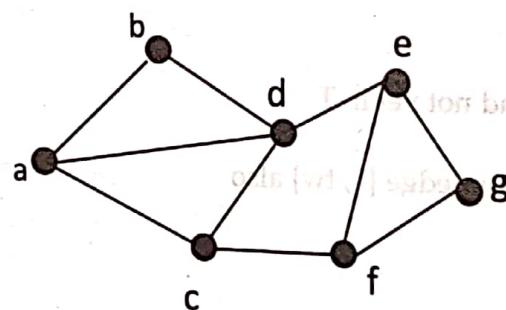


Set of vertices (L)	Stack (S)	Popped element of Stack (v)	Adjacent vertex (w)	Edge list (T)
A	A
.....	A	B	AB
A,B	B	B	C	AB, BC
A,B,C	C	C	D	AB, BC, CD
A,B,C,D	D	D	G	AB, BC, CD, DG
A,B,C,D,G	G	G	E	AB, BC, CD, DG, GE
A,B,C,D,G,E	E	E	F	AB, BC, CD, DG, GE, EF
A,B,C,D,G,E,F	F	F	AB, BC, CD, DG, GE, EF
A,B,C,D,G,E,F	AB, BC, CD, DG, GE, EF

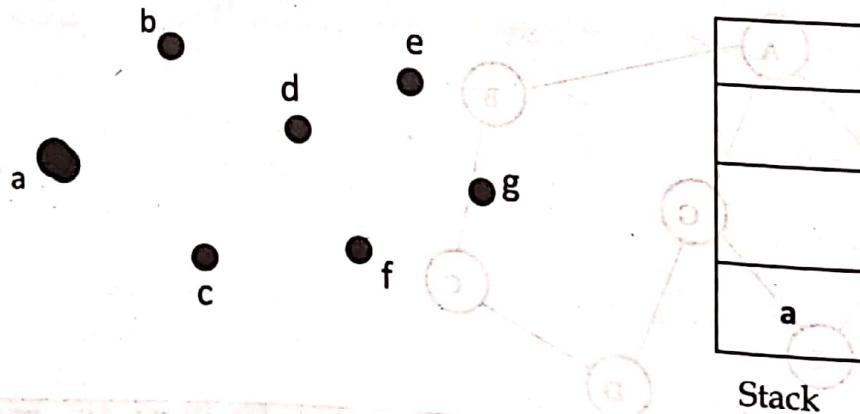
The resulting BFS order of visitation is returned in the list $L = (A, B, C, D, G, E, F)$, and the resulting DFS spanning tree is returned in the set $T = \{AB, BC, CD, DG, GE, EF\}$



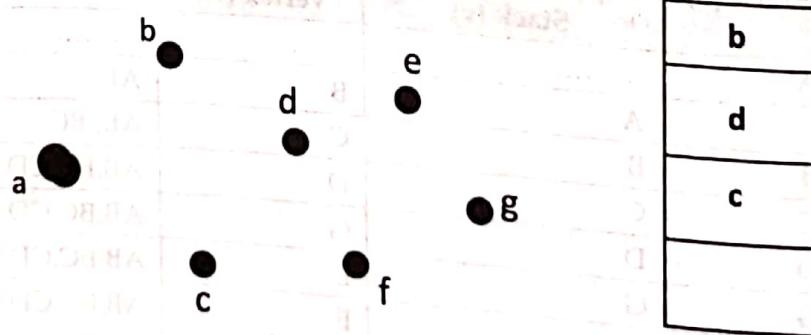
Tracing method 2 for DFS

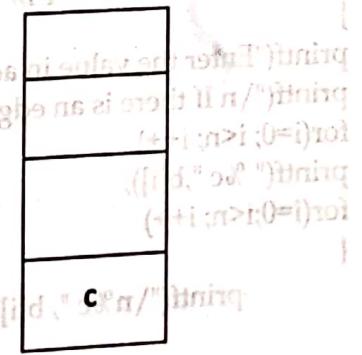
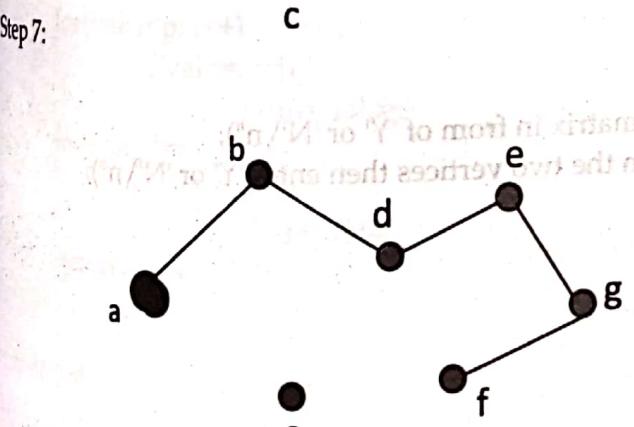
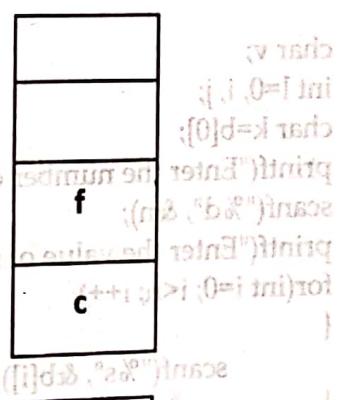
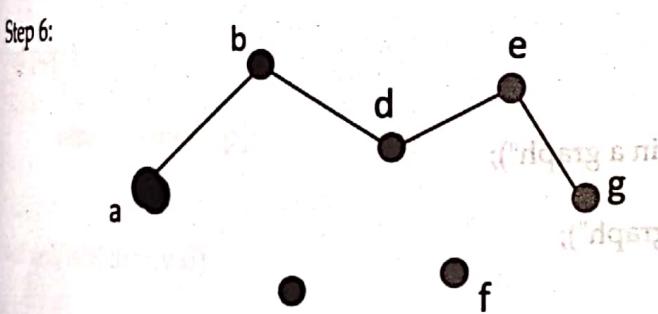
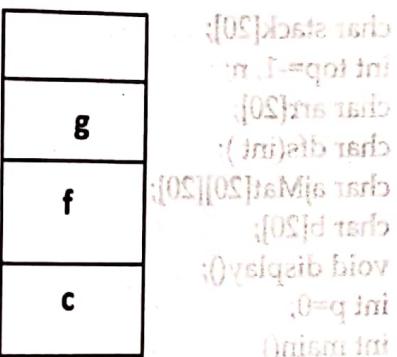
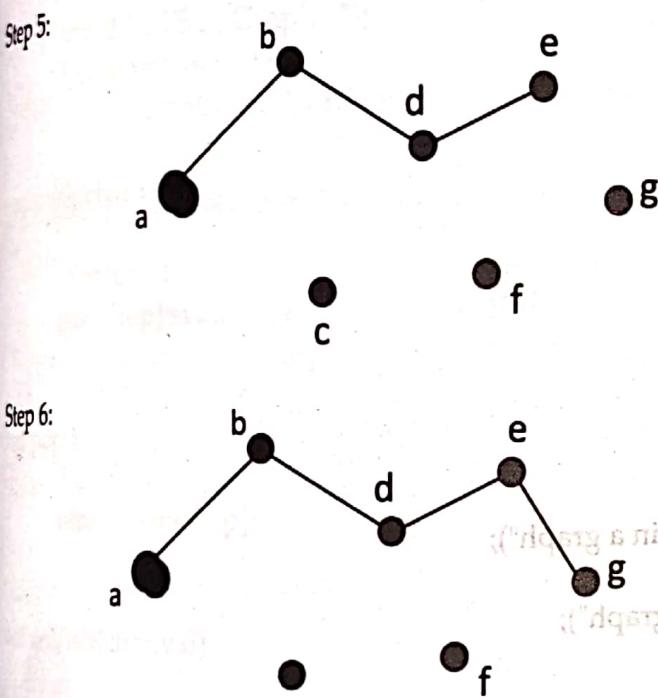
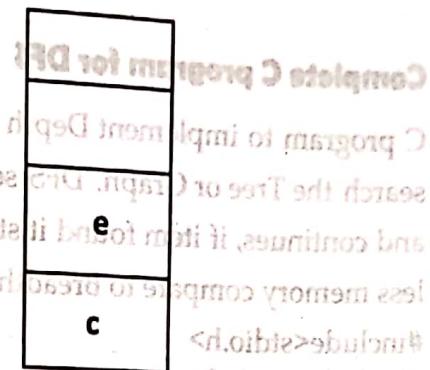
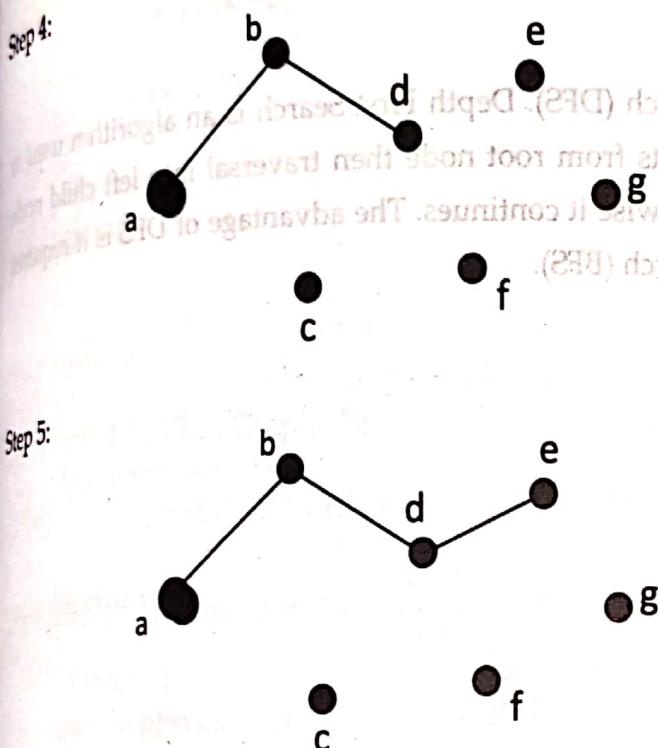
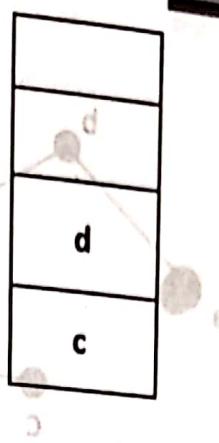
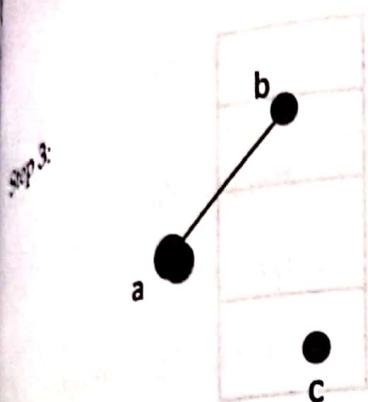


Step 1:



Step 2:

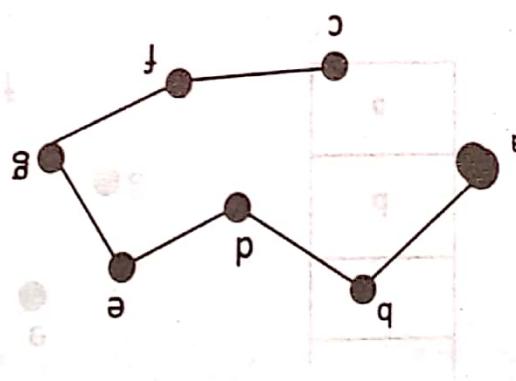
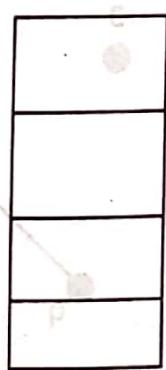




```

#include<stdio.h>
char stack[20];
char arr[20];
int top=-1, n;
char dfs(int);
char adjMat[20][20];
void display();
int p=0;
int main()
{
    int k=b[0];
    char v;
    int i=0, j;
    scanf("%d", &n);
    printf("Enter the number of nodes in a graph:");
    scanf("%d", &k);
    for(i=0; i<n; i++)
    {
        printf("Enter the value of node %c", i+1);
        scanf("%c", &b[i]);
    }
    printf("Enter the value in adjacency matrix in form of 'y' or 'N':");
    for(i=0; i<n; i++)
    {
        for(j=0; j<n; j++)
        {
            if(b[i] == 'y')
                adjMat[i][j] = 1;
            else
                adjMat[i][j] = 0;
        }
    }
    display();
}
char dfs(int i)
{
    if(p==i)
        return 1;
    p=i;
    printf("Enter the value of node %c", i+1);
    printf("Enter the value between the two vertices then enter 'y' or 'N':");
    char c=b[i];
    if(c=='y')
        printf("There is an edge between the two vertices in form of 'y' or 'N':");
    else
        printf("There is no edge between the two vertices in form of 'y' or 'N':");
    int j;
    for(j=0; j<n; j++)
    {
        if(adjMat[i][j]==1)
            dfs(j);
    }
}

```

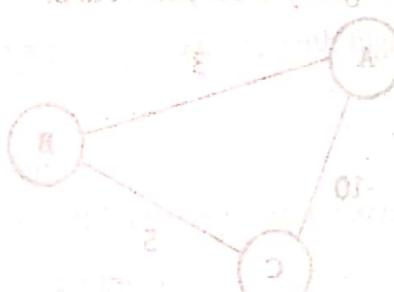


Complete C program for DFS

```

        for(j=0; j<n; j++)
        {
            printf("%c ", v=getch());
            ajMat[i][j]=v;
        }
        printf("\n\n");
    }
    for(int i=0; i<n; i++)
    {
        l=0;
        while(k!=b[l])
            l++;
        k=dfs(l);
    }
    display();
    getch();
}
void display()
{
    printf(" DFS of Graph : ");
    for(int i=0; i<n; i++)
        printf("%c ", arr[i]);
}
void push(char val)
{
    top=top+1;
    stack[top]=val;
}
char pop()
{
    return stack[top];
}
bool unVisit(char val)
{
    for(i=0; i<p; i++)
        if(val==arr[i])
            return false;
    for(i=0; i<=top; i++)
        if(val==stack[i])
            return false;
    return true;
}
char dfs(int i)
{

```



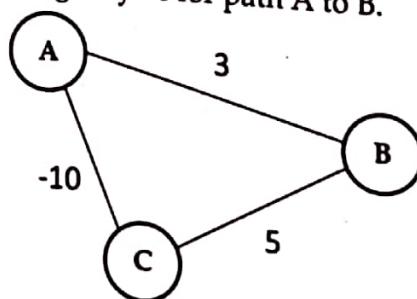
```

int k;
char m;
if(top== -1)
{
    push(b[i]);
}
m=pop();
top--;
arr[p]=m;
p++;
for(int j=0; j<n; j++)
{
    if(aJMat[i][j]== 'y')
    {
        if(unVisit(b[j]))
        {
            push(b[j]);
        }
    }
}
return stack[top];
}

```

SHORTEST PATHS

Given a weighted graph $G = (V, E)$, then it has weight for every path $p = \langle v_0, v_1, \dots, v_{k-1}, v_k \rangle$ as $w(p) = w(v_0, v_1) + w(v_1, v_2) + \dots + w(v_{k-1}, v_k)$. A shortest path from u to v is the path from u to v with minimum weight. Shortest path from u to v is denoted by $d(u, v)$. It is important to remember that the shortest path may exist in a graph or may not i.e. if there is negative weight cycle then there is no shortest path. For e.g. the below graph has no shortest path from A to C . You can notice the negative weight cycle for path A to B .



As a matter of fact, even the positive weight cycle doesn't constitute shortest path but there will be shortest path. Some of the variations of shortest path problem include:

Single Source shortest path problem
This type of problem asks us to find the shortest path from the given vertex (source) to all other vertices in a connected graph. Here only one source vertex and lot of destination vertices are take place.

Single Destination shortest path problem

This type of problem asks us to find the shortest path to the given vertex (destination) from all other vertices in a connected graph.

Single Pair shortest path problem

This type of problem asks us to find the shortest path from the given vertex (source) to another given vertex (destination). In this type of problem, we need to find shortest path form a fixed source vertex to one of the destination vertex.

All Pairs

This type of problem asks us to find the shortest path from the all vertices to all other vertices in a connected graph.

Relaxation

Relaxation of an edge (u, v) is a process of testing the total weight of the shortest path to v by going through u and if we get the weight less than the previous one then replacing the record of previous shortest path by new one.

DIJKSTRA'S ALGORITHM

This is another approach of getting single source shortest paths. In this algorithm it is assumed that there is no negative weight edge. Dijkstra's algorithm works using greedy approach, as we will see later. Dijkstra's algorithm finds the shortest path from one vertex v_0 to each other vertex in a digraph. When it has finished, the length of the shortest distance from v_0 to v is stored in the vertex v , and the shortest path from v_0 to v is recorded in the back pointers of v and the other vertices along that path.

Steps in Dijkstra's algorithm

Precondition: $G = (V, w)$ is a weighted graph with initial vertex v_0 then it holds following steps:

1. Initialize the distance field to 0 for v_0 and to for each of the other vertices.
2. Enqueue all the vertices into a priority queue Q with highest priority being the lowest distance field value.
3. Repeat steps 4-10 until Q is empty.
4. The distance and back reference fields of every vertex that is not in Q are correct
5. Dequeue the highest priority vertex into v .
6. Do steps 7-10 for each vertex w that are adjacent to v and in the priority queue.
7. Let S be the sum of the v 's distance field plus the weight of the edge from v to w .
8. If S is less than w 's distance field, do steps 9-10; otherwise go back to Step3.
9. Assign s to w 's distance field.
10. Assign v to w 's back reference field.

AlgorithmDijkstra_Algorithm(G, w, s)

```

for each vertex  $v \in V$ 
     $d[v] = \Phi$ 
 $d[s] = 0$ 
 $S = \Phi$ 
 $Q = V$ 
while ( $Q \neq \Phi$ )
{

```

initialization: u = Take minimum from Q and delete.

$S = S \cup \{u\}$

for each vertex v adjacent to u

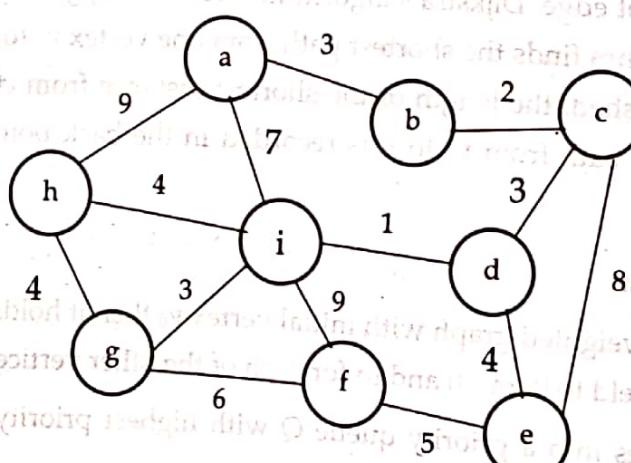
if $d[v] > d[u] + w(u, v)$ then

{
 $d[v] = d[u] + w(u, v)$
 $Q = Q \setminus \{v\}$
}

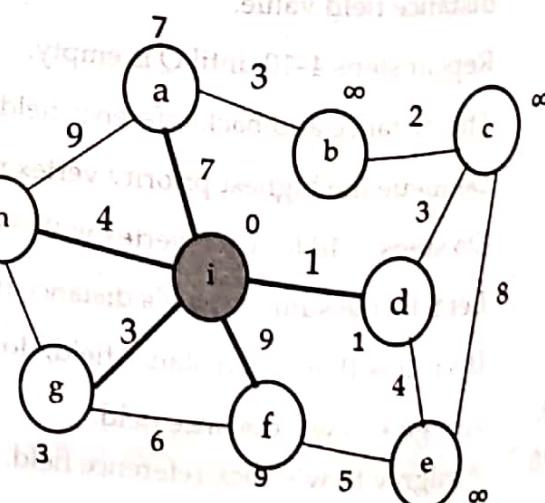
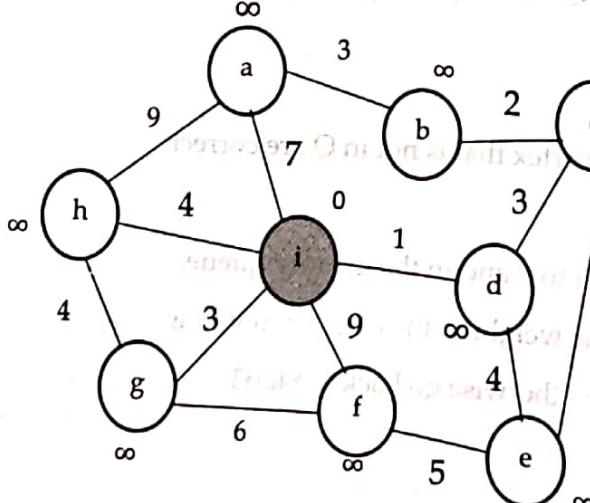
Analysis

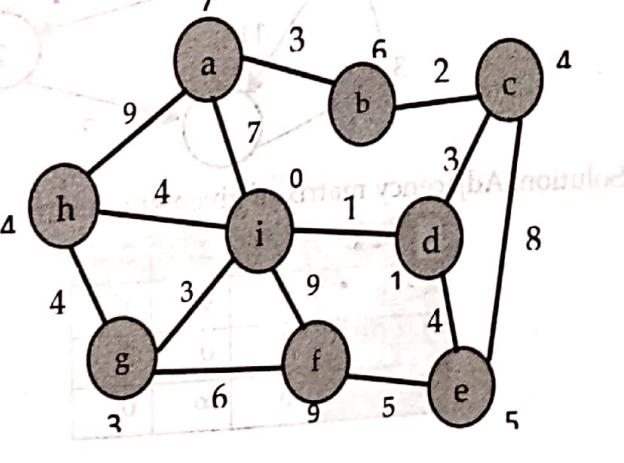
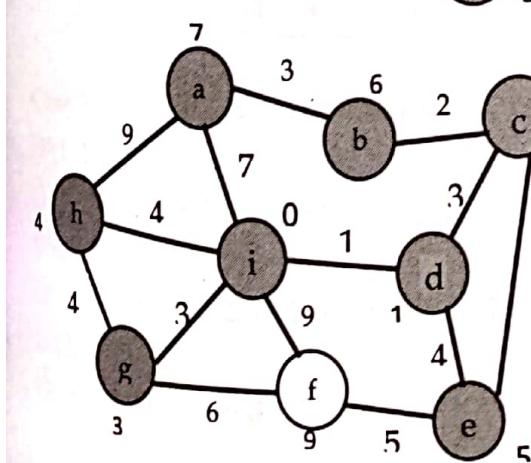
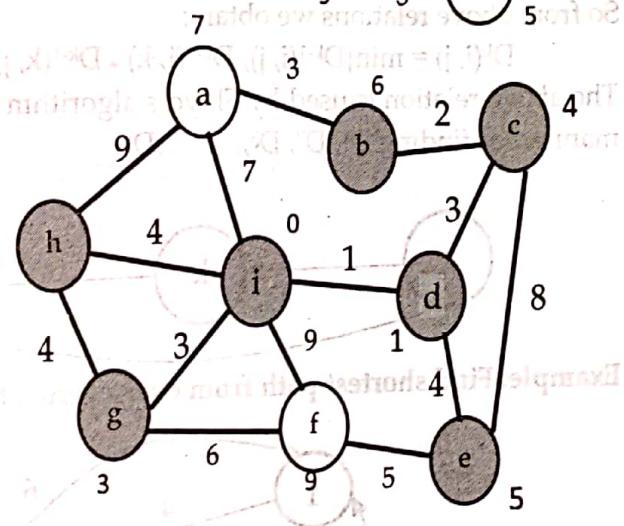
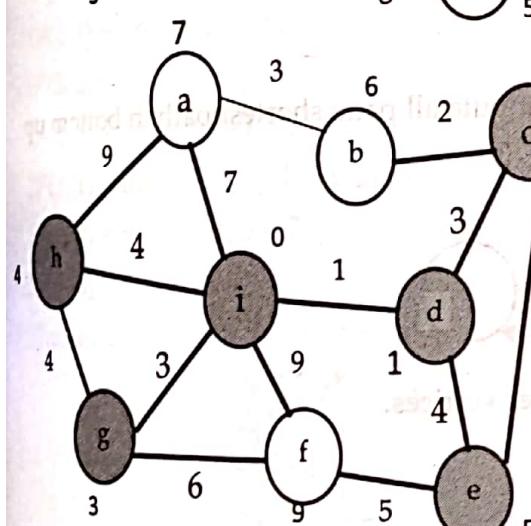
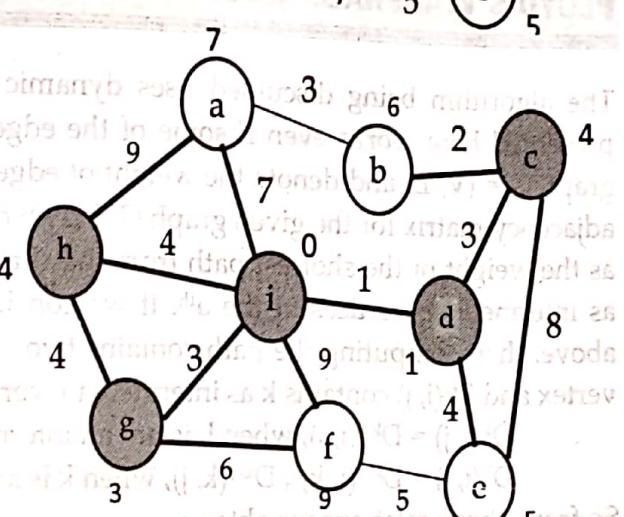
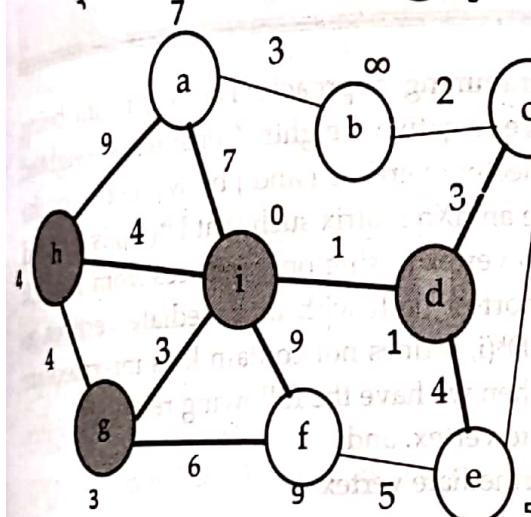
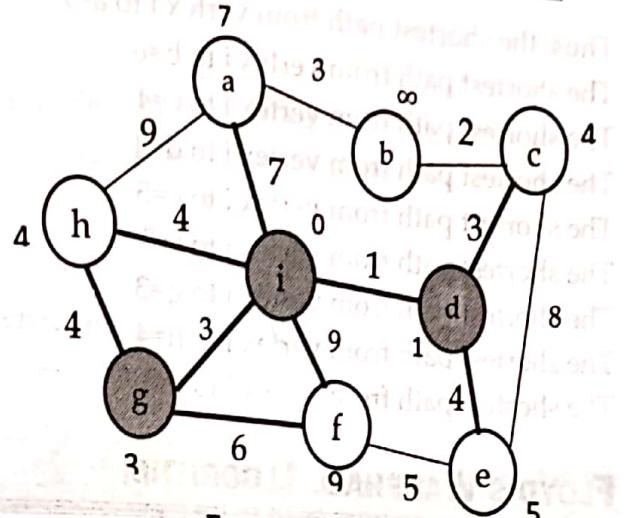
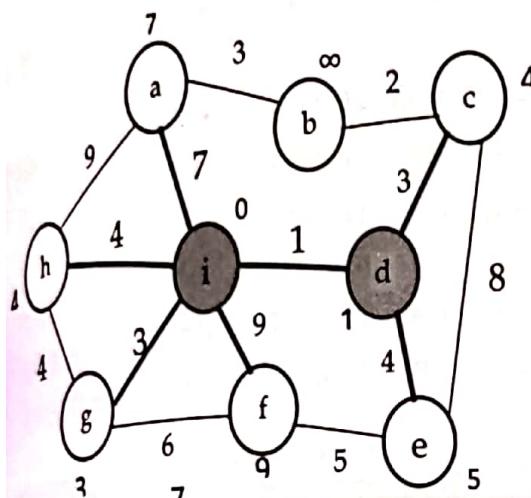
In the above algorithm, the first for loop block takes $O(V)$ time. Initialization of priority queue Q takes $O(V)$ time. The while loop executes for $O(V)$, where for each execution the block inside the loop takes $O(V)$ times. Hence the total running time is $O(V^2)$.

Example Find the shortest paths from the source node i to all other vertices using Dijkstra's algorithm.



Solution:





Thus, the shortest path from vertex i to a=7

The shortest path from vertex i to b=6

The shortest path from vertex i to c=4

The shortest path from vertex i to d=1

The shortest path from vertex i to e=5

The shortest path from vertex i to f=9

The shortest path from vertex i to g=3

The shortest path from vertex i to h=4

The shortest path from vertex i to i=0



FLOYD'S WARSHALL ALGORITHM

The algorithm being discussed uses dynamic programming approach. The algorithm being presented here works even if some of the edges have negative weights. Consider a weighted graph $G = (V, E)$ and denote the weight of edge connecting vertices i and j by w_{ij} . Let W be the adjacency matrix for the given graph G . Let D^k denote an $n \times n$ matrix such that $D^k(i, j)$ is defined as the weight of the shortest path from the vertex i to vertex j using only vertices from $1, 2, \dots, k$ as intermediate vertices in the a^{th} . If we consider shortest path with intermediate vertices as above, then computing the path contains two cases. $D^k(i, j)$ does not contain k as intermediate vertex and $D^k(i, j)$ contains k as intermediate vertex. Then we have the following relations

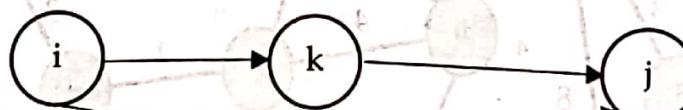
$$D^k(i, j) = D^{k-1}(i, j), \text{ when } k \text{ is not an intermediate vertex, and}$$

$$D^k(i, j) = D^{k-1}(i, k) + D^{k-1}(k, j), \text{ when } k \text{ is an intermediate vertex}$$

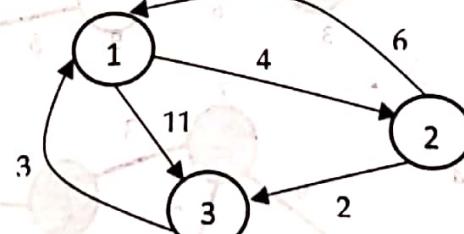
So from above relations we obtain:

$$D^k(i, j) = \min\{D^{k-1}(i, j), D^{k-1}(i, k) + D^{k-1}(k, j)\}$$

The above relation is used by Floyd's algorithm to compute all pairs shortest path in bottom up manner for finding $D^1, D^2, D^3, \dots, D^n$.



Example: Find shortest path from every vertex to other vertices.



Solution: Adjacency matrix of given graph is;

W or D^0	1	2	3
1	0	4	11
2	6	0	2
3	3	∞	0

Case 1: When vertex (1) as intermediate vertex:

$$D^1(1, 1) = 0$$

$$D^1(1, 2) = \min\{D^0(1, 2), D^0(1, 1) + D^0(1, 2)\} = \text{unchanged} = \min\{4, 0+4\} = 4$$

$$D^1(1, 3) = \min\{D^0(1, 3), D^0(1, 1) + D^0(1, 3)\} = \text{unchanged} = \min\{11, 0+11\} = 11$$

$$D^1(2, 1) = \min\{D^0(2, 1), D^0(2, 1) + D^0(1, 1)\} = \text{unchanged} = \min\{6, 6+0\} = 6$$

$$D^1(2, 2) = 0$$

$$D^1(2, 3) = \min\{D^0(2, 3), D^0(2, 1) + D^0(1, 3)\} = \text{may change} = \min\{2, 6+11\} = 2$$

$$D^1(3, 1) = \min\{D^0(3, 1), D^0(3, 1) + D^0(1, 1)\} = \text{unchanged} = \min\{3, 3+0\} = 3$$

$$D^1(3, 2) = \min\{D^0(3, 2), D^0(3, 1) + D^0(1, 2)\} = \text{may change} = \min\{\infty, 3+4\} = 7$$

$$D^1(3, 3) = 0$$

Thus adjacency matrix can be modified as

D ¹	1	2	3
1	0	4	11
2	6	0	2
3	3	7	0

Case 2: When vertex (2) as intermediate vertex:

$$D^2(1, 1) = 0$$

$$D^2(1, 2) = \text{unchanged} = 4$$

$$D^2(1, 3) = \min\{D^1(1, 3), D^1(1, 2) + D^1(2, 3)\} = \text{may change} = \min\{11, 4+2\} = 6$$

$$D^2(2, 1) = \text{unchanged} = 6$$

$$D^2(2, 2) = 0$$

$$D^2(2, 3) = \text{unchanged} = 2$$

$$D^2(3, 1) = \min\{D^1(3, 1), D^1(3, 2) + D^1(2, 1)\} = \text{may change} = \min\{3, 7+6\} = 3$$

$$D^2(3, 2) = \text{unchanged} = 7$$

$$D^2(3, 3) = 0$$

Thus adjacency matrix can be modified as

D ²	1	2	3
1	0	4	6
2	6	0	2
3	3	7	0

Case 3: When vertex (3) as intermediate vertex:

$$D^3(1, 1) = 0$$

$$D^3(1, 2) = \min\{D^2(1, 2), D^2(1, 3) + D^2(3, 2)\} = \text{may change} = \min\{4, 6+7\} = 4$$

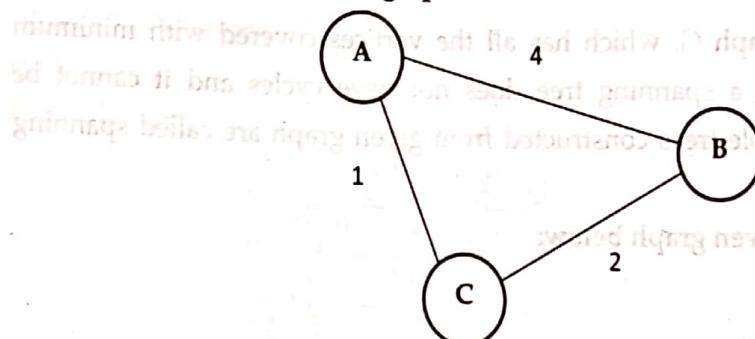
$$D^3(1, 3) = \text{unchanged} = 6$$

$$D^3(2, 1) = \min\{D^2(2, 1), D^2(2, 3) + D^2(3, 1)\} = \text{may change} = \min\{6, 2+3\} = 5$$

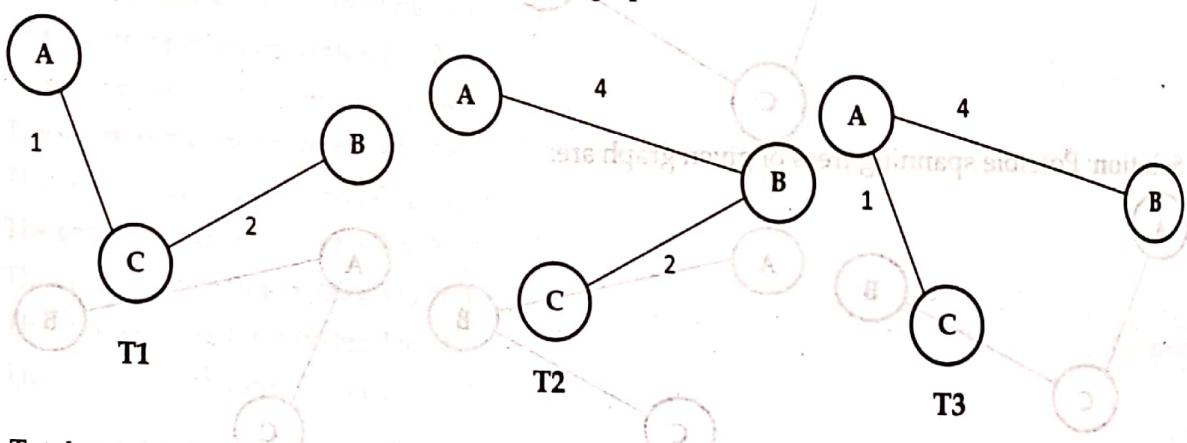
$$D^3(2, 2) = 0$$

$$D^3(2, 3) = \text{unchanged} = 2$$

Example: Find MST of given graph



Solution: The possible spanning trees of given graph are:



Total weight of $T_1 = 3$

Total weight of $T_2 = 6$

Total weight of $T_3 = 5$

Since T_1 has minimum weight out of all possible spanning trees of given graph hence, tree T_1 acts as minimum spanning tree of given graph.

KRUSKAL'S ALGORITHM

Kruskal's algorithm is a minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph which:

- form a tree that includes every vertex
- has the minimum sum of weights among all the trees that can be formed from the graph

It is the procedure for producing a minimum spanning tree of a given weighted graph that successively adds edges of least weight that are not already in the tree such that no edges produce a simple circuit when it is added.

Algorithm

1. Start
2. Sort all the edges from low weight to high
3. Take the edge with the lowest weight and add it to the spanning tree. If adding the edge created a cycle, then reject this edge.

Keep adding edges until we reach all vertices.

4. Stop

5. Pseudo code

Let G be a weighted connected undirected graph with n vertices)

KruskalMST(G)

$T = \{V\}$ // forest of n nodes

E = set of edges sorted in non-decreasing order of weight

while ($|T| < n-1$ and $E \neq \emptyset$)

{

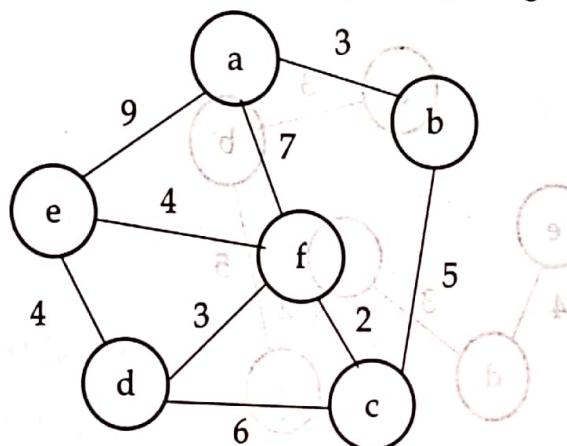
- Select (u, v) from E in order
- Remove (u, v) from E
- If $((u, v)$ does not create a cycle in T)
 $T = T \cup \{(u, v)\}$

}

Analysis

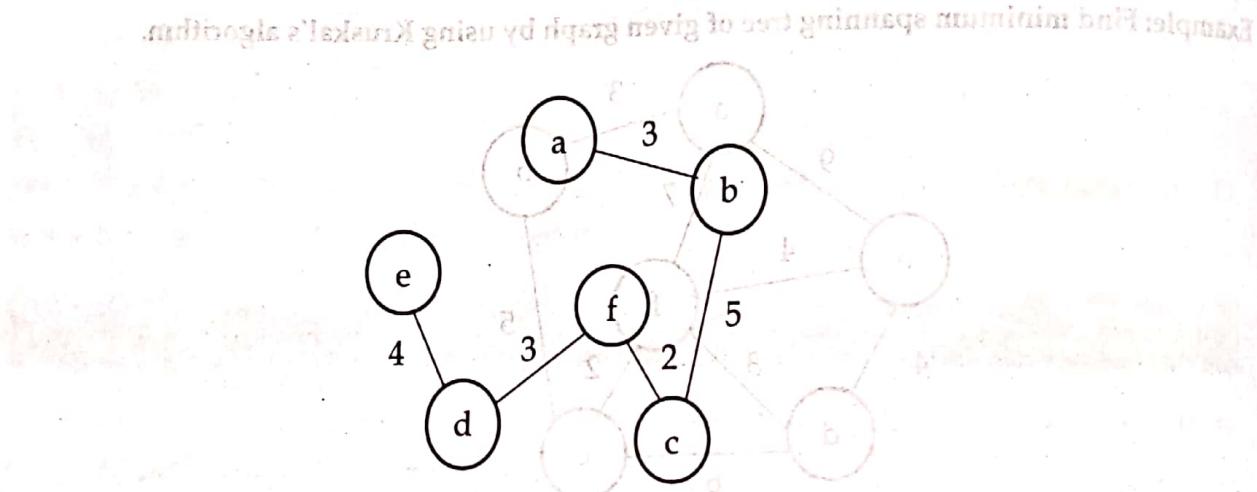
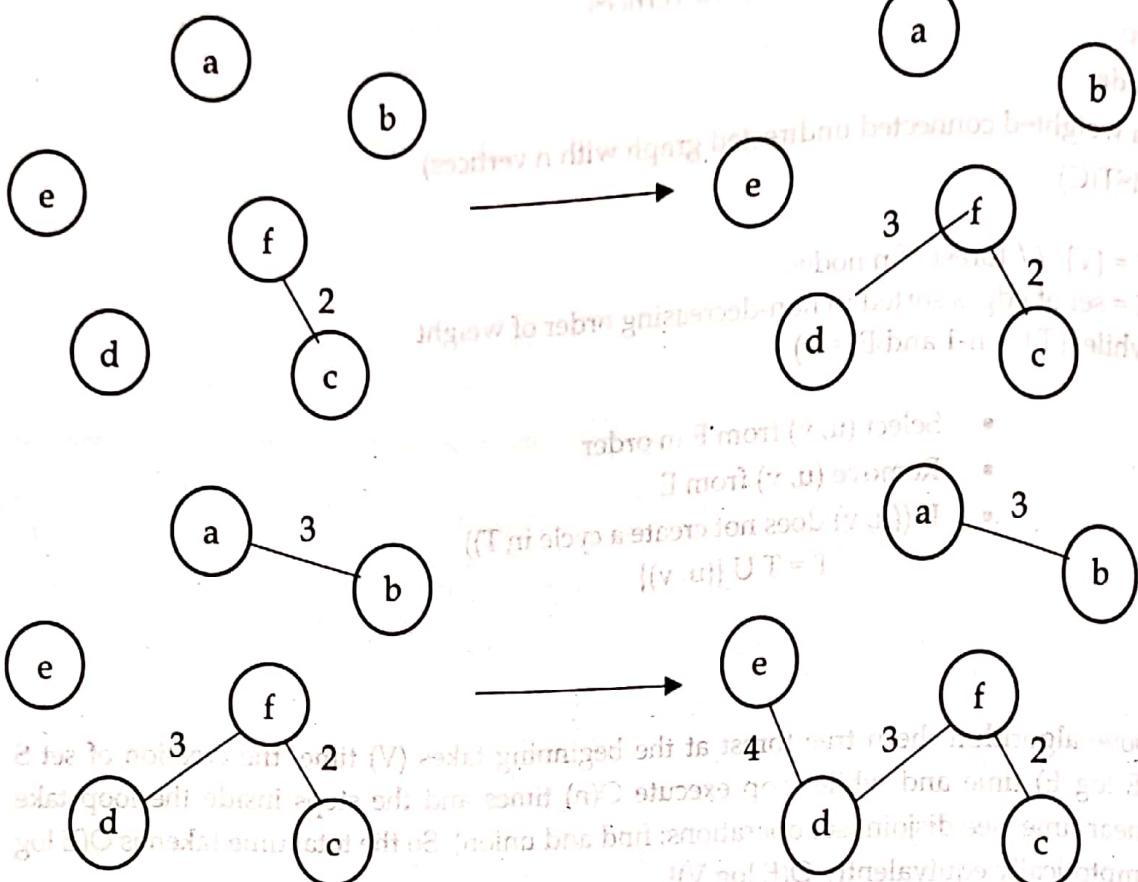
In the above algorithm the n tree forest at the beginning takes (V) time, the creation of set S takes $O(E \log E)$ time and while loop execute $O(n)$ times and the steps inside the loop take almost linear time (see disjoint set operations; find and union). So the total time taken is $O(E \log E)$ or asymptotically equivalently $O(E \log V)$!

Example: Find minimum spanning tree of given graph by using Kruskal's algorithm.



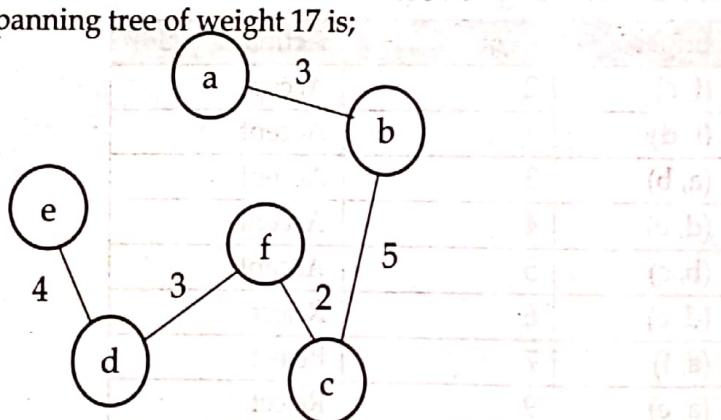
Solution: Edges in sorted order are given below:

Edges	Cost	Action
(f, c)	2	Accept
(f, d)	3	Accept
(a, b)	3	Accept
(d, e)	4	Accept
(b, c)	5	Accept
(d, c)	6	Reject
(a, f)	7	Reject
(a, e)	9	Reject



The edges (d, c), (a, f) and (a, e) forms cycle so discard these edges from the tree.

Thus the minimum spanning tree of weight 17 is;



Complete C program for Kruskal's algorithm to find Minimum Spanning Tree

```

#include <stdio.h>
#include <stdlib.h>
#include <string.h>
struct Edge
{
    int src;
    int dest;
    int weight;
};

struct Graph
{
    int V, E;
    struct Edge* edge;
};

Graph* createGraph(int V, int E)
{
    struct Graph* graph = (struct Graph*) malloc( sizeof(struct Graph) );
    graph->V = V;
    graph->E = E;
    graph->edge = (struct Edge*) malloc( graph->E * sizeof( struct Edge ) );
    return graph;
}

struct subset
{
    int parent;
    int rank;
};

int find(struct subset subsets[], int i)
{
    if (subsets[i].parent != i)
        subsets[i].parent = find(subsets, subsets[i].parent);
    return subsets[i].parent;
}

void Union(struct subset subsets[], int x, int y)
{
    int xroot = find(subsets, x);
    int yroot = find(subsets, y);
    if (subsets[xroot].rank < subsets[yroot].rank)
        subsets[xroot].parent = yroot;
    else if (subsets[xroot].rank > subsets[yroot].rank)
        subsets[yroot].parent = xroot;
}

```

```

    else
    {
        subsets[yroot].parent = xroot;
        subsets[xroot].rank++;
    }
}

int myComp(const void* a, const void* b)
{
    struct Edge* a1 = (struct Edge*)a;
    struct Edge* b1 = (struct Edge*)b;
    return a1->weight > b1->weight;
}

void KruskalMST(struct Graph* graph)
{
    int V = graph->V;
    struct Edge result[V];
    int e = 0;
    int i = 0;
    qsort(graph->edge, graph->E, sizeof(graph->edge[0]), myComp);
    struct subset *subsets = (struct subset*) malloc( V * sizeof(struct subset) );
    for (int v = 0; v < V; ++v)
    {
        subsets[v].parent = v;
        subsets[v].rank = 0;
    }
    while (e < V - 1)
    {
        struct Edge next_edge = graph->edge[i++];
        int x = find(subsets, next_edge.src);
        int y = find(subsets, next_edge.dest);
        if (x != y)
        {
            result[e++] = next_edge;
            Union(subsets, x, y);
        }
    }
    printf("Following are the edges in the constructed MST\n");
    for (i = 0; i < e; ++i)
    {
        printf("%d -- %d == %d\n", result[i].src, result[i].dest, result[i].weight);
    }
    return;
}

```

PRIM'S ALGORITHM

Prim's algorithm is a minimum spanning tree algorithm that takes a graph as input and finds the subset of the edges of that graph which:

- form a tree that includes every vertex
- has the minimum sum of weights among all the trees that can be formed from the graph

The idea behind this algorithm is just take any arbitrary vertex and choose the edge with minimum weight incident on the chosen vertex. Add the vertex and continue the above process until all the vertices are not added to the list. Remember the cycle must be avoided.

Algorithm

Start

1. Initialize the minimum spanning tree with a vertex chosen at random.
2. Find all the edges that connect the tree to new vertices, find the minimum and add it to the tree
3. Keep repeating step 3 until we get a minimum spanning tree

Stop

Pseudo code

PrimMST(G)

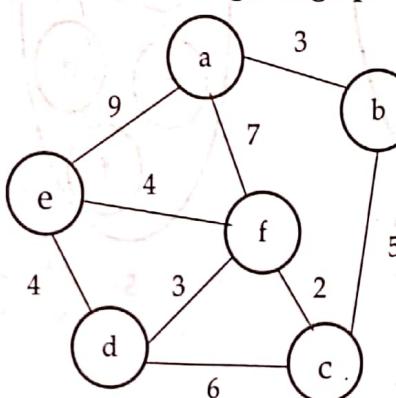
```

1   T =  $\Phi$ ;           // T is a set of edges of MST
2   S = {s} ;          // s is randomly chosen vertex and S is set of vertices
3   while ( $S \neq V$ )
4     {
5       e = (u, v) an edge of minimum weight incident to vertices in T and not forming
6       simple circuit in T if added to T i.e.  $u \in S$  and  $v \in V - S$ 
7       T =  $T \cup \{(u, v)\}$ ;
8       S =  $S \cup \{v\}$ ;
9     }
10  }
```

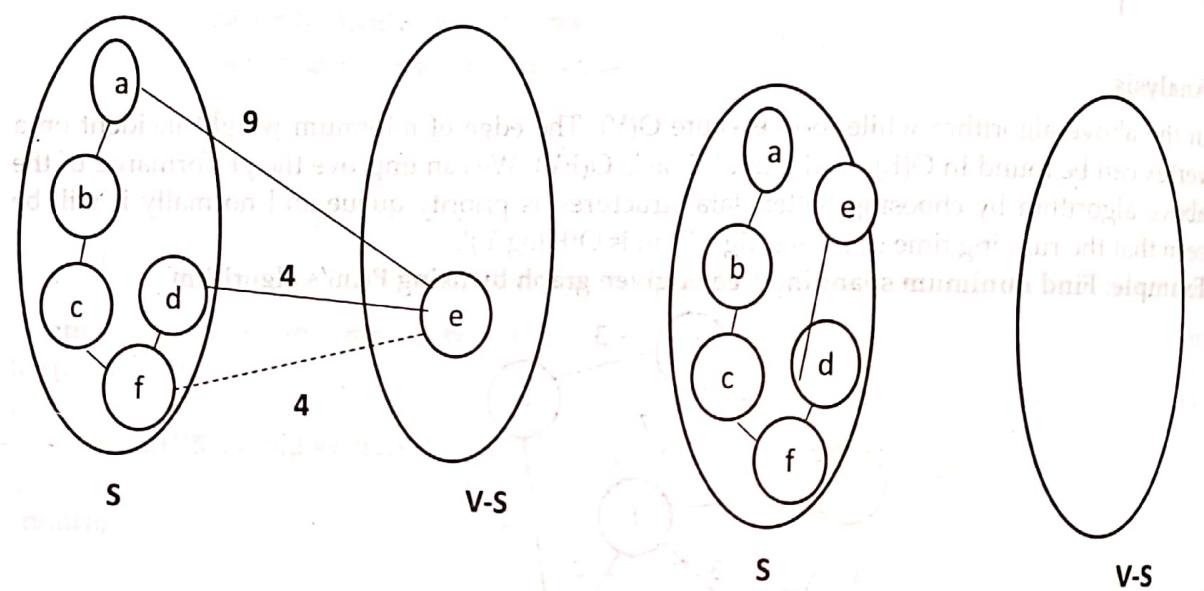
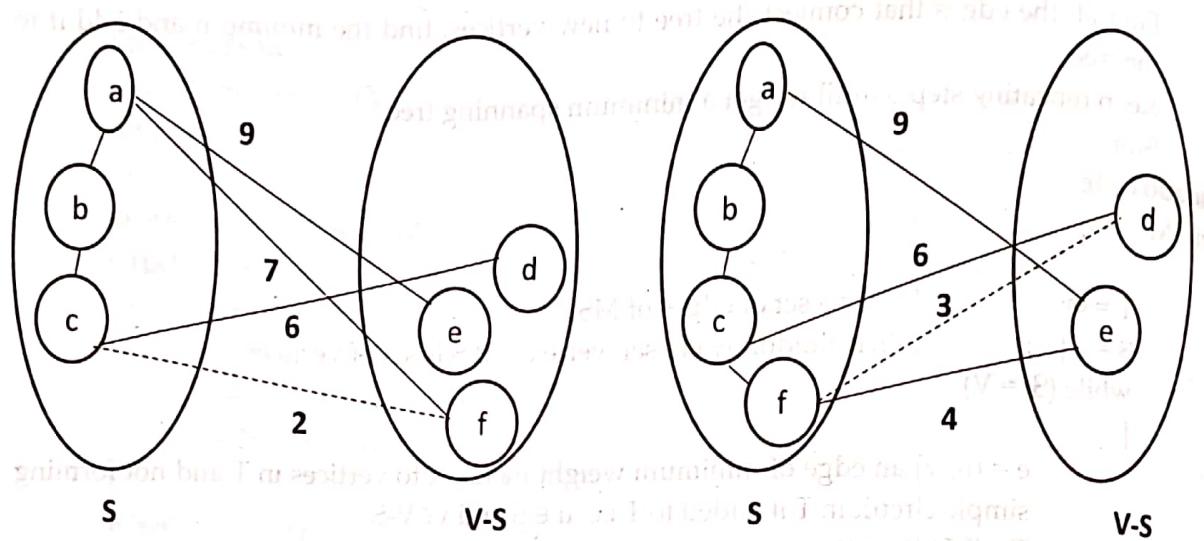
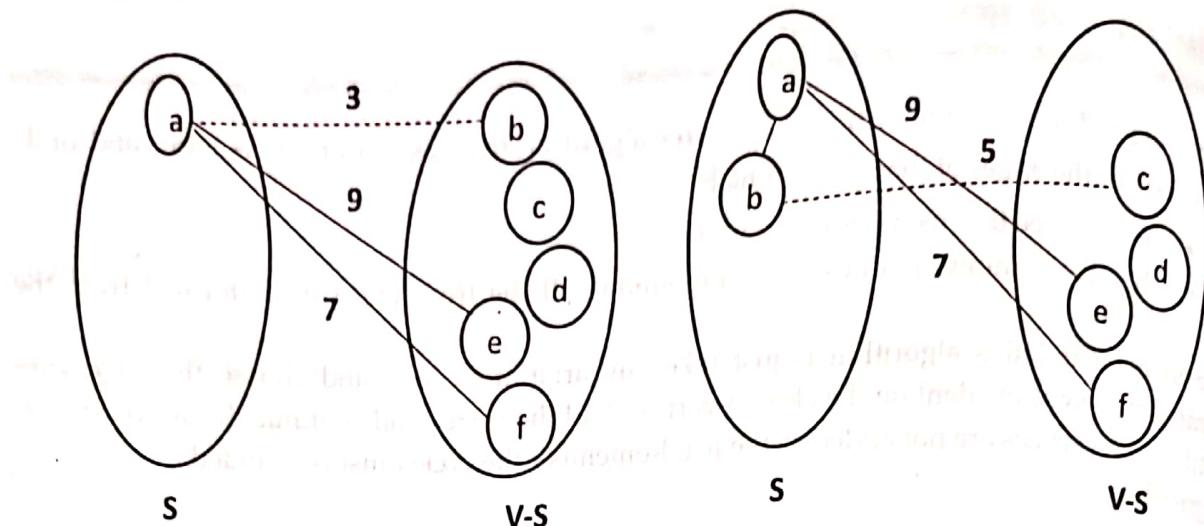
Analysis

In the above algorithm while loop execute $O(V)$. The edge of minimum weight incident on a vertex can be found in $O(E)$, so the total time is $O(EV)$. We can improve the performance of the above algorithm by choosing better data structures as priority queue and normally it will be seen that the running time of prim's algorithm is $O(E \log V)$!

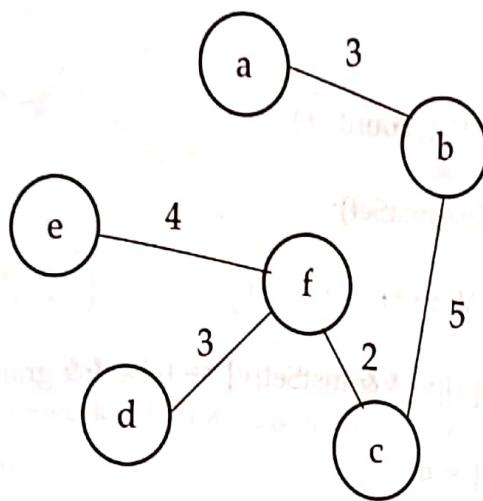
Example: Find minimum spanning tree of given graph by using Prim's algorithm



Solution:



Thus final MST given by Prim's algorithm is given below;



Thus the minimum spanning tree of weight 17 is shown in fig above.

Complete C program to find MST from given graph

```

#include <stdio.h>
#include <limits.h>
#define V 5
int minKey(int key[ ], bool mstSet[])
{
    int min = INT_MAX, min_index; //INT_MAX gives maximum value for int
    for (int v = 0; v < V; v++)
    {
        if (mstSet[v] == false && key[v] < min)
            min = key[v], min_index = v;
    }
    return min_index;
}
int printMST(int parent[ ], int n, int graph[V][V])
{
    printf("Edge  Weight\n");
    for (int i = 1; i < V; i++)
    {
        printf("%d - %d %d \n", parent[i], i, graph[i][parent[i]]);
    }
}
void primMST(int graph[V][V])
{
    int parent[V];
    int key[V];
    bool mstSet[V];
    for (int i = 0; i < V; i++)
    {
        key[i] = INT_MAX;
        mstSet[i] = false;
    }
}
    
```

```

    }
    key[0] = 0;
    parent[0] = -1;
    for (int count = 0; count < V-1; count++)
    {
        int u = minKey(key, mstSet);
        mstSet[u] = true;
        for (int v = 0; v < V; v++)
        {
            if (graph[u][v] && mstSet[v] == false && graph[u][v] < key[v])
            {
                parent[v] = u;
                key[v] = graph[u][v];
            }
        }
    }
    printMST(parent, V, graph);
}
int main()
{
    int graph[V][V] = {{0, 2, 0, 6, 0},
                        {2, 0, 3, 8, 5},
                        {0, 3, 0, 0, 7},
                        {6, 8, 0, 0, 9},
                        {0, 5, 7, 9, 0}};
    primMST(graph);
    return 0;
}

```

NETWORK FLOW PROBLEMS

Some real life problems like those involving the flow of liquids through pipe, current through wires, and delivery of goods can be modeling by using flow of networks.

Transport network

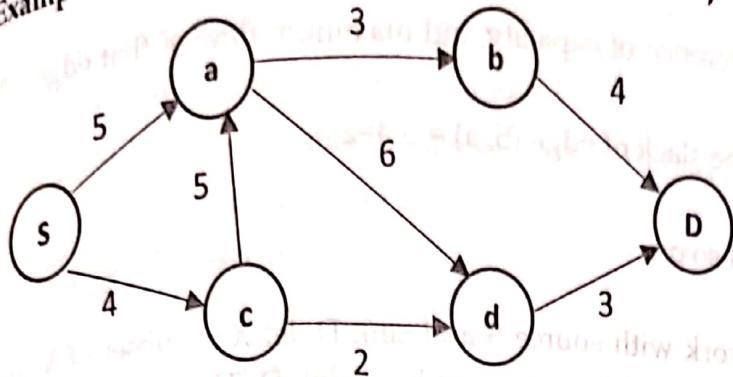
A flow network or transport network is a connected directed graph $G = (V, E)$ that does not contain any loops such that,

- There are exactly two distinguished vertices S and D , called the source and sink (destination) of graph G respectively
- There is a non-negative real valued function K defined on edge E called the capacity function of G .

Then (G, K) is called transport network and function K is called capacity function of G .

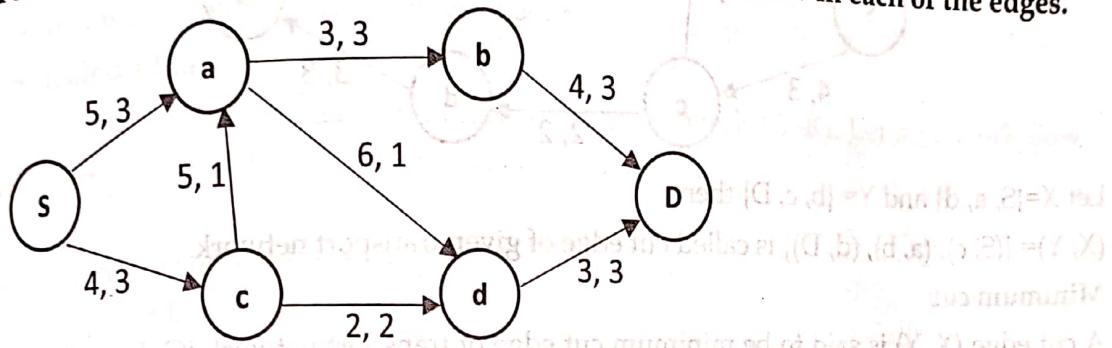
The vertices distinct for S and D are called intermediate vertices. If $e \in E$ then $K(e)$ is called capacity of e .

Example: Example of transport network is shown in fig below;



Flow
Let (G, K) be a transport network. Then a flow in G is a non-negative real valued function F defined on edge E such that,
 $0 \leq F(e) \leq K(e)$ for each edge e belongs to E . Where, $F(e)$ = flow to the edge e and $K(e)$ = maximum capacity to the edge e .

Example: Following transport network show that the capacity and flow in each of the edges.



From the above figure the flow-in of vertex 'a' is $F(S, a) + F(c, a) = 3+1=4$
 The flow out of vertex 'a' is $F(a, b)+F(a, d)=3+1=4$

The flow in $S=0$

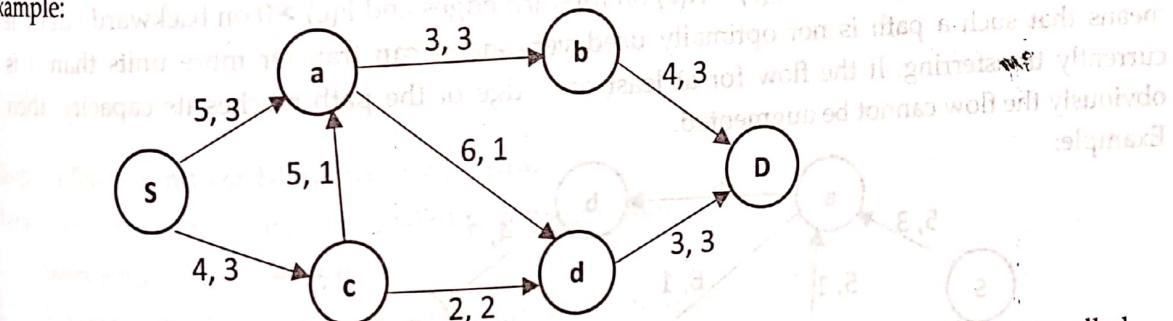
Flow in of vertex $D=F(b, D) + F(d, D)=3+3=6$ and so on

Note: Flow out of source vertex = flow in of sink vertex

Saturated and Unsaturated edges

The edges for which the flow and capacity are equal are called saturated edges. Otherwise they are called unsaturated edges.

Example:



Here edges (a, b) , (c, d) and (d, D) are called saturated edges. And remaining edges are called unsaturated edges.

Slack of edge

The slack of edge 'e' is the difference of capacity and maximum flow of that edge. The slack of each saturated edge is zero.

Example: In the above figure the slack of edge (S, a) = 5-3=2

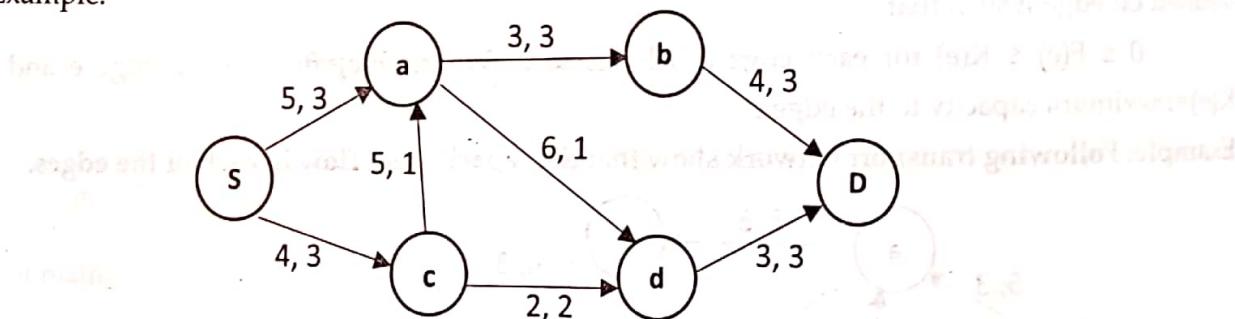
Slack of edge (a, b) = 3-3 = 0

Slack of edge (a, d) = 6-1=5 and so on

Cut

Let (G, K) is a transport network with source S and sink D. let X is subset of V and Y= V-X where X contains at least source S and Y contains at least sink D. Then the edge list (X, Y) are called cut edges of given network flow.

Example:



Let $X=\{S, a, d\}$ and $Y= \{b, c, D\}$ then

$(X, Y)= \{(S, c), (a, b), (d, D)\}$ is called cut edge of given transport network.

Minimum cut

A cut edge (X, Y) is said to be minimum cut edge of transport network (G, K) if there is no any cut edge (W, Z) that is less than (X, Y) . I.e. $K(W, Z) < K(X, Y)$ is false.

Example: In the above figure possible cut edges and their capacity are,

Case 1: let $X=\{S\}$ and $Y= \{a, b, c, d, D\}$ then

$$(X, Y)= \{(S, a), (S, c)\} = 5+4=9$$

Case 2: let $X=\{S, a\}$ and $Y= \{b, c, d, D\}$ then

$$(X, Y)= \{(S, c)\} = 4$$

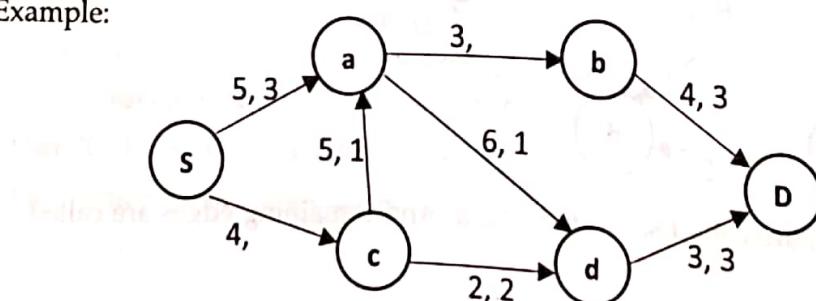
Case 3: let $X=\{S\}$ and $Y= \{a, b, c, d, D\}$ then

$$(X, Y)= \{(S, a), (S, c)\} = 5+4=9$$

F-augmenting path or flow augmenting path

A flow-augmenting path from source s to sink t is a sequence of edges from s to t such that, for each edge in this path, the flow $F(e) < K(e)$ on forward edges and $F(e) > 0$ on backward edges. It means that such a path is not optimally used yet, and it can transfer more units than it is currently transferring. If the flow for at least one edge of the path reaches its capacity, then obviously the flow cannot be augmented.

Example:



In the above figure the possible f-augmenting paths are: $\{S \rightarrow a \rightarrow b \rightarrow D\}$, $\{S \rightarrow c \rightarrow a \rightarrow b \rightarrow D\}$
 If we reach the sink t, the flows of the edges on the augmenting path that was just found are updated by increasing flows of forward edges and decreasing flows of backward edges, and the process restarts in the quest for another augmenting path. Here is a summary of the algorithm.
augmentPath (network with source s and sink t)
 for each edge e in the path from s to t

if forward(e)

$$f(e) += \text{slack}(t);$$

else

$$f(e) -= \text{slack}(t);$$

Maximum flow

A flow F in a network (G, K) is called a maximum flow if $|F'| \leq |F|$, for every flow F' in (G, K) . Simply a flow F in a network (G, K) is called maximum flow if the value of F is the largest possible value of any flow other possible flows in (G, K) .

Ford Fulkerson algorithm

This algorithm is used to find the maximum flow of given network (G, K) . Let a network flow diagram with source s and sink t.

1. Start
2. set flow of all edges and vertices to 0;
 Label = (null, ∞);
 Labeled = {s};
3. while labeled is not empty // while not stuck;
 - 3.1. Detach a vertex v from labeled;
 - 3.2. for all unlabeled vertices u adjacent to v
 - 3.2.1. if forward(edge(vu)) and slack(edge(vu)) > 0
 Label (u) = $(v+, \min(\text{slack}(v), \text{slack}(\text{edge}(vu))))$
 - 3.2.2. else if backward(edge(vu)) and $f(\text{edge}(uv)) > 0$
 Label (u) = $(v^-, \min(\text{slack}(v), f(\text{edge}(uv))))$;
 - 3.2.3. if u got labeled
 - 3.2.3.1. if $u == t$
 augmentPath (network);
 Labeled = {s}; // look for another path;
 - 3.2.3.2. else
 Include u in labeled;
4. Stop

Simple idea behind Ford-Fulkerson Algorithm

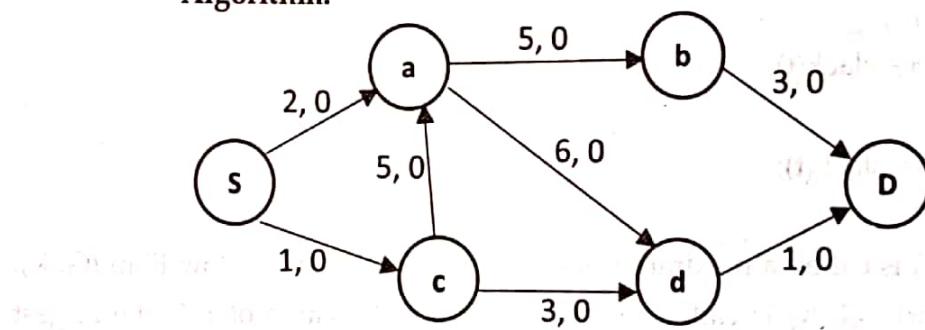
The following is simple idea of Ford-Fulkerson algorithm

1. Start with initial flow as 0
2. While there is 'a' augmenting path from source to sink.
 - Add this path-flow to flow
3. Return flow.

Time Complexity

Time complexity of the above algorithm is $O(\text{max_flow} * E)$. We run a loop while there is an augmenting path. In worst case, we may add 1 unit flow in every iteration. Therefore the time complexity becomes $O(\text{max_flow} * E)$.

Example 1: Find max flow of the following network flow graph by using Ford Fulkerson Algorithm.



Solution: At first listing f-augmenting paths as

$$\{S \rightarrow a \rightarrow b \rightarrow D\}, \{S \rightarrow c \rightarrow d \rightarrow D\}, \{S \rightarrow a \rightarrow d \rightarrow D\}, \{S \rightarrow a \rightarrow c \rightarrow d \rightarrow D\}, \{S \rightarrow c \rightarrow d \rightarrow a \rightarrow b \rightarrow D\}$$

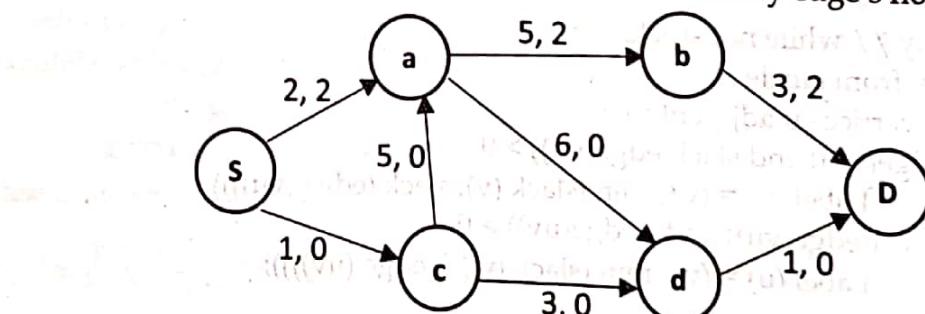
Step 1: In the f-augmenting path $\{S \rightarrow a \rightarrow b \rightarrow D\}$,

$$\text{Slack of edge } (S, a) = 2 - 0 = 2 \text{ (minimum)}$$

$$\text{Slack of edge } (a, b) = 5 - 0 = 5$$

$$\text{Slack of edge } (b, D) = 3 - 0 = 3$$

Since the minimum slack is 2 hence add 2 to every edge's flow of the path $\{S \rightarrow a \rightarrow b \rightarrow D\}$,



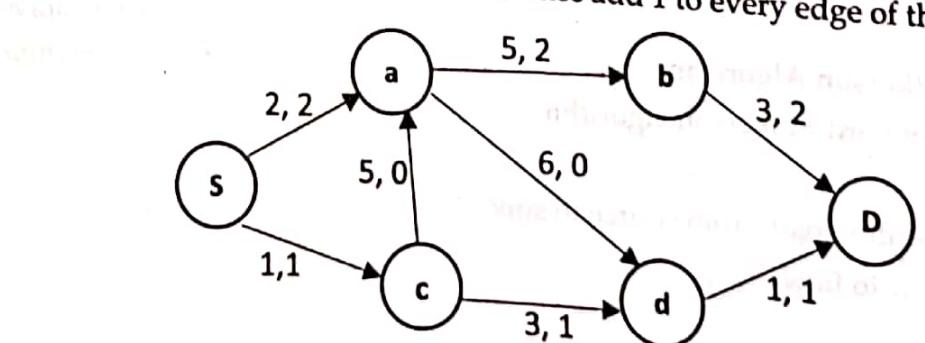
Step 2: In the f-augmenting path $\{S \rightarrow c \rightarrow d \rightarrow D\}$,

$$\text{Slack of edge } (S, c) = 1 - 0 = 1 \text{ (minimum)}$$

$$\text{Slack of edge } (c, d) = 3 - 0 = 3$$

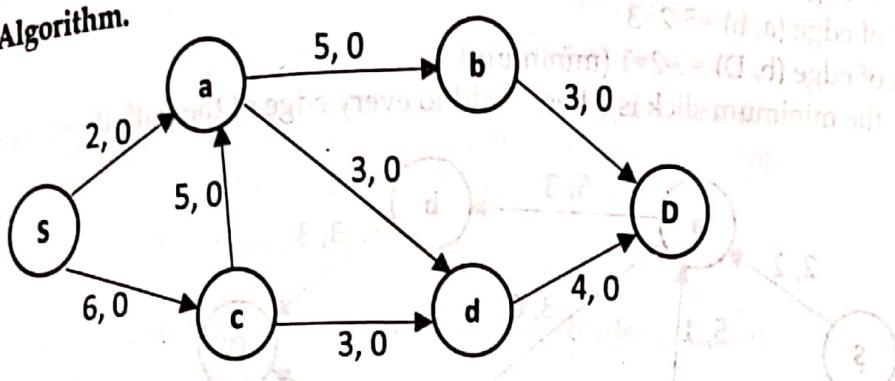
$$\text{Slack of edge } (d, D) = 1 - 0 = 1$$

Since the minimum slack is 1 hence add 1 to every edge of the path $\{S \rightarrow c \rightarrow d \rightarrow D\}$,



Now there is no any possible path from source to sink without saturated edge
 Hence maximum flow of given network graph is $2+1=3$
 Hence flow out from source = $2+1=3$ = flow in to the sink = $2+1=3$.

Example 2: Find max flow of the following network flow graph by using Ford Fulkerson Algorithm.



Solution: At first listing f-augmenting paths as

$\{S \rightarrow a \rightarrow b \rightarrow D\}, \{S \rightarrow c \rightarrow d \rightarrow D\}, \{S \rightarrow a \rightarrow d \rightarrow D\}, \{S \rightarrow a \rightarrow c \rightarrow d \rightarrow D\}, \{S \rightarrow c \rightarrow a \rightarrow b \rightarrow D\},$
 $\{S \rightarrow c \rightarrow a \rightarrow d \rightarrow D\}, \{S \rightarrow c \rightarrow d \rightarrow a \rightarrow b \rightarrow D\}$

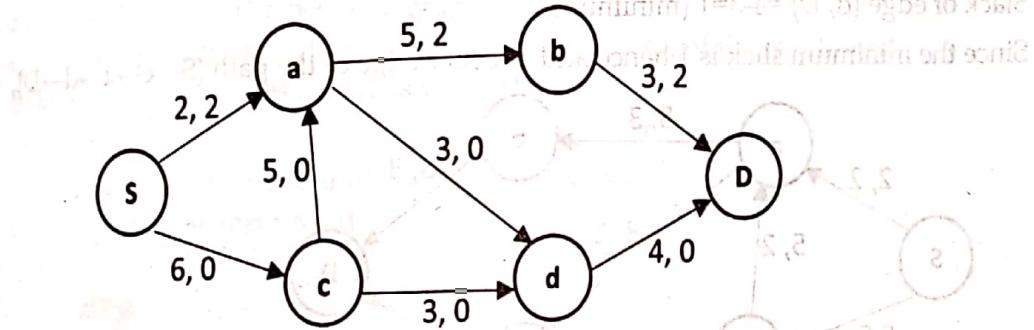
Step 1: In the f-augmenting path $\{S \rightarrow a \rightarrow b \rightarrow D\}$,

Slack of edge $(S, a) = 2 - 0 = 2$ (minimum)

Slack of edge $(a, b) = 5 - 0 = 5$

Slack of edge $(b, D) = 3 - 0 = 3$

Since the minimum slack is 2 hence add 2 to every edge's flow of the path $\{S \rightarrow a \rightarrow b \rightarrow D\}$,



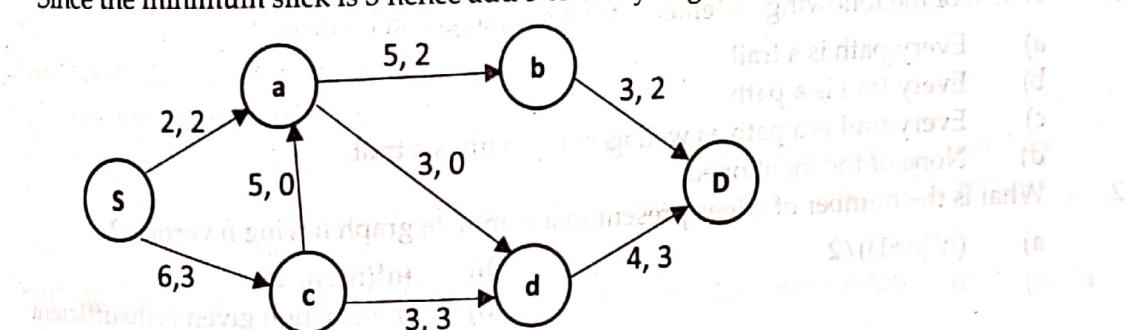
Step 2: In the f-augmenting path $\{S \rightarrow c \rightarrow d \rightarrow D\}$,

Slack of edge $(S, c) = 6 - 0 = 6$

Slack of edge $(c, d) = 3 - 0 = 3$ (minimum)

Slack of edge $(d, D) = 4 - 0 = 4$

Since the minimum slack is 3 hence add 3 to every edge of the path $\{S \rightarrow c \rightarrow d \rightarrow D\}$,



Step 3: In the f-augmenting path $\{S \rightarrow a \rightarrow d \rightarrow D\}$,

No change

Step 4: In the f-augmenting path $\{S \rightarrow a \rightarrow c \rightarrow d \rightarrow D\}$,

No change

Step 5: In the f-augmenting path $\{S \rightarrow c \rightarrow a \rightarrow b \rightarrow D\}$,

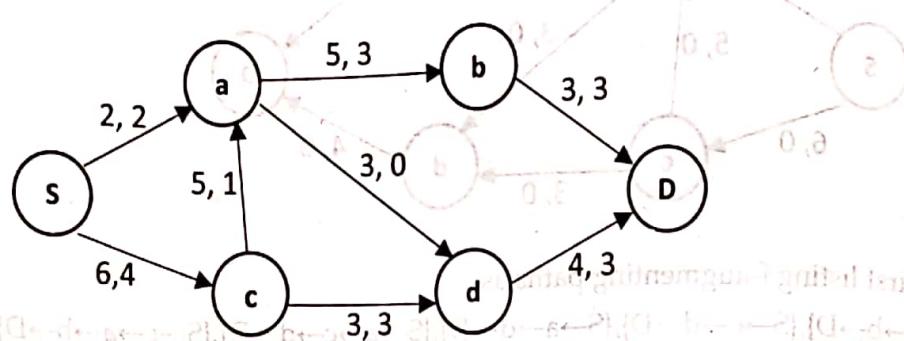
Slack of edge $(S, c) = 6 - 3 = 3$

Slack of edge $(c, a) = 5 - 0 = 5$

Slack of edge $(a, b) = 5 - 2 = 3$

Slack of edge $(b, D) = 3 - 2 = 1$ (minimum)

Since the minimum slack is 1 hence add to every edge of the path $\{S \rightarrow c \rightarrow a \rightarrow b \rightarrow D\}$,



Step 6: In the f-augmenting path $\{S \rightarrow c \rightarrow a \rightarrow d \rightarrow D\}$,

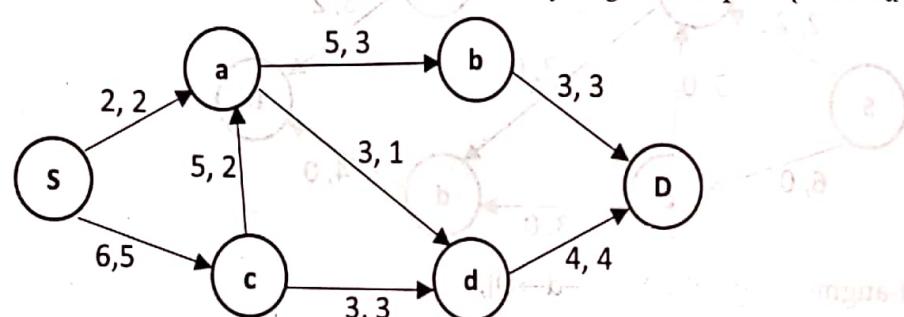
Slack of edge $(S, c) = 6 - 4 = 2$

Slack of edge $(c, a) = 5 - 1 = 4$

Slack of edge $(a, d) = 3 - 0 = 3$

Slack of edge $(d, D) = 4 - 3 = 1$ (minimum)

Since the minimum slack is 1 hence add to every edge of the path $\{S \rightarrow c \rightarrow a \rightarrow d \rightarrow D\}$,



MULTIPLE CHOICE QUESTIONS

1. Which of the following statements for a simple graph is correct?
 - Every path is a trail
 - Every trail is a path
 - Every trail is a path as well as every path is a trail
 - None of the mentioned
2. What is the number of edges present in a complete graph having n vertices?
 - $(n*(n+1))/2$
 - $(n*(n-1))/2$
 - n
 - Information given is insufficient

3. A connected planar graph having 6 vertices, 7 edges contains _____ regions.
- 15
 - 3
 - 1
 - 11
4. Which of the following properties does a simple graph not hold?
- Must be connected
 - Must be un-weighted
 - Must have no loops or multiple edges
 - All of the mentioned
5. A graph with all vertices having equal degree is known as a _____.
- Multi Graph
 - Regular Graph
 - Simple Graph
 - Complete Graph
6. Which of the following ways can be used to represent a graph?
- Adjacency List and Adjacency Matrix
 - Incidence Matrix
 - Adjacency List, Adjacency Matrix as well as Incidence Matrix
 - None of the mentioned
7. Breadth First Search is equivalent to which of the traversal in the Binary Trees?
- Pre-order Traversal
 - Post-order Traversal
 - Level-order Traversal
 - In-order Traversal
8. The Data structure used in standard implementation of Breadth First Search is?
- Stack
 - Queue
 - Linked List
 - None of the mentioned
9. What can be the applications of Breadth First Search?
- Finding shortest path between two nodes
 - Finding bi-partiteness of a graph
 - GPS navigation system
 - All of the mentioned
10. In BFS, how many times a node is visited?
- Once
 - Twice
 - Equivalent to number of in-degree of the node
 - None of the mentioned



DISCUSSION EXERCISE

- What is graph? How it is differ from tree? Show that a tree with n vertices has $n - 1$ edge.
- Which data structure is used by DFS and BFS?
- What are the application areas of graph? Explain.
- What is minimum spanning tree? How it is differ from maximum spanning tree? Draw a maximum spanning tree of a graph containing 5 vertices and 9 edges with arbitrary edge cost.
- What do you mean by representation of graph? Which representation of graph is more suitable to represent complete graph?

6. What is the relationship between the sum of the degrees of all vertices and the number of edges of graph $G = (V, E)$?
7. What is DAG? How it works? Explain with suitable example.
8. What is shortest path problem? Describe all pair shortest path with suitable example.
9. What is negative weight cycle graph? Describe with suitable example.
10. What is the complexity of breadth First Search algorithm?
11. Show that a simple graph is connected if it has a spanning tree.
12. What do you mean by topological sort? Explain.
13. What do you mean by all pair shortest path problem? Describe Floyd Warshall algorithm.
14. What do you mean by graph traversing? Which data structure is used to efficiently implement the DFS and BFS?
15. How can Dijkstra's Algorithm be applied to undirected graphs? Explain.
16. How can Dijkstra's Algorithm be modified to become an algorithm for finding the shortest path from vertex 'a' to 'b'?
17. What is the application of spanning tree? Draw a minimum spanning tree of a graph containing any 8 vertices and 11 edges with arbitrary edge costs.
18. What is network flow? Explain with suitable example.
19. Write down statement of max flow min cut theorem. And verify with suitable example.
20. What do you mean by all pair shortest path problem? Describe Floyd Warshall algorithm.

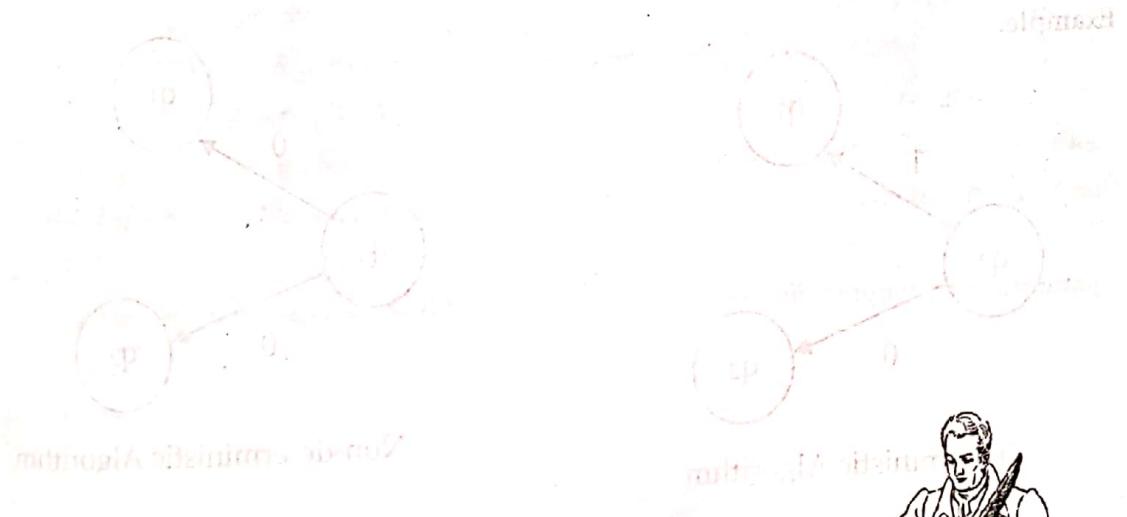


DISCUSSION EXERCISE



Algorithm is a step by step procedure to solve a problem. It is a well defined set of instructions which can be followed to get the solution. An algorithm is a finite sequence of well defined, unambiguous, ordered steps which must be followed to get the solution. An algorithm is a well defined set of steps which must be followed to get the solution. An algorithm is a well defined set of steps which must be followed to get the solution.

ALGORITHMS



CHAPTER OUTLINE

After studying this chapter, the reader will be able to understand the

- Deterministic and Non-deterministic algorithm, divide and conquer algorithm, series and parallel algorithm, heuristic and approximate algorithms.

DETERMINISTIC AND NON-DETERMINISTIC ALGORITHM

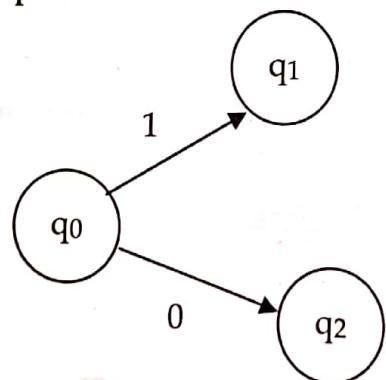
In computer science, a deterministic algorithm is an algorithm which, given a particular input, will always produce the same output, with the underlying machine always passing through the same sequence of states. A deterministic algorithm is an algorithm that is purely determined by its inputs, where no randomness is involved in the model. Deterministic algorithms will always come up with the same result given the same inputs.

A non-deterministic algorithm can provide different outputs for the same input on different executions. Unlike a deterministic algorithm which produces only a single output for the same input even on different runs, a non-deterministic algorithm travels in various routes to arrive at the different outcomes. Non-deterministic algorithms are useful for finding approximate solutions, when an exact solution is difficult or expensive to derive using a deterministic algorithm.

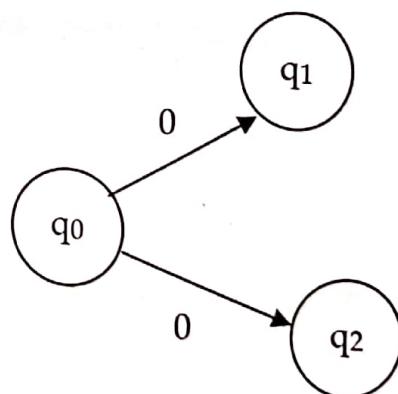
A non-deterministic algorithm is capable of execution on a deterministic computer which has an unlimited number of parallel processors. A non-deterministic algorithm usually has two phases and output steps. The first phase is the guessing phase, which makes use of arbitrary characters to run the problem.

The second phase is the verifying phase, which returns true or false for the chosen string. There are many problems which can be conceptualized with help of non-deterministic algorithms including the unresolved problem of P vs. NP in computing theory. Non-deterministic algorithms are used in solving problems which allow multiple outcomes. Every outcome the non-deterministic algorithm produces is valid, regardless of the choices made by the algorithm during execution.

Example:



Deterministic Algorithm



Non-deterministic Algorithm

Difference between Deterministic and Non-deterministic Algorithms	
Deterministic algorithm	Non-deterministic algorithm
For a particular input the computer will give always same output.	For a particular input the computer will give different output on different execution.
Can solve the problem in polynomial time.	Can't solve the problem in polynomial time.
Can determine the next step of execution.	Cannot determine the next step of execution due to more than one path the algorithm can take.
Backtracking is allow	Backtracking is not always allow

DIVIDE AND CONQUER ALGORITHM

Divide and conquer is an algorithm design paradigm based on multi-branched recursion. A divide-and-conquer algorithm works by recursively breaking down a problem into two or more sub-problems of the same or related type, until these become simple enough to be solved directly. The solutions to the sub-problems are then combined to give a solution to the original problem. Generally, divide-and-conquer algorithms have three parts:

- **Divide/Break:** This step involves breaking the problem into smaller sub-problems. Sub-problems should represent a part of the original problem. This step generally takes a recursive approach to divide the problem until no sub-problem is further divisible. At this stage, sub-problems become atomic in nature but still represent some part of the actual problem.
- **Conquer/Solve:** This step receives a lot of smaller sub-problems to be solved. Generally, at this level, the problems are considered 'solved' on their own.
- **Merge/Combine:** When the smaller sub-problems are solved, this stage recursively combines them until they formulate a solution of the original problem. This algorithmic approach works recursively and conquer & merge steps works so close that they appear as one.

Examples: The following computer algorithms are based on divide-and-conquer programming approach:

- Merge Sort
- Quick Sort
- Binary Search
- Strassen's Matrix Multiplication
- Closest pair (points)

There are various ways available to solve any computer problem, but the mentioned are a good example of divide and conquer approach.

Advantages of divide and conquer algorithm

- In a perfect world, where the problem is easy to divide, and the sub-problem at some level is easy to solve, divide and conquer can be optimal for a general case solution, like merge sort.
- Parallel availability, divide and conquer by its very nature lends itself well to parallel processing.

Disadvantages of divide and conquer algorithm

- Problem decomposition may be very complex and thus not really suitable to divide and conquer.
- Recursive nature of the solution may end up duplicating sub-problems, dynamic solutions may be better in some of these cases, like Fibonacci.
- Recursion into small/tiny base cases may lead to huge recursive stacks, and efficiency can be lost by not applying solutions earlier for larger base cases.

SERIES AND PARALLEL ALGORITHM

A sequential algorithm or serial algorithm is an algorithm that is executed sequentially once through, from start to finish, without other processing executing as opposed to concurrently or in parallel. The term is primarily used to contrast with concurrent algorithm or parallel algorithm; most standard computer algorithms are sequential algorithms, and not specifically identified as such, as sequentialness is a background assumption. Concurrency and parallelism are in general distinct concepts, but they often overlap - many distributed algorithms are both concurrent and parallel - and thus "sequential" is used to contrast with both, without distinguishing which one. If these need to be distinguished, the opposing pairs sequential/concurrent and serial/parallel may be used.

In real time example, people standing in a queue and waiting for a railway ticket. In this case, one person can get a ticket at a time. Suppose there are two queues of people and one cashier is handling both the queues then one person can get a ticket at a time from both queues. Similarly, processor gets lists of tasks and each task is completed at a time and all other tasks wait till the first one completes. This type of processing is also known as sequential processing.

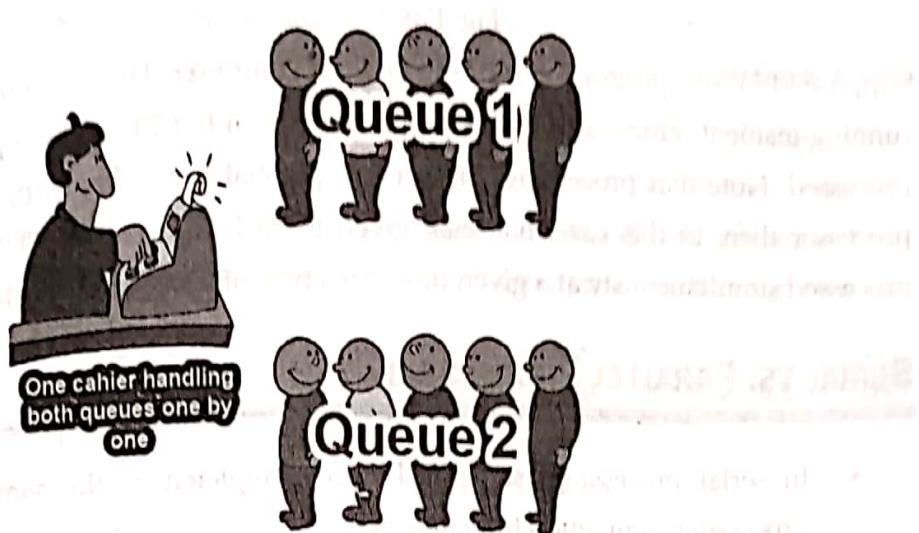


Fig: Sequential Processing

PARALLEL ALGORITHM

A parallel algorithm, as opposed to a traditional serial algorithm, is an algorithm which can do multiple operations in a given time. It has been a tradition of computer science to describe serial algorithms in abstract machine models, often the one known as Random-access machine. The problem is divided into sub-problems and is executed in parallel to get individual outputs. Later on, these individual outputs are combined together to get the final desired output.

A parallel algorithm can be executed simultaneously on many different processing devices and then combined together to get the correct result. Parallel algorithms are highly useful in processing huge volumes of data in quick time. In real time example, there are multiple queues of people standing to get railway tickets. In this case, each queue is handled by multiple people, so multiple people will get tickets at a time. Similarly, in the operating system, there are multiple queues of tasks and multiple tasks are completed by different processors at a time.

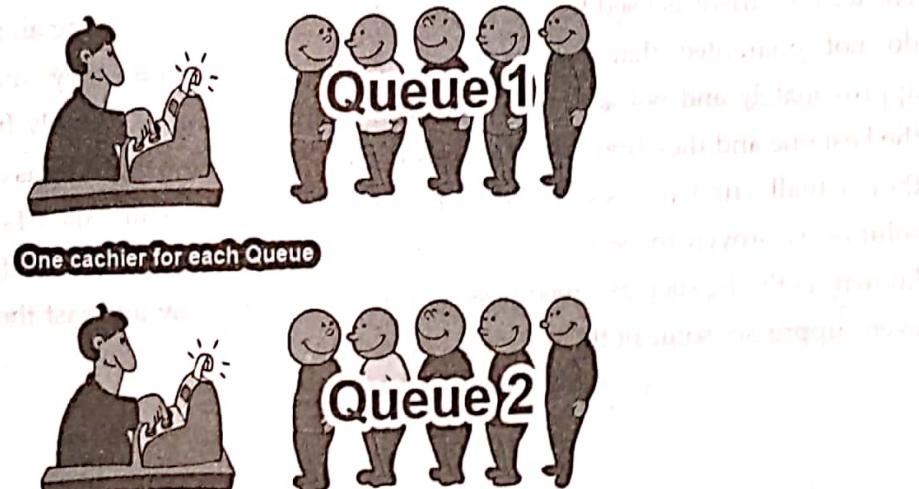


Fig: Parallel processing

Suppose MS Word program is running on your computer. This program may have several tasks running inside it. First Task1 of the program is given to CPU and get processed then Task2 get processed. Note that processing goes in a sequential way. Now suppose you have multi-core processor then, in this case, one task given to each processor (CPU) and all three tasks get processed simultaneously at a given time. This type of processing is called parallel processing.

SERIAL VS. PARALLEL PROCESSING

- In serial processing, same tasks are completed at the same time but in parallel processing completion time may vary.
- In sequential processing, the load is high on single core processor and processor heats up quickly.
- In serial processing data transfers in bit by bit form while In parallel processing data transfers in byte form i.e. in 8 bits form
- Parallel processor is costly as compared to serial processor
- Serial processing takes more time than parallel processor

HEURISTIC AND APPROXIMATE ALGORITHMS

A heuristic algorithm is one that is designed to solve a problem in a faster and more efficient fashion than traditional methods by sacrificing optimality, accuracy, precision, or completeness for speed. Heuristic algorithms often times used to solve NP-complete problems, a class of decision problems. In these problems, there is no known efficient way to find a solution quickly and accurately although solutions can be verified when given. Heuristics can produce a solution individually or be used to provide a good baseline and are supplemented with optimization algorithms. Heuristic algorithms are most often employed when approximate solutions are sufficient and exact solutions are necessarily computationally expensive.

The term heuristic is used for algorithms which find solutions among all possible ones, but they do not guarantee that the best will be found, therefore they may be considered as approximately and not accurate algorithms. These algorithms, usually find a solution close to the best one and they find it fast and easily. Sometimes these algorithms can be accurate, that is they actually find the best solution, but the algorithm is still called heuristic until this best solution is proven to be the best. The method used from a heuristic algorithm is one of the known methods, such as greediness, but in order to be easy and fast the algorithm ignores or even suppresses some of the problem's demands.

APPROXIMATE ALGORITHMS

An Approximate Algorithm is a way of approach NP-Completeness for the optimization problem. This technique does not guarantee the best solution. The goal of an approximation algorithm is to come as close as possible to the optimum value in a reasonable amount of time which is at the most polynomial time. Such algorithms are called approximation algorithm or heuristic algorithm.

- For the traveling salesperson problem, the optimization problem is to find the shortest cycle, and the approximation problem is to find a short cycle.
- For the vertex cover problem, the optimization problem is to find the vertex cover with fewest vertices, and the approximation problem is to find the vertex cover with few vertices.

QUESTION EXERCISE

MULTIPLE CHOICE QUESTIONS

- The indirect change of the values of a variable in one module by another module is called
 - Internal change
 - Inter-module change
 - Side effect
 - Side-module update
- Approach of dynamic programming is similar to
 - Parsing
 - Hash table
 - Divide and Conquer algorithm
 - Greedy algorithm
- Algorithms like merge sort, quick sort and binary search are based on
 - Greedy algorithm
 - Divide and Conquer algorithm
 - Hash table
 - Parsing
- Sub-problems in dynamic programming are solved
 - Dependently
 - Independently
 - Parallel
 - Concurrent
- In Divide and Conquer process, breaking problem into smaller sub-problems is responsibility of
 - Divide/Break
 - Sorting/Divide
 - Conquer/Solve
 - Merge/Combine
- The disadvantage of using a parallel mode of communication is _____
 - It is costly
 - Leads to erroneous data transfer
 - Security of data
 - All of the mentioned
- In the analysis of algorithms, what plays an important role?
 - Text Analysis
 - Growth factor
 - Time
 - None of the above
- The time factor when determining the efficiency of algorithm is measured by counting
 - Microseconds
 - The number of key operations
 - The number of statements
 - the kilobytes of algorithm

9. The space factor when determining the efficiency of algorithm is measured by counting the
- Maximum memory needed by the algorithm
 - Minimum memory needed by the algorithm
 - Average memory needed by the algorithm
 - Maximum disk space needed by the algorithm
10. Let there be an array of length 'N', and the selection sort algorithm is used to sort it, how many times a swap function is called to complete the execution?
- $N \log N$ times
 - $\log N$ times
 - N^2 times
 - $N-1$ times



DISCUSSION EXERCISE

- Compare divide and conquer algorithm and recursive algorithm.
- What is deterministic algorithm? How it is differ from non-deterministic algorithm? Explain.
- What is approximations algorithm? Explain with suitable example.
- What is heuristic algorithm? How it is differ from approximation algorithm? Explain.
- List out possible divide and conquer algorithm. Explain any two divide and conquer algorithms with suitable example.
- What is serial algorithm? How it is differ from parallel algorithm? Explain.
- Write down the advantages and disadvantages of Divide and conquer algorithm.
- Write down the advantages of approximation algorithm.
- What is parallel algorithm? Write down advantages of parallel algorithms.
- Explain the properties of an algorithm with an example.
- Write the General method of Divide - And - Conquer approach.
- Explain Reliability Design Problem with suitable example.
- Write Control Abstraction of Divide - and - Conquer.
- Distinguish between Divide and conquer and Greedy method.
- Explain recursive functions algorithm analysis with an example.
- Explain the method of determining the complexity of procedure by the step count approach. Illustrate with an example.
- Do you know any profilers that show parallelizable regions and non-parallelizable regions?
- How can we compute time complexity for parallel algorithms in form of asymptotic notations?
- How do we convert the serial algorithm to a parallel algorithm?
- List out the possible types of algorithms used in DSA.

□□□

TRIBHUVAN UNIVERSITY

Faculty of Humanities & Social Sciences

Office of the Dean

2019

Bachelor in Computer Applications

Course Title: Date Structure & Algorithms

Code No: CACS 201

Semester: III

Candidates are required to answers all the questions in their own words as far as practicable. Figures in brackets indicate full marks.

Full Marks: 60

Pass Marks: 24

Time: 3 hours

Group "B"

Attempt any Six Questions.

$6 \times 5 = 30$

2. What is Data Structure ? Show the status of stack converting following infix expression to prefix P + Q (R*S/T+U)-V*W. 1+4
3. Write binary search. Consider a hash table of size 10; insert the keys 62, 37, 36, 44, 67, 91 and 107 using linear probing. 2+3
4. What are determinants and non-deterministic algorithms ? Explain greedy algorithm. 3+2
5. Draw a BST from the string DATASTRUCTURE and traverse the tree in post order and preorder. 3+2
6. Define circular queue ? How does circular queue overcome the limitation of linear queue ? Explain. 2+3
7. What is singly linked list ? Write an algorithm to add a node at the beginning and end of singly linked list. 1+4
8. Define AVL tree. Construct AVL tree from given daa set: 4, 6, 12, 9, 5, 2, 13, 8, 3, 7, 11. 2+3

Group "C"

Attempt any two Questions.

$2 \times 10 = 20$

9. What is stack ? List the applications of stack. Write an algorithm or procedure to perform PUSH and POP operation in stack. 1+2+7
10. What is heap ? Explain quick sort algorithm with Big-Oh notation in best case, average case and worst case and trace it to sort the data: 8, 10, 5, 12, 14, 5, ,7, 13 2+2+6
11. Define graph and tree data structure. Explain breadth first traversal and depth first traversal with example. 4+6

