UNIT I

INTRODUCTION TO DATA STRUCTURE

OUTLINE

INTRODUCTION TO DATA STRUCTURE

1.1 NEED FOR DATA STRUCTURE

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

1. **Processor speed**

   To handle very large amount of data, high speed processing is required, but as the data is growing day by day to the billions of files per entity, processor may fail to deal with that much amount of data.

2. **Data Search**

   Consider an inventory size of 106 items in a store, If our application needs to search for a particular item, it needs to traverse 106 items every time, results in slowing down the search process

3. **Multiple requests**

   If thousands of users are searching the data simultaneously on a web server, then there are the chances that a very large server can be failed during that process. In order to solve the above problems, data structures are used.

1.2 DEFINITIONS

Data structure is representation of the logical relationship existing between individual elements of data. In other words, a data structure is a way of organizing all data items that considers not only the elements stored but also their relationship to each other.

- Data Structure can be defined as the group of data elements which provides an efficient way of storing and organizing data in the computer.
- Some examples of Data Structures are arrays, Linked List, Stack, Queue, etc.

1.3 ARRAYS

1.3.1. INTRODUCTION

Arrays are defined as the collection of similar type of data items stored at contiguous memory locations. Simply, declaration of array is as follows:

int arr[10]

- Where int specifies the data type or type of elements arrays stores.
- "arr" is the name of array & the number specified inside the square brackets is the number of elements an array can store, this is also called sized or length of array.

1.3.2. RANGE OF AN ARRAY

Given an array **arr** of integer elements, the task is to find the range and coefficient of range of the given array where:

Range: Difference between the maximum value and the minimum value in the distribution.

Coefficient of Range: (Max – Min) / (Max + Min).

Examples:

Input: arr[] = {15, 16, 10, 9, 6, 7, 17}

Output: Range : 11

Coefficient of Range : 0.478261

Max = 17, Min = 6

Range = Max – Min = 17 – 6 = 11

Coefficient of Range = (Max – Min) / (Max + Min) = 11 / 23 = 0.478261

Input: arr[] = {5, 10, 15}

Output: Range : 10

Coefficient of Range : 0.5

1.3.3. ONE DIMENSIONAL ARRAY

A one-dimensional array as a row, where elements are stored one after another.

datatype array_name[size];

datatype: It denotes the type of the elements in the array.

array_name: Name of the array. It must be a valid identifier.

s**ize:** Number of elements an array can hold.

Here is some example of array declarations.

int num[100]; float temp[20]; char ch[50];

num is an array of type **int**, which can only store 100 elements of type int.

temp is an array of type **float**, which can only store 20 elements of type float.

ch is an array of type **char**, which can only store 50 elements of type char.

For example: Reading an array

```
For(i=0;i<=9;i++)
 scanf("%d",&arr[i]);
```

For example: Writing an array

```
For(i=0;i<=9;i++)
 printf("%d",arr[i])
```

1.3.4. TWO DIMENSIONAL ARRAY

2D array can be defined as an array of arrays. The 2D array is organized as matrices which can be represented as the collection of rows and columns.

- The syntax of declaring two dimensional array is very much similar to that of a one dimensional array, given as follows.
- For ex **:int arr[max_rows][max_columns];**

The two dimensional array, the elements are organized in the form of

rows and columns. First element of the first row is represented by a[0][0] where the number shown in the first index is the number of that row while the number shown in the second index is the number of the column.

1.3.5 SPECIAL TYPE OF MATRICES

1. **Row Matrix** - A row matrix is formed by a single row.

$$\begin{pmatrix} 2 & 3 & -1 \end{pmatrix}$$

2. **Column Matrix** - A column matrix is formed by a single column.

$$\begin{pmatrix} -7 \\ 1 \\ 6 \end{pmatrix}$$

3. **Rectangular Matrix** - A rectangular matrix is formed by a different number of rows and columns, and its dimension is noted as: m x n.

$$\begin{pmatrix} 1 & 2 & 5 \\ 9 & 1 & 3 \end{pmatrix}$$

4. **Square Matrix** - A square matrix is formed by the same number of rows and columns.

The elements of the form a_{ii} constitute the principal diagonal.

The secondary diagonal is formed by the elements with i+j = n+1.

5.**Zero Matrix** - In a zero matrix, all the elements are zeros.

$$\begin{pmatrix} 0 & 0 \\ 0 & 0 \end{pmatrix}$$

6. **Upper Triangular Matrix** - In an upper triangular matrix, the elements located below the diagonal are zeros.

$$\begin{pmatrix} 1 & 7 & -2 \\ 0 & -3 & 4 \\ 0 & 0 & 2 \end{pmatrix}$$

7. **Lower Triangular Matrix** - In a lower triangular matrix, the elements above the diagonal are zeros.

$$\begin{pmatrix} 2 & 0 & 0 \\ 1 & 2 & 0 \\ 3 & 5 & 6 \end{pmatrix}$$

8. **Diagonal Matrix** - In a diagonal matrix, all the elements above and below the diagonal are zeros.

$$\begin{pmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 6 \end{pmatrix}$$

9. **Scalar Matrix** - A scalar matrix is a diagonal matrix in which the diagonal elements are equal.

$$\begin{pmatrix} 2 & 0 & 0 \\ 0 & 2 & 0 \\ 0 & 0 & 2 \end{pmatrix}$$

10. **Identity Matrix** - An identity matrix is a diagonal matrix in which the diagonal elements are equal to 1.

$$\begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix}$$

11. **Transpose Matrix** - Given matrix A, the transpose of matrix A is another matrix where the elements in the columns and rows have switched. In other words, the rows become the columns and the columns become the rows.

$$A = \begin{pmatrix} 2 & 3 & 0 \\ 1 & 2 & 0 \\ 3 & 5 & 6 \end{pmatrix} \qquad A^t = \begin{pmatrix} 2 & 1 & 3 \\ 3 & 2 & 5 \\ 0 & 0 & 6 \end{pmatrix}$$

$(A^t)^t = A$

$(A + B)^t = A^t + B^t$

$(\alpha \cdot A)^t = \alpha \cdot A^t$

$(A \cdot B)^t = B^t \cdot A^t$

12. **Regular Matrix** - A regular matrix is a square matrix that has an inverse.

13. **Singular Matrix** - A singular matrix is a square matrix that has no inverse.

14. **Idempotent Matrix** - The matrix A is idempotent if:

$A^2 = A$.

15. **Involutive Matrix** - The matrix A is involutive if:

$A^2 = I$.

16. **Symmetric Matrix** - A symmetric matrix is a square matrix that verifies:

$A = A^t$.

17. **Antisymmetric Matrix** - An antisymmetric matrix is a square matrix that verifies:

$A = -A^t$.

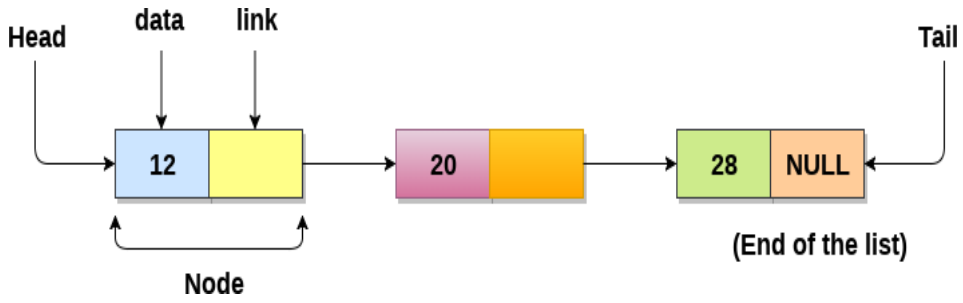
18. **Orthogonal Matrix** - A matrix is orthogonal if it verifies that:

$A \cdot A^t = I$

1.4 LINKED LISTS

1.4.1. INTRODUCTION

- 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.



1.4.2. BENEFITS AND LIMITATIONS OF LINKED LIST

BENEFITS

- **Dynamic Data Structure**

  Linked list is a dynamic data structure so it can grow and shrink at runtime by allocating and deallocating memory. So there is no need to give initial size of linked list.

- **Insertion and Deletion**

  Insertion and deletion of nodes are easier. In linked list we just have to update the address present in next pointer of a node.

- **No Memory Wastage**

  As size of linked list can increase or decrease at run time so there is no memory wastage. In linked list the memory is allocated only when required.

- **Implementation**

  Data structures such as stack and queues can be easily implemented using linked list.

LIMITATIONS

- **Memory Usage**

  More memory is required to store elements in linked list as compared to array. Because in linked list each node contains a pointer and it requires extra memory for itself.

- **Traversal**

  Elements or nodes traversal is difficult in linked list. For example if we want to access a node at position n then we have to traverse all the nodes before it. So, time required to access a node is large

- **Reverse Traversing**

  In linked list reverse traversing is really difficult. In case of doubly linked list its easier but extra memory is required for back pointer hence wastage of memory**.**

1.4.3. TYPES OF LINKED LIST

There are three types of Linked List.

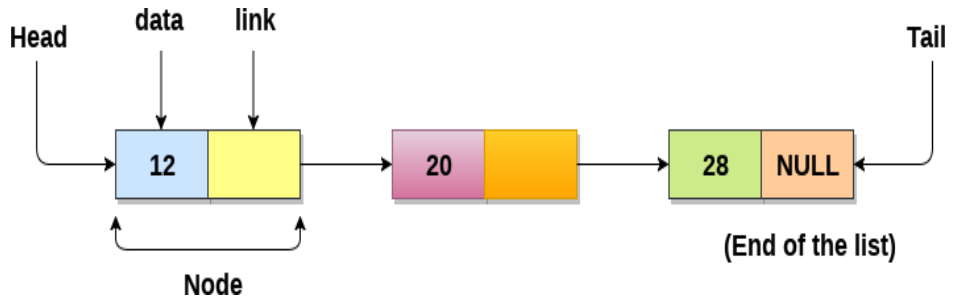
1. **Singly linked list.**
2. **Circular linked list.**
3. **Doubly linked list.**

1.4.3.1. SINGLY LINKED LIST

Singly linked list can be defined as the collection of ordered set of elements.

A node in the singly linked list consist of two parts:

- **data part** - Data part of the node stores actual information that is to be represented by the node
- **link part**- the link part of the node stores the address of its immediate successor**.**



OPERATIONS ON SINGLY LINKED LIST

There are various operations which can be performed on singly linked list.

Node Creation

```
struct node
{
    int data;
    struct node *next;
};
struct node *head, *ptr;
ptr = (struct node *)malloc(sizeof(struct node *));
```

INSERTION

1. **Insertion at beginning**-It involves inserting any element at the front of the list.

2. **Insertion at end of the list**-It involves insertion at the last of the linked list.

3. **Insertion after specified node**-It involves insertion after the specified node of the linked list.

DELETION & TRAVERSING

1. **Deletion at beginning**-It involves deletion of a node from the beginning of the list.

2. **Deletion at the end of the list**-It involves deleting the last node of the list.

3. **Deletion after specified node**-It involves deleting the node after the specified node in the list.

4. **Traversing**-In traversing, we simply visit each node of the list at least once in order to perform some specific operation on it.

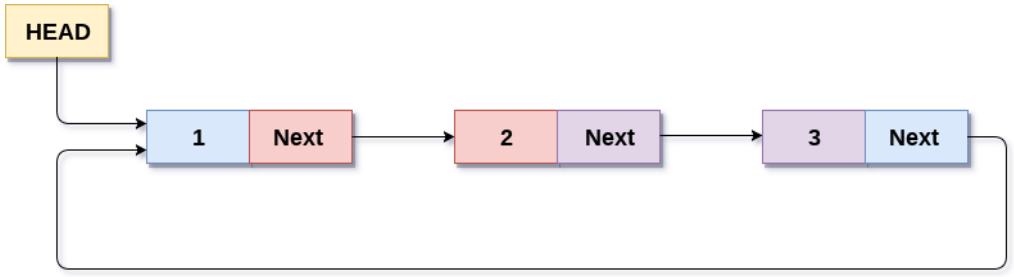
5. **Searching**-In searching, we match each element of the list with the given element. If the element is found on any of the location then location of that element is returned otherwise null is returned.

Program:

```
#include<stdio.h>
#include<stdlib.h>
void beginsert(int);
struct node
{
int data;
struct node *next;
};
struct node *head
void main ()
{
 int choice,item;
 do
 {
 printf("\nEnter the item which you want to insert?\n");
scanf("%d",&item);
 beginsert(item);
  printf("\nPress 0 to insert more ?\n");
  scanf("%d",&choice);
 }while(choice == 0);
}
  ptr->next = head;
  head = ptr;
  printf("\nNode inserted\n");
  }                }
```

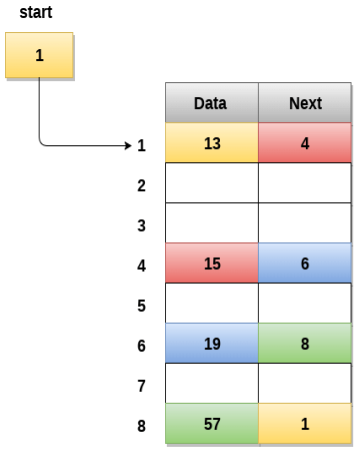
1.4.3.2. CIRCULAR LINKED LIST

- In a circular linked list, the last node of the list contains a pointer to the first node of the list.
- We traverse a circular singly linked list until we reach the same node where we started.
- The circular singly liked list has no beginning and no ending. There is no null value present in the next part of any of the nodes.



Circular Singly Linked List

Memory Representation of circular linked list:



Memory Representation of a circular linked list

Operations on Circular Singly linked list:

Insertion

1. **Insertion at beginning**-Adding a node into circular singly linked list at the beginning.

2. **Insertion at the end**-Adding a node into circular singly linked list at the end.

Deletion & Traversing

1. **Deletion at beginning**-Removing the node from circular singly linked list at the beginning.

2. **Deletion at the end-**Removing the node from circular singly linked list at the end.

3. **Searching**-Compare each element of the node with the given item and return the location at which the item is present in the list otherwise return null.

4. **Traversing**-Visiting each element of the list at least once in order to perform some specific operation.

Algorithm

- **Step 1:** IF PTR = NULL

  Write OVERFLOW
  Go to Step 11
  [END OF IF]

- **Step 2:** SET NEW_NODE = PTR
- **Step 3:** SET PTR = PTR -> NEXT
- **Step 4:** SET NEW_NODE -> DATA = VAL
- **Step 5:** SET TEMP = HEAD

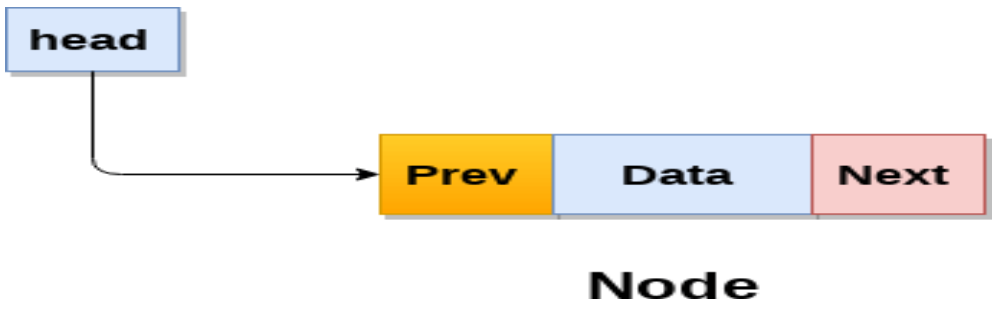
- **Step 6:** Repeat Step 8 while TEMP -> NEXT != HEAD
- **Step 7:** SET TEMP = TEMP -> NEXT
- [END OF LOOP]
- **Step 8:** SET NEW_NODE -> NEXT = HEAD
- **Step 9:** SET TEMP → NEXT = NEW_NODE
- **Step 10:** SET HEAD = NEW_NODE
- **Step 11:** EXIT

1.4.3.3. DOUBLY LINKED LIST

Doubly linked list is a complex type of linked list in which a node contains a pointer to the previous as well as the next node in the sequence.

In a doubly linked list, a node consists of three parts:

1. node data,
2. pointer to the next node in sequence (next pointer) ,
3. pointer to the previous node (previous pointer)

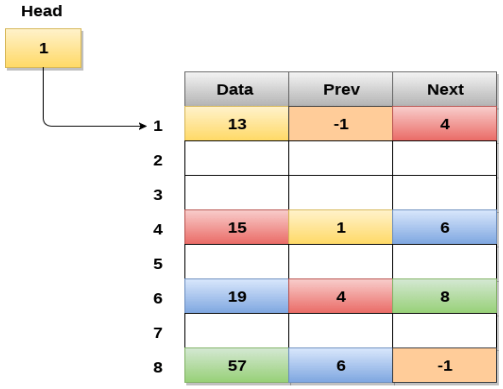


.

In C, structure of a node in doubly linked list can be given as:

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

Memory Representation of a doubly linked list



Memory Representation of a Doubly linked list

Node Creation

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

Operations on doubly linked list

1. Insertion at beginning-Adding the node into the linked list at beginning.

2. Insertion at end-Adding the node into the linked list to the end.

3. Insertion after specified node-Adding the node into the linked list after the specified node.

4. Deletion at beginning-Removing the node from beginning of the list

5. Deletion at the end-Removing the node from end of the list.

6. Deletion of the node having given data-Removing the node which is present just after the node containing the given data.

7. Searching-Comparing each node data with the item to be searched and return the location of the item in the list if the item found else return null.

8. Traversing-Visiting each node of the list at least once in order to perform some specific operation like searching, sorting, display, etc.

UNIT II

STACK AND QUEUE

OUTLINE

STACK

- **2.1 INTRODUCTION**
- **2.2 ADT STACK**
- **2.3 IMPLEMENTATION OF STACK**
- **2.4 APPLICATION OF STACK**

QUEUE

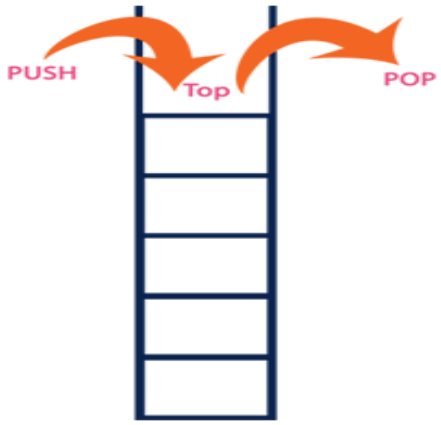
- **2.5 INTRODUCTION**
- **2.6 IMPLEMENTATION OF BASIC OPERATIONS ON ARRAY BASED AND LINKED LIST BASED QUEUE**
- **2.7 CIRCULAR QUEUES**

STACK AND QUEUE

STACK

2.1 INTRODUCTION

- Stack is a linear data structure in which the insertion and deletion operations are performed at only one end.
- In a stack, adding and removing of elements are performed at a single position which is known as "top".
- In stack, the insertion and deletion operations are performed based on LIFO (Last In First Out) principle.



- In a stack, the insertion operation is performed using a function called **"push"** and deletion operation is performed using a function called **"pop"**.

2.2 ADT STACK

- The abstract datatype is special kind of datatype, whose behavior is defined by a set of values and set of operations. The keyword “Abstract” is used as we can use these datatypes, we can perform different operations.

- But how those operations are working that is totally hidden from the user.
- The ADT is made of with primitive datatypes, but operation logics are hidden.

Here we will see the stack ADT. These are few operations or functions of the Stack ADT.

- **isFull()** - This is used to check whether stack is full or not.
- **isEmpty()** - This is used to check whether stack is empty or not.
- **push(x)** - This is used to push x into the stack.
- **pop()** - This is used to delete one element from top of the stack.
- **peek()** - This is used to get the top most element of the stack.
- **size()** - This function is used to get number of elements present into the stack.

2.3 IMPLEMENTATION OF STACK

Stack data structure can be implemented in two ways. They are as follows...

- **Using Array**
- **Using Linked List**

Stack Using Array

- A stack data structure can be implemented using a one-dimensional array. But stack implemented using array stores only a fixed number of data values. Initially, the top is set to -1.

- Whenever we want to insert a value into the stack, increment the top value by one and then insert.
- Whenever we want to delete a value from the stack, then delete the top value and decrement the top value by one.

Stack Operations using Array

Before implementing actual operations, first follow the steps to create an empty stack.

- **Step 1** - Include all the header files which are used in the program and define a constant **'SIZE'** with specific value.
- **Step 2** - Declare all the functions used in stack implementation.
- **Step 3** - Create a one dimensional array with fixed size (int stack[SIZE]).
- **Step 4** - Define a integer variable 'top' and initialize with '-1'. (int top = -1).
- **Step 5** - In main method, display menu with list of operations and make suitable function calls to perform operation selected by the user on the stack.

push(value) - Inserting value into the stack

In a stack, push() is a function used to insert an element into the stack. In a stack, the new element is always inserted at top position. Push function takes one integer value as parameter and inserts that value into the stack.

- **Step 1** - Check whether stack is FULL. (top == SIZE-1)
- **Step 2** - If it is FULL, then display "Stack is FULL!!! Insertion is not possible!!!" and terminate the function.
- **Step 3** - If it is NOT FULL, then increment top value by one (top++) and set stack[top] to value (stack[top] = value).

pop() - Delete a value from the Stack

In a stack, pop() is a function used to delete an element from the stack. In a stack, the element is always deleted from **top** position. Pop function does not take any value as parameter.

- **Step 1 -** Check whether **stack** is **EMPTY**. (**top == -1**)
- **Step 2 -** If it is **EMPTY**, then display **"Stack is EMPTY!!! Deletion is not possible!!!"** and terminate the function.
- **Step 3 -** If it is **NOT EMPTY**, then delete **stack[top]** and decrement **top** value by one (**top--**).

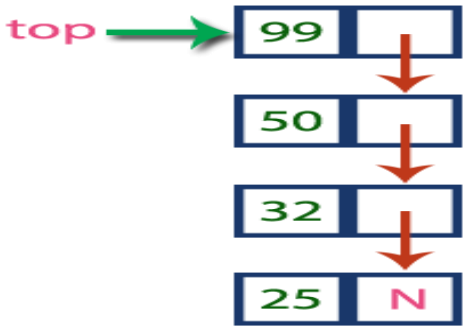
display() - Displays the elements of a Stack

To display the elements of a stack is,

- **Step 1 -** Check whether **stack** is **EMPTY**. (**top == -1**)
- **Step 2 -** If it is **EMPTY**, then display **"Stack is EMPTY!!!"** and terminate the function.
- **Step 3 -** If it is **NOT EMPTY**, then define a variable '**i**' and initialize with top. Display **stack[i]** value and decrement **i** value by one (**i--**).
- **Step 3 -** Repeat above step until **i** value becomes '0'.

Stack Using Linked List

- In linked list implementation of a stack, every new element is inserted as '**top**' element. That means every newly inserted element is pointed by '**top**'.
- Whenever we want to remove an element from the stack, simply remove the node which is pointed by '**top**' by moving '**top**' to its previous node in the list.
- The **next** field of the first element must be always **NULL**.



Stack Operations using Linked List

- **Step 1** - Include all the header files which are used in the program. And declare all the user defined functions.
- **Step 2** - Define a 'Node' structure with two members data and next.
- **Step 3** - Define a Node pointer 'top' and set it to NULL.
- **Step 4** - Implement the main method by displaying Menu with list of operations and make suitable function calls in the main method.

push(value) - Inserting an element into the Stack

To insert a new node into the stack.

- **Step 1 -** Create a **newNode** with given value.
- **Step 2 -** Check whether stack is **Empty** (**top** == **NULL**)
- **Step 3 -** If it is **Empty**, then set **newNode → next** = **NULL**.
- **Step 4 -** If it is **Not Empty**, then set **newNode → next** = **top**.
- **Step 5 -** Finally, set **top** = **newNode**.

pop() - Deleting an Element from a Stack

To delete a node from the stack.

- **Step 1 -** Check whether **stack** is **Empty** (**top == NULL**).
- **Step 2 -** If it is **Empty**, then display **"Stack is Empty!!! Deletion is not possible!!!"** and terminate the function
- **Step 3 -** If it is **Not Empty**, then define a **Node** pointer '**temp**' and set it to '**top**'.

- **Step 4 -** Then set '**top** = **top → next**'.
- **Step 5 -** Finally, delete '**temp**'. (**free(temp)**).

display() - Displaying stack of elements

To display the elements (nodes) of a stack**.**

- **Step 1** - Check whether stack is Empty (top == NULL).
- **Step 2** - If it is Empty, then display 'Stack is Empty!!!' and terminate the function.
- **Step 3** - If it is Not Empty, then define a Node pointer 'temp' and initialize with top.
- **Step 4** - Display 'temp → data --->' and move it to the next node. Repeat the same until temp reaches to the first node in the stack. (temp → next != NULL).
- **Step 5** - Finally! Display 'temp → data ---> NULL'.

2.4 APPLICATION OF STACK

1. Stacks can be used for expression evaluation.

2. Stacks can be used to check parenthesis matching in an expression.

3. Stacks can be used for Conversion from one form of expression to another.

4. Stacks can be used for Memory Management.

5. Stack data structures are used in backtracking problems.

1. **Expression Evaluation**

Stack data structure is used for evaluating the given expression. For example, consider the following expression

5 * (6 + 2) - 12 / 4

Since parenthesis has the highest precedence among the arithmetic operators, (6 +2) = 8 will be evaluated first. Now, the expression becomes

5 * 8 - 12 / 4

* and / have equal precedence and their associativity is from left-to-right. So, start evaluating the expression from left-to-right.

5 * 8 = 40 and 12 / 4 = 3

Now, the expression becomes 40 - 3

And the value returned after the subtraction operation is 37.

2. Parenthesis Matching

Given an expression, you have to find if the parenthesis is either correctly matched or not. For example, consider the expression

(a + b) * (c + d).

In the above expression, the opening and closing of the parenthesis are given properly and hence it is said to be a correctly matched parenthesis expression. Whereas, the expression, (a + b * [c + d) is not a valid expression as the parenthesis are incorrectly given.

3. Expression Conversion

Converting one form of expressions to another is one of the important applications of Stacks.

1. Infix to prefix
2. Infix to postfix

3. Prefix to Infix
4. Prefix to Postfix
5. Postfix to Infix
6. Postfix to Infix

Infix	Prefix	Postfix
a + b	+ b a	a b +
(a + b) * (c + d)	* + d c + b a	a b + c d + *
b * b - 4 * a * c	- * c * a 4 * b b	b b * 4 a * c * -

4. Memory management

- The assignment of memory takes place in contiguous memory blocks. We call this stack memory allocation because the assignment takes place in the function call stack.
- The size of the memory to be allocated is known to the compiler. When a function is called, its variables get memory allocated on the stack.
- When the function call is completed, the memory for the variables is released.
- All this happens with the help of some predefined routines in the compiler.
- The user does not have to worry about memory allocation and release of stack variables.

5. Backtracking Problems

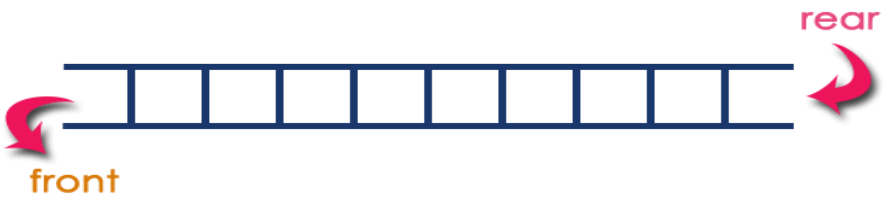
- Consider the N-Queens problem for an example. The solution of this problem is that N queens should be positioned on a chessboard so that none of the queens can attack another queen.

- In the generalized N-Queens problem, N represents the number of rows and columns of the board and the number of queens which must be placed in safe positions on the board.
- The basic strategy we will use to solve this problem is to use backtracking.
- Backtracking means we will perform a safe move for a queen at the time we make the move.

QUEUE

2.5 INTRODUCTION

- Queue is a linear data structure in which the insertion and deletion operations are performed at two different ends. The insertion is performed at one end and deletion is performed at another end.
- In a queue data structure, the insertion operation is performed at a position which is known as ‘rear‘and the deletion operation is performed at a position which is known as 'front'.
- In queue data structure, the insertion and deletion operations are performed based on FIFO (First In First Out) principle. In a queue data structure, the insertion operation is performed using a function called "enQueue()" and deletion operation is performed using a function called deQueue()".



2.6 IMPLEMENTATION OF BASIC OPERATIONS ON ARRAY BASED AND LINKED LIST BASED QUEUE

Queue data structure can be implemented in two ways. They are as follows...

1. Using Array
2. Using Linked List

1. Queue Using Array

A queue data structure can be implemented using one dimensional array. The queue implemented using array stores only fixed number of data values.

Just define a one dimensional array of specific size and insert or delete the values into that array by using FIFO (First In First Out) principle with the help of variables 'front' and 'rear'. Initially both 'front' and 'rear' are set to -1.

Queue Operations using Array

- **Step 1 -** Include all the **header files** which are used in the program and define a constant **'SIZE'** with specific value.
- **Step 2 -** Declare all the **user defined functions** which are used in queue implementation.
- **Step 3 -** Create a one dimensional array with above defined SIZE (**int queue[SIZE]**)
- **Step 4 -** Define two integer variables **'front'** and '**rear**' and initialize both with **'-1'**.

  (**int front = -1, rear = -1**)

- **Step 5 -** Then implement main method by displaying menu of operations list and make suitable function calls to perform operation selected by the user on queue.

enQueue(value) - Inserting value into the queue

- **Step 1 -** Check whether **queue** is **FULL**. (**rear == SIZE-1**)
- **Step 2 -** If it is **FULL**, then display **"Queue is FULL!!! Insertion is not possible!!!"** and terminate the function.
- **Step 3 -** If it is **NOT FULL**, then increment **rear** value by one (**rear++**) and set **queue[rear]** = **value**.

deQueue() - Deleting a value from the Queue

- **Step 1 -** Check whether **queue** is **EMPTY**. (**front == rear**)
- **Step 2 -** If it is **EMPTY**, then display **"Queue is EMPTY!!! Deletion is not possible!!!"** and terminate the function.
- **Step 3 -** If it is **NOT EMPTY**, then increment the **front** value by one (**front ++**). Then display **queue[front]** as deleted element. Then check whether both **front** and **rear** are equal (**front** == **rear**), if it **TRUE**, then set both **front** and **rear** to '**-1**' (**front** = **rear** = **-1**).

display() - Displays the elements of a Queue

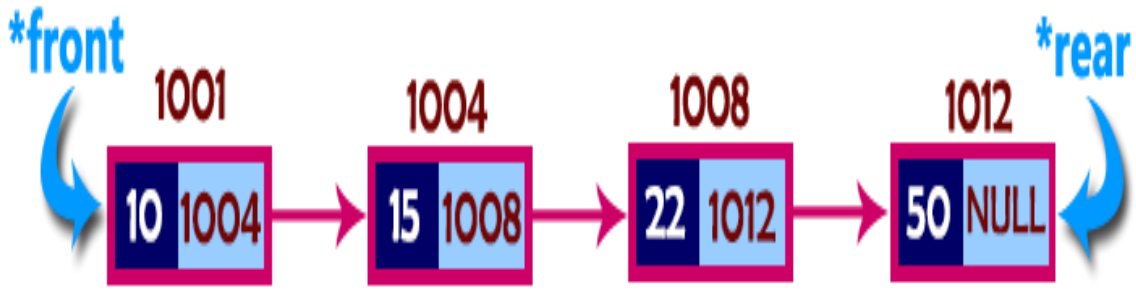
We can use the following steps to display the elements of a queue.

- **Step 1 -** Check whether **queue** is **EMPTY**. (**front == rear**)
- **Step 2 -** If it is **EMPTY**, then display **"Queue is EMPTY!!!"** and terminate the function.
- **Step 3 -** If it is **NOT EMPTY**, then define an integer variable '**i**' and set '**i** = **front+1**'.

- **Step 4 -** Display '**queue[i]**' value and increment '**i**' value by one (**i++**). Repeat the same until '**i**' value reaches to **rear** (**i** <= **rear**)

Queue Using Linked List

In linked list implementation of a queue, the last inserted node is always pointed by 'rear' and the first node is always pointed by 'front'.



Operations

- **Step 1 -** Include all the **header files** which are used in the program. And declare all the **user defined functions**.
- **Step 2 -** Define a '**Node**' structure with two members **data** and **next**.
- **Step 3 -** Define two **Node** pointers '**front**' and '**rear**' and set both to **NULL**.
- **Step 4 -** Implement the **main** method by displaying Menu of list of operations and make suitable function calls in the **main** method to perform user selected operation.

enQueue(value) - Inserting an element into the Queue

To insert a new node into the queue,

- **Step 1 -** Create a **newNode** with given value and set '**newNode → next**' to **NULL**.
- **Step 2 -** Check whether queue is **Empty** (**rear** == **NULL**)
- **Step 3 -** If it is **Empty** then, set **front** = **newNode** and **rear** = **newNode**.

- **Step 4 -** If it is **Not Empty** then, set **rear → next** = **newNode** and **rear** = **newNode**.

deQueue() - Deleting an Element from Queue

To delete a node from the queue,

- **Step 1 -** Check whether **queue** is **Empty** (**front == NULL**).
- **Step 2 -** If it is **Empty**, then display **"Queue is Empty!!! Deletion is not possible!!!"** and terminate from the function
- **Step 3 -** If it is **Not Empty** then, define a Node pointer '**temp**' and set it to '**front**'.
- **Step 4 -** Then set '**front** = **front → next**' and delete '**temp**' (**free(temp)**).

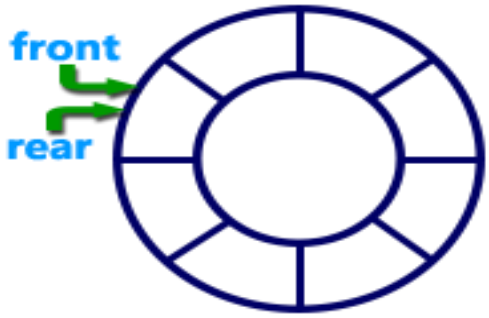
display() - Displaying the elements of Queue

To display the elements (nodes) of a queue,

- **Step 1 -** Check whether queue is **Empty** (**front** == **NULL**).
- **Step 2 -** If it is **Empty** then, display **'Queue is Empty!!!'** and terminate the function.
- **Step 3 -** If it is **Not Empty** then, define a Node pointer **'temp'** and initialize with **front**.
- **Step 4 -** Display '**temp → data** --->' and move it to the next node. Repeat the same until '**temp**' reaches to '**rear**' (**temp → next** != **NULL**).
- **Step 5 -** Finally! Display '**temp → data** ---> **NULL**'.

2.7 CIRCULAR QUEUES

A circular queue is a linear data structure in which the operations are performed based on FIFO (First In First Out) principle and the last position is connected back to the first position to make a circle.



- **Step 1 -** Include all the **header files** which are used in the program and define a constant **'SIZE'** with specific value.
- **Step 2 -** Declare all **user defined functions** used in circular queue implementation.
- **Step 3 -** Create a one dimensional array with above defined SIZE (**int cQueue[SIZE]**)
- **Step 4 -** Define two integer variables **'front'** and '**rear**' and initialize both with **'-1'**. (**int front = -1, rear = -1**)
- **Step 5 -** Implement main method by displaying menu of operations list and make suitable function calls to perform operation selected by the user on circular queue.

enQueue(value) - Inserting value into the Circular Queue

- **Step 1 -** Check whether **queue** is **FULL**. (**(rear == SIZE-1 && front == 0) || (front == rear+1)**)
- **Step 2 -** If it is **FULL**, then display **"Queue is FULL!!! Insertion is not possible!!!"** and terminate the function.
- **Step 3 -** If it is **NOT FULL**, then check **rear == SIZE - 1 && front != 0** if it is **TRUE**, then set **rear = -1**.

- **Step 4 -** Increment **rear** value by one (**rear++**), set **queue[rear]** = **value** and check '**front == -1**' if it is **TRUE**, then set **front = 0**.

deQueue() - Deleting a value from the Circular Queue

- **Step 1 -** Check whether **queue** is **EMPTY**. (**front == -1 && rear == -1**)
- **Step 2 -** If it is **EMPTY**, then display **"Queue is EMPTY!!! Deletion is not possible!!!"** and terminate the function.
- **Step 3 -** If it is **NOT EMPTY**, then display **queue[front]** as deleted element and increment the **front** value by one (**front ++**). Then check whether **front == SIZE**, if it is **TRUE**, then set **front = 0**. Then check whether both **front - 1** and **rear** are equal (**front -1** == **rear**), if it **TRUE**, then set both **front** and **rear** to '**-1**' (**front** = **rear** = **-1**).

display() - Displays the elements of a Circular Queue

We can use the following steps to display the elements of a circular queue...

- **Step 1 -** Check whether **queue** is **EMPTY**. (**front == -1**)
- **Step 2 -** If it is **EMPTY**, then display **"Queue is EMPTY!!!"** and terminate the function.
- **Step 3 -** If it is **NOT EMPTY**, then define an integer variable '**i**' and set '**i** = **front**'.
- **Step 4 -** Check whether '**front <= rear**', if it is **TRUE**, then display '**queue[i]**' value and increment '**i**' value by one (**i++**). Repeat the same until '**i <= rear**' becomes **FALSE**.

- **Step 5 -** If '**front <= rear**' is **FALSE**, then display '**queue[i]**' value and increment '**i**' value by one (**i++**). Repeat the same until'**i <= SIZE - 1**' becomes **FALSE**.
- **Step 6 -** Set **i** to **0**.
- **Step 7 -** Again display '**cQueue[i]**' value and increment **i** value by one (**i++**). Repeat the same until '**i <= rear**' becomes **FALSE**.

UNIT III

TREES

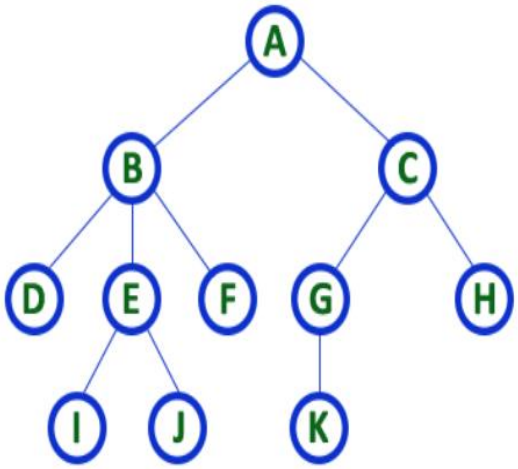
OUTLINE

TREES

3.1 INTRODUCTION

- Tree is a non-linear data structure which organizes data in hierarchical structure and this is a recursive definition.
- Tree data structure is a collection of data (Node) which is organized in hierarchical structure recursively
- In tree data structure, every individual element is called as Node. Node in a tree data structure stores the actual data of that particular element and link to next element in hierarchical structure.

  In a tree data structure, if we have N number of nodes then we can have a maximum of N-1 number of links.

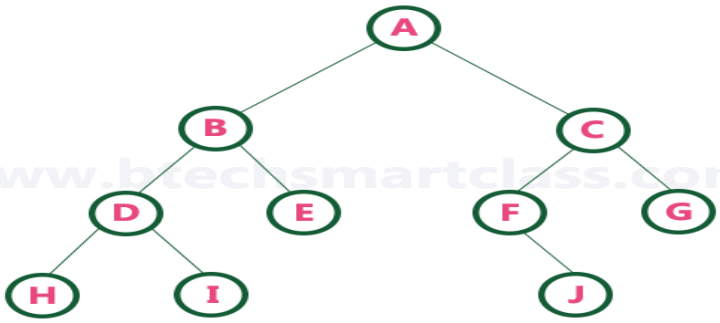


TREE with 11 Nodes and 10 Edges

- **In any tree with 'N' nodes there will be maximum of 'N-1' edges.**
- **In a tree every individual element is called as 'NODE'**

3.2 BINARY TREES

- In a normal tree, every node can have any number of children. A binary tree is a special type of tree data structure in which every node can have a maximum of 2 children. One is known as a left child and the other is known as right child.
- A tree in which every node can have a maximum of two children is called Binary Tree.
- In a binary tree, every node can have either 0 children or 1 child or 2 children but not more than 2 children.

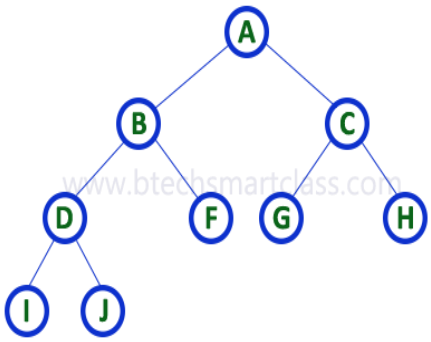


Types of Binary trees:

1. Strictly Binary Tree
2. Complete Binary Tree
3. Extended Binary Tree

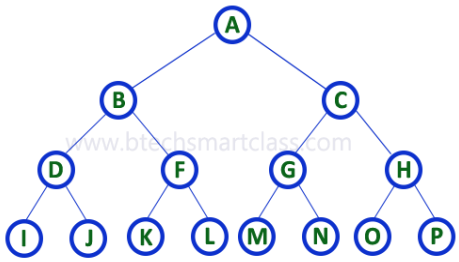
1. Strictly Binary Tree

- In strictly binary tree, every node should have exactly two children or none. That means every internal node must have exactly two children.
- Strictly binary tree is also called as Full Binary Tree or Proper Binary Tree or 2-Tree.



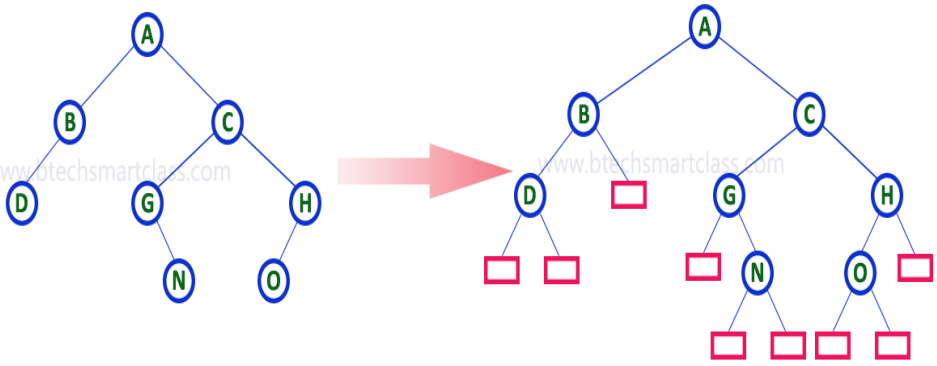
2. Complete Binary Tree

- In complete binary tree all the nodes must have exactly two children and at every level of complete binary tree there must be 2^{level} number of nodes.
- For example at level 2 there must be $2^2 = 4$ nodes and at level 3 there must be $2^3 = 8$ nodes.
- Complete binary tree is also called as Perfect Binary Tree.



3. Extended Binary Tree

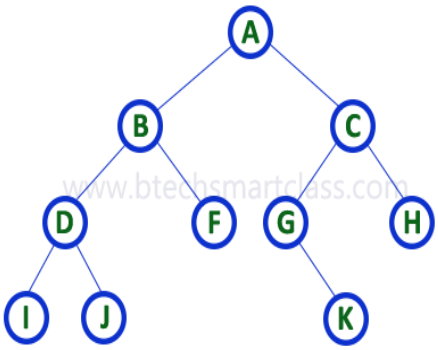
- A binary tree can be converted into Full Binary tree by adding dummy nodes to existing nodes wherever required.
- The full binary tree obtained by adding dummy nodes to a binary tree is called as Extended Binary Tree.



3.3 REPRESENTATION OF BINARY TREES

A binary tree data structure is represented using two methods.

1. Array Representation
2. Linked List Representation



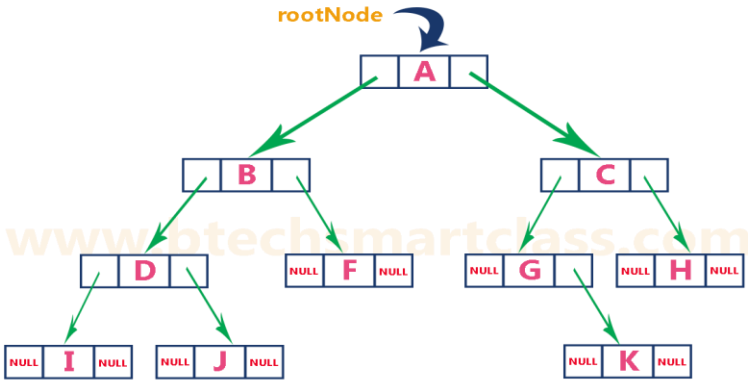
1. Array Representation of Binary Tree

- In array representation of a binary tree, we use one-dimensional array (1-D Array) to represent a binary tree.
- To represent a binary tree of depth 'n' using array representation, we need one dimensional array with a maximum size of 2n + 1.

2. Linked List Representation of Binary Tree

- We use a double linked list to represent a binary tree. In a double linked list, every node consists of three fields.
- First field for storing left child address
- Second for storing actual data
- Third for storing right child address.

Structure:

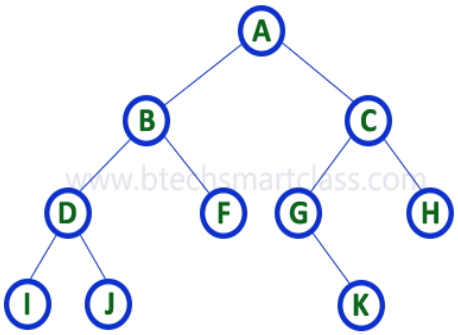


3.4 BINARY TREE TRAVERSAL

Here are three types of binary tree traversals.

1. **In - Order Traversal**
2. **Pre - Order Traversal**
3. **Post - Order Traversal**

Consider the following binary tree...



3.5 RECURSIVE PROCEDURES OF TRAVERSAL METHODS

1. In - Order Traversal (leftChild - root - rightChild)

- In the above example of a binary tree, first we try to visit left child of root node 'A', but A's left child 'B' is a root node for left subtree.
- so we try to visit its (B's) left child 'D' and again D is a root for subtree with nodes D, I and J.
- So we try to visit its left child 'I' and it is the leftmost child. So first we visit **'I'** then go for its root node **'D'** and later we visit D's right child **'J'**.
- With this we have completed the left part of node B.
- Then visit **'B'** and next B's right child **'F'** is visited.
- With this we have completed left part of node A.
- Then visit root node **'A'**.
- With this we have completed left and root parts of node A.
- Then we go for the right part of the node A.
- In right of A again there is a subtree with root C.
- So go for left child of C and again it is a subtree with root G.
- But G does not have left part so we visit **'G'** and then visit G's right child K.
- With this we have completed the left part of node C.

- Then visit root node **'C'** and next visit C's right child **'H'** which is the rightmost child in the tree.

So we stop the process.

In-Order Traversal for above example of binary tree is

I - D - J - B - F - A - G - K - C – H

2. Pre - Order Traversal (root - leftChild - rightChild)

- In the above example of binary tree, first we visit root node **'A'** then visit its left child **'B'** which is a root for D and F.
- So we visit B's left child **'D'** and again D is a root for I and J.
- So we visit D's left child **'I'** which is the leftmost child.
- So next we go for visiting D's right child **'J'**.
- With this we have completed root, left and right parts of node D and root, left parts of node B. Next visit B's right child **'F'**.
- With this we have completed root and left parts of node A.
- So we go for A's right child **'C'** which is a root node for G and H.
- After visiting C, we go for its left child **'G'** which is a root for node K.
- So next we visit left of G, but it does not have left child so we go for G's right child **'K'**.
- With this, we have completed node C's root and left parts.
- Next visit C's right child **'H'** which is the rightmost child in the tree.

So we stop the process.

That means here we have visited in the order of **A-B-D-I-J-F-C-G-K-H** using Pre-Order Traversal.

3. Post - Order Traversal (leftChild - rightChild - root)

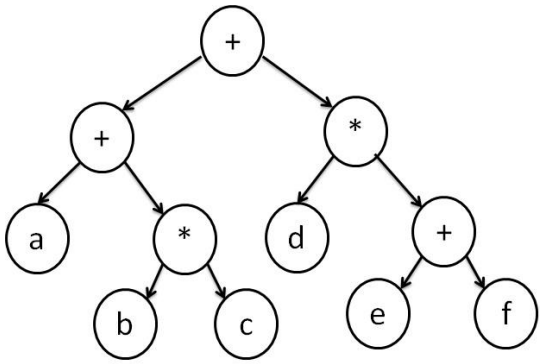
- In Post-Order traversal, the root node is visited after left child and right child.
- In this traversal, left child node is visited first, then its right child and then its root node.
- This is recursively performed until the right most node is visited.

  Here we have visited in the order of **I - J - D - F - B - K - G - H - C - A** using Post-Order Traversal.

3.6 EXPRESSION TREES

- Expression Tree is used to represent expressions.
- An expression and expression tree shown below

  a + (b * c) + d * (e + f)



Expressions may includes constants value as well as variables

- `a * 6`
- `16`
- `(a^2)+(b^2)+(2 * a * b)`
- `(a/b) + (c)`

```
- m * (c ^ 2)
```

It is quite common to use parenthesis in order to ensure correct evaluation of expression as shown above

There are different types of expression formats:

- Prefix expression
- Infix expression and
- Postfix expression

Expression Tree is a special kind of binary tree with the following properties:

- Each leaf is an operand. Examples: a, b, c, 6, 100
- The root and internal nodes are operators. Examples: +, -, *, /, ^
- Subtrees are subexpressions with the root being an operator.

Traversal Techniques

Inorder Traversal

We can produce an infix expression by recursively printing out

- the left expression,
- the root, and
- the right expression.

Postorder Traversal

The postfix expression can be evaluated by recursively printing out

- the left expression,
- the right expression and
- then the root

Preorder Traversal

We can also evaluate prefix expression by recursively printing out:

- the root,
- the left expressoion and
- the right expression.

If we apply all these strategies to the sample tree above, the outputs are:

- **Infix expression**:

```
(a+(b*c))+(d*(e + f))
```

- **Postfix Expression:**

```
a b c * + d e f + * +
```

- **Prefix Expression:**

```
+ + a * b c * d + e f
```

Construction of Expression Tree

We consider that a postfix expression is given as an input for constructing an expression tree. Following are the step to construct an expression tree:

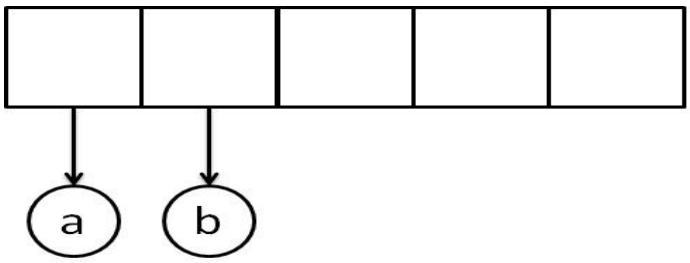
1. Read one symbol at a time from the postfix expression.
2. Check if the symbol is an operand or operator.
3. If the symbol is an operand, create a one node tree and pushed a pointer onto a stack

4. If the symbol is an operator, pop two pointer from the stack namely T_1 & T_2 and form a new tree with root as the operator, T_1 & T_2 as a left and right child
5. A pointer to this new tree is pushed onto the stack

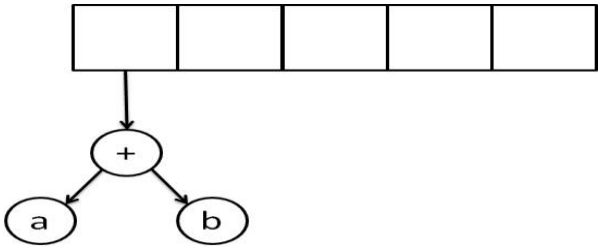
Example

The input is: a b + c *

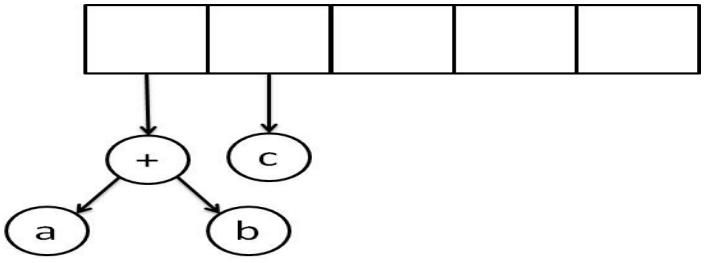
1. The first two symbols are operands, we create one-node tree and push a pointer to them onto the stack.



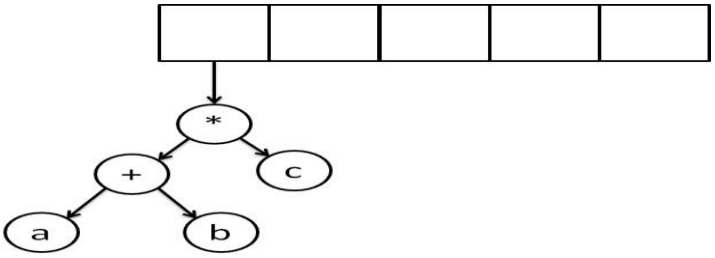
2. Next, read a**'+'** symbol, so two pointers to tree are popped, a new tree is formed and push a pointer to it onto the stack.



3. Next, **'c'** is read, we create one node tree and push a pointer to it onto the stack.



4. Finally, the last symbol is read ' * ', we pop two tree pointers and form a new tree with a, ' * ' as root, and a pointer to the final tree remains on the stack.

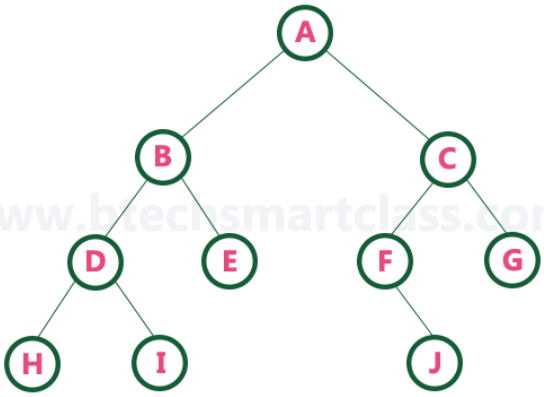


3.7 THREADED TREES

1. A. J. Perlis and C. Thornton have proposed new binary tree called "**Threaded Binary Tree**", which makes use of NULL pointers to improve its traversal process.

2. In a threaded binary tree, NULL pointers are replaced by references of other nodes in the tree. These extra references are called as **threads**.

3. If there is no in-order predecessor or in-order successor, then it points to the root node.



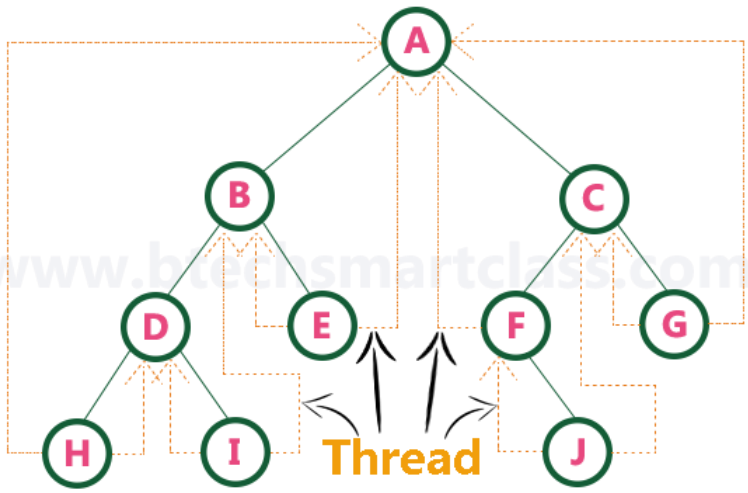
To convert the above example binary tree into a threaded binary tree, first find the in-order traversal of that tree...

In-order traversal of above binary tree...

H - D - I - B - E - A - F - J - C - G

- When we represent the above binary tree using linked list representation, nodes **H, I, E, F, J** and **G** left child pointers are NULL.
- This NULL is replaced by address of its in-order predecessor respectively (I to D, E to B, F to A, J to F and G to C), but here the node H does not have its in-order predecessor, so it points to the root node A and nodes **H, I, E, J** and **G** right child pointers are NULL.
- These NULL pointers are replaced by address of its in-order successor respectively (H to D, I to B, E to A, and J to C), but here the node G does not have its in-order successor, so it points to the root node A.

Above example binary tree is converted into threaded binary tree as follows.



In the above figure, threads are indicated with dotted links.

3.8 APPLICATION OF TREES

1. Binary Search Trees (BSTs) are used to quickly check whether an element is present in a set or not.

2. Heap is a kind of tree that is used for heap sort.

3 .A modified version of a tree called Tries is used in modern routers to store routing information.

4. Most popular databases use B-Trees and T-Trees, which are variants of the tree structure we learned above to store their data

5. Compilers use a syntax tree to validate the syntax of every program you write.

DATA STRUCTURE AND COMPUTER ALGORITHMS–7BCE3C1

UNIT–4

4.1. ALGORITHM

4.1.1WHAT IS AN ALGORITHM?

- An *algorithm* is a finite set of instructions which, if followed, accomplish a particular task. In addition every algorithm must satisfy the following criteria:

1. input: there are zero or more quantities which are externally supplied;
2. output: at least one quantity is produced;
3. definiteness: each instruction must be clear and unambiguous;
4. finiteness: if we trace out the instructions of an algorithm, then for all cases the algorithm will terminate after a finite number of steps;
5. effectiveness: every instruction must be sufficiently basic that it can in principle be carried out by a person using only pencil and paper. It is not enough that each operation be definite, but it must also be feasible.

==========

4.1.2. ALGORITHM SPECIFICATIONS

Recursive Algorithms

Recursion is similar to the method of induction which is often used to prove mathematical statements. In mathematical induction, a statement about integers (e.g., the sum of the first n positive integers is n (n +1)12) is proved by showing that the statement can be proved for integer k if it is assumed to be true for integer k -1.

To understand a recursive function, you must.

(1) Formulate in your mind a statement of what it is that the function is supposed to do, for a given input.

(2) Verify that the function does achieve its goal if the recursive invocations to itself do what they are supposed to.

(3) Ensure that a finite number of recursive invocations of the function eventually lead to an invocation which satisfies the terminating condition (otherwise, the function will keep calling itself and not terminate!).·

(4) The function should perform the correct computations if the terminating condition is encountered. .

Example 1.5 [Permutation generator]:·

• Given a set of n > 1 elements, the problem is to print all possible permutations of this set. For example if the set is {a, b, c}, then the set of permutations is {(a, b, c), (a, c, b),(b, a, c),(b, c, a),(c, a, b),(c, b, a)}. It is easy to see that given n elements, there are n ! different permutations.

• A simple algorithm can be obtained by looking at the the case of four elements (a,b,c,d). The answer can be constructed by writing.

• a followed by all permutations of (b,c,d)
(2) b followed by all permutations of (a,c,d)
(3) c followed by all permutations of (a,b,d)
(4) d followed by all permutations of (a,b,c)

• The expression “followed by all permutations” is the clue to ‘recursion. It implies that we can solve the problem for a set with n elements if we have an algorithm that works on n – 1 elements. These observations lead to Program 1.11, which is invoked by perm (a, 0, n).

==========

4.1.3. PERFORMANCE ANALYSIS:

SPACE COMPLEXITY:

- Amount of memory space required to solve the algorithm.

i)Fixed Space Requirements (C)

- Independent of the characteristics of the inputs and outputs
  instruction space
- space for simple variables, fixed-size structured variable, constants

ii)Variable Space Requirements ($S_P(I)$)

depend on the instance characteristic I

- number, size, values of inputs and outputs associated with I
- recursive stack space, formal parameters, local variables, return address

Space Complexity

$$S(P)=C+S_P(I)$$

*Program 1.10: Iterative function for summing a list of numbers (p.20)

```
float sum(float list[ ], int n)
{
  float tempsum = 0;
inti;
  for (i = 0; i<n; i++)
tempsum += list [i];
  return tempsum;
}
```

*Program 1.11: Recursive function for summing a list of numbers (p.20)

```
float rsum(float list[ ], int n)
{
   if (n) return rsum(list, n-1) + list[n-1];
   return 0;
 }
```

Time Complexity :

- Amount of compilation time and run time to execute algorithm

- A *program step* is a syntactically or semantically meaningful program segment whose execution time is independent of the instance characteristics.

$$T_P(n)=c_a ADD(n)+c_s SUB(n)+c_l LDA(n)+c_{st} STA(n)$$

*Program 1.12: Program 1.10 with count statements (p.23)

```
float sum(float list[ ], int n)
{
   float tempsum = 0; count++; /* for assignment */
inti;
   for (i = 0; i< n; i++) {
        count++;            /*for the for loop */
tempsum += list[i]; count++;  /* for assignment */
   }
   count++;         /* last execution of for */
   return tempsum;
   count++;         /* for return */
}
```

*Program 1.13: Simplified version of Program 1.12 (p.23)

```
float sum(float list[ ], int n)
{
   float tempsum = 0;
inti;
   for (i = 0; i< n; i++)
       count += 2;
   count += 3;
   return 0;
}
```

==========

4.2. DIVIDE AND CONQUER

4.2.1INTRODUCTION

General Method

Divide and conquer is a design strategy which is well known to breaking down efficiency barriers. When the method applies, it often leads to a large improvement in time complexity. For example, from O (n^2) to O (n log n) to sort the elements.

Divide and conquer strategy is as follows: divide the problem instance into two or more smaller instances of the same problem, solve the smaller instances recursively, and assemble the

solutions to form a solution of the original instance. The recursion stops when an instance is reached which is too small to divide. When dividing the instance, one can either use whatever division comes most easily to hand or invest time in making the division carefully so that the assembly is simplified.

Divide and conquer algorithm consists of two parts:

Divide : Divide the problem into a number of sub problems. The sub problems are solved recursively.

Conquer : The solution to the original problem is then formed from the solutions to the sub problems (patching together the answers).

Traditionally, routines in which the text contains at least two recursive calls are called divide and conquer algorithms, while routines whose text contains only one recursive call are not. Divide-and-conquer is a very powerful use of recursion.

```
DANDC (P)
{
if
SMAL
L (P)
then
return
S (p);
else
{
divide p into smaller instances p1, p2, …. Pk, k · 1;
apply DANDC to each of these sub problems;
return (COMBINE (DANDC (p1) , DANDC (p2),….DANDC (pk));
}
}
```

SMALL (P) is a Boolean valued function which determines whether the input size is small enough so that the answer can be computed without splitting. If this is so function „S" is invoked otherwise, the problem „p" into smaller sub problems. These sub problems p_1, p_2, . . . , p_k are solved by recursive application of DANDC.

==========

4.2.2. BINARY SEARCH

In Binary search we jump into the middle of the file, where we find key a[mid], and compare „x" with a[mid]. If x = a[mid] then the desired record has been found. If x < a[mid] then „x" must be in that portion of the file that precedes a[mid], if there at all. Similarly, if a[mid] > x, then further search is only necessary in that past of the file which follows a[mid]. If we use recursive procedure of finding the middle key a[mid] of the un-searched portion of a file, then every un-successful comparison of „x" with a[mid] will eliminate roughly half the un-searched portion from consideration.

Since the array size is roughly halved often each comparison between „x" and a[mid], and since an array of length „n" can be halved only about $\log_2 n$ times before

reaching a trivial length, the worst case complexity of Binary search is about $\log_2 n$

Algorithm Algorithm

BINSRCH (a, n, x)
// array a(1 : n) of elements in increasing order, n · 0,
// determine whether „x" is present, and if so, set j such that x = a(j)
// else return j

```
{
low :=1 ; high :=n ;
while (low ≤high) do
{
mid :=|(low + high)/2|
if (x < a [mid]) then high:=mid - 1;
else if (x > a [mid]) then low:= mid + 1
                                    else return mid;
}
return 0;
}
```

low and *high* are integer variables such that each time through the loop either „x" is found or *low* is increased by at least one or *high* is decreased by at least one. Thus we have two sequences of integers approaching each other and eventually *low* will become greater than *high* causing termination in a finite number of steps if „x" is not present.

Example for Binary Search

Let us illustrate binary search on the following 9 elements:

Index	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
Elemer	-	-	0	7	9	2	5	8	1

The number of comparisons required for searching different elements is as follows:

1.Searching for x = 101

Number of comparisons = 4

2.Searching for x = 82

Number of comparisons = 3

3.Searching for x = 42

Number of comparisons = 4

4.Searching for x = -14

Number of comparisons = 3

low high mid
1 9 5
6 9 7
8 9 8
9 9 9
found

low high mid
1 9 5
6 9 7
8 9 8
found

low high mid
1 9 5
6 9 7
6 6 6
7 6 not found

low high mid
1 9 5
1 4 2
1 1 1
2 1 not found

Continuing in this manner the number of element comparisons needed to find each of nine elements is:

Index	*1*	*2*	*3*	*4*	*5*	*6*	*7*	*8*	*9*
Elements	-	-	0	7	9	2	5	8	1
Compariso	*3*	*2*	*3*	*4*	*1*	*3*	*2*	*3*	*4*

No element requires more than 4 comparisons to be found. Summing the comparisons needed to find all nine items and dividing by 9, yielding 25/9 or approximately 2.77 comparisons per successful search on the average.

There are ten possible ways that an un-successful search may terminate depending upon the value of x.

==========

4.2.3. FINDING MAXIMUM AND MINIMUM

1. Let us consider simple problem that can be solved by the divide-and conquer technique.
2. The problem is to find the maximum and minimum value in a set of 'n' elements.
3. By comparing numbers of elements, the time complexity of this algorithm can be analyzed.
4. Hence, the time is determined mainly by the total cost of the element comparison.
5. comparison.

```
Algorithm straight MaxMin (a, n, max, min)
// Set max to the maximum & min to the minimum of a [1: n]
{
Max = Min = a [1];
For i = 2 to n do
{
If (a [i] > Max) then Max = a [i];
If (a [i] < Min) then Min = a [i];
}}
```

Explanation:

a. Straight MaxMin requires 2(n-1) element comparisons in the best, average & worst cases.

b. By realizing the comparison of a [i]max is false, improvement in a algorithm can be done.

c. Hence we can replace the contents of the for loop by, If (a [i]> Max) then Max = a [i]; Else if (a [i]< 2(n-1)

d. On the average a[i] is > max half the time, and so, the avg. no. of comparison is 3n/2-1.

A Divide and Conquer Algorithm for this problem would proceed as follows:

a. Let P = (n, a [i],……,a [j]) denote an arbitrary instance of the problem.

b. Here 'n' is the no. of elements in the list (a [i],….,a[j]) and we are interested in finding the maximum and minimum of the list.

c. If the list has more than 2 elements, P has to be divided into smaller instances.

d. For example, we might divide 'P' into the 2 instances, P1=([n/2],a[1],……..a[n/2]) & P2= (n-[n/2], a[[n/2]+1],….., a[n]) After having divided 'P' into 2 smaller sub problems, we can solve them by recursively invoking the same divide-and-conquer algorithm.

Algorithm:

```
MaxMin (i, j, max, min)
 // a [1: n] is a global array, parameters i& j are integers, 1<= i<=j <=n. The effect is
to4.
// Set max & min to the largest & smallest value 5 in a [i: j], respectively.
{
If (i=j) then Max = Min = a[i];
Else if (i=j-1) then
{
if (a[i] < a[j]) then
        {
                Max = a[j];
                Min = a[i];
        }
        Else
        {
                Max = a[i];
                Min = a[j];
        }
}Else
{
Mid = (i + j) / 2;
MaxMin (I, Mid, Max, Min);
MaxMin (Mid +1, j, Max1, Min1);
If (Max < Max1) then Max = Max1;
If (Min > Min1) then Min = Min1;
}}
The procedure is initially invoked by the statement, MaxMin (1, n, x, y)
```

Example:

A	1	2	3	4	5	6	7	8	9
Values	22	13	-5	-8	15	60	17	31	47

Tree Diagram:

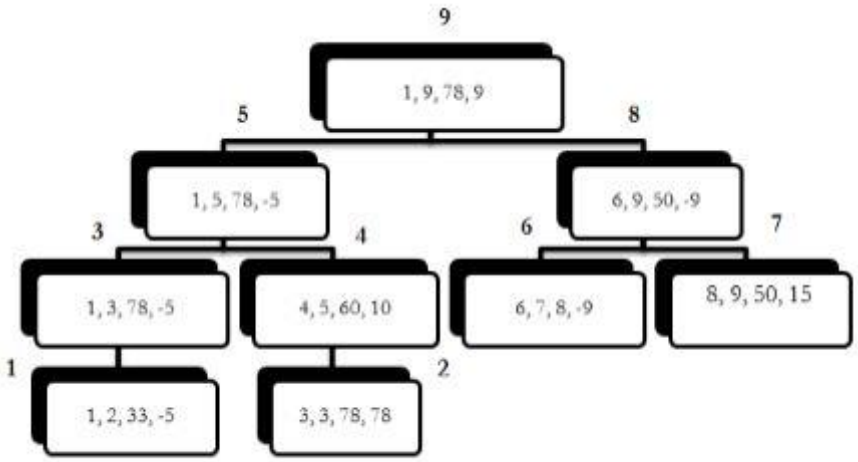
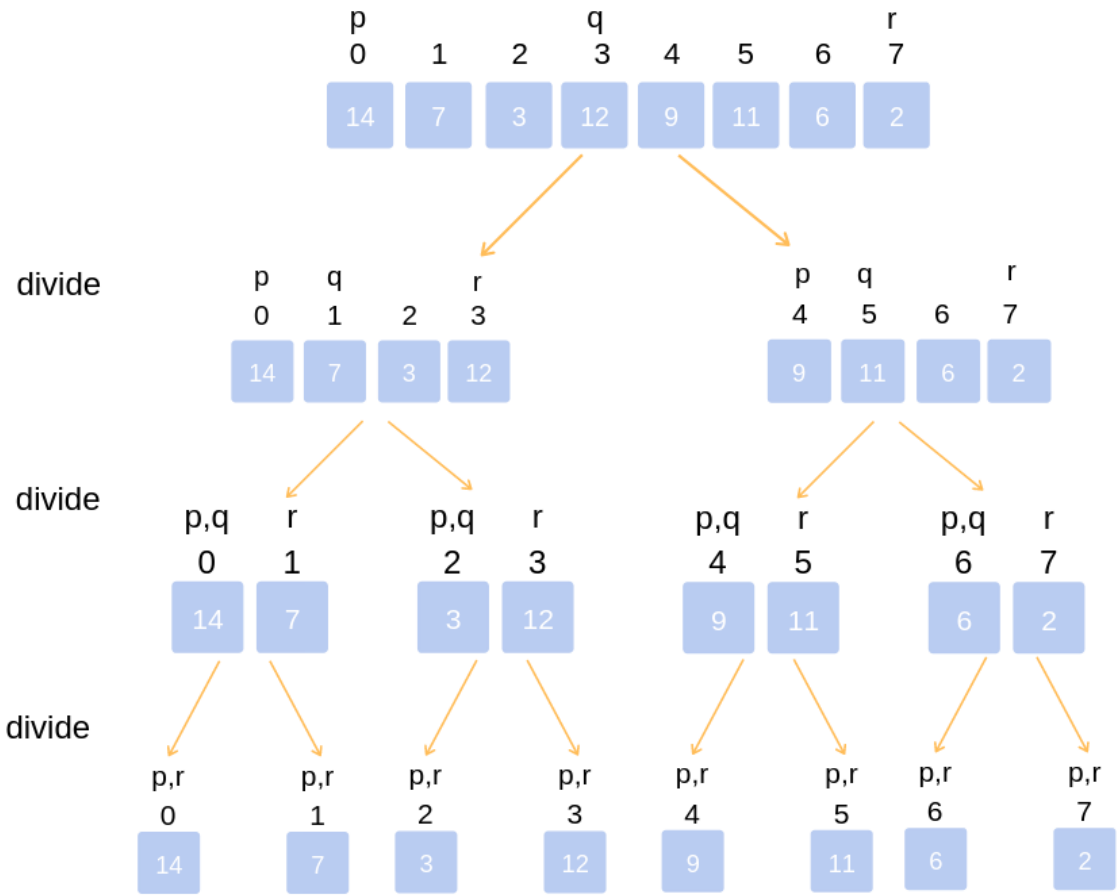


Figure 4

==========

4.2.4. MERGE SORT

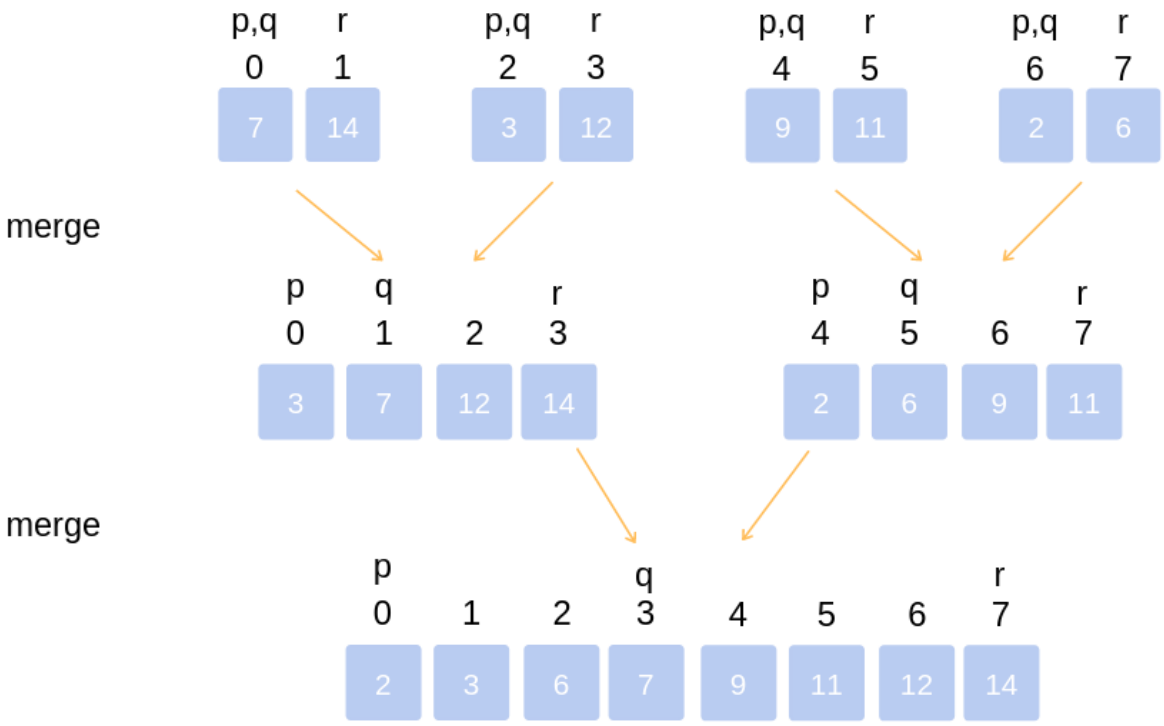
- Merge sort is one of the most efficient sorting algorithms. It works on the principle of Divide and Conquer.
- Merge sort repeatedly breaks down a list into several sublists until each sublist consists of a single element and merging those sublists in a manner that results into a sorted list.
- Example: Let us consider an example to understand the approach better.
- Divide the unsorted list into n sublists, each comprising 1 element (a list of 1 element is supposed sorted).



- Repeatedly merge sublists to produce newly sorted sublists until there is only 1 sublist remaining. This will be the sorted list.

Merging of two lists done as follows:

- The first element of both lists is compared. If sorting in ascending order, the smaller element among two becomes a new element of the sorted list.
- This procedure is repeated until both the smaller sublists are empty and the newly combined sublist covers all the elements of both the sublists.




```
voidmerge(int*Arr,intstart,intmid,intend){
        // create a temp array
        inttemp[end-start+1];

        // crawlers for both intervals and for temp
        inti=start,j=mid+1,k=0;

        // traverse both arrays and in each iteration add smaller of both elem
ents in temp
        while(i<=mid&&j<=end){
                if(Arr[i]<=Arr[j]){
                        temp[k]=Arr[i];
                        k+=1;i+=1;
                }
                else{
                        temp[k]=Arr[j];
                        k+=1;j+=1;
                }
        }

        // add elements left in the first interval
        while(i<=mid){
                temp[k]=Arr[i];
                k+=1;i+=1;
        }
```

```
        // add elements left in the second interval
        while(j<=end){
                temp[k]=Arr[j];
                k+=1;j+=1;
        }

        // copy temp to original interval
        for(i=start;i<=end;i+=1){
                Arr[i]=temp[i-start]
        }
}

// Arr is an array of integer type
// start and end are the starting and ending index of current interval of Arr

voidmergeSort(int*Arr,intstart,intend)
{

        if(start<end){
                intmid=(start+end)/2;
                mergeSort(Arr,start,mid);
                mergeSort(Arr,mid+1,end);
                merge(Arr,start,mid,end);
        }
}
```

==========

4.2.5.QUICK SORT:

- Technically, quick sort follows the below steps:
  Step 1 − Make any element as pivot
  Step 2 − Partition the array on the basis of pivot
  Step 3 − Apply quick sort on left partition recursively
  Step 4 − Apply quick sort on right partition recursively

- Consider the following array: 50, 23, 9, 18, 61, 32.
- the pivot (32) comes at its actual position and all elements to its left are lesser, and all elements to the right are greater than itself.
- **Step 2**: The main array after the first step becomes

- 23, 9, 18, 32, 61, 50

- **Step 3**: Now the list is divided into two parts:
- Sublist before pivot element
- Sublist after pivot element
- **Step 4**: Repeat the steps for the left and right sublists recursively. The final array thus becomes
  9, 18, 23, 32, 50, 61.

```
void swap(int *a, int *b)
{
        int temp;
        temp = *a;
        *a = *b;
        *b = temp;
}

// Partitioning the array on the basis of values at high as pivot value.
int Partition(int a[], int low, int high)
{
        int pivot, index, i;
        index = low;
        pivot = high;


        for(i=low; i< high; i++)
        {
                if(a[i] < a[pivot])
                {


        return index;
}



intQuickSort(int a[], int low, int high)
{
        intpindex;
        if(low < high)
                pindex = RandomPivotPartition(a, low, high);

                QuickSort(a, low, pindex-1);
                QuickSort(a, pindex+1, high);
        }
```

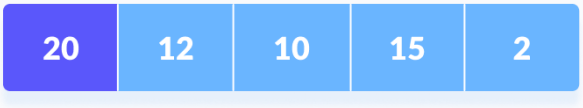
```
        return 0;
}

int main()
{
        int n, i;
        cout<<"\nEnter the number of data elements to be sorted: ";
        cin>>n;

        intarr[n];
        for(i = 0; i< n; i++)
        {
                cout<<"Enter element "<<i+1<<": ";
                cin>>arr[i];
        }

        QuickSort(arr, 0, n-1);


        cout<<"\nSorted Data ";
        for (i = 0; i< n; i++)
        cout<<"->"<<arr[i];

        return 0;
}
```

4.2.6 **SELECTION SORT:**

- Selection sort is an algorithm that selects the smallest element from an unsorted list in each iteration and places that element at the beginning of the unsorted list.

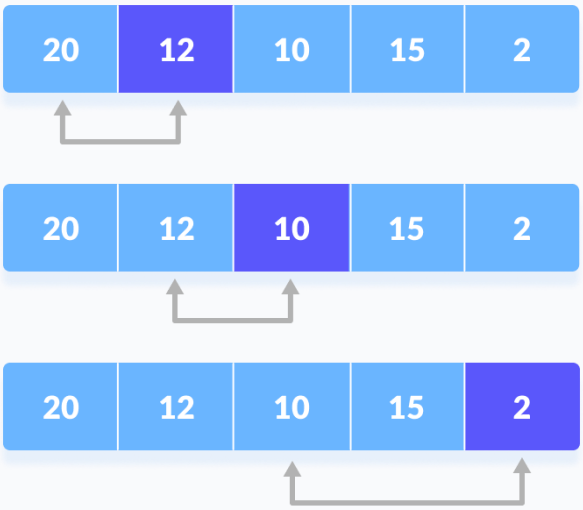
How Selection Sort Works?

1. Set the first element as minimum.



Select first element as minimum

2. Compare minimum with the second element. If the second element is smaller than minimum, assign the second element as minimum.
3. Compare minimum with the third element. Again, if the third element is smaller, then assign minimum to the third element otherwise do nothing. The process goes on until the last element.



4. Compare minimum with the remaining elements
5. After each iteration, minimum is placed in the front of the unsorted list.

Swap the first with minimum

6. For each iteration, indexing starts from the first unsorted element. Step 1 to 3 are repeated until all the elements are placed at their correct positions.

ALGORITHM FOR SELECTION SORT

```
voidselectionSort(intarray[], int size){
for (int step = 0; step < size - 1; step++) {
intmin_idx = step;
for (int i = step + 1; i< size; i++) {

// To sort in descending order, change > to < in this line.
// Select the minimum element in each loop.
if (array[i] <array[min_idx])
min_idx = i;
   }

// put min at the correct position
   swap(&array[min_idx], &array[step]);
 }
}
voidswap(int *a, int *b){
int temp = *a;
 *a = *b;
 *b = temp;
}

// function to print an array
voidprintArray(intarray[], int size){
for (inti = 0; i< size; i++) {
cout<<array[i] <<" ";
 }
cout<<endl;
}
```

4.2.7. Strassen’ s Matrix Multiplication

- For multiplying the two 2*2 dimension matrices **Strassen's** used some formulas in which there are seven multiplication and eighteen addition,

subtraction, and in brute force algorithm, there is eight multiplication and four addition.

- The utility of Strassen's formula is shown by its asymptotic superiority when order **n** of matrix reaches infinity. Let us consider two matrices **A** and **B**, **n*n** dimension, where **n** is a power of two.
- . It can be observed that we can contain four **n/2*n/2** submatrices from **A**, **B** and their product **C**. **C** is the resultant matrix of **A** and **B**.

Procedure of Strassen matrix multiplication

There are some procedures:

1. Divide a matrix of order of 2*2 recursively till we get the matrix of 2*2.
2. Use the previous set of formulas to carry out 2*2 matrix multiplication.
3. In this eight multiplication and four additions, subtraction are performed.
4. Combine the result of two matrixes to find the final product or final matrix.

Formulas for Stassen's matrix multiplication

In **Strassen's matrix multiplication** there are seven multiplication and four addition, subtraction in total.

1.	D1 = (a11 + a22) (b11 + b22)
2.	D2 = (a21 + a22).b11
3.	D3 = (b12 – b22).a11
4.	D4 = (b21 – b11).a22
5.	D5 = (a11 + a12).b22
6.	D6 = (a21 – a11) . (b11 + b12)
7.	D7 = (a12 – a22) . (b21 + b22)
C11 = d1 + d4 – d5 + d7	
C12 = d3 + d5	
C21 = d2 + d4	
C22 = d1 + d3 – d2 – d6	

Algorithm for Strassen's matrix multiplication

Algorithm Strassen(n, a, b, d)

```
begin
        If n = threshold then compute
```

```
                    C = a * b is a conventional matrix.
          Else
                    Partition a into four sub matrices  a11, a12, a21, a22.
                    Partition b into four sub matrices b11, b12, b21, b22.
                    Strassen ( n/2, a11 + a22, b11 + b22, d1)
                    Strassen ( n/2, a21 + a22, b11, d2)
                    Strassen ( n/2, a11, b12 – b22, d3)
                    Strassen ( n/2, a22, b21 – b11, d4)
                    Strassen ( n/2, a11 + a12, b22, d5)
                    Strassen (n/2, a21 – a11, b11 + b22, d6)
                    Strassen (n/2, a12 – a22, b21 + b22, d7)

                    C = d1+d4-d5+d7     d3+d5
                    d2+d4          d1+d3-d2-d6

          end if

          return (C)
end.
```

DATA STRUCTURE AND COMPUTER ALGORITHMS-7BCE3C1

UNIT-5

5.1 GREEDY METHOD :

5.1.1. INTRODUCTION:

- The greedy method is a simple strategy of progressively building up a solution,one element at a time, by choosing the best possible element at each stage. At each stage, a decision is made regarding whether or not a particular input is in an optimal solution.

- This is done by considering the inputs in an order determined by some selection procedure. If the inclusion of the next input, into the partially constructed optimal solution will result in an infeasible solution then this input is not added to the partial solution.
- The selection procedure itself is based on some optimization measure. Several optimization measures are plausible for a given problem. Most of them, however, will result in algorithms that generate sub-optimal solutions. This version of greedy technique is called *subset paradigm*. Some problems like Knapsack, Job sequencing with deadlines and minimum cost spanning trees are based on *subset paradigm.*

- For the problems that make decisions by considering the inputs in some order, each decision is made using an optimization criterion that can be computed using decisions already made. This version of greedy method is *ordering paradigm*. Some problems like optimal storage on tapes, optimal merge patterns and single source shortest path are based on *ordering paradigm.*

CONTROL ABSTRACTION

Algorithm Greedy (a, n)

```
// a(1 : n) contains the „n“ inputs
{
        solution  := □;                 // initialize the solution to empty
        for i:=1 to n do
        {
                x := select (a);
                if  feasible (solution, x) then
                        solution := Union (Solution, x);
        }
        return solution;
}
```

- Procedure Greedy describes the essential way that a greedy based algorithm will look, once a particular problem is chosen and the functions select, feasible and union are properly implemented.
- The function select selects an input from „a“, removes it and assigns its value to„x“.
  Feasible is a Boolean valued function, which determines if „x“ can be included intothe solution vector.
- The function Union combines „x“ with solution and updates the objective function.

5.1.2. KNAPSACK PROBLEM

- Let us apply the greedy method to solve the knapsack problem. We are given „n“
  objects and a knapsack.
- The object „i“ has a weight w_i and the knapsack has a capacity „m“. If a fraction x_i, $0 < x_i < 1$ of object i is placed into the knapsack then a profit of p_i x_i is earned. The objective is to fill the knapsack that maximizes the total profit earned.
- Since the knapsack capacity is „m“, we require the total weight of all chosen objects to be at most „m“. The problem is stated as:

$$\text{maximize} \sum_{i=1}^{n} p_i x_i$$

$$\text{subject to} \sum_{i=1}^{n} a_i x_i \leq M \qquad \text{where, } 0 \leq x_i \leq 1 \text{ and } 1 \leq i \leq n$$

The profits and weights are positive numbers.

Algorithm

If the objects are already been sorted into non-increasing order of p[i] / w[i] then the algorithm given below obtains solutions corresponding to this strategy.

Algorithm GreedyKnapsack (m, n)

```
// P[1 : n] and w[1 : n] contain the profits
and weights respectively of // Objects
ordered so that p[i] / w[i] > p[i + 1] / w[i
+ 1].
// m is the knapsack size and x[1: n] is the
solution vector.
{
        for i := 1 to n do x[i]  := 0.0                  // initialize x
        U := m;
        for i := 1 to n do
        {
                i
                f

                   (w(i) > U) then break;
                x

                [
                i
                ]

                :
                =

                1
                .
                0
                ;

                U

                :
                =

                U

                -

                w
                [
                i
                ]
                ;
```

```
        }
        if (i ≤n) then x[i] := U / w[i];
}
```

Running time:

The objects are to be sorted into non-decreasing order of p_i / w_i ratio. But if we
disregard the time to initially sort the objects, the algorithm requires only O(n) time.

Example:

Consider the following instance of the knapsack problem: n = 3, m = 20, $(p_1, p_2, p_3) = (25, 24, 15)$ and $(w_1, w_2, w_3) = (18, 15, 10)$.

considered the objects in the order of the ratio p_i / w_i .

p_1/w_1	p_2/w_2	p_3/w_3
25/18	24/15	15/10
1.4	1.6	1.5

Sort the objects in order of the non-increasing order of the ratio p_i / x_i. Select the object with the maximum p_i / x_i ratio, so, $x_2 = 1$ and profit earned is 24. Now, only 5 units of space is left, select the object with next largest p_i / x_i ratio, so $x_3 = ½$ and the profit earned is 7.5.

This solution is the optimal solution.

==========

5.1.3. JOB SEQUENCING WITH DEADLINES

- Let us consider, a set of ***n*** given jobs which are associated with deadlines and profit is earned, if a job is completed by its deadline.
- These jobs need to be ordered in such a way that there is maximum profit.
- It may happen that all of the given jobs may not be completed within their deadlines.

- Assume, deadline of i^{th} job J_i is d_i and the profit received from this job is p_i.
- Hence, the optimal solution of this algorithm is a feasible solution with maximum profit.
- Thus, D(i)>0D(i)>0 for 1⩽i⩽n1⩽i⩽n.
- Initially, these jobs are ordered according to profit, i.e. p1⩾p2⩾p3⩾...⩾pnp1⩾p2⩾p3⩾...⩾pn.

Algorithm: Job-Sequencing-With-Deadline (D, J, n, k)

```
D(0) := J(0) := 0
k := 1
J(1) := 1   // means first job is selected
for i = 2 … n do
  r := k
  while D(J(r)) > D(i) and D(J(r)) ≠ r do
    r := r – 1
  if D(J(r)) ≤ D(i) and D(i) > r then
    for l = k … r + 1 by -1 do
      J(l + 1) := J(l)
      J(r + 1) := i
      k := k + 1
```

- Let us consider a set of given jobs as shown in the following table.
- We have to find a sequence of jobs, which will be completed within their deadlines and will give maximum profit.
- Each job is associated with a deadline and profit.

Job	J_1	J_2	J_3	J_4	J_5
Deadline	2	1	3	2	1
Profit	60	100	20	40	20

Solution

- To solve this problem, the given jobs are sorted according to their profit in a descending order.
- Hence, after sorting, the jobs are ordered as shown in the following table.

Job	J_2	J_1	J_4	J_3	J_5
Deadline	1	2	2	3	1

Profit	100	60	40	20	20

From this set of jobs, first we select J_2, as it can be completed within its deadline and contributes maximum profit.

- Next, J_1 is selected as it gives more profit compared to J_4.
- In the next clock, J_4 cannot be selected as its deadline is over, hence J_3 is selected as it executes within its deadline.
- The job J_5 is discarded as it cannot be executed within its deadline.

Thus, the solution is the sequence of jobs (J_2, J_1, J_3), which are being executed within their deadline and gives maximum profit.

Total profit of this sequence is **100 + 60 + 20 = 180**.

==========

5.1.4. OPTIMAL STORAGE ON TAPES

- **Input:** We are given 'n' problem that are to be stored on computer tape of length L and the length of program i is Li
- Such that $1 \leq i \leq n$ and $\Sigma\ 1 \leq k \leq j\ Li \leq 1$
- **Output:** A permutation from all n! For the n programs so that when they are stored on tape in the order the MRT is minimized.
- **Example:**
- Let $n = 3$, $(l1, l2, l3) = (8, 12, 2)$. As $n = 3$, there are $3! = 6$ possible ordering.
- All these orderings and their respective d value are given below:

Ordering	d (i)	Value
1, 2, 3	$8 + (8+12) + (8+12+2)$	50
1, 3, 2	$8 + 8 + 2 + 8 + 2 + 12$	40

-

2, 1, 3	12 + 12 + 8 +12 + 8 + 2	54
2, 3, 1	12 + 12 +2 +12 + 2 + 8	48
3, 1, 2	2 + 2 + 8 + 2 + 8+ 12	34
3, 2, 1	2 + 2 +12 + 2 + 12 + 8	38

The optimal ordering is 3, 1, 2.

- The greedy method is now applied to solve this problem.
- It requires that the programs are stored in non-decreasing order which can be done in O (nlogn) time.

Greedy solution:

i. Make tape empty

ii. Fori = 1 to n do;

iii. Grab the next shortest path

iv. Put it on next tape.

The algorithm takes the best shortest term choice without checking to see whether it is a big long term decision.

Algorithm:

```
Algorithm Optimal_Storage (n, m)
{
K = 0; // Next tape to be stored.
For i = 1 to n do
        {
        Write (i, k); // "Assign program", j, "to tape", k;
        k = (k +1) mod m;
        }
}
```

==========

5.1.5. OPTIMAL MERGE PATTERNS

- **Optimal merge pattern** is a pattern that relates to the merging of two or more sorted files in a single sorted file. This type of merging can be done by the two-way merging method.
- If we have two sorted files containing n and m records respectively then they could be merged together, to obtain one sorted file in time **O (n+m)**.
- There are many ways in which pairwise merge can be done to get a single sorted file. Different pairings require a different amount of computing time.
- The main thing is to pairwise merge the n sorted files so that the number of comparisons will be less.

The formula of external merging cost is:

$$\sum_{i=1}^{n} f(i)d(i)$$

Where, `f (i)` represents the number of records in each file and `d (i)` represents the depth.

Algorithm for optimal merge pattern

```
Algorithm Tree(n)
//list is a global list of n single node
{
       For  i=1 to i= n-1 do
       {
              // get a new tree node
              Pt: new treenode;
              // merge two trees with smallest length
              (Pt = lchild) = least(list);
              (Pt = rchild) = least(list);
              (Pt =weight) = ((Pt = lchild) = weight) = ((Pt =
rchild) = weight);
              Insert (list , Pt);
       }
       // tree left in list
       Return least(list);
}
```

Example:

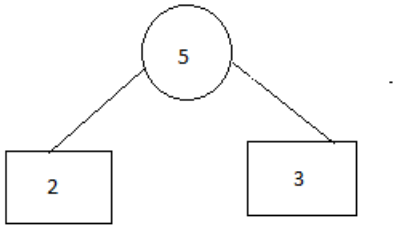
- Given a set of unsorted files: 5, 3, 2, 7, 9, 13
- Now, arrange these elements in ascending order: 2, 3, 5, 7, 9, 13

- After this, pick two smallest numbers and repeat this until we left with only one number.

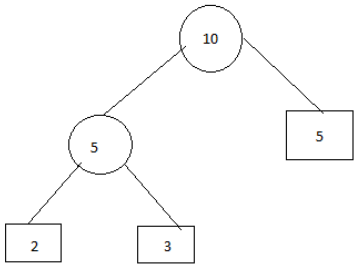
  Now follow following steps:

Step 1: Insert 2, 3

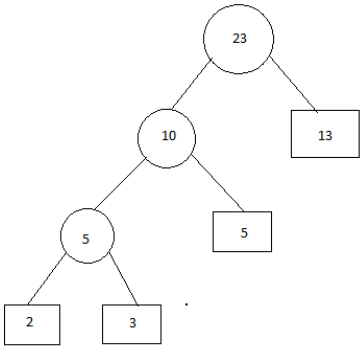
Step 2:



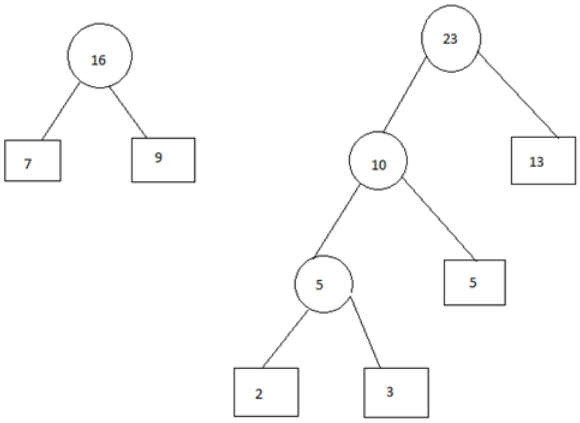
Step 3: Insert 5



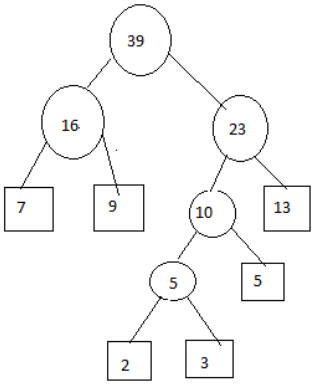
Step 4: Insert 13



Step 5: Insert 7 and 9



Step 6:



So, The merging cost = 5 + 10 + 16 + 23 + 39 = 93

==========

5.2 MINIMUM COST SPANNING TREE

- A **Minimum Spanning Tree (MST)** is a subset of edges of a connected weighted undirected graph that connects all the vertices together with the minimum possible total edge weight.
- To derive an MST, Prim's algorithm or Kruskal's algorithm can be used.
- If there are **n** number of vertices, the spanning tree should have ***n* - 1** number of edges.

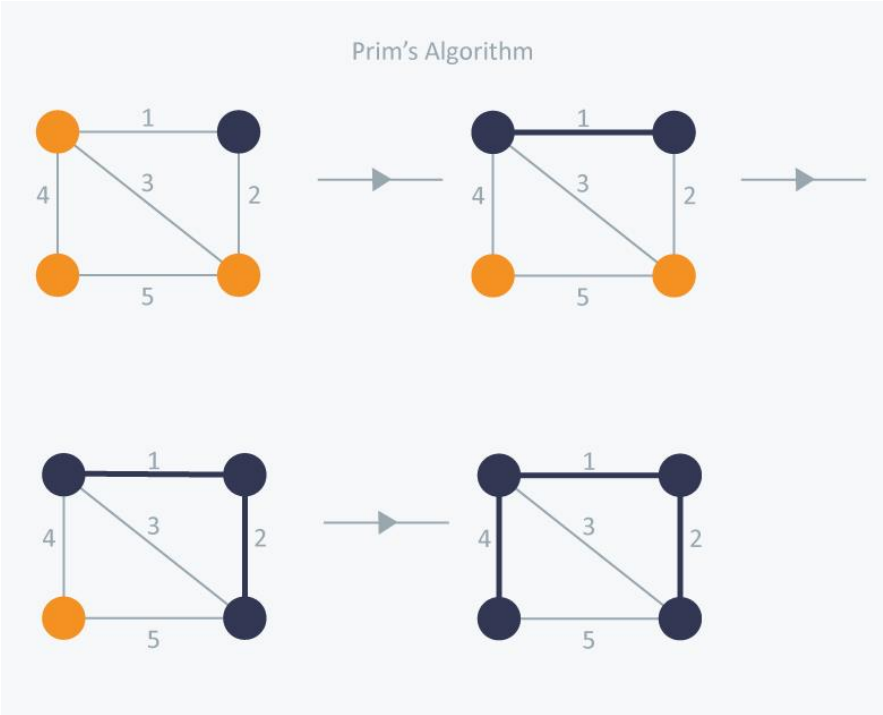
5.2.1. PRIM'S ALGORITHM :

In Prim's Algorithm we grow the spanning tree from a starting position.

Algorithm Steps:

- Maintain two disjoint sets of vertices. One containing vertices that are in the growing spanning tree and other that are not in the growing spanning tree.
- Select the cheapest vertex that is connected to the growing spanning tree and is not in the growing spanning tree and add it into the growing spanning tree. This can be done using Priority Queues. Insert the vertices, that are connected to growing spanning tree, into the Priority Queue.
- 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:



5.2.2. KRUSKAL'S ALGORITHM

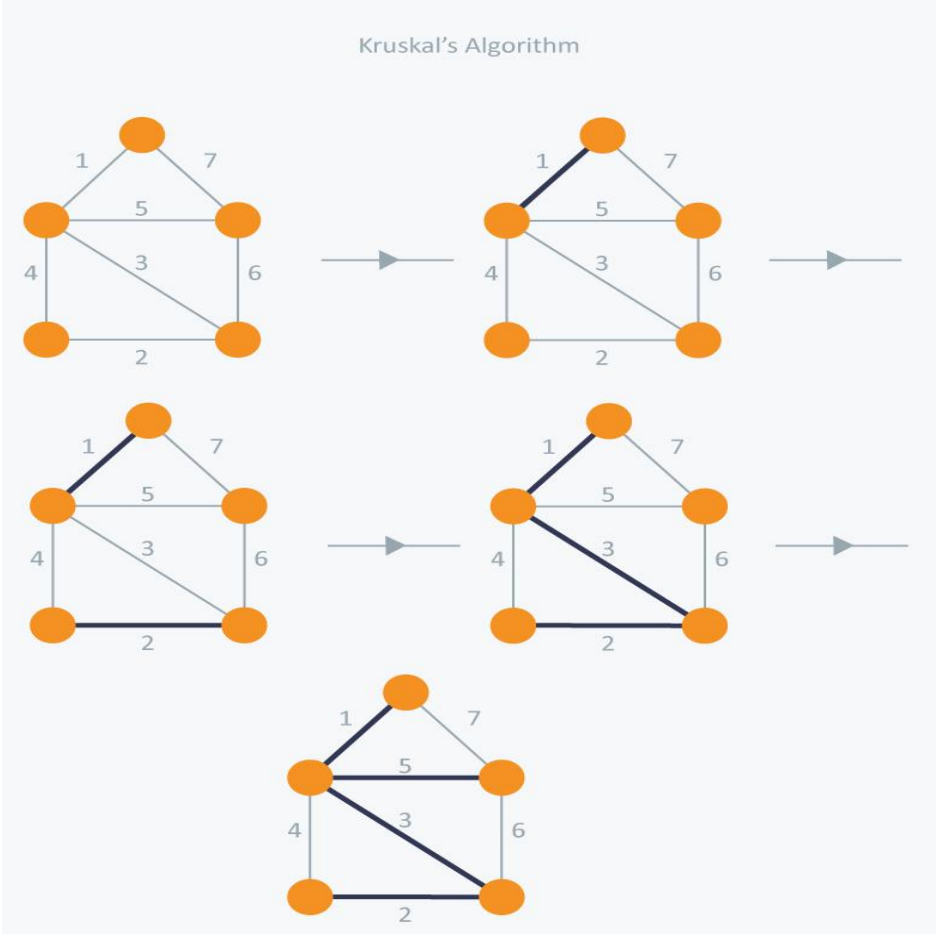
- Kruskal's Algorithm builds the spanning tree by adding edges one by one into a growing spanning tree.

- 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:

- Sort the graph edges with respect to their weights.
- Start adding edges to the MST from the edge with the smallest weight until the edge of the largest weight.
- Only add edges which doesn't form a cycle , edges which connect only disconnected components.

- Consider following example:

- 

==========

5.3. DYNAMIC PROBLEM

5.3.1. INTRODUCTION

- Dynamic Programming solves problems by combining the solutions of subproblems.
- Moreover, Dynamic Programming algorithm solves each sub-problem just once and then saves its answer in a table, thereby avoiding the work of re-computing the answer every time.

The steps in a dynamic programming solution are:

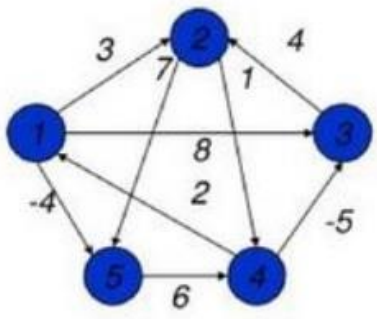
- ➢ Verify that the principle of optimality holds
- ➢ Set up the dynamic-programming recurrence equations
- ➢ Solve the dynamic-programming recurrence equations for the value of the
- ➢ optimal solution.

➢
➢ Perform a trace back step in which the solution itself is constructed.

==========

5.3.2. ALL PAIRS SHORTEST PATH

- The all pair shortest path algorithm is also known as Floyd-Warshall algorithm is used to find all pair shortest path problem from a given weighted graph.
- As a result of this algorithm, it will generate a matrix, which will represent the minimum distance from any node to all other nodes in the graph.



$$\begin{bmatrix} 0 & 1 & -3 & 2 & -4 \\ 3 & 0 & -4 & 1 & -1 \\ 7 & 4 & 0 & 5 & 3 \\ 2 & -1 & -5 & 0 & -2 \\ 8 & 5 & 1 & 6 & 0 \end{bmatrix}$$

- At first the output matrix is same as given cost matrix of the graph. After that the output matrix will be updated with all vertices k as the intermediate vertex.
- The time complexity of this algorithm is O(V3), here V is the number of vertices in the graph.
- Input − The cost matrix of the graph.

```
0 3 6 ∞ ∞ ∞ ∞
3 0 2 1 ∞ ∞ ∞
6 2 0 1 4 2 ∞
∞ 1 1 0 2 ∞ 4
∞ ∞ 4 2 0 2 1
∞ ∞ 2 ∞ 2 0 1
∞ ∞ ∞ 4 1 1 0
```

Output − Matrix of all pair shortest path.

```
0 3 4 5 6 7 7
3 0 2 1 3 4 4
4 2 0 1 3 2 3
5 1 1 0 2 3 3
6 3 3 2 0 2 1
7 4 2 3 2 0 1
7 4 3 3 1 1 0
```

Algorithm
floydWarshal(cost)

Input − The cost matrix of given Graph.

Output − Matrix to for shortest path between any vertex to any vertex.

```
Begin
  for k := 0 to n, do
    for i := 0 to n, do
      for j := 0 to n, do
        if cost[i,k] + cost[k,j] < cost[i,j], then
          cost[i,j] := cost[i,k] + cost[k,j]
        done
      done
    done
    display the current cost matrix
End
```

5.3.3. SINGLE SOURCE SHORTEST PATH

- Dijkstra's algorithm solves the single-source shortest-paths problem on a directed weighted graph *G = (V, E)*, where all the edges are non-negative (i.e., *w(u, v)* ≥ 0 for each edge *(u, v) Є E*).

```
Algorithm: Dijkstra's-Algorithm (G, w, s)
for each vertex v Є G.V
v.d := ∞
   v.∏ := NIL
s.d := 0
S := Φ
Q := G.V
while Q ≠ Φ
   u := Extract-Min (Q)
   S := S U {u}
   for each vertex v Є G.adj[u]
      if v.d>u.d + w(u, v)
v.d := u.d + w(u, v)
         v.∏ := u
```

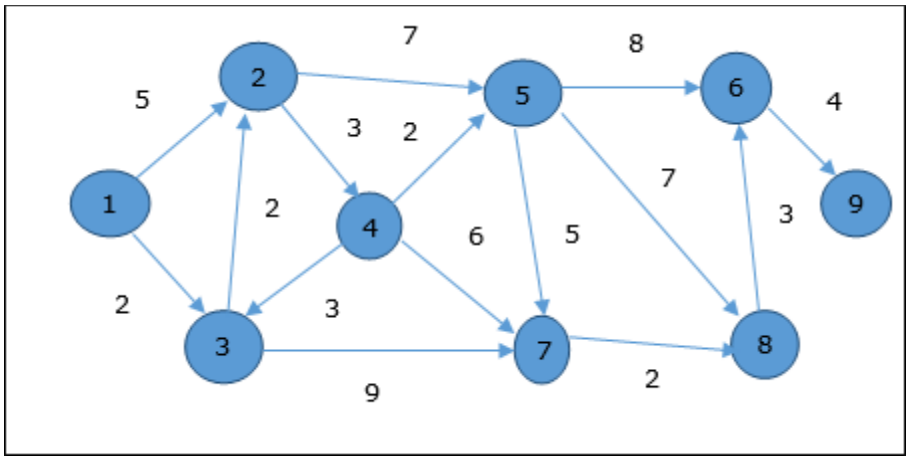
Example

- Let us consider vertex ***1*** and ***9*** as the start and destination vertex respectively. Initially, all the vertices except the start vertex are marked by ∞ and the start vertex is marked by ***0***.

Vertex	Initial	Step1 V_1	Step2 V_3	Step3 V_2	Step4 V_4	Step5 V_5	Step6 V_7	Step7 V_8	Step8 V_6

1	0	0	0	0	0	0	0	0	0
2	∞	5	4	4	4	4	4	4	4
3	∞	2	2	2	2	2	2	2	2
4	∞	∞	∞	7	7	7	7	7	7
5	∞	∞	∞	11	9	9	9	9	9
6	∞	∞	∞	∞	∞	17	17	16	16
7	∞	∞	11	11	11	11	11	11	11
8	∞	∞	∞	∞	∞	16	13	13	13
9	∞	∞	∞	∞	∞	∞	∞	∞	20

- Hence, the minimum distance of vertex ***9*** from vertex ***1*** is ***20***. And the path is
- 1→ 3→ 7→ 8→ 6→ 9
- This path is determined based on predecessor information.



==========

5.3.4. TRAVELLING SALESMAN PROBLEM

- In the traveling salesman Problem, a salesman must visits n cities.
- We can say that salesman wishes to make a tour or Hamiltonian cycle, visiting each city exactly once and finishing at the city he starts from. There is a non-negative cost c (i, j) to travel from the city i to city j.
- The goal is to find a tour of minimum cost. We assume that every two cities are connected. Such problems are called Traveling-salesman problem (TSP).
- We can model the cities as a complete graph of n vertices, where each vertex represents a city.
- Let us consider a graph ***G = (V, E)***, where ***V*** is a set of cities and ***E*** is a set of weighted edges.
- An edge ***e(u, v)*** represents that vertices ***u*** and ***v*** are connected. Distance between vertex ***u*** and ***v*** is ***d(u, v)***, which should be non-negative.
- When $|S| > 1$, we define $C(S, 1) = \propto$ since the path cannot start and end at **1**.
- Now, let express **C(S, j)** in terms of smaller sub-problems. We need to start at ***1*** and end at **j**. We should select the next city in such a way that

$$C(S,j)=\min C(S-\{j\},i)+d(i,j)$$

Where

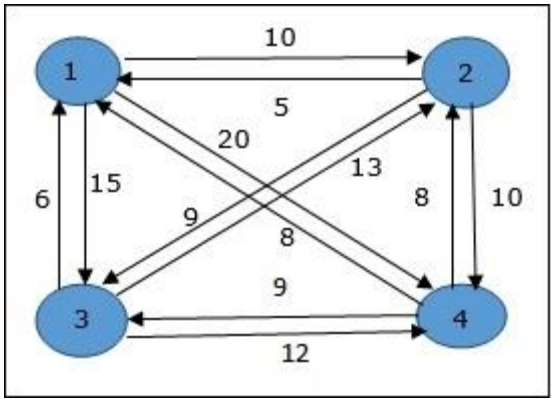
$$i \in S \text{ and } i \neq j \quad c(S,j)=\min C(s-\{j\},i)+d(i,j)$$

$$\text{where} i \in S \text{ and } i \neq j \; C(S,j)=\min C(S-\{j\},i)+d(i,j) \text{where} i \in S \text{ and } i \neq j \; c(S,j)=\min C(s-\{j\},i)+d(i,j) \text{where} i \in S \text{ and } i \neq j$$

Algorithm: Traveling-Salesman-Problem

```
C ({1}, 1) = 0
for s = 2 to n do
  for all subsets S Є {1, 2, 3, … , n} of size s and containing 1
    C (S, 1) = ∞
  for all j Є S and j ≠ 1
    C (S, j) = min {C (S – {j}, i) + d(i, j) for i Є S and i ≠ j}
Return minj C ({1, 2, 3, …, n}, j) + d(j, i)
```

In the following example, we will illustrate the steps to solve the travelling salesman problem.

- 

- From the above graph, the following table is prepared.

	1	2	3	4
1	0	10	15	20
2	5	0	9	10
3	6	13	0	12
4	8	8	9	0

- **S = Φ**
  - Cost(2,Φ,1)=d(2,1)=5Cost(2,Φ,1)=d(2,1)=5Cost(2,Φ,1)=d(2,1)=5Cost(2,Φ,1)=d(2,1)=5
  - Cost(3,Φ,1)=d(3,1)=6Cost(3,Φ,1)=d(3,1)=6Cost(3,Φ,1)=d(3,1)=6Cost(3,Φ,1)=d(3,1)=6
  - Cost(4,Φ,1)=d(4,1)=8Cost(4,Φ,1)=d(4,1)=8Cost(4,Φ,1)=d(4,1)=8Cost(4,Φ,1)=d(4,1)=8
  -

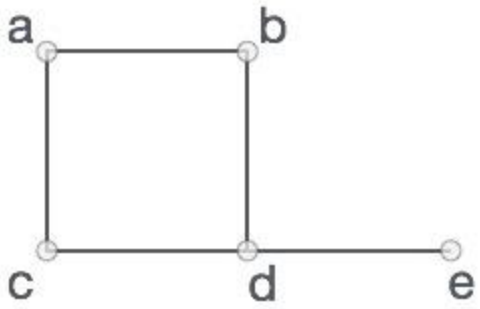
- The minimum cost path is 35.
- Start from cost **{1, {2, 3, 4}, 1}**, we get the minimum value for **d [1, 2]**

==========

5.4. GRAPH

5.4.1. GRAPH AND TERMINOLOGY

- A graph is a pictorial representation of a set of objects where some pairs of objects are connected by links.
- The interconnected objects are represented by points termed as **vertices**, and the links that connect the vertices are called **edges**.
- Formally, a graph is a pair of sets **(V, E)**, where **V** is the set of vertices and **E** is the set of edges, connecting the pairs of vertices. Take a look at the following graph −



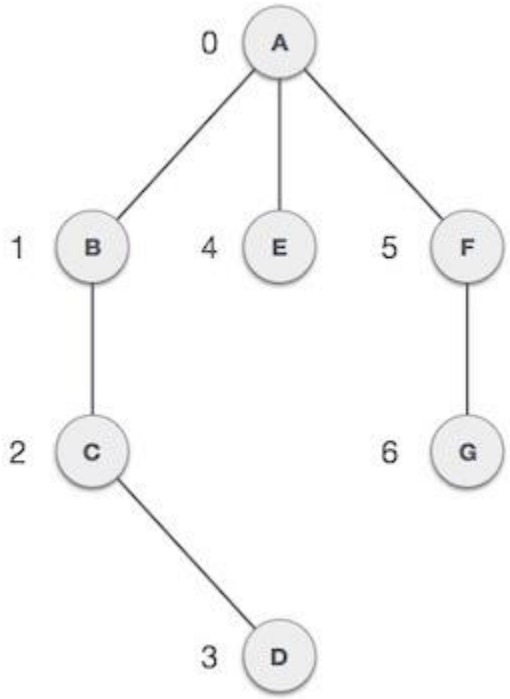
In the above graph,

V = {a, b, c, d, e}

E = {ab, ac, bd, cd, de}

Graph Data Structure

- Mathematical graphs can be represented in data structure.
- We can represent a graph using an array of vertices and a two-dimensional array of edges.
- Before we proceed further, let's familiarize ourselves with some important terms −
- **Vertex** − Each node of the graph is represented as a vertex. In the following example, the labeled circle represents vertices. Thus, A to G are vertices. We can represent them using an array as shown in the following image. Here A can be identified by index 0. B can be identified using index 1 and so on.
- **Edge** − Edge represents a path between two vertices or a line between two vertices. In the following example, the lines from A to B, B to C, and so on represents edges. We can use a two-dimensional array to represent an array as shown in the following image. Here AB can be represented as 1 at row 0, column 1, BC as 1 at row 1, column 2 and so on, keeping other combinations as 0.
- **Adjacency** − Two node or vertices are adjacent if they are connected to each other through an edge. In the following example, B is adjacent to A, C is adjacent to B, and so on.
- **Path** − Path represents a sequence of edges between the two vertices. In the following example, ABCD represents a path from A to D.



Basic Operations

Following are basic primary operations of a Graph −

- **Add Vertex** − Adds a vertex to the graph.
- **Add Edge** − Adds an edge between the two vertices of the graph.
- **Display Vertex** − Displays a vertex of the graph.

Adjacency Matrix

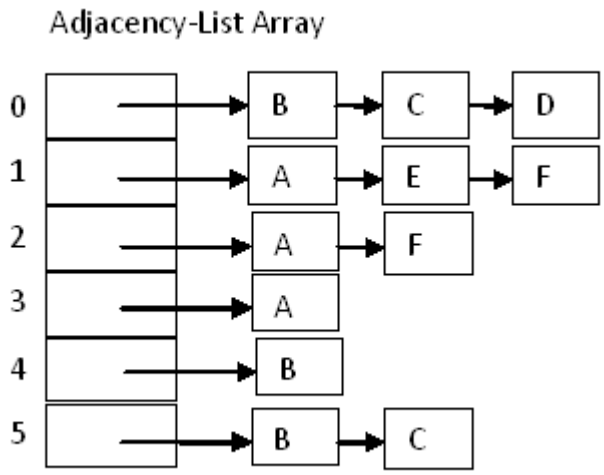
- It is a two dimensional array with Boolean flags. As an example, we can represent the edges for the above graph using the following adjacency matrix

	A	B	C	D	E	F
A	0	1	1	1	0	0
B	1	0	0	0	1	1
C	1	0	0	0	0	1
D	1	0	0	0	0	0
E	0	1	0	0	0	0
F	0	1	1	0	0	0

  - In the given graph, A is connected with B, C and D nodes, so adjacency matrix will have 1s in the ‘A’ row for the ‘B’, ‘C’ and ‘D’ column

Adjacency List

- It is an array of linked list nodes. In other words, it is like a list whose elements are a linked list.
- For the given graph example, the edges will be represented by the below adjacency list:

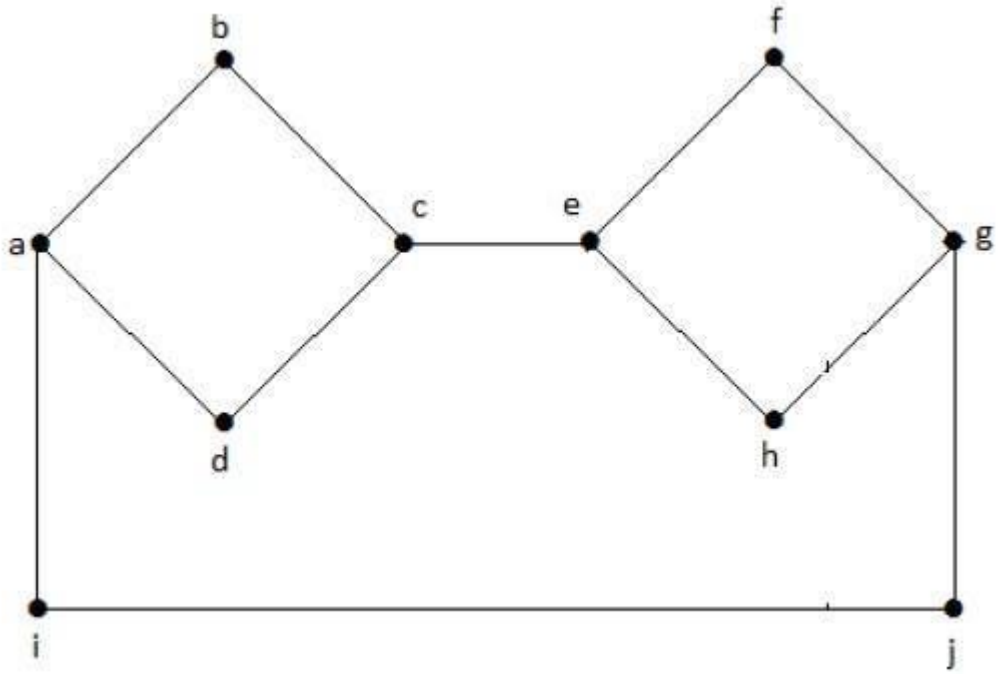


5.4.2. CONNECTED GRAPH

- A graph G is said to be connected **if there exists a path between every pair of vertices**. There should be at least one edge for every vertex in the graph.
- So that we can say that it is connected to some other vertex at the other side of the edge.

Example

In the following graph, each vertex has its own edge connected to other edge. Hence it is a connected graph.

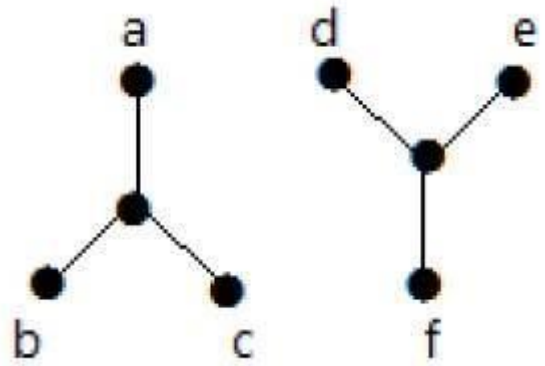


Disconnected Graph

A graph G is disconnected, if it does not contain at least two connected vertices.

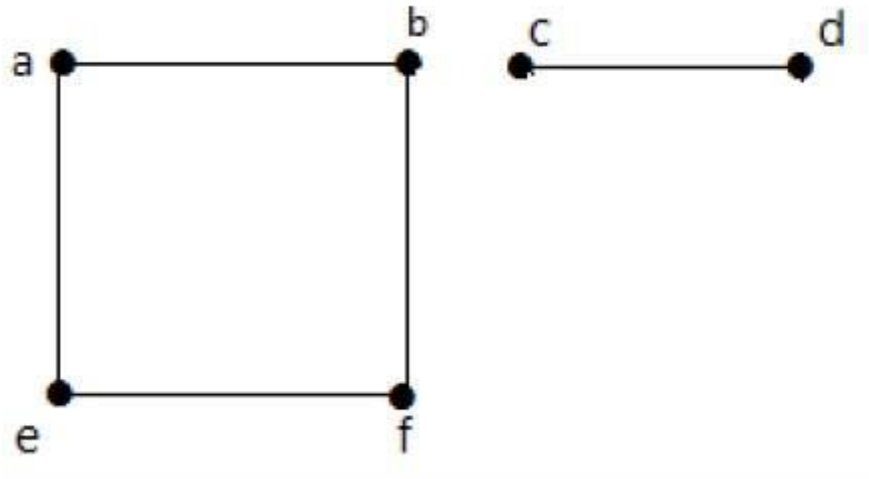
Example 1

The following graph is an example of a Disconnected Graph, where there are two components, one with ‘a’, ‘b’, ‘c’, ‘d’ vertices and another with ‘e’, ’f’, ‘g’, ‘h’ vertices.



The two components are independent and not connected to each other. Hence it is called disconnected graph.

Example 2



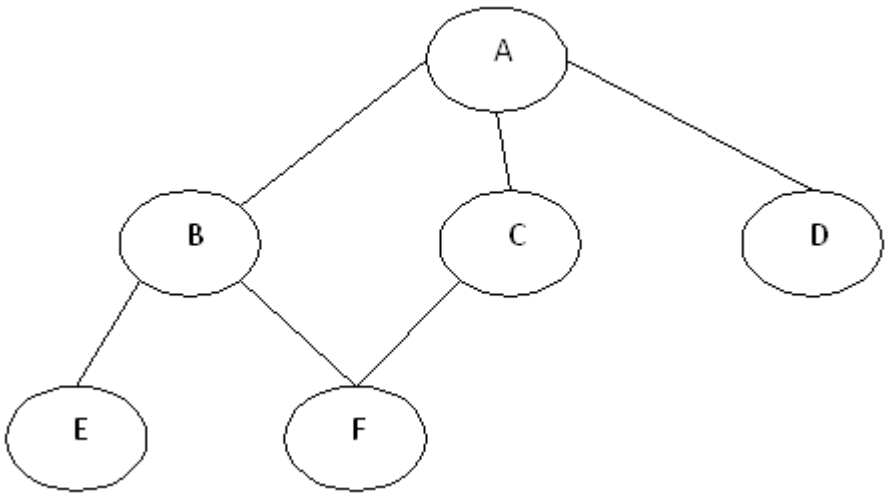
In this example, there are two independent components, a-b-f-e and c-d, which are not connected to each other. Hence this is a disconnected graph.

==========

5.4.3. GRAPH TRAVERSAL TECHNIQUES

They can also be used to find out whether a node is reachable from a given node or not.

Depth First Search (DFS)



The aim of DFS algorithm is to traverse the graph in such a way that it tries to go far from the root node.

- If we do the depth first traversal of the above graph and print the visited node, it will be “A B E F C D”.
- DFS visits the root node and then its children nodes until it reaches the end node, i.e. E and F nodes, then moves up to the parent nodes.

BREADTH FIRST SEARCH

- The aim of BFS algorithm is to traverse the graph as close as possible to the root node. Queue is used in the implementation of the breadth first search.
- If we do the breadth first traversal of the above graph and print the visited node as the output, it will print the following output. “A B C D E F”.