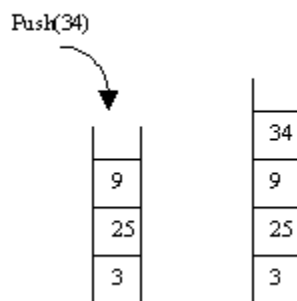


What is a Stack?

A *stack* is like a pile of plates where we can only remove a plate from the top and can only add a new plate on the top. In computer science we commonly place numbers on a stack, or perhaps place records on the stack.

The `Push` operation places a new data item on top of the stack. `Pop` removes the top item and sends it back via the parameter. Finally, the `Empty` operation tells us whether or not the stack object is empty, the `Full` operation tells us whether or not the stack object is full if stack is implemented using arrays.

Here are a picture of a small stack of integers and another picture of the stack after 34 has been pushed:



Top is a variable which always hold the top element of the stack.

A stack is called a "last in, first out" (LIFO) data structure since the last item pushed onto a stack will be the first one popped off. It can also be called a "first in, last out" (FILO) data structure since the first item pushed onto it will be the last one popped off. Because of these features, a stack is the ideal data structure to use in reversing a sequence of items.

//Stack implementation using arrays

```
#include<iostream>
using namespace std;
template<class T>
class AStack
{
    T *s;
    int ms,top;
public:
    AStack(int size=10)
    {
        if(size<1)
            throw "negative size not allowed";
        ms=size;
    }
};
```

```

        s=new T[ms];
        top=-1;
    }

    ~AStack()
    {
        delete []s;
    }
    inline bool IsEmpty(){return top==-1;}
    inline bool IsFull(){return top==ms-1;}
    inline T TopElement()
    {
        if(IsEmpty())
            throw "Stack is empty.No top element";
        return s[top];
    }
    void Push(T a);
    T Pop();
    void Display();
};

template<class T>
void AStack<T>::Push(T a)
{
    if(IsFull())
    {
        ms=ms*2;
        T *temp=new T[ms];
        for(int i=top;i>=0;i--)
            temp[i]=s[i];
        delete [] s;
        s=temp;
    }
    s[++top]=a;
}

template<class T>
T AStack<T>::Pop()
{
    if(IsEmpty())
        throw "stack is empty.No delete";
    return s[top--];
}

```

```

}
template<class T>
void AStack<T>::Display()
{
    if(IsEmpty())
        throw "Stack is empty.No elements";
    cout<<"\n STACK ELEMENTS are...\n";

    for(int i=top;i>=0;i--)
        cout<<s[i]<<endl;
}
#include"AStack.h"
int menu()
{
    int i;
    cout<<"\n\n1.push\n2.pop\n3.top element\n4.display\n5.exit";
    cout<<"\nEnter the option:";
    cin>>i;
    return i;
}
int main()
{
    AStack<int> obj(5);
    int ch=menu(),a;
    try{
        do
        {
            switch(ch)
            {
                case 1: {cout<<"Enter element:";
                        cin>>a;
                        obj.Push(a);
                        break;}
                case 2: {int x=obj.Pop();
                        cout<<x<<" is popped from the stack";
                        break;}
                case 3: {int y=obj.TopElement();
                        cout<<y<<" is the top element of the stack";
                        break;}
                case 4: {obj.Display();
                        break;}
            }
        }
    }
}

```

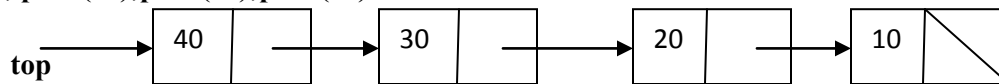
```

        ch=menu();
    }while(ch!=5);
} catch(const char *s)
{
    cout<<"EXCEPTION RAISED: "<<s<<endl;
}
return 0;
}

```

//Stack implementation using linked list

Push(10); push(20);push(30);push(40)



if we perform pop() operation node which is having 40 element is deleted & top will hold next node ie 30 node address.

```

#include<iostream>
using namespace std;
template<class T>
class LStack;
template<class T>
class node
{
    T ele;
    node<T> * next;
    friend class LStack<T>;
};
template<class T>
class LStack
{
    node<T> *top;
public:
    LStack(){top=NULL;}
    bool IsEmpty(){return top==NULL;};
    T TopElement()
    {

```

```
        if(IsEmpty())
        {
            cout<<"\nSTACK IS EMPTY";
            return -1;
        }
        return top->ele;
    }
    void Push(T a);
    T Pop();
    void Display();
};
template<class T>
void LStack<T>::Push(T a)
{
    node<T>* temp;
    temp=new node<T>;
    temp->ele=a;
    temp->next=top;
    top=temp;
}
template<class T>
T LStack<T>::Pop()
{
    if(IsEmpty())
    {
        cout<<"\n STACK UNDERFLOW";
        //return ;
    }
    node<T>* temp;
    T a=top->ele;
    temp=top;
    top=temp->next;
    delete temp;
    return a;
}
template<class T>
void LStack<T>::Display()
{
    if(IsEmpty())
        cout<<"\n STACK UNDERFLOW";
    else
    {
```

```

        for(node<T>* ptr=top;ptr!=NULL;ptr=ptr->next)
            cout<<ptr->ele<<endl;
    }
}
int menu()
{
    int i;
    cout<<"\n\n1.push\n2.pop\n3.top element\n4.display\n5.exit";
    cout<<"\nEnter the option:";
    cin>>i;
    return i;
}

int main()
{
    LStack<int> obj;
    int ch=menu(),a;
    do
    {
        switch(ch)
        {
            case 1: {cout<<"Enter element:";
                    cin>>a;
                    obj.Push(a);
                    break;}
            case 2: {int x=obj.Pop();
                    if(x!=-1)
                        cout<<x<<" is popped from the stack";
                    break;}
            case 3: {int y=obj.TopElement();
                    if(y!=-1)
                        cout<<y<<" is the top element of the stack";
                    break;}
            case 4: { obj.Display();break;}
        }
    }
}

```

```
        ch=menu();
    } while(ch!=5);
    return 0;
}
```

// Implementation of parenthesis check

```
#include "LStack.h"
```

```
void checkparen(char *a)
```

```
{
```

```
    LStack<char> s;
```

```
    char ch;
```

```
    int valid;
```

```
    for(int i=0;a[i]!='\0';i++)
```

```
    {
```

```
        if(a[i]=='(')
```

```
            s.Push(a[i]);
```

```
        else if(a[i]==')')
```

```
        {
```

```
            if(s.IsEmpty())
```

```
            {
```

```
                cout<<"\n More number of right parenthesis\n";
```

```
                return;
```

```
            }
```

```
                else
                {
                    ch=s.Pop();
                    if(ch=='(' && a[i]=='')
                        valid=1;
                }
            }
        }
        if(!s.IsEmpty())
            valid=0;
        if(valid==1)
            cout<<"\n Expression is valid";
        else
            cout<<"\n Expression id invalid";
    }
    int main()
    {
        char e[20];
        cout<<"\n Enter the expression:";
        cin>>e;
        checkparen(e);
        return 0;
    }
```

//infix to post fix implementation

```
#include "AStack.h"
#include <ctype.h>
int pred(char c)
{
    switch(c)
    {
        case '^': return 5;
        case '*':
        case '/':
        case '%': return 4;
        case '+':
        case '-': return 3;
        default: return 2;
    }
```



```

    }
}
void InfixToPost(char *e)
{
    AStack<char> obj(30);
    char post[30];
    int j=0;
    obj.Push('#');
    for(int i=0;e[i]!='\0';i++)
    {
        if(isalpha(e[i]))
            post[j++]=e[i];
        else if(e[i]=='(')
            obj.Push(e[i]);
        else if(e[i]==')')
        {
            while(obj.TopElement()!='(')
                post[j++]=obj.Pop();
            obj.Pop();
        }
        else
        {
            while(pred(e[i])<=pred(obj.TopElement()))
                post[j++]=obj.Pop();
            obj.Push(e[i]);
        }
    }
    while(obj.TopElement()!='#')
        post[j++]=obj.Pop();
    post[j]='\0';
    cout<<"\n Postfix expression is : "<<post<<endl;
}
int main()
{
    char e[20];
    cout<<"\n Enter Infix expression:";
    cin>>e;
    InfixToPost(e);
    return 0;
}
Enter Infix expression:a+b/c*(e*f/g)/h

```

INPUT	POST	STACK
a	a	#
+	a	+ #
b	ab	+ #
/	ab	+ #
c	abc	/+ #
*	abc/	*+ #
(abc/	(*+ #
e	abc/e	(*+ #
*	abc/e	*(+ #
f	abc/ef	*(+ #
/	abc/ef*	/(+ #
g	abc/ef*g	/(+ #
)	abc/ef*g	*+ #
/	abc/ef*g/*	/+ #
h	abc/ef*g/*h	/+ #

Still 2 operators are there those are popped & placed in the POST array variable.

Postfix expression is :abc/ef*g/*h/+

//infix to prefix implementation

```
#include "AStack.h"
#include <ctype.h>
#include <string.h>
int pred(char c)
{
```

```

        switch(c)
        {
            case '^': return 5;
            case '*':
            case '/':
            case '%': return 4;
            case '+':
            case '-': return 3;
            default: return 2;
        }
    }
}

void InfixToPrefix(char *e)
{
    AStack<char> opstk(30);
    AStack<char> alpstk(30);
    int len;
    len=strlen(e);
    opstk.Push('#');
    alpstk.Push('#');
    for(int i=len-1;i>=0;i--)
    {
        if(isalpha(e[i]))
            alpstk.Push(e[i]);
        else if(e[i]==')')
            opstk.Push(e[i]);
        else if(e[i]=='(')
        {
            while(opstk.TopElement()!='')
                alpstk.Push(opstk.Pop());
            opstk.Pop();
        }
        else
        {
            while(pred(e[i])<pred(opstk.TopElement()))
                alpstk.Push(opstk.Pop());
            opstk.Push(e[i]);
        }
    }
    while(opstk.TopElement()!='#')
        alpstk.Push(opstk.Pop());
    cout<<"\n Prefix expression is:";
    while(alpstk.TopElement()!='#')

```

```
        cout<<alpstk.Pop();
    cout<<endl;
}
int main()
{
    char e[30];
    cout<<"\n Enter Infix expression:";
    cin>>e;
    InfixToPrefix(e);
    return 0;
}
```

//evaluate the postfix expression

```
#include "AStack.h"
#include <ctype.h>
#include <math.h>
double operation(int x,int y,char ch);
void evalpost(char *p)
{
    AStack<int> obj(20);
    char ich;
    int t, opd1,opd2;
    for(int i=0;p[i]!='\0';i++)
    {
        ich=p[i];
        if(isalpha(ich))
        {
            cout<<"\n Enter value for "<<ich<<" : ";
            cin>>t;
            obj.Push(t);
        }
        else
        {
            opd2=obj.Pop();
            opd1=obj.Pop();
            int result;
            result=(int)operation(opd1,opd2,ich);
            obj.Push(result);
        }
    }
    cout<<"Result:"<<obj.Pop();
}
double operation(int x,int y,char ch)
{
    switch(ch)
    {
        case '+':return(x+y);
        case '-':return x-y;
        case '*':return x*y;
        case '/':return x/y;
        case '%':return x%y;
        case '$':return pow(x,y);
    }
}
```

```

    }
int main()
{
    char a[20];
    cout<<"\n Enter the postfix expression for evaluation :";
    cin>>a;
    evalpost(a);    return 0;    }

//Implementation of Evaluation of prefix expression
#include"AStack.h"
#include<ctype.h>
#include<math.h>
double operation(int x,int y,char ch)
{
    switch(ch)
    {
        case'+':return(x+y);
        case'-':return x-y;
        case'*':return x*y;
        case'/':return x/y;
        case'%':return x%y;
        case'$':return pow(x,y);
    }
}

void evalpre(char *p)
{
    AStack<int> obj(20);
    char ich;
    int t,opd1,opd2;
    int len=strlen(p),ele;
    for(int i=len-1;i>=0;i--)
    {
        ich=p[i];
        if(isalpha(ich))
        {
            cout<<"\n Enter value for "<<ich<<" : ";
            cin>>t;
            obj.Push(t);
        }
        else
        {
            opd1=obj.Pop();
            opd2=obj.Pop();
            int result=(int)operation(opd1,opd2,ich);
            obj.Push(result);
        }
    }
}

```

```

    }
    cout<<"\nresult:"<<obj.Pop();
}
int main()
{
    char a[20];
    cout<<"\n Enter the prefix expression for evaluation :";
    cin>>a;
    evalpre(a);
    return 0;    }

//rat in a maze
#include<iostream>
#include"AStack.h"
using namespace std;
enum directions{N,NE,E,SE,S,SW,W,NW};
struct offsets
{
    int a,b;    };
struct Items
{
    int x,y,dir;
    Items(){}
    Items(int i,int j,int d)
    {
        x=i;y=j;dir=d;    }
};
class StackApp
{
    bool **maze,**mark;
    int row,col;
    offsets move[8];
public:
    StackApp(int r,int c)
    {
        row=r;col=c;
        maze=new bool*[r+2];
        for(int i=0;i<r+2;i++)
            maze[i]=new bool[c+2];
        mark=new bool*[r+2];
        for(int i=0;i<r+2;i++)
            mark[i]=new bool[c+2];
        move[0].a=-1;move[0].b=0;move[1].a=-1;move[1].b=1;
        move[2].a=0;move[2].b=1;move[3].a=1;move[3].b=1;
        move[4].a=1;move[4].b=0;move[5].a=1;move[5].b=-1;
        move[6].a=0;move[6].b=-1;move[7].a=-1;move[7].b=-1;
    }
};

```

```

        //placing boundaries
    }
    void setboundaries()
    {
        for(int i=0;i<col+2;i++)
        {
            //placing boundaries in the first & last row
            maze[0][i]=1;maze[row+1][i]=1;
        }
        for(int i=0;i<row+2;i++)
        {
            //placing boundaries in the first & last col
            maze[i][0]=1;maze[i][col+1]=1;
        }
    }
    void readmaze()
    {
        setboundaries();
        cout<<"\n Give the maze input\n";
        for(int i=1;i<=row;i++)
        {
            cout<<"Enter "<<i<<" row info(only 1's & 0's):";
            for(int j=1;j<=col;j++)
                cin>>maze[i][j];
        }
        //placing all 0's in the mark[][]
        for(int i=0;i<row+2;i++)
        {
            for(int j=0;j<col+2;j++)
                mark[i][j]=0;
        }
    }
    void display()
    {
        cout<<" ";
        for(int i=0;i<col+2;i++)
            cout<<i<<" ";
        cout<<endl;
        for(int i=0;i<row+2;i++)
        {
            cout<<i<<" ";
            for(int j=0;j<col+2;j++)

```



```

        cout<<maze[i][j]<<" ";
    cout<<endl;
    }
}
void path()
{
    mark[1][1]=1;
    AStack<Items> s(row*col);
    Items temp(1,1,E);
    s.Push(temp);
    while(!s.IsEmpty())
    {
        temp=s.Pop();
        int i=temp.x;
        int j=temp.y;
        int d=temp.dir;
        while(d<8)
        {
            int g=i+move[d].a;
            int h=j+move[d].b;
            if(g==row&&h==col)
            {
                mark[g][h]=1;
                cout<<"\n WE GOT THE PATH.PATH IS....\n";
                cout<<"("<<g<<","<<h<<")"<<endl;
                cout<<"("<<i<<","<<j<<")"<<endl;
                while(!s.IsEmpty())
                {
                    temp=s.Pop();
                    cout<<"("<<temp.x<<","<<temp.y<<")"<<endl;
                }
                cout<<"\n see the path in the mark array where 1's are placed in
the path\n";
                for(int i=0;i<row+2;i++)
                {
                    for(int j=0;j<col+2;j++)
                        cout<<mark[i][j]<<" ";
                    cout<<endl;
                }
                return;
            }
        }
        if(!maze[g][h]&&!mark[g][h])

```

```

        {
            mark[g][h]=1;
            temp.x=i;
            temp.y=j;
            temp.dir=d+1;
            s.Push(temp);
            i=g;j=h;d=N;
        }
        else
            d++;
    }

    }
    cout<<"\n NO path exists\n";
}

};
main()
{
    StackApp obj(5,6);
    obj.readmaze();
    obj.display();
    obj.path();
}

```

What is a Queue?

A queue is a linear data structure in which inserting an element into the queue will be performed from rear side & deleting an element from the queue will be performed from the front side of the queue.

Insert(10);insert(20);insert(30)

10	20	30		
0	1	2	3	4
Front=0		rear=2		

Delete()

	20	30		
0	1	2	3	4
Front=1		rear=2		

//QUEUE implementation using arrays

```

#include<iostream>
using namespace std;
template<class T>
class AQueue
{
    T *q;

```

```

int rear,front,max;
public:
    AQueue(int maxsize)
    {
        max=maxsize;
        q=new T[max];
        rear=front=-1;
    }
    ~AQueue(){delete []q;}
    bool IsEmpty() const
    {
        if(front==-1 && rear==-1 || rear<front )
            return true;
        else
            return false;
    }
    bool IsFull()const
    {
        return rear==max-1;
    }
    T First()const
    {
        if(IsEmpty())
            throw "Queue is empty.No front element";
        return q[front];
    }
    T Last()const
    {
        if(IsEmpty())
            throw "Queue is empty.No rear element";
        return q[rear];
    }
    void Insert(T x);
    T Delete();
    void Display();
};

template<class T>
void AQueue<T>::Insert(T x)
{
    if(IsFull())
        throw "Queue is full";
    if(front==-1 && rear==-1)

```

```

        front=rear=0;
    else
        rear=rear+1;
    q[rear]=x;
}
template<class T>
T AQueue<T>::Delete()
{
    if(IsEmpty())
        throw "Queue is empty.No deletion";
    T ele=q[front];
    front=front+1;
    return ele;
}
template<class T>
void AQueue<T>::Display()
{
    if(IsEmpty())
        throw "Queue is empty.No elements";
    cout<<"\n Queue elements are....\n";
    for(int i=front;i<=rear;i++)
        cout<<q[i]<<"\t";
}
#include"AQueue.h"
int menu()
{
    int i;
    cout<<"1.Insert\n2.Delete\n3.Display\n4.First\n5.Last\n6.exit\n";
    cin>>i;
    return i;
}
int main()
{
    int i,ele,ch;
    AQueue<int> obj(5);
    ch=menu();
    try{
        do{
            switch(ch)
            {
                case 1:cout<<"\n Enter element:";
                    cin>>ele;

```

```
                obj.Insert(ele);
                break;
            case 2:cout<<obj.Delete()<<" is deleted from the queue\n";
                break;
            case 3:obj.Display();break;
            case 4:cout<<"\n First element is "<<obj.First();
                break;
            case 5:cout<<"\n Last element is "<<obj.Last();
                break;
        }
        ch=menu();
    }while(ch!=6);
} catch(const char *s)
{
    cout<<"\nException raised:"<<s<<endl;
}
return 0;
}
```

//QUEUE implementation using linked list

```
#include<iostream>
using namespace std;
template<class T>
class LQueue;
template<class T>
class node
```

```

{
    T data;
    node<T> * next;
    friend class LQueue<T>;
};
template<class T>
class LQueue
{
    node<T> *front,*rear;
public:
    LQueue()
    {
        rear=front=NULL;
    }
    ~LQueue()
    {
        node<T>* ptr;
        for(ptr=front;ptr!=NULL;ptr=ptr->next)
            delete ptr;
        delete front;
        delete rear;
    }
    bool IsEmpty() const
    {
        return front==NULL;
    }
    T First() const
    {
        if(IsEmpty())
            throw "Queue is empty.No front element";
        return front->data;
    }
    T Last() const
    {
        if(IsEmpty())
            throw "Queue is empty.No rear element";
        else
            return rear->data;
    }
    void Insert(T x);
    T Delete();
    void Display();

```

```

};
template<class T>
void LQueue<T>::Insert(T x)
{
    node<T>* temp;
    temp=new node<T>;
    temp->data=x;
    temp->next=NULL;
    if(front==NULL)
        front=temp;
    else
        rear->next=temp;
    rear=temp;
}
template<class T>
T LQueue<T>::Delete()
{
    if(IsEmpty())
        throw "Queue is empty.No deletion";
    node<T>* temp;
    temp=front;
    T x=front->data;
    front=front->next;
    delete temp;
    return x;
}
template<class T>
void LQueue<T>::Display()
{
    if(IsEmpty())
        throw "Queue is empty.No element";
    node<T>* ptr;
    for(ptr=front;ptr!=NULL;ptr=ptr->next)
        cout<<ptr->data<<"\t";
}
#include"LQueue.h"
int menu()
{
    int i;
    cout<<"1.Insert\n2.Delete\n3.Display\n4.First\n5.Last\n6.exit\n";
    cin>>i;
    return i;
}

```

```

}
int main()
{
    int i,ele,ch;
    LQueue<int> obj;
    ch=menu();
    try{
        do{
            switch(ch)
            {
                case 1:cout<<"\n Enter element:";
                    cin>>ele;
                    obj.Insert(ele);
                    break;
                case 2:cout<<endl<<obj.Delete()<<" is deleted from the queue\n";break;
                case 3:obj.Display();break;
                case 4:cout<<"\n First element is "<<obj.First()<<endl;break;
                case 5:cout<<"\n Last element is "<<obj.Last()<<endl;break;
            }
            ch=menu();
        }while(ch!=6);
    }catch(const char *s)
    {
        cout<<"Exception raised:"<<s<<endl;
    }
    return 0;
}

```


//CIRCULAR QUEUE implementation

```
#include<iostream>
using namespace std;
template<class T>
class ACQueue
{
    T *q;
    int front,rear,max;
public:
    ACQueue(int size)
    {
        if(size<0)
            throw "negative size for the array is not allowed";
        max=size;
        q=new T[max];
        front=rear=0;
    }
    ~ACQueue(){ delete []q;}
    bool IsEmpty() const
    {
        return rear==front;
    }
    bool IsFull() const
    {
        if((rear+1)%max==front)
            return true;
        else
            return false;
    }
    T First()
    {
        if(IsEmpty())
            throw "Queue is empty.no first element";
        return q[(front+1)%max];
    }
    T Last()
    {
```

```

        if(IsEmpty())
            throw"Queue is empty.no last element";
        else
            return q[rear];
    }
    void Insert(T x);
    T Delete();
    void Display();
};

template<class T>
void ACQueue<T>::Insert(T x)
{
    if(IsFull())
    {
        T *nq=new T[2*max];
        int j=0;
        if(front>rear)
        {
            for(int i=(front+1)%max;i<max;i++,j++)
                nq[j]=q[i];
            for(int i=0;i<=rear;i++,j++)
                nq[j]=q[i];
        }
        else
        {
            for(int i=(front+1)%max;i<=rear;i++,j++)
                nq[j]=q[i];
        }
        front=2*max-1;
        rear=max-2;
        max=max*2;
        delete[]q;
        q=nq;
        //throw"QUEUE IS OVERFLOW.no insertion";
    }
    rear=(rear+1)%max;
    q[rear]=x;
    cout<<"\n f="<<front<<"\tr="<<rear<<endl;
}

template<class T>
T ACQueue<T>::Delete()
{

```

```

        if(IsEmpty())
            throw"QUEUE IS UNDERFLOW.no delete";
        front=(front+1)%max;
        cout<<"\n f="<<front<<"\tr="<<rear<<endl;
        return q[front];
    }
template<class T>
void ACQueue<T>::Display()
{
    if(IsEmpty())
        throw"QUEUE IS UNDERFLOW.no elements";
    if(front>rear&&front!=max-1)
    {
        for(int i=(front+1)%max;i<max;i++)
            cout<<q[i]<<"\t";
        for(int i=0;i<=rear;i++)
            cout<<q[i]<<"\t";
    }
    else
    {
        for(int i=(front+1)%max;i<=rear;i++)
            cout<<q[i]<<"\t";
    }
}
}

```

//Polynomial addition using arrays

The Term class object is having 2 fields coef & exp in which 1 term of a polynomial can be stored. If we want to store one polynomial get the number of terms in the polynomial and create an array of term objects.

Example:

$$3x^2+2x+5$$

Term *ta=new Term[3]

ta[0]		ta[1]		ta[2]	
coef	exp	coef	exp	coef	exp
3	2	2	1	5	0

```
#include<iostream>
using namespace std;
class poly;
class Term
{
    friend class poly;
    int coef,exp;
};
class poly
{
    Term *ta;
    int maxsize,nt;
public:
    poly()
    {
```

```

        maxsize=1;nt=0;
        ta=new Term[maxsize];
    }
    poly(int ms,int n)
    {
        maxsize=ms;nt=n;
        ta=new Term[maxsize];
    }
    void create();
    poly add(poly);
    void display();
    void attach(const int,const int);
};

void poly::create()
{
    for(int i=0;i<nt;i++)
    {
        cout<<"\nEnter the coefficient & exponent :";
        cin>>ta[i].coef;
        cin>>ta[i].exp;
    }
}

void poly::display()
{
    for(int i=0;i<nt;i++)
    {
        cout<<ta[i].coef<<"X^"<<ta[i].exp;
        if(i!=(nt-1))
            cout<<" + ";
    }
}

poly poly::add(poly b)
{
    int apos=0,bpos=0;
    poly c;
    while((apos<nt)&&(bpos<b.nt))
    {
        if(ta[apos].exp==b.ta[bpos].exp)
        {
            int t=ta[apos].coef + b.ta[bpos].coef;

```

```

        if(t)
            c.attach(t,ta[apos].exp);
        apos++;bpos++;
    }
    else if(ta[apos].exp<b.ta[bpos].exp)
    {
        c.attach(b.ta[bpos].coef,b.ta[bpos].exp);
        bpos++;
    }
    else
    {
        c.attach(ta[apos].coef,ta[apos].exp);
        apos++;
    }
}
for(;apos<nt;apos++)
    c.attach(ta[apos].coef,ta[apos].exp);
for(;bpos<b.nt;bpos++)
    c.attach(b.ta[bpos].coef,b.ta[bpos].exp);
return c;
}
void poly::attach(const int c,const int e)
{
    if(nt==maxsize)
    {
        maxsize*=2;
        Term *temp=new Term[maxsize];
        for(int i=0;i<nt;i++)
            temp[i]=ta[i];
        delete []ta;
        ta=temp;
    }
    ta[nt].coef=c;ta[nt].exp=e;
    nt++;
}
main()
{
    poly p1(3,3),p2(4,4);
    poly p3;
    cout<<"\n Enter first polynomial:";
    p1.create();
    cout<<"\n Enter second polynomial:";

```

```

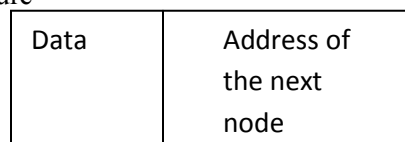
    p2.create();
    p3=p1.add(p2);
    cout<<"\nfirst polynomial is.....\n";
    p1.display();
    cout<<"\nsecond polynomial is.....\n";
    p2.display();
    cout<<"\nthird polynomial is.....\n";
    p3.display();
    cout<<endl;
}

```

//poly add using chain iterator & linked list

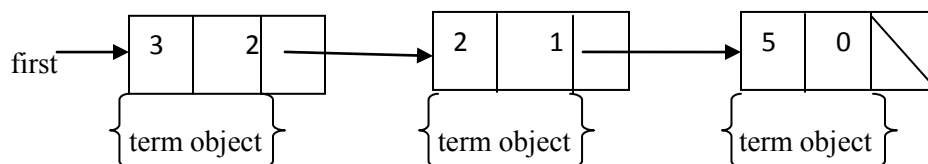
In linked list representation for each term a node will be created and coef & exp of the term will be placed in the node. And all terms related to a polynomial will be linked each other to form a single linked list

Node structure



Example:

$3x^2+2x+5$



The type of data field in the node is a term object in which coef & exp will be there.

```
#include<iostream>
using namespace std;
template<class T>
class Chain;
template<class T>
class CNode
{
    T data;
    CNode<T> *next;
    friend class Chain<T>;
public:
    CNode(T e=0,CNode<T>* p=0)    {        data=e;next=p;        }
};
template<class T>
class Chain
{
    CNode<T> *first,*last;
public:
    class ChainIterator
    {
        CNode<T> *current;
    public:
        ChainIterator(CNode<T> *sn=0)
        {
            current=sn;
        }
        T operator->(){return current->data;}
        ChainIterator& operator++()
        {
            current=current->next; return *this;
        }
        CNode<T> * getNextNode()
        {
            return current->next;
        }
        bool operator!=(const ChainIterator right) const
        {
            return current!=right.current;
        }
    };
};
```



```

Chain(){ first=0;last=0;}
void InsertLast(T &ele)
{
    CNode<T> *temp=new CNode<T>(ele);
    if(last)
    {
        temp->next=last->next;
        last->next=temp;
        last=temp;
    }
    else
        first=last=temp;
}
void display()
{
    for(CNode<T> *ptr=first;ptr!=0;ptr=ptr->next)
    {
        cout<<ptr->data;
        if(ptr->next!=0)
            cout<<"-->";
    }
}
ChainIterator begin(){ return ChainIterator(first);}
ChainIterator end() { return ChainIterator(0);}

};
#include"chainiterator.h"
struct Term
{
    int coef,exp;
    Term set(int c,int e)
    {
        coef=c;exp=e;
        return *this;
    }
};
class polynomial
{
    Chain<Term> poly;
public:
    void createpoly()
    {

```

```

    int c,e;
    cout<<"\n enter coef & exp (give exp as -1 for exit):";
    cin>>c>>e;
    while(e!=-1)
    {
        Term t;
        poly.InsertLast(t=t.set(c,e));
        cout<<"\n t.c="<<t.coef<<"\t t.e="<<t.exp;
        cout<<"\n enter coef & exp (give exp as -1 for exit):";
        cin>>c>>e;
    }
}

void display()
{
    Chain<Term>::ChainIterator ptr=poly.begin();
    for(;ptr!=0;++ptr)
    {
        cout<<ptr->coef<<"X^"<<ptr->exp;
        if(ptr.getNextNode()!=0)
            cout<<" + ";
    }
}

polynomial& operator+(polynomial &b)
{
    polynomial c;
    Term t;
    Chain<Term>::ChainIterator ai=poly.begin();
    Chain<Term>::ChainIterator bi=b.poly.begin();
    while((ai!=0)&&(bi!=0))
    {
        if(ai->exp==bi->exp)
        {
            int sum=ai->coef+bi->coef;
            if(sum)
                c.poly.InsertLast(t=t.set(sum,ai->exp));
            ++ai;++bi;
        }
        else if(ai->exp<bi->exp)
        {
            c.poly.InsertLast(t=t.set(bi->coef,bi->exp));
            ++bi;
        }
    }
}

```

```

        else
        {
            c.poly.InsertLast(t=t.set(ai->coef,ai->exp));
            ++ai;
        }
    }
    while(ai!=0)
    {
        c.poly.InsertLast(t=t.set(ai->coef,ai->exp));
        ++ai;
    }
    while(bi!=0)
    {
        c.poly.InsertLast(t=t.set(bi->coef,bi->exp));
        ++bi;
    }
    return c;
}

};

main()
{
    polynomial p1,p2;
    cout<<"\n Enter first polynomial\n";
    p1.createpoly();
    cout<<"\n Enter second polynomial\n";
    p2.createpoly();
    cout<<"\n First polynomial ....\n";
    p1.createpoly();
    cout<<"\n Enter second polynomial\n";
    p2.createpoly();
    cout<<"\n First polynomial ....\n";
    p1.display();
    cout<<"\n second polynomial ....\n";
    p2.display();
    polynomial p3=p1+p2;
    cout<<"\n third polynomial ....\n";
    p3.display();
    cout<<endl;
}

```

Binary Search

The binary search algorithm **can only be applied if the data are sorted**. You can exploit the knowledge that they are sorted to speed up the search.

The idea is analogous to the way people look up an entry in a dictionary or telephone book. You don't start at page 1 and read every entry! Instead, you turn to a page somewhere about where you expect the item to be. If you are lucky you find the item straight away. If not, you know which part of the book will contain the item (if it is there), and repeat the process with just that part of the book.

If you always split the data in half and check the middle item, you halve the number of remaining items to check each time. This is much better than linear search, where each unsuccessful comparison eliminates just one item.

Seek the value 123

2 6 7 34 76 123 234 567 677 986
 ↑ ↑ ↑
 first (1) mid (5) (10) last

2 6 7 34 76 123 234 567 677 986
 ↑ ↑ ↑
 (6) first (8) mid (10) last

2 6 7 34 76 123 234 567 677 986
 ↑↑ ↑
 (6) first last (7)
 mid

2 6 7 34 76 123 234 567 677 986
 ↑↑↑
 first mid last (6)

//Binary Search Implementation recursive

```
template<class T>
int Array<T>::bsearch(T key,int low,int high)
{
    if(low<=high)
    {
        int mid=(low+high)/2;
        if(key>a[mid])
            return bsearch(key,mid+1,high);
        else if(key<a[mid])
            return bsearch(key,low,mid-1);
        else
            return mid;
    }
    return -1;
}
```

//Binary Search Implementation non-recursive

```
template<class T>
void Array<T>::bsearch(T key)
{
    int low=0,high=ms-1;
    while(low<=high)
    {
        int mid=(low+high)/2;
        if(key>a[mid])
            low=mid+1;
        else if(key<a[mid])
            high=mid-1;
        else
        {
            cout<<"\n element found";
            return;
        }
    }
    cout<<"\n element not found";
}
```

Hashing

The Division Method

Map a key k into one of m slots by taking the remainder of k divided by m . That is, the hash function is
$$h(k) = k \bmod m.$$

Example:

If table size $m = 12$

key $k = 100$

than

$$\begin{aligned} h(100) &= 100 \bmod 12 \\ &= 4 \end{aligned}$$

these are the keys that are to be stored in 0-8 memory locations in an array:

5, 28, 19, 15, 20, 33, 12, 17, 10

slots: 9

hash function = $h(k) = k \bmod 9$

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

$$h(28) = 28 \bmod 9 = 1$$

0	1	2	3	4	5	6	7	8
	28			5				

$$h(19) = 19 \bmod 9 = 1$$

Collision occurred. Collision is nothing but as keys are inserted in the table, it is possible that two keys may hash to the same table slot. If the hash function distributes the elements uniformly over the table, the number of conclusions cannot be too large on the average, but the birthday paradox makes it very likely that there will be at least one collision, even for a lightly loaded table.

There are two basic methods for handling collisions in a hash table: **Chaining** and **Open addressing**.

Open addressing or Linear probing is if $H(k)=h$ ie key k has to placed in h location, if it is not free place key k in $h+1$, if it not free place key k in $h+2$ and soon

Ie $h, h+1, h+2, h+3, \dots$

$h(19) = 19 \bmod 9 = 1$ place 19 in 2nd location because 1st location is not free.

0	1	2	3	4	5	6	7	8
	28	19		5				

$h(15) = 15 \bmod 9 = 6$

$h(20) = 20 \bmod 9 = 2$

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

$h(12) = 12 \bmod 9 = 3$

$h(17) = 17 \bmod 9 = 8$

$h(10) = 10 \bmod 9 = 1$

0	1	2	3	4	5	6	7	8
10	28	19	20	5	12	15	33	17

// implementation of hashing using modulo division & collision solving using open addressing

```
#include<iostream>
```

```
using namespace std;
```

```
class Hash
```

```
{
```

```
    int *ht;
```

```
    bool *empty;
```

```
    int D;
```

```
    public:
```

```
        Hash(int size)
```

```
        {
```

```
            D=size;
```

```
            ht=new int[D];
```

```
            empty=new bool[D];
```

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

```
            {
```

```
                ht[i]=0;
```

```
                empty[i]=true;
```

```
            }
```

```
        }
```

```
        void insert(int ele)
```

```
        {
```

```
            int i=ele%D;
```

```
            if(empty[i])
```

```
            {
```

```
                ht[i]=ele;
```

```

        empty[i]=false;
    }
    else
    {
        int j=(i+1)%D;
        do{
            if(empty[j])
            {
                ht[j]=ele;
                empty[j]=false;
                return;
            }
            j=(j+1)%D;
        }while(j!=i);
        cout<<"\n Hash Table is Full";
    }
}

int search(int ele)
{
    int i=ele%D;
    int j=i;
    do{
        if(ht[i]==ele&&!empty[i])
            return i;
        i=(i+1)%D;
    }while(i!=j);
    return -1;
}

void Delete(int ele)
{
    int i=search(ele);
    if(i!=-1)
    {
        ht[i]=0;
        empty[i]=true;
        cout<<"\n Deleted element is :"<<ele;
    }
    else
        cout<<"\n Element does not exist";
}

void display()
{

```



```

        cout<<"\n Hash Table elements are...\n";
        for(int i=0;i<D;i++)\
            cout<<ht[i]<<"\t";
    }
};
int menu()
{
    int i;
    cout<<"1.Insert\n2.Delete\n3.Display\n4.search\n5.exit\n";
    cin>>i;
    return i;
}
int main()
{
    int i,ele,ch;
    Hash obj(11);
    ch=menu();
    do{
        switch(ch)
        {
            case 1:cout<<"\n Enter element:"; cin>>ele;
                    obj.insert(ele); break;
            case 2:cout<<"\n Enter the element to be deleted:";
                    cin>>ele;    obj.Delete(ele);break;
            case 3:obj.display();break;
            case 4:cout<<"\n enter the element to be searched:";
                    cin>>ele;i=obj.search(ele);
                    if(i!=-1)
                        cout<<"\n element found at "<<i;
                    else
                        cout<<"\n Element not found";
                    break;
        }
        ch=menu();
    }while(ch!=5);
    return 0;
}

```

//All Sortings

```
#include<iostream>
using namespace std;
template<class T>
class Sorting
{
    int *a;
    int n;
public:
    Sorting(int size)
    {
        n=size;
        a=new int[n];
    }
    void Read();
    void Display();
    void inssort();
    void Bubblesort();
    void selectionsort();
}
```

```

        int largest(int size);
};
template<class T>
void Sorting<T>::Read()
{
    for(int i=0;i<n;i++)
        cin>>a[i];
}
template<class T>
void Sorting<T>::Display()
{
    for(int i=0;i<n;i++)
        cout<<a[i]<<"\t";
}


```

Insertion Sorting


The first class of sorting algorithm that we consider comprises algorithms that *sort by insertion*. An algorithm that sorts by insertion takes the initial, unsorted sequence, $S = \{s_1, s_2, s_3, \dots, s_n\}$, and computes a series of *sorted* sequences $S'_0, S'_1, S'_2, \dots, S'_n$, as follows:

1. The first sequence in the series, S'_0 , is the empty sequence. That is, $S'_0 = \{\}$.
2. Given a sequence S'_i in the series, for $0 \leq i < n$, the next sequence in the series, S'_{i+1} , is obtained by inserting the $(i+1)^{th}$ element of the unsorted sequence s_{i+1} into the correct position in S'_i .

Each sequence S'_i , $0 \leq i < n$, contains the first i elements of the unsorted sequence S . Therefore, the final sequence in the series, S'_n , is the sorted sequence we seek. That is, $S' = S'_n$.

Figure  illustrates the insertion sorting algorithm. The figure shows the progression of the insertion sorting algorithm as it sorts an array of ten integers. The array is sorted *in place*. That is, the initial unsorted sequence, S , and the series of sorted sequences, S'_0, S'_1, \dots , occupy the same array.

In the i^{th} step, the element at position i in the array is inserted into the sorted sequence S'_i which occupies array positions 0 to $(i-1)$. After this is done, array positions 0 to i contain the $i+1$ elements of S'_{i+1} . Array positions $(i+1)$ to $(n-1)$ contain the remaining $n-i-1$ elements of the unsorted sequence S .

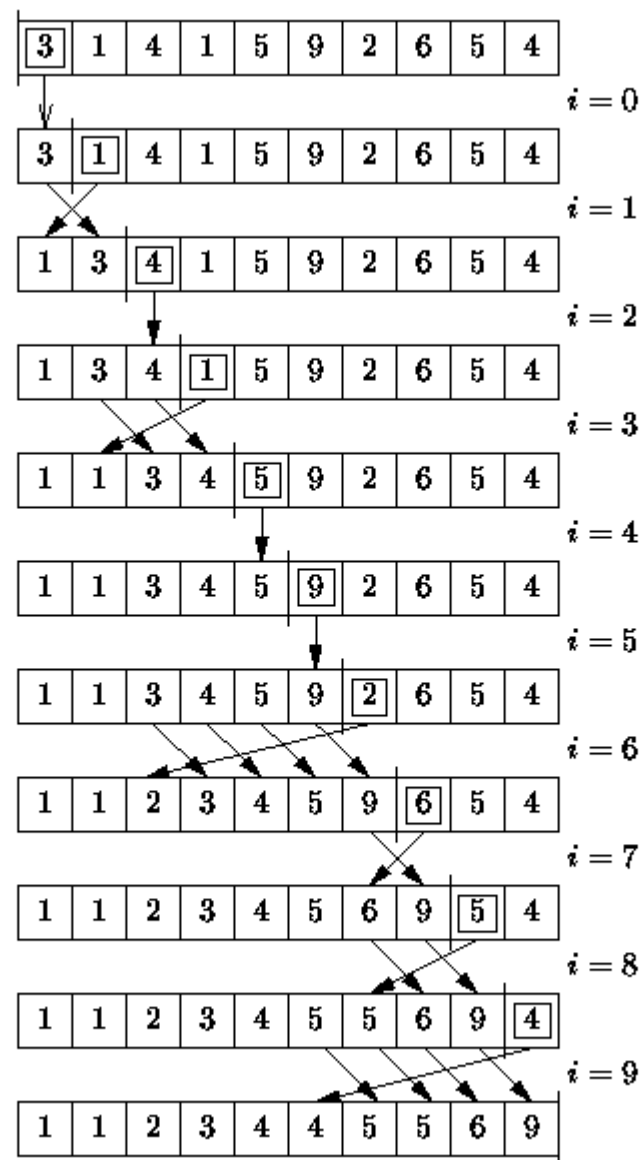
As shown in Figure , the first step ($i=0$) is trivial--inserting an element into the empty list involves no work. Altogether, $n-1$ non-trivial insertions are required to sort a list of n elements.

Best, worst, and average cases

The best case input is an array that is already sorted. In this case insertion sort has a linear running time (i.e., $\Theta(n)$). During each iteration, the first remaining element of the input is only compared with the right-most element of the sorted subsection of the array.

The worst case input is an array sorted in reverse order. In this case every iteration of the inner loop will scan and shift the entire sorted subsection of the array before inserting the next element. For this case insertion sort has a quadratic running time (i.e., $O(n^2)$).

The average case is also quadratic, which makes insertion sort impractical for sorting large arrays. However, insertion sort is one of the fastest algorithms for sorting very small arrays, even faster than quicksort; indeed, good quicksort implementations use insertion sort for arrays smaller than a certain threshold, also when arising as subproblems; the exact threshold must be determined experimentally and depends on the machine, but is commonly around ten.

**Figure:** Insertion sorting.**//Implementation of insertion sort**

```

template<class T>
void Sorting<T>::insort()
{
    int k,j;
    for(int i=1;i<n;i++)
    {
        k=a[i];j=i-1;
        while(a[j]>k&& j>=0)
        {

```

```

        a[j+1]=a[j];
        j=j-1;
    }
    a[j+1]=k;    }    }

```

Bubble sort

Bubble sort, also known as **sinking sort**, is a simple sorting algorithm that works by repeatedly stepping through the list to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order. The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted. The algorithm gets its name from the way smaller elements "bubble" to the top of the list. Because it only uses comparisons to operate on elements, it is a comparison sort. Although the algorithm is simple, it is not efficient for sorting large lists; other algorithms are better.

Step-by-step example

Let us take the array of numbers "5 1 4 2 8", and sort the array from lowest number to greatest number using bubble sort algorithm. In each step, elements written in **bold** are being compared. Three passes will be required.

First Pass:

(**5** 1 4 2 8) → (**1** 5 4 2 8), Here, algorithm compares the first two elements, and swaps them.
 (1 **5** 4 2 8) → (1 **4** 5 2 8), Swap since 5 > 4
 (1 4 **5** 2 8) → (1 4 **2** 5 8), Swap since 5 > 2
 (1 4 2 **5** 8) → (1 4 2 **5** 8), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

Second Pass:

(1 4 2 5 8) → (1 4 2 5 8)
 (1 4 **2** 5 8) → (1 **2** 4 5 8), Swap since 4 > 2
 (1 2 **4** 5 8) → (1 2 **4** 5 8)
 (1 2 4 **5** 8) → (1 2 4 **5** 8)

Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one **whole** pass without **any** swap to know it is sorted.

Third Pass:

(1 2 4 5 8) → (1 2 4 5 8)
 (1 2 4 5 8) → (1 2 4 5 8)
 (1 2 **4** 5 8) → (1 2 **4** 5 8)
 (1 2 4 **5** 8) → (1 2 4 **5** 8)

//implementation of bubble sort

```

template<class T>
void Sorting<T>::Bubblesort()
{
    for(int i=0;i<n-1;i++)
    {
        for(int j=0;j<n-i-1;j++)
        {
            if(a[j]>a[j+1])
            {

```

```

        T temp=a[j];
        a[j]=a[j+1];
        a[j+1]=temp;
    }

}
} }

```

Selection sort

The idea of selection sort is rather simple: we repeatedly find the next largest (or smallest) element in the array and move it to its final position in the sorted array. Assume that we wish to sort the array in increasing order, i.e. the smallest element at the beginning of the array and the largest element at the end. We begin by selecting the largest element and moving it to the highest index position. We can do this by swapping the element at the highest index and the largest element. We then reduce the *effective size* of the array by one element and repeat the process on the smaller (sub)array. The process stops when the effective size of the array becomes 1 (an array of 1 element is already sorted).

For example, consider the following array, shown with array elements in sequence separated by commas:

63, 75, 90, 12, 27

The leftmost element is at index zero, and the rightmost element is at the highest array index, in our case, 4 (the effective size of our array is 5). The largest element in this effective array (index 0-4) is at index 2. We have shown the largest element and the one at the highest index in **bold**. We then swap the element at index 2 with that at index 4. The result is:

63, 75, 27, 12, 90

We reduce the effective size of the array to 4, making the highest index in the effective array now 3. The largest element in this effective array (index 0-3) is at index 1, so we swap elements at index 1 and 3 (in **bold**):

63, 12, 27, 75, 90

The next two steps give us:

27, 12, 63, 75, 90
12, 27, 63, 75, 90

The last effective array has only one element and needs no sorting. The entire array is now sorted.

//implementation of selection sort

```
template<class T>
void Sorting<T>::selectionsort()
{
    for(int i=n-1;i>0;i--)
    {
        int pos=largest(i+1);
        int temp=a[pos];
        a[pos]=a[i];
        a[i]=temp;
    }
}
```

```
template<class T>
int Sorting<T>::largest(int size)
{
    int max=0;
    for(int i=1;i<size;i++)
    {
        if(a[i]>a[max])
            max=i;
    }
    return max;
}
```

Quick Sort

Bold number is the pivot element.

Before sorting:

30 36 45 66 14 12 1 17

lb=0, ub=7,pivot=

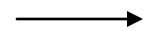
First pass

30 36 45 66 14 12 1 17



Find smallest element exists in the right side of the array when compare with pivot. And place it in the left side. 17 is the smallest element which is under lined.

30 36 45 66 14 12 1 17



Find largest element exists in the left side of the array when compare with pivot. And place it in the right side. 36 is the largest element which is under lined.

Now

After pass

17 **30** 45 66 14 12 1 36

Similarly the remaining pass as shown below

After pass 2nd

17 1 **30** 66 14 12 45 36

After pass 3rd

17 1 12 **30** 14 66 45 36

After pass 4th

17 1 12 14 **30** 66 45 36

Now the pivot element 30 place in the exact location. Now apply quick sort to left array of the pivot & right array of the pivot

14 1 12 17 **30** 66 45 36

After pass 5th

14 1 12 17 30 66 45 36

After pass 6th

12 1 **14** 17 30 66 45 36

After pass 7th

1 12 14 17 30 **66** 45 36

After pass 8th

1 12 14 17 30 36 45 **66**

After pass 9th

1 12 14 17 30 **36** 45 66

After sorting:

1 12 14 17 30 36 45 66

//Implementaion of quick sort

```
template<class T>
void Qsort<T>::qsort(int left,int right)
{
    T pivot=a[left];
    int l=left,r=right;
    while(l<r)
    {
        while(a[r]>=pivot&& l<r)
            r--;
        if(l!=r)
            a[l]=a[r];
        while(a[l]<=pivot&& l<r)
            l++;
        if(l!=r)
            a[r]=a[l];
        a[l]=pivot;
    }
    pivot=l;
    if(left<l)
        qsort(left,l-1);
    if(right>l)
        qsort(l+1,right);
}
```

Merge Sort

Merge sort is based on the **divide-and-conquer** paradigm. Its worst-case running time has a lower order of growth than insertion sort. Since we are dealing with subproblems, we state each subproblem as sorting a subarray $A[p \dots r]$. Initially, $p = 1$ and $r = n$, but these values change as we recurse through subproblems.

To sort $A[p \dots r]$:

1. Divide Step

If a given array A has zero or one element, simply return; it is already sorted. Otherwise, split $A[p \dots r]$ into two subarrays $A[p \dots q]$ and $A[q + 1 \dots r]$, each containing about half of the elements of $A[p \dots r]$. That is, q is the halfway point of $A[p \dots r]$.

2. Conquer Step

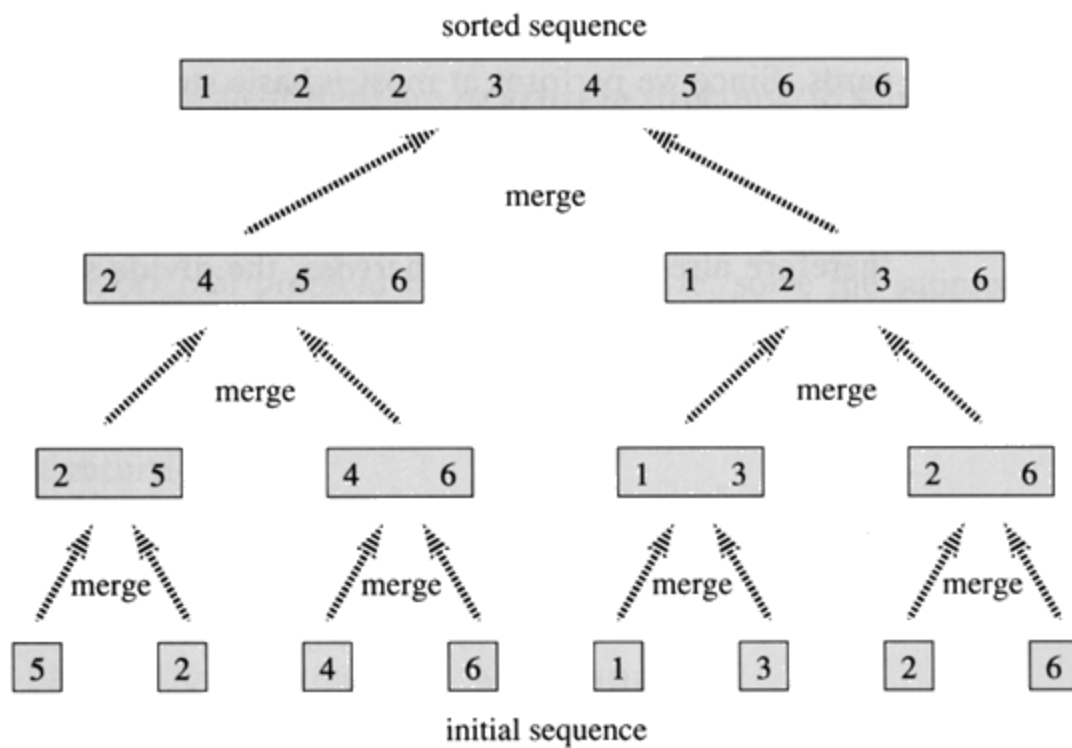
Conquer by recursively sorting the two subarrays $A[p \dots q]$ and $A[q + 1 \dots r]$.

3. Combine Step

Combine the elements back in $A[p \dots r]$ by merging the two sorted subarrays $A[p \dots q]$ and $A[q + 1 \dots r]$ into a sorted sequence. To accomplish this step, we will define a procedure $\text{MERGE}(A, p, q, r)$.

Note that the recursion bottoms out when the subarray has just one element, so that it is trivially sorted.

Bottom-up view of the above procedure for $n = 8$.



//Implementations Recursive & iterative merge sort

```

template<class T>

void MergeSort<T>::msort(int left,int right)
{
    if(left<right)
    {
        int m=(left+right)/2;
        msort(left,m);
        msort(m+1,right);
        recmerge(left,m,right);
    }
}

template<class T>
void MergeSort<T>::recmerge(int left,int mid,int right)
{
    T *b=new T[right];
    int i=left,j=mid+1,k=left;
    while(i<=mid&& j<=right)
    {
        if(a[i]<=a[j])
            b[k++]=a[i++];
        else
            b[k++]=a[j++];
    }
    while(i<=mid)
        b[k++]=a[i++];
    while(j<=right)
        b[k++]=a[j++];
    for(int q=left;q<=right;q++)
        a[q]=b[q];
}

template<class T>
void MergeSort<T>::nonrecmergesort()
{
    T *b=new T[n];
    int len=1;
    while(len<=n-1)
    {
        mpass(len,b);
        len=2*len;
    }
}

```

```

        for(int i=0;i<=n-1;i++)
            a[i]=b[i];
    }
}
template<class T>
void MergeSort<T>::mpass(int len,T *b)
{
    int i=0;
    while(i<=n-(2*len))
    {
        nonrecmerge(i,i+len-1,i+2*len-1,b);
        i=i+2*len;
    }
    if(i+len<n)
        nonrecmerge(i,i+len-1,n-1,b);
    else
        for(int j=i;j<=n-1;j++)
            b[j]=a[j];
}
template<class T>
void MergeSort<T>::nonrecmerge(int left,int mid,int right,T *b)
{
    int i=left,j=mid+1,k=left;
    while(i<=mid&& j<=right)
    {
        if(a[i]<=a[j])
            b[k++]=a[i++];
        else
            b[k++]=a[j++];
    }
    while(i<=mid)
        b[k++]=a[i++];
    while(j<=right)
        b[k++]=a[j++];
}

```

Shell Sort

Shell sort is also called as diminishing sort because using a s k value the insertion sort is applied on the elements. K is the diminishing variable

I	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
N	16	4	3	13	5	6	8	9	10	11	12	17	15	18	19	7	1	2	14	20

I-Index**N-Numbers that are to be sorted**

We can divide this into three smaller slices by placing every 7th element as a part of the array

ie k=7

Subarray 1

16	4	3	13	5	6	8	0	1	2	3	4	5	6	Subarray 2
9	10	11	12	17	15	18	7	8	9	10	11	12	13	
19	7	1	2	14	20		14	15	16	17	18	19		

Subarray 3

Once we have performed this slicing we can sort each slice:

First we sort the first column (16 9 and 19) to give 9 16 19

Next the second column (4 10 and 7) to give 4 7 10

Column three (3 11 and 1) to give 1 3 11

Column four (13 12 and 2) to give 2 12 13

Column five (5 17 and 14) to give 5 14 17

Column six (6 15 and 20) to give 6 15 20

and column seven (8 and 18) to give 8 18

Reassembling the array we now have:

9 4 1 2 5 6 8 16 7 3 12 14 15 18 19 10 11 13 17 20

Notice that elements such as 1 have moved a large distance (from position 17 to position 3).

Now reduce k=3 then apply insertion sort.

Then k=1 then apply insertion sort.

```
template<class T>
void Shellsort<T>::ShellSort()
{
    int k=n,p=0;
    while(k>1)
    {
        k=k/3+1;
```

```

        ShellPass(p,k);
    }
}
template<class T>
void Shellsort<T>:: ShellPass(int index,int k)
{
    for(int i=index+k;i<=n-1;i++)
    {
        InsSort(i,a[i],k);
        cout<<"\n After\n";
        display();
    }
}

```

Heap Sort

Given an array of 6 elements: 15, 19, 10, 7, 17, 16, sort it in ascending order using heap sort.

Steps:

1. Consider the values of the elements as priorities and build the heap tree.
 2. Start deleteMin operations, storing each deleted element at the end of the heap array.
- After performing step 2, the order of the elements will be opposite to the order in the heap tree. Hence, if we want the elements to be sorted in ascending order, we need to build the heap tree in descending order - the greatest element will have the highest priority.

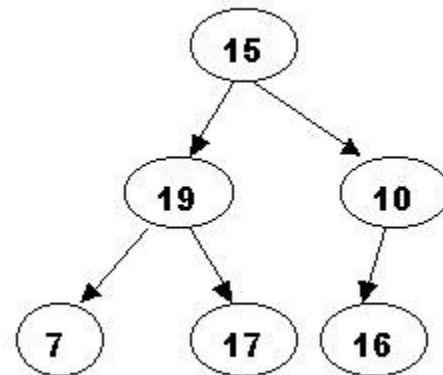
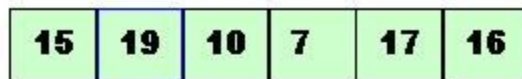
Note that we use only one array , treating its parts differently:

- a. when building the heap tree, part of the array will be considered as the heap, and the rest part - the original array.
 - b. when sorting, part of the array will be the heap, and the rest part - the sorted array.
- This will be indicated by colors: white for the original array, blue for the heap and red for the sorted array

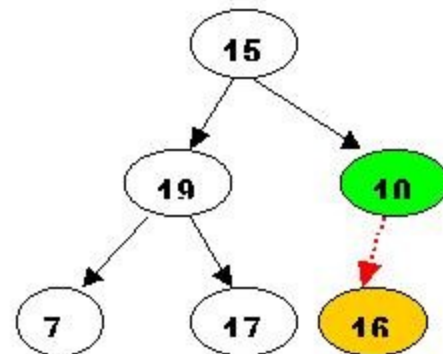
Here is the array: 15, 19, 10, 7, 17, 6

A. Building the heap tree

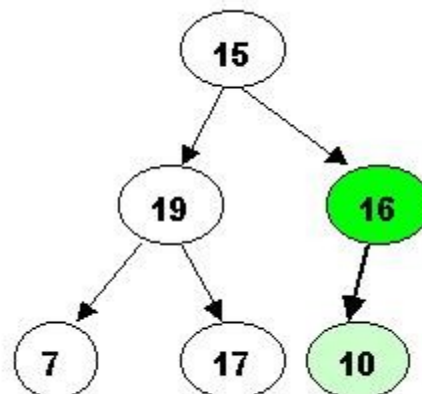
The array represented as a tree, complete but not ordered:



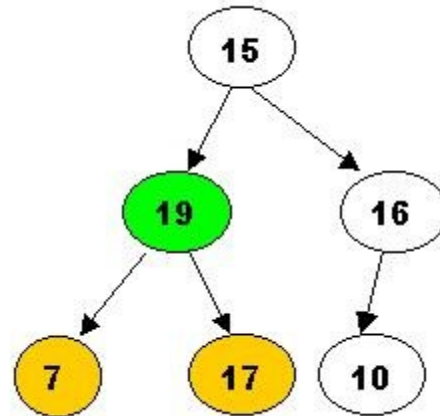
Start with the rightmost node at height 1 - the node at position 3 = Size/2.
It has one greater child and has to be percolated down:



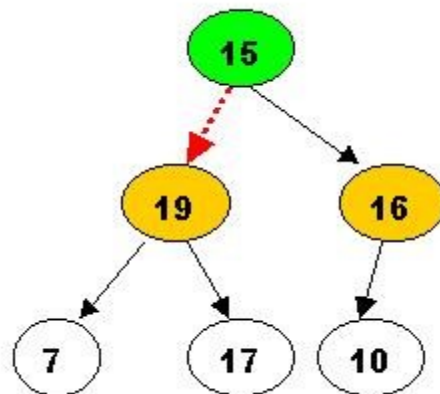
After processing array[3] the situation is:



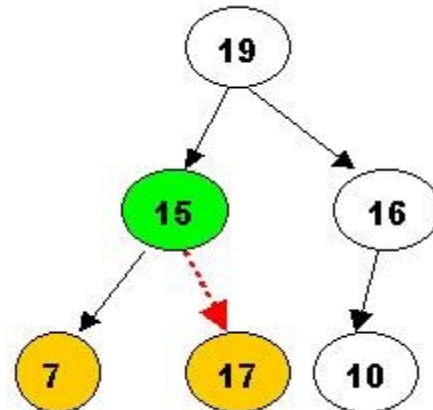
Next comes array[2]. Its children are smaller, so no percolation is needed.



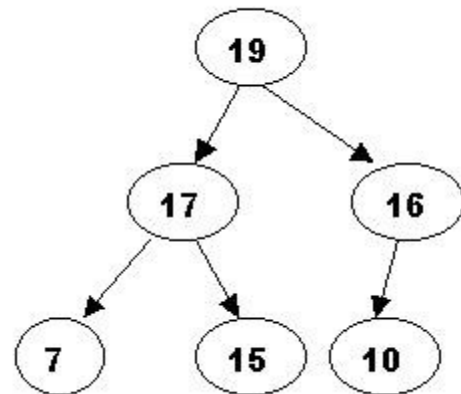
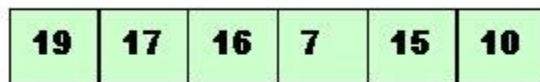
The last node to be processed is array[1]. Its left child is the greater of the children. The item at array[1] has to be percolated down to the left, swapped with array[2].



As a result the situation is:



The children of array[2] are greater, and item 15 has to be moved down further, swapped with array[5].



Now the tree is ordered, and the binary heap is built.

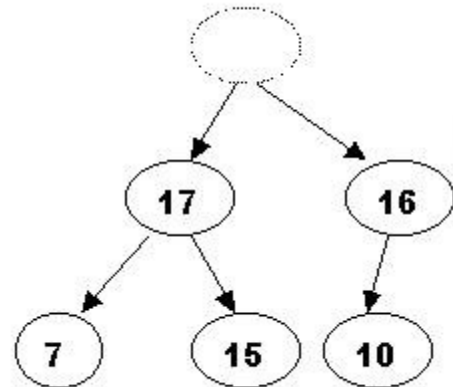
B. Sorting - performing deleteMax operations:

1. Delete the top element 19.

1.1. Store 19 in a temporary place. A hole is created at the top



19



1.2. Swap 19 with the last element of the heap.

As 10 will be adjusted in the heap, its cell will no longer be a part of the heap. Instead it becomes a cell from the sorted array

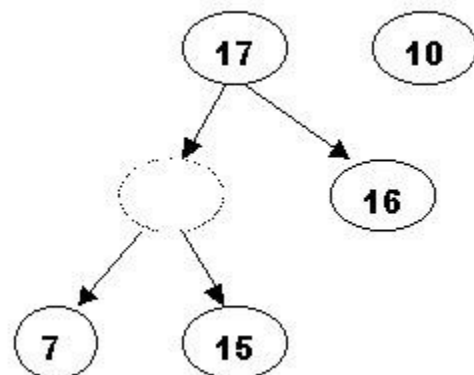


10

1.3. Percolate down the hole



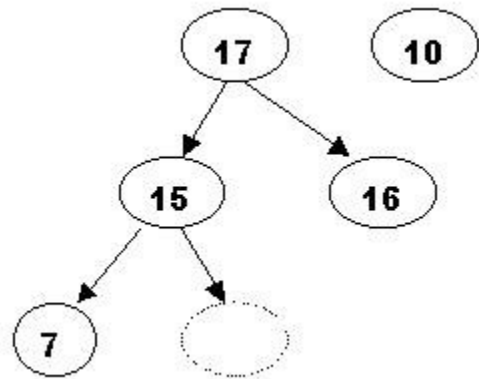
10



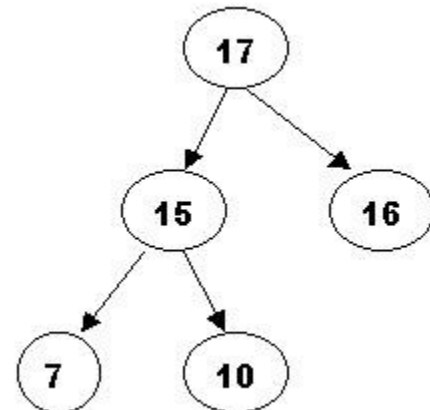
1.4. Percolate once more (10 is less than 15, so it cannot be inserted in the previous hole)



10

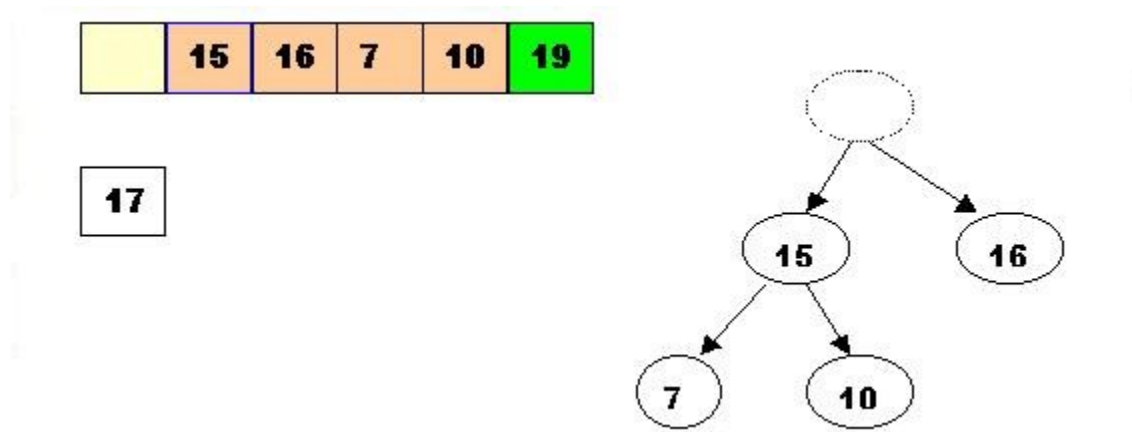


Now 10 can be inserted in the hole



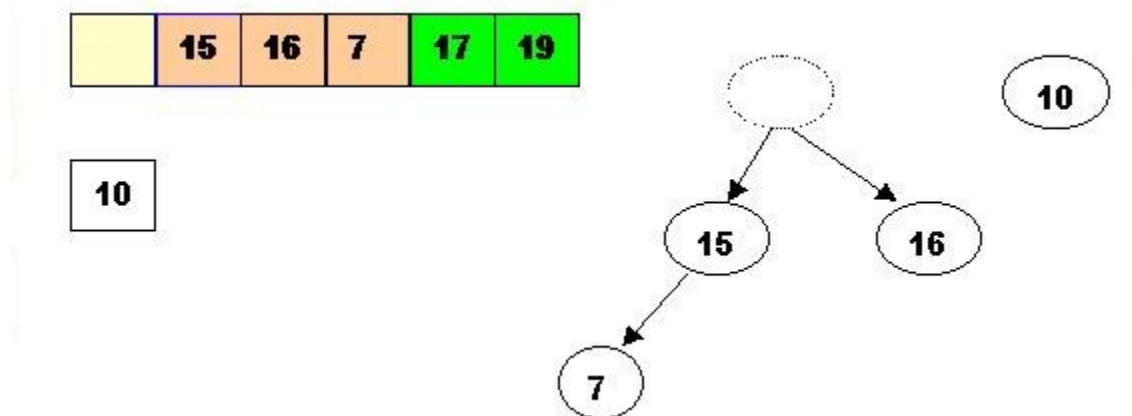
2. DeleteMax the top element 17

2.1. Store 17 in a temporary place. A hole is created at the top

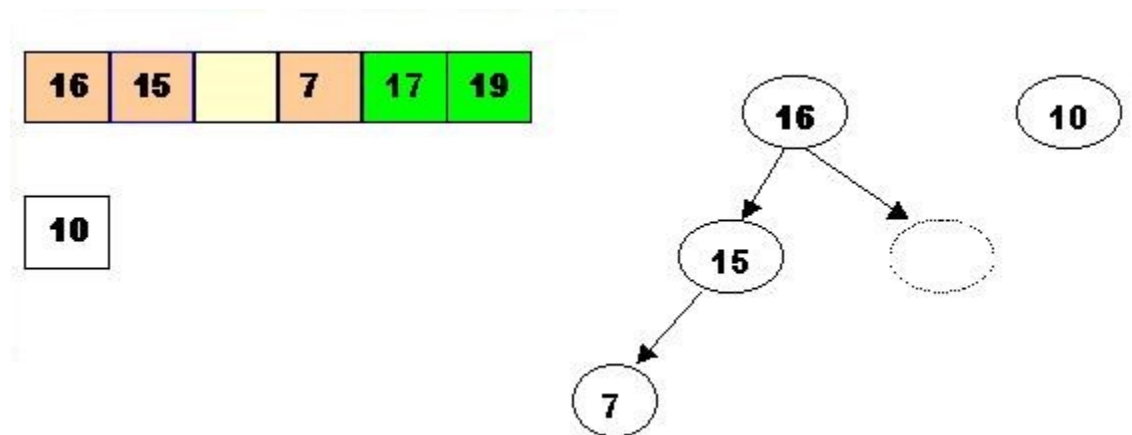


2.2. Swap 17 with the last element of the heap.

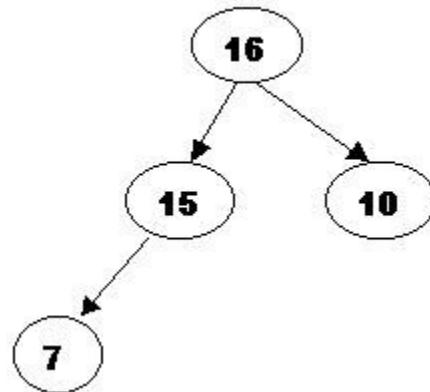
As 10 will be adjusted in the heap, its cell will no longer be a part of the heap.
Instead it becomes a cell from the sorted array



2.3. The element 10 is less than the children of the hole, and we percolate the hole down:

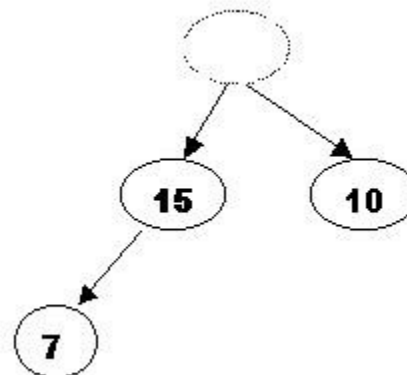


2.4. Insert 10 in the hole



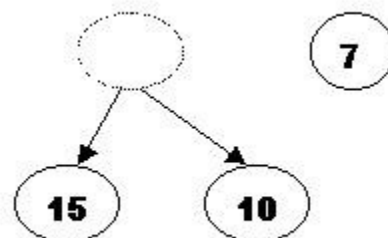
3. DeleteMax 16

3.1. Store 16 in a temporary place. A hole is created at the top

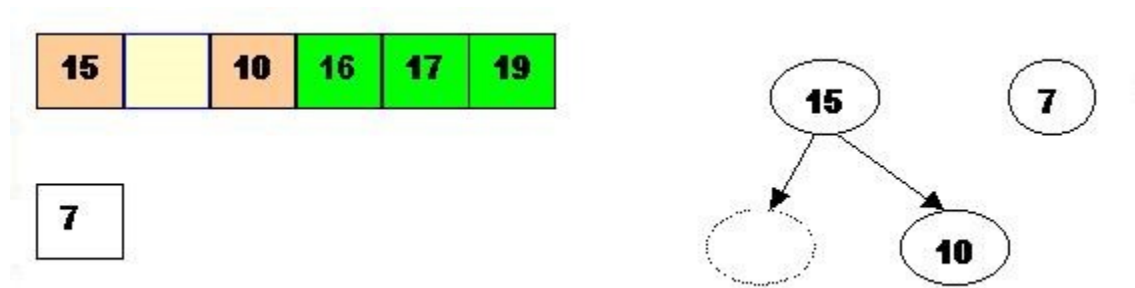


3.2. Swap 16 with the last element of the heap.

As 7 will be adjusted in the heap, its cell will no longer be a part of the heap. Instead it becomes a cell from the sorted array



3.3. Percolate the hole down (7 cannot be inserted there - it is less than the children of the hole)



3.4. Insert 7 in the hole



4. DeleteMax the top element 15

4.1. Store 15 in a temporary location. A hole is created.

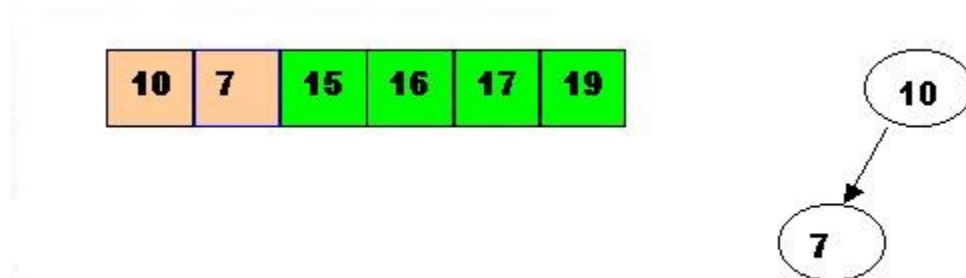


4.2. Swap 15 with the last element of the heap.

As 10 will be adjusted in the heap, its cell will no longer be a part of the heap.
Instead it becomes a position from the sorted array



4.3. Store 10 in the hole (10 is greater than the children of the hole)



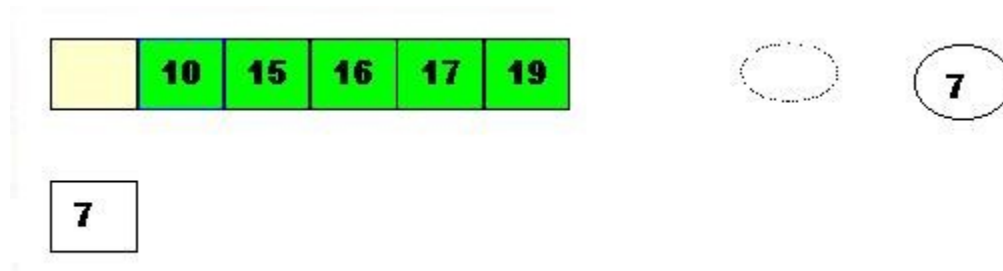
5. DeleteMax the top element 10.

5.1. Remove 10 from the heap and store it into a temporary location.



5.2. Swap 10 with the last element of the heap.

As 7 will be adjusted in the heap, its cell will no longer be a part of the heap. Instead it becomes a cell from the sorted array



5.3. Store 7 in the hole (as the only remaining element in the heap)



7 is the last element from the heap, so now the array is sorted



```
template<class T>
void Hsort<T>::heapsort()
{
    // constructing a heap tree
    for(int i=maxsize/2-1;i>=0;i--)
        adjustH(i,maxsize);
    for(int i=maxsize-1;i>=1;i--)
    {
        T temp=heap[0];
        heap[0]=heap[i];
        heap[i]=temp;
        adjustH(0,i);
    }
}

template<class T>
void Hsort<T>::adjustH(int i,int size)
{
    int ci;
    ci=2*i+1; //lchild of i
    while(ci<=size-1)
    {
        if(ci<size-1 && heap[ci]<heap[ci+1])//rchild exist and it larger than lchild
            ci++;
    }
}
```



```
    if(heap[i]<heap[ci])
    {    T temp=heap[i];
        heap[i]=heap[ci];
        heap[ci]=temp;
    }
    i=ci;        ci=2*i+1;    }
}
```

Comparison of all Sorting techniques

n- number of elements, **k**- the size of each key, **d**- the digit size used by the implementation

Name of the Sorting Technique	Best Case	Average Case	Worst Case
Bubble Sort	n	n^2	n^2
Insertion Sort	n	n^2	n^2
Selection Sort	n^2	n^2	n^2
Quick Sort	nlogn	nlogn	n^2
Merge Sort	nlogn	nlogn	nlogn
Heap Sort	nlogn	nlogn	nlogn
Shell Sort	n	$n^{3/2}$	$n(\log n)^2$
Radix Sort	-	$n*(k/d)$	$n*(k/d)$

```
// list sort
template<class T>
void Sorting<T>::ListinsertSort()
{
    int * link = new int[n];
    int head =0;
    int next,cur,i;
    link[0] = -1;
    for (i = 1 ; i < n; i++)
    {
        if (a[head] > a[i])
        {
            link[i] = head;
            head = i;
        }
    }
}
```

```

    }
    else
    {
        for (cur = head ; cur != -1 && a[cur] <= a[i]; cur = link[cur])
            next = cur;
        link[next] = i;
        link[i] = cur;
    }
    cout<<"\n link values\n";
    for(int j=0;j<n;j++)
    cout<<" "<<link[j];
}
cout<<"\n*****\n";
cur = head;
for( i = 0 ; i < n-1;i++)
{
    while(cur < i)
        cur = link [cur];
    next = link[cur];
    if (cur != i)
    {
        T swap = a[i]; a[i] = a[cur]; a[cur] = swap;
        link[cur] = link[i];
        link[i] = cur;
    }
    cur = next;
    cout<<"\n a elements are\n";
    Display();
    cout<<"\nlink values\n";
    for(int j=0;j<n;j++)
        cout<<" "<<link[j];
}
delete[] link;
}

```

//Implementation of binary tree using linked list

```

#include<iostream>
#include"LStack.h"
using namespace std;
template<class T>
class BT;
template<class T>
class BTreeNode
{
    friend class BT<T>;
    T ele;
    BTreeNode<T> *lc,*rc;
public:
    BTreeNode(T e=0,BTreeNode<T> *l=0,BTreeNode<T> *r=0)
    {
        ele=e;
        lc=l;rc=r;
    }
};
template<class T>
class BT
{
    BTreeNode<T> *root;
public:
    BT() {root=0;}
    void makeTree(T e,BT<T> &a,BT<T> &b)
    {
        root=new BTreeNode<T>(e,a.root,b.root);
        a.root=b.root=0;
    }
    void Inorder()
    {
        Inorder(root);
        cout<<"\n Height of the tree is :"<<Height(root);
    }
}

```

```

void Inorder(BTNode<T> *r)
{
    if(r)
    {
        Inorder(r->lc);
        cout<<r->ele<<"\t";
        Inorder(r->rc);
    }
}

void Preorder(BTNode<T> *r)
{
    if(r)
    {
        cout<<r->ele<<"\t";
        Preorder(r->lc);
        Preorder(r->rc);
    }
}

void Postorder(BTNode<T> *r)
{
    if(r)
    {
        Postorder(r->lc);
        Postorder(r->rc);
        cout<<r->ele<<"\t";
    }
}

int Height(BTNode<T> *r)
{
    if(!r)
        return 0;
    int hl=Height(r->lc);
    int hr=Height(r->rc);
    if(hl>hr)
        return ++hl;
    else
        return ++hr;
}

//non-recursive or iterative inorder traversals implementation using Lstack
void nonrecinorder()
{
    BTNode<T> *currentNode=root;
    LStack<BTNode<T>*> s;
    while(1)

```

```

    {
        while(currentNode)
        {
            s.Push(currentNode);
            currentNode=currentNode->lc;
        }
        if(s.IsEmpty())
            return ;
        currentNode=s.Pop();
        cout<<currentNode->ele<<"\t";
        currentNode=currentNode->rc;
    }
}

```

//non-recursive inorder using iterator inner class in BT class without using stack

```

void nonrecinorderiterator()
{
    InorderIterator obj(root);
    while(1)
    {
        T temp=obj.Next();
        if(temp==0)
            return;
        cout<<temp<<"\t";
    }
}

class InorderIterator
{
    LStack<BTNode<T>*> s;
    BTNode<T> *currentNode;
public:
    InorderIterator(BTNode<T>* cn){ currentNode=cn;}
    T Next()
    {
        while(currentNode)
        {
            s.Push(currentNode);
            currentNode=currentNode->lc;
        }
        if(s.IsEmpty())
            return 0;
    }
}

```

```

        currentNode=s.Pop();
        T temp=currentNode->ele;
        currentNode=currentNode->rc;
        return temp;
    }
};

int main()
{
    BT<char> X,Y,Z,a,b,c;
    X.makeTree('G',a,a);
    Y.makeTree('H',b,b);
    Z.makeTree('E',X,Y);
    c.makeTree('D',a,a);
    X.makeTree('B',c,Z);
    Y.makeTree('F',a,a);
    Z.makeTree('C',a,Y);
    Y.makeTree('A',X,Z);

    cout<<"\n inorder traversal using recursive function\n";
    Y.Inorder();

    cout<<"\n inorder traversal using non recursive function\n";
    Y.nonrecinorder();

    cout<<"\n inorder traversal using non recursive Inorder Iterator nested class of binary tree\n";
    Y.nonrecinorderiterator();
    return 0;
}

```

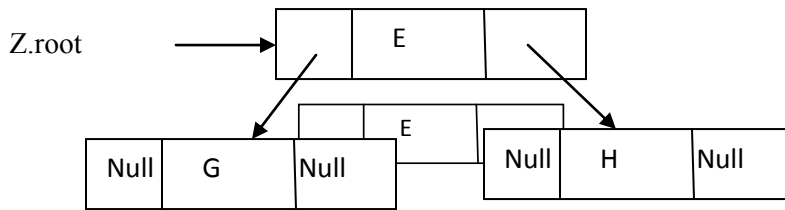
makeTree function will create the tree just as follows

X,Y,Z,a,b,c are objects of BT class which is having root variable. So every BT object will have root & initial value is NULL.

Therefore

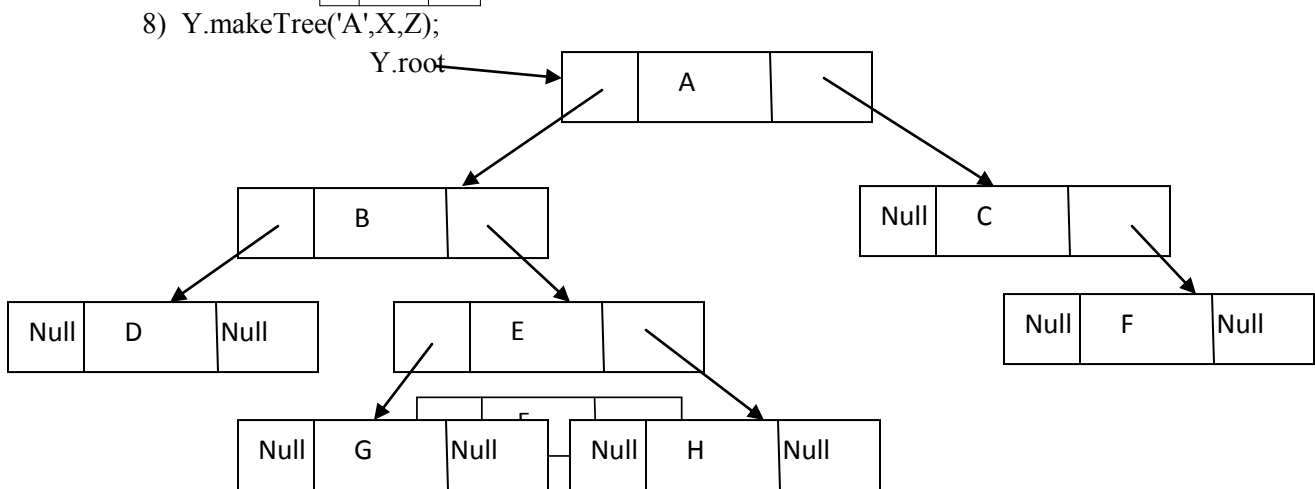
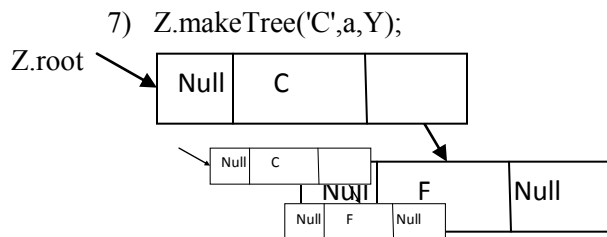
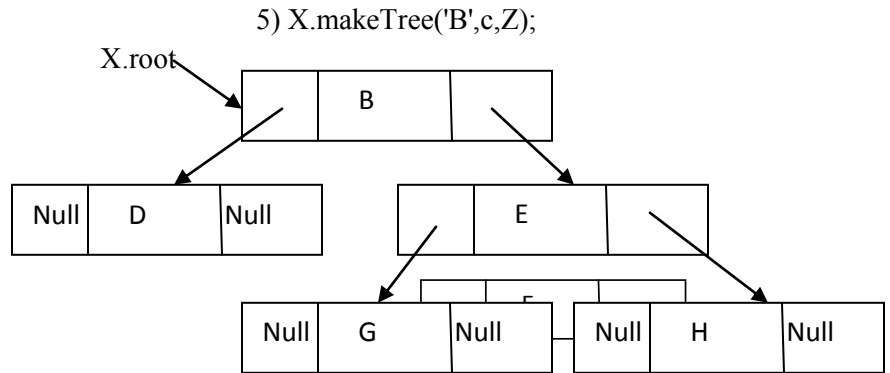
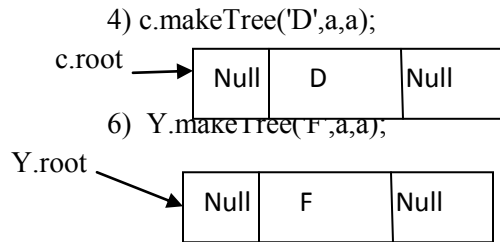


3) Z.makeTree('E',X,Y);



Now X & Y objects of BT class are free We can use those objects for

Creating next node.



Binary Search Tree

Binary Search Tree(BST) is a binary tree which follows the following conditions-

The left subtree of a node contains only nodes with keys less than the node's key

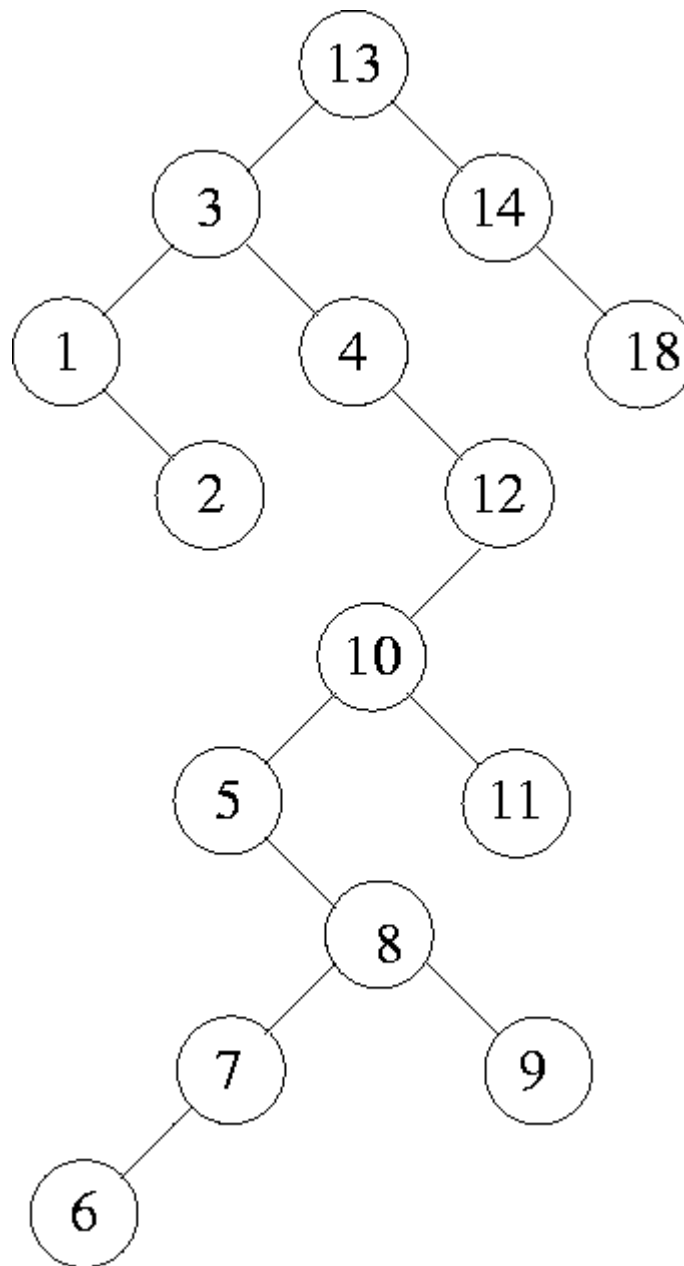
The right subtree of a node contains only nodes with keys greater than the node's key.

Both the left & right subtrees must also be the BST.

Search Operations

We begin by examining the root node. If the tree is null, the value we are searching for does not exist in the tree. Otherwise, if the value equals the root, the search is successful. If the value is less than the root, search the left subtree. Similarly, if it is greater than the root, search the right subtree. This process is repeated until the value is found or the indicated subtree is null. If the searched value is not found before a null subtree is reached, then the item must not be present in the tree.

Insertion Operation

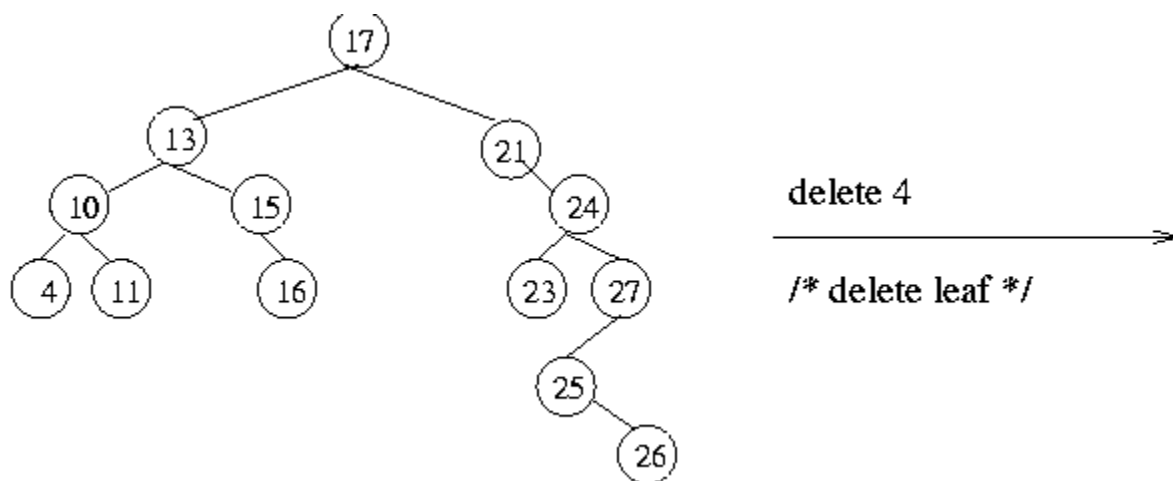


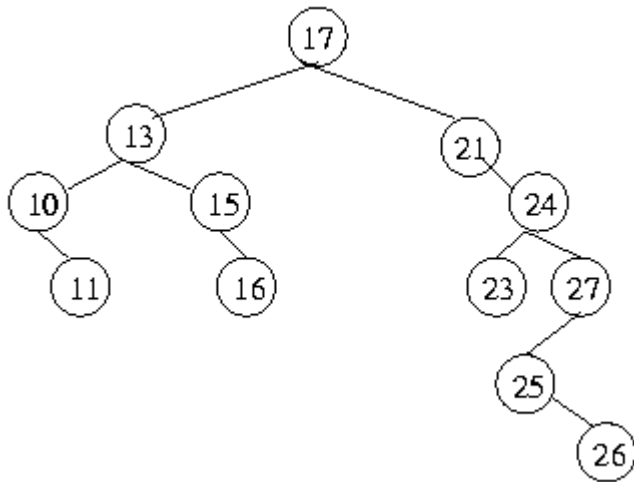
Insertion begins as a search would begin; if the root is not equal to the value, we search the left or right subtrees as before. Eventually, we will reach an external node and add the value as its right or left child, depending on the node's value. In other words, we examine the root and recursively insert the new node to the left subtree if the new value is less than the root, or the right subtree if the new value is greater than or equal to the root.

Deletion in BST

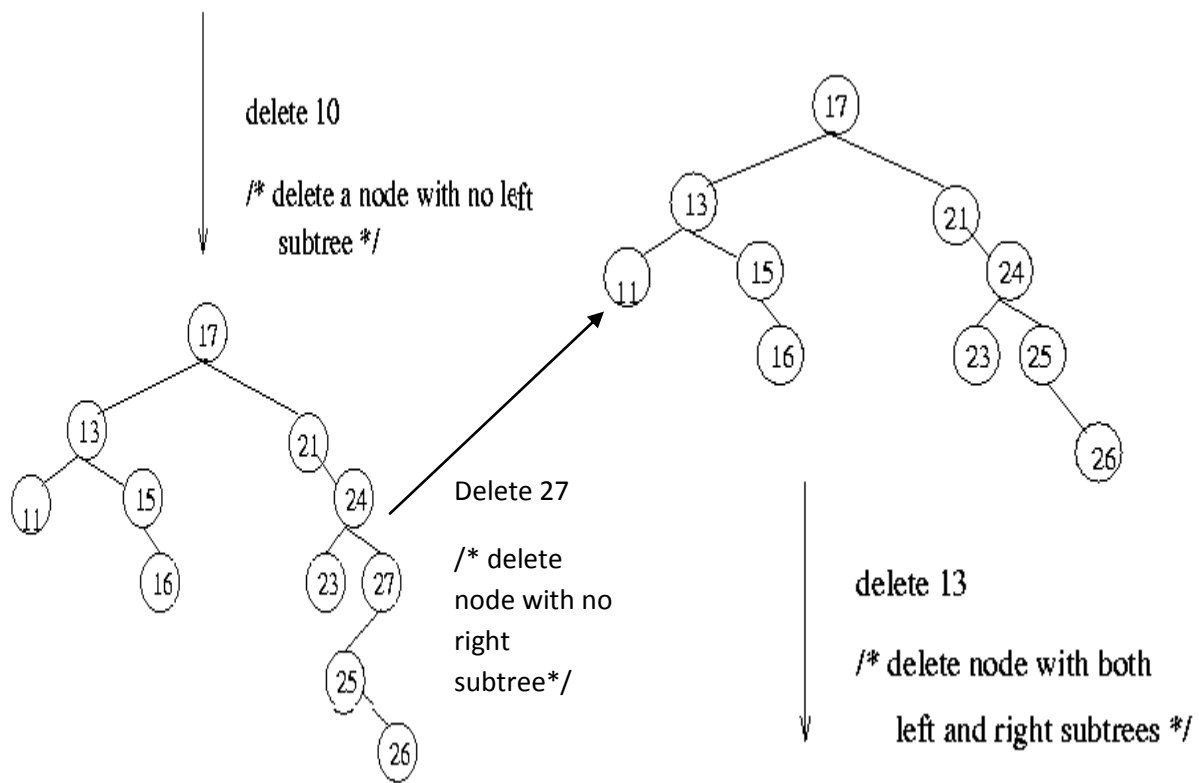
Let x be a value to be deleted from the BST and let X denote the node containing the value x . Deletion of an element in a BST again uses the BST property in a critical way. When we delete the node X containing x , it would create a "void" that should be filled by a suitable existing node of the BST. There are two possible candidate nodes that can fill this void, in a way that the BST property is not violated: (1). Node containing highest valued element among all descendants of left child of X . (2). Node containing the lowest valued element among all the descendants of the right child of X . In case (1), the selected node will necessarily have a null right link which can be conveniently used in patching up the tree. In case (2), the selected node will necessarily have a null left link which can be used in patching up the tree. Figure illustrates several scenarios for deletion in BSTs.

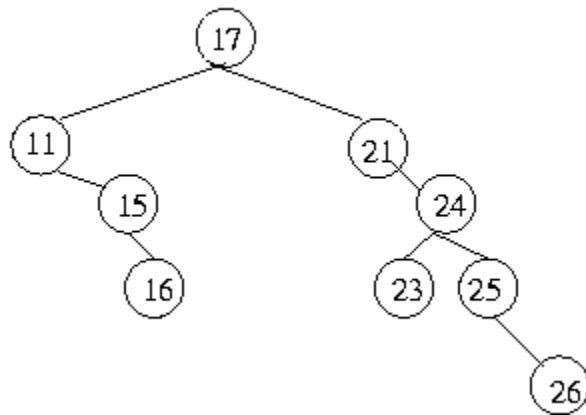
Figure 4.15: Deletion in binary search trees: An example





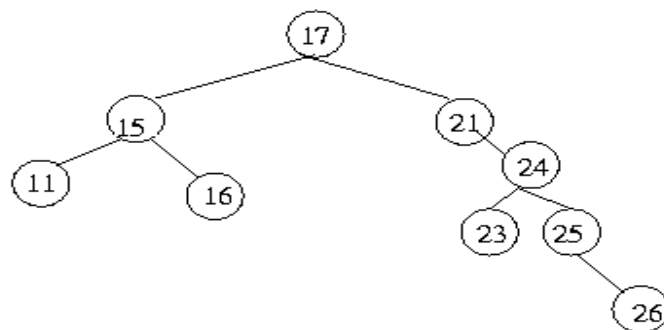
From the above tree deleting node 10 which is not having left child





Method 1.

Find highest valued element
among the descendants of
left child



Method 2

Find lowest valued element
among the descendants of
right child

//Binary Search Tree implementation

```
#include<iostream>
using namespace std;
template<class T>
class BST;
template<class T>
class BSTnode
{
    T data;
    BSTnode<T> *lc,*rc;
    friend class BST<T>;
public:
    BSTnode(T &e,BSTnode<T> *l=0,BSTnode<T> *r=0)
    {
```

```

        data=e; lc=l; rc=r;
    }
};
template<class T>
class BST
{
    BSTnode<T> *root;
public:
    BST(){root=NULL;}
    void Insert(T& ele);
    void Delete(T ele);
    void Search(T ele);
    void Inorder(BSTnode<T> *);
    void Inorder(){Inorder(root);}
};
template<class T>
void BST<T>::Insert(T &ele)
{
    BSTnode<T> *pp,*p=root;
    while(p!=NULL)
    {
        pp=p;
        if(p->data<ele)
            p=p->rc;
        else if(p->data>ele)
            p=p->lc;
        else
        {
            cout<<"duplicate element's not allowed";
            return;
        }
    }
    BSTnode<T> *temp=new BSTnode<T>(ele);
    if(root==NULL)
        root=temp;
    else
    {
        if(pp->data<ele)
            pp->rc=temp;
        else
            pp->lc=temp;
    }
}

```

```

}
template<class T>
void BST<T>::Delete(T ele)
{
    T e;
    BSTnode<T> *p=root, *pp=NULL;
    if(root==NULL)
    {
        cout<<"\n Tree is Empty";
        return;
    }
    while(p!=NULL&& p->data!=ele)
    {
        pp=p;
        if(p->data>ele)
            p=p->lc;
        else
            p=p->rc;
    }
    if(p==NULL)
    {
        cout<<"\n Node does not exist";
        return;
    }
    e=p->data;
    if(p->lc!=NULL&&p->rc!=NULL)
    {
        BSTnode<T> *s,*ps;
        s=p->lc;ps=p;
        while(s->rc!=NULL)
        {
            ps=s;
            s=s->rc;
        }
        p->data=s->data;
        p=s;
        pp=ps;
    }
    BSTnode<T>*c;
    if(p->lc!=NULL)
        c=p->lc;
    else

```

```

        c=p->rc;
    if(p==root)
        root=c;
    else
    {
        if(pp->lc==p)
            pp->lc=c;
        else
            pp->rc=c;
    }
    cout<<e<<" node is deleted\n";
    delete p;
}
template<class T>
void BST<T>::Search(T ele)
{
    BSTnode<T> *ptr=root;
    if(root==NULL)
    {
        cout<<"\n Tree is empty";
        return;
    }
    while(ptr!=NULL)
    {
        if(ptr->data>ele)
            ptr=ptr->lc;
        else if(ptr->data<ele)
            ptr=ptr->rc;
        else
        {
            cout<<"\n Element exist";
            return;
        }
    }
    cout<<"\n Element does not exist";
}
template<class T>
void BST<T>::Inorder(BSTnode<T>* p)
{
    if(p!=NULL)
    {
        Inorder(p->lc);

```

```

        cout<<p->data<<"\t";
        Inorder(p->rc);
    }
}
int menu()
{
    int i;
    cout<<"\n 1.Insert\n 2.Delete\n3.Search\n4.inorder\n5.exit";
    cin>>i;
    return i;
}
int main()
{
    int i,ele;
    BST<int> obj;
    i=menu();
    while(i!=5)
    {
        switch(i)
        {
            case 1: cout<<"\n Enter element:";
                    cin>>ele;
                    obj.Insert(ele);break;
            case 2: cout<<"\n Enter element for delete:";
                    cin>>ele;
                    obj.Delete(ele); break;
            case 3:cout<<"\n Enter element for searching:";
                    cin>>ele;
                    obj.Search(ele);break;
            case 4:obj.Inorder();break;
        }
        i=menu();
    }
    return 0;
}

```

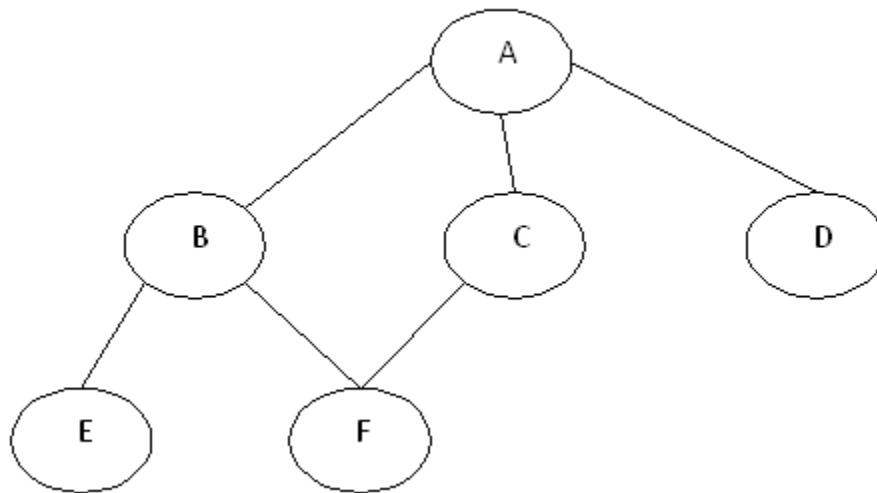

What is a Graph?

Graphs are one of the most interesting data structures in computer science. Graphs and the trees are somewhat similar by their structure. In fact, tree is derived from the graph data structure. However there are two important differences between trees and graphs.

1. Unlike trees, in graphs, a node can have many parents.
2. The link between the nodes may have values or weights.

Graphs are good in modeling real world problems like representing cities which are connected by roads and finding the paths between cities, modeling air traffic controller system, etc. These kinds of problems are hard to represent using simple tree structures.

The following example shows a very simple graph:



In the above graph, A,B,C,D,E,F are called nodes and the connecting lines between these nodes are called edges. The edges can be directed edges which are shown by arrows; they can also be weighted edges in which some numbers are assigned to them. Hence, a graph can be a directed/undirected and weighted/un-weighted graph. In this article, we will discuss undirected and un-weighted graphs.

Edges represent the connection between nodes. There are two ways to represent edges.

Adjacency Matrix

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

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

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

The advantages of representing the edges using adjacency matrix are:

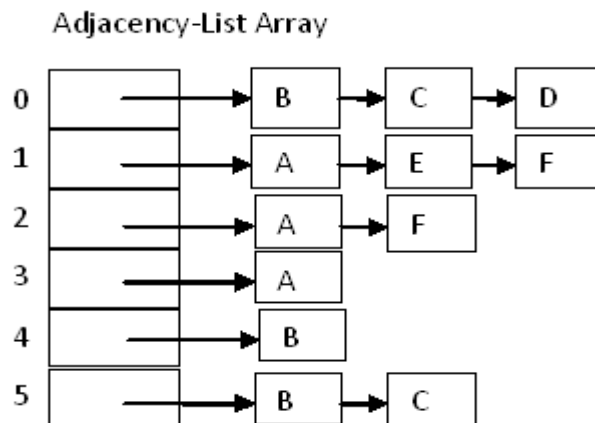
1. Simplicity in implementation as you need a 2-dimensional array
2. Creating edges/removing edges is also easy as you need to update the Booleans

The drawbacks of using the adjacency matrix are:

1. Increased memory as you need to declare $N \times N$ matrix where N is the total number of nodes.
2. Redundancy of information, i.e. to represent an edge between A to B and B to A , it requires to set two Boolean flag in an adjacency matrix.

Adjacency List

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

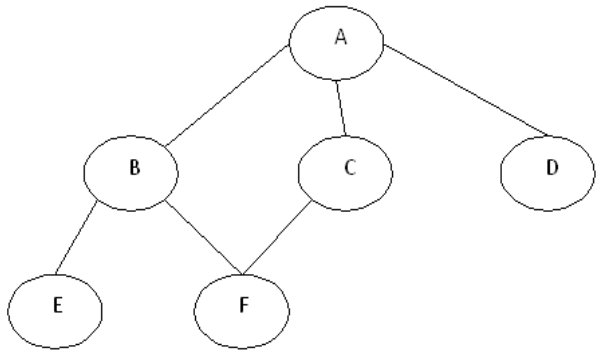


Graph Traversal

The breadth first search (BFS) and the depth first search (DFS) are the two algorithms used for traversing and searching a node in a graph. They can also be used to find out whether a node is reachable from a given node or not.

Depth First Search (DFS)

The aim of DFS algorithm is to traverse the graph in such a way that it tries to go far from the root node. Stack is used in the implementation of the depth first search. Let's see how depth first search works with respect to the following graph:



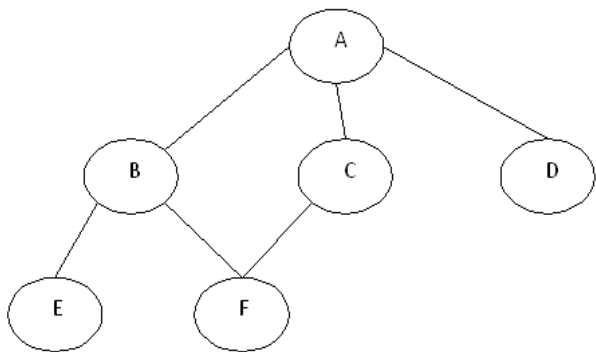
As stated before, in DFS, nodes are visited by going through the depth of the tree from the starting node. If we do the depth first traversal of the above graph and print the visited node, it will be “A B E F C D”. DFS visits the root node and then its children nodes until it reaches the end node, i.e. E and F nodes, then moves up to the parent nodes.

Algorithmic Steps

1. **Step 1:** Push the root node in the Stack.
2. **Step 2:** Loop until stack is empty.
3. **Step 3:** Peek the node of the stack.
4. **Step 4:** If the node has unvisited child nodes, get the unvisited child node, mark it as traversed and push it on stack.
5. **Step 5:** If the node does not have any unvisited child nodes, pop the node from the stack.

Breadth First Search (BFS)

This is a very different approach for traversing the graph nodes. The aim of BFS algorithm is to traverse the graph as close as possible to the root node. Queue is used in the implementation of the breadth first search. Let's see how BFS traversal works with respect to the following graph:



If we do the breadth first traversal of the above graph and print the visited node as the output, it will print the following output. “A B C D E F”. The BFS visits the nodes level by level, so it will start with level 0 which is the root node, and then it moves to the next levels which are B, C and D, then the last levels which are E and F.

Algorithmic Steps

1. **Step 1:** Push the root node in the Queue.
2. **Step 2:** Loop until the queue is empty.
3. **Step 3:** Remove the node from the Queue.
4. **Step 4:** If the removed node has unvisited child nodes, mark them as visited and insert the unvisited children in the queue.

//Graph implementation using linked list

```
#include<iostream>
using namespace std;
template<class T>
class chain;
template<class T>
class graph;
template<class T>
class CNode
{
    T data;
    CNode<T> *next;
public:
    CNode(T &ele=0,CNode<T> *n=0)
    {
        data=ele; next=n;
    }
    friend class chain<T>;
    friend class graph<T>;
};
template<class T>
class chain
{

```

```

CNode<T> *first,*last;
public:
    chain(){first=0;last=0;}
    ~chain()
    {
        for(CNode<T> *p=first;p!=0;p=p->next)
            delete p;
        delete first;
        delete last;
    }
    CNode<T>* begin(){return first;}
    void insertLast(T &ele)
    {
        CNode<T> *temp=new CNode<T>(ele);
        if(last)
        {
            temp->next=last->next;
            last->next=temp;
            last=temp;
        }
        Else
            first=last=temp;
    }
    void display()
    {
        for(CNode<T> *ptr=first;ptr!=0;ptr=ptr->next)
            cout<<ptr->data<<"\t";
    }
};
#include"chain.h"
#include"LQueue.h"
template<class T>
class graph
{
    int n;
    chain<T> *list;
    bool *visit;
public:
    graph(int vetx)
    {
        n=vetx;
        list=new chain<T>[n+1];
    }
};

```

```

    }
    ~graph()
    {
        delete []visit;
        delete []list;
    }
    void create()
    {
        int i,cv,ele;
        for(i=1;i<=n;i++)
        {
            cout<<"\n How many adjacent vertices for "<<i<<" vertex:";
            cin>>cv;
            cout<<"\n Enter "<<cv<<" adjacent vertices of "<<i<<" vertex in
ascending order";
            int j=1;
            while(j<=cv)
            {
                cout<<"\nEnter "<<j<<" vertex:";
                cin>>ele;
                list[i].insertLast(ele);
                j++;
            }
        }
    }
    void display()
    {
        for(int i=1;i<=n;i++)
        {
            cout<<"\n adjacent vertices of "<<i<<" vertex\n";
            list[i].display();
        }
    }
    void bfs()
    {
        CNode<T> *temp;
        LQueue<T> Q;
        int i,j;
        visit=new bool[n+1];
        for(i=1;i<=n;i++)
            visit[i]=false;
        cout<<"\n Enter vertex number to find the reachable vertices:";

```

```

        cin>>i;
        visit[i]=true;
        cout<<i<<"\t";
        Q.Insert(i);
        //cout<<" is insert in q";
        while(!Q.IsEmpty())
        {
            j=Q.Delete();
            /* extract all adjacent vertices of vertex j. if they are not at visited just insert into Queue*/
            for(temp=list[j].begin();temp!=NULL;temp=temp->next)
            {
                if(!visit[temp->data])
                {
                    // cout<<"\n Queue is..\n";
                    cout<<temp->data<<"\t"; visit[temp->data]=true;
                    Q.Insert(temp->data);
                    //Q.Qdisplay();
                }
            }
        }
    }
}

void dfs()
{
    int i;
    visit=new bool[n+1];
    for(i=1;i<=n;i++)
        visit[i]=false;
    cout<<"\n Enter vertex number to find the reachable vertices:";
    cin>>i;
    traverse(i);
}

void traverse(int i)
{
    CNode<T> *temp;
    if(!visit[i])
    {
        cout<<i<<"\t";
        visit[i]=true;
        for(temp=list[i].begin();temp!=NULL;temp=temp->next)
            traverse(temp->data);
    }
}

```



```
};

main()
{
    int n;
    cout<<"\n Enter number of vertex:";
    cin>>n;
    graph<int> obj(n);
    obj.create();
    obj.display();
    cout<<"\n BFS \n";
    obj.bfs();
    cout<<"\n DFS \n";
    obj.dfs();
}
```

AVL Trees

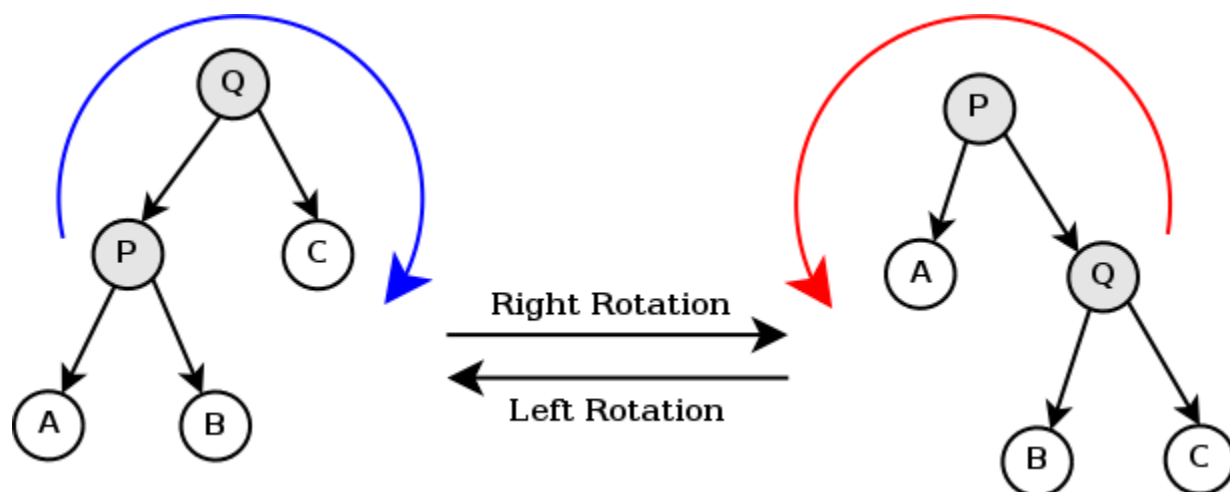
For an AVL tree every node is having balance factor along with left child address, right child address and data.

Balance factor is nothing but the height difference of its left subtree and right subtree ie hl-hr. Balance factor value should not exceed -1,0,1.

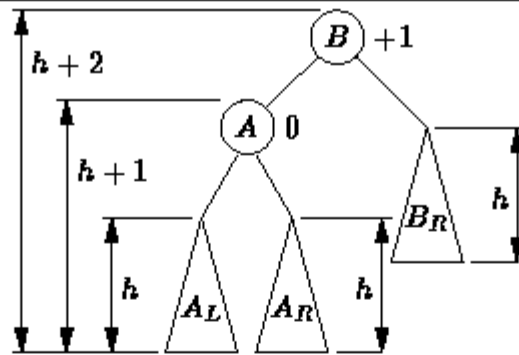
But while performing insert & delete operation on AVL tree the tree will become imbalance by exceed the values to ...-3,-2 or 2,3....

We are having Single rotations & Double rotations to make the tree as balance.

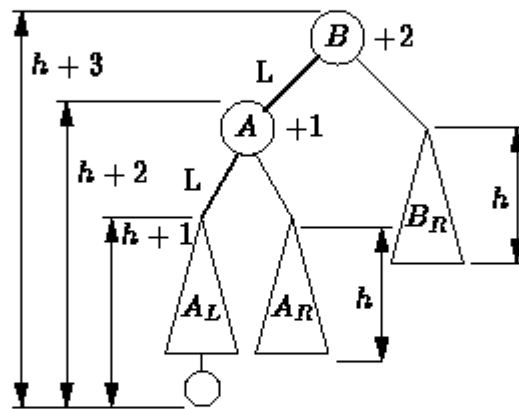
Single Rotations----- LL(left rotation) & RR



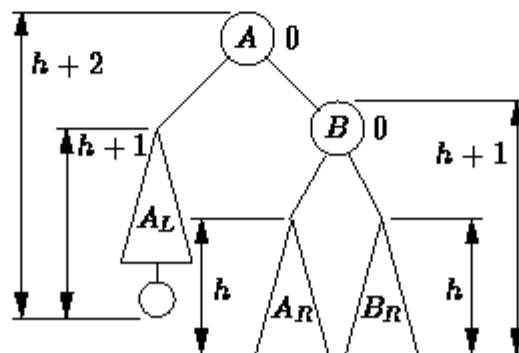
Single Rotation (LL)



(a)

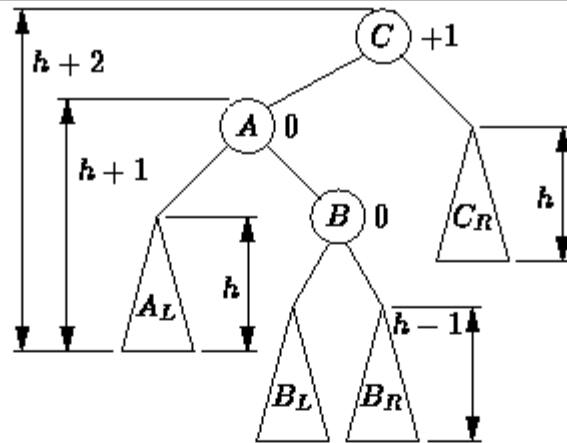


(b)

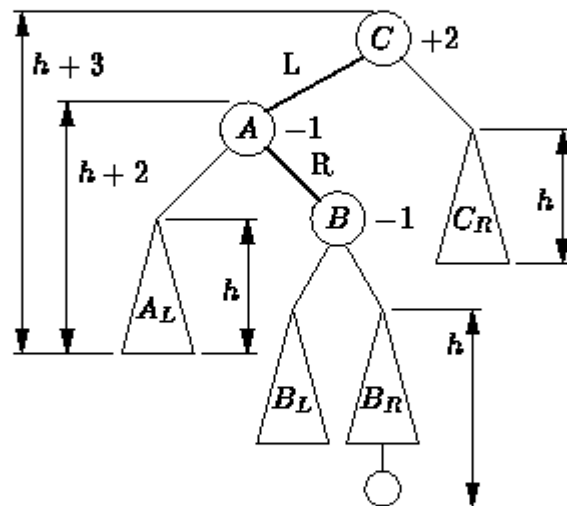


(c)

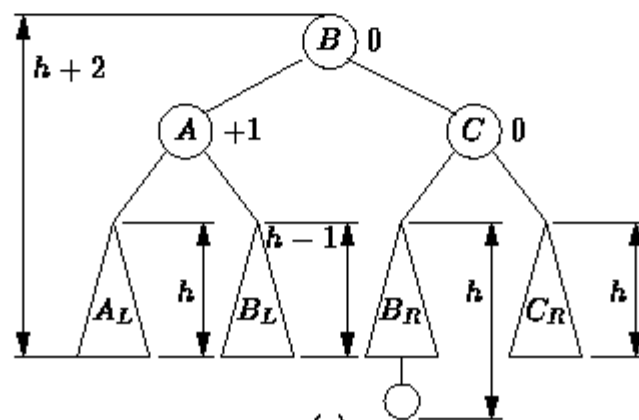
Double Rotation (LR)



(a)



(b)



(c)

//AVL Tree implementation

```

#include <iostream>
#include <stdlib.h>
using namespace std;
class AVLNode
{
    public:
        int data ;
        int balfact ;
        AVLNode *left ;
        AVLNode *right ;
};

class avltree
{
    private :
        AVLNode *root ;
    public :
        avltree() ;
        AVLNode* insert ( int data, int *h ) ;
        static AVLNode* buildtree ( AVLNode *root, int data, int *h ) ;
        void display( AVLNode *root ) ;
        AVLNode* deldata ( AVLNode* root, int data, int *h ) ;
        static AVLNode* del ( AVLNode *node, AVLNode* root, int *h ) ;
        static AVLNode* balright ( AVLNode *root, int *h ) ;
        static AVLNode* balleft ( AVLNode* root, int *h ) ;
        void setroot ( AVLNode *avl ) ;
        ~avltree() ;
        static void deltree ( AVLNode *root ) ;
};

avltree :: avltree()
{
    root = NULL ;
}
AVLNode* avltree :: insert ( int data, int *h )
{
    root = buildtree ( root, data, h ) ;
    return root ;
}
AVLNode* avltree :: buildtree ( AVLNode *root, int data, int *h )
{
    AVLNode *node1, *node2 ;

    if ( root == NULL )
    {
        root = new AVLNode ;
        root -> data = data ;
        root -> left = NULL ;
        root -> right = NULL ;
    }
}

```

```

        root -> balfact = 0 ;
        *h = TRUE ;
        return ( root ) ;
    }
    if ( data < root -> data )
    {
        root -> left = buildtree ( root -> left, data, h ) ;

        // If left subtree is higher
        if ( *h )
        {
            switch ( root -> balfact )
            {
                case 1 :
                    node1 = root -> left ;
                    if ( node1 -> balfact == 1 )
                    {
                        cout << "\nRight rotation." ;
                        root -> left = node1 -> right ;
                        node1 -> right = root ;
                        root -> balfact = 0 ;
                        root = node1 ;
                    }
                    else
                    {
                        cout << "\nDouble rotation, left then right." ;
                        node2 = node1 -> right ;
                        node1 -> right = node2 -> left ;
                        node2 -> left = node1 ;
                        root -> left = node2 -> right ;
                        node2 -> right = root ;
                        if ( node2 -> balfact == 1 )
                            root -> balfact = -1 ;
                        else
                            root -> balfact = 0 ;
                        if ( node2 -> balfact == -1 )
                            node1 -> balfact = 1 ;
                        else
                            node1 -> balfact = 0 ;
                        root = node2 ;
                    }
                    root -> balfact = 0 ;
                    *h = FALSE ;
                    break ;

                case 0 :
                    root -> balfact = 1 ;
                    break ;
            }
        }
    }

```

```

                                case -1 :
                                    root -> balfact = 0 ;
                                    *h = FALSE ;
                                }
                            }
    }

    if ( data > root -> data )
    {
        root -> right = buildtree ( root -> right, data, h ) ;

        if ( *h )
        {
            switch ( root -> balfact )
            {
                case 1 :
                    root -> balfact = 0 ;
                    *h = FALSE ;
                    break ;
                case 0 :
                    root -> balfact = -1 ;
                    break ;
                case -1 :
                    node1 = root -> right ;
                    if ( node1 -> balfact == -1 )
                    {
                        cout << "\nLeft rotation." ;
                        root -> right = node1 -> left ;
                        node1 -> left = root ;
                        root -> balfact = 0 ;
                        root = node1 ;
                    }
                    else
                    {
                        cout << "\nDouble rotation, right then left." ;
                        node2 = node1 -> left ;
                        node1 -> left = node2 -> right ;
                        node2 -> right = node1 ;
                        root -> right = node2 -> left ;
                        node2 -> left = root ;
                        if ( node2 -> balfact == -1 )
                            root -> balfact = 1 ;
                        else
                            root -> balfact = 0 ;
                        if ( node2 -> balfact == 1 )
                            node1 -> balfact = -1 ;
                        else
                            node1 -> balfact = 0 ;
                    }
                }
            }
        }
    }

```



```

                                root = node2 ;
                                }
                                root -> balfact = 0 ;
                                *h = FALSE ;
                                }
                                }
                                }
                                return ( root ) ;
                                }
                                }
void avltree :: display ( AVLNode* root )
{
    if ( root != NULL )
    {
        display ( root -> left ) ;
        cout << root -> data << "\t" ;
        display ( root -> right ) ;
    }
}
AVLNode* avltree :: deldata ( AVLNode *root, int data, int *h )
{
    AVLNode *node ;
    if ( root -> data == 13 )
        cout << root -> data ;
    if ( root == NULL )
    {
        cout << "\nNo such data." ;
        return ( root ) ;
    }
    else
    {
        if ( data < root -> data )
        {
            root -> left = deldata ( root -> left, data, h ) ;
            if ( *h )
                root = balright ( root, h ) ;
        }
        else
        {
            if ( data > root -> data )
            {
                root -> right = deldata ( root -> right, data, h ) ;
                if ( *h )
                    root = balleft ( root, h ) ;
            }
            else
            {
                node = root ;
                if ( node -> right == NULL )

```

```

        {
            root = node -> left ;
            *h = TRUE ;
            delete ( node ) ;
        }
        else
        {
            if ( node -> left == NULL )
            {
                root = node -> right ;
                *h = TRUE ;
                delete ( node ) ;
            }
            else
            {
                node -> right = del ( node -> right, node, h ) ;
                if ( *h )
                    root = balleft ( root, h ) ;
            }
        }
    }
}
return ( root ) ;
}
AVLNode* avltree :: del ( AVLNode *succ, AVLNode *node, int *h )
{
    AVLNode *temp = succ ;

    if ( succ -> left != NULL )
    {
        succ -> left = del ( succ -> left, node, h ) ;
        if ( *h )
            succ = balright ( succ, h ) ;
    }
    else
    {
        temp = succ ;
        node -> data = succ -> data ;
        succ = succ -> right ;
        delete ( temp ) ;
        *h = TRUE ;
    }
    return ( succ ) ;
}
AVLNode* avltree :: balright ( AVLNode *root, int *h )
{
    AVLNode *temp1, *temp2 ;

```

```

switch ( root -> balfact )
{
    case 1 :
        root -> balfact = 0 ;
        break ;
    case 0 :
        root -> balfact = -1 ;
        *h = FALSE ;
        break ;
    case -1 :
        temp1 = root -> right ;
        if ( temp1 -> balfact <= 0 )
        {
            cout << "\nLeft rotation." ;
            root -> right = temp1 -> left ;
            temp1 -> left = root ;
            if ( temp1 -> balfact == 0 )
            {
                root -> balfact = -1 ;
                temp1 -> balfact = 1 ;
                *h = FALSE ;
            }
            else
            {
                root -> balfact = temp1 -> balfact = 0 ;
            }
            root = temp1 ;
        }
        else
        {
            cout << "\nDouble rotation, right then left." ;
            temp2 = temp1 -> left ;
            temp1 -> left = temp2 -> right ;
            temp2 -> right = temp1 ;
            root -> right = temp2 -> left ;
            temp2 -> left = root ;
            if ( temp2 -> balfact == -1 )
                root -> balfact = 1 ;
            else
                root -> balfact = 0 ;
            if ( temp2 -> balfact == 1 )
                temp1 -> balfact = -1 ;
            else
                temp1 -> balfact = 0 ;
            root = temp2 ;
            temp2 -> balfact = 0 ;
        }
    }
}

```

```

        return ( root ) ;
    }
AVLNode* avltree :: balleft ( AVLNode *root, int *h )
{
    AVLNode *temp1, *temp2 ;
    switch ( root -> balfact )
    {
        case -1 :
            root -> balfact = 0 ;
            break ;

        case 0 :
            root -> balfact = 1 ;
            *h = FALSE ;
            break ;

        case 1 :
            temp1 = root -> left ;
            if ( temp1 -> balfact >= 0 )
            {
                cout << "\nRight rotation." ;
                root -> left = temp1 -> right ;
                temp1 -> right = root ;

                if ( temp1 -> balfact == 0 )
                {
                    root -> balfact = 1 ;
                    temp1 -> balfact = -1 ;
                    *h = FALSE ;
                }
                else
                {
                    root -> balfact = temp1 -> balfact = 0 ;
                }
                root = temp1 ;
            }
            else
            {
                cout << "\nDouble rotation, left then right." ;
                temp2 = temp1 -> right ;
                temp1 -> right = temp2 -> left ;
                temp2 -> left = temp1 ;
                root -> left = temp2 -> right ;
                temp2 -> right = root ;
                if ( temp2 -> balfact == 1 )
                    root -> balfact = -1 ;
                else
                    root -> balfact = 0 ;
            }
    }
}

```

```

        if ( temp2-> balfact == -1 )
            temp1 -> balfact = 1 ;
        else
            temp1 -> balfact = 0 ;
        root = temp2 ;
        temp2 -> balfact = 0 ;
    }
}
return ( root ) ;
}
void avltree :: setroot ( AVLNode *avl )
{
    root = avl ;
}
avltree :: ~avltree( )
{
    deltree ( root ) ;
}

void avltree :: deltree ( AVLNode *root )
{
    if ( root != NULL )
    {
        deltree ( root -> left ) ;
        deltree ( root -> right ) ;
    }
    delete ( root ) ;
}
void main( )
{
    avltree at ;
    AVLNode *avl = NULL ;
    int h ;
    clrscr();
    avl = at.insert ( 20, &h ) ;
    at.setroot ( avl ) ;
    avl = at.insert ( 6, &h ) ;
    at.setroot ( avl ) ;
    avl = at.insert ( 29, &h ) ;
    at.setroot ( avl ) ;
    avl = at.insert ( 5, &h ) ;
    at.setroot ( avl ) ;
    avl = at.insert ( 12, &h ) ;
    at.setroot ( avl ) ;
    avl = at.insert ( 25, &h ) ;
    at.setroot ( avl ) ;
    avl = at.insert ( 32, &h ) ;

```

```

        at.setroot ( avl ) ;
        avl = at.insert ( 10, &h ) ;
        at.setroot ( avl ) ;
        avl = at.insert ( 15, &h ) ;
        at.setroot ( avl ) ;
        avl = at.insert ( 27, &h ) ;
        at.setroot ( avl ) ;
        avl = at.insert ( 13, &h ) ;
        at.setroot ( avl ) ;
        cout << endl << "AVL tree:\n" ;
        at.display ( avl ) ;
        avl = at.deldata ( avl, 20, &h ) ;
        at.setroot ( avl ) ;
        avl = at.deldata ( avl, 12, &h ) ;
        at.setroot ( avl ) ;
        cout << endl << "AVL tree after deletion of a node:\n" ;
        at.display ( avl ) ;
        getch();
    }

```

Splay Trees

A **splay tree** is a self-adjusting binary search tree with the additional property that recently accessed elements are quick to access again. It performs basic operations such as insertion, look-up and removal in $O(\log n)$ amortized time. For many sequences of nonrandom operations, splay trees perform better than other search trees, even when the specific pattern of the sequence is unknown. The splay tree was invented by Daniel Dominic Sleator and Robert Endre Tarjan in 1985.

All normal operations on a binary search tree are combined with one basic operation, called *splaying*. Splaying the tree for a certain element rearranges the tree so that the element is placed at the root of the tree. One way to do this is to first perform a standard binary tree search for the element in question, and then use tree rotations in a specific fashion to bring the element to the top. Alternatively, a top-down algorithm can combine the search and the tree reorganization into a single phase.

Splaying

When a node x is accessed, a splay operation is performed on x to move it to the root. To perform a splay operation we carry out a sequence of *splay steps*, each of which moves x closer to the root. By performing a splay operation on the node of interest after every access, the recently accessed nodes are kept near the root and the tree remains roughly balanced, so that we achieve the desired amortized time bounds.

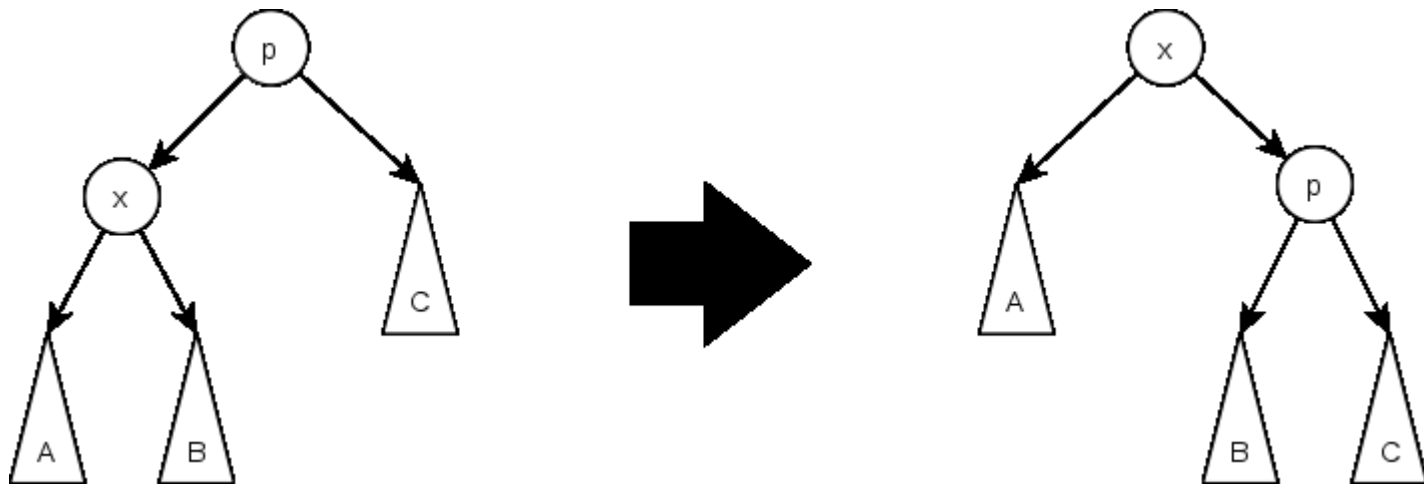
Each particular step depends on three factors:

- Whether x is the left or right child of its parent node, p ,
- whether p is the root or not, and if not
- whether p is the left or right child of *its* parent, g (the *grandparent* of x).

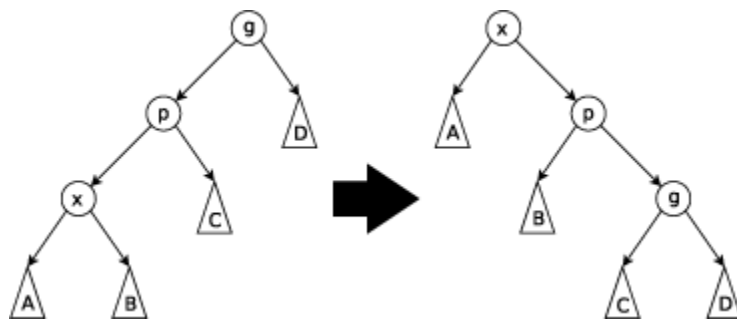
It is important to remember to set gg (the *great-grandparent* of x) to now point to x after any splay operation. If gg is null, then x obviously is now the root and must be updated as such.

The three types of splay steps are:

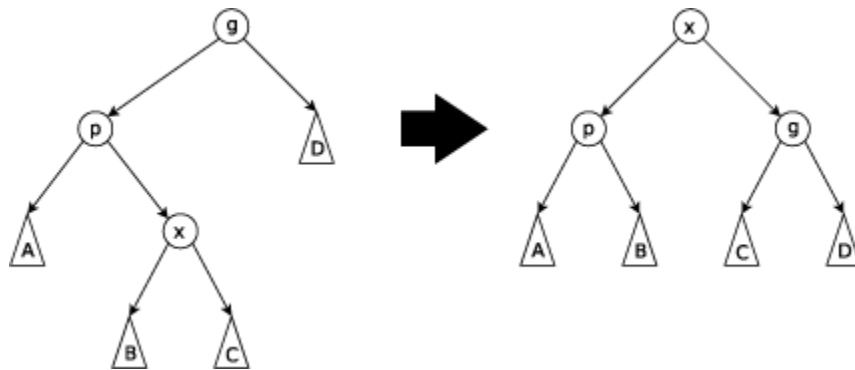
Zig Step: This step is done when p is the root. The tree is [rotated](#) on the edge between x and p . Zig steps exist to deal with the parity issue and will be done only as the last step in a splay operation and only when x has odd depth at the beginning of the operation.



Zig-zig Step: This step is done when p is not the root and x and p are either both right children or are both left children. The picture below shows the case where x and p are both left children. The tree is rotated on the edge joining p with *its* parent g , then rotated on the edge joining x with p . Note that zig-zig steps are the only thing that differentiate splay trees from the *rotate to root* method introduced by Allen and Munro prior to the introduction of splay trees.



Zig-zag Step: This step is done when p is not the root and x is a right child and p is a left child or vice versa. The tree is rotated on the edge between x and p , then rotated on the edge between x and its new parent g .



Insertion

To insert a node x into a splay tree:

1. First insert the node as with a normal binary search tree.
2. Then splay the newly inserted node x to the top of the tree.

Deletion

To delete a node x , we use the same method as with a binary search tree: if x has two children, we swap its value with that of either the rightmost node of its left sub tree (its in-order predecessor) or the leftmost node of its right subtree (its in-order successor). Then we remove that node instead. In this way, deletion is reduced to the problem of removing a node with 0 or 1 children.

Unlike a binary search tree, in a splay tree after deletion, we splay the parent of the removed node to the top of the tree. **OR** The node to be deleted is first splayed, i.e. brought to the root of the tree and then deleted. This leaves the tree with two sub trees.

The maximum element of the left sub tree (: **METHOD 1**), or minimum of the right sub tree (: **METHOD 2**) is then splayed to the root. The right sub tree is made the right child of the resultant left sub tree (for **METHOD 1**). The root of left sub tree is the root of melded tree.

//Splay Trees implementation

```
#include<iostream>
using namespace std;
class node
{
    public:
    int data;
    node *parent;
    node *left;
    node *right;
};
class SplayTree
{
    node* root;
    public:
    SplayTree()
    {
        root=NULL;
    }
    void splay (node *x)
    {
        node *p,*g;
        /*check if node x is the root node*/
        if(x==root)
            return;
        /*Performs Zig step*/
        else if(x->parent==root)
        {
            if(x==x->parent->left)
                root=rightrotation(root);
            else
                root=leftrotation(root);
        }
        else
        {
            p=x->parent; /*now points to parent of x*/
            g=p->parent; /*now points to parent of x's parent*/
            /*Performs the Zig-zig step when x is left and x's parent is left*/
            if(x==p->left&&g==p->left)
            {
                root=rightrotation(g,root);
                root=rightrotation(p,root);
            }
        }
    }
}
```

```

/*Performs the Zig-zig step when x is right and x's parent is right*/
else if(x==p->right&&g==p->right)
{
    root=leftrotation(g,root);
    root=leftrotation(p,root);
}
/*Performs the Zig-zag step when x's is right and x's parent is left*/
else if(x==p->right&&g==p->left)
{
    root=leftrotation(p,root);
    root=rightrotation(g,root);
}
/*Performs the Zig-zag step when x's is left and x's parent is right*/
else if(x==p->left&&g==p->right)
{
    root=rightrotation(p,root);
    root=leftrotation(g,root);
}
splay(x, root);
}
}
node *rightrotation(node *p)
{
    node *x;
    x = p->left;
    p->left = x->right;
    if (x->right!=NULL) x->right->parent = p;
    x->right = p;
    if (p->parent!=NULL)
        if(p==p->parent->right) p->parent->right=x;
        else
            p->parent->left=x;
    x->parent = p->parent;
    p->parent = x;
    if (p==root)
        return x;
    else
        return root;
}
node *leftrotation(node *p)
{
    node *x;
    x = p->right;
    p->right = x->left;
    if (x->left!=NULL) x->left->parent = p;
    x->left = p;
    if (p->parent!=NULL)
        if (p==p->parent->left) p->parent->left=x;

```

```

        else
            p->parent->right=x;
x->parent = p->parent;
p->parent = x;
if(p==root)
    return x;
else
    return root;
}
void insert(int value)
{
    node *temp1,*temp2,*par,*x;
    if(root == NULL)
    {
        root=new node;
        if(root != NULL)
        {
            root->data = value;
            root->parent = NULL;
            root->left = NULL;
            root->right = NULL;
        }
        else
        {
            cout<<"No memory is allocated\n";
            exit(0);
        }
    } //the case 2 says that we must splay newly inserted node to root
    else
    {
        temp2 = root;
        while(temp2 != NULL)
        {
            temp1 = temp2;
            if(temp2->data > value)
                temp2 = temp2->left;
            else if(temp2->data < value)
                temp2 = temp2->right;
            else
                if(temp2->data == value)
                    return temp2;
        }
        if(temp1->data > value)
        {
            par = temp1;//temp1 having the parent address,so that's it
            temp1->left = new node;
            temp1= temp1->left;
            if(temp1 != NULL)

```

```

        {
            temp1->data = value;
            temp1->parent = par;//store the parent address.
            temp1->left = NULL;
            temp1->right = NULL;
        }
        else
        {
            cout<<"No memory is allocated\n";
            exit(0);
        }
    }
    else
    {
        par = temp1;//temp1 having the parent node address.
        temp1->right = new node;
        temp1 = temp1->right;
        if(temp1 != NULL)
        {
            temp1->data = value;
            temp1->parent = par;//store the parent address
            temp1->left = NULL;
            temp1->right = NULL;
        }
        else
        {
            cout<<"No memory is allocated\n";
            exit(0);
        }
    }
}
splay(temp1);//temp1 will be new root after splaying
root=temp1;
}
void inorder( node *p)
{
    if(p != NULL)
    {
        inorder(p->left);
        cout<<"CURRENT %d\t"<<p->data;
        cout<<"LEFT %d\t"<<data_print(p->left);
        cout<<"PARENT %d\t"<<data_print(p->parent);
        cout<<"RIGHT %d\t\n"<<data_print(p->right);
        inorder(p->right);
    }
}
void Delete(int value)
{

```

```

node *x,*y,*p1;
node *s;
x = lookup(root,value);
if(x->data == value)
{ //if the deleted element is leaf
  if((x->left == NULL) && (x->right == NULL))
  {
    y = x->parent;
    if(x == (x->parent->right))
      y->right = NULL;
    else
      y->left = NULL;
    delete x;
  }
  //if deleted element having left child only
  else if((x->left != NULL) && (x->right == NULL))
  {
    if(x == (x->parent->left))
    {
      y = x->parent;
      x->left->parent = y;
      y->left = x->left;
      delete x;
    }
    else
    {
      y = x->parent;
      x->left->parent = y;
      y->right = x->left;
      delete x;
    }
  }
  //if deleted element having right child only
  else if((x->left == NULL) && (x->right != NULL))
  {
    if(x == (x->parent->left))
    {
      y = x->parent;
      x->right->parent = y;
      y->left = x->right;
      delete x;
    }
    else
    {
      y = x->parent;
      x->right->parent = y;
      y->right = x->right;
      delete x;
    }
  }
}

```

```

    }
}
//if the deleted element having two child
else if((x->left != NULL) && (x->right != NULL))
{
    if(x == (x->parent->left))
    {
        s = sucessor(x);
        if(s != x->right)
        {
            y = s->parent;
            if(s->right != NULL)
            {
                s->right->parent = y;
                y->left = s->right;
            }
            else y->left = NULL;
            s->parent = x->parent;
            x->right->parent = s;
            x->left->parent = s;
            s->right = x->right;
            s->left = x->left;
            x->parent->left = s;
        }
    }
    else
    {
        y = s;
        s->parent = x->parent;
        x->left->parent = s;
        s->left = x->left;
        x->parent->left = s;
    }
    delete x;
}
else if(x == (x->parent->right))
{
    s = sucessor(x);
    if(s != x->right)
    {
        y = s->parent;
        if(s->right != NULL)
        {
            s->right->parent = y;
            y->left = s->right;
        }
        else y->left = NULL;
        s->parent = x->parent;
        x->right->parent = s;
    }
}

```

```

        x->left->parent = s;
        s->right = x->right;
        s->left = x->left;
        x->parent->right = s;
    }
    else
    {
        y = s;
        s->parent = x->parent;
        x->left->parent = s;
        s->left = x->left;
        x->parent->right = s;
    }
    delete x;
}

}
splay(y);
}
else
{
    splay(x);
}
}
}
node *sucessor(node *x)
{
    node *temp,*temp2;
    temp=temp2=x->right;
    while(temp != NULL)
    {
        temp2 = temp;
        temp = temp->left;
    }
    return temp2;
}
//p is a root element of the tree
node *lookup(node *p,int value)
{
    node *temp1,*temp2;
    if(p != NULL)
    {
        temp1 = p;
        while(temp1 != NULL)
        {
            temp2 = temp1;
            if(temp1->data > value)
                temp1 = temp1->left;
            else if(temp1->data < value)

```

```

        temp1 = temp1->right;
    else
        return temp1;
    }
    return temp2;
}
else
{
    cout<<"NO element in the tree\n";
    exit(0);
}
}
node *search(int value)
{
    node *x;
    x = lookup(root,value);
    if(x->data == value)
    {
        cout<<"Inside search if\n";
        splay(x,root);
    }
    else
    {
        cout<<"Inside search else\n";
        splay(x,root);
    }
}
int data_print(struct node *x)
{
    if ( x==NULL )
        return 0;
    else
        return x->data;
}
};
main()
{
    SplayTree st;
    int i,choice = 0,ele;
    while(1)
    {
        cout<<"\n\n 1.Insert";
        cout<<"\n\n 2.Delete";
        cout<<"\n\n 3.Search";
        cout<<"\n\n 4.Display\n";
        cout<<"\n\n Enter your choice:";
        cin>>choice;
        if(choice==5)

```



```
        exit(0);
    switch(choice)
    {
        case 1:
            cout<<"\n\n Enter the element to be inserted:";
            cin>>ele;
            st.insert(ele);
            st.splay(x);
            break;
        case 2:
            cout<<"\n\n Enter the element to be delete:";
            cin>>ele;
            st.Delete(ele);
            break;
        case 3:
            cout<<"Enter the element to be search\n";
            cin>>ele;
            st.search(ele);
            break;
        case 4:
            cout<<"The elements are\n";
            st.inorder(root);
            break;
        default:
            cout<<"Wrong choice\n";
            break;
    }
}
```