

# Unit 1: Chapter 1

## Introduction to Data Structures and Algorithms

- 1.0 Objective
- 1.1 Introduction:
- 1.2 Basic Concepts of Data Structures
  - 1.2.1 Basic Terminology
  - 1.2.2 Need for Data Structures
  - 1.2.3 Goals of Data Structure
  - 1.2.4 Features of Data Structure
- 1.3 Classification of Data Structures
- 1.4 Static Data Structure vs Dynamic Data Structure
- 1.5 Operations on Data Structures
- 1.6 Abstract Data Type
- 1.7 Algorithms
- 1.8 Algorithm Complexity
  - 1.8.1 Time Complexity
  - 1.8.2 Space Complexity
- 1.9 Algorithmic Analysis
  - 1.7.1 Worst-case
  - 1.7.2 Average-case
  - 1.7.3 Best-case
- 1.10 Mathematical Notation
  - 1.10.1 Asymptotic
  - 1.10.2 Asymptotic Notations
    - 1.10.2.1 Big-Oh Notation ( $O$ )
    - 1.10.2.2 Big-Omega Notation ( $\Omega$ )
    - 1.10.2.3 Big-Theta Notation ( $\Theta$ )
- 1.11 Algorithm Design technique
  - 1.11.1 Divide and Conquer
  - 1.11.2 Back Tracking Method
  - 1.11.3 Dynamic programming
- 1.12 Summary
- 1.13 Model Questions
- 1.14 List of References

## **1.0 OBJECTIVE**

After studying this unit, you will be able to:

- Discuss the concept of data structure
- Discuss the need for data structures
- Explain the classification of data structures
- Discuss abstract data types
- Discuss various operations on data structures
- Explain algorithm complexity
- Understand the basic concepts and notations of data structures

## **1.1 INTRODUCTION**

The study of data structures helps to understand the basic concepts involved in organizing and storing data as well as the relationship among the data sets. This in turn helps to determine the way information is stored, retrieved and modified in a computer's memory.

## **1.2 BASIC CONCEPT OF DATA STRUCTURE**

Data structure is a branch of computer science. The study of data structure helps you to understand how data is organized and how data flow is managed to increase efficiency of any process or program. Data structure is the structural representation of logical relationship between data elements. This means that a data structure organizes data items based on the relationship between the data elements.

Example:

A house can be identified by the house name, location, number of floors and so on. These structured set of variables depend on each other to identify the exact house. Similarly, data structure is a structured set of variables that are linked to each other, which forms the basic component of a system

### **1.2.1 Basic Terminology**

Data structures are the building blocks of any program or the software. Choosing the appropriate data structure for a program is the most difficult task for a programmer. Following terminology is used as far as data structures are concerned

**Data:** Data can be defined as an elementary value or the collection of values, for example, student's name and its id are the data about the student.

**Group Items:** Data items which have subordinate data items are called Group item, for example, name of a student can have first name and the last name.

**Record:** Record can be defined as the collection of various data items, for example, if we talk about the student entity, then its name, address, course and marks can be grouped together to form the record for the student.

**File:** A File is a collection of various records of one type of entity, for example, if there are 60 employees in the class, then there will be 20 records in the related file where each record contains the data about each employee.

**Attribute and Entity:** An entity represents the class of certain objects. it contains various attributes. Each attribute represents the particular property of that entity.

**Field:** Field is a single elementary unit of information representing the attribute of an entity.

### 1.2.2 Need for Data Structure

- It gives different level of organization data.
- It tells how data can be stored and accessed in its elementary level.
- Provide operation on group of data, such as adding an item, looking up highest priority item.
- Provide a means to manage huge amount of data efficiently.
- Provide fast searching and sorting of data.

### 1.2.3 Goals of Data Structure

Data structure basically implements two complementary goals.

**Correctness:** Data structure is designed such that it operates correctly for all kinds of input, which is based on the domain of interest. In other words, correctness forms the primary goal of data structure, which always depends on the specific problems that the data structure is intended to solve.

**Efficiency:** Data structure also needs to be efficient. It should process the data at high speed without utilizing much of the computer resources such as memory space. In a real time state, the efficiency of a data structure is an important factor that determines the success and failure of the process.

### 1.2.4 Features of Data Structure

Some of the important features of data structures are:

**Robustness:** Generally, all computer programmers wish to produce software that generates correct output for every possible input provided to it, as well as execute

efficiently on all hardware platforms. This kind of robust software must be able to manage both valid and invalid inputs.

**Adaptability:** Developing software projects such as word processors, Web browsers and Internet search engine involves large software systems that work or execute correctly and efficiently for many years. Moreover, software evolves due to ever changing market conditions or due to emerging technologies.

**Reusability:** Reusability and adaptability go hand-in-hand.

It is a known fact that the programmer requires many resources for developing any software, which makes it an expensive enterprise. However, if the software is developed in a reusable and adaptable way, then it can be implemented in most of the future applications. Thus, by implementing quality data structures, it is possible to develop reusable software, which tends to be cost effective and time saving.

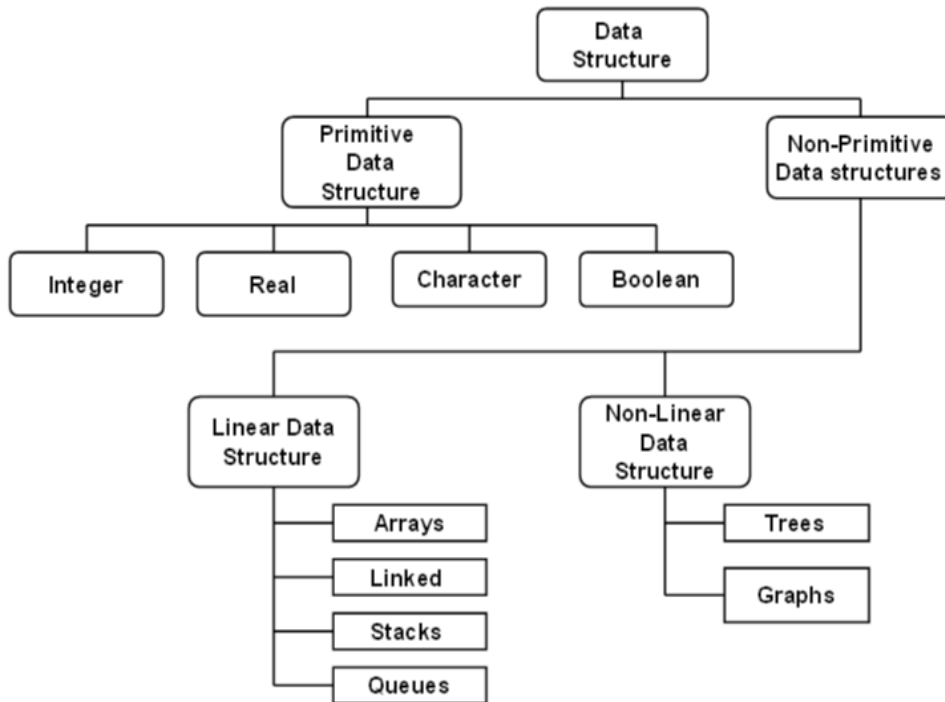
### **1.3 CLASSIFICATION OF DATA STRUCTURES**

A data structure provides a structured set of variables that are associated with each other in different ways. It forms a basis of programming tool that represents the relationship between data elements and helps programmers to process the data easily.

Data structure can be classified into two categories:

- 1.3.1 Primitive data structure
- 1.3.2 Non-primitive data structure

Figure 1.1 shows the different classifications of data structures.



**Figure 1.1 Classifications of data structures.**

### 1.3.1 Primitive Data Structure

Primitive data structures consist of the numbers and the characters which are built in programs. These can be manipulated or operated directly by the machine level instructions. Basic data types such as integer, real, character, and Boolean come under primitive data structures. These data types are also known as simple data types because they consist of characters that cannot be divided.

### 1.3.2 Non-primitive Data Structure

Non-primitive data structures are those that are derived from primitive data structures. These data structures cannot be operated or manipulated directly by the machine level instructions. They focus on formation of a set of data elements that is either homogeneous (same data type) or heterogeneous (different data type).

These are further divided into linear and non-linear data structure based on the structure and arrangement of data.

#### 1.3.2.1 Linear Data Structure

A data structure that maintains a linear relationship among its elements is called a linear data structure. Here, the data is arranged in a linear fashion. But in the memory, the arrangement may not be sequential.

Ex: Arrays, linked lists, stacks, queues.

### 1.3.2.1 Non-linear Data Structure

Non-linear data structure is a kind of data structure in which data elements are not arranged in a sequential order. There is a hierarchical relationship between individual data items. Here, the insertion and deletion of data is not possible in a linear fashion. Trees and graphs are examples of non-linear data structures.

### D) Array

Array, in general, refers to an orderly arrangement of data elements. Array is a type of data structure that stores data elements in adjacent locations. Array is considered as linear data structure that stores elements of same data types. Hence, it is also called as a linear homogenous data structure.

When we declare an array, we can assign initial values to each of its elements by enclosing the values in braces { }.

```
int Num [5] = { 26, 7, 67, 50, 66 };
```

This declaration will create an array as shown below:

	0	1	2	3	4
Num	26	7	67	50	66

**Figure 1.2 Array**

The number of values inside braces { } should be equal to the number of elements that we declare for the array inside the square brackets [ ]. In the example of array Paul, we have declared 5 elements and in the list of initial values within braces { } we have specified 5 values, one for each element. After this declaration, array Paul will have five integers, as we have provided 5 initialization values.

Arrays can be classified as one-dimensional array, two-dimensional array or multidimensional array.

**One-dimensional Array:** It has only one row of elements. It is stored in ascending storage location.

**Two-dimensional Array:** It consists of multiple rows and columns of data elements. It is also called as a matrix.

**Multidimensional Array:** Multidimensional arrays can be defined as array of arrays. Multidimensional arrays are not bounded to two indices or two dimensions. They can include as many indices as required.

### **Limitations:**

- Arrays are of fixed size.

- Data elements are stored in contiguous memory locations which may not be always available.
- Insertion and deletion of elements can be problematic because of shifting of elements from their positions.

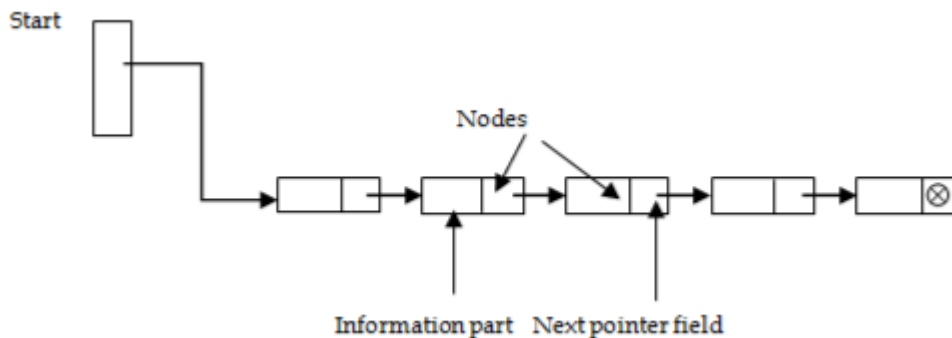
However, these limitations can be solved by using linked lists.

### Applications:

- Storing list of data elements belonging to same data type
- Auxiliary storage for other data structures
- Storage of binary tree elements of fixed count
- Storage of matrices

## II) Linked List

A linked list is a data structure in which each data element contains a pointer or link to the next element in the list. Through linked list, insertion and deletion of the data element is possible at all places of a linear list. Also in linked list, it is not necessary to have the data elements stored in consecutive locations. It allocates space for each data item in its own block of memory. Thus, a linked list is considered as a chain of data elements or records called nodes. Each node in the list contains information field and a pointer field. The information field contains the actual data and the pointer field contains address of the subsequent nodes in the list.



**Figure 1.3: A Linked List**

Figure 1.3 represents a linked list with 4 nodes. Each node has two parts. The left part in the node represents the information part which contains an entire record of data items and the right part represents the pointer to the next node. The pointer of the last node contains a null pointer.

**Advantage:** Easier to insert or delete data elements

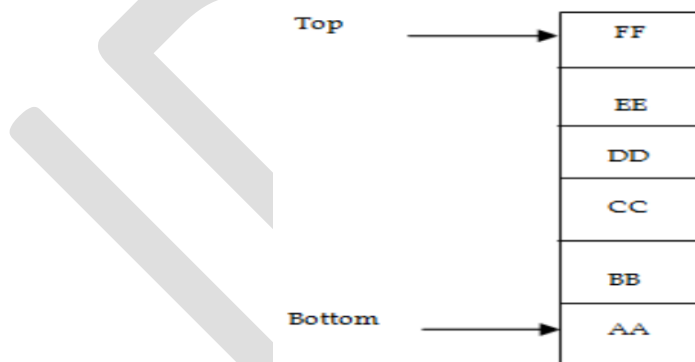
**Disadvantage:** Slow search operation and requires more memory space

**Applications:**

- Implementing stacks, queues, binary trees and graphs of predefined size.
- Implement dynamic memory management functions of operating system.
- Polynomial implementation for mathematical operations
- Circular linked list is used to implement OS or application functions that require round robin execution of tasks.
- Circular linked list is used in a slide show where a user wants to go back to the first slide after last slide is displayed.
- Doubly linked list is used in the implementation of forward and backward buttons in a browser to move backwards and forward in the opened pages of a website.
- Circular queue is used to maintain the playing sequence of multiple players in a game.

**III) Stacks**

A stack is a linear data structure in which insertion and deletion of elements are done at only one end, which is known as the top of the stack. Stack is called a last-in, first-out (LIFO) structure because the last element which is added to the stack is the first element which is deleted from the stack.



**Figure 1.4: A Stack**

In the computer's memory, stacks can be implemented using arrays or linked lists. Figure 1.4 is a schematic diagram of a stack. Here, element FF is the top of the stack and element AA is the bottom of the stack. Elements are added to the stack from the top. Since it follows LIFO pattern, EE cannot be deleted before FF is deleted, and similarly DD cannot be deleted before EE is deleted and so on.

**Applications:**

- Temporary storage structure for recursive operations

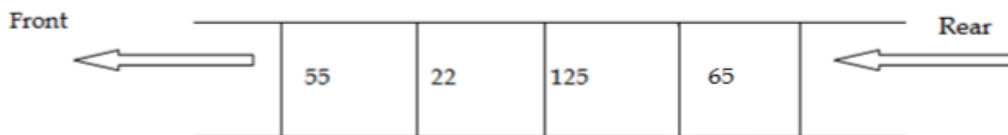


- Auxiliary storage structure for nested operations, function calls, deferred/postponed functions
- Manage function calls
- Evaluation of arithmetic expressions in various programming languages
- Conversion of infix expressions into postfix expressions
- Checking syntax of expressions in a programming environment
- Matching of parenthesis
- String reversal
- In all the problems solutions based on backtracking.
- Used in depth first search in graph and tree traversal.
- Operating System functions
- UNDO and REDO functions in an editor.

#### IV) Queues

A queue is a first-in, first-out (FIFO) data structure in which the element that is inserted first is the first one to be taken out. The elements in a queue are added at one end called the rear and removed from the other end called the front. Like stacks, queues can be implemented by using either arrays or linked lists.

Figure 1.5 shows a queue with 4 elements, where 55 is the front element and 65 is the rear element. Elements can be added from the rear and deleted from the front.



**Figure 1.5: A Queue**

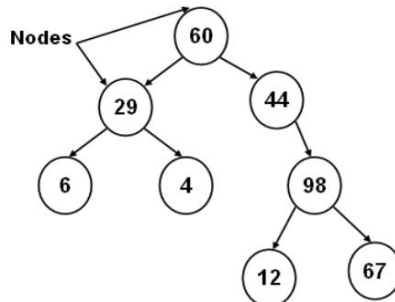
#### Applications:

- It is used in breadth search operation in graphs.
- Job scheduler operations of OS like a print buffer queue, keyboard buffer queue to store the keys pressed by users
- Job scheduling, CPU scheduling, Disk Scheduling
- Priority queues are used in file downloading operations in a browser
- Data transfer between peripheral devices and CPU.
- Interrupts generated by the user applications for CPU
- Calls handled by the customers in BPO

## V) Trees

A tree is a non-linear data structure in which data is organized in branches. The data elements in tree are arranged in a sorted order. It imposes a hierarchical structure on the data elements.

Figure 1.6 represents a tree which consists of 8 nodes. The root of the tree is the node 60 at the top. Node 29 and 44 are the successors of the node 60. The nodes 6, 4, 12 and 67 are the terminal nodes as they do not have any successors.



**Figure 1.6: A Tree**

**Advantage:** Provides quick search, insert, and delete operations

**Disadvantage:** Complicated deletion algorithm

### Applications:

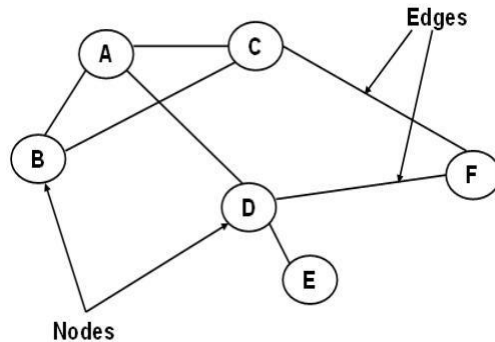
- Implementing the hierarchical structures in computer systems like directory and file system.
- Implementing the navigation structure of a website.
- Code generation like Huffman's code.
- Decision making in gaming applications.
- Implementation of priority queues for priority-based OS scheduling functions
- Parsing of expressions and statements in programming language compilers
- For storing data keys for DBMS for indexing
- Spanning trees for routing decisions in computer and communications networks
- Hash trees
- path-finding algorithm to implement in AI, robotics and video games applications

## VI) Graphs

A graph is also a non-linear data structure. In a tree data structure, all data elements are stored in definite hierarchical structure. In other words, each node has only one parent node. While in graphs, each data element is called a

vertex and is connected to many other vertexes through connections called edges.

Thus, a graph is considered as a mathematical structure, which is composed of a set of vertexes and a set of edges. Figure shows a graph with six nodes A, B, C, D, E, F and seven edges [A, B], [A, C], [A, D], [B, C], [C, F], [D, F] and [D, E].



**Figure 1.7 Graph**

**Advantage:** Best models real-world situations

**Disadvantage:** Some algorithms are slow and very complex

**Applications:**

- Representing networks and routes in communication, transportation and travel applications
- Routes in GPS
- Interconnections in social networks and other network-based applications
- Mapping applications
- Ecommerce applications to present user preferences
- Utility networks to identify the problems posed to municipal or local corporations
- Resource utilization and availability in an organization
- Document link map of a website to display connectivity between pages through hyperlinks
- Robotic motion and neural networks

## **1.4 STATIC DATA STRUCTURE VS DYNAMIC DATA STRUCTURE**

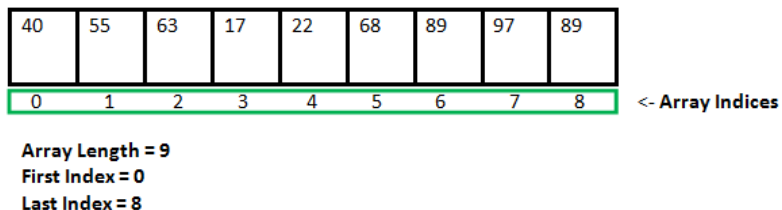
Data structure is a way of storing and organising data efficiently such that the required operations on them can be performed be efficient with respect to time as well as memory. Simply, Data Structure are used to reduce complexity (mostly the time complexity) of the code.

Data structures can be two types:

1. Static Data Structure
2. Dynamic Data Structure

### What is a Static Data structure?

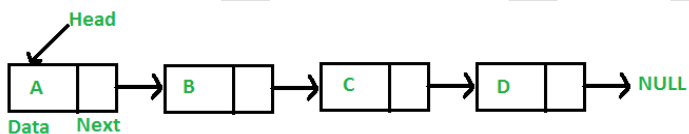
In Static data structure the size of the structure is fixed. The content of the data structure can be modified but without changing the memory space allocated to it.



Example of Static Data Structures: **Array**

### What is Dynamic Data Structure?

In Dynamic data structure the size of the structure is not fixed and can be modified during the operations performed on it. Dynamic data structures are designed to facilitate change of data structures in the run time.



Example of Dynamic Data Structures: **Linked List**

### Static Data Structure vs Dynamic Data Structure

Static Data structure has fixed memory size whereas in Dynamic Data Structure, the size can be randomly updated during run time which may be considered efficient with respect to memory complexity of the code. Static Data Structure provides more easier access to elements with respect to dynamic data structure. Unlike static data structures, dynamic data structures are flexible.

## 1.5 OPERATIONS ON DATA STRUCTURES

This section discusses the different operations that can be performed on the various data structures previously mentioned.

**Traversing** It means to access each data item exactly once so that it can be processed. For example, to print the names of all the students in a class.

**Searching** It is used to find the location of one or more data items that satisfy the given constraint. Such a data item may or may not be present in the given collection of data items. For example, to find the names of all the students who secured 100 marks in mathematics.

**Inserting** It is used to add new data items to the given list of data items. For example, to add the details of a new student who has recently joined the course.

**Deleting** It means to remove (delete) a particular data item from the given collection of data items. For example, to delete the name of a student who has left the course.

**Sorting** Data items can be arranged in some order like ascending order or descending order depending on the type of application. For example, arranging the names of students in a class in an alphabetical order, or calculating the top three winners by arranging the participants' scores in descending order and then extracting the top three.

**Merging** Lists of two sorted data items can be combined to form a single list of sorted data items.

## 1.6 ABSTRACT DATA TYPE

According to National Institute of Standards and Technology (NIST), a data structure is an organization of information, usually in the memory, for better algorithm efficiency. Data structures include queues, stacks, linked lists, dictionary, and trees. They could also be a conceptual entity, such as the name and address of a person.

From the above definition, it is clear that the operations in data structure involve higher-level abstractions such as, adding or deleting an item from a list, accessing the highest priority item in a list, or searching and sorting an item in a list. When the data structure does such operations, it is called an abstract data type.

It can be defined as a collection of data items together with the operations on the data. The word “abstract” refers to the fact that the data and the basic operations defined on it are being studied independently of how they are implemented. It involves **what** can be done with the data, not **how** has to be done. For ex, in the below figure the user would be involved in checking that what can be done with the data collected not how it has to be done.

An implementation of ADT consists of storage structures to store the data items and algorithms for basic operation. All the data structures i.e. array, linked list, stack, queue etc are examples of ADT.

## **Advantage of using ADTs**

In the real world, programs *evolve* as a result of new requirements or constraints, so a modification to a program commonly requires a change in one or more of its data structures. For example, if you want to add a new field to a student's record to keep track of more information about each student, then it will be better to replace an array with a linked structure to improve the program's efficiency. In such a scenario, rewriting every procedure that uses the changed structure is not desirable. Therefore, a better alternative is to *separate* the use of a data structure from the details of its implementation. This is the principle underlying the use of abstract data types.

## **1.7 ALGORITHM**

Algorithm is a step-by-step procedure, which defines a set of instructions to be executed in a certain order to get the desired output. Algorithms are generally created independent of underlying languages, i.e. an algorithm can be implemented in more than one programming language.

From the data structure point of view, following are some important categories of algorithms –

- **Search** – Algorithm to search an item in a data structure.
- **Sort** – Algorithm to sort items in a certain order.
- **Insert** – Algorithm to insert item in a data structure.
- **Update** – Algorithm to update an existing item in a data structure.
- **Delete** – Algorithm to delete an existing item from a data structure.

### **1.7.1 Characteristics of an Algorithm**

Not all procedures can be called an algorithm. An algorithm should have the following characteristics –

An algorithm should have the following characteristics –

- **Clear and Unambiguous:** Algorithm should be clear and unambiguous. Each of its steps should be clear in all aspects and must lead to only one meaning.
- **Well-Defined Inputs:** If an algorithm says to take inputs, it should be well-defined inputs.
- **Well-Defined Outputs:** The algorithm must clearly define what output will be yielded and it should be well-defined as well.
- **Finite-ness:** The algorithm must be finite, i.e. it should not end up in an infinite loops or similar.

- **Feasible:** The algorithm must be simple, generic and practical, such that it can be executed upon will the available resources. It must not contain some future technology, or anything.
- **Language Independent:** The Algorithm designed must be language-independent, i.e. it must be just plain instructions that can be implemented in any language, and yet the output will be same, as expected.

## 1.7.2 Advantages and Disadvantages of Algorithm

### Advantages of Algorithms:

- It is easy to understand.
- Algorithm is a step-wise representation of a solution to a given problem.
- In Algorithm the problem is broken down into smaller pieces or steps hence, it is easier for the programmer to convert it into an actual program.

### Disadvantages of Algorithms:

- Writing an algorithm takes a long time so it is time-consuming.
- Branching and Looping statements are difficult to show in Algorithms.

## 1.7.3 Different approach to design an algorithm

1. **Top-Down Approach:** A top-down approach starts with identifying major components of system or program decomposing them into their lower level components & iterating until desired level of module complexity is achieved . In this we start with topmost module & incrementally add modules that is calls.

2. **Bottom-Up Approach:** A bottom-up approach starts with designing most basic or primitive component & proceeds to higher level components. Starting from very bottom , operations that provide layer of abstraction are implemented

## 1.7.4 How to Write an Algorithm?

There are no well-defined standards for writing algorithms. Rather, it is problem and resource dependent. Algorithms are never written to support a particular programming code.

As we know that all programming languages share basic code constructs like loops (do, for, while), flow-control (if-else), etc. These common constructs can be used to write an algorithm.

We write algorithms in a step-by-step manner, but it is not always the case. Algorithm writing is a process and is executed after the problem domain is well-defined. That is, we should know the problem domain, for which we are designing a solution.

### Example

Let's try to learn algorithm-writing by using an example.

**Problem** – Design an algorithm to add two numbers and display the result.

**Step 1** – START

**Step 2** – declare three integers **a**, **b** & **c**

**Step 3** – define values of **a** & **b**

**Step 4** – add values of **a** & **b**

**Step 5** – store output of step 4 to **c**

**Step 6** – print **c**

**Step 7** – STOP

Algorithms tell the programmers how to code the program. Alternatively, the algorithm can be written as –

**Step 1** – START ADD

**Step 2** – get values of **a** & **b**

**Step 3** –  $c \leftarrow a + b$

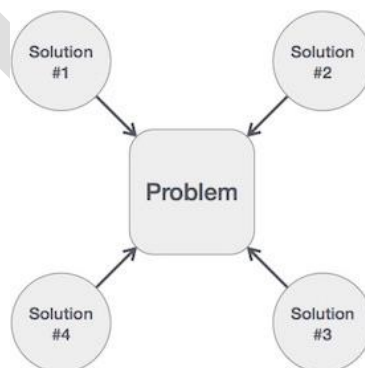
**Step 4** – display **c**

**Step 5** – STOP

In design and analysis of algorithms, usually the second method is used to describe an algorithm. It makes it easy for the analyst to analyze the algorithm ignoring all unwanted definitions. He can observe what operations are being used and how the process is flowing.

Writing **step numbers**, is optional.

We design an algorithm to get a solution of a given problem. A problem can be solved in more than one ways.



Hence, many solution algorithms can be derived for a given problem. The next step is to analyze those proposed solution algorithms and implement the best suitable solution.



## 1.8 ALGORITHM COMPLEXITY

Suppose **X** is an algorithm and **n** is the size of input data, the time and space used by the algorithm **X** are the two main factors, which decide the efficiency of **X**.

- **Time Factor** – Time is measured by counting the number of key operations such as comparisons in the sorting algorithm.
- **Space Factor** – Space is measured by counting the maximum memory space required by the algorithm.

The complexity of an algorithm **f(n)** gives the running time and/or the storage space required by the algorithm in terms of **n** as the size of input data.

### 1.8.1 Space Complexity

Space complexity of an algorithm represents the amount of memory space required by the algorithm in its life cycle. The space required by an algorithm is equal to the sum of the following two components –

- A fixed part that is a space required to store certain data and variables, that are independent of the size of the problem. For example, simple variables and constants used, program size, etc.
- A variable part is a space required by variables, whose size depends on the size of the problem. For example, dynamic memory allocation, recursion stack space, etc.

Space complexity  $S(P)$  of any algorithm **P** is  $S(P) = C + SP(I)$ , where **C** is the fixed part and **S(I)** is the variable part of the algorithm, which depends on instance characteristic **I**. Following is a simple example that tries to explain the concept –

Algorithm: SUM(A, B)  
Step 1 - START  
Step 2 -  $C \leftarrow A + B + 10$   
Step 3 - Stop

Here we have three variables **A**, **B**, and **C** and one constant. Hence  $S(P) = 1 + 3$ . Now, space depends on data types of given variables and constant types and it will be multiplied accordingly.

### 1.8.2 Time Complexity

Time complexity of an algorithm represents the amount of time required by the algorithm to run to completion. Time requirements can be defined as a numerical function  $T(n)$ , where  $T(n)$  can be measured as the number of steps, provided each step consumes constant time.

For example, addition of two  $n$ -bit integers takes  $n$  steps. Consequently, the total computational time is  $T(n) = c * n$ , where  $c$  is the time taken for the addition of two bits. Here, we observe that  $T(n)$  grows linearly as the input size increases.

## 1.9 ALGORITHM ANALYSIS

Efficiency of an algorithm can be analyzed at two different stages, before implementation and after implementation. They are the following –

- **A *Priori* Analysis** or Performance or Asymptotic Analysis – This is a theoretical analysis of an algorithm. Efficiency of an algorithm is measured by assuming that all other factors, for example, processor speed, are constant and have no effect on the implementation.
- **A *Posterior* Analysis** or Performance Measurement – This is an empirical analysis of an algorithm. The selected algorithm is implemented using programming language. This is then executed on target computer machine. In this analysis, actual statistics like running time and space required, are collected.

We shall learn about *a priori* algorithm analysis. Algorithm analysis deals with the execution or running time of various operations involved. The running time of an operation can be defined as the number of computer instructions executed per operation.

Analysis of an algorithm is required to determine the amount of resources such as time and storage necessary to execute the algorithm. Usually, the efficiency or running time of an algorithm is stated as a function which relates the input length to the time complexity or space complexity.

Algorithm analysis framework involves finding out the time taken and the memory space required by a program to execute the program. It also determines how the input size of a program influences the running time of the program.

In theoretical analysis of algorithms, it is common to estimate their complexity in the asymptotic sense, i.e., to estimate the complexity function for arbitrarily large input. Big-O notation, Omega notation, and Theta notation are used to estimate the complexity function for large arbitrary input.

### 1.9.1 Types of Analysis

The efficiency of some algorithms may vary for inputs of the same size. For such algorithms, we need to differentiate between the worst case, average case and best case efficiencies.

#### 1.9.1.1 Best Case Analysis

If an algorithm takes the least amount of time to execute a specific set of input, then it is called the best case time complexity. The best case efficiency of an algorithm is the efficiency for the best case input of size  $n$ . Because of this input, the algorithm runs the fastest among all the possible inputs of the same size.

### **1.9.1.2 Average Case Analysis**

If the time complexity of an algorithm for certain sets of inputs are on an average, then such a time complexity is called average case time complexity.

Average case analysis provides necessary information about an algorithm's behavior on a typical or random input. You must make some assumption about the possible inputs of size  $n$  to analyze the average case efficiency of algorithm.

### **1.9.1.3 Worst Case Analysis**

If an algorithm takes maximum amount of time to execute for a specific set of input, then it is called the worst case time complexity. The worst case efficiency of an algorithm is the efficiency for the worst case input of size  $n$ . The algorithm runs the longest among all the possible inputs of the similar size because of this input of size  $n$ .

## **1.10 MATHEMATICAL NOTATION**

Algorithms are widely used in various areas of study. We can solve different problems using the same algorithm. Therefore, all algorithms must follow a standard. The mathematical notations use symbols or symbolic expressions, which have a precise semantic meaning.

### **1.10.1 Asymptotic Notations**

A problem may have various algorithmic solutions. In order to choose the best algorithm for a particular process, you must be able to judge the time taken to run a particular solution. More accurately, you must be able to judge the time taken to run two solutions, and choose the better among the two.

To select the best algorithm, it is necessary to check the efficiency of each algorithm. The efficiency of each algorithm can be checked by computing its time complexity. The asymptotic notations help to represent the time complexity in a shorthand way. It can generally be represented as the fastest possible, slowest possible or average possible.

The notations such as O (Big-O),  $\Omega$  (Omega), and  $\theta$  (Theta) are called as asymptotic notations. These are the mathematical notations that are used in three different cases of time complexity.

### 1.10.1.1 Big-O Notation

‘O’ is the representation for Big-O notation. Big -O is the method used to express the upper bound of the running time of an algorithm. It is used to describe the performance or time complexity of the algorithm. Big-O specifically describes the worst-case scenario and can be used to describe the execution time required or the space used by the algorithm.

Table 2.1 gives some names and examples of the common orders used to describe functions. These orders are ranked from top to bottom.

Table 2.1: Common Orders			
Time complexity			Examples
1	O(1)	Constant	Adding to the front of a linked list
2	O(log n)	Logarithmic	Finding an entry in a sorted array
3	O(n)	Linear	Finding an entry in an unsorted array
4	O(n log n)	Linearithmic	Sorting ‘n’ items by ‘divide-and-conquer’
5	O(n <sup>2</sup> )	Quadratic	Shortest path between two nodes in a graph
6	O(n <sup>3</sup> )	Cubic	Simultaneous linear equations
7	O(2 <sup>n</sup> )	Exponential	The Towers of Hanoi problem

Big-O notation is generally used to express an ordering property among the functions. This notation helps in calculating the maximum amount of time taken by an algorithm to compute a problem. Big-O is defined as:

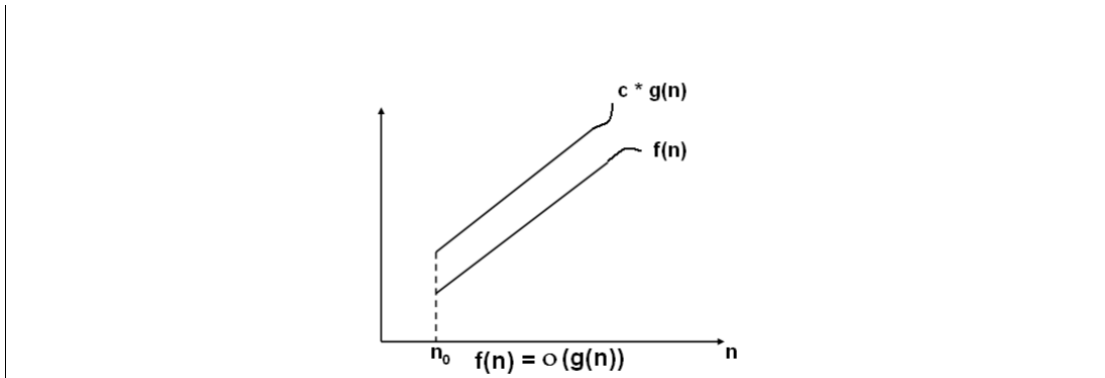
$$f(n) \leq c * g(n)$$

where, **n** can be any number of inputs or outputs and **f(n)** as well as **g(n)** are two non-negative functions. These functions are true only if there is a constant **c** and a non-negative integer **n<sub>0</sub>** such that,  $n \geq n_0$ .

The Big-O can also be denoted as  $f(n) = O(g(n))$ , where **f(n)** and **g(n)** are two non -negative functions and  $f(n) < g(n)$  if **g(n)** is multiple of some constant **c**. The graphical representation of  $f(n) = O(g(n))$  is shown in figure 2.1, where the running time increases considerably when **n** increases.

**Example:** Consider  $f(n)=15n^3+40n^2+2n\log n+2n$ . As the value of  $n$  increases,  $n^3$  becomes much larger than  $n^2$ ,  $n\log n$ , and  $n$ . Hence, it dominates the function  $f(n)$  and we can consider the running time to grow by the order of  $n^3$ . Therefore, it can be written as  $f(n)=O(n^3)$ .

The values of  $n$  for  $f(n)$  and  $C * g(n)$  will not be less than  $n_0$ . Therefore, the values less than  $n_0$  are not considered relevant.



**Figure 1.8: Big-O Notation  $f(n) = O(g(n))$**

Let us take an example to understand the Big-O notation more clearly.

**Example:**

Consider function  $f(n) = 2(n)+2$  and  $g(n) = n^2$ .

We need to find the constant  $c$  such that  $f(n) \leq c * g(n)$ .

Let  $n = 1$ , then

$$f(n) = 2(n)+2 = 2(1)+2 = 4$$

$$g(n) = n^2 = 1^2 = 1$$

Here,  $f(n) > g(n)$

Let  $n = 2$ , then

$$f(n) = 2(n)+2 = 2(2)+2 = 6$$

$$g(n) = n^2 = 2^2 = 4$$

Here,  $f(n) > g(n)$

Let  $n = 3$ , then

$$f(n) = 2(n)+2 = 2(3)+2 = 8$$

$$g(n) = n^2 = 3^2 = 9$$

Here,  $f(n) < g(n)$

Thus, when  $n$  is greater than 2, we get  $f(n) < g(n)$ . In other words, as  $n$  becomes larger, the running time increases considerably. This concludes that the Big-O helps to determine the 'upper bound' of the algorithm's run-time.

### Limitations of Big O Notation

There are certain limitations with the Big O notation of expressing the complexity of algorithms. These limitations are as follows:

- Many algorithms are simply too hard to analyse mathematically.
- There may not be sufficient information to calculate the behaviour of the algorithm in the average case.
- Big O analysis only tells us how the algorithm grows with the size of the problem, not how efficient it is, as it does not consider the programming effort.
- It ignores important constants. For example, if one algorithm takes  $O(n^2)$  time to execute and the other takes  $O(100000n^2)$  time to execute, then as per Big O, both algorithms have equal time complexity. In real-time systems, this may be a serious consideration.

#### 1.10.1.2 Omega Notation

' $\Omega$ ' is the representation for Omega notation. Omega describes the manner in which an algorithm performs in the best case time complexity. This notation provides the minimum amount of time taken by an algorithm to compute a problem. Thus, it is considered that omega gives the "lower bound" of the algorithm's run-time. Omega is defined as:

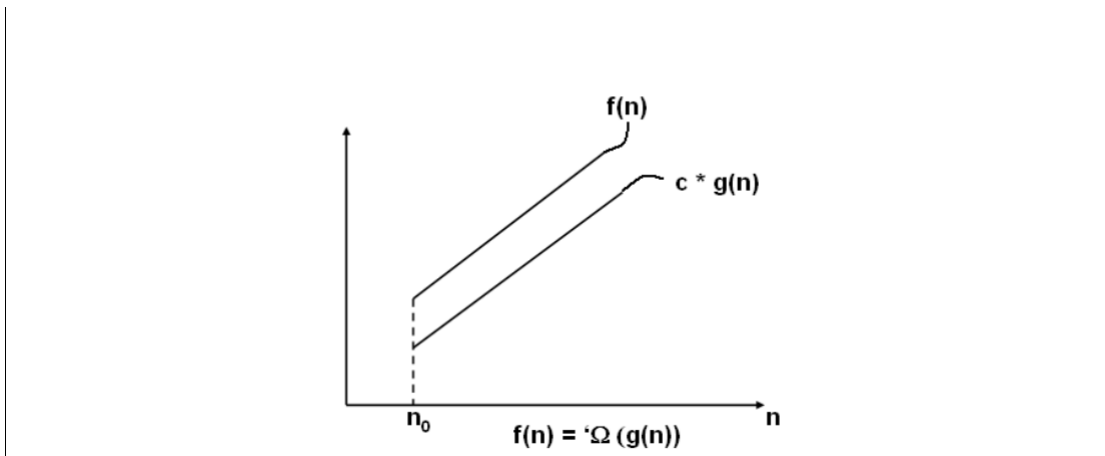
$$f(n) \geq c * g(n)$$

Where,  $n$  is any number of inputs or outputs and  $f(n)$  and  $g(n)$  are two non-negative functions. These functions are true only if there is a constant  $c$  and a non-negative integer  $n_0$  such that  $n > n_0$ .

Omega can also be denoted as  $f(n) = \Omega(g(n))$  where,  $f$  of  $n$  is equal to Omega of  $g$  of  $n$ . The graphical representation of  $f(n) = \Omega(g(n))$  is shown in figure 2.2. The function  $f(n)$  is said to be in  $\Omega(g(n))$ , if  $f(n)$  is bounded below by some constant multiple of  $g(n)$  for all large values of  $n$ , i.e., if there exists some positive constant  $c$  and some non-negative integer  $n_0$ , such that  $f(n) \geq c * g(n)$  for all  $n \geq n_0$ .

Figure 2.2 shows Omega notation.

**Figure 1.9 Omega Notation  $f(n) = \Omega(g(n))$**



Let us take an example to understand the Omega notation more clearly.

**Example:**

Consider function  $f(n) = 2n^2+5$  and  $g(n) = 7n$ .

We need to find the constant  $c$  such that  $f(n) \geq c * g(n)$ .

Let  $n = 0$ , then

$$f(n) = 2n^2+5 = 2(0)^2+5 = 5$$

$$g(n) = 7(n) = 7(0) = 0$$

Here,  $f(n) > g(n)$

Let  $n = 1$ , then

$$f(n) = 2n^2+5 = 2(1)^2+5 = 7$$

$$g(n) = 7(n) = 7(1) = 7$$

Here,  $f(n) = g(n)$

Let  $n = 2$ , then

$$f(n) = 2n^2+5 = 2(2)^2+5 = 13$$

$$g(n) = 7(n) = 7(2) = 14$$

Here,  $f(n) < g(n)$

Thus, for  $n=1$ , we get  $f(n) \geq c * g(n)$ . This concludes that Omega helps to determine the "lower bound" of the algorithm's run-time.

**1.10.1.3 Theta Notation**

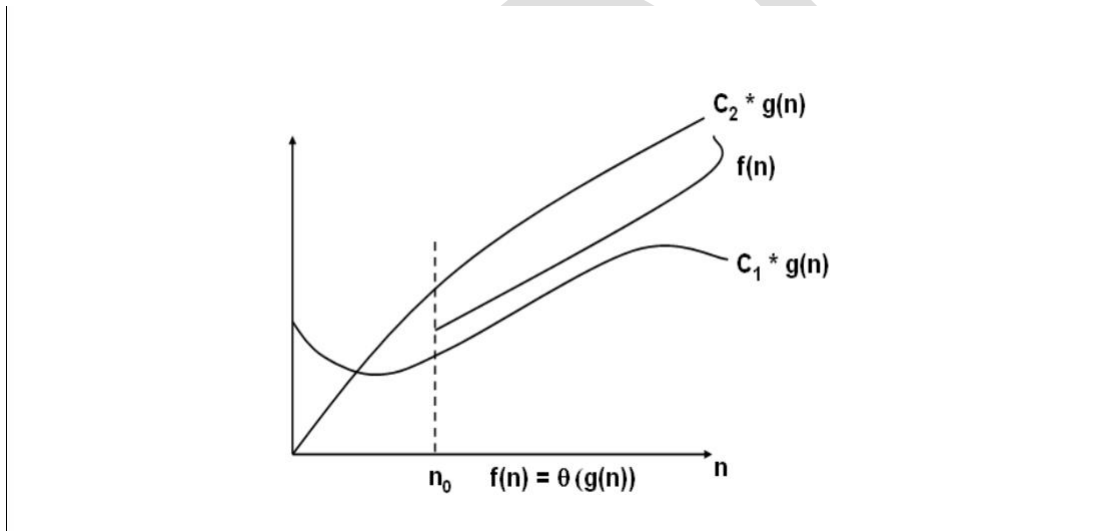
' $\theta$ ' is the representation for Theta notation. Theta notation is used when the upper bound and lower bound of an algorithm are in the same order of magnitude. Theta can be defined as:

$$c_1 * g(n) \leq f(n) \leq c_2 * g(n) \quad \text{for all } n > n_0$$

Where,  $n$  is any number of inputs or outputs and  $f(n)$  and  $g(n)$  are two non-negative functions. These functions are true only if there are two constants namely,  $c_1$ ,  $c_2$ , and a non-negative integer  $n_0$ .

Theta can also be denoted as  $f(n) = \theta(g(n))$  where,  $f$  of  $n$  is equal to Theta of  $g$  of  $n$ . The graphical representation of  $f(n) = \theta(g(n))$  is shown in figure 2.3. The function  $f(n)$  is said to be in  $\theta(g(n))$  if  $f(n)$  is bounded both above and below by some positive constant multiples of  $g(n)$  for all large values of  $n$ , i.e., if there exists some positive constant  $c_1$  and  $c_2$  and some non-negative integer  $n_0$ , such that  $C_2g(n) \leq f(n) \leq C_1g(n)$  for all  $n \geq n_0$ .

Figure shows Theta notation.



**Figure 1.10: Theta Notation  $f(n) = \theta(g(n))$**

Let us take an example to understand the Theta notation more clearly.

**Example:** Consider function  $f(n) = 4n + 3$  and  $g(n) = 4n$  for all  $n \geq 3$ ; and  $f(n) = 4n + 3$  and  $g(n) = 5n$  for all  $n \geq 3$ .

Then the result of the function will be:

Let  $n = 3$

$$f(n) = 4n + 3 = 4(3) + 3 = 15$$

$$g(n) = 4n = 4(3) = 12 \text{ and}$$

$$f(n) = 4n + 3 = 4(3) + 3 = 15$$

$$g(n) = 5n = 5(3) = 15 \text{ and}$$



here,  $c_1$  is 4,  $c_2$  is 5 and  $n_0$  is 3  
Thus, from the above equation we get  $c_1 g(n) f(n) c_2 g(n)$ . This concludes that Theta notation depicts the running time between the upper bound and lower bound.

## 1.11 ALGORITHM DESIGN TECHNIQUE

- 1.11.1 Divide and Conquer
- 1.11.2 Back Tracking Method
- 1.11.3 Dynamic programming

### 1.11.1 Divide and Conquer

#### Introduction

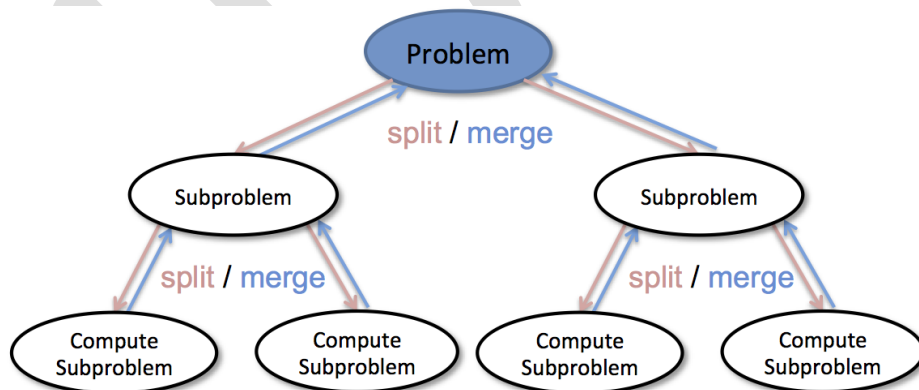
Divide and Conquer approach basically works on breaking the problem into sub problems that are similar to the original problem but smaller in size & simpler to solve. once divided sub problems are solved recursively and then combine solutions of sub problems to create a solution to original problem.

At each level of the recursion the divide and conquer approach follows three steps:

**Divide:** In this step whole problem is divided into several sub problems.

**Conquer:** The sub problems are conquered by solving them recursively, only if they are small enough to be solved, otherwise step1 is executed.

**Combine:** In this final step, the solution obtained by the sub problems are combined to create solution to the original problem.



Generally, we can follow the **divide-and-**

**conquer** approach in a three-step process.

**Examples:** The specific computer algorithms are based on the Divide & Conquer approach:

1. Maximum and Minimum Problem
2. Binary Search

3. Sorting (merge sort, quick sort)
4. Tower of Hanoi.

## **Fundamental of Divide & Conquer Strategy:**

There are two fundamentals of Divide & Conquer Strategy:

1. Relational Formula
2. Stopping Condition

**1. Relational Formula:** It is the formula that we generate from the given technique. After generation of Formula, we apply D&C Strategy, i.e., we break the problem recursively & solve the broken subproblems.

**2. Stopping Condition:** When we break the problem using Divide & Conquer Strategy, then we need to know that for how much time, we need to apply divide & Conquer. So, the condition where the need to stop our recursion steps of D&C is called as Stopping Condition.

## **Applications of Divide and Conquer Approach:**

Following algorithms are based on the concept of the Divide and Conquer Technique:

1. **Binary Search:** The binary search algorithm is a searching algorithm, which is also called a half-interval search or logarithmic search. It works by comparing the target value with the middle element existing in a sorted array. After making the comparison, if the value differs, then the half that cannot contain the target will eventually eliminate, followed by continuing the search on the other half. We will again consider the middle element and compare it with the target value. The process keeps on repeating until the target value is met. If we found the other half to be empty after ending the search, then it can be concluded that the target is not present in the array.
2. **Quicksort:** It is the most efficient sorting algorithm, which is also known as partition-exchange sort. It starts by selecting a pivot value from an array followed by dividing the rest of the array elements into two sub-arrays. The partition is made by comparing each of the elements with the pivot value. It compares whether the element holds a greater value or lesser value than the pivot and then sort the arrays recursively.
3. **Merge Sort:** It is a sorting algorithm that sorts an array by making comparisons. It starts by dividing an array into sub-array and then recursively sorts each of them. After the sorting is done, it merges them back.

## **Advantages of Divide and Conquer**

- Divide and Conquer tend to successfully solve one of the biggest problems, such as the Tower of Hanoi, a mathematical puzzle. It is challenging to solve complicated problems for which you have no basic idea, but with the help of the divide and conquer approach, it has lessened the effort as it works on dividing the main problem into two halves and then solve them recursively. This algorithm is much faster than other algorithms.
- It efficiently uses cache memory without occupying much space because it solves simple subproblems within the cache memory instead of accessing the slower main memory.

### **Disadvantages of Divide and Conquer**

- Since most of its algorithms are designed by incorporating recursion, so it necessitates high memory management.
- An explicit stack may overuse the space.
- It may even crash the system if the recursion is performed rigorously greater than the stack present in the CPU.

### **1.11.2 Backtracking**

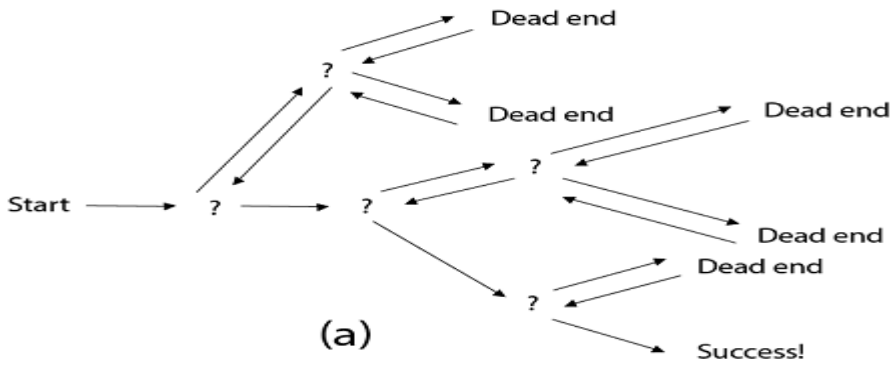
#### **Introduction**

The Backtracking is an algorithmic-method to solve a problem with an additional way. It uses a recursive approach to explain the problems. We can say that the backtracking is needed to find all possible combination to solve an optimization problem.

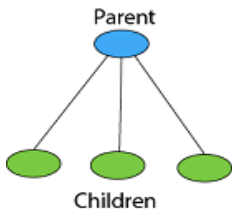
**Backtracking** is a systematic way of trying out different sequences of decisions until we find one that "works."

In the following Figure:

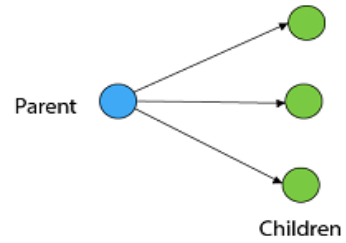
- Each non-leaf node in a tree is a parent of one or more other nodes (its children)
- Each node in the tree, other than the root, has exactly one parent



Generally, however, we draw our trees downward, with the root at the top.

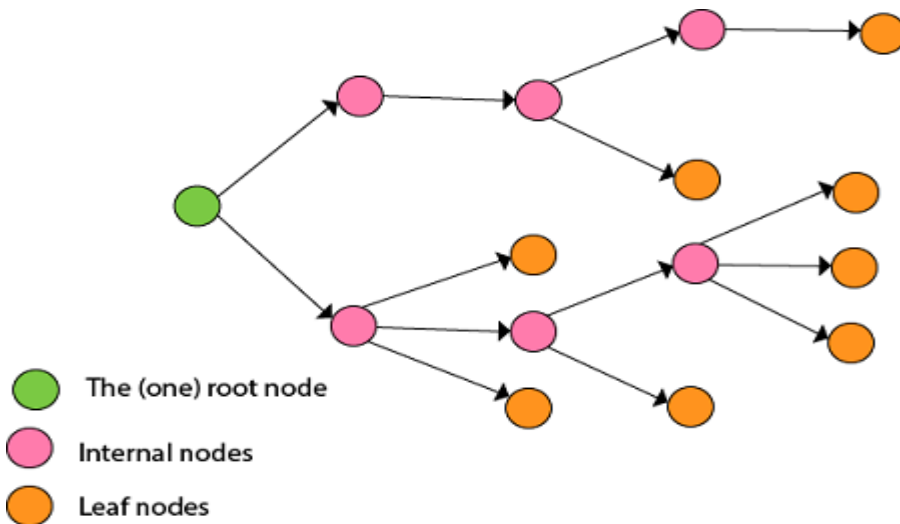


or



(b)

A tree is composed of nodes.



(c)

**Backtracking can understand of as searching a tree for a particular "goal" leaf node.**

Backtracking is undoubtedly quite simple - we "explore" each node, as follows:

**To "explore" node N:**

1. If N is a goal node, return "success"
2. If N is a leaf node, return "failure"
3. For each child C of N,  
     Explore C  
     If C was successful, return "success"
4. Return "failure"

Backtracking algorithm determines the solution by systematically searching the solution space for the given problem. Backtracking is a **depth-first search** with any bounding function. All solution using backtracking is needed to satisfy a complex set of constraints. The constraints may be explicit or implicit.

**Explicit Constraint** is ruled, which restrict each vector element to be chosen from the given set.

**Implicit Constraint** is ruled, which determine which each of the tuples in the solution space, actually satisfy the criterion function.

### 1.11.3 Dynamic programming

Dynamic Programming Technique is similar to divide-and-conquer technique. Both techniques solve a problem by breaking it down into several sub-problems that can be solved recursively. The main difference between is that, Divide & Conquer approach partitions the problems into independent sub-problems, solve the sub-problems recursively, and then combine their solutions to solve the original problems. Whereas dynamic programming is applicable when the sub-problems are not independent, that is, when sub-problems share sub subproblems. Also, A dynamic programming algorithms solves every sub problem just once and then saves its answer in a table, thereby avoiding the work of recomputing the answer every time the sub subproblems is encountered.

Therefore "**Dynamic programming is a applicable when sub problem are not independent, that is when sub problem share sub problems.**"

As Greedy approach, Dynamic programming is typically applied to optimization problems and for them there can be many possible solutions and the requirement is to find the optimal solution among those. But Dynamic programming approach is little different greedy approach. In greedy solutions are computed by making choices in serial forward way and in this no backtracking & revision of choices is done where as Dynamic programming computes its solution bottom up by producing them from smaller sub problems, and by trying many possibilities and choices before it arrives at the optimal set of choices.

The Development of a dynamic-programming algorithm can be broken into a sequence of four steps:

**Divide, Sub problems:** The main problems are divided into several smaller sub problems. In this the solution of the main problem is expressed in terms of the solution for the smaller sub problems. Basically, it is all about characterizing the structure of an optimal solution and recursively define the value of an optimal solution.

**Table, Storage:** The solution for each sub problem is stored in a table, so that it can be used many times whenever required.

**Combine, bottom-up Computation:** The solution to main problem is obtained by combining the solutions of smaller sub problems. i.e., compute the value of an optimal solution in a bottom-up fashion.

Construct an optimal solution from computed information. **(This step is optional and is required in case if some additional information is required after finding out optimal solution.)**

Now for any problem to be solved through dynamic programming approach it must follow the following conditions:

**Principle of Optimality:** It states that for solving the master problem optimally, its sub problems should be solved optimally. It should be noted that not all the times each sub problem(s) is solved optimally, so in that case we should go for optimal majority.

**Polynomial Breakup:** For solving the main problem, the problem is divided into several sub problems and for efficient performance of dynamic programming the total number of sub problems to be solved should be at-most a polynomial number.

Various algorithms which make use of Dynamic programming technique are as follows:

1. Knapsack problem.
2. Chain matrix multiplication.
3. All pair shortest path.
4. Travelling sales man problem.
5. Tower of hanoi.
6. Checker Board.
7. Fibonacci Sequence.
8. Assembly line scheduling.
9. Optimal binary search trees.

## 1.12 SUMMARY

A data structure is a particular way of storing and organizing data either in computer's memory or on the disk storage so that it can be used efficiently.

There are two types of data structures: primitive and non-primitive data structures.

Primitive data structures are the fundamental data types which are supported by a programming language. Nonprimitive data structures are those data structures which are created using primitive data structures.

Non-primitive data structures can further be classified into two categories: linear and non-linear data structures.

If the elements of a data structure are stored in a linear or sequential order, then it is a linear data structure. However, if the elements of a data structure are not stored in sequential order, then it is a non-linear data structure.

An array is a collection of similar data elements which are stored in consecutive memory locations.

A linked list is a linear data structure consisting of a group of elements (called nodes) which together represent a sequence.

A stack is a last-in, first-out (LIFO) data structure in which insertion and deletion of elements are done at only one end, which is known as the top of the stack.

A queue is a first-in, first-out (FIFO) data structure in which the element that is inserted first is the first to be taken out. The elements in a queue are added at one end called the rear and removed from the other end called the front.

A tree is a non-linear data structure which consists of a collection of nodes arranged in a hierarchical tree structure.

A graph is often viewed as a generalization of the tree structure, where instead of a purely parent-to-child relationship between tree nodes, any kind of complex relationships can exist between the nodes.

An abstract data type (ADT) is the way we look at a data structure, focusing on what it does and ignoring how it does its job.

An algorithm is basically a set of instructions that solve a problem.

The time complexity of an algorithm is basically the running time of the program as a function of the input size.

The space complexity of an algorithm is the amount of computer memory required during the program execution as a function of the input size.

The worst-case running time of an algorithm is an upper bound on the running time for any input.

The average-case running time specifies the expected behaviour of the algorithm when the input is randomly drawn from a given distribution.

The efficiency of an algorithm is expressed in terms of the number of elements that has to be processed and the type of the loop that is being used.

### 1.13 MODEL QUESTIONS

1. Define data structures. Give some examples.
2. In how many ways can you categorize data structures? Explain each of them.
3. Discuss the applications of data structures.
4. Write a short note on different operations that can be performed on data structures.
5. Write a short note on abstract data type.
6. Explain the different types of data structures. Also discuss their merits and demerits.
7. Define an algorithm. Explain its features with the help of suitable examples.
8. Explain and compare the approaches for designing an algorithm.
9. What do you understand by a graph?
10. Explain the criteria that you will keep in mind while choosing an appropriate algorithm to solve a particular problem.
11. What do you understand by time–space trade-off?
12. What do you understand by the efficiency of an algorithm?
13. How will you express the time complexity of a given algorithm?
14. Discuss the significance and limitations of the Big O notation.
15. Discuss the best case, worst case and average case complexity of an algorithm.
16. Categorize algorithms based on their running time complexity.
17. Give examples of functions that are in Big O notation as well as functions that are not in Big O notation.
18. Explain the  $\Omega$  notation.



19. Give examples of functions that are in  $\Omega$  notation as well as functions that are not in  $\Omega$  notation.
20. Explain the  $\Theta$  notation.
21. Give examples of functions that are in  $\Theta$  notation as well as functions that are not in  $\Theta$  notation.
22. Explain the  $\omega$  notation.
23. Give examples of functions that are in  $\omega$  notation as well as functions that are in  $\omega$  notation.

#### **1.14 List of References**

<https://www.javatpoint.com/>  
<https://www.studytonight.com>  
<https://www.tutorialspoint.com>  
<https://www.geeksforgeeks.org/heap-sort/>  
<https://www.programiz.com/dsa/heap-sort>  
<https://www.2braces.com/data-structures>

## Unit 2: CHAPTER-2

# Sorting Techniques

2.0 Objective

2.1 Introduction to Sorting

2.2 Sorting Technique

2.2.1 Bubble Sort

2.2.2 Insertion Sort

2.2.3 Selection Sort

2.2.4 Quick Sort

2.2.5 Heap Sort

2.2.6 Merge Sort

2.2.7 Shell Sort

2.2.8 Comparison of Sorting Methods

2.3 Summary

2.4 Model Questions

2.5 List of References

### 2.0 OBJECTIVE :

After studying this unit, you will be able to:

- To study basic concepts of sorting.
- To study different sorting methods and their algorithms.
- Study different sorting methods and compare with their time complexity

### 2.1 INTRODUCTION TO SORTING

Arranging the data in ascending or descending order is known as sorting.

Sorting is very important from the point of view of our practical life.

The best example of sorting can be phone numbers in our phones. If, they are not maintained in an alphabetical order we would not be able to search any number effectively.

There are two types of sorting:

#### **Internal Sorting:**

If all the data that is to be sorted can be adjusted at a time in the main memory, the internal sorting method is being performed.

#### **External Sorting:**

When the data that is to be sorted cannot be accommodated in the memory at the same time and some has to be kept in auxiliary memory such as hard disk, floppy disk, magnetic tapes etc, then external sorting methods are performed.

### **Application of sorting**

1. The sorting is useful in database applications for arranging the data in desired ORDER.
2. In the dictionary like applications the data is arranged in sorted order.
3. For searching the element from the list of elements the sorting is required
4. For checking the uniqueness of the element the sorting is required.
5. For finding the closest pair from the list of elements the sorting is required.

## **2.2 SORTING TECHNIQUES**

- 1) Bubble sort
- 2) Insertion sort
- 3) Radix Sort
- 4) Quick sort
- 5) Merge sort
- 6) Heap sort
- 7) Selection sort
- 8) shell Sort

### **2.2.1 BUBBLE SORT**

In *bubble sorting*, consecutive adjacent pairs of elements in the array are compared with each other. If the element at the lower index is greater than the element at the higher index, the two elements are interchanged so that the element is placed before the bigger one. This process will continue till the list of unsorted elements exhausts.

This procedure of sorting is called bubble sorting because elements 'bubble' to the top of the list. Note that at the end of the first pass, the largest element in the list will be placed at its proper position (i.e., at the end of the list).

**Note :** If the elements are to be sorted in descending order, then in first pass the smallest element is moved to the highest index of the array.

**Example** To discuss bubble sort in detail, let us consider an array  $A[]$  that has the following elements:

$A[] = \{30, 52, 29, 87, 63, 27, 19, 54\}$

**Pass 1:**

Compare 30 and 52. Since  $30 < 52$ , no swapping is done.  
Compare 52 and 29. Since  $52 > 29$ , swapping is done. 30, **29, 52**, 87, 63, 27, 19, 54

Compare 52 and 87. Since  $52 < 87$ , no swapping is done.  
Compare 87 and 63. Since  $87 > 63$ , swapping is done. 30, 29, 52, **63, 87**, 27, 19, 54

Compare 87 and 27. Since  $87 > 27$ , swapping is done. 30, 29, 52, 63, **27, 87**, 19, 54

Compare 87 and 19. Since  $87 > 19$ , swapping is done. 30, 29, 52, 63, 27, **19, 87**, 54

Compare 87 and 54. Since  $87 > 54$ , swapping is done. 30, 29, 52, 63, 27, 19, **54, 87**

Observe that after the end of the first pass, the largest element is placed at the highest index of the array. All the other elements are still unsorted.

### **Pass 2:**

Compare 30 and 29. Since  $30 > 29$ , swapping is done. **29, 30**, 52, 63, 27, 19, 54, 87

Compare 30 and 52. Since  $30 < 52$ , no swapping is done.

Compare 52 and 63. Since  $52 < 63$ , no swapping is done.

Compare 63 and 27. Since  $63 > 27$ , swapping is done. 29, 30, 52, **27, 63**, 19, 54, 87

Compare 63 and 19. Since  $63 > 19$ , swapping is done.  
29, 30, 52, 27, **19, 63**, 54, 87

Compare 63 and 54. Since  $63 > 54$ , swapping is done. 29, 30, 52, 27, 19, **54, 63**, 87

Observe that after the end of the second pass, the second largest element is placed at the second highest index of the array. All the other elements are still unsorted.

### **Pass 3:**

Compare 29 and 30. Since  $29 < 30$ , no swapping is done.

Compare 30 and 52. Since  $30 < 52$ , no swapping is done.

Compare 52 and 27. Since  $52 > 27$ , swapping is done. 29, 30, **27, 52**, 19, 54, 63, 87

Compare 52 and 19. Since  $52 > 19$ , swapping is done. 29, 30, 27, **19, 52**, 54, 63, 87

Compare 52 and 54. Since  $52 < 54$ , no swapping is done.

Observe that after the end of the third pass, the third largest element is placed at the third highest index of the array. All the other elements are still unsorted.

#### **Pass 4:**

Compare 29 and 30. Since  $29 < 30$ , no swapping is done.

Compare 30 and 27. Since  $30 > 27$ , swapping is done. 29, **27, 30**, 19, 52, 54, 63, 87

Compare 30 and 19. Since  $30 > 19$ , swapping is done. 29, 27, **19, 30**, 52, 54, 63, 87

Compare 30 and 52. Since  $30 < 52$ , no swapping is done.

Observe that after the end of the fourth pass, the fourth largest element is placed at the fourth highest index of the array. All the other elements are still unsorted.

#### **Pass 5:**

Compare 29 and 27. Since  $29 > 27$ , swapping is done. **27, 29**, 19, 30, 52, 54, 63, 87

Compare 29 and 19. Since  $29 > 19$ , swapping is done. 27, **19, 29**, 30, 52, 54, 63, 87

Compare 29 and 30. Since  $29 < 30$ , no swapping is done.

Observe that after the end of the fifth pass, the fifth largest element is placed at the fifth highest index of the array. All the other elements are still unsorted.

#### **Pass 6:**

Compare 27 and 19. Since  $27 > 19$ , swapping is done. **19, 27**, 29, 30, 52, 54, 63, 87

Compare 27 and 29. Since  $27 < 29$ , no swapping is done.

Observe that after the end of the sixth pass, the sixth largest element is placed at the sixth largest index of the array. All the other elements are still unsorted.

### Pass 7:

(a) Compare 19 and 27. Since  $19 < 27$ , no swapping is done.

Observe that the entire list is sorted now.

<b>Algorithm for bubble sort</b>
<b>BUBBLE_SORT(A, N)</b>
Step 1: Repeat Step 2 For $i = 0$ to $N-1$
Step 2: Repeat For $J =$ to $N - i$
Step 3:    IF $A[J] > A[J + 1]$
SWAP $A[J]$ and $A[J+1]$
[END OF INNER LOOP]
[END OF OUTER LOOP]
Step 4: EXIT

### Advantages :

- Simple and easy to implement
- In this sort, elements are swapped in place without using additional temporary storage, so the space requirement is at a minimum.

### Disadvantages :

- It is slowest method .  $O(n^2)$
- Inefficient for large sorting lists.

<b>Program</b>
<pre>#include&lt;stdio.h&gt; void main ()</pre>

```
{
    int i, j, temp;
    int a[10] = { 10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
    for(i = 0; i<10; i++)
    {
        for(j = i+1; j<10; j++)
        {
            if(a[j] > a[i])
            {
                temp = a[i];
                a[i] = a[j];
                a[j] = temp;
            }
        }
    }
    printf("Printing Sorted Element List ...\n");
    for(i = 0; i<10; i++)
    {
        printf("%d\n",a[i]);
    }
}
```

**Output:**

Printing Sorted Element List . . .

7  
9  
10  
12  
23  
34  
34  
44  
78  
101

## Complexity of Bubble Sort

The complexity of any sorting algorithm depends upon the number of comparisons. In bubble sort, we have seen that there are  $N-1$  passes in total. In the first pass,  $N-1$  comparisons are made to place the highest element in its correct position. Then, in Pass 2, there are  $N-2$  comparisons and the second highest element is placed in its position. Therefore, to compute the complexity of bubble sort, we need to calculate the total number of comparisons. It can be given as:

$$f(n) = (n - 1) + (n - 2) + (n - 3) + \dots + 3 + 2 + 1$$

$$f(n) = n(n - 1)/2$$

$$f(n) = n^2/2 + O(n) = O(n^2)$$

Therefore, the complexity of bubble sort algorithm is  $O(n^2)$ . It means the time required to execute bubble sort is proportional to  $n^2$ , where  $n$  is the total number of elements in the array.

### 2.2.2 INSERTION SORT

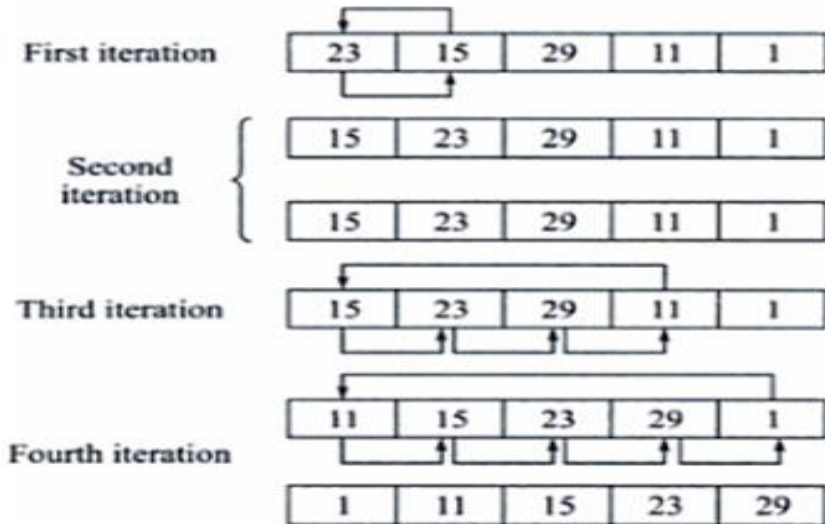
Insertion sort is a very simple sorting algorithm in which the sorted array (or list) is built one element at a time. We all are familiar with this technique of sorting, as we usually use it for ordering a deck of cards while playing bridge.

Insertion sort inserts each item into its proper place in the final list. In insertion sort, the first iteration starts with comparison of 1<sup>st</sup> element with 0<sup>th</sup> element. In the second iteration 2<sup>nd</sup> element is compared with the 0<sup>th</sup> and 1<sup>st</sup> element and so on. In every iteration an element is compared with all elements. The main idea is to insert the  $i^{\text{th}}$  pass the  $i^{\text{th}}$  element in  $A[1], A[2] \dots A[i]$  in its proper place.

**Example** Consider an array of integers given below. We will sort the values in the array using insertion sort

23    15    29    11    1





**Algorithm for insertion sort**

**INSERTION-SORT (ARR, N)**

Step 1: Repeat Steps 2 to 5 for  $K = 1$  to  $N - 1$

Step 2: SET  $TEMP = ARR[K]$

Step 3: SET  $J = K - 1$

Step 4: Repeat while  $TEMP <= ARR[J]$   
 SET  $ARR[J + 1] = ARR[J]$   
 SET  $J = J - 1$   
 [END OF INNER LOOP]

Step 5: SET  $ARR[J + 1] = TEMP$   
 [END OF LOOP]

Step 6: EXIT

**Advantages:**

The advantages of this sorting algorithm are as follows:

- Relatively simple and Easy to implement.
- It is easy to implement and efficient to use on small sets of data.
- It can be efficiently implemented on data sets that are already substantially sorted.
- The insertion sort is an in-place sorting algorithm so the space requirement is minimal.

**Disadvantages:**

- Inefficient for large list  $O(n^2)$ .

### Program

```
#include<stdio.h>
void main ()
{
    int i,j, k,temp;
    int a[10] = { 10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
    printf("\nprinting sorted elements...\n");
    for(k=1; k<10; k++)
    {
        temp = a[k];
        j= k-1;
        while(j>=0 && temp <= a[j])
        {
            a[j+1] = a[j];
            j = j-1;
        }
        a[j+1] = temp;
    }
    for(i=0;i<10;i++)
    {
        printf("\n%d\n",a[i]);
    }
}
```

### Output:

Printing Sorted Elements . . .

7  
9  
10  
12  
23  
23  
34

44  
78  
101

### Complexity of Insertion Sort

If the initial file is sorted, only one comparison is made on each iteration, so that the sort is  $O(n)$ . If the file is initially sorted in the reverse order the worst case complexity is  $O(n^2)$ . Since the total number of comparisons is:

$$(n-1) + (n-2) + \dots + 3 + 2 + 1 = (n-1) * n/2, \text{ which is } O(n^2)$$

The average case or the average number of comparisons in the simple insertion sort is  $O(n^2)$ .

### 2.2.3 SELECTION SORT

Selection sorting is conceptually the simplest sorting algorithm. This algorithm first finds the smallest element in the array and exchanges it with the element in the first position, then finds the second smallest element and exchange it with the element in the second position, and continues in this way until the entire array is sorted.

**Example 1 :** 3, 6, 1, 8, 4, 5

Original Array	After 1st pass	After 2nd pass	After 3rd pass	After 4th pass	After 5th pass
3	1	1	1	1	1
6	6	3	3	3	3
1	3	6	4	4	4
8	8	8	8	5	5
4	4	4	6	6	6
5	5	5	5	8	8

#### Algorithm for selection sort

**SELECTION SORT(ARR, N)**

Step 1: Repeat Steps 2 and 3 for  $K = 1$  to  $N-1$

Step 2: CALL SMALLEST(ARR, K, N, POS)

Step 3: SWAP A[K] with ARR[POS]

[END OF LOOP]

Step 4: EXIT

### **SMALLEST (ARR, K, N, POS)**

Step 1: [INITIALIZE] SET SMALL = ARR[K]

Step 2: [INITIALIZE] SET POS = K

Step 3: Repeat for J = K+1 to N

    IF SMALL > ARR[J]

        SET SMALL = ARR[J]

        SET POS = J

    [END OF IF]

[END OF LOOP]

Step 4: RETURN POS

### **Advantages:**

- It is simple and easy to implement.
- It can be used for small data sets.
- It is 60 per cent more efficient than bubble sort.

### **Disadvantages:**

- Running time of Selection sort algorithm is very poor of  $O(n^2)$ .
- However, in case of large data sets, the efficiency of selection sort drops as compared to insertion sort.

### **Program**

```
#include<stdio.h>
int smallest(int[],int,int);
void main ()
{
    int a[10] = {10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
    int i,j,k,pos,temp;
    for(i=0;i<10;i++)
    {
        pos = smallest(a,10,i);
        temp = a[i];
        a[i]=a[pos];
        a[pos] = temp;
    }
}
```

```
    }
    printf("\nprinting sorted elements...\n");
    for(i=0;i<10;i++)
    {
        printf("%d\n",a[i]);
    }
}
int smallest(int a[], int n, int i)
{
    int small,pos,j;
    small = a[i];
    pos = i;
    for(j=i+1;j<10;j++)
    {
        if(a[j]<small)
        {
            small = a[j];
            pos=j;
        }
    }
    return pos;
}
```

### Output:

```
printing sorted elements...
7
9
10
12
23
23
34
44
78
101
```

## Complexity of Selection Sort

The first element is compared with the remaining  $n-1$  elements in pass 1. Then  $n-2$  elements are taken in pass 2, this process is repeated until the last element is encountered. The mathematical expression for these iterations will be equal to:  $(n-1)+(n-2)+\dots+(n-(n-1))$ . Thus the expression become  $n*(n-1)/2$ . Thus, the number of comparisons is proportional to  $(n^2)$ . The time complexity of selection sort is  $O(n^2)$ .

### 2.2.4 MERGE SORT

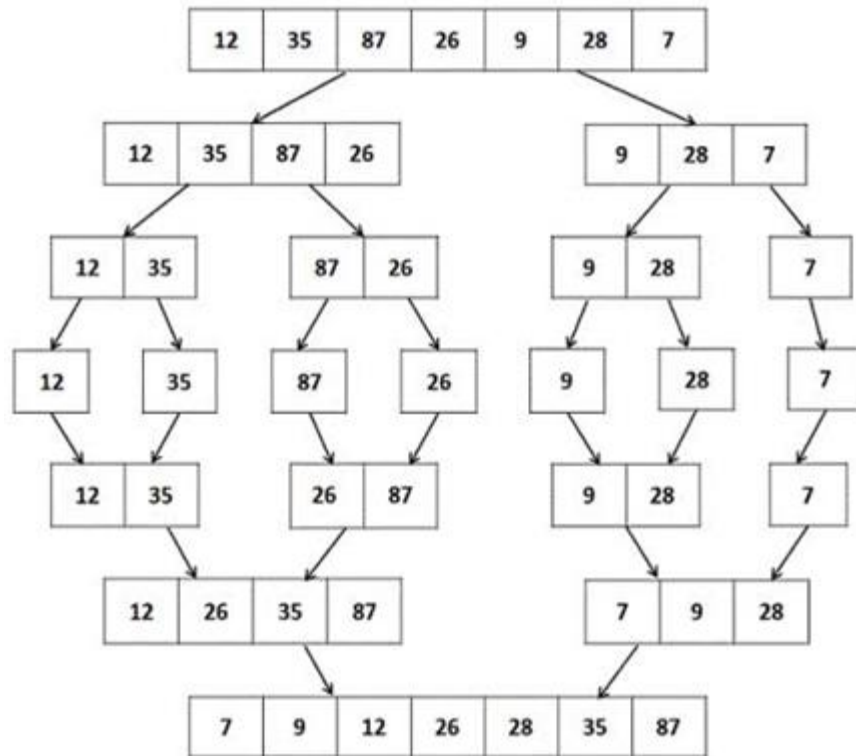
Merging means combining two sorted lists into one-sorted list. The merge sort splits the array to be sorted into two equal halves and each array is recursively sorted, then merged back together to form the final sorted array. The logic is to split the array into two sub arrays each sub array is individually sorted and the resulting sequence is then combined to produce a single sorted sequence of  $n$  elements.

The merge sort recursively follows the steps:

- 1) Divide the given array into equal parts.
- 2) Recursively sorts the elements on the left side of the partitions.
- 3) Recursively sorts the elements on the right side of the partitions.
- 4) Combine the sorted left and right partitions into a single sorted array.

**Example** Sort the array given below using merge sort.

**12 , 35 ,87, 26, 9, 28,7**



Merge Sort

### Algorithm for merge sort

**MERGE\_SORT(ARR, BEG, END)**

Step 1: IF BEG < END

    SET MID = (BEG + END)/2

    CALL MERGE\_SORT (ARR, BEG, MID)

    CALL MERGE\_SORT (ARR, MID + 1, END)

    MERGE (ARR, BEG, MID, END)

    [END OF IF]

Step 2: END

**MERGE (ARR, BEG, MID, END)**

Step 1: [INITIALIZE] SET I = BEG, J = MID + 1, INDEX = 0

Step 2: Repeat while (I <= MID) AND (J <= END)

    IF ARR[I] < ARR[J]

```

        SET TEMP[INDEX] = ARR[I]
        SET I = I + 1
    ELSE
        SET TEMP[INDEX] = ARR[J]
        SET J = J + 1
    [END OF IF]
    SET INDEX = INDEX + 1
[END OF LOOP]
Step 3: [Copy the remaining elements of right sub-array, if any]
    IF I > MID
        Repeat while J <= END
            SET TEMP[INDEX] = ARR[J]
            SET INDEX = INDEX + 1, SET J = J + 1
        [END OF LOOP]
    [Copy the remaining elements of left sub-array, if any]
    ELSE
        Repeat while I <= MID
            SET TEMP[INDEX] = ARR[I]
            SET INDEX = INDEX + 1, SET I = I + 1
        [END OF LOOP]
    [END OF IF]
Step 4: [Copy the contents of TEMP back to ARR] SET K = 0
Step 5: Repeat while K < INDEX
    SET ARR[K] = TEMP[K]
    SET K = K + 1
[END OF LOOP]
Step 6: END

```



## Advantages :

- Merge sort algorithm is best case for sorting slow-access data e.g) tape drive.
- Merge sort algorithm is better at handling sequential - accessed lists.

## Disadvantages:

- Slower comparative to the other sort algorithms for smaller tasks.
- Merge sort algorithm requires additional memory space of  $O(n)$  for the temporary array .
- It goes through the whole process even if the array is sorted.

## Program

```
#include<stdio.h>
void mergeSort(int[],int,int);
void merge(int[],int,int,int);
void main ()
{
    int a[10]= {10, 9, 7, 101, 23, 44, 12, 78, 34, 23};
    int i;
    mergeSort(a,0,9);
    printf("printing the sorted elements");
    for(i=0;i<10;i++)
    {
        printf("\n%d\n",a[i]);
    }
}
void mergeSort(int a[], int beg, int end)
{
    int mid;
    if(beg<end)
    {
        mid = (beg+end)/2;
        mergeSort(a,beg,mid);
        mergeSort(a,mid+1,end);
```

```

    merge(a,beg,mid,end);
}
}
void merge(int a[], int beg, int mid, int end)
{
    int i=beg,j=mid+1,k,index = beg;
    int temp[10];
    while(i<=mid && j<=end)
    {
        if(a[i]<a[j])
        {
            temp[index] = a[i];
            i = i+1;
        }
        else
        {
            temp[index] = a[j];
            j = j+1;
        }
        index++;
    }
    if(i>mid)
    {
        while(j<=end)
        {
            temp[index] = a[j];
            index++;
            j++;
        }
    }
    else
    {
        while(i<=mid)

```

```
    {
        temp[index] = a[i];
        index++;
        i++;
    }
}
k = beg;
while(k<index)
{
    a[k]=temp[k];
    k++;
}
}
```

### **Output:**

printing the sorted elements

7  
9  
10  
12  
23  
23  
34  
44  
78  
101

### **Complexity of Merge Sort**

The running time of merge sort in the average case and the worst case can be given as  $O(n \log n)$ . Although merge sort has an optimal time complexity, it needs an additional space of  $O(n)$  for the temporary array TEMP.

### **Applications**

- Merge Sort is useful for sorting linked lists in  $O(n \log n)$  time.
- Inversion Count Problem
- Used in External Sorting

### **2.2.5 QUICK SORT**

Quick sort sorts by employing a divide and conquer strategy to divide a list into two sub-lists.

The steps are:

1. Pick an element, called a *pivot*, from the list.
2. Reorder the list so that all elements which are less than the pivot come before the pivot and so that all elements greater than the pivot come after it (equal values can go either way).

After

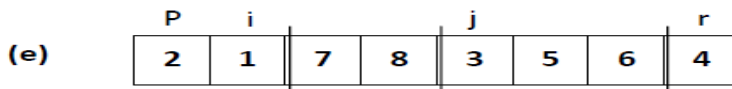
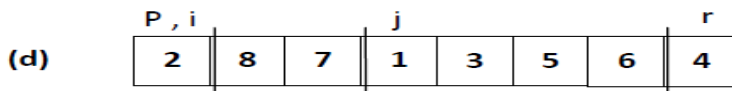
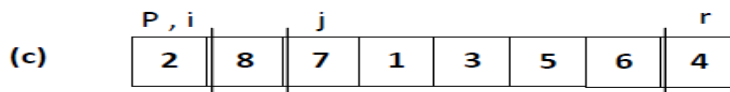
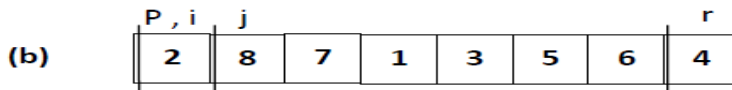
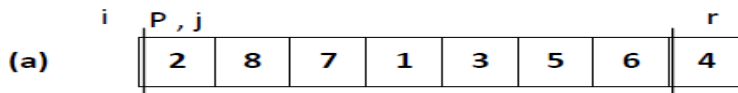
this partitioning, the pivot is in its final position. This is called the **partition** operation.

3. Recursively sort the sub-list of lesser elements and the sub-list of greater elements.

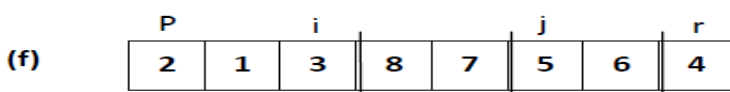
### Example

Sort given array using Quick Sort:

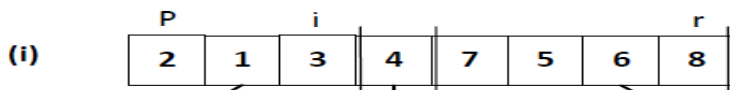
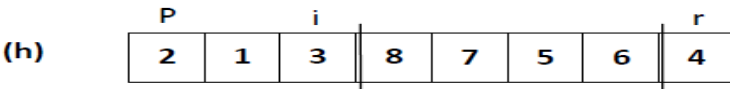
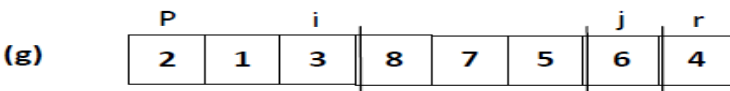
2	8	7	1	3	5	6	4
---	---	---	---	---	---	---	---



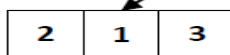
Exchange 8 and 1



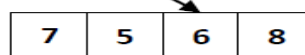
Exchange 7 and 3



Exchange 8 and 4



Left Sub Array



Right Sub Array

Apply same method for left and right sub array finally we will get sorted

## Algorithm for Quick Sort

### QUICK\_SORT (ARR, BEG, END)

Step 1: IF (BEG < END)  
    CALL PARTITION (ARR, BEG, END, LOC)  
    CALL QUICKSORT(ARR, BEG, LOC - 1)  
    CALL QUICKSORT(ARR, LOC + 1, END)  
    [END OF IF]  
Step 2: END

### PARTITION (ARR, BEG, END, LOC)

Step 1: [INITIALIZE] SET LEFT = BEG, RIGHT = END, LOC = BEG, FLAG =  
Step 2: Repeat Steps 3 to 6 while FLAG =

Step 3: Repeat while ARR[LOC] <= ARR[RIGHT] AND  
LOC != RIGHT SET RIGHT = RIGHT - 1  
    [END OF LOOP]

Step 4: IF LOC = RIGHT  
    SET FLAG = 1  
    ELSE IF ARR[LOC] > ARR[RIGHT]  
        SWAP ARR[LOC] with ARR[RIGHT]  
        SET LOC = RIGHT  
    [END OF IF]

Step 5: IF FLAG = 0  
    Repeat while ARR[LOC] >= ARR[LEFT] AND LOC != LEFT  
    SET LEFT = LEFT + 1  
    [END OF LOOP]

Step 6: IF LOC = LEFT  
    SET FLAG = 1  
    ELSE IF ARR[LOC] < ARR[LEFT]  
        SWAP ARR[LOC] with ARR[LEFT]  
        SET LOC = LEFT  
    [END OF IF]  
    [END OF IF]

Step 7: [END OF LOOP]  
Step 8: END

### Advantages:

- Extremely fast  $O(n \log_2 n)$
- Gives good results when an array is in random order.

- Quick sort can be used to sort arrays of small size, medium size, or large size.

### Disadvantages:

- Algorithm is very complex
- In worst case of quick sort algorithm, the time efficiency is very poor which is very much likely to selection sort algorithm i.e)  $n(\log_2 n)$ .

### Program

```
#include <stdio.h>
int partition(int a[], int beg, int end);
void quickSort(int a[], int beg, int end);
void main()
{
    int i;
    int arr[10]={90,23,101,45,65,23,67,89,34,23};
    quickSort(arr, 0, 9);
    printf("\n The sorted array is: \n");
    for(i=0;i<10;i++)
        printf(" %d\t", arr[i]);
}
int partition(int a[], int beg, int end)
{
    int left, right, temp, loc, flag;
    loc = left = beg;
    right = end;
    flag = 0;
    while(flag != 1)
    {
        while((a[loc] <= a[right]) && (loc!=right))
            right--;
        if(loc==right)
            flag = 1;
        else if(a[loc]>a[right])
        {
```

```
    temp = a[loc];
    a[loc] = a[right];
    a[right] = temp;
    loc = right;
}
if(flag!=1)
{
    while((a[loc] >= a[left]) && (loc!=left))
    left++;
    if(loc==left)
    flag =1;
    else if(a[loc] <a[left])
    {
        temp = a[loc];
        a[loc] = a[left];
        a[left] = temp;
        loc = left;
    }
}
}
```

```
void quickSort(int a[], int beg, int end)
{

    int loc;
    if(beg<end)
    {
        loc = partition(a, beg, end);
        quickSort(a, beg, loc-1);
        quickSort(a, loc+1, end);
    }
}
```

## Output:

The sorted array is:

23  
23  
23  
34  
45  
65  
67  
89  
90  
101

### Complexity of Quick Sort

Pass 1 will have  $n$  comparisons. Pass 2 will have  $2*(n/2)$  comparisons. In the subsequent passes will have  $4*(n/4)$ ,  $8*(n/8)$  comparisons and so on. The total comparisons involved in this case would be  $O(n)+O(n)+O(n)+\dots+s$ . The value of expression will be  $O(n \log n)$ . Thus time complexity of quick sort is  $O(n \log n)$ . Space required by quick sort is very less, only  $O(n \log n)$  additional space is required.

### 2.2.6 RADIX SORT

Radix sort is one of the sorting algorithms used to sort a list of integer numbers in order. In radix sort algorithm, a list of integer numbers will be sorted based on the digits of individual numbers. Sorting is performed from **least significant digit to the most significant digit**.

Radix sort algorithm requires the number of passes which are equal to the number of digits present in the largest number among the list of numbers. For example, if the largest number is a 3 digit number then that list is sorted with 3 passes.

#### Step by Step Process

The Radix sort algorithm is performed using the following steps...

- **Step 1** - Define 10 queues each representing a bucket for each digit from 0 to 9.
- **Step 2** - Consider the least significant digit of each number in the list which is to be sorted.
- **Step 3** - Insert each number into their respective queue based on the least significant digit.
- **Step 4** - Group all the numbers from queue 0 to queue 9 in the order they have inserted into their respective queues.
- **Step 5** - Repeat from step 3 based on the next least significant digit.
- **Step 6** - Repeat from step 2 until all the numbers are grouped based on the most significant digit.



### Algorithm for Radix Sort

#### Algorithm for RadixSort (ARR, N)

- Step 1: Find the largest number in ARR as LARGE  
Step 2: [INITIALIZE] SET NOP = Number of digits in LARGE  
Step 3: SET PASS = 0  
Step 4: Repeat Step 5 while PASS <= NOP-1  
Step 5: SET I = 0 and INITIALIZE buckets  
Step 6: Repeat Steps 7 to 9 while I < N-1  
Step 7: SET DIGIT = digit at PASSth place in A[I]  
Step 8: Add A[I] to the bucket numbered DIGIT  
Step 9: INCREMENT bucket count for bucket numbered DIGIT  
[END OF LOOP]  
Step 10: Collect the numbers in the bucket  
[END OF LOOP]  
Step 11: END

**Example** Sort the numbers given below using radix sort.

345, 654, 924, 123, 567, 472, 555, 808, 911

In the first pass, the numbers are sorted according to the digit at ones place. The buckets are pictured upside down as shown below.

Number	0	1	2	3	4	5	6	7	8	9
345						345				
654					654					
924					924					
123				123						
567								567		
472			472							
555						555				
808									808	
911		911								

After this pass, the numbers are collected bucket by bucket. The new list thus formed is used as an input for the next pass. In the second pass, the numbers are sorted according to the digit at the tens place. The buckets are pictured upside down.

Number	0	1	2	3	4	5	6	7	8	9
911		911								
472								472		
123			123							
654						654				
924			924							
345					345					
555						555				
567							567			
808	808									

In the third pass, the numbers are sorted according to the digit at the hundreds place. The buckets are pictured upside down.

Number	0	1	2	3	4	5	6	7	8	9
808									808	
911										911
123		123								
924										924
345				345						
654							654			
555						555				
567						567				
472					472					

The numbers are collected bucket by bucket. The new list thus formed is the final sorted result. After the third pass, the list can be given as

123, 345, 472, 555, 567, 654, 808, 911, 924.

### Advantages:

- Radix sort algorithm is well known for its fastest sorting algorithm for numbers and even for strings of letters.
- Radix sort algorithm is the most efficient algorithm for elements which are arranged in descending order in an array.

### Disadvantages:

- Radix sort takes more space than other sorting algorithms.
- Poor efficiency for most elements which are already arranged in ascending order in an array.
- When Radix sort algorithm is applied on very small sets of data (numbers or strings), then algorithm runs in  $O(n)$  asymptotic time.

## Program

```
#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++)
    {
        // sort the numbers according to the digit at passth place
        remainder = (a[i]/divisor)%10;
        bucket[remainder][bucket_count[remainder]] = a[i];
        bucket_count[remainder] += 1;
    }
    // collect the numbers after PASS pass
    i=0;
    for(k=0;k<10;k++)
    {
        for(j=0;j<bucket_count[k];j++)
        {
            a[i] = bucket[k][j];
            i++;
        }
    }
    divisor *= 10;
}
}

```

### Output:

The sorted array is:

23  
23  
23  
34  
45  
65  
67  
89

### Complexity of Radix Sort

To sort an unsorted list with 'n' number of elements, Radix sort algorithm needs the following complexities...

- Worst Case :  $O(n)$
- Best Case :  $O(n)$
- Average Case :  $O(n)$

### 2.2.7 HEAP SORT

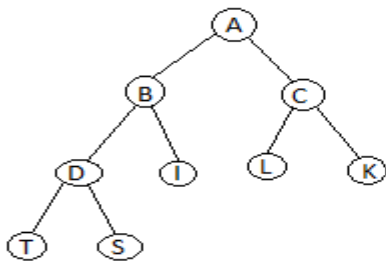
#### A) Terms in Heap:

##### a) Heap:

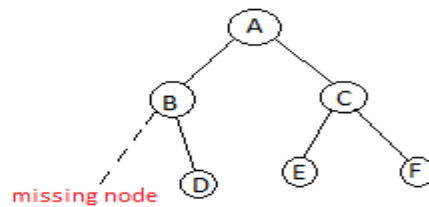
Heap is a special tree-based data structure that satisfies the following special heap properties

##### b) Shape Property:

Heap data structure is always a complete Binary Tree, which means all levels of the tree are fully filled.



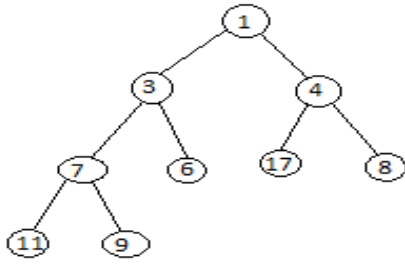
Complete Binary Tree



In-Complete Binary Tree

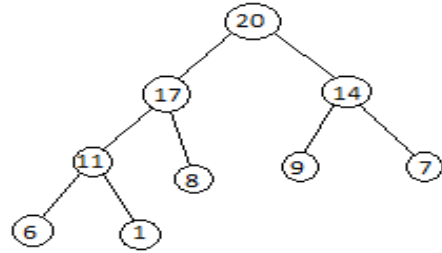
##### c) Heap Property:

All nodes are either (greater than or equal to) or (less than or equal to) each of its children. If the parent nodes are greater than their children, heap is called a Max-Heap, and if the parent nodes are small than their child nodes, heap is called Min-Heap.



**Min-Heap**

In min-heap, first element is the smallest. So when we want to sort a list in ascending order, we create a Min-heap from that list, and pick the first element, as it is the smallest, then we repeat the process with remaining elements.



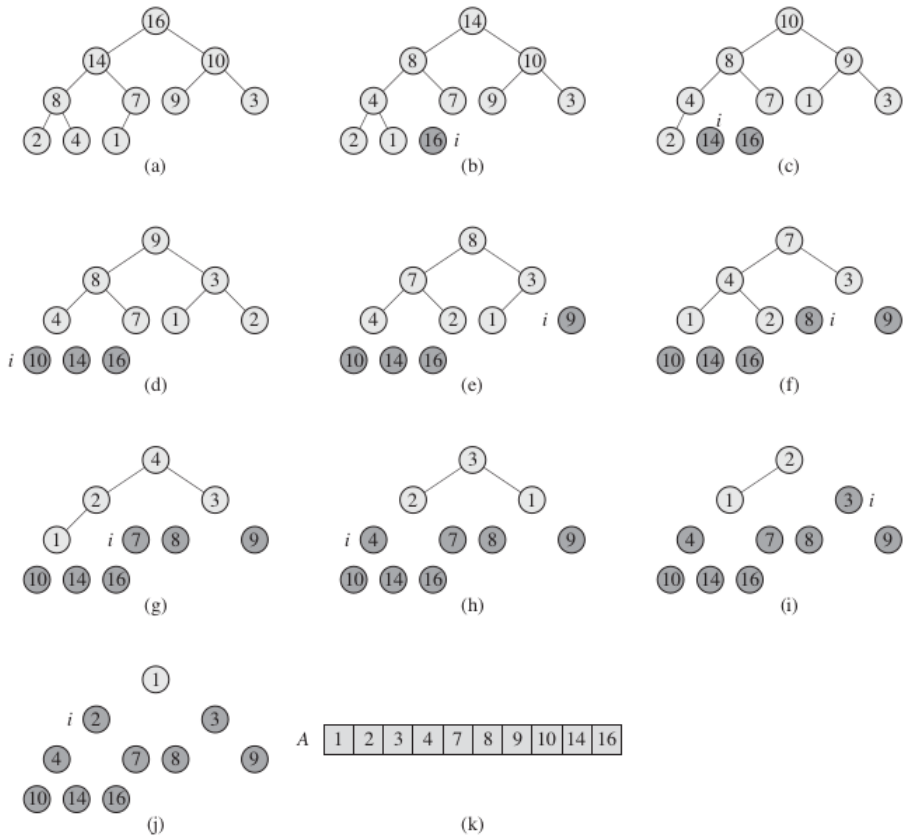
**Max-Heap**

In max-heap, the first element is the largest, hence it is used when we need to sort a list in descending order.

### **B) Working of Heap Sort:**

Initially on receiving an unsorted list, the first step in heap sort is to create a Heap data structure (Max-Heap or Min-Heap). Once heap is built, the first element of the Heap is either largest or smallest (depending upon Max-Heap or min-Heap), so put the first element of the heap in array. Then again make heap using the remaining elements, to again pick the first element of the heap and put it into the array. Keep on doing the same repeatedly until we have the complete sorted list in array.

**Example 2 : 4 ,1,3 ,2 ,16,9,10,14, 8,7**



**Operations on the heap :**

**1. Inserting An Element into Heap :**

The elements are always inserted at the bottom of the original heap. After insertion, the heap remains complete but the order property is not followed so we use an UPHEAP or HEAPIFY operation. This involves moving the elements upward from the last position where it satisfies the order property. If the value of last node is greater than its parent, exchange it's value with it's parent and repeat the process with each parent node until the order property is satisfied.

**2. Deleting an Element from Heap :**

Elements are always deleted from the root of the heap.

**Algorithm for insertion of element :**  
**INHEAP (TREE, N, ITEM)**

A heap H with N elements is stored in the array TREE, and an item of information is given. This procedure inserts ITEM as a new element of H. PTR gives the location of ITEM as it rises in the tree, and PAR denotes the location of the parent of ITEM.

- |  |   |
|--|---|
| 1. Set $N = N + 1$ and $PTR = N$       | [Add new node to H and initialize PTR]. |
| 2. Repeat Steps 3 to 6 while $PTR < 1$ | [Find location to insert ITEM].         |
| 3. Set $PAR = [PTR/2]$ .               | [Location of parent node].              |
| 4. If $ITEM \leq TREE[PAR]$ , then :   |   |
| Set $TREE[PTR] = ITEM$ , and return.   |   |
| [End of If Structure].                 |   |
| 5. Set $TREE[PTR] = TREE[PAR]$ .       | [Moves node down]                       |
| 6. Set $PTR = PAR$                     | [Updates PTR]                           |
| [End of step 2 loop].                  |   |
| 7. Set $TREE[1] = ITEM$                | [Assign ITEM as a root of H].           |
| 8. Exit                                |   |

### Algorithm for deletion of element :

#### DELHEAP(TREE,N,ITEM)

A heap H with N elements is stored in the array TREE. This procedure assigns the root  $TREE[1]$  of H to the variable ITEM and then reheaps the remaining elements. The variable LAST saves the value of the original last node of H. The pointers PTR, LEFT and RIGHT give the locations of LAST and its left and right children as LAST sinks in the tree.

- |   |                          |
|---|--------------------------|
| 1. Set $ITEM = TREE[1]$ .   | [Removes root of H].     |
| 2. Set $LAST = TREE[N]$ and $N = N - 1$                           | [Removes last node of H] |
| 3. Set $PTR = 1$ , $LEFT = 2$ and $RIGHT = 3$                     | [Initialize Pointers]    |
| 4. Repeat Steps 5 to 7 while $RIGHT \leq N$ :                     |                          |
| 5. If $LAST \geq TREE[LEFT]$ and $LAST \geq TREE[RIGHT]$ , then : |                          |
| 6. If $TREE[RIGHT] \leq TREE[L$                                   |                          |
| EFT] and $PTR = LEFT$   |                          |
| Set $TREE[PTR] = TREE[LEFT]$                                      |                          |
| and $PTR = LEFT$ . Else   |                          |
| Set $TREE[PTR] = TREE[RIGHT]$ and                                 |                          |
| $PTR = RIGHT$ . [End of If structure]                             |                          |
| 7. Set $LEFT = 2 * PTR$ and $RIGHT = LEFT + 1$                    |                          |
| [End of Step 4 loop].   |                          |



8. If  $LEFT = N$  and if  $LAST \ M \ TREE[LEFT]$ , then  $Set \ PTR = LEFT$ .
9.  $Set \ TREE[PTR] = LAST$
10. Exit

#### Advantages:

- Heap Sort is very fast and is widely used for sorting.
- Heap sort algorithm can be used to sort large sets of data.

#### Disadvantages:

- Heap sort is not a Stable sort, and requires a constant space for sorting a list.
- Heap sort algorithm's worst case comes with the running time of  $O(n \log (n))$  which is more likely to merge sort algorithm.

#### Program

```
#include<stdio.h>
int temp;

void heapify(int arr[], int size, int i)
{
    int largest = i;
    int left = 2*i + 1;
    int right = 2*i + 2;

    if (left < size && arr[left] >arr[largest])
        largest = left;

    if (right < size && arr[right] > arr[largest])
        largest = right;

    if (largest != i)
    {
        temp = arr[i];
```

```
    arr[i]= arr[largest];  
    arr[largest] = temp;  
    heapify(arr, size, largest);  
}  
}
```

```
void heapSort(int arr[], int size)
```

```
{  
int i;  
for (i = size / 2 - 1; i >= 0; i--)  
    heapify(arr, size, i);  
for (i=size-1; i>=0; i--)  
{  
    temp = arr[0];  
    arr[0]= arr[i];  
    arr[i] = temp;  
    heapify(arr, i, 0);  
}  
}
```

```
void main()
```

```
{  
int arr[] = {1, 10, 2, 3, 4, 1, 2, 100,23, 2};  
int i;  
int size = sizeof(arr)/sizeof(arr[0]);  
  
    heapSort(arr, size);  
  
    printf("printing sorted elements\n");  
for (i=0; i<size; ++i)  
    printf("%d\n",arr[i]);  
}
```

## Output:

printing sorted elements

```
1
1
2
2
2
3
4
10
23
100
```

## Complexity of Heap Sort

To sort an unsorted list with 'n' number of elements, following are the complexities...

**Worst Case :  $O(n \log n)$**

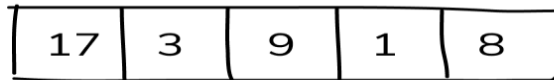
**Best Case :  $O(n \log n)$**

**Average Case :  $O(n \log n)$**

## 2.2.8 SHELL SORT

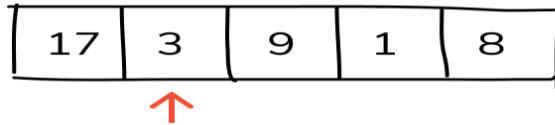
Shell sort algorithm is **very similar to that of the Insertion sort algorithm**. In case of Insertion sort, we move elements one position ahead to insert an element at its correct position. Whereas here, Shell sort starts by sorting pairs of elements far apart from each other, then progressively reducing the gap between elements to be compared. Starting with far apart elements, it can move some out-of-place elements into the position faster than a simple nearest-neighbor exchange.

Here is an example to help you understand the working of Shell sort on array of elements name  $A = \{17, 3, 9, 1, 8\}$



**Comparisons:**

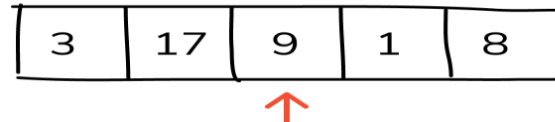
3 < 17 ? Yes, so swap



**Comparisons:**

9 < 17 ? Yes, so swap

9 < 3 ? No

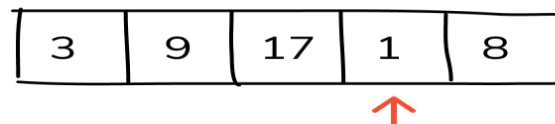


**Comparisons:**

1 < 17 ? Yes, so swap

1 < 9 ? Yes, so swap

1 < 3 ? Yes, so swap

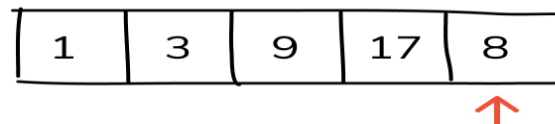


**Comparisons:**

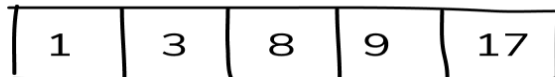
8 < 17 ? Yes, so swap

8 < 9 ? Yes, so swap

8 < 3 ? No



Remaining comparison are not required as we know for sure that elements on the left hand side of 3 are less than 3



### Algorithm for shell sort

**Shell\_Sort(Arr, n)**

Step 1: SET FLAG = 1, GAP\_SIZE = N

Step 2: Repeat Steps 3 to 6 while FLAG = 1 OR GAP\_SIZE > 1

Step 3: SET FLAG = 0

Step 4: SET GAP\_SIZE = (GAP\_SIZE + 1) / 2

Step 5: Repeat Step 6 for I = 0 to I < (N - GAP\_SIZE)

Step 6: IF Arr[I + GAP\_SIZE] > Arr[I]

SWAP Arr[I + GAP\_SIZE], Arr[I]

SET FLAG = 0

Step 7: END

**Advantages:**

- Shell sort algorithm is only efficient for finite number of elements in an array.
- Shell sort algorithm is 5.32 x faster than bubble sort algorithm.

### Disadvantages:

- Shell sort algorithm is complex in structure and bit more difficult to understand.
- Shell sort algorithm is significantly slower than the merge sort, quick sort and heap sort algorithms.

### Program

```

#include <stdio.h>
void shellsort(int arr[], int num)
{
    int i, j, k, tmp;
    for (i = num / 2; i > 0; i = i / 2)
    {
        for (j = i; j < num; j++)
        {
            for(k = j - i; k >= 0; k = k - i)
            {
                if (arr[k+i] >= arr[k])
                    break;
                else
                {
                    tmp = arr[k];
                    arr[k] = arr[k+i];
                    arr[k+i] = tmp;
                }
            }
        }
    }
}
int main()
{

```

```

int arr[30];
int k, num;
printf("Enter total no. of elements : ");
scanf("%d", &num);
printf("\nEnter %d numbers: ", num);

for (k = 0 ; k < num; k++)
{
scanf("%d", &arr[k]);
}
shellsort(arr, num);
printf("\n Sorted array is: ");
for (k = 0; k < num; k++)
printf("%d ", arr[k]);
return 0;
}

```

### Output:

Enter total no. of elements : 6

Enter 6 numbers: 3

2

4

10

2

1

**Sorted array is: 1 2 2 3 4 10**

### Complexity of Shell Sort

Time complexity of above implementation of shellsort is  $O(n^2)$ . In the above implementation, gap is reduced by half in every iteration.

## 2.2.7 TABLE OF COMPARISON OF ALL SORTING TECHNIQUES

<b>Time complexity of various sorting algorithms</b>
--

S.No.	Algorithm	Worst Case	Average Case	Best Case
1.	<b>Selection Sort</b>	$O(n^2)$	$O(n^2)$	$O(n^2)$
2.	<b>Bubble Sort</b>	$O(n^2)$	$O(n^2)$	$O(n^2)$
3.	<b>Insertion Sort</b>	$O(n^2)$	$O(n^2)$	$n - 1$
4.	<b>Quick Sort</b>	$O(n^2)$	$\log_2 n$	$\log_2 n$
5.	<b>Heap Sort</b>	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
6.	<b>Merge Sort</b>	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
7.	<b>Radix Sort</b>	$O(n^2)$	$O(n \log n)$	$O(n \log n)$
8.	<b>Shell sort</b>	$O(n \log^2 n)$	–	–

## 2.3 SUMMERY

Sorting deals with sorting the data stored in the memory, whereas external sorting deals with sorting the data stored in files.

In bubble sorting, consecutive adjacent pairs of elements in the array are compared with each other.

Insertion sort works by moving the current data element past the already sorted values and repeatedly interchanging it with the preceding value until it is in the correct place.

Selection sort works by finding the smallest value and placing it in the first position. It then finds the second smallest value and places it in the second position. This procedure is repeated until the whole array is sorted.

Merge sort is a sorting algorithm that uses the divide, conquer, and combine algorithmic paradigm. *Divide* means partitioning the  $n$ -element array to be sorted into two sub-arrays of  $n/2$  elements in each sub-array. *Conquer* means sorting the two sub-arrays recursively using merge sort. *Combine* means merging the two sorted sub-arrays of size  $n/2$  each to produce a sorted array of  $n$  elements. The running time of merge sort in average case and worst case can be given as  $O(n \log n)$ .

Quick sort works by using a divide-and-conquer strategy. It selects a pivot element and rearranges the elements in such a way that all elements less than pivot appear before it and all elements greater than pivot appear after it.

Radix sort is a linear sorting algorithm that uses the concept of sorting names in alphabetical order.

Heap sort sorts an array in two phases. In the first phase, it builds a heap of the given array. In the second phase, the root element is deleted repeatedly and inserted into an array.

Shell sort is considered as an improvement over insertion sort, as it compares elements separated by a gap of several positions.

## 2.4 MODEL QUESTIONS

1. Define sorting. What is the importance of sorting?
2. What are the different types of sorting techniques? Which sorting technique has the least worst case?
3. Explain the difference between bubble sort and quick sort. Which one is more efficient?
4. Sort the elements 77, 49, 25, 12, 9, 33, 56, 81 using
  - (a) insertion sort (b) selection sort
  - (b) bubble sort (d) merge sort
  - (e) quick sort (f) radix sort
  - (g) shell sort
5. Compare heap sort and quick sort.

## 2.5 LIST OF REFERENCES

<https://www.javatpoint.com/>  
<https://www.studytonight.com>  
<https://www.tutorialspoint.com>  
<https://www.geeksforgeeks.org/heap-sort/>  
<https://www.programiz.com/dsa/heap-sort>  
<https://www.2braces.com/data-structures>



## Unit 2: CHAPTER-3

# Searching Techniques

3.0 Objective

3.1 Introduction to Searching

3.2 Searching Techniques

3.2.1 Linear Search /Sequential Search

3.2.2 Binary Search

3.3 Summary

3.4 Model Questions

3.5 List of References

### 3.0 OBJECTIVE

- To study basic concepts of searching.
- To study different searching techniques like linear search and binary search.
- Comparisons of searching methods and their time complexity

### 3.1 INTRODUCTION TO SEARCHING

Searching is the process of finding some particular element in the list. If the element is present in the list, then the process is called successful and the process returns the location of that element, otherwise the search is called unsuccessful.

### 3.2 SEARCHING TECHNIQUES

There are two popular search methods that are widely used in order to search some item into the list. However, choice of the algorithm depends upon the arrangement of the list.

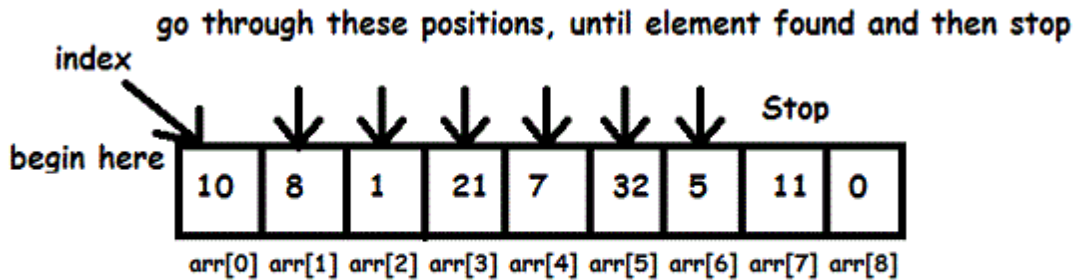
- Linear Search
- Binary Search

#### 3.2.1 LINEAR SEARCH / SEQUENTIAL SEARCH

Linear search, also called as *sequential search*, is a very simple method used for searching an array for a particular value. It works by comparing the value to be searched with every element of the array one by one in a sequence until a match is found. Linear search is mostly used to search an unordered list of elements (array in which data elements are not sorted).

For example, if an array A[] is declared and initialized as,

```
int A[] = {10, 8, 1, 21, 7, 32, 5, 11, 0};
```



**Element to search : 5**

and the value to be searched is  $VAL = 5$ , then searching means

to find whether the value '5' is present in the array or not. If yes, then it returns the position of its occurrence. Here,  $POS = 6$  (index starting from 0).

#### **Algorithm for linear search**

**LINEAR\_SEARCH(A, N, VAL)**

Step 1: [INITIALIZE] SET  $POS = -1$

Step 2: [INITIALIZE] SET  $I = 1$

while

Step 3: Repeat Step 4  $I \leq N$

Step 4:       IF  $A[I] = VAL$

SET  $POS = I$

PRINT  $POS$

Go to Step 6

[END OF IF]

SET  $I = I + 1$

[END OF LOOP]

Step 5: IF  $POS = -1$

PRINT "VALUE IS NOT PRESENT IN THE ARRAY"

[END OF IF]

Step 6: EXIT

The algorithm for linear search.

In Steps 1 and 2 of the algorithm, we initialize the value of POS and I.

In Step 3, a while loop is executed that would be executed till I is less than N (total number of elements in the array).

In Step 4, a check is made to see if a match is found between the current array element and VAL.

If a match is found, then the position of the array element is printed, else the value of I is incremented to match the next element with VAL.

However, if all the array elements have been compared with VAL and no match is found, then it means that VAL is not present in the array.

### Programming Example

Write a program to search an element in an array using the linear search technique.

```
#include<stdio.h>
void main ()
{
    int a[10] = {10, 23, 40, 1, 2, 0, 14, 13, 50, 9};
    int item, i, flag;
    printf("\nEnter Item which is to be searched\n");
    scanf("%d",&item);
    for (i = 0; i < 10; i++)
    {
        if(a[i] == item)
        {
            flag = i+1;
            break;
        }
        else
            flag = 0;
    }
    if(flag != 0)
        printf("\nItem found at location %d\n",flag);
}
```

```
lse
```

```
printf("\nItem not found\n");
```

### Output:

```
Enter Item which is to be searched
```

```
20
```

```
Item not found
```

```
Enter Item which is to be searched
```

```
23
```

```
Item found at location 2
```

### Advantages of a linear search

- **Will perform fast searches of small to medium lists.** With today's powerful computers, small to medium arrays can be searched relatively quickly.
- **The list does not need to be sorted.** Unlike a binary search, linear searching does not require an ordered list.
- **Not affected by insertions and deletions.** As the linear search does not require the list to be sorted, additional elements can be added and deleted. As other searching algorithms may have to reorder the list after insertions or deletions, this may sometimes mean a linear search will be more efficient.

### Disadvantages of a linear search

- **Slow searching of large lists.** For example, when searching through a database of everyone in the Northern Ireland to find a particular name, it might be necessary to search through 1.8 million names before you found the one you wanted. This speed disadvantage is why other search methods have been developed.

### Complexity of Linear Search Algorithm

The time complexity of the linear search is  $O(N)$  because each element in an array is compared only once.

## 3.2.2 BINARY SEARCH

Binary search is the search technique which works efficiently on the sorted lists. Hence, in order to search an element into some list by using binary search technique, we must ensure that the list is sorted.

Binary search follows divide and conquer approach in which, the list is divided into two halves and the item is compared with the middle element of the list. If the match is found then, the location of middle element is returned otherwise, we search into either of the halves depending upon the result produced through the match.

Let us consider an array  $arr = \{1, 5, 7, 8, 13, 19, 20, 23, 29\}$ . Find the location of the item 23 in the array.

**In 1<sup>st</sup> step :**

1.  $BEG = 0$
2.  $END = 8$
3.  $MID = 4$
4.  $a[mid] = a[4] = 13 < 23$ , therefore

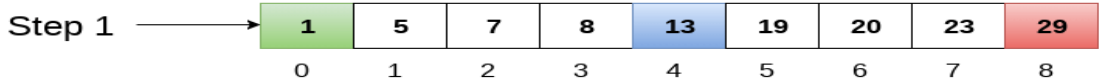
**in Second step:**

1.  $Beg = mid + 1 = 5$
2.  $End = 8$
3.  $mid = \lfloor 13/2 \rfloor = 6$
4.  $a[mid] = a[6] = 20 < 23$ , therefore;

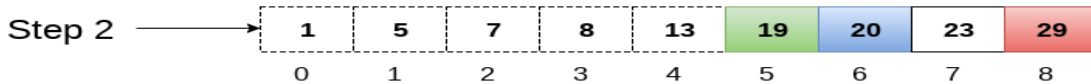
**in third step:**

1.  $beg = mid + 1 = 7$
2.  $End = 8$
3.  $mid = \lfloor 15/2 \rfloor = 7$
4.  $a[mid] = a[7]$
5.  $a[7] = 23 = item$ ;
6. therefore, set location = mid;
7. The location of the item will be 7.

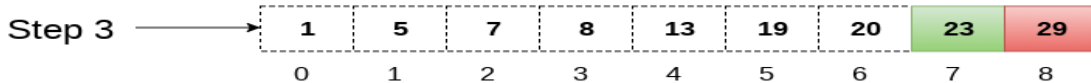
Item to be searched = 23



$a[\text{mid}] = 13$   
 $13 < 23$   
 $\text{beg} = \text{mid} + 1 = 5$   
 $\text{end} = 8$   
 $\text{mid} = (\text{beg} + \text{end})/2 = 13 / 2 = 6$



$a[\text{mid}] = 20$   
 $20 < 23$   
 $\text{beg} = \text{mid} + 1 = 7$   
 $\text{end} = 8$   
 $\text{mid} = (\text{beg} + \text{end})/2 = 15 / 2 = 7$



$a[\text{mid}] = 23$   
 $23 = 23$   
 $\text{loc} = \text{mid}$

Return location 7

Binary search algorithm is given below.

### Algorithm for binary search

**BINARY\_SEARCH(A, lower\_bound, upper\_bound, VAL)**

Step 1: [INITIALIZE] SET BEG = lower\_bound

END = upper\_bound, POS = - 1

Step 2: Repeat Steps 3 and 4 while BEG <= END

Step 3: SET MID = (BEG + END)/2

Step 4: IF A[MID] = VAL

SET POS = MID

PRINT POS

Go to Step 6

ELSE IF A[MID] > VAL

SET END = MID - 1

ELSE

SET BEG = MID + 1

[END OF IF]

[END OF LOOP]

Step 5: IF POS = -1

```
PRINT "VALUE IS NOT PRESENT IN THE ARRAY"  
[END OF IF]
```

Step 6: EXIT

### The algorithm for binary search.

In Step 1, we initialize the value of variables, BEG, END, and POS.

In Step 2, a while loop is executed until BEG is less than or equal to END.

In Step 3, the value of MID is calculated.

In Step 4, we check if the array value at MID is equal to VAL (item to be searched in the array). If a match is found, then the value of POS is printed and the algorithm exits.

However, if a match is not found, and if the value of A[MID] is greater than VAL, the value of END is modified, otherwise if A[MID] is greater than VAL, then the value of BEG is altered.

In Step 5, if the value of POS = -1, then VAL is not present in the array and an appropriate message is printed on the screen before the algorithm exits.

### Programming Example

Write a program to search an element in an array using binary search.

```
#include<stdio.h>  
int binarySearch(int[], int, int, int);  
void main ()  
{  
    int arr[10] = { 16, 19, 20, 23, 45, 56, 78, 90, 96, 100};  
    int item, location=-1;  
    printf("Enter the item which you want to search ");  
    scanf("%d",&item);  
    location = binarySearch(arr, 0, 9, item);  
    if(location != -1)  
        printf("Item found at location %d",location);  
    else  
        printf("Item not found");
```

```
binarySearch(int a[], int beg, int end, int item)
```

```
int mid;
```

```
if(end >= beg)
```

```
mid = (beg + end)/2;
```

```
if(a[mid] == item)
```

```
{
```

```
return mid+1;
```

```
}
```

```
else if(a[mid] < item)
```

```
{
```

```
return binarySearch(a,mid+1,end,item);
```

```
}
```

```
else
```

```
{
```

```
return binarySearch(a,beg,mid-1,item);
```

```
}
```

```
return -1;
```

**Output:**

Enter the item which you want to search

19

Item found at location 2

### Advantages-

The advantages of binary search algorithm are-

- It eliminates half of the list from further searching by using the result of each comparison.
- It indicates whether the element being searched is before or after the current position in the list.



- This information is used to narrow the search.
- For large lists of data, it works significantly better than linear search.

### Disadvantages-

The disadvantages of binary search algorithm are-

- It employs recursive approach which requires more stack space.
- Programming binary search algorithm is error prone and difficult.
- The interaction of binary search with memory hierarchy i.e. caching is poor.  
(because of its random access nature)

### Complexity of Binary Search Algorithm

As we dispose off one part of the search case during every step of binary search, and perform the search operation on the other half, this results in a worst case time complexity of  $O(\log 2N)$ .

### Comparison of Searching methods in Data Structures

Sequential Search	Binary Search
Time complexity is $O(n)$	Time complexity is $O(\log n)$
Finds the key present at first position in constant time	Finds the key present at centre position in constant time
Sequence of elements in the container does not affect.	The elements must be sorted in the container
Arrays and linked lists can be used to implement this	It cannot be implemented directly into the linked list. We need to change the basic rules of the list to implement this
Algorithm is iterative in nature	Algorithm technique is Divide and Conquer.
Algorithm is easy to implement, and requires less amount of code.	Algorithm is slightly complex. It takes more amount of code to implement.
$N$ number of comparisons are required for worst case.	$\log n$ number of comparisons are sufficient in worst case.

### 3.3 SUMMARY

Searching refers to finding the position of a value in a collection of values. Some of the popular searching techniques are linear search, binary search.

Linear search works by comparing the value to be searched with every element of the array one by one in a sequence until a match is found.

Binary search works efficiently with a sorted list. In this algorithm, the value to be searched is compared with the middle element of the array segment.

### **3.4 MODEL QUESTIONS**

1. Explain Linear Search with example?
2. Explain Binary Search with example?
3. Compare Linear and Binary Search?
4. Which technique of searching an element in an array would you prefer to use and in which situation?

### **3.5 LIST OF REFERENCES**

<https://www.javatpoint.com/>  
<https://www.studytonight.com>  
<https://www.tutorialspoint.com>  
<https://www.geeksforgeeks.org/heap-sort/>  
<https://www.programiz.com/dsa/heap-sort>  
<https://www.2braces.com/data-structures>

## Unit 1: CHAPTER 3

# Searching Techniques

3.0 Objective

3.1 Introduction to Searching

3.2 Searching Techniques

3.2.1 Linear Search /Sequential Search

3.2.2 Binary Search

3.3 Summary

3.4 Model Questions

3.5 List of References

### 3.0 OBJECTIVE

- To study basic concepts of searching.
- To study different searching techniques like linear search and binary search.
- Comparisons of searching methods and their time complexity

### 3.1 INTRODUCTION TO SEARCHING

Searching is the process of finding some particular element in the list. If the element is present in the list, then the process is called successful and the process returns the location of that element, otherwise the search is called unsuccessful.

### 3.2 SEARCHING TECHNIQUES

There are two popular search methods that are widely used in order to search some item into the list. However, choice of the algorithm depends upon the arrangement of the list.

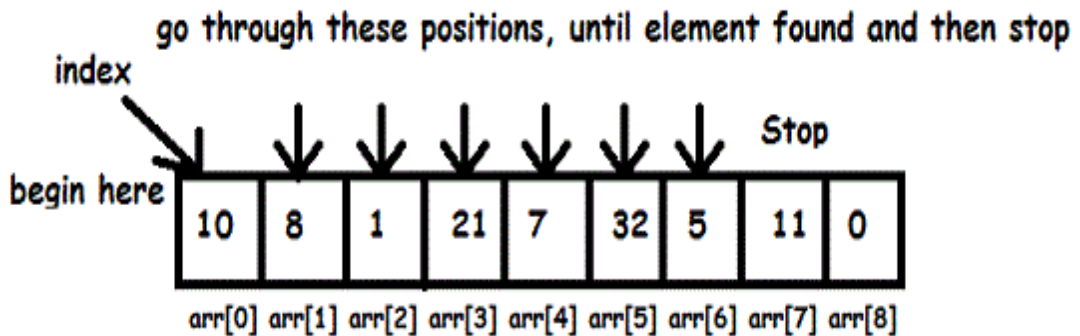
- Linear Search
- Binary Search

#### 3.2.1 LINEAR SEARCH / SEQUENTIAL SEARCH

Linear search, also called as *sequential search*, is a very simple method used for searching an array for a particular value. It works by comparing the value to be searched with every element of the array one by one in a sequence until a match is found. Linear search is mostly used to search an unordered list of elements (array in which data elements are not sorted).

For example, if an array A[] is declared and initialized as,

```
int A[] = {10, 8, 1, 21, 7, 32, 5, 11, 0};
```



**Element to search : 5**

and the value to be searched is  $VAL = 5$ , then searching means

to find whether the value '5' is present in the array or not. If yes, then it returns the position of its occurrence. Here,  $POS = 6$  (index starting from 0).

#### **Algorithm for linear search**

**LINEAR\_SEARCH(A, N, VAL)**

Step 1: [INITIALIZE]

SET POS = -1

Step 2: [INITIALIZE]

SET I = 1

while

Step 3: Repeat Step 4  $I \leq N$

Step 4: IF  $A[I] = VAL$

SET POS = I

PRINT POS

Go to Step 6

[END OF IF]

SET I = I + 1

[END OF LOOP]

Step 5: IF POS = -1

PRINT "VALUE IS NOT PRESENT IN THE ARRAY"

[END OF IF]

Step 6: EXIT

The algorithm for linear search.

In Steps 1 and 2 of the algorithm, we initialize the value of POS and I.

In Step 3, a while loop is executed that would be executed till I is less than N (total number of elements in the array).

In Step 4, a check is made to see if a match is found between the current array element and VAL.

If a match is found, then the position of the array element is printed, else the value of I is incremented to match the next element with VAL.

However, if all the array elements have been compared with VAL and no match is found, then it means that VAL is not present in the array.

### Programming Example

Write a program to search an element in an array using the linear search technique.

```
#include<stdio.h>
void main ()
{
    int a[10] = {10, 23, 40, 1, 2, 0, 14, 13, 50, 9};
    int item, i, flag;
    printf("\nEnter Item which is to be searched\n");
    scanf("%d",&item);
    for (i = 0; i < 10; i++)
    {
        if(a[i] == item)
        {
            flag = i+1;
            break;
        }
        else
            flag = 0;
    }
    if(flag != 0)
        printf("\nItem found at location %d\n",flag);
}
```

```
lse
```

```
printf("\nItem not found\n");
```

### Output:

```
Enter Item which is to be searched
```

```
20
```

```
Item not found
```

```
Enter Item which is to be searched
```

```
23
```

```
Item found at location 2
```

### Advantages of a linear search

- **Will perform fast searches of small to medium lists.** With today's powerful computers, small to medium arrays can be searched relatively quickly.
- **The list does not need to be sorted.** Unlike a binary search, linear searching does not require an ordered list.
- **Not affected by insertions and deletions.** As the linear search does not require the list to be sorted, additional elements can be added and deleted. As other searching algorithms may have to reorder the list after insertions or deletions, this may sometimes mean a linear search will be more efficient.

### Disadvantages of a linear search

- **Slow searching of large lists.** For example, when searching through a database of everyone in the Northern Ireland to find a particular name, it might be necessary to search through 1.8 million names before you found the one you wanted. This speed disadvantage is why other search methods have been developed.

### Complexity of Linear Search Algorithm

The time complexity of the linear search is  $O(N)$  because each element in an array is compared only once.

### 3.2.2 BINARY SEARCH

Binary search is the search technique which works efficiently on the sorted lists. Hence, in order to search an element into some list by using binary search technique, we must ensure that the list is sorted.

Binary search follows divide and conquer approach in which, the list is divided into two halves and the item is compared with the middle element of the list. If the match is found then, the location of middle element is returned otherwise, we search into either of the halves depending upon the result produced through the match.

Let us consider an array  $arr = \{1, 5, 7, 8, 13, 19, 20, 23, 29\}$ . Find the location of the item 23 in the array.

**In 1<sup>st</sup> step :**

5.  $BEG = 0$
6.  $END = 8$
7.  $MID = 4$
8.  $a[mid] = a[4] = 13 < 23$ , therefore

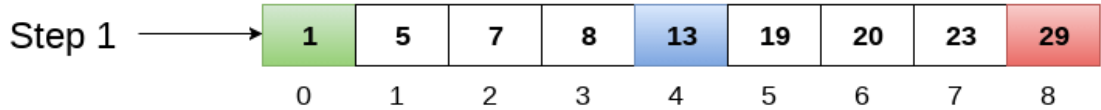
**in Second step:**

5.  $Beg = mid + 1 = 5$
6.  $End = 8$
7.  $mid = (5+8)/2 = 6$
8.  $a[mid] = a[6] = 20 < 23$ , therefore;

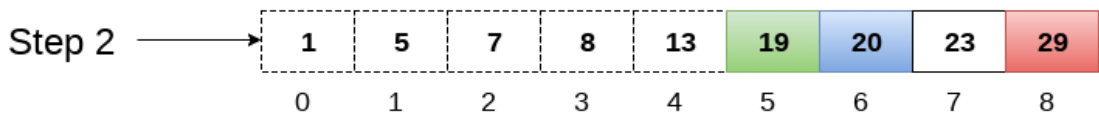
**in third step:**

8.  $beg = mid + 1 = 7$
9.  $End = 8$
10.  $mid = (7+8)/2 = 7$
11.  $a[mid] = a[7]$
12.  $a[7] = 23 = item$ ;
13. therefore, set location = mid;
14. The location of the item will be 7.

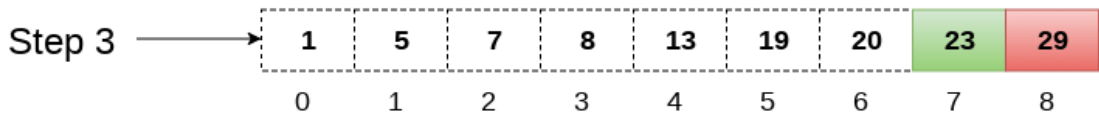
Item to be searched = 23



$a[\text{mid}] = 13$   
 $13 < 23$   
 $\text{beg} = \text{mid} + 1 = 5$   
 $\text{end} = 8$   
 $\text{mid} = (\text{beg} + \text{end})/2 = 13 / 2 = 6$



$a[\text{mid}] = 20$   
 $20 < 23$   
 $\text{beg} = \text{mid} + 1 = 7$   
 $\text{end} = 8$   
 $\text{mid} = (\text{beg} + \text{end})/2 = 15 / 2 = 7$



$a[\text{mid}] = 23$   
 $23 = 23$   
 $\text{loc} = \text{mid}$

Return location 7

Binary search algorithm is given below.

**Algorithm for binary search**



### **BINARY\_SEARCH(A, lower\_bound, upper\_bound, VAL)**

Step 1: [INITIALIZE]

SET BEG = lower\_bound

    END = upper\_bound,

POS = - 1

Step 2: Repeat Steps 3 and 4 while BEG <= END

Step 3: SET MID = (BEG + END)/2

Step 4: IF A[MID] = VAL

    SET POS = MID

    PRINT POS

    Go to Step 6

    ELSE IF A[MID] > VAL

        SET END = MID - 1

    ELSE

        SET BEG = MID + 1

    [END OF IF]

    [END OF LOOP]

Step 5: IF POS = -1

    PRINT "VALUE IS NOT PRESENT IN THE ARRAY"

    [END OF IF]

Step 6: EXIT

### **The algorithm for binary search.**

In Step 1, we initialize the value of variables, BEG, END, and POS.

In Step 2, a while loop is executed until BEG is less than or equal to END.

In Step 3, the value of MID is calculated.

In Step 4, we check if the array value at MID is equal to VAL (item to be searched in the array). If a match is found, then the value of POS is printed and the algorithm exits. However, if a match is not found, and if the value of A[MID] is greater than VAL, the value of END is modified, otherwise if A[MID] is greater than VAL, then the value of BEG is altered.

In Step 5, if the value of POS = -1, then VAL is not present in the array and an appropriate message is printed on the screen before the algorithm exits.

### **Programming Example**

Write a program to search an element in an array using binary search.

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

```
int binarySearch(int[], int, int, int);
```

```

void main ()
{
    int arr[10] = { 16, 19, 20, 23, 45, 56, 78, 90, 96, 100};
    int item, location=-1;
    printf("Enter the item which you want to search ");
    scanf("%d",&item);
    location = binarySearch(arr, 0, 9, item);
    if(location != -1)

        printf("Item found at location %d",location);
    else

        printf("Item not found");
}

int binarySearch(int a[], int beg, int end, int item)
{
    int mid;
    if(end >= beg)

        mid = (beg + end)/2;
        if(a[mid] == item)
        {
            return mid+1;
        }
        else if(a[mid] < item)
        {
            return binarySearch(a,mid+1,end,item);
        }
        else
        {
            return binarySearch(a,beg,mid-1,item);
        }
}

```

```
return -1;
```

**Output:**

Enter the item which you want to search

19

Item found at location 2

### **Advantages-**

The advantages of binary search algorithm are-

- It eliminates half of the list from further searching by using the result of each comparison.
- It indicates whether the element being searched is before or after the current position in the list.
- This information is used to narrow the search.
- For large lists of data, it works significantly better than linear search.

### **Disadvantages-**

The disadvantages of binary search algorithm are-

- It employs recursive approach which requires more stack space.
- Programming binary search algorithm is error prone and difficult.
- The interaction of binary search with memory hierarchy i.e. caching is poor.  
(because of its random access nature)

### **Complexity of Binary Search Algorithm**

As we dispose off one part of the search case during every step of binary search, and perform the search operation on the other half, this results in a worst case time complexity of  $O(\log_2 N)$ .

### **Comparison of Searching methods in Data Structures**

<b>Sequential Search</b>	<b>Binary Search</b>
Time complexity is $O(n)$	Time complexity is $O(\log n)$
Finds the key present at first position in constant time	Finds the key present at centre position in constant time
Sequence of elements in the container does not affect.	The elements must be sorted in the container
Arrays and linked lists can be used to implement this	It cannot be implemented directly into the linked list. We need to change the basic rules of the list to implement this
Algorithm is iterative in nature	Algorithm technique is Divide and Conquer.
Algorithm is easy to implement, and requires less amount of code.	Algorithm is slightly complex. It takes more amount of code to implement.
N number of comparisons are required for worst case.	Log n number of comparisons are sufficient in worst case.

### 3.3 SUMMARY

Searching refers to finding the position of a value in a collection of values. Some of the popular searching techniques are linear search, binary search.

Linear search works by comparing the value to be searched with every element of the array one by one in a sequence until a match is found.

Binary search works efficiently with a sorted list. In this algorithm, the value to be searched is compared with the middle element of the array segment.

### 3.4 MODEL QUESTIONS

5. Explain Linear Search with example?
6. Explain Binary Search with example?
7. Compare Linear and Binary Search?
8. Which technique of searching an element in an array would you prefer to use and in which situation?

### **3.5 LIST OF REFERENCES**

<https://www.javatpoint.com/>

<https://www.studytonight.com>

<https://www.tutorialspoint.com>

<https://www.geeksforgeeks.org/heap-sort/>

<https://www.programiz.com/dsa/heap-sort>

<https://www.2braces.com/data-structures>

## Unit 3:Hashing

### Chapter 4

#### 4.0 Objective

##### 4.1. Hashing

##### 4.2. Why we need Hashing?

##### 4.3.Universal Hashing

##### 4.4.Rehashing

##### 4.5.Hash Tables

##### 4.6.Why use HashTable?

##### 4.7.Application of Hash Tables:

##### 4.8.Methods of Hashing

###### 4.8.1. Hashing with Chaining

###### 4.8.2.Collision Resolution by Chaining

###### 4.8.3. Analysis of Hashing with Chaining:

##### 4.9.Hashing with Open Addressing

##### 4.10.Open Addressing Techniques

###### 4.10.1.Linear Probing.

###### 4.10.2.Quadratic Probing.

###### 4.10.3.Double Hashing.

##### 4.11.Hash Function

##### 4.12.Characteristics of Good Hash Function:

##### 4.13.Some Popular Hash Function

###### 4.13.1. Division Method

###### 4.13.2 Duplication Method

###### 4.13.3. Mid Square Method

###### 4.13.4.Folding Method

#### 4.0.Objective

This chapter would make you understand the following concepts:

- Different Hashing Techniques
- Address calculation Techniques
- Collision resolution techniques

#### 4.1 Hashing

Hashing is the change of a line of character into a normally more limited fixed-length worth or key that addresses the first string.

Hashing is utilized to list and recover things in a data set since it is quicker to discover the thing utilizing the briefest hashed key than to discover it utilizing the first worth. It is likewise utilized in numerous encryption calculations.

A hash code is produced by utilizing a key, which is an exceptional worth.

Hashing is a strategy where given key field esteem is changed over into the location of capacity area of the record by applying a similar procedure on it.

The benefit of hashing is that permits the execution season of fundamental activity to stay consistent in any event, for the bigger side.

#### **4.2 Why we need Hashing?**

Assume we have 50 workers, and we need to give 4 digit key to every representative (with respect to security), and we need subsequent to entering a key, direct client guide to a specific position where information is put away.

In the event that we give the area number as per 4 digits, we should hold 0000 to 9999 locations since anyone can utilize anybody as a key. There is a great deal of wastage.

To tackle this issue, we use hashing which will deliver a more modest estimation of the record of the hash table relating to the key of the client.

#### **4.3 Universal Hashing**

Leave  $H$  alone a limited assortment of hash works that map a given universe  $U$  of keys into the reach  $\{0, 1, \dots, m-1\}$ . Such an assortment is supposed to be all inclusive if for each pair of particular keys  $k, l \in U$ , the quantity of hash capacities  $h \in H$  for which  $h(k) = h(l)$  is all things considered  $|H|/m$ . As such, with a hash work haphazardly browsed  $H$ , the possibility of an impact between particular keys  $k$  and  $l$  is close to the opportunity  $1/m$  of a crash if  $h(k)$  and  $h(l)$  were arbitrarily and autonomously looked over the set  $\{0, 1, \dots, m-1\}$ .

#### **4.4.Rehashing**

On the off chance that any stage the hash table turns out to be almost full, the running time for the activities of will begin taking an excess of time, embed activity may fall flat in such circumstance, the most ideal arrangement is as per the following:

1. Make another hash table twofold in size.
2. Output the first hash table, register new hash worth and addition into the new hash table.
3. Free the memory involved by the first hash table.

Model: Consider embeddings the keys 10, 22, 31, 4, 15, 28, 17, 88 and 59 into a hash table of length  $m = 11$  utilizing open tending to with the essential hash work  $h'(k) = k \bmod m$ . Illustrate the aftereffect of embeddings these keys utilizing straight examining, utilizing quadratic testing with  $c_1=1$  and  $c_2=3$ , and utilizing twofold hashing with  $h_2(k) = 1 + (k \bmod (m-1))$ .

Arrangement: Using Linear Probing the last condition of hash table would be:

0	22
1	88
2	/
3	/
4	4
5	15
6	28
7	17
8	59
9	31
10	10

Using Quadratic Probing with  $c_1=1$ ,  $c_2=3$ , the final state of hash table would be  $h(k, i) = (h'(k) + c_1*i + c_2*i^2) \bmod m$  where  $m=11$  and  $h'(k) = k \bmod m$ .

0	22
1	88
2	/
3	17
4	4
5	/
6	28
7	59
8	15
9	31
10	10

Using Double Hashing, the final state of the hash table would be:



0	22
1	/
2	59
3	17
4	4
5	15
6	28
7	88
8	/
9	31
10	10

#### 4.5 Hash Tables

It is an assortment of things which are put away so as to make it simple to discover them later.

Each position in the hash table is called opening, can hold a thing and is named by a number worth beginning at 0.

The planning between a thing and a space where the thing should be in a hash table is known as a Hash Function. A hash Function acknowledges a key and returns its hash coding, or hash esteem.

Expect we have a bunch of whole numbers 54, 26, 93, 17, 77, 31. Our first hash work needed to be as "leftover portion strategy" just takes the thing and separation it by table size, returning remaining portion as its hash esteem for example

$h \text{ item} = \text{item} \% (\text{size of table})$

Let us say the size of table = 11, then

$54 \% 11 = 10$     $26 \% 11 = 4$     $93 \% 11 = 5$

$17 \% 11 = 6$     $77 \% 11 = 0$     $31 \% 11 = 9$

ITEM	HASH VALUE
54	10
26	4
93	5
17	6
77	0
31	9

0	1	2	3	4	5	6	7	8	9	10
77				26	93	17			31	54

Fig: Hash Table

Now when we need to search any element, we just need to divide it by the table size, and we get the hash value. So we get the  $O(1)$  search time.

Now taking one more element 44 when we apply the hash function on 44, we get  $(44 \% 11 = 0)$ , But 0 hash value already has an element 77. This Problem is called as Collision.

Collision: According to the Hash Function, two or more item would need in the same slot. This is said to be called as Collision.

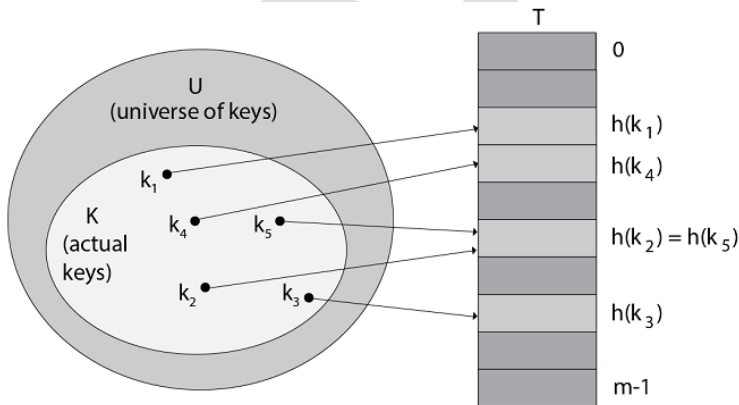


Figure: using a hash function  $h$  to map keys to hash-table slots. Because keys  $K_2$  and  $k_5$  map to the same slot, they collide.

#### 4.6. Why use HashTable?

1. If  $U$  (Universe of keys) is large, storing a table  $T$  of size  $[U]$  may be impossible.
2. Set  $k$  of keys may be small relative to  $U$  so space allocated for  $T$  will waste.

So Hash Table requires less storage. Indirect addressing element with key  $k$  is stored in slot  $k$  with hashing it is stored in  $h(k)$  where  $h$  is a hash fn and  $hash(k)$  is the value of key  $k$ . Hash fn required array range.

#### 4.7. Application of Hash Tables:

Some use of Hash Tables are:

1. Data set System: Specifically, those that are required effective arbitrary access. Generally, information base frameworks attempt to create between two sorts of access techniques: successive and arbitrary. Hash Table is an essential piece of proficient arbitrary access since they give an approach to find information in a consistent measure of time.
2. Image Tables: The tables used by compilers to keep up information about images from a program. Compilers access data about images much of the time. In this manner, it is fundamental that image tables be actualized productively.
3. Information Dictionaries: Data Structure that supports adding, erasing, and looking for information. Albeit the activity of hash tables and an information word reference are comparable, other Data Structures might be utilized to actualize information word references.
4. Cooperative Arrays: Associative Arrays comprise of information orchestrated so nth components of one exhibit compare to the nth component of another. Cooperative Arrays are useful for ordering a consistent gathering of information by a few key fields.

#### **4.8 .Methods of Hashing**

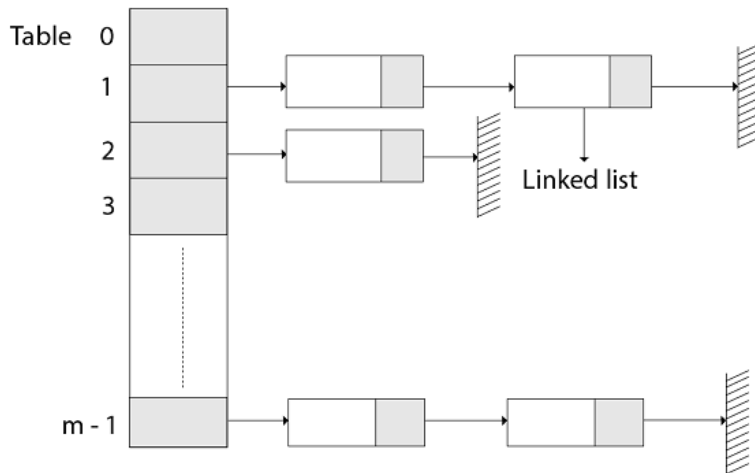
There are two main methods used to implement hashing:

- 2.4.1.Hashing with Chaining
- 2.4.2.Hashing with open addressing

##### **4.8.1.Hashing with Chaining**

In Hashing with Chaining, the component in  $S$  is put away in Hash table  $T [0...m-1]$  of size  $m$ , where  $m$  is to some degree bigger than  $n$ , the size of  $S$ . The hash table is said to have  $m$  spaces. Related with the hashing plan is a hash work  $h$  which is planning from  $U$  to  $\{0...m-1\}$ . Each key  $k \in S$  is put away in area  $T [h(k)]$ , and we say that  $k$  is hashed into opening  $h(k)$ . In the event that more than one key in  $S$  hashed into a similar opening, we have a crash.

In such case, all keys that hash into a similar space are put in a connected rundown related with that opening, this connected rundown is known as the chain at opening. The heap factor of a hash table is characterized to be  $\alpha = n/m$  it addresses the normal number of keys per opening. We normally work in the reach  $m = \theta(n)$ , so  $\alpha$  is typically a consistent by and large  $\alpha < 1$ .



#### 4.8.2. Collision Resolution by Chaining:

In anchoring, we place all the components that hash to a similar opening into a similar connected rundown, As fig shows that Slot  $j$  contains a pointer to the top of the rundown of all put away components that hash to  $j$ ; if there are no such components, space  $j$  contains NIL.

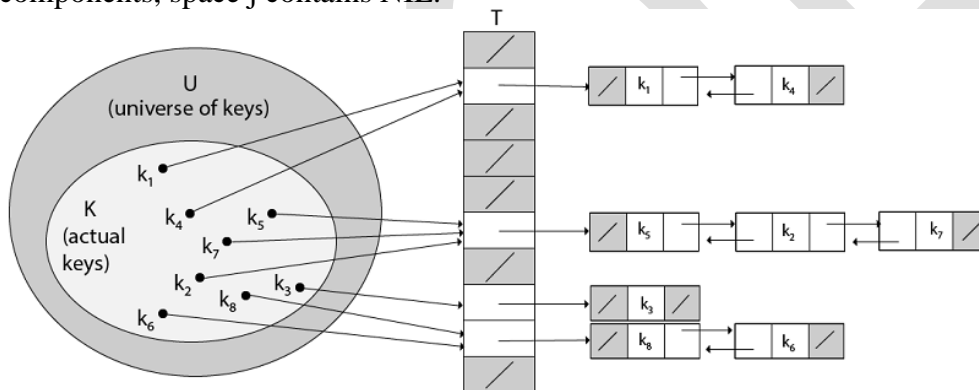


Fig: Collision resolution by chaining.

Each hash-table slot  $T[j]$  contains a linked list of all the keys whose hash value is  $j$ . For example,  $h(k_1) = h(k_4)$  and  $h(k_5) = h(k_7) = h(k_2)$ . The linked list can be either singly or doubly linked; we show it as doubly linked because deletion is faster that way.

#### 4.8.3. Analysis of Hashing with Chaining:

Given a hash table  $T$  with  $m$  spaces that stores  $n$  components, we characterize the heap factors  $\alpha$  for  $T$  as  $n/m$  that is the normal number of components put away in a chain. The most pessimistic scenario running time for looking is in this manner  $\theta(n)$  in addition to an opportunity to figure the hash work no better compared to on the off chance that we utilized one connected rundown for all the components. Obviously, hash tables are not utilized for their most pessimistic scenario execution.

The normal presentation of hashing relies upon how well the hash work  $h$  circulates the arrangement of keys to be put away among the  $m$  openings, all things considered.

Model: let us think about the inclusion of components 5, 28, 19,15,20,33,12,17,10 into an anchored hash table. Allow us to assume the hash table has 9 openings and the hash work be  $h(k) = k \bmod 9$ .

Arrangement: The underlying condition of tied hash table

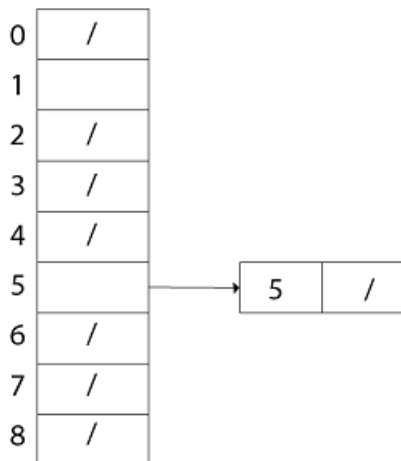
0	/
1	/
2	/
3	/
4	/
5	/
6	/
7	/
8	/

T

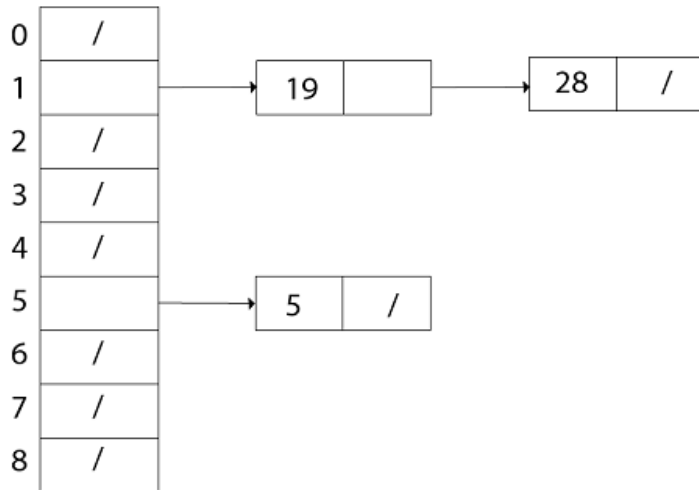
**Insert 5:**

$h(5) = 5 \bmod 9 = 5$

Create a linked list for T [5] and store value 5 in it.



Similarly, insert 28.  $h(28) = 28 \bmod 9 = 1$ . Create a Linked List for T [1] and store value 28 in it. Now insert 19  $h(19) = 19 \bmod 9 = 1$ . Insert value 19 in the slot T [1] at the beginning of the linked-list.



Now insert  $h(15)$ ,  $h(15) = 15 \bmod 9 = 6$ . Create a link list for  $T[6]$  and store value 15 in it.

Similarly, insert 20,  $h(20) = 20 \bmod 9 = 2$  in  $T[2]$ .

Insert 33,  $h(33) = 33 \bmod 9 = 6$

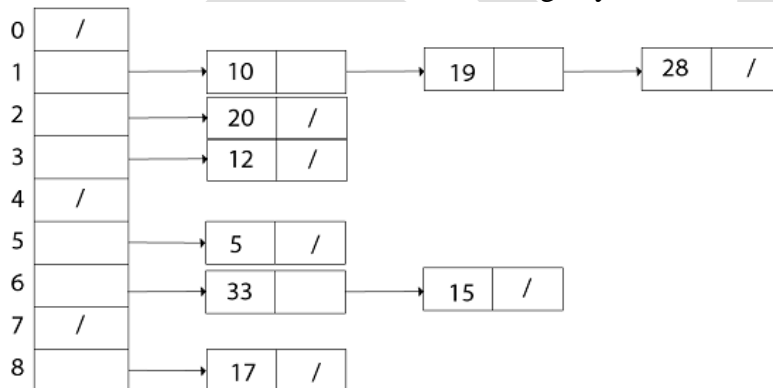
In the beginning of the linked list  $T[6]$ . Then,

Insert 12,  $h(12) = 12 \bmod 9 = 3$  in  $T[3]$ .

Insert 17,  $h(17) = 17 \bmod 9 = 8$  in  $T[8]$ .

Insert 10,  $h(10) = 10 \bmod 9 = 1$  in  $T[1]$ .

Thus the chained-hash-table after inserting key 10 is



#### 4.9.Hashing with Open Addressing

In Open Addressing, all components are put away in hash table itself. That is, each table passage comprises of a segment of the powerful set or NIL. While looking for a thing, we reliably analyze table openings until possibly we locate the ideal article or we have verified that the component isn't in the table. Accordingly, in open tending to, the heap factor  $\alpha$  can never surpass 1.

The upside of open tending to is that it dodges Pointer. In this, we figure the grouping of spaces to be inspected. The additional memory liberated by not sharing pointers furnishes the hash table with a bigger number of openings for a similar measure of memory, conceivably yielding less crash and quicker recovery.

The way toward looking at the area in the hash table is called Probing.

In this manner, the hash work becomes

$$h : U \times \{0,1,\dots,m-1\} \rightarrow \{0,1,\dots,m-1\}.$$

With open addressing, we require that for every key  $k$ , the probe sequence

$$\{h(k, 0), h(k, 1), \dots, h(k, m-1)\}$$

Be a Permutation of  $\{0, 1, \dots, m-1\}$

The HASH-INSERT procedure takes as input a hash table  $T$  and a key  $k$

HASH-INSERT ( $T, k$ )

1.  $i \leftarrow 0$
2. repeat  $j \leftarrow h(k, i)$
3. if  $T[j] = \text{NIL}$
4. then  $T[j] \leftarrow k$
5. return  $j$
6. else  $i \leftarrow i + 1$
7. until  $i = m$
8. error "hash table overflow"

The procedure HASH-SEARCH takes as input a hash table  $T$  and a key  $k$ , returning  $j$  if it finds that slot  $j$  contains key  $k$  or NIL if key  $k$  is not present in table  $T$ .

HASH-SEARCH. $T(k)$

1.  $i \leftarrow 0$
2. repeat  $j \leftarrow h(k, i)$
3. if  $T[j] = k$
4. then return  $j$
5.  $i \leftarrow i + 1$
6. until  $T[j] = \text{NIL}$  or  $i = m$
7. return NIL

0 1 2 3 4 5 6 7 8 9 10

/	65	21	/	26	37	/	/	/	59	76
---	----	----	---	----	----	---	---	---	----	----

#### 4.10 Open Addressing Techniques

Three techniques are commonly used to compute the probe sequence required for open addressing:

**4.10.1. Linear Probing.**

**4.10.2. Quadratic Probing.**

**4.10.3. Double Hashing.**

### 4.10.1. Linear Probing:

It is a Scheme in Computer Programming for settling impact in hash tables.

Assume another record R with key k is to be added to the memory table T however that the memory areas with the hash address H (k). H is as of now filled.

Our regular key to determine the impact is to intersection R to the most readily accessible area following T (h). We accept that the table T with m area is round, so T [i] comes after T [m].

The above impact goal is designated "Direct Probing".

Direct examining is easy to execute, yet it experiences an issue known as essential grouping. Long runs of involved spaces develop, expanding the normal pursuit time. Bunches emerge on the grounds that a vacant space continued by I full openings gets filled next with likelihood (I + 1)/m. Long runs of involved spaces will in general get longer, and the normal hunt time increments.

Given a normal hash work h': U {0, 1...m-1}, the technique for direct examining utilizes the hash work.

$$h(k, i) = (h'(k) + i) \text{ mod } m$$

Where 'm' is the size of hash table and  $h'(k) = k \text{ mod } m$ . for  $i=0, 1, \dots, m-1$ .

Given key k, the first slot is T [h' (k)]. We next slot T [h' (k) + 1] and so on up to the slot T [m-1]. Then we wrap around to slots T [0], T [1]....until finally slot T [h' (k)-1]. Since the initial probe position dispose of the entire probe sequence, only m distinct probe sequences are used with linear probing.

**Example:** Consider inserting the keys 24, 36, 58,65,62,86 into a hash table of size  $m=11$  using linear probing, consider the primary hash function is  $h'(k) = k \text{ mod } m$ .

Solution: Initial state of hash table

0	1	2	3	4	5	6	7	8	9	10
T										
/	/	/	/	/	/	/	/	/	/	/

Insert 24. We know  $h(k, i) = [h'(k) + i] \text{ mod } m$

$$\begin{aligned} \text{Now } h(24, 0) &= [24 \text{ mod } 11 + 0] \text{ mod } 11 \\ &= (2+0) \text{ mod } 11 = 2 \text{ mod } 11 = 2 \end{aligned}$$

Since T [2] is free, insert key 24 at this place.

$$\begin{aligned} \text{Insert 36. Now } h(36, 0) &= [36 \text{ mod } 11 + 0] \text{ mod } 11 \\ &= [3+0] \text{ mod } 11 = 3 \end{aligned}$$

Since T [3] is free, insert key 36 at this place.

$$\begin{aligned} \text{Insert 58. Now } h(58, 0) &= [58 \text{ mod } 11 + 0] \text{ mod } 11 \\ &= [3+0] \text{ mod } 11 = 3 \end{aligned}$$

Since T [3] is not free, so the next sequence is

$$\begin{aligned} h(58, 1) &= [58 \text{ mod } 11 + 1] \text{ mod } 11 \\ &= [3+1] \text{ mod } 11 = 4 \text{ mod } 11 = 4 \end{aligned}$$

T [4] is free; Insert key 58 at this place.

Insert 65. Now  $h(65, 0) = [65 \text{ mod } 11 + 0] \text{ mod } 11$



$$= (10 + 0) \bmod 11 = 10$$

T [10] is free. Insert key 65 at this place.

Insert 62. Now  $h(62, 0) = [62 \bmod 11 + 0] \bmod 11$

$$= [7 + 0] \bmod 11 = 7$$

T [7] is free. Insert key 62 at this place.

Insert 86. Now  $h(86, 0) = [86 \bmod 11 + 0] \bmod 11$

$$= [9 + 0] \bmod 11 = 9$$

T [9] is free. Insert key 86 at this place.

Thus,

	0	1	2	3	4	5	6	7	8	9	10
T	/	/	24	36	58	/	/	62	/	86	65

#### 4.10.2. Quadratic Probing:

Assume a record R with key k has the hash address  $H(k) = h$  then as opposed to looking through the area with addresses  $h, h+1, \text{ and } h+2 \dots$  We directly search the areas with addresses

$$h, h+1, h+4, h+9 \dots h+i^2$$

Quadratic Probing uses a hash capacity of the structure

$$h(k, i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$$

Where (as in direct testing)  $h'$  is a helper hash work  $c_1$  and  $c_2 \neq 0$  are assistant constants and  $i=0, 1 \dots m-1$ . The underlying position is  $T[h'(k)]$ ; later position tested is counterbalanced by the sum that depend in a quadratic way on the test number  $i$ .

Model: Consider embeddings the keys 74, 28, 36, 58, 21, 64 into a hash table of size  $m = 11$  utilizing quadratic examining with  $c_1=1$  and  $c_2=3$ . Further consider that the essential hash work is  $h'(k) = k \bmod m$ .

Arrangement: For Quadratic Probing, we have

$$h(k, i) = [k \bmod m + c_1 i + c_2 i^2] \bmod m$$

This is the initial state of hash table

Here  $c_1=1$  and  $c_2=3$

$$h(k, i) = [k \bmod m + i + 3i^2] \bmod m$$

Insert 74.

$$\begin{aligned}h(74,0) &= (74 \bmod 11 + 0 + 3 \times 0) \bmod 11 \\ &= (8 + 0 + 0) \bmod 11 = 8\end{aligned}$$

T [8] is free; insert the key 74 at this place.

Insert 28.

$$\begin{aligned}h(28,0) &= (28 \bmod 11 + 0 + 3 \times 0) \bmod 11 \\ &= (6 + 0 + 0) \bmod 11 = 6.\end{aligned}$$

T [6] is free; insert key 28 at this place.

Insert 36.

$$\begin{aligned}h(36,0) &= (36 \bmod 11 + 0 + 3 \times 0) \bmod 11 \\ &= (3 + 0 + 0) \bmod 11 = 3\end{aligned}$$

T [3] is free; insert key 36 at this place.

Insert 58.

$$\begin{aligned}h(58,0) &= (58 \bmod 11 + 0 + 3 \times 0) \bmod 11 \\ &= (3 + 0 + 0) \bmod 11 = 3\end{aligned}$$

T [3] is not free, so next probe sequence is computed as

$$\begin{aligned}h(58,1) &= (58 \bmod 11 + 1 + 3 \times 1) \bmod 11 \\ &= (3 + 1 + 3) \bmod 11 \\ &= 7 \bmod 11 = 7\end{aligned}$$

T [7] is free; insert key 58 at this place.

Insert 21.

$$\begin{aligned}h(21,0) &= (21 \bmod 11 + 0 + 3 \times 0) \\ &= (10 + 0) \bmod 11 = 10\end{aligned}$$

T [10] is free; insert key 21 at this place.

Insert 64.

$$\begin{aligned}h(64,0) &= (64 \bmod 11 + 0 + 3 \times 0) \\ &= (9 + 0 + 0) \bmod 11 = 9.\end{aligned}$$

T [9] is free; insert key 64 at this place.

Thus, after inserting all keys, the hash table is

0	1	2	3	4	5	6	7	8	9	10
/	/	/	36	/	/	28	58	74	64	21

### 4.10.3. Double Hashing:

Double Hashing is one of the best techniques available for open addressing because the permutations produced have many of the characteristics of randomly chosen permutations.

Double hashing uses a hash function of the form

$$h(k, i) = (h_1(k) + i h_2(k)) \bmod m$$

Where  $h_1$  and  $h_2$  are auxiliary hash functions and  $m$  is the size of the hash table.

$h_1(k) = k \bmod m$  or  $h_2(k) = k \bmod m'$ . Here  $m'$  is slightly less than  $m$  (say  $m-1$  or  $m-2$ ).

Example: Consider inserting the keys 76, 26, 37, 59, 21, 65 into a hash table of size  $m = 11$  using double hashing. Consider that the auxiliary hash functions are  $h_1(k) = k \bmod 11$  and  $h_2(k) = k \bmod 9$ .

Solution: Initial state of Hash table is

	0	1	2	3	4	5	6	7	8	9	10
T	/	/	/	/	/	/	/	/	/	/	/

1. Insert 76.

$$h_1(76) = 76 \bmod 11 = 10$$

$$h_2(76) = 76 \bmod 9 = 4$$

$$h(76, 0) = (10 + 0 \times 4) \bmod 11$$

$$= 10 \bmod 11 = 10$$

T [10] is free, so insert key 76 at this place.

2. Insert 26.

$$h_1(26) = 26 \bmod 11 = 4$$

$$h_2(26) = 26 \bmod 9 = 8$$

$$h(26, 0) = (4 + 0 \times 8) \bmod 11$$

$$= 4 \bmod 11 = 4$$

T [4] is free, so insert key 26 at this place.

3. Insert 37.

$$h_1(37) = 37 \bmod 11 = 4$$

$$h_2(37) = 37 \bmod 9 = 1$$

$$h(37, 0) = (4 + 0 \times 1) \bmod 11 = 4 \bmod 11 = 4$$

T [4] is not free, the next probe sequence is

$$h(37, 1) = (4 + 1 \times 1) \bmod 11 = 5 \bmod 11 = 5$$

T [5] is free, so insert key 37 at this place.

4. Insert 59.

$$h_1(59) = 59 \bmod 11 = 4$$

$$h_2(59) = 59 \bmod 9 = 5$$

$$h(59, 0) = (4 + 0 \times 5) \bmod 11 = 4 \bmod 11 = 4$$

Since, T [4] is not free, the next probe sequence is

$$h(59, 1) = (4 + 1 \times 5) \bmod 11 = 9 \bmod 11 = 9$$

T [9] is free, so insert key 59 at this place.

5. Insert 21.

$$h_1(21) = 21 \bmod 11 = 10$$

$$h_2(21) = 21 \bmod 9 = 3$$

$$h(21, 0) = (10 + 0 \times 3) \bmod 11 = 10 \bmod 11 =$$

10

T [10] is not free, the next probe sequence is

$$h(21, 1) = (10 + 1 \times 3) \bmod 11 = 13 \bmod 11 = 2$$

T [2] is free, so insert key 21 at this place.

6. Insert 65.

$$h_1(65) = 65 \bmod 11 = 10$$

$$h_2(65) = 65 \bmod 9 = 2$$

$$h(65, 0) = (10 + 0 \times 2) \bmod 11 = 10$$

$\bmod 11 = 10$

T [10] is not free, the next probe sequence is

$$h(65, 1) = (10 + 1 \times 2) \bmod 11 = 12$$

$\bmod 11 = 1$

T [1] is free, so insert key 65 at this place.

Thus, after insertion of all keys the final hash table is

#### 4.11 Hash Function

Hash Function is utilized to list the first worth or key and afterward utilized later each time the information related with the worth or key is to be recovered. Along these lines, hashing is consistently a single direction activity. There is no compelling reason to "figure out" the hash work by examining the hashed values.

#### 4.12 Characteristics of Good Hash Function:

1. The hash esteem is completely dictated by the information being hashed.
2. The hash Function utilizes all the information.
3. The hash work "consistently" conveys the information across the whole arrangement of conceivable hash esteems.
4. The hash work creates convoluted hash esteems for comparative strings.

### 4.13 Some Popular Hash Function is:

**4.13.1. Division Method:** Pick a number  $m$  more modest than the quantity of  $n$  of keys in  $k$  (The number  $m$  is typically picked to be an indivisible number or a number without little divisors, since this often a base number of crashes).

The hash work is:

$$h(k) = k \bmod m$$

$$h(k) = k \bmod m + 1$$

For Example: if the hash table has size  $m = 12$  and the key is  $k = 100$ , at that point  $h(k) = 4$ . Since it requires just a solitary division activity, hashing by division is very quick.

### 4.13.2. Duplication Method:

The increase technique for making hash capacities works in two stages. In the first place, we increase the vital  $k$  by a steady  $A$  in the reach  $0 < A < 1$  and concentrate the fragmentary piece of  $kA$ . At that point, we increment this incentive by  $m$  and take the floor of the outcome.

The hash work is:

$$h(k) = \lfloor m(kA \bmod 1) \rfloor$$

Where " $kA \bmod 1$ " means the fractional part of  $kA$ , that is,  $kA - \lfloor kA \rfloor$ .

### 4.13.3. Mid Square Method:

The key  $k$  is squared. Then function  $H$  is defined by

$$H(k) = L$$

Where  $L$  is obtained by deleting digits from both ends of  $k^2$ . We emphasize that the same position of  $k^2$  must be used for all of the keys.

### 4.13.4. Folding Method:

The key  $k$  is partitioned into a number of parts  $k_1, k_2, \dots, k_n$  where each part except possibly the last, has the same number of digits as the required address.

Then the parts are added together, ignoring the last carry.

$$H(k) = k_1 + k_2 + \dots + k_n$$

Example: Company has 68 employees, and each is assigned a unique four-digit employee number. Suppose  $L$  consist of 2-digit addresses: 00, 01, and 02....99. We apply the above hash functions to each of the following employee numbers:

3205, 7148, 2345

(a) Division Method: Choose a Prime number  $m$  close to 99, such as  $m = 97$ , Then

$$H(3205) = 4, \quad H(7148) = 67, \quad H(2345) = 17.$$

That is dividing 3205 by 17 gives a remainder of 4, dividing 7148 by 97 gives a remainder of 67, dividing 2345 by 97 gives a remainder of 17.

**(b) Mid-Square Method:**

$k = 3205 \quad 7148 \quad 2345$   
 $k^2 = 10272025 \quad 51093904 \quad 5499025$   
 $h(k) = 72 \quad 93 \quad 99$

Observe that fourth & fifth digits, counting from right are chosen for hash address.

**(c) Folding Method:**

Divide the key  $k$  into 2 parts and adding yields the following hash address:

$H(3205) = 32 + 50 = 82$      $H(7148) = 71 + 84 = 55$   
 $H(2345) = 23 + 45 = 68$

## Unit 4: Chapter 5

### Linear Data Structures

#### 5.0 Objective

##### 5.1. What is a Stack?

##### 5.2. Working of Stack

##### 5.3. Standard Stack Operations

###### 5.3.1. PUSH operation

###### 5.3.2. POP operation

##### 5.4. Applications of Stack

###### 5.4.1. Cluster execution of Stack

###### 5.4.2. Implementation of push algorithm in C language

###### 5.4.3. Visiting each element of the stack (Peek operation)

###### 5.4.4. Menu Driven program in C implementing all the stack operations

##### 5.5. Linked list implementation of stack

###### 5.5.2. Deleting a hub from the stack (POP activity)

###### 5.5.3. Display the nodes (Traversing)

###### 5.5.4. Menu Driven program in C implementing all the stack operations using linked list

#### 5.0 Objective

This chapter would make you understand the following concepts:

- **What is mean by Stack**
- **Different Operations on stack**
- **Application of stack**
- **Link List Implementation of stack**

#### 5.1. What is a Stack?

A Stack is a straight information structure that follows the LIFO (Last-In-First-Out) guideline. Stack has one end, though the Queue has two finishes (front and back). It contains just a single pointer top pointer highlighting the highest component of the stack. At whatever point a component is included the stack, it is added on the highest point of the stack, and the component can be erased uniquely from the stack. All in all, a stack can be characterized as a compartment in which inclusion and erasure should be possible from the one end known as the highest point of the stack.

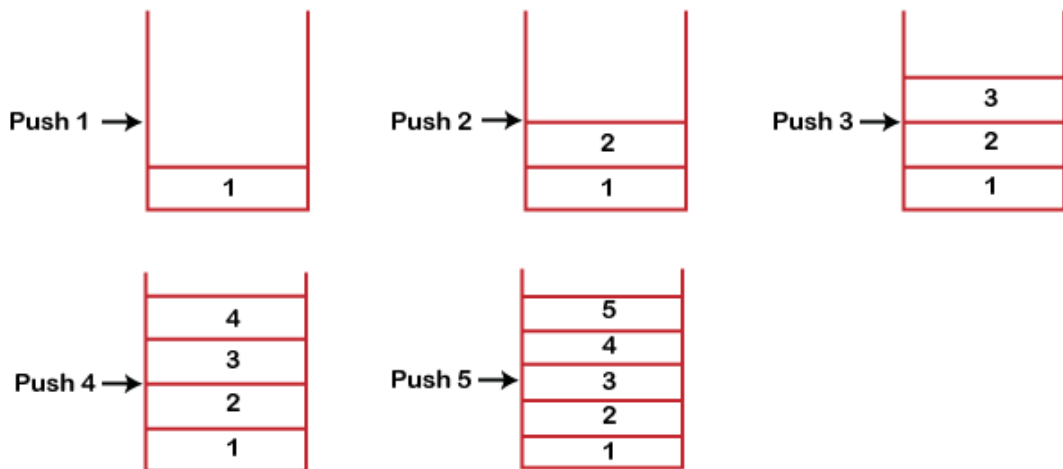
#### **Some key points related to stack**

o It is called as stack since it carries on like a certifiable stack, heaps of books, and so forth

- o A Stack is a theoretical information type with a pre-characterized limit, which implies that it can store the components of a restricted size.
- o It is an information structure that follows some request to embed and erase the components, and that request can be LIFO or FILO.

### 5.2.Working of Stack

Stack chips away at the LIFO design. As we can see in the underneath figure there are five memory blocks in the stack; along these lines, the size of the stack is 5. Assume we need to store the components in a stack and how about we expect that stack is vacant. We have taken the pile of size 5 as appeared underneath in which we are pushing the components individually until the stack turns out to be full.



Since our stack is full as the size of the stack is 5. In the above cases, we can see that it goes from the top to the base when we were entering the new component in the stack. The stack gets topped off from the base to the top.

At the point when we play out the erase procedure on the stack, there is just a single route for passage and exit as the opposite end is shut. It follows the LIFO design, which implies that the worth entered first will be eliminated last. In the above case, the worth 5 is entered first, so it will be taken out simply after the cancellation of the multitude of different components.

### 5.3. Standard Stack Operations

**Coming up next are some basic activities actualized on the stack:**

- o **push():** When we embed a component in a stack then the activity is known as a push. On the off chance that the stack is full, at that point the flood condition happens.
- o **pop():** When we erase a component from the stack, the activity is known as a pop. In the event that the stack is unfilled implies that no component exists in the stack, this state is known as an undercurrent state.
- o **isEmpty():** It decides if the stack is unfilled or not.
- o **isFull():** It decides if the stack is full or not.'
- o **peek():** It restores the component at the given position.
- o **count():** It restores the all out number of components accessible in a stack.
- o **change():** It changes the component at the given position.

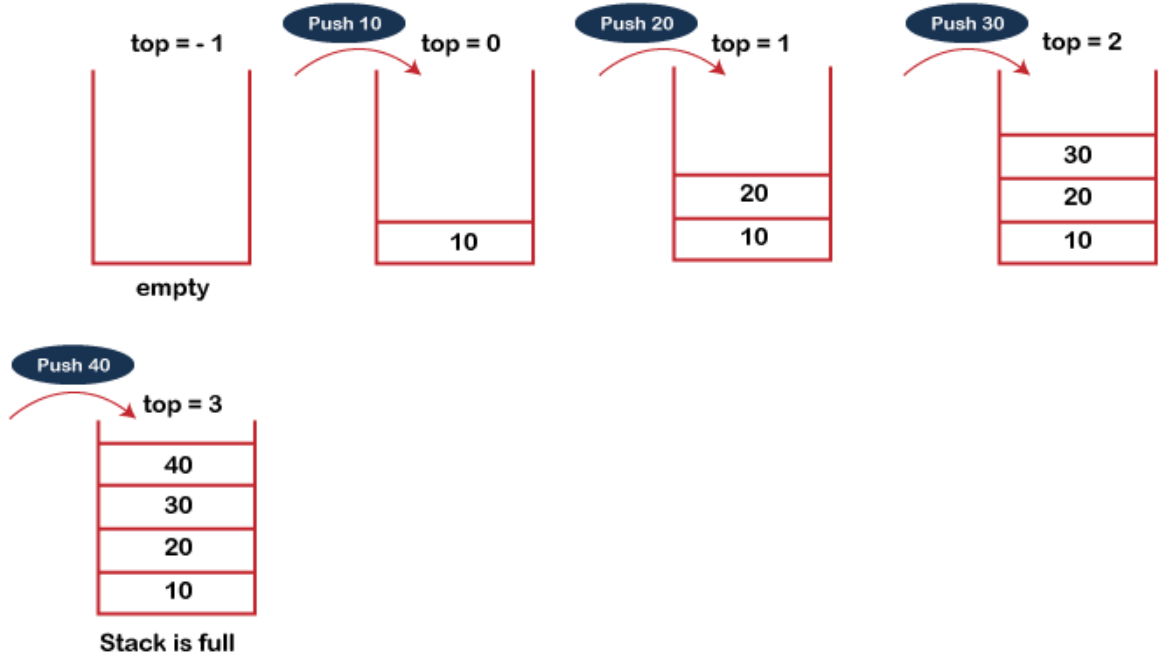


- o **display():** It prints all the components accessible in the stack.

### 5.3.1. PUSH operation

**The means engaged with the PUSH activity is given beneath:**

- o Before embeddings a component in a stack, we check whether the stack is full.
- o If we attempt to embed the component in a stack, and the stack is full, at that point the flood condition happens.
- o When we introduce a stack, we set the estimation of top as - 1 to watch that the stack is unfilled.
- o When the new component is pushed in a stack, first, the estimation of the top gets increased, i.e.,  $top=top+1$ , and the component will be put at the new situation of the top.
- o The components will be embedded until we arrive at the maximum size of the stack.

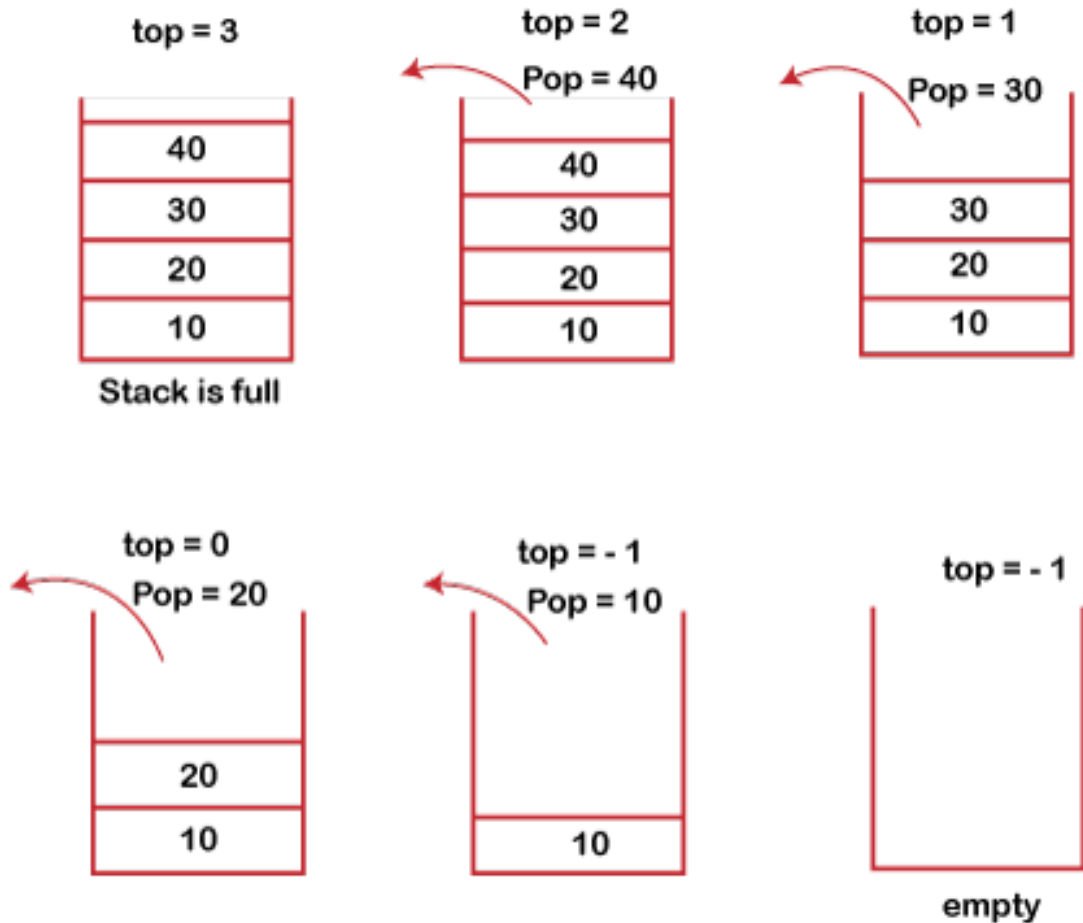


### 5.3.2. POP operation

**The means engaged with the POP activity is given beneath:**

- o Before erasing the component from the stack, we check whether the stack is vacant.
- o If we attempt to erase the component from the vacant stack, at that point the sub-current condition happens.
- o If the stack isn't unfilled, we first access the component which is pointed by the top

- o Once the pop activity is played out, the top is decremented by 1, i.e.,  $top = top - 1$



#### 5.4. Applications of Stack

The following are the applications of the stack:

**Balancing of symbols:** Stack is used for balancing a symbol. For example, we have the following program:

```
int main()
{
cout<<"Hello";
cout<<"Data Structure";
}
```

As we probably are aware, each program has an opening and shutting supports; when the initial supports come, we push the supports in a stack, and when the end supports show up, we pop the initial supports from the stack. Subsequently, the net worth comes out to be zero. In the event that any image is left in the stack, it implies that some language structure happens in a program.

o **String inversion:** Stack is additionally utilized for switching a string. For instance, we need to switch a "Information Structure" string, so we can accomplish this with the assistance of a stack.

To start with, we push all the characters of the string in a stack until we arrive at the invalid character.

In the wake of pushing all the characters, we begin taking out the character individually until we arrive at the lower part of the stack.

o **UNDO/REDO:** It can likewise be utilized for performing UNDO/REDO tasks. For instance, we have a manager where we compose 'a', at that point 'b', and afterward 'c'; hence, the content written in a supervisor is abc. Thus, there are three expresses, a, stomach muscle, and abc, which are put away in a stack. There would be two stacks in which one stack shows UNDO state, and different shows REDO state.

In the event that we need to perform UNDO activity, and need to accomplish 'stomach muscle' state, at that point we actualize pop activity.

o **Recursion:** The recursion implies that the capacity is calling itself once more. To keep up the past states, the compiler makes a framework stack in which all the past records of the capacity are kept up.

o **DFS(Depth First Search):** This hunt is actualized on a Graph, and Graph utilizes the stack information structure.

o **Backtracking:** Suppose we need to make a way to tackle a labyrinth issue. In the event that we are moving in a specific way, and we understand that we please the incorrect way. To come toward the start of the way to make another way, we need to utilize the stack information structure.

o **Expression change:** Stack can likewise be utilized for articulation transformation. This is perhaps the main utilizations of stack. The rundown of the articulation change is given underneath: Infix to prefix

Infix to postfix

Prefix to infix

Prefix to postfix

Postfix to infix

o **Memory the board:** The stack deals with the memory. The memory is allotted in the touching memory blocks. The memory is referred to as stack memory as all the

factors are allocated in a capacity call stack memory. The memory size allotted to the program is known to the compiler. At the point when the capacity is made, every one of its factors are doled out in the stack memory. At the point when the capacity finished its execution, all the factors doled out in the stack are delivered.

#### **5.4.1. Cluster execution of Stack**

In cluster execution, the stack is shaped by utilizing the exhibit. All the activities with respect to the stack are performed utilizing exhibits. Lets perceive how every activity can be actualized on the stack utilizing cluster information structure.

Adding a component onto the stack (push activity)

Adding a component into the highest point of the stack is alluded to as push activity.

Push activity includes following two stages.

1. Increment the variable Top with the goal that it can now refer to the following memory area.
2. Add component at the situation of increased top. This is alluded to as adding new component at the highest point of the stack.

Stack is overflown when we attempt to embed a component into a totally filled stack thusly, our primary capacity should consistently stay away from stack flood condition.

#### **Algorithm:**

begin

if top = n then stack full

top = top + 1

stack (top) := item;

end

**Time Complexity :  $O(1)$**

#### **5.4.2. Implementation of push algorithm in C language**

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

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

#### **Algorithm :**

begin

if top = 0 then stack empty;

item := stack(top);

top = top - 1;

end;

**Time Complexity :  $O(1)$**

#### **Implementation of POP algorithm using C language**

```

int pop ()
{
if(top == -1)
{
printf("Underflow");
return 0;
}
else
{
return stack[top - - ];
}
}

```

#### 5.4.3. Visiting each element of the stack (Peek operation)

Look actKivity includes restoring the component which is available at the highest point of the stack without erasing it. Sub-current condition can happen in the event that we attempt to restore the top compo:.,nent in an all around void stack.

##### Algorithm :

PEEK (STACK, TOP)

Begin

if top = -1 then stack empty

item = stack[top]

return item

End

**Time complexity: o(n)**

#### Implementation of Peek algorithm in C language

```

int peek()
{
if (top == -1)
{
printf("Underflow");
return 0;
}
else
{
return stack [top];
}
}

```

#### 5.4.4. Menu Driven program in C implementing all the stack operations

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

```
int stack[100],i,j,choice=0,n,top=-1;
```

```
void push();
```

```

void pop();
void show();
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:
{
show();
break;
}
case 4:
{
printf("Exiting....");
break;
}
default:
{
printf("Please Enter valid choice ");
}
};
}
}

```

```

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

```

```

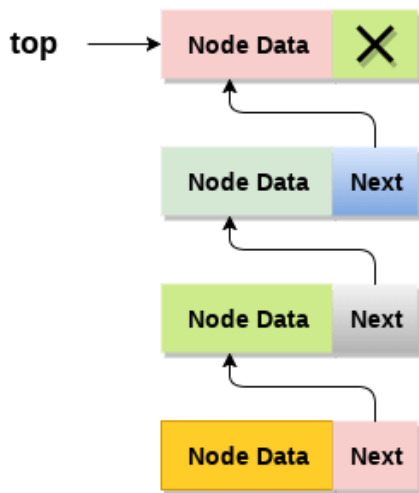
void pop ()
{
if(top == -1)
printf("Underflow");
else
top = top -1;
}
void show()
{
for (i=top;i>=0;i--)
{
printf("%d\n",stack[i]);
}
if(top == -1)
{
printf("Stack is empty");
}
}

```

### **5.5. Linked list implementation of stack**

Rather than utilizing cluster, we can likewise utilize connected rundown to execute stack. Connected rundown assigns the memory powerfully. Nonetheless, time unpredictability in both the situation is same for all the tasks for example push, pop and look.

In connected rundown execution of stack, the hubs are kept up non-coterminously in the memory. Every hub contains a pointer to its nearby replacement hub in the stack. Stack is supposed to be overflown if the space left in the memory pile isn't sufficient to make a hub.



### Stack

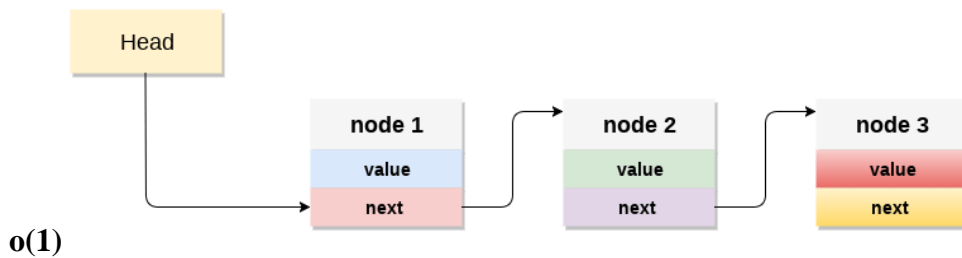
The top most hub in the stack consistently contains invalid in its location field. Lets examine the manner by which, every activity is acted in connected rundown usage of stack.

#### 5.5.1. Adding a node to the stack (Push operation)

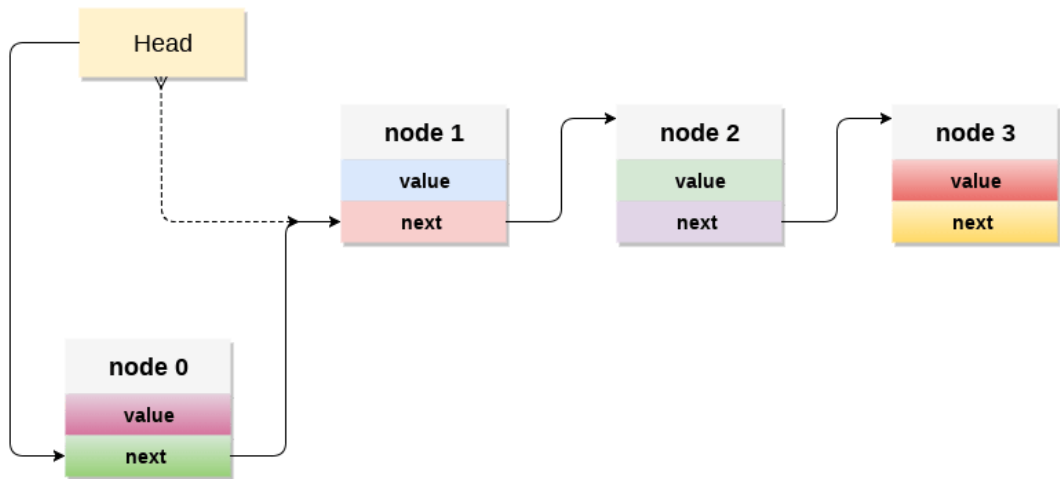
Adding a hub to the stack is alluded to as push activity. Pushing a component to a stack in connected rundown execution is not quite the same as that of a cluster usage. To push a component onto the stack, the accompanying advances are included.

1. Create a hub first and designate memory to it.
2. If the rundown is vacant then the thing is to be pushed as the beginning hub of the rundown. This incorporates doling out an incentive to the information part of the hub and allot invalid to the location part of the hub.
3. If there are a few hubs in the rundown effectively, at that point we need to add the new component in the start of the rundown (to not disregard the property of the stack). For this reason, allocate the location of the beginning component to the location field of the new hub and make the new hub, the beginning hub of the rundown.

**Time Complexity :**







### New Node

#### C implementation:

```

void push ()
{
    int val;
    struct node *ptr =(struct node*)malloc(sizeof(struct node));
    if(ptr == NULL)
    {
        printf("not able to push the element");
    }
    else
    {
        printf("Enter the value");
        scanf("%d",&val);
        if(head==NULL)
        {
            ptr->val = val;
            ptr -> next = NULL;
            head=ptr;
        }
        else
        {
            ptr->val = val;
            ptr->next = head;
            head=ptr;
        }
        printf("Item pushed");
    }
}
  
```

### 5.5.2. Deleting a hub from the stack (POP activity)

Erasing a hub from the highest point of stack is alluded to as pop activity. Erasing a hub from the connected rundown usage of stack is not quite the same as that in the exhibit execution. To pop a component from the stack, we need to follow the accompanying advances :

Check for the undercurrent condition: The sub-current condition happens when we attempt to fly from a generally unfilled stack. The stack will be unfilled if the head pointer of the rundown focuses to invalid.

Change the head pointer in like manner: In stack, the components are popped uniquely from one end, thusly, the worth put away in the head pointer should be erased and the hub should be liberated. The following hub of the head hub presently turns into the head hub.

**Time Complexity:  $O(n)$**

#### C implementation

```
void pop()
{
int item;
struct node *ptr;
if (head == NULL)
{
printf("Underflow");
}
else
{
item = head->val;
ptr = head;
head = head->next;
free(ptr);
printf("Item popped");
}
}
```

### 5.5.3. Display the nodes (Traversing)

Showing all the hubs of a stack requires navigating all the hubs of the connected rundown coordinated as stack. For this reason, we need to follow the accompanying advances.

1. Copy the head pointer into an impermanent pointer.
2. Move the brief pointer through all the hubs of the rundown and print the worth field joined to each hub.

**Time Complexity:  $O(n)$**

#### C Implementation

```
void display()
{
inti;
```

```

struct node *ptr;
ptr=head;
if(ptr == NULL)
{
printf("Stack is empty\n");
}
else
{
printf("Printing Stack elements \n");
while(ptr!=NULL)
{
printf("%d\n",ptr->val);
ptr = ptr->next;
}
}
}

```

#### 5.5.4. Menu Driven program in C implementing all the stack operations using linked list:

```

#include <stdio.h>
#include <stdlib.h>
void push();
void pop();
void display();
struct node
{
intval;
struct node *next;
};
struct node *head;
void main ()
{
int choice=0;
printf("\n*****Stack operations using linked list*****\n");
printf("\n-----\n");
while(choice != 4)
{
printf("\n\nChose 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 ()
{
intval;
struct node *ptr = (struct node*)malloc(sizeof(struct node));
if(ptr == NULL)
    {
printf("not able to push the element");
    }
else
    {
printf("Enter the value");
scanf("%d",&val);
if(head==NULL)
    {
ptr->val = val;
ptr -> next = NULL;
head=ptr;
    }
else
    {
ptr->val = val;
ptr->next = head;

```

```

head=ptr;

    }
printf("Item pushed");

    }
}

void pop()
{
int item;
struct node *ptr;
if (head == NULL)
    {
printf("Underflow");
    }
else
    {
item = head->val;
ptr = head;
head = head->next;
free(ptr);
printf("Item popped");

    }
}

void display()
{
inti;
struct node *ptr;
ptr=head;
if(ptr == NULL)
    {
printf("Stack is empty\n");
    }
else
    {
printf("Printing Stack elements \n");
while(ptr!=NULL)
    {
printf("%d\n",ptr->val);
ptr = ptr->next;
    }
}
}
}

```

## Unit 4 - Chapter 6

### Queue

#### 6.0 Objective

#### 6.1.Queue

#### 6.2.Applications of Queue

#### 6.3.Types of Queues

#### 6.4.Operations on Queue

#### 6.5.Implementation of Queue

##### 6.5.1.Consecutive assignment:

##### 6.5.2.LinkedList allocation:

#### 6.6.What are the utilization instances of Queue?

#### 6.7.Types of Queue

##### 6.7.1.Linear Queue

##### 6.7.2.Circular Queue

##### 6.7.3.Priority Queue

##### 6.7.4.Dequeue

#### 6.8.Array representation of Queue

#### 6.9.Queue

##### 6.9.1.Algorithm

##### 6.9.2.C Function

##### 6.9.3.Algorithm

##### 6.9.4.Menu driven program to implement queue using array

#### 6.10.LinkedList implementation of Queue

#### 6.11.Operation on Linked Queue

##### 6.11.1.Insert operation

##### 6.11.2.Algorithm

##### 6.11.3.C Function

#### 6.12.Deletion

##### 6.12.1.Algorithm

##### 6.12.2.C Function

##### 6.12.3.Menu-Driven Program implementing all the operations on Linked Queue

#### 6.0.Objective

This chapter would make you understand the following concepts:

- Queue
- Definition
- Operations, Implementation of simple
- queue (Array and Linked list) and applications of queue-BFS
- Types of queues: Circular, Double ended, Priority,

#### 6.1.Queue

1. A Queue can be characterized as an arranged rundown which empowers embed tasks to be performed toward one side called REAR and erase activities to be performed at another end called FRONT.
2. Queue is alluded to be as First In First Out rundown.
3. For instance, individuals sitting tight in line for a rail ticket structure a Queue



## 6.2.Applications of Queue

Because of the way that line performs activities on first in first out premise which is very reasonable for the requesting of activities. There are different uses of queues examined as beneath.

1. Queues are generally utilized as hanging tight records for a solitary shared asset like printer, plate, CPU.
2. Queues are utilized in offbeat exchange of information (where information isn't being moved at similar rate between two cycles) for eg. pipes, document IO, attachments.
3. Queues are utilized as cradles in the greater part of the applications like MP3 media player, CD player, and so on
4. Queue are utilized to keep up the play list in media major parts to add and eliminate the tunes from the play-list.
5. Queues are utilized in working frameworks for dealing with interferes.

## Complexity

Data Structure	Time Complexity								Space Compleity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Queue	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$

## 6.3.Types of Queues

Prior to understanding the sorts of queues, we first glance at 'what is Queue'.

### What is the Queue?

A queue in the information construction can be viewed as like the queue in reality. A queue is an information structure in which whatever starts things out will go out first.

It follows the FIFO (First-In-First-Out) arrangement. In Queue, the inclusion is done from one end known as the backside or the tail of the queue, though the erasure is done from another end known as the front end or the top of the line. At the end of the day, it very well may be characterized as a rundown or an assortment with an imperative that the addition can be performed toward one side called as the backside or tail of the queue and cancellation is performed on another end called as the front end or the top of the queue.



#### 6.4.Operations on Queue

- o Enqueue: The enqueue activity is utilized to embed the component at the backside of the queue. It brings void back.
- o Dequeue: The dequeue activity plays out the erasure from the front-finish of the queue. It additionally restores the component which has been eliminated from the front-end. It restores a number worth. The dequeue activity can likewise be intended to void.
- o Peek: This is the third activity that profits the component, which is pointed by the front pointer in the queue yet doesn't erase it.
- o Queue flood (isfull): When the Queue is totally full, at that point it shows the flood condition.
- o Queue undercurrent (isempty): When the Queue is unfilled, i.e., no components are in the Queue then it tosses the sub-current condition.

A Queue can be addressed as a compartment opened from both the sides in which the component can be enqueued from one side and dequeued from another side as demonstrated in the beneath figure:

#### 6.5.Implementationof Queue

**There are two different ways of executing the Queue:**

**6.5.1.Consecutive assignment:** The successive distribution in a Queue can be executed utilizing a cluster.

**6.5.2.Linked list allocation:** The linked list portion in a Queue can be actualized utilizing a linked list.

#### 6.6.What are the utilization instances of Queue?

Here, we will see this present reality situations where we can utilize the Queue information structure. The Queue information structure is primarily utilized where



there is a shared asset that needs to serve the different asks for however can serve a solitary solicitation at a time. In such cases, we need to utilize the Queue information structure for lining up the solicitations. The solicitation that shows up first in the line will be served first. Coming up next are this present reality situations in which the Queue idea is utilized:

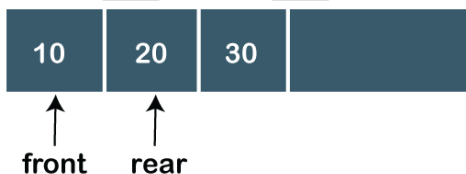
- o Suppose we have a printer divided among different machines in an organization, and any machine or PC in an organization can send a print solicitation to the printer. In any case, the printer can serve a solitary solicitation at a time, i.e., a printer can print a solitary archive at a time. At the point when any print demand comes from the organization, and if the printer is occupied, the printer's program will put the print demand in a line.
- o If the solicitations are accessible in the Queue, the printer takes a solicitation from the front of the Queue, and serves it.
- o The processor in a PC is likewise utilized as a shared asset. There are numerous solicitations that the processor should execute, however the processor can serve a solitary ask for or execute a solitary interaction at a time. Consequently, the cycles are kept in a Queue for execution.

## 6.7.Types of Queue

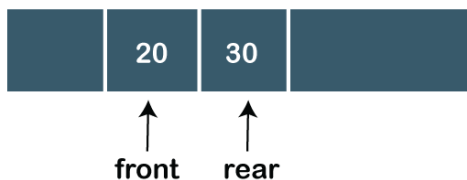
**There are four kinds of Queues:**

### 6.7.1.Linear Queue

In Linear Queue, an inclusion happens from one end while the erasure happens from another end. The end at which the addition happens is known as the backside, and the end at which the erasure happens is known as front end. It carefully keeps the FIFO rule. The straight Queue can be addressed, as demonstrated in the beneath figure:



The above figure shows that the components are embedded from the backside, and on the off chance that we embed more components in a Queue, at that point the back worth gets increased on each addition. In the event that we need to show the cancellation, at that point it tends to be addressed as:

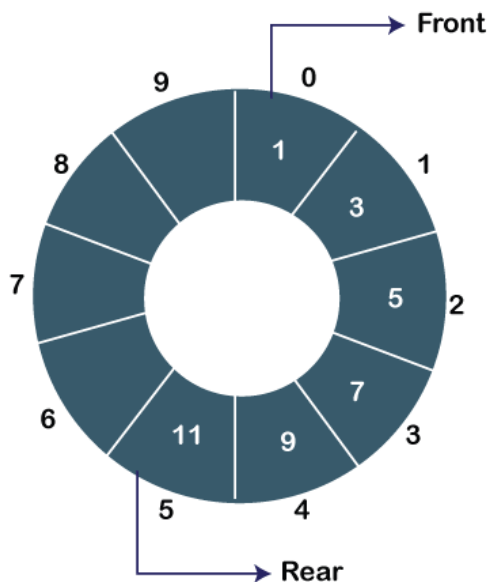


In the above figure, we can see that the front pointer focuses to the following component, and the component which was recently pointed by the front pointer was erased.

The significant disadvantage of utilizing a straight Queue is that inclusion is done distinctly from the backside. In the event that the initial three components are erased from the Queue, we can't embed more components despite the fact that the space is accessible in a Linear Queue. For this situation, the straight Queue shows the flood condition as the back is highlighting the last component of the Queue.

### 6.7.2. Circular Queue

In Circular Queue, all the hubs are addressed as round. It is like the direct Queue aside from that the last component of the line is associated with the principal component. It is otherwise called Ring Buffer as all the finishes are associated with another end. The round line can be addressed as:



The disadvantage that happens in a direct line is overwhelmed by utilizing the roundabout queue. On the off chance that the unfilled space is accessible in a round line, the new component can be included a vacant space by just augmenting the estimation of back.

### 6.7.3. Priority Queue

A need queue is another exceptional sort of Queue information structure in which every component has some need related with it. In view of the need of the component, the components are organized in a need line. In the event that the components happen with a similar need, at that point they are served by the FIFO rule.

In need Queue, the inclusion happens dependent on the appearance while the cancellation happens dependent on the need. The need Queue can be appeared as:

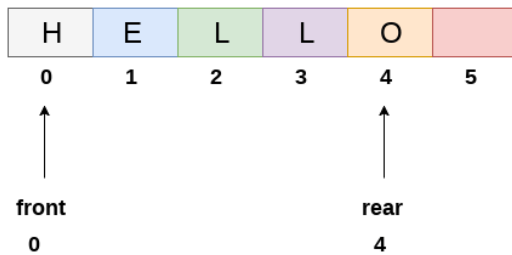
The above figure shows that the most elevated need component starts things out and the components of a similar need are organized dependent on FIFO structure.

### 6.7.4. Deque

Both the Linear Queue and Deque are distinctive as the direct line follows the FIFO standard while, deque doesn't follow the FIFO rule. In Deque, the inclusion and erasure can happen from the two closures.

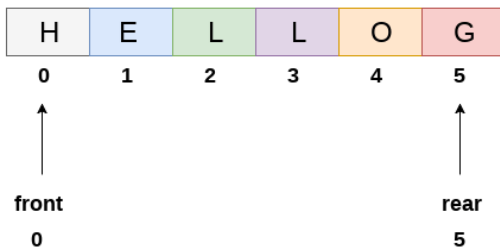
### 6.8. Array representation of Queue

We can without much of a stretch address queue by utilizing direct exhibits. There are two factors for example front and back, that are actualized on account of each queue. Front and back factors highlight the situation from where inclusions and cancellations are acted in a queue. At first, the estimation of front and queue is - 1 which addresses an unfilled queue. Cluster portrayal of a queue containing 5 components alongside the separate estimations of front and back, is appeared in the accompanying figure.



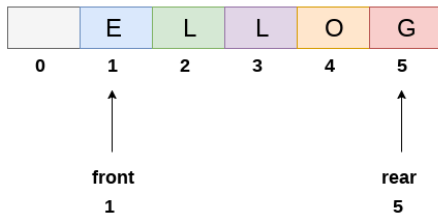
### 6.9. Queue

The above figure shows the queue of characters shaping the English word "Hi". Since, No cancellation is acted in the line till now, thusly the estimation of front remaining parts - 1 . Be that as it may, the estimation of back increments by one each time an addition is acted in the queue. Subsequent to embeddings a component into the queue appeared in the above figure, the queue will look something like after. The estimation of back will become 5 while the estimation of front remaining parts same.



#### Queue after inserting an element

After deleting an element, the value of front will increase from -1 to 0. however, the queue will look something like following.



### Queue after deleting an element

Algorithm to embed any component in a line

Check if the line is as of now full by contrasting back with  $\text{max} - 1$ . assuming this is the case, at that point return a flood blunder.

In the event that the thing is to be embedded as the principal component in the rundown, all things considered set the estimation of front and back to 0 and addition the component at the backside.

In any case continue to expand the estimation of back and addition every component individually having back as the file.

#### 6.9.1. Algorithm

Step 1: IF REAR = MAX - 1

Write OVERFLOW

Go to step

[END OF IF]

Step 2: IF FRONT = -1 and REAR = -1

SET FRONT = REAR = 0

ELSE

SET REAR = REAR + 1

[END OF IF]

Step 3: Set QUEUE[REAR] = NUM

Step 4: EXIT

#### 6.9.2.C Function

```
void insert (int queue[], int max, int front, int rear, int item)
```

```
{
```

```
if (rear + 1 == max)
```

```
{
```

```
printf("overflow");
```

```
}
```

```
else
```

```
{
```

```
if(front == -1 && rear == -1)
```

```
{
```

```

front = 0;
rear = 0;
    }
else
    {
rear = rear + 1;
    }
queue[rear]=item;
    }
}

```

Algorithm to delete an element from the queue

If, the value of front is -1 or value of front is greater than rear , write an underflow message and exit.

Otherwise, keep increasing the value of front and return the item stored at the front end of the queue at each time.

### 6.9.3.Algorithm

Step 1: IF FRONT = -1 or FRONT > REAR

Write UNDERFLOW

ELSE

SET VAL = QUEUE[FRONT]

SET FRONT = FRONT + 1

[END OF IF]

Step 2: EXIT

C Function

```
int delete (int queue[], int max, int front, int rear)
```

```
{
int y;
if (front == -1 || front > rear)
```

```
{
printf("underflow");
}
```

```
else
```

```
{
    y = queue[front];
if(front == rear)
{
```

```

front = rear = -1;
else
front = front + 1;

    }
return y;
    }
}

```

#### 6.9.4.Menu driven program to implement queue using array

```

#include<stdio.h>
#include<stdlib.h>
#define maxsize 5
void insert();
void delete();
void display();
int front = -1, rear = -1;
int queue[maxsize];
void main ()
{
int choice;
while(choice != 4)
    {
printf("\n*****Main
Menu*****\n");
printf("\n=====
=====\\n");
printf("\n1.insert an element\n2.Delete an element\n3.Display the
queue\n4.Exit\n");
printf("\nEnter your choice ?");
scanf("%d",&choice);
switch(choice)
    {
case 1:
insert();
break;
case 2:
delete();
break;
case 3:
display();

```

```
break;
case 4:
exit(0);
break;
default:
printf("\nEnter valid choice??\n");
    }
    }
}
void insert()
{
int item;
printf("\nEnter the element\n");
scanf("\n%d",&item);
if(rear == maxsize-1)
    {
printf("\nOVERFLOW\n");
return;
    }
if(front == -1 && rear == -1)
    {
front = 0;
rear = 0;
    }
else
    {
rear = rear+1;
    }
queue[rear] = item;
printf("\nValue inserted ");

}
void delete()
{
int item;
if (front == -1 || front > rear)
    {
printf("\nUNDERFLOW\n");
return;
    }
```

```

    }
else
    {
item = queue[front];
if(front == rear)
    {
front = -1;
rear = -1 ;
    }
else
    {
front = front + 1;
    }
printf("\nvalue deleted ");
    }

}

void display()
{
inti;
if(rear == -1)
    {
printf("\nEmpty queue\n");
    }
else
    { printf("\nprinting values ..... \n");
for(i=front;i<=rear;i++)
    {
printf("\n%d\n",queue[i]);
    }
    }
}
}

```

**Output:**

\*\*\*\*\*Main Menu\*\*\*\*\*

- 
- 1.insert an element
  - 2.Delete an element
  - 3.Display the queue



4.Exit

Enter your choice ?1

Enter the element

123

Value inserted

\*\*\*\*\*Main Menu\*\*\*\*\*

=====

- 1.insert an element
- 2.Delete an element
- 3.Display the queue
- 4.Exit

Enter your choice ?1

Enter the element

90

Value inserted

\*\*\*\*\*Main Menu\*\*\*\*\*

=====

- 1.insert an element
- 2.Delete an element
- 3.Display the queue
- 4.Exit

Enter your choice ?2

value deleted

\*\*\*\*\*Main Menu\*\*\*\*\*

=====

- 1.insert an element
- 2.Delete an element
- 3.Display the queue
- 4.Exit

Enter your choice ?3

printing values .....

90

\*\*\*\*\*Main Menu\*\*\*\*\*

=====

- 1.insert an element
- 2.Delete an element
- 3.Display the queue
- 4.Exit

Enter your choice ?4

### **6.10.Linked List implementation of Queue**

Because of the disadvantages examined in the past part of this instructional exercise, the exhibit usage can not be utilized for the huge scope applications where the queues are actualized. One of the option of cluster usage is connected rundown execution of queue.

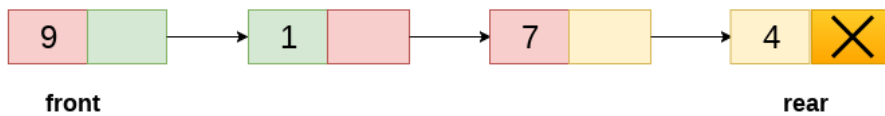
The capacity prerequisite of connected portrayal of a queue with n components is  $o(n)$  while the time necessity for tasks is  $o(1)$ .

In a linked queue, every hub of the queue comprises of two sections for example information part and the connection part. Every component of the queue focuses to its nearby next component in the memory.

In the linked queue, there are two pointers kept up in the memory for example front pointer and back pointer. The front pointer contains the location of the beginning component of the queue while the back pointer contains the location of the last component of the queue.

Inclusion and erasures are performed at back and front end separately. On the off chance that front and back both are NULL, it shows that the line is vacant.

The connected portrayal of queue is appeared in the accompanying figure.



## Linked Queue

### 6.11.Operation on Linked Queue

There are two basic operations which can be implemented on the linked queues. The operations are Insertion and Deletion.

#### 6.11.1.Insert operation

The addition activity attach the line by adding a component to the furthest limit of the line. The new component will be the last component of the line.

Right off the bat, assign the memory for the new hub ptr by utilizing the accompanying assertion.

```
Ptr = (struct node *) malloc (sizeof(struct node));
```

There can be the two situation of embeddings this new hub ptr into the connected line.

In the principal situation, we embed component into an unfilled queue. For this situation, the condition `front = NULL` turns out to be valid. Presently, the new component will be added as the lone component of the queue and the following pointer of front and back pointer both, will highlight NULL.

```
ptr -> data = item;
if(front == NULL)
{
front = ptr;
rear = ptr;
front -> next = NULL;
rear -> next = NULL;
}
```

In the subsequent case, the queue contains more than one component. The condition `front = NULL` turns out to be bogus. In this situation, we need to refresh the end pointer back with the goal that the following pointer of back will highlight the new hub ptr. Since, this is a connected line, consequently we likewise need to make the back pointer highlight the recently added hub ptr. We additionally need to make the following pointer of back highlight NULL.

```
rear -> next = ptr;
rear = ptr;
rear->next = NULL;
```

In this way, the element is inserted into the queue. The algorithm and the C implementation is given as follows.

### 6.11.2. Algorithm

```
Step 1: Allocate the space for the new node PTR
Step 2: SET PTR -> DATA = VAL
Step 3: IF FRONT = NULL
SET FRONT = REAR = PTR
SET FRONT -> NEXT = REAR -> NEXT = NULL
ELSE
SET REAR -> NEXT = PTR
SET REAR = PTR
SET REAR -> NEXT = NULL
[END OF IF]
Step 4: END
```

### 6.11.3.C Function

```
void insert(struct node *ptr, int item; )
{
ptr = (struct node *) malloc (sizeof(struct node));
if(ptr == NULL)
{
printf("\nOVERFLOW\n");
return;
}
else
{
ptr -> data = item;
if(front == NULL)
{
front = ptr;
rear = ptr;
front -> next = NULL;
rear -> next = NULL;
}
else
{
rear -> next = ptr;
rear = ptr;
}
```

```

    rear->next = NULL;
    }
}
}

```

## 6.12.Deletion

Cancellation activity eliminates the component that is first embedded among all the queue components. Right off the bat, we need to check either the rundown is unfilled or not. The condition `front == NULL` turns out to be valid if the rundown is unfilled, for this situation, we essentially compose undercurrent on the comfort and make exit. Else, we will erase the component that is pointed by the pointer `front`. For this reason, duplicate the hub pointed by the front pointer into the pointer `ptr`. Presently, move the front pointer, highlight its next hub and free the hub pointed by the hub `ptr`. This is finished by utilizing the accompanying assertions.

```

ptr = front;
front = front -> next;
free(ptr);

```

The algorithm and C function is given as follows.

### 6.12.1.Algorithm

```

Step 1: IF FRONT = NULL
Write " Underflow "
Go to Step 5
[END OF IF]
Step 2: SET PTR = FRONT
Step 3: SET FRONT = FRONT -> NEXT
Step 4: FREE PTR
Step 5: END

```

### 6.12.2.C Function

```

void delete (struct node *ptr)
{
if(front == NULL)
{
printf("\nUNDERFLOW\n");
return;
}
else
{
ptr = front;
front = front -> next;
free(ptr);
}
}

```

```
    }  
}
```

### 6.12.3.Menu-Driven Program implementing all the operations on Linked Queue

```
#include<stdio.h>  
#include<stdlib.h>  
struct node  
{  
int data;  
struct node *next;  
};  
struct node *front;  
struct node *rear;  
void insert();  
void delete();  
void display();  
void main ()  
{  
int choice;  
while(choice != 4)  
{  
printf("\n*****Main  
Menu*****\n");  
printf("\n===== \n");  
printf("\n1.insert an element\n2.Delete an element\n3.Display the  
queue\n4.Exit\n");  
printf("\nEnter your choice ?");  
scanf("%d",& choice);  
switch(choice)  
{  
case 1:  
insert();  
break;  
case 2:  
delete();  
break;  
case 3:  
display();  
break;
```

```

case 4:
exit(0);
break;
default:
printf("\nEnter valid choice??\n");
    }
    }
}
void insert()
{
struct node *ptr;
int item;

ptr = (struct node *) malloc (sizeof(struct node));
if(ptr == NULL)
    {
printf("\nOVERFLOW\n");
return;
    }
else
    {
printf("\nEnter value?\n");
scanf("%d",&item);
ptr -> data = item;
if(front == NULL)
    {
front = ptr;
rear = ptr;
front -> next = NULL;
rear -> next = NULL;
    }
else
    {
rear -> next = ptr;
rear = ptr;
rear->next = NULL;
    }
}
}
void delete ()

```

```
{
struct node *ptr;
if(front == NULL)
    {
printf("\nUNDERFLOW\n");
return;
    }
else
    {
ptr = front;
front = front -> next;
free(ptr);
    }
}
void display()
{
struct node *ptr;
ptr = front;
if(front == NULL)
    {
printf("\nEmpty queue\n");
    }
else
    { printf("\nprinting values ..... \n");
while(ptr != NULL)
    {
printf("\n%d\n",ptr -> data);
ptr = ptr -> next;
    }
    }
}
```

Output:

\*\*\*\*\*Main Menu\*\*\*\*\*

=====

- 1.insert an element
- 2.Delete an element
- 3.Display the queue



4.Exit

Enter your choice ?1

Enter value?

123

\*\*\*\*\*Main Menu\*\*\*\*\*

=====

- 1.insert an element
- 2.Delete an element
- 3.Display the queue
- 4.Exit

Enter your choice ?1

Enter value?

90

\*\*\*\*\*Main Menu\*\*\*\*\*

=====

- 1.insert an element
- 2.Delete an element
- 3.Display the queue
- 4.Exit

Enter your choice ?3

printing values .....

123

90

\*\*\*\*\*Main Menu\*\*\*\*\*

=====

- 1.insert an element
- 2.Delete an element
- 3.Display the queue
- 4.Exit

Enter your choice ?2

\*\*\*\*\*Main Menu\*\*\*\*\*

=====

- 1.insert an element
- 2.Delete an element
- 3.Display the queue
- 4.Exit

Enter your choice ?3

printing values .....

90

\*\*\*\*\*Main Menu\*\*\*\*\*

=====

- 1.insert an element
- 2.Delete an element
- 3.Display the queue
- 4.Exit

Enter your choice 4

## Unit 4 - Chapter 7

### Types of Queue

#### 7.0.Objective

#### 7.1.Circular Queue

#### 7.2.What is a Circular Queue?

##### 7.2.1.Procedure on Circular Queue

#### 7.3.Uses of Circular Queue

#### 7.4.Enqueue operation

#### 7.5.Algorithm to insert an element in a circular queue

#### 7.6.Dequeue Operation

##### 7.6.1.Algorithm to delete an element from the circular queue

#### 7.7.Implementation of circular queue using Array

#### 7.8.Implementation of circular queue using linked list

#### 7.9.Deque

#### 7.10.Operations on Deque

##### 7.10.1.Memory Representation

##### 7.10.2.What is a circular array?

##### 7.10.3.Applications of Deque

#### 7.11.Implementation of Deque using a circular array

#### 7.12.Dequeue Operation

#### 7.13.Program for deque Implementation

#### 7.14.What is a priority queue?

#### 7.15.Characteristics of a Priority queue

#### 7.16.Types of Priority Queue

##### 7.16.1.Ascending order priority queue

##### 7.16.2.Descending order priority queue

##### 7.16.3.Representation of priority queue

##### 7.17.Implementation of Priority Queue

#### 7.17.1.Analysis of complexities using different implementations

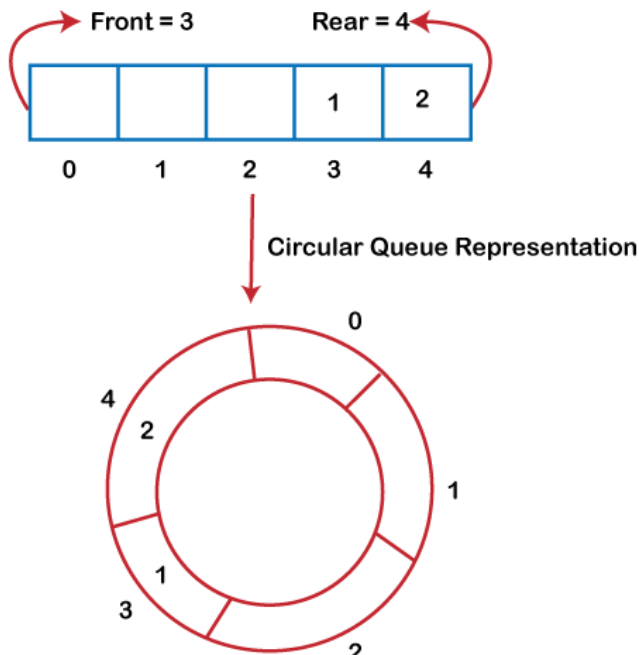
#### 7.0. Objective

This chapter would make you understand the following concepts:

- Understand the concept of Circular Queue
- Operation of Circular Queue
- Application of Circular Queue
- Implementation of Circular Queue

#### 7.1.Circular Queue

There was one limit in the exhibit usage of Queue. On the off chance that the back spans to the end position of the Queue, at that point there may be plausibility that some empty spaces are left to start with which can't be used. Thus, to defeat such restrictions, the idea of the round line was presented.



As we can find in the above picture, the back is at the last situation of the Queue and front is pointing some place as opposed to the 0<sup>th</sup> position. In the above exhibit, there are just two components and other three positions are unfilled. The back is at the last situation of the Queue; in the event that we attempt to embed the component, at that point it will show that there are no unfilled spaces in the Queue. There is one answer for maintain a strategic distance from such wastage of memory space by moving both the components at the left and change the front and backside as needs be. It's anything but a for all intents and purposes great methodology since moving all the components will burn-through loads of time. The effective way to deal with stay away from the wastage of the memory is to utilize circular queue data structure.

## 7.2.What is a Circular Queue?

A circular queue is like a linear queue as it is likewise founded on the FIFO (First In First Out) rule aside from that the last position is associated with the principal position in a round line that shapes a circle. It is otherwise called a Ring Buffer.

### 7.2.1.Procedure on Circular Queue

Coming up next are the activities that can be performed on a circular queue:

**Front:** It is utilized to get the front component from the Queue.

**Back:** It is utilized to get the back component from the Queue.

**enqueue(value):** This capacity is utilized to embed the new incentive in the Queue. The new component is constantly embedded from the backside.

**deQueue():** This capacity erases a component from the Queue. The cancellation in a Queue

consistently happens from the front end.

### 7.3.Uses of Circular Queue

The roundabout Queue can be utilized in the accompanying situations:

**Memory the board:** The roundabout queue gives memory the executives. As we have just seen that in linear queue, the memory isn't overseen proficiently. Yet, if there should arise an occurrence of a roundabout queue, the memory is overseen effectively by putting the components in an area which is unused.

**CPU Scheduling:** The working framework likewise utilizes the circular queue to embed the cycles and afterward execute them.

**Traffic framework:** In a PC control traffic framework, traffic signal is probably the best illustration of the circular queue. Each light of traffic signal gets ON individually after each jinterval of time. Like red light gets ON briefly then yellow light briefly and afterward green light. After green light, the red light gets ON.

### 7.4.Enqueue operation

**The steps of enqueue operation are given below:**

First, we will check whether the Queue is full or not.

Initially the front and rear are set to -1. When we insert the first element in a Queue, front and rear both are set to 0.

When we insert a new element, the rear gets incremented, i.e.,  $rear=rear+1$ .

#### Scenarios for inserting an element

There are two scenarios in which queue is not full:

If  $rear \neq \max - 1$ , then rear will be incremented to  $\text{mod}(\text{maxsize})$  and the new value will be inserted at the rear end of the queue.

If  $front \neq 0$  and  $rear = \max - 1$ , it means that queue is not full, then set the value of rear to 0 and insert the new element there.

There are two cases in which the element cannot be inserted:

When  $front == 0$  &&  $rear = \max - 1$ , which means that front is at the first position of the Queue and rear is at the last position of the Queue.

$front == rear + 1$ ;

### 7.5.Algorithm to insert an element in a circular queue

**Step 1:** IF  $(REAR+1)\%MAX = FRONT$

Write " OVERFLOW "

Goto step 4  
[End OF IF]

**Step 2:** IF FRONT = -1 and REAR = -1  
SET FRONT = REAR = 0  
ELSE IF REAR = MAX - 1 and FRONT != 0  
SET REAR = 0  
ELSE  
SET REAR = (REAR + 1) % MAX  
[END OF IF]

**Step 3:** SET QUEUE[REAR] = VAL

**Step 4:** EXIT

### **7.6.Dequeue Operation**

The means of dequeue activity are given underneath:

To start with, we check if the Queue is vacant. In the event that the queue is unfilled, we can't play out the dequeue activity.

At the point when the component is erased, the estimation of front gets decremented by 1.

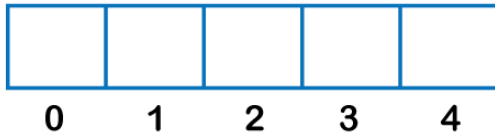
On the off chance that there is just a single component left which is to be erased, at that point the front and back are reset to - 1.

#### **7.6.1.Algorithm to delete an element from the circular queue**

Step 1: IF FRONT = -1  
Write " UNDERFLOW "  
Goto Step 4  
[END of IF]  
Step 2: SET VAL = QUEUE[FRONT]  
Step 3: IF FRONT = REAR  
SET FRONT = REAR = -1  
ELSE  
IF FRONT = MAX -1  
SET FRONT = 0  
ELSE  
SET FRONT = FRONT + 1  
[END of IF]  
[END OF IF]

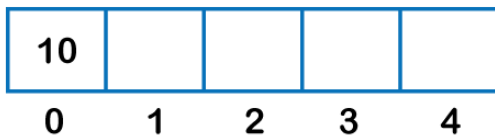
#### Step 4: EXIT

Let's understand the enqueue and dequeue operation through the diagrammatic representation.



**Front = -1**

**Rear = -1**



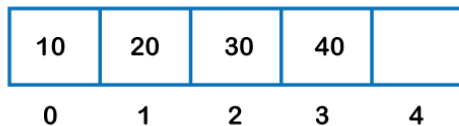
**Front = 0**

**Rear = 0**



**Front = 0**

**Rear = 2**



**Front = 0**

**Rear = 3**





```

else if((rear+1)%max==front) // condition to check queue is full
{
    printf("Queue is overflow..");
}
else
{
    rear=(rear+1)%max;    // rear is incremented
    queue[rear]=element; // assigning a value to the queue at the rear position.
}
}

```

// function to delete the element from the queue

```

int dequeue()
{
    if((front==-1) && (rear==-1)) // condition to check queue is empty
    {
        printf("\nQueue is underflow..");
    }
    else if(front==rear)
    {
        printf("\nThe dequeued element is %d", queue[front]);
        front=-1;
        rear=-1;
    }
    else
    {
        printf("\nThe dequeued element is %d", queue[front]);
        front=(front+1)%max;
    }
}

```

// function to display the elements of a queue

```

void display()
{
    int i=front;
    if(front==-1 && rear==-1)
    {
        printf("\n Queue is empty..");
    }
    else
    {

```

```

printf("\nElements in a Queue are :");
while(i<=rear)
{
    printf("%d,", queue[i]);
    i=(i+1)%max;
}
}
}
int main()
{
    int choice=1,x; // variables declaration

    while(choice<4 && choice!=0) // while loop
    {
        printf("\n Press 1: Insert an element");
        printf("\nPress 2: Delete an element");
        printf("\nPress 3: Display the element");
        printf("\nEnter your choice");
        scanf("%d", &choice);

        switch(choice)
        {

            case 1:

                printf("Enter the element which is to be inserted");
                scanf("%d", &x);
                enqueue(x);
                break;
            case 2:
                dequeue();
                break;
            case 3:
                display();

        }
    }
    return 0;
}

```

**Output:**

## Output:

```
Press 1: Insert an element
Press 2: Delete an element
Press 3: Display the element
Enter your choice
1
Enter the element which is to be inserted
10

Press 1: Insert an element
Press 2: Delete an element
Press 3: Display the element
Enter your choice
1
Enter the element which is to be inserted
20

Press 1: Insert an element
Press 2: Delete an element
Press 3: Display the element
Enter your choice
1
Enter the element which is to be inserted
30

Press 1: Insert an element
Press 2: Delete an element
Press 3: Display the element
Enter your choice
3

Elements in a Queue are :10,20,30,
Press 1: Insert an element
Press 2: Delete an element
Press 3: Display the element
Enter your choice
2

The dequeued element is 10
```

## 7.8.Implementation of circular queue using linked list

As we realize that connected rundown is a direct information structure that stores two sections, i.e., information part and the location part where address part contains the location of the following hub. Here, connected rundown is utilized to execute the roundabout line; in this way, the connected rundown follows the properties of the Queue. At the point when we are actualizing the roundabout line utilizing connected rundown then both the enqueue and dequeue tasks take  $O(1)$  time.

```
#include <stdio.h>
// Declaration of struct type node
struct node
{
```

```

int data;
struct node *next;
};
struct node *front=-1;
struct node *rear=-1;
// function to insert the element in the Queue
void enqueue(int x)
{
struct node *newnode; // declaration of pointer of struct node type.
newnode=(struct node *)malloc(sizeof(struct node)); // allocating the memory to the
newnode
newnode->data=x;
newnode->next=0;
if(rear==-1) // checking whether the Queue is empty or not.
{
front=rear=newnode;
rear->next=front;
}
else
{
rear->next=newnode;
rear=newnode;
rear->next=front;
}
}

// function to delete the element from the queue
void dequeue()
{
struct node *temp; // declaration of pointer of node type
temp=front;
if((front==-1)&&(rear==-1)) // checking whether the queue is empty or not
{
printf("\nQueue is empty");
}
else if(front==rear) // checking whether the single element is left in the queue
{
front=rear=-1;
free(temp);
}
}

```

```
else
{
front=front->next;
rear->next=front;
free(temp);
}
}
```

```
// function to get the front of the queue
```

```
int peek()
{
if((front==-1) &&(rear==-1))
{
printf("\nQueue is empty");
}
else
{
printf("\nThe front element is %d", front->data);
}
}
```

```
// function to display all the elements of the queue
```

```
void display()
{
struct node *temp;
temp=front;
printf("\n The elements in a Queue are : ");
if((front==-1) && (rear==-1))
{
printf("Queue is empty");
}
}
```

```
else
{
while(temp->next!=front)
{
printf("%d", temp->data);
temp=temp->next;
}
printf("%d", temp->data);
}
```

```

    }
}

void main()
{
enqueue(34);
enqueue(10);
enqueue(23);
display();
dequeue();
peek();
}

```

**Output:**

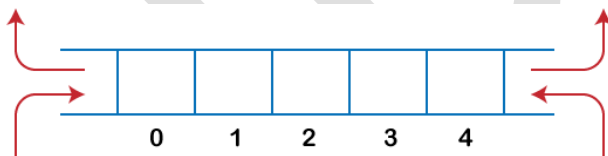
```

The elements in a Queue are : 34,10,23
The front element is 10
...Program finished with exit code 24
Press ENTER to exit console.

```

**7.9.Deque**

The dequeue represents Double Ended Queue. In the queue, the inclusion happens from one end while the erasure happens from another end. The end at which the addition happens is known as the backside while the end at which the erasure happens is known as front end.

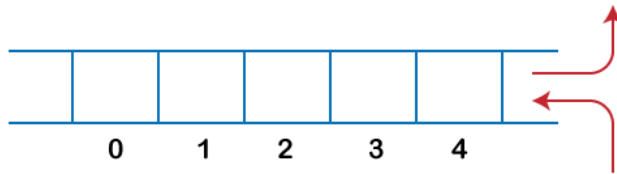


Deque is a direct information structure in which the inclusion and cancellation tasks are performed from the two finishes. We can say that deque is a summed up form of the line.

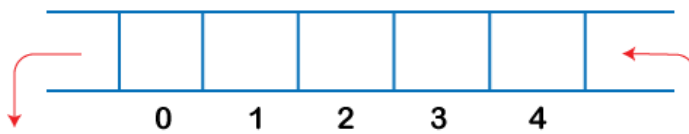
How about we take a gander at certain properties of deque.

Deque can be utilized both as stack and line as it permits the inclusion and cancellation procedure on the two finishes.

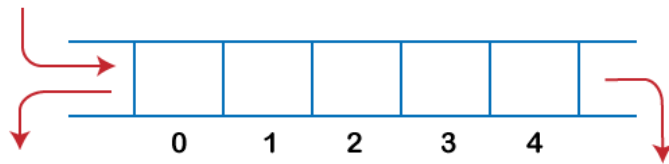
In deque, the inclusion and cancellation activity can be performed from one side. The stack adheres to the LIFO rule in which both the addition and erasure can be performed distinctly from one end; in this way, we reason that deque can be considered as a stack.



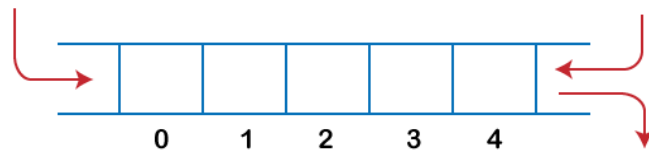
In deque, the addition can be performed toward one side, and the erasure should be possible on another end. The queue adheres to the FIFO rule in which the component is embedded toward one side and erased from another end. Hence, we reason that the deque can likewise be considered as the queue.



There are two types of Queues, Input-restricted queue, and output-restricted queue. Information confined queue: The info limited queue implies that a few limitations are applied to the inclusion. In info confined queue, the addition is applied to one end while the erasure is applied from both the closures.



Yield confined queue: The yield limited line implies that a few limitations are applied to the erasure activity. In a yield limited queue, the cancellation can be applied uniquely from one end, while the inclusion is conceivable from the two finishes.



### 7.10.Operations on Deque

The following are the operations applied on deque:

Insert at front

Delete from end

insert at rear

delete from rear

Other than inclusion and cancellation, we can likewise perform look activity in deque. Through look activity, we can get the front and the back component of the deque.

**We can perform two additional procedure on dequeue:**

**isFull():** This capacity restores a genuine worth if the stack is full; else, it restores a bogus worth.

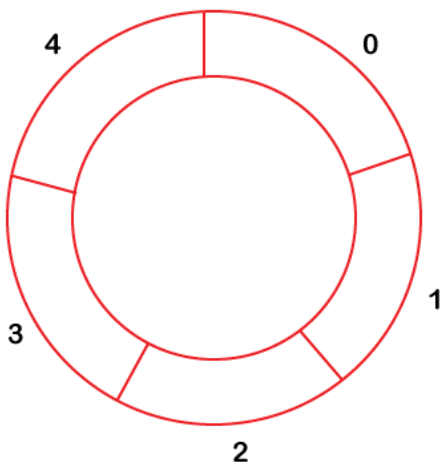
**isEmpty():** This capacity restores a genuine worth if the stack is vacant; else it restores a bogus worth.

### 7.10.1. Memory Representation

The deque can be executed utilizing two information structures, i.e., round exhibit, and doubly connected rundown. To actualize the deque utilizing round exhibit, we initially should realize what is roundabout cluster.

### 7.10.2. What is a circular array?

An exhibit is supposed to be roundabout if the last component of the cluster is associated with the primary component of the exhibit. Assume the size of the cluster is 4, and the exhibit is full however the primary area of the cluster is unfilled. In the event that we need to embed the exhibit component, it won't show any flood condition as the last component is associated with the primary component. The worth which we need to embed will be included the primary area of the exhibit.



### 7.10.3. Applications of Deque

- The deque can be utilized as a stack and line; subsequently, it can perform both re-try and fix activities.
- It tends to be utilized as a palindrome checker implies that in the event that we read the string from the two closures, at that point the string would be the equivalent.



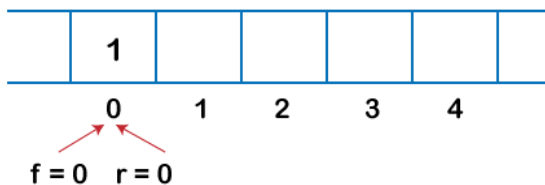
- It tends to be utilized for multiprocessor planning. Assume we have two processors, and every processor has one interaction to execute. Every processor is appointed with an interaction or a task, and each cycle contains numerous strings. Every processor keeps a deque that contains strings that are prepared to execute. The processor executes an interaction, and on the off chance that a cycle makes a kid cycle, at that point that cycle will be embedded at the front of the deque of the parent interaction. Assume the processor P2 has finished the execution of every one of its strings then it takes the string from the backside of the processor P1 and adds to the front finish of the processor P2. The processor P2 will take the string from the front end; thusly, the erasure takes from both the closures, i.e., front and backside. This is known as the A-take calculation for planning.

### 7.11.Implementation of Deque using a circular array

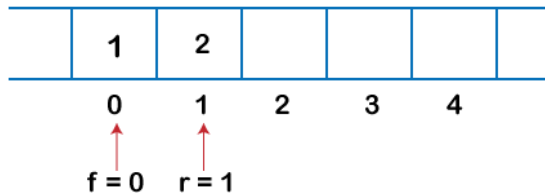
The following are the steps to perform the operations on the Deque:

#### Enqueue operation

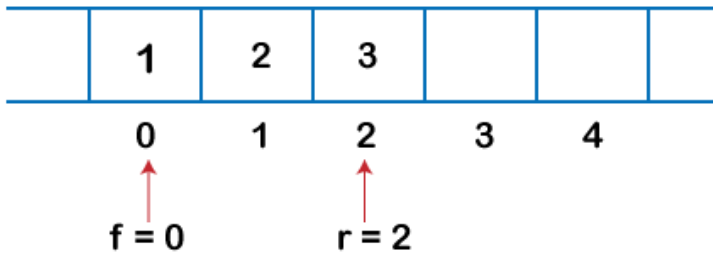
1. At first, we are thinking about that the deque is unfilled, so both front and back are set to - 1, i.e.,  $f = - 1$  and  $r = - 1$ .
2. As the deque is vacant, so embeddings a component either from the front or backside would be something very similar. Assume we have embedded component 1, at that point front is equivalent to 0, and the back is likewise equivalent to 0.



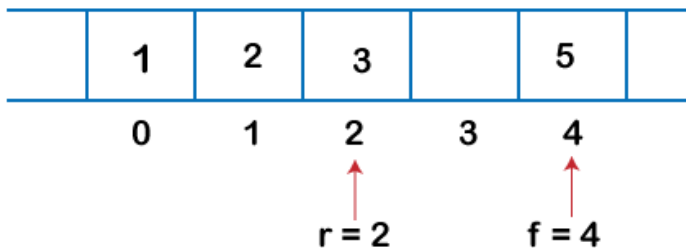
3. Assume we need to embed the following component from the back. To embed the component from the backside, we first need to augment the back, i.e.,  $\text{rear}=\text{rear}+1$ . Presently, the back is highlighting the subsequent component, and the front is highlighting the main component.



4. Assume we are again embeddings the component from the backside. To embed the component, we will first addition the back, and now back focuses to the third component.

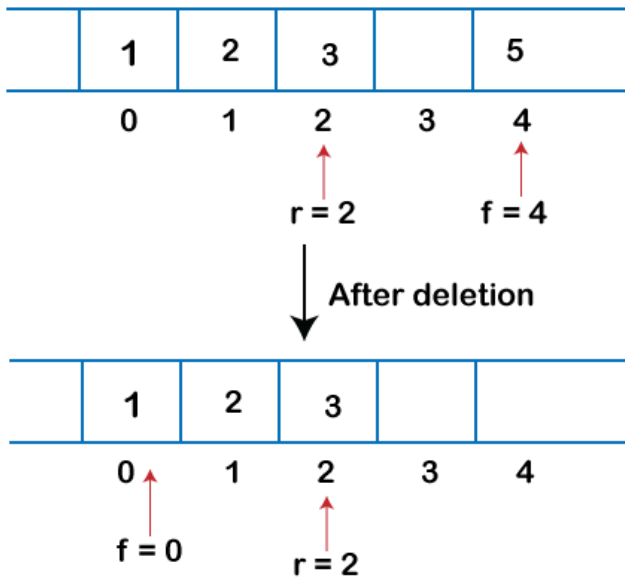


5. In the event that we need to embed the component from the front end, and addition a component from the front, we need to decrement the estimation of front by 1. In the event that we decrement the front by 1, at that point the front focuses to - 1 area, which isn't any substantial area in an exhibit. Thus, we set the front as  $(n - 1)$ , which is equivalent to 4 as  $n$  is 5. When the front is set, we will embed the incentive as demonstrated in the beneath figure:

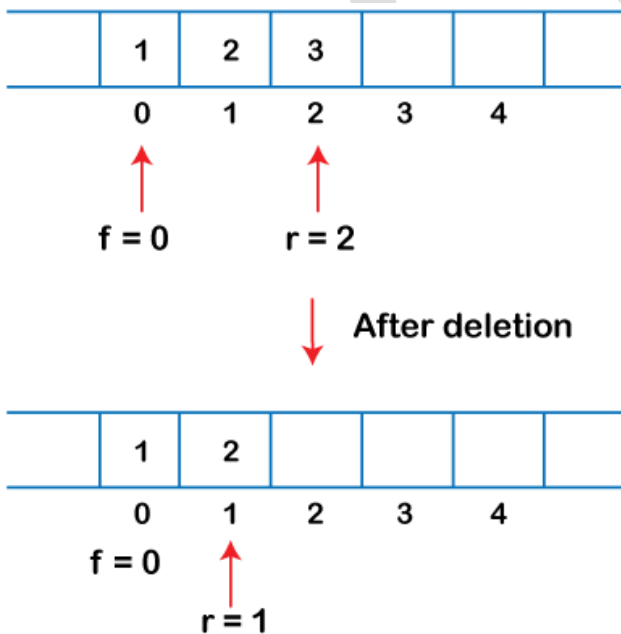


### 7.12.Dequeue Operation

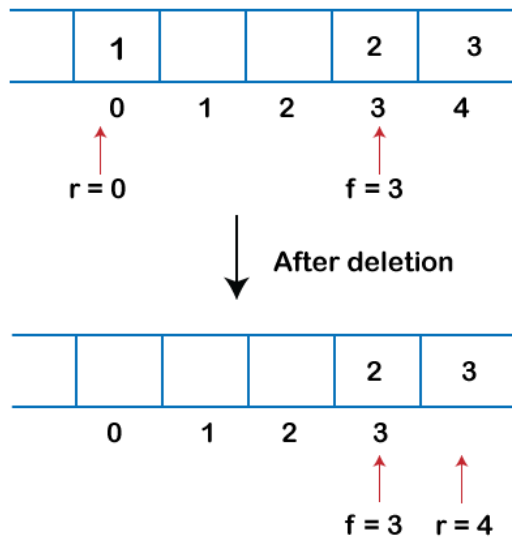
1. On the off chance that the front is highlighting the last component of the exhibit, and we need to play out the erase activity from the front. To erase any component from the front, we need to set  $\text{front} = \text{front} + 1$ . At present, the estimation of the front is equivalent to 4, and in the event that we increase the estimation of front, it becomes 5 which is definitely not a substantial list. Thusly, we presume that in the event that front focuses to the last component, at that point front is set to 0 if there should be an occurrence of erase activity.



2. If we want to delete the element from rear end then we need to decrement the rear value by 1, i.e.,  $rear = rear - 1$  as shown in the below figure:



3. In the event that the back is highlighting the principal component, and we need to erase the component from the backside then we need to set  $rear = n - 1$  where  $n$  is the size of the exhibit as demonstrated in the beneath figure:



Let's create a program of deque.

The following are the six functions that we have used in the below program:

- **enqueue\_front():** It is used to insert the element from the front end.
- **enqueue\_rear():** It is used to insert the element from the rear end.
- **dequeue\_front():** It is used to delete the element from the front end.
- **dequeue\_rear():** It is used to delete the element from the rear end.
- **getfront():** It is used to return the front element of the deque.
- **getrear():** It is used to return the rear element of the deque.

### 7.13. Program for deque Implementation

```
#define size 5
#include <stdio.h>
int deque[size];
int f=-1, r=-1;
// enqueue_front function will insert the value from the front
void enqueue_front(int x)
{
    if((f==0 && r==size-1) || (f==r+1))
    {
        printf("deque is full");
    }
    else if((f==-1) && (r==-1))
    {
```

```
    f=r=0;
    deque[f]=x;
}
else if(f==0)
{
    f=size-1;
    deque[f]=x;
}
else
{
    f=f-1;
    deque[f]=x;
}
}
```

// enqueue\_rear function will insert the value from the rear

```
void enqueue_rear(int x)
{
    if((f==0 && r==size-1) || (f==r+1))
    {
        printf("deque is full");
    }
    else if((f==-1) && (r==-1))
    {
        r=0;
        deque[r]=x;
    }
    else if(r==size-1)
    {
        r=0;
        deque[r]=x;
    }
    else
    {
        r++;
        deque[r]=x;
    }
}
```

// display function prints all the value of deque.

```
void display()
{
    int i=f;
    printf("\n Elements in a deque : ");

    while(i!=r)
    {
        printf("%d ",deque[i]);
        i=(i+1)%size;
    }
    printf("%d",deque[r]);
}
```

// getfront function retrieves the first value of the deque.

```
void getfront()
{
    if((f==-1) && (r==-1))
    {
        printf("Deque is empty");
    }
    else
    {
        printf("\nThe value of the front is: %d", deque[f]);
    }
}
```

// getrear function retrieves the last value of the deque.

```
void getrear()
{
    if((f==-1) && (r==-1))
    {
        printf("Deque is empty");
    }
    else
    {
        printf("\nThe value of the rear is: %d", deque[r]);
    }
}
```

```
}
```

// dequeue\_front() function deletes the element from the front

```
void dequeue_front()
{
    if((f==0) && (r==0))
    {
        printf("Deque is empty");
    }
    else if(f==r)
    {
        printf("\nThe deleted element is %d", deque[f]);
        f=-1;
        r=-1;
    }
    else if(f==(size-1))
    {
        printf("\nThe deleted element is %d", deque[f]);
        f=0;
    }
    else
    {
        printf("\nThe deleted element is %d", deque[f]);
        f=f+1;
    }
}
```

// dequeue\_rear() function deletes the element from the rear

```
void dequeue_rear()
{
    if((f==0) && (r==0))
    {
        printf("Deque is empty");
    }
    else if(f==r)
    {
        printf("\nThe deleted element is %d", deque[r]);
        f=-1;
        r=-1;
    }
}
```

```

}
else if(r==0)
{
    printf("\nThe deleted element is %d", deque[r]);
    r=size-1;
}
else
{
    printf("\nThe deleted element is %d", deque[r]);
    r=r-1;
}
}

```

```

int main()
{
    // inserting a value from the front.
    enqueue_front(2);
    // inserting a value from the front.
    enqueue_front(1);
    // inserting a value from the rear.
    enqueue_rear(3);
    // inserting a value from the rear.
    enqueue_rear(5);
    // inserting a value from the rear.
    enqueue_rear(8);
    // Calling the display function to retrieve the values of deque
    display();
    // Retrieve the front value
    getfront();
    // Retrieve the rear value.
    getrear();
    // deleting a value from the front
    dequeue_front();
    //deleting a value from the rear
    dequeue_rear();
    // Calling the display function to retrieve the values of deque
    display();
    return 0;
}

```



## Output:

```
Elements in a deque : 1 2 3 5 8
The value of the front is: 1
The value of the rear is: 8
The deleted element is 1
The deleted element is 8
Elements in a deque : 2 3 5
...Program finished with exit code 0
Press ENTER to exit console.
```

### 7.14. What is a priority queue?

A need queue is a theoretical information type that carries on comparatively to the ordinary queue aside from that every component has some need, i.e., the component with the most elevated need would start things out in a need line. The need of the components in a need queue will decide the request where components are taken out from the need line.

The need queue underpins just similar components, which implies that the components are either masterminded in a rising or slipping request.

For instance, assume we have a few qualities like 1, 3, 4, 8, 14, 22 embedded in a need queue with a requesting forced on the qualities is from least to the best. Along these lines, the 1 number would have the most elevated need while 22 will have the least need.

### 7.15. Characteristics of a Priority queue

A need queue is an expansion of a line that contains the accompanying qualities:

- o Every component in a need line has some need related with it.
- o An component with the higher need will be erased before the cancellation of the lesser need.
- o If two components in a need queue have a similar need, they will be organized utilizing the FIFO rule.

**Let's understand the priority queue through an example.**

We have a priority queue that contains the following values:

1, 3, 4, 8, 14, 22

All the qualities are orchestrated in climbing request. Presently, we will see how the need line will take care of playing out the accompanying activities:

**poll():** This capacity will eliminate the most elevated need component from the need line. In the above need line, the '1' component has the most elevated need, so it will be eliminated from the need line.

**add(2):** This capacity will embed '2' component in a need line. As 2 is the littlest component among all the numbers so it will acquire the most elevated need.

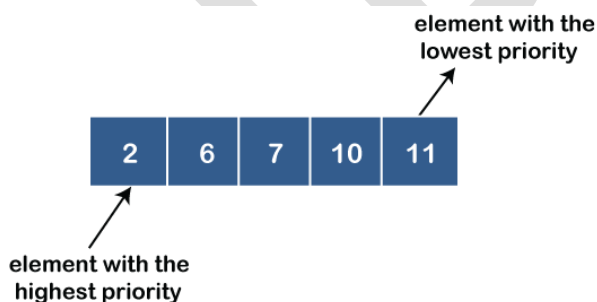
**poll()** It will eliminate '2' component from the need line as it has the most elevated need line.

**add(5):** It will embed 5 component after 4 as 5 is bigger than 4 and lesser than 8, so it will acquire the third most noteworthy need in a need line.

## 7.16.Types of Priority Queue

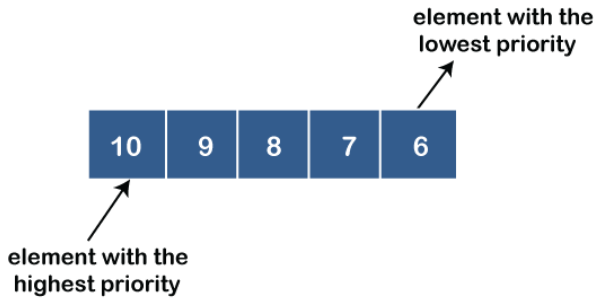
There are two types of priority queue:

**7.16.1.Ascending order priority queue:** In rising request need line, a lower need number is given as a higher need in a need. For instance, we take the numbers from 1 to 5 orchestrated in a rising request like 1,2,3,4,5; in this manner, the most modest number, i.e., 1 is given as the most noteworthy need in a need line.



## 7.16.2.Descending order priority queue:

In plunging request need line, a higher need number is given as a higher need in a need. For instance, we take the numbers from 1 to 5 orchestrated in diving request like 5, 4, 3, 2, 1; along these lines, the biggest number, i.e., 5 is given as the most elevated need in a need line.



### 7.16.3. Representation of priority queue

Presently, we will perceive how to address the need line through a single direction list.

We will make the need line by utilizing the rundown given underneath in which INFO list contains the information components, PRN list contains the need quantities of every information component accessible in the INFO rundown, and LINK essentially contains the location of the following hub.

	INFO	PNR	LINK
0	200	2	4
1	400	4	2
2	500	4	6
3	300	1	0
4	100	2	5
5	600	3	1
6	700	4	

Let's create the priority queue step by step.

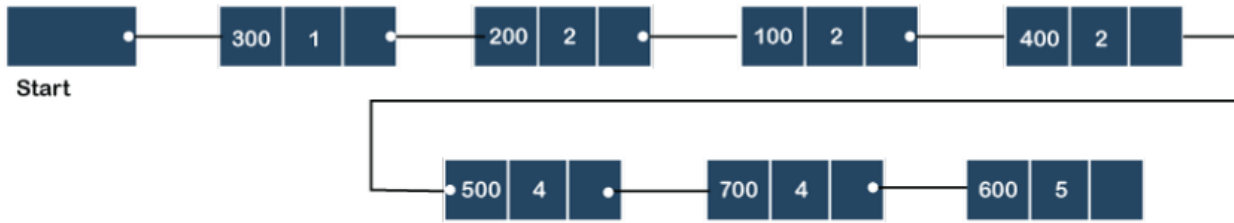
In the case of priority queue, lower priority number is considered the higher priority, i.e., lower priority number = higher priority.

**Step 1:** In the list, lower priority number is 1, whose data value is 333, so it will be inserted in the list as shown in the below diagram:

**Step 2:** After inserting 333, priority number 2 is having a higher priority, and data values associated with this priority are 222 and 111. So, this data will be inserted based on the FIFO principle; therefore 222 will be added first and then 111.

**Step 3:** After inserting the elements of priority 2, the next higher priority number is 4 and data elements associated with 4 priority numbers are 444, 555, 777. In this case, elements would be inserted based on the FIFO principle; therefore, 444 will be added first, then 555, and then 777.

**Step 4:** After inserting the elements of priority 4, the next higher priority number is 5, and the value associated with priority 5 is 666, so it will be inserted at the end of the queue.



### 7.17.Implementation of Priority Queue:

The need queue can be actualized in four different ways that incorporate clusters, connected rundown, stack information construction and twofold pursuit tree. The load information structure is the most productive method of executing the need queue, so we will actualize the need queue utilizing a store information structure in this subject. Presently, first we comprehend the motivation behind why pile is the most productive route among the wide range of various information structures.

#### 7.17.1.Analysis of complexities using different implementations

Implementation	add	Remove	peek
Linked list	$O(1)$	$O(n)$	$O(n)$
Binary heap	$O(\log n)$	$O(\log n)$	$O(1)$
Binary search tree	$O(\log n)$	$O(\log n)$	$O(1)$

## **Unit 4 : Chapter 8**

### **Linked List**

#### **8.0 Objective**

#### **8.1.What is Linked List?**

#### **8.2.How can we declare the Linked list?**

#### **8.3.Advantages of using a Linked list over Array**

#### **8.4.Applications of Linked List**

#### **8.5.Types of Linked List**

##### **8.5.1.Singly Linked list**

##### **8.5.2.Doubly linked list**

##### **8.5.3.Circular linked list**

##### **8.5.4.Doubly Circular linked list**

#### **8.6.Linked List**

#### **8.7.Uses of Linked List**

#### **8.8.Why use linked list over array?**

##### **8.8.1.Singly linked list or One way chain**

##### **8.8.2.Operations on Singly Linked List**

##### **8.8.3.Linked List in C: Menu Driven Program**

#### **8.9.Doubly linked list**

##### **8.9.1.Memory Representation of a doubly linked list**

##### **8.9.2.Operations on doubly linked list**

##### **8.9.3.Menu Driven Program in C to implement all the operations of doubly linked list**

#### **8.0.Objective**

**This chapter would make you understand the following concepts:**

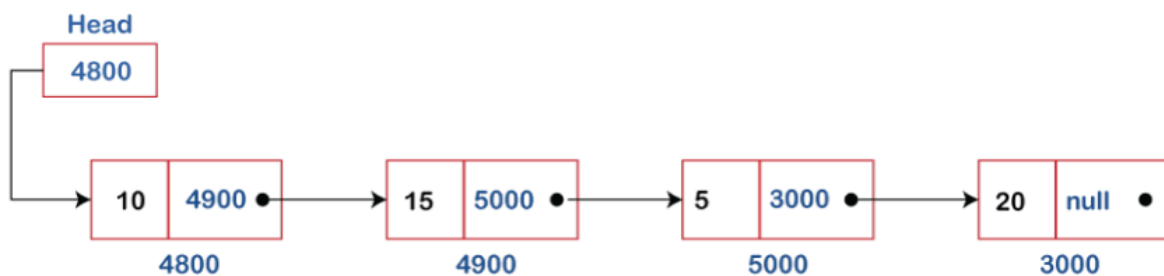
- To understand the concept of Linked List**
- To understand Types of Linked List**
- To Singly Linked list**
- To Doubly Linked list**

## 8.1.What is Linked List?

A linked list is also a collection of elements, but the elements are not stored in a consecutive location. Suppose a programmer made a request for storing the integer value then size of 4-byte memory block is assigned to the integer value. The programmer made another request for storing 3 more integer elements; then, three different memory blocks are assigned to these three elements but the memory blocks are available in a random location. So, how are the elements connected?.

These elements are linked to each other by providing one additional information along with an element, i.e., the address of the next element. The variable that stores the address of the next element is known as a pointer. Therefore, we conclude that the linked list contains two parts, i.e., the first one is the data element, and the other is the pointer. The pointer variable will occupy 4 bytes which is pointing to the next element.

*A linked list can also be defined as the collection of the nodes in which one node is connected to another node, and node consists of two parts, i.e., one is the data part and the second one is the address part, as shown in the below figure:*



In the above figure, we can observe that each node contains the data and the address of the next node. The last node of the linked list contains the NULL value in the address part.

## 8.2.How can we declare the Linked list?

The need line can be actualized in four different ways that incorporate clusters, connected rundown, stack information construction and twofold pursuit tree. The load information structure is the most productive method of executing the need line, so we will actualize the need line utilizing a store information structure in this

subject. Presently, first we comprehend the motivation behind why pile is the most productive route among the wide range of various information structures.

The structure of a linked list can be defined as:

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

In the above declaration, we have defined a structure named as a node consisting of two variables: an integer variable (data), and the other one is the pointer (next), which contains the address of the next node.

### **8.3. Advantages of using a Linked list over Array**

**The following are the advantages of using a linked list over an array:**

#### **Dynamic data structure:**

The size of the linked list is not fixed as it can vary according to our requirements.

#### **Insertion and Deletion:**

Insertion and deletion in linked list are easier than array as the elements in an array are stored in a consecutive location. In contrast, in the case of a linked list, the elements are stored in a random location. The complexity for insertion and deletion of elements from the beginning is  $O(1)$  in the linked list, while in the case of an array, the complexity would be  $O(n)$ . If we want to insert or delete the element in an array, then we need to shift the elements for creating the space. On the other hand, in the linked list, we do not have to shift the elements. In the linked list, we just need to update the address of the pointer in the node.

#### **Memory efficient**

Its memory consumption is efficient as the size of the linked list can grow or shrink according to our requirements.

#### **Implementation**

Both the stacks and queues can be implemented using a linked list.

#### **Disadvantages of Linked list**

The following are the disadvantages of linked list:

### Memory usage

The node in a linked list occupies more memory than array as each node occupies two types of variables, i.e., one is a simple variable, and another is a pointer variable that occupies 4 bytes in the memory.

### Traversal

In a linked list, the traversal is not easy. If we want to access the element in a linked list, we cannot access the element randomly, but in the case of an array, we can randomly access the element by index. For example, if we want to access the 3rd node, then we need to traverse all the nodes before it. So, the time required to access a particular node is large.

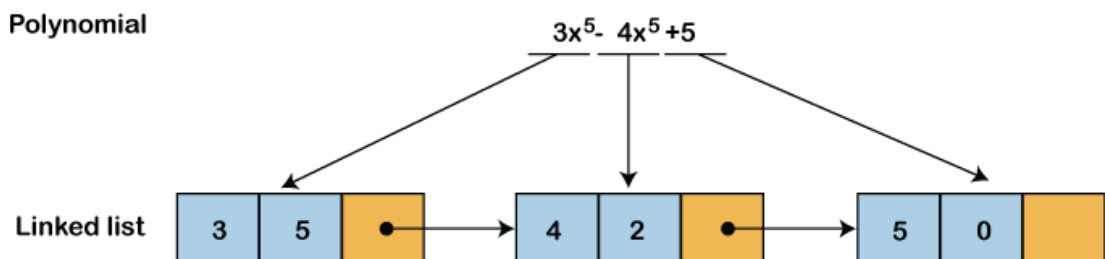
### Reverse traversing

In a linked list, backtracking or reverse traversing is difficult. In a doubly linked list, it is easier but requires more memory to store the back pointer.

## 8.4.Applications of Linked List

The applications of the linked list are given below:

- With the assistance of a connected rundown, the polynomials can be addressed just as we can play out the procedure on the polynomial. We realize that polynomial is an assortment of terms where each term contains coefficient and force. The coefficients and force of each term are put away as hub and connection pointer focuses to the following component in a connected rundown, so connected rundown can be utilized to make, erase and show the polynomial.





- An inadequate grid is utilized in logical calculation and mathematical investigation. In this way, a connected rundown is utilized to address the scanty grid.
- The different tasks like understudy's subtleties, worker's subtleties or item subtleties can be executed utilizing the connected rundown as the connected rundown utilizes the design information type that can hold distinctive information types.
- Stack, Queue, tree and different other information constructions can be executed utilizing a connected rundown.
- The chart is an assortment of edges and vertices, and the diagram can be addressed as a nearness lattice and contiguousness list. In the event that we need to address the diagram as a nearness network, at that point it very well may be actualized as an exhibit. In the event that we need to address the diagram as a nearness list, at that point it tends to be actualized as a connected rundown.
- To actualize hashing, we require hash tables. The hash table contains sections that are executed utilizing connected rundown.
- A connected rundown can be utilized to execute dynamic memory designation. The powerful memory distribution is the memory designation done at the run-time.

## **8.5.Types of Linked List**

Before knowing about the types of a linked list, we should know what is linked list. So, to know about the linked list, click on the link given below:

### **Types of Linked list**

**The following are the types of linked list:**

Singly Linked list

Doubly Linked list

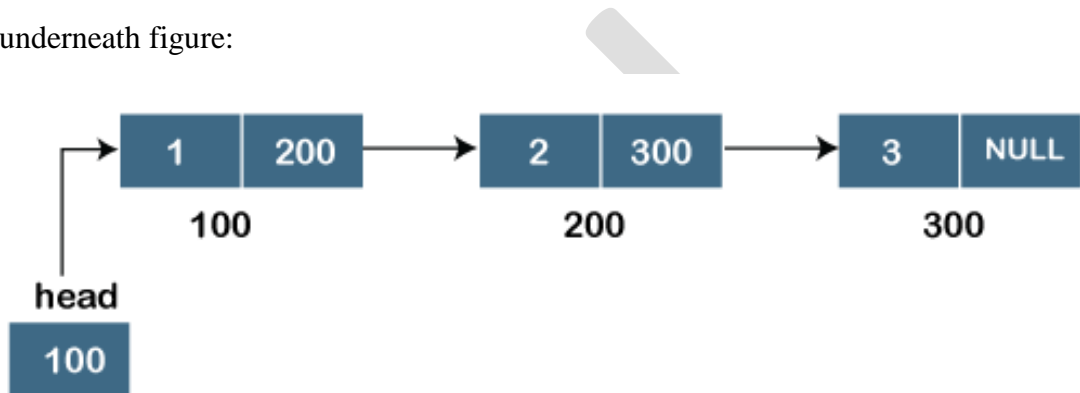
Circular Linked list

Doubly Circular Linked list

### **8.5.1.Singly Linked list**

It is the normally utilized connected rundown in projects. In the event that we are discussing the connected show, it implies it is a separately connected rundown. The separately connected rundown is an information structure that contains two sections, i.e., one is the information part, and the other one is the location part, which contains the location of the following or the replacement hub. The location part in a hub is otherwise called a pointer.

Assume we have three hubs, and the locations of these three hubs are 100, 200 and 300 separately. The portrayal of three hubs as a connected rundown is appeared in the underneath figure:



We can see in the above figure that there are three unique hubs having address 100, 200 and 300 individually. The principal hub contains the location of the following hub, i.e., 200, the subsequent hub contains the location of the last hub, i.e., 300, and the third hub contains the NULL incentive in its location part as it doesn't highlight any hub. The pointer that holds the location of the underlying hub is known as a head pointer.

The connected rundown, which is appeared in the above outline, is referred to as an independently connected rundown as it contains just a solitary connection. In this rundown, just forward crossing is conceivable; we can't navigate the regressive way as it has just one connection in the rundown.

Representation of the node in a singly linked list

struct node

{

int data;

```

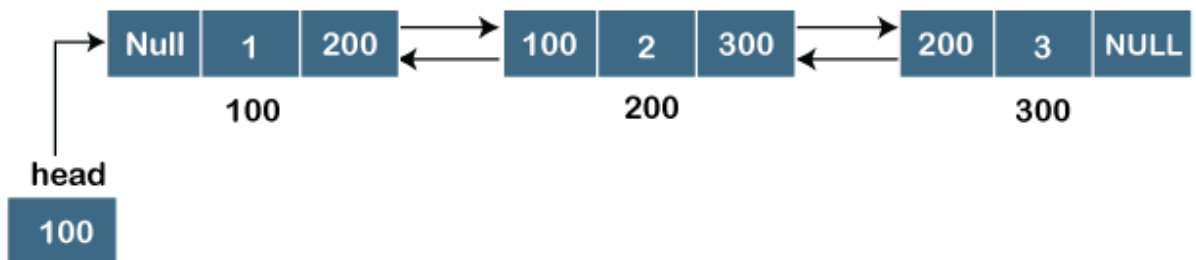
struct node *next;
}

```

### 8.5.2. Doubly linked list

As the name recommends, the doubly connected rundown contains two pointers. We can characterize the doubly connected rundown as a straight information structure with three sections: the information part and the other two location part. All in all, a doubly connected rundown is a rundown that has three sections in a solitary hub, incorporates one information section, a pointer to its past hub, and a pointer to the following hub.

Assume we have three hubs, and the location of these hubs are 100, 200 and 300, separately. The portrayal of these hubs in a doubly-connected rundown is appeared beneath:



As we can see in the above figure, the hub in a doubly-connected rundown has two location parts; one section stores the location of the following while the other piece of the hub stores the past hub's location. The underlying hub in the doubly connected rundown has the NULL incentive in the location part, which gives the location of the past hub.

#### Representation of the node in a doubly linked list

```

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

```

In the above portrayal, we have characterized a client characterized structure named a hub with three individuals, one is information of number sort, and the other two are the pointers, i.e., next and prev of the hub type. The following pointer variable holds the location of the following hub, and the prev pointer holds the location of the past hub. The sort of both the pointers, i.e., next and prev is struct hub as both the pointers are putting away the location of the hub of the struct hub type.

### 8.5.3.Circular linked list

A round connected rundown is a variety of an independently connected rundown. The lone contrast between the separately connected rundown and a round connected rundown is that the last hub doesn't highlight any hub in an independently connected rundown, so its connection part contains a NULL worth. Then again, the roundabout connected rundown is a rundown where the last hub interfaces with the principal hub, so the connection a piece of the last hub holds the main hub's location. The round connected rundown has no beginning and finishing hub. We can navigate toward any path, i.e., either in reverse or forward. The diagrammatic portrayal of the round connected rundown is appeared underneath:

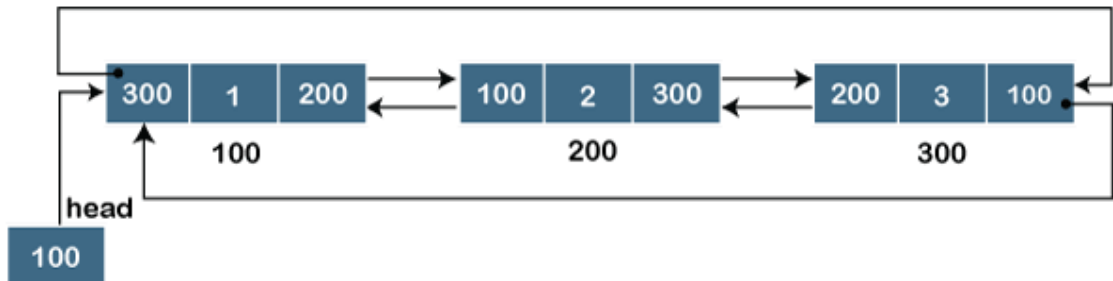
```
struct node
{
int data;
struct node *next;
}
```

A circular linked list is a sequence of elements in which each node has a link to the next node, and the last node is having a link to the first node. The representation of the circular linked list will be similar to the singly linked list, as shown below:



#### 8.5.4. Doubly Circular linked list

The doubly circular linked list has the features of both the circular linked list and doubly linked list.



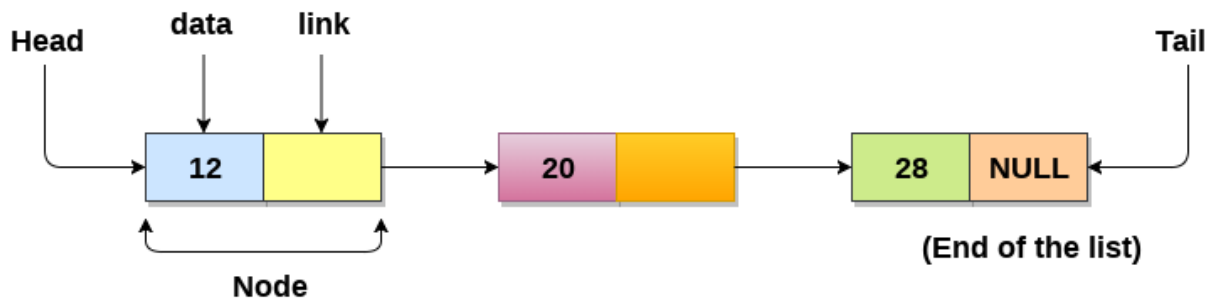
The above figure shows the portrayal of the doubly round connected rundown wherein the last hub is appended to the principal hub and consequently makes a circle. It is a doubly connected rundown likewise in light of the fact that every hub holds the location of the past hub too. The primary distinction between the doubly connected rundown and doubly roundabout connected rundown is that the doubly roundabout connected rundown doesn't contain the NULL incentive in the past field of the hub. As the doubly roundabout connected contains three sections, i.e., two location parts and one information part so its portrayal is like the doubly connected rundown.

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

#### 8.6. Linked List

- Linked List can be defined as collection of objects called nodes that are randomly stored in the memory.
- A node contains two fields i.e. data stored at that particular address and the pointer which contains the address of the next node in the memory.

- The last node of the list contains pointer to the null.



### 8.7.Uses of Linked List

- The rundown isn't needed to be adjoiningly present in the memory. The hub can dwell anyplace in the memory and connected together to make a rundown. This accomplishes advanced usage of room.
- list size is restricted to the memory size and shouldn't be announced ahead of time.
- Void hub can not be available in the connected rundown.
- We can store estimations of crude sorts or items in the separately connected rundown.

### 8.8.Why use linked list over array?

Till now, we were utilizing cluster information construction to sort out the gathering of components that are to be put away separately in the memory. Nonetheless, Array has a few points of interest and hindrances which should be known to choose the information structure which will be utilized all through the program.

#### Array contains following limitations:

- The size of cluster should be known ahead of time prior to utilizing it in the program.
- Expanding size of the cluster is a period taking cycle. It is practically difficult to grow the size of the exhibit at run time.
- All the components in the cluster require to be adorningly put away in the memory. Embedding's any component in the cluster needs moving of every one of its archetypes.

Linked list is the data structure which can overcome all the limitations of an array. Using linked list is useful because

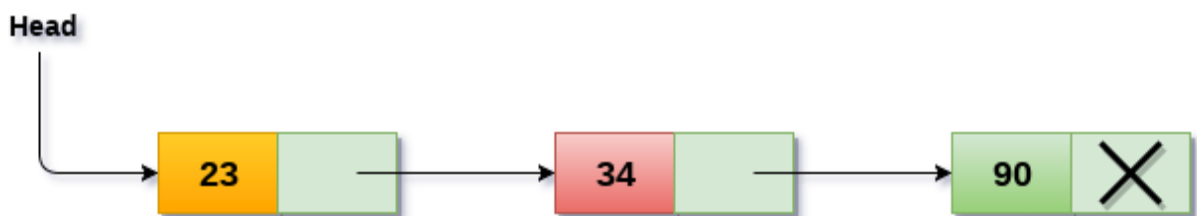
- It dispenses the memory progressively. All the hubs of connected rundown are non-adjacently put away in the memory and connected along with the assistance of pointers.
- Measuring is not, at this point an issue since we don't have to characterize its size at the hour of affirmation. Rundown develops according to the program's interest and restricted to the accessible memory space.

### 8.8.1.Singly linked list or One way chain

Separately connected rundown can be characterized as the assortment of requested arrangement of components. The quantity of components may shift as indicated by need of the program. A hub in the independently connected rundown comprise of two sections: information part and connection part. Information some portion of the hub stores real data that will be addressed by the hub while the connection a piece of the hub stores the location of its nearby replacement.

One way chain or separately connected rundown can be crossed distinctly one way. As such, we can say that every hub contains just next pointer, hence we can not cross the rundown the opposite way.

Consider a model where the imprints acquired by the understudy in three subjects are put away in a connected rundown as demonstrated in the figure.



In the above figure, the bolt addresses the connections. The information a piece of each hub contains the imprints acquired by the understudy in the diverse subject. The last hub in the rundown is recognized by the invalid pointer which is available in the location part of the last hub. We can have as numerous components we need, in the information part of the rundown.

## Complexity

Data Structure	Time Complexity								Space Compleity
	Average				Worst				Worst
	Access	Search	Insertion	Deletion	Access	Search	Insertion	Deletion	
Singly Linked List	$\theta(n)$	$\theta(n)$	$\theta(1)$	$\theta(1)$	$O(n)$	$O(n)$	$O(1)$	$O(1)$	$O(n)$

### 8.8.2.Operations on Singly Linked List

There are various operations which can be performed on singly linked list. A list of all such operations is given below.

#### Node Creation

```
struct node
{
int data;
struct node *next;
};
struct node *head, *ptr;
ptr = (struct node *)malloc(sizeof(struct node *));
```

#### Insertion

The insertion into a singly linked list can be performed at different positions. Based on the position of the new node being inserted, the insertion is categorized into the following categories.

SN	Operation	Description
1	Insertion at beginning	It involves inserting any element at the front of the list. We just need to a few link adjustments to make the new node as the head of the list.



2	Insertion at end of the list	It involves insertion at the last of the linked list. The new node can be inserted as the only node in the list or it can be inserted as the last one. Different logics are implemented in each scenario.
3	Insertion after specified node	It involves insertion after the specified node of the linked list. We need to skip the desired number of nodes in order to reach the node after which the new node will be inserted. .

### Deletion and Traversing

The Deletion of a node from a singly linked list can be performed at different positions. Based on the position of the node being deleted, the operation is categorized into the following categories.

SN	Operation	Description
1	Deletion at beginning	It involves deletion of a node from the beginning of the list. This is the simplest operation among all. It just need a few adjustments in the node pointers.
2	Deletion at the end of the list	It involves deleting the last node of the list. The list can either be empty or full. Different logic is implemented for the different scenarios.
3	Deletion after specified node	It involves deleting the node after the specified node in the list. we need to skip the desired number of nodes to reach the node after which the node will be deleted. This requires traversing through the list.
4	Traversing	In traversing, we simply visit each node of the list at least once in order to perform some specific operation on it, for example, printing data part of each node present in the list.

5	Searching	In searching, we match each element of the list with the given element. If the element is found on any of the location then location of that element is returned otherwise null is returned. .
---	-----------	--

### 8.8.3. Linked List in C: Menu Driven Program

```

#include<stdio.h>
#include<stdlib.h>
struct node
{
int data;
struct node *next;
};
struct node *head;

voidbegininsert ();
voidlastinsert ();
voidrandominsert();
voidbegin_delete();
voidlast_delete();
voidrandom_delete();
void display();
void search();
void main ()
{
int choice =0;
while(choice != 9)
{
printf("\n\n*****Main Menu*****\n");
printf("\nChoose one option from the following list ...\n");
printf("\n===== \n");

```

```
printf("\n1.Insert in begining\n2.Insert at last\n3.Insert at any random
location\n4.Delete from Beginning\n
5.Delete from last\n6.Delete node after specified location\n7.Search for an
element\n8.Show\n9.Exit\n");
printf("\nEnter your choice?\n");
scanf("\n%d",&choice);
switch(choice)
    {
case 1:
begininsert();
break;
case 2:
lastinsert();
break;
case 3:
randominsert();
break;
case 4:
begin_delete();
break;
case 5:
last_delete();
break;
case 6:
random_delete();
break;
case 7:
search();
break;
case 8:
display();
```

```
break;
case 9:
exit(0);
break;
default:
printf("Please enter valid choice..");
    }
}
}
voidbeginsert()
{
struct node *ptr;
int item;
ptr = (struct node *) malloc(sizeof(struct node *));
if(ptr == NULL)
    {
printf("\nOVERFLOW");
    }
else
    {
printf("\nEnter value\n");
scanf("%d",&item);
ptr->data = item;
ptr->next = head;
head = ptr;
printf("\nNode inserted");
    }
}
voidlastinsert()
{
```

```
struct node *ptr,*temp;
int item;
ptr = (struct node*)malloc(sizeof(struct node));
if(ptr == NULL)
    {
printf("\nOVERFLOW");
    }
else
    {
printf("\nEnter value?\n");
scanf("%d",&item);
ptr->data = item;
if(head == NULL)
    {
ptr -> next = NULL;
head = ptr;
printf("\nNode inserted");
    }
else
    {
temp = head;
while (temp -> next != NULL)
    {
temp = temp -> next;
    }
temp->next = ptr;
ptr->next = NULL;
printf("\nNode inserted");

    }
}
```

```

}
void randominsert()
{
    int i, loc, item;
    struct node *ptr, *temp;
    ptr = (struct node *) malloc (sizeof(struct node));
    if(ptr == NULL)
    {
        printf("\nOVERFLOW");
    }
    else
    {
        printf("\nEnter element value");
        scanf("%d",&item);
        ptr->data = item;
        printf("\nEnter the location after which you want to insert ");
        scanf("\n%d",&loc);
        temp=head;
        for(i=0;i<loc;i++)
        {
            temp = temp->next;
            if(temp == NULL)
            {
                printf("\ncan't insert\n");
                return;
            }
        }
        ptr ->next = temp ->next;
        temp ->next = ptr;
        printf("\nNode inserted");
    }
}

```

```

    }
}
voidbegin_delete()
{
struct node *ptr;
if(head == NULL)
    {
printf("\nList is empty\n");
    }
else
    {
ptr = head;
head = ptr->next;
free(ptr);
printf("\nNode deleted from the begining ...\n");
    }
}
voidlast_delete()
{
struct node *ptr,*ptr1;
if(head == NULL)
    {
printf("\nlist is empty");
    }
else if(head -> next == NULL)
    {
head = NULL;
free(head);
printf("\nOnly node of the list deleted ...\n");
    }
}

```

```

else
    {
ptr = head;
while(ptr->next != NULL)
    {
        ptr1 = ptr;
ptr = ptr ->next;
    }
    ptr1->next = NULL;
free(ptr);
printf("\nDeleted Node from the last ...\n");
    }
}
voidrandom_delete()
{
struct node *ptr,*ptr1;
intloc,i;
printf("\n Enter the location of the node after which you want to perform deletion
\n");
scanf("%d",&loc);
ptr=head;
for(i=0;i<loc;i++)
    {
        ptr1 = ptr;
ptr = ptr->next;

if(ptr == NULL)
    {
printf("\nCan't delete");
return;
    }
}

```



```
    }
    ptr1 ->next = ptr ->next;
free(ptr);
printf("\nDeleted node %d ",loc+1);
}
void search()
{
struct node *ptr;
int item,i=0,flag;
ptr = head;
if(ptr == NULL)
    {
printf("\nEmpty List\n");
    }
else
    {
printf("\nEnter item which you want to search?\n");
scanf("%d",&item);
while (ptr!=NULL)
    {
if(ptr->data == item)
        {
printf("item found at location %d ",i+1);
flag=0;
        }
else
        {
flag=1;
        }
i++;
ptr = ptr -> next;
```

```
    }
    if(flag==1)
    {
    printf("Item not found\n");
    }
}

void display()
{
struct node *ptr;
ptr = head;
if(ptr == NULL)
{
printf("Nothing to print");
}
else
{
printf("\nprinting values . . . .\n");
while (ptr!=NULL)
{
printf("\n%d",ptr->data);
ptr = ptr -> next;
}
}
}
```

Output:

\*\*\*\*\*Main Menu\*\*\*\*\*

Choose one option from the following list ...

=====

- 1.Insert in begining
- 2.Insert at last
- 3.Insert at any random location
- 4.Delete from Beginning
- 5.Delete from last
- 6.Delete node after specified location
- 7.Search for an element
- 8.Show
- 9.Exit

Enter your choice?

1

Enter value

1

Node inserted

\*\*\*\*\*Main Menu\*\*\*\*\*

Choose one option from the following list ...

=====

- 1.Insert in begining
- 2.Insert at last
- 3.Insert at any random location
- 4.Delete from Beginning
- 5.Delete from last
- 6.Delete node after specified location
- 7.Search for an element
- 8.Show
- 9.Exit

Enter your choice?

2

Enter value?

2

Node inserted

\*\*\*\*\*Main Menu\*\*\*\*\*

Choose one option from the following list ...

=====

- 1.Insert in begining
- 2.Insert at last
- 3.Insert at any random location
- 4.Delete from Beginning
- 5.Delete from last
- 6.Delete node after specified location
- 7.Search for an element
- 8.Show
- 9.Exit

Enter your choice?

3

Enter element value1

Enter the location after which you want to insert 1

Node inserted

\*\*\*\*\*Main Menu\*\*\*\*\*

Choose one option from the following list ...

=====

- 1.Insert in begining
- 2.Insert at last
- 3.Insert at any random location
- 4.Delete from Beginning
- 5.Delete from last
- 6.Delete node after specified location
- 7.Search for an element
- 8.Show
- 9.Exit

Enter your choice?

8

printing values . . . . .

1

2

1

\*\*\*\*\*Main Menu\*\*\*\*\*

Choose one option from the following list ...

=====

- 1.Insert in begining
- 2.Insert at last
- 3.Insert at any random location
- 4.Delete from Beginning
- 5.Delete from last
- 6.Delete node after specified location
- 7.Search for an element
- 8.Show
- 9.Exit

Enter your choice?

2

Enter value?

123

Node inserted

\*\*\*\*\*Main Menu\*\*\*\*\*

Choose one option from the following list ...

=====

- 1.Insert in begining
- 2.Insert at last
- 3.Insert at any random location
- 4.Delete from Beginning
- 5.Delete from last

6.Delete node after specified location

7.Search for an element

8.Show

9.Exit

Enter your choice?

1

Enter value

1234

Node inserted

\*\*\*\*\*Main Menu\*\*\*\*\*

Choose one option from the following list ...

=====

1.Insert in beginning

2.Insert at last

3.Insert at any random location

4.Delete from Beginning

5.Delete from last

6.Delete node after specified location

7.Search for an element

8.Show

9.Exit

Enter your choice?

4

Node deleted from the beginning ...

\*\*\*\*\*Main Menu\*\*\*\*\*

Choose one option from the following list ...

- 
- 
- 1.Insert in begining
  - 2.Insert at last
  - 3.Insert at any random location
  - 4.Delete from Beginning
  - 5.Delete from last
  - 6.Delete node after specified location
  - 7.Search for an element
  - 8.Show
  - 9.Exit

Enter your choice?

5

Deleted Node from the last ...

\*\*\*\*\*Main Menu\*\*\*\*\*

Choose one option from the following list ...

---

---

- 1.Insert in begining
- 2.Insert at last
- 3.Insert at any random location
- 4.Delete from Beginning
- 5.Delete from last
- 6.Delete node after specified location
- 7.Search for an element
- 8.Show

9.Exit

Enter your choice?

6

Enter the location of the node after which you want to perform deletion

1

Deleted node 2

\*\*\*\*\*Main Menu\*\*\*\*\*

Choose one option from the following list ...

- 
- 1.Insert in begining
  - 2.Insert at last
  - 3.Insert at any random location
  - 4.Delete from Beginning
  - 5.Delete from last
  - 6.Delete node after specified location
  - 7.Search for an element
  - 8.Show
  - 9.Exit

Enter your choice?

8

printing values . . . . .



1

1

\*\*\*\*\*Main Menu\*\*\*\*\*

Choose one option from the following list ...

=====

- 1.Insert in begining
- 2.Insert at last
- 3.Insert at any random location
- 4.Delete from Beginning
- 5.Delete from last
- 6.Delete node after specified location
- 7.Search for an element
- 8.Show
- 9.Exit

Enter your choice?

7

Enter item which you want to search?

1

item found at location 1

item found at location 2

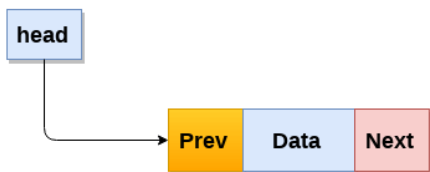
\*\*\*\*\*Main Menu\*\*\*\*\*

Choose one option from the following list ...

- 
- 1.Insert in begining
  - 2.Insert at last
  - 3.Insert at any random location
  - 4.Delete from Beginning
  - 5.Delete from last
  - 6.Delete node after specified location
  - 7.Search for an element
  - 8.Show
  - 9.Exit
- Enter your choice?
- 9

### 8.9.Doubly linked list

Doubly connected rundown is a mind boggling kind of connected rundown wherein a hub contains a pointer to the past just as the following hub in the arrangement. Subsequently, in a doubly connected rundown, a hub comprises of three sections: hub information, pointer to the following hub in arrangement (next pointer) , pointer to the past hub (past pointer). An example hub in a doubly connected rundown is appeared in the figure.



**Node**

A doubly linked list containing three nodes having numbers from 1 to 3 in their data part, is shown in the following image.



## Doubly Linked List

In C, structure of a node in doubly linked list can be given as :

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

The prev part of the first node and the next part of the last node will always contain null indicating end in each direction.

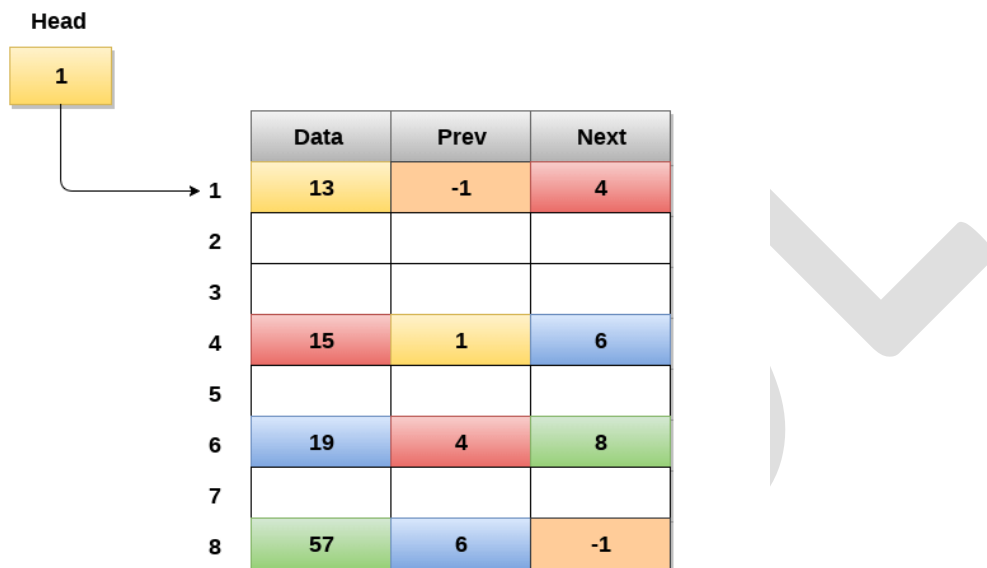
Doubly connected rundown is a mind boggling kind of connected rundown wherein a hub contains a pointer to the past just as the following hub in the arrangement. Subsequently, in a doubly connected rundown, a hub comprises of three sections: hub information, pointer to the following hub in arrangement (next pointer) , pointer to the past hub (past pointer). An example hub in a doubly connected rundown is appeared in the figure.

### 8.9.1.Memory Representation of a doubly linked list

Memory Representation of a doubly connected rundown is appeared in the accompanying picture. For the most part, doubly connected rundown burns-through more space for each hub and in this way, causes more sweeping fundamental activities, for example, inclusion and erasure. Notwithstanding, we can without much of a stretch control the components of the rundown since the rundown keeps up pointers in both the ways (forward and in reverse).

In the accompanying picture, the principal component of the rundown that is for example 13 put away at address 1. The head pointer focuses to the beginning location 1. Since this is the primary component being added to the rundown along these lines the prev of the rundown contains invalid. The following hub of the rundown lives at address 4 accordingly the first hub contains 4 in quite a while next pointer.

We can cross the rundown in this manner until we discover any hub containing invalid or - 1 in its next part.



### Memory Representation of a Doubly linked list

#### 8.9.2.Operations on doubly linked list

##### Node Creation

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

All the remaining operations regarding doubly linked list are described in the following table.

SN	Operation	Description
1	Insertion at beginning	Adding the node into the linked list at beginning.
2	Insertion at end	Adding the node into the linked list to the end.
3	Insertion after specified node	Adding the node into the linked list after the specified node.
4	Deletion at beginning	Removing the node from beginning of the list
5	Deletion at the end	Removing the node from end of the list.
6	Deletion of the node having given data	Removing the node which is present just after the node containing the given data.
7	Searching	Comparing each node data with the item to be searched and return the location of the item in the list if the item found else return null.
8	Traversing	Visiting each node of the list at least once in order to perform some specific operation like searching, sorting, display, etc.

### **8.9.3.Menu Driven Program in C to implement all the operations of doubly linked list**

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
```

```

struct node *prev;
struct node *next;
int data;
};
struct node *head;
void insertion_beginning();
void insertion_last();
void insertion_specified();
void deletion_beginning();
void deletion_last();
void deletion_specified();
void display();
void search();
void main ()
{
int choice =0;
while(choice != 9)
    {
printf("\n*****Main Menu*****\n");
printf("\nChoose one option from the following list ...\n");
printf("\n===== \n");
printf("\n1.Insert in beginning\n2.Insert at last\n3.Insert at any random
location\n4.Delete from Beginning\n
5.Delete from last\n6.Delete the node after the given
data\n7.Search\n8.Show\n9.Exit\n");
printf("\nEnter your choice?\n");
scanf("\n%d",&choice);
switch(choice)
    {
case 1:
insertion_beginning();

```

```
break;
case 2:
insertion_last();
break;
case 3:
insertion_specified();
break;
case 4:
deletion_beginning();
break;
case 5:
deletion_last();
break;
case 6:
deletion_specified();
break;
case 7:
search();
break;
case 8:
display();
break;
case 9:
exit(0);
break;
default:
printf("Please enter valid choice..");
    }
    }
}
void insertion_beginning()
```

```
{
struct node *ptr;
int item;
ptr = (struct node *)malloc(sizeof(struct node));
if(ptr == NULL)
{
printf("\nOVERFLOW");
}
else
{
printf("\nEnter Item value");
scanf("%d",&item);

if(head==NULL)
{
ptr->next = NULL;
ptr->prev=NULL;
ptr->data=item;
head=ptr;
}
else
{
ptr->data=item;
ptr->prev=NULL;
ptr->next = head;
head->prev=ptr;
head=ptr;
}
printf("\nNode inserted\n");
}
```



```
}  
void insertion_last()  
{  
    struct node *ptr,*temp;  
    int item;  
    ptr = (struct node *) malloc(sizeof(struct node));  
    if(ptr == NULL)  
        {  
        printf("\nOVERFLOW");  
        }  
    else  
        {  
        printf("\nEnter value");  
        scanf("%d",&item);  
        ptr->data=item;  
        if(head == NULL)  
            {  
            ptr->next = NULL;  
            ptr->prev = NULL;  
            head = ptr;  
            }  
        else  
            {  
            temp = head;  
            while(temp->next!=NULL)  
                {  
                temp = temp->next;  
                }  
            temp->next = ptr;  
            ptr ->prev=temp;  
            ptr->next = NULL;
```

```

    }

    }
printf("\nnode inserted\n");
    }
void insertion_specified()
{
struct node *ptr,*temp;
int item,loc,i;
ptr = (struct node *)malloc(sizeof(struct node));
if(ptr == NULL)
{
printf("\n OVERFLOW");
}
else
{
temp=head;
printf("Enter the location");
scanf("%d",&loc);
for(i=0;i<loc;i++)
{
temp = temp->next;
if(temp == NULL)
{
printf("\n There are less than %d elements", loc);
return;
}
}
printf("Enter value");
scanf("%d",&item);
ptr->data = item;

```

```
ptr->next = temp->next;
ptr ->prev = temp;
temp->next = ptr;
temp->next->prev=ptr;
printf("\nnode inserted\n");
}
}
voiddeletion_beginning()
{
struct node *ptr;
if(head == NULL)
{
printf("\n UNDERFLOW");
}
else if(head->next == NULL)
{
head = NULL;
free(head);
printf("\nnode deleted\n");
}
else
{
ptr = head;
head = head -> next;
head ->prev = NULL;
free(ptr);
printf("\nnode deleted\n");
}
}
voiddeletion_last()
{
```

```

struct node *ptr;
if(head == NULL)
{
printf("\n UNDERFLOW");
}
else if(head->next == NULL)
{
head = NULL;
free(head);
printf("\nnode deleted\n");
}
else
{
ptr = head;
if(ptr->next != NULL)
{
ptr = ptr -> next;
}
ptr ->prev -> next = NULL;
free(ptr);
printf("\nnode deleted\n");
}
}
void deletion_specified()
{
struct node *ptr, *temp;
int val;
printf("\n Enter the data after which the node is to be deleted : ");
scanf("%d", &val);
ptr = head;
while(ptr -> data != val)

```

```
ptr = ptr -> next;
if(ptr -> next == NULL)
{
printf("\nCan't delete\n");
}
else if(ptr -> next -> next == NULL)
{
ptr ->next = NULL;
}
else
{
temp = ptr -> next;
ptr -> next = temp -> next;
temp -> next ->prev = ptr;
free(temp);
printf("\nnode deleted\n");
}
}
void display()
{
struct node *ptr;
printf("\n printing values...\n");
ptr = head;
while(ptr != NULL)
{
printf("%d\n",ptr->data);
ptr=ptr->next;
}
}
void search()
{
```

```
struct node *ptr;
int item, i=0, flag;
ptr = head;
if(ptr == NULL)
    {
printf("\nEmpty List\n");
    }
else
    {
printf("\nEnter item which you want to search?\n");
scanf("%d",&item);
while (ptr!=NULL)
    {
if(ptr->data == item)
    {
printf("\nitem found at location %d ",i+1);
flag=0;
break;
    }
else
    {
flag=1;
    }
i++;
ptr = ptr -> next;
    }
if(flag==1)
    {
printf("\nItem not found\n");
    }
    }
```

}

## Output

\*\*\*\*\*Main Menu\*\*\*\*\*

Choose one option from the following list ...

=====

- 1.Insert in beginning
- 2.Insert at last
- 3.Insert at any random location
- 4.Delete from Beginning
- 5.Delete from last
- 6.Delete the node after the given data
- 7.Search
- 8.Show
- 9.Exit

Enter your choice?

8

printing values...

\*\*\*\*\*Main Menu\*\*\*\*\*

Choose one option from the following list ...

=====

- 1.Insert in beginning
- 2.Insert at last
- 3.Insert at any random location
- 4.Delete from Beginning
- 5.Delete from last
- 6.Delete the node after the given data
- 7.Search
- 8.Show
- 9.Exit

Enter your choice?

1

Enter Item value12

Node inserted

\*\*\*\*\*Main Menu\*\*\*\*\*

Choose one option from the following list ...

=====

- 1.Insert in begining
- 2.Insert at last
- 3.Insert at any random location
- 4.Delete from Beginning
- 5.Delete from last
- 6.Delete the node after the given data
- 7.Search
- 8.Show
- 9.Exit

Enter your choice?

1

Enter Item value123

Node inserted

\*\*\*\*\*Main Menu\*\*\*\*\*

Choose one option from the following list ...

=====

- 1.Insert in begining
- 2.Insert at last
- 3.Insert at any random location
- 4.Delete from Beginning
- 5.Delete from last
- 6.Delete the node after the given data
- 7.Search
- 8.Show
- 9.Exit

Enter your choice?



1

Enter Item value1234

Node inserted

\*\*\*\*\*Main Menu\*\*\*\*\*

Choose one option from the following list ...

---

- 1.Insert in begining
- 2.Insert at last
- 3.Insert at any random location
- 4.Delete from Beginning
- 5.Delete from last
- 6.Delete the node after the given data
- 7.Search
- 8.Show
- 9.Exit

Enter your choice?

8

printing values...

1234

123

12

\*\*\*\*\*Main Menu\*\*\*\*\*

Choose one option from the following list ...

---

- 1.Insert in begining
- 2.Insert at last
- 3.Insert at any random location
- 4.Delete from Beginning
- 5.Delete from last
- 6.Delete the node after the given data
- 7.Search

8.Show

9.Exit

Enter your choice?

2

Enter value89

node inserted

\*\*\*\*\*Main Menu\*\*\*\*\*

Choose one option from the following list ...

=====

- 1.Insert in begining
- 2.Insert at last
- 3.Insert at any random location
- 4.Delete from Beginning
- 5.Delete from last
- 6.Delete the node after the given data
- 7.Search
- 8.Show
- 9.Exit

Enter your choice?

3

Enter the location1

Enter value12345

node inserted

\*\*\*\*\*Main Menu\*\*\*\*\*

Choose one option from the following list ...

=====

- 1.Insert in begining
- 2.Insert at last
- 3.Insert at any random location
- 4.Delete from Beginning
- 5.Delete from last

- 6.Delete the node after the given data
- 7.Search
- 8.Show
- 9.Exit

Enter your choice?

8

printing values...

1234

123

12345

12

89

\*\*\*\*\*Main Menu\*\*\*\*\*

Choose one option from the following list ...

=====

- 1.Insert in beginning
- 2.Insert at last
- 3.Insert at any random location
- 4.Delete from Beginning
- 5.Delete from last
- 6.Delete the node after the given data
- 7.Search
- 8.Show
- 9.Exit

Enter your choice?

4

node deleted

\*\*\*\*\*Main Menu\*\*\*\*\*

Choose one option from the following list ...

=====

- 1.Insert in begining
- 2.Insert at last
- 3.Insert at any random location
- 4.Delete from Beginning
- 5.Delete from last
- 6.Delete the node after the given data
- 7.Search
- 8.Show
- 9.Exit

Enter your choice?

5

node deleted

\*\*\*\*\*Main Menu\*\*\*\*\*

Choose one option from the following list ...

=====

- 1.Insert in begining
- 2.Insert at last
- 3.Insert at any random location
- 4.Delete from Beginning
- 5.Delete from last
- 6.Delete the node after the given data
- 7.Search
- 8.Show
- 9.Exit

Enter your choice?

8

printing values...

123

12345

\*\*\*\*\*Main Menu\*\*\*\*\*

Choose one option from the following list ...

=====

- 1.Insert in beginning
- 2.Insert at last
- 3.Insert at any random location
- 4.Delete from Beginning
- 5.Delete from last
- 6.Delete the node after the given data
- 7.Search
- 8.Show
- 9.Exit

Enter your choice?

6

Enter the data after which the node is to be deleted : 123

\*\*\*\*\*Main Menu\*\*\*\*\*

Choose one option from the following list ...

=====

- 1.Insert in begining
- 2.Insert at last
- 3.Insert at any random location
- 4.Delete from Beginning
- 5.Delete from last
- 6.Delete the node after the given data
- 7.Search
- 8.Show
- 9.Exit

Enter your choice?

8

printing values...

123

\*\*\*\*\*Main Menu\*\*\*\*\*

Choose one option from the following list ...

=====

- 1.Insert in begining
- 2.Insert at last
- 3.Insert at any random location
- 4.Delete from Beginning
- 5.Delete from last
- 6.Delete the node after the given data
- 7.Search
- 8.Show
- 9.Exit

Enter your choice?

7

Enter item which you want to search?

123

item found at location 1

\*\*\*\*\*Main Menu\*\*\*\*\*

Choose one option from the following list ...

=====

- 1.Insert in begining
- 2.Insert at last
- 3.Insert at any random location
- 4.Delete from Beginning
- 5.Delete from last
- 6.Delete the node after the given data
- 7.Search

8.Show

9.Exit

Enter your choice?

6

Enter the data after which the node is to be deleted : 123

Can't delete

\*\*\*\*\*Main Menu\*\*\*\*\*

Choose one option from the following list ...

=====

1.Insert in begining

2.Insert at last

3.Insert at any random location

4.Delete from Beginning

5.Delete from last

6.Delete the node after the given data

7.Search

8.Show

9.Exit

Enter your choice?

9

Exited..

## Unit 4 : Chapter 9

### Operations on Linked List

#### 9.0.Objective

#### 9.1.Circular Singly Linked List

#### 9.2.Memory Representation of circular linked list

#### 9.3.Operations on Circular Singly linked list

#### 9.4.Deletion& Traversing

#### 9.5.Menu-driven program in C implementing all operations on circular singly linked list

#### 9.6.Circular Doubly Linked List

#### 9.7.Memory Management of Circular Doubly linked list

#### 9.8.Operations on circular doubly linked list

#### 9.9.C program to implement all the operations on circular doubly linked list

#### 9.0.Objective

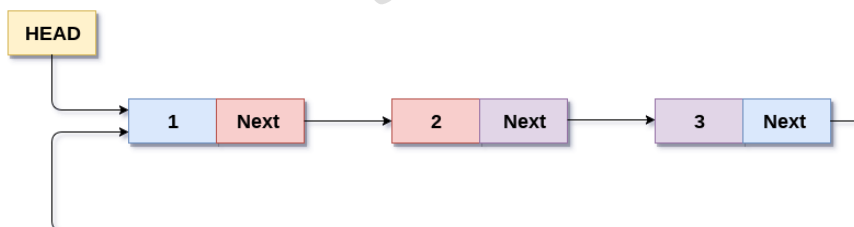
- To Understand the concept of Circular Singly Linked List
- Operations on Circular Singly linked list
- To Understand the concept of Circular Doubly Linked List
- Memory Management of Circular Doubly linked list

#### 9.1.Circular Singly Linked List

In a roundabout Singly connected rundown, the last hub of the rundown contains a pointer to the primary hub of the rundown. We can have roundabout separately connected rundown just as roundabout doubly connected rundown.

We navigate a roundabout separately connected rundown until we arrive at a similar hub where we began. The roundabout separately loved rundown has no start and no consummation. There is no invalid worth present in the following piece of any of the hubs.

The accompanying picture shows a round separately connected rundown.



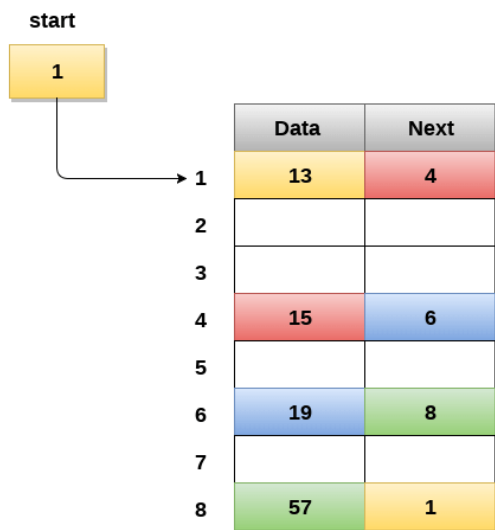
Circular Singly Linked List



Circular linked list are generally utilized in errand support in working frameworks. There are numerous models where round connected rundown are being utilized in software engineering including program riding where a record of pages visited in the past by the client, is kept up as roundabout connected records and can be gotten to again on tapping the past catch.

### 9.2.Memory Representation of circular linked list:

In the accompanying picture, memory portrayal of a round connected rundown containing signs of an understudy in 4 subjects. Nonetheless, the picture shows a brief look at how the round rundown is being put away in the memory. The beginning or top of the rundown is highlighting the component with the file 1 and containing 13 imprints in the information part and 4 in the following part. Which implies that it is connected with the hub that is being put away at fourth list of the rundown. Notwithstanding, because of the way that we are thinking about roundabout connected rundown in the memory in this manner the last hub of the rundown contains the location of the primary hub of the rundown.



### Memory Representation of a circular linked list

We can likewise have more than one number of connected rundown in the memory with the distinctive beginning pointers highlighting the diverse beginning hubs in the rundown. The last hub is distinguished by its next part which contains the location of

the beginning hub of the rundown. We should have the option to recognize the last hub of any connected rundown with the goal that we can discover the quantity of cycles which should be performed while navigating the rundown.

### **9.3.Operations on Circular Singly linked list:**

#### **Insertion**

<b>SNOperation</b>	<b>Description</b>
1 Insertion at beginning	Adding a node into circular singly linked list at the beginning.
2 Insertion at the end	Adding a node into circular singly linked list at the end.

### **9.4.Deletion& Traversing**

<b>SNOperation</b>	<b>Description</b>
1 Deletion at beginning	Removing the node from circular singly linked list at the beginning.
2 Deletion at the end	Removing the node from circular singly linked list at the end.
3 Searching	Compare each element of the node with the given item and return the location at which the item is present in the list otherwise return null.
4 Traversing	Visiting each element of the list at least once in order to perform some specific operation.

### **9.5.Menu-driven program in C implementing all operations on circular singly linked list**

```
#include<stdio.h>
#include<stdlib.h>
struct node
{
int data;
struct node *next;
};
struct node *head;
```

```

voidbegininsert ();
voidlastinsert ();
voidrandominsert();
voidbegin_delete();
voidlast_delete();
voidrandom_delete();
void display();
void search();
void main ()
{
int choice =0;
while(choice != 7)
{
printf("\n*****Main Menu*****\n");
printf("\nChoose one option from the following list ...\n");
printf("\n===== \n");
printf("\n1.Insert in begining\n2.Insert at last\n3.Delete from Beginning\n4.Delete
from last\n5.Search for an element\n6.Show\n7.Exit\n");
printf("\nEnter your choice?\n");
scanf("\n%d",&choice);
switch(choice)
{
case 1:
begininsert();
break;
case 2:
lastinsert();
break;
case 3:
begin_delete();
break;

```

```
case 4:
last_delete();
break;
case 5:
search();
break;
case 6:
display();
break;
case 7:
exit(0);
break;
default:
printf("Please enter valid choice..");
    }
}
}
voidbeginsert()
{
struct node *ptr,*temp;
int item;
ptr = (struct node *)malloc(sizeof(struct node));
if(ptr == NULL)
{
printf("\nOVERFLOW");
}
else
{
printf("\nEnter the node data?");
scanf("%d",&item);
ptr -> data = item;
```

```
if(head == NULL)
    {
head = ptr;
ptr -> next = head;
    }
else
    {
temp = head;
while(temp->next != head)
temp = temp->next;
ptr->next = head;
temp -> next = ptr;
head = ptr;
    }
printf("\nnode inserted\n");
    }
}
voidlastinsert()
{
struct node *ptr,*temp;
int item;
ptr = (struct node *)malloc(sizeof(struct node));
if(ptr == NULL)
    {
printf("\nOVERFLOW\n");
    }
else
    {
printf("\nEnter Data?");
scanf("%d",&item);
ptr->data = item;
```

```
if(head == NULL)
    {
head = ptr;
ptr -> next = head;
    }
else
    {
temp = head;
while(temp -> next != head)
    {
temp = temp -> next;
    }
temp -> next = ptr;
ptr -> next = head;
    }
printf("\nnode inserted\n");
}
}
voidbegin_delete()
{
struct node *ptr;
if(head == NULL)
    {
printf("\nUNDERFLOW");
    }
else if(head->next == head)
    {
head = NULL;
free(head);
printf("\nnode deleted\n");
    }
}
```

```

else
    { ptr = head;
while(ptr -> next != head)
ptr = ptr -> next;
ptr->next = head->next;
free(head);
head = ptr->next;
printf("\nnode deleted\n");

    }
}
voidlast_delete()
{
struct node *ptr, *preptr;
if(head==NULL)
    {
printf("\nUNDERFLOW");
    }
else if (head ->next == head)
    {
head = NULL;
free(head);
printf("\nnode deleted\n");
    }
else
    {
ptr = head;
while(ptr ->next != head)
    {
preptr=ptr;
ptr = ptr->next;

```

```
    }
preptr->next = ptr -> next;
free(ptr);
printf("\nnode deleted\n");
    }
}
void search()
{
struct node *ptr;
int item,i=0,flag=1;
ptr = head;
if(ptr == NULL)
    {
printf("\nEmpty List\n");
    }
else
    {
printf("\nEnter item which you want to search?\n");
scanf("%d",&item);
if(head ->data == item)
    {
printf("item found at location %d",i+1);
flag=0;
    }
else
    {
while (ptr->next != head)
    {
if(ptr->data == item)
    {
printf("item found at location %d ",i+1);
```



```
flag=0;
break;
    }
else
    {
flag=1;
    }
i++;
ptr = ptr -> next;
    }
    }
if(flag != 0)
    {
printf("Item not found\n");
    }
}
void display()
{
struct node *ptr;
ptr=head;
if(head == NULL)
    {
printf("\nnothing to print");
    }
else
    {
printf("\n printing values ... \n");
while(ptr -> next != head)
    {
printf("%d\n", ptr -> data);
```

```
ptr = ptr -> next;
    }
printf("%d\n", ptr -> data);
    }
}
```

Output:

\*\*\*\*\*Main Menu\*\*\*\*\*

Choose one option from the following list ...

=====

- 1.Insert in begining
- 2.Insert at last
- 3.Delete from Beginning
- 4.Delete from last
- 5.Search for an element
- 6.Show
- 7.Exit

Enter your choice?

1

Enter the node data?10

node inserted

\*\*\*\*\*Main Menu\*\*\*\*\*

Choose one option from the following list ...

=====

- 1.Insert in beginning
- 2.Insert at last
- 3.Delete from Beginning
- 4.Delete from last
- 5.Search for an element
- 6.Show
- 7.Exit

Enter your choice?

2

Enter Data?20

node inserted

\*\*\*\*\*Main Menu\*\*\*\*\*

Choose one option from the following list ...

=====

- 1.Insert in beginning
- 2.Insert at last
- 3.Delete from Beginning
- 4.Delete from last
- 5.Search for an element
- 6.Show
- 7.Exit

Enter your choice?

2

Enter Data?30

node inserted

\*\*\*\*\*Main Menu\*\*\*\*\*

Choose one option from the following list ...

=====

- 1.Insert in beginning
- 2.Insert at last
- 3.Delete from Beginning
- 4.Delete from last
- 5.Search for an element
- 6.Show
- 7.Exit

Enter your choice?

3

node deleted

\*\*\*\*\*Main Menu\*\*\*\*\*

Choose one option from the following list ...

=====

- 1.Insert in beginning
- 2.Insert at last
- 3.Delete from Beginning
- 4.Delete from last
- 5.Search for an element
- 6.Show
- 7.Exit

Enter your choice?

4

node deleted

\*\*\*\*\*Main Menu\*\*\*\*\*

Choose one option from the following list ...

=====

- 1.Insert in beginning
- 2.Insert at last
- 3.Delete from Beginning
- 4.Delete from last
- 5.Search for an element
- 6.Show
- 7.Exit

Enter your choice?

5

Enter item which you want to search?

20

item found at location 1

\*\*\*\*\*Main Menu\*\*\*\*\*

Choose one option from the following list ...

=====

- 1.Insert in beginning

- 2.Insert at last
- 3.Delete from Beginning
- 4.Delete from last
- 5.Search for an element
- 6.Show
- 7.Exit

Enter your choice?

6

printing values ...

20

\*\*\*\*\*Main Menu\*\*\*\*\*

Choose one option from the following list ...

- 
- 1.Insert in beginning
  - 2.Insert at last
  - 3.Delete from Beginning
  - 4.Delete from last
  - 5.Search for an element
  - 6.Show
  - 7.Exit

Enter your choice?

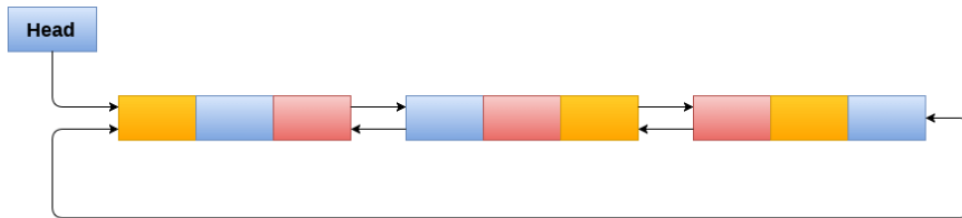
7

### **9.6.Circular Doubly Linked List**

Circular Doubly Linked List rundown is a more complexed kind of information structure in which a hub contain pointers to its past hub just as the following hub.

Round doubly connected rundown doesn't contain NULL in any of the hub. The last hub of the rundown contains the location of the main hub of the rundown. The main hub of the rundown additionally contain address of the last hub in its past pointer.

A circular doubly linked list is shown in the following figure.

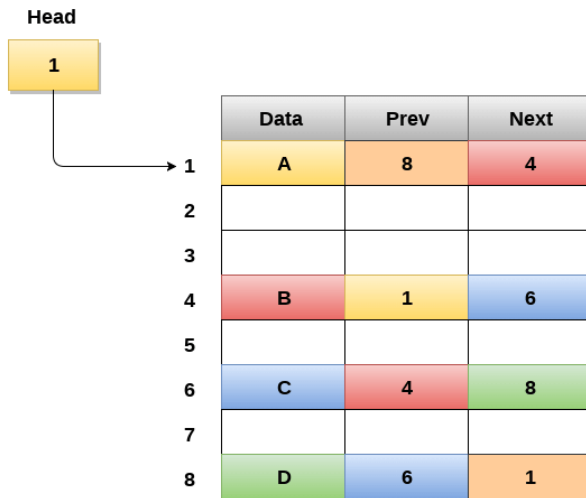


### Circular Doubly Linked List

Because of the way that a round doubly connected rundown contains three sections in its design hence, it requests more space per hub and more costly essential activities. Be that as it may, a round doubly connected rundown gives simple control of the pointers and the looking turns out to be twice as proficient.

#### 9.7.Memory Management of Circular Doubly linked list

The accompanying figure shows the manner by which the memory is designated for a round doubly connected rundown. The variable head contains the location of the principal component of the rundown for example 1 consequently the beginning hub of the rundown contains information An is put away at address 1. Since, every hub of the rundown should have three sections along these lines, the beginning hub of the rundown contains address of the last hub for example 8 and the following hub for example 4. The last hub of the rundown that is put away at address 8 and containing information as 6, contains address of the primary hub of the rundown as demonstrated in the picture for example 1. In roundabout doubly connected rundown, the last hub is recognized by the location of the main hub which is put away in the following piece of the last hub hence the hub which contains the location of the principal hub, is really the last hub of the rundown.



### Memory Representation of a Circular Doubly linked list

#### 9.8.Operations on circular doubly linked list :

There are different tasks which can be performed on round doubly connected rundown. The hub design of a roundabout doubly connected rundown is like doubly connected rundown. Be that as it may, the procedure on round doubly connected rundown is portrayed in the accompanying table.

SN	Operation	Description
1	Insertion at beginning	Adding a node in circular doubly linked list at the beginning.
2	Insertion at end	Adding a node in circular doubly linked list at the end.
3	Deletion at beginning	Removing a node in circular doubly linked list from beginning.
4	Deletion at end	Removing a node in circular doubly linked list at the end.

Traversing and searching in circular doubly linked list is similar to that in the circular singly linked list.

#### 9.9.C program to implement all the operations on circular doubly linked list

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

struct node
{
struct node *prev;

```

```

struct node *next;
int data;
};
struct node *head;
void insertion_beginning();
void insertion_last();
void deletion_beginning();
void deletion_last();
void display();
void search();
void main ()
{
int choice =0;
while(choice != 9)
{
printf("\n*****Main Menu*****\n");
printf("\nChoose one option from the following list ...\n");
printf("\n=====");
printf("\n1.Insert in Beginning\n2.Insert at last\n3.Delete from Beginning\n4.Delete
from last\n5.Search\n6.Show\n7.Exit\n");
printf("\nEnter your choice?\n");
scanf("\n%d",&choice);
switch(choice)
{
case 1:
insertion_beginning();
break;
case 2:
insertion_last();
break;
case 3:

```



```
deletion_beginning();
break;
case 4:
deletion_last();
break;
case 5:
search();
break;
case 6:
display();
break;
case 7:
exit(0);
break;
default:
printf("Please enter valid choice..");
    }
}
}
void insertion_beginning()
{
struct node *ptr,*temp;
int item;
ptr = (struct node *)malloc(sizeof(struct node));
if(ptr == NULL)
{
printf("\nOVERFLOW");
}
else
{
printf("\nEnter Item value");
```

```
scanf("%d",&item);
ptr->data=item;
if(head==NULL)
{
head = ptr;
ptr -> next = head;
ptr ->prev = head;
}
else
{
temp = head;
while(temp -> next != head)
{
temp = temp -> next;
}
temp -> next = ptr;
ptr ->prev = temp;
head ->prev = ptr;
ptr -> next = head;
head = ptr;
}
printf("\nNode inserted\n");
}
}
voidinsertion_last()
{
struct node *ptr,*temp;
int item;
ptr = (struct node *) malloc(sizeof(struct node));
if(ptr == NULL)
{
```

```
printf("\nOVERFLOW");
}
else
{
printf("\nEnter value");
scanf("%d",&item);
ptr->data=item;
if(head == NULL)
{
head = ptr;
ptr -> next = head;
ptr ->prev = head;
}
else
{
temp = head;
while(temp->next !=head)
{
temp = temp->next;
}
temp->next = ptr;
ptr ->prev=temp;
head ->prev = ptr;
ptr -> next = head;
}
}
printf("\nnode inserted\n");
}
voiddeletion_beginning()
{
struct node *temp;
```

```
if(head == NULL)
    {
printf("\n UNDERFLOW");
    }
else if(head->next == head)
    {
head = NULL;
free(head);
printf("\nnode deleted\n");
    }
else
    {
temp = head;
while(temp -> next != head)
    {
temp = temp -> next;
    }
temp -> next = head -> next;
head -> next ->prev = temp;
free(head);
head = temp -> next;
    }

}

voiddeletion_last()
{
struct node *ptr;
if(head == NULL)
    {
printf("\n UNDERFLOW");
    }
}
```

```
else if(head->next == head)
{
head = NULL;
free(head);
printf("\nnode deleted\n");
}
else
{
ptr = head;
if(ptr->next != head)
{
ptr = ptr -> next;
}
ptr ->prev -> next = head;
head ->prev = ptr ->prev;
free(ptr);
printf("\nnode deleted\n");
}
}
void display()
{
struct node *ptr;
ptr=head;
if(head == NULL)
{
printf("\nnothing to print");
}
else
{
printf("\n printing values ... \n");
```

```

while(ptr -> next != head)
    {
printf("%d\n", ptr -> data);
ptr = ptr -> next;
    }
printf("%d\n", ptr -> data);
    }
}

void search()
{
struct node *ptr;
int item,i=0,flag=1;
ptr = head;
if(ptr == NULL)
    {
printf("\nEmpty List\n");
    }
else
    {
printf("\nEnter item which you want to search?\n");
scanf("%d",&item);
if(head ->data == item)
    {
printf("item found at location %d",i+1);
flag=0;
    }
else
    {
while (ptr->next != head)
    {
if(ptr->data == item)

```

```
        {
printf("item found at location %d ",i+1);
flag=0;
break;
        }
else
        {
flag=1;
        }
i++;
ptr = ptr -> next;
        }
        }
if(flag != 0)
        {
printf("Item not found\n");
        }
        }
}
```

Output:

```
*****Main Menu*****
```

Choose one option from the following list ...

```
=====
```

- 1.Insert in Beginning
- 2.Insert at last
- 3.Delete from Beginning
- 4.Delete from last
- 5.Search
- 6.Show
- 7.Exit

Enter your choice?

1

Enter Item value123

Node inserted

\*\*\*\*\*Main Menu\*\*\*\*\*

Choose one option from the following list ...

=====

1.Insert in Beginning

2.Insert at last

3.Delete from Beginning

4.Delete from last

5.Search

6.Show

7.Exit

Enter your choice?

2

Enter value234

node inserted

\*\*\*\*\*Main Menu\*\*\*\*\*

Choose one option from the following list ...

=====

1.Insert in Beginning

2.Insert at last

3.Delete from Beginning

4.Delete from last

5.Search

6.Show

7.Exit

Enter your choice?

1

Enter Item value90

Node inserted



\*\*\*\*\*Main Menu\*\*\*\*\*

Choose one option from the following list ...

=====

- 1.Insert in Beginning
- 2.Insert at last
- 3.Delete from Beginning
- 4.Delete from last
- 5.Search
- 6.Show
- 7.Exit

Enter your choice?

2

Enter value80

node inserted

\*\*\*\*\*Main Menu\*\*\*\*\*

Choose one option from the following list ...

=====

- 1.Insert in Beginning
- 2.Insert at last
- 3.Delete from Beginning
- 4.Delete from last
- 5.Search
- 6.Show
- 7.Exit

Enter your choice?

3

\*\*\*\*\*Main Menu\*\*\*\*\*

Choose one option from the following list ...

=====

- 1.Insert in Beginning
- 2.Insert at last

3.Delete from Beginning

4.Delete from last

5.Search

6.Show

7.Exit

Enter your choice?

4

node deleted

\*\*\*\*\*Main Menu\*\*\*\*\*

Choose one option from the following list ...

=====

1.Insert in Beginning

2.Insert at last

3.Delete from Beginning

4.Delete from last

5.Search

6.Show

7.Exit

Enter your choice?

6

printing values ...

123

\*\*\*\*\*Main Menu\*\*\*\*\*

Choose one option from the following list ...

=====

1.Insert in Beginning

2.Insert at last

3.Delete from Beginning

4.Delete from last

5.Search

6.Show

7.Exit

Enter your choice?

5

Enter item which you want to search?

123

item found at location 1

\*\*\*\*\*Main Menu\*\*\*\*\*

Choose one option from the following list ...

=====

1.Insert in Beginning

2.Insert at last

3.Delete from Beginning

4.Delete from last

5.Search

6.Show

7.Exit

Enter your choice?

7

### Unit Structure

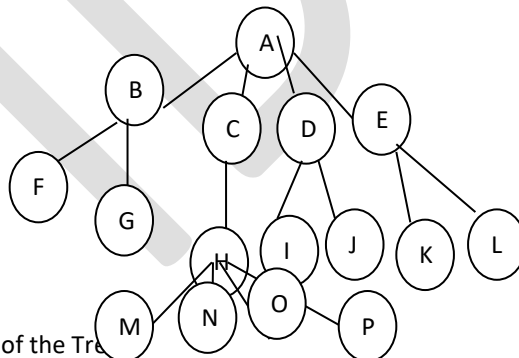
- 10.0 General Tree & Definition
- 10.1 Properties of the Tree
- 10.2 Binary Tree
  - 10.2.1 Binary Tree
  - 10.2.2 Strictly Binary Tree
  - 10.2.3 Almost complete Binary Tree
  - 10.2.4 Complete Binary Tree
- 10.3 Conversion of Tree to Binary tree
- 10.4 Construction of Binary Tree
- 10.5 Binary Search Tree
  - 10.5.1 Binary Search Tree
  - 10.5.2 Operations on Binary Search Tree
- 10.6 Tree Traversal
- 10.7 Construction of Binary Tree from the Traversal of the tree
- 10.8 Expression Tree
- 10.9 Threaded Binary tree
- 10.10 Huffman Tree
- 10.11 AVL Tree
- 10.12 Heap
  - 10.12.1 Heap Tree
  - 10.12.2 How to Construct Heap Tree
  - 10.12.3 Heap Sort

10.0 Trees: A Tree is a collection of nodes. The collection can be empty also. There is a specially designated node called Root Node. The Remaining nodes are partitioned in sub-trees like  $T_1, T_2, \dots, T_n$ . The tree will have unique path from root node to its children or leaf node. The tree does not have cycle.

Dig 1.

$H(O)=0; H(A)=3$

$D(P)=3$



$H(I)=2; H(E)=1;$

$D(H)=2; D(K)=2;$

#### 10.1 Properties of the Tree

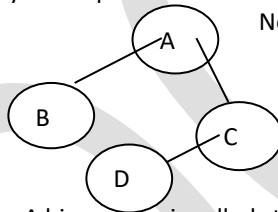
1. The root of each sub-tree is a child of a Root node and root R is the parent of each root of the sub-tree.
2. Every node except the root node has one parent node and all the parent nodes have at least one child. The parent can have one or more children except the leaf node.
3. The node which has no child called leaf node or terminal nodes. A tree can have one or more leaf nodes or terminal nodes.
4. The nodes with the same parent are called siblings.

5. The depth of a node n is the unique path from root node to the node n. The depth of the tree is the unique path from root node to the deepest leaf node.
6. The height of a node n is the unique path from the node n to the root node. The height of the tree is the unique path from deepest leaf node to the root node.
7. The height of the tree must be equal to the depth of the tree.  
Therefore  $H(T) = D(T)$ . Where H represents the Height, D represents the Depth and T represents the Tree.
8. All the leaves at height zero and the depth of the root is zero.
9. If there is a direct path from n1 to n2 then n1 is an ancestor of n2 and n2 is descendant of n1.
10. A tree should not have multiple paths from node n1 to node n2.

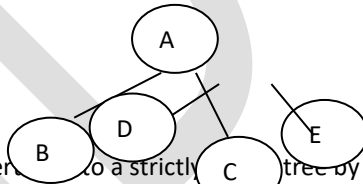
The above tree, height and depth of the tree: 3  
 Height of root node: 3 ; Depth of all the leaf nodes: 3  
 Depth of Root node: 0 ; Height of all the leaves : 0

10.2.1 Binary Tree: A tree is called binary tree if all the nodes in the tree has at the most two children. In a binary tree a parent can have either zero child or one child or two children.

Note: All the nodes have at the most two children.



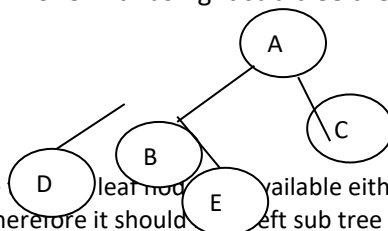
10.2.2 Strictly Binary Tree: A binary tree is called strictly binary tree if every non leaf node in a binary tree has non empty left sub-tree and right sub-tree.



Any binary tree can be converted to a strictly binary tree by adding left sub-tree or right sub-tree. In the above tree all the non leaf nodes have left sub-tree as well as right sub-tree also. A is a non leaf node has left and right sub-tree similarly C is a non leaf node has left and right sub-tree. Therefore the above tree is a strictly binary tree.

10.2.3 Almost complete Binary Tree: A binary tree is called almost complete binary tree if it satisfies the following two properties:

1. All the leaf nodes should be present either at level d or d-1.
2. Any non leaf node if it has right sub-tree then there should be left sub-tree also.

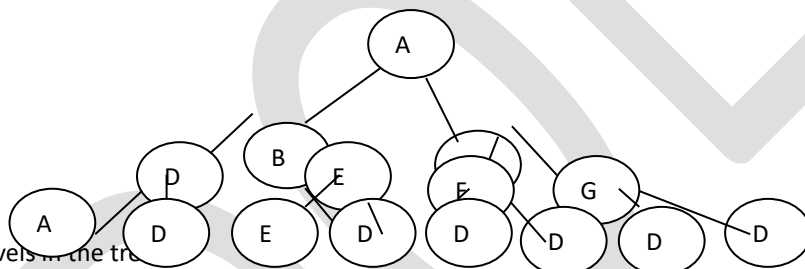


In this tree C, D, E are leaf nodes available either at depth d or d-1. B has right sub-tree therefore it should have left sub-tree as well. The node B has both the sub-trees hence it is an almost complete binary tree.

10.2.4 Complete Binary Tree: A binary tree is called complete binary tree if it satisfies the following properties:

1. Each node in tree must have at the most two children.
2. In a complete binary tree with n node at position  $i$ , the left child node must be present at position  $2i$  such that  $2i \leq N$  where  $N$  is the total number of nodes in a tree and the right child node must be present at position  $2i+1$  such that  $2i+1 \leq N$  where  $N$  is the total number of nodes in a tree.
3. The parent of left child node must be present at position  $i/2$  such that  $i/2 < N$  where  $N$  is the total number of nodes in a tree and the parent of right child node must be present at position  $(i-1)/2$  such that  $(i-1)/2 < N$  where  $N$  is the total number of nodes in a tree.
4. The complete binary tree at depth  $d$  is a strictly binary tree in which all the leaves are present at depth  $d$ .
5. The complete binary tree of level  $l$  contains  $2^l$  at any level of a complete binary tree.
6. The total number of nodes in a complete binary tree will be calculated  $\sum_{l=0}^d 2^l$ .

L0



L1  
L2  
L3  
L4

There are 4 levels in the tree.

Each level number of nodes in Complete Binary :  $2^l$

Total Number of nodes at level 0:  $2^0=1$

Total Number of nodes at level 1:  $2^1=2$

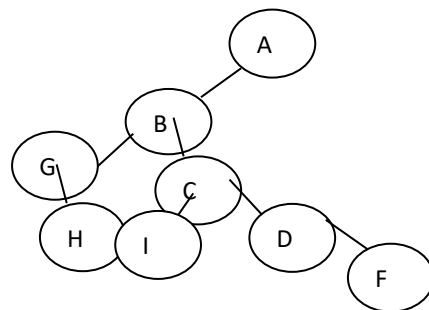
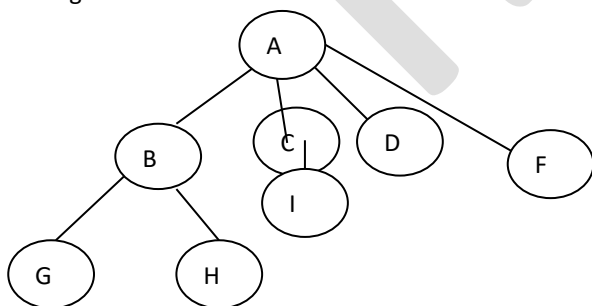
Total Number of nodes at level 2:  $2^2=4$

Total Number of nodes at level 3:  $2^3=8$

Total Number of nodes in the tree:  $L0 + L1 + L2 + L3 = 1+2+4+8 = 15$

10.3 Conversion of Tree to Binary tree: A tree can be converted to a binary tree with the help of following statement:

The left child of a node  $n$  will remain the left child of a node  $n$  and the right sibling will become the right child of a node  $n$ .



10.4 Construction of Binary Tree: The binary tree can be constructed with the help of following norms:

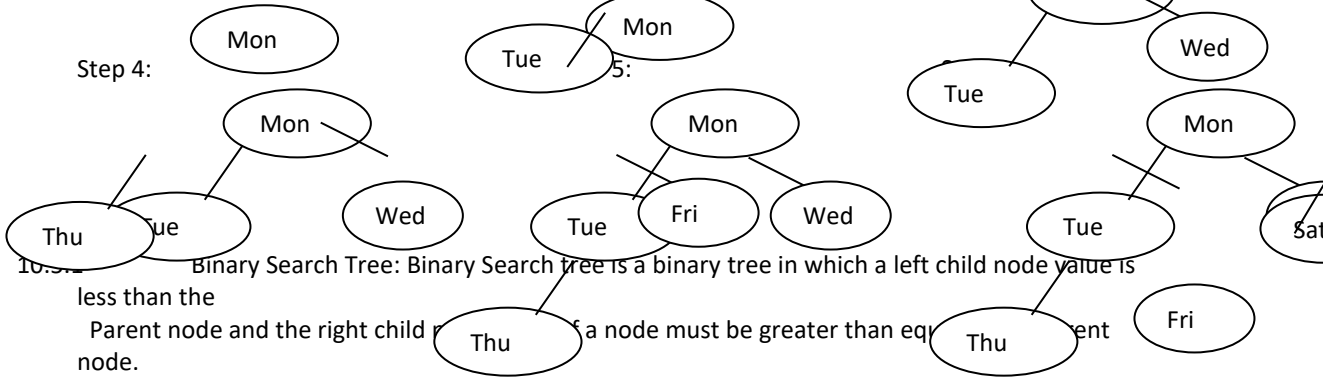
1. Start with the first value, become the root node.
2. The next value will be added at the last position of the tree.
3. Always first add the left child of the parent then the right child to the parent.

Problem: Construct Binary tree of Mon, Tue, Wed, Thu, Fri, Sat

Step 1:

Step 2:

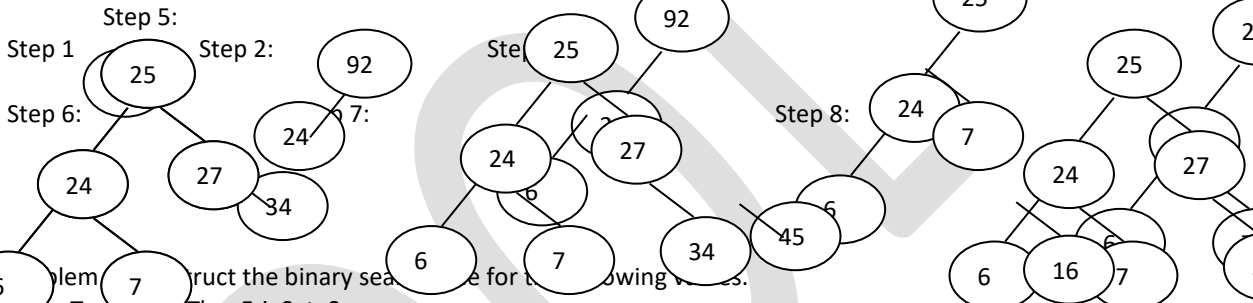
Step 3:



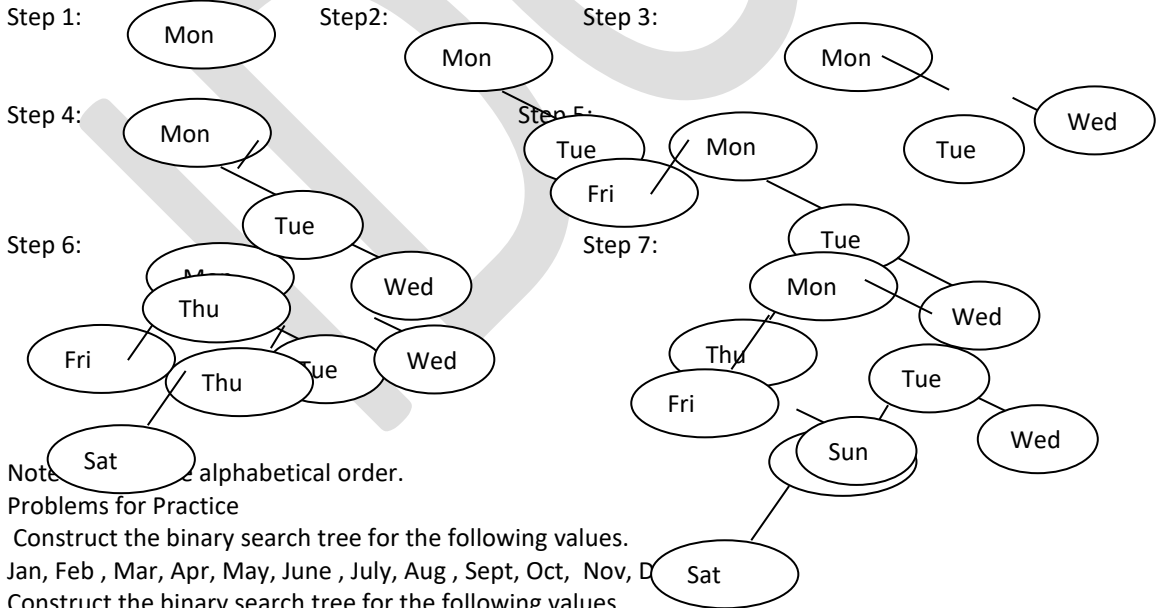
10.5.1 Binary Search Tree: Binary Search tree is a binary tree in which a left child node value is less than the parent node and the right child node value is greater than or equal to the parent node.

Problem 1: Construct the binary search tree for the following values.

25, 24, 6, 7, 11, 8, 27, 34, 45, 16,



Problem 2: Construct the binary search tree for the following values. Mon, Tue, Wed, Thu, Fri, Sat, Sun



Note: The values are in alphabetical order.

Problems for Practice

Construct the binary search tree for the following values.

Jan, Feb, Mar, Apr, May, June, July, Aug, Sept, Oct, Nov, Dec

Construct the binary search tree for the following values.

Dec, Nov, Oct, Sept, Aug, July, June, May, Apr, Mar, Feb, Jan

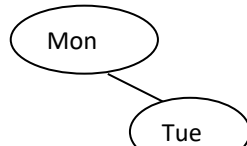
10.5.2 Operations on Binary Search Tree: The following operations can be performed on Binary Search.

How to insert a node in a binary search tree: If the new node value is less than the parent node value then the new node value will be the part of left sub-tree otherwise right sub-tree.

Inserted Value Thu



How to delete a node from a binary search tree: Any node can be deleted from the bellow binary search tree.



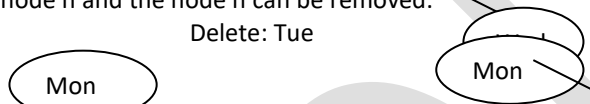
If the deleted node is a leaf node then delete the node directly from the binary tree.

Delete Thu :



If the node to be deleted has only one child then the child of node n will be connected with the parent of the node n and the node n can be removed.

Delete: Tue



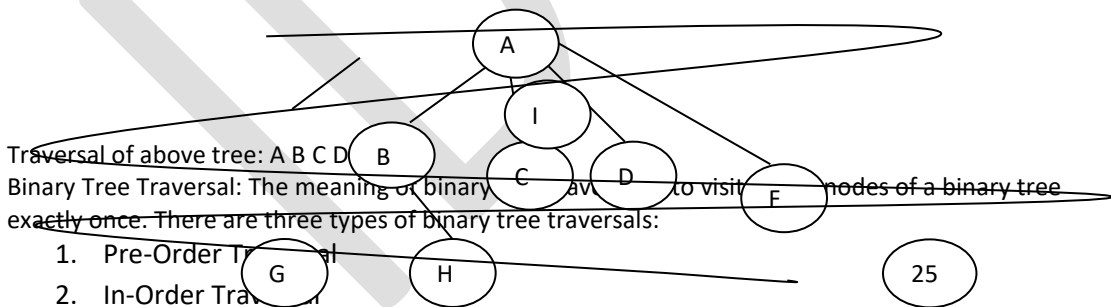
If a deleted node n has two children then either replace the highest value of left sub-tree from the deleted node value or replace the lowest value of right sub-tree from the deleted node value.

Delete

Replaced by lowest value from RST



10.6 Tree Traversal: Tree traversal is to visit all the nodes of a tree. Tree traversal is possible only for Simple tree and all binary trees. Tree traversal does not require any algorithm. In case of Binary tree the traverse will start from root node and covers all the nodes from left to right.

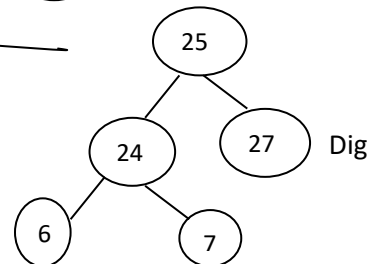


Traversal of above tree: A B C D

Binary Tree Traversal: The meaning of binary tree traversal is to visit all nodes of a binary tree exactly once. There are three types of binary tree traversals:

1. Pre-Order Traversal
2. In-Order Traversal
3. Post-Order Traversal

8



The Algorithms of Binary Tree Traversals:

Pre-Order Traversal

1. Visit the root node.
2. Traverse the left sub-tree in Pre-Order.
3. Traverse the Right sub-tree in Pre-Order.

Pre Order Traversal of tree in dig 8: 25, 24, 6, 7, 27



In-Order Traversal

1. Traverse the left sub-tree in In-Order.
2. Visit the root node.
3. Traverse the Right sub-tree in In-Order.

In Order Traversal of tree in dig 8: 6, 24, 7, 25, 27

Post-Order Traversal

1. Traverse the left sub-tree in In-Order.
2. Traverse the right sub-tree in In-Order.
3. Visit the root node.

Post Order Traversal of tree in dig 8: 6, 7, 24, 27, 25

Problems for Practice:

Problem 1: Construct and Traverse the binary tree in Pre-Order, In-Order and Post-Order .

92, 24 6,7,11,8,22,4,5,16,19,20,78

Problem 2: Construct and Traverse the binary tree in Pre-Order, In-Order and Post-Order .

Mon, Tue, Wed, Thu, Fri, Sat, Sun

Problem 3: Construct and Traverse the binary tree in Pre-Order, In-Order and Post-Order .

Jan, Feb, Mar, Apr, May, June, July, Aug, Sept, Oct, Nov, Dec.

Problem 4: Construct and Traverse the binary tree in Pre-Order, In-Order and Post-Order .

Dec, Nov, Oct, Sept, Aug, July, June, May, Apr, Mar, Feb, Jan

10.7 Construction of Binary Tree from the Traversal of the tree:

If any of the traversal pre-order or post-order is given the tree can be constructed using following method:

Pre-Order and In-Order

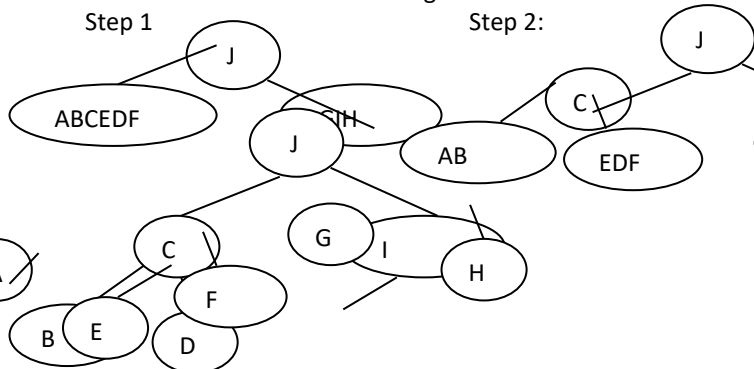
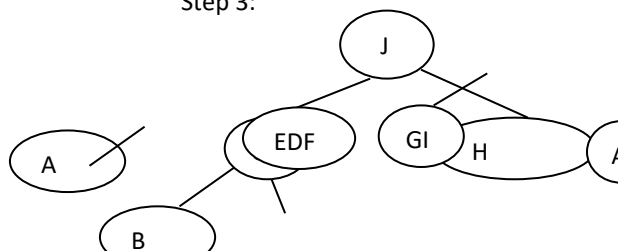
1. The first value of pre-order traversal becomes the root node value.
2. All the values lying left to the same value in in-order will be the part of left sub-tree and the values which are right to the in-order of the same value will be part of right sub-tree.

Problem: Construct a binary tree of a traversal of the tree for which the following is the in-order and Pre-order Traversal.

In-Order: A B C E D F J G I H

Pre-Order: J C B A D E F I G H

Step 3:



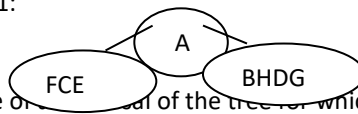
Step 1

Step 2:

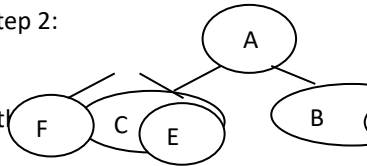
Problem: Construct a binary tree of a traversal of the tree for which the following is the in-order and Post-order Traversal.

Post-Order: F E C H G D B A

Step 1:



Step 2:



In-Order: F C E A B H D G

Step 3:

Practice Problem: Construct a binary tree of a traversal of the tree for which the following is the in-order and Post-order Traversal.

Pre-Order: A B E C

Post-Order: D C E F G

10.8 Expression Tree: All the algebraic equations can be represented in the form of binary tree. An algebraic equation is represented in the form of infix notation in which the operator is coming between the operands. For example  $A+B$  where  $A$  and  $B$  are the operands and  $+$  is an operator which is coming between operands  $A$  and  $B$ . While representing an algebraic equation in the form of binary tree, the entire operator will be either root node or internal nodes and all the operands will be the leaf nodes.

Expression tree satisfy following properties:

1. It represents an algebraic equation in the form of binary tree.
2. All the operators will be either root node or an internal node and all the operands will always be the leaf nodes.
3. Expression is an infix notation of a binary tree.
4. If you traverse the expression tree in an in-order then it is converted in to an algebraic equation.

Algebraic equation  $A+B$  can be represented as

Problem:

How to construct an expression Tree:

Step1: Convert the algebraic equation either in to pre fix notation or postfix notation.

Step 2: By Prefix / Postfix notation identify the root.

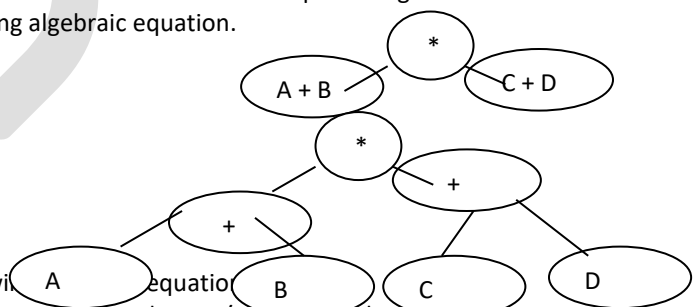
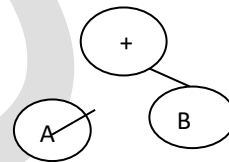
Step 3: All the values which comes left to prefix value in infix notation will be part of left sub tree and the values come to right to the prefix value in infix notation will be the part of right sub tree.

Construct an expression tree of the following algebraic equation.

$A + B * C + D$

Infix Notation:  $A + B * C + D$

Prefix Notation:  $*+AB+CD$



Construct an expression Tree for the following algebraic equation

A.  $A+B*C$

B.  $(A+B)*(C-D)$

E.  $A*C + D/F$

F.  $X*Y-Z*U$

Construct an expression Tree for the following algebraic equation:

A.  $2+P+Q*C$

B.  $(A-B)/(C+D)$

E.  $A*C * D/F$

F.  $P/Q+Z-U$

Convert the expression in prefix notation and post fix notation and then construct the tree.

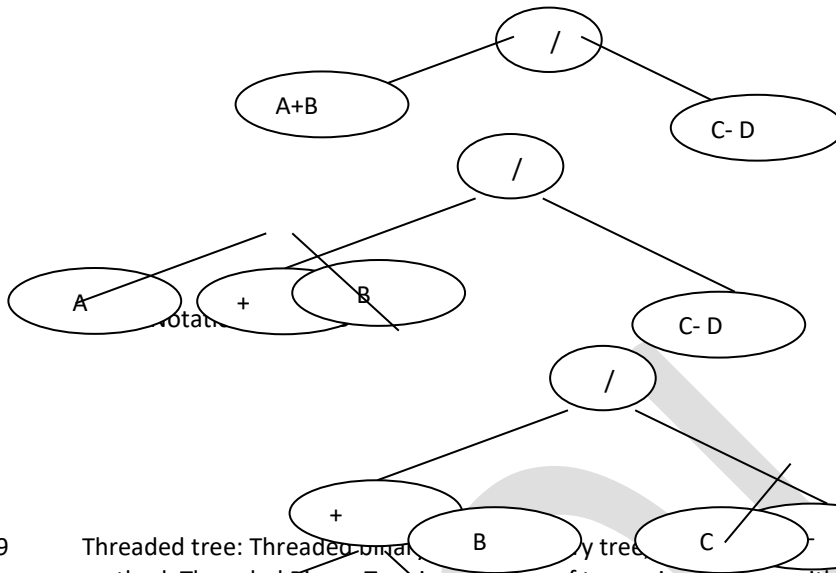
A.  $A/B*C$

B.  $(A+B*C)/(E-D)$

E.  $A*C * D/F$

F.  $X/Y-Z*U$

Example: Infix Notation:  $(A+B)/(C-D)$



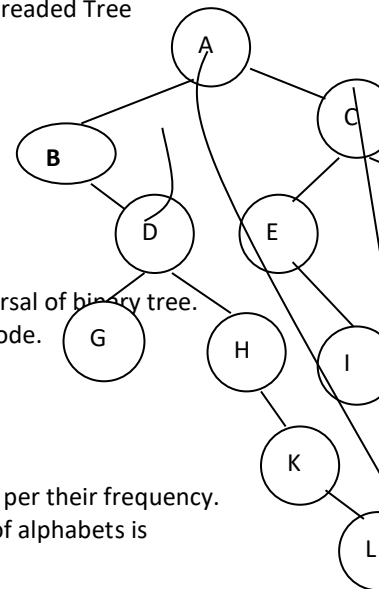
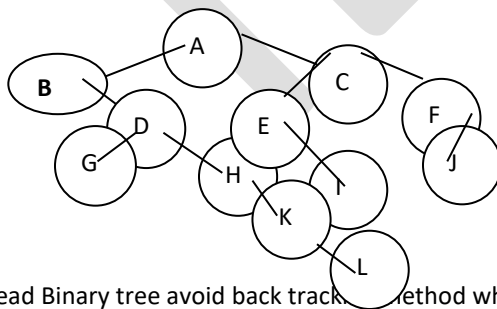
10.9

Threaded tree: Threaded Binary Tree is a solution of traversing the tree without using back tracking method. If tree is constructed and traverse the same tree in in-order then to visit the next node it passes through a previous node value. Threaded binary tree is a solution of back tracking method in which while traversing the tree in in-order, it will not pass through the previous node value.

Process: Traverse the binary tree in in-order and mark all the successor node of all the leaf node of in-order traversal tree of the same tree. These leaf nodes will be connected to their successor node value by a thread which is redirecting the leaf node value to connect with successor node and avoid going to previous node value.

In-order: B G D H K L A E I C J  
 F  
 Leaf Nodes: G L I J  
 Successor Leaf nodes : G-> D; L->A I-> C J->F

Therefore G will be connected with D by a threaded. Similarly L by A, I by C and J by F.  
 Binary Threaded Tree



Thread Binary tree avoid back tracking method which is a drawback of traversal of binary tree. Threads are proving an alternate path to move from one node to another node.

10.10

Huffman tree: Huffman Tree is a representation of alphabets or numbers as per their frequency. Many times we see an alphabet is repeated multiple times or combination of alphabets is

repeated in the same combination. Huffman coding is a lossless data compression technique. In this all the alphabets are assigned variable length unique code by forming the tree. This tree is called Huffman Tree.

How to construct a Huffman Tree:

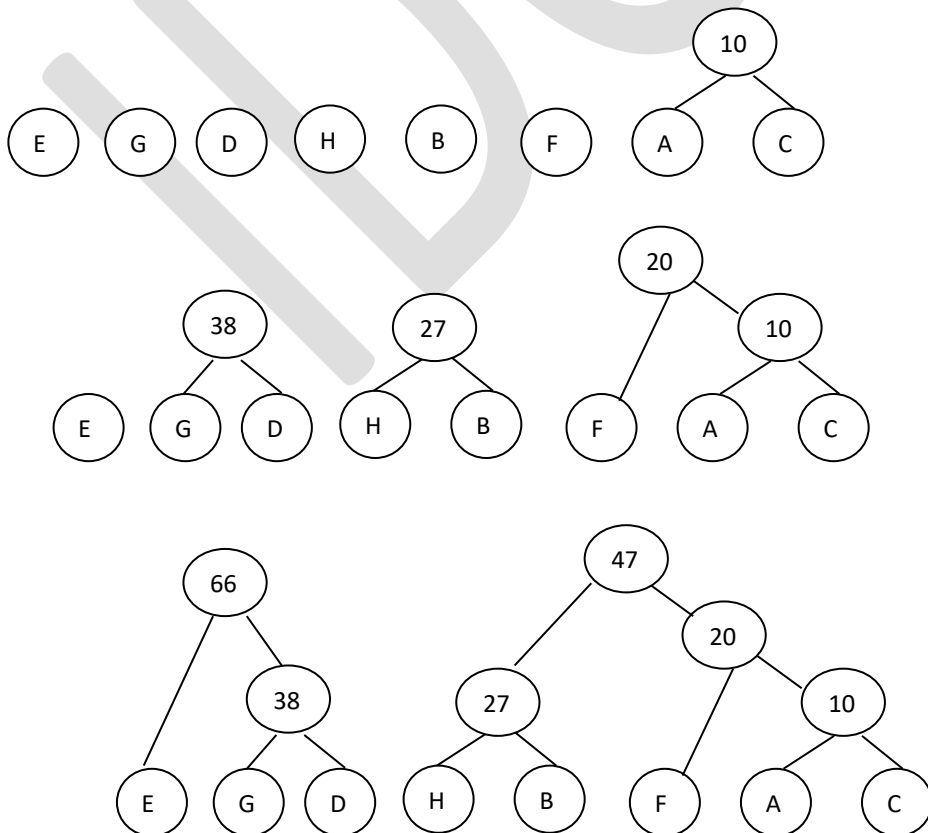
1. Arrange all the alphabets in ascending or descending order and place them a leaf node of a tree.
2. Add two lowest frequency nodes and form a new node which will be one level up. The value of this new node will be the addition of both the nodes which are added.
3. Keep adding two lowest frequency nodes which are not yet added.
4. Repeat step number three un till a root is formed.
5. Assign 0 to all the left branch of the tree and 1 to all the right branch of the tree.
6. A code of each alphabet is identified by reading the branch values from top to bottom which is a unique path from root to the leaf node.

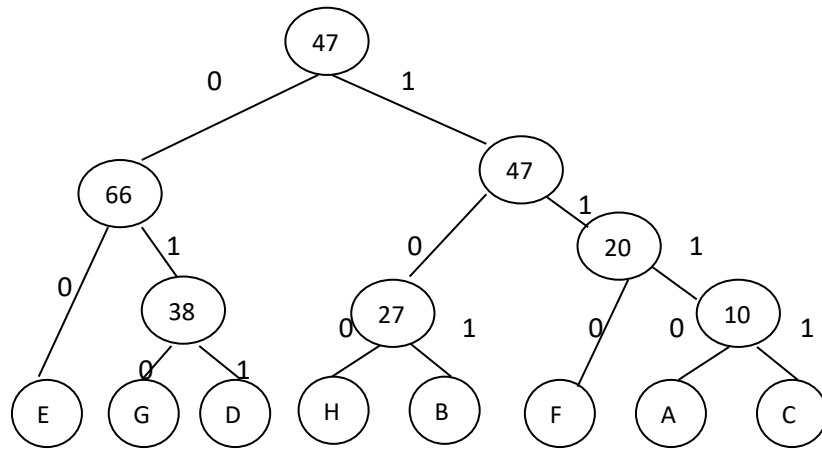
Problem: Construct Huffman Tree for the following data values:

Symbol	A	B	C	D	E	F	G	H
Frequency	6	13	4	18	28	10	20	14

Arrange all the alphabets in descending order as per their frequency:

E      G      D      H      B      F      A      C  
 28    20    18    14    13    10    6      4





Huffman Code:

E: 00	D: 011	B: 101	A: 1110
G: 010	H: 100	F: 110	C: 1111

- 10.11 AVL tree: AVL Tree is identified by Adelson Velskii and Landis. That is why it is called as AVL Tree. AVL tree is a solution of a binary search tree which is constructed for either all the values of increasing order or decreasing order. It has following problems:
1. When a binary search tree is constructed of ascending or descending values it will go one side either left hand side or right hand side.
  2. The tree will not look like a tree at the same time and it will not be balanced.

Therefore AVL Tree is a solution to overcome from both the above problems.

AVL Tree is a binary search tree in which the difference between the height of left sub tree and the height of right sub tree must be less than equal to one. The condition can be represented by following equation.

$$H_L \sim H_R \leq 1$$

$H_L$  is Height of Left sub tree

$H_R$  is Height of Right sub-tree.

Since the tree will be balanced therefore it is called as Height Balance tree or Balanced Binary Search Tree.

AVL Tree must satisfy the following properties:

1. The difference between the height of left sub tree and the height of right sub tree must be less than equal to one.
2. A new node can be added to the AVL tree only if the tree is balanced.
3. All the nodes must have balance factor either 0 or 1.

How to Balance Binary search tree: Use rotation by finding the unbalancing is due to which sub tree.

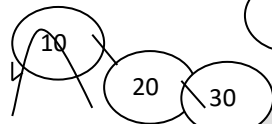
1. If the tree is unbalance due to the height of left sub tree then rotate the tree in to right hand side. Dig

- If the tree is unbalance due to the height of right sub tree then rotate the tree in to left hand side. Dig
- If the tree is unbalance due to the right sub tree of left sub tree then rotate the tree first left then right hand side. [In this case there will be multiple rotations to balance the tree.]
- If the tree is unbalance due to the left sub tree of right sub tree then rotate the tree first right then left hand side. [In this case there will be multiple rotations to balance the tree.]
- Dig 14.1

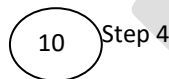
Construct AVL Tree for the following values:

10 20 30 40

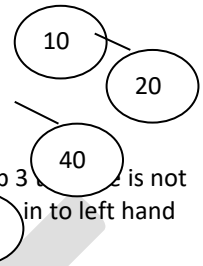
Step 3:



Step 1



Step 2:



Note: Step 1 and 2 the tree is balance but step 3 the tree is not balance. In step 3 the tree is not balance because of unbalancing due to right hand side therefore rotate the tree in to left hand side.

The AVL Tree is constructed at step 4.

Problem: Construct the AVL Tree for the following values.

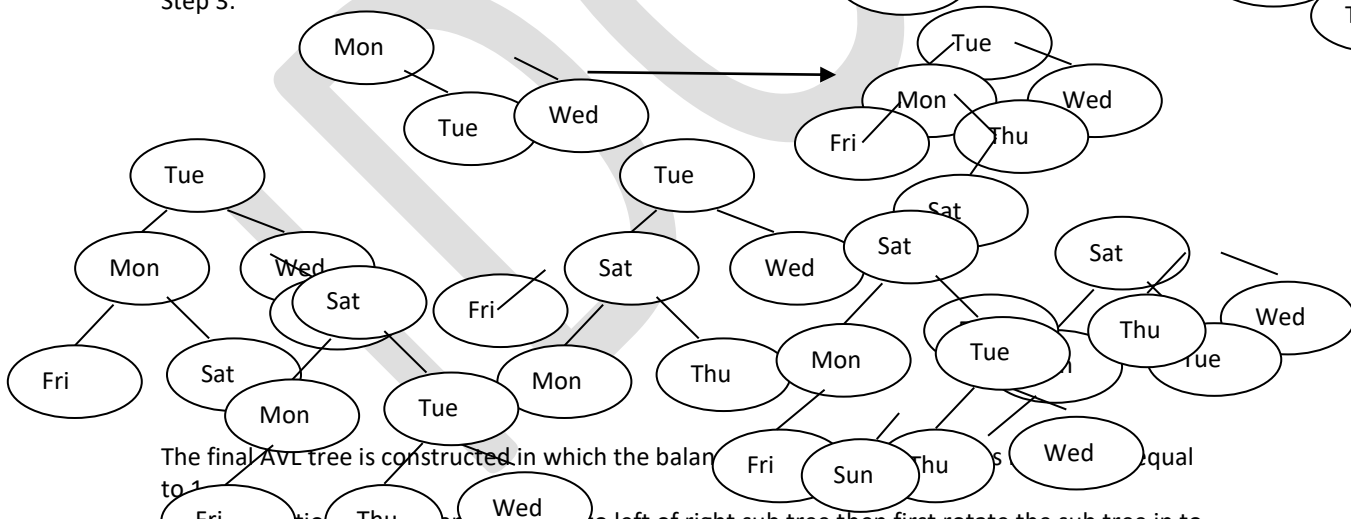
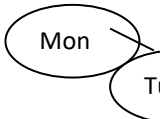
Mon, Tue, Wed, Thu, Fri, Sat, Sun

Step 3:

Step 1:



Step 2:



The final AVL tree is constructed in which the balance factor is equal to 1.

If unbalancing is due to left of right sub tree then first rotate the sub tree in to right side than rotate the sub tree in to left side and if unbalancing is due to right of left sub tree then first rotate the sub tree in to left side than rotate the sub tree in to right.

Practice Problems:

- Construct and AVL Tree for the following values.  
92, 24 6,7,11,8,22,4,5,16,19,20,78
- Construct and AVL Tree for the following values.  
Sun, Mon, Tue, Wed, Thu, Fri, Sat
- Construct and AVL Tree for the following values.  
Jan, Feb , Mar, Apr, May, June , July, Aug , Sept, Oct, Nov, Dec.

- Construct and AVL Tree for the following values.

Dec, Nov, Oct, Sept, Aug, July, June, May, Apr, Mar, Feb, Jan

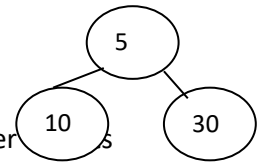
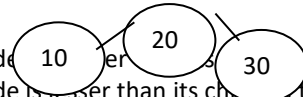
10.12.1 Heap: Heap Tree is binary tree in which parent node is either greater than its children or lesser than its children. There are two types of Heap Tree.

Max Heap

Min Heap

If the parent node is greater than its children then it is called as Max Heap.

If the parent node is lesser than its children then it is called as Min Heap.



1. Max Heap: Max Heap Tree is binary tree in which parent node is greater than its children. Using max heap tree the values will be coming in descending order if the same heap tree is used for sorting.
2. Min Heap: Min Heap Tree is binary tree in which parent node is lesser than its children. Using min heap tree the values will be coming in ascending order if the same heap tree is used for sorting.

10.12.2 How to construct a Heap Tree

Steps to construct Max Heap: Follow the following steps to construct max heap.

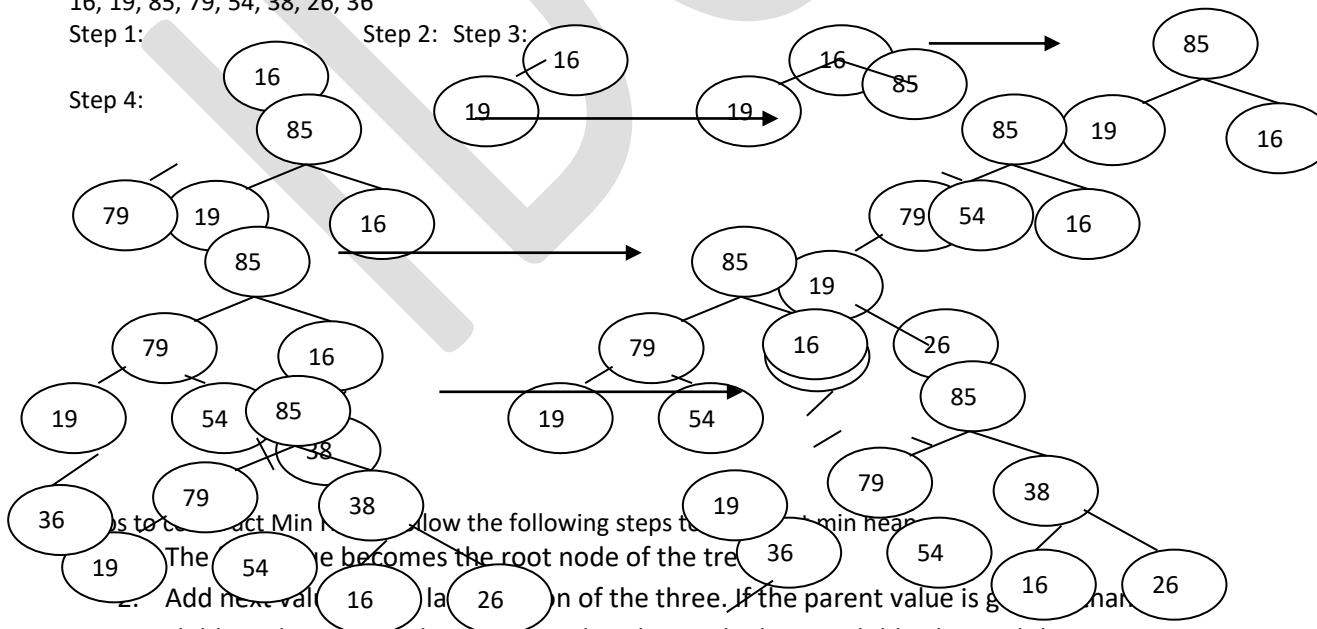
1. The first value becomes the root node of the tree.
2. Add next value at the last position of the tree. If the parent value is lesser than its children the replace the parent node value to the greater child value and the tree is converted in to heap.
3. Repeat the step number 2 until all the values are added in to the heap tree.

Construct Max Heap for the following values:

16, 19, 85, 79, 54, 38, 26, 36

Step 1:

Step 2: Step 3:

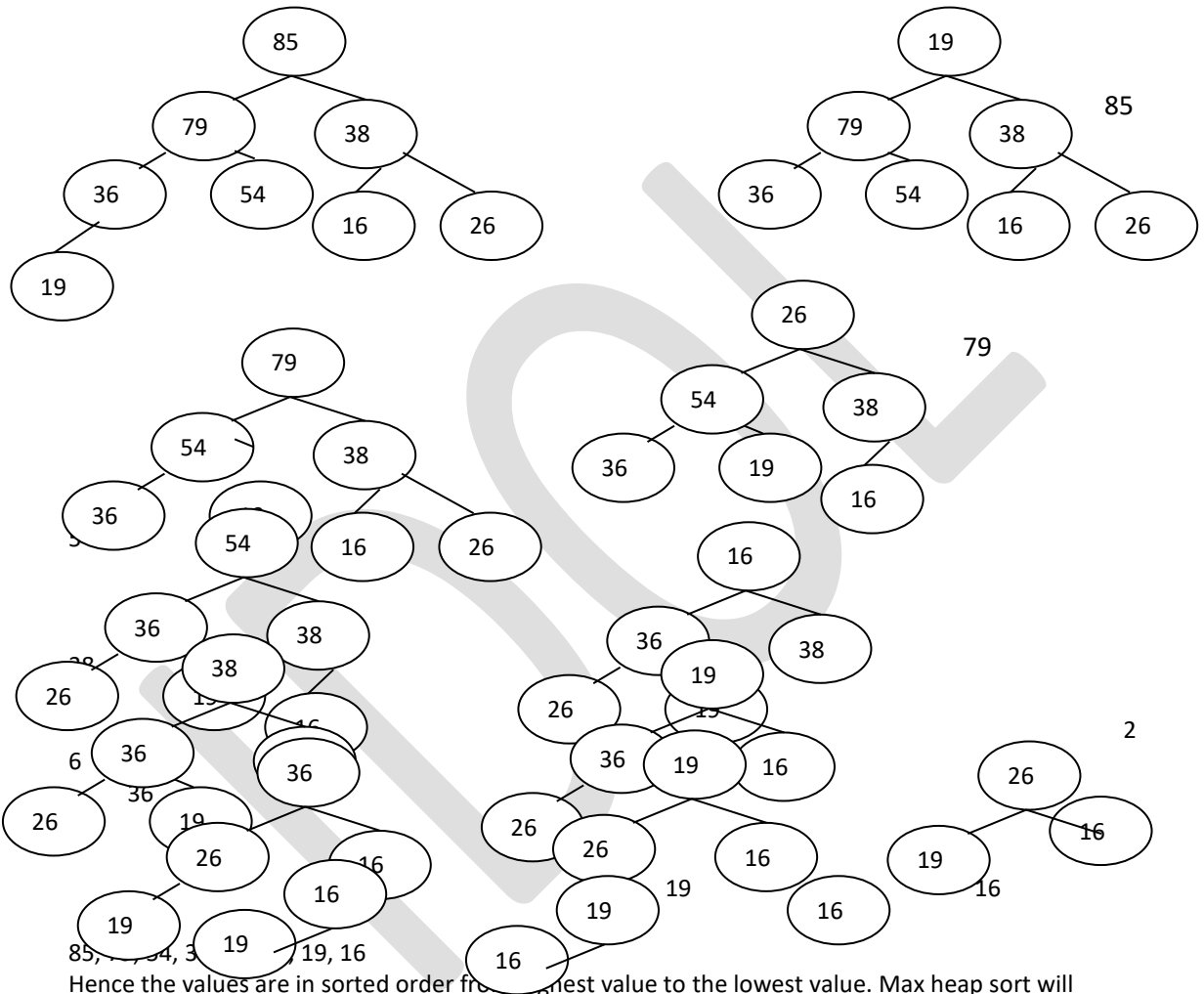


1. The first value becomes the root node of the tree.
2. Add next value at the last position of the tree. If the parent value is greater than its children the replace the parent node value to the lowest child value and the tree is converted in to heap.
3. Repeat the step number 2 until all the values are added in to the heap tree.

The way max heap is constructed; min heap can also be constructed similarly.

10.12.3 Steps to apply Heap Sort:

1. Convert the tree in to heap.
2. Take out the value from the root.
3. Replace last node value from the root node value.
4. Go to step 1 until number of nodes are greater than one.



Hence the values are in sorted order from highest value to the lowest value. Max heap sort will arrange the values in descending order while min heap arrange the values in ascending order. Reheap up and Reheap down: The concept of reheap up and reheap down is how the values are shifting upward direction or downward direction while constructing the heap tree. If the values are shifting upward direction then it is call Reheapup and if the values are shifting downward direction then it is call reheap down.

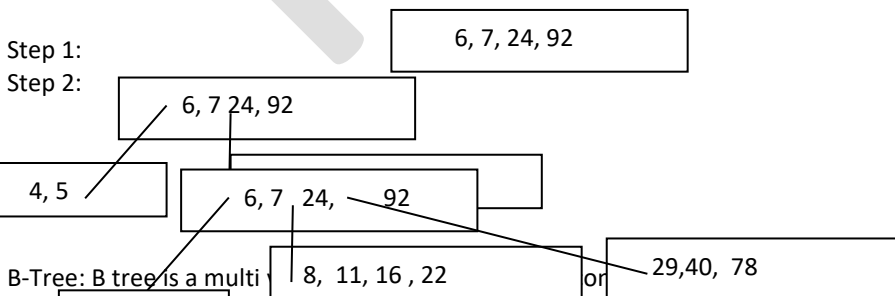
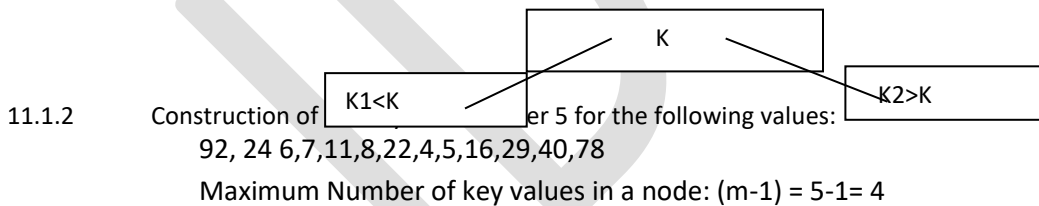


Unit 5: Chapter 11  
M- Way Tree

- 11.1 M- Way Tree
  - 11.1.1 M Way-Tree
  - 11.1.2 Construction of M-Way Tree
- 11.2 B-Tree
  - 11.2.1 B- Tree
  - 11.2.2 Construction of B-Tree
- 11.3 B\* Tree
  - 11.3.1 B\* Tree
  - 11.3.2 Construct of B\*- Tree
- 11.4 Deletion from B-Tree/ B\* Tree
- 11.5 Similarities and Difference from B-Tree and B\* Tree
  - 11.5.1 Similarities in B-Tree and B\* Tree
  - 11.5.2 Difference from B-Tree and B\* Tree
- 11.6 Practice Problem based on B-Tree and B\* Tree

11.1.1 M-Way Tree: M-way tree is a multi valued tree in which a node consists of multiple values and satisfy the following properties.

1. M-Way tree of order m will have maximum m pointers and m-1 key values in a node. All the keys must be placed in an increasing order or ascending order.
2. If a node has m-1 key values then the node is full. If a node is full then the new key value will be either left child value of the parent node if it is lesser then its parent node value or right pointer child value if it is greater than its parent value.
3. A new key value will be added at the leaf node value.
4. The leaves may not be at the same level.



11.2.1 B-Tree: B tree is a multi valued tree in which a node consists of multiple values and satisfy the following properties.

1. B tree of order m will have maximum m pointers and m-1 key values in a node. All the keys must be placed in an increasing order or ascending order.
2. If a node has m-1 key values then the node is full. If a node is full then split the node. Select the median value which becomes the parent value. The values

coming left of median value will become the left child of parent value and the values coming to right to the median value will become the right child of the median value. The new key value will be inserted either left child of the parent node (if it is lesser than its parent node value) or right child value of the parent node ( if it is greater than its parent value).

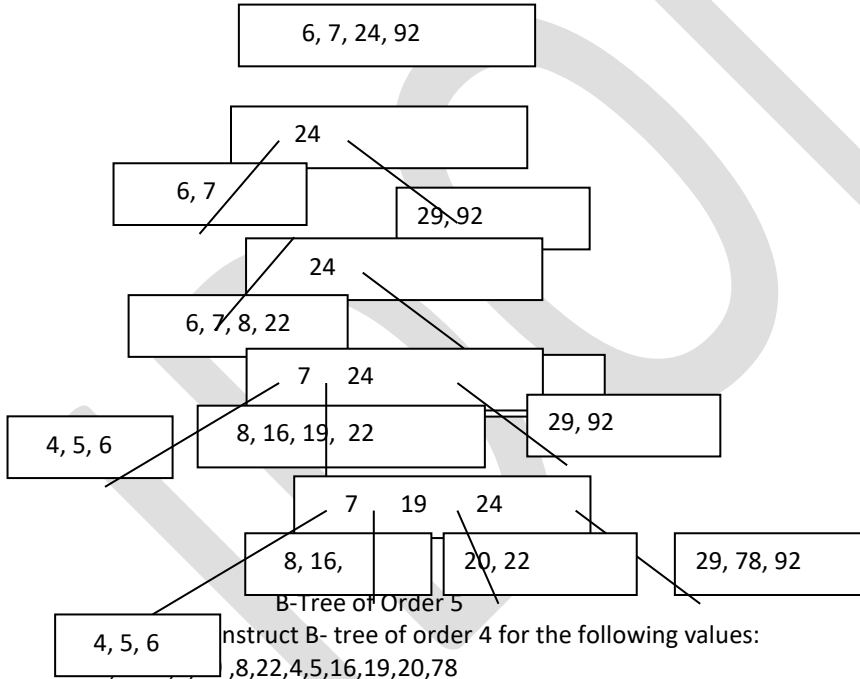
3. A new key value will be added at the leaf node value.
4. The leaves must be at the same level.
5. The height of B-tree will always be lesser than M-Way Tree.
6. The new value will be inserted in the leaf node.

11.2.2

Construction of B Tree: Construct B- tree of order 5 for the following values:

92, 24 6,7,29 ,8,22,4,5,16,19,20,78

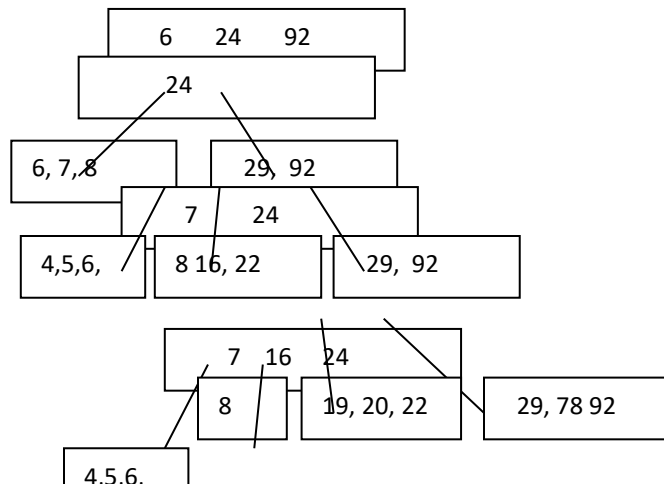
Maximum Number of Key values in a node=  $m-1=5-1=4$



Construct B- tree of order 4 for the following values:

,8,22,4,5,16,19,20,78

Maximum Number of Key values in a node=  $m-1=4-1=3$



## B Tree of Order 4

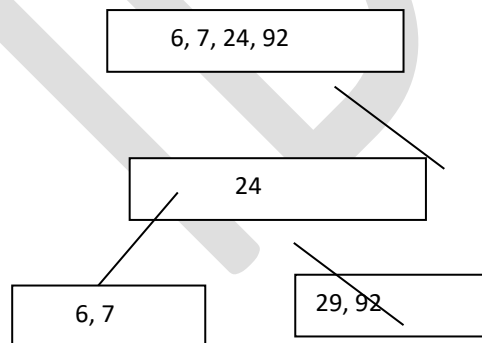
11.3.1 B\*-Tree: B\* tree is a multi valued tree in which a node consists of multiple values and satisfy the following properties.

1. B\* tree of order m will have maximum m pointers and m-1 key values in a node. All the keys must be placed in an increasing order or ascending order.
2. If a node has m-1 key values then the node is full. If a node is full then split the node only if all the siblings are full. If all the siblings are full then only select the median value which becomes the parent value. The values coming left of median value will become the left child of parent value and the values coming to right to the median value will become the right child of the median value. The new key value will be inserted either left child of the parent node (if it is lesser then its parent node value) or right child value of the parent node ( if it is greater than its parent value). If siblings are not full the rotate the values of the leaf node and make the place empty for the new key value.
3. A new key value will be added at the leaf node value.
4. The leaves must be at the same level.
5. The height of B\*-tree will always be lesser than B Tree.
6. B\* tree is referred for classification of topic. B\* tree is also referred for the purpose of indexing.
7. The new value will be inserted in the leaf node.

11.3.2 Construction of B\* Tree: Construct B\*- tree of order 5 for the following values:

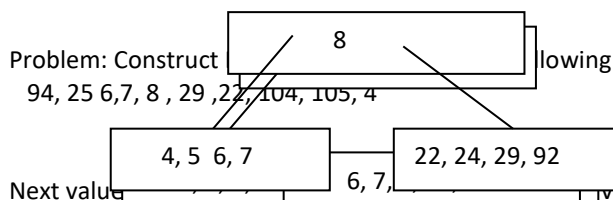
92, 24, 6, 7, 29, 8, 22, 4, 5

Maximum Number of Key values in a node =  $m-1=5-1=4$

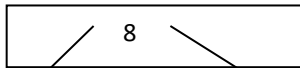


Problem: Construct B\*-tree of order 5 for the following values:

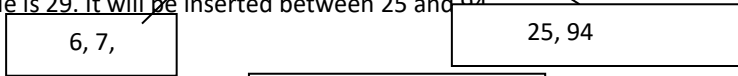
94, 25, 6, 7, 8, 29, 22, 104, 105, 4



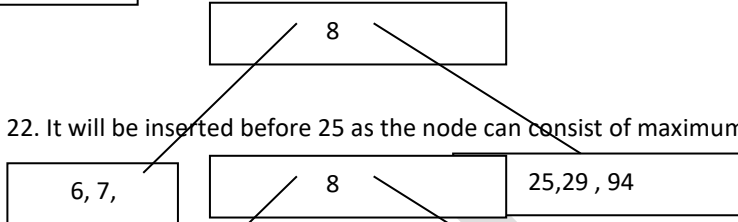
Next value 6, 7, 8, 24, 92. 8 will become the root node value and the values which are coming left to 8 (6,7) will become the left child of root node and the values which are coming right to 8 (24, 82) will become the right child of median value.



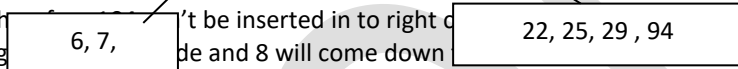
Next value is 29. It will be inserted between 25 and 94



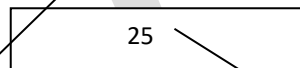
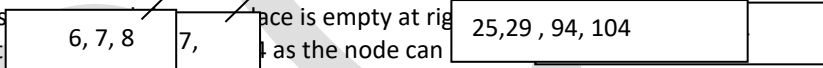
Next value is 22. It will be inserted before 25 as the node can consist of maximum 4 key values.



Next value is 104. It will be inserted after 94 as the node can consist of maximum 4 key values But the node is full therefore it can't be inserted in to right child of 8. Hence new value can be inserted after rotating 22 as a root node and 8 will come down to the left hand side.



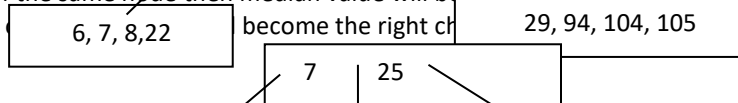
Once the key values 6, 7, 8, 7, 22 are present in the tree, the place is empty at right child of 22. Next value is 105. It can't be inserted in to right child of 22 as the node is full therefore 105 can't be inserted in to right child of 8. Hence new value can be inserted after rotating 22 as a root node and 8 will come down to the left hand side.



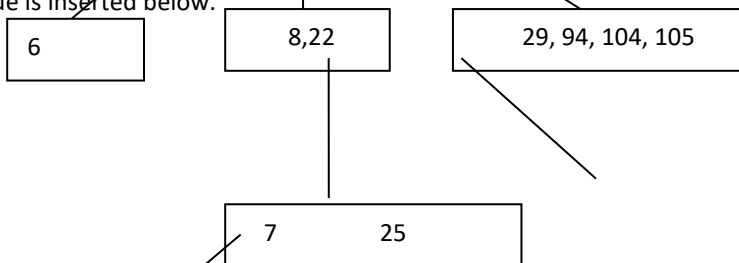
Now the place is empty at the right child of 25 therefore 105 can be inserted in to the right child of 25.



Next value is 4 but both the siblings are full therefore break the left node of 25. Since 4 is the left most value of the same node then median value will become 7. 7 will move up and 4,6 will become the left child of 7. 22 will become the right child of 7.



Here the new value is inserted below.

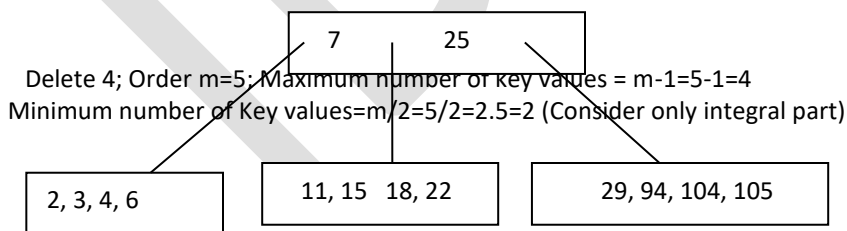


#### 11.4 Deletion from B Tree:

Deletion from B tree is the deleting the key values from a node. Below are the rules to delete the key values from the node.

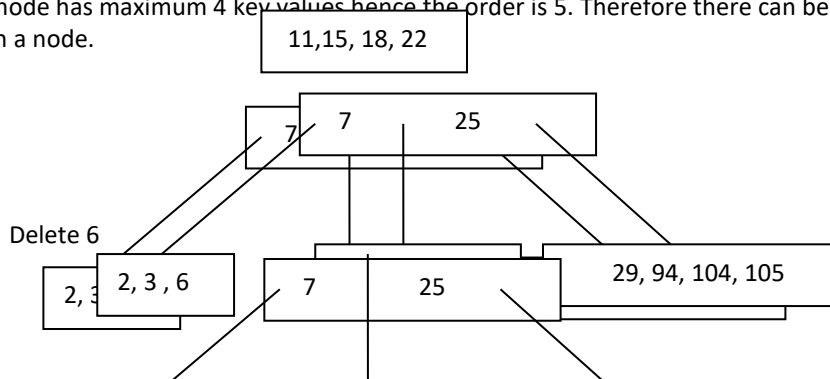
1. The key values are deleted from the leaf node only.
2. At a time only one key value is deleted.
3. The key value can't be deleted from root node and internal node.
4. Key value can't be deleted if the number of key values is less the  $m/2$  key values.
5. If the key value which is available at leaf node is supposed to be deleted and the node have more than  $m/2$  key values than delete the key value from the leaf node directly.
6. If the key value which is available at leaf node is supposed to be deleted and the node have  $m/2$  key values or less than  $m/2$  key values then merge the leaf node siblings and then delete the key value from the leaf node directly.
7. If the key value which is not available at leaf node is supposed to be deleted and the node have more than  $m/2$  key values then move key value at the leaf node and then delete the key value from the leaf node directly.

Delete the following key values from the below B-Tree.



Four is deleted from the leaf node directly since it was available at the leaf node and the node has more than  $m/2$  Key values.

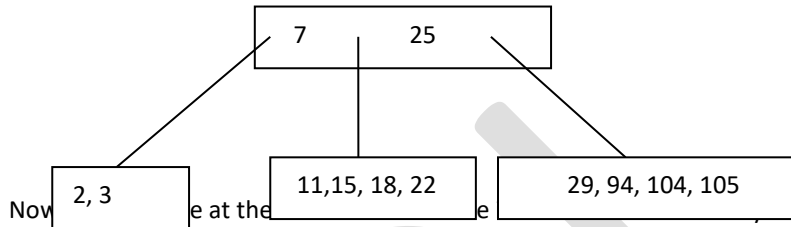
Since a node has maximum 4 key values hence the order is 5. Therefore there can be minimum 2 key values in a node.



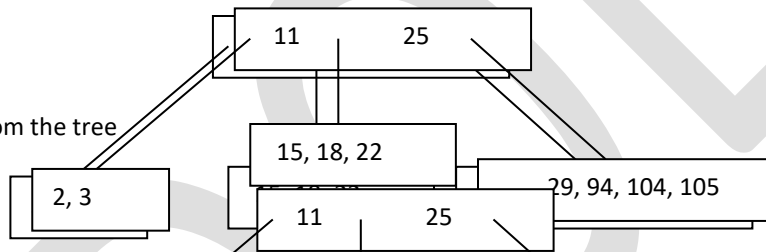
6 is deleted from the leaf node since it was available at the leaf node and number of key values in a node was more than  $m/2$ .

Delete 7

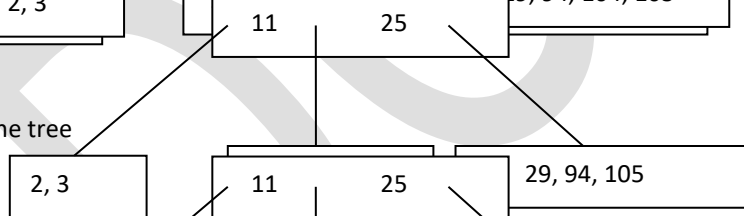
7 can't be deleted directly since it is not available at the leaf node therefore rotate the key values.



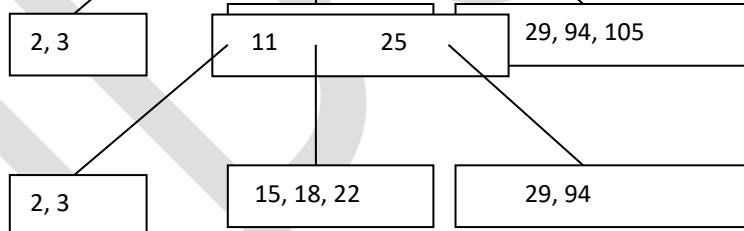
Delete 104 from the tree



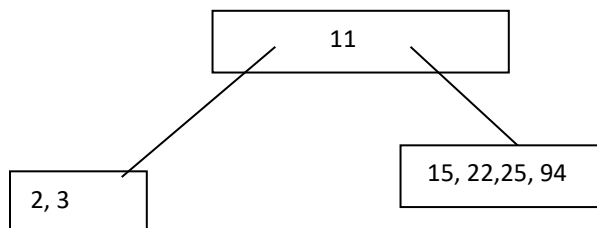
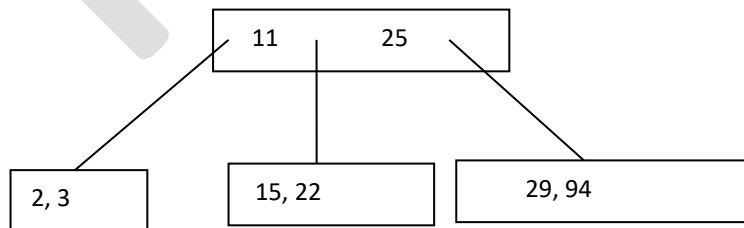
Delete 105 from the tree



Delete 18 from the tree

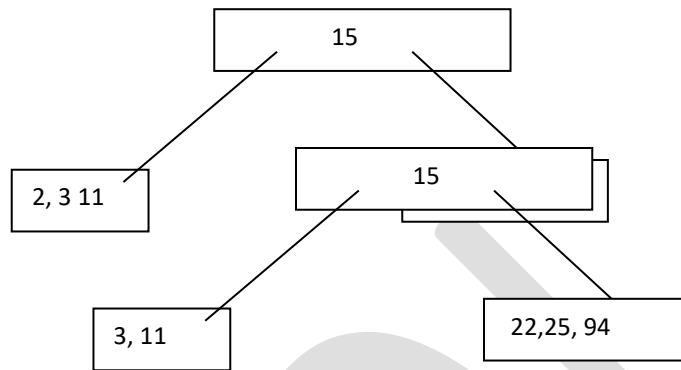


Delete 29 from the tree

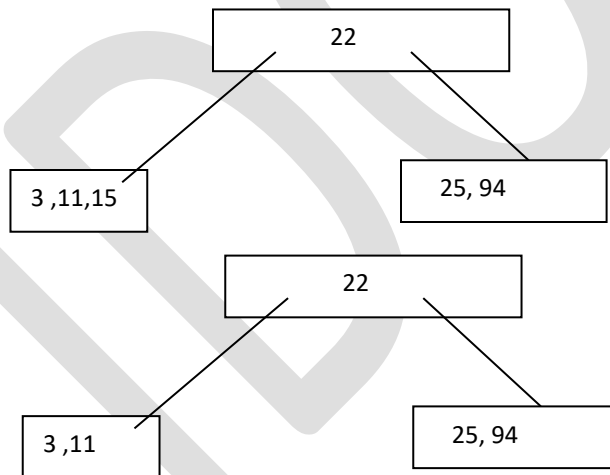


Since the node containing the value 29, will have less than  $m/2$  key values therefore both the siblings will be merged.

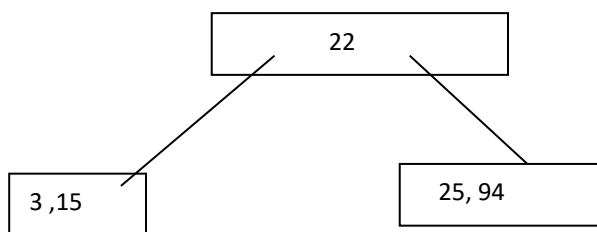
Delete 2 from the tree: First rotate the key values in which 11 will move down and 15 will move up.



Delete 15 from the tree: 22 will move up and 15 will move down then 15 will be deleted from the leaf.



Delete 11 from the tree



Delete 25 from the Tree: 25 is available at the leaf node but both siblings has  $m/2$  key values therefore merger will take place.

3, 15, 22, 94

Delete 94 from the Tree:

3, 15, 22

Delete 15 from the Tree:

3, 22

Further values from the tree can't be deleted since the node is containing  $m/2$  key values and neither rotation is possible nor merger is possible.

11.5 Similarities in B-Tree and B\* Tree

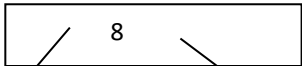
- Both have maximum  $m-1$  key values in a node if the order of the tree is  $m$ .
- Both the trees can have maximum  $m$  pointers in a node if the order of the tree is  $m$ .
- Both the trees the key values in a node must be in increasing order.
- Both the trees leaves must be at the same level.
- At the time of deletion the rotation is possible in both the trees.

Difference in B-Tree and B\* Tree

- The height of B\*-Tree is either lesser or equal to the B-Tree for same number of key values and for the same order of the tree.
- In case of B\* Tree, a node will have more number of key values compare to B Tree.
- B Tree, rotation of key values is not allowed while in B\* Tree rotation of key values is allowed at the time of inserting the key values.
- We do not split the node in B\*-Tree until sibling are full while B tree, we split the node once the node is full.

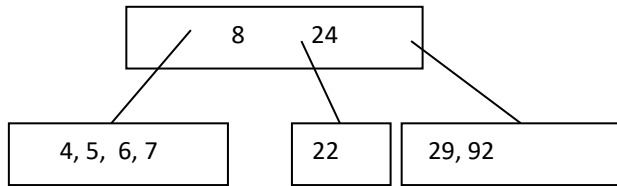
11.6 Practice Problems on B Tree and B\* Tree

Problem: Insert 107 in the following B-tree of order 5.



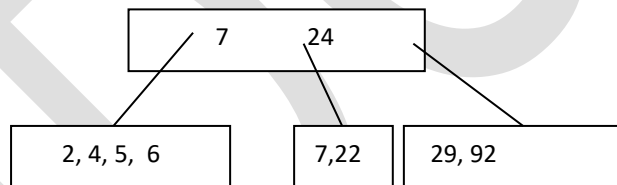
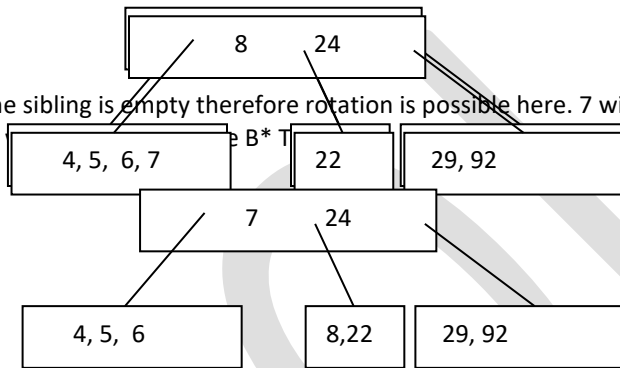


Solution: The value 107 can't be inserted directly in to the B tree of order 5 and a node can contain maximum 4 values in this case the node. Therefore split the right child node.



Problem: Insert 2 in the following B\* Tree of Order 5.

Solution: Since the sibling is empty therefore rotation is possible here. 7 will move up and 8 will come down and then 2



## Unit 6: Chapter 12

### Graphs

#### Introduction to Graphs

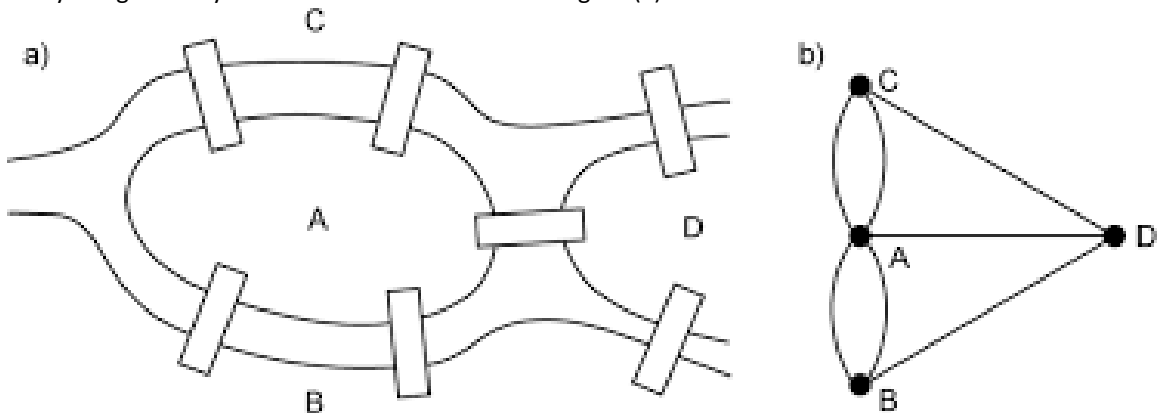
12.0 Objectives .....	274
12.1 Introduction.....	275
12.2 Basic Concepts of Graphs.....	275
12.2.1 Types of Graphs .....	277
12.2.2 Representing Graphs .....	278
12.2.2.1 Adjacency Matrix.....	278
12.2.2.2 Adjacency List:.....	280
12.2.2.3 Adjacency Multi-lists.....	281
12.2.2.4 Weighted edges.....	282
12.2.3 Operations on Graphs.....	282
12.2.3.1 Depth-First Search .....	282
12.2.3.2 Breadth-First Search .....	283
12.2.3.3 Connected Components .....	285
12.3 Graph Traversals.....	289
12.3.1 In-order.....	289
12.3.2 Pre-order.....	290
12.3.3 Post-order .....	290
12.4 Summary .....	291
12.5 Review Your Learning.....	291
12.6 Questions.....	291
12.7 Further Reading .....	292
12.8 References.....	292

#### 12.0 Objectives

1. Explain basic graph terminologies.
2. Understand adjacency matrix and convert to graphs.
3. Describe various operations like BFS, DFS performed on graphs
4. Analyse the applications of graphs in day-to-day life

## 12.1 Introduction

A Graph in data structure is a type of non-linear data structure. Map is a well-established example of a graph. In a map, various cities are connected using links. These links can be considered as roads, railway lines or aerial network. Leonhard Euler was a scientist and he used graph theory to solve Seven Bridges of Konigsberg problem in 1736. He laid the foundations of Graph Theory idea of topology. The problem of Konigsberg bridge was to find whether there is a possible way to traverse every bridge exactly once. This is shown in below in figure (a) and is called as Euler's Tour.



**Figure: (a) Seven Bridges of Konigsberg and (b) Graphical Representation of figure (a)**

As we can see the graphical representation of Konigsberg's seven bridges in figure b, here the points A, B, C and D are called as Vertex in graph terminologies and the paths joining to these vertices are called as edges.

We can represent any graphical scenario with the help of graphs and find a solution for the same.

Applications of Graphs in real life:

1. Solving Electricity Distribution problem
2. Maps like Cities, Rivers, Countries and so on
3. Water distribution in various areas
4. CAD/CAM applications
5. Finding Disaster Relief Solutions

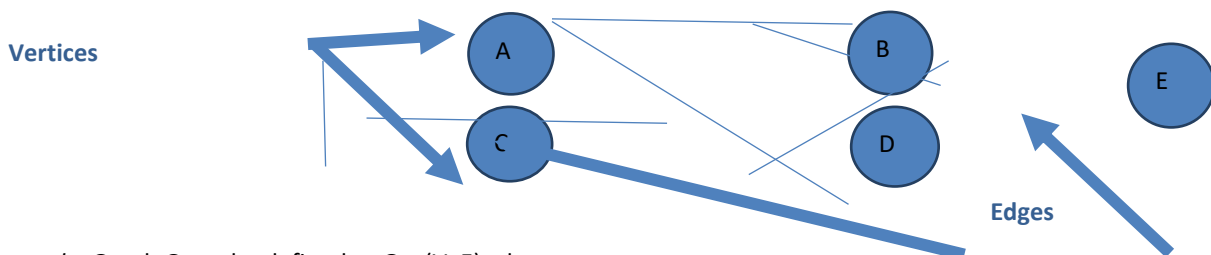
## 12.2 Basic Concepts of Graphs

**Nodes / Vertices:** A graph contains a set of points known as nodes or vertices

**Edge / Link / Arc:** A link joining any two-vertex known as edge or Arc.

**Graph:** A graph is a collection of vertices and arcs which connects vertices in the graph.

A graph G is represented as  $G = (V, E)$ , where V is set of vertices and E is set of edges.



**Example:** Graph G can be defined as  $G = (V, E)$  where,

$V = \{A, B, C, D, E\}$  and

$E = \{(A, B), (A, C), (A, D), (B, D), (C, D), (B, E), (E, D)\}$ .

This is a graph with 5 vertices and 6 edges.

### Graph Terminology

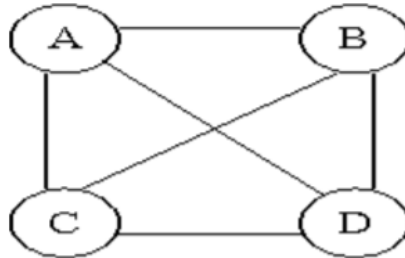
1. **Vertex:** An individual data element of a graph is called as Vertex. Vertex is also known as node.  
In above example graph, A, B, C, D & E are known as vertices.
2. **Edge:** An edge is a connecting link between two vertices. Edge is also known as Arc. An edge is represented as (starting Vertex, ending Vertex).  
Example: In above graph, the link between vertices A and B is represented as (A,B)

Edges are of three types:

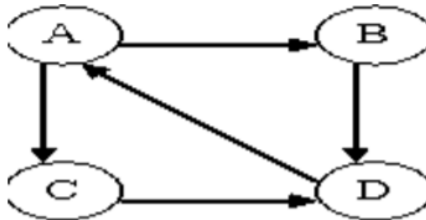
- a. **Undirected Edge-** An undirected edge is a bidirectional edge. If there is an undirected edge between vertices A and B then edge (A, B) is equal to edge (B, A).
  - b. **Directed Edge** - A directed edge is a unidirectional edge. If there is a directed edge between vertices A and B then edge (A, B) is not equal to edge (B, A).
  - c. **Weighted Edge** - A weighted edge is an edge with cost as weight on it.
3. **Degree of a Vertex:** The degree of a vertex is said to be the number of edges incident on it.  
  
Euler showed that there is a path beginning at any vertex, going through each edge exactly once and terminating at the start vertex iff the degree of each vertex is even. A walk which does this is called Eulerian.  
Ex: There is no Eulerian walk for the Königsberg bridge problem as all four vertices are of odd degree.
  4. **Outgoing Edge:** A directed edge is said to be outgoing edge on its origin vertex.
  5. **Incoming Edge:** A directed edge is said to be incoming edge on its destination vertex.
  6. **Degree:** Total number of edges connected to a vertex is said to be degree of that vertex.
  7. **Indegree:** Total number of incoming edges connected to a vertex is said to be indegree of that vertex.
  8. **Outdegree:** Total number of outgoing edges connected to a vertex is said to be outdegree of that vertex.
  9. **Parallel edges or Multiple edges:** If there are two undirected edges to have the same end vertices, and for two directed edges to have the same origin and the same destination. Such edges are called parallel edges or multiple edges.
  10. **Self-loop:** An edge (undirected or directed) is a self-loop if its two endpoints coincide.
  11. **Simple Graph**
  12. A graph is said to be simple if there are no parallel and self-loop edges.

### 12.2.1 Types of Graphs

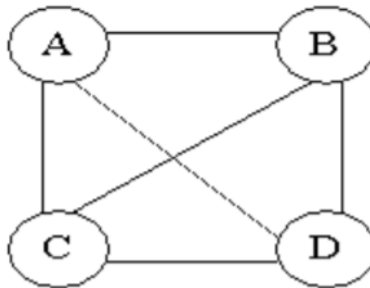
1. **Undirected Graph:** A graph with only undirected edges is said to be undirected graph.



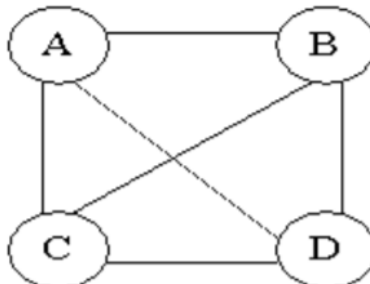
2. **Directed Graph:** A graph with only directed edges is said to be directed graph.



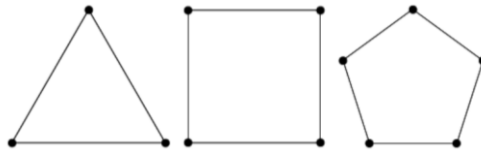
3. **Complete Graph:** A graph in which any V node is adjacent to all other nodes present in the graph is known as a complete graph. An undirected graph contains the edges that are equal to  $\text{edges} = \frac{n(n-1)}{2}$  where n is the number of vertices present in the graph. The following figure shows a complete graph.



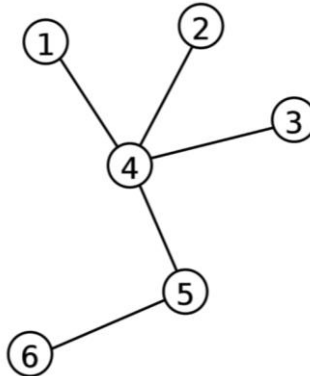
4. **Regular Graph:** Regular graph is the graph in which nodes are adjacent to each other, i.e., each node is accessible from any other node.



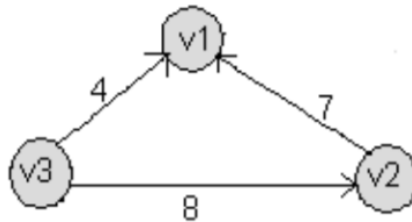
5. **Cycle Graph:** A graph having cycle is called cycle graph. In this case the first and last nodes are the same. A closed simple path is a cycle.



6. **Acyclic Graph:** A graph without cycle is called acyclic graphs.



7. **Weighted Graph:** A graph is said to be weighted if there are some non-negative value assigned to each edges of the graph. The value is equal to the length between two vertices. Weighted graph is also called a network.



## 12.2.2 Representing Graphs

Graph data structure is represented using following representation types:

1. Adjacency Matrix
2. Adjacency List
3. Adjacency Multi-list

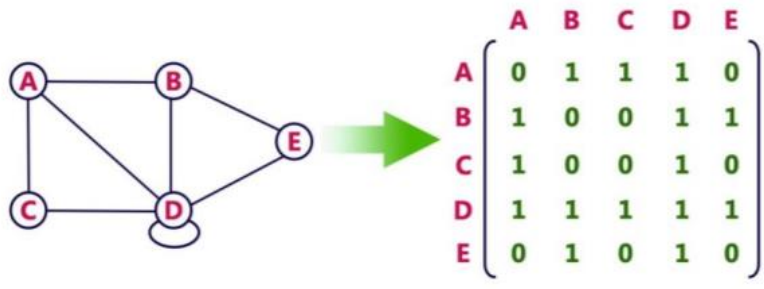
### 10.2.2.1 Adjacency Matrix

In this representation, graph can be represented using a matrix of size total number of vertices by total number of vertices; means if a graph with 4 vertices can be represented using a matrix of 4X4 size.

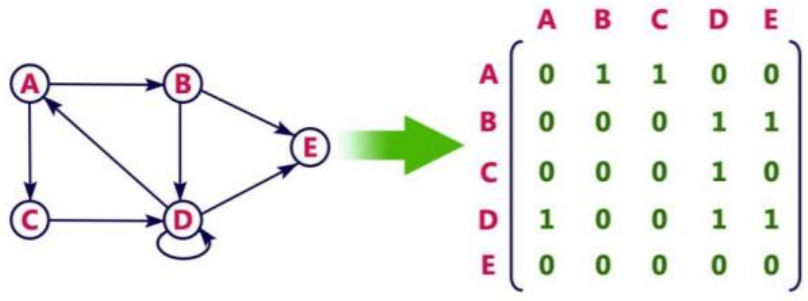
In this matrix, rows and columns both represent vertices. This matrix is filled with either 1 or 0. Here, 1 represents there is an edge from row vertex to column vertex and 0 represents there is no edge from row vertex to column vertex.

**Adjacency Matrix:** Let  $G = (V, E)$  with  $n$  vertices,  $n \geq 1$ . The adjacency matrix of  $G$  is a 2-dimensional  $n \times n$  matrix,  $A$ ,  $A(i, j) = 1$  iff  $(v_i, v_j) \in E(G)$  ( $\langle v_i, v_j \rangle$  for a digraph),  $A(i, j) = 0$  otherwise.

Example: For undirected graph



For a Directed graph



The adjacency matrix for an undirected graph is symmetric but the adjacency matrix for a digraph need not be symmetric.

Merits of Adjacency Matrix: From the adjacency matrix, to determine the connection of vertices is easy.

The degree of a vertex is

$$\sum_{j=0}^{n-1} adj\_mat[i][j]$$

For a digraph, the row sum is the out\_degree, while the column sum is the in\_degree.

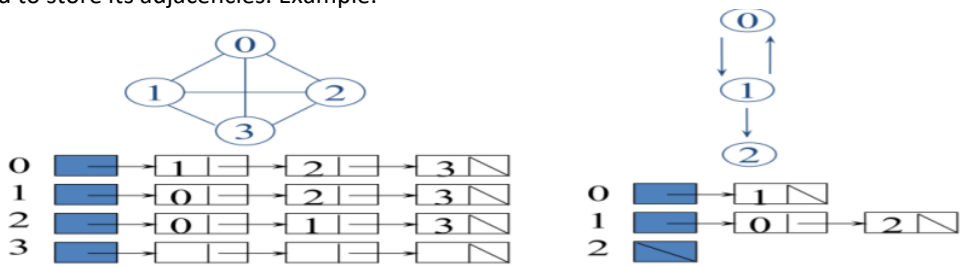
$$ind(v_i) = \sum_{j=0}^{n-1} A[j, i] \qquad outd(v_i) = \sum_{j=0}^{n-1} A[i, j]$$

The space needed to represent a graph using adjacency matrix is  $n^2$  bits. To identify the edges in a graph, adjacency matrices will require at least  $O(n^2)$  time.

## 2. Adjacency List

In this representation, every vertex of graph contains list of its adjacent vertices. The  $n$  rows of the adjacency matrix are represented as  $n$  chains. The nodes in chain  $i$  represent the vertices that are adjacent to vertex  $i$ .

It can be represented in two forms. In one form, array is used to store  $n$  vertices and chain is used to store its adjacencies. Example:



So that we can access the adjacency list for any vertex in  $O(1)$  time.

Adjlist[i] is a pointer to the first node in the adjacency list for vertex  $i$ . Structure is

```
#define MAX_VERTICES 50
```

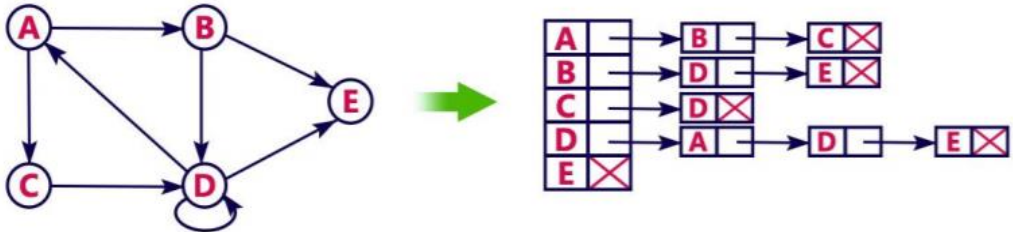
```

typedef struct node *node_pointer;
typedef struct node { int vertex; struct node *link; };
node_pointer graph[MAX_VERTICES];
int n=0; /* vertices currently in use */

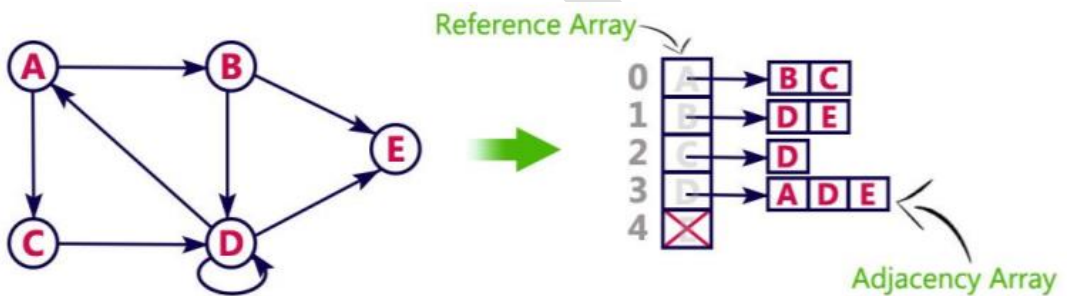
```

Another type of representation is given below.

Example: Consider the following directed graph representation implemented using linked list.



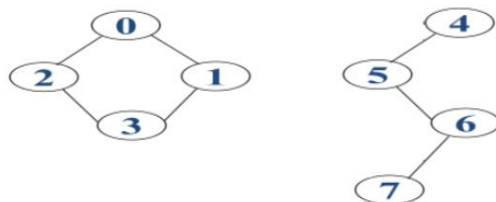
This representation can also be implemented using array



### 12.2.2.2 Adjacency List:

Sequential representation of adjacency list and its conversion to graph is:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22
9	11	13	15	17	18	20	22	23	2	1	3	0	0	3	1	2	5	6	4	5	7	6

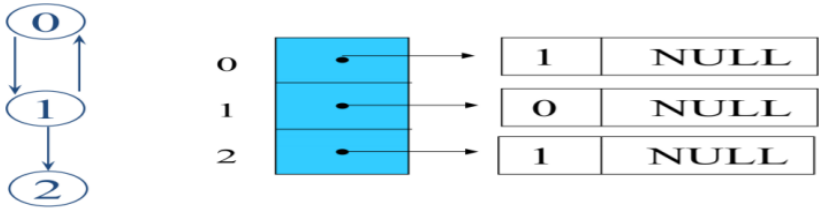


Instead of chains, we can use sequential representation into an integer array with size  $n+2e+1$ . For  $0 \leq i < n$ ,  $Array[i]$  gives starting point of the list for vertex  $i$ , and  $array[n]$  is set to  $n+2e+1$ . The adjacent vertices of node  $i$  are stored sequentially from  $array[i]$ .

For an undirected graph with  $n$  vertices and  $e$  edges, linked adjacency list requires an array of size  $n$  and  $2e$  chain nodes. For a directed graph, the number of list nodes is only  $e$ . The out degree of any vertex may be determined by counting the number of nodes in its adjacency list. To find in-degree of vertex  $v$ , we must traverse complete list.

To avoid this, inverse adjacency list is used which contain in-degree.





Determine in-degree of a vertex in a fast way.

### 12.2.2.3 Adjacency Multi-lists

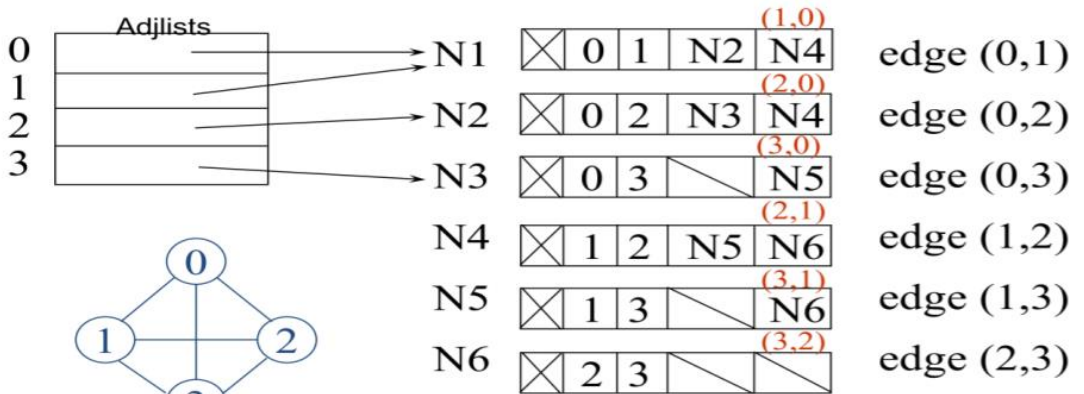
In the adjacency-list representation of an undirected graph each edge  $(u, v)$  is represented by two entries one on the list for  $u$  and the other on the list for  $v$ . As we shall see in some situations it is necessary to be able to determine efficiently for a particular edge and mark that edge as having been examined. This can be accomplished easily if the adjacency lists are actually maintained as multilists (i.e., lists in which nodes may be shared among several lists). For each edge there will be exactly one node but this node will be in two lists (i.e. the adjacency lists for each of the two nodes to which it is incident).

```

For adjacency multilists, node structure is
typedef struct edge *edge_pointer;
typedef struct edge {
    short int marked;
    int vertex1, vertex2;
    edge_pointer path1, path2;
};
edge_pointer graph[MAX_VERTICES];
  
```

marked	vertex1	vertex2	path1	path2
--------	---------	---------	-------	-------

Lists: vertex 0: N0->N1->N2, vertex 1: N0->N3->N4  
 vertex 2: N1->N3->N5, vertex 3: N2->N4->N5

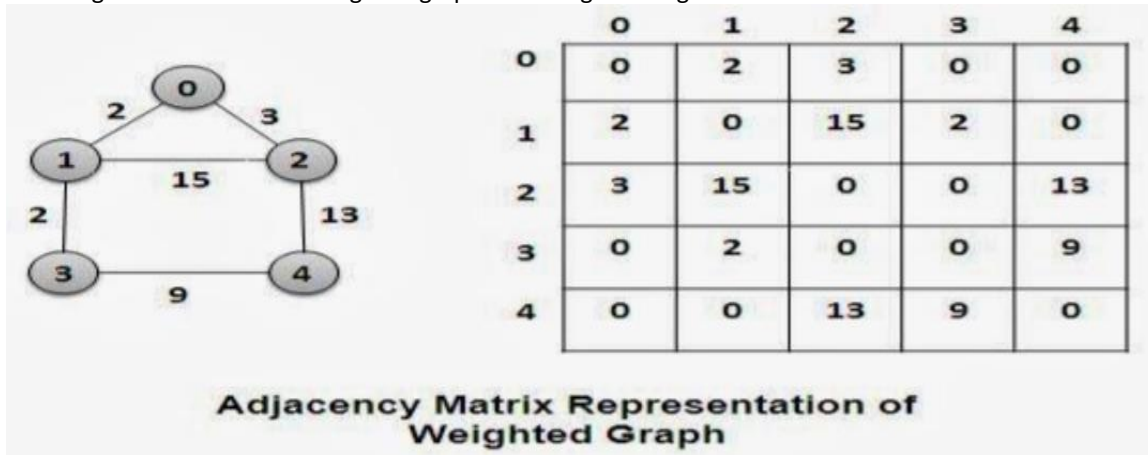


six edges

Figure: Adjacency multilists for given graph

### 12.2.2.4 Weighted edges

In many applications the edges of a graph have weights assigned to them. These weights may represent the distance from one vertex to another or the cost of going from one; vertex to an adjacent vertex. In these applications the adjacency matrix entries  $A[i][j]$  would keep this information too. When adjacency lists are used the weight information may be kept in the list's nodes by including an additional field weight. A graph with weighted edges is called a network.



### 12.2.3 Operations on Graphs

Given a graph  $G = (V, E)$  and a vertex  $v$  in  $V(G)$  we wish to visit all vertices in  $G$  that are reachable from  $v$  (i.e., all vertices that are connected to  $v$ ). We shall look at two ways of doing this: depth-first search and breadth-first search. Although these methods work on both directed and undirected graphs the following discussion assumes that the graphs are undirected.

#### 12.2.3.1 Depth-First Search

- ☑ Begin the search by visiting the start vertex  $v$ 
  - If  $v$  has an unvisited neighbor, traverse it recursively
  - Otherwise, backtrack
- ☑ Time complexity
  - Adjacency list:  $O(|E|)$
  - Adjacency matrix:  $O(|V|^2)$

We begin by visiting the start vertex  $v$ . Next an unvisited vertex  $w$  adjacent to  $v$  is selected, and a depth-first search from  $w$  is initiated. When a vertex  $u$  is reached such that all its adjacent vertices have been visited, we back up to the last vertex visited that has an unvisited vertex  $w$  adjacent to it and initiate a depth-first search from  $w$ .

The search terminates when no unvisited vertex can be reached from any of the visited vertices. 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.

We use the following steps to implement DFS traversal...

Step 1: Define a Stack of size total number of vertices in the graph.

Step 2: Select any vertex as starting point for traversal. Visit that vertex and push it on to the Stack.

Step 3: 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.

Step 4: Repeat step 3 until there are no new vertex to be visit from the vertex on top of the stack.  
 Step 5: When there is no new vertex to be visit then use back tracking and pop one vertex from the stack.  
 Step 6: Repeat steps 3, 4 and 5 until stack becomes Empty.  
 Step 7: When stack becomes Empty, then produce final spanning tree by removing unused edges from the graph.  
 This function is best described recursively as in Program.

```

#define FALSE 0
#define TRUE 1
int visited[MAX_VERTICES];
void dfs(int v)
{
  node_pointer w;
  visited[v]= TRUE;
  printf("%d", v);
  for (w=graph[v]; w; w=w->link)
    if (!visited[w->vertex])
      dfs(w->vertex);
}

```

Consider the graph G of Figure 6.16(a), which is represented by its adjacency lists as in Figure 6.16(b). If a depthfirst search is initiated from vertex 0 then the vertices of G are visited in the following order: 0 1 3 7 4 5 2 6. Since DFS(0) visits all vertices that can be reached from 0 the vertices visited, together with all edges in G incident to these vertices form a connected component of G.

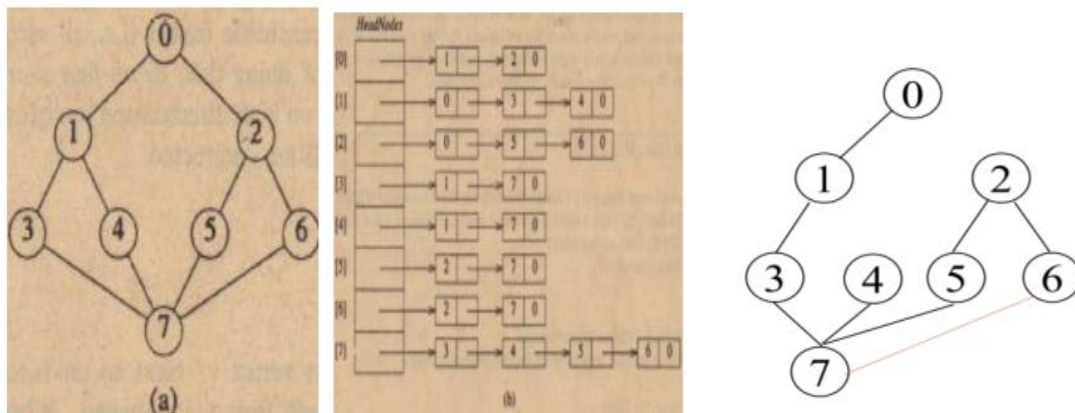


Figure: Graph and its adjacency list representation, DFS spanning tree

**Analysis or DFS:**

When G is represented by its adjacency lists, the vertices w adjacent to v can be determined by following a chain of links. Since DFS examines each node in the adjacency lists at most once and there are 2e list nodes the time to complete the search is O(e). If G is represented by its adjacency matrix then the time to determine all vertices adjacent to v is O(n). Since at most n vertices are visited the total time is O(n<sup>2</sup>).

**12.2.3.2 Breadth-First Search**

In a breadth-first search, we begin by visiting the start vertex v. Next all unvisited vertices adjacent to v are

visited. Unvisited vertices adjacent to these newly visited vertices are then visited and so on.  
Algorithm BFS.

**Program:**

```
typedef struct queue *queue_pointer;
typedef struct queue {
    int vertex;
    queue_pointer link;
};
void addq(queue_pointer *,
queue_pointer *, int);
int deleteq(queue_pointer *);
void bfs(int v)
{
    node_pointer w;
    queue_pointer front, rear;
    front = rear = NULL;
    printf("%d", v);
    visited[v] = TRUE;
    addq(&front, &rear, v);
    while (front) {
        v = deleteq(&front);
        for (w = graph[v]; w; w = w->link)
            if (!visited[w->vertex]) {
                printf("%d", w->vertex);
                addq(&front, &rear, w->vertex);
                visited[w->vertex] = TRUE;
            }
    }
}
```

**Steps:**

BFS traversal of a graph, produces a spanning tree as final result. Spanning Tree is a graph without any loops. We use Queue data structure with maximum size of total number of vertices in the graph to implement BFS traversal of a graph.

We use the following steps to implement BFS traversal...

**Step 1:** Define a Queue of size total number of vertices in the graph.

**Step 2:** Select any vertex as starting point for traversal. Visit that vertex and insert it into the Queue.

**Step 3:** Visit all the adjacent vertices of the vertex which is at front of the Queue which is not visited and insert them into the Queue.

**Step 4:** When there is no new vertex to be visit from the vertex at front of the Queue then delete that vertex from the Queue.

**Step 5:** Repeat step 3 and 4 until queue becomes empty.

**Step 6:** When queue becomes Empty, then produce final spanning tree by removing unused edges from the graph

**Analysis of BFS:**

Each visited vertex enters the queue exactly once. So the while loop is iterated at most n times If an adjacency matrix is used the loop takes  $O(n)$  time for each vertex visited. The total time is therefore,  $O(n^2)$ . If adjacency lists are used the loop has a total cost of  $d_0 + \dots + d_{n-1} = O(e)$ , where d is the degree of vertex i. As in the case of DFS all visited vertices together with all edges incident to them, form a connected component of G.

### 12.2.3.3 Connected Components

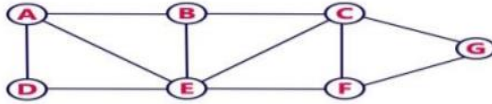
If G is an undirected graph, then one can determine whether or not it is connected by simply making a call to either DFS or BFS and then determining if there is any unvisited vertex. The connected components of a graph may be obtained by making repeated calls to either DFS(v) or BFS(v); where v is a vertex that has not yet been visited. This leads to function Connected as given below in program which determines the connected components of G. The algorithm uses DFS (BFS may be used instead if desired). The computing time is not affected. Function connected –Output outputs all vertices visited in the most recent invocation of DFS together with all edges incident on these vertices.

```
void connected(void){
    for (i=0; i<n; i++) {
        if (!visited[i]) {
            dfs(i);
            printf("\n");
        }
    }
}
```

#### Analysis of Components:

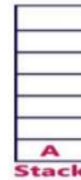
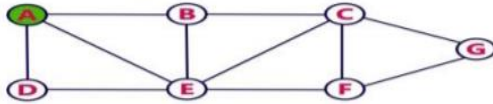
If G is represented by its adjacency lists, then the total time taken by dfs is O(e). Since the for loops take O(n) time, the total time to generate all the Connected components is O(n+e). If adjacency matrices are used, then the time required is O(n<sup>2</sup>).

Consider the following example graph to perform DFS traversal



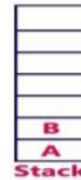
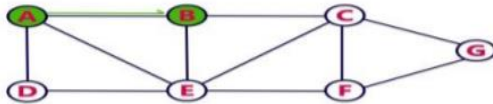
**Step 1:**

- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.



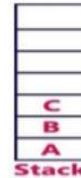
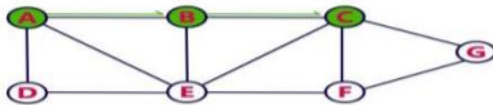
**Step 2:**

- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex **B** on to the Stack.



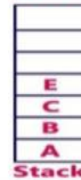
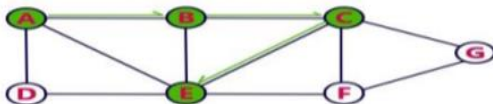
**Step 3:**

- Visit any adjacent vertex of **B** which is not visited (**C**).
- Push **C** on to the Stack.



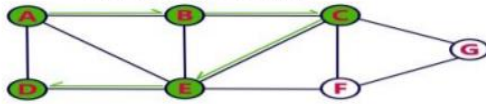
**Step 4:**

- Visit any adjacent vertex of **C** which is not visited (**E**).
- Push **E** on to the Stack



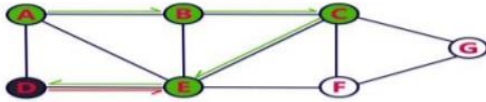
**Step 5:**

- Visit any adjacent vertex of **E** which is not visited (**D**).
- Push **D** on to the Stack



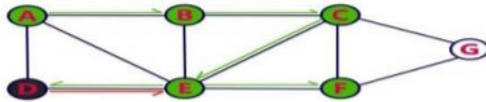
**Step 6:**

- There is no new vertex to be visited from **D**. So use back track.
- Pop **D** from the Stack.



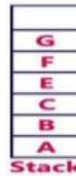
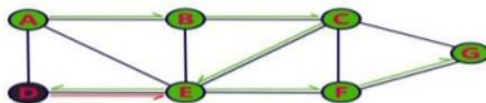
**Step 7:**

- Visit any adjacent vertex of **E** which is not visited (**F**).
- Push **F** on to the Stack.



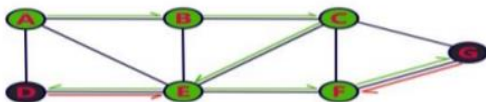
**Step 8:**

- Visit any adjacent vertex of **F** which is not visited (**G**).
- Push **G** on to the Stack.

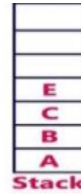
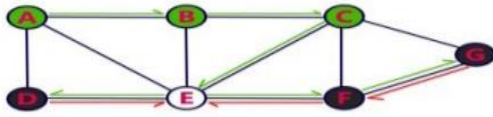


**Step 9:**

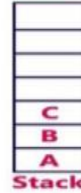
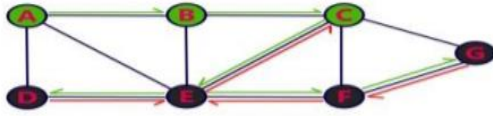
- There is no new vertex to be visited from **G**. So use back track.
- Pop **G** from the Stack.



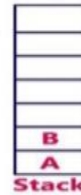
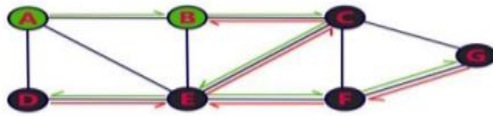
**Step 10:**  
 - There is no new vertex to be visited from F. So use back track.  
 - Pop F from the Stack.



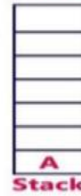
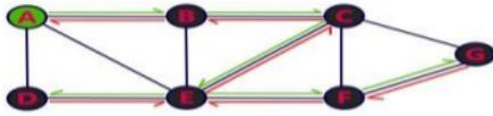
**Step 11:**  
 - There is no new vertex to be visited from E. So use back track.  
 - Pop E from the Stack.



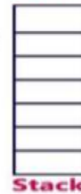
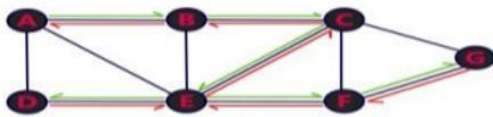
**Step 12:**  
 - There is no new vertex to be visited from C. So use back track.  
 - Pop C from the Stack.



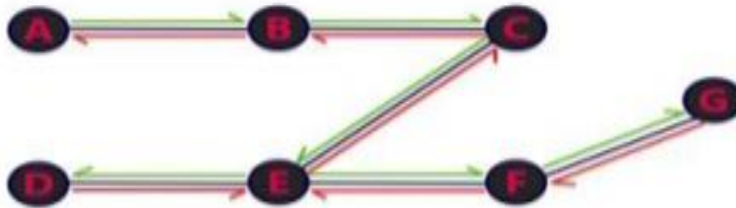
**Step 13:**  
 - There is no new vertex to be visited from B. So use back track.  
 - Pop B from the Stack.



**Step 14:**  
 - There is no new vertex to be visited from A. So use back track.  
 - Pop A from the Stack.

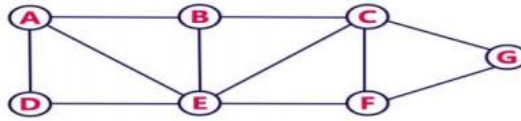


- Stack became Empty. So stop DFS Traversal.  
 - Final result of DFS traversal is following spanning tree.



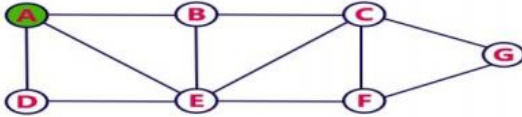
Consider the following example for BFS traversal.





**Step 1:**

- Select the vertex **A** as starting point (visit **A**).
- Insert **A** into the Queue.

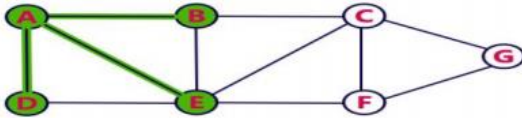


Queue



**Step 2:**

- Visit all adjacent vertices of **A** which are not visited (**D, E, B**).
- Insert newly visited vertices into the Queue and delete A from the Queue..

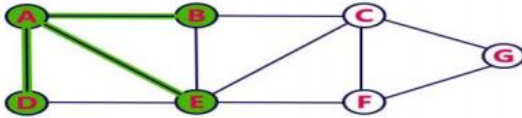


Queue



**Step 3:**

- Visit all adjacent vertices of **D** which are not visited (there is no vertex).
- Delete D from the Queue.

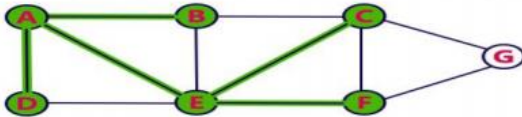


Queue



**Step 4:**

- Visit all adjacent vertices of **E** which are not visited (**C, F**).
- Insert newly visited vertices into the Queue and delete E from the Queue.



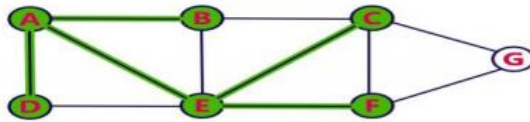
Queue





**Step 5:**

- Visit all adjacent vertices of **B** which are not visited (**there is no vertex**).
- Delete **B** from the Queue.

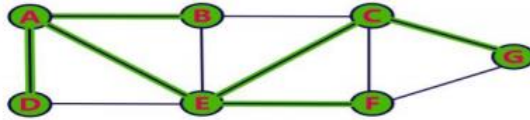


Queue



**Step 6:**

- Visit all adjacent vertices of **C** which are not visited (**G**).
- Insert newly visited vertex into the Queue and delete **C** from the Queue.

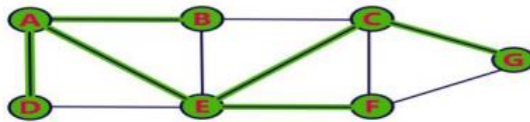


Queue



**Step 7:**

- Visit all adjacent vertices of **F** which are not visited (**there is no vertex**).
- Delete **F** from the Queue.

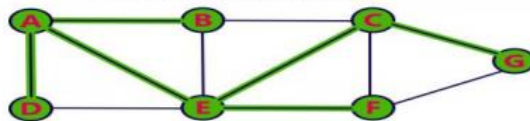


Queue



**Step 8:**

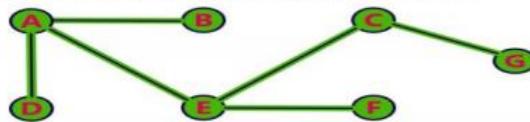
- Visit all adjacent vertices of **G** which are not visited (**there is no vertex**).
- Delete **G** from the Queue.



Queue



- Queue became Empty. So, stop the BFS process.
- Final result of BFS is a Spanning Tree as shown below...



## 12.3 Graph Traversals

Unlike linear data structures (Array, Linked List, Queues, Stacks, etc) which have only one logical way to traverse them, trees can be traversed in different ways. Following are the generally used ways for traversing trees.

### 12.3.1 In-order

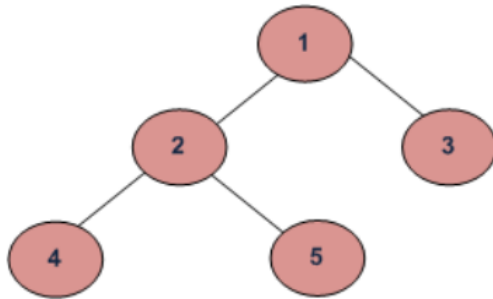
Algorithm Inorder(tree)

1. Traverse the left subtree, i.e., call Inorder(left subtree)
2. Visit the root.
3. Traverse the right subtree, i.e., call Inorder(right-subtree)

#### Uses of Inorder

In case of binary search trees (BST), Inorder traversal gives nodes in non-decreasing order. To get nodes of BST in non-increasing order, a variation of Inorder traversal where Inorder traversal is reversed can be used.

**Example-1:** Inorder traversal for the below-given tree is 4 2 5 1 3.



**Example-2:** Consider input as given below:

**Input:**



**Output:** 3 1 2

### 12.3.2 Pre-order

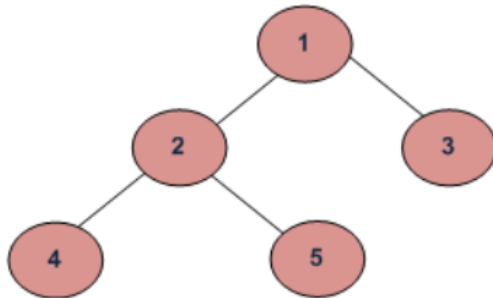
Algorithm Preorder(tree)

1. Visit the root.
2. Traverse the left subtree, i.e., call Preorder(left-subtree)
3. Traverse the right subtree, i.e., call Preorder(right-subtree)

**Uses of Preorder**

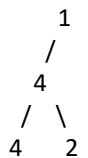
- Preorder traversal is used to create a copy of the tree.
- Preorder traversal is also used to get prefix expression on of an expression tree.

**Example-1:** Preorder traversal for the below given figure is 1 2 4 5 3.



**Example-2:** Consider Input as given below:

**Input:**



**Output:** 1 4 4 2

### 12.3.3 Post-order

Algorithm Postorder(tree)

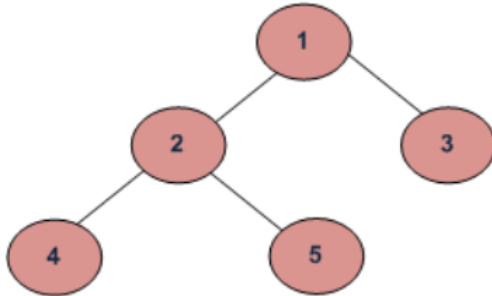
1. Traverse the left subtree, i.e., call Postorder(left-subtree)
2. Traverse the right subtree, i.e., call Postorder(right-subtree)

3. Visit the root.

### Uses of Postorder

- Postorder traversal is used to delete the tree.
- Postorder traversal is also useful to get the postfix expression of an expression tree.

**Example-1:** Postorder traversal for the below given figure is 4 5 2 3 1.



**Example-2:** Consider input as given below:

**Input:**

```
    19
   /  \
  10   8
 /  \
11  13
```

**Output:** 11 13 10 8 19

## 12.4 Summary

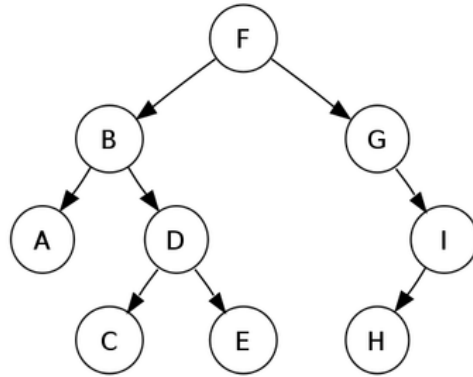
In this chapter, we have seen what graphs are, what are the various terminologies used in graphs. We have also seen the graph operations like Breadth First Search (BFS) & Depth First Search (DFS). We have seen the various graph traversal methods like Inorder, Preorder and Postorder.

## 12.5 Review Your Learning

- Can you explain what are graphs?
- Can you explain what is BFS and DFS?
- Are you able to write the graph nodes sequence using Inorder, Preorder and Postorder traversal methods?
- Can you relate day to day real problems using graphical notations?

## 12.6 Questions

1. Draw a directed graph with five vertices and seven edges. Exactly one of the edges should be a loop, and do not have any multiple edges.
2. Draw an undirected graph with five edges and four vertices. The vertices should be called v1, v2, v3 and v4--and there must be a path of length three from v1 to v4. Draw a squiggly line along this path from v1 to v4.
3. Explain Inorder, Preorder and Postorder traversal methods. Write the sequence for following given graph.



4. Explain BFS and DFA Algorithms with examples.
5. Draw the directed graph that corresponds to this adjacency matrix:

	0	1	2	3
0	True	False	True	False
1	True	False	False	False
2	False	False	False	True
3	True	False	True	False

### 12.7 Further Reading

- <https://lpuguidecom.files.wordpress.com/2017/04/fundamentals-of-data-structures-ellis-horowitz-sartaj-sahni.pdf>
- <https://www.guru99.com/data-structure-algorithms-books.html>
- <https://edutechlearners.com/data-structures-with-c-by-schaum-series-pdf/>

### 12.8 References

1. <http://www.musaliarcollege.com/e-Books/CSE/Data%20structures%20algorithms%20and%20applications%20in%20C.pdf>
2. <https://lpuguidecom.files.wordpress.com/2017/04/fundamentals-of-data-structures-ellis-horowitz-sartaj-sahni.pdf>
3. <https://www.geeksforgeeks.org/>

## Unit 6: Chapter 13

### Graph Algorithms

#### Graph Algorithms

13.0 Objectives .....	293
13.1 Introduction.....	293
13.2 Minimum Spanning Tree .....	295
13.2.1 Spanning Tree .....	295
13.2.2 Minimum Spanning Tree .....	298
13.3 Graph Algorithms .....	299
13.3.1 Kruskal's Algorithm.....	299
13.3.2 Prim's Algorithm.....	303
13.3 Summary .....	307
13.4 Review Your Learning.....	307
13.5 Questions.....	307
13.6 Further Reading .....	308
13.7 References.....	308

#### 13.0 Objectives

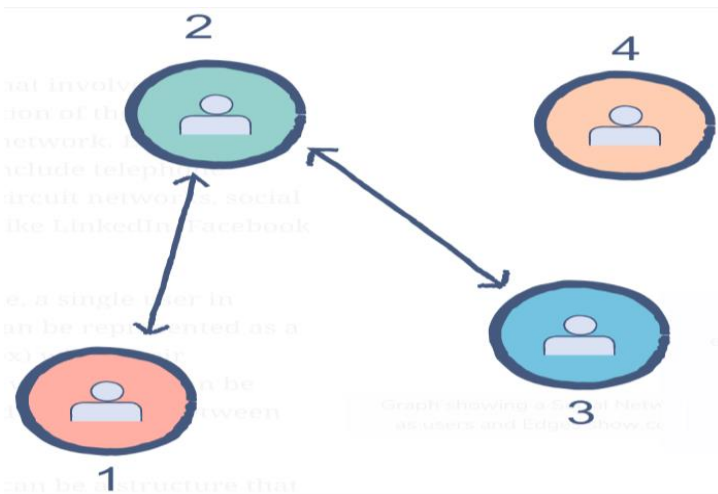
5. Identify Minimum Spanning Tree in a given graph
6. Explain Kruskal's Algorithm
7. Explain Prim's Algorithm
8. Analyse working with Kruskal's Algorithm, Prim's Algorithm
9. Explain Greedy Algorithms.

#### 13.1 Introduction

A graph is a common data structure that consists of a finite set of nodes (or vertices) and a set of edges connecting them.

A pair  $(x,y)$  is referred to as an edge, which communicates that the  $x$  vertex connects to the  $y$  vertex.

In the examples below, circles represent vertices, while lines represent edges.



Graphs are used to solve real-life problems that involve representation of the problem space as a network. Examples of networks include telephone networks, circuit networks, social networks (like LinkedIn, Facebook etc.).

For example, a single user in Facebook can be represented as a node (vertex) while their connection with others can be represented as an edge between nodes.

Each node can be a structure that contains information like user's id, name, gender, etc.

### Types of graphs:

#### A. Undirected Graph:

In an undirected graph, nodes are connected by edges that are all bidirectional. For example, if an edge connects node 1 and 2, we can traverse from node 1 to node 2, and from node 2 to 1.

#### B. Directed Graph

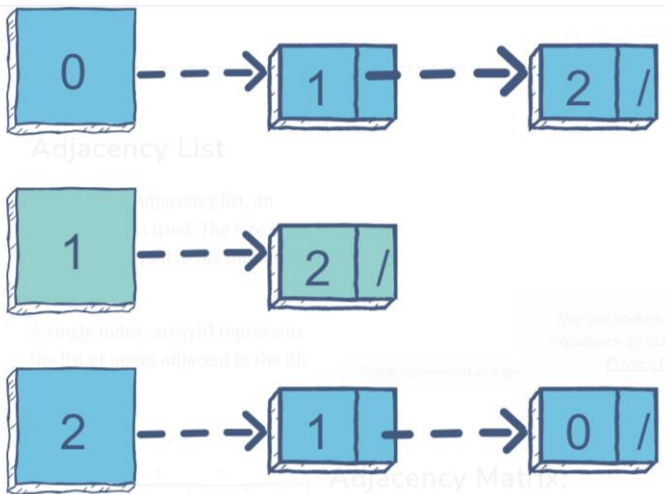
In a directed graph, nodes are connected by directed edges – they only go in one direction. For example, if an edge connects node 1 and 2, but the arrowhead points towards 2, we can only traverse from node 1 to node 2 – not in the opposite direction.

### Types of Graph Representations:

#### A. Adjacency List

To create an Adjacency list, an array of lists is used. The size of the array is equal to the number of nodes.

A single index,  $array[i]$  represents the list of nodes adjacent to the  $i$ th node.



## B. Adjacency Matrix:

An Adjacency Matrix is a 2D array of size  $V \times V$  where  $V$  is the number of nodes in a graph. A slot  $matrix[i][j] = 1$  indicates that there is an edge from node  $i$  to node  $j$ .

	0	1	2	3
0	1			1
1			1	
2		1		
3	1			1

## 13.2 Minimum Spanning Tree

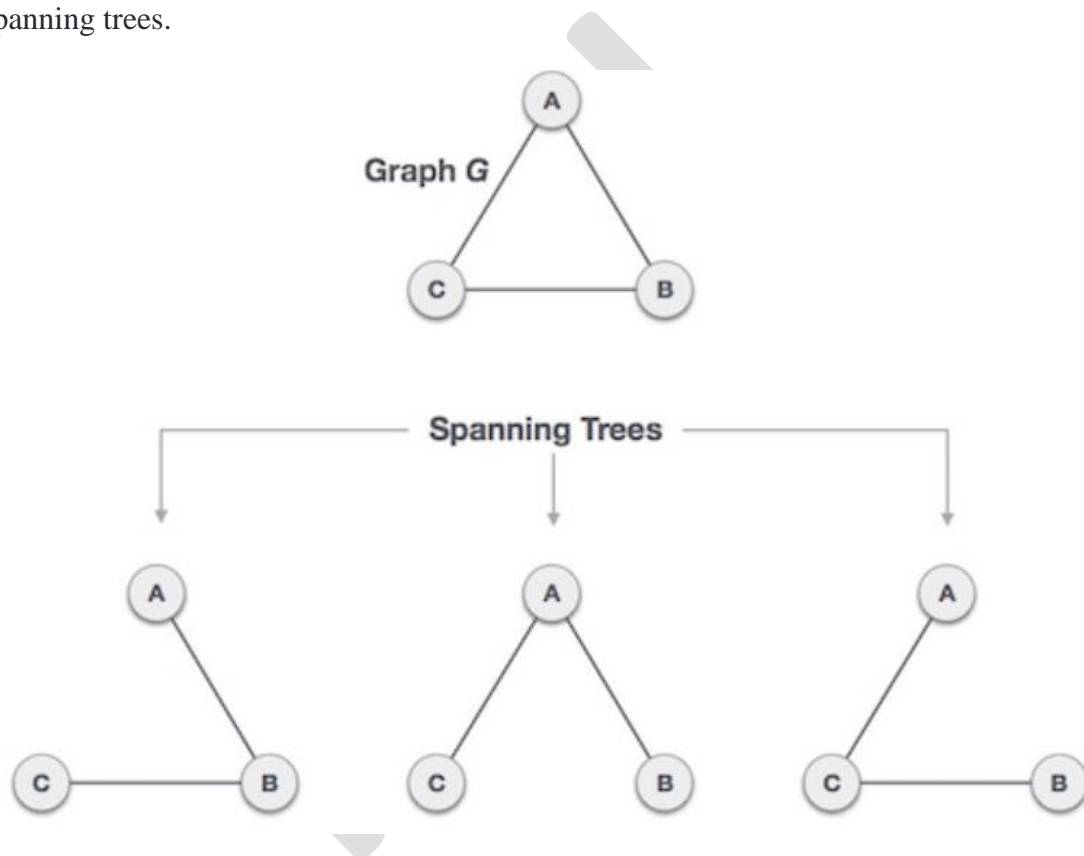
### 13.2.1 Spanning Tree

A spanning tree is a subset of Graph  $G$ , which has all the vertices covered with minimum possible number of edges. Hence, a spanning tree does not have cycles and it cannot be disconnected.

By this definition, we can draw a conclusion that every connected and undirected Graph  $G$  has at least one spanning tree. A disconnected graph does not have any spanning tree, as it cannot be spanned to all its vertices.

Given an undirected and connected graph  $G=(V, E)$ , a spanning tree of the graph  $G$  is a tree that spans  $G$  (that is, it includes every vertex of  $G$ ) and is a subgraph of  $G$  (every edge in the tree belongs to  $G$ ).

Following figure shows the original undirected graph and its various possible spanning trees.



We found three spanning trees off one complete graph. A complete undirected graph can have maximum  $n^{n-2}$  number of spanning trees, where  $n$  is the number of nodes. In the above addressed example,  $n$  is 3, hence  $3^{3-2} = 3$  spanning trees are possible.

### General Properties of Spanning Tree

As one graph can have more than one spanning tree. Following are a few properties of the spanning tree connected to graph  $G$  –

1. A connected graph  $G$  can have more than one spanning tree.



2. All possible spanning trees of graph  $G$ , have the same number of edges and vertices.
3. The spanning tree does not have any cycle (loops).
4. Removing one edge from the spanning tree will make the graph disconnected, i.e. the spanning tree is minimally connected.
5. Adding one edge to the spanning tree will create a circuit or loop, i.e. the spanning tree is maximally acyclic.

### **Mathematical Properties of Spanning Tree**

1. Spanning tree has  $n-1$  edges, where  $n$  is the number of nodes (vertices).
2. From a complete graph, by removing maximum  $e - n + 1$  edges, we can construct a spanning tree.
3. A complete graph can have maximum  $n^{n-2}$  number of spanning trees.
4. Thus, we can conclude that spanning trees are a subset of connected Graph  $G$  and disconnected graphs do not have spanning tree.

### **Application of Spanning Tree**

Spanning tree is basically used to find a minimum path to connect all nodes in a graph. Common application of spanning trees are –

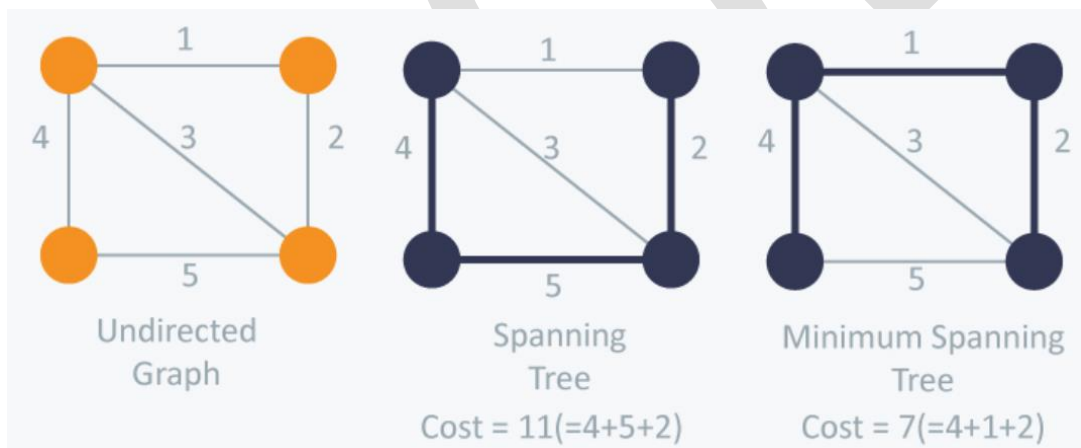
- Civil Network Planning
- Computer Network Routing Protocol
- Cluster Analysis

### 13.2.2 Minimum Spanning Tree

The cost of the spanning tree is the sum of the weights of all the edges in the tree. There can be many spanning trees. Minimum spanning tree is the spanning tree where the cost is minimum among all the spanning trees. There also can be many minimum spanning trees.

Minimum spanning tree has direct application in the design of networks. It is used in algorithms approximating the travelling salesman problem, multi-terminal minimum cut problem and minimum cost weighted perfect matching. Other practical applications are:

1. Cluster Analysis
2. Handwriting recognition
3. Image segmentation



There are two famous algorithms for finding the Minimum Spanning Tree:

1. Kruskal's Algorithm
2. Prim's Algorithm

These both algorithms are Greedy Algorithms.

## 13.3 Graph Algorithms

### 13.3.1 Kruskal's Algorithm

Kruskal's Algorithm builds the spanning tree by adding edges one by one into a growing spanning tree. Kruskal's algorithm follows greedy approach as in each iteration it finds an edge which has least weight and add it to the growing spanning tree.

#### Algorithm Steps:

1. Sort the graph edges with respect to their weights.
2. Start adding edges to the MST from the edge with the smallest weight until the edge of the largest weight.
3. Only add edges which doesn't form a cycle , edges which connect only disconnected components.

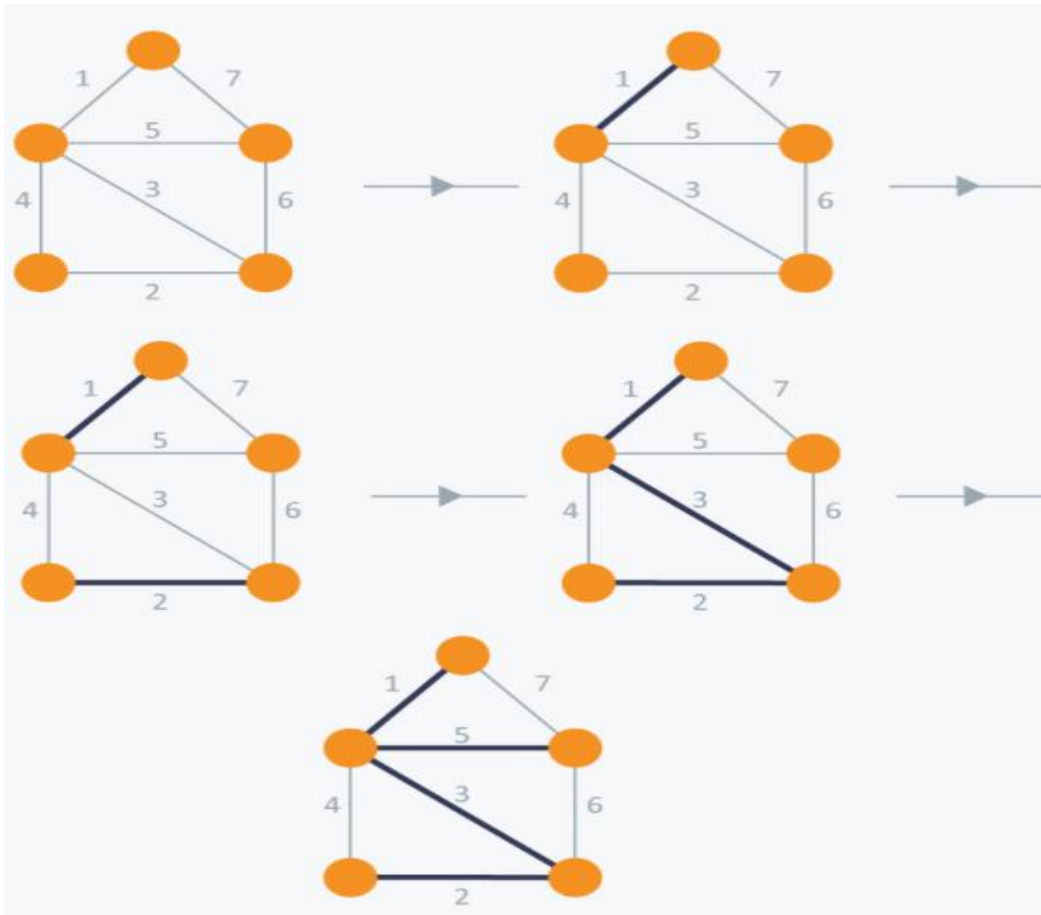
So now the question is how to check if 2 vertices are connected or not ?

This could be done using DFS which starts from the first vertex, then check if the second vertex is visited or not. But DFS will make time complexity large as it has an order of  $O(V+E)$  where V is the number of vertices, E is the number of edges. So the best solution is "**Disjoint Sets**".

#### DisjointSets:

Disjoint sets are sets whose intersection is the empty set so it means that they don't have any element in common.

Consider following example:



In Kruskal's algorithm, at each iteration we will select the edge with the lowest weight. So, we will start with the lowest weighted edge first i.e., the edges with weight 1. After that we will select the second lowest weighted edge i.e., edge with weight 2. Notice these two edges are totally disjoint. Now, the next edge will be the third lowest weighted edge i.e., edge with weight 3, which connects the two disjoint pieces of the graph. Now, we are not allowed to pick the edge with weight 4, that will create a cycle and we can't have any cycles. So we will select the fifth lowest weighted edge i.e., edge with weight 5. Now the other two edges will create cycles so we will ignore them. In the end, we end up with a minimum spanning tree with total cost 11 ( $= 1 + 2 + 3 + 5$ ).

**Program:**

```
#include <iostream>

#include <vector>

#include <utility>

#include <algorithm>
```

```
using namespace std;

const int MAX = 1e4 + 5;

int id[MAX], nodes, edges;

pair <long long, pair<int, int>> p[MAX];
```

```
void initialize()
```

```
{
for(int i = 0; i < MAX; ++i)
    id[i] = i;
}
```

```
int root(int x)
```

```
{
while(id[x] != x)
{
    id[x] = id[id[x]];
    x = id[x];
}
return x;
}
```

```
void union1(int x, int y)
```

```
{
    int p = root(x);
    int q = root(y);
    id[p] = id[q];
}
```

```
}
```

```
long longkruskal(pair<long long, pair<int, int>> p[])
```

```
{
```

```
    int x, y;
```

```
    long long cost, minimumCost = 0;
```

```
    for(int i = 0; i < edges; ++i)
```

```
    {
```

```
        // Selecting edges one by one in increasing order from the beginning
```

```
        x = p[i].second.first;
```

```
        y = p[i].second.second;
```

```
        cost = p[i].first;
```

```
        // Check if the selected edge is creating a cycle or not
```

```
        if(root(x) != root(y))
```

```
        {
```

```
            minimumCost += cost;
```

```
            union1(x, y);
```

```
        }
```

```
    }
```

```
    return minimumCost;
```

```
}
```

```
int main()
```

```
{
```

```
    int x, y;
```

```
    long long weight, cost, minimumCost;
```

```

initialize();

cin>> nodes >>edges;

for(int i = 0;i < edges;++i)
{
cin>> x >> y >>weight;

    p[i] = make_pair(weight, make_pair(x, y));

}

// Sort the edges in the ascending order

sort(p, p + edges);

minimumCost = kruskal(p);

cout<<minimumCost<<endl;

return 0;

}

```

**TimeComplexity:**

**Time Complexity:**

In Kruskal's algorithm, most time consuming operation is sorting because the total complexity of the Disjoint-Set operations will be  $O(E \log V)$ , which is the overall Time Complexity of the algorithm.

### ***13.3.2 Prim's Algorithm***

Prim's Algorithm also use Greedy approach to find the minimum spanning tree. In Prim's Algorithm we grow the spanning tree from a starting position. Unlike an edge in Kruskal's, we add vertex to the growing spanning tree in Prim's.

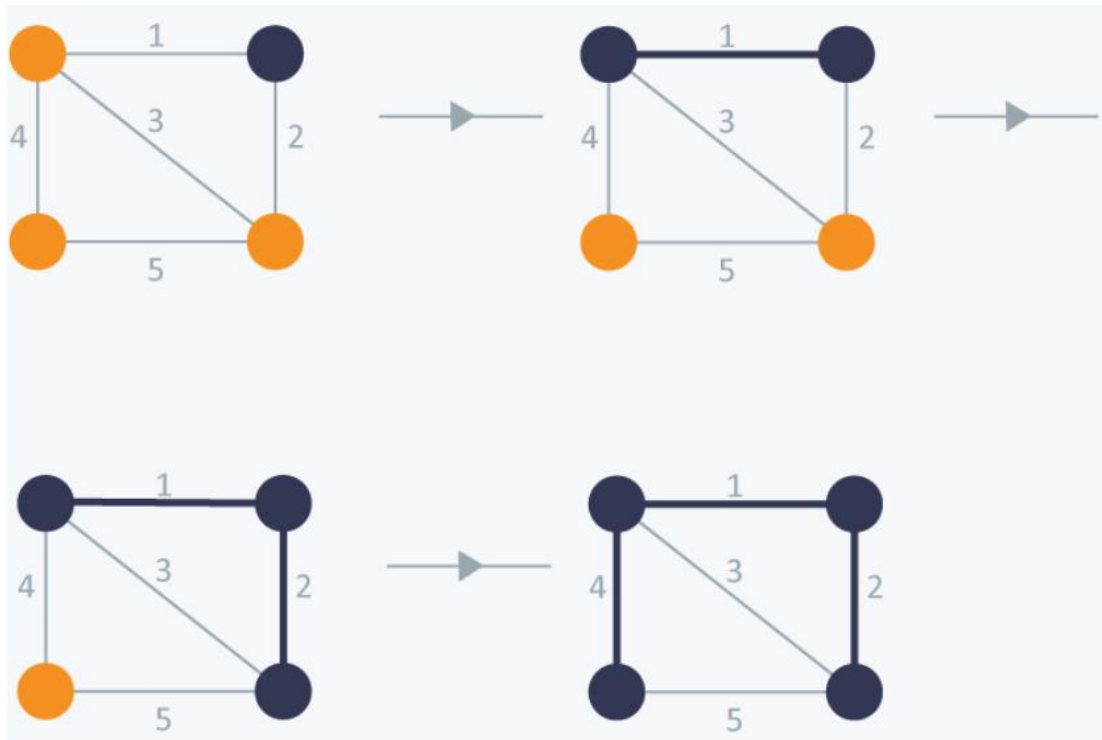
**Algorithm Steps:**

1. Maintain two disjoint sets of vertices. One containing vertices that are in the growing spanning tree and other that are not in the growing spanning tree.
2. Select the cheapest vertex that is connected to the growing spanning tree and is not in the growing spanning tree and add it into the growing spanning tree. This can be

done using Priority Queues. Insert the vertices, that are connected to growing spanning tree, into the Priority Queue.

3. Check for cycles. To do that, mark the nodes which have been already selected and insert only those nodes in the Priority Queue that are not marked.

Consider the example below:



In Prim's Algorithm, we will start with an arbitrary node (it doesn't matter which one) and mark it. In each iteration we will mark a new vertex that is adjacent to the one that we have already marked. As a greedy algorithm, Prim's algorithm will select the cheapest edge and mark the vertex. So we will simply choose the edge with weight 1. In the next iteration we have three options, edges with weight 2, 3 and 4. So, we will select the edge with weight 2 and mark the vertex. Now again we have three options, edges with weight 3, 4 and 5. But we can't choose edge with weight 3 as it is creating a cycle. So we will select the edge with weight 4 and we end up with the minimum spanning tree of total cost  $7 (= 1 + 2 + 4)$ .

**Program:**

```
#include <iostream>
```

```
#include <vector>
```

```
#include <queue>
```



```
#include <functional>

#include <utility>

using namespace std;

const int MAX = 1e4 + 5;

typedef pair<long long, int>Pll;

bool marked[MAX];

vector <Pll>adj[MAX];

long longprim(int x)
{
    priority_queue<Pll, vector<Pll>, greater<Pll>>Q;

    int y;

    long longminimumCost = 0;

    Pll p;

    Q.push(make_pair(0, x));

    while(!Q.empty())
    {
        // Select the edge with minimum weight

        p = Q.top();

    Q.pop();

        x = p.second;

        // Checking for cycle

        if(marked[x] == true)

            continue;

        minimumCost += p.first;
```

```
        marked[x] = true;
    for(int i = 0; i < adj[x].size(); ++i)
    {
        y = adj[x][i].second;
        if(marked[y] == false)
        Q.push(adj[x][i]);
    }
}
return minimumCost;
}

int main()
{
    int nodes, edges, x, y;
    long long weight, minimumCost;
    cin >> nodes >> edges;
    for(int i = 0; i < edges; ++i)
    {
        cin >> x >> y >> weight;
        adj[x].push_back(make_pair(weight, y));
        adj[y].push_back(make_pair(weight, x));
    }

    // Selecting 1 as the starting node
    minimumCost = prim(1);

    cout << minimumCost << endl;

    return 0;
}
```

}

#### **Time Complexity:**

The time complexity of the Prim's Algorithm is  $O((V+E)\log V)$  because each vertex is inserted in the priority queue only once and insertion in priority queue take logarithmic time.

### **13.3 Summary**

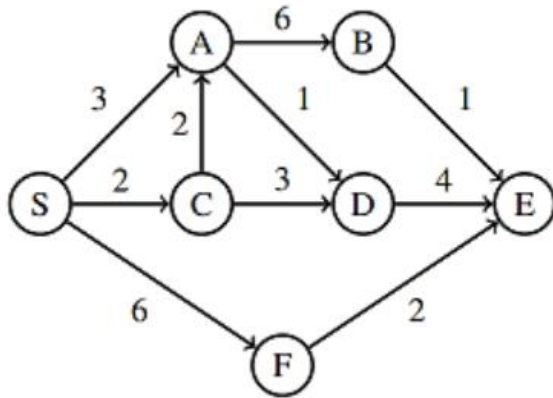
In this chapter, we have studied Minimum Spanning Tree (MST) and its use in various algorithms like Kruskal's Algorithm, Prim's Algorithm. We have also seen all-pair shortest path algorithms like Floyd Warshall's Algorithm, Dijkstra's Algorithm.

### **13.4 Review Your Learning**

1. Are you able to explain what is Minimum Spanning Tree (MST)?
2. Are you able to explain Greedy algorithms working on graph data structures?
3. Are you able to analyse the condition when to use Prim's and Kruskal's Algorithm?
4. Are you able to find shortest path using graph algorithms?
5. Can you explain the applications of Graph data structures in day-to-day life?

### **13.5 Questions**

1. Explain Minimum Spanning Tree (MST). Explain its use in various algorithms which works on graphs.
2. Explain applications of Graph data structures in day-to-day life?
3. Explain various graph algorithms to find shortest path using Greedy Approach?
4. Find shortest path using Prim's Algorithm on graph given below:



5. Explain difference between Greedy and Dynamic programming approach. Explain the examples of graph algorithms in each categories.

### 13.6 Further Reading

1. <https://runestone.academy/runestone/books/published/pythonds/Graphs/DijkstraAlgorithm.html>
2. [https://www.oreilly.com/library/view/data-structures-and/9781118771334/18\\_chap14.html](https://www.oreilly.com/library/view/data-structures-and/9781118771334/18_chap14.html)
3. <https://medium.com/javarevisited/10-best-books-for-data-structure-and-algorithms-for-beginners-in-java-c-c-and-python-5e3d9b478eb1>
4. <https://examradar.com/data-structure-graph-mcq-based-online-test-3/>
5. [https://www.tutorialspoint.com/data\\_structures\\_algorithms/data\\_structures\\_algorithms\\_online\\_quiz.htm](https://www.tutorialspoint.com/data_structures_algorithms/data_structures_algorithms_online_quiz.htm)

### 13.7 References

1. [http://www.nitjsr.ac.in/course\\_assignment/CS01CS1302A%20Book%20Fundamentals%20of%20Data%20Structure%20\(1982\)%20by%20Ellis%20Horowitz%20and%20Sartaj%20Sahni.pdf](http://www.nitjsr.ac.in/course_assignment/CS01CS1302A%20Book%20Fundamentals%20of%20Data%20Structure%20(1982)%20by%20Ellis%20Horowitz%20and%20Sartaj%20Sahni.pdf)
2. <https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/>
3. <https://www.geeksforgeeks.org/prims-mst-for-adjacency-list-representation-greedy-algo-6/>
4. <https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2/>
5. [http://wccclab.cs.nchu.edu.tw/www/images/Data\\_Structure\\_105/chapter6.pdf](http://wccclab.cs.nchu.edu.tw/www/images/Data_Structure_105/chapter6.pdf)

## Unit 6: Chapter 14

### Dynamic Graph Algorithms

#### Dynamic Graph Algorithms

14.0 Objectives .....	309
14.1 Introduction.....	310
14.2 Dynamic Graph Algorithms.....	311
14.2.1 Floyd Warshall’s Algorithm .....	313
14.2.2 Dijkstra’s Algorithms.....	320
14.3 Applications of Graphs .....	325
14.4 Summary .....	328
14.5 Review Your Learning.....	328
14.6 Questions.....	328
14.7 Further Reading .....	331
14.8 References.....	331

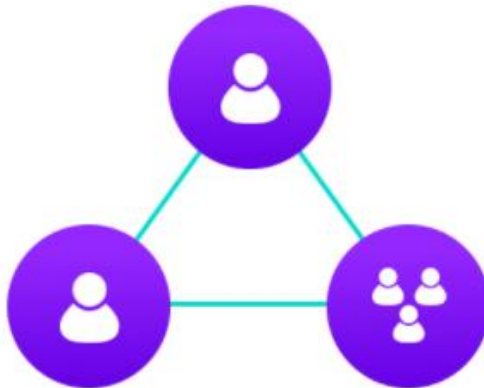
#### **14.0 Objectives**

10. Analyse working with Kruskal’s Algorithm, Prim’s Algorithm, Warshall’s Algorithm
11. Explain Shortest Path Algorithm using Dijkstra’s Algorithm
12. Explain difference between single source shortest path and all pair shortest path algorithms.
13. Explain difference between greedy and dynamic algorithm categories.
14. Analyse the applications of graphs in day-to-day life

## 14.1 Introduction

A Graph is a network of interconnected items. Each item is known as a node and the connection between them is known as the edge.

You probably use social media like Facebook, LinkedIn, Instagram, and so on. Social media is a great example of a graph being used. Social media uses graphs to store information about each user. Here, every user is a node just like in Graph. And, if one user, let's call him Jack, becomes friends with another user, Rose, then there exists an edge (connection) between Jack and Rose. Likewise, the more we are connected with people, the nodes and edges of the graph keep on increasing.



Similarly, Google Map is another example where Graphs are used. In the case of the Google Map, every location is considered as nodes, and roads between locations are considered as edges. And, when one has to move from one location to another, the Google Map uses various Graph-based algorithms to find the shortest path. We will discuss this later in this blog.



### **Need for Dynamic Graph Algorithms:**

The goal of a dynamic graph algorithm is to support query and update operations as quickly as possible (usually much faster than recomputing from scratch). Graphs subject to insertions only, or deletions only, but not both. Graphs subject to intermixed sequences of insertions and deletions.

## **14.2 Dynamic Graph Algorithms**

Let us say that we have a machine, and to determine its state at time  $t$ , we have certain quantities called state variables. There will be certain times when we have to make a decision which affects the state of the system, which may or may not be known to us in advance. These decisions or changes are equivalent to transformations of state variables. The results of the previous decisions help us in choosing the future ones.

What do we conclude from this? We need to break up a problem into a series of overlapping sub-problems and build up solutions to larger and larger sub-problems. If you are given a problem, which can be broken down into smaller sub-problems, and these smaller sub-problems can still be broken into smaller ones - and if you manage to find out that there are some overlapping sub-problems, then you've encountered a DP problem.

Some famous Dynamic Programming algorithms are:

- Unix diff for comparing two files
- Bellman-Ford for shortest path routing in networks
- TeX the ancestor of LaTeX
- WASP - Winning and Score Predictor

The core idea of Dynamic Programming is to avoid repeated work by remembering partial results and this concept finds its application in a lot of real life situations.

In programming, Dynamic Programming is a powerful technique that allows one to solve different types of problems in time  $O(n^2)$  or  $O(n^3)$  for which a naive approach would take exponential time.

## Dynamic Programming and Recursion:

Dynamic programming is basically, recursion plus using common sense. What it means is that recursion allows you to express the value of a function in terms of other values of that function. Where the common sense tells you that if you implement your function in a way that the recursive calls are done in advance, and stored for easy access, it will make your program faster. This is what we call Memorization - it is memorizing the results of some specific states, which can then be later accessed to solve other sub-problems.

The intuition behind dynamic programming is that we trade space for time, i.e. to say that instead of calculating all the states taking a lot of time but no space, we take up space to store the results of all the sub-problems to save time later.

Let's try to understand this by taking an example of Fibonacci numbers.

$$\text{Fibonacci}(n) = 1; \text{ if } n = 0$$

$$\text{Fibonacci}(n) = 1; \text{ if } n = 1$$

$$\text{Fibonacci}(n) = \text{Fibonacci}(n-1) + \text{Fibonacci}(n-2)$$

So, the first few numbers in this series will be: 1, 1, 2, 3, 5, 8, 13, 21... and so on!

Majority of the Dynamic Programming problems can be categorized into two types:

1. Optimization problems.
2. Combinatorial problems.

The optimization problems expect you to select a feasible solution, so that the value of the required function is minimized or maximized. Combinatorial problems expect you to figure out the number of ways to do something, or the probability of some event happening.

Every Dynamic Programming problem has a schema to be followed:

- Show that the problem can be broken down into optimal sub-problems.
- Recursively define the value of the solution by expressing it in terms of optimal solutions for smaller sub-problems.
- Compute the value of the optimal solution in bottom-up fashion.
- Construct an optimal solution from the computed information.



### **Bottom up vs. Top Down:**

**Bottom Up** - I'm going to learn programming. Then, I will start practicing. Then, I will start taking part in contests. Then, I'll practice even more and try to improve. After working hard like crazy, I'll be an amazing coder.

**Top Down** - I will be an amazing coder. How? I will work hard like crazy. How? I'll practice more and try to improve. How? I'll start taking part in contests. Then? I'll practicing. How? I'm going to learn programming.

### **Dynamic Programming in Graph Data Structures:**

Dynamic programming is “an algorithmic technique which is usually based on a recurrent formula and one (or some) starting states.” When it’s applied to graphs, we can solve for the shortest paths with one source or shortest paths for every pair.

**Examples:** Bellman-Ford Algorithm, Floyd Warshall’s Algorithm, Dijkstra’s Algorithm

#### ***14.2.1 Floyd Warshall’s Algorithm***

The Floyd Warshall Algorithm is for solving the All-Pairs Shortest Path problem. The problem is to find shortest distances between every pair of vertices in a given edge weighted directed Graph. Floyd Warshall Algorithm is an example of dynamic programming approach.

Floyd Warshall’s Algorithm has the following main advantages-

1. It is extremely simple.
2. It is easy to implement.

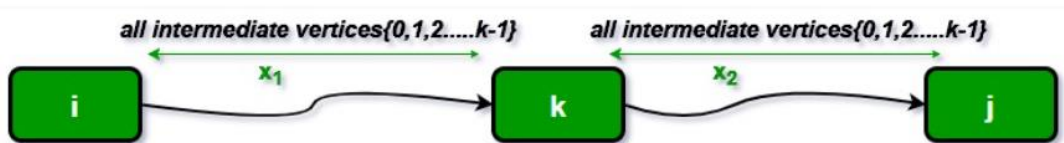
#### **Algorithm Steps:**

1. Initialize the solution matrix same as the input graph matrix as a first step.
2. Then update the solution matrix by considering all vertices as an intermediate vertex. The idea is to one by one pick all vertices and updates all shortest paths

which include the picked vertex as an intermediate vertex in the shortest path. When we pick vertex number  $k$  as an intermediate vertex, we already have considered vertices  $\{0, 1, 2, \dots, k-1\}$  as intermediate vertices. For every pair  $(i, j)$  of the source and destination vertices respectively, there are two possible cases.

- $k$  is not an intermediate vertex in shortest path from  $i$  to  $j$ . We keep the value of  $\text{dist}[i][j]$  as it is.
- $k$  is an intermediate vertex in shortest path from  $i$  to  $j$ . We update the value of  $\text{dist}[i][j]$  as  $\text{dist}[i][k] + \text{dist}[k][j]$  if  $\text{dist}[i][j] > \text{dist}[i][k] + \text{dist}[k][j]$

The following figure shows the above optimal substructure property in the all-pairs shortest path problem.



### Program:

```
// C++ Program for Floyd Warshall Algorithm
```

```
#include <bits/stdc++.h>
```

```
using namespace std;
```

```
// Number of vertices in the graph
```

```
#define V 4
```

```
/* Define Infinite as a large enough value.This value will be used for vertices not
connected to each other */
```

```
#define INF 99999
```

```
// A function to print the solution matrix
```

```
void printSolution(int dist[][V]);
```

```
// Solves the all-pairs shortest path problem using Floyd Warshall algorithm
```

```
void floydWarshall (int graph[][V])
```

```
{
```

```
    /* dist[][] will be the output matrix that will finally have the shortest distances
between every pair of vertices */
```

```
    int dist[V][V], i, j, k;
```

```
    /* Initialize the solution matrix same as input graph matrix. Or we can say the
initial values of shortest distances are based on shortest paths considering no
intermediate vertex. */
```

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

```
        for (j = 0; j < V; j++)
```

```
            dist[i][j] = graph[i][j];
```

```
    /* Add all vertices one by one to the set of intermediate vertices. ---> Before
start of an iteration, we have shortest distances between all pairs of vertices such that
the shortest distances consider only the vertices in set {0, 1, 2, .. k-1} as
intermediate vertices. ----> After the end of an iteration, vertex no. k is added to the
set of intermediate vertices and the set becomes {0, 1, 2, .. k} */
```

```
    for (k = 0; k < V; k++)
```

```

{
    // Pick all vertices as source one by one
    for (i = 0; i < V; i++)
    {
        // Pick all vertices as destination for the above picked source
        for (j = 0; j < V; j++)
        {
            // If vertex k is on the shortest path from i to j, then
            // value of dist[i][j]
            if (dist[i][k] + dist[k][j] < dist[i][j])
                dist[i][j] = dist[i][k] + dist[k][j];
        }
    }
}
// Print the shortest distance matrix
printSolution(dist);
}

```

/\* A utility function to print solution \*/

```
void printSolution(int dist[][V])
```

```
{
```

```
    cout<<"The following matrix shows the shortest distances"
```

```
        " between every pair of vertices \n";
```

```

for (int i = 0; i < V; i++)
{
    for (int j = 0; j < V; j++)
    {
        if (dist[i][j] == INF)
            cout << "INF" << " ";
        else
            cout << dist[i][j] << " ";
    }
    cout << endl;
}
}
// Driver code
int main()
{
    /* Let us create the weighted graph */
    int graph[V][V] = { {0, 5, INF, 10},
                        {INF, 0, 3, INF},
                        {INF, INF, 0, 1},
                        {INF, INF, INF, 0}
    };

    // Print the solution
    floydWarshall(graph);
    return 0;
}

```

}

### Time Complexity-

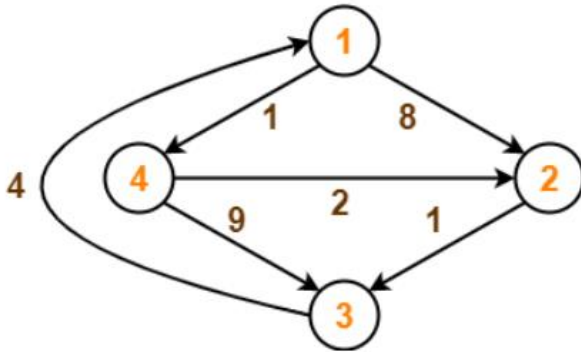
- Floyd Warshall Algorithm consists of three loops over all the nodes.
- The inner most loop consists of only constant complexity operations.
- Hence, the asymptotic complexity of Floyd Warshall algorithm is  $O(n^3)$ .
- Here,  $n$  is the number of nodes in the given graph.

### When Floyd Warshall Algorithm Is Used?

- Floyd Warshall Algorithm is best suited for dense graphs.
- This is because its complexity depends only on the number of vertices in the given graph.
- For sparse graphs, Johnson's Algorithm is more suitable.

Example-1:

Consider the following directed weighted graph. Using Floyd Warshall Algorithm, find the shortest path distance between every pair of vertices.



### Step-1:

- Remove all the self-loops and parallel edges (keeping the lowest weight edge) from the graph.
- In the given graph, there are neither self-edges nor parallel edges.

### Step-2:

Write the initial distance matrix.

- It represents the distance between every pair of vertices in the form of given weights.
- For diagonal elements (representing self-loops), distance value = 0.
- For vertices having a direct edge between them, distance value = weight of that edge.
- For vertices having no direct edge between them, distance value =  $\infty$ .

Initial distance matrix for the given graph is-

$$D_0 = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 8 & \infty & 1 \\ \infty & 0 & 1 & \infty \\ 4 & \infty & 0 & \infty \\ \infty & 2 & 9 & 0 \end{bmatrix} \end{matrix}$$

**Step-3:**

Using Floyd Warshall Algorithm, write the following 4 matrices-

$$D_1 = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 8 & \infty & 1 \\ \infty & 0 & 1 & \infty \\ 4 & 12 & 0 & 5 \\ \infty & 2 & 9 & 0 \end{bmatrix} \end{matrix}$$

$$D_3 = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 8 & 9 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 12 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix} \end{matrix}$$

$$D_2 = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 8 & 9 & 1 \\ \infty & 0 & 1 & \infty \\ 4 & 12 & 0 & 5 \\ \infty & 2 & 3 & 0 \end{bmatrix} \end{matrix}$$

$$D_4 = \begin{matrix} & \begin{matrix} 1 & 2 & 3 & 4 \end{matrix} \\ \begin{matrix} 1 \\ 2 \\ 3 \\ 4 \end{matrix} & \begin{bmatrix} 0 & 3 & 4 & 1 \\ 5 & 0 & 1 & 6 \\ 4 & 7 & 0 & 5 \\ 7 & 2 & 3 & 0 \end{bmatrix} \end{matrix}$$

Please note:

- In the above problem, there are 4 vertices in the given graph.
- So, there will be total 4 matrices of order 4 x 4 in the solution excluding the initial distance matrix.
- Diagonal elements of each matrix will always be 0.

The last matrix D4 represents the shortest path distance between every pair of vertices.

### ***14.2.2 Dijkstra's Algorithms***

Dijkstra's algorithm, published in 1959 and named after its creator Dutch computer scientist Edsger Dijkstra, can be applied on a weighted [graph](#). The graph can either be directed or undirected. One stipulation to using the algorithm is that the graph needs to have a nonnegative weight on every edge.

If you are given a directed or undirected weighted graph with  $n$  vertices and  $m$  edges. The weights of all edges are non-negative. You are also given a starting vertex  $s$ . To find the lengths of the shortest paths from a starting vertex  $s$  to all other vertices, and output the shortest paths themselves, we use Dijkstra's Algorithm.

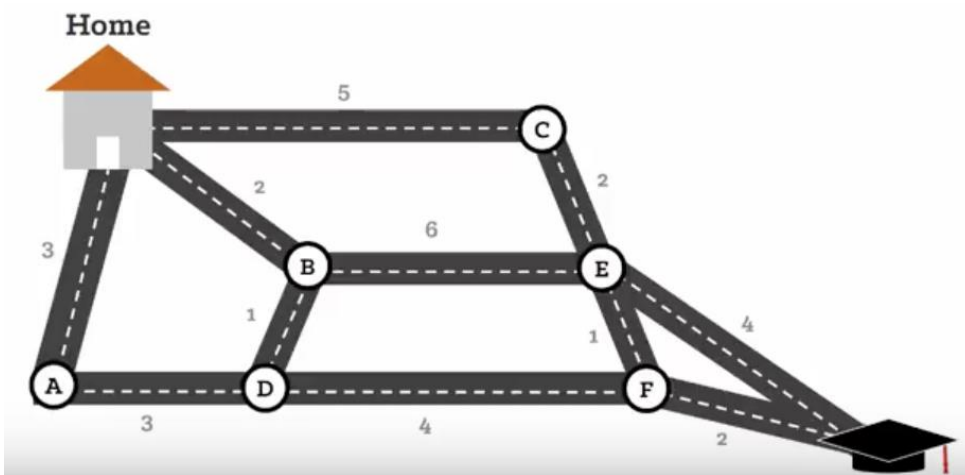
This problem is also called **single-source shortest paths problem**.

One algorithm for finding the shortest path from a starting node to a target node in a weighted graph is Dijkstra's algorithm. The [algorithm](#) creates a [tree](#) of shortest paths from the starting vertex, the source, to all other points in the graph.

Suppose a student wants to go from home to school in the shortest possible way. She knows some roads are heavily congested and difficult to use. In Dijkstra's algorithm, this means the edge has a large weight--the shortest path tree found by the algorithm will try to avoid edges with larger weights. If the student looks up directions using a map service, it is likely they may use Dijkstra's algorithm, as well as others.

**Example:** Find the shortest path from home to school in the following graph:





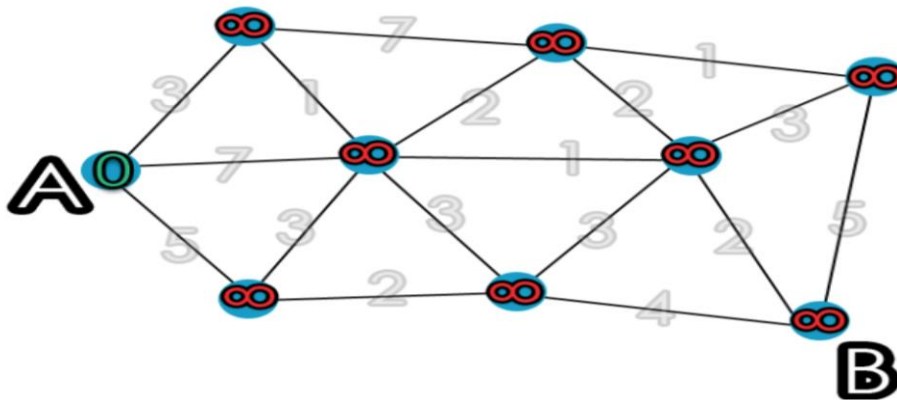
The shortest path, which could be found using Dijkstra's algorithm, is

Home → B → D → F → School

Algorithm:

Dijkstra's algorithm finds a shortest path tree from a single source node, by building a set of nodes that have minimum distance from the source.

The graph has the following:



- Vertices, or nodes, denoted in the algorithm by  $v_v$  or  $u_u$ ;
- Weighted edges that connect two nodes:  $(u, v)$  denotes an edge, and  $w(u, v)$  denotes its weight. In the diagram on the right, the weight for each edge is written in gray.

This is done by initializing three values:

- *dist*, an array of distances from the source node *ss* to each node in the graph, initialized the following way:  $dist(s) = 0$ ; and for all other nodes *v*,  $dist(v) = \infty$ . This is done at the beginning because as the algorithm proceeds, the *dist* from the source to each node *v* in the graph will be recalculated and finalized when the shortest distance to *v* is found
- *Q*, a queue of all nodes in the graph. At the end of the algorithm's progress, *Q* will be empty.
- *S*, an empty set, to indicate which nodes the algorithm has visited. At the end of the algorithm's run, *S* will contain all the nodes of the graph.

The algorithm proceeds as follows:

1. While *Q* is not empty, pop the node *v*, that is not already in *SS*, from *Q* with the smallest *dist* (*v*). In the first run, source node *ss* will be chosen because *dist*(*s*) was initialized to 0. In the next run, the next node with the smallest *dist* value is chosen.
2. Add node *v* to *S*, to indicate that *v* has been visited
3. Update *dist* values of adjacent nodes of the current node *v* as follows: for each new adjacent node *u*,
4. if  $dist(v) + \text{weight}(u,v) < dist(u)$ , there is a new minimal distance found for *u*, so update *dist*(*u*) to the new minimal distance value;
5. otherwise, no updates are made to *dist*(*u*).

The algorithm has visited all nodes in the graph and found the smallest distance to each node. *dist* now contains the shortest path tree from source *s*.

*Note:* The weight of an edge (*u,v*) is taken from the value associated with (*u,v*) on the graph.

### **Program:**

function Dijkstra(Graph, source):

```
dist[source] := 0           // Distance from source to source is set to 0
```

```

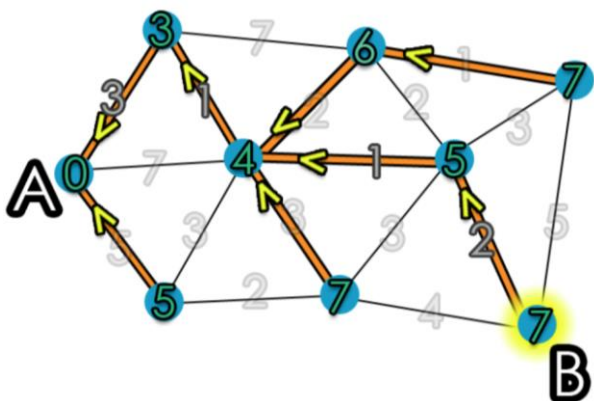
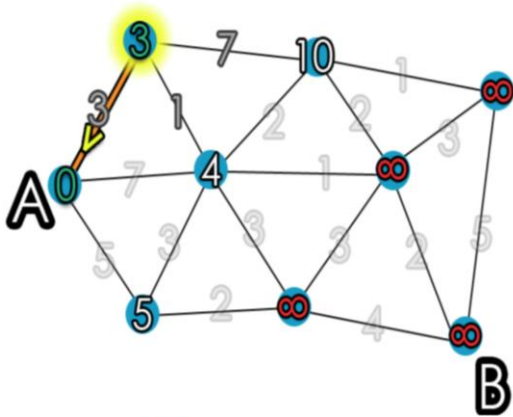
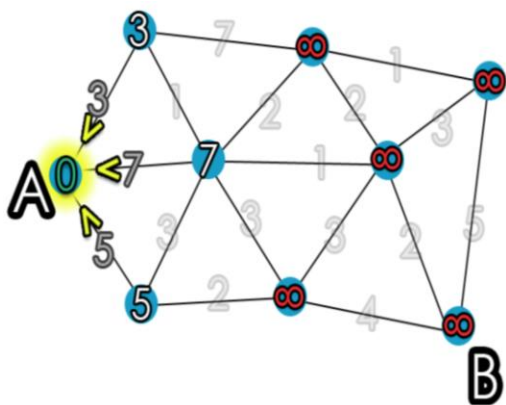
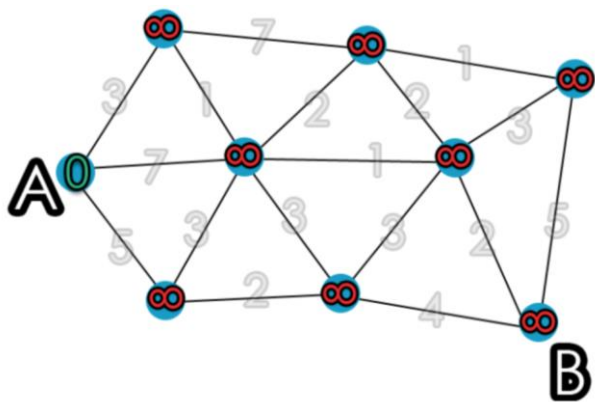
for each vertex v in Graph:      // Initializations
    if v ≠ source
dist[v] := infinity           // Unknown distance function from source to each node set
                                // to infinity
    add v to Q                 // All nodes initially in Q
while Q is not empty:         // The main loop
v := vertex in Q with min dist[v] // In the first run-through, this vertex is the source
node
    remove v from Q
    for each neighbor u of v:   // where neighbor u has not yet been removed
from Q.
alt := dist[v] + length(v, u)
    if alt < dist[u]:          // A shorter path to u has been found
        dist[u] := alt        // Update distance of u
return dist[]
end function

```

### Example:

Let us solve above example using following steps through Dijkstra's algorithm:

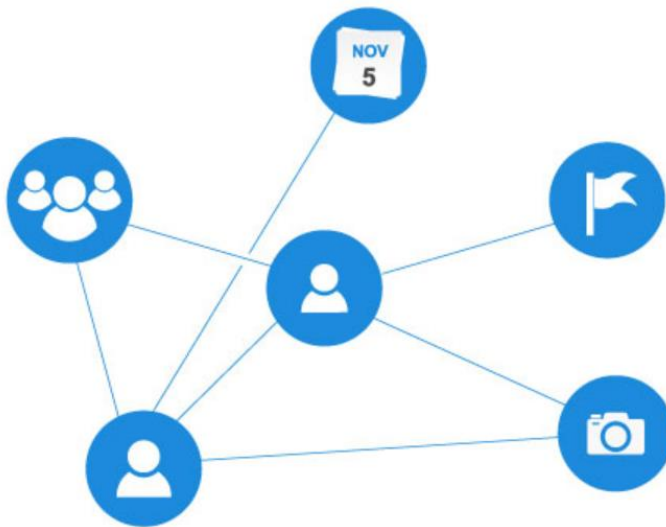
2. Initialize distances according to the algorithm
3. Pick first node and calculate distances to adjacent nodes.
4. Pick next node with minimal distance; repeat adjacent node distance calculations.
5. Final result of shortest-path tree



## 14.3 Applications of Graphs

1. In Computer science graphs are used to represent the flow of computation.
2. Google maps uses graphs for building transportation systems, where intersection of two (or more) roads are considered to be a vertex and the road connecting two vertices is considered to be an edge, thus their navigation system is based on the algorithm to calculate the shortest path between two vertices.
  - Google Maps and Routes APIs are classic Shortest Path APIs. This a graph problem that's very easy to solve with edge-weighted directed graphs (digraphs).
  - The idea of a Map API is to find the shortest path from one vertex to every other as in a single source shortest path variant, from your current location to every other destination you might be interested in going to on the map.
  - The idea of by contrast Routing API to find the shortest path from one vertex to another as in a source sink shortest path variant, from s to t.
  - Shortest Path APIs are typically directed graphs. The underlying data structures and graphs too.
3. In Facebook, users are considered to be the vertices and if they are friends then there is an edge running between them. Facebook's Friend suggestion algorithm uses graph theory. Facebook is an example of undirected graph.
  - Facebook's Graph API is perhaps the best example of application of graphs to real life problems. The Graph API is a revolution in large-scale data provision.
  - On The Graph API, everything is a vertice or node. This are entities such as Users, Pages, Places, Groups, Comments, Photos, Photo Albums, Stories, Videos, Notes, Events and so forth. Anything that has properties that store data is a vertice.

- And every connection or relationship is an edge. This will be something like a User posting a Photo, Video or Comment etc., a User updating their profile with their Place of birth, a relationship status Users, a User liking a Friend's Photo etc.
- The Graph API uses this collection of vertices and edges (essentially graph data structures) to store its data. The Graph API is also a GraphQL API. This is the language it uses to build and query the schema.
- The Graph API has come into some problems because of it's ability to obtain unusually rich info about user's friends.



4. In World Wide Web, web pages are considered to be the vertices. There is an edge from a page  $u$  to other page  $v$  if there is a link of page  $v$  on page  $u$ . This is an example of Directed graph. It was the basic idea behind Google Page Ranking Algorithm.
5. In Operating System, we come across the Resource Allocation Graph where each process and resources are considered to be vertices. Edges are drawn from resources to the allocated process, or from requesting process to the requested resource. If this leads to any formation of a cycle, then a deadlock will occur.

6. Google Knowledge Graph: knowledge graph has something to do with linking data and graphs...graph-based representation of knowledge. It still isn't what is can and can't do yet.
7. Path Optimization Algorithms: Path optimizations are primarily occupied with finding the best connection that fits some predefined criteria e.g. speed, safety, fuel etc or set of criteria e.g. procedures, routes.
  - In unweighted graphs, the Shortest Path of a graph is the path with the least number of edges. Breadth First Search (BFS) is used to find the shortest paths in graphs—we always reach a node from another node in the fewest number of edges in breadth graph traversals.
  - Any Spanning Tree is a Minimum Spanning Tree unweighted graphs using either BFS or Depth First Search.
8. Flight Networks: For flight networks, efficient route optimizations perfectly fit graph data structures. Using graph models, airport procedures can be modeled and optimized efficiently. Computing best connections in flight networks is a key application of algorithm engineering. In flight network, graph data structures are used to compute shortest paths and fuel usage in route planning, often in a multi-modal context. The vertices in flight networks are places of departure and destination, airports, aircrafts, cargo weights. The flight trajectories between airports are the edges. Turns out it's very feasible to fit graph data structures in route optimizations because of precompiled full distance tables between all airports. Entities such as flights can have properties such as fuel usage, crew pairing which can themselves be more graphs.
9. GPS Navigation Systems: Car navigations also use Shortest Path APIs. Although this is still a type of a routing API it would differ from the Google Maps Routing API because it is single source (from one vertex to every other i.e. it computes locations from where you are to any other location you might be interested in going.) BFS is used to find all neighbouring locations.

## 14.4 Summary

In previous chapter, we have studied Minimum Spanning Tree (MST) and its use in various algorithms like Kruskal's Algorithm, Prim's Algorithm. Based on this we have studied all-pair shortest path algorithms like Floyd Warshall's Algorithm, Dijkstra's Algorithm. We have studied the applications of various graph algorithms and data structures in day-to-day life.

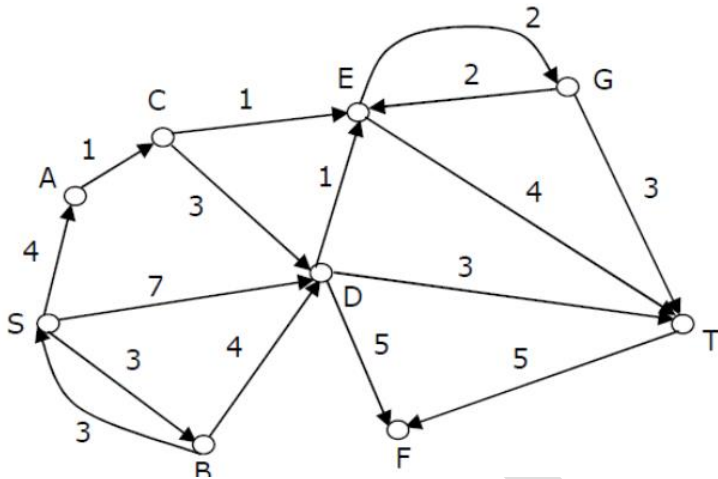
## 14.5 Review Your Learning

1. Are you able to find shortest path using Dijkstra's algorithm?
2. Are you able to find distance matrix and use it in Floyd Warshall's algorithm?
3. Explain the difference between Dynamic and Greedy programming approach. Give examples of shortest path algorithms in both categories.
4. Explain applications of all-pair shortest path algorithms.
5. Analyse how graph data structure is used in Google maps navigations.
6. Write a case study on Facebook friends list creation and recommendation techniques used in Facebook for finding close related friends/relatives.

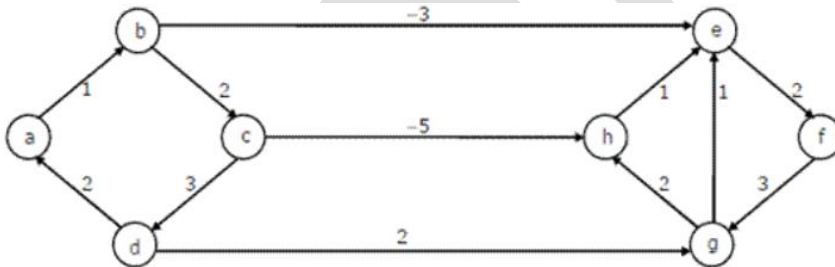
## 14.6 Questions

6. Consider the directed graph shown in the figure below. There are multiple shortest paths between vertices S and T. Which one will be reported by Dijkstra's shortest path algorithm? Assume that, in any iteration, the shortest path to a vertex  $v$  is updated only when a strictly shorter path to  $v$  is discovered.

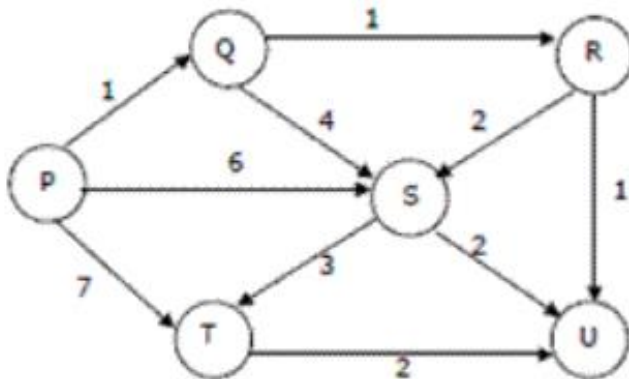




7. Dijkstra's single source shortest path algorithm when run from vertex a in the below graph, computes the correct shortest path distance to??

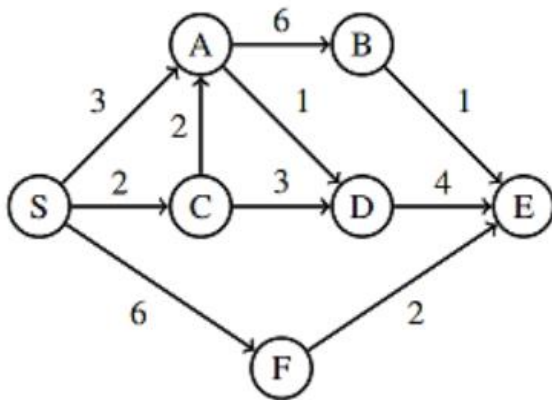


8. Suppose we run Dijkstra's single source shortest-path algorithm on the following edge weighted directed graph with vertex P as the source. In what order do the nodes get included into the set of vertices for which the shortest path distances are finalized?

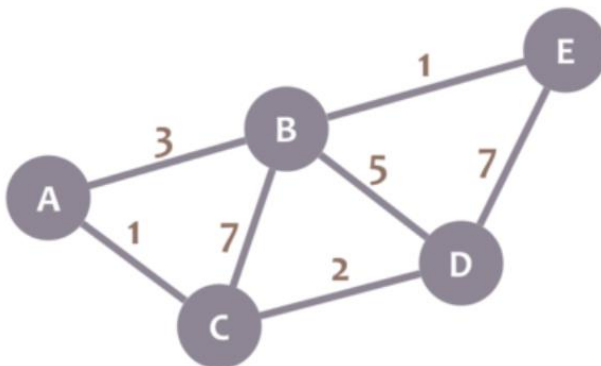


9. Dijkstra's Algorithm is used to solve \_\_\_\_\_ problems.
- a) All pair shortest path
  - b) Single source shortest path

- c) Network flow
  - d) Sorting
10. Dijkstra's Algorithm cannot be applied on \_\_\_\_\_
- a) Directed and weighted graphs
  - b) Graphs having negative weight function
  - c) Unweighted graphs
  - d) Undirected and unweighted graphs
11. Run Dijkstra's on the following graph and determine the resulting shortest path tree.



12. Find Shortest Path using Dijkstra's Algorithm for following graph.



13. Enlist and explain all-pair shortest path algorithms.
14. Explain Warshall's Algorithm with example.
15. Explain application of Floyd Warshall's and Dijkstra's Algorithm in day-to-day life.

## 14.7 Further Reading

6. <https://runestone.academy/runestone/books/published/pythonds/Graphs/DijkstraAlgorithm.html>
7. [https://www.oreilly.com/library/view/data-structures-and/9781118771334/18\\_chap14.html](https://www.oreilly.com/library/view/data-structures-and/9781118771334/18_chap14.html)
8. <https://medium.com/javarevisited/10-best-books-for-data-structure-and-algorithms-for-beginners-in-java-c-c-and-python-5e3d9b478eb1>
9. <https://examradar.com/data-structure-graph-mcq-based-online-test-3/>
10. [https://www.tutorialspoint.com/data\\_structures\\_algorithms/data\\_structures\\_algorithms\\_online\\_quiz.htm](https://www.tutorialspoint.com/data_structures_algorithms/data_structures_algorithms_online_quiz.htm)
11. <https://www.mdpi.com/2227-7390/8/9/1595/htm>

## 14.8 References

1. [http://www.nitjsr.ac.in/course\\_assignment/CS01CS1302A%20Book%20Fundamentals%20of%20Data%20Structure%20\(1982\)%20by%20Ellis%20Horowitz%20and%20Sartaj%20Sahni.pdf](http://www.nitjsr.ac.in/course_assignment/CS01CS1302A%20Book%20Fundamentals%20of%20Data%20Structure%20(1982)%20by%20Ellis%20Horowitz%20and%20Sartaj%20Sahni.pdf)
2. <https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/>
3. <https://www.geeksforgeeks.org/prims-mst-for-adjacency-list-representation-greedy-algo-6/>
4. <https://www.geeksforgeeks.org/kruskals-minimum-spanning-tree-algorithm-greedy-algo-2/>
5. [http://wccclab.cs.nchu.edu.tw/www/images/Data\\_Structure\\_105/chapter6.pdf](http://wccclab.cs.nchu.edu.tw/www/images/Data_Structure_105/chapter6.pdf)