

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 (Ω)
 - 1.10.2.3 Big-Theta Notation (Θ)
- 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.

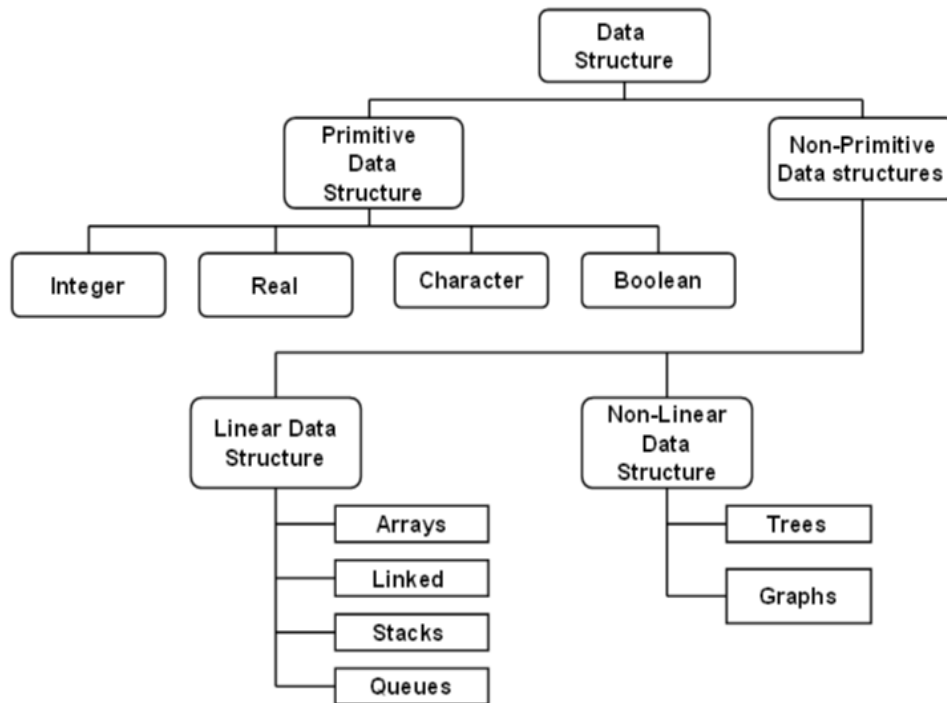


Figure 1.1 Classifications of data structures.

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.

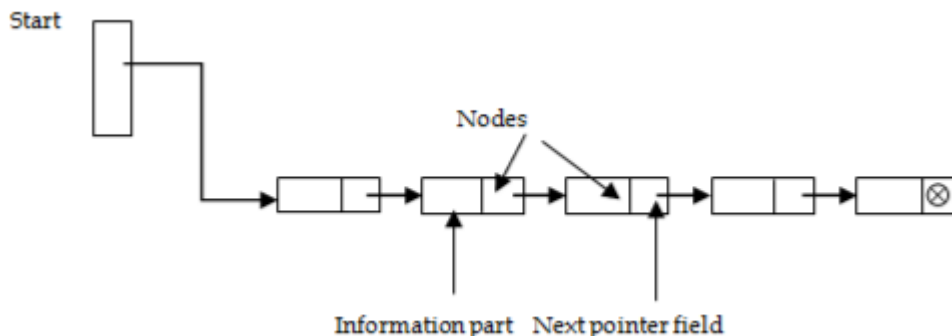


Figure 1.3: A Linked List

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.

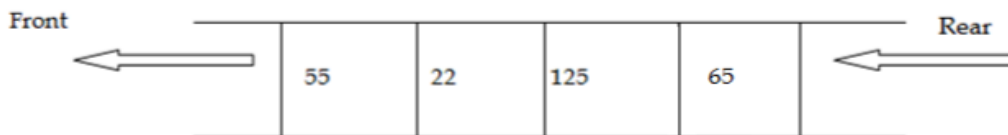


Figure 1.5: A Queue

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.

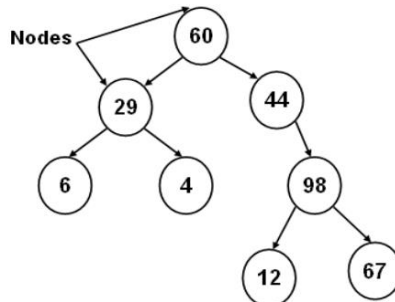


Figure 1.6: A Tree

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].

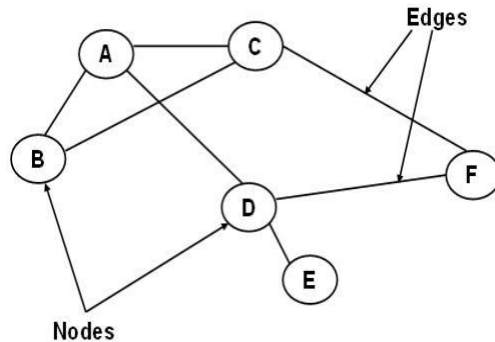


Figure 1.7 Graph

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

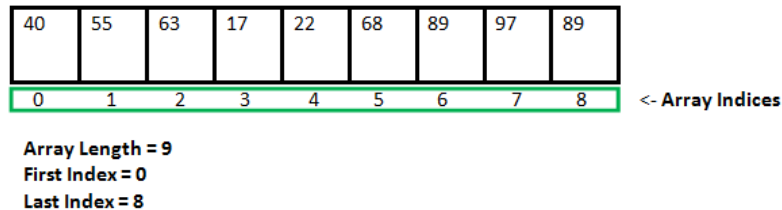
Data structure is a way of storing and organising data efficiently such that the required operations on them can be performed be efficient with respect to time as well as memory. Simply, Data Structure are used to reduce complexity (mostly the time complexity) of the code.

Data structures can be two types:

1. Static Data Structure
2. Dynamic Data Structure

What is a Static Data structure?

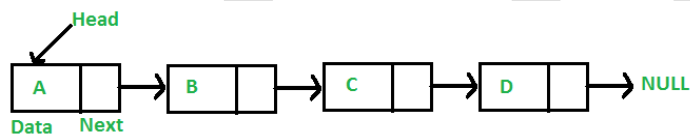
In Static data structure the size of the structure is fixed. The content of the data structure can be modified but without changing the memory space allocated to it.



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 – $c \leftarrow a + b$

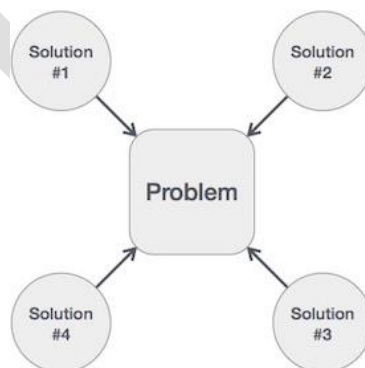
Step 4 – display **c**

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), Ω (Omega), and θ (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	$O(n^2)$	Quadratic	Shortest path between two nodes in a graph
6	$O(n^3)$	Cubic	Simultaneous linear equations
7	$O(2^n)$	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 **n₀** 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.

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

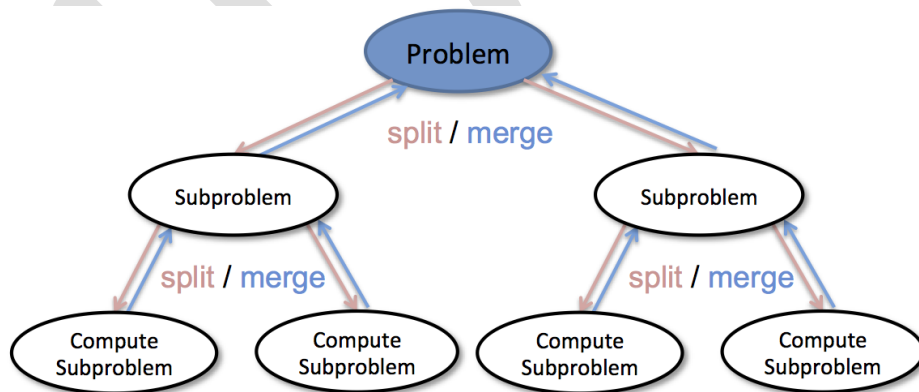
Divide and Conquer approach basically works on breaking the problem into sub problems that are similar to the original problem but smaller in size & simpler to solve. once divided sub problems are solved recursively and then combine solutions of sub problems to create a solution to original problem.

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."

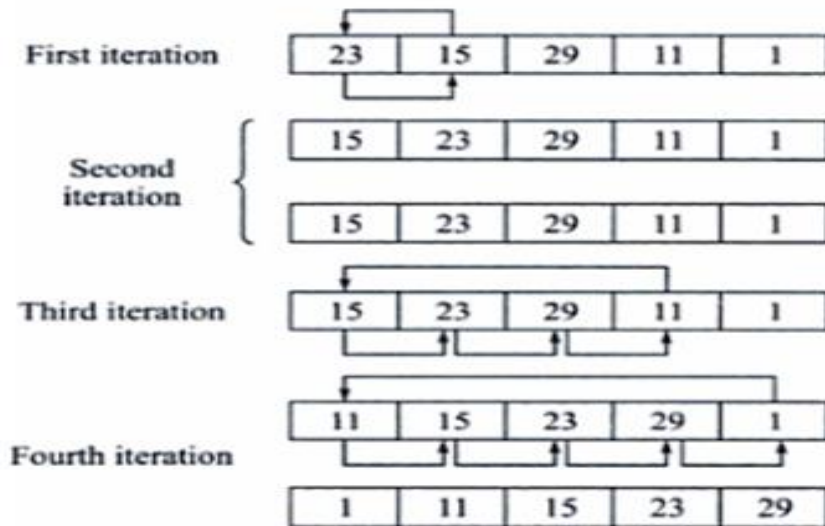
In the following Figure:

- 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



rd, with the





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 $TEMP \leq ARR[J]$	
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 tile 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:

$(n-1) + (n-2) + \dots + 3 + 2 + 1 = (n-1) * n/2$, 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 6 1 8 4 5	1 --- 6 3 8 4 5	1 3 --- 6 8 4 5	1 3 4 --- 8 6 5	1 3 4 5 6 8	1 3 4 5 6 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;
    }
}
```

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

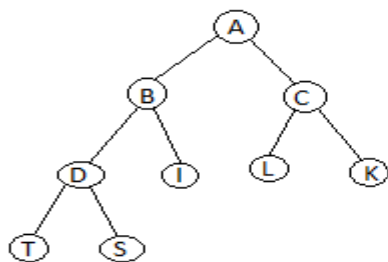
A) Terms in Heap:

a) Heap:

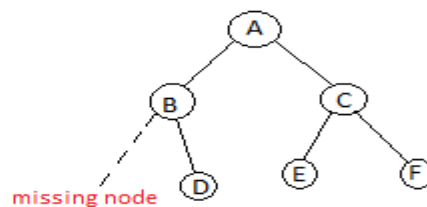
Heap is a special tree-based data structure that satisfies the following special heap properties

b) Shape Property:

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



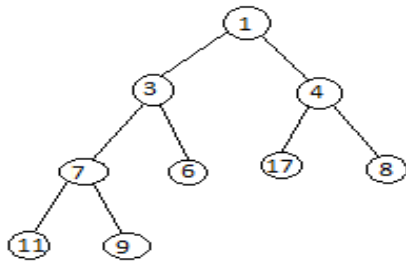
Complete Binary Tree



In-Complete Binary Tree

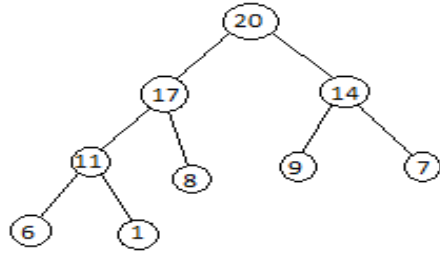
c) Heap Property:

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.



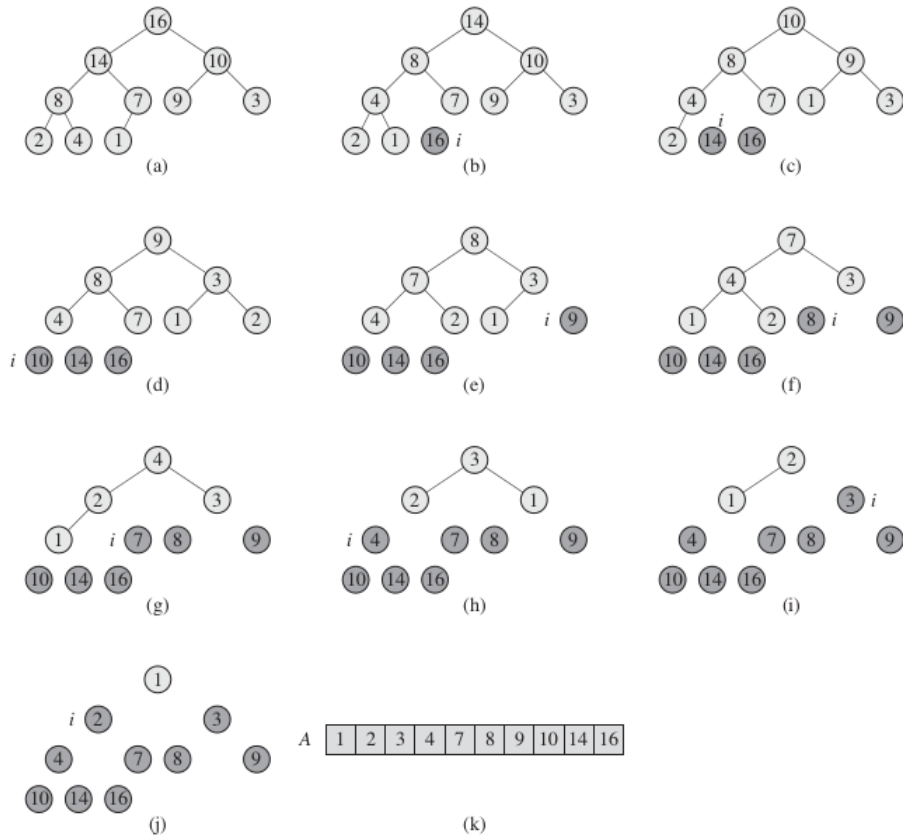
Max-Heap

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 $TREE[RIGHT] \leq TREE[LEFT]$ | |
| 7. $LEFT$ 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]. | |


```
    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;
```

```
while(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.

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.

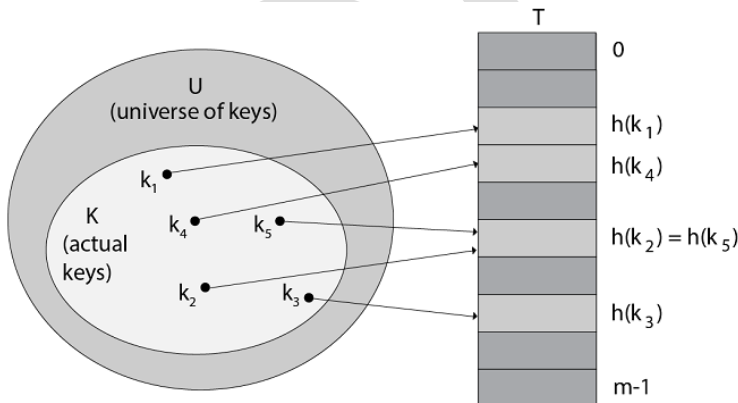


Figure: using a hash function h to map keys to hash-table slots. Because keys K_2 and k_5 map to the same slot, they collide.

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

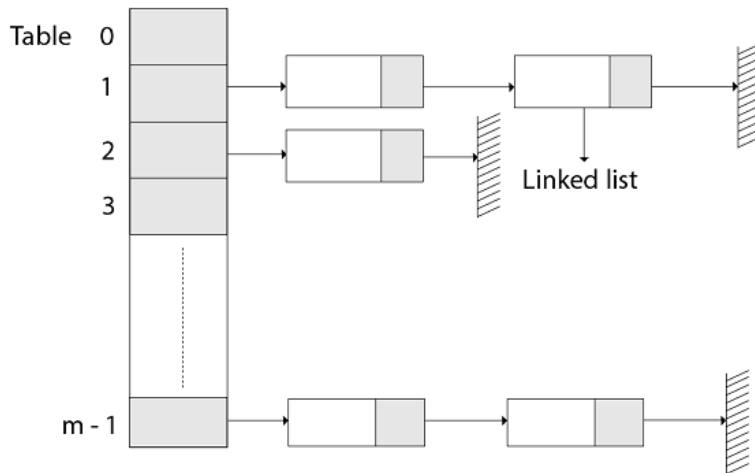
There are two main methods used to implement hashing:

- 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 α 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.

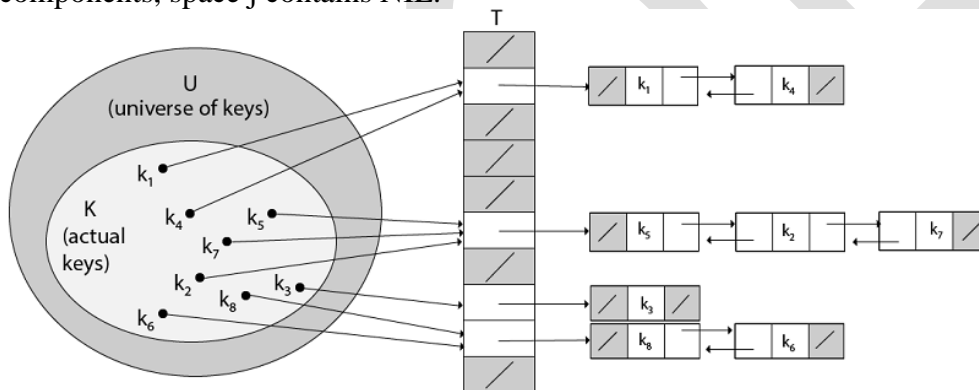


Fig: Collision resolution by chaining.

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 α 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.

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 (intval,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 at Kivity 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 component 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();
```

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;
```

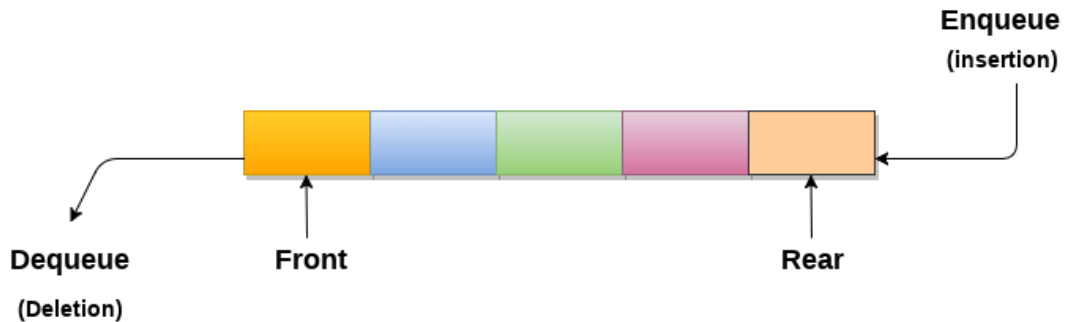


```

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;

```

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'.

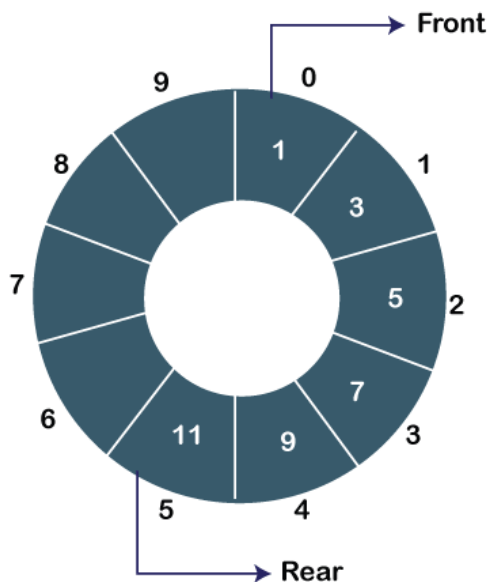
What is the Queue?

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.

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:

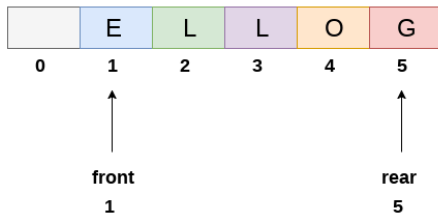


The disadvantage that happens in a direct line is overwhelmed by utilizing the roundabout queue. On the off chance that the unfilled space is accessible in a round line, the new component can be included a vacant space by just augmenting the estimation of back.

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.



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 $\text{REAR} = \text{MAX} - 1$

Write OVERFLOW

Go to step

[END OF IF]

Step 2: IF $\text{FRONT} = -1$ and $\text{REAR} = -1$

SET $\text{FRONT} = \text{REAR} = 0$

ELSE

SET $\text{REAR} = \text{REAR} + 1$

[END OF IF]

Step 3: Set $\text{QUEUE}[\text{REAR}] = \text{NUM}$

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)
    {

```

```
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;
    }
```

- 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.

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 `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);
}
}

```

```

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 ()

```

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*****

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

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.

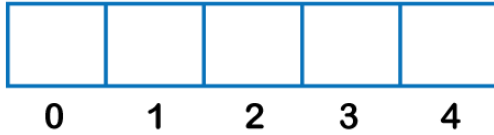
On the off chance that there is just a single component left which is to be erased, at that point the front and back are reset to - 1.

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]

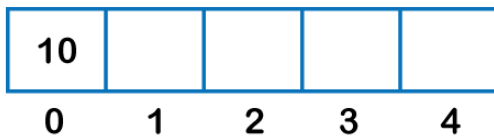
Step 4: EXIT

Let's understand the enqueue and dequeue operation through the diagrammatic representation.



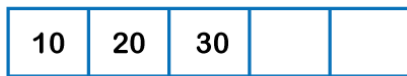
Front = -1

Rear = -1

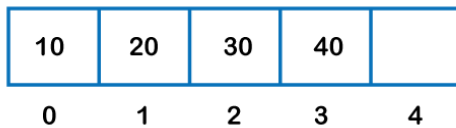


Front = 0

Rear = 0



Front = 0 **Rear = 2**



Front = 0 **Rear = 3**

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);
}
}
```

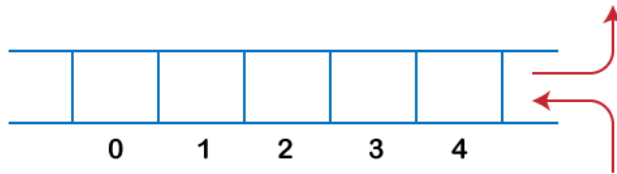
// function to get the front of the queue

```
int peek()
{
if((front==-1) &&(rear==-1))
{
printf("\nQueue is empty");
}
else
{
printf("\nThe front element is %d", front->data);
}
}
```

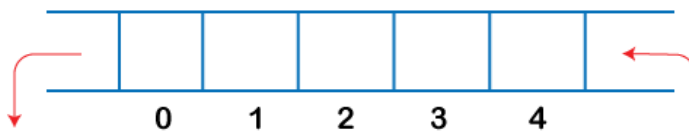
// function to display all the elements of the queue

```
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);
}
```



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.

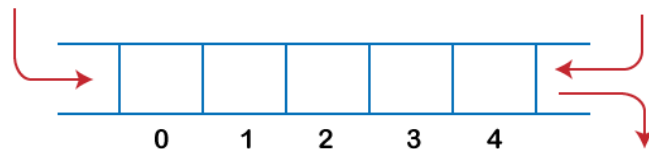


There are two types of Queues, Input-restricted queue, and output-restricted queue.

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.

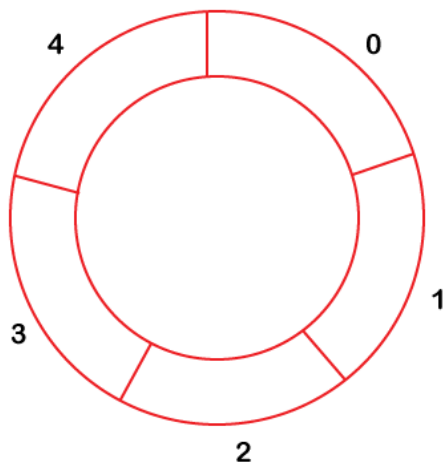
isEmpty(): This capacity restores a genuine worth if the stack is vacant; else it restores a bogus worth.

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.

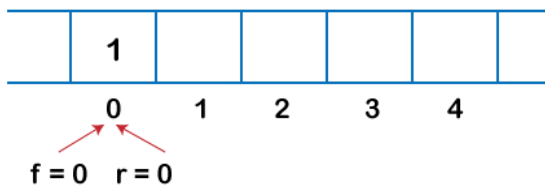
- It tends to be utilized for multiprocessor planning. Assume we have two processors, and every processor has one interaction to execute. Every processor is appointed with an interaction or a task, and each cycle contains numerous strings. Every processor keeps a deque that contains strings that are 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

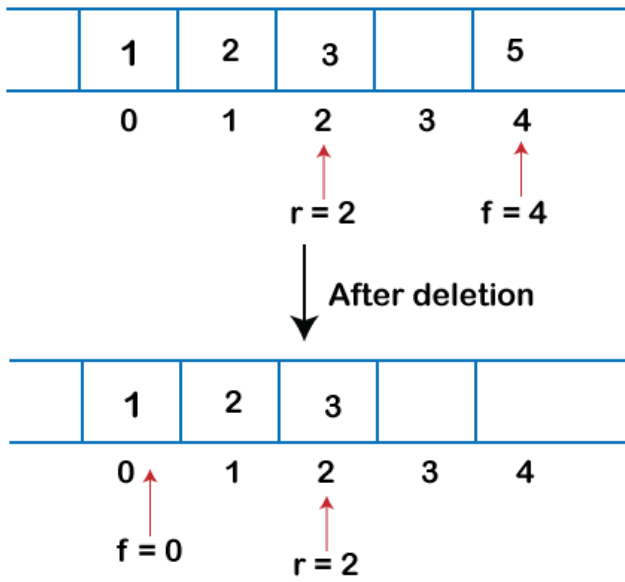
The following are the steps to perform the operations on the Deque:

Enqueue operation

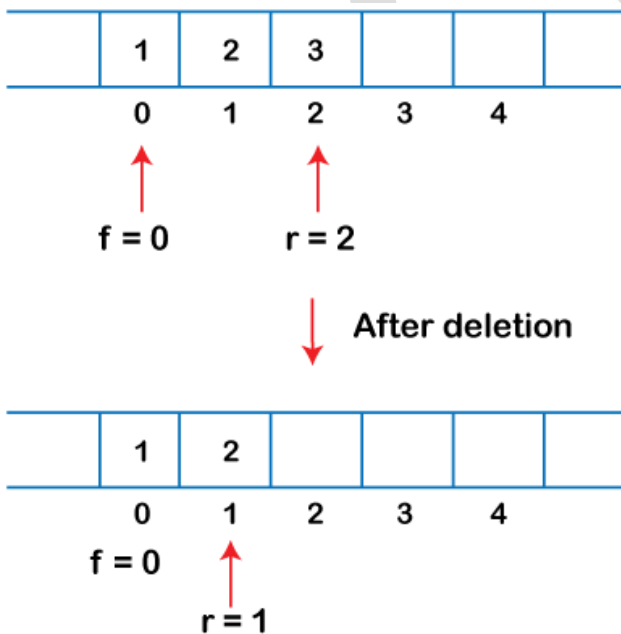
1. At first, we are thinking about that the deque is unfilled, so both front and back are set to - 1, i.e., $f = - 1$ and $r = - 1$.
2. As the deque is vacant, so embeddings a component either from the front or backside would be something very similar. Assume we have embedded component 1, at that point 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.



2. If we want to delete the element from rear end then we need to decrement the rear value by 1, i.e., $\text{rear} = \text{rear} - 1$ 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 $\text{rear} = n - 1$ where n is the size of the exhibit as demonstrated in the beneath figure:

// 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]);
    }
}
```

```

}
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;
}
}

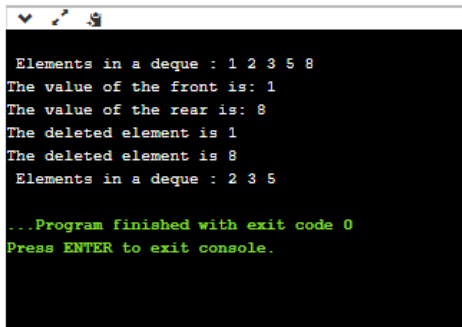
```

```

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.

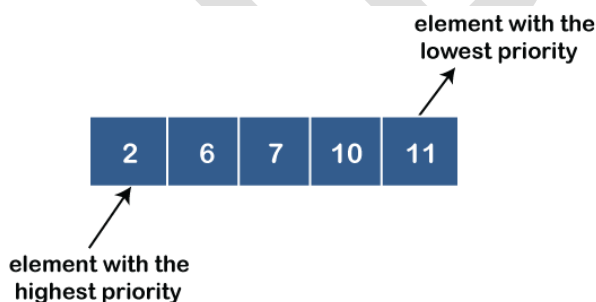
poll() It will eliminate '2' component from the need line as it has the most elevated need line.

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

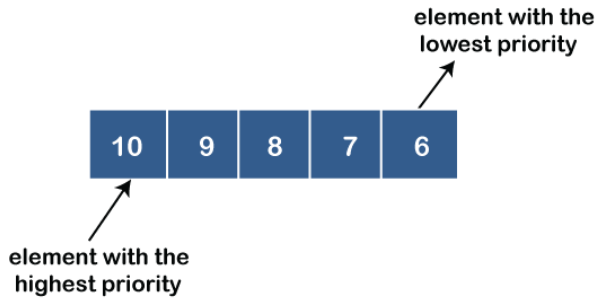
There are two types of priority queue:

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.

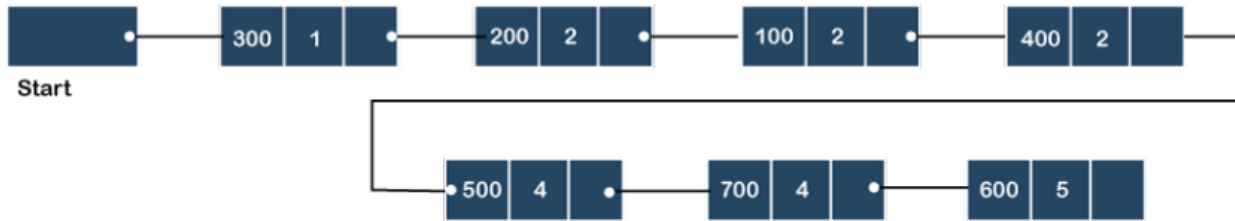
In the case of priority queue, lower priority number is considered the higher priority, i.e., lower priority number = higher priority.

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:

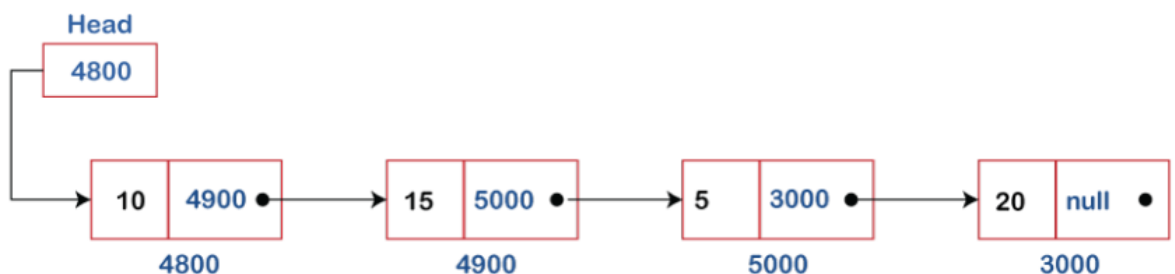
- **To understand the concept of Linked List**
- **To understand Types of Linked List**
- **To Singly Linked list**
- **To Doubly Linked list**

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.

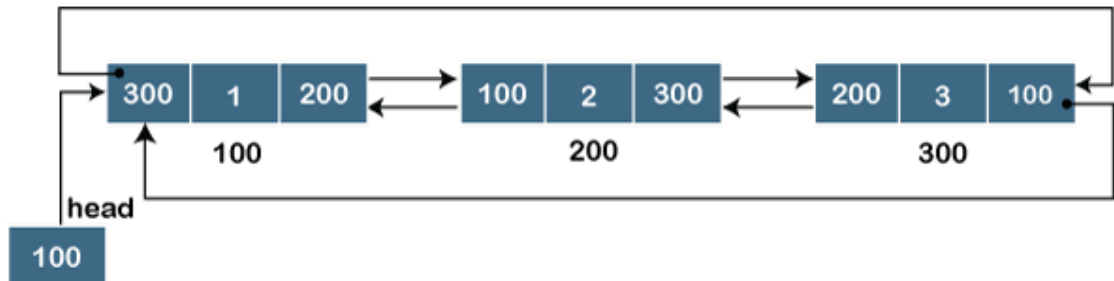
Implementation

Both the stacks and queues can be implemented using a linked list.

Disadvantages of Linked list

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.



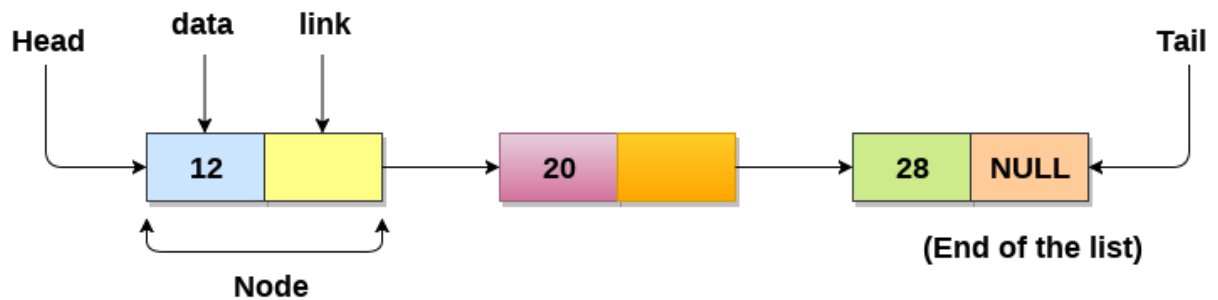
The above figure shows the portrayal of the doubly round connected rundown wherein the last hub is appended to the principal hub and consequently makes a circle. It is a doubly connected rundown likewise in light of the fact that every hub holds the location of the past hub too. The primary distinction 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.

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.

```
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();
```

```
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");

}
}
```

```

else
{
ptr = head;
while(ptr->next != NULL)
{
ptr1 = ptr;
ptr = ptr ->next;
}
ptr1->next = NULL;
free(ptr);
printf("\nDeleted Node from the last ...\n");
}
}

void random_delete()
{
struct node *ptr,*ptr1;
int loc,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;
}
}

```

```
    }  
    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*****

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

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

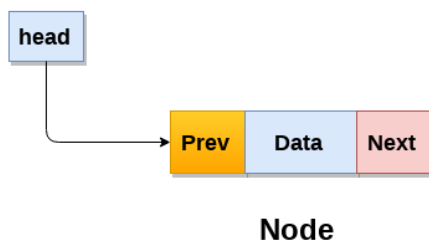
*****Main Menu*****

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.



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.

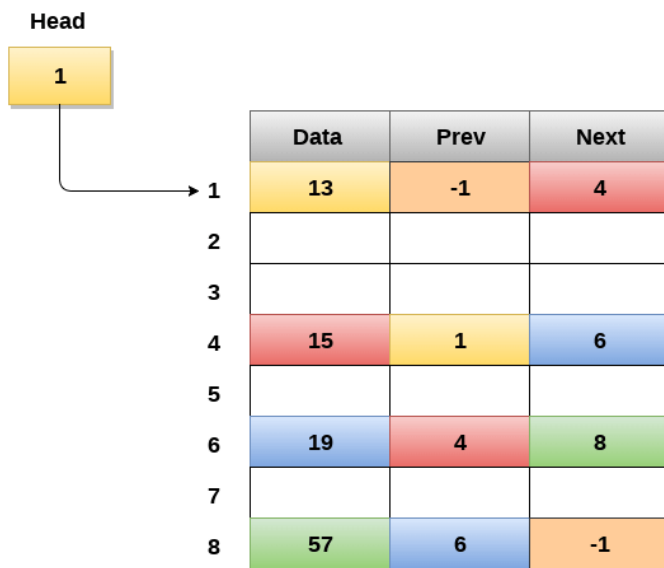
Doubly connected rundown is a mind boggling kind of connected rundown wherein a hub contains a pointer to the past just as the following hub in the arrangement. Subsequently, in a doubly connected rundown, a hub comprises of three sections: hub information, pointer to the following hub in arrangement (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;
```

```
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?

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 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...

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..

```
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");
}
}
void lastinsert()
{
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;
```

```

    }
    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);
```

```
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;
```

*****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

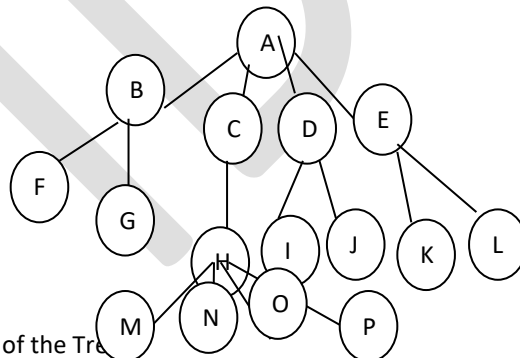
- 10.0 General Tree & Definition
- 10.1 Properties of the Tree
- 10.2 Binary Tree
 - 10.2.1 Binary Tree
 - 10.2.2 Strictly Binary Tree
 - 10.2.3 Almost complete Binary Tree
 - 10.2.4 Complete Binary Tree
- 10.3 Conversion of Tree to Binary tree
- 10.4 Construction of Binary Tree
- 10.5 Binary Search 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 n_1 to n_2 then n_1 is an ancestor of n_2 and n_2 is descendant of n_1 .
10. A tree should not have multiple paths from node n_1 to node n_2 .

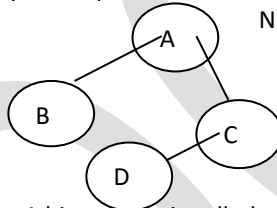
The above tree, height and depth of the tree: 3

Height of root node: 3 ; Depth of all the leaf nodes: 3

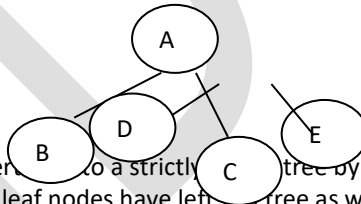
Depth of Root node: 0 ; Height of all the leaves : 0

- 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.

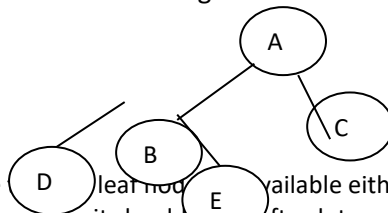


Any binary tree can be converted to a strictly binary tree by adding left sub-tree or right sub-tree.

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.

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

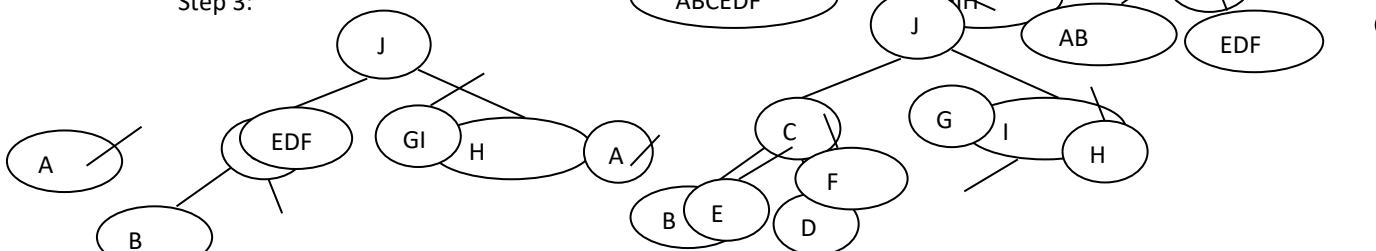
1. The first value of pre-order traversal becomes the root node value.
2. All the values lying left to the same value in in-order will be the part of left sub-tree and the values which are right to the in-order of the same value will be part of right sub-tree.

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:



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

Pre-order and Post-order Traversal.

Pre-Order: A B E C

Post-Order: C 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.

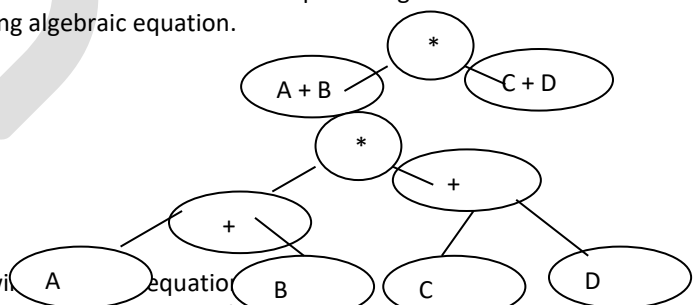
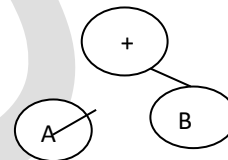
Step 3: All the values which comes left to prefix value in infix notation will be part of left sub tree and the values come to right to the prefix value in infix notation will be the part of right sub tree.

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.

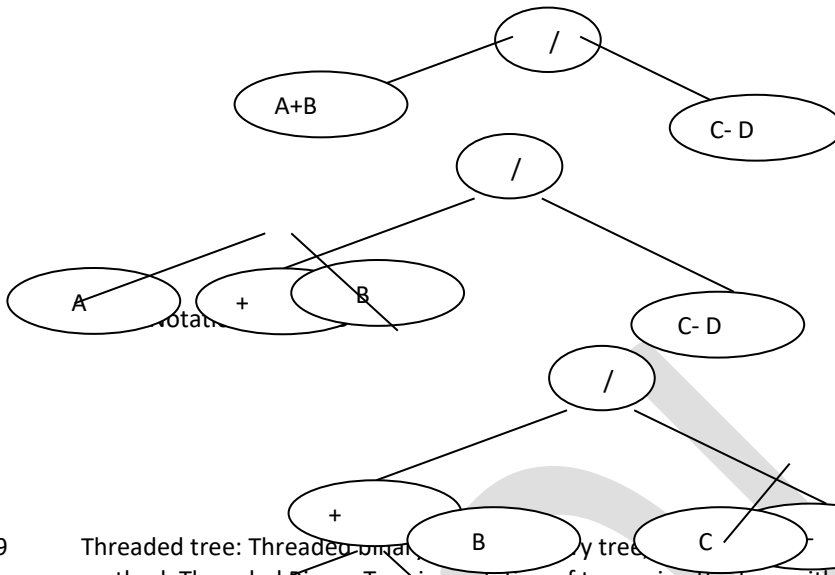
A. $A/B*C$

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 binary tree in which the null pointers of a binary tree are replaced by threads to the next node in the in-order traversal. Threaded Binary Tree is a solution of traversing the tree without using back tracking method.

If a binary 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.

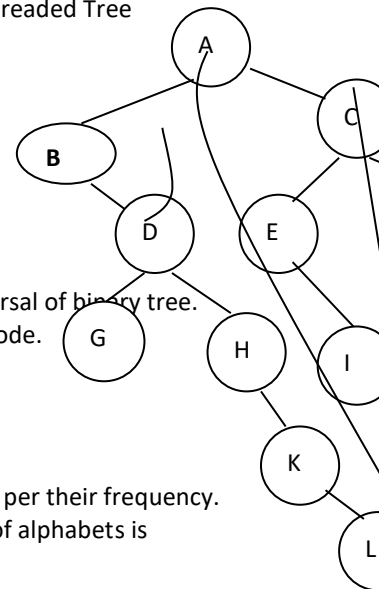
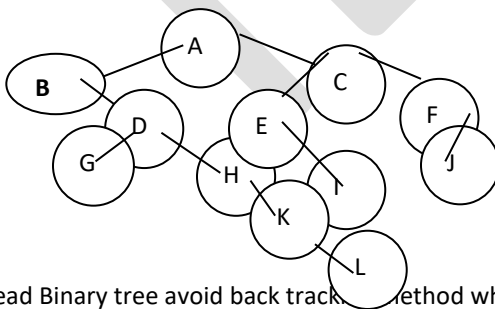
In-order: B G D H K L A E I C J

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:

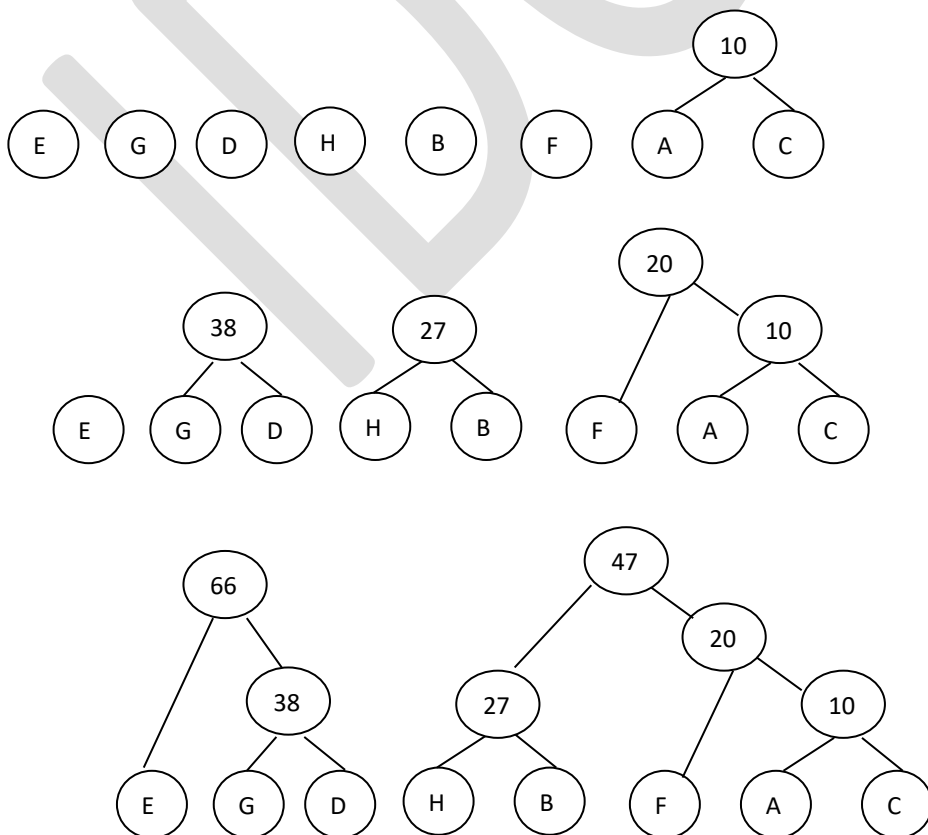
1. Arrange all the alphabets in ascending or descending order and place them a leaf node of a tree.
2. Add two lowest frequency nodes and form a new node which will be one level up. The value of this new node will be the addition of both the nodes which are added.
3. Keep adding two lowest frequency 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



Unit 5: Chapter 11

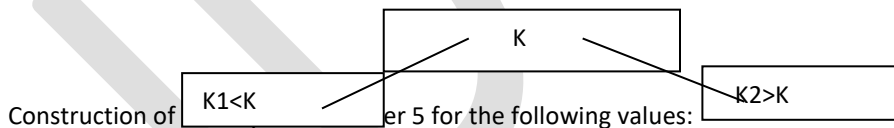
M- Way Tree

- 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.

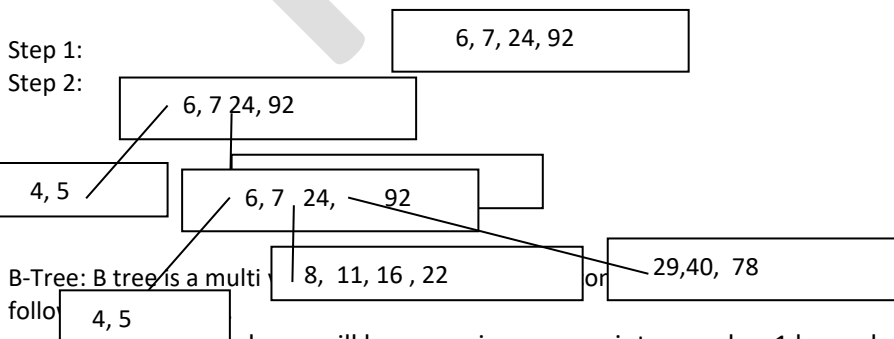
1. M-Way tree of order m will have maximum m pointers and m-1 key values in a node. All the keys must be placed in an increasing order or ascending order.
2. If a node has m-1 key values then the node is full. If a node is full then the new key value will be either left child value of the parent node 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.

11.1.2



92, 24 6,7,11,8,22,4,5,16,29,40,78

Maximum Number of key values in a node: $(m-1) = 5-1 = 4$



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 then its parent node value) or right child value of the parent node (if it is greater than its parent value).

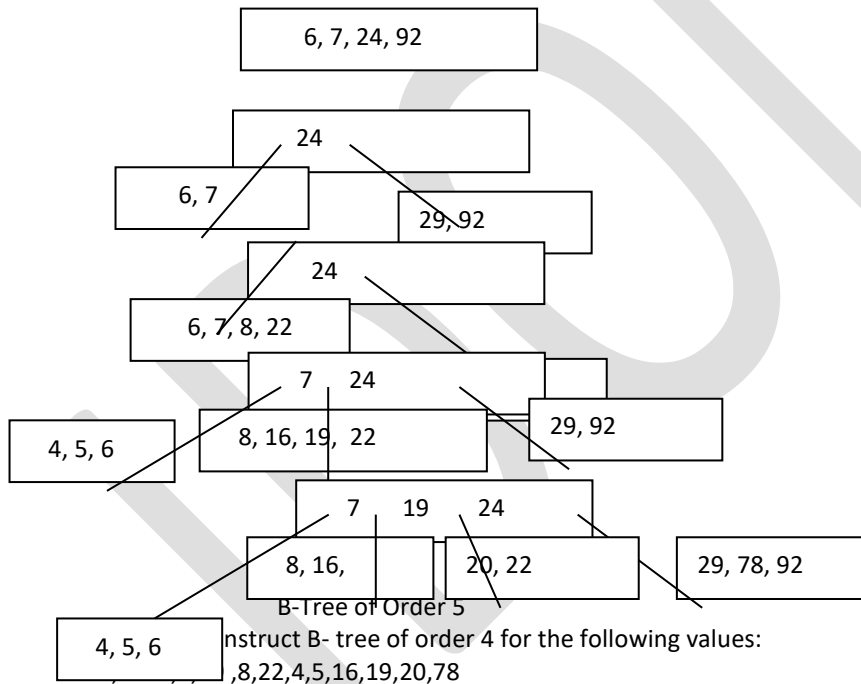
3. A new key value will be added at the leaf node value.
4. The leaves must be at the same level.
5. The height of B-tree will always be lesser than M-Way Tree.
6. The new value will be inserted in the leaf node.

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

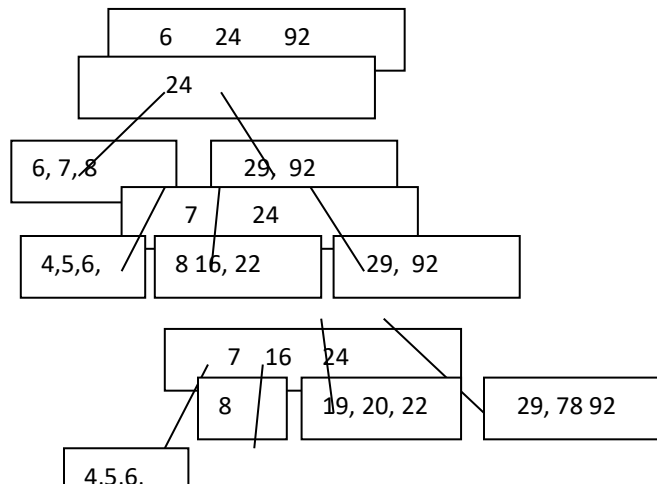
Maximum Number of Key values in a node= $m-1=5-1=4$



Construct B- tree of order 4 for the following values:

92,24,4,5,16,19,20,78

Maximum Number of Key values in a node= $m-1=4-1=3$



B Tree of Order 4

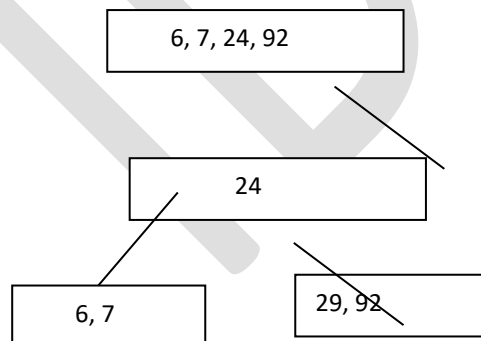
11.3.1 B*-Tree: B* tree is a multi valued tree in which a node consists of multiple values and satisfy the following properties.

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:

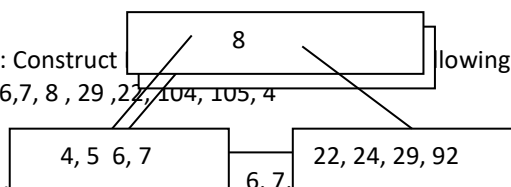
92, 24, 6, 7, 29, 8, 22, 4, 5

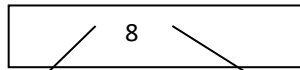
Maximum Number of Key values in a node = $m-1=5-1=4$



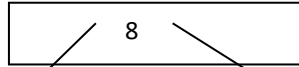
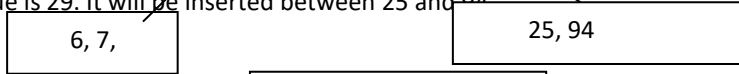
Problem: Construct B*-tree of order 5 for the following values:
94, 25, 6, 7, 8, 29, 22, 104, 105, 4

Next value 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.

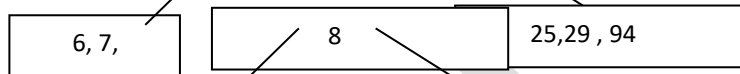




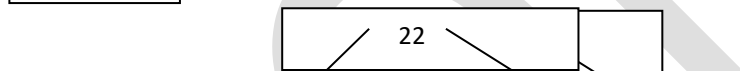
Next value is 29. It will be inserted between 25 and 94



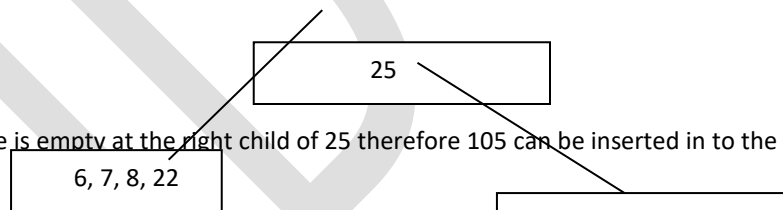
Next value is 22. It will be inserted before 25 as the node can consist of maximum 4 key values.



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 104 can be inserted after rotating 22 as a root node and 8 will come down

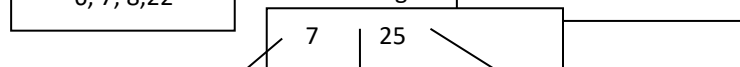


Once the key values 6, 7, 8, 7, 25, 29, 94, 104 are present in the B-tree, the place is empty at right child of 22. Next value is 105. It can't be inserted in to right child of 22 as the node is full therefore 105 can't be inserted in to right child of 8. Hence new value can be inserted after rotating 22 as a root node and 8 will come down to the left hand side.

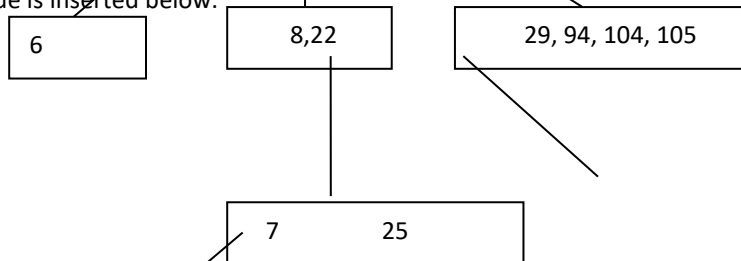


Now the place is empty at the right child of 25 therefore 105 can be inserted in to the right child of 25.

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 25. 29, 94, 104, 105 will become the right child of 25.



Here the new value is inserted below.

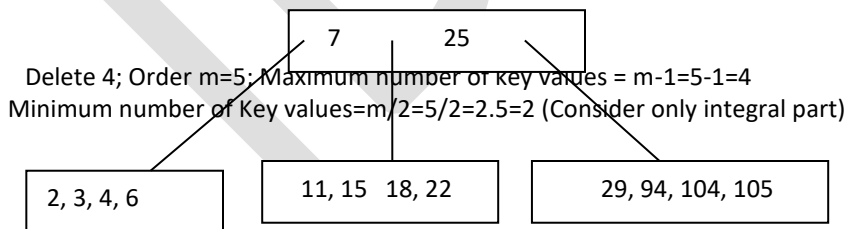


11.4 Deletion from B Tree:

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.

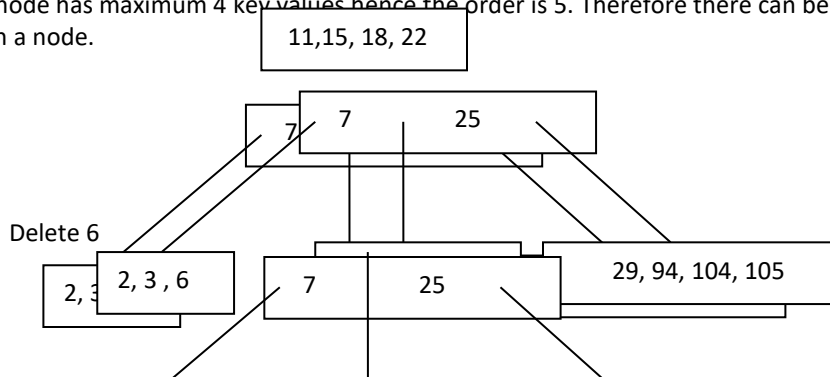
1. The key values are deleted from the leaf node only.
2. At a time only one key value is deleted.
3. The key value can't be deleted from root node and internal node.
4. Key value can't be deleted if the number of key values is less than $m/2$ key values.
5. If the key value which is available at leaf node is supposed to be deleted and the node have more than $m/2$ key values then 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.



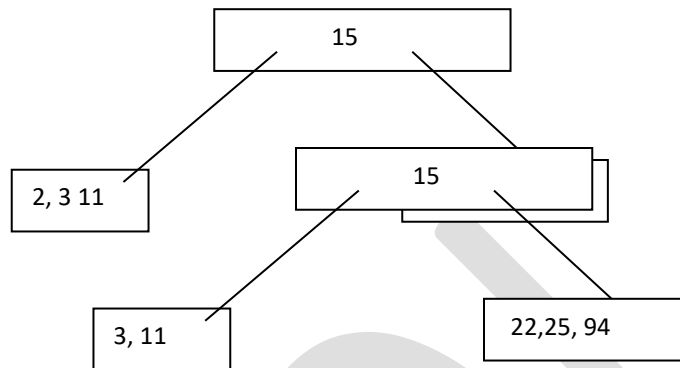
Four is deleted from the leaf node directly since it was available at the leaf node and the node has more than $m/2$ Key values.

Since a node has maximum 4 key values hence the order is 5. Therefore there can be minimum 2 key values in a node.

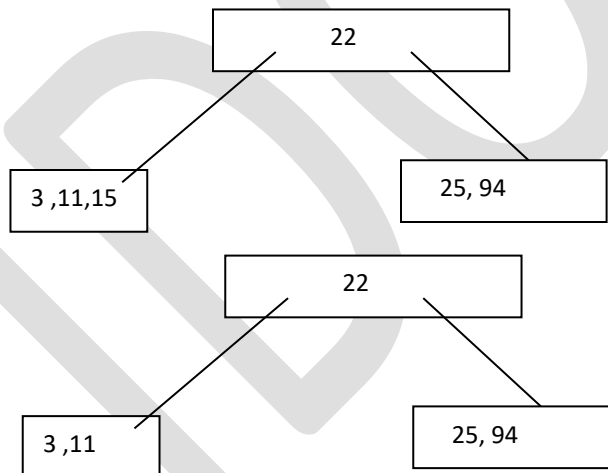


Since the node containing the value 29, will have less than $m/2$ key values therefore both the siblings will be merged.

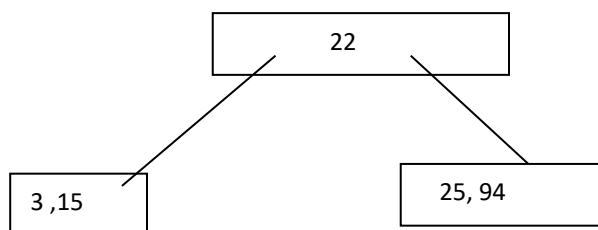
Delete 2 from the tree: First rotate the key values in which 11 will move down and 15 will move up.



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.

Unit 6: Chapter 12

Graphs

Introduction to Graphs

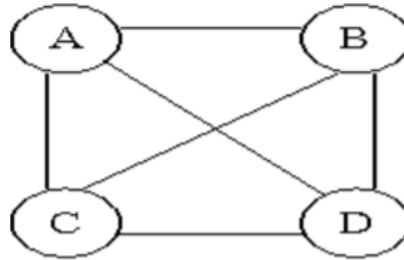
12.0 Objectives	274
12.1 Introduction.....	275
12.2 Basic Concepts of Graphs	275
12.2.1 Types of Graphs	277
12.2.2 Representing Graphs	278
12.2.2.1 Adjacency Matrix.....	278
12.2.2.2 Adjacency List:	280
12.2.2.3 Adjacency Multi-lists.....	281
12.2.2.4 Weighted edges	282
12.2.3 Operations on Graphs.....	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

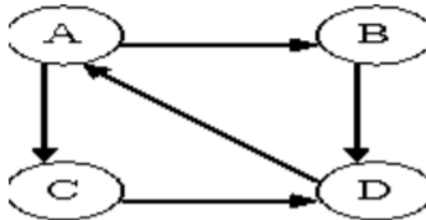
1. Explain basic graph terminologies.
2. Understand adjacency matrix and convert to graphs.
3. Describe various operations like BFS, DFS performed on graphs
4. Analyse the applications of graphs in day-to-day life

12.2.1 Types of Graphs

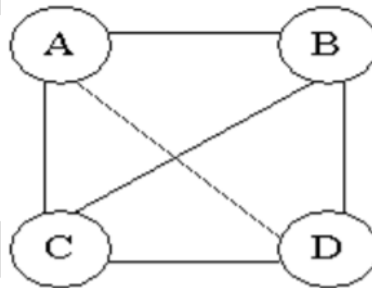
1. **Undirected Graph:** A graph with only undirected edges is said to be undirected graph.



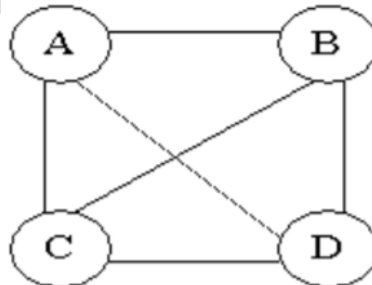
2. **Directed Graph:** A graph with only directed edges is said to be directed graph.



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 $\text{edges} = n(n-1)/2$ 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.



0		1	NULL
1		0	NULL
2		1	NULL

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

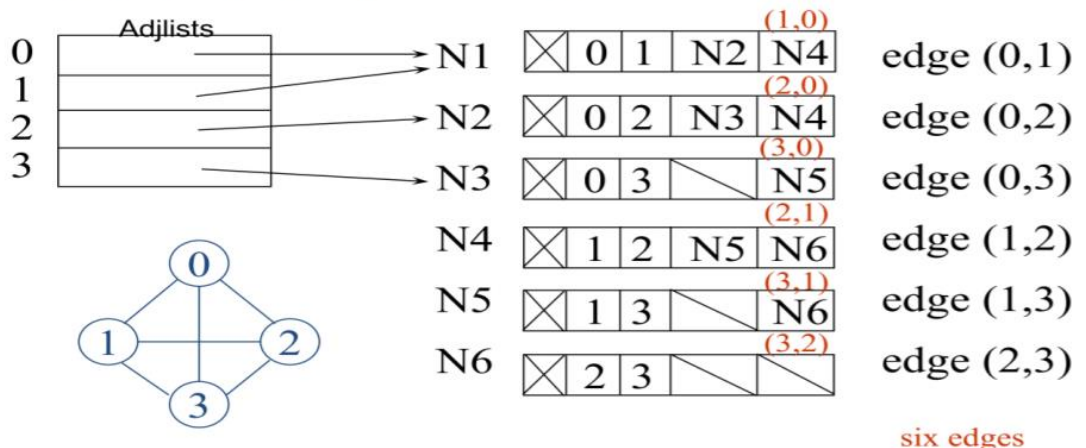


Figure: Adjacency multilists for given graph

12.2.3.3 Connected Components

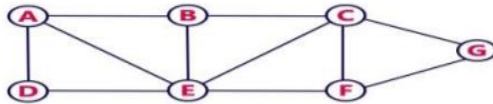
If G is an undirected graph, then one can determine whether or not it is connected by simply making a call to either DFS or BFS and then determining if there is any unvisited vertex. The connected components of a graph may be obtained by making repeated calls to either $\text{DFS}(v)$ or $\text{BFS}(v)$; 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:

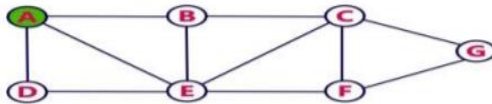
If G is represented by its adjacency lists, then the total time taken by dfs is $O(e)$. Since the for loops take $O(n)$ time, the total time to generate all the Connected components is $O(n+e)$. If adjacency matrices are used, then the time required is $O(n^2)$.

Consider the following example graph to perform DFS traversal



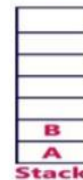
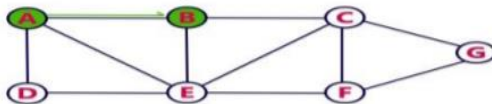
Step 1:

- Select the vertex **A** as starting point (visit **A**).
- Push **A** on to the Stack.



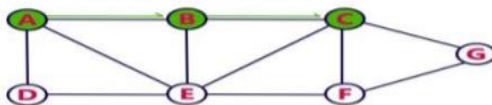
Step 2:

- Visit any adjacent vertex of **A** which is not visited (**B**).
- Push newly visited vertex **B** on to the Stack.



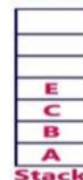
Step 3:

- 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.

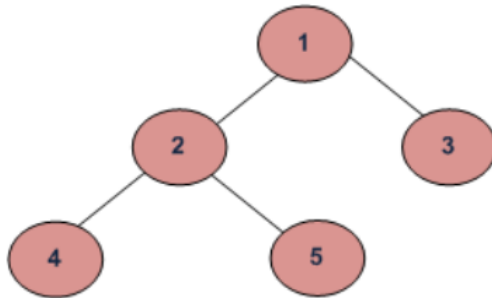


3. Visit the root.

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.

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.

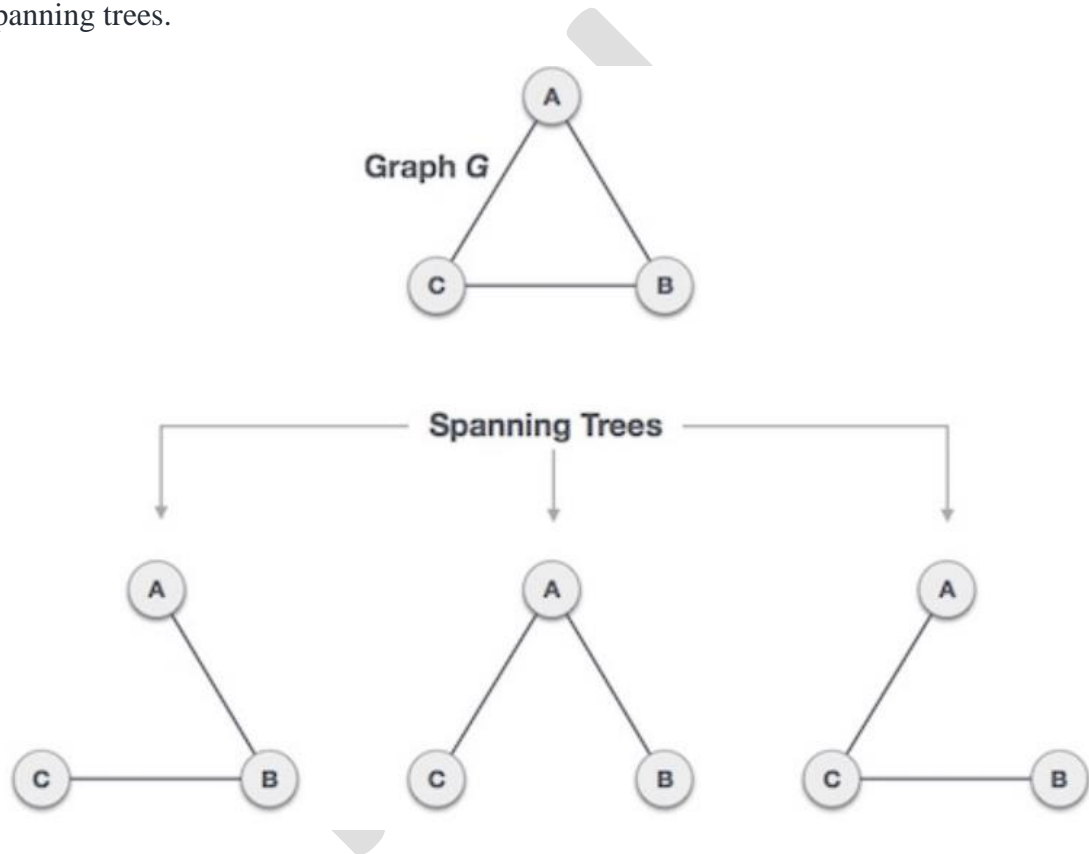
A pair (x,y) is referred to as an edge, which communicates that the x vertex connects to the y vertex.

In the examples below, circles represent vertices, while lines represent edges.

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

13.3 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.

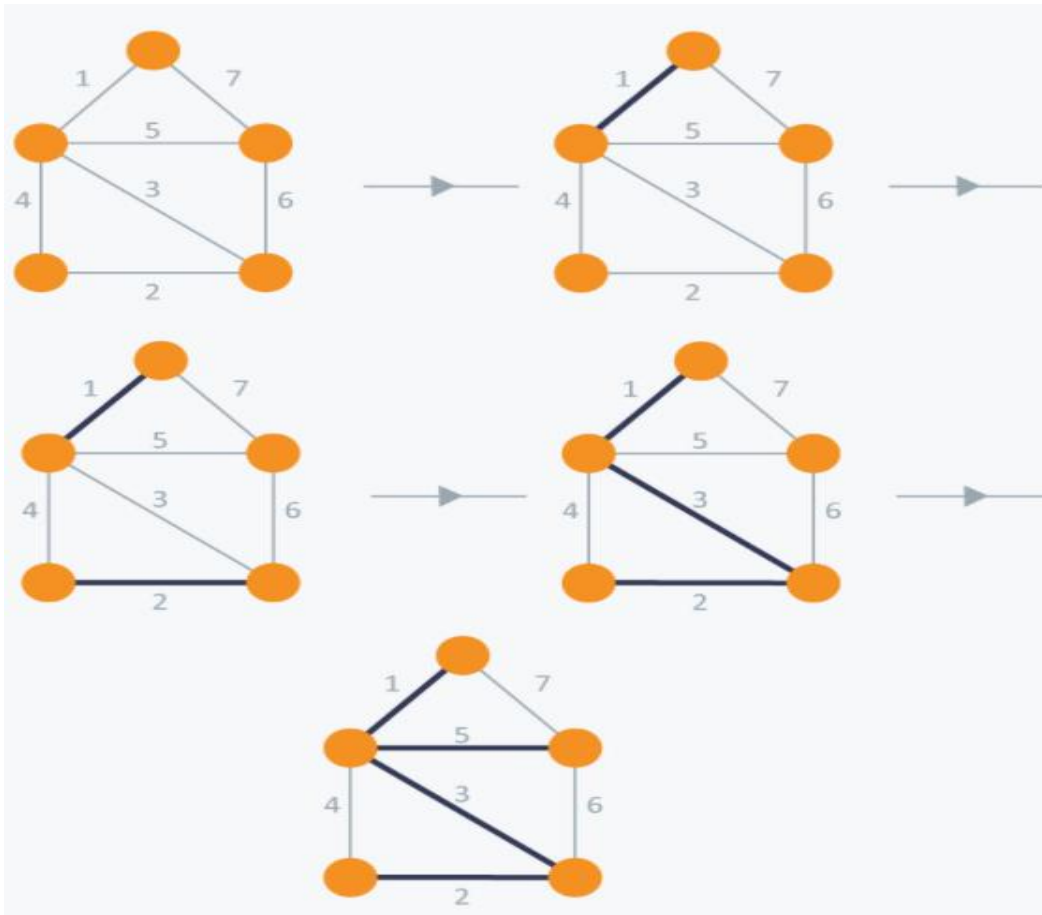
So now the question is how to check if 2 vertices are connected or not ?

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;
}
```

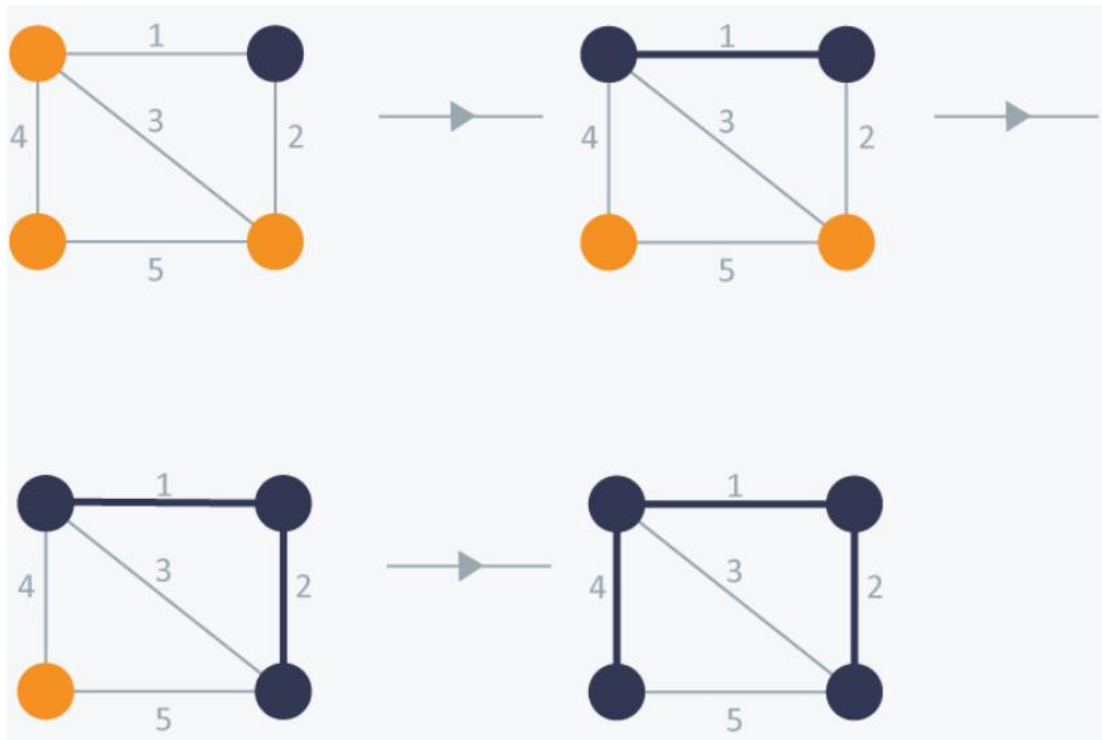
```
int root(int x)
{
    while(id[x] != x)
    {
        id[x] = id[id[x]];
        x = id[x];
    }
    return x;
}
```

```
void union1(int x, int y)
{
    int p = root(x);
    int q = root(y);
    id[p] = id[q];
}
```

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> PII;
```

```
bool marked[MAX];
```

```
vector <PII>adj[MAX];
```

```
long longprim(int x)
```

```
{
```

```
priority_queue<PII, vector<PII>, greater<PII>>Q;
```

```
int y;
```

```
long longminimumCost = 0;
```

```
PII p;
```

```
Q.push(make_pair(0, x));
```

```
while(!Q.empty())
```

```
{
```

```
// Select the edge with minimum weight
```

```
p = Q.top();
```

```
Q.pop();
```

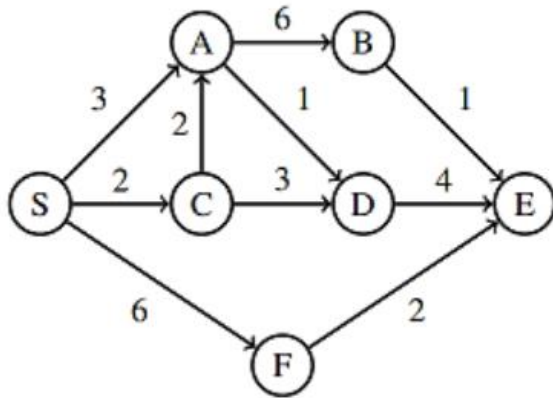
```
x = p.second;
```

```
// Checking for cycle
```

```
if(marked[x] == true)
```

```
continue;
```

```
minimumCost += p.first;
```



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
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

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

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

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.

$\text{Fibonacci}(n) = 1$; if $n = 0$

$\text{Fibonacci}(n) = 1$; 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.

```

{
    // 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-

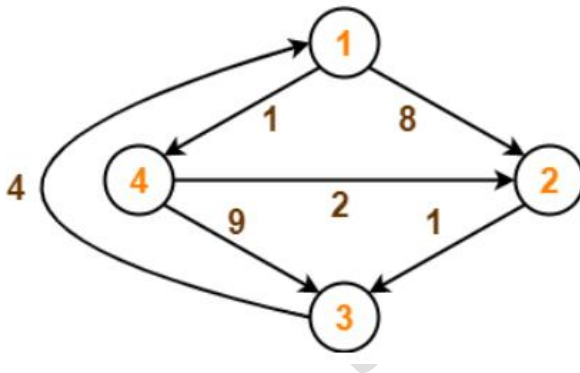
- Floyd Warshall Algorithm consists of three loops over all the nodes.
- The inner most loop consists of only constant complexity operations.
- Hence, the asymptotic complexity of Floyd Warshall algorithm is $O(n^3)$.
- Here, n is the number of nodes in the given graph.

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 = ∞ .

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.

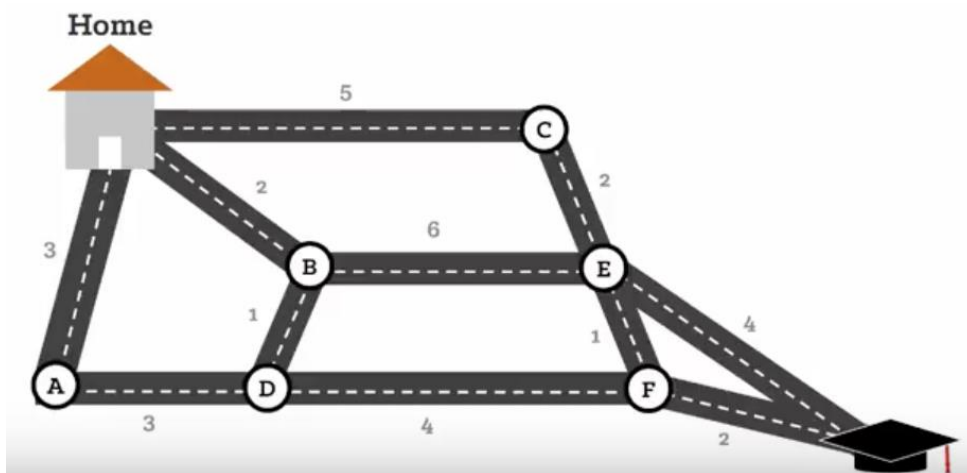
If you are given a directed or undirected weighted graph with n vertices and m edges. The weights of all edges are non-negative. You are also given a starting vertex s . To find the lengths of the shortest paths from a starting vertex s to all other vertices, and output the shortest paths themselves, 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:



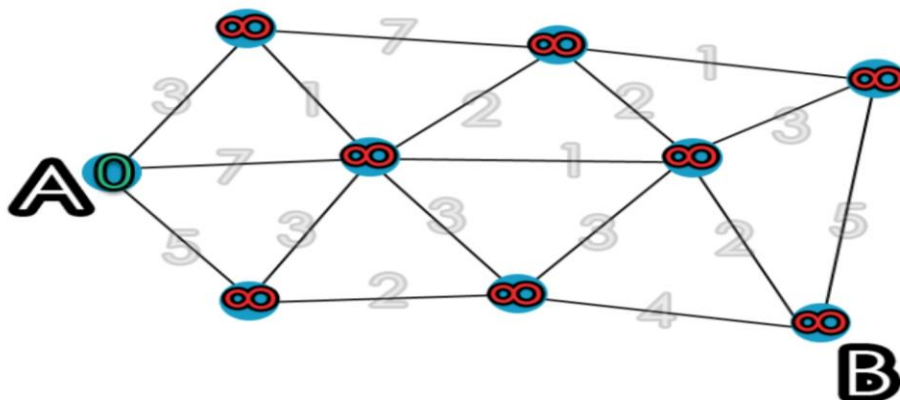
The shortest path, which could be found using Dijkstra's algorithm, is

Home → B → D → F → School

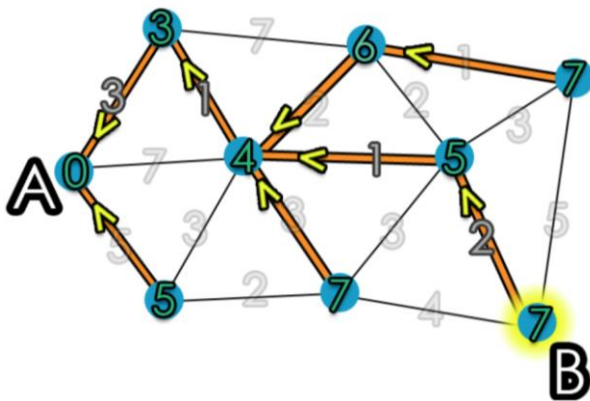
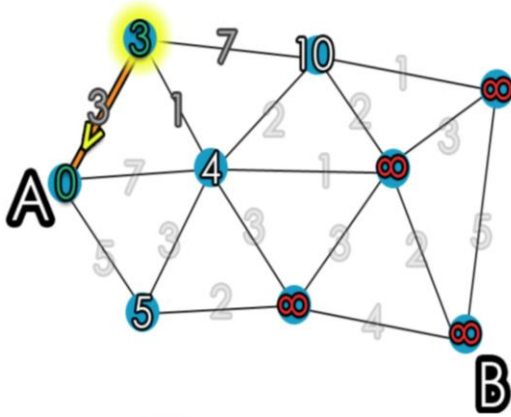
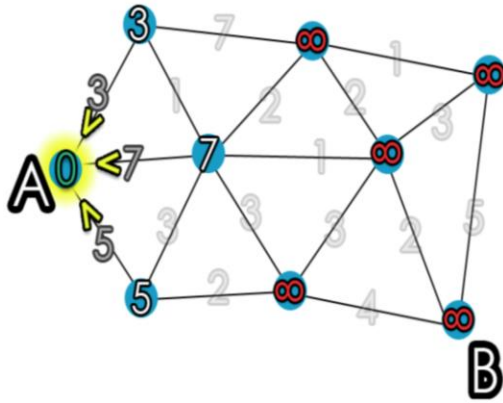
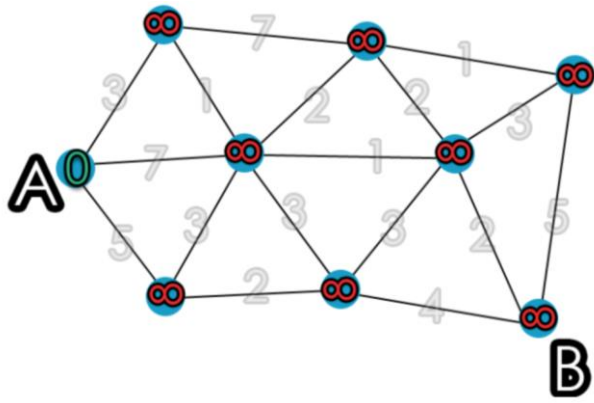
Algorithm:

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.



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.

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

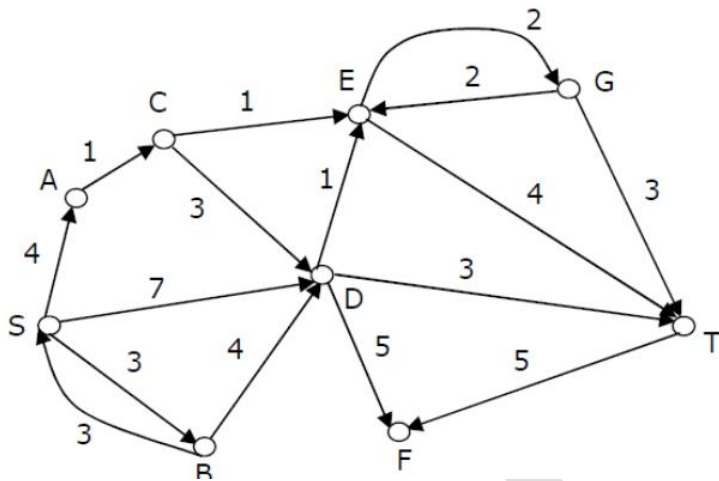
In previous chapter, we have studied Minimum Spanning Tree (MST) and its use in various algorithms like Kruskal's Algorithm, Prim's Algorithm. Based on this we have studied all-pair shortest path algorithms like Floyd Warshall's Algorithm, Dijkstra's Algorithm. We have studied the applications of various graph algorithms and data structures in day-to-day life.

14.5 Review Your Learning

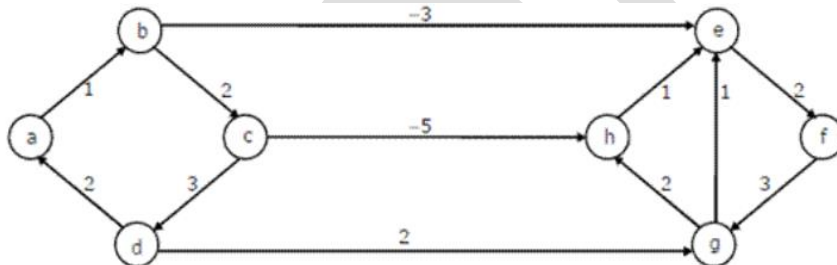
1. Are you able to find shortest path using Dijkstra's algorithm?
2. Are you able to find distance matrix and use it in Floyd Warshall's algorithm?
3. Explain the difference between Dynamic and Greedy programming approach. Give examples of shortest path algorithms in both categories.
4. Explain applications 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

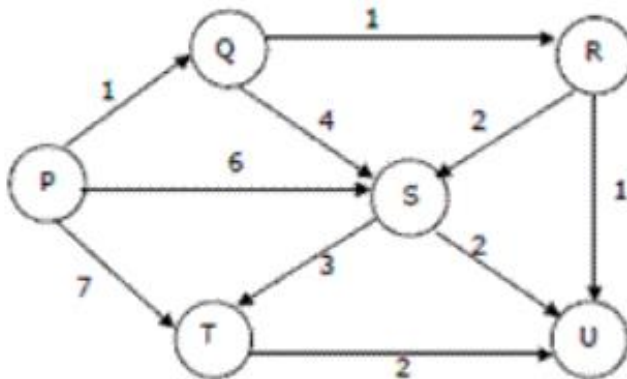
6. Consider the directed graph shown in the figure below. There are multiple shortest paths between vertices S and T. Which one will be reported by Dijkstra's shortest path algorithm? Assume that, in any iteration, the shortest path to a vertex v is updated only when a strictly shorter path to v is discovered.



7. Dijkstra's single source shortest path algorithm when run from vertex a in the below graph, computes the correct shortest path distance to??

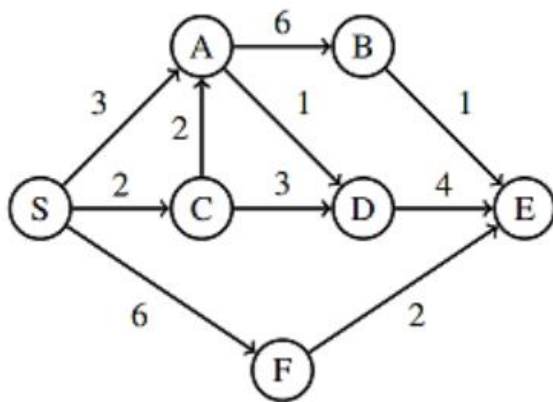


8. Suppose we run Dijkstra's single source shortest-path algorithm on the following edge weighted directed graph with vertex P as the source. In what order do the nodes get included into the set of vertices for which the shortest path distances are finalized?

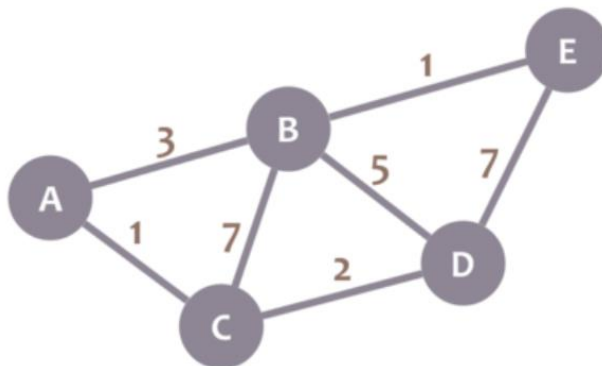


9. Dijkstra's Algorithm is used to solve _____ problems.
- All pair shortest path
 - Single source shortest path

- 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
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
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