Malla Reddy College Engineering (Autonomous)

Maisammaguda, Dhulapally (Post Via. Hakimpet), Secunderabad ,Telangana-500100

www.mrec.ac.in

Department of Computer Science and Engineering

(AIML,CS,DS,IOT)

II B.TECH I SEM (A.Y.2021-22)

Lecture Notes

On

DATA STRUCTURES

2020-21 Onwards (MR-20)	MALLA REDDY ENGINEERING COLLEGE (Autonomous)	B.Tech. III Semester		
Code: A0509	Data Structures (Common for CSE, CSE (DS), CSE (AI and ML),CSE (Cyber Security), CSE (IOT) and IT)	L	T	P
Credits: 3		3	-	-

Prerequisites: A course on "Programming for Problem Solving "

Course Objectives:
- Exploring basic data structures such as linked list, stacks and queues.
- Introduces a variety of data structures such as dictionaries and hash tables
- To learn non linear data structures i.e. Binary search trees and height balanced trees.
- To understand the graph traversal algorithms and heap sort.
- Introduces the pattern matching and tries algorithms

MODULE-I: [10 Periods]
Introduction to Data Structures, abstract data types, Linear list – singly linked list implementation, insertion, deletion and searching operations on linear list, Stacks-Operations, array and linked representations of stacks, stack applications, Queues-operations, array and linked representations.

MODULE-II: [09 Periods]
Dictionaries: linear list representation, skip list representation, operations - insertion, deletion and searching.

Hash table representation: hash functions, collision resolution-separate chaining, open addressing-linear probing, quadratic probing, double hashing, rehashing, extendible hashing.

MODULE-III: [10 Periods]
Search Trees: Binary Search Trees, Definition, Implementation, Operations- Searching, Insertion and Deletion, AVL Trees, Definition, Height of an AVL Tree, Operations – Insertion, Deletion and Searching, Definition and example of Red –Black, Splay Trees.

MODULE-IV: [10 Periods]
Graphs: Graph Implementation Methods. Graph Traversal Methods.

Sortings: Max Heap, Min Heap,Heap Sort. External Sorting: Model for external sorting, Merge sort.

MODULE-V: [09 Periods]
Pattern matching and Tries: Pattern matching algorithms-Brute force, the Boyer –Moore algorithm, the Knuth-Morris-Pratt algorithm, Standard Tries, Compressed Tries, Suffix tries.

TEXT BOOKS:
1. Jean Paul Tremblay, Paul G Sorenson, "An Introduction to Data Structures with Applications", Tata McGraw Hills, 2nd Edition, 1984.
2. Richard F. Gilberg, Behrouz A. Forouzan, "Data Structures: A Pseudo code approach with C ", Thomson (India), 2nd Edition, 2004.

REFERENCES:

1. Horowitz, Ellis, Sahni, Sartaj, Anderson-Freed, Susan, "Fundamentals of Data Structure in C", University Press (India), 2nd Edition, 2008.
2. A. K. Sharma, "Data structures using C", Pearson, 2nd Edition, June, 2013.
3. R. Thareja, "Data Structures using C", Oxford University Press, 2nd Edition, 2014.

E-RESOURCES:
1. http://gvpcse.azurewebsites.net/pdf/data.pdf
2. http://www.sncwgs.ac.in/wp-content/uploads/2015/11/Fundamental-Data-Structures.pdf
3. http://www.learnerstv.com/Free-Computer-Science-Video-lectures-ltv247-Page1.htm
4. http://ndl.iitkgp.ac.in/document/yVCWqd6u7wgye1qwH9xY7-
   3lcmoMApVUMmjlExpIb1zste4YXX1pSpX8a2mLgDzZ-
   E41CJ6PVmY4S0MqVbxsFQ
5. http://nptel.ac.in/courses/106102064/1

Course Outcomes:
At the end of the course, students will be able to
1. **Implement** the linear data structures such as linked list, stacks and queues.
2. **Understand** the Dictionaries and Hash table representation.
3. **Analyze** the various non linear data structures with its operations
4. **Develop** the programs by using Graph Traversal and heap sort
5. **Apply** data structure concepts for the implementation of pattern matching and tries.

CO- PO,PSO Mapping (3/2/1 indicates strength of correlation) 3-Strong, 2-Medium, 1-Weak															
COs	Programme Outcomes (POs)												PSOs		
	PO1	PO2	PO3	PO4	PO5	PO6	PO7	PO8	PO9	PO10	PO11	PO12	PSO1	PSO2	PSO3
CO1	2	3	2										2	3	
CO2	2	2	3										3	2	
CO3		2	2											2	1
CO4		2	3										2	3	
CO5	2	3	3										2	3	

Data structure A data structure is a specialized format for organizing and storing data. General data structure types include the array, the file, the record, the table, the tree, and so on. Any data structure is designed to organize data to suit a specific purpose so that it can be accessed and worked with in appropriate ways

Abstract Data Type

In computer science, an abstract data type (ADT) is a mathematical model for data types where a data type is defined by its behavior (semantics) from the point of view of a user of the data, specifically in terms of possible values, possible operations on data of this type, and the behavior of these operations. When a class is used as a type, it is an abstract type that refers to a hidden representation. In this model an ADT is typically implemented as a class, and each instance of the ADT is usually a n object of that class.

In ADT all the implementation details are hidden
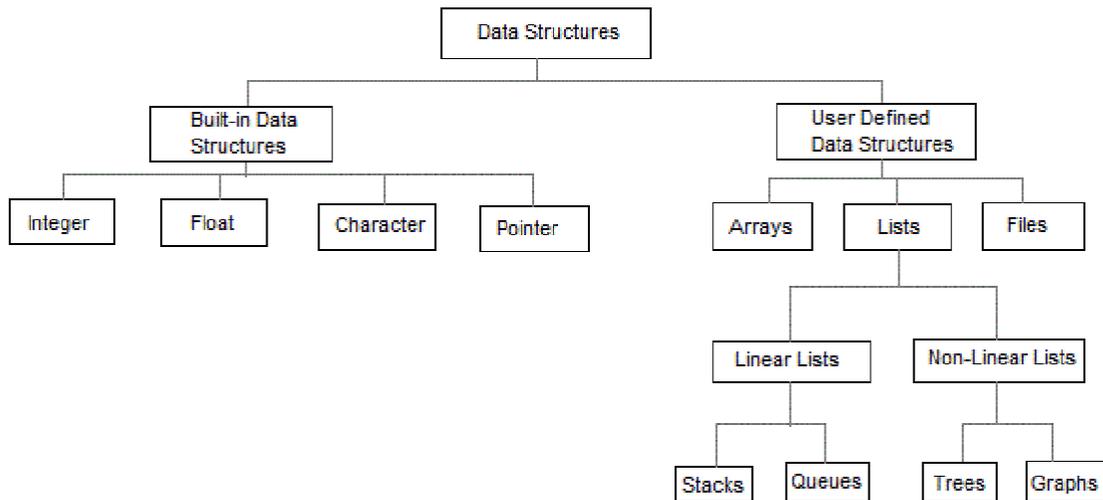
- Linear data structures are the data structures in which data is arranged in a list or in a sequence.
- Non linear data structures are the data structures in which data may be arranged in a hierarchic al manner

LIST ADT

List is basically the collection of elements arrange d in a sequential manner. In memory we can store the list in two ways: one way is we can store the elements in sequential memory locations. That means we can store the list in arrays.

The other way is we can use pointers or links to associate elements sequentially.

This is known as linked list.

LINKED LISTS

The linked list is very different type of collection from an array. Using such lists, we can store collections of information limited only by the total amount of memory that the OS will allow us to use.Further more, there is no need to specify our needs in advance. The linked list is very flexible dynamic data structure : items may be added to it or deleted from it at will. A programmer need not worry about how many items a program will have to accommodate in advance. This allows us to write robust programs which require much less maintenance.

The linked allocation has the following draw backs:
1. No direct access to a particular element.
2. Additional memory required for pointers.

Linked list are of 3 types:
1. Singly Linked List
2. Doubly Linked List
3. Circularly Linked List

SINGLY LINKED LIST

A singly linked list, or simply a linked list, is a linear collection of data items. The linear order is given by means of POINTERS. These types of lists are often referred to as **linear linked list**.
* Each item in the list is called a node.
* Each node of the list has two fields:
1. Information- contains the item being stored in the list.
2. Next address- contains the address of the next item in the list.
* The last node in the list contains NULL pointer to indicate that it is the end of the list.

Conceptual view of Singly Linked List

Data	Ptr		Data	Ptr		Data	Ptr		Data	NULL

Operations on Singly linked list:
- Insertion of a node
- Deletions of a node
- Traversing thelist

Structure of a node:

Method -1:

```
struct node
{
        int data;
        struct node *link;
};
```

| Data	link
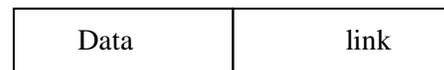
Method -2:

```
class node
{
public:
        int data;
        node *link;
};
```

Insertions: To place an elements in the list there are 3 cases :
1. At the beginning
2. End of the list
3. At a given position

case 1:Insert at the beginning

Eg:
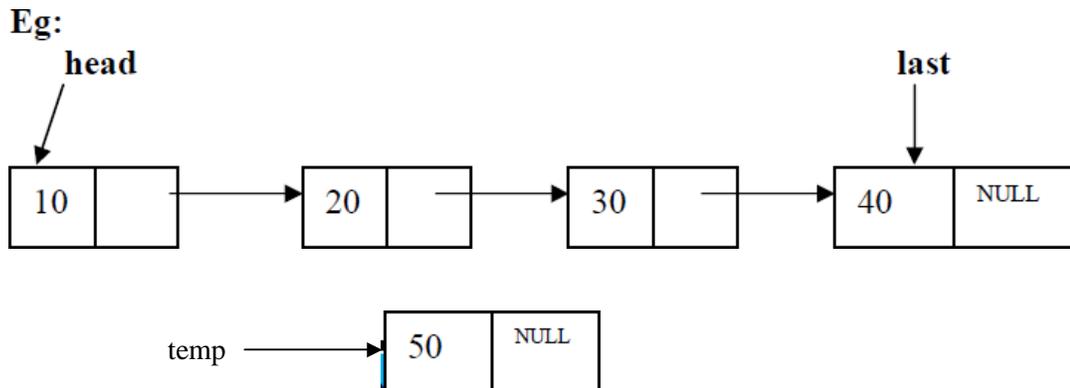
 head is the pointer variable which contains address of the first node and **temp** contains address of new node to be inserted then sample code is

```
temp->link=head;
head=temp;
```

After insertion:

Insert   at the beginning of list.
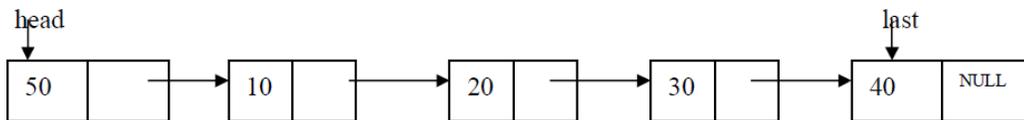
Code for insert front:-
```
template <class T>
void list<T>::insert_front()
{
struct node <T>*t,*temp;
        cout<<"Enter data into node:";
        cin>>item;
        temp=create_node(item);
        if(head==NULL)
                head=temp;
        else
        {       temp->link=head;
                head=temp;
        }
}
```

case 2: Inserting end of the list
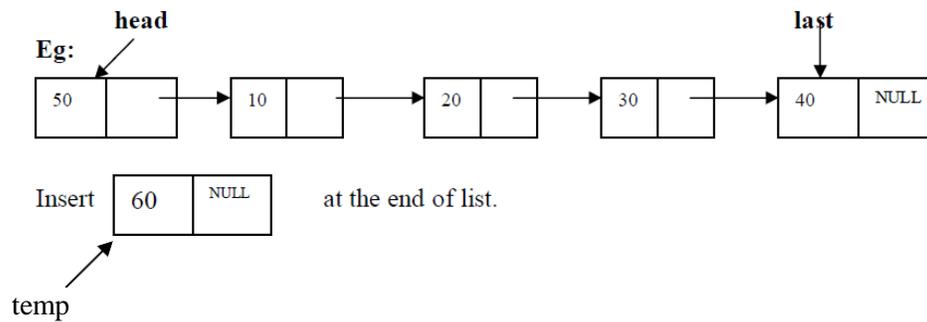
head is the pointer variable which contains address of the first node and **temp** contains address of new node to be inserted then sample code is

```
t=head;
while(t->link!=NULL)
{
        t=t->link;
}
t->link=temp;
```

After insertion the linked list is

Code for insert End:-

```
template <class T>
void list<T>::insert_end()
{
struct node<T> *t,*temp;
int n;
        cout<<"Enter data into node:";
        cin>>n;
        temp=create_node(n);
        if(head==NULL)
                head=temp;
        else
        {       t=head;
                while(t->link!=NULL)
                        t=t->link;
                t->link=temp;
        }
}
```

case 3: Insert at a position


```
c=1;
while(c<pos)
{
        prev=cur;
        cur=cur->link;
        c++;
}
prev->link=temp;
temp->link=cur;
```
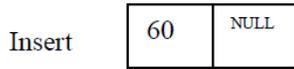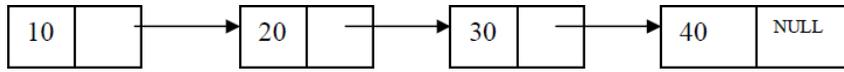
insert node at position 3
head is the pointer variable which contains address of the first node and **temp** contains address of new node to be inserted then sample code is
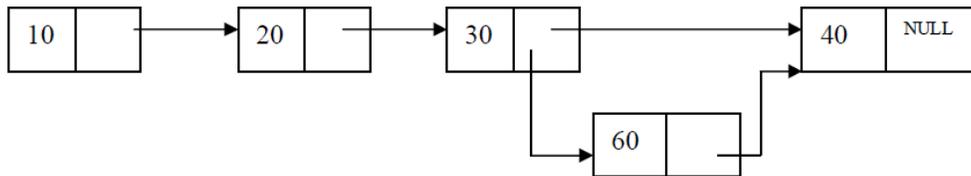

Code for inserting a node at a given position:-

```
template <class T>
void  list<T>::Insert_at_pos(int  pos)
{struct node<T>*cur,*prev,*temp;
int c=1;
        cout<<"Enter data into node:";
        cin>>item
        temp=create_node(item);
        if(head==NULL)
                head=temp;
         else
          {
                 prev=cur=head;
                if(pos==1)
                {
                        temp->link=head;
```

```
                head=temp;
            }
            else
            {
            while(c<pos)
            {       c++;
                    prev=cur;
                    cur=cur->link;
            }
            prev->link=temp;
            temp->link=cur;
            }
        }
}
```

Deletions: Removing an element from the list, without destroying the integrity of the list itself.
To place an element from the list there are 3 cases :
1. Delete a node at beginning of the list
2. Delete a node at end of the list
3. Delete a node at a given position

Case 1: Delete a node at beginning of the list
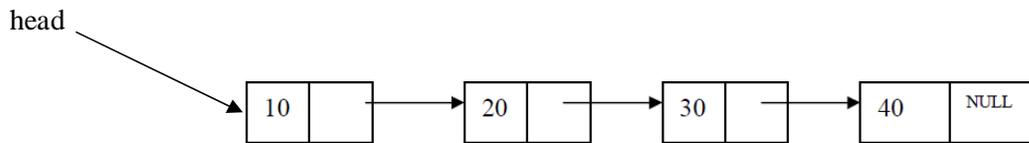
head is the pointer variable which contains address of the first node

sample code is

```
t=head;
head=head->link;
cout<<"node "<<t->data<<" Deletion is sucess";
delete(t);
```


code for deleting a node at front

```
template <class T>
void list<T>::delete_front()
{
struct node<T>*t;
        if(head==NULL)
                cout<<"List is Empty\n";
        else
        {       t=head;
```

```
                head=head->link;
                cout<<"node "<<t->data<<" Deletion is sucess";
                delete(t);
        }
}
```

Case 2. Delete a node at end of the list
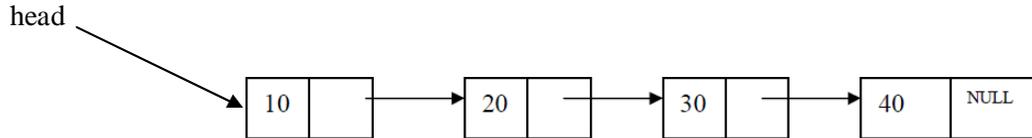
To delete last node , find the node using following code

```
struct node<T>*cur,*prev;
cur=prev=head;
while(cur->link!=NULL)
{       prev=cur;
        cur=cur->link;
}
prev->link=NULL;
cout<<"node "<<cur->data<<" Deletion is sucess";
free(cur);
```
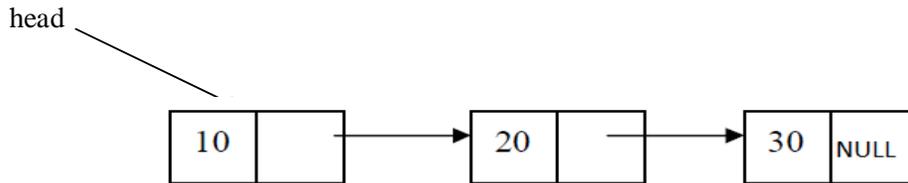

code for deleting a node at end of the list

```
        template <class T>
        void list<T>::delete_end()
        {
        struct node<T>*cur,*prev;
                cur=prev=head;
                if(head==NULL)
                        cout<<"List is Empty\n";
                else
                {       cur=prev=head;
                        if(head->link==NULL)
                        {
                                cout<<"node "<<cur->data<<" Deletion is sucess";
                                free(cur);
                                head=NULL;
                        }
```

```
            else
            {       while(cur->link!=NULL)
                    {       prev=cur;
                            cur=cur->link;
                    }
                    prev->link=NULL;
                    cout<<"node "<<cur->data<<" Deletion is sucess";
                    free(cur);
            }
        }
    }
```

CASE 3. Delete a node at a given position

head

|10| | → |20| | → |30| | → |40|NULL|
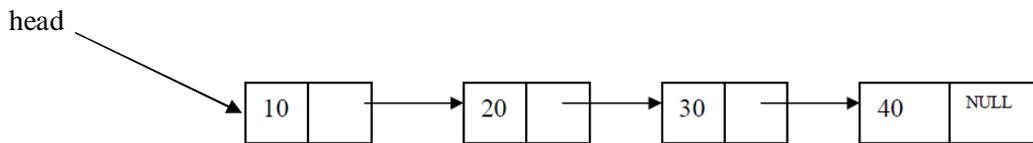
 Delete node at position 3
head is the pointer variable which contains address of the first node. Node to be deleted is node containing value 30.
Finding node at position 3

```
c=1;
while(c<pos)
{       c++;
        prev=cur;
        cur=cur->link;
}
```

prev   cur

|10| | → |20| | → |30| | → |40|NULL|
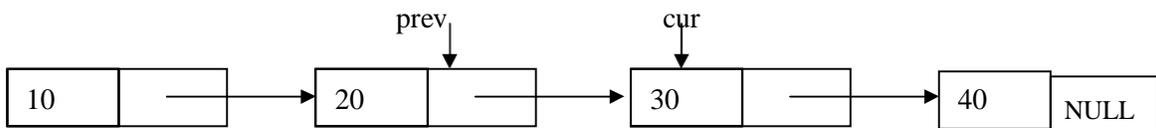
 cur is the node to be deleted . before deleting update links

code to update links

```
prev->link=cur->link;
cout<<cur->data <<"is deleted successfully";
delete cur;
```

prev   cur

|10| | → |20| | → |30| | → |40|NULL|

8
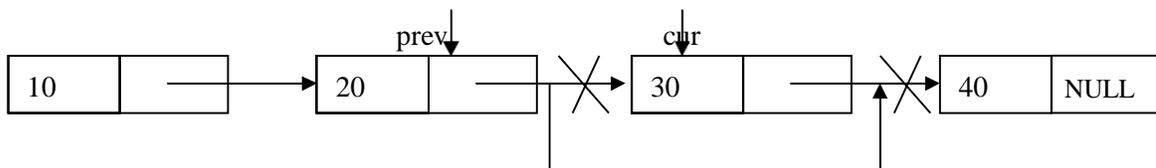
Traversing the list: Assuming we are given the pointer to the head of the list, how do we get the end of the list.

```cpp
template <class T>
void list<T>:: display()
{
        struct node<T>*t;

        if(head==NULL)
         {
                        cout<<"List is Empty\n";
         }
         else
         {      t=head;
                while(t!=NULL)
                {      cout<<t->data<<"->";
                        t=t->link;
                }
         }
}
```

Dynamic Implementation of list ADT

```cpp
#include<iostream.h>
#include<stdlib.h>
template <class T>
struct node
{
        T data;
        struct node<T> *link;
};
template <class T>
class list
{
        int item;
        struct node<T>*head;
        public:
        list();
        void display();
        struct node<T>*create_node(int n);
        void insert_end();
        void insert_front();
        void Insert_at_pos(int pos);
        void delete_end();
        void delete_front();
        void Delete_at_pos(int pos);
        void Node_count();
};
```

```cpp
template <class T>
 list<T>::list()
        {
                head=NULL;
        }

template <class T>
void list<T>:: display()
{
        struct node<T>*t;

        if(head==NULL)
         {
                        cout<<"List is Empty\n";
         }
         else
         {      t=head;
                while(t!=NULL)
                {       cout<<t->data<<"->";
                        t=t->link;
                }
         }
}

template <class T>
struct node<T>* list<T>::create_node(int n)
{struct node<T> *t;
        t=new struct node<T>;
        t->data=n;
        t->link=NULL;
return t;
}

template <class T>
void list<T>::insert_end()
{struct node<T> *t,*temp;
int n;
        cout<<"Enter data into node:";
        cin>>n;
        temp=create_node(n);
        if(head==NULL)
                head=temp;
        else
        {       t=head;
                while(t->link!=NULL)
                        t=t->link;
                t->link=temp;
        }
}
```

```
template <class T>
void list<T>::insert_front()
{
struct node <T>*t,*temp;
        cout<<"Enter data into node:";
        cin>>item;
        temp=create_node(item);
        if(head==NULL)
                head=temp;
        else
        {       temp->link=head;
                head=temp;
        }
}

template <class T>
void list<T>::delete_end()
{
struct node<T>*cur,*prev;
        cur=prev=head;
        if(head==NULL)
                cout<<"List is Empty\n";
        else
        {       cur=prev=head;
                if(head->link==NULL)
                {
                        cout<<"node "<<cur->data<<" Deletion is sucess";
                        free(cur);
                        head=NULL;
                }
                else
                {       while(cur->link!=NULL)
                        {       prev=cur;
                                cur=cur->link;
                        }
                        prev->link=NULL;
                        cout<<"node "<<cur->data<<" Deletion is sucess";
                        free(cur);
                }
        }
}

template <class T>
void list<T>::delete_front()
{
struct node<T>*t;
        if(head==NULL)
                cout<<"List is Empty\n";
        else
        {       t=head;
                head=head->link;
```

```
                    cout<<"node "<<t->data<<" Deletion is sucess";
                    delete(t);
        }
}

template <class T>
void list<T>::Node_count()
{
struct node<T>*t;
int c=0;
        t=head;
        if(head==NULL)
            {
                cout<<"List is Empty\n";

            }
        else
        {       while(t!=NULL)
                {       c++;
                        t=t->link;
                }
                cout<<"Node Count="<<c<<endl;
        }
}

template <class T>
void list<T>::Insert_at_pos(int pos)
{struct node<T>*cur,*prev,*temp;
int c=1;
        cout<<"Enter data into node:";
                    cin>>item
                    temp=create_node(item);
        if(head==NULL)
                head=temp;
        else
        {  prev=cur=head;
                if(pos==1)
                {
                        temp->link=head;
                        head=temp;
                }
                else
                {
                while(c<pos)
                {       c++;
                        prev=cur;
                        cur=cur->link;
                }
                prev->link=temp;
                temp->link=cur;
                }
```

```cpp
        }
}

template <class T>
void list<T>::Delete_at_pos(int pos)
{
struct node<T>*cur,*prev,*temp;
int c=1;

        if(head==NULL)
        {
                        cout<<"List is Empty\n";
                }
        else
          { prev=cur=head;
                if(pos==1)
                {
                head=head->link;
                cout<<cur->data <<"is deleted sucesfully";
                delete cur;
                }
                else
                {
                        while(c<pos)
                        {       c++;
                                prev=cur;
                                cur=cur->link;
                        }
                        prev->link=cur->link;
                        cout<<cur->data <<"is deleted sucesfully";
                        delete cur;
                }
        }
}

int main()
{
int ncount,ch,pos;
list <int> L;
        while(1)
        {
                cout<<"\n ***Operations on Linked List***"<<endl;
                cout<<"\n1.Insert node at End"<<endl;
                cout<<"2.Insert node at Front"<<endl;
                cout<<"3.Delete node at END"<<endl;
                cout<<"4.Delete node at Front"<<endl;
                cout<<"5.Insert at a position "<<endl;
                cout<<"6.Delete at a position "<<endl;
                cout<<"7.Node Count"<<endl;
                cout<<"8.Display nodes "<<endl;
                cout<<"9.Clear Screen "<<endl;
```

```
cout<<"10.Exit "<<endl;
cout<<"Enter Your choice:";
cin>>ch;
switch(ch)
{
        case 1: L.insert_end();
                        break;
        case 2: L.insert_front();
                        break;
        case 3:L.delete_end();
                        break;
        case 4:L.delete_front();
                        break;
        case 5: cout<<"Enter position to insert";
                        cin>>pos;
                        L.Insert_at_pos(pos);
                        break;
        case 6: cout<<"Enter position to insert";
                        cin>>pos;
                        L.Delete_at_pos(pos);
                        break;
        case 7: L.Node_count();
                        break;
        case 8: L.display();
                        break;
        case 9:system("cls");
                        break;
        case 10:exit(0);

        default:cout<<"Invalid choice";
        }
    }
}
```

DOUBLY LINKED LIST

A singly linked list has the disadvantage that we can only traverse it in one direction. Many applications require searching backwards and forwards through sections of a list. A useful refinement that can be made to the singly linked list is to create a doubly linked list. The distinction made between the two list types is that while singly linked list have pointers going in one direction, doubly linked list have pointer both to the next and to the previous element in the list. The main advantage of a doubly linked list is that, they permit traversing or searching of the list in both directions.

In this linked list each node contains three fields.
 a) One to store data
 b) Remaining are self referential pointers which points to previous and next nodes in the list

prev	data	next

Implementation of node using structure

Method -1:

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

Implementation of node using class

Method -2:

```
class node
{
public:
        int data;
        node *prev;
        node * next;
};
```


Operations on Doubly linked list:
- ➢ Insertion of a node
- ➢ Deletions of a node
- ➢ Traversing the list

Doubly linked list ADT:

```
template <class T>
class dlist
{
        int data;
        struct dnode<T>*head;
public:
        dlist()
        {
                head=NULL;
        }
        void display();
        struct dnode<T>*create_dnode(int n);
        void insert_end();
        void insert_front();
        void delete_end();
        void delete_front();
        void dnode_count();
```

```
        void Insert_at_pos(int pos);
        void Delete_at_pos(int pos);
};
```

Insertions: To place an elements in the list there are 3 cases
- ➢ 1. At the beginning
- ➢ 2. End of the list
- ➢ 3. At a given position

**case 1:Insert at the beginning**

head is the pointer variable which contains address of the first node and **temp** contains address of new node to be inserted then sample code is

```
 temp->next=head;
 head->prev=temp;
 head=temp;
```
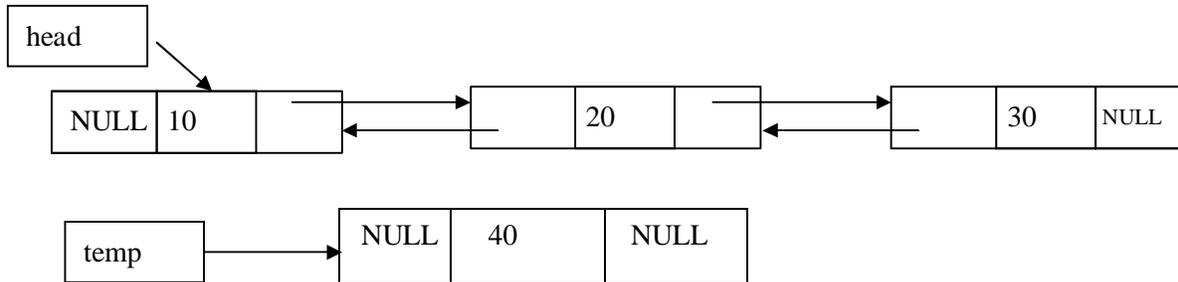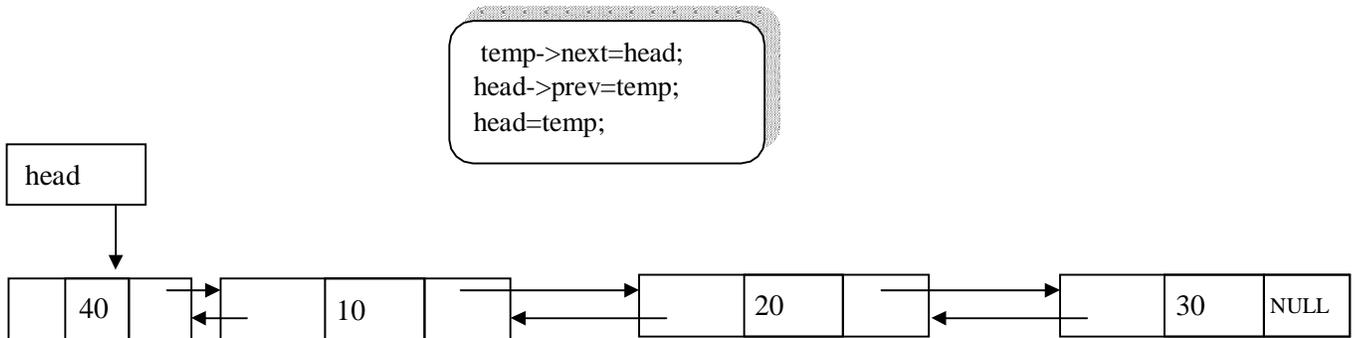

Code for insert front:-
```
template <class T>
void DLL<T>::insert_front()
{
struct dnode <T>*t,*temp;
        cout<<"Enter data into node:";
        cin>>data;
        temp=create_dnode(data);
        if(head==NULL)
                head=temp;
        else
        { temp->next=head; head-
                >prev=temp;
                head=temp;
        }
}
```

case 2: Inserting end of the list
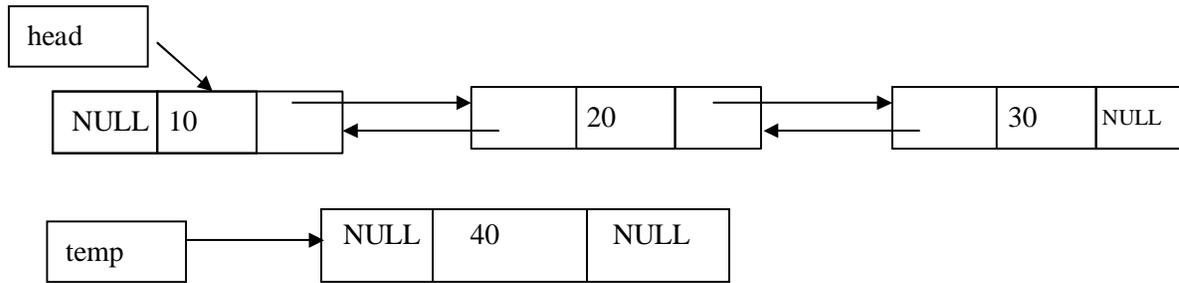
```
head
```

| NULL | 10 | | | 20 | | | 30 | NULL |

| | temp | | NULL | 40 | NULL |

head is the pointer variable which contains address of the first node and temp contains address of new node to be inserted then sample code is

```
t=head;
while(t->next!=NULL)
        t=t->next;
t->next=temp;
temp->prev=t;
```

```
head
```

| NULL | 10 | | | 20 | | | 30 | NULL |

| NULL | 40 | NULL |
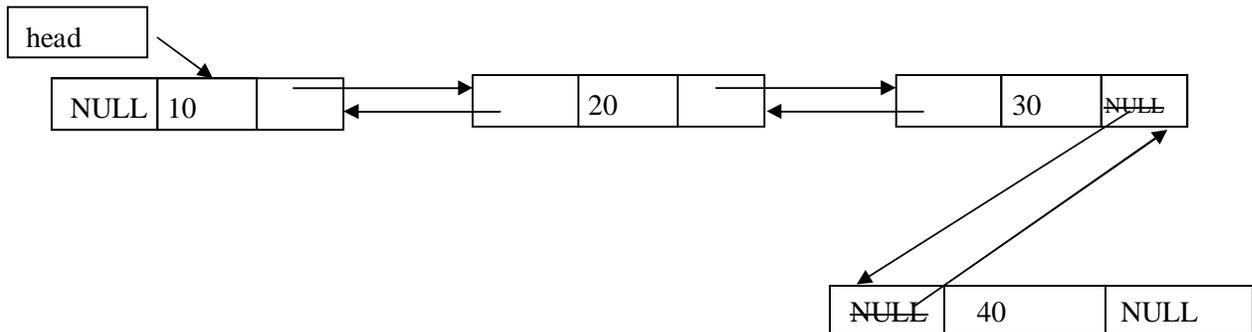
Code to insert a node at End:-

```
template <class T>
void DLL<T>::insert_end()
{
struct dnode<T> *t,*temp;
int n;
        cout<<"Enter data into dnode:";
        cin>>n;
        temp=create_dnode(n);
        if(head==NULL)
                head=temp;
        else
        {       t=head;
                while(t->next!=NULL)
                        t=t->next;
```

```
                                    t->next=temp;
                                    temp->prev=t;
                        }
            }
```

case 3: Inserting at a give position
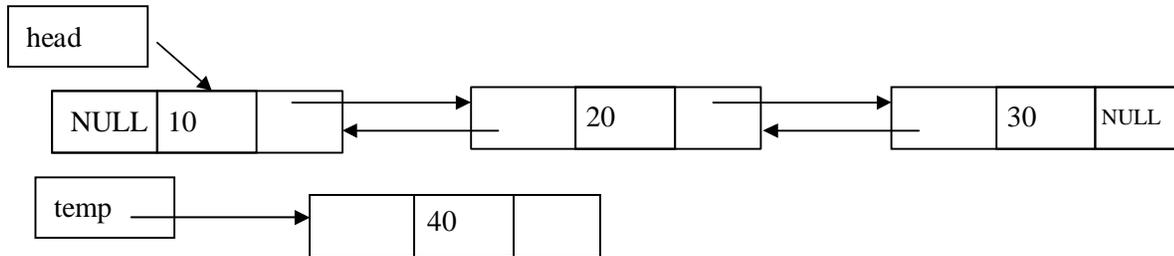
insert 40 at position 2
head is the pointer variable which contains address of the first node and **temp** contains address of new
node to be inserted then sample code is

```
while(count<pos)
{       count++;
        pr=cr;
        cr=cr->next;
}
pr->next=temp;
temp->prev=pr;
temp->next=cr;
cr->prev=temp;
```
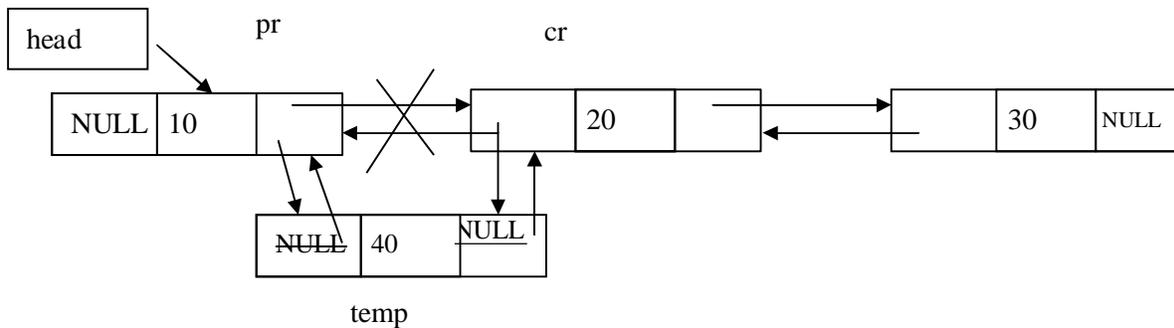
Code to insert a node at a position

```
template <class T>
void dlist<T>::Insert_at_pos(int pos)
{
struct dnode<T>*cr,*pr,*temp;
int count=1;
        cout<<"Enter data into dnode:";
        cin>>data;
        temp=create_dnode(data);
        display();
        if(head==NULL)
        {//when list is empty
                head=temp;
        }
        else
        {       pr=cr=head;
                if(pos==1)
                {       //inserting at pos=1
                        temp->next=head;
                        head=temp;
                }
                else
                {
                        while(count<pos)
                        {       count++;
                                pr=cr;
                                cr=cr->next;
                        }
                        pr->next=temp;
                        temp->prev=pr;
                        temp->next=cr;
                        cr->prev=temp;
                }
        }
}
```

Deletions: Removing an element from the list, without destroying the integrity of the list itself.
To place an element from the list there are 3 cases :
1. Delete a node at beginning of the list
2. Delete a node at end of the list
3. Delete a node at a given position

Case 1: Delete a node at beginning of the list

head

head is the pointer variable which contains address of the first node

sample code is

```
t=head;
head=head->next;
head->prev=NULL;
cout<<"dnode "<<t->data<<" Deletion is sucess";
delete(t);
```
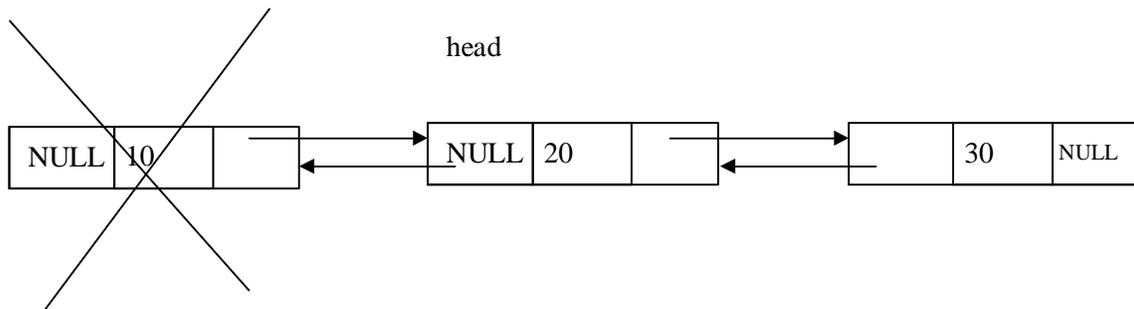
head

```
NULL | 10 |        NULL | 20 |        | 30 | NULL
```

code for deleting a node at front

```
template <class T>
void dlist<T>:: delete_front()
{struct dnode<T>*t;
        if(head==NULL)
                cout<<"List is Empty\n";
        else
        {       t=head;
                head=head->next;
                head->prev=NULL;
                cout<<"dnode "<<t->data<<" Deletion is sucess";
                delete(t);
        }
}
```

Case 2. Delete a node at end of the list
To deleted the last node find the last node. find the node using following code

```
struct  dnode<T>*pr,*cr;
        pr=cr=head;
while(cr->next!=NULL)
{       pr=cr;
        cr=cr->next;
}
pr->next=NULL;
cout<<"dnode "<<cr->data<<" Deletion is sucess";
delete(cr);
```
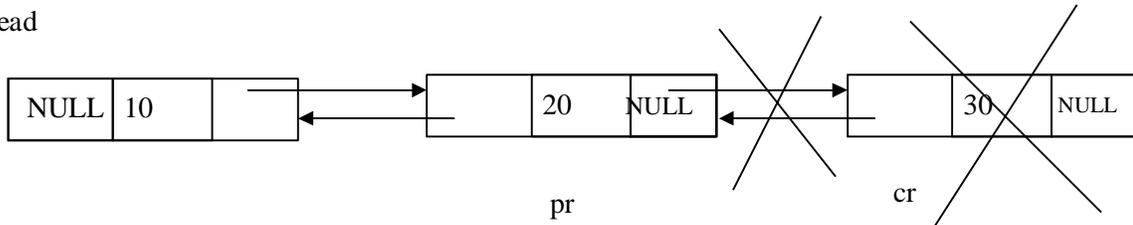
head

code for deleting a node at end of the list

```
template <class T>
void dlist<T>::delete_end()
{
struct dnode<T>*pr,*cr;
        pr=cr=head;
        if(head==NULL)
                cout<<"List is Empty\n";
        else
        {       cr=pr=head;
                if(head->next==NULL)
                {
                        cout<<"dnode "<<cr->data<<" Deletion is sucess";
                        delete(cr);
                        head=NULL;
                }
                else
                {       while(cr->next!=NULL)
                        {       pr=cr;
                                cr=cr->next;
                        }
                        pr->next=NULL;
                        cout<<"dnode "<<cr->data<<" Deletion is sucess";
                        delete(cr);
                }
        }
}
```

CASE 3. Delete a node at a given position

head

Delete node at position 2
head is the pointer variable which contains address of the first node. Node to be deleted is node containing value 30.
Finding node at position 2.

```
while(count<pos)
{        pr=cr;
         cr=cr->next;
         count++;
}
pr->next=cr->next;
cr->next->prev=pr;
```

head

| NULL | 10 | | | | 30 | | | | 20 | NULL |
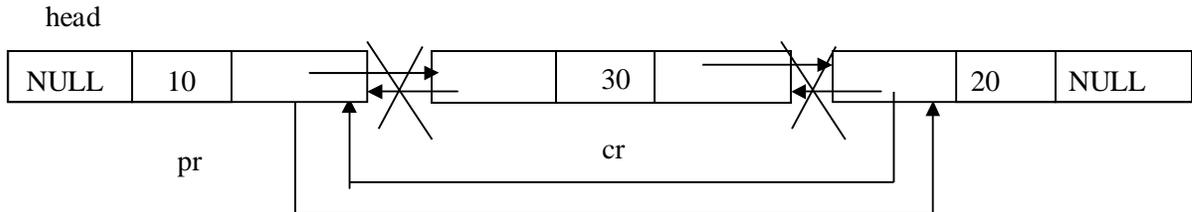
pr   cr

code for deleting a node at a position

```
template <class T>
void dlist<T>::Delete_at_pos(int pos)
{
struct dnode<T>*cr,*pr,*temp;
int count=1;
        display();
        if(head==NULL)
        {
                    cout<<"List is Empty\n";
                }
        else
        {  pr=cr=head;
              if(pos==1)
              {
                      head=head->next;
                      head->prev=NULL;
                      cout<<cr->data <<"is deleted sucesfully";
                      delete cr;
              }
              else
              {
                      while(count<pos)
                      {        count++;
                          pr=cr;
                          cr=cr->next;
                      }
                      pr->next=cr->next;
                      cr->next->prev=pr;
                      cout<<cr->data <<"is deleted sucesfully";
                      delete cr;
              }
        }
}
```

Dynamic Implementation of Doubly linked list ADT

```cpp
#include<iostream.h>
template <class T>
struct dnode
{
T data;
struct dnode<T> *prev;
struct dnode<T> *next;
};
template <class T>
class dlist
{
        int data;
        struct dnode<T>*head;
public:
        dlist();
        struct dnode<T>*create_dnode(int n);
        void insert_front();
        void insert_end();
        void Insert_at_pos(int pos);
        void delete_front();
        void delete_end();
        void Delete_at_pos(int pos);
        void dnode_count();
        void display();
};

template <class T>
 dlist<T>::dlist()
        {
                head=NULL;
        }

template <class T>
struct dnode<T>*dlist<T>::create_dnode(int n)
{
struct dnode<T> *t;
        t=new struct dnode<T>;
        t->data=n;
        t->next=NULL;
        t->prev=NULL;
return t;
}
template <class T>
void dlist<T>::insert_front()
{
struct dnode <T>*t,*temp;
        cout<<"Enter data into  dnode:";
```

```
            cin>>data;
            temp=create_dnode(data);
            if(head==NULL)
                    head=temp;
            else
            { temp->next=head; head-
                    >prev=temp;
                    head=temp;
            }
}
template <class T>
void dlist<T>::insert_end()
{
struct dnode<T> *t,*temp;
int n;
            cout<<"Enter data into dnode:";
            cin>>n;
            temp=create_dnode(n);
            if(head==NULL)
                    head=temp;
            else
            {        t=head;
                    while(t->next!=NULL)
                            t=t->next;
                    t->next=temp;
                    temp->prev=t;
            }
}


template <class T>
void dlist<T>::Insert_at_pos(int pos)
{
struct dnode<T>*cr,*pr,*temp;
int count=1;
            cout<<"Enter data into dnode:";
            cin>>data;
            temp=create_dnode(data);
            display();
            if(head==NULL)
            {//when list is empty
                    head=temp;
            }
             else
            {        pr=cr=head;
                    if(pos==1)
                    {        //inserting at pos=1
                            temp->next=head;
                            head=temp;
                    }
                    else
```

```
                {
                        while(count<pos)
                        {       count++;
                                pr=cr;
                                cr=cr->next;
                        }
                        pr->next=temp;
                        temp->prev=pr;
                        temp->next=cr;
                        cr->prev=temp;
                }
        }
}

template <class T>
void dlist<T>:: delete_front()
{struct dnode<T>*t;
        if(head==NULL)
                cout<<"List is Empty\n";
        else
        {       display();
                t=head;
                head=head->next;
                head->prev=NULL;
                cout<<"dnode "<<t->data<<" Deletion is sucess";
                delete(t);
        }
}
template <class T>
void dlist<T>::delete_end()
{
struct dnode<T>*pr,*cr;
        pr=cr=head;
        if(head==NULL)
                cout<<"List is Empty\n";
        else
        {       cr=pr=head;
                if(head->next==NULL)
                {
                        cout<<"dnode "<<cr->data<<" Deletion is sucess";
                        delete(cr);
                        head=NULL;
                }
                else
                {       while(cr->next!=NULL)
                        {       pr=cr;
                                cr=cr->next;
                        }
                        pr->next=NULL;
                        cout<<"dnode "<<cr->data<<" Deletion is sucess";
                        delete(cr);
```

```
                }
            }
}
template <class T>
void dlist<T>::Delete_at_pos(int pos)
{
struct dnode<T>*cr,*pr,*temp;
int count=1;
        display();
        if(head==NULL)
        {
                    cout<<"List is Empty\n";
                }
        else
        {  pr=cr=head;
                if(pos==1)
                {
                        head=head->next;
                        head->prev=NULL;
                        cout<<cr->data <<"is deleted sucesfully";
                        delete cr;
                }
                else
                {
                        while(count<pos)
                        {       count++;
                                pr=cr;
                                cr=cr->next;
                        }
                        pr->next=cr->next;
                        cr->next->prev=pr;
                        cout<<cr->data <<"is deleted sucesfully";
                        delete cr;
                }
            }
}
template <class T>
void dlist<T>::dnode_count()
{
struct dnode<T>*t;
int count=0;
        display();
        t=head;
        if(head==NULL)
                cout<<"List is Empty\n";
        else
        {       while(t!=NULL)
                {       count++;
                        t=t->next;
                }
                cout<<"node count is "<<count;
```

```
        }
}
template <class  T>
void dlist<T>::display()
{
        struct dnode<T>*t;
        if(head==NULL)
         {
                        cout<<"List is Empty\n";
         }
         else
        {       cout<<"Nodes in the linked list are ...\n";
                t=head;
                while(t!=NULL)
                {       cout<<t->data<<"<=>";
                        t=t->next;
                }
        }
}
int main()
{
int  ch,pos;
dlist <int> DL;
        while(1)
        {
                cout<<"\n ***Operations on Doubly List***"<<endl;
                cout<<"\n1.Insert dnode at End"<<endl;
                cout<<"2.Insert dnode at Front"<<endl;
                cout<<"3.Delete dnode at END"<<endl;
                cout<<"4.Delete dnode at Front"<<endl;
                cout<<"5.Display nodes "<<endl;
                cout<<"6.Count Nodes"<<endl;
                cout<<"7.Insert at a position "<<endl;
                cout<<"8.Delete at a position "<<endl;
                cout<<"9.Exit "<<endl;
                cout<<"10.Clear Screen "<<endl;
                cout<<"Enter Your choice:";
                cin>>ch;
                switch(ch)
                {
                        case 1: DL.insert_end();
                                        break;
                        case 2: DL.insert_front();
                                        break;
                        case 3:DL.delete_end();
                                        break;
                        case 4:DL.delete_front();
                                        break;
                        case 5://display contents
                                        DL.display();
                                        break;
```

```
                    case 6: DL.dnode_count();
                                    break;
                    case 7: cout<<"Enter position to insert";
                                    cin>>pos;
                                    DL.Insert_at_pos(pos);
                                    break;
                    case 8: cout<<"Enter position to Delete";
                                    cin>>pos;
                                    DL.Delete_at_pos(pos);
                                    break;
                    case 9:exit(0);
                    case 10:system("cls");
                                    break;
                    default:cout<<"Invalid choice";
            }
        }
}
```

CIRCULARLY LINKED LIST
A circularly linked list, or simply circular list, is a linked list in which the last node is always points to the first node. This type of list can be build just by replacing the NULL pointer at the end of the list with a pointer which points to the first node. There is no first or last node in the circular list.

Advantages:
  ➢ Any node can be traversed starting from any other node in the list.
  ➢ There is no need of NULL pointer to signal the end of the list and hence, all pointers contain valid addresses.
  ➢ In contrast to singly linked list, deletion operation in circular list is simplified as the search for the previous node of an element to be deleted can be started from that item itself.
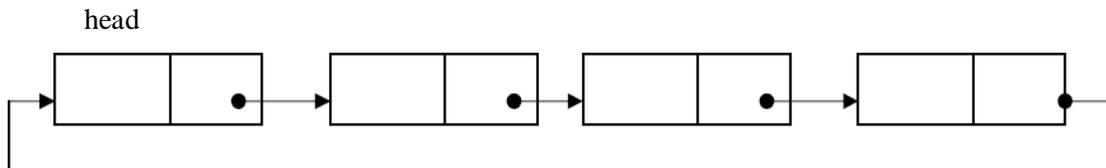
head

Dynamic Implementation of Circular linked list ADT

```
#include<iostream.h>
#include<stdlib.h>
template <class T>
struct cnode
{
T data;
struct cnode<T> *link;
};
//Code fot circular linked List ADT
template <class T>
```

```
class clist
{
        int data;
        struct cnode<T>*head;
        public:
                clist();
                struct cnode<T>* create_cnode(int n);
                void display();
                void insert_end();
                void insert_front();
                void delete_end();
                void delete_front();
                void cnode_count();

};
//code for defaut constructor
template <class T>
clist<T>::clist()
        {
                head=NULL;
        }

//code to display elements in the list
template <class T>
void clist<T>::display()
{
        struct cnode<T>*t;
        if(head==NULL)
         {
                cout<<"clist is Empty\n";
         }
         else
         {      t=head;
                if(t->link==head)
                        cout<<t->data<<"->";
                else
                {
                        cout<<t->data<<"->";
                        t=t->link;
                        while(t!=head)
                        {
                                cout<<t->data<<"->";
                                t=t->link;
                        }
                }
         }
}

//Code to create node
template <class T>
struct cnode<T>* clist<T>::create_cnode(int n)
```

```
{
struct cnode<T> *t;
        t=new struct cnode<T>;
        t->data=n;
        t->link=NULL;
return t;
}
//Code to insert node at the end
template <class T>
void clist<T>::insert_end()
{
struct cnode<T>*t;
struct cnode<T>*temp;
int n;
        cout<<"Enter data into cnode:";
        cin>>n;
        temp=create_cnode(n);
        if(head==NULL)
        {
                head=temp;
                temp->link=temp;
        }
        else
        {
                t=head;
                if(t->link==head)// list containing only one node
                {
                        t->link=temp;
                        temp->link=t;
                }
                else
                {
                        while(t->link!=head)
                        {
                                t=t->link;
                        }
                        t->link=temp;
                        temp->link=head;
                }
        }
        cout<<"Node inerted"<<endl;
}

//Code to insert node at front
template <class T>
void clist<T>::insert_front()
{
struct cnode <T>*t;
struct cnode<T>*temp;
        cout<<"Enter data into cnode:";
        cin>>data;
```

```
        temp=create_cnode(data);
        if(head==NULL)
        {
                head=temp;
                temp->link=temp;
        }
        else
        {
                t=head;
                if(t->link==head)
                {
                        t->link=temp;
                        temp->link=t;
                }
                else
                {
                        //code to find last node
                        while(t->link!=head)
                        {
                                t=t->link;
                        }
                        t->link=temp; //linking last and first node
                        temp->link=head;
                        head=temp;

                }
        }
cout<<"Node inserted \n";
}

//Code to delete node at end
template <class T>
void clist<T>::delete_end()
{
struct cnode<T>*cur,*prev;
        cur=prev=head;
        if(head==NULL)
                cout<<"clist is Empty\n";
        else
        {       cur=prev=head;
                if(cur->link==head)
                {
                        cout<<"cnode "<<cur->data<<" Deletion is sucess";
                        free(cur);
                        head=NULL;
                }
                else
                {       while(cur->link!=head)
                        {       prev=cur;
                                cur=cur->link;
                        }
```

```
                        //prev=cur;
                        //cur=cur->link;
                        prev->link=head;//points to head
                        cout<<"cnode "<<cur->data<<" Deletion is sucess";
                        free(cur);
                }
        }
}
//Code to delete node at front
template <class T>
void clist<T>::delete_front()
{
struct cnode<T>*t,*temp;
        if(head==NULL)
                cout<<"circular list is Empty\n";
        else
        {       t=head;
                //head=head->link;
                if(t->link==head)
                {
                        head=NULL;
                        cout<<"cnode "<<t->data<<" Deletion is sucess";
                        delete(t);
                }
                else
                {
                 //code to find last node
                        while(t->link!=head)
                        {
                                t=t->link;
                        }
                        temp=head;
                        t->link=head->link;     //linking last and first node
                cout<<"cnode "<<temp->data<<" Deletion is sucess";
                head=head->link;
                delete(temp);
                }
        }
}
//Code to count nodes in the circular linked list
template <class T>
void clist<T>::cnode_count()
{
struct cnode<T>*t;
int c=0;
        t=head;
         if(head==NULL)
                {
                cout<<"circular list is Empty\n";

                }
```

32

```
        else
        {       t=t->link;
                c++;
                while(t!=head)
                {       c++;
                        t=t->link;
                }
        cout<<"Node Count="<<c;

        }

}
int main()
{
int ch,pos;
clist <int> L;
        while(1)
        {
                cout<<"\n ***Operations on Circular Linked clist***"<<endl;
                cout<<"\n1.Insert cnode at End"<<endl;
                cout<<"2.Insert Cnode at Front"<<endl;
                cout<<"3.Delete Cnode at END"<<endl;
                cout<<"4.Delete Cnode at Front"<<endl;
                cout<<"5.Display Nodes "<<endl;
                cout<<"6.Cnode Count"<<endl;
                cout<<"7.Exit "<<endl;
                cout<<"8.Clear Screen "<<endl;
                cout<<"Enter Your choice:";
                cin>>ch;
                switch(ch)
                {
                        case 1: L.insert_end();
                                        break;
                        case 2: L.insert_front();
                                        break;
                        case 3:L.delete_end();
                                        break;
                        case 4:L.delete_front();
                                        break;
                        case 5://display contents
                                        L.display();
                                        break;
                        case 6: L.cnode_count();
                                        break;
                        case 7:exit(0);
                        case 8:system("cls");
                                        break;
                        default:cout<<"Invalid choice";
                }
        }
}
```

STACK ADT:- A Stack is a linear data structure where insertion and deletion of items takes place at one end called top of the stack. A Stack is defined as a data structure which operates on a last-in first-out basis. So it is also is referred as Last-in First-out(LIFO).

Stack uses a single index or pointer to keep track of the information in the stack. The basic operations associated with the stack are:

a) push(insert) an item onto the stack.
b) pop(remove) an item from the stack.

The general terminology associated with the stack is as follows:

A stack pointer keeps track of the current position on the stack. When an element is placed on the stack, it is said to be **pushed** on the stack. When an object is removed from the stack, it is said to be **popped** off the stack. Two additional terms almost always used with stacks are **overflow**, which occurs when we try to push more information on a stack that it can hold, and **underflow**, which occurs when we try to pop an item off a stack which is empty.

Pushing items onto the stack:
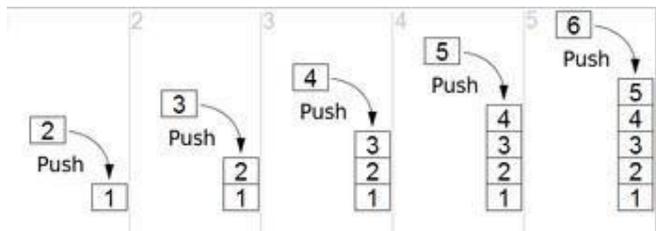
Assume that the array elements begin at 0 (because the array subscript starts from 0)
and the maximum elements that can be placed in stack is max. The stack pointer, **top**, is considered to be pointing to the top element of the stack. A push operation thus involves adjusting the stack pointer to point to next free slot and then copying data into that slot of the stack. Initially the top is initialized to -1.

```
//code to push an element on to stack;
template<class T>
void stack<T>::push()
{
        if(top==max-1)
                cout<<"Stack Overflow...\n";
        else
        {
                cout<<"Enter an element to be pushed:";
                top++;
                cin>>data;
                stk[top]=data;
                cout<<"Pushed Sucesfully....\n";
        }
}
```

Popping an element from stack:

To remove an item, first extract the data from top position in the stack and then decrement the stack pointer, top.
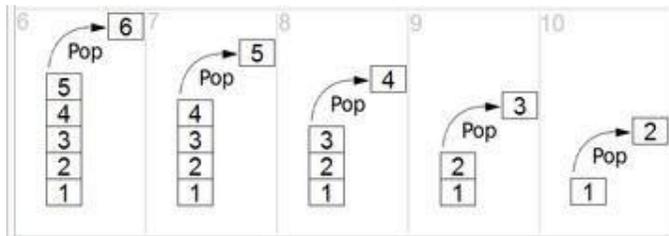

```
//code to remove an element from stack
template<class T>
void stack<T>::pop()
{
        if(top= =-1)
                cout<<"Stack is Underflow";
        else
        {
                data=stk[top];
                top--;
                cout<<data<<" is poped Sucesfully ....\n";
        }
}
```

Static implementation of Stack ADT

```
#include<stdlib.h>
#include<iostream.h>
#define max 4
template<class T>
class stack
{
private:
        int top;
        T stk[max],data;
public:
        stack();
        void push();
        void pop();
        void display();
};
template<class T>
stack<T>::stack()
{
        top=-1;
```

```cpp
}
//code to push an element on to stack;
template<class T>
void stack<T>::push()
{
        if(top==max-1)
                cout<<"Stack Overflow...\n";
        else
        {
                cout<<"Enter an element to be pushed:";
                top++;
                cin>>data;
                stk[top]=data;
                cout<<"Pushed Sucesfully....\n";
        }
}
//code to remove an element from stack
template<class T>
void stack<T>::pop()
{
        if(top==-1)
                cout<<"Stack is Underflow";
        else
        {
                data=stk[top];
                top--;
                cout<<data<<" is poped Sucesfully ....\n";
        }
}
//code to display stack elements
template<class T>
void stack<T>::display()
{
        if(top==-1)
                cout<<"Stack Under Flow";
        else
        {       cout<<"Elements in the Stack are....\n";
                for(int i=top;i>-1;i--)
                {
                        cout<<<<stk[i]<<"\n";
                }
        }
}
int main()
{
int choice;
stack <int>st;
        while(1)
        {
                cout<<"\n*****Menu for Stack operations*****\n";
                cout<<"1.PUSH\n2.POP\n3.DISPLAY\n4.EXIT\n";
```

```
                        cout<<"Enter Choice:";
                        cin>>choice;
                        switch(choice)
                        {
                                case 1: st.push();
                                        break;
                                case 2: st.pop();
                                        break;
                                case 3: st.display();
                                        break;
                                case 4: exit(0);
                                default:cout<<"Invalid choice...Try again...\n";
                        }
                }
}
output:
*****Menu for Stack operations*****
1.PUSH
2.POP
3.DISPLAY
4.EXIT
Enter Choice:1
Enter an element to be pushed:11
Pushed Sucesfully....
*****Menu for Stack operations*****
1.PUSH
2.POP
3.DISPLAY
4.EXIT
Enter Choice:1
Enter an element to be pushed:22
Pushed Sucesfully....

*****Menu for Stack operations*****
1.PUSH
2.POP
3.DISPLAY
4.EXIT
Enter Choice:1
Enter an element to be pushed:44
Pushed Sucesfully....

*****Menu for Stack operations*****
1.PUSH
2.POP
3.DISPLAY
4.EXIT
Enter Choice:1
Enter Choice:1
Enter an item to be pushed:55
Pushed Sucesfully....
```

```
*****Menu for Stack operations*****
1.PUSH
2.POP
3.DISPLAY
4.EXIT
Enter Choice:1
Stack Overflow...

*****Menu for Stack operations*****
1.PUSH
2.POP
3.DISPLAY
4.EXIT
Enter Choice:2
55 is poped Sucesfully....

*****Menu for Stack operations*****
1.PUSH
2.POP
3.DISPLAY
4.EXIT
Enter Choice:3
Elements in the Stack are....
44
22
11

*****Menu for Stack operations*****
1.PUSH
2.POP
3.DISPLAY
4.EXIT
Enter Choice:4
```

Dynamic implementation of Stack ADT

```cpp
#include<iostream.h>
template <class T>
struct node
{
T data;
struct node<T> *link;
};
template <class T>
class stack
{
        int data;
        struct node<T>*top;
```

```
        public:
        stack()
        {
                top=NULL;
        }
        void display();
        void push();
        void pop();
};
template <class T>
void stack<T>::display()
{
        struct node<T>*t;

        if(top==NULL)
         {
                        cout<<"stack is Empty\n";
          }
         else
        {       t=top;
                while(t!=NULL)
                {       cout<<"|"<<t->data<<"|"<<endl;
                        t=t->link;
                }
        }
}

template <class T>
void stack<T>::push()
{
struct node <T>*t,*temp;
        cout<<"Enter data into node:";
        cin>>data;
        temp=new struct node<T>;
        temp->data=data;
        temp->link=NULL;
        if(top==NULL)
                top=temp;
        else
        {       temp->link=top;
                top=temp;
        }
}


template <class T>
void stack<T>::pop()
{
struct node<T>*t;
        if(top==NULL)
                cout<<"stack is Empty\n";
```

```
            else
            {       t=top;
                    top=top->link;
                    cout<<"node "<<t->data<<" Deletion is sucess";
                    delete(t);
            }
}

int main()
{
int ch;
stack <int> st;
        while(1)
        {
                cout<<"\n ***Operations on Dynamic stack***"<<endl;
                cout<<"\n1.PUSH"<<endl;
                cout<<"2.POP"<<endl;
                cout<<"3.Display "<<endl;
                cout<<"4.Exit "<<endl;
                cout<<"Enter Your choice:";
                cin>>ch;
                switch(ch)
                {
                        case 1: st.push();
                                        break;
                        case 2: st.pop();
                                        break;
                        case 3:st.display();;
                                        break;
                        case 4:exit(0);
                        default:cout<<"Invalid choice";
                }
        }
}
```

Applications of Stack:
1. Stacks are used in conversion of infix to postfix expression.
2. Stacks are also used in evaluation of postfix expression.
3. Stacks are used to implement recursive procedures.
4. Stacks are used in compilers.
5. Reverse String

An arithmetic expression can be written in three different but equivalent notations, i.e., without changing the essence or output of an expression. These notations are−
   1. Infix Notation
   2. Prefix (Polish) Notation
   3. Postfix (Reverse-Polish) Notation

Expression	Example	Note
Infix	a + b	Operator Between Operands
Prefix	+ a b	Operator before Operands
Postfix	a b +	Operator after Operands

Conversion of Infix Expressions to Prefix and Postfix

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

Convert following infix expression to prefix and postfix
(A + B) * C - (D - E) * (F + G)

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

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

Prefix   Postfix

- * + A B C * - D E + F G   A B + C * D E - F G + * -
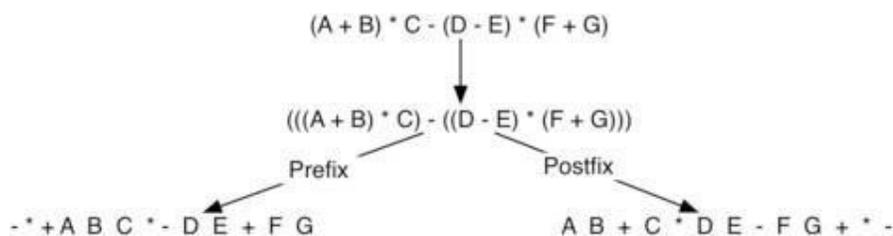
The Tower of Hanoi (also called the Tower of Brahma or Lucas' Tower,[1] and sometimes pluralized) is a mathematical game or puzzle. It consists of three rods, and a number of disks of different sizes which can slide onto any rod. The puzzle starts with the disks in a neat stack in ascending order of size on one rod, the smallest at the top, thus making a conical shape.

The objective of the puzzle is to move the entire stack to another rod, obeying the following simple rules:

1. Only one disk can be moved at a time.
2. Each move consists of taking the upper disk from one of the stacks and placing it on top of another stack i.e. a disk can only be moved if it is the uppermost disk on a stack.
3. No disk may be placed on top of a smaller disk.
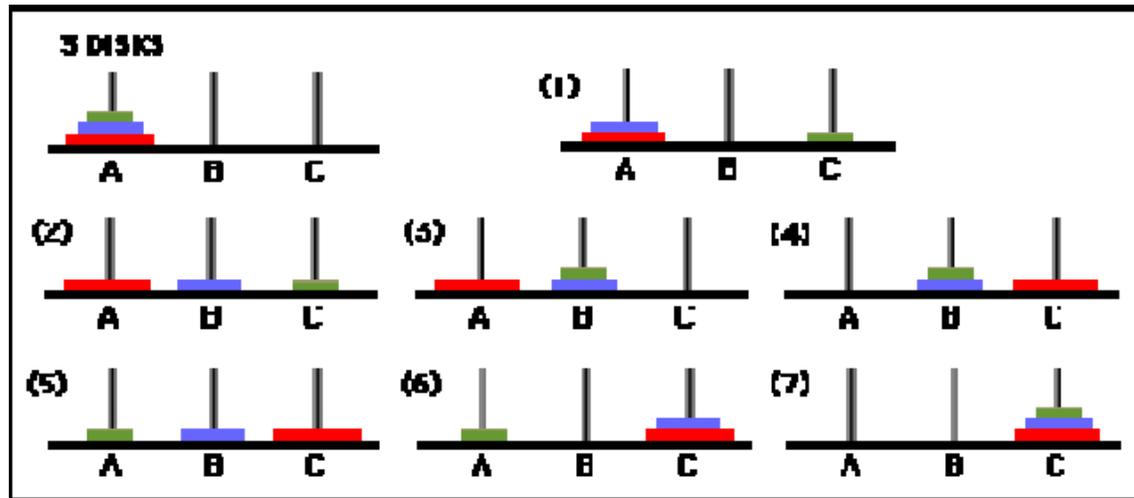
QUEUE ADT

A queue is an ordered collection of data such that the data is inserted at one end and deleted from another end. The key difference when compared stacks is that in a queue the information stored is processed first-in first-out or FIFO. In other words the information receive from a queue comes in the same order that it was placed on the queue.
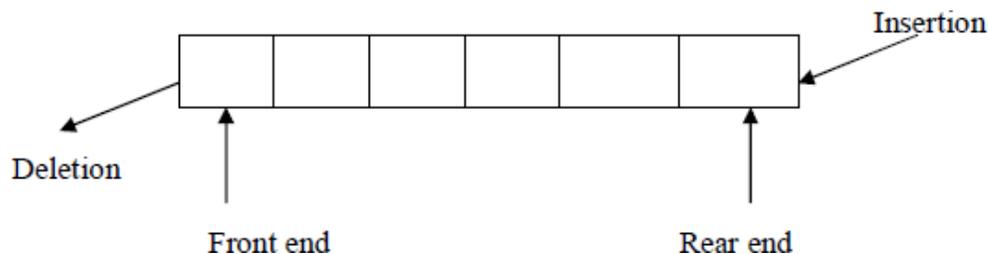
Representing a Queue:

One of the most common way to implement a queue is using array. An easy way to do so is to define an array Queue, and two additional variables front and rear. The rules for manipulating these variables are simple:

- ➤ Each time information is added to the queue, increment rear.
- ➤ Each time information is taken from the queue, increment front.
- ➤ Whenever **front >rear or front=rear=-1** the queue is empty.

Array implementation of a Queue do have drawbacks. The maximum queue size has to be set at compile time, rather than at run time. Space can be wasted, if we do not use the full capacity of the array.

Operations on Queue:
A queue have two basic operations:
a) adding new item to the queue
b) removing items from queue.
The operation of adding new item on the queue occurs only at one end of the queue called the **rea**r or back.
 The operation of removing items of the queue occurs at the other end called the **front.**

For insertion and deletion of an element from a queue, the array elements begin at 0 and the maximum elements of the array is **maxSize.** The variable front will hold the index of the item that is considered the front of the queue, while the rear variable will hold the index of the last item in the queue.
Assume that initially the front and rear variables are initialized to -1. Like stacks, underflow and overflow conditions are to be checked before operations in a queue.

Queue empty or underflow condition is

```
if((front>rear)||front= =-1)
    cout<"Queue is empty";
```

Queue Full or overflow condition is

```
if((rear==max)
    cout<"Queue is full";
```

Static implementation of Queue ADT

```
#include<stdlib.h>
#include<iostream.h>
#define max 4
template <class T>
class queue
{
            T  q[max],item;
              int front,rear;
public:        queue();
              void   insert_q();
              void   delete_q();
              voiddisplay_q();
};
template <class T>
queue<T>::queue()
{
            front=rear=-1;
}
//code to insert an item into queue;
template <class T>
void queue<T> ::insert_q()
```

```cpp
        {
                if(front>rear)
                        front=rear=-1;
                if(rear==max-1)
                    cout<<"queue Overflow...\n";
                else
                {
                    if(front==-1)
                                front=0;
                    rear++;
                    cout<<"Enter an item to be inserted:";
                    cin>>item;
                    q[rear]=item;
                    cout<<"inserted Sucesfully..into queue..\n";
                }
        }
        template <class T>
        void queue<T>::delete_q()
        {
                if((front==-1&&rear==-1)||front>rear)
                {
                    front=rear=-1;
                    cout<<"queue is Empty .. \n";
                }
                else
                {
                    item=q[front];
                    front++;
                    cout<<item<<" is deleted Sucesfully ...\n";
                }
        }
        template <class T>
        void queue<T>::display_q()
        {
                if((front==-1&&rear==-1)||front>rear)
                {
                    front=rear=-1;
                    cout<<"queue is Empty .. \n";
                }
                else
                {
                    for(int i=front;i<=rear;i++)
                            cout<<"|"<<q[i]<<"|<--";
                }
        }
        int main()
        {
        int choice;
        queue<int> q;
                while(1)
                {
                cout<<"\n\n*****Menu for operations on QUEUE*****\n\n";
                cout<<"1.INSERT\n2.DELETE\n3.DISPLAY\n4.EXIT\n";
```

44

```
                    cout<<"Enter Choice:";
                    cin>>choice;
                       switch(choice)
                       {
                               case 1: q.insert_q();
                                               break;
                               case 2: q.delete_q();
                                               break;
                               case 3: cout<<"Elements in the queue are... \n";
                                               q.display_q();
                                               break;
                               case 4: exit(0);
                               default: cout<<"Invalid choice...Try again...\n";
                       }
                    }
}
```

Dynamic implementation of Queue ADT

```
#include<stdlib.h>
#include<iostream.h>
template <class T>
struct node
{
            T data;
struct node<T>*next;
};
template <class T>
class queue
{
private:
            T item;
            node<T> *front,*rear;
public:
                queue();
                void   insert_q();
                void   delete_q();
                voiddisplay_q();
};
template <class T>
queue<T>::queue()
{
            front=rear=NULL;
}
//code to insert an item into queue;
template <class T>
void queue<T>::insert_q()
{
node<T>*p;
            cout<<"Enter an element to be inserted:";
```

45

```cpp
            cin>>item;
            p=new node<T>;
            p->data=item;
            p->next=NULL;
            if(front==NULL)
            {
                rear=front=p;
            }
            else
            {
                rear->next=p;
                rear=p;
            }
            cout<<"\nInserted into Queue Sucesfully... \n";
}
//code to delete an elementfrom queue
template <class T>
void queue<T>::delete_q()
{
node<T>*t;
            if(front==NULL)
                cout<<"\nQueue is Underflow";
            else
            {
                item=front->data;
                t=front;
                front=front->next;
                cout<<"\n"<<item<<" is deletedfrom Queue... \n";
            }
            delete(t);
}
//code to display elements in queue
template <class T>
void queue<T>::display_q()
{
node<T>*t;
            if(front==NULL)
                cout<<"\nQueue Under Flow";
            else
            {
                cout<<"\nElements in the Queue are... \n";
                t=front;
                while(t!=NULL)
                {
                        cout<<"|"<<t->data<<"|<-";
                        t=t->next;
                }
            }
}
int main()
{
int choice;
queue<int>q1;
```

```
        while(1)
        {
        cout<<"\n\n***Menu for operations on Queue***\n\n";
        cout<<"1.Insert\n2.Delete\n3.DISPLAY\n4.EXIT\n";
        cout<<"Enter Choice:";
        cin>>choice;
        switch(choice)
        {
            case 1: q1.insert_q();
                            break;
            case 2: q1.delete_q();
                            break;
            case 3: q1.display_q();
                            break;
            case 4: exit(0);
            default: cout<<"Invalid choice...Try again...\n";
        }
        }
}
```

Application of Queue:

Queue, as the name suggests is used whenever we need to have any group of objects in an order in which the first one coming in, also gets out first while the others wait for there turn, like in the following scenarios :

1. Serving requests on a single shared resource, like a printer, CPU task scheduling etc.

2. In real life, Call Center phone systems will use Queues, to hold people calling them in an order, until a service representative is free.

3. Handling of interrupts in real-time systems. The interrupts are handled in the same order as they arrive, First come first served.

CIRCULAR QUEUE

Once the queue gets filled up, no more elements can be added to it even if any element is removed from it consequently. This is because during deletion, rear pointer is not adjusted.
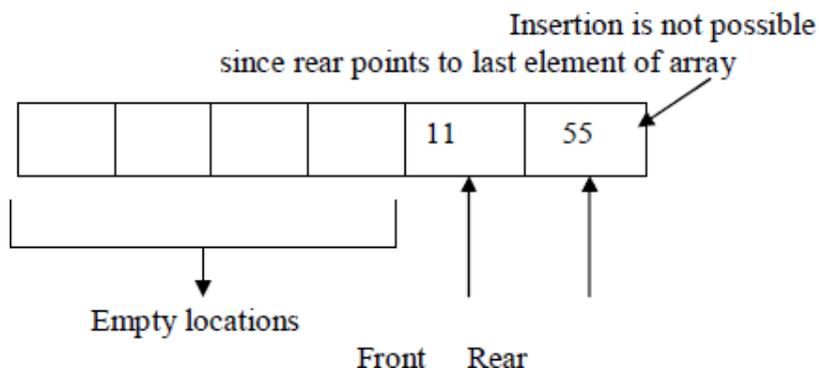
When the queue contains very few items and the rear pointer points to last element. i.e. rear=maxSize-1, we cannot insert any more items into queue because the overflow condition satisfies. That means a lot of space is wasted

    .Frequent reshuffling of elements is time consuming. One solution to this is arranging all elements in a circular fashion. Such structures are often referred to as **Circular Queues.**

A circular queue is a queue in which all locations are treated as circular such that the first location CQ[0] follows the last location CQ[max-1].

Circular Queue empty or underflow condition is

```
if(front==-1)
            cout<<"Queue is empty";
```

Circular Queue Full or overflow condition is

```
if(front==(rear+1)%max)
{
            cout<<"Circular Queue is full\n";
}
```

Example: Consider the following circular queue with N = 5.

1. Initially, Rear = 0, Front = 0.
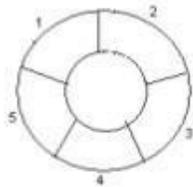
4. Insert 20, Rear = 3, Front = 0.
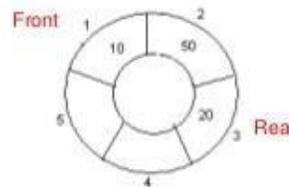
2. Insert 10, Rear = 1, Front = 1.
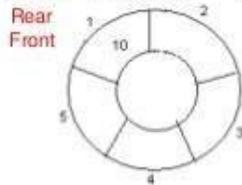
5. Insert 70, Rear = 4, Front = 1.
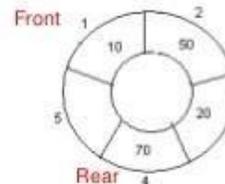
3. Insert 50, Rear = 2, Front = 1.
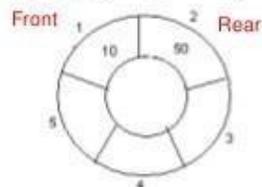
6. Delete front, Rear = 4, Front = 2.
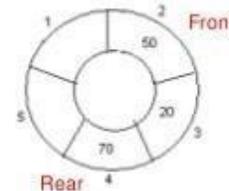
Insertion into a Circular Queue:

Algorithm CQueueInsertion(Q,maxSize,Front,Rear,item)

Step 1: If Rear = maxSize-1 then

            Rear = 0

        else

            Rear=Rear+1

Step 2: If Front = Rear then

        print "Queue Overflow"

        Return

Step 3: Q[Rear] = item

Step 4: If Front = 0 then
        Front = 1
Step 5: Return

Deletion from Circular Queue:

Algorithm   CQueueDeletion(Q,maxSize,Front,Rear,item)
Step 1: If Front = 0 then
        print "Queue Underflow"
        Return
Step 2: K=Q[Front]
Step 3: If Front = Rear then
        begin
                Front = -1
                Rear = -1
        end
        else
        If Front = maxSize-1 then
                Front = 0
        else
                Front = Front + 1
Step 4: Return K

        printArray(list, n);

Dictionaries:- linear list representation, skip list representation, operations insertion, deletion and searching, hash table representation, hash functions, collision resolution-separate chaining, open addressing-linear probing, quadratic probing, double hashing, rehashing, extendible hashing.

DICTIONARIES:

Dictionary is a collection of pairs of key and value where every value is associated with the corresponding key.

Basic operations that can be performed on dictionary are:
1. Insertion of value in the dictionary
2. Deletion of particular value from dictionary
3. Searching of a specific value with the help of key

Linear List Representation

The dictionary can be represented as a linear list. The linear list is a collection of pair and value.
There are two method of representing linear list.
1. Sorted Array- An array data structure is used to implement the dictionary.
2. Sorted Chain- A linked list data structure is used to implement the dictionary

Structure of linear list for dictionary:

class dictionary
{

private:
　　　　 int k,data;
　　　　struct node
　　　　{
　　　　public: int key;
　　　　int value;
　　　　struct node *next;
　　　　} *head;

public:
　　　　 dictionary();
　　　　void insert_d( );
　　　　void delete_d( );
　　　　void display_d( );
　　　　void length();
};

Insertion of new node in the dictionary:
Consider that initially dictionary is empty then
head = NULL
We will create a new node with some key and value contained in it.

New

| 1 | 10 | NULL |

Now as head is NULL, this new node becomes head. Hence the dictionary contains only one record. this node will be 'curr' and 'prev' as well. The 'cuur' node will always point to current visiting node and 'prev' will always point to the node previous to 'curr' node. As now there is only one node in the list mark as 'curr' node as 'prev' node.

New/head/curr/prev

| 1 | 10 | NULL |
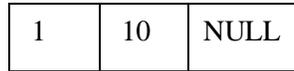
Insert a record, key=4 and value=20,

New

| 4 | 20 | NULL |
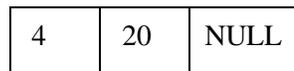
Compare the key value of 'curr' and 'New' node. If New->key > Curr->key then attach New node to 'curr' node.

prev/head   New   curr->next=New

                                                    prev=curr

| 1 | 10 | → | | 4 | 20 | NULL |

Add a new node <7,80> then

head/prev   curr   New

| 1 | 10 | → | | 4 | 20 | → | | 7 | 80 | NULL |
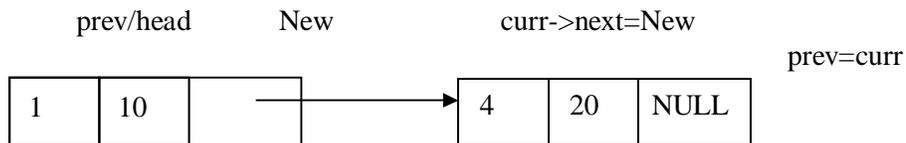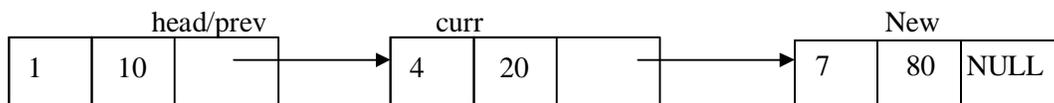
If we insert <3,15> then we have to search for it proper position by comparing key value.

(curr->key < New->key) is false. Hence else part will get executed.

| 1 | 10 | | | 4 | 20 | → | | 7 | 80 | NULL |

| 3 | 15 | |
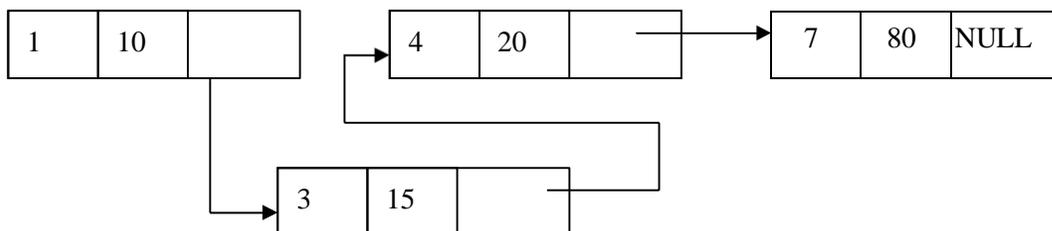
```
void dictionary::insert_d( )
{
node *p,*curr,*prev;
        cout<<"Enter an key and value to be inserted:";
        cin>>k;
        cin>>data;
```

```
 p=new node;
 p->key=k;
p->value=data;
p->next=NULL;
 if(head==NULL)
         head=p;
 else
 {
      curr=head;
       while((curr->key<p->key)&&(curr->next!=NULL))
       {
              prev=curr;
              curr=curr->next;
        }
       if(curr->next==NULL)
       {
              if(curr->key<p->key)
               {
                      curr->next=p;
                       prev=curr;
               }
              else
              {
                      p->next=prev->next;
                       prev->next=p;
              }
        }
        else
       {
              p->next=prev->next;
              prev->next=p;
       }
       cout<<"\nInserted  into dictionary Sucesfully.... \n";
 }
}
```

The delete operation:

Case 1: Initially assign 'head' node as 'curr' node.Then ask for a key value of the node which   is to be deleted. Then starting from head node key value of each jode is cked and compared with the desired node's key value. We will get node which is to be deleted in variable 'curr'. The   node given by variable 'prev' keeps track of previous node of 'cuu' node. For eg, delete node with key value 4 then

53
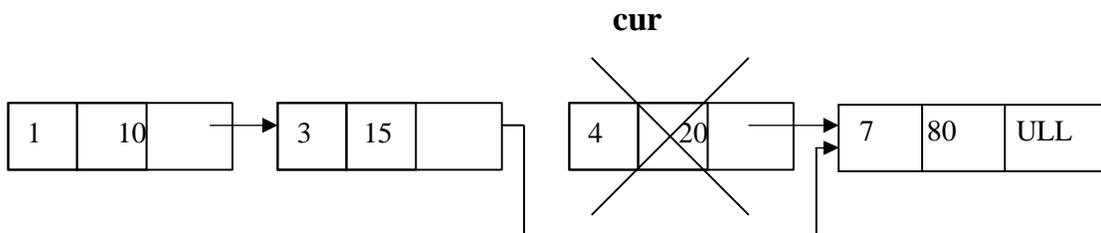
Case 2:

If the node to be deleted is head node
i.e.. if(curr==head)

Then, simply make 'head' node as next node and delete 'curr'
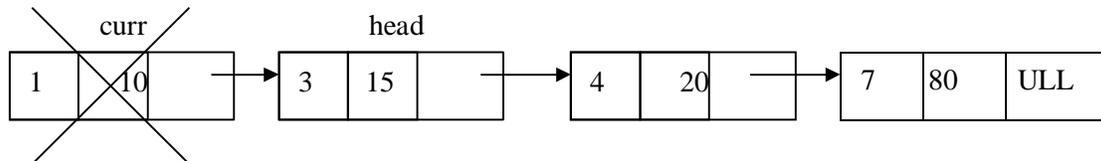
Hence the list becomes
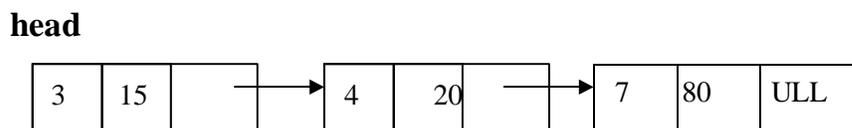

```cpp
void dictionary::delete_d( )
{
node*curr,*prev;
        cout<<"Enter key value that you want to delete...";
        cin>>k;
        if(head==NULL)
            cout<<"\ndictionary is Underflow";
        else
          {    curr=head;
               while(curr!=NULL)
               {
               if(curr->key==k)
                    break;
               prev=curr;
               curr=curr->next;
               }
          }
        if(curr==NULL)
            cout<<"Node not found...";
        else
          {
              if(curr==head)
```

```
            head=curr->next;
        else
            prev->next=curr->next;
    delete curr;
    cout<<"Item deleted from dictionary...";
    }
}
```

The length operation:
int dictionary**::length()**
```
{
struct node *curr;
int count;
    count=0;
    curr=head;
    if(curr==NULL)
    {
        cout<<"The list is empty";
        return 0;
    }
    while(curr!=NULL)
    {
        count++;
        cur=curr->next;
    }
return count;
}
```

SKIP LIST REPRESENTATION

Skip list is a variant list for the linked list. Skip lists are made up of a series of nodes connected one after the other. Each node contains a key and value pair as well as one or more references, or pointers, to nodes further along in the list. The number of references each node contains is determined randomly. This gives skip lists their probabilistic nature, and the number of references a node contains is called its node level.

There are two special nodes in the skip list one is head node which is the starting node of the list and tail node is the last node of the list
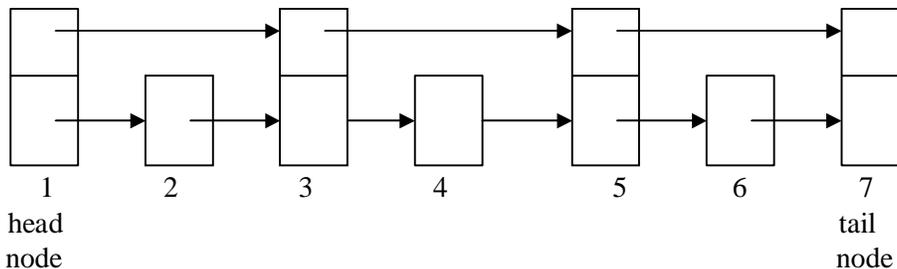
The skip list is an efficient implementation of dictionary using sorted chain. This is because in skip list each node consists of forward references of more than one node at a time.

Eg:
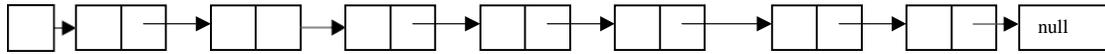
Now to search any node from above given sorted chain we have to search the sorted chain from head node by visiting each node. But this searching time can be reduced if we add one level in every alternate node. This extra level contains the forward pointer of some node. That means in sorted chain come nodes can holds pointers to more than one node.
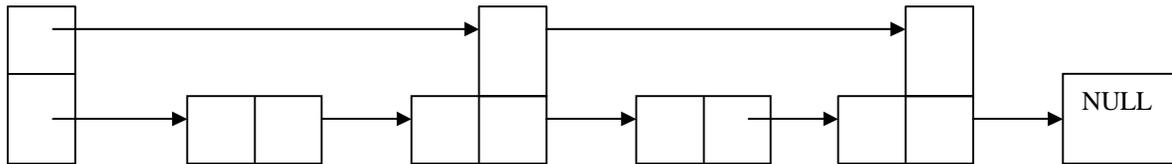
If we want to search node 40 from above chain there we will require comparatively less time. This search again can be made efficient if we add few more pointers forward references.
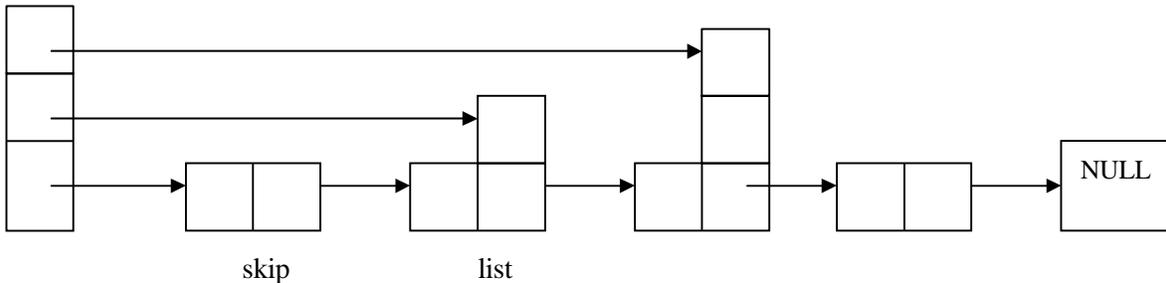
skip   list

Node structure of skip list:

```
template <class K, class E>
struct skipnode
{
        typedef pair<const K,E> pair_type;
        pair_type element;
        skipnode<K,E> **next;
        skipnode(const pair_type &New_pair, int MAX):element(New_pair)
        {
                next=new skipnode<K,E>*[MAX];
        }
};
```

The individual node looks like this:

Key	value	array of pointer
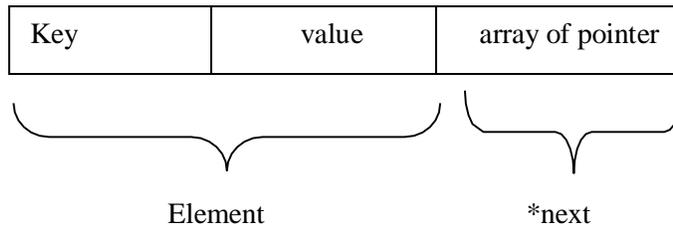
Element   *next

Searching:

The desired node is searched with the help of a key value.

```
template<class K, class E>
skipnode<K,E>* skipLst<K,E>::search(K& Key_val)
{
skipnode<K,E>* Forward_Node = header;
for(int i=level;i>=0;i--)
{
while (Forward_Node->next[i]->element.key < key_val)
Forward_Node = Forward_Node->next[i];
last[i] = Forward_Node;
}
return Forward_Node->next[0];
}
```

Searching for a key within a skip list begins with starting at header at the overall list level and moving forward in the list comparing node keys to the key_val. If the node key is less than the key_val, the search continues moving forward at the same level. If o the other hand, the node key is equal to or greater than the key_val, the search drops one level and continues forward. This process continues until the desired key_val has been found if it is present in the skip list. If it is not, the search will either continue at the end of the list or until the first key with a value greater than the search key is found.

Insertion:

There are two tasks that should be done before insertion operation:

1. Before insertion of any node the place for this new node in the skip list is searched. Hence before any insertion to take place the search routine executes. The last[] array in the search routine is used to keep track of the references to the nodes where the search, drops down one level.
2. The level for the new node is retrieved by the routine randomelevel()

```
template<class K,class E>
void skipLst<K,E>::insert(pair<K,E>& New_pair)
{
if(New_pair.key >= tailkey)
{
cout<<"Key is too large";
}

skipNode<K,E>* temp = search(New_pair.key);
if(temp->element.key == New_pair.key)
```

```
{
temp->element.value=New_pair.value;
return;
}

if*New_Level > levels)
{
New_Level = ++levels;
last[New_Level] = header;
}

skipNode<K,E> *newNode = new skipNode<K,E>(New_pair, New_Level+1);

for(int i=0;i<=New_Level;i++)
{
newNode->next[i] = last[i]->next[i];
last[i]->next[i] = newNode;
}
len++;
return;
}
```

Determining the level of each node:

```
template <class K, class E>
int skipLst<K,E>::randomlevel()
{
int lvl=0;
while(rand() <= Lvl_No)
lvl=lvl+1;
if(lvl<=MaxLvl)
return lvl;
else
return MaxLvl;
}
```

Deletion:
First of all, the deletion makes use of search algorithm and searches the node that is to be deleted.
If the key to be deleted is found, the node containing the key is removed.

```
template<class K, class E>
void skipLst<K,E>::delet(K& Key_val)
{
if(key_val>=tailKey)
return;
skipNode<K,E>* temp = search(Key_val);
if(temp->elemnt.key != Key_val)
return;

for(int i=0;i<=levels;i++)
```

```
{
if(last[i]->next[i] == temp)
last[i]=>next[i] = temp->next[i];
}

while(level>0 && header->next[level] == tail)
levels--;
delete temp;
len--;
}
```

HASH TABLE REPRESENTATION

> ➤ Hash table is a data structure used for storing and retrieving data very quickly. Insertion of data in the hash table is based on the key value. Hence every entry in the hash table is associated with some key.
> ➤ Using the hash key the required piece of data can be searched in the hash table by few or more key comparisons. The searching time is then dependent upon the size of   the hash table.
> ➤ The effective representation of dictionary can be done using hash table. We can place the dictionary entries in the hash table using hash function.

HASH FUNCTION

> ➤ Hash function is a function which is used to put the data in the hash table.   Hence one can use the same hash function to retrieve the data from the hash table. Thus hash function is used to implement the hash table.
> ➤ The integer returned by the hash function is called hash key.

For example: Consider that we want place some employee records in the hash table The record of employee is placed with the help of key: employee ID. The employee ID is a 7 digit number for placing the record in the hash table.   To place the record 7 digit number is converted into 3 digits by taking only last three digits of the key.

If the key is 496700 it can be stored at 0th position. The second key 8421002, the record of those key is placed at 2nd position in the array.
Hence the hash function will be- H(key) = key%1000
Where key%1000 is a hash function and key obtained by hash function is called hash key.

> ➤ **Bucket and Home bucket:** The hash function H(key) is used to map several dictionary entries in the hash table. Each position of the hash table is called bucket.

The function H(key) is home bucket for the dictionary with pair whose value is key.

TYPES OF HASH FUNCTION
There are various types of hash functions that are used to place the record in the hash table-

1. Division Method: The hash function depends upon the remainder of division.
Typically the divisor is table length.
For eg; If the record 54, 72, 89, 37 is placed in the hash table and if the table size is 10 then

h(key) = record % table size

54%10=4
72%10=2
89%10=9
37%10=7

0	
1	
2	72
3	
4	54
5	
6	
7	37
8	
9	89

2. Mid Square:

In the mid square method, the key is squared and the middle or mid part of the result is used as the index. If the key is a string, it has to be preprocessed to produce a number.
Consider that if we want to place a record 3111 then

$3111^2 = 9678321$
for the hash table of size 1000
H(3111) = 783 (the middle 3 digits)

3. Multiplicative hash function:

The given record is multiplied by some constant value. The formula for computing the hash key is-

H(key) = floor(p *(fractional part of key*A)) where p is integer constant and A is constant real number.

Donald Knuth suggested to use constant A = 0.61803398987

If key 107 and p=50 then

 H(key) = floor(50*(107*0.61803398987))
        = floor(3306.4818458045)
        = 3306
 At 3306 location in the hash table the record 107 will be placed.

4. Digit Folding:

The key is divided into separate parts and using some simple operation these parts are combined to produce the hash key.
For eg; consider a record 12365412 then it is divided into separate parts as 123 654 12 and these are added together

H(key) = 123+654+12
       = 789
The record will be placed at location 789

5. Digit Analysis:

The digit analysis is used in a situation when all the identifiers are known in advance. We first transform the identifiers into numbers using some radix, r. Then examine the digits of each identifier. Some digits having most skewed distributions are deleted. This deleting of digits is continued until the number of remaining digits is small enough to give an address in the range of the hash table. Then these digits are used to calculate the hash address.

UNIT -2

COLLISION

the hash function is a function that returns the key value using which the record can be placed in the hash table. Thus this function helps us in placing the record in the hash table at appropriate position and due to this we can retrieve the record directly from that location. This   function need to be designed very carefully and it should not return the same hash key address for two different records. This is an undesirable situation in hashing.

Definition: The situation in which the hash function returns the same hash key (home bucket) for more than one record is called collision and two same hash keys returned for different records is called synonym.

Similarly when there is no room for a new pair in the hash table then such a situation is called overflow. Sometimes when we handle collision it may lead to overflow conditions. Collision and overflow show the poor hash functions.

For example,

Consider a hash function.

H(key) = recordkey%10 having the hash table size of 10

The record keys to be placed are

131, 44, 43, 78, 19, 36, 57 and 77
131%10=1
44%10=4
43%10=3
78%10=8
19%10=9
36%10=6
57%10=7
77%10=7

0	
1	131
2	
3	43
4	44
5	
6	36
7	57
8	78
9	19

Now if we try to place 77 in the hash table then we get the hash key to be 7 and at index 7 already the record key 57 is placed. This situation is called **collision**. From the index 7 if we look for next vacant position at subsequent indices 8.9 then we find that there is no room to place 77 in the hash table. This situation is called **overflow**.

COLLISION RESOLUTION TECHNIQUES
If collision occurs then it should be handled by applying some techniques. Such a technique is called collision handling technique.
      1.Chaining
      2.Open addressing (linear probing)
      3.Quadratic probing
      4.Double hashing
      5. Double hashing
      6.Rehashing

CHAINING

In collision handling method chaining is a concept which introduces an additional field with data i.e. chain. A separate chain table is maintained for colliding data. When collision occurs then a linked list(chain) is maintained at the home bucket.

For eg;

Consider the keys to be placed in their home buckets are
131, 3, 4, 21, 61, 7, 97, 8, 9

then we will apply a hash function as H(key) = key % D

Where D is the size of table. The hash table will be-

Here D = 10
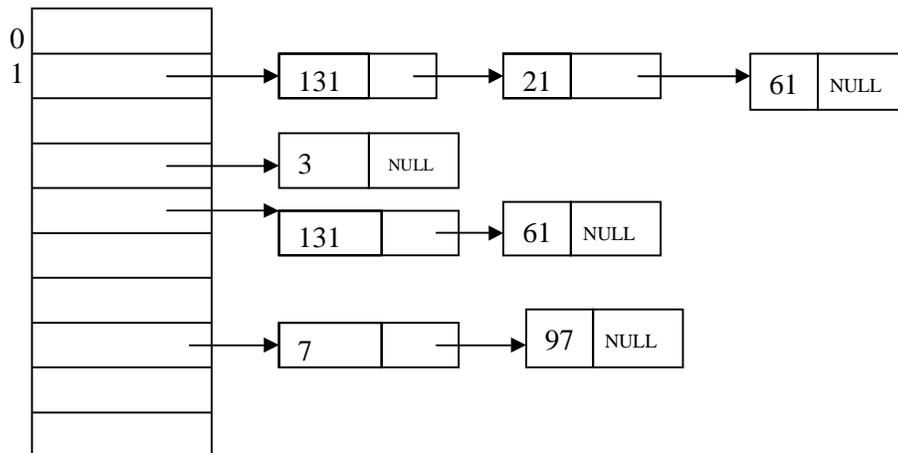
A chain is maintained for colliding elements. for instance 131 has a home bucket (key) 1. similarly key 21 and 61 demand for home bucket 1. Hence a chain is maintained at index 1.

OPEN ADDRESSING – LINEAR PROBING

This is the easiest method of handling collision. When collision occurs i.e. when two records demand for the same home bucket in the hash table then collision can be solved by placing the second record linearly down whenever the empty bucket is found. When use linear probing (open addressing), the hash table is represented as a one-dimensional array with indices that range from 0 to the desired table size-1. Before inserting any elements into this table, we must initialize the table to represent the situation where all slots are empty. This allows us to detect overflows and collisions when we inset elements into the table. Then using some suitable hash function the element can be inserted into the hash table.

For example:

Consider that following keys are to be inserted in the hash table

131, 4, 8, 7, 21, 5, 31, 61, 9, 29

Initially, we will put the following keys in the hash table.
We will use Division hash function. That means the keys are placed using the formula

H(key) = key % tablesize
H(key) = key % 10

For instance the element 131 can be placed at

H(key) = 131 % 10
          = 1

Index 1 will be the home bucket for 131. Continuing in this fashion we will place 4, 8, 7.

Now the next key to be inserted is 21. According to the hash function

H(key)=21%10
H(key) = 1

But the index 1 location is already occupied by 131 i.e.   collision occurs. To resolve this collision we will linearly move down and at the next empty location we   will prob   the element.   Therefore 21 will be placed at the index 2. If the next element is 5 then we   get   the home bucket for 5 as index 5 and this bucket is empty so we will put the element 5 at index 5.

Index	Key		Key		Key
0	NULL		NULL		NULL
1	131		131		131
2	NULL		21		21
3	NULL		NULL		31
4	4		4		4
5	NULL		5		5
6	NULL		NULL		61
7	7		7		7
8	8		8		8
9	NULL		NULL		NULL

after placing keys 31, 61

The next record key is 9. According to decision hash function it demands for the home bucket 9. Hence we will place 9 at index 9. Now the next final record key 29 and it hashes a key 9. But home bucket 9 is already occupied. And there is no next empty bucket as the table size is limited to index 9. The overflow occurs. To handle it we move back to bucket 0 and is the location over there is empty 29 will be placed at 0th index.

Problem with linear probing:

One major problem with linear probing is primary clustering. Primary clustering is a process in which a block of data is formed in the hash table when collision is resolved.

Key
39
29
8
18
19

$19\%10 = 9$ cluster is formed

$18\%10 = 8$

$39\%10 = 9$

$29\%10 = 9$

$8\%10 = 8$

          rest of the table is empty

this cluster problem can be solved by quadratic probing.

QUADRATIC PROBING:

Quadratic probing operates by taking the original hash value and adding successive values of an arbitrary quadratic polynomial to the starting value. This method uses following formula.

$$H(key) = (Hash(key) + i^2) \% m)$$

where m can be table size or any prime number.

for eg; If we have to insert following elements in the hash table with table size 10:

37, 90, 55, 22, 17, 49, 87

0	90
1	11
2	22
3	
4	
5	55
6	
7	37
8	
9	

$37 \% 10 = 7$

$90 \% 10 = 0$

$55 \% 10 = 5$

$22 \% 10 = 2$

$11 \% 10 = 1$

Now if we want to place 17 a collision will occur as $17\%10 = 7$ and bucket 7 has already an element 37. Hence we will apply quadratic probing to insert this record in the hash table.

$H_i (key) = (Hash(key) + i^2) \% m$

Consider $i = 0$ then

$(17 + 0^2) \% 10 = 7$

$(17 + 1^2)$ % 10 = 8, when i =1

The bucket 8 is empty hence we will place the element at index 8. Then comes 49 which will be placed at index 9.

49 % 10 = 9

Index	Value
0	90
1	11
2	22
3	
4	
5	55
6	
7	37
8	49
9	

Now to place 87 we will use quadratic probing.

$(87 + 0)$ % 10 = 7
$(87 + 1)$ % 10 = 8… but already occupied
$(87 + 2^2)$ % 10 = 1.. already occupied
$(87 + 3^2)$ % 10 = 6

It is observed that if we want place all the necessary elements in the hash table the size of divisor (m) should be twice as large as total number of elements.

Index	Value
0	90
1	11
2	22
3	
4	
5	
6	55
7	87
	37
8	49
9	

DOUBLE HASHING

Double hashing is technique in which a second hash function is applied to the key when a collision occurs. By applying the second hash function we will get the number of positions from the point of collision to insert.
There are two important rules to be followed for the second function:
- it must never evaluate to zero.
- must make sure that all cells can be probed.

The formula to be used for double hashing is

$H_1(key)$ = key mod tablesize

$H_2(key)$ = M – (key mod M)

where M is a prime number smaller than the size of the table.

Consider the following elements to be placed in the hash table of size 10
37, 90, 45, 22, 17, 49, 55
Initially insert the elements using the formula for $H_1(key)$.
Insert 37, 90, 45, 22

$H_1(37)$ = 37 % 10 = 7
$H_1(90)$ = 90 % 10 = 0
$H_1(45)$ = 45 % 10 = 5
$H_1(22)$ = 22 % 10 = 2
$H_1(49)$ = 49 % 10 = 9

Key
90
22
45
37
49

Now if 17 to be inserted then

$H_1(17) = 17 \% 10 = 7$
$H_2(key) = M - (key \% M)$

Here M is prime number smaller than the size of the table. Prime number
smaller than table size 10 is 7

Hence $M = 7$

$H_2(17) = 7-(17 \% 7)$
$\qquad = 7 - 3 = 4$

Key
90
17
22
45
37
49

That means we have to insert the element 17 at 4 places from 37. In short we ha v
jumps. Therefore the 17 will be placed at index 1.

Now to insert number 55

$H_1(55) = 55 \% 10 = 5 \longrightarrow$ Collision

$H_2(55) = 7-(55 \% 7)$
$\qquad = 7 - 6 = 1$

That means we have to take one jump from index 5 to place 55.
Finally the hash table will be -

Key
90
17
22
45
55
37
49

Comparison of Quadratic Probing & Double Hashing

The double hashing requires another hash function whose probing efficiency is same as
some another hash function required when handling random collision.
The double hashing is more complex to implement than quadratic probing. The quadratic
probing is fast technique than double hashing.

REHASHING

Rehashing is a technique in which the table is resized, i.e., the size of table is doubled by creating
a new table. It is preferable is the total size of table is a prime number. There are situations in
which the rehashing is required.

- When table is completely full
- With quadratic probing when the table is filled half.
- When insertions fail due to overflow.

In such situations, we have to transfer entries from old table to the new table by re computing their positions using hash functions.

Consider we have to insert the elements 37, 90, 55, 22, 17, 49, and 87. the table size is 10 and will use hash function.,

H(key) = key mod tablesize

37 % 10 = 7
90 % 10= 0
55 % 10 = 5
22 % 10 = 2
17 % 10 = 7 Collision solved by linear probing
49 % 10 = 9

Now this table is almost full and if we try to insert more elements collisions will occur and eventually further insertions will fail. Hence we will rehash by doubling the table size. The old table size is 10 then we should double this size for new table, that becomes 20. But 20 is not a prime number, we will prefer to make the table size as 23. And new hash function will be

H(key) key mod 23

37 % 23 = 14
90 % 23 = 21
55 % 23 = 9
22 % 23 = 22
17 % 23 = 17
49 % 23 = 3
87 % 23 = 18

0	90
1	11
2	22
3	
4	
5	55
6	87
7	37
8	49
9	
10	
11	
12	
13	
14	
15	
16	
17	
18	
19	
20	
21	
22	
23	

Now the hash table is sufficiently large to accommodate new insertions.

Advantages:

1. This technique provides the programmer a flexibility to enlarge the table size if required.
2. Only the space gets doubled with simple hash function which avoids occurrence of collisions.

EXTENSIBLE HASHING

➤ Extensible hashing is a technique which handles a large amount of data. The data to be placed in the hash table is by extracting certain number of bits.
➤ Extensible hashing grow and shrink similar to B-trees.
➤ In extensible hashing referring the size of directory the elements are to be placed in buckets. The levels are indicated in parenthesis.

For eg:
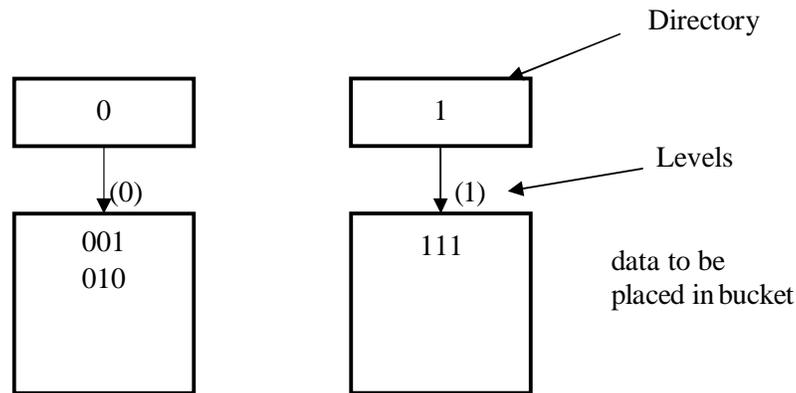
➤ The bucket can hold the data of its global depth. If data in bucket is more than global depth then, split the bucket and double the directory.

Consider we have to insert 1, 4, 5, 7, 8, 10. Assume each page can hold 2 data entries (2 is the depth).

Step 1: Insert 1, 4
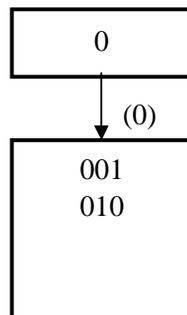
1 = 001

4 = 100

We will examine last bit of data and insert the data in bucket.

Insert 5. The bucket is full. Hence double the directory.

0

↓ (0)

100

1

↓ (1)

001
010
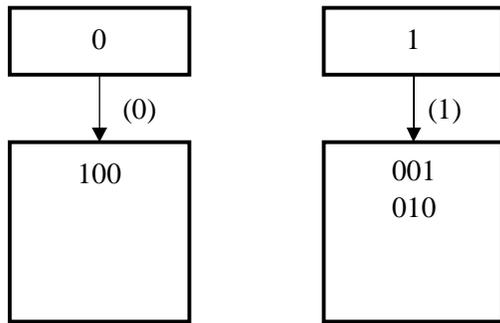
1 = 001

4 = 100

5 = 101

Based on last bit the data
is inserted.

Step 2: Insert 7
7 = 111
But as depth is full we can not insert 7 here. Then double the directory and split the bucket.
After insertion of 7. Now consider last two bits.

00

↓ (1)

100

01

↓ (2)

001
010

10

11

↓ (2)

111

Step 3: Insert 8 i.e. 1000

00

↓ (1)

100
1000
	0

01

↓ (2)

001
010

10

11

111
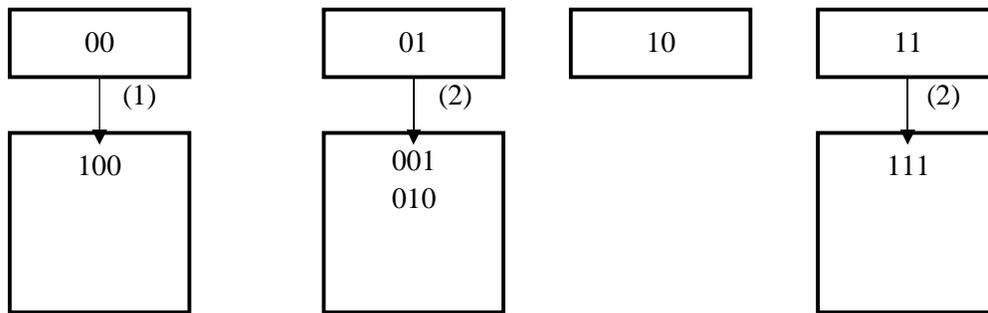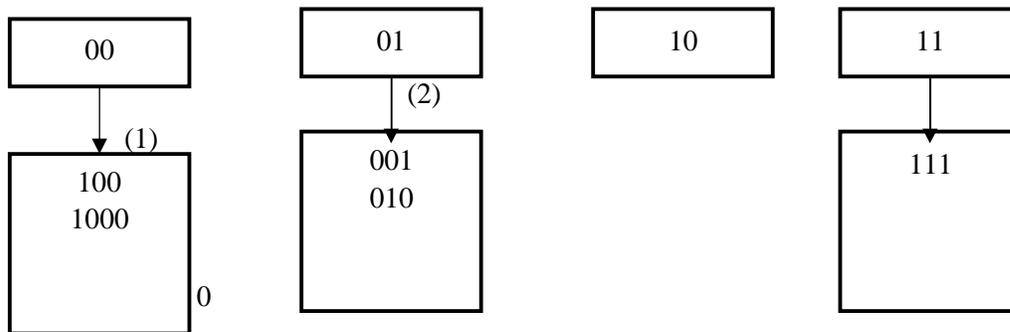
Thus the data is inserted using extensible hashing.

Deletion Operation:

If we wan tot delete 10 then, simply make the bucket of 10 empty.

00	01	10	11

↓ (1)	↓ (2)		↓ (2)

100 1000	001 101		111

Delete 7.

00	01	10	11

↓ (1)	↓ (1)		

| 100
1000 | 001
101 | | |

Note that the level was increased when we insert 7. Now on deletion of 7, the level should get decremented.
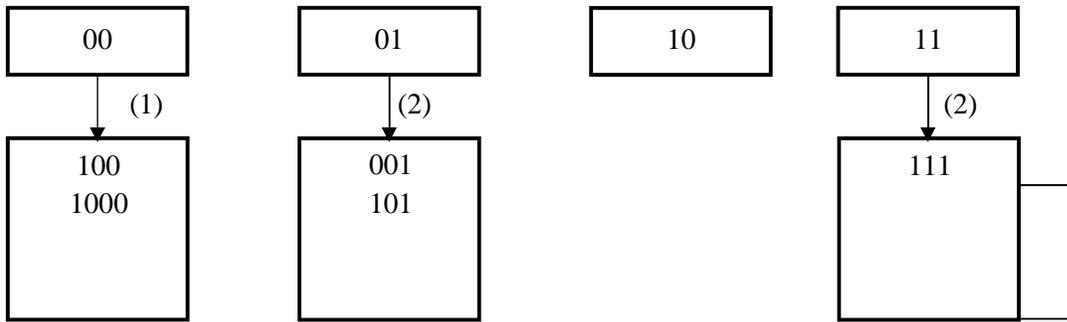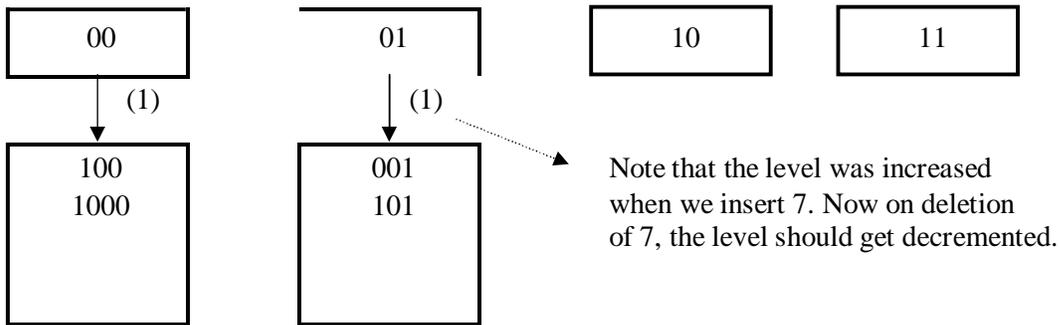
Delete 8. Remove entry from directory 00.

00	00	10	11

↓ (1)	↓ (1)		

| 100 | 001
101 | | |
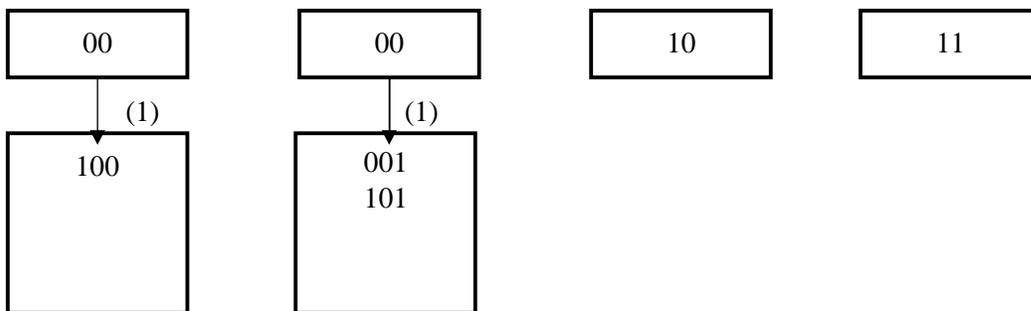
Applications of hashing:

1. In compilers to keep track of declared variables.
2. For online spelling checking the hashing functions are used.
3. Hashing helps in Game playing programs to store the moves made.
4. For browser program while caching the web pages, hashing is used.
5. Construct a *message authentication code* (MAC)
6. Digital signature.
7. Time stamping
8. Key updating: key is hashed at specific intervals resulting in new key

UNIT-3

TREES

A Tree is a data structure in which each element is attached to one or more elements directly beneath it.
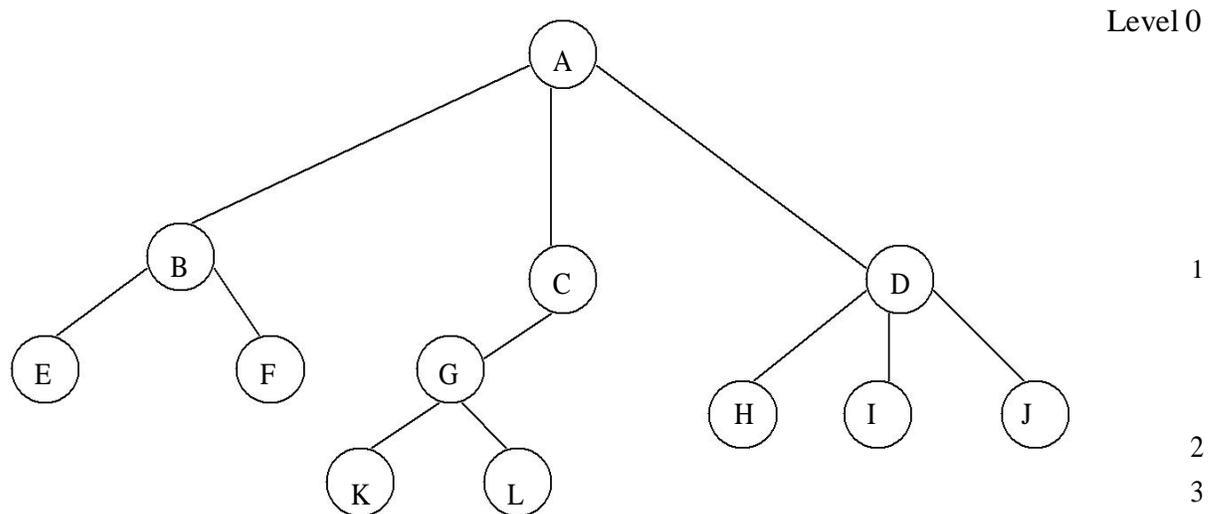
Terminology

- The connections between elements are called **branches**.
- A tree has a single root, called **root** node, which is shown at the top of the tree. i.e. root is always at the highest level 0.
- Each node has exactly one node above it, called **parent.** Eg: A is the parent of B,C and D.
- The nodes just below a node are called its **children**. ie. child nodes are one level lower than the parent node.
- A node which does not have any child called **leaf or terminal node**. Eg: E, F, K, L, H, I and M are leay.
- Nodes with at least one child are called **non terminal or internal nodes**.
- The child nodes of same parent are said to be **siblings**.
- A **path** in a tree is a list of distinct nodes in which successive nodes are connected by branches in the tree.
- The **length** of a particular path is the number of branches in that path. The **degree** of a node of a tree is the number of children of that node.
- The maximum number of children a node can have is often referred to as the **order** of a tree. The **height or depth** of a tree is the length of the longest path from root to any leaf.

1. **Root:** This is the unique node in the tree to which further sub trees are attached. Eg: A

Degree of the node: The total number of sub-trees attached to the node is called the degree of the node.Eg: For node A degree is 3. For node K degree is 0

3. Leaves: These are the terminal nodes of the tree. The nodes with degree 0 are always the leaf nodes.

Eg: E, F, K, L,H, I, J

4. Internal nodes: The nodes other than the root node and the leaves are called the internal nodes. Eg: B, C, D, G

5. Parent nodes: The node which is having further sub-trees(branches) is called the parent node of those sub-trees. Eg: B is the parent node of E and F.

6. Predecessor: While displaying the tree, if some particular node occurs previous to some other node then that node is called the predecessor of the other node. Eg: E is the predecessor of the node B.

7. Successor: The node which occurs next to some other node is a successor node. Eg: B is the successor of E and F.

8. Level of the tree: The root node is always considered at level 0, then its adjacent children are supposed to be at level 1 and so on. Eg: A is at level 0, B,C,D are at level 1, E,F,G,H,I,J are at level 2, K,L are at level 3.

9. Height of the tree: The maximum level is the height of the tree. Here height of the tree is 3. The height if the tree is also called depth of the tree.

10. Degree of tree: The maximum degree of the node is called the degree of the tree.

BINARY TREES

Binary tree is a tree in which each node has at most two children, a left child and a right child. Thus the order of binary tree is 2.

A binary tree is either empty or consists
of a) a node called the root
b) left and right sub trees are themselves binary trees.

A binary tree is a finite set of nodes which is either empty or consists of a root and two disjoint trees called left sub-tree and right sub-tree.
In binary tree each node will have one data field and two pointer fields for representing the sub-branches. The degree of each node in the binary tree will be at the most two.

Types Of Binary Trees:

There are 3 types of binary trees:

1. **Left skewed binary tree:** If the right sub-tree is missing in every node of a tree we call it as left skewed tree.
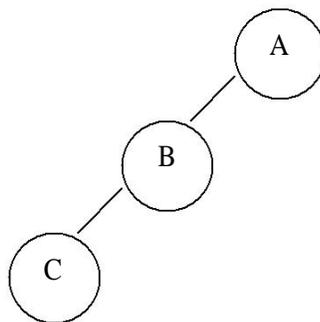
2. **Right skewed binary tree:** If the left sub-tree is missing in every node of a tree we call it is right sub-tree.
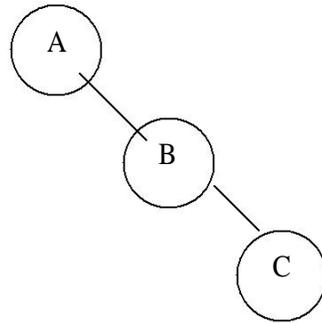
3. **Complete binary tree:**

The tree in which degree of each node is at the most two is called a complete binary tree. In a complete binary tree there is exactly one node at level 0, two nodes at level 1 and four nodes at level 2 and so on. So we can say that a complete binary tree depth d will contain exactly 2^l nodesat each level l, where l is from 0 to d.
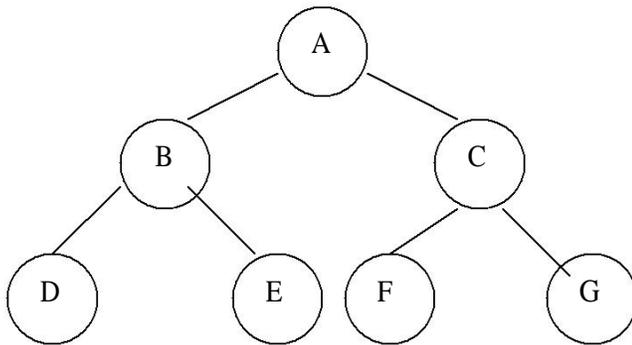
Note:

1. A binary tree of depth n will have maximum 2^n-1 nodes.
2. A complete binary tree of level l will have maximum $2l$ nodes at each level, where l starts from 0.
3. Any binary tree with n nodes will have at the most n+1 null branches.
4. The total number of edges in a complete binary tree with n terminal nodes are 2(n-1).

<u>Binary Tree Representation</u>

A binary tree can be represented mainly in 2 ways:

a) Sequential Representation
b) Linked Representation

a) Sequential Representation
The simplest way to represent binary trees in memory is the sequential representation that uses one-dimensional array.
1) The root of binary tree is stored in the 1 st location of array
2) If a node is in the jth location of array, then its left child is in the location 2J+1 and its right child in the location 2J+2

The maximum size that is required for an array to store a tree is $2^{d+1}-1$, where d is the depth of the tree.
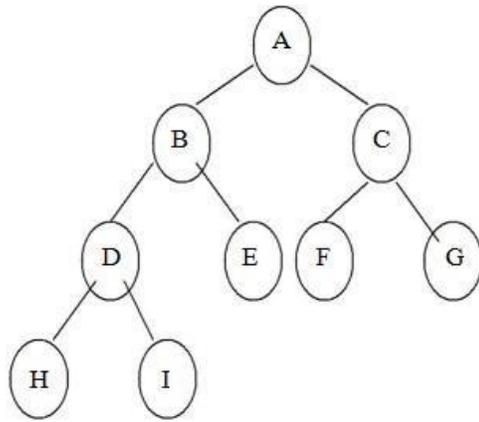
POSITION	ARRAY
0	A
1	B
2	C
3	D
4	E
5	F
6	G
7	H
8	I
.	.
.	.
.	.
.	.
.	.

Advantages of sequential representation:

The only advantage with this type of representation is that the direct access to any node can be possible and finding the parent or left children of any particular node is fast because of the random access.

Disadvantages of sequential representation:

1. The major disadvantage with this type of representation is wastage of memory. For example in the skewed tree half of the array is unutilized.

2. In this type of representation the maximum depth of the tree has to be fixed. Because we have decide the array size. If we choose the array size quite larger than the depth of the tree, then it will be wastage of the memory. And if we coose array size lesser than the depth of the tree then we will be unable to represent some part of the tree.

3. The insertions and deletion of any node in the tree will be costlier as other nodes has to be adjusted at appropriate positions so that the meaning of binary tree can bepreserved.

As these drawbacks are there with this sequential type of representation, we will search for more flexible representation. So instead of array we will make use of linked list to represent the tree.

b) Linked Representation

Linked representation of trees in memory is implemented using pointers. Since each node in a binary tree can have maximum two children, a node in a linked representation has two pointers for both left and right child, and one information field. If a node does not have any child, the corresponding pointer field is made NULL pointer.

In linked list each node will look like this:

Left Child	Data	Right Child

Advantages of linked representation:

1. This representation is superior to our array representation as there is no wastage of memory. And so there is no need to have prior knowledge of depth of the tree. Using dynamic memory concept one can create as much memory(nodes) as required. By chance if some nodes are unutilized one can delete the nodes by making the address free.

2. Insertions and deletions which are the most common operations can be done without moving the nodes.

75

Disadvantages of linked representation:

1. This representation does not provide direct access to a node and special algorithms are required.
2. This representation needs additional space in each node for storing the left and right sub-trees.

TRAVERSING A BINARY TREE

Traversing a tree means that processing it so that each node is visited exactly once. A binary tree can be
traversed a number of ways. The most common tree traversals are

- In-order
- Pre-order and
- Post-order

Pre-order	1.Visit the root	Root | Left | Right
	2.Traverse the left sub tree in pre-order	
	3.Traverse the right sub tree in pre-order.	
In-order	1.Traverse the left sub tree in in-order	Left | Root | Right
	2.Visit the root	
	3.Traverse the right sub tree in in-order.	
Post-order	1.Traverse the left sub tree in post-order	Left | Right | Root
	2.Traverse the right sub tree in post-order.	
	3.Visit the root	
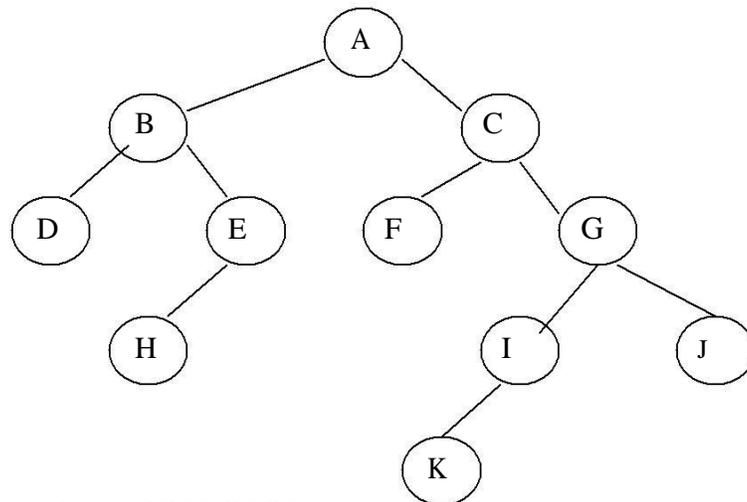
The pre-order traversal is: ABDEHCFGIKJ
The in-order traversal is : DBHEAFCKIGJ
The post-order traversal is:DHEBFKIJGCA

Inorder Traversal:
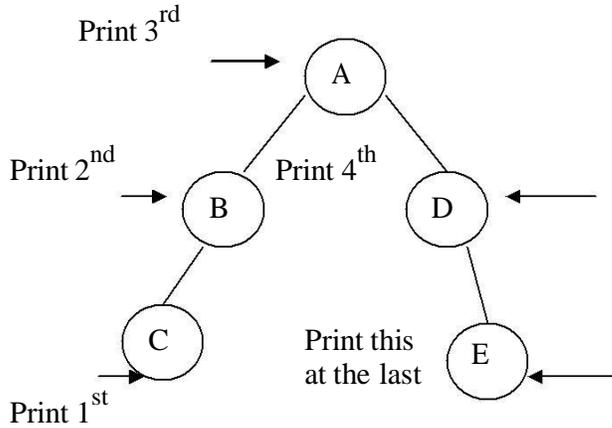
C-B-A-D-E is the inorder traversal i.e. first we go towards the leftmost node. i.e. C so print that node C. Then go back to the node B and print B. Then root node A then move towards the right sub-tree print D and finally E. Thus we are following the tracing sequence of Left|Root|Right. This type of traversal is called inorder traversal. The basic principle is to traverse left sub-tree then root and then the right sub-tree.

Pseudo Code:

```
template <class T>
void inorder(bintree<T> *temp)
{
        if(temp!=NULL)
        {
                inorder(temp->left);
                cout<<"temp->data";
                inorder(temp->right);
        }
}
```

Preorder Traversal:
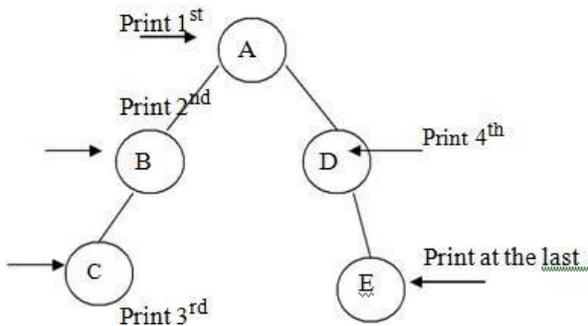
is the preorder traversal of the above fig. We are following Root|Left|Right path i.e. data at the root node will be printed first then we move on the left sub-tree and go on printing the data till we reach to the left most node. Print the data at that node and then move to the right sub- tree. Follow the same principle at each sub-tree and go on printing the data accordingly.

```
template <class T>
void preorder(bintree<T> *temp)
```

77

```
{
        if(temp!=NULL)
        {
                cout<<"temp->data";            preorder(temp->left);
                preorder(temp->right);
        }
}
```

Postorder Traversal:


Print at the last
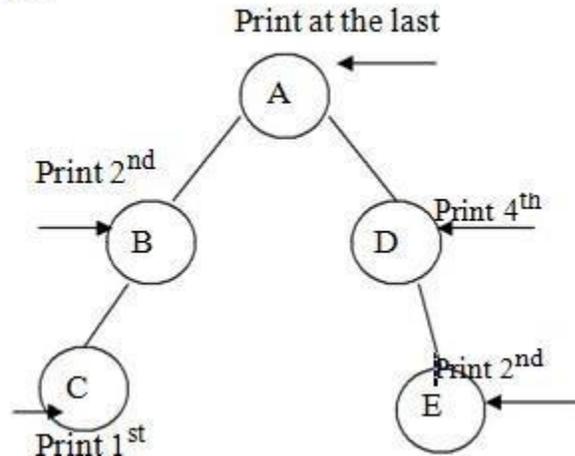
From figure the postorder traversal is C-D-B-E-A. In the postorder traversal we are following the Left|Right|Root principle i.e. move to the leftmost node, if right sub-tree is there or not if not then print the leftmost node, if right sub-tree is there move towards the right most node. The key idea here is that at each sub-tree we are following the Left|Right|Root principle and print the data accordingly.

Pseudo Code:

```
template <class T>
void postorder(bintree<T> *temp)
{
        if(temp!=NULL)
        {
                postorder(temp->left);
                postorder(temp->right);
                cout<<"temp->data";
        }
}
```

BINARY SEARCH TREE

        In the simple binary tree the nodes are arranged in any fashion. Depending on user's desire the new nodes can be attached as a left or right child of any desired node. In such a case finding for any node is a long cut procedure, because in that case we have to search the entire tree. And thus the searching time complexity will get increased unnecessarily. So to make the searching algorithm faster in a binary tree we will go for building the binary search tree. The binary search tree is based on the binary search algorithm. While creating the binary search tree the data is systematically arranged. That means values at **left sub-tree < root node value < right sub-tree values.**
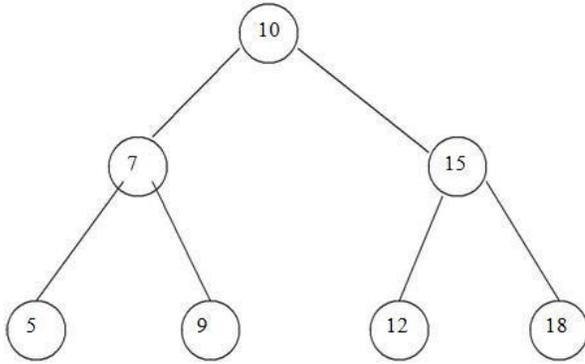
Operations On Binary Search Tree:

The basic operations which can be performed on binary search tree are.

1. Insertion of a node in binary search tree.
2. Deletion of a node from binary search tree.
3. Searching for a particular node in binary search tree.

Insertion of a node in binary search tree.

While inserting any node in binary search tree, look for its appropriate position in the binary search tree. We start comparing this new node with each node of the tree. If the value of the node which is to be inserted is greater than the value of the current node we move on to the right sub-branch otherwise we move on to the left sub-branch. As soon as the appropriate position is found we attach this new node as left or right child appropriately.
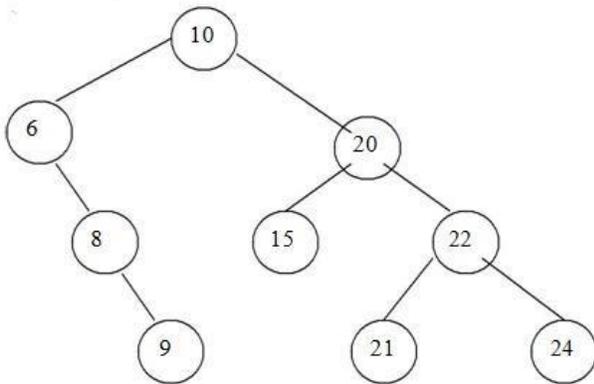
Before Insertion

In the above fig, if we wan to insert 23. Then we will start comparing 23 with value of root node i.e. 10. As 23 is greater than 10, we will move on right sub-tree. Now we will compare 23 with 20 and move right, compare 23 with 22 and move right. Now compare 23 with 24 but it is less than 24. We will move on left branch of 24. But as there is node as left child of 24, we can attach 23 as left child of 24.

After Insertion
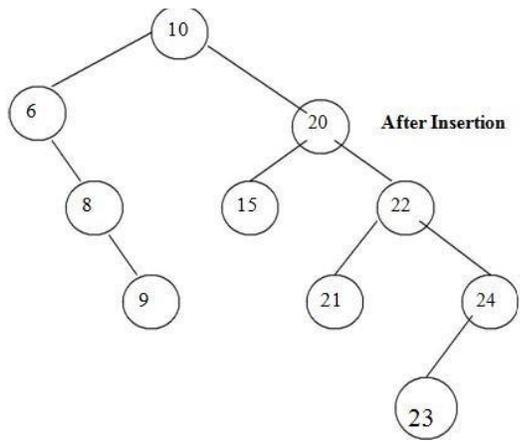
Deletion of a node from binary search tree.

For deletion of any node from binary search tree there are three   which are possible.

   i.   Deletion of leaf node.
   ii.   Deletion of a node having one child.
   iii.   Deletion of a node having two children.

Deletion of leaf node.

This is the simplest deletion, in which we set the left or right pointer of parent node as NULL.


Before deletion
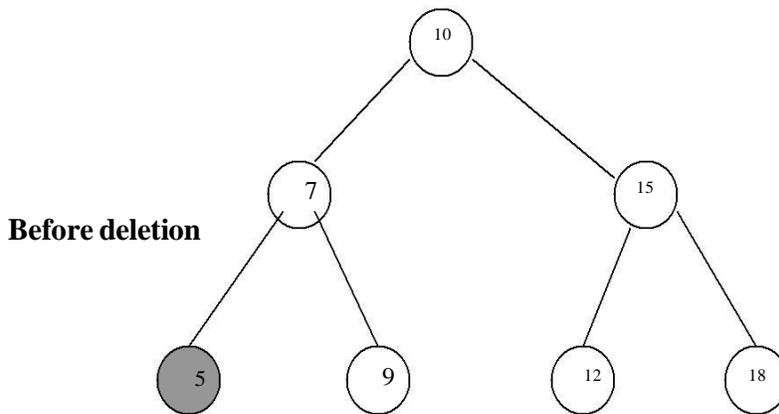
From the above fig, we want to delete the node having value 5 then we will set left pointer of its parent node as NULL. That is left pointer of node having value 7 is set to NULL.


After deletion
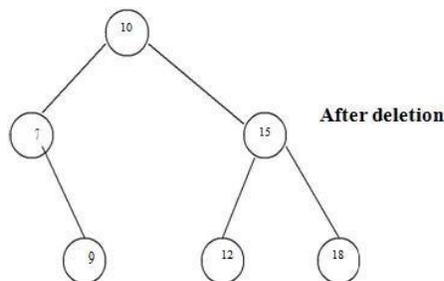
Deletion of a node having one child.

To explain this kind of deletion, consider a tree as given below.
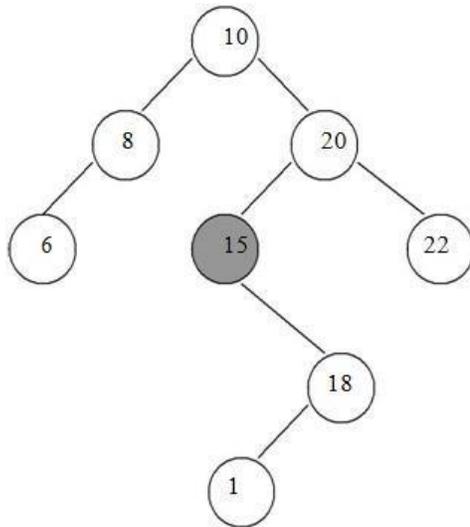
If we want to delete the node 15, then we will simply copy node 18 at place of 16 and then set the node free
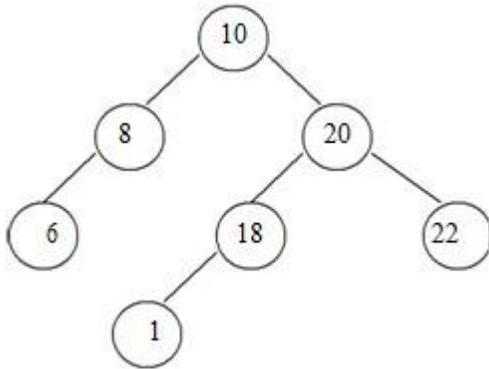
Deletion of a node having two children.
Consider a tree as given below.
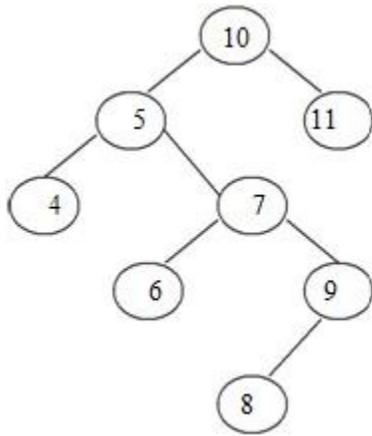
Let us consider that we want to delete node having value 7. We will then find out the inorder successor of node 7. We will then find out the inorder successor of node 7. The inorder successor will be simply copied at location of node 7.

That means copy 8 at the position where value of node is 7. Set left pointer of 9 as NULL. This completes the deletion procedure.
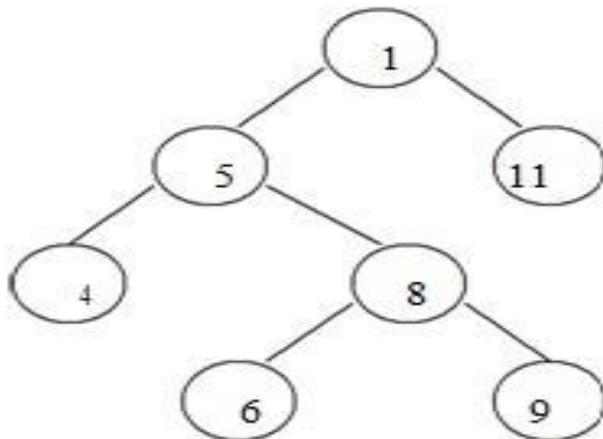
Searching for a node in binary search tree.

In searching, the node which we want to search is called a key node. The key node will be compared with each node starting from root node if value of key node is greater than current node then we search for it on right sub branch otherwise on left sub branch. If we reach to leaf node and still we do not get the value of key node then we declare "node is not present in the tree".
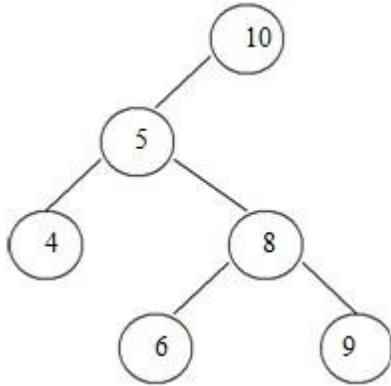
In the above tree, if we want to search for value 9. Then we will compare 9 with root node 10. As 9 is less than 10 we will search on left sub branch. Now compare 9 with 5, but 9 is greater than 5. So we will move on right sub tree. Now compare 9 with 8 but 9 is greater than 8 we will move on right sub branch. As the node we will get holds the value 9. Thus the desired node can be searched.

AVL TREES

Adelsion Velski and Lendis in 1962 introduced binary tree structure that is balanced with respect to height of sub trees. The tree can be made balanced and because of this retrieval

of any node can be done in $O(\log n)$ times, where n is total number of nodes. From the name of these scientists the tree is called AVL tree.

Definition:

An empty tree is height balanced if T is a non empty binary tree with T_L and T_R as its left and right sub trees. The T is height balanced if and only if

    i. T_L and T_R are height balanced.

    ii. $h_L - h_R <= 1$ where h_L and h_R are heights of T_L and T_R.

The idea of balancing a tree is obtained by calculating the balance factor of a tree.

Definition of Balance Factor:

The balance factor BF(T) of a node in binary tree is defined to be $h_L - h_R$ where h_L and h_R are heights of left and right sub trees of T.

For any node in AVL tree the balance factor i.e. BF(T) **is -1, 0 or +1**.

AVL Tree

BF=2 (at node 10)

BF=2 (at node 8)

BF=0 (at node 15)

BF=2 (at node 7)

BF=0 (at node 12)

BF=0 (at node 18)

BF=1 (at node 5)

BF=0 (at node 4)
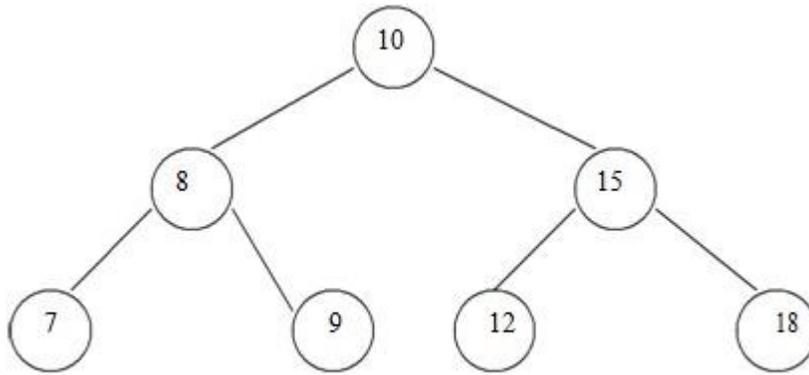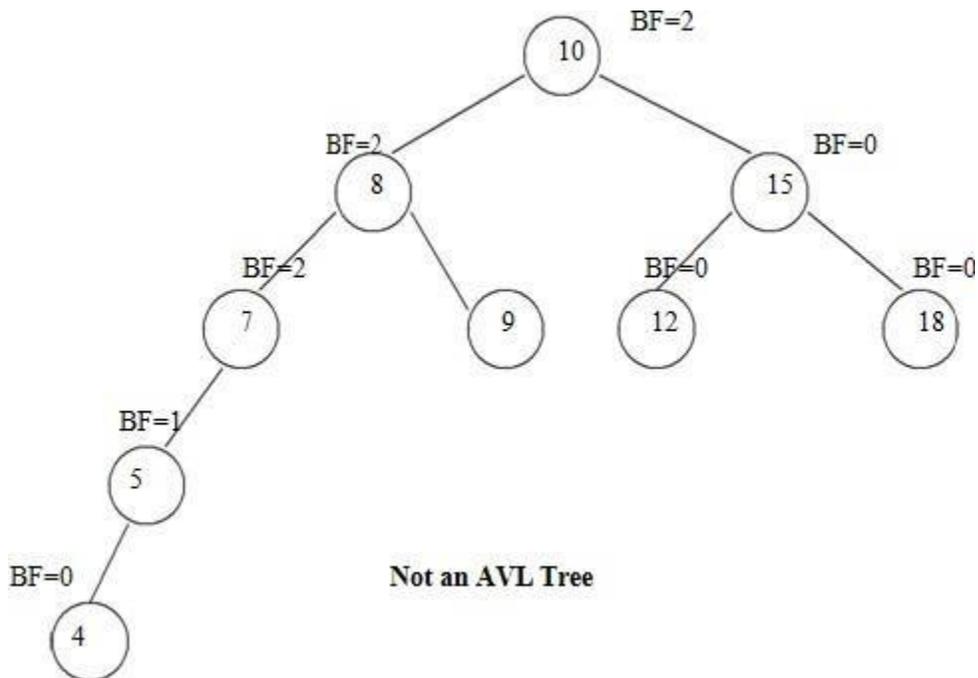
Not an AVL Tree

Height of AVL Tree:

Theorem: The height of AVL tree with n elements (nodes) is $O(\log n)$.

Proof: Let an AVL tree with n nodes in it. N_h be the minimum number of nodes in an AVL tree of height h.

In worst case, one sub tree may have height h-1 and other sub tree may have height h-2. And both these sub trees are AVL trees. Since for every node in AVL tree the height of left and right sub trees differ by at most 1.

Hence

$$N_h = N_{h-1} + N_{h-2} + 1$$

Where N_h denotes the minimum number of nodes in an AVL tree of height h.

$N_0 = 0$ $N_1 = 2$

We can also write it as

$N > N_h = N_{h-1} + N_{h-2} + 1$

$> 2N_{h-2}$

$> 4N_{h-4}$

.

.

$> 2iNh-2i$

If value of h is even, let $i = h/2-1$

Then equation becomes

$N > 2h/2-1N2$

$= N > 2(h-1)/2x4$ $(N2 = 4)$

$= O(\log N)$

If value of h is odd, let $I = (h-1)/2$ then equation becomes

$N > 2(h-1)/2$ N1

$N > 2(h-1)/2$ x 1

$H = O(\log N)$

This proves that height of AVL tree is always O(log N). Hence search, insertion and deletion can be carried out in logarithmic time.

Representation of AVL Tree

⌋ The AVL tree follows the property of binary search tree. In fact AVL trees are basically binary search trees with balance factors as -1, 0, or +1.

⌋ After insertion of any node in an AVL tree if the balance factor of any node becomes other than -1, 0, or +1 then it is said that AVL property is violated. Then we have to restore the destroyed balance condition. The balance factor is denoted at right top corner inside the node.

For example:

Original AVL Tree

Insert 13 Property violated
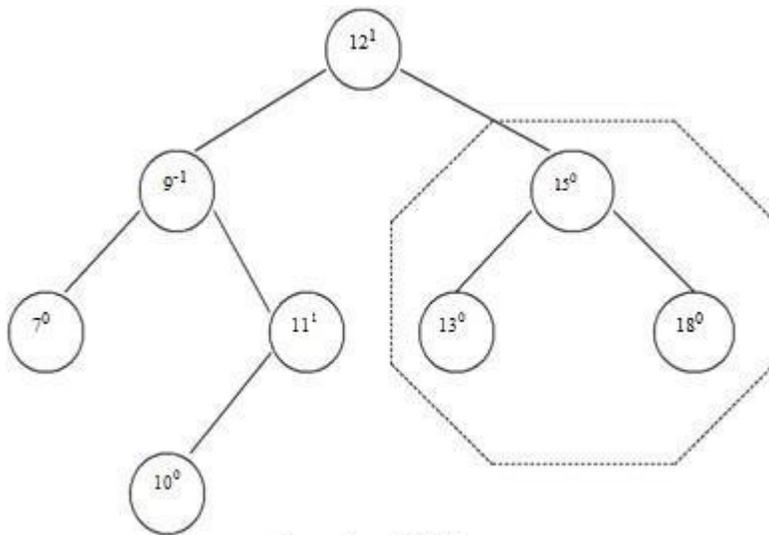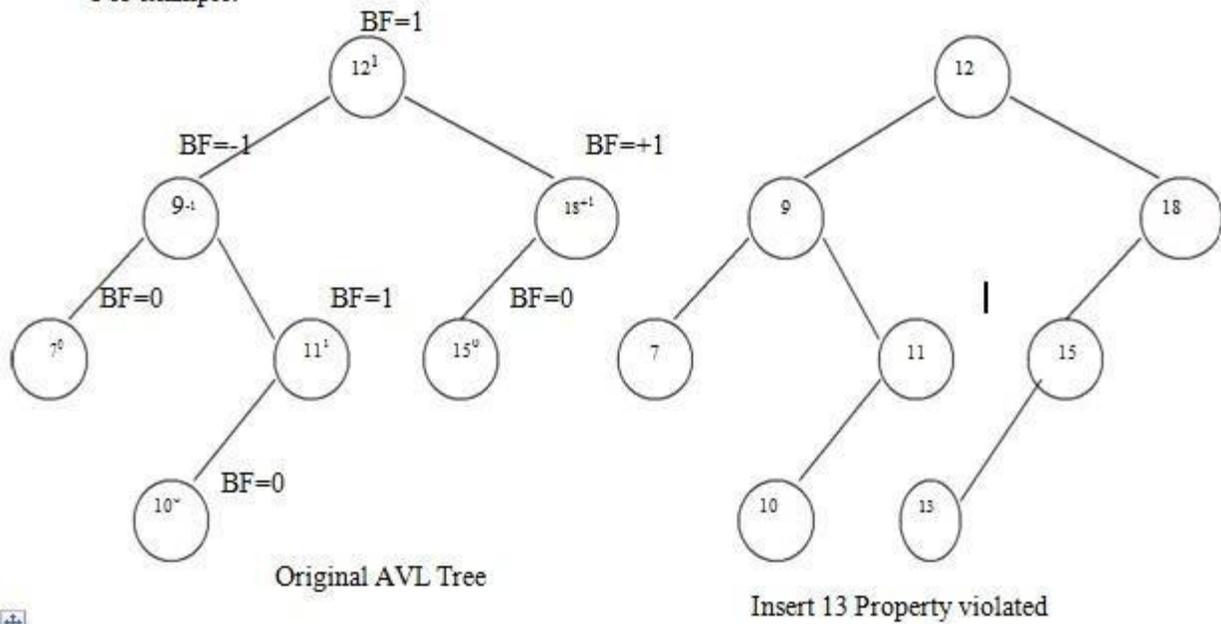
Restoring AVL Property

⌋ After insertion of a new node if balance condition gets destroyed, then the nodes on that path(new node insertion point to root) needs to be readjusted. That means only the affected sub tree is to be rebalanced.

⌋ The rebalancing should be such that entire tree should satisfy AVL property.

In above given example-

86
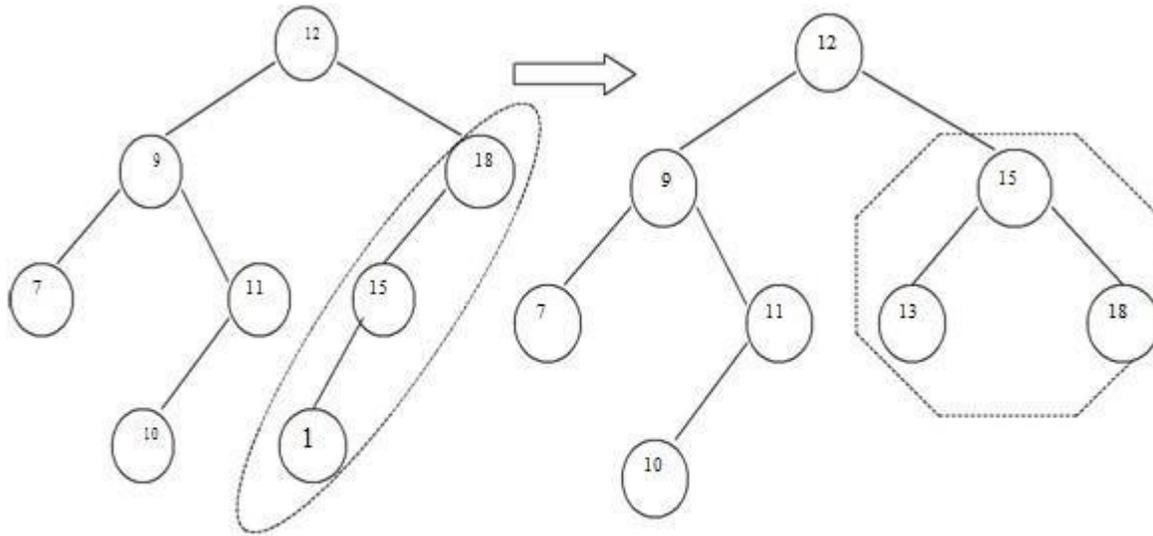
Nodes 18, 15, 13 are to be adjusted

By adjusting 15 the entire
Tree satisfies AVL property

Insertion of a node.

There are four different cases when rebalancing is required after insertion of new node.

1.   An insertion of new node into left sub tree of left child. (LL).
2.   An insertion of new node into right sub tree of left child. (LR).
3.   An insertion of new node into left sub tree of right child. (RL).
4.   An insertion of new node into right sub tree of right child.(RR).

Some modifications done on AVL tree in order to rebalance it is called rotations of AVL tree

There are two types of rotations:

Single rotation   Double rotation
        Left-Left(LL rotation)   Left-Right(LR rotation)
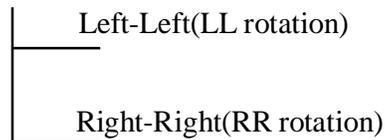
   Right-Right(RR rotation)   Right-Left(RL rotation)
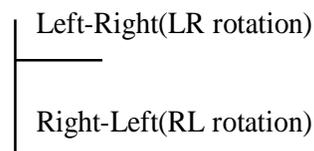
Insertion Algorithm:
1. Insert a new node as new leaf just as an ordinary binary search tree.
2. Now trace the path from insertion point(new node inserted as leaf) towards root. For each node
'n' encountered, check if heights of left (n) and right (n) differ by at most 1.
a) If yes, move towards parent (n).
b) Otherwise restructure by doing either a single rotation or a double rotation.
Thus once we perform a rotation at node 'n' we do not require to perform any rotation at any
ancestor on 'n'.

Different rotation in AVL tree:
1. LL rotation:
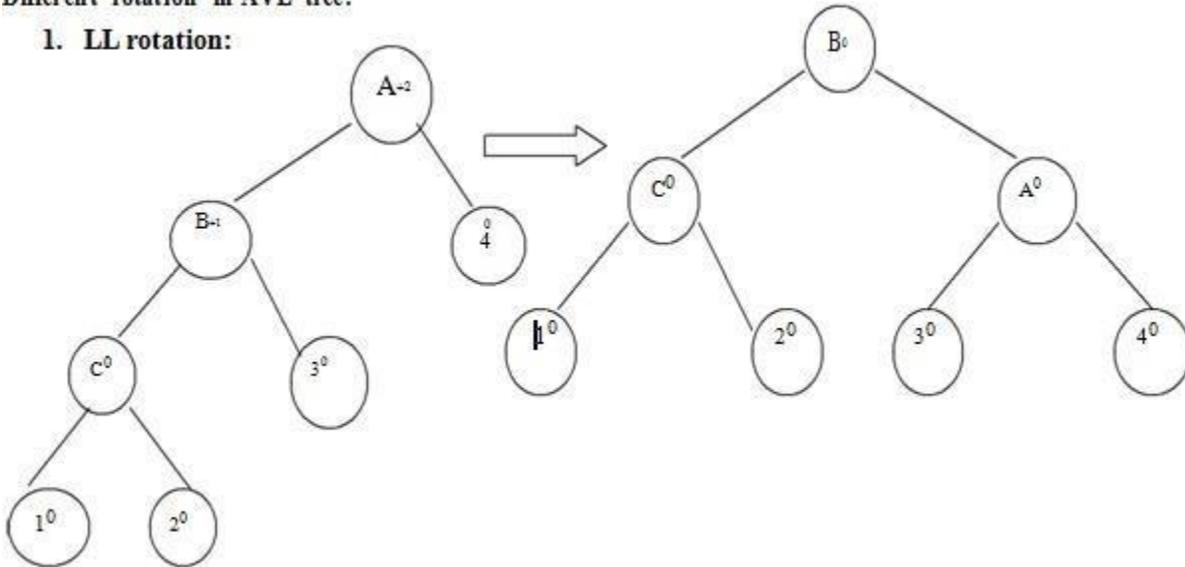
When node '1' gets inserted as a left child of node 'C' then AVL property gets destroyed i.e. node A has balance factor +2.

The LL rotation has to be applied to rebalance the nodes.

2. RR rotation:

When node '4' gets attached as right child of node 'C' then node 'A' gets unbalanced. The rotation which needs to be applied is RR rotation as shown in fig.
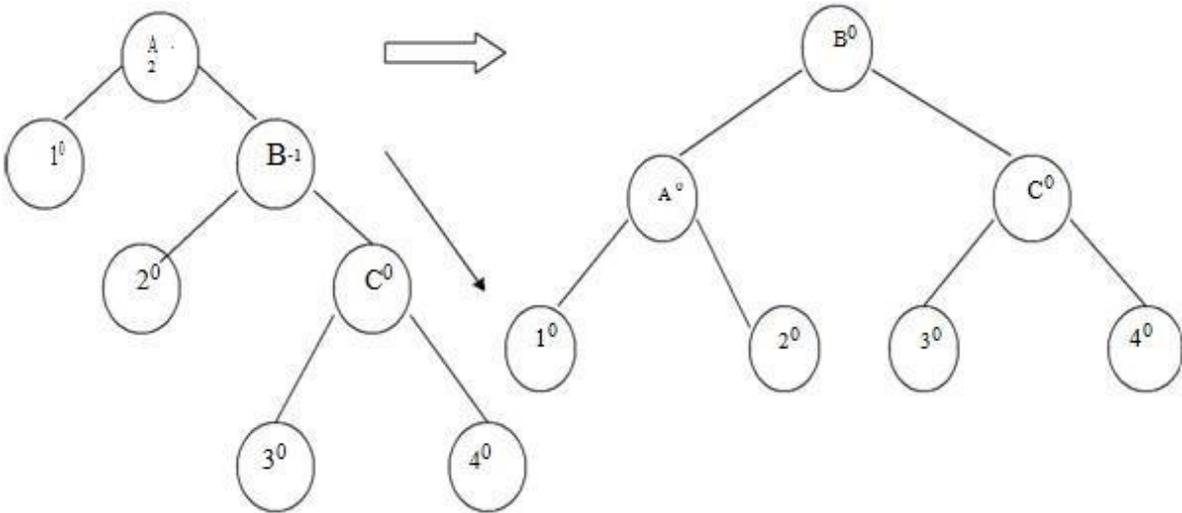
3. LR rotation:
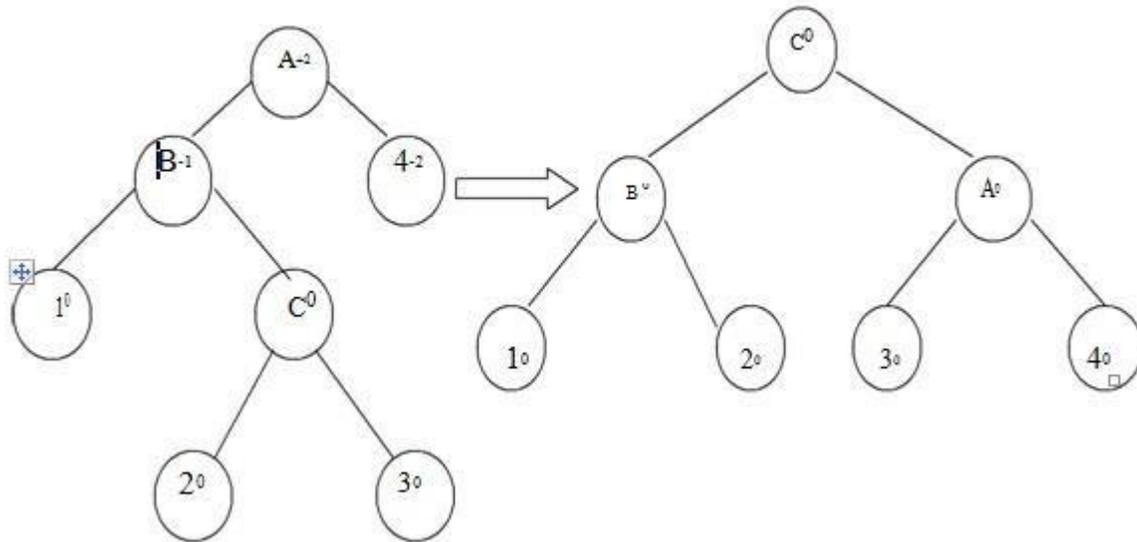
When node '3' is attached as a right child of node 'C' then unbalancing occurs because of LR. Hence LR rotation needs to be applied.

4. RL rotation
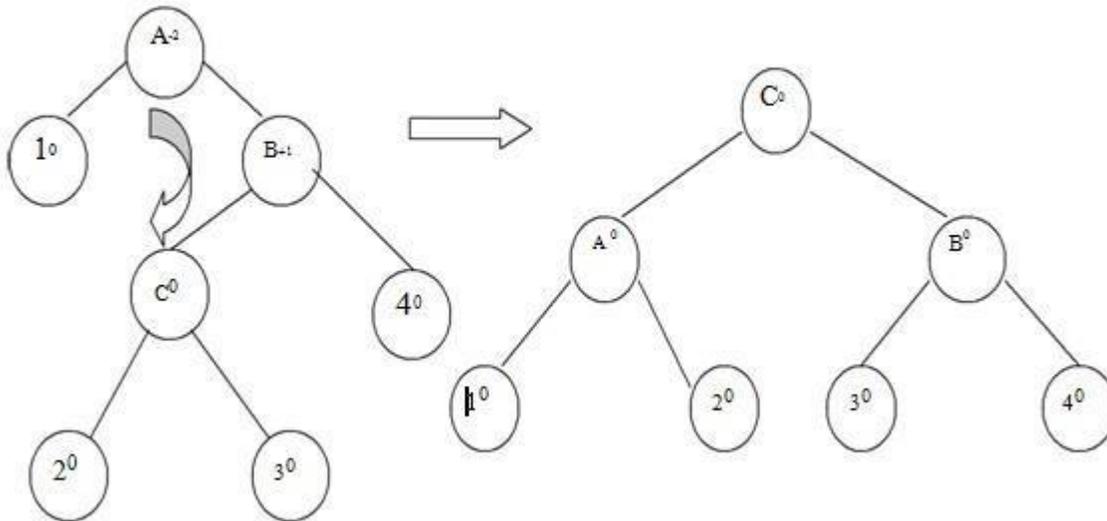
When node '2' is attached as a left child of node 'C' then node 'A' gets unbalanced as its balance factor becomes -2. Then RL rotation needs to be applied to rebalance the AVL tree.
Example:

Insert 1, 25, 28, 12 in the following AVL tree.
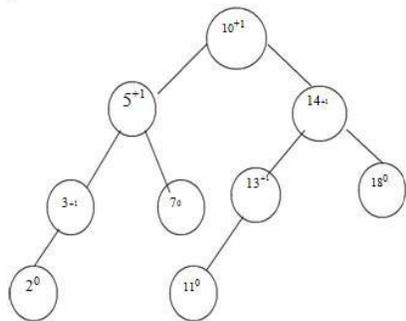
Insert 1

To insert node '1' we have to attach it as a left child of '2'. This will unbalance the tree as follows. We will apply LL rotation to preserve AVL property of it.
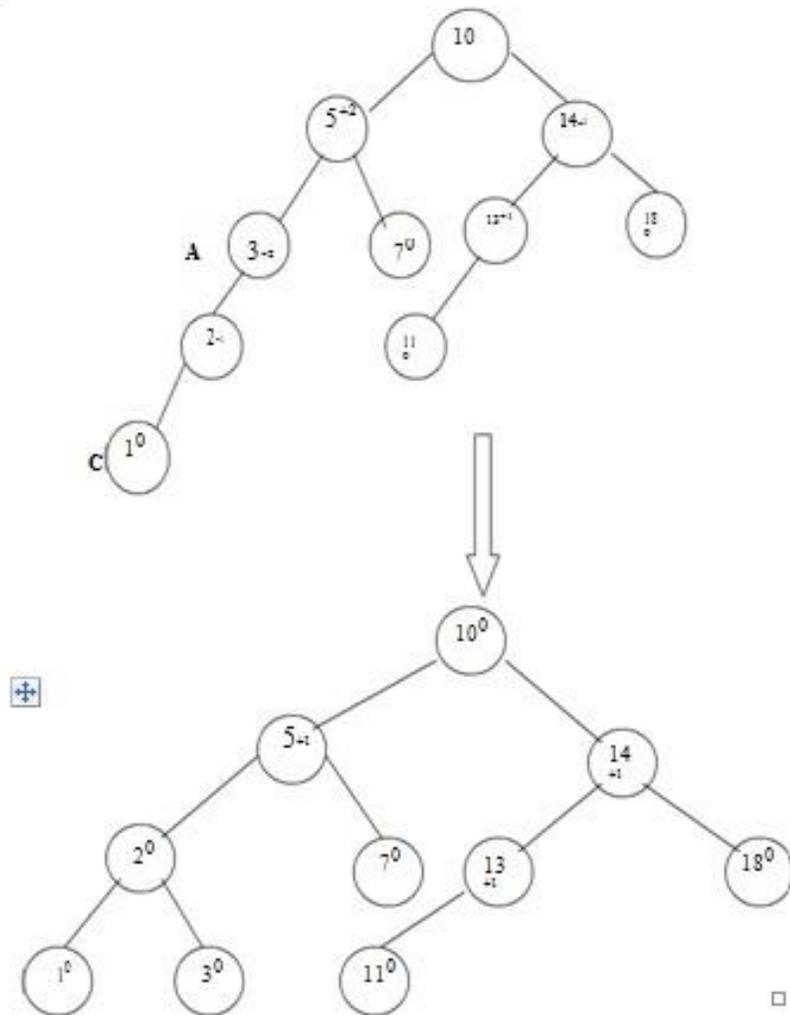
Insert 25

We will attach 25 as a right child of 18. No balancing is required as entire tree preserves the AVL property
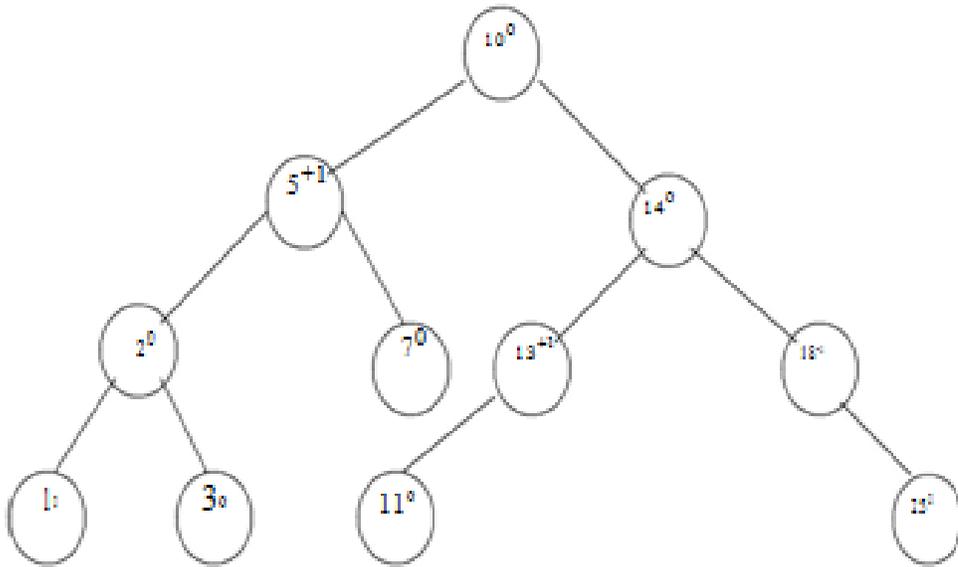
Insert 28

The node '28' is attached as a right child of 25. RR rotation is required to rebalance.
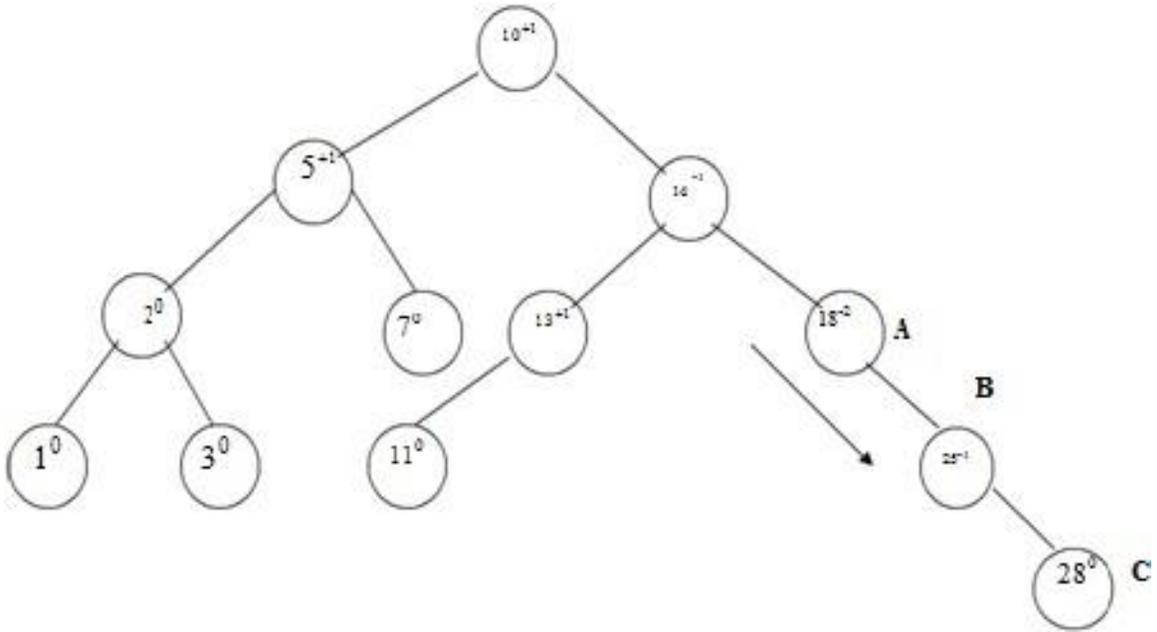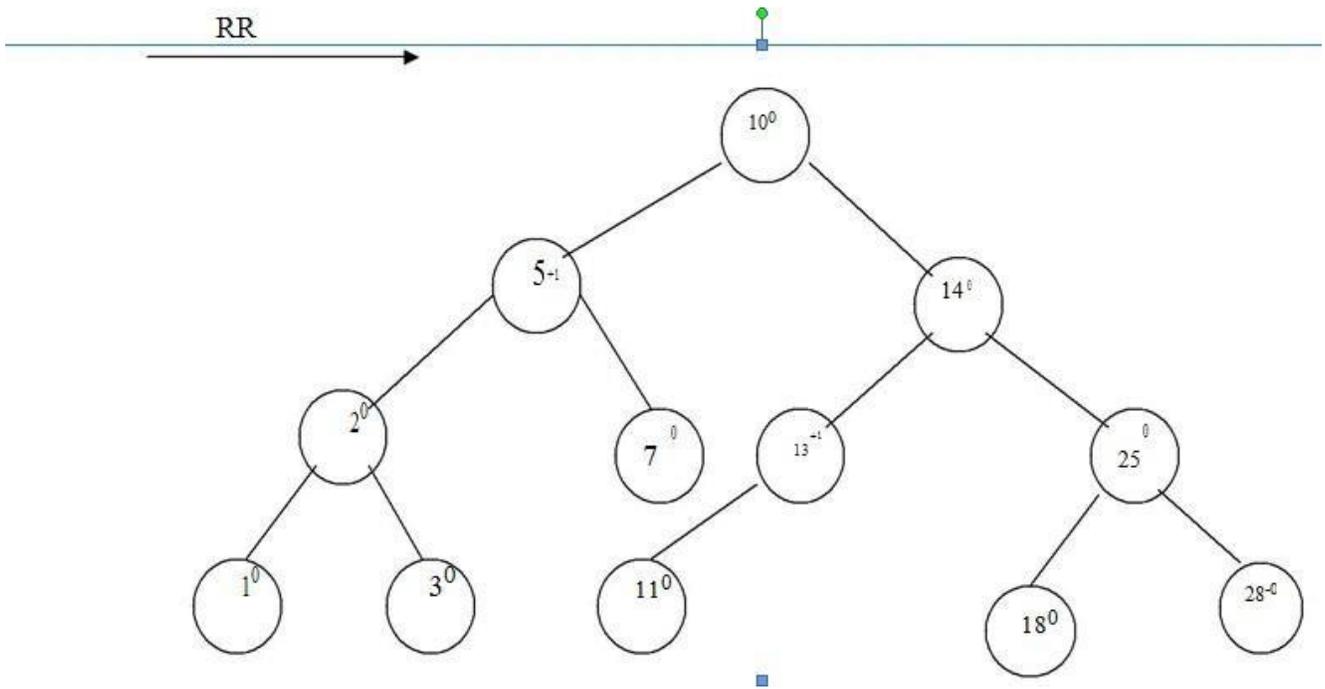
Insert 12
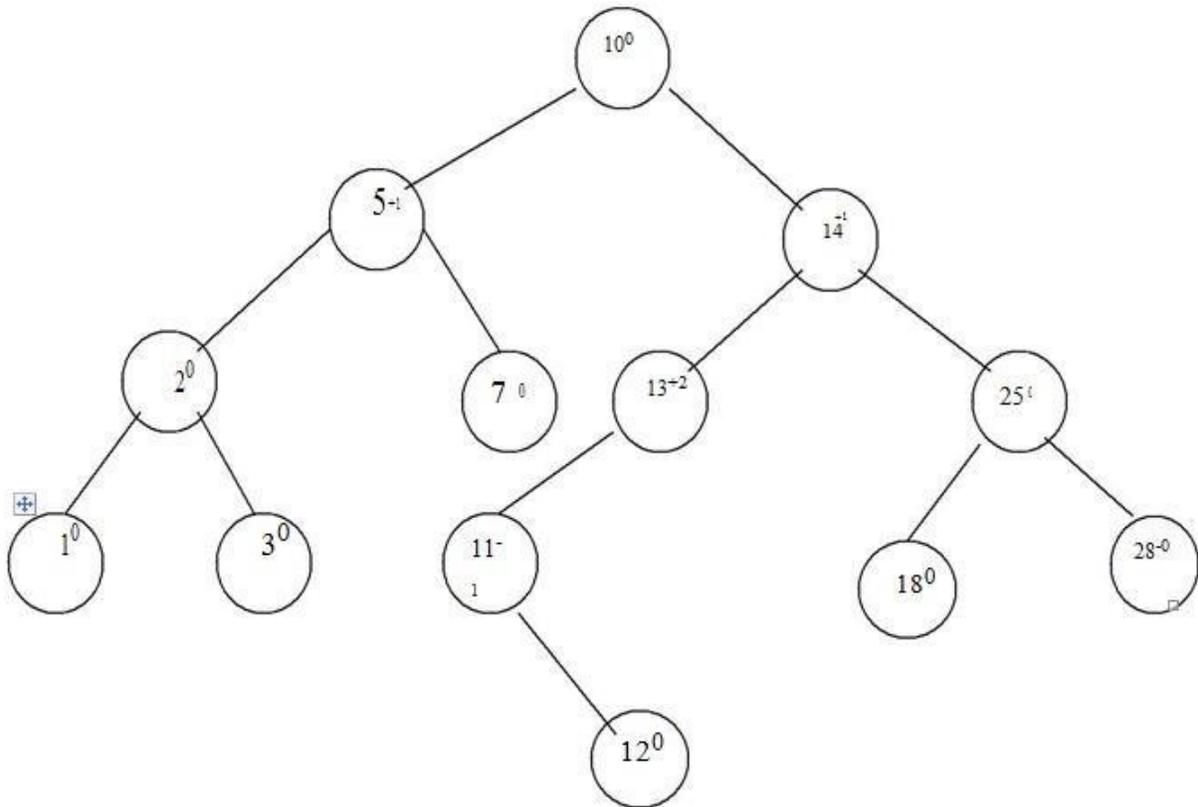
To rebalance the tree we have to apply LR rotation.
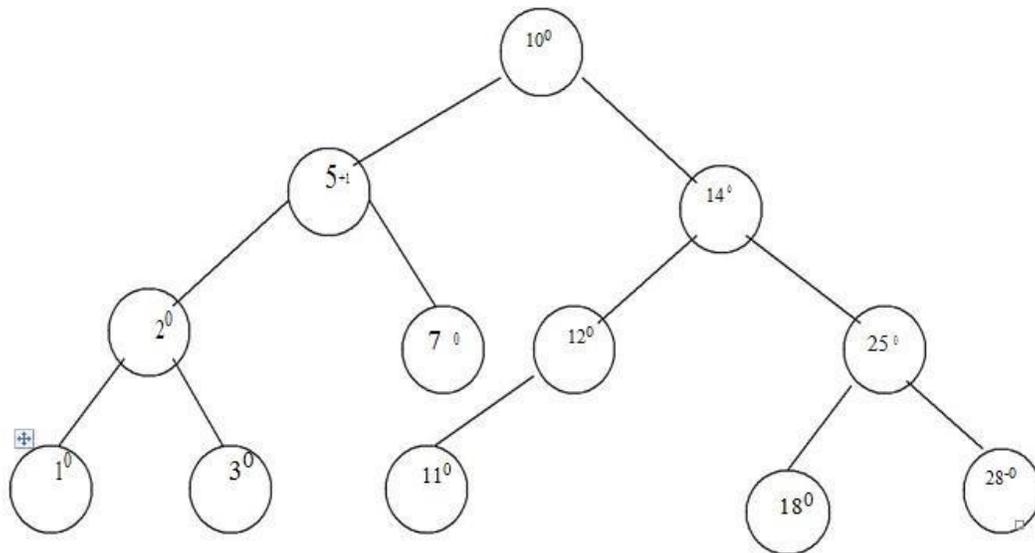
Thus by applying various rotations depending upon direction of insertion of new node the AVL tree can be restructured.

Deletion:

Even after deletion of any particular node from AVL tree, the tree has to be restructured in order to preserve AVL property. And thereby various rotations need to be applied.

Algorithm for deletion:

The deletion algorithm is more complex than insertion algorithm.
1. Search the node which is to be deleted.
2. a) If the node to be deleted is a leaf node then simply make it NULL to remove.

   b) If the node to be deleted is not a leaf node i.e. node may have one or two children, then the node must be swapped with its inorder successor. Once the node is swapped, we can remove this node.
3. Now we have to traverse back up the path towards root, checking the balance factor of every node along the path. If we encounter unbalancing in some sub tree

   then balance that sub tree using appropriate single or double
rotations. The deletion algorithm takes O(log n) time to delete any node.

Consider an AVL tree.
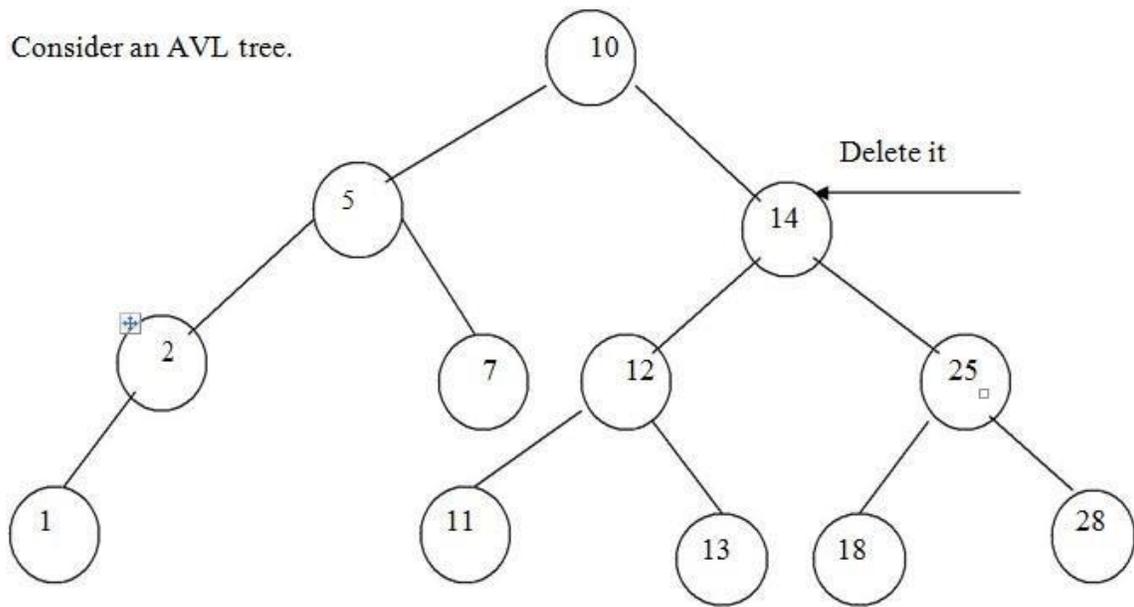
The tree becomes
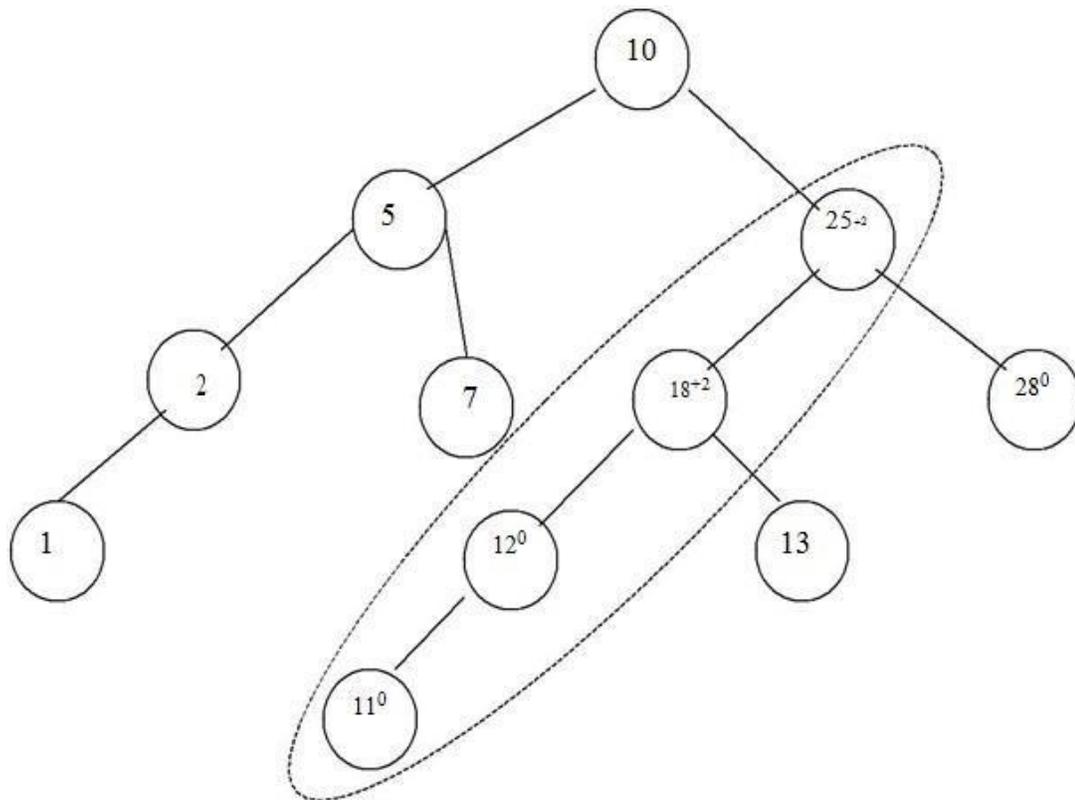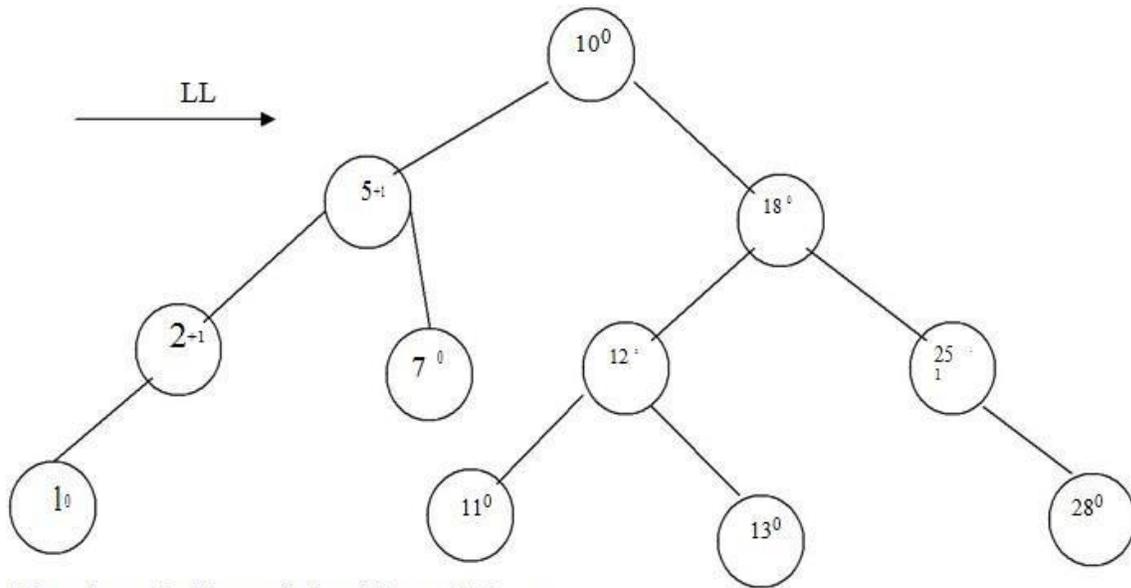
Thus the node 14 gets deleted from AVL tree.

Searching:

The searching of a node in an AVL tree is very simple. As AVL tree is basically binary search tree, the algorithm used for searching a node from binary search tree is the same one is used to search a node from AVL tree.

The searching of a node from AVL tree takes O(log n) time.

BTREES

➢ Multi-way trees are tree data structures with more than two branches at a node. The data structures of m-way search trees, B trees and Tries belong to this category of tree structures.

➢ AVL search trees are height balanced versions of binary search trees, provide efficient retrievals and storage operations. The complexity of insert, delete and search operations on AVL search trees id O(log n).

➢ Applications such as File indexing where the entries in an index may be very large, maintaining the index as m-way search trees provides a better option than AVL search trees which are but only balanced binary search trees.

➢ While binary search trees are two-way search trees, m-way search trees are extended binary search trees and hence provide efficient retrievals.

➢ B trees are height balanced versions of m-way search trees and they do not recommend representation of keys with varying sizes.

➢ Tries are tree based data structures that support keys with varying sizes.

Terminology of Graph

Graphs:-

A graph G is a discrete structure consisting of nodes (called vertices) and lines joining the nodes (called edges). Two vertices are adjacent to each other if they are joint by an edge. The edge joining the two vertices is said to be an edge incident with them. We use V (G) and E(G) to denote the set of vertices and edges of G respectively.

Example
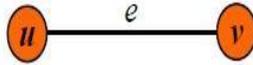
u and v are adjacent vertices; e is an edge incident with u and v. e can also be denoted by uv or vu.

Loops and Multiple Edges

An edge joining only one vertex is called a *loop*.   If there are more than one edge joining u and v of G, then all edges joining u and v form a *multiple edge* of G.
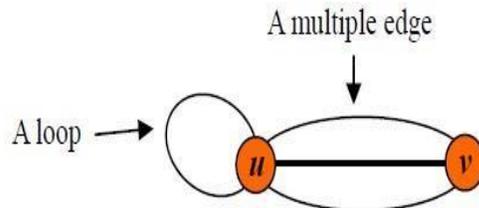
Simple Graph

A *simple graph* is a graph containing no loops and multiple edges.

Degrees of Vertices

The degree of a vertex is the number of edges incident with it, except that a loop at a vertex contributes twice to the degree of that vertex. The degree of the vertex v is denoted by $\deg(v)$ or $d(v)$.

Example

$d(u) = 5$, $d(v) = 3$ and $d(w) = 2$
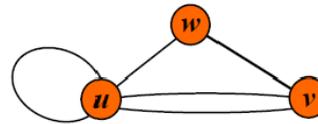
Complete Graphs

The *complete graph* on n vertices, denoted by K_n, is the simple graph in which any pair of vertices are adjacent.

Examples

I) K_1

II) K_2

III) K_3
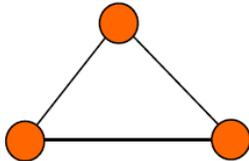
IV) K_4
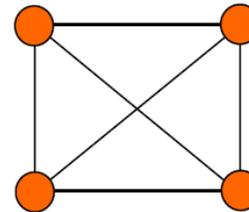
Subgraphs

A *subgraph* of a graph G is a graph H where $V(H) \subseteq V(G)$ and $E(H) \subseteq E(G)$.

Example

is a subgraph of
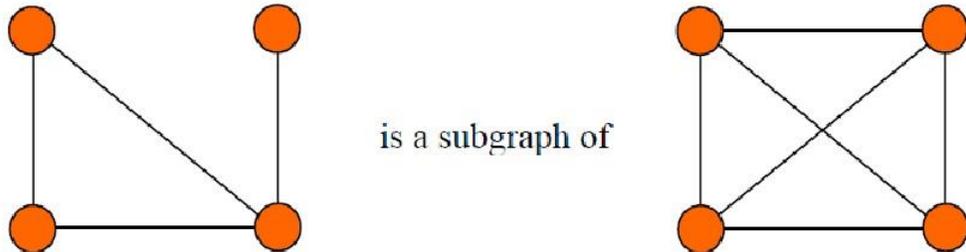
Connected Graphs

A graph is *connected* if there is a path between every pair of distinct vertices of the graph. An edge *uv* in a connected graph G is called a *bridge* if $G - uv$, the graph obtained by deleting *uv* from G, is not connected.

Example

A connected graph

A Bridge
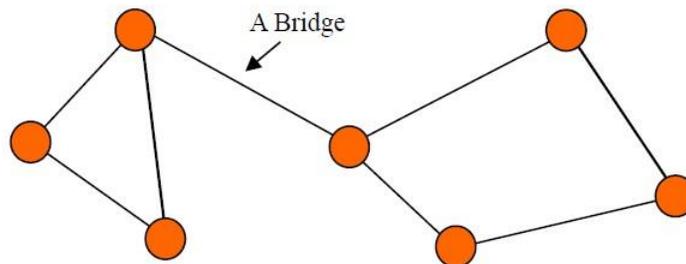
Euler Circuit and Euler Path

An *Euler circuit* in a graph G is a simple circuit containing every edge of G. An *Euler path* in G is a simple path containing every edge of G.

Graph Representations

Graph data structure is represented using following representations...

1. **Adjacency Matrix**
2. **Incidence Matrix**
3. **Adjacency List**

Adjacency Matrix

In this representation, graph can be represented using a matrix of size total number of vertices by total number of vertices. That means if a graph with 4 vertices can be represented using a matrix of 4X4 class. In this matrix, rows and columns both represents vertices. This matrix is filled with either 1 or 0. Here, 1 represents there is an edge from row vertex to column vertex and 0 represents there is no edge from row

vertex to column vertex.

For example, consider the following undirected graph representation...

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

Directed graph representation...

	A	B	C	D	E
A	0	1	1	0	0
B	0	0	0	1	1
C	0	0	0	1	0
D	1	0	0	1	1
E	0	0	0	0	0
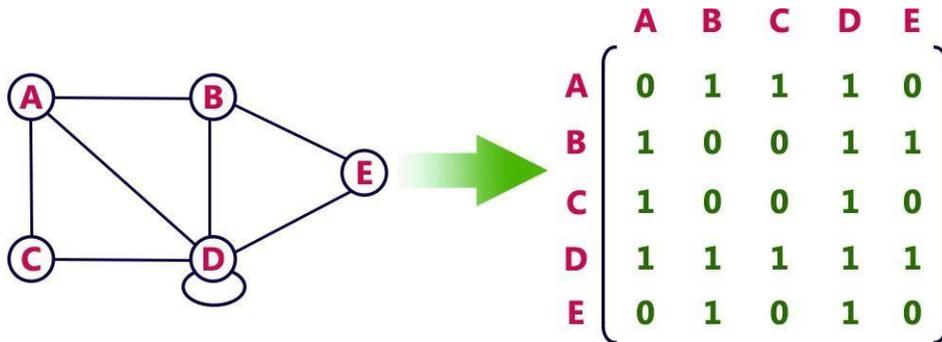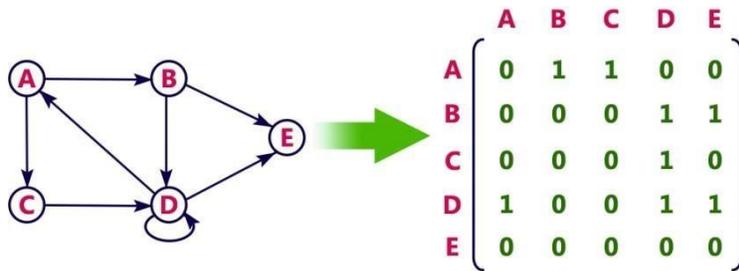
Incidence Matrix

In this representation, graph can be represented using a matrix of size total number of vertices by total number of edges. That means if a graph with 4 vertices and 6 edges can be represented using a matrix of 4X6 class. In this matrix, rows represents vertices and columns represents edges. This matrix is filled with either 0 or 1 or -1. Here, 0 represents row edge is not connected to column vertex, 1 represents row edge is connected as outgoing edge to column vertex and -1 represents row edge is connected as incoming edge to column vertex.

For example, consider the following directed graph representation...

	E1	E2	E3	E4	E5	E6	E7	E8
A	1	1	-1	0	0	0	0	0
B	-1	0	0	1	0	1	0	0
C	0	-1	0	0	1	0	0	0
D	0	0	1	-1	-1	0	1	1
E	0	0	0	0	0	-1	-1	0
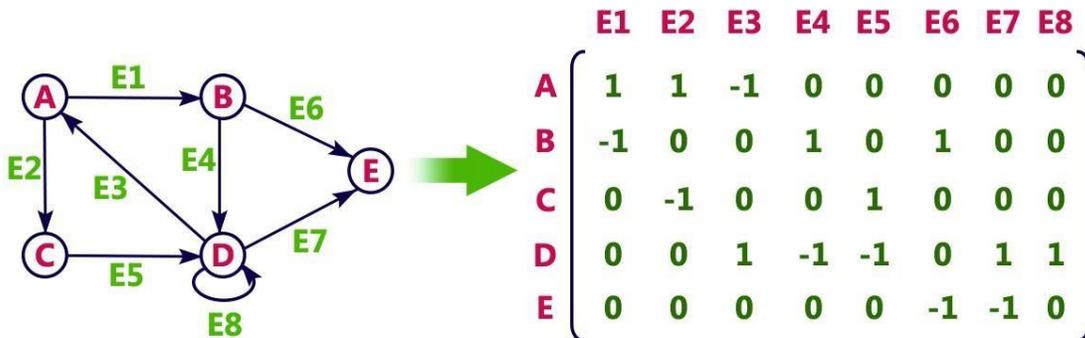
Adjacency List

In this representation, every vertex of graph contains list of its adjacent vertices.

For example, consider the following directed graph representation implemented using linked list...
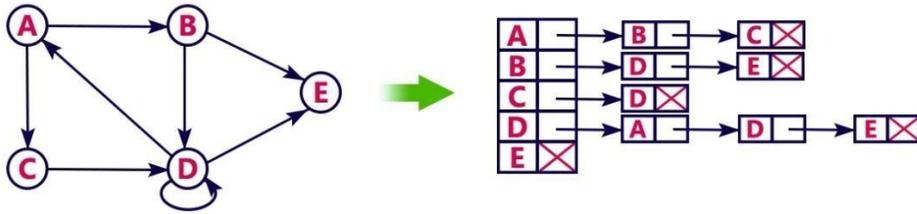
This representation can also be implemented using array as follows..
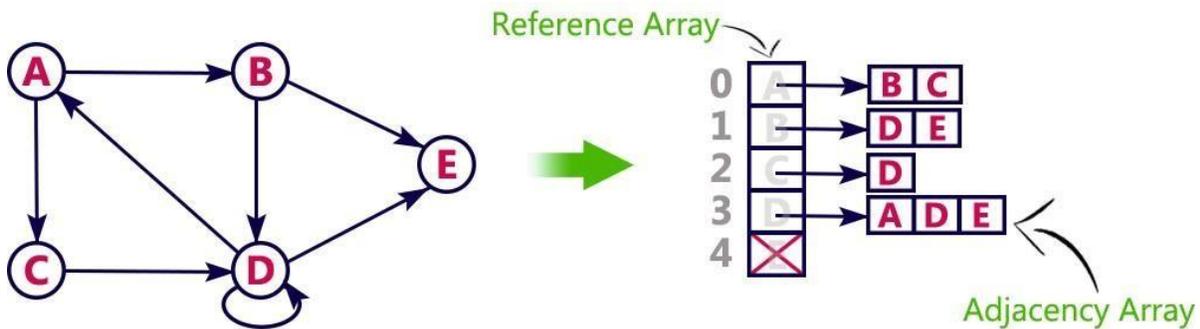
Graph traversals

Graph traversal means visiting every vertex and edge exactly once in a well-defined order. While using certain graph algorithms, you must ensure that each vertex of the graph is visited exactly once. The order in which the vertices are visited are important and may depend upon the algorithm or question that you are solving.

During a traversal, it is important that you track which vertices have been visited. The most common way of tracking vertices is to mark them.

Depth First Search (DFS)

The DFS algorithm is a recursive algorithm that uses the idea of backtracking. It involves exhaustive searches of all the nodes by going ahead, if possible, else by backtracking.

Here, the word backtrack means that when you are moving forward and there are no more nodes along the current path, you move backwards on the same path to find nodes to traverse. All the nodes will be visited on the current path till all the unvisited nodes have been traversed after which the next path will be selected.

This recursive nature of DFS can be implemented using stacks. The basic idea is as follows:
Pick a starting node and push all its adjacent nodes into a stack.
Pop a node from stack to select the next node to visit and push all its adjacent nodes into a stack.
Repeat this process until the stack is empty. However, ensure that the nodes that are visited are marked. This will prevent you from visiting the same node more than once. If you do not mark the nodes that are visited and you visit the same node more than once, you may end up in an infinite loop.

```
DFS-iterative (G, s):                          //Where G is graph and s is source vertex
    let S be stack
    S.push( s )           //Inserting s in stack
    mark s as visited.
    while ( S is not empty):
        //Pop a vertex from stack to visit next
        v = S.top( )
        S.pop( )
        //Push all the neighbours of v in stack that are not visited
        for all neighbours w of v in Graph G:
            if w is not visited :
                S.push( w )
                mark w as visited
```

```
DFS-recursive(G, s):
    mark s as visited
    for all neighbours w of s in Graph G:
        if w is not visited:
            DFS-recursive(G, w)
```


10
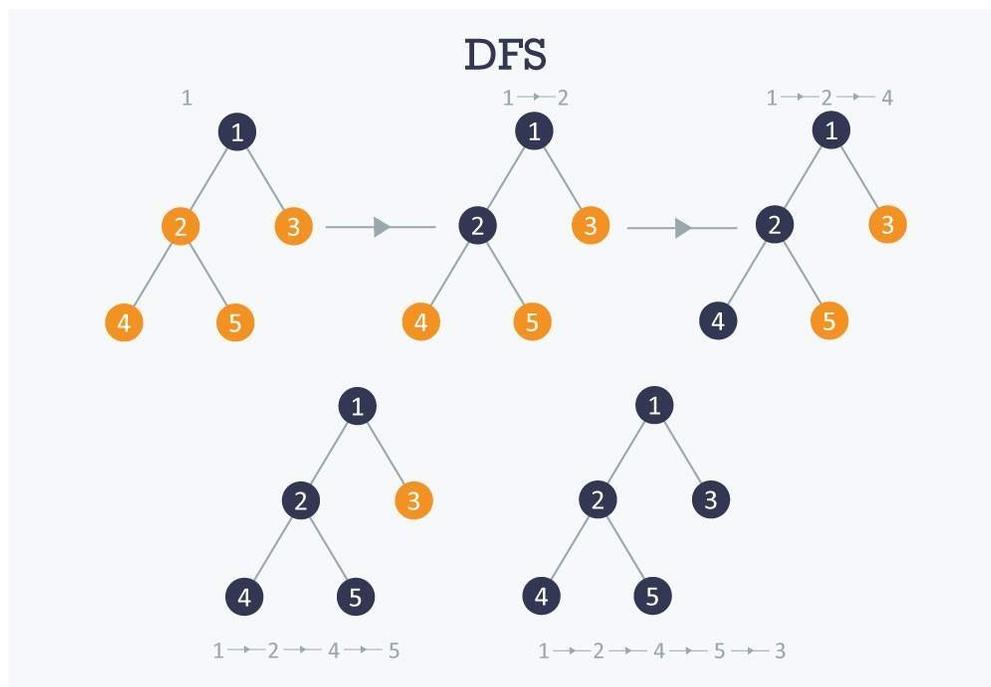
Breadth First Search (BFS);

There are many ways to traverse graphs. BFS is the most commonly used approach.BFS is a traversing algorithm where you should start traversing from a selected node (source or starting node) and traverse the graph layerwise thus exploring the neighbour nodes (nodes which are directly connected to source node). You must then move towards the next-level neighbour nodes.As the name BFS suggests, you are required to traverse the graph breadthwise as follows:

1.First move horizontally and visit all the nodes of the current layer
2.Move to the next layer
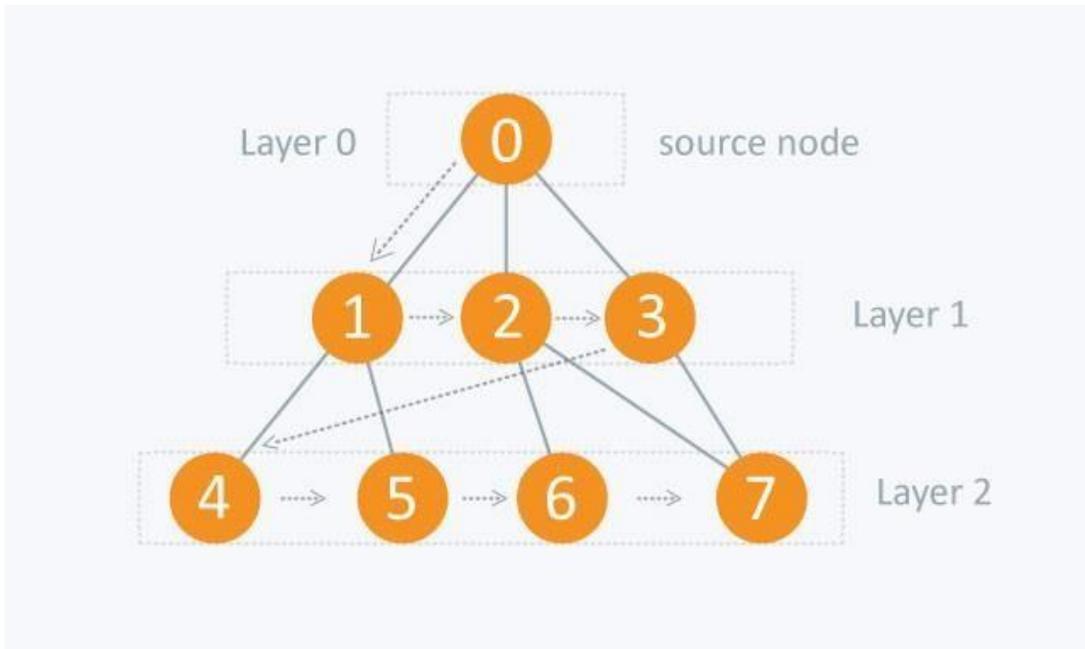
Merge sort is a sorting technique based on divide and conquer technique. In merge sort the unsorted list is divided into N sublists, each having one element, because a list of one element is considered sorted. Then, it repeatedly merge these sublists, to produce new sorted sublists, and at lasts one sorted list is produced. Merge Sort is quite fast, and has a time complexity of O(n log n).

Conceptually, merge sort works as follows:
1. Divide the unsorted list into two sub lists of about half the size.
2. Divide each of the two sub lists recursively until we have list sizes of length 1, in which case the list itself is returned.
3. Merge the two sub lists back into one sorted list.
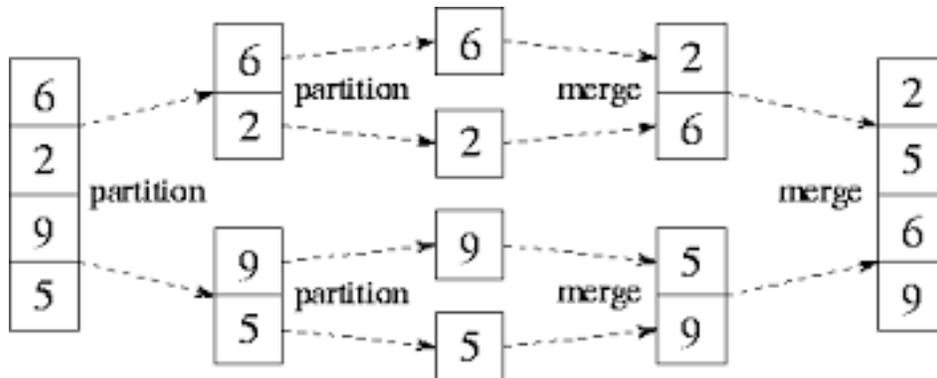

```
#include<iostream>
using namespace std;
void merge(int a[ ],int low,int mid,int high)
{
int temp[100];
```

```
int i,j,k;
        i=low;
        j=mid+1;
        k=low;
         while((i<=mid)&&(j<=high))
          {
                if(a[i]<=a[j])
                  {
                        temp[k]=a[i];
                         ++i;
                  }
                else
                 {
                        temp[k]=a[j];
                        ++j;
                 }
                 ++k;
          }
        if(i>mid)
        {
                while(j<=high)
                {
                        temp[k]=a[j];
                         ++j;
                        ++k;
                }
        }
        else
         {
                while(i<=mid)
                {
                        temp[k]=a[i];
                        ++i;
                        ++k;
                }
         }
        for(int i=low;i<=high;i++)
        a[i]=temp[i];
}

 void mergesort(int a[],int low,int high)
  {
 int mid;
 if(low<high)
  {
        mid=(low+high)/2;
        mergesort(a,low,mid);
        mergesort(a,mid+1,high);
        merge(a,low,mid,high);
        }
}
```

```
int main()
{
int n,i;
int list[30];
        cout<<"enter no of elements\n";
        cin>>n;
        cout<<"enter "<<n<<" numbers ";
        for(i=0;i<n;i++)
        cin>>list[i];
        mergesort (list,0,n-1);
        cout<<" after sorting\n";
        for(i=0;i<n;i++)
        cout<<list[i]<<"\t";
return 0;
}
```

RUN 1:

enter no of elements 5
enter 5 numbers 44 33 55 11 -1
  after sorting -1 11 33 44 55

Heap sort

It is a completely binary tree with the property that a parent is always greater   than or equal to either of its children (if they exist). first the heap (max or min) is created using binary tree and then heap is sorted using priority queue.

Steps Followed:

a) Start with just one element. One element will always satisfy heap property.
b) Insert next elements and make this heap.
c) Repeat step b, until all elements are included in the heap.
Steps of Sorting:
a) Exchange the root and last element in the heap.
b) Make this heap again, but this time do not include the last node.
c) Repeat steps a and b until there is no element left.

C++ program for implementation of Heap Sort

```
#include <iostream>
using namespace std;
// To heapify a subtree rooted with node i which is
// an index in arr[]. n is size of heap
void heapify(int arr[], int n, int i)
{
   int largest = i; // Initialize largest as root
   int L= 2*i + 1; // left = 2*i +  1
   int R= 2*i + 2; // right = 2*i + 2
```

10

```cpp
  // If left child is larger than root
  if (L < n && arr[L] > arr[largest])
    largest = L;
  // If right child is larger than largest so far
  if (R < n && arr[R] > arr[largest])
    largest = R;
  // If largest is not root
  if (largest != i)
  {
    swap(arr[i], arr[largest]);
    // Recursively heapify the affected sub-tree
    heapify(arr, n, largest);
  }
}

void heapSort(int arr[], int n)
{ int i;
  // Build heap (rearrange array)
  for ( i = n / 2 - 1; i >= 0; i--)
    heapify(arr, n, i);

  // One by one extract an element from heap
  for ( i=n-1; i>=0; i--)
  {
    // Move current root to end
    swap(arr[0], arr[i]);

    // call max heapify on the reduced heap
    heapify(arr, i, 0);
  }
}

/* A utility function to print array of size n */
void printArray(int arr[], int n)
{
  for (int i=0; i<n; ++i)
    cout << arr[i] << " ";
  cout << "\n";
}
int main()
{
int n,i;
int list[30];
    cout<<"enter no of elements\n";
    cin>>n;
    cout<<"enter "<<n<<" numbers ";
    for(i=0;i<n;i++)
    cin>>list[i];
    heapSort(list, n);
    cout << "Sorted array is \n";
  printArray(list, n);
```

return 0;
}
RUN 1:
enter no of elements 5
enter 5 numbers 11 99 22 101 1
Sorted array is
1 11 22 99 101

Time complexities:

Algorithm	Worst case	Average case	Best case
Bubble sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Insertion sort	$O(n^2)$	$O(n^2)$	$O(n^2)$
Quick sort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$
Merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Heap sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$
Linear search	$O(n)$	$O(n)$	$O(1)$
Binary search	$O(\log n)$	$O(\log n)$	$O(1)$

UNIT - V
PATTERN MATCHING & TRIES

Pattern matching :——

Pattern matching is the act of checking a given sequence of tokens (alphaties or symbols etc) for the presence of consti- -tuents of some patterns.

Means., pattern matching is to find a pattern which is relatively small, in a text which is to be very large.

- Pattern & text can be one-dimensional or two-dimensional.

- 1-dimensional example → text editor & DNA.

  Text editor :- have a 26 characters & some special symbols.

  DNA :- DNA has 4 characters.

  2-dimensional eg.,- computer vision.

- Either 1-dimensional (or) 2-dimensional the text is very large & therefore., a fast algorithm to find the occurance of pattern in it needed.

- In the classic string pattern matching problem we are given text string 'T' of length 'n'. and pattern string 'P' of length 'm'.

- To find whether `P` is a substring of `T`. The notation of a match is that there is substring of `T` starting at some index `i`. That matches, `P` character by character so that such as., $P[0] = T[i]$, $P[1] = T[i+1]$; $P[2] = T[i+2]$ ... ... $P[m-1] = T[i+n-1]$.

- The output of a pattern matching alg., could put either be some indication. That the pattern `P` does not exist in `T` or an integer indicating the starting index in `T` of a substring matching `P`.

<u>Applications of pattern matching</u>:—

* <u>Text editor</u>:— In text editor we have No. of lines of text data for finding required string from the editor we use string pattern match.

* <u>Search engine</u>:— The query submitted by the user in search engine uses the pattern matching.

* <u>Biological search</u>:— Eg: DNA

* Pattern matching is used to find the common ~~personali~~ functionalites of different research things.

Pattern matching algorithm:—

The popular pattern matching alg,'s are:-
• Naive pattern matching algorithm.
• Brute force algorithm.
• Boyer-moore algorithm.
• Knuth-morris pratt algorithm [KMP].

• Boyer-moore algorithm:—

Introduction: It is a pattern matching algorithm. It was developed by Robert S Boyer and J S. Moore in 1977.

Definition: The boyer-moore algorithm is an sufficient string search alg,. This alg,, preprocess the string being searched for the pattern but not the string being searched in the text.

It is well suitable for applications in which pattern is much shorter than the text, does persist (carry on) across multiple searches.

Purpose of Boyer-moore alg,,:
It is usually used in text editor & commands substitution.

Text editor :— In text editor for search & substitute commands implementation we use this alg,,.

Working Procedure :—

This alg„ compare's the pattern p with the sub string of sequence T with in a sliding window in the right to left order.

Means it can scan the characters of pattern from right to left beginning with the right most one.

This alg„ uses two heuristics (set of rules)

(1) Bad character rule. (BCR).

(2) Good suffix rule

(1) BCR :—

Suppose the P_1 is aligned to T_s. Now we perform a pair wise comparing b/w text 'T' & pattern 'P' from right to left.

Assume that the first mis-match occurs when comparing T_{s+j-1} with P_j

Since„ $T_{s+j-1} \neq P_j$, we moves the pattern P to the right. Such that the largest position 'c' in the left of P_j is equal to T_{s+j-1}.

We shift the pattern atleast j−c positions.

The diagram shows arrays T and P with characters and indices:

T: positions s, $s+j-1$, $s+j+1$ with characters x, t

P: with characters x, y, t at positions c, j, m labeled "mismatch" and "match"

Second row P: x, y, t at positions j, m
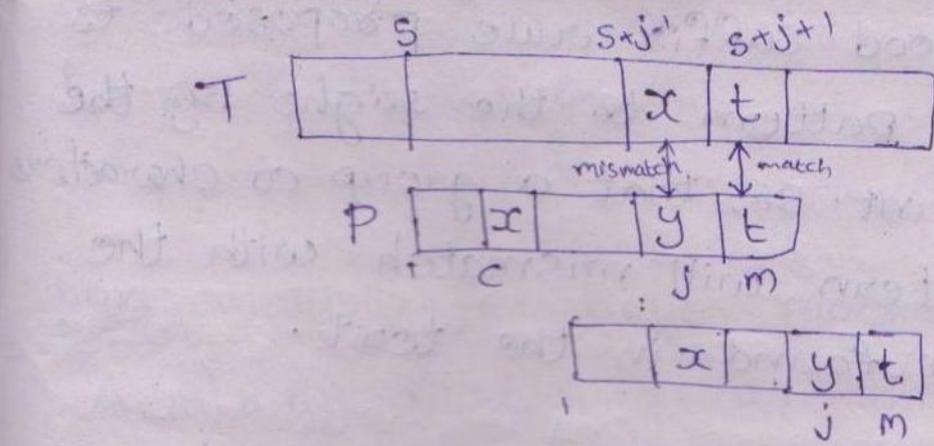
* BCR uses rule 2-1. [character matching rule].
* For any character x $ in T find the nearest x in P which is to be left of x in T.
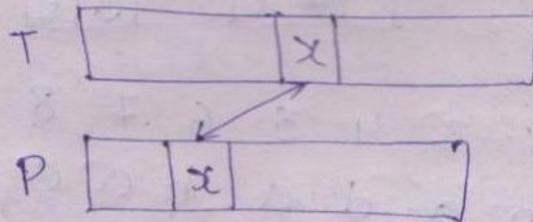
Move 'P' so that the two x's are match.
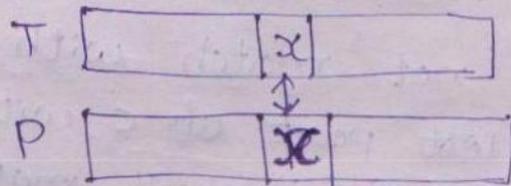
If no such a 'x' in P. consider, the partial window defined by x. In T & the string to the left of it.

Partial window

T: x

2) **Good Suffix rule:** —
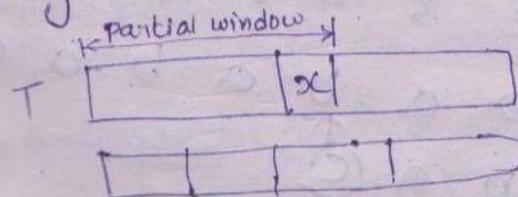
Good Suffix rule used when BCR failed.

The good suffix rule proposed to move the pattern to the right by the least amount. So, that a group of characters in the pattern will mismatch with the good suffix found in the text.

P $\rightarrow$ z t   y t

z t   y t

Eg:- 1

	0	1	2	3	4	5	6	7	8	9	10
T $\rightarrow$	a	b	b	a	d	a	b	a	c	b	a
P $\rightarrow$	b	a	b	a	ⓒ						

b a b a c

Here 'c' does not match with 'd' then compare the left part of c with left part with d if 'd' don't match with left part of c then shift the 'P' to right part of 'c'.

Eg:- 2

	0	1	2	3	4	5	6	7	8	9	10
T $\rightarrow$	a	b	b	a	b	a	b	a	c	b	a
P $\rightarrow$	b	a	b	a	ⓒ						

b a b a ⓒ (mismatch)

b a b a c (match)

case-3:

	0	1	2	3	4	5	6	7	8	9	10
eg. T →	a	b	a	a	b	a	b	a	c	b	a

P → c a b a b

c a b a b

Time analysis for boyer-moore algorithm:-

A string matching algorithm pre-process a pattern P [IPI=n] → size of pattern

For a text T [ITI=m] → no of characters in text.

Find all of the occurance of P in T.

Time complexity is $\boxed{O(n+m)}$

Right to left pattern matching.

Worst case complexity is $\boxed{O(n \cdot m)}$

Best case complexity is $\boxed{O(n/m)}$

(Time complexity is calculated by based on the no. of lines executed).

Knuth - Morris Pratt alg, (KMP):—

Intro: KMP is a pattern matching algorithm. It was conceived by Donald Knuth, Vaughan Prat in 1974. and

→ independently J. H. Morris.

→ The three published jointly in 1977.

Debinition:—

KMP alg,, searches for occurance a word 'w' within a main text 's' by employing. (applying)

the observation. That when a mismatch occurs the word itself embodies (represent) subbicient information to determine where

the next match could begin, that by
passing re-examination of previously
matched characters.

Working process:—

It is a tight analysis of naive algorithm.
KMP algorithm keeps the information that
naive approach wasted gathering during
the scan of the text.
By avoiding the waste of information it
acheives a running time of text is $O(m+n)$

Eg: W = ABCDABD

S = ABC ABCDAB, ABCDABCDABDE.

At given time., the algorithm is in a stage
determined by two integers.

1st: $m \rightarrow$ which denotes the position in text
which is the begining of a
perspective (future/potential) match for W.

2nd: $i \rightarrow$ The index in W denoting the character
currently under consideration.

In each step we compare $S[m+i]$
with $W[i]$ advance if they are equal.

Eg: m: 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2
S: ABC ABCDAB ABCDABCDABDE

W : A B C D A B D

i : 0 1 2 3 4 5 6

S[3] = Space ⎱ mismatch.
W[3] = D ⎰

We start at S[0] but it fail. Now we start from S[1]. But we note that no 'A' occurs b/w positions 0 & 3 expect at 0. having checked all those characters previously. we know that there is no chance of finding the beginning of a match.

∴ We move onto the next character. M=4, i=0

II) M : 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2

S : A B C   A B C D A B   A B C D A B C D A B D E

W :   A B C D A B D

i :   0 1 2 3 4 5 6.

S[10] = Space ⎱ mismatch.
W[6] = D ⎰

III)

M : 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2

S : A B C   A B C D A B   A B C D A B C D A B D E

W :   A B C D A B D

i :   0 1 2 3 4 5 6.

S[10] = Space ⎱ mismatch.
W[2] = C ⎰

IV) M : 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2

S : A B C A A B C D A B   A B C D A B C D A B D E

W :   A B C D A B D

i :   0 1 2 3 4 5 6

$$S[17] = C \quad \bigg\} \text{ mismatch.}$$
$$W[6] = D$$

v)

```
M:  0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2
S:  A B C   A B C D A B   A B C D A B C D A B D E
W:                                    A B C D A B D
i:                                    0 1 2 3 4 5 6
```

This time we are able to complete the match

- Whoose 1st character is <u>S[15]</u>.

<u>Algorithm:-</u>

Alg KMP-Search (S[ ], W[ ])

{

// I/p :- S → array of characters (The text. to be searched)

W → array of characters. (The text to be sought (looking for)

// 0/p :- an integer.
The ~~0~~ zero placed position in S at which W is found.

Integer: M ← 0 // The begining of the current match in S.

Integer: i ← 0 // The position of the current character in W.

Array integer : T // the table compute else where.

while (m + i < length (s) do
  if W[i] = S[m + i] then
    if i = length (w) - 1 then
```

```
        return m
           i ← i+1
    else
          m ← m+i - T[i]
     if T[i] > -1 then
             i ← T[i]
    else
           i ← 0
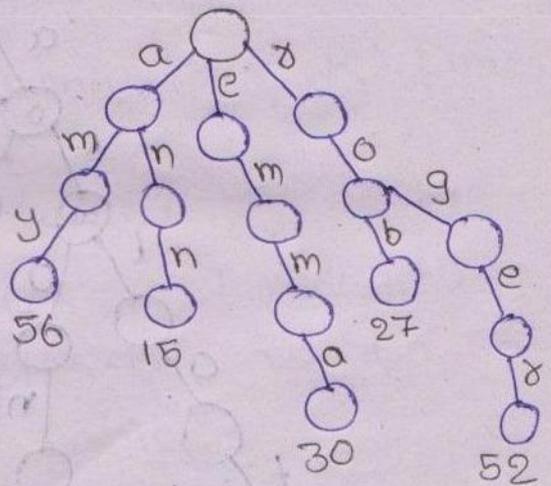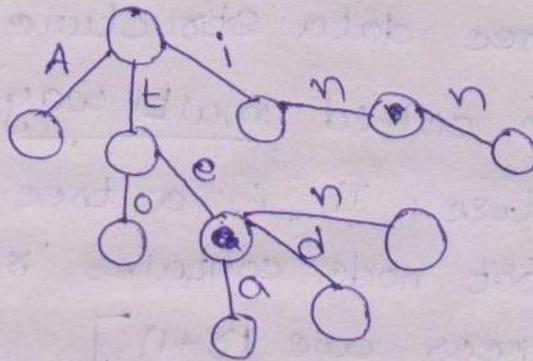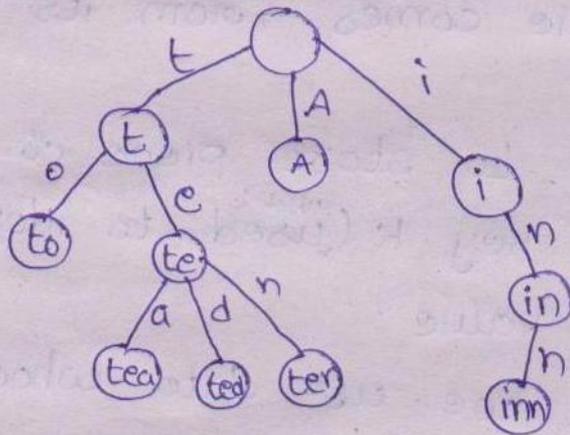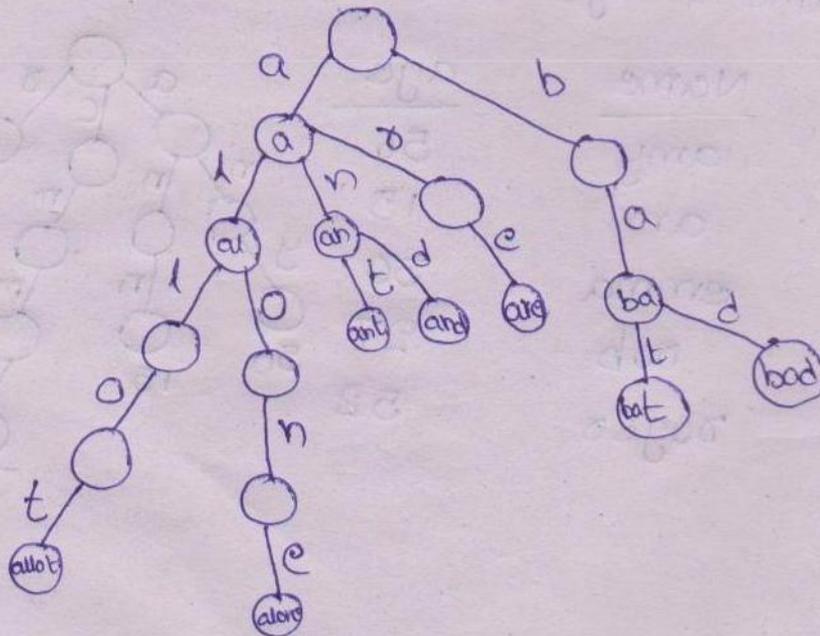
   end while.
```

If above alg,, return m value ...∴ ω found in S at position m. If above alg,, did not return m. we say we haved searched all of 'S' unsuccessful.

KMP - time analysis :-

For finding 'P' of size 'n' ($|P| = n$) In text T of size 'm' ($|T| = m$). By using naive alg,, time complexity → $O(n \cdot m)$

The KMP makes use of information gained by previous symbol comparisions. It never recompares a text symbol that has matched a pattern symbol. Time complexity for text is $O(m)$ & pattern is $O(n)$.

Overall time complexity is $O(m+n)$

Trie :—

Trie is an efficient information retrieval ordered tree data structure.

- It is also called multi way tree d·s

  [Multiway tree : It is a tree d·s · In this if the root node contains 'r' elements then the subtrees are (r+1)]

- The name trie comes from its use of retrieval.

- We use trie to store piece of data that have a key K (which is used to identify a data) & possibly a value.

  In trie we use data whoose keys are strings.

Ex:- Name & age for set of people.

Name	age
amy	56
ann	15
emma	30
rob	27
roger	52

Ex:- A trie for keys "A" to "tea" "ted" "i" ✗ "ten"

"in", "inn"

Another representation:—

Ex:- "allot", alone, ant, and, are, bat, bad.

Unlike., a binary search tree, no node in the tree stores the key associated with that node. Instead., its position in the tree defines the key with which it is associated.

· All the descendats of a node have a common prefix of the string associated with that node.

· Root is associated with empty string.

· Values are not associated with every node.

↗ Only with leab nodes & some inner nodes that corresponds to keys of insert.

Advantages :——

* The pattern matching can be done efficiently.

* In tries, the keys are searched using common prefix.

  ↳ It takes O(K) lookup → searching time. where k is the size of a key.

* Lookup can takes less than (<) K times if it is not there.

* Comparing with hash table :——

· Look up can be faster in time. in worst case time complexity as to compared with hash table.

· There is no collision in trie.

· There is no hash function in trie.

* comparing with BST :——

Disadvantages:

* Some times data retrieval of trie is very much slower than hash table.

* Representation of keys a string is complex-

Eg:- Representation of floating point numbers using string is really complicated in tries.

* It always takes more space.

* It is not available in programming tool.

Applications :——

• Tries has an ability to insert, delete, or search for the entries. Hence., there are used in building dictionaries. Such as., English words, Telephone numbers.

• These are also used in spelling check s/w

• These are well suited for approximate matching algorithm.

Digital Search tree (DST) :——

• A DST is a binary tree in which each node contains one element.

• The element to node assignment is

determined by the binary representation of element key.

- DST represents one possible data structure which allow us to store, search, & delete data using the key.

- DST is similar to BST but., the main difference is instead of comparing key values, they make use of the digital representation. of the keys.

- If the key can be represented as binary number. it makes seen sense to refer the bth bit of a key, where the bits are numbered from left to right.

- To insert a record (k, data) with key K into a DST, we said 'x' to set point to root. and "b" to 1.

Steps for DST:-

* Step-1: If x is nill, then store (k, data) in new node and terminate.

Step-2: If key(x) = K ,then terminate (K is already stored in the tree).

Step-3: Otherwise.,
  ↳ If bth bit of K is 0 then set x to left(x).
  ↳ If bth bit of K is 1 then set x to right(x).

Step-4: Set b to (b+1)

Step-5: Go to step-1

Example :—

Create a DST by using keys.,

1000 , 0010 , 1001 , 0001, 1100 , 0000

Ex-2)

insert 0110

insert 0010

insert 1001

insert 1011

insert 0000

Eg·3) create DST,

A-00001 , S-10011 , E-00101 , R-10010

C-00011 , H-01000, I-01001, N-01110

G-00111 , X-11000, M-01101, P-10000

Advantages of DST :—

In DST we compare keys bitwise, by this we avoid full length of key which is not suitable for searched element key. By this we eliminate time for searching full length of key.

Drawbacks of DST :—

* When we dealing with very large key, the cost of a key comparision is very high.

* To Solving this problem, we have to reduce no.of comparisions.

* For reducing no.of comparisions we using another data structure called "Patricia".

* This patricia structure is developed in three steps.

Step-1: First we introduce a structure called trie. binary trie.

Step-2: Then we transform binary-trie into compressed binary-tries.

Step-3: Finally from compressed binary tries we obtain patricia.

Binary trie :—

* A binary trie is a binary tree that has two kinds of nodes.

(i) Branch Node

(ii) Element Node.

(i) Branch Node: Branch node has two data members. * Left child
                                                    * Right child

(ii) Element Node:

Element node has single data member called Data.

+ Branch nodes are used to build a binary tree search structure. Similary to the DST This leads to element nodes.

Ex., for binary trie:

Compressed binary trie :-

* The binary trie's contains branch whoose degree is one.

* By adding another data member bit-number, to each branch node. We can eliminate all degree one branch node from the trie

* The bit-member data num, member of a branch nodes gives the bit-number of the key. i.e., to be used at at this node.

Patricia :—

Patricia

```
|_____ Pratical
 |_____ Algorithm
  |_____ to
   |_____ Retrieve
    |_____ information
     |_____ Code
      |_____ In
       |_____ Alphanumeric.
```

Introduction :—

It is related to 'trie' & developed by D.R. Morrison in 1968

Purpose :—

- This is for solving the comparision Problem in DST.
- It is also called <u>radix tree</u>.
  [Radix / base : is the no. of unique digits including zero].

Definition :—

Patricia is a space optimized trie d·s. where each node with only one child is merged with its parent.

This obtained from compressed binary trie in the following way.,

Step-1: Replace each branch node by a argumented branch node.

Step-2: Eliminate the element nodes.

Step-3: Store the data previously in the element node., data members of argumented branch node since every non-empty compressed binary tries has one less branch node then it has element nodes. It is necessary to add 1 argument branch node. This node is called head node.

Step-4: Replace the original pointers to the element node by pointers to be respective argumented branch nodes.

Multiway tries :—

Multiway tries is a one of the trie ds for fast data retrieving.

It is for come's over to the drawbacks for binary trie.

* Binary trie uses radix search with radix =2

* Coming to multiway tries, we uses radix search with $R > 2$

* Multiway tries sometimes called as R-ary tries.

* In each digit in a key has 'r'-bits, then the radix is $R = 2^r$.

* If the key have atmost B-bits, the worst case time for the no. of comparisions would be "B/r".

Ex-1) Keys are words made up of lower case letters in english. There are 26 different lower case letters in english. So, R-ary tries with $R=26$ could hold these keys. This type of tries sometimes rebbers as alphabet tries.

Properties :———

* The structure of multiway trie depends only on keys in it, not on the order in which they are inserted.

* Multiway tries have strong key ordering property.

 ⊙ At a node 'x', all keys in 'x' left most sub tree are smaller than keys in 'x'.

 ⊙ At a node x, all keys in x right most

Sub tree are larger than keys in x.

- So, tree order traversal of a multiway trie visit's keys in ~~shortest~~ sorted order.

* The worst case time for the no. of comparisions would be $\boxed{B/\gamma}$.

⊙ worst case height in BST contains $\boxed{N}$

⊙ worst case height in DST contains $\boxed{\log_R N}$

Drawbacks : —

There is a space cost disadvantage in multiway trie. ~~So, To solve~~

To solve this problem we go for another data structure <u>ternary</u> trie.