

UNIT- I INTRODUCTION

Syllabus:

Data objects and structures – Performance analysis- Space complexity – Time complexity- Classification of data structures -The linear list data structure -Implementation of List data Structures- Singly linked list– Circular lists- Doubly linked list

Data: - Raw information or unprocessed information is called data.

Data Structure: - Data Structure is a mathematical model to store and organize data so that it can be used efficiently.

Characteristics of a Data Structure:

- ❖ **Correctness** – Data structure implementation should implement its interface correctly.
- ❖ **Time Complexity** – Running time or the execution time of operations of data structure must be as small as possible.
- ❖ **Space Complexity** – Memory usage of a data structure operation should be as little as possible.

Need for Data Structure:

As applications are getting complex and data rich, there are three common problems that applications face now-a-days.

- **Data Search**
- **Processor speed**
- **Multiple requests**

To solve the above-mentioned problems, data structures come to rescue. Data can be organized in a data structure in such a way that all items may not be required to be searched, and the required data can be searched almost instantly.

PERFORMANCE ANALYSIS:

There are *three cases* which are usually used to compare various data structure's execution time in a relative manner.

1. **Worst Case:** - This is the scenario where a particular data structure operation takes maximum time it can take. If an operation's *worst case time is $f(n)$* then this *operation* will not take more than *$f(n)$ time* where $f(n)$ represents function of n .
2. **Average Case:** - This is the scenario depicting the average execution time of an operation of a data structure. If an *operation* takes *$f(n)$ time* in execution, then *m operations* will take *$mf(n)$ time*.
3. **Best Case:** - If an *operation takes $f(n)$ time* in execution, then the *actual operation* may take time as the random number which would be *maximum as $f(n)$* .

Algorithm:

- ❖ Algorithm is a *step-by-step procedure*, which defines a set of instructions to be executed in a certain order to get the desired output.
- ❖ Algorithms are generally created independent of underlying languages.

Algorithm Analysis:

- ❖ Efficiency of an algorithm can be analyzed at two different stages, *before implementation* and *after implementation*.

What Is Asymptotic Analysis Of An Algorithm?

Asymptotic analysis of an algorithm refers to defining the mathematical boundation/framing of its run-time performance. Using asymptotic analysis, we can very well conclude the best case, average case and worst case scenario of an algorithm.

Algorithm Complexity:

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

1. **Time Factor** – Time is measured by counting the number of key operations such as comparisons in the sorting algorithm.

2. **Space Factor** – Space is measured by counting the maximum memory space required by the algorithm.

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

Space Complexity:

- ❖ *Space complexity* of an algorithm represents the *amount of memory space required* by the algorithm in its life cycle.
- ❖ The space required by an algorithm is equal to the sum of the following two statements -
 - a) A *fixed part* that is a space required to store certain data and variables that are independent of the size of the problem.
 - b) A *variable part* is a space required by variables, whose size depends on the size of the problem.
- ❖ *Space complexity* $S(P)$ of any algorithm P is $S(P) = C + SP(I)$, where C is *the fixed part* and $S(I)$ is *the variable part* of the algorithm, which depends on instance characteristic I .
- ❖ **Example:**

Algorithm: SUM(A, B)

Step 1: START

Step 2: $C \leftarrow A + B + 10$

Step 3: Stop

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

Time Complexity:

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

❖ Example:

Addition of two n-bit integers takes **n** steps.

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

ASYMPTOTIC ANALYSIS:

- ❖ Asymptotic analysis of an algorithm refers to defining the mathematical *boundation/framing* of its run-time performance.
- ❖ Using asymptotic analysis, we can very well conclude the *best case, average case, and worst case* scenario of an algorithm.
- ❖ Usually, the time required by an algorithm falls under three types –
 - **Best Case** – Minimum time required for program execution.
 - **Average Case** – Average time required for program execution.
 - **Worst Case** – Maximum time required for program execution.

Asymptotic Notations:

Following are the commonly used asymptotic notations to calculate the running time complexity of an algorithm.

- **O** Notation
- **Ω** Notation
- **θ** Notation

Big Oh Notation (O):

- ❖ The notation **O(n)** is the formal way to express the *upper bound of an algorithm's running time*.
- ❖ It measures the *worst case time complexity* or the *longest amount of time an algorithm* can possibly take to complete.

Omega Notation (Ω):

- ❖ The notation **$\Omega(n)$** is the formal way to express the *lower bound of an algorithm's* running time.

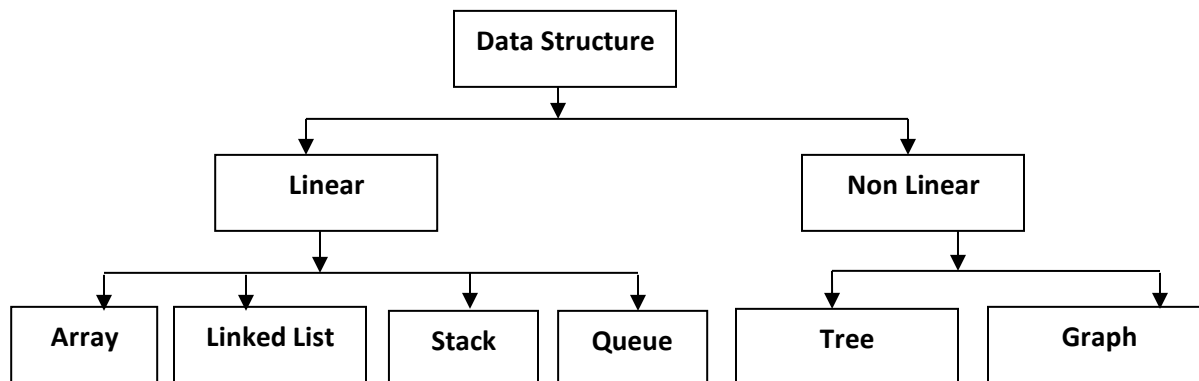
- ❖ It measures the *best case time complexity* or the *best amount of time an algorithm* can possibly take to complete.

Theta Notation (θ):

- ❖ The notation $\theta(n)$ is the formal way to express *both the lower bound and the upper bound* of an algorithm's running time.

What is the difference between time complexity and space complexity?

- Time and Space complexity are *different aspects* of calculating the efficiency of an algorithm.
- *Time complexity* deals with *finding* out how the *computational time* of an algorithm changes with the change in size of the input.
- *Space complexity* deals with *finding* out *how much* (extra) *space* would be required by the algorithm with change in the input size.
- To calculate time complexity of the algorithm the best way is to check if we increase in the size of the input, will the number of comparison (or computational steps) also increase and to calculate space complexity the best bet is to see additional memory requirement of the algorithm also changes with the change in the size of the input.

CLASSIFICATION OF DATA STRUCTURES:**Data Objects:**

Group of one or more piece of data is called data object. Data object represents object having data.

Linear Data Structures:

- ❖ A data structure is called linear if all of its elements are *arranged in the linear order*.
- ❖ In linear data structures, the elements are stored in *non-hierarchical way* where each element has the *successors* and *predecessors* except the first and last element.

Non Linear Data Structures:

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

Data Type:

- ❖ Data type is a way to classify various types of data such as integer, string, etc. which determines the values that can be used with the corresponding type of data, the type of operations that can be performed on the corresponding type of data.
- ❖ There are two data types –
 1. **Built-in Data Type**
 2. **Derived Data Type**

Built-in Data Type:

- ❖ Those data types for which a language has built-in support are known as Built-in Data types.
- ❖ For example, most of the languages provide the following built-in data types.
 1. **Integers**
 2. **Boolean (true, false)**
 3. **Floating (Decimal numbers)**
 4. **Character and Strings**

Derived Data Type:

- ❖ Those data types which are implementation independent as they can be implemented in one or the other way are known as derived data types.
- ❖ These data types are normally built by the combination of primary or built-in data types and associated operations on them. For example –
 1. **List**
 2. **Array**
 3. **Stack**
 4. **Queue**

THE LINEAR LIST DATA STRUCTURE:

LIST:

A list is a data structure. It is an ordered set of elements.

In general list is formatted as follows -

A ₁	A ₂	A ₃	A _n
----------------	----------------	----------------	-------	----------------

Where,

A₁ is the *first element*.

A_n is the *last element*.

n is the *size of the List*.

We can implement the list data structure by following three ways –

1. Array implementation.
2. Linked List Implementation.
3. Cursor Implementation.

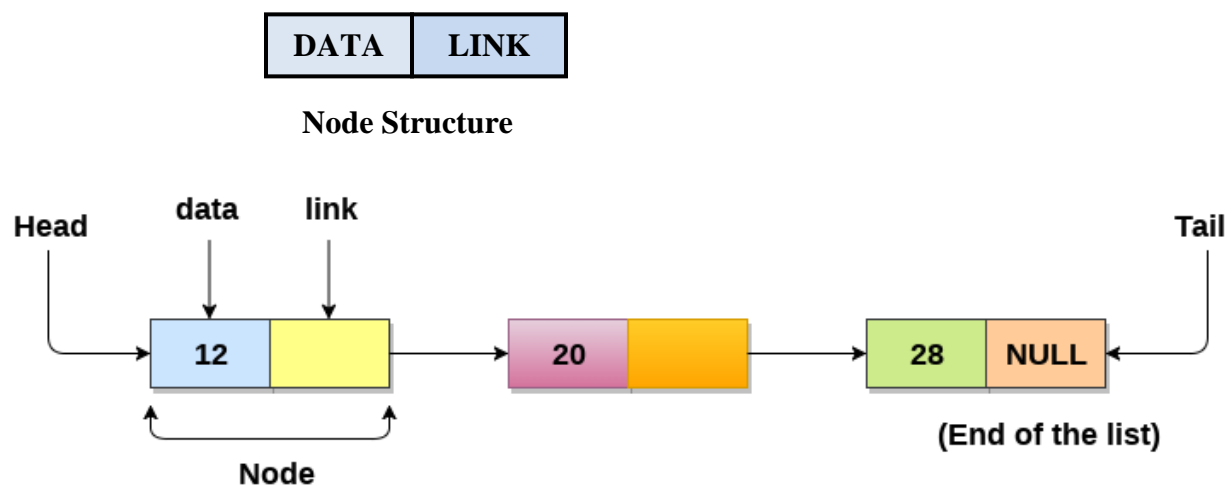
ADT:

Define ADT.

- ❖ ADT stands for Abstract Data Type.
- ❖ It mentions what operations are to be performed but not how these operations will be implemented.
- ❖ It does not specify how data will be organized in memory and what algorithms will be used for implementing the operations.
- ❖ It is called “abstract” because it gives an implementation-independent view.

LINKED LIST IMPLEMENTATION OF LIST ADT:

- ❖ 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**.

**Uses of Linked List**

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

Types of Linked List:

1. Singly Linked List.
2. Doubly Linked List.
3. Circular Linked List.

SINGLY LINKED LIST:

- ❖ Singly linked list can be defined as the collection of ordered set of elements. A node in the singly linked list consists of two parts: **data part** and **link part**.

- ❖ Data part of the node stores *actual information* that is to be represented by the node while the link part of the node stores the *address of its immediate successor*.
- ❖ Singly linked list can be traversed *only in one direction*. We *cannot traverse* the list in the *reverse direction*.

Singly Linked List ADT:

There are various operations which can be performed on singly linked list. A list of all such operations is given below.

Node Creation:

```
struct node
{
    int data;
    struct node *next;
}head=null;
```

INSERTION:

The insertion into a singly linked list can be performed at different positions. Based on the position of the new node being inserted, the insertion is categorized into the following categories.

1. Insert at Beginning.
2. Insert at Last.
3. Insert at particular Position.

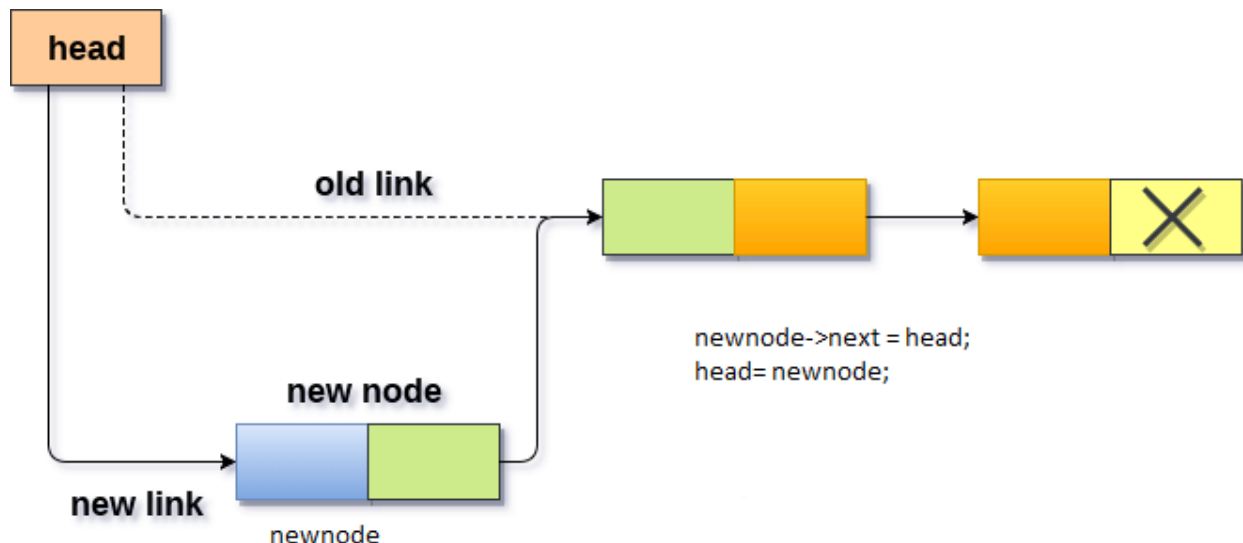
DELETION AND TRAVERSING:

The Deletion of a node from a singly linked list can be performed at different positions. Based on the position of the node being deleted, the operation is categorized into the following categories.

1. Delete a First Node.
2. Delete a Last Node.
3. Delete at particular Node.
4. Search an element.
5. Print the Linked List.

INSERTION ADT:**Insert at Beginning:**

- ❖ Inserting a new element into a singly linked list at beginning is quite simple. We just need to make a few adjustments in the node links.
- ❖ There are the following steps which need to be followed in order to insert a new node in the list at beginning.
 1. First create a new node named as “newnode” and allocate memory space.
 2. Then check whether the linked list is empty or not.
 3. **If the list is empty** then we can directly insert our new node with head node.
 - i. Assign data to data part of newnode.
 - ii. Assign null value at next part of newnode.
 - iii. Assign address of new node to head.
 4. **If the list is not empty** then do the following –
 - i. Assign data to data part of newnode.
 - ii. Assign value of head (First Node) to next part of newnode.
 - iii. Assign address of newnode to head.

**//Insert @ First**

```
void insert_first(int x)
```

```
{
```

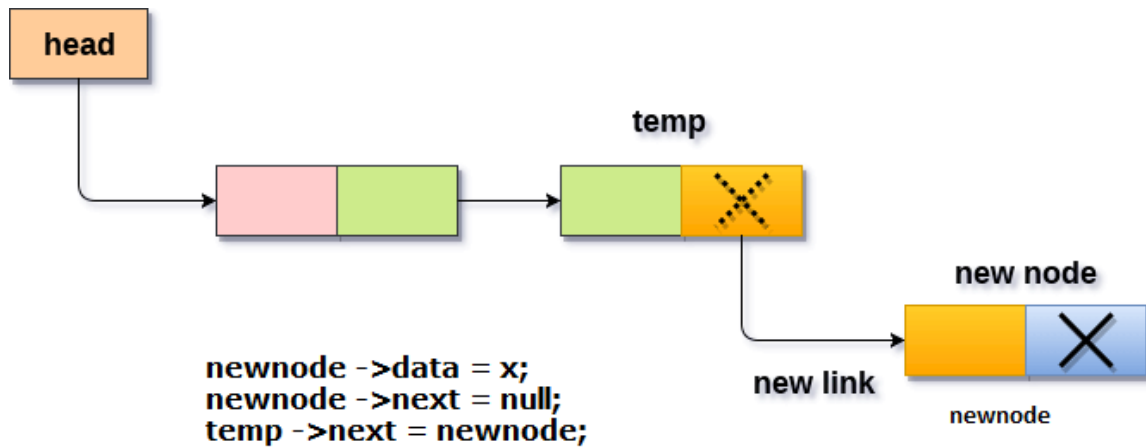
```
    struct node *newnode = (struct node *) malloc(sizeof(struct node *));
```

```
if(head == NULL)           //List is Empty
{
    printf("\n List is empty\n");
    newnode->data=x;
    newnode->next= null;
    head = newnode;
    printf("\nNode inserted @ First\n");
}
else                         //List is Not Empty
{
    newnode->data = x;
    newnode->next = head;
    head = newnode;
    printf("\n Node inserted @ First\n");
}
}
```

Insert at Last:

- ❖ In order to insert a node at the last, there are two following scenarios which need to be mentioned.
 1. The node is being added to an empty list.
 2. The node is being added to the end of the linked list.
- ❖ There are the following steps which need to be followed in order to insert a new node in the list at last.
 1. First **create a new node** named as “**newnode**” and **allocate memory** space.
 2. Second **check** whether **the linked list is empty or not**.
 3. **If the list is empty** then we can directly insert our new node with head node as a last node as well as first node.
 - i. Assign data to data part of newnode.
 - ii. Assign null value at next part of newnode.
 - iii. Assign address of new node to head.

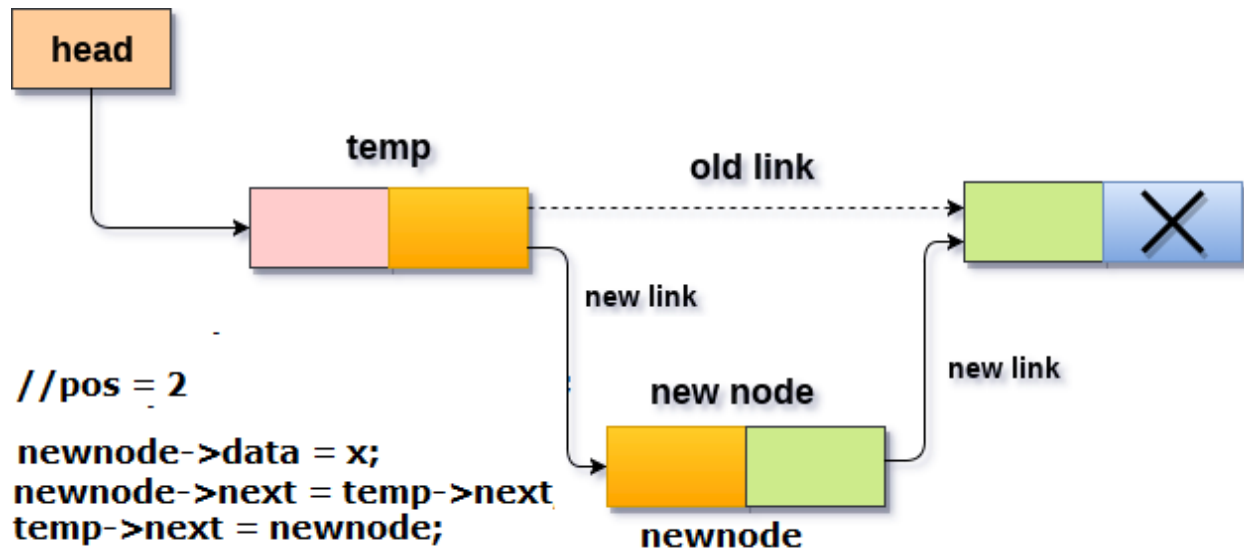
4. **If the list is not empty** then do the following –
- Create a temporary node named as “temp”.
 - Assign value of head to temp. (Now temp pointed @ First node)
`temp = head;`
 - Then, traverse through the entire linked list using the statements:
`while(temp->next!=null)`
`temp = temp->next;`
 - Now temp pointed at last node.
 - Assign data to data part of newnode.
 - Assign null value to next part of new node.
 - Assign address of newnode to next part of temp node.



Inserting node at the last into a non-empty list

Insert at particular Position:

- ❖ It involves insertion a new element at the specified position of the linked list.
- ❖ We need to skip the desired number of nodes in order to reach the previous position of the desired position.
- ❖ Then we can insert the new element at desired position.



//Insert @ Specified Position

```
void insert_position(int x, int pos)
{
    struct node *newnode = (struct node *) malloc(sizeof(struct node *));
    struct node *temp=head;
    if(pos==1)    //Insert an element as a first element
    {
        newnode->data=x;
        newnode->next= head;
        head = newnode;
        printf("\nNode inserted @ First\n");
        //return;
    }
    else
    {
        if(head==NULL)
        {
            printf("Invalid Position Entered \n");
            return;
        }
    }
}
```

```
    }
    else
    {
        for(int i=0; i<pos-2; i++)
            temp = temp->next;    //temp now pointed @ prev node of desired
node.

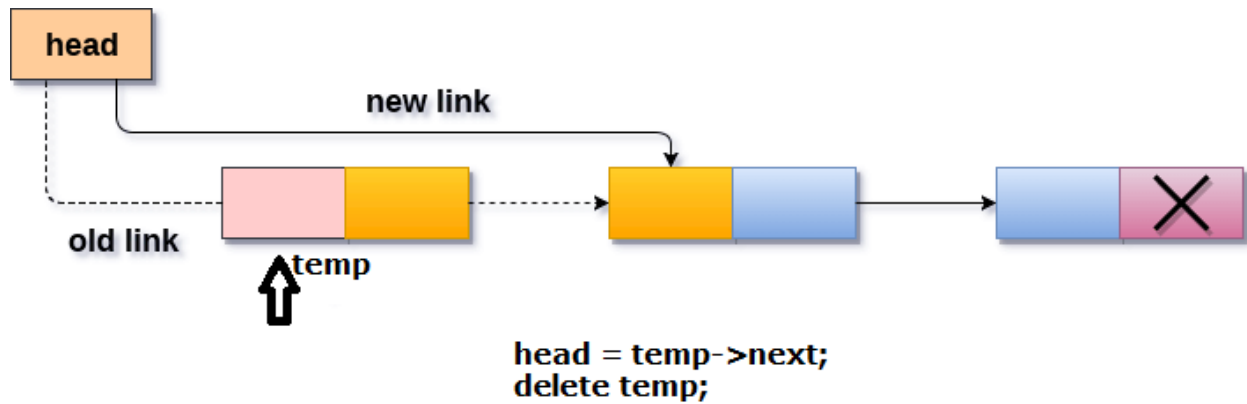
        if(temp->next==NULL)        //Means End of the LIST
            void insertlast(int x);
        else
        {
            newnode->data = x;
            newnode->next = temp->next;
            temp->next = newnode;
        }
        printf("The Element\t %d \t inserted successfully at the
position:%d\n",x,pos);
    }    //End of Else
}    //End of Else
}    //End of Insert_position
```

DELETION:

- ❖ The Deletion of a node from a singly linked list can be performed at different positions.
- ❖ Based on the position of the node is being deleted.

Delete a First Node:

- ❖ Deleting a node from the beginning of the list is the simplest operation of all. It just needs a few adjustments in the node pointers.
- ❖ Since the first node of the list is to be deleted, therefore, we just need to make the head, point to the next of the head.
- ❖ This will be done by using the following statements.



Deleting a node from the beginning

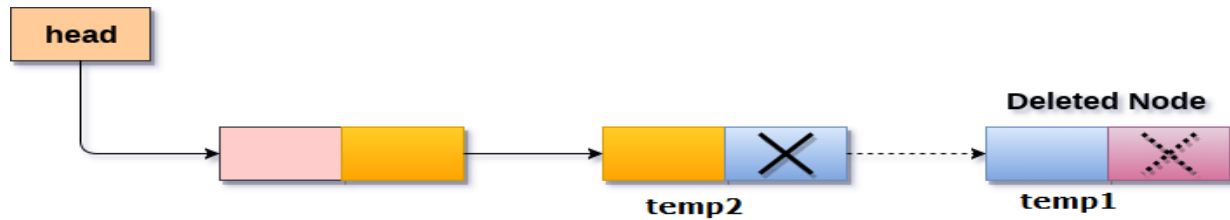
//Delete @ First

```
void delete_first()
{
    if(head == NULL)
    {
        printf(" LIST IS EMPTY..");
        return;
    }
    else
    {
        Struct node *temp;
        temp = head;
        head = temp->next;
        delete temp;
        return;
    }
    printf("\n The First element is deleted from the linked list \n");
}
```

Delete a Last Node:

There are *two scenarios* in which, a node is deleted from the end of the linked list.

1. There is only one node in the list and that needs to be deleted.
2. There are more than one node in the list and the last node of the list will be deleted.



```
while(temp1->next != null)
{
    temp2 = temp1;
    temp1 = temp1 -> next;
}

temp2 -> next = null;
delete temp1;
```

Deleting a node from the last**//Delete Last**

```
void delete_last()
{
    struct node *temp1,*temp2;
    temp1=head;
    if(head==NULL)
    {
        printf("LIST IS EMPTY..\n");
        return;
    }
    else if(temp1->next==NULL)
    {
        head=temp1->next;
        delete temp1;
        return;
    }
    else
```



```
{
    while(temp1->next!=NULL)
    {
        temp2=temp1;
        temp1=temp1->next;
    }
    temp2->next=NULL;
    delete temp1;      //Last node is deleted
    return;
    printf("\n THE LIST AFTER DELETION \n");
}
//End of Delete Last
```

Delete At Specified Position Node:

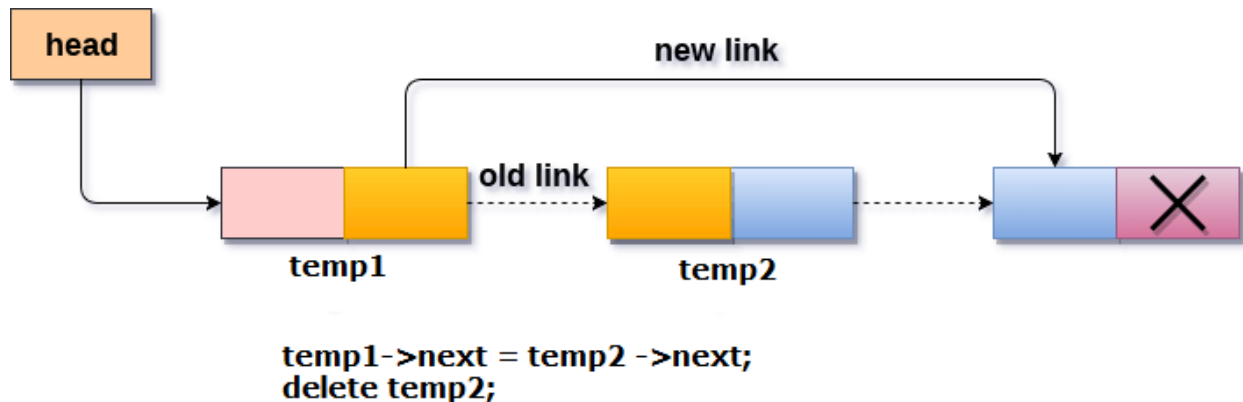
- ❖ In order to delete the node, which is present before the specified node, we need to skip the desired number of nodes to reach the previous node before which the node will be deleted.
- ❖ We need to keep **track of the two nodes**. The one which is to be deleted the other one if the node which is present before that node.

//Delete @ Specified Node

```
void delete_position(int pos)
```

```
{
    struct node *temp1, *temp2;
    temp1 = head;
    if(head == null)
    {
        printf("\n List is Empty....Deletion is not possible....\n");
        return;
    }
    else
    {
        if(position==1)
```

```
{
    head = temp1->next;
    delete temp1;
    return;
}
else
{
    for(int i=0; i<pos-2; i++)
    {
        temp1 = temp1->next;
    }
    temp2 = temp1->link; //temp2 is desired positioned node to be deleted
    temp1->link = temp2->link;
    delete temp2;        //The desired position element is deleted
    return;
} //End of Else
}
```



Deletion a node from specified position

SEARCH AN ELEMENT:

- ❖ Searching is performed in order to find the location of a particular element in the list.

- ❖ Searching any element in the list needs traversing through the list and makes the comparison of every element of the list with the specified element.
- ❖ If the element is matched with any of the list element then the location of the element is returned from the function.

//Search Element from the List

```
void search(int val)
```

```
{
    struct node *temp=head;
    int position=1, flag;
    if(head==NULL)
    {
        printf("\n....List is Empty....Search is not Possible....\n");
        return;
    }
    else
    {
        while(temp!=NULL)          //Don't use temp->next != null
        {
            if(temp->data==val)
            {
                printf("Element \t %d \t present in the position: \t %d", val, position);
                flag = 0;
                return; // Exit from while loop
            }
            else
            {
                flag=1;
            }
            temp = temp->next;
            position++;
        }
    }
}
```

```
    }
    if(flag==1)
        printf("The Element:\t %d \tNot found\n", val);
    }
    return;
} //End of Search Element
```

DISPLAY THE ELEMENTS IN THE LINKED LIST:

1. First check whether the list is empty or not.
2. If empty, then display “List is Empty”.
3. If not empty –
 1. Create a temporary node named as “temp” and assign value of head to temp.
 2. Use the following code to traverse the linked list upto end of the linked list.
 3. While traversing, print the elements in the linked list.

//Display List

```
void display_list()
{
    node *temp;
    if(head==NULL)
    {
        printf("THE LIST IS EMPTY..\n");
        return; //Exit from display_list()
    }
    else
    {
        temp=head;
        printf("The items present in the list are:\n");
        printf("Head");
        while(temp!=NULL)
        {
            printf("->%d", temp->data); //print the elements
            temp = temp->next;
        }
    }
}
```

```
    }  
} //End of Display List
```

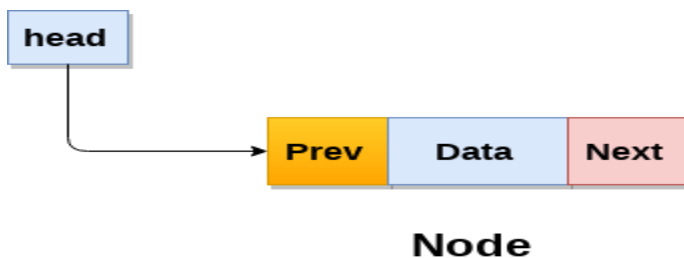
Comparison between Array and Linked List

Basis Comparison	Array	Linked List
Size	Specified during declaration.	No need to specify; grow and shrink during execution.
Storage Allocation	Element location is allocated during compile time.	Element position is assigned during run time.
Order of the elements.	Stored consecutively	Stored randomly

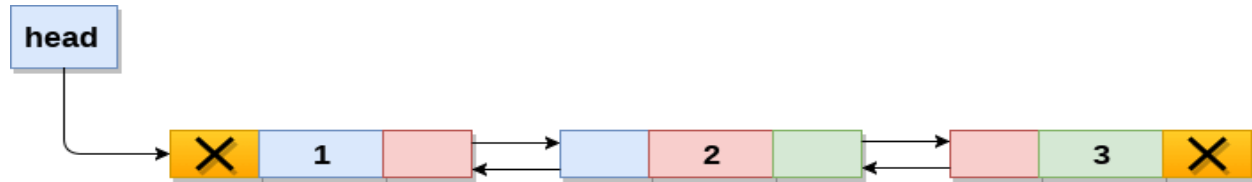
Basis Comparison	Array	Linked List
Accessing the element	Direct or randomly accessed, i.e., Specify the array index or subscript.	Sequentially accessed, i.e., Traverse starting from the first node in the list by the pointer.
Insertion and deletion of element	Slow relatively as shifting is required.	Easier, fast and efficient.
Searching	Binary search and linear search	linear search
Memory required	less	More

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.
- ❖ Therefore, in a doubly linked list, a node consists of *three parts: node data, pointer to the next node* in sequence (next pointer), and *pointer to the previous node* (previous pointer).
- ❖ A sample node structure is shown in the figure.

**Example:**

A doubly linked list containing three nodes having numbers from 1 to 3 in their data part, is shown in the following image.



Doubly Linked List

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

```
struct node
{
    struct node *prev;
    int data;
    struct node *next;
} *head=null;
```

❖ The **prev part of the first node** and the **next part of the last node** will always contain **null** indicating end in each direction.

DOUBLY LINKED LIST ADT:

Node Creation:

```
struct node
{
    struct node *prev;
    int data;
    struct node *next;
} *head=null;
```

1. INSERTION ADT

- Insert at First
- Insert at Last
- Insert at particular position

2. DELETION ADT

- a. Delete at First
- b. Delete at last
- c. Delete at particular position

3. SEARCH ADT

- a. Search an Element

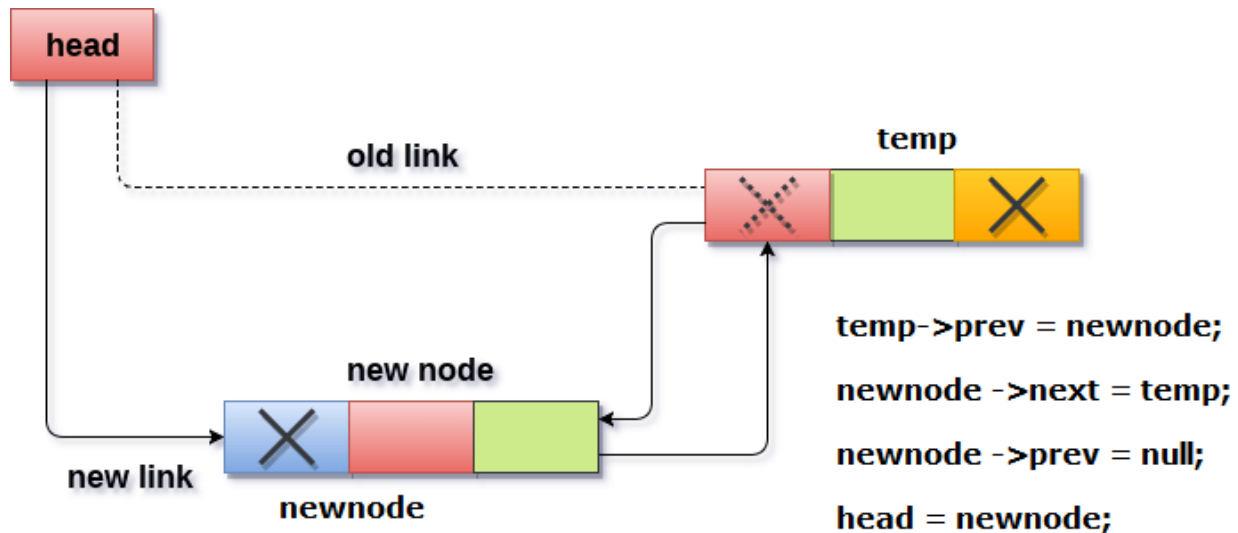
4. PRINT ADT

- a. Print the Elements in the Linked List

INSERTION ADT:

Insert at First:

- ❖ As in doubly linked list, each node of the list contains **double pointers**.
- ❖ There are two scenarios of inserting any element into doubly linked list.
 - ✚ Either the **list is empty** or it contains **at least one element**.



Insertion into doubly linked list at beginning

#Insert at First

```
void insert_first(int x)
```

```
{
```

```
    struct node *newnode, *temp;
```

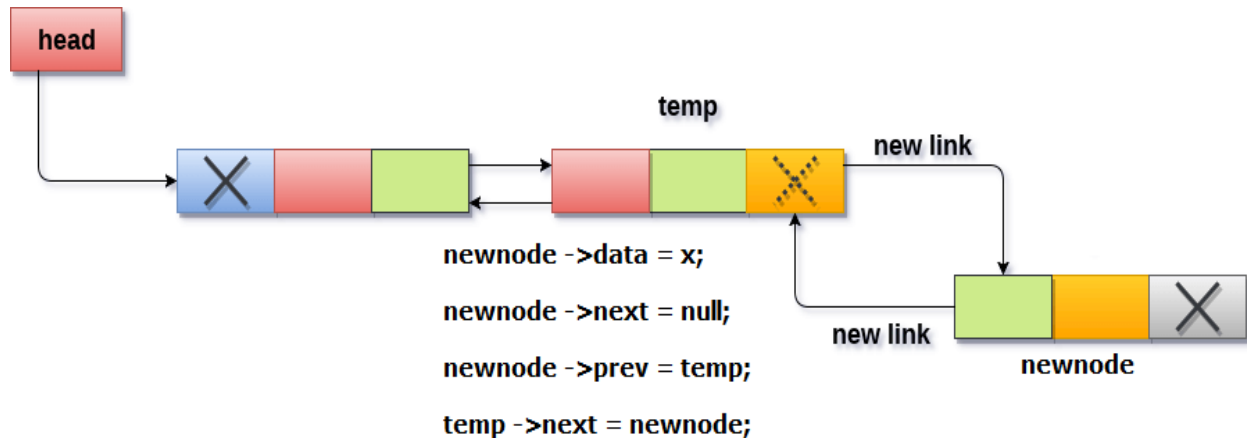
```
    struct node *newnode = (struct node *) malloc(sizeof(struct node *));
```



```
if(head == null)           //List is Empty
{
    newnode ->data = x;
    newnode ->next = null;
    newnode ->prev = null;
    head = newnode;
}
else
{
    temp = head;
    newnode ->data= x;
    temp ->prev = newnode;
    newnode ->next = temp;
    newnode ->prev = null;
    head = newnode;
}
```

Insert at Last:

- ❖ An insert a node in doubly linked list at the end, we must make sure whether the **list is empty** or it **contains any element**.
- ❖ Use the following steps in order to insert the node in doubly linked list at the end.

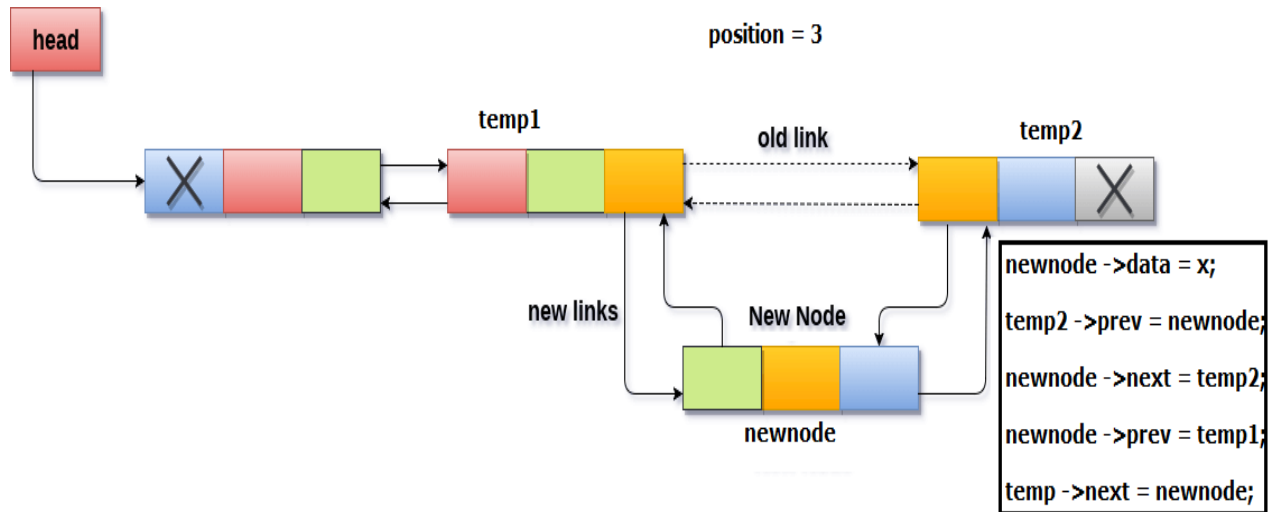
**Insertion into doubly linked list at the end****#Insert at Last:**

```
void insert_last(int x)
```

```
{
    struct *newnode, *temp;
    newnode = (struct node*) malloc(sizeof(struct node*));
    if(head == null)
    {
        void insert_first(int x);
    }
    else
    {
        temp = head;
        while(temp -> next != null)
        {
            temp =temp -> next;
        }
        newnode -> data = x;
        newnode -> next = null;
        newnode -> prev = temp;
        temp ->next = newnode;
    }
} //End of Insert_Last()
```

Insert at particular position:

- ❖ An insert a node in the specified position in the list, we need to skip the required number of nodes in order to reach the previous node of desired positioned node and then make the pointer adjustments as required.



Insertion into doubly linked list after specified node

#Insert at Position:

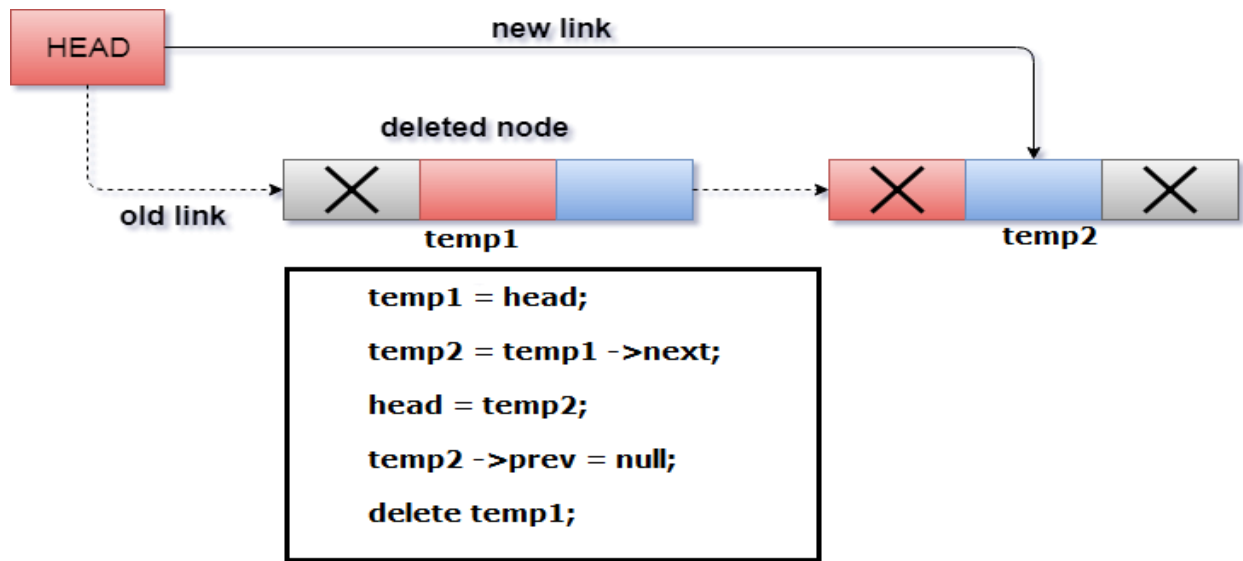
```

void insert_position(int x, int pos)
{
    struct node *newnode, *temp1, *temp2;
    newnode = (struct node*) malloc(sizeof(struct node*));
    if(pos == 1)
    {
        void insert_first(x);
    }
    else
    {
        if(head==NULL)
        {
            printf("Invalid Position Entered \n");
            return;
        }
        for(int i=0; i<pos-2; i++)
        {
            temp1 = temp->next; //temp1 is pointed @ prev node of desired position.
        }
        if(temp1->next==NULL) //Means End of the LIST
    }
}
    
```

```
{
    void insert_last(x);
}
else
{
    temp2 = temp->next;          //temp2 is pointed at desired position
    newnode ->data = x;
    temp2 ->prev = newnode;
    newnode ->next = temp2;
    newnode->prev = temp1;
    temp1->next = newnode;
}
printf("The Element\t %d \t inserted successfully at the position:%d\n", x, pos);
}
//End of Insert Last
```

DELETION ADT:

Delete at First:



Deletion in doubly linked list from beginning

#Delete at First:

```
void delete_first()
```

```
{
```

```
    struct node *temp1, *temp2;
```

```
    if(head == null)
```

```
    {
```

```
        printf("\n List is Empty....Deletion not possible \n");
```

```
        return;
```

```
    }
```

```
    else
```

```
    {    temp1 = head;
```

```
        temp2 = temp1 -> next    //temp2 is pointed at 2nd node
```

```
        temp2 -> prev = null;
```

```
        head = temp2;
```

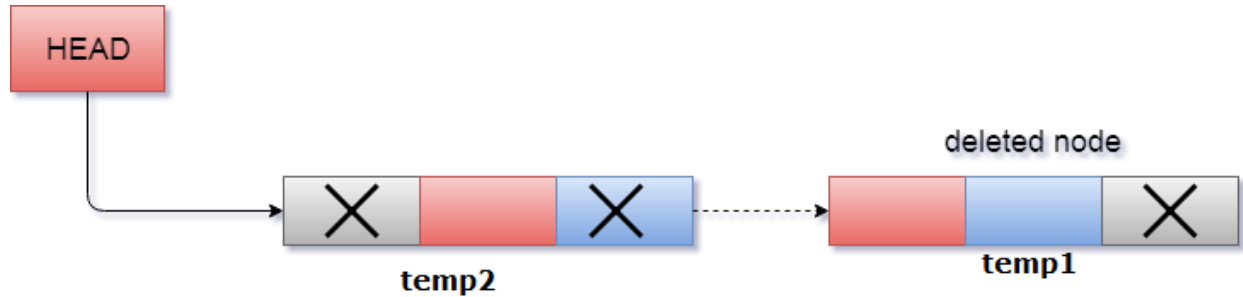
```
        delete temp1;    //1st is Deleted
```

```
    }
```

```
}
```

Delete at Last Node:

- ❖ Deletion of the last node in a doubly linked list needs traversing the list in order to reach the last node of the list and then make pointer adjustments at that position.



```
temp2 = temp1 ->prev;           // temp1 is the last node
temp2 -> next = null;
delete temp1;
```

Deletion in doubly linked list at the end

#Delete at Last Node:

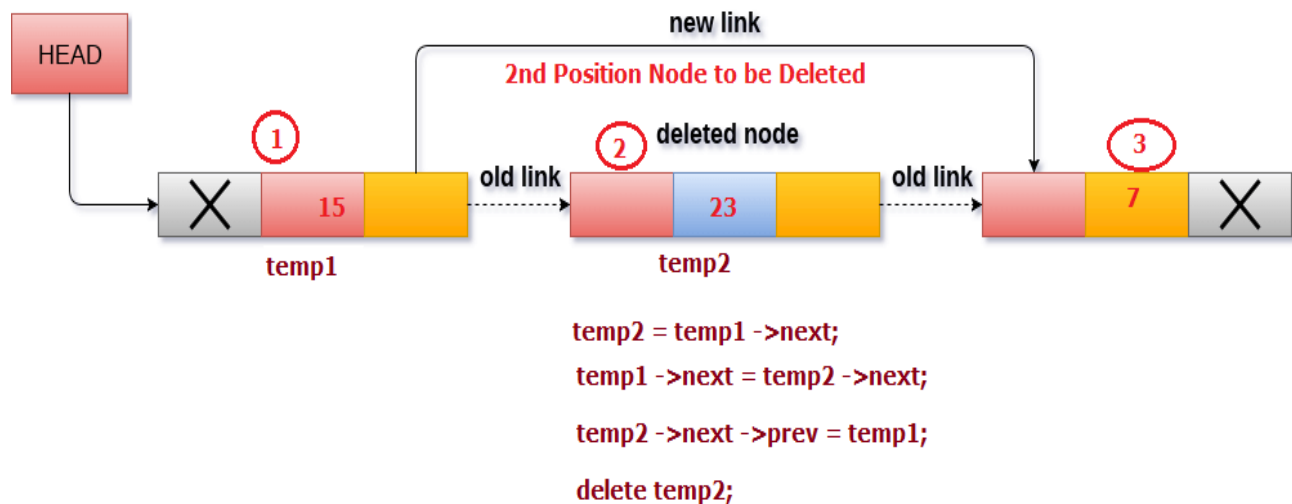
```
void delete_last()
{
    struct node *temp1, *temp2;
    temp1 = head;
    if(head == null)
    {
        printf("\n List is Empty.....Deletion is not possible \n");
        return;
    }
    else if(temp1->next==NULL)
    {
        head=temp1->next;
        delete temp1;
        return;
    }
    else
    {
        while(temp1->next!=NULL)
```

```
{    temp1=temp1->next;
}

temp2 = temp1 ->prev;    //temp2 is pointed at previous node of last node
temp2->next=NULL;
delete temp1;    //Last node is deleted
return;
}

}    //End of Delete Last
```

Delete at specified position Node:



Deletion of a specified node in doubly linked list

#Delete at specified Positioned Node:

```
void delete_position(int pos)
```

```
{
    struct node *temp1, *temp2;
    if(head == null)
    {
        printf("\n....List is Empty.....Deletion is not possible....\n");
        return;
    }
}
```

```
else           //List is not Empty
{
    if(pos == 1)
    {
        head = temp1->next;
        delete temp1;
        return;
    }
    else
    {
        for(int i=1; i<pos-2; i++)
        {
            temp1 = temp1 ->next;
        }
        temp2 = temp1 ->next;
        temp1 ->next = temp2 ->next;
        temp2 ->next ->prev = temp1;
        delete temp2;
    }
}
}
```

SEARCH AN ELEMENT:

- ❖ Searching is performed in order to find the location of a particular element in the list.
- ❖ Searching any element in the list needs traversing through the list and makes the comparison of every element of the list with the specified element.
- ❖ If the element is matched with any of the list element then the location of the element is returned from the function.

//Search Element from the List


```
void search(int val)
{
    node *temp;
    temp=head;
    int position=1;
    if(head == null)
    {
        printf("\n....List is Empty....Search is not Possible....\n");
        return;
    }
    else
    {
        while(temp!=NULL)          //Don't use temp->next != null
        {
            if(temp->data==val)
            {
                printf("Element \t %d \t present in the position: \t %d", val, position);
                return; // Exit from while loop
            }
            else
            {
                temp = temp->next;
                position++;
            }
        }
        printf("The Element:\t %d \tNot found\n", val);
        return;
    } //End of Search Element
}
```

Display the Elements in The Linked List:

1. First check whether the list is empty or not.
2. If empty, then display “List is Empty”.
3. If not empty –
 1. Create a temporary node named as “temp” and assign value of head to temp.
 2. Use the following code to traverse the linked list upto end of the linked list.
 3. While traversing, print the elements in the linked list.

//Display List

```
void display_list()
{
    node *temp;
    if(head==NULL)
    {
        printf("THE LIST IS EMPTY..\n");
        return; //Exit from display_list()
    }
    else
    {
        temp=head;
        printf("The items present in the list are:\n");
        printf("Head");
        while(temp!=NULL)
        {
            printf("->%d", temp->data); //print the elements
            temp = temp->next;
        }
    }
} //End of Display List
```

DIFFERENCE BETWEEN SINGLY AND DOUBLY LINKED LISTS:

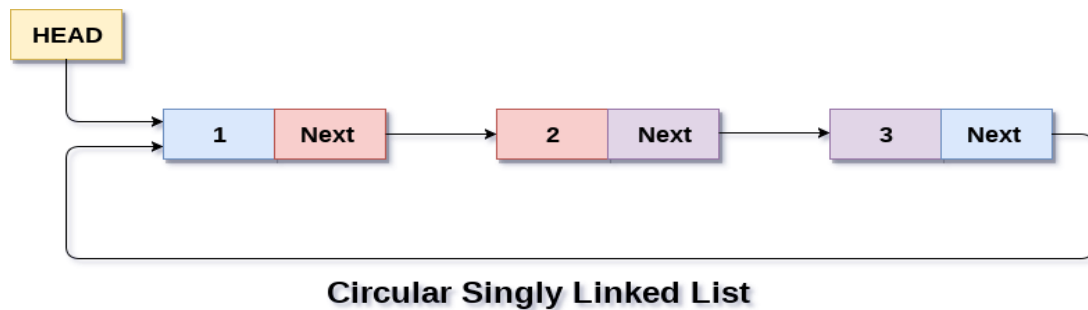
Singly Linked List	Doubly Linked List
SLL has nodes with only a data field and next link field.	DLL has nodes with a data field, a previous link field and a next link field.
In SLL, the traversal can be done using the next node link only.	In DLL, the traversal can be done using the previous node link or the next node link.
The SLL occupies less memory than DLL as it has only 2 fields.	The DLL occupies more memory than SLL as it has 3 fields.
Less efficient access to elements.	More efficient access to elements.

CIRCULAR LINKED LIST:

- ❖ Circular Linked List is a variation of Linked list in which the first element points to the last element and the last element points to the first element.
- ❖ Both Singly Linked List and Doubly Linked List can be made into a circular linked list.

1. Circular Singly Linked List:

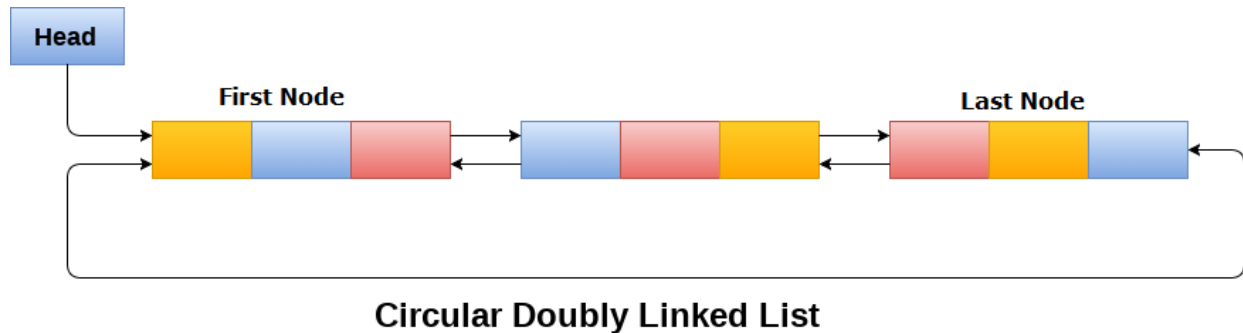
In singly linked list, the next pointer of the last node points to the first node.

**Operations on Circular singly linked list:**

1. Insert at Beginning
2. Insert at Last
3. Delete at Beginning
4. Delete at Last

2. Circular Doubly Linked List:

- ❖ Circular doubly linked list is a more complicated type of data structure in which a node contains pointers to its *previous node* as well as the *next node*.
- ❖ Circular doubly linked list *doesn't contain NULL* in any of the node.
- ❖ The last node of the list contains the address of the first node of the list.
- ❖ The first node of the list also contain address of the last node in its previous pointer.

**Operations on Circular doubly linked list:**

1. Insert at Beginning
2. Insert at Last
3. Delete at Beginning
4. Delete at Last

Applications of Linked List:

1. Implementation of stacks and queues
2. Implementation of graphs.
3. Dynamic memory allocation: We use linked list of free blocks.
4. Maintaining directory of names
5. Performing arithmetic operations on long integers
6. Manipulation of polynomials by storing constants in the node of linked list
7. Representing sparse matrices

Stack

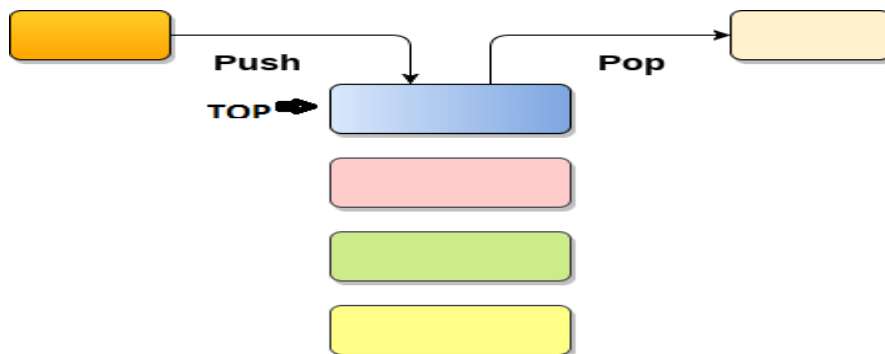
1. Stack is an ordered list in which, insertion and deletion can be performed only at one end that is called **top**.
2. Stack is a recursive data structure having pointer to its top element.
3. Stacks are sometimes called as Last-In-First-Out (LIFO) lists i.e. the element which is inserted first in the stack, will be deleted last from the stack.

Applications of Stack

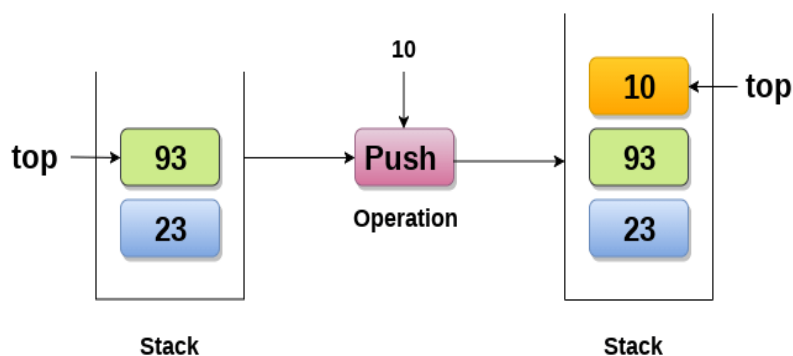
1. Recursion
2. Expression evaluations and conversions
3. Parsing
4. Browsers
5. Editors
6. Tree Traversals

Operations on Stack

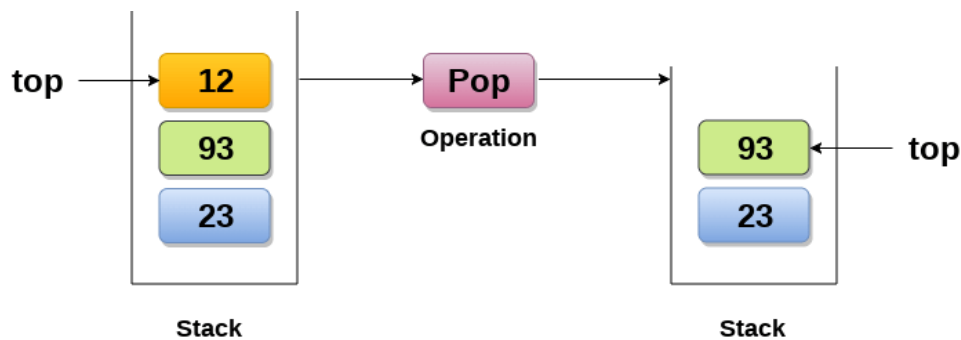
There are various operations which can be performed on stack.



1. **Push:** Adding an element onto the stack



2. **Pop:** Removing an element from the stack



Stack Implementation:

There are several ways to implement stack. They are –

1. Array Implementation
2. Linked List implementation

Array Implementation:

- ❖ Stack can be implemented using one-dimensional array. One-dimensional array is used to hold elements of a stack.
- ❖ Implementing a stack using array can store fixed number of data values.
- ❖ In a stack, initially **top is set to -1**.
- ❖ Top is used to keep track of the index of the top most elements.

```
#include<stdio.h>
#include<conio.h>
#define MAX_SIZE 101
int Arr[MAX_SIZE];
int top = -1;
void Push(int x)
{
    if(top == MAX_SIZE -1)
    {
        printf("Error: stack overflow\n");
        return;
    }
    Arr[++top] = x;
}
```

void Pop()

```
{
    if(top == -1)
    {
        printf("Error: No element to pop\n");
        return;
    }
    top--;
}
```

void Print()

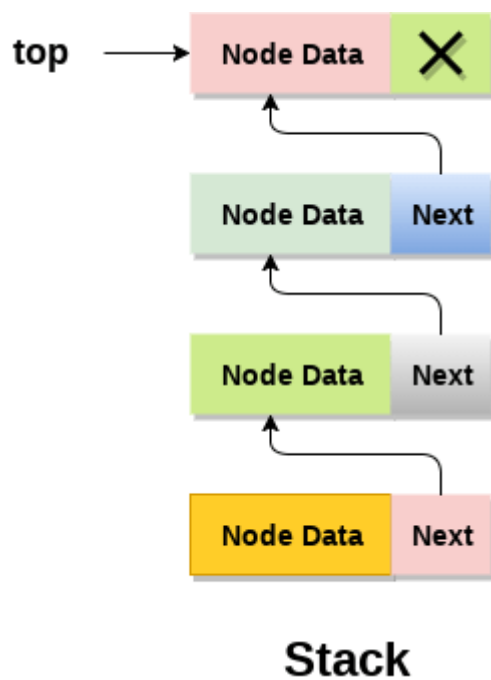
```
{
    int i;
    printf("Stack: ");
    for(i = 0; i<=top; i++)
        printf("%d ",Arr[i]);
    printf("\n");
}
```

void main()

```
{
    int choice, x;
    while (1)
    {
        printf("1.Push ...2.Pop.....3.Print....4.Exit\n");
        printf("\nEnter your choice: ");
        scanf("%d", &choice);
        switch (choice)
        {
            case 1: printf("Enter an element to push:");
                    scanf("%d",&x);
                    Push(x);
                    break;
            case 2: Pop();
                    break;
            case 3: Print();
                    break;
            case 4: exit(1);
            default: printf("Inavlid choice \n");
        }
    }
}
```

Linked List implementation of Stack:

- ❖ Instead of using array, we can also use linked list to implement stack.
- ❖ Linked list allocates the memory dynamically. However, time complexity in both the scenario is same for all the operations i.e. push, pop and peek.
- ❖ In linked list implementation of stack, the nodes are maintained non-contiguously in the memory.
- ❖ Each node contains a pointer to its immediate successor node in the stack.



Program:

//Stack using Linked List:

```
#include<stdio.h>
#include<conio.h>
struct node
{
    int data;
    struct node *next;
};
struct *Top = NULL;
void push(int x)
{
    struct node *newnode = (struct *node) malloc(sizeof(struct node));
```



```

        newnode->data = x;
        newnode->next = Top;
        Top = newnode;
    }
void pop();
{
    struct node *temp;
    if(Top == NULL)
    {
        printf("\n Stack is Empty....\n");
        return;
    }
    temp = Top;
    Top = Top->next;
    free(temp);
}
void Print()
{
    struct node *temp=Top;
    if(Top == NULL)
    {
        printf("\nStack is Empty.....\n");
        return;
    }
    printf("Stack Elements are....\n");
    while(temp!=NULL)
    {
        printf("\n%d", temp->data);
        temp = temp->next;
    }
}
void main()
{
    int x, choice;

```

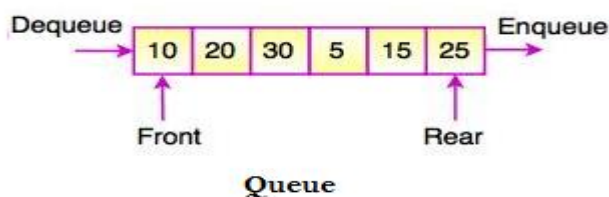
```

while(1)
{
    printf("1.Push ...2.Pop.....3.Print....4.Exit\n");
    printf("\nEnter your choice: ");
    scanf("%d", &choice);
    switch (choice)
    {
        case 1: printf("Enter an element to push:");
                scanf("%d",&x);
                Push(x);
                break;
        case 2: Pop();
                break;
        case 3: Print();
                break;
        case 4: exit(1);
        default: printf("Invalid choice \n");
    }
}
}

```

QUEUE:

- ❖ Queue is a linear data structure where the element is inserted at one end called **REAR** and deleted from the other end called as **FRONT**.
- ❖ **Front** points to the **beginning** of the queue and **Rear** points to the **end** of the queue.
- ❖ Queue follows the **FIFO (First - In - First Out)** structure.
- ❖ According to its FIFO structure, element inserted first will also be removed first.
- ❖ The **enqueue()** and **dequeue()** are two important operations used in a queue.



Operations on Queue:

Following are the basic operations performed on a Queue.

Operations	Description
enqueue()	This function defines adding an element into queue.
dequeue()	This function defines removing an element from queue.

Queue Implementation:

There are several ways to implement queue. They are

1. Array implementation
2. Linked List implementation

Array implementation:

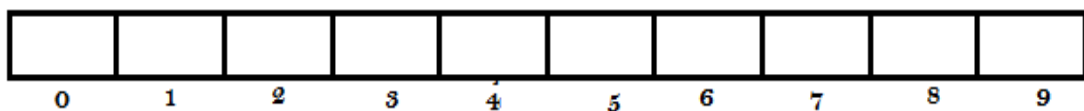
Array is the easiest way to implement a queue.

Array Name: Array_queue[10]

Input Values: 11, 12, 13, 14, 15, 16, 17, 18, 19, 20

Step 1: Initially front and rear value is -1. (Queue is Empty)

front = -1

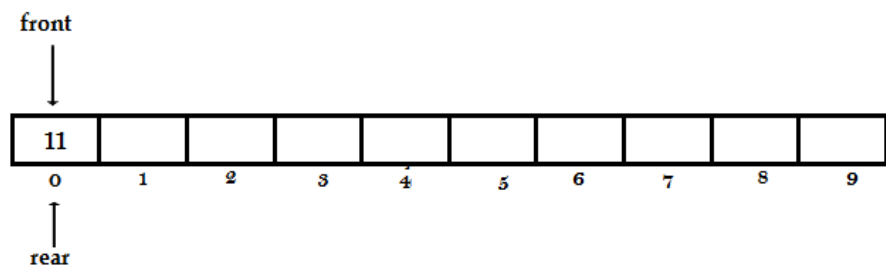


rear = -1

Step 2:

Enqueue (11)

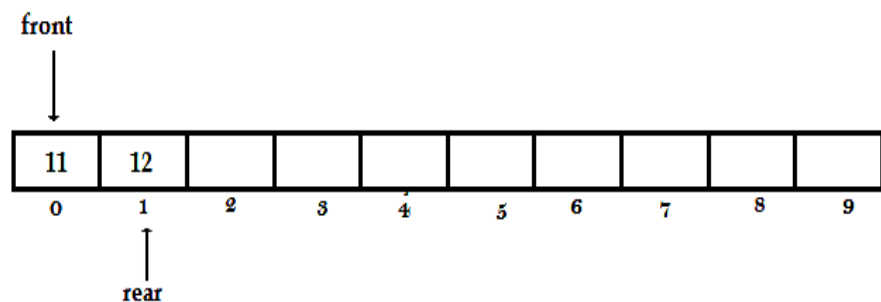
front = rear = 0
Qarray[rear] = 11



Step 3:

Enqueue (12)

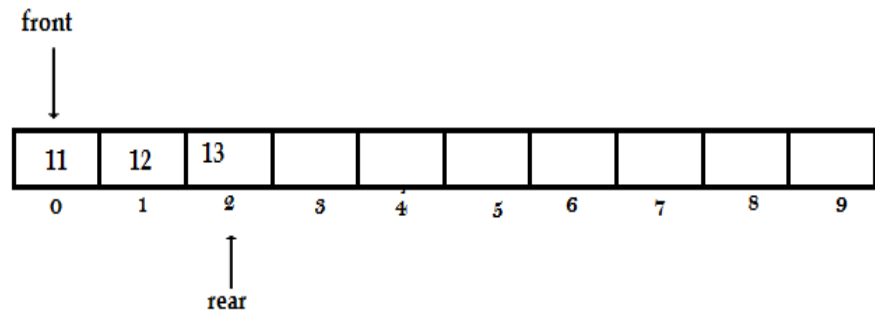
rear = rear + 1
Qarray[rear] = 12



Step 4:

Enqueue (13)

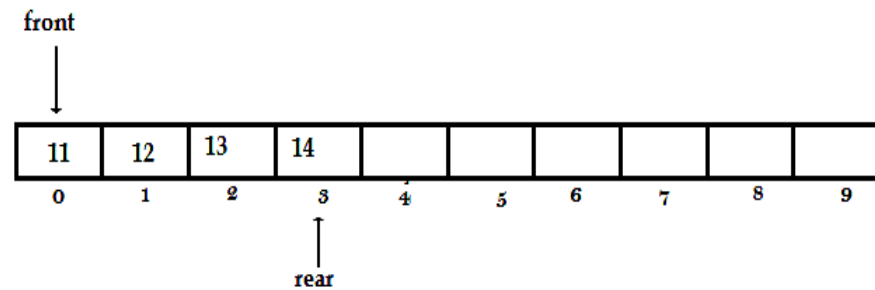
$\text{rear} = \text{rear} + 1$
 $\text{Qarray}[\text{rear}] = 13$



Step 5:

Enqueue (14)

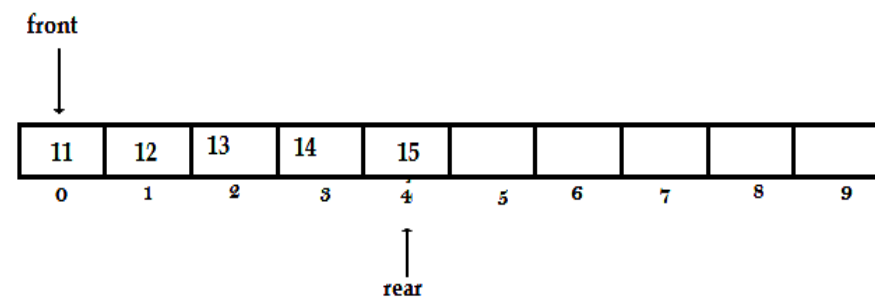
$\text{rear} = \text{rear} + 1$
 $\text{Qarray}[\text{rear}] = 14$



Step 6:

Enqueue (15)

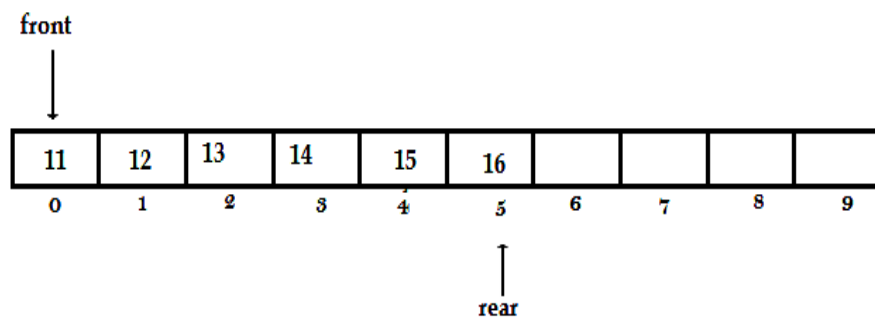
$\text{rear} = \text{rear} + 1$
 $\text{Qarray}[\text{rear}] = 15$



Step 7:

Enqueue (16)

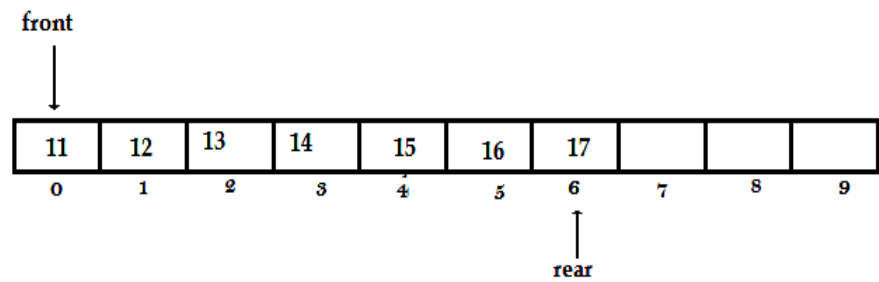
$\text{rear} = \text{rear} + 1$
 $\text{Qarray}[\text{rear}] = 16$



Step 8:

Enqueue (17)

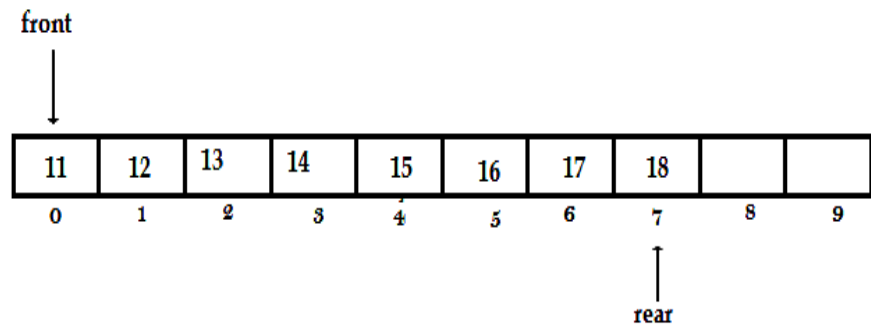
$\text{rear} = \text{rear} + 1$
 $\text{Qarray}[\text{rear}] = 17$



Step 9:

Enqueue (18)

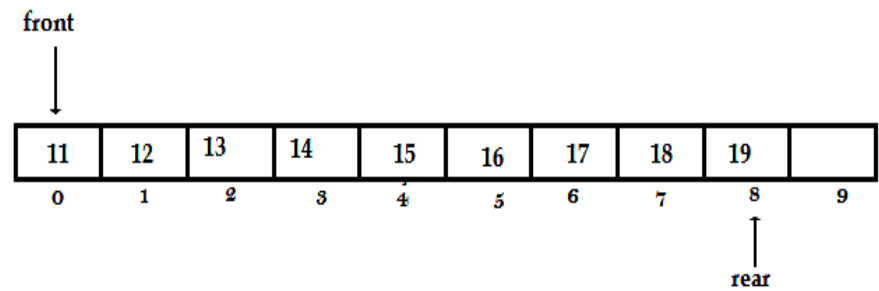
$\text{rear} = \text{rear} + 1$
 $\text{Qarray}[\text{rear}] = 18$



Step 9:

Enqueue (19)

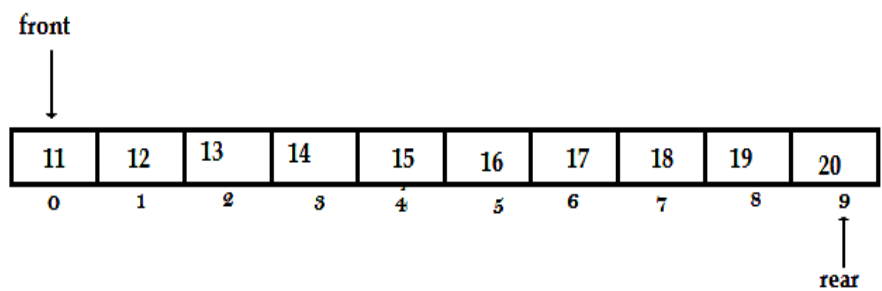
$\text{rear} = \text{rear} + 1$
 $\text{Qarray}[\text{rear}] = 19$



Step 10:

Enqueue (20)

$\text{rear} = \text{rear} + 1$
 $\text{Qarray}[\text{rear}] = 20$

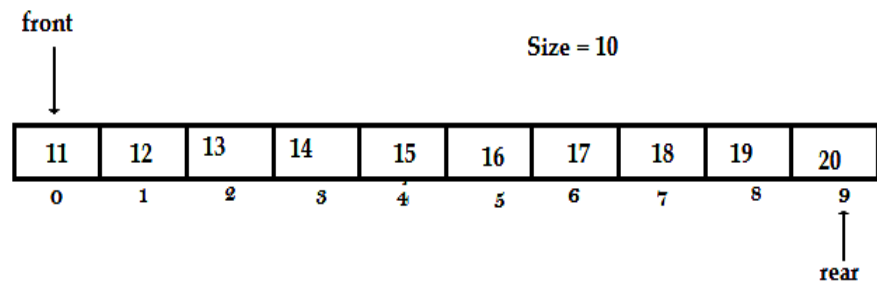


Queue Overflow:

Enqueue (22)

if(rear == Size -1)

" Queue if Overflow"

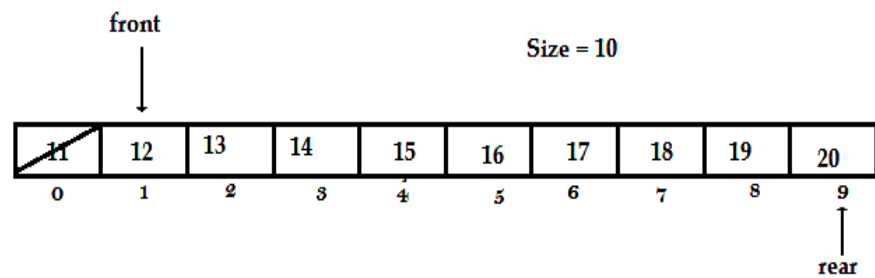


DEQUEUE:

Step 1:

Dequeue()

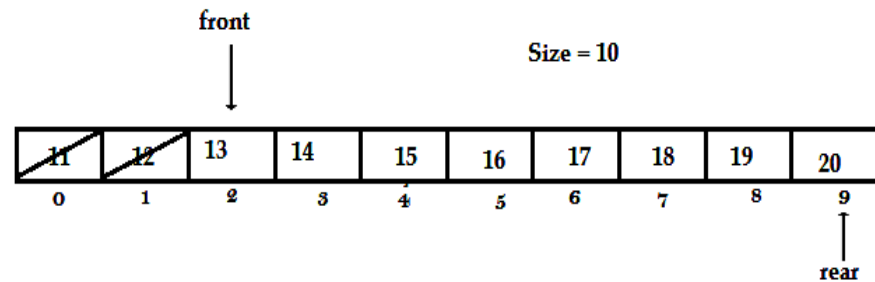
print Qarray[front]
front = front + 1



Step 2:

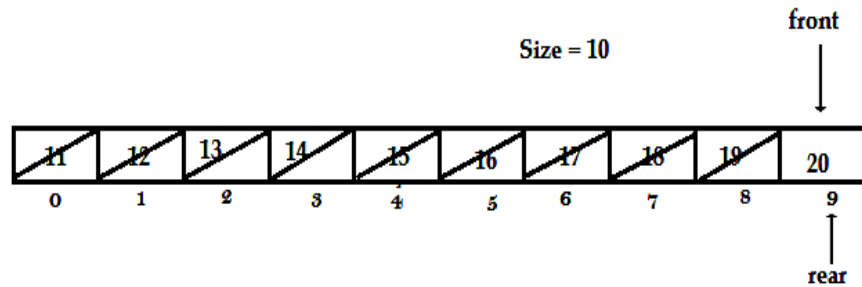
Dequeue()

print Qarray[front]
front = front + 1



Step 3 to Step 9:

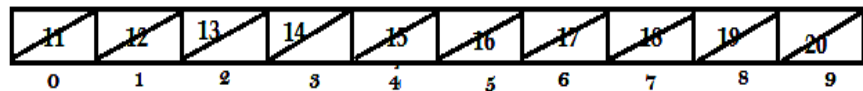
Step 10:



Dequeue()

front = -1

```
if(rear == front)
    front = rear = -1
```



rear = -1

Program:

//Queue implementation using Array

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

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

```
#define MAX 50
```

```
int Qarray[MAX];
```

```
void Enqueue(int);
```

```
void Dequeue();
```

```
void Printqueue();
```

```
int rear = - 1;
```

```
int front = - 1;
```

```
void main()
```

```
{
```

```
    int choice, x;
```

```
    while (1)
```

```
    {
```

```
        printf("1.Enqueue ...2.DeQueue.....3.PrintQueue....4.Exit\n");
```

```
        printf("\nEnter your choice: ");
```

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

```
        switch (choice)
```

```
        {
```

```
            case 1: printf("Enter an element to Enqueue:");
```

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

```
                    Enqueue(x);
```

```
                    break;
```

```
            case 2:delete();
```

```
                    break;
```

```
            case 3: display();
```

```
                    break;
```

```
            case 4: exit(1);
```

```
            default:      printf("Invalid choice \n");
```

```
        }
```

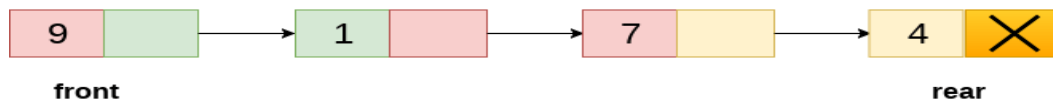
```

    }
}
void Enqueue(int x)
{
    if (rear == MAX - 1)
    {
        printf("Error:.....Queue Overflow \n");
    }
    else if (front == -1 && rear == -1)
        front = rear = 0;
    else
        rear = rear + 1;
    Qarray[rear] = x;
}
}
Dequeue()
{
    if (front == - 1 || front == -1)
    {
        printf("\nError...Queue Underflow \n");
        return ;
    }
    else if(front == rear)
    {
        printf("Dequeued Element is : %d\n", Qarray[front]);
        front = rear = -1;
    }
    else
    {
        printf("Dequeued Element is : %d\n", Qarray[front]);
        front = front + 1;
    }
}
}
void display()
{
    int i;
    if (front == -1 && rear == -1)
    {
        printf("\nError....Queue is empty \n");
        return;
    }
    else
    {
        printf("Queue is : \n");
        for (i = front; i <= rear; i++)
            printf("%d ", Qarray[i]);
        printf("\n");
    }
}
}

```


Linked List implementation of Queue:

- ❖ The array implementation cannot be used for the large scale applications where the queues are implemented.
- ❖ One of the alternatives of array implementation is linked list implementation of queue.
- ❖ In a linked queue, each node of the queue consists of two parts i.e. **data part** and the **link part**.
- ❖ Each element of the queue points to its immediate next element in the memory.



Linked Queue

/*Queue - Linked List implementation*/

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

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

```
struct Node
```

```
{
```

```
    int data;
```

```
    struct Node* next;
```

```
};
```

```
struct Node* front = NULL;
```

```
struct Node* rear = NULL;
```

```
int main()
```

```
{
```

```
    int choice, x;
```

```
    clrscr();
```

```
    while(1)
```

```
    {
```

```
        printf("1.Enqueue element to Queue \n");
```

```
        printf("2.Dequeue element from Queue \n");
```

```
        printf("3.Print all elements of queue \n");
```

```
        printf("4.Quit \n");
```

```
        printf("Enter your choice : ");
```

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

```
        switch (choice)
```

```
        {
```

```
            case 1:
```

```
                printf("Enter the element to Enqueue:");
```

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

```
                Enqueue(x);
```

```
                break;
```

```
            case 2:
```

```
                Dequeue();
```

```
                break;
```

```
            case 3:
```

```

        PrintQ();
        break;
    case 4:
        exit(1);
    default:
        printf("\n Wrong Choice");
    }
}

void Enqueue(int x)
{
    struct Node* newnode = (struct Node*)malloc(sizeof(struct Node));
    newnode->data = x;
    newnode->next = NULL;
    if(front == NULL && rear == NULL)
    {
        front = rear = newnode;
        return;
    }
    rear->next = newnode;
    rear = newnode;
}

```

```

void Dequeue()
{
    struct Node* temp = front;
    if(front == NULL)
    {
        printf("Queue is Empty\n");
        return;
    }
    if(front == rear)
    {
        front = rear = NULL;
    }
    else
    {
        front = front->next;
    }
    free(temp);
}

```

```

void PrintQ()
{
    struct Node* temp = front;
    while(temp != NULL)
    {
        printf("%d ",temp->data);
        temp = temp->next;
    }
}

```

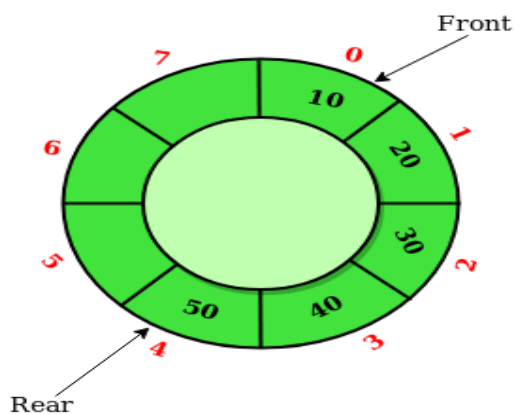
```

    printf("\n");
}

```

CIRCULAR QUEUE:

- ❖ In Circular Queue, Enqueue of a new element is performed at the very first location of the queue if the last location of the queue is full.
- ❖ In such case the first element comes after the last element.



Program for Circular Queue:

```

#include<stdio.h>
#include<conio.h>
#include<stdbool.h>
#define MAX_SIZE 101
int A[MAX_SIZE];
int front=-1, rear=-1;
bool IsEmpty();
bool IsFull();
void Enqueue(int);
void Dequeue();
int Front();
void PrintQ();
int main()
{
    int choice, x;
    clrscr();
    while(1)
    {
        printf("1.Enqueue element to Queue \n");
        printf("2.Dequeue element from Queue \n");
        printf("3.Display all elements of queue \n");
        printf("4.Quit \n");
        printf("Enter your choice : ");
        scanf("%d", &choice);

```

```

        switch (choice)
        {
            case 1:
                printf("Enter the element to Enqueue:");
                scanf("%d",&x);
                Enqueue(x);
                break;
            case 2:
                Dequeue();
                break;
            case 3:
                PrintQ();
                break;
            case 4:
                exit(1);
            default:
                printf("\n Wrong Choice");
        }
    }
}

bool IsEmpty()
{
    return(front == -1 && rear == -1);
}

bool IsFull()
{
    return(rear+1)%MAX_SIZE == front ? true : false;
}

void Enqueue(int x)
{
    printf("Enqueuing...%d\n",x);
    if(IsFull())
    {
        printf("Error: Queue is Full\n");
        return;
    }
    if(IsEmpty())
    {
        front=rear=0;
    }
    else
    {
        rear = (rear+1)%MAX_SIZE;
    }
    A[rear] = x;
}

void Dequeue()
{
    printf("Dequeuing\n");

```

```

        if(IsEmpty())
        {
            printf("Error: Queue is Empty\n");
            return;
        }
        else if(front==rear)
        {
            rear=front=-1;
        }
        else
        {
            front=(front+1)%MAX_SIZE;
        }
    }
    void PrintQ()
    {
        int count=(rear+MAX_SIZE-front)%MAX_SIZE+1;
        printf("Queue  :");
        for(int i=0;i<count;i++)
        {
            int index=(front+i)%MAX_SIZE;
            printf("%d",A[index]);
        }
        printf("\n\n");
    }
}

```

Difference between Linear Queue and Circular Queue:

Comparison	Linear Queue	Circular Queue
Basic	Organizes the data elements and instructions in a sequential order one after the other.	Arranges the data in the circular pattern where the last element is connected to the first element.
Order of task execution	Tasks are executed in order they were placed before (FIFO).	Order of executing a task may change.
Insertion & Deletion	The new element is added from the rear end and removed from the front.	Insertion and deletion can be done at any position.

1. Conversion of Infix to Postfix

1. Read the characters from left to right.
2. If the **character is operand**, then insert it into postfix string.
3. If the character is operator, then if the stack is empty or contains a left parenthesis on top, push the incoming operator onto the stack.
4. If the character is a left parenthesis, push it on the stack.
5. If the character is a right parenthesis, pop the stack and print the operators **until you see a left parenthesis**. Discard the pair of parentheses.
6. If the **incoming operator has higher precedence** than the top of the stack, push it on the stack.
7. If the **incoming operator has lower precedence** than the symbol on the top of the stack, pop the stack and print the top operator. Then test the incoming operator against the new top of stack.
8. If the **incoming operator has equal precedence** with the top of the stack, use association. If the association is **left to right**, pop and print the top of the stack and then push the incoming operator. If the association is **right to left**, push the incoming operator.
9. At the end of the expression, pop and print all operators on the stack. (No parentheses should remain.)

U18PCCS301 - DATA STRUCTURES & ALGORITHM
Unit – III TREES

Unit – III TREE

Syllabus:

Trees - Tree Terminologies, Different types of Trees – Binary trees – Properties of binary trees – Representation of binary trees – Binary tree traversal –Heap – Binary search tree- Binary search tree implementations –AVL trees-Threaded Binary Tree-B Tree, B+ Tree – Splay tree – Applications

- ❖ A Tree is a **non-linear** data structure containing the set of one or more data nodes where one node is designated as the **root** of the tree while the remaining nodes are called as the **children** of the root.
- ❖ The nodes other than the root node are partitioned into the **non empty sets** where each one of them is to be called **sub-tree**.
- ❖ A node can have **any number of children** nodes but it can have **only one parent**.
- ❖ The following image shows a tree, where the node A is the root node of the tree while the other nodes can be seen as the children of A.

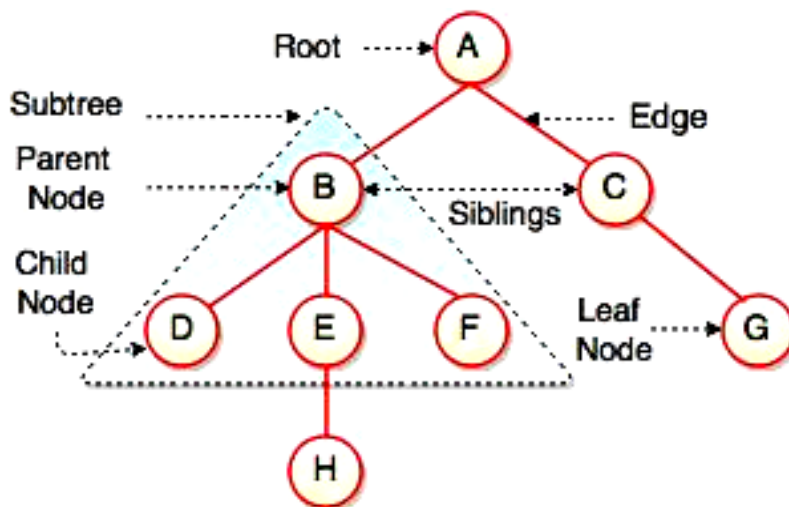


Fig. Structure of Tree

Basic terminology:

Root Node: - The root node is the topmost node in the tree hierarchy. In other words, the root node is the one which doesn't have any parent.

Sub Tree: - If the root node is not null, the tree T1, T2 and T3 is called **sub-trees** of the root node.

Leaf Node: - The node of tree, which **doesn't have any child node**, is called **leaf node**. Leaf node is the **bottom most node** of the tree. There can be any **number of leaf nodes** present in a general tree. Leaf nodes can also be called **external nodes**.

U18PCCS301 - DATA STRUCTURES & ALGORITHM
Unit – III TREES

Path: - The sequence of consecutive edges is called path. In the tree shown in the above image, path to the node E is $A \rightarrow B \rightarrow E$.

Levels of a node:

- ❖ Levels of a node represent the number of connections between the node and the root. It represents generation of a node.
- ❖ If the root node is at level 0, its next node is at level 1, its grand child is at level 2 and so on. Levels of a node can be shown as follows:

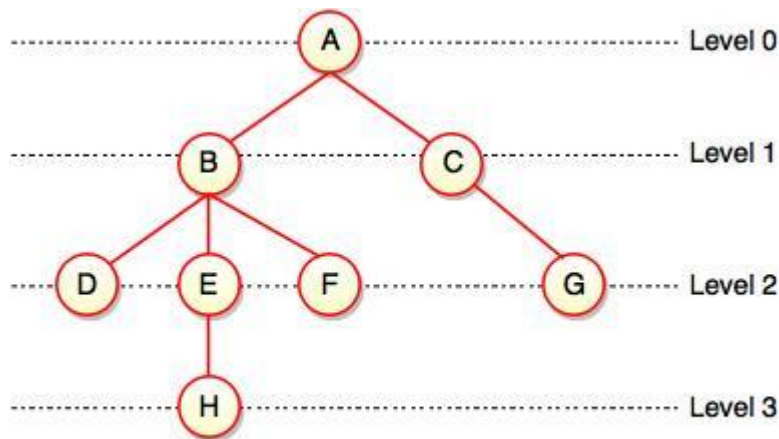


Fig. Levels of Tree

Height of a Node:

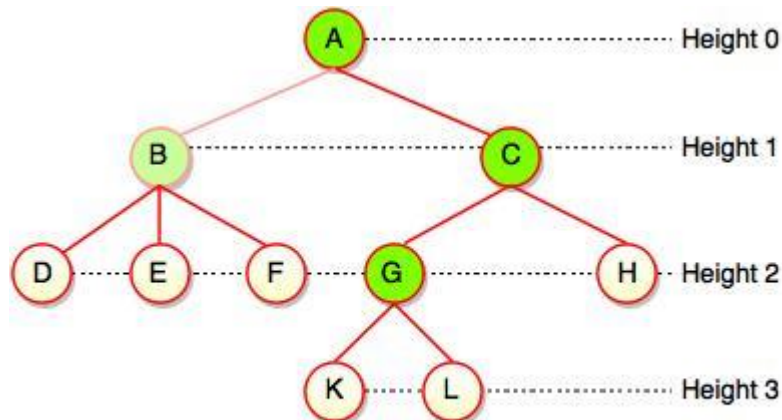


Fig. Height of a Node

- ❖ Height of a node is a number of edges on the longest path between that node and a leaf. Each node has height.
- ❖ In the above figure, A, B, C, D can have height.
- ❖ Leaf cannot have height as there will be no path starting from a leaf.
- ❖ Node A's height is the number of edges of the path to K not to D. And its height is 3.

U18PCCS301 - DATA STRUCTURES & ALGORITHM
Unit – III TREES

Degree of a node:

- ❖ The degree of the tree is the total number of it's children i-e the total number nodes that originate from it.
- ❖ The degree of a node is the number of partitions in the subtree which has that node as the root.
- ❖ The leaf node does not have any child so its degree is zero.

Advantages of Tree:

- 1) Tree reflects structural relationships in the data.
- 2) It is used to represent hierarchies.
- 3) It provides an efficient insertion and searching operations.
- 4) Trees are flexible. It allows moving sub-trees around with minimum effort.

BINARY TREE:

- ❖ Binary tree is a special type of data structure. In binary tree, every node can have a maximum of 2 children, which are known as **Left child** and **Right Child**.
- ❖ **Root** of the tree.
- ❖ **Left sub-tree** which is also a binary tree.
- ❖ **Right sub-tree** which is also a binary tree.

Definition: "A tree in which every node can have maximum of two children is called as Binary Tree."

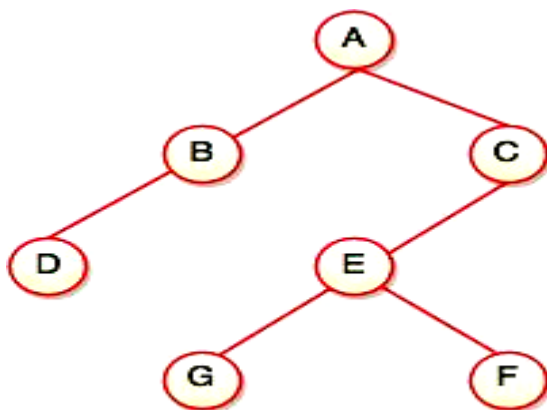


Fig. Binary Tree

Representation of Binary Tree:

There are two ways to represent binary tree, they are

- 1) Array Representation
- 2) Linked Representation

U18PCCS301 - DATA STRUCTURES & ALGORITHM
Unit – III TREES

Array Representation of Trees:

- ❖ Arrays are used to represent the binary trees.
- ❖ Then an array of **size 2^k** is declared where **k is the depth** of the tree. For e.g. if the depth of the tree is 3, the maximum $2^3-1=7$ elements will be present in the tree and hence the array size is 8.
- ❖ For any element in position i, the left child is in position $2i+1$, the right child is in position $(2i + 2)$, and the parent is in position $[(i-1)/2]$

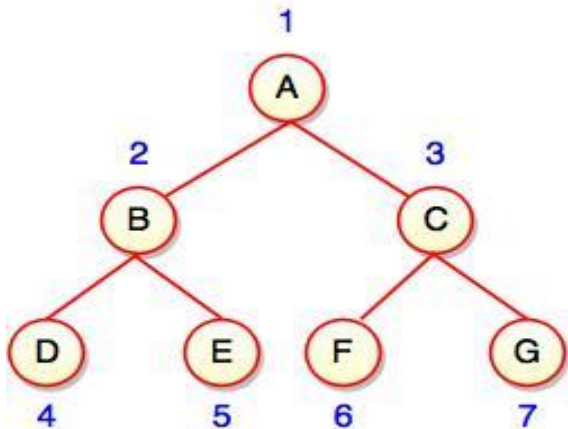


Fig. Binary Tree using Array

0	1	2	3	4	5	6
A	B	C	D	E	F	G

Location Number of an Array in a Tree

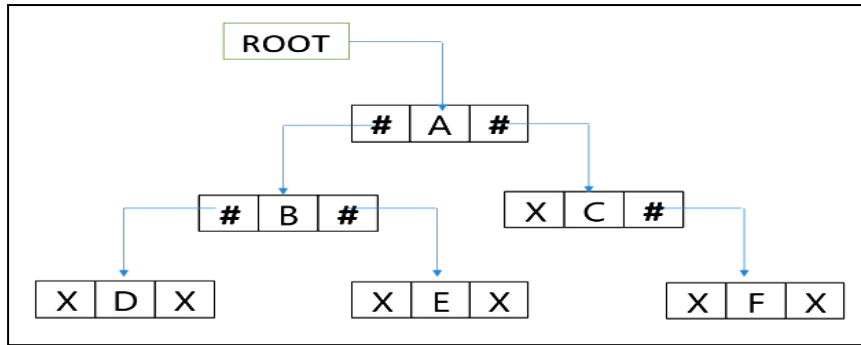
Linked List Representation of Binary Tree:

The elements are represented using pointers. Each node in linked representation has three fields, namely,

Left	Data	Right
-------------	-------------	--------------

- ❖ Left Pointer points to the left sub-tree.
- ❖ Data field stores actual data.
- ❖ Right Pointer points to the right sub-tree.
- ❖ The leaf nodes, both the pointer fields are assigned as NULL.

U18PCCS301 - DATA STRUCTURES & ALGORITHM
Unit – III TREES



Properties of Binary Trees:

Property 1: There is exactly one path connecting any two nodes in a tree.

Property 2: A tree with N nodes has N-1 Edges.

Property 3: A Full Binary Tree with N Internal nodes has N+ 1 External node.

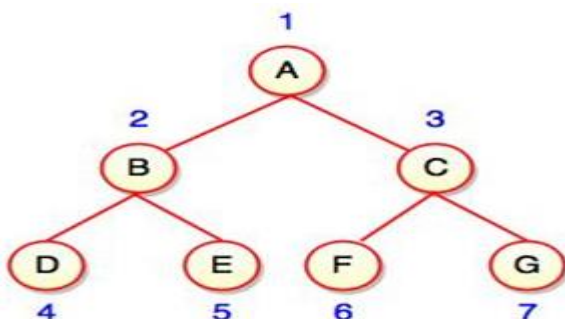
Property 4: The Height of Complete Binary Tree with N Internal nodes is about $\log_2 N$.

Types of Binary Trees:

- 1) Full Binary Tree
- 2) Perfect Binary Tree
- 3) Complete Binary Tree
- 4) Skewed Binary Tree
- 5) Strict Binary Tree

1. Full Binary Tree:

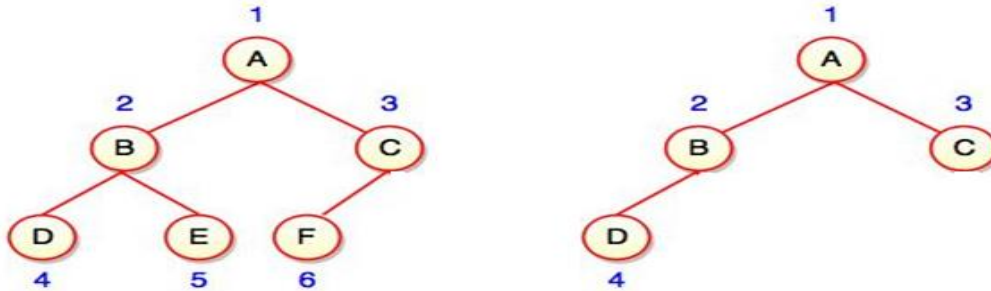
- i. A binary tree is said to be full binary tree if each of its node has **two children** or **no children** at all.
- ii. Every level is completely filled up.
- iii. Number of node at any level i in a full binary tree is 2^i .



2. Complete Binary Tree:

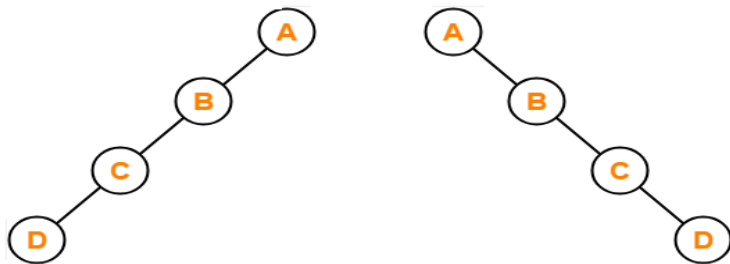
A complete binary tree is defined as a binary tree where

- i. All leaf nodes are on level n or $n-1$.
- ii. Levels are filled from left to right.
- iii. Example: Heap



3. Skewed Binary Tree:

- i. A skewed binary tree could be skewed to the left or skewed to the right.
- ii. In a left skewed binary tree, most of the nodes have the left child without corresponding right child.
- iii. In a right skewed binary tree, most of the nodes have the right child without corresponding left child.
- iv. Example: Binary Search Tree.

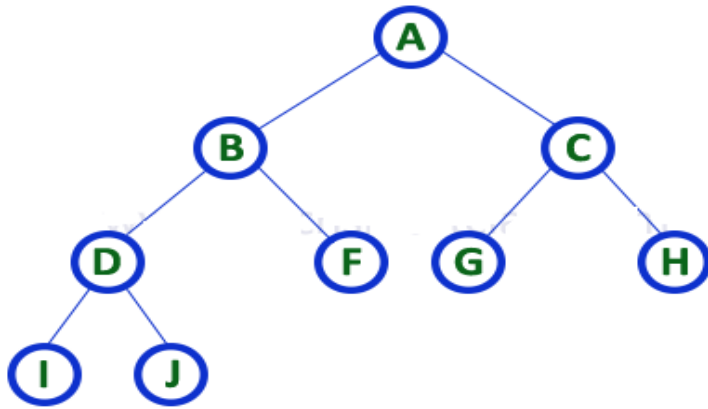


Left Skewed Binary Tree

Right Skewed Binary Tree

4. Strict Binary Tree:

- i. If every non-terminal node in binary tree consists of non-empty left sub-tree and right sub-tree.
- ii. A node will have either two children or no children at all.



BINARY TREE TRAVERSAL:

Traversing means visiting each node only once. Tree traversal is a method for visiting all the nodes in the tree exactly once. There are three types of tree traversal techniques, namely

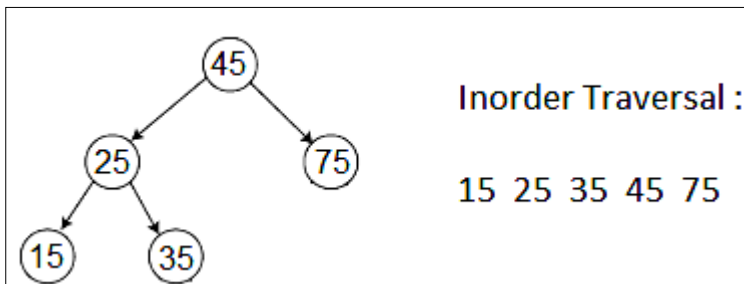
1. Inorder Traversal
2. Preorder Traversal
3. Postorder Traversal

1. Inorder Traversal (Left – Root – Right)

The Inorder traversal of a binary tree is performed as

1. Traverse the left subtree in Inorder
2. Visit the root
3. Traverse the right subtree in Inorder.

Example:



Recursive Routine for Inorder Traversal

```
void Inorder (Tree T)
{
    if (T != NULL)
    {
        Inorder (T->left);
```

U18PCCS301 - DATA STRUCTURES & ALGORITHM
Unit – III TREES

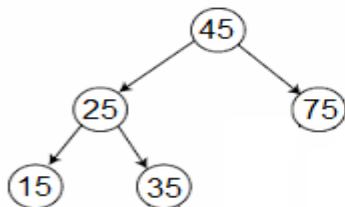
```
        printElement (T->Element);  
        Inorder (T->right);  
    }  
}
```

2. Preorder Traversal (Root – Left – Right):

The preorder traversal of a binary tree is performed as follows,

1. Visit the root
2. Traverse the left subtree in Preorder
3. Traverse the right subtree in Preorder.

Example:



Preorder Traversal

45 25 15 35 75

Recursive Routine for Preorder Traversal

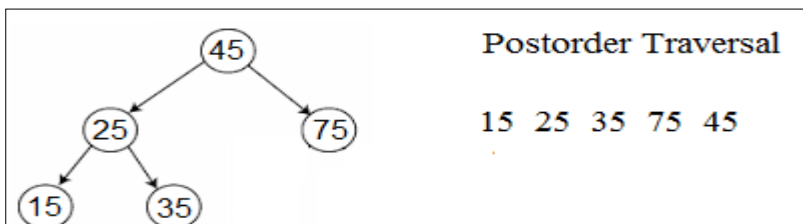
```
void Preorder (Tree T)  
{  
    if (T != NULL)  
    {  
        printElement (T->Element);  
        Preorder (T->left);  
        Preorder (T->right);  
    }  
}
```

3. Postorder Traversal (Left – Right – Root)

The Postorder traversal of a binary tree is performed by the following steps.

1. Traverse the left subtree in Postorder.
2. Traverse the right subtree in Postorder.
3. Visit the root.

Example:



Postorder Traversal

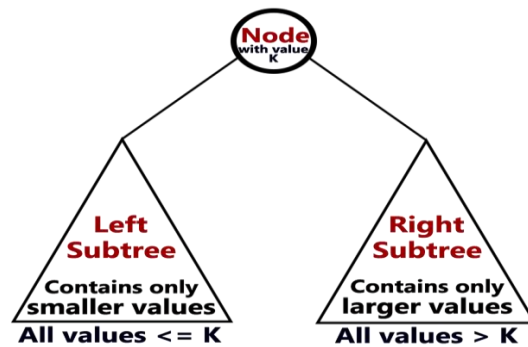
15 25 35 75 45

Recursive Routine for Postorder Traversal

```
void Postorder (Tree T)
{
    if (T != NULL)
    {
        Postorder (T->Left);
        Postorder (T->Right);
        PrintElement (T->Element);
    }
}
```

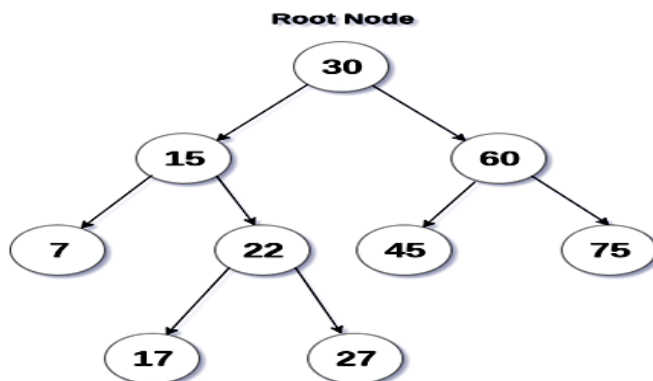
BINARY SEARCH TREE:

1. Binary Search tree is a binary tree.
2. In a binary search tree, the value of all the nodes in the *left sub-tree* is *less than* the value of the *root*.
3. Similarly, value of all the nodes in the *right sub-tree* is *greater than* to the value of the *root*.



Definition:

"Binary Search Tree is a binary tree where each node contains only smaller values in its left sub-tree and only larger values in its right sub-tree."



U18PCCS301 - DATA STRUCTURES & ALGORITHM
Unit – III TREES

Advantages of using binary search tree

1. Searching becomes very efficient in a binary search tree.
2. The binary search tree is considered as efficient data structure in compare to arrays and linked lists.
3. It also speed up the insertion and deletion operations as compare to that in array and linked list.

Operations on Binary Search Tree:

There are many operations which can be performed on a binary search tree.

S. No	Operation	Description
1	Searching in BST	Finding the location of some specific element in a binary search tree.
2	Insertion in BST	Adding a new element to the binary search tree at the appropriate location so that the property of BST do not violate.
3	Deletion in BST	Deleting some specific node from a binary search tree. However, there can be various cases in deletion depending upon the number of children, the node have.

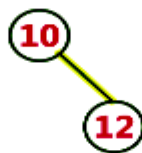
Q. Create the binary search tree using the following data elements.

10, 12, 5, 4, 20, 8, 7, 15 and 13

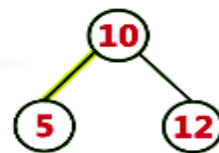
insert (10)



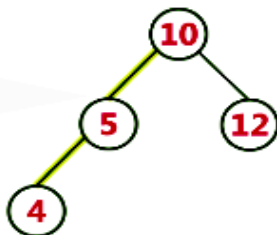
insert (12)



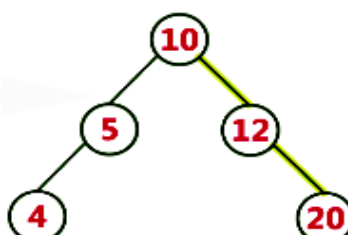
insert (5)



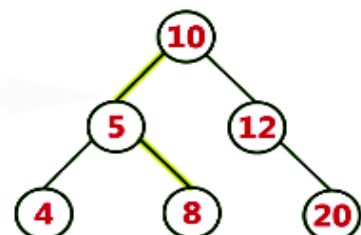
insert (4)



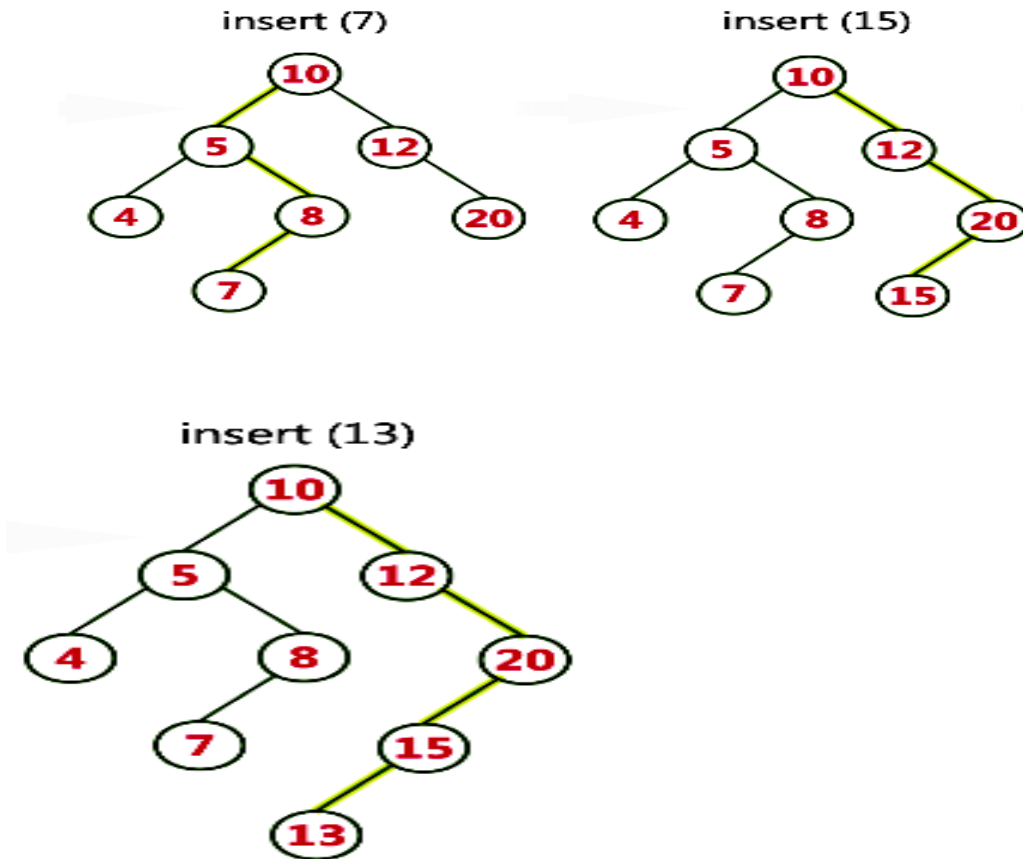
insert (20)



insert (8)



U18PCCS301 - DATA STRUCTURES & ALGORITHM
Unit – III TREES



Program Code:

```
#include<stdio.h>
#include<conio.h>
#include<stdlib.h>
struct node
{
    int data;
    struct node *left;
    struct node *right;
};
void inorder(struct node *root)
{
    if(root)
    {
        inorder(root->left);
        printf(" %d",root->data);
        inorder(root->right);
    }
}
int main()
{
    int n , i;
    struct node *p , *q , *root;
    printf("Enter the number of nodes to be insert: ");
    scanf("%d",&n);
```

U18PCCS301 - DATA STRUCTURES & ALGORITHM
Unit – III TREES

```
printf("\nPlease enter the numbers to be insert: ");

for(i=0;i<i++)
{
    p = (struct node*)malloc(sizeof(struct node));
    scanf("%d",&p->data);
    p->left = NULL;
    p->right = NULL;
    if(i == 0)
    {
        root = p; // root always point to the root node
    }
    else
    {
        q = root; // q is used to traverse the tree
        while(1)
        {
            if(p->data > q->data)
            {
                if(q->right == NULL)
                {
                    q->right = p;
                    break;
                }
                else
                {
                    q = q->right;
                }
            }
            else
            {
                if(q->left == NULL)
                {
                    q->left = p;
                    break;
                }
                else
                {
                    q = q->left;
                }
            }
        }
    }
}

printf("\nBinary Search Tree nodes in Inorder Traversal: ");
inorder(root);
printf("\n");

return 0;
}
```

Deletion Operation in BST:

Deletion operation is the complex operation in the Binary search tree. To delete an element, consider the following three possibilities.

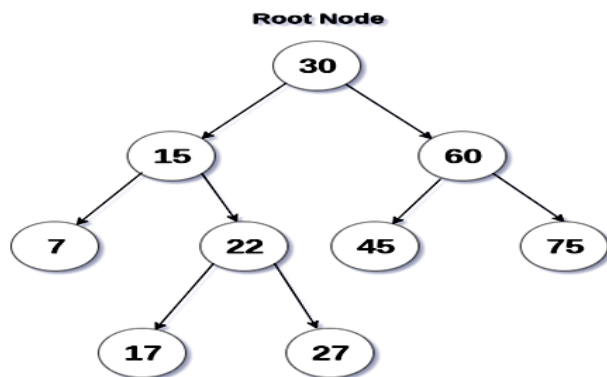
CASE 1: Node has no children.

CASE 2: Node has one child.

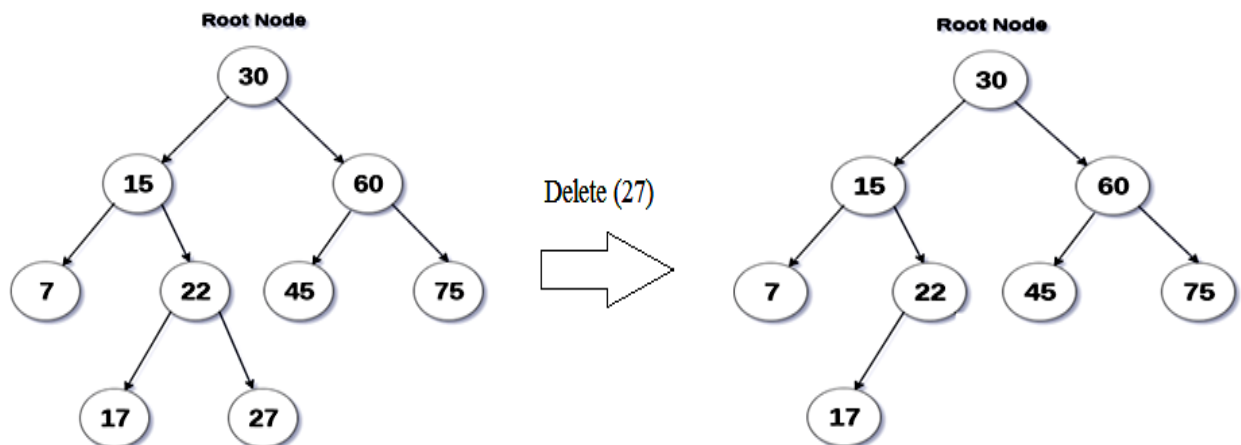
CASE 3: Node has two children.

CASE: 1 [Node with no children]

If the node is a leaf node, it can be deleted immediately.



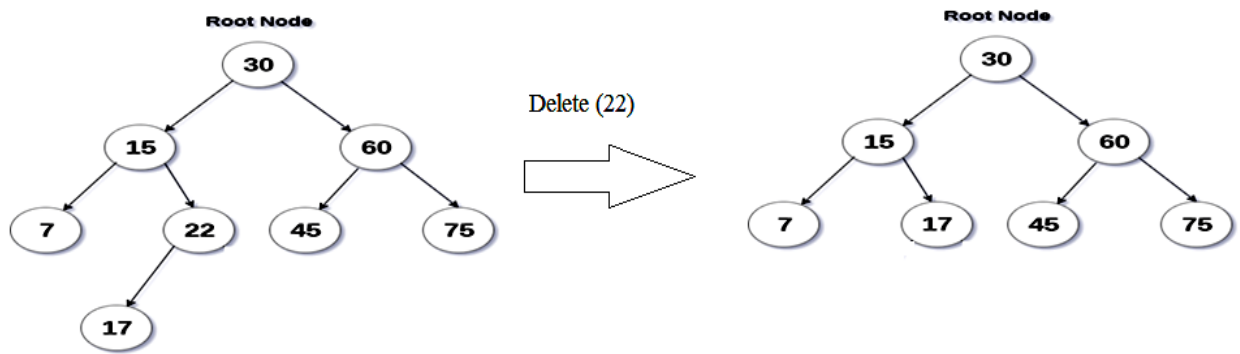
Delete (27)



CASE: 2 [Node with one child]

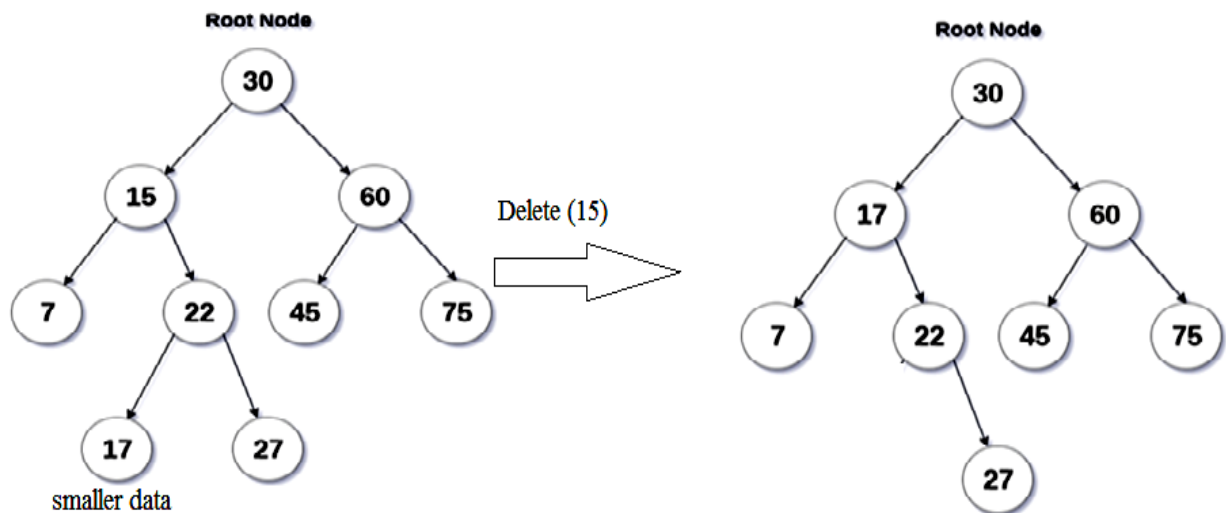
If the node has one child, it can be deleted by adjusting its parent pointer that points to its child node.

U18PCCS301 - DATA STRUCTURES & ALGORITHM
Unit – III TREES



Case: 3 [Node with two children]

- It is difficult to delete a node which has two children.
- The general strategy is to replace the data of the node to be deleted with its smallest data of the right subtree or largest data of the left subtree.



AVL TREE:

- i. An AVL tree is a height balanced Binary Search tree.
- ii. AVL tree is a binary search tree in which the height of two siblings is not permitted to differ more than one.

$$| \text{Height of the Left subtree} - \text{Height of Right subtree} | \leq 1$$

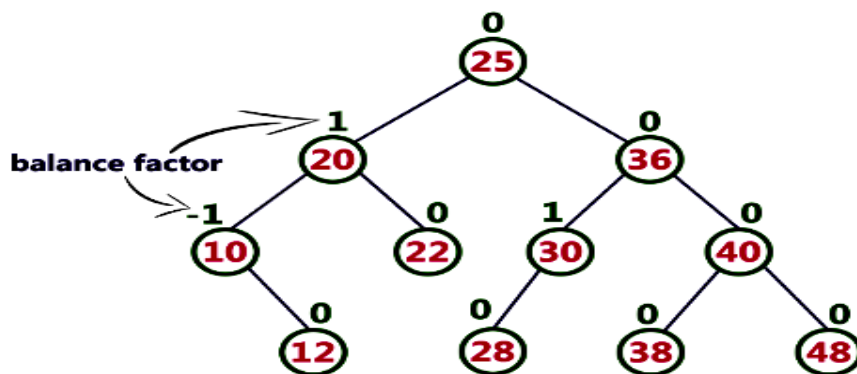
Balance Factor:

The balance factor (BF) should be either 0, 1, -1.

$$\text{Balance Factor (BF)} = \text{Height of Left subtree} - \text{Height of Right subtree}$$

Structure of a Node in AVL tree:

```
struct node
{
    int data;
    struct node *left, *right;
    int height;
}*node;
```



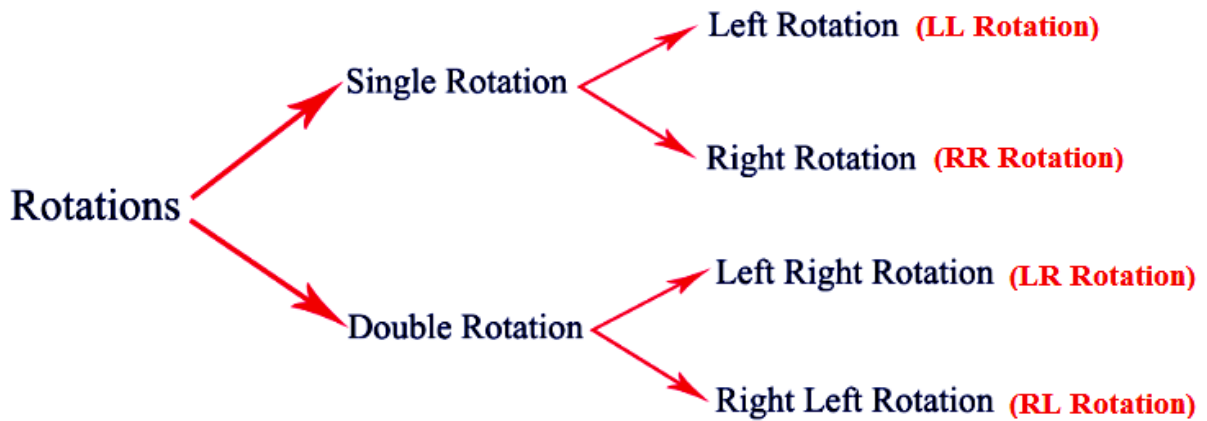
❖ The above tree is a binary search tree and every node is satisfying balance factor condition. So this tree is said to be an AVL tree.

Note: Every AVL Tree is a binary search tree but every Binary Search Tree need not be AVL tree.

AVL Tree Rotations:

- ❖ In AVL tree, after performing operations like *insertion* and *deletion* we need to *check the balance factor* of every node in the tree.
- ❖ If every node *satisfies the balance factor* condition then we conclude the operation otherwise we must make it balanced.
- ❖ Whenever the tree *becomes imbalanced* due to any operation we *use rotation* operations to make the tree balanced.
- ❖ Rotation operations are used to *make the tree balanced*.
- ❖ Rotation is the process of moving nodes either to left or to right to make the tree balanced.

There are *four rotations* and they are classified into *two types*.



1. Rotate Left:

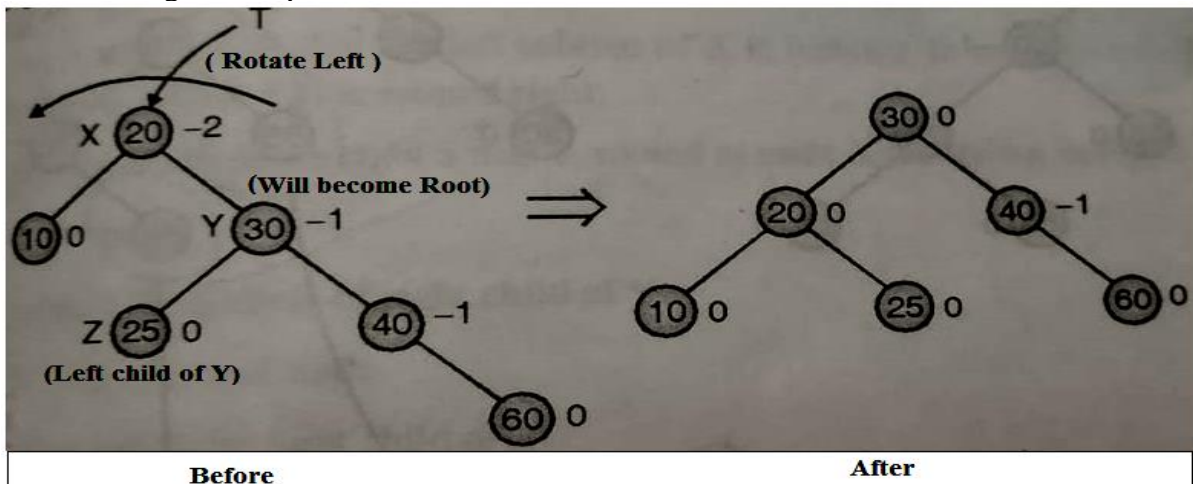
- ❖ X is the Root node.
- ❖ Y is the Right Child of X.

Rules:

1. Y will become the Root.
2. The node X will become the left child of Y.
3. Left child of Y will become Right child of X

Algorithm:

```
temp = Y->left
Root = Y
Y->left = X
X->right = temp
```



2. Rotate Right:

- ❖ X is the Root node.
- ❖ Y is the Left Child of X.

Rules:

1. Y will become the Root.

U18PCCS301 - DATA STRUCTURES & ALGORITHM

Unit – III TREES

2. The node X will become the left child of Y.
3. Left child of Y will become Right child of X

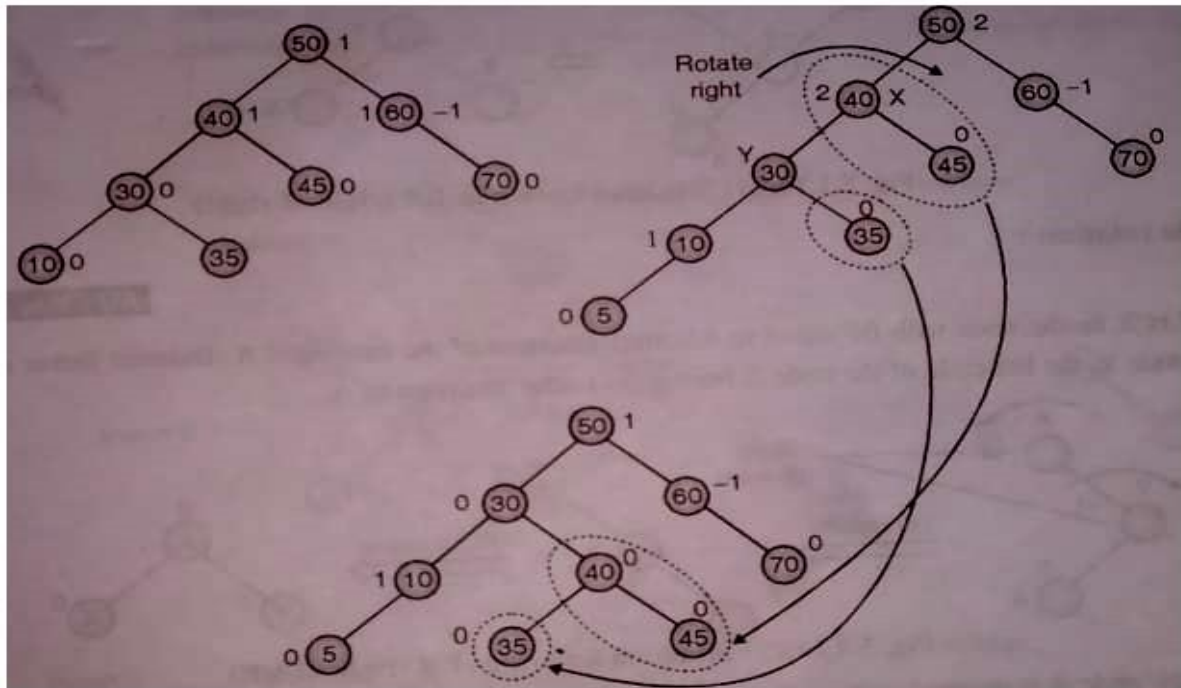
Algorithm:

temp = Y->right

Root = Y

Y->right = X

X->left = temp

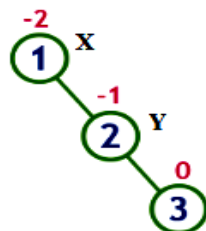


Single Rotation:

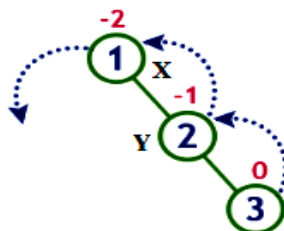
The node's BF is imbalanced when we insert a new node at **Left Subtree of Left Subtree (LL)** or **Right Subtree of Right Subtree (RR)**.

1. Single Left Rotation (LL Rotation)

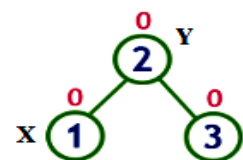
insert 1, 2 and 3



Tree is imbalanced



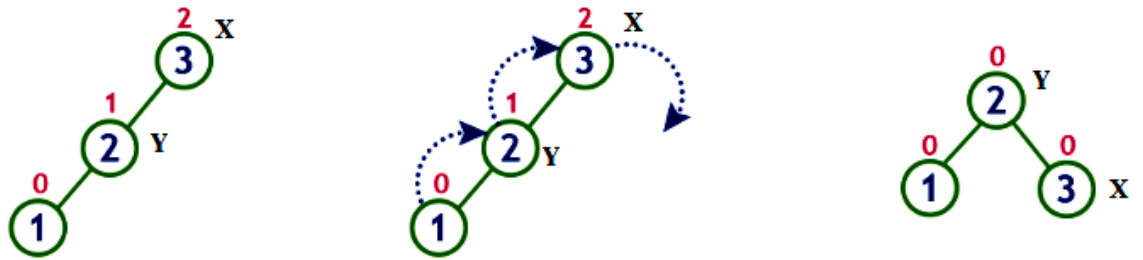
To make balanced we use LL Rotation which moves nodes one position to left



After LL Rotation Tree is Balanced

2. Single Right Rotation (RR Rotation)

insert 3, 2 and 1



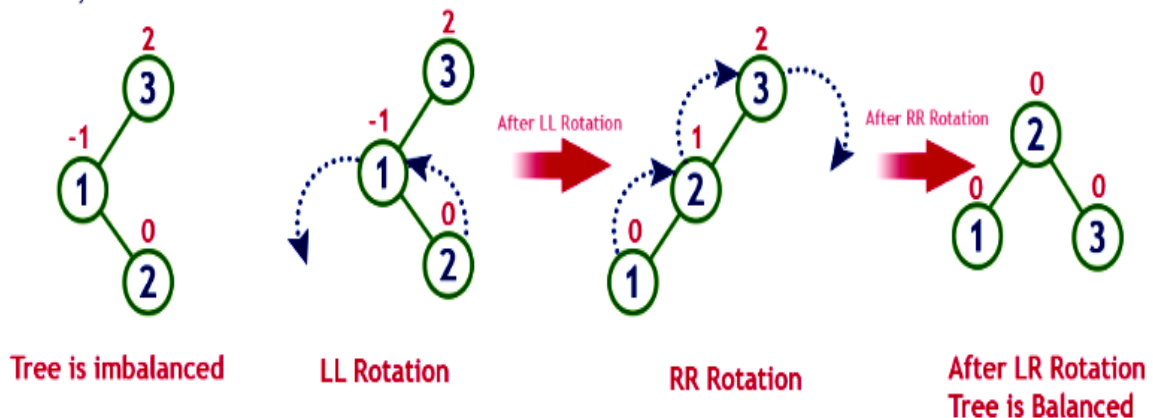
Double Rotation:

The node's BF is imbalanced when we insert a new node at **Left Subtree of Right Subtree (LR)** or **Right Subtree of Left Subtree (RL)**.

3. Left Right Rotation (LR Rotation)

- The LR Rotation is a sequence of single left rotation followed by a single right rotation.
- In LR Rotation, at first, every node moves one position to the left and one position to right from the current position.

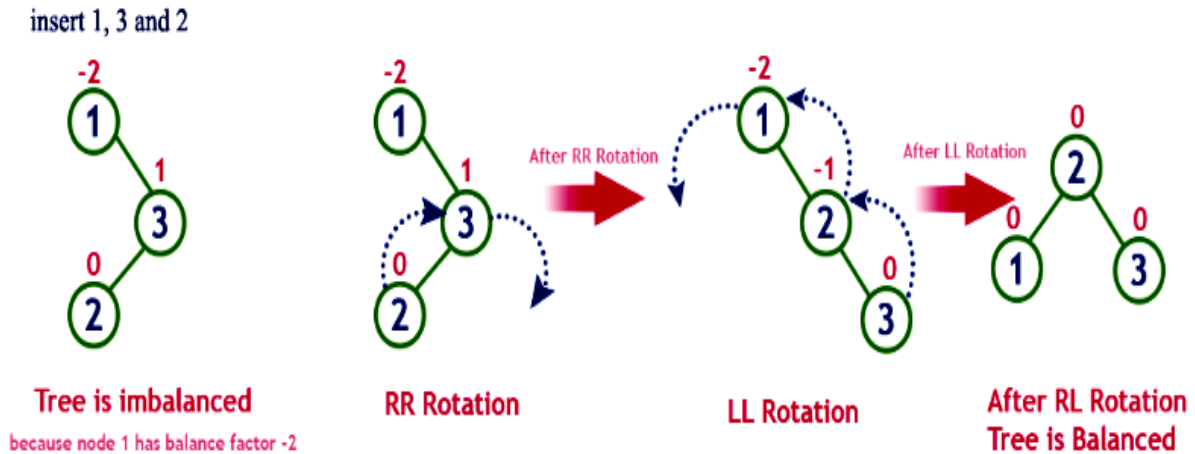
insert 3, 1 and 2



4. Right Left Rotation (RL Rotation)

- The RL Rotation is sequence of single right rotation followed by single left rotation.
- In RL Rotation, at first every node moves one position to right and one position to left from the current position.

U18PCCS301 - DATA STRUCTURES & ALGORITHM
Unit – III TREES



Insertion Operation in AVL Tree:

- ❖ In an AVL tree, the insertion operation is performed with $O(\log n)$ time complexity.
- ❖ In AVL Tree, a new node is always inserted as a leaf node.
- ❖ The insertion operation is performed as follows...

Step 1 - Insert the new element into the tree using Binary Search Tree insertion logic.

Step 2 - After insertion, check the Balance Factor of every node.

Step 3 - If the Balance Factor of every node is 0 or 1 or -1 then go for next operation.

Step 4 - If the Balance Factor of any node is other than 0 or 1 or -1 then that tree is said to be imbalanced. In this case, perform suitable Rotation to make it balanced and go for next operation.

Example: Construct an AVL Tree by inserting numbers from 1 to 8.

insert 1

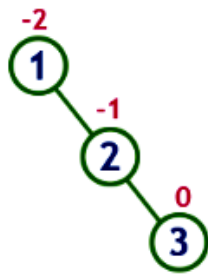


insert 2

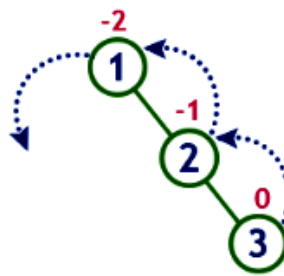


U18PCCS301 - DATA STRUCTURES & ALGORITHM
Unit – III TREES

insert 3

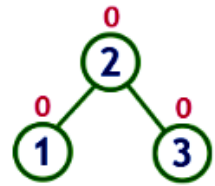


Tree is imbalanced



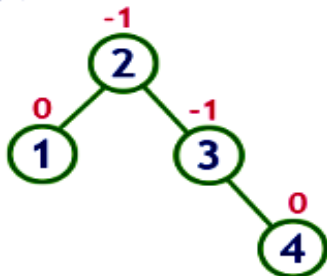
LL Rotation

After LL Rotation



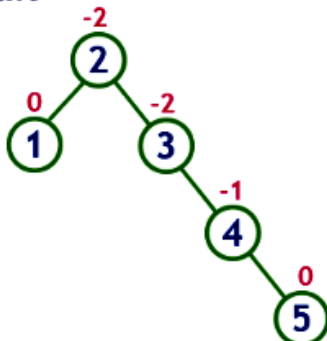
Tree is balanced

insert 4

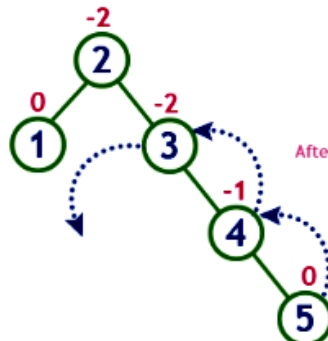


Tree is balanced

insert 5

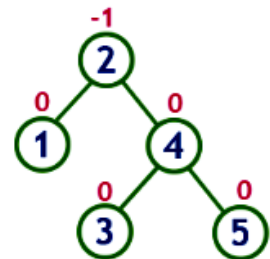


Tree is imbalanced



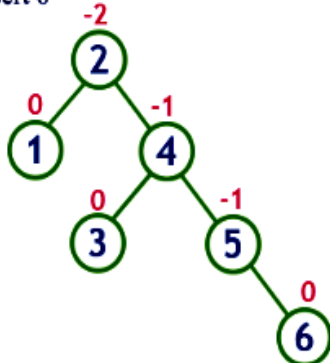
LL Rotation at 3

After LL Rotation at 3

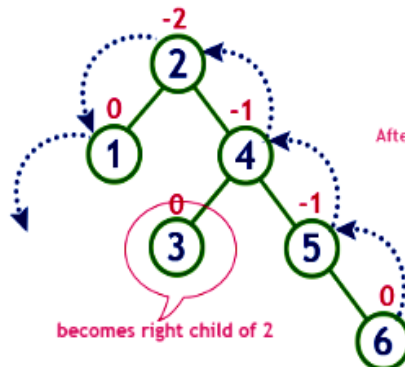


Tree is balanced

insert 6

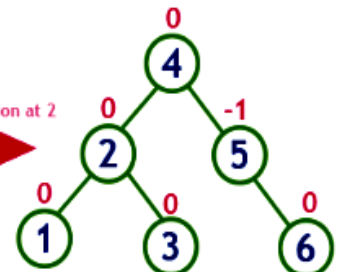


Tree is imbalanced



LL Rotation at 2

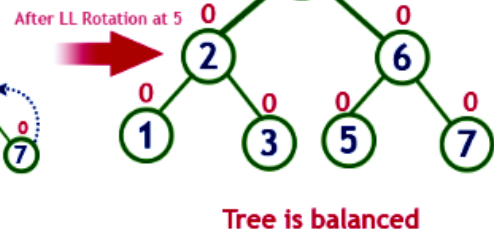
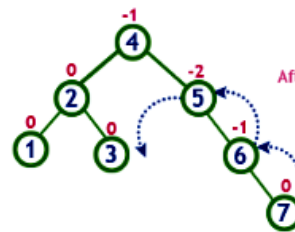
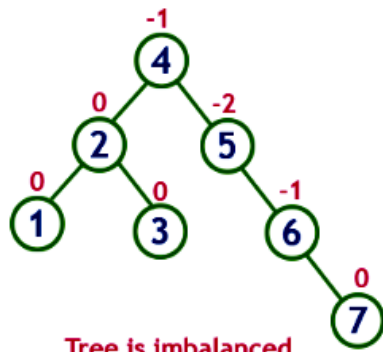
After LL Rotation at 2



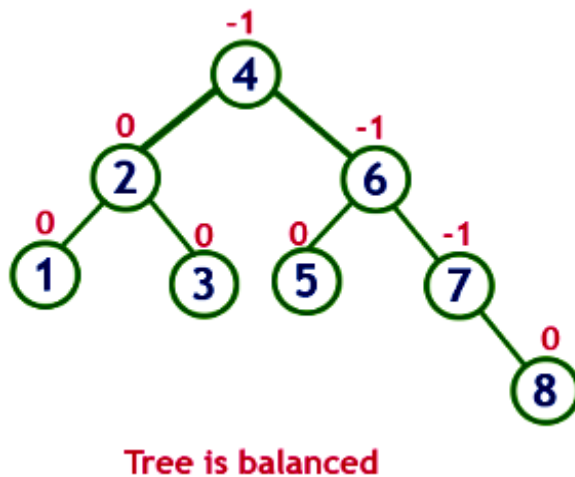
Tree is balanced

U18PCCS301 - DATA STRUCTURES & ALGORITHM
Unit – III TREES

insert 7



insert 8



Advantages of AVL Tree:

1. Since AVL trees are always close to balanced, search is $O(\log n)$.
2. Insertion and deletion are also $O(\log n)$.
3. For both insertion and deletion, height re-balancing adds no more than a constant factor to the runtime.

B-TREE:

- ❖ In search trees like binary search tree, AVL Tree, Red-Black tree, etc., every node contains only one value (key) and a maximum of two children.
- ❖ But there is a special type of search tree called B-Tree in which a **node contains more than one value** (key) and **more than two children**.
- ❖ B-Tree was developed in the year 1972 by Bayer and McCreight with the name Height Balanced ***m-way Search Tree***. Later it was named as B-Tree.
- ❖ B-trees are primarily meant for secondary storage, where as binary trees like AVL tree is stored in primary memory.

U18PCCS301 - DATA STRUCTURES & ALGORITHM
Unit – III TREES

Definition:

B-Tree is a self-balanced search tree in which every node contains multiple keys and has more than two children.

B-Tree of Order m has the following properties:

Property #1 - All leaf nodes must be at same level.

Property #2 - All nodes except root must have at least $\lceil m/2 \rceil - 1$ keys and maximum of $m-1$ keys.

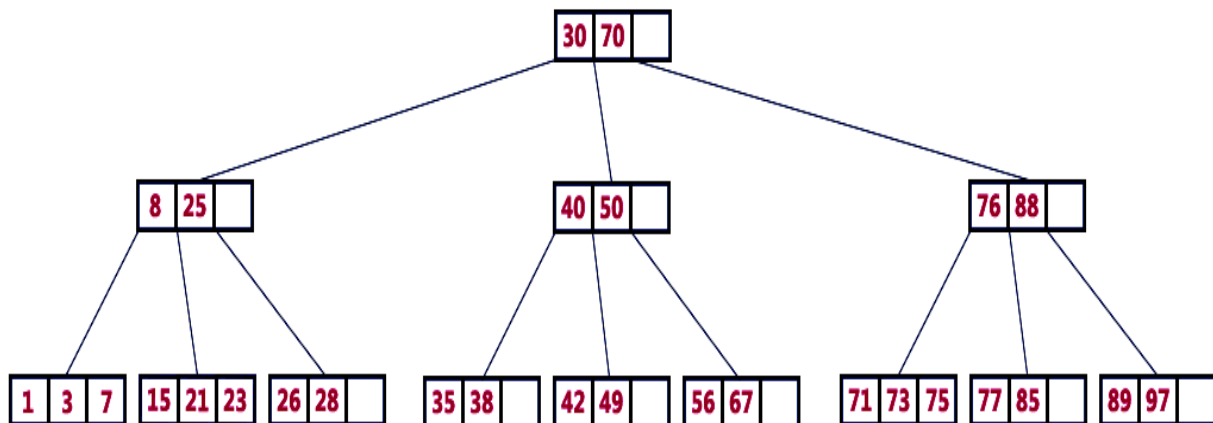
Property #3 - All non-leaf nodes except root (i.e. all internal nodes) must have at least $m/2$ children.

Property #4 - If the root node is a non-leaf node, then it must have at least 2 children.

Property #5 - A non-leaf node with $n-1$ keys must have n number of children.

Property #6 - All the key values in a node must be in *Ascending Order*.

B-Tree of Order 4



Operations on a B-Tree

The following operations are performed on a B-Tree...

- ❖ Search
- ❖ Insertion
- ❖ Deletion

Insertion Operation in B-Tree

- ❖ In a B-Tree, a new element must be added only at the leaf node.
- ❖ That means, the new key value is always attached to the leaf node only.
- ❖ The insertion operation is performed as follows...

Step 1 - Check whether tree is Empty.

U18PCCS301 - DATA STRUCTURES & ALGORITHM
Unit – III TREES

Step 2 - If tree is *Empty*, then create a new node with new key value and insert it into the tree as a root node.

Step 3 - If tree is *Not Empty*, then find the suitable leaf node to which the new key value is added using Binary Search Tree logic.

Step 4 - If that *leaf node has empty* position, add the new key value to that leaf node in ascending order of key value within the node.

Step 5 - If that *leaf node is already full*, split that leaf node by sending middle value to its parent node. Repeat the same until the sending value is fixed into a node.

Step 6 - If the splitting is performed at root node then the middle value becomes new root node for the tree and the height of the tree is increased by one.

Example: Construct a B-Tree of Order 3 by inserting numbers from 1 to 7.

Insert (1): Since '1' is the first element into the tree that is inserted as a leaf node as well as root node.

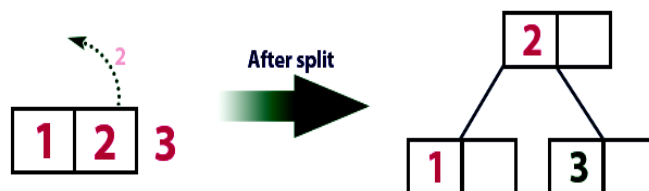


Insert (2): The leaf node has an empty position, so element '2' can be inserted at that empty position.



Insert (3):

1. The element '3' is added to existing leaf node. This leaf node does not have empty position.
2. So, we split the node by sending middle value (2) to its parent node. But here, this node does have parent node.
3. So, this middle value (2) becomes a root node for this tree.

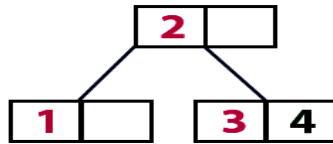


Insert (4):

1. Element '4' is larger than 2. So, move to right of '2'.

U18PCCS301 - DATA STRUCTURES & ALGORITHM
Unit – III TREES

2. Element 4 is larger than '3', and we have empty position.
3. So, element '4' is inserted at that empty position.



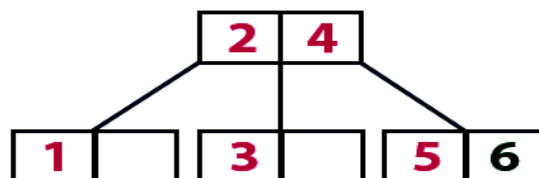
Insert (5):

1. Element '5' is larger than 2, so we moved to right of '2'.
2. Element '5' is larger than 3 and 4, But there is no empty position.
3. So, we split that node by sending the middle value (4) to its parent node.
4. The new element is added to leaf node.



Insert (6):

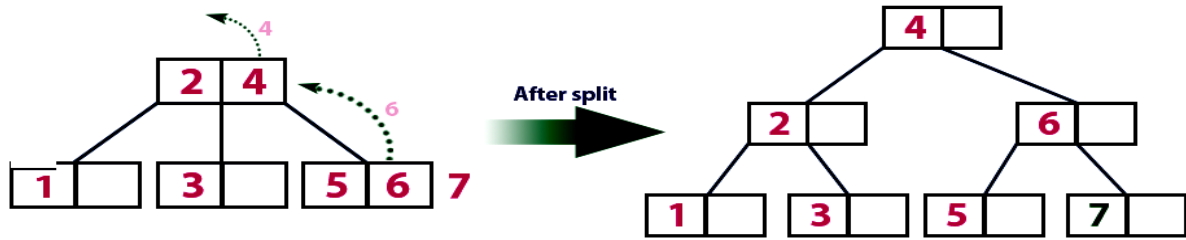
1. An element '6' is larger than 2 and 4, so we moved to right of '4'.
2. We reach leaf node and it has value '5' and it has empty position.
3. Then, a new element '6' is inserted at that empty position.



Insert (7):

1. An element '7' is larger than 2 and 4, so we moved to right of 4.
2. We reach leaf node which has 5 and 6 and it has no empty position.
3. So, we split that node by sending the middle value (6) to its parent node(2&4). But its parent node is already full.
4. So, we again split the parent (2&4) by sending the middle value(4) to its parent node.

U18PCCS301 - DATA STRUCTURES & ALGORITHM
Unit – III TREES



Advantages of B-tree:

1. Keeps keys in sorted order for sequential traversing.
2. Uses a hierarchical index to minimize the number of disk reads.
3. Uses partially full blocks to speed insertions and deletions.
4. Keeps the index balanced with a recursive algorithm.

SPLAY TREE:

1. Splay tree is another variant of a binary search tree. In a splay tree, recently accessed element is placed at the root of the tree.
2. **Definition:** Splay Tree is a self - adjusted Binary Search Tree in which every operation on element rearranges the tree so that the element is placed at the root position of the tree.
3. Splaying an element is the process of bringing it to the root position by performing suitable rotation operations.
4. In a splay tree, every operation is performed at the root of the tree. All the operations in splay tree are involved with a common operation called "*Splaying*".

❖ In a splay tree, splaying an element rearranges all the elements in the tree so that splayed element is placed at the root of the tree.

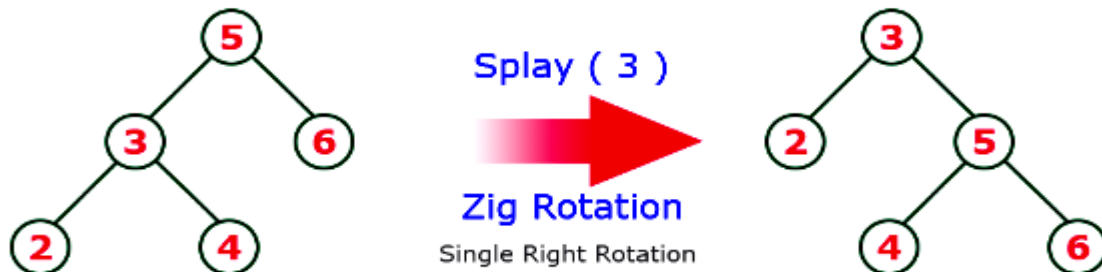
❖ In splay tree, to splay any element we use the following rotation operations...

❖ **Rotations in Splay Tree**

1. Zig Rotation
2. Zag Rotation
3. Zig - Zig Rotation
4. Zag - Zag Rotation
5. Zig - Zag Rotation
6. Zag - Zig Rotation

Zig Rotation:

- ❖ The Zig Rotation in splay tree is similar to the single right rotation in AVL Tree rotations.
- ❖ In zig rotation, every node moves one position to the right from its current position.



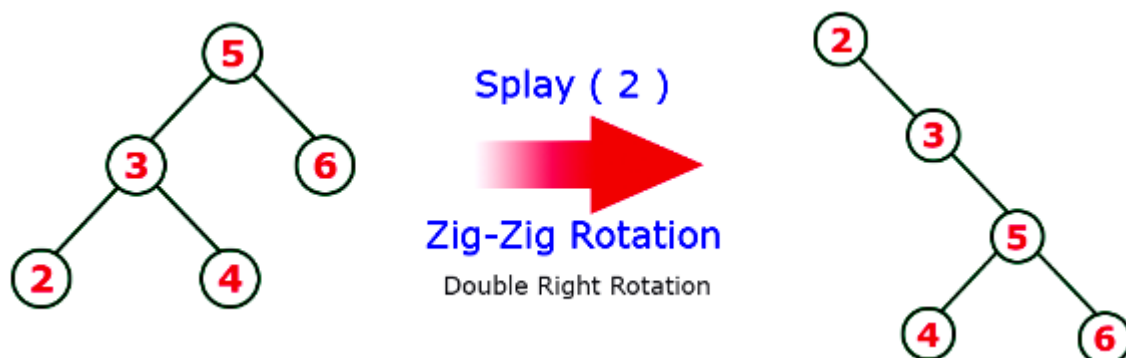
Zag Rotation:

- ❖ The Zag Rotation in splay tree is similar to the single left rotation in AVL Tree rotations.
- ❖ In zag rotation, every node moves one position to the left from its current position.



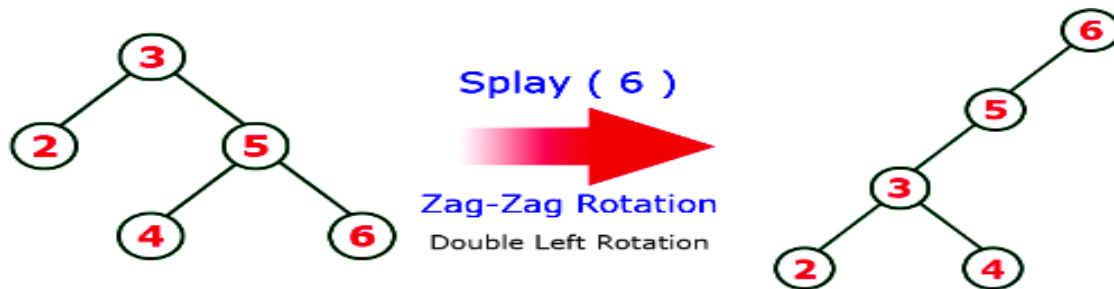
Zig-Zig Rotation:

- ❖ The Zig-Zig Rotation in splay tree is a double zig rotation.
- ❖ In zig-zig rotation, every node moves two positions to the right from its current position.



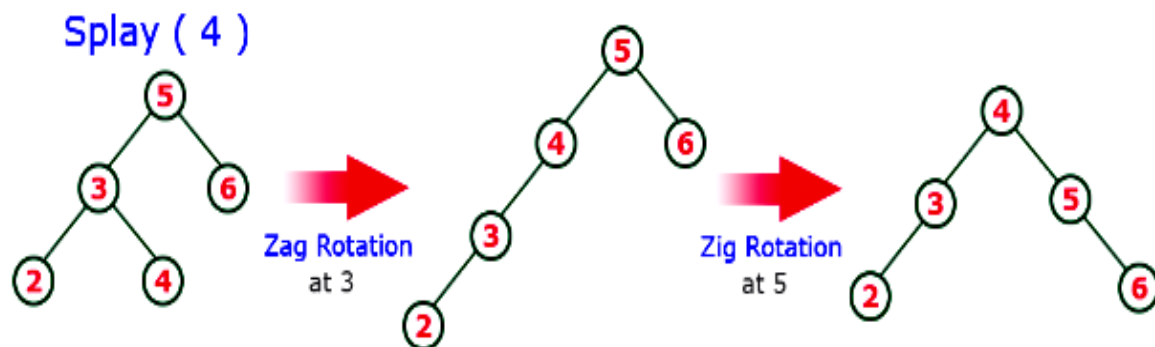
Zag-Zag Rotation:

- ❖ The Zag-Zag Rotation in splay tree is a double zag rotation.
- ❖ In zag-zag rotation, every node moves two positions to the left from its current position.



Zag-Zig Rotation:

- ❖ The Zag-Zig Rotation in splay tree is a sequence of zag rotation followed by zig rotation.
- ❖ In zag-zig rotation, every node moves one position to the left followed by one position to the right from its current position.



Insertion Operation in Splay Tree:

The insertion operation in Splay tree is performed using following steps...

Step 1 - Check whether tree is Empty.

Step 2 - If tree is Empty then insert the newNode as Root node and exit from the operation.

Step 3 - If tree is not Empty then insert the newNode as leaf node using Binary Search tree insertion logic.

Step 4 - After insertion, Splay the newNode

Deletion Operation in Splay Tree:

1. The deletion operation in splay tree is similar to deletion operation in Binary Search Tree.
2. But before deleting the element, we first need to splay that element and then delete it from the root position.

U18PCCS301 - DATA STRUCTURES & ALGORITHM
Unit – III TREES

3. Finally join the remaining tree using binary search tree logic.

Comparison of Search Trees:

- ❖ The comparison of search trees is performed based on the Time complexity of search, insertion and deletion operations in search trees.
- ❖ The following table provides the Time complexities of search trees. These Time complexities are defined for 'n' number of elements.

Search Tree	Average Case			Worst Case		
	Insert	Delete	Search	Insert	Delete	Search
BST	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$
AVL Tree	$O(\log_2 n)$	$O(\log_2 n)$	$O(\log_2 n)$	$O(\log_2 n)$	$O(\log_2 n)$	$O(\log_2 n)$
B - Tree	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
Red - Black Tree	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$
Splay Tree	$O(\log_2 n)$	$O(\log_2 n)$	$O(\log_2 n)$	$O(\log_2 n)$	$O(\log_2 n)$	$O(\log_2 n)$

2 Mark Question and answers

1) Define tree.

A tree is a finite set of one or more nodes such that there is a specially designated node called the Root, and zero or more non empty sub trees T_1, T_2, \dots, T_k , each of whose roots are connected by a directed edge from Root R.

2) Define path of tree.

A path from node n_1 to n_k is defined as a sequence of nodes $n_1, n_2, n_3, \dots, n_k$ such that n_i is the parent of n_{i+1} . There is exactly only one path from each node to root.

3) Define Length and degree of tree.

U18PCCS301 - DATA STRUCTURES & ALGORITHM
Unit – III TREES

Length: The length is defined as the number of edges on the path.

Degree: The number of subtrees of a node is called its degree.

4) Define level and Depth of a tree.

Level: The level of a node is defined by initially letting the root be at level Zero; if a node is at level L then its children are at level L + 1.

Depth: For any node n, the depth of n is the length of the unique path from root to n. The depth of the root is zero.

5) Define height of a tree.

For any node n, the height of the node n is the length of the longest path from n to the leaf.

6) Define Binary tree.

Binary Tree is a tree in which no node can have more than two children. Maximum number of nodes at level i of a binary tree is 2^{i-1} .

7) Write short notes on types of binary tree?

Types of Binary Tress

Rooted Binary tree: - In which every node has atmost two children.

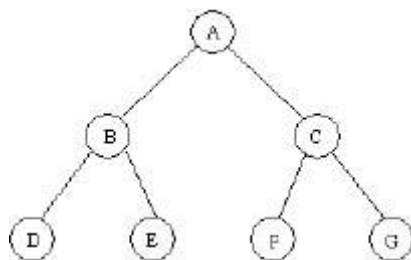
Full Binary Tree:-In which every node has zero or two children.

Perfect Binary Tree:- In which all the leaves are at same depth.

Complete Binary Tree:- In which all the leaves are at same depth n or n-1 for some n.

8) Write the linear representation of Binary tree?

The elements are represented using arrays. For any element in position i, the left child is in position $2i$, the right child is in position $(2i + 1)$, and the parent is in position $(i/2)$



A	B	C	D	E	F	G
---	---	---	---	---	---	---

1 2 3 4 5 6 7

E.g $i=0$, For node A, Left child at $2i \Rightarrow 2 \times 1 + 1 = 1 \Rightarrow B$, Right child at $2i+2 \Rightarrow (2 \times 1) + 2 = 2 \Rightarrow C$

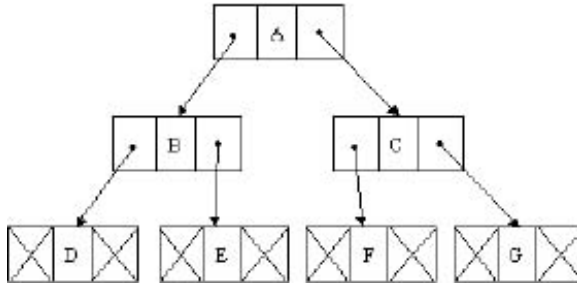
9) Write the linked representation of Binary tree?

The elements are represented using pointers. Each node in linked representation has three fields, namely,

U18PCCS301 - DATA STRUCTURES & ALGORITHM
Unit – III TREES

- * Pointer to the left subtree
- * Data field
- * Pointer to the right subtree

In leaf nodes, both the pointer fields are assigned as NULL.



10) What is meant by traversing?

Traversing a tree means processing it in such a way, that each node is visited only once.

11) What are the different types of traversing?

The different types of traversing are

- a. Pre-order traversal-yields prefix form of expression.
- b. In-order traversal-yields infix form of expression.
- c. Post-order traversal-yields postfix form of expression.

12) Define pre-order traversal?

Pre-order traversal entails the following steps;

- a. Visit the root node
- b. Traverse the left subtree
- c. Traverse the right subtree

13) Define post-order traversal?

Post order traversal entails the following steps;

- a. Traverse the left subtree
- b. Traverse the right subtree
- c. Visit the root node

14) Define in-order traversal?

In-order traversal entails the following steps;

- a. Traverse the left subtree
- b. Visit the root node
- c. Traverse the right subtree

U18PCCS301 - DATA STRUCTURES & ALGORITHM
Unit – III TREES

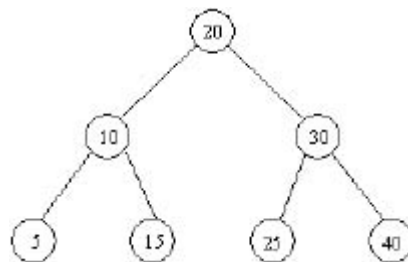
15) Define BST?

Binary search tree is a binary tree in which for every node X in the tree, the values of all the keys in its left subtree are smaller than the key value in X, and the values of all the keys in its right subtree are larger than the key value in X.

16) Define Expression Tree?

Expression Tree is a binary tree in which the leaf nodes are operands and the interior nodes are operators. Like binary tree, expression tree can also be traversed by Inorder, Preorder and Postorder traversal.

17) Write down the Inorder, Preorder, Postorder for the following tree.



Inorder: 5 10 15 20 25 30 40

Preorder: 20 10 5 15 30 25 40

Postorder: 5 15 10 25 40 30 20

18) Define AVL tree.

- i. An AVL tree is a height balanced Binary Search tree.
- ii. AVL tree is a binary search tree in which the height of two siblings is not permitted to differ more than one.

| Height of the Left subtree – Height of Right subtree | ≤ 1

19) Define Splay tree.

Splay Tree is a self - adjusted Binary Search Tree in which every operation on element rearranges the tree so that the element is placed at the root position of the tree.

20) Define B tree.

B-Tree is a self-balanced search tree in which every node contains multiple keys and has more than two children.

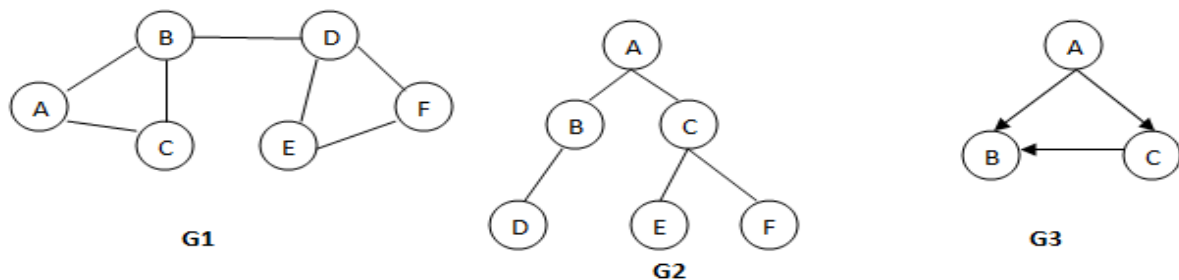
Unit – IV GRAPH**Syllabus:**

Graph- Basic Terminologies – Representations- Graph types - Graph search methods - Graph traversal algorithms - complexity analysis– Applications of Graph.

GRAPH:

Definition: A graph G is a set of vertices (V) and set of edges (E).

The set V is finite, nonempty set of vertices. The set E is a set of pairs of vertices representing edges. $G = (V, E)$.



The set representation for each of these graphs are given by

$$V(G1) = \{A, B, C, D, E, F\}$$

$$V(G2) = \{A, B, C, D, E, F\}$$

$$V(G3) = \{A, B, C\}$$

$$E(G1) = \{(A,B), (A,C), (B,C), (B,D), (D,E), (D,F), (E,F)\}$$

$$E(G2) = \{(A,B), (A,C), (B,D), (C,E), (C,F)\}$$

$$E(G3) = \{(A,B), (A,C), (C,B)\}$$

Applications

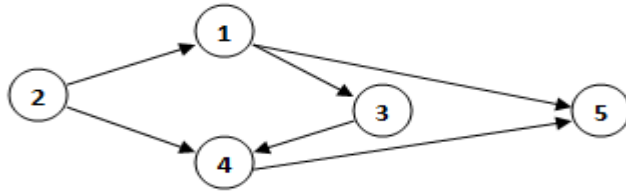
Graphs have many important applications as shown below:

- ✓ Analysis of electrical circuit
- ✓ Finding shortest routes
- ✓ Project planning
- ✓ To represent highway structures
- ✓ Communication lines
- ✓ Railway lines

BASIC TERMINOLOGIES:**1. Directed Graph (or) Digraph:**

A graph containing ordered pair of vertices is called a directed graph.

Directed graph is a graph which consists of directed edges, where each edge in E is unidirectional. It is also referred as Digraph.



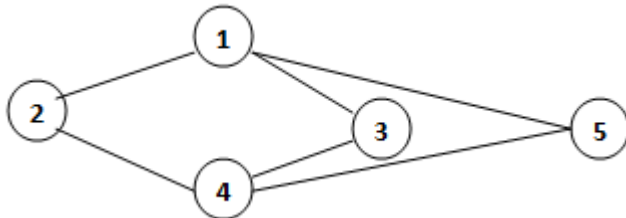
The set of vertices $V = \{1, 2, 3, 4, 5\}$

The set of Edges $E = \{(1,3), (1,5), (2,1), (2,4), (3,4), (4,5)\}$

2. Undirected Graph:

An undirected graph is a graph, which consists of undirected edges.

A graph containing unordered pair of vertices is called an undirected graph.



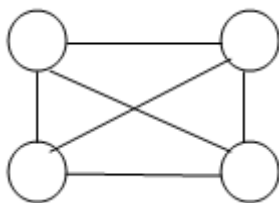
The set of Vertices $V = \{1, 2, 3, 4, 5\}$

The set of Edges $E = \{(1,2), (1,3), (1,5), (2,4), (3,4), (4,5)\}$

3. Complete Graph:

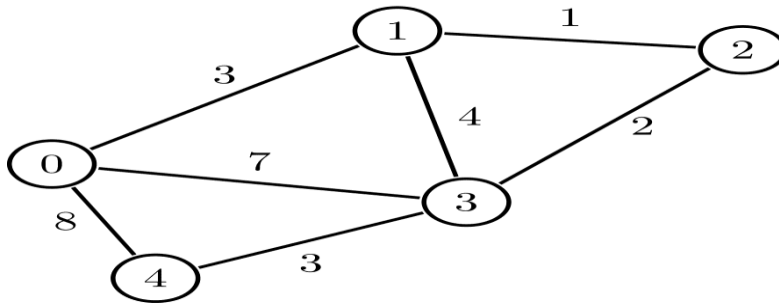
A complete graph is a graph in which there is an edge between every pair of vertices.

A complete graph with N vertices has $N(N-1)/2$ edges.



4. Weighted Graph:

A graph is said to be weighted graph if every edge in the graph is assigned a weight or value. It can be directed or undirected graph.



5. Degree of a Vertex:

The total number of edges linked to a vertex is called its degree.

Indegree: The indegree of a vertex is the total number of edges coming to that node.

Outdegree: The outdegree of a node is the total number of edges going out from that node.

Source: A vertex, which has only outgoing edges and no incoming edges, is called a source.

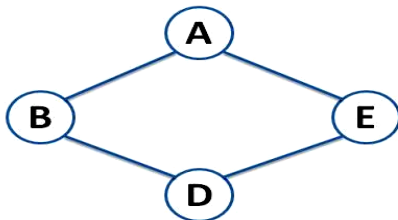
Sink: A vertex having only incoming edges and no outgoing edges is called sink.

Pendant: When indegree of a vertex is one and outdegree is zero then such a vertex is called Pendant vertex.

Isolated: When the degree of a vertex is zero, it is an isolated vertex.

6. Connected Graph

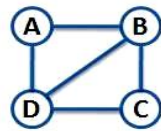
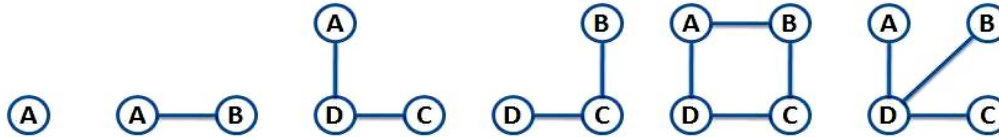
A graph is said to be connected if there exists a path between every pair of vertices V_i and V_j .



Connected Graph

7. Subgraph:

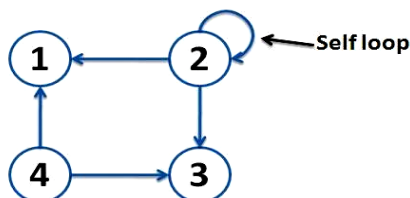
A subgraph of G is a graph G_1 such that $V(G_1)$ is a subset of $V(G)$ and $E(G_1)$ is a subset of $E(G)$.

**Graph (G)****Subgraphs of Graph (G)****8. Path:**

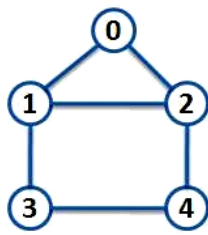
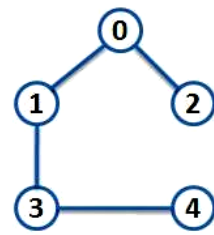
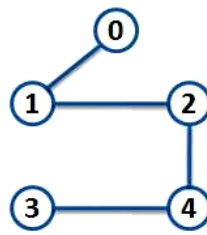
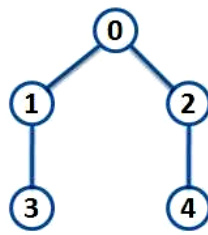
- ✓ A path from vertices V_0 to V_1 is a sequence of vertices $V_0, V_1, V_2, \dots, V_{n-1}, V_n$.
- ✓ Here V_0 is adjacent to V_1 and V_{n-1} is adjacent to V_n .
- ✓ The length of a path is the number of edges on the path.
- ✓ A path with n vertices has a length of $n-1$.

9. Loops

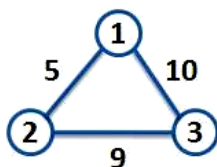
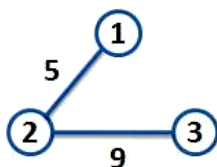
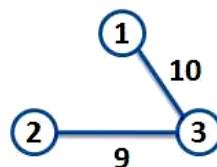
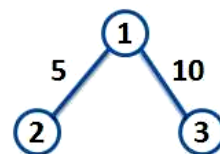
An edge of the form (V, V) is known as self edge or self loop.

**Graph with self loop****10. Spanning Tree:**

- ✓ A spanning tree of a graph $G = (V, E)$ is a connected subgraph of G having all vertices of G and no cycles in it.
- ✓ If the graph G is not connected then there is no spanning tree of G .
- ✓ A graph may have multiple spanning trees.

**Connected Graph (G)****Spanning Trees of Graph (G)****11. Minimal Spanning Tree:**

- ✓ The cost of a graph is the sum of the costs of the edges in the weighted graph.
- ✓ A spanning tree of a group $G = (V, E)$ is called a minimal cost spanning tree or simply the minimal spanning tree of G if its cost is minimum.

**G****T₁****T₂****T₃****Minimal Spanning Tree**

G → A sample weighted graph

T₁ → A spanning tree of G with cost $5 + 9 = 14$

T₂ → A spanning tree of G with cost $10 + 9 = 19$

T₃ → A spanning tree of G with cost $5 + 10 = 15$

Therefore T₁ with cost 14 is the minimal cost spanning tree of the graph G.

12. Cycle:

A cycle in a graph is a path in which first and last vertex are same. A graph which has cycle is referred to as cyclic graph.

Graph Representation:

Graph can be represented by Adjacency Matrix and Adjacency list.

Adjacency Matrix

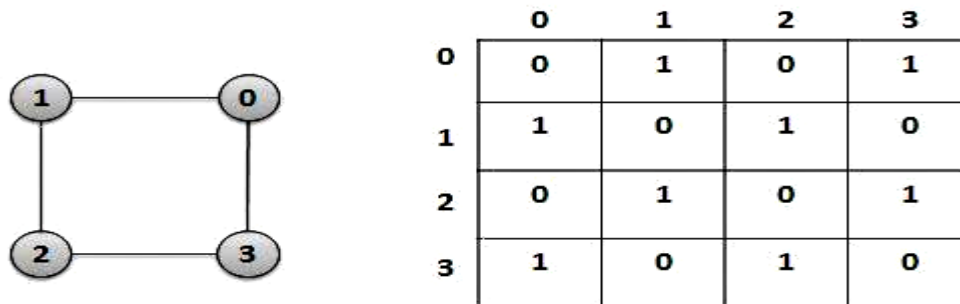
One simple way to represent a graph is Adjacency Matrix.

The adjacency Matrix A for a graph $G = (V, E)$ with n vertices is an $n \times n$ matrix, such that

$A[i][j] = 1$, if there is an edge V_i to V_j

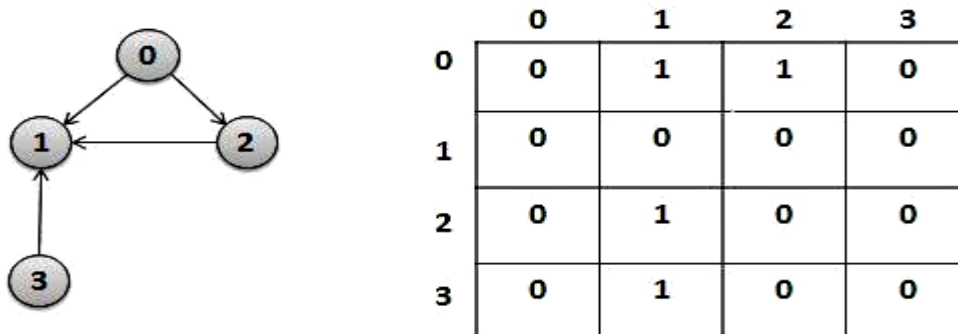
$A[i][j] = 0$, if there is no edge.

Adjacency Matrix for Undirected Graph



Adjacency Matrix Representation of Undirected Graph

Adjacency Matrix for Directed Graph



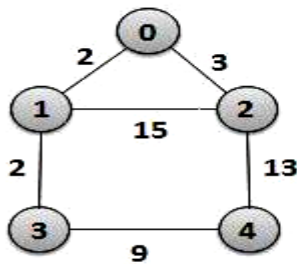
Adjacency Matrix Representation of Directed Graph

Adjacency Matrix for Weighted Graph:

For weighted graph, the matrix $adj[][]$ is represented as:

If there is an edge between vertices i and j then $adj[i][j] = \text{weight of the edge } (i, j)$ otherwise $adj[i][j] = 0$.

An example of representation of weighted graph is given below:

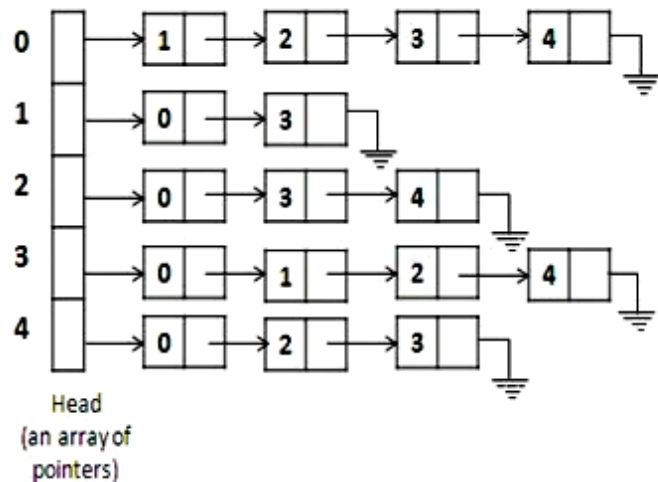
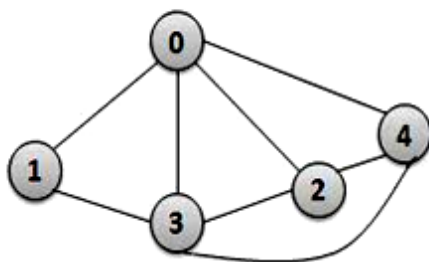


	0	1	2	3	4
0	0	2	3	0	0
1	2	0	15	2	0
2	3	15	0	0	13
3	0	2	0	0	9
4	0	0	13	9	0

Adjacency Matrix Representation of Weighted Graph

Adjacency List Representation:

- ✓ A graph can also be represented using a linked list. For each vertex, a list of adjacent vertices is maintained using a linked list.
- ✓ It creates a separate linked list for each vertex V_i in the graph $G = (V, E)$.
- ✓ Adjacency list of a graph with n nodes can be represented by an array of pointers.
- ✓ Each pointer points to a linked list of the corresponding vertex.



Graph Traversal:

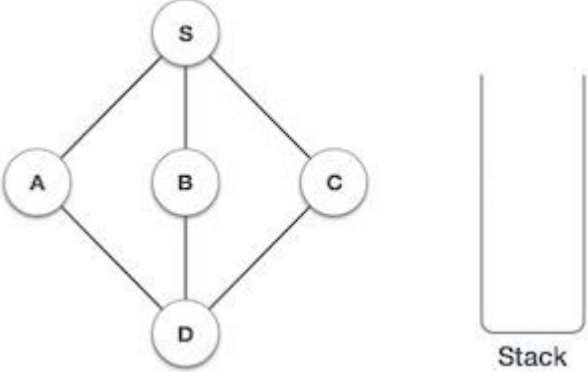
- ✓ Most of graph problems involve traversal of a graph. Traversal of a graph means visiting each node and visiting exactly once.
- ✓ There are two types of traversal in graphs i.e. **Depth First Search** (DFS) and **Breadth First Search** (BFS).

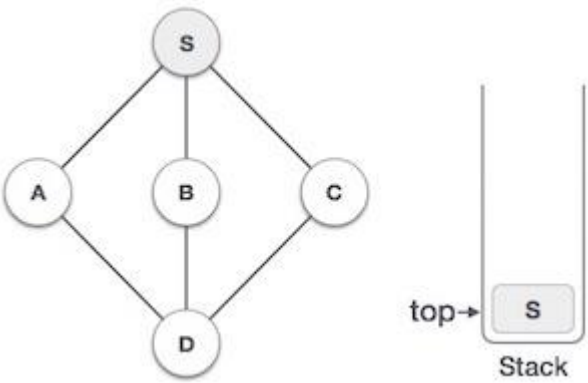
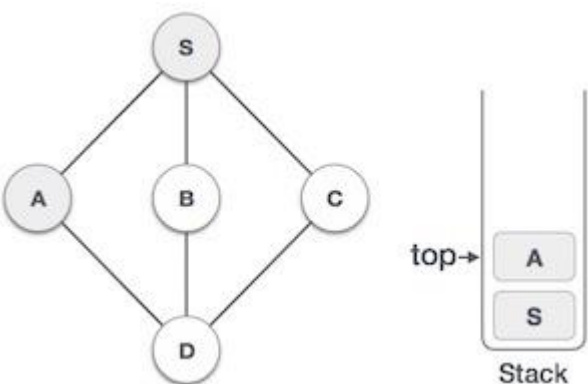
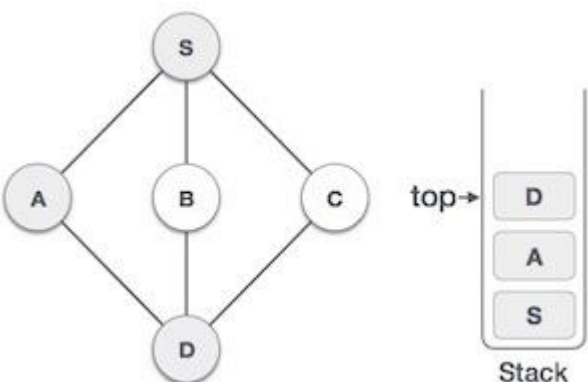
DEPTH FIRST SEARCH:

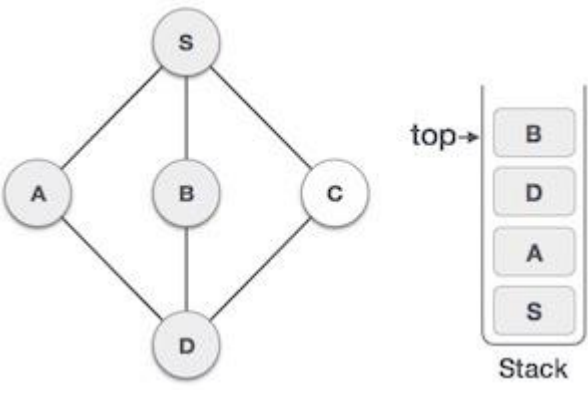
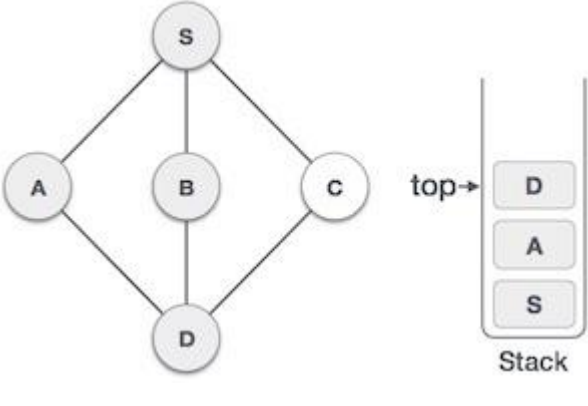
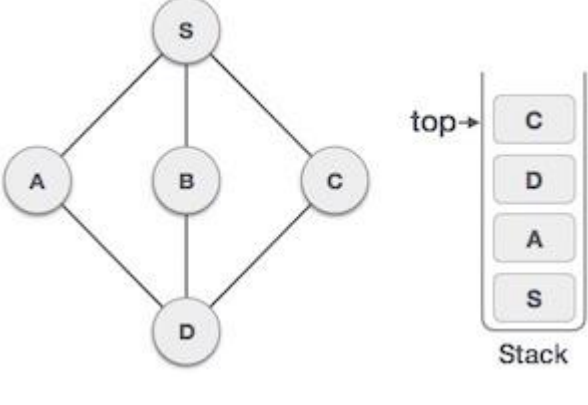
Depth First Search (DFS) algorithm traverses a graph in a depthward motion and uses a stack to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

1. It is like tree. Traversal can start from any vertex, say V_i .
2. V_i is visited and then all vertices adjacent to V_i are traversed recursively using DFS.
3. Since, a graph can have cycles. We must avoid revisiting a node.
4. To do this, when we visit a vertex V , we mark it visited.
5. A node that has already been marked as visited should not be selected for traversal.

- ✓ **Rule 1** – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Push it in a stack.
- ✓ **Rule 2** – If no adjacent vertex is found, pop up a vertex from the stack. (It will pop up all the vertices from the stack, which do not have adjacent vertices.)
- ✓ **Rule 3** – Repeat Rule 1 and Rule 2 until the stack is empty.

Step	Traversal	Description
1		Initialize the stack.

2		<p>Mark S as visited and put it onto the stack.</p> <p>Explore any unvisited adjacent node from S.</p> <p>We have three nodes and we can pick any of them.</p> <p>For this example, we shall take the node in an alphabetical order.</p>
3		<p>Mark A as visited and put it onto the stack.</p> <p>Explore any unvisited adjacent node from A.</p> <p>Both S and D are adjacent to A but we are concerned for unvisited nodes only.</p>
4		<p>Visit D and mark it as visited and put onto the stack.</p> <p>Here, we have B and C nodes, which are adjacent to D and both are unvisited.</p> <p>However, we shall again choose in an alphabetical order.</p>

5		<p>We choose B, mark it as visited and put onto the stack.</p> <p>Here B does not have any unvisited adjacent node.</p> <p>So, we pop B from the stack.</p>
6		<p>We check the stack top for return to the previous node and check if it has any unvisited nodes.</p> <p>Here, we find D to be on the top of the stack.</p>
7		<p>Only unvisited adjacent node is from D is C now.</p> <p>So we visit C, mark it as visited and put it onto the stack.</p>

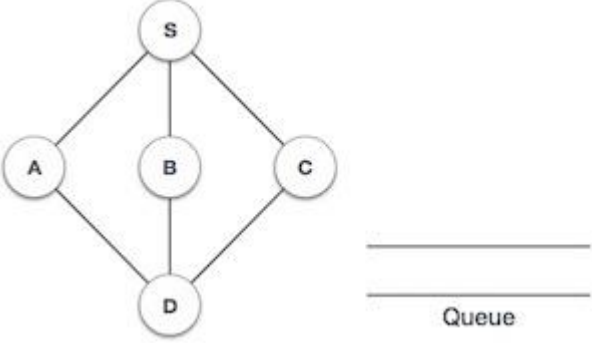
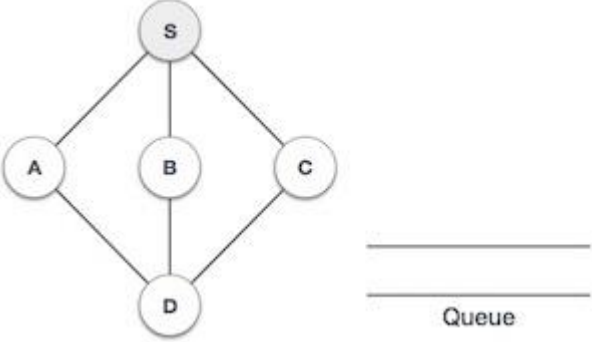
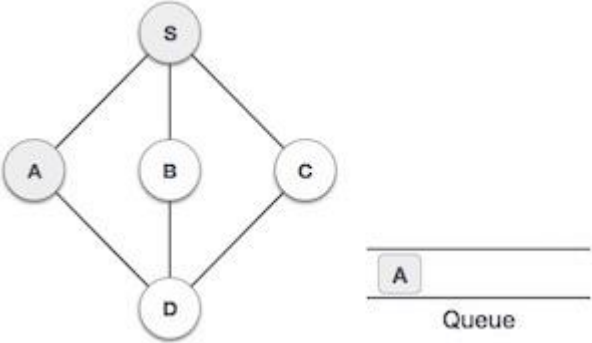
```
#include<stdio.h>
#include<conio.h>
void DFS(int);
int G[10][10],visited[10], n; //n is no of vertices and graph is sorted in array G[10][10]
void main()
{
    int i,j;
    printf("Enter number of vertices:");
    scanf("%d",&n);
    printf("\nEnter adjacency matrix of the graph:");
    for(i=0;i<n;i++)
        for(j=0;j<n;j++)
            scanf("%d",&G[i][j]);
    for(i=0;i<n;i++)
        visited[i]=0;
    DFS(0);
}
void DFS(int i)
{
    int j;
    printf("\n%d",i);
    visited[i]=1;
    for(j=0;j<n;j++)
        if(!visited[j]&&G[i][j]==1)
            DFS(j);
}
```

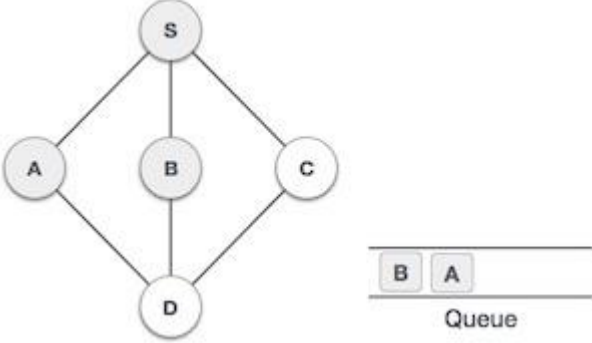
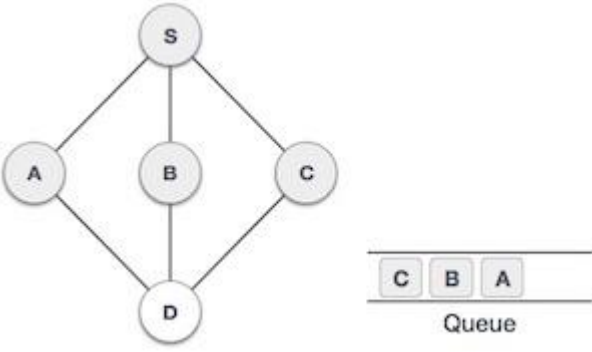
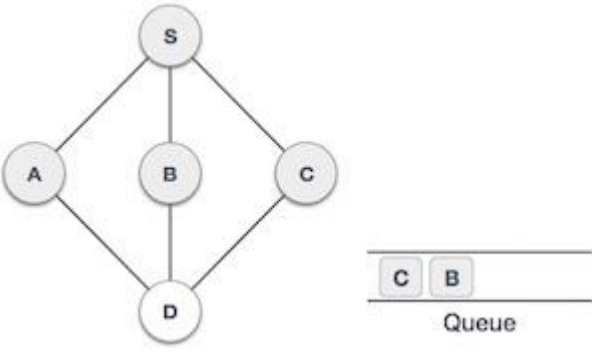
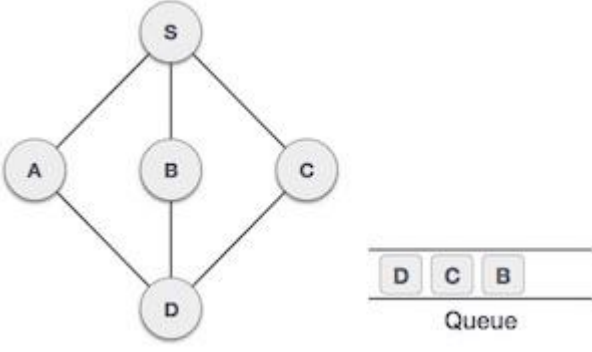
BREADTH FIRST TRAVERSAL:

Breadth First Search (BFS) algorithm traverses a graph in a breadthward motion and uses a queue to remember to get the next vertex to start a search, when a dead end occurs in any iteration.

- ✓ **Rule 1** – Visit the adjacent unvisited vertex. Mark it as visited. Display it. Insert it in a queue.
- ✓ **Rule 2** – If no adjacent vertex is found, remove the first vertex from the queue.

✓ **Rule 3** – Repeat Rule 1 and Rule 2 until the queue is empty.

Step	Traversal	Description
1		Initialize the queue.
2		We start from visiting S (starting node), and mark it as visited.
3		We then see an unvisited adjacent node from S . In this example, we have three nodes but alphabetically we choose A , mark it as visited and enqueue it.

4		<p>Next, the unvisited adjacent node from S is B. We mark it as visited and enqueue it.</p>
5		<p>Next, the unvisited adjacent node from S is C. We mark it as visited and enqueue it.</p>
6		<p>Now, S is left with no unvisited adjacent nodes. So, we dequeue and find A.</p>
7		<p>From A we have D as unvisited adjacent node. We mark it as visited and enqueue it.</p>

At this stage, we are left with no unmarked (unvisited) nodes. But as per the algorithm we keep on dequeuing in order to get all unvisited nodes. When the queue gets emptied, the program is over.

Objective and properties of different sorting algorithms: Selection Sort, Bubble Sort, Insertion Sort, Quick Sort, Merge Sort, Heap Sort; Performance and Comparison among all the methods, Hashing.

SORTING:

Sorting is a mechanism to arranging the unsorted elements in either ascending or descending order.

There are two kinds of sorting.

1. **Internal Sort** – Utilizes internal memory
2. **External Sort** – Utilizes external memory or secondary storage devices.

INSERTION SORT:

- ❖ Insertion sort algorithm arranges a list of elements in a particular order.
- ❖ In insertion sort algorithm, every iteration moves an element from unsorted portion to sorted portion until all the elements are sorted in the list.

Insertion Sort Algorithm:

Step: 1 To store unsorted elements in an array.

Step: 2 Take first element and marked it as sorted zone. Then remaining elements are marked as in unsorted zone.

Step: 3 Take first element from unsorted zone and compare it with sorted zone elements then insert it their appropriate position in sorted zone.

Step: 4 Repeat step 3 until unsorted zone will become empty.

Step: 5 Print the sorted elements.

Step: 6 Stop the sorting process

Example:

Given unsorted list of elements are: 15, 20, 10, 30, 50, 18, 5, 45

Step: 1 To store the given unsorted elements in to unsorted array.

15	20	10	30	50	18	5	45
----	----	----	----	----	----	---	----

Step: 1 Take first element (15) and mark it as in **sorted zone**.

Sorted		Unsorted					
15	20	10	30	50	18	5	45

Step: 3 Take 20 from unsorted zone and compare it with 15. And insert it at correct position.

Sorted		Unsorted					
15	20	10	30	50	18	5	45

Step: 4 Take 10 from unsorted zone and compare it with sorted zone elements. Compare 10 with 20 and it is smaller so swap. Then compare 10 with 15 it is smaller so swap. And 10 is inserted at it correct position in sorted zone.

Sorted				Unsorted			
10	15	20	30	50	18	5	45

Step: 5 Take 30 from unsorted zone and compare it with sorted zone elements. Compare 30 with 20, 15 and 10 it is greater than all so need of swap. And 30 is inserted at its correct position in sorted zone.

Sorted				Unsorted			
10	15	20	30	50	18	5	45

Step: 6 Take 50 from unsorted zone and compare it with sorted zone elements. Compare 50 with 30, 20, 15 and 10 it is greater than all so need of swap. And 50 is inserted at its correct position in sorted zone.

Sorted					Unsorted		
10	15	20	30	50	18	5	45

Step: 7 Take 18 from unsorted zone and compare it with sorted zone elements. Compare 18 with 50, 30, 20, 15 and 10. 18 is larger than 15 so move 20, 30, and 50 to one position right. And insert 18 its correct position in sorted zone.

Sorted						Unsorted	
10	15	18	20	30	50	5	45

Step: 8 Take 5 from unsorted zone and compare it with sorted zone elements. Compare 5 with 50, 30, 20, 18, 15 and 10. The element 5 is smaller than all move all the elements to one position right. And insert 5 its correct position in sorted zone.



Step: 9 Take 45 from unsorted zone and compare it with sorted zone elements. Compare 45 with 50, 30, 20, 18, 15, 10 and 5. 45 is larger than 30 so move 50 to one position right. And insert 45 its correct position in sorted zone.



Step: 9 The Unsorted zone has became empty. So we stop process. Print the sorted zone elements.



Complexity of the Insertion Sort Algorithm

Worst Case: $O(n^2)$

Best Case: $\Omega(n)$

Average Case: $\Theta(n^2)$

Program:

```
#include<stdio.h>
#include<conio.h>
void main()
{
    int size, i, j, temp, list[100];
    printf("Enter the size of the list: ");
    scanf("%d", &size);
    printf("Enter %d integer values: ", size);
    for (i = 0; i < size; i++)
        scanf("%d", &list[i]);
```

```

        for (i = 1; i < size; i++)
        {
            temp = list[i];
            j = i - 1;
            while ((temp < list[j]) && (j >= 0))
            {
                list[j + 1] = list[j];
                j = j - 1;
            }
            list[j + 1] = temp;
        }
        printf("List after Sorting is: ");
        for (i = 0; i < size; i++)
            printf(" %d", list[i]);
        getch();
    }

```

Output:

Enter the size of the list: 5

Enter 5 integer values: 63 45 76 98 80

List after sorting is: 45 63 76 80 98

BUBBLE SORT:

- ❖ Bubble sort is one of the easiest sorting method.
- ❖ In this method, each data item is compared with its neighbour.
- ❖ If it is an ascending order, then the larger element is moved to top of all the elements and the smaller elements slowly moved to the bottom position.
- ❖ It is also called as exchange sort.

Algorithm:

Step: 1 Read number of elements to be sorted.

Step: 2 Store the elements in an unsorted array.

Step: 3 Compare adjacent elements. Larger elements moved to highest index position in the array.

Step: 4 Repeat step(3) for all the elements in the array.

Step: 5 Print the sorted elements.

Step: 6 Stop the sorting process.

Example: Consider the elements 70, 30, 20, 50, 60, 10, 40.

We can store the elements in the array.

70	30	20	50	60	10	40
A[0]	A[1]	A[2]	A[3]	A[4]	A[5]	A[6]

Working procedure:

1. Largest element moves to the highest index position.
2. Second largest element bubbles up, then third highest element bubbles and so on.
3. This process will end when all the elements get sorted.

Phase: 1

70	30	20	50	60	10	40	Compare(70, 30) => swap(70, 30)
30	70	20	50	60	10	40	Compare(70, 20) => swap(70, 20)
30	20	70	50	60	10	40	Compare(70, 50) => swap(70, 50)
30	20	50	70	60	10	40	Compare(70, 60) => swap(70, 60)
30	20	50	60	70	10	40	Compare(70, 10) => swap(70, 10)
30	20	50	60	10	70	40	Compare(70, 40) => swap(70, 40)
30	20	50	60	10	40	70	Finish Phase-1

Phase: 2

30	20	50	60	10	40	70	Compare(30, 20) => swap(30, 20)
20	30	50	60	10	40	70	Compare(30, 50) => No need to swap
20	30	50	60	10	40	70	Compare(50, 60) => No need to swap
20	30	50	60	10	40	70	Compare(60, 10) => swap(60, 10)
20	30	50	10	60	40	70	Compare(60, 40) => swap(60, 40)
20	30	50	10	40	60	70	Compare(60, 70) => No need to swap

Phase: 3

20	30	50	10	40	60	70	Compare(20, 30) => No need to swap
20	30	50	10	40	60	70	Compare(30, 50) => No need to swap
20	30	50	10	40	60	70	Compare(50, 10) => swap(50, 10)
20	30	10	50	40	60	70	Compare(50, 40) => swap(50, 40)

20	30	10	40	50	60	70	Compare(50, 60) => No need to swap
20	30	10	40	50	60	70	Compare(60, 70) => No need to swap

Phase: 3

20	30	10	40	50	60	70	Compare(20, 30) => No need to swap
20	30	10	40	50	60	70	Compare(30, 10) => swap(30, 10)
20	10	30	40	50	60	70	Compare(30, 40) => No need to swap
20	10	30	40	50	60	70	Compare(40, 50) => No need to swap
20	10	30	40	50	60	70	Compare(50, 60) => No need to swap
20	10	30	40	50	60	70	Compare(60, 70) => No need to swap

Phase: 4

20	10	30	40	50	60	70	Compare(20, 10) => swap(20, 10)
10	20	30	40	50	60	70	Compare(20, 30) => No need to swap
10	20	30	40	50	60	70	Compare(30, 40) => No need to swap
10	20	30	40	50	60	70	Compare(40, 50) => No need to swap
10	20	30	40	50	60	70	Compare(50, 60) => No need to swap
10	20	30	40	50	60	70	Compare(60, 70) => No need to swap

Sorted array is:

10	20	30	40	50	60	70
----	----	----	----	----	----	----

Time complexity analysis:

$O(n^2)$ for all cases.

QUICK SORT:

Quick sort is sorting algorithm that uses divide and conquer methodology.

In this method, division is dynamically carried out.

Divide:

- ❖ Divide the array into two sub arrays by picking any key value in the array called pivot element.
- ❖ Each element in the left sub array is less than or equal to pivot element.
- ❖ Each element in the right sub tree is greater than the pivot element.

Conquer:

- ❖ Recursively divide the sub arrays until the array will contain only one element.

Combine:

- ❖ Combine all the sorted elements in a group to form a list of sorted elements.

Algorithm:

Step(1): Let take the first element as the ‘pivot’ element.

Step(2): Set low index of an array to “i”.

Set high index of an array to “j”.

Step(3): if($A[i] < \text{pivot}$) \Rightarrow then increment “i”

Step(4): if($A[j] > \text{pivot}$) \Rightarrow then decrement “j”.

Step(5): if 'i' cant increment and 'j' cant decrement, then swap(A[i], A[j])

Step(6): if 'i' and 'j' are crossed (or) if($A[j] < \text{pivot}$) \Rightarrow then swap($A[j]$, pivot)

Example:

Consider the list of elements $-50, 30, 10, 90, 80, 20, 40, 70$.

Step: 1

i							j
50	30	10	90	80	20	40	70
pivot							

Step: 2

i				j			
50	30	10	90	80	20	40	70

pivot

We will increment 'i', if(A[i] < pivot), we will continue to increment 'i' until the element the element of A[i] is greater than pivot.

Step: 3

		i					j
50	30	10	90	80	20	40	70
pivot							

Increment 'i' as $A[i] < \text{pivot}$

Step: 4

			i				j
50	30	10	90	80	20	40	70
pivot							

A[i] > pivot, So stop incrementing 'i'

Step: 5

A[j] > pivot, So decrement 'j'

		i			j		
50	30	10	90	80	20	40	70
pivot							

Step: 5

A[j] < pivot, So stop decrementing 'j'

			i				j
50	30	10	90	80	20	40	70
pivot							

Step: 6

'i' cant increment and 'j' cant decrement. Then swap(A[i], A[j])

		i			j		
50	30	10	40	80	20	90	70
pivot							

Step: 7

$A[i] < \text{pivot} \Rightarrow \text{Increment 'i'}$

				i	j		
50	30	10	40	80	20	90	70

pivot

Step: 8

$A[i] > \text{pivot} \Rightarrow$ Stop incrementing 'i'

				i					j
50	30	10	40	80	20	90	70		
pivot									

Step: 9

$A[j] > \text{pivot} \Rightarrow$ Decrement 'j'

				i	j			
50	30	10	40	80	20	90	70	
pivot								

Step: 10

$A[j] < \text{pivot} \Rightarrow$ Stop Decrementing 'j'

				i	j			
50	30	10	40	80	20	90	70	
pivot								

Step: 11

'i' cant increment and 'j' cant decrement. Then swap($A[i]$, $A[j]$)

				i	j			
50	30	10	40	20	80	90	70	
pivot								

Step: 12

$A[i] < \text{pivot} \Rightarrow$ Increment 'i'

				i, j				
50	30	10	40	20	80	90	70	
pivot								

Step: 13

$A[i] > \text{pivot} \Rightarrow$ Stop Incrementing 'i'

i, j							
50	30	10	40	20	80	90	70
pivot							

Step: 14

$A[j] > \text{pivot} \Rightarrow \text{Decrement 'j'}$

j				i			
50	30	10	40	20	80	90	70
pivot							

Step: 15

$A[j] < \text{pivot} \Rightarrow \text{Stop Decrementing 'j'}$

j				i			
50	30	10	40	20	80	90	70
pivot							

Step: 16

'i' and 'j' are crossed and $A[j] < \text{pivot} \Rightarrow \text{swap}(A[j], \text{pivot})$

i				j			
20	30	10	40	50	80	90	70
pivot							

Left sub array

Right Sub Array

HEAP SORT:

- ❖ Heap sort is one of the sorting algorithms used to arrange a list of elements in order.
- ❖ Heapsort algorithm uses one of the tree concepts called Heap Tree.
- ❖ In this sorting algorithm, we use Max Heap to arrange list of elements in ascending order and Min Heap to arrange list elements in descending order.

Step by Step Process

The Heap sort algorithm to arrange a list of elements in ascending order is performed using following steps...

Step 1 - Construct a Binary Tree with given list of Elements.

Step 2 - Transform the Binary Tree into Min Heap.

Step 3 - Delete the root element from Min Heap using Heapify method.

Step 4 - Put the deleted element into the Sorted list.

Step 5 - Repeat the same until Min Heap becomes empty.

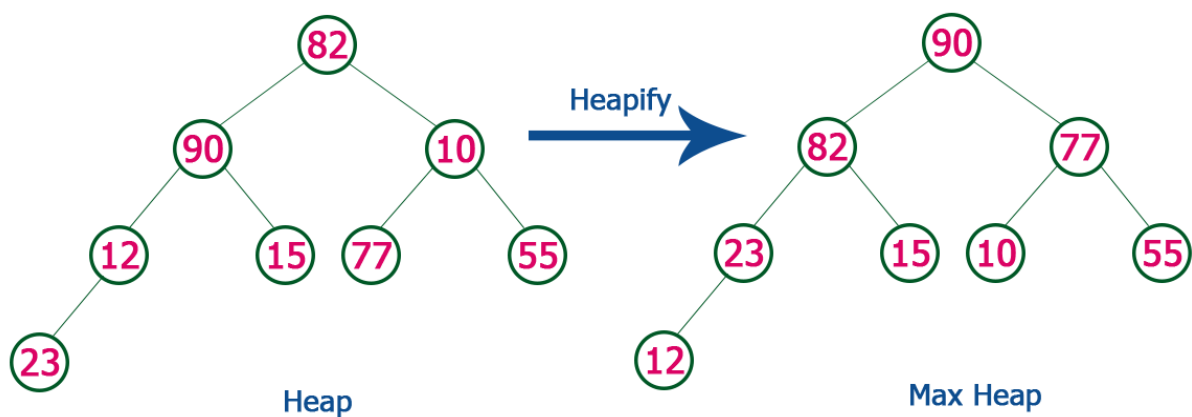
Step 6 - Display the sorted list.

Example

Consider the following list of unsorted numbers which are to be sort using Heap Sort –

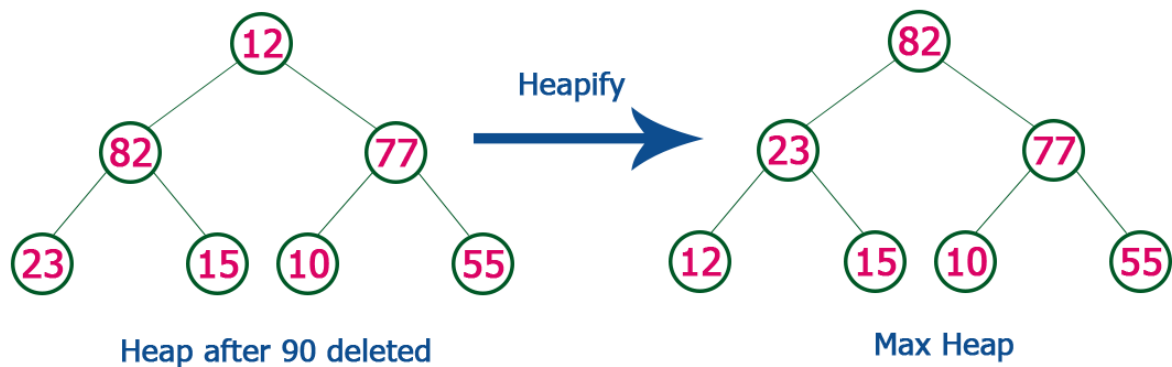
82, 90, 10, 12, 15, 77, 55, 23

Step -1 Construct a heap with given list of unsorted numbers and convert to Max-Heap.



List of numbers after heap converted to Max-Heap: **90, 82, 77, 23, 15, 10, 55, 12**

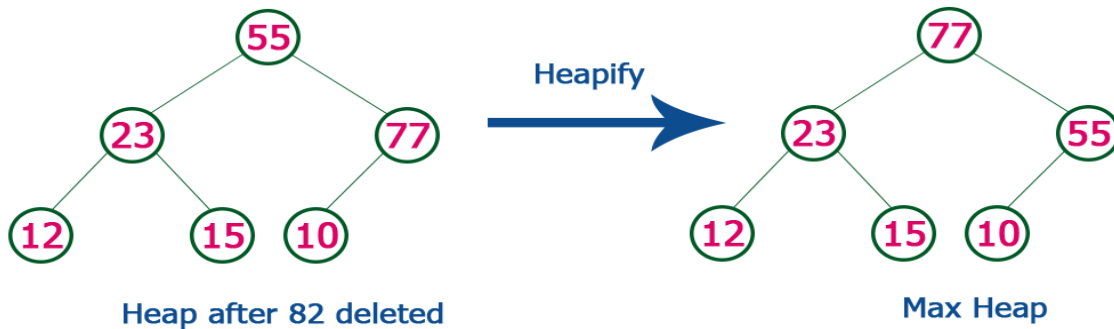
Step: 2 Delete Root (90) from Max-Heap. To delete root node it needs to be swapped with a last node (12). After deletion, Tree needs to be heapify to make it Max-Heap.



List of number after swapping 90 with 12:

12	82	77	23	15	10	55	90
----	----	----	----	----	----	----	-----------

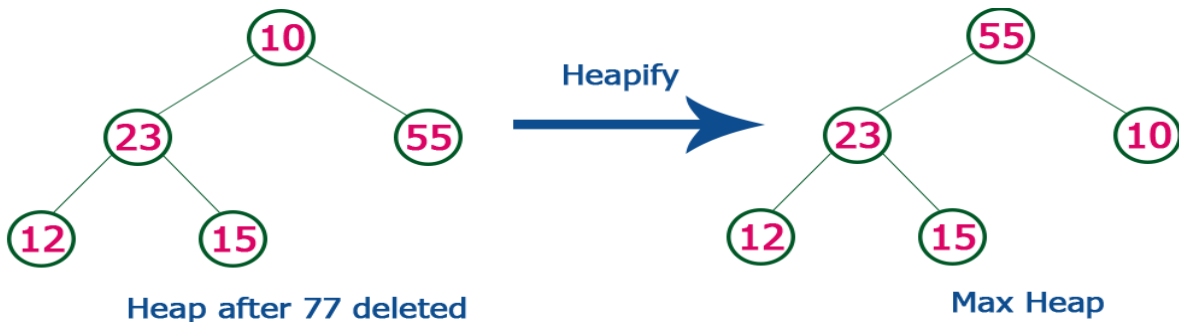
Step: 3 Delete Root (82) from Max-Heap. To delete root node it needs to be swapped with a last node (55). After deletion, Tree needs to be heapifying to make it Max-Heap.



List of number after swapping 82 with 55:

12	55	77	23	15	10	82	90
----	----	----	----	----	----	-----------	-----------

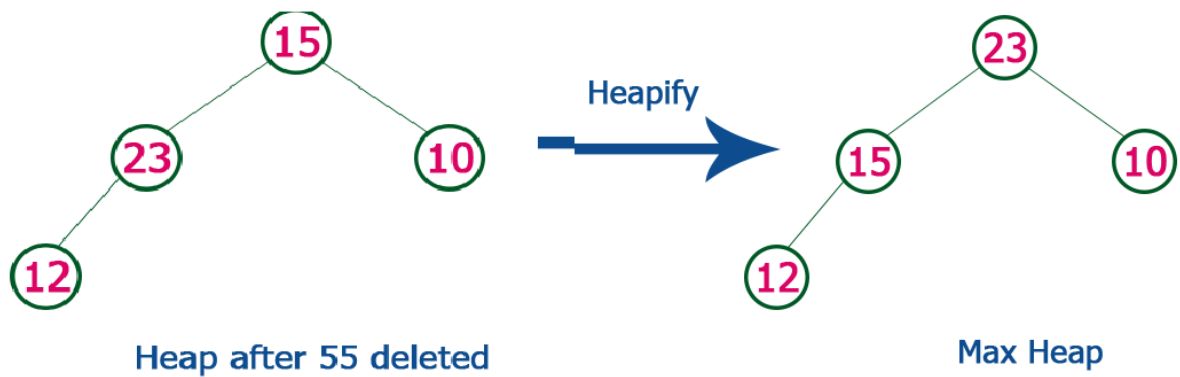
Step: 4 Delete Root (77) from Max-Heap. To delete root node it needs to be swapped with a last node (10). After deletion, Tree needs to be heapifying to make it Max-Heap.



List of number after swapping 77 with 10:

12	55	10	23	15	77	82	90
----	----	----	----	----	-----------	-----------	-----------

Step: 5 Delete Root (55) from Max-Heap. To delete root node it needs to be swapped with a last node (15). After deletion, Tree needs to be heapifying to make it Max-Heap.



List of number after swapping 55 with 15:

12	15	10	23	55	77	82	90
----	----	----	----	----	----	----	----

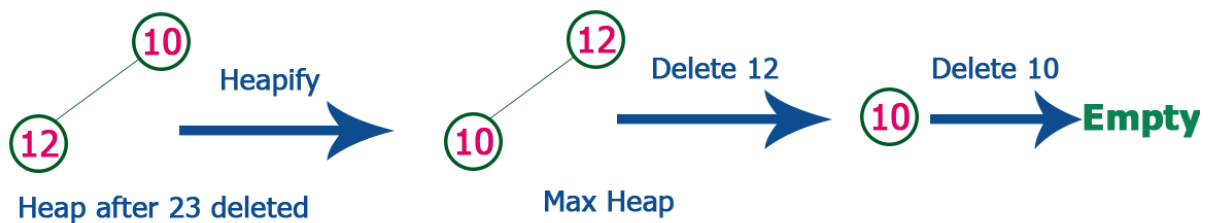
Step: 6 Delete Root (23) from Max-Heap. To delete root node it needs to be swapped with a last node (12). After deletion, Tree needs to be heapifying to make it Max-Heap.



List of number after swapping 23 with 12:

12	15	10	23	55	77	82	90
----	----	----	----	----	----	----	----

Step: 7 Delete Root (15) from Max-Heap. To delete root node it needs to be swapped with a last node (10). After deletion, Tree needs to be heapifying to make it Max-Heap.



List of number after swapping 23 with 12:

10	12	15	23	55	77	82	90
----	----	----	----	----	----	----	----

Whenever the Max-Heap becomes empty, the list gets sorted in ascending order.

Complexity of the Heap Sort Algorithm

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

- ❖ Worst Case: $O(n \log n)$
- ❖ Best Case: $O(n \log n)$
- ❖ Average Case: $O(n \log n)$

Sorting Method	Worst Case	Average Case	Best Case	Space Complexity
Bubble Sort	$n(n-1)/2 = O(n^2)$	$n(n-1)/2 = O(n^2)$	$n(n-1)/2 = O(n^2)$	Constant
Insertion Sort	$n(n-1)/2 = O(n^2)$	$n(n-1)/4 = O(n^2)$	$O(n)$	Constant
Selection Sort	$n(n-1)/2 = O(n^2)$	$n(n-1)/2 = O(n^2)$	$n(n-1)/2 = O(n^2)$	Constant
Quick Sort	$n(n+3)/2 = O(n^2)$	$O(n \log n)$	$O(n \log n)$	Constant
Heap Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Constant
Merge Sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	Depends