PRACTICAL

**1. i. Implement Insertion Sort (The program should report the number of comparisons)**

**Source code**

#include<iostream.h>

#include<conio.h>

void main()

{

clrscr();

int n;

int p[20];

cout<<"Enter the number of elements to be sorted"<<endl;

cin>>n;

cout<<"Enter the elements to be sorted"<<endl;

for(int i=0;i<n;i++)

cin>>p[i];

int c=0;

for(i=1;i<n;i++)

{

int value=p[i];

int hole=i;

while(++c && hole>0 && p[hole-1]>value)

{

p[hole]=p[hole-1];

hole--;

}

p[hole]=value;

}

cout<<"the array is"<<endl;

for(int u=0;u<n;u++)

{

cout<<p[u]<<endl;

}

cout<<"The number of comparisons are "<<c<<endl;

getch();

}

**Output**



**ii. Implement Merge Sort(The program should report the number of comparisons)**

**Source code** #include<iostream.h>

#include<conio.h>

int c1=0;

void merge(int a[],int l,int m,int r)

{

int L[10];

int c[10];

int i,j,k;

i=l;

k=l;

j=m+1;

while(++c1 && i <=m && j <=r )

{

if(a[i]<=a[j])

{

c[k]=a[i];

k++;

i++;

}

else

{

c[k]=a[j];

k++;

j++;

}

}

while(i<=m)

{

c[k]=a[i];

k++;

i++;

}

while(j<=r)

{

c[k]=a[j];

k++;

j++;

}

for(i=l;i<k;i++)

{

a[i]=c[i];

}

}

void sort(int a[],int l,int r)

{

if(l<r)

{

int m=(l+r)/2;

sort(a,l,m);

sort(a,m+1,r);

merge(a,l,m,r);

}

}

int main()

{

clrscr();

int n;

int f[10];

cout<<"Enter the number of elements"<<endl;

cin>>n;

cout<<"Enter the elements"<<endl;

for(int i=0;i<n;i++)

{

cin>>f[i];

}

sort(f,0,n-1);

cout<<"Sorted array using merge sort is "<<endl;

for( i=0;i<n;i++)

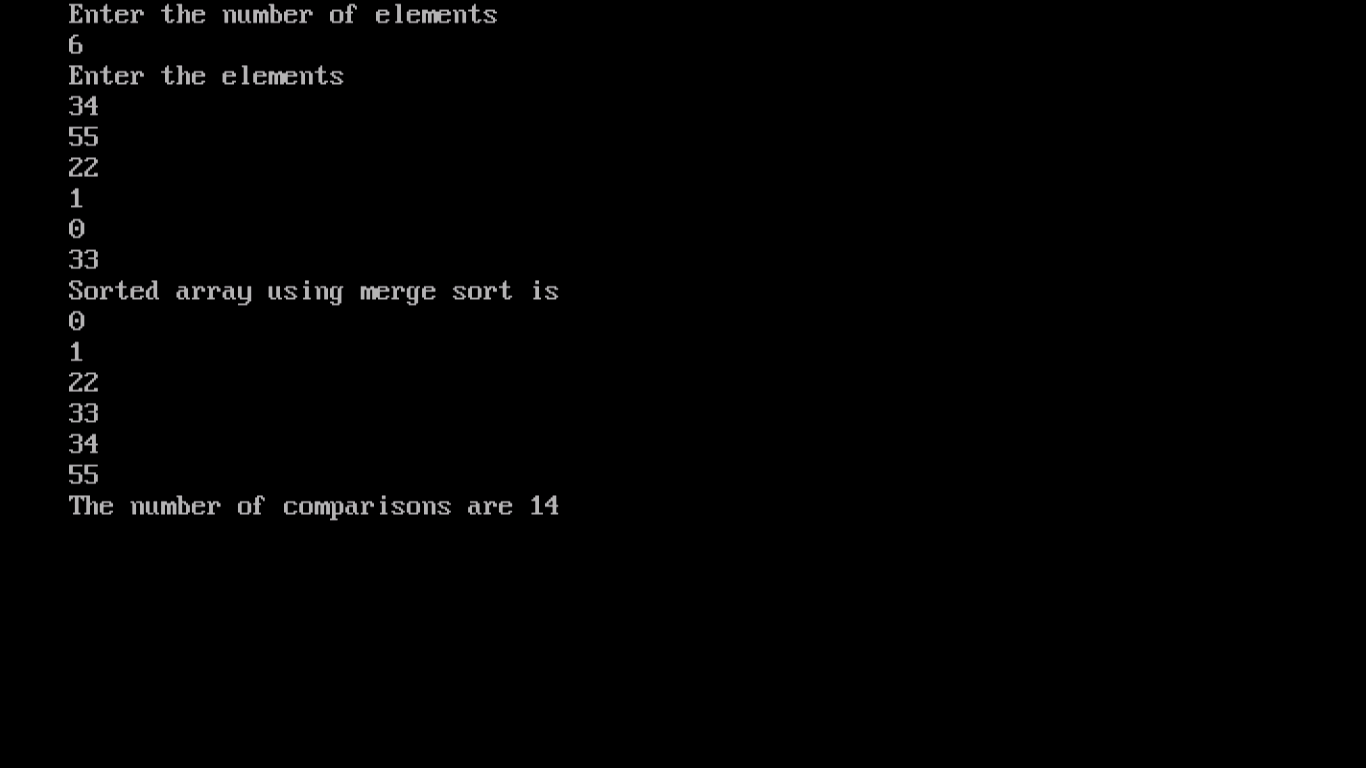
cout<<f[i]<<endl;

cout<<"The number of comparisons are "<<c1<<endl;

getch();

}

**Output**



**2. Implement Heap Sort(The program should report the number of comparisons)**

**Source code**

#include<iostream.h>

#include<conio.h>

static int c=0;

void maxheapify(int a[],int n,int i)

{

int large=i;

int left=2\*i+1;

int right=2\*i+2;

if(++c && a[large]<a[left] && left<n)

{

large=left;

}

if(++c && a[large]<a[right] && right<n)

{

large=right;

}

if(large!=i)

{

int t=a[large];

a[large]=a[i];

a[i]=t;

maxheapify(a,n,large);

}

}

void buildheap(int a[],int n)

{

for(int i=n/2-1;i>=0;i--)

{

maxheapify(a,n,i);

}

for(i=n-1;i>=0;i--)

{

int t=a[0];

a[0]=a[i];

a[i]=t;

maxheapify(a,i,0);

}

cout<<"sorted heap is "<<endl;

for( i=0;i<n;i++)

{

cout<<a[i]<<endl;

}

}

void main()

{

int n;

int a[20];

clrscr();

cout<<"enter the number of elements to be sorted"<<endl;

cin>>n;

cout<<"enter the elements to be sorted"<<endl;

for(int i=0;i<n;i++)

cin>>a[i];

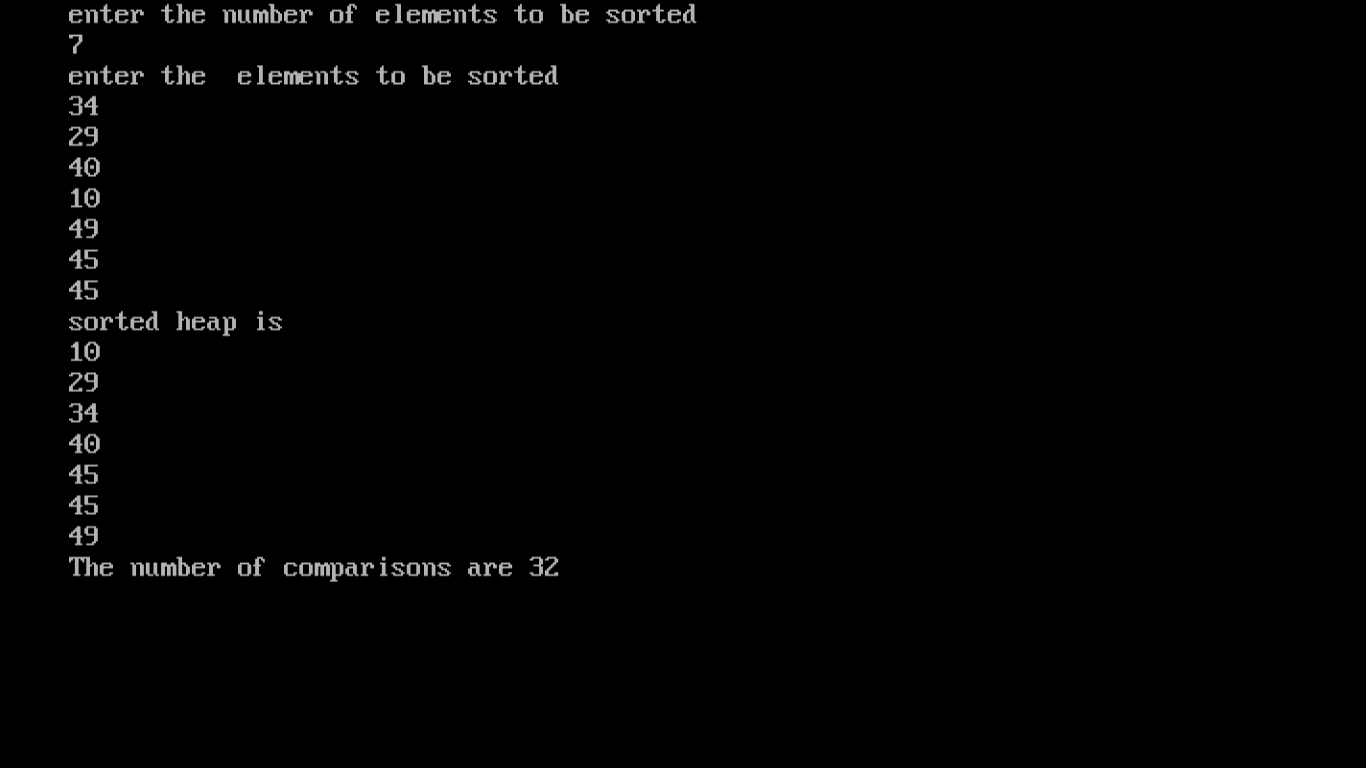
buildheap(a,n);

cout<<"The number of comparisons are "<<c<<endl;

getch();

}

**Output**



**3. Implement Randomized Quick sort (The program should report the number of comparisons)**

**Source code**

#include<iostream.h>

#include<conio.h>

#include<stdlib.h>

int c=0;

int partition(int a[],int p,int r)

{

int x=a[r];

int i=p-1;

for(int j=p;j<=(r-1);j++)

{

if(++c && a[j]<=x)

{

i=i+1;

int temp=a[i];

a[i]=a[j];

a[j]=temp;

}

}

int temp=a[r];

a[r]=a[i+1];

a[i+1]=temp;

return i+1;

}

int randomizedpartition(int a[],int p,int r)

{

int i=p+(rand()%(int) (r-p));

int temp=a[r];

a[r]=a[i];

a[i]=temp;

return partition(a,p,r);

}

void randomizedquicksort(int a[],int p,int r)

{

if(p<r)

{

int q=randomizedpartition(a,p,r);

randomizedquicksort(a,p,q-1);

randomizedquicksort(a,q+1,r);

}

}

int main()

{

clrscr();

int a[10];

int n;

cout<<"Enter the number of elements"<<endl;

cin>>n;

cout<<"Enter the elements"<<endl;

for(int i=0;i<n;i++)

cin>>a[i];

randomizedquicksort(a,0,n-1);

cout<<"The sorted elements are"<<endl;

for( i=0;i<n;i++)

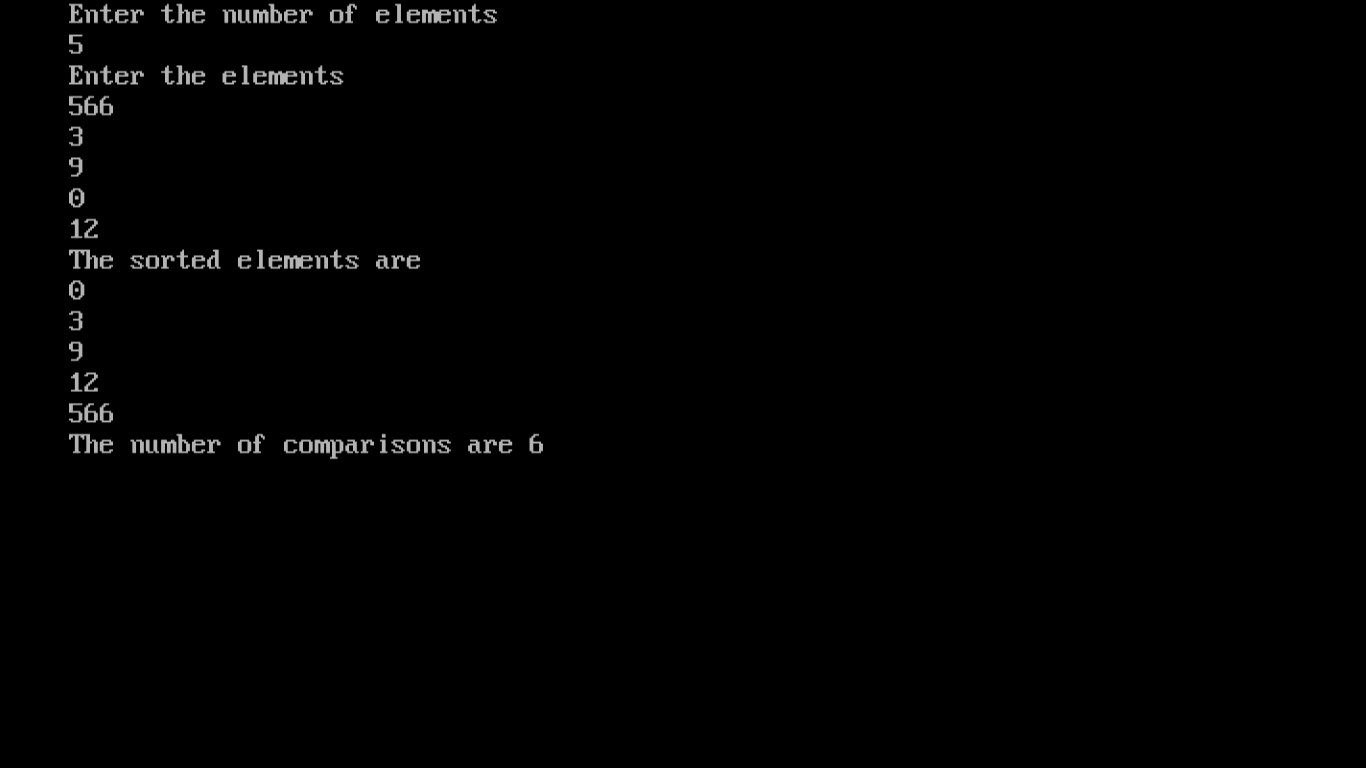
cout<<a[i]<<endl;

cout<<"The number of comparisons are "<<c<<endl;

getch();

}

**Output**



**4.Implement Radix Sort**

**Source code**

#include<iostream.h>

#include<conio.h>

void countsort(int a[],int n,int k,int e)

{

int c[100];

for(int i=0;i<=k;i++)

{

c[i]=0;

}

int b[100];

for(i=0;i<n;i++)

{

c[(a[i]/e)%k]++;

}

for(i=1;i<=k;i++)

{

c[i]=c[i]+c[i-1];

}

for(i=0;i<n;i++)

b[i]=0;

for(int j=n-1;j>=0;j--)

{

b[c[(a[j]/e)%k]-1]=a[j];

c[(a[j]/e)%k]--;

}

for(i=0;i<n;i++)

a[i]=b[i];

}

void main()

{

clrscr();

int a[100],b[10];

int n,max;

cout<<"Enter the upper bound or the maximum element to be stored in the array"<<endl;

cin>>max;;

cout<<"Enter the number of elements of the array"<<endl;

cin>>n;

cout<<"Enter the elements"<<endl;

for(int i=0;i<n;i++)

{

cin>>a[i];

}

for(i=1;max/i>0;i=i\*10)

{

countsort(a,n,10,i);

}

cout<<"The elements are"<<endl;

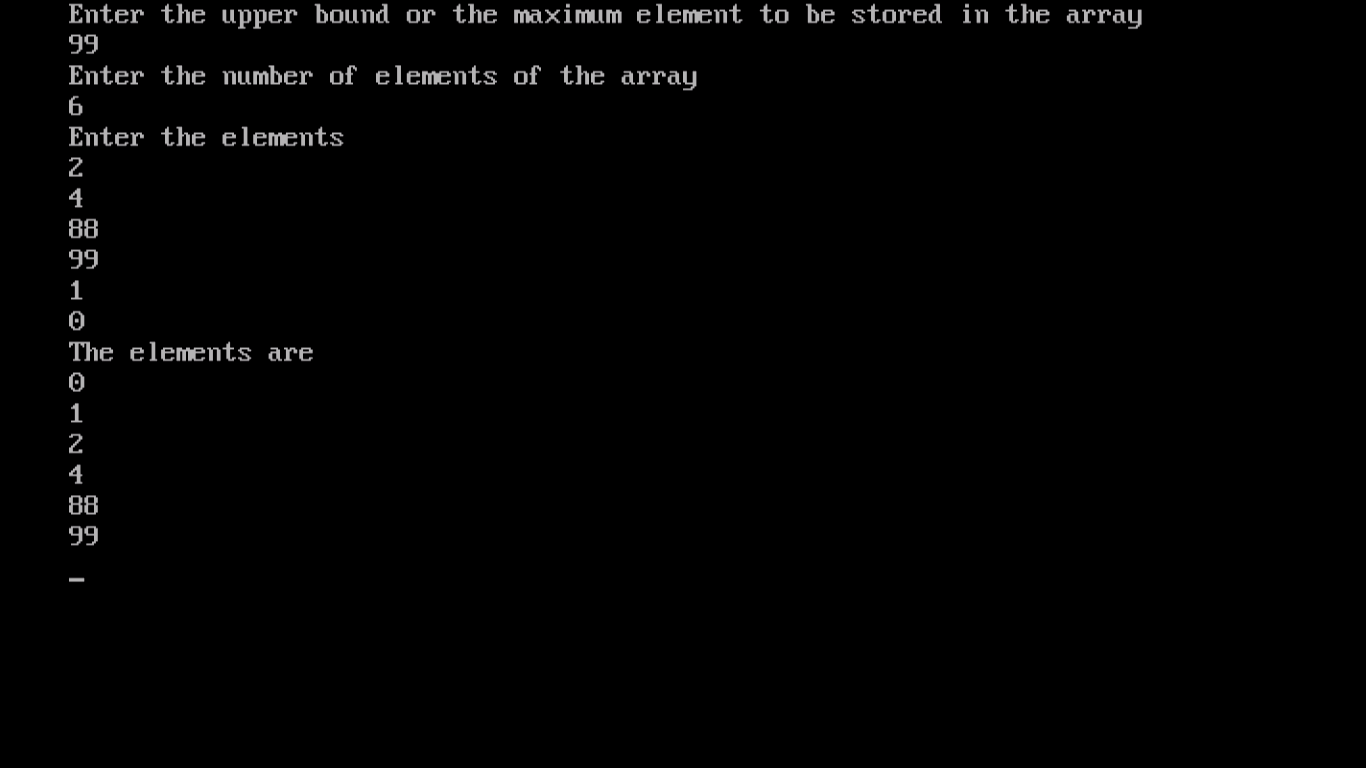
for(i=0;i<n;i++)

cout<<a[i]<<endl;

getch();

}

**Output**



**5. Create a Red-Black Tree and perform following operations on it:**

**i. Insert a node**

**ii. Delete a node**

**iii. Search for a number & also report the colour of the node containing this number.**

**Source code**

#include<iostream.h>

#include<conio.h>

struct node

{

int key;

node \*parent;

char color;

node \*left;

node \*right;

};

class RBtree

{

node \*root;

node \*q;

public :

RBtree()

{

q=NULL;

root=NULL;

}

void insert();

void insertfix(node \*);

void leftrotate(node \*);

void rightrotate(node \*);

void del();

node\* successor(node \*);

void delfix(node \*);

void disp();

void display( node \*);

void search();

};

void RBtree::insert()

{

int z,i=0;

cout<<"\nEnter key of the node to be inserted: ";

cin>>z;

node \*p,\*q;

node \*t=new node;

t->key=z;

t->left=NULL;

t->right=NULL;

t->color='r';

p=root;

q=NULL;

if(root==NULL)

{

root=t;

t->parent=NULL;

}

else

{

while(p!=NULL)

{

q=p;

if(p->key<t->key)

p=p->right;

else

p=p->left;

}

t->parent=q;

if(q->key<t->key)

q->right=t;

else

q->left=t;

}

insertfix(t);

}

void RBtree::insertfix(node \*t)

{

node \*u;

if(root==t)

{

t->color='b';

return;

}

while(t->parent!=NULL&&t->parent->color=='r')

{

node \*g=t->parent->parent;

if(g->left==t->parent)

{

if(g->right!=NULL)

{

u=g->right;

if(u->color=='r')

{

t->parent->color='b';

u->color='b';

g->color='r';

t=g;

}

}

else

{

if(t->parent->right==t)

{

t=t->parent;

leftrotate(t);

}

t->parent->color='b';

g->color='r';

rightrotate(g);

}

}

else

{

if(g->left!=NULL)

{

u=g->left;

if(u->color=='r')

{

t->parent->color='b';

u->color='b';

g->color='r';

t=g;

}

}

else

{

if(t->parent->left==t)

{

t=t->parent;

rightrotate(t);

}

t->parent->color='b';

g->color='r';

leftrotate(g);

}

}

root->color='b';

}

}

void RBtree::del()

{

if(root==NULL)

{

cout<<"\nEmpty Tree." ;

return ;

}

int x;

cout<<"\nEnter the key of the node to be deleted: ";

cin>>x;

node \*p;

p=root;

node \*y=NULL;

node \*q=NULL;

int found=0;

while(p!=NULL&&found==0)

{

if(p->key==x)

found=1;

if(found==0)

{

if(p->key<x)

p=p->right;

else

p=p->left;

}

}

if(found==0)

{

cout<<"\nElement Not Found.";

return ;

}

else

{

cout<<"\nDeleted Element: "<<p->key;

cout<<"\nColour: ";

if(p->color=='b')

cout<<"Black\n";

else

cout<<"Red\n";

if(p->parent!=NULL)

cout<<"\nParent: "<<p->parent->key;

else

cout<<"\nThere is no parent of the node. ";

if(p->right!=NULL)

cout<<"\nRight Child: "<<p->right->key;

else

cout<<"\nThere is no right child of the node. ";

if(p->left!=NULL)

cout<<"\nLeft Child: "<<p->left->key;

else

cout<<"\nThere is no left child of the node. ";

cout<<"\nNode Deleted.";

if(p->left==NULL||p->right==NULL)

y=p;

else

y=successor(p);

if(y->left!=NULL)

q=y->left;

else

{

if(y->right!=NULL)

q=y->right;

else

q=NULL;

}

if(q!=NULL)

q->parent=y->parent;

if(y->parent==NULL)

root=q;

else

{

if(y==y->parent->left)

y->parent->left=q;

else

y->parent->right=q;

}

if(y!=p)

{

p->color=y->color;

p->key=y->key;

}

if(y->color=='b')

delfix(q);

}

}

void RBtree::delfix(node \*p)

{

node \*s;

while(p!=root&&p->color=='b')

{

if(p->parent->left==p)

{

s=p->parent->right;

if(s->color=='r')

{

s->color='b';

p->parent->color='r';

leftrotate(p->parent);

s=p->parent->right;

}

if(s->right->color=='b'&&s->left->color=='b')

{

s->color='r';

p=p->parent;

}

else

{

if(s->right->color=='b')

{

s->left->color=='b';

s->color='r';

rightrotate(s);

s=p->parent->right;

}

s->color=p->parent->color;

p->parent->color='b';

s->right->color='b';

leftrotate(p->parent);

p=root;

}

}

else

{

s=p->parent->left;

if(s->color=='r')

{

s->color='b';

p->parent->color='r';

rightrotate(p->parent);

s=p->parent->left;

}

if(s->left->color=='b'&&s->right->color=='b')

{

s->color='r';

p=p->parent;

}

else

{

if(s->left->color=='b')

{

s->right->color='b';

s->color='r';

leftrotate(s);

s=p->parent->left;

}

s->color=p->parent->color;

p->parent->color='b';

s->left->color='b';

rightrotate(p->parent);

p=root;

}

}

p->color='b';

root->color='b';

}

}

void RBtree::leftrotate(node \*p)

{

if(p->right==NULL)

return ;

else

{

node \*y=p->right;

if(y->left!=NULL)

{

p->right=y->left;

y->left->parent=p;

}

else

p->right=NULL;

if(p->parent!=NULL)

y->parent=p->parent;

if(p->parent==NULL)

root=y;

else

{

if(p==p->parent->left)

p->parent->left=y;

else

p->parent->right=y;

}

y->left=p;

p->parent=y;

}

}

void RBtree::rightrotate(node \*p)

{

if(p->left==NULL)

return ;

else

{

node \*y=p->left;

if(y->right!=NULL)

{

p->left=y->right;

y->right->parent=p;

}

else

p->left=NULL;

if(p->parent!=NULL)

y->parent=p->parent;

if(p->parent==NULL)

root=y;

else

{

if(p==p->parent->left)

p->parent->left=y;

else

p->parent->right=y;

}

y->right=p;

p->parent=y;

}

}

node\* RBtree::successor(node \*p)

{

node \*y=NULL;

if(p->left!=NULL)

{

y=p->left;

while(y->right!=NULL)

y=y->right;

}

else

{

y=p->right;

while(y->left!=NULL)

y=y->left;

}

return y;

}

void RBtree::disp()

{

display(root);

}

void RBtree::display(node \*p)

{

if(root==NULL)

{

cout<<"\nEmpty Tree.";

return ;

}

if(p!=NULL)

{

cout<<"\n\t NODE: ";

cout<<"\n Key: "<<p->key;

cout<<"\n Colour: ";

if(p->color=='b')

cout<<"Black";

else

cout<<"Red";

if(p->parent!=NULL)

cout<<"\n Parent: "<<p->parent->key;

else

cout<<"\n There is no parent of the node. ";

if(p->right!=NULL)

cout<<"\n Right Child: "<<p->right->key;

else

cout<<"\n There is no right child of the node. ";

if(p->left!=NULL)

cout<<"\n Left Child: "<<p->left->key;

else

cout<<"\n There is no left child of the node. ";

cout<<endl;

if(p->left)

{

cout<<"\n\nLeft:\n";

display(p->left);

}

/\*else

cout<<"\nNo Left Child.\n";\*/

if(p->right)

{

cout<<"\n\nRight:\n";

display(p->right);

}

/\*else

cout<<"\nNo Right Child.\n"\*/

}

}

void RBtree::search()

{

if(root==NULL)

{

cout<<"\nEmpty Tree\n" ;

return ;

}

int x;

cout<<"\n Enter key of the node to be searched: ";

cin>>x;

node \*p=root;

int found=0;

while(p!=NULL&& found==0)

{

if(p->key==x)

found=1;

if(found==0)

{

if(p->key<x)

p=p->right;

else

p=p->left;

}

}

if(found==0)

cout<<"\nElement Not Found.";

else

{

cout<<"\n\t FOUND NODE: ";

cout<<"\n Key: "<<p->key;

cout<<"\n Colour: ";

if(p->color=='b')

cout<<"Black";

else

cout<<"Red";

if(p->parent!=NULL)

cout<<"\n Parent: "<<p->parent->key;

else

cout<<"\n There is no parent of the node. ";

if(p->right!=NULL)

cout<<"\n Right Child: "<<p->right->key;

else

cout<<"\n There is no right child of the node. ";

if(p->left!=NULL)

cout<<"\n Left Child: "<<p->left->key;

else

cout<<"\n There is no left child of the node. ";

cout<<endl;

}

}

int main()

{

clrscr();

int ch,y=0;

RBtree obj;

do

{

cout<<"\n\t \*\*\*\*\*\*\*\*RED BLACK TREE\*\*\*\*\*\*\*\*\*\* " ;

cout<<"\n 1. Insert in the tree ";

cout<<"\n 2. Delete a node from the tree";

cout<<"\n 3. Search for an element in the tree";

cout<<"\n 4. Display the tree ";

cout<<"\n 5. Exit " ;

cout<<"\nEnter Your Choice: ";

cin>>ch;

switch(ch)

{

case 1 :

obj.insert();

cout<<"\nNode Inserted.\n";

break;

case 2 :

obj.del();

break;

case 3 : obj.search();

break;

case 4 : obj.disp();

getch();

break;

case 5 : y=1;

break;

default : cout<<"\nEnter a Valid Choice.";

}

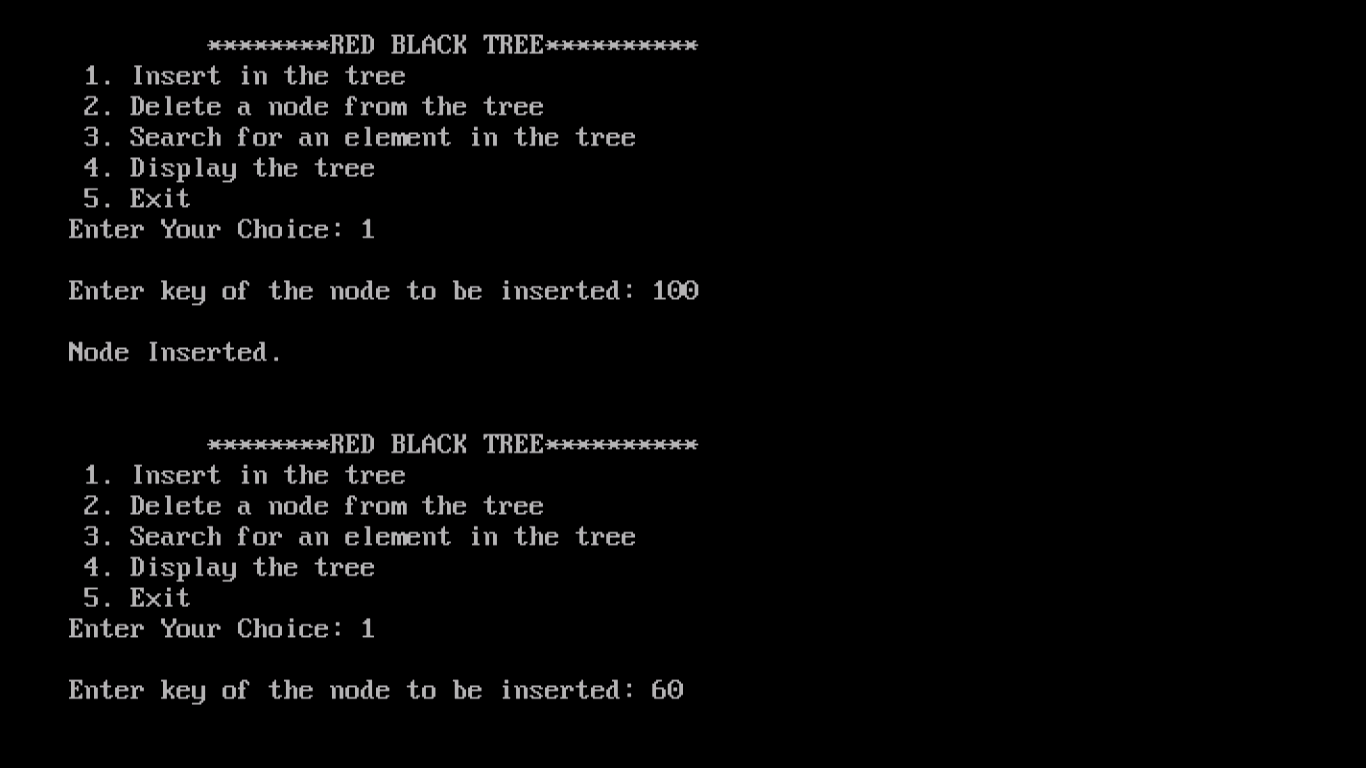
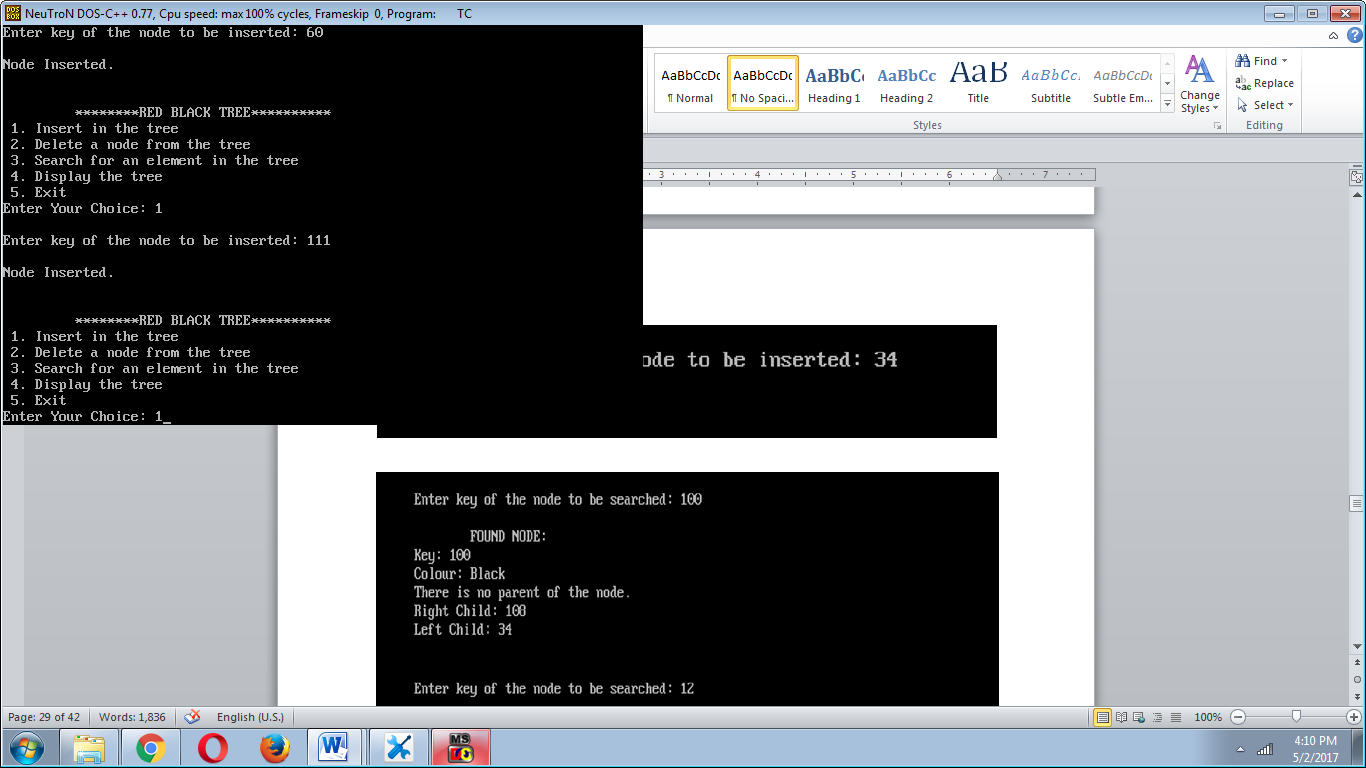
cout<<endl;

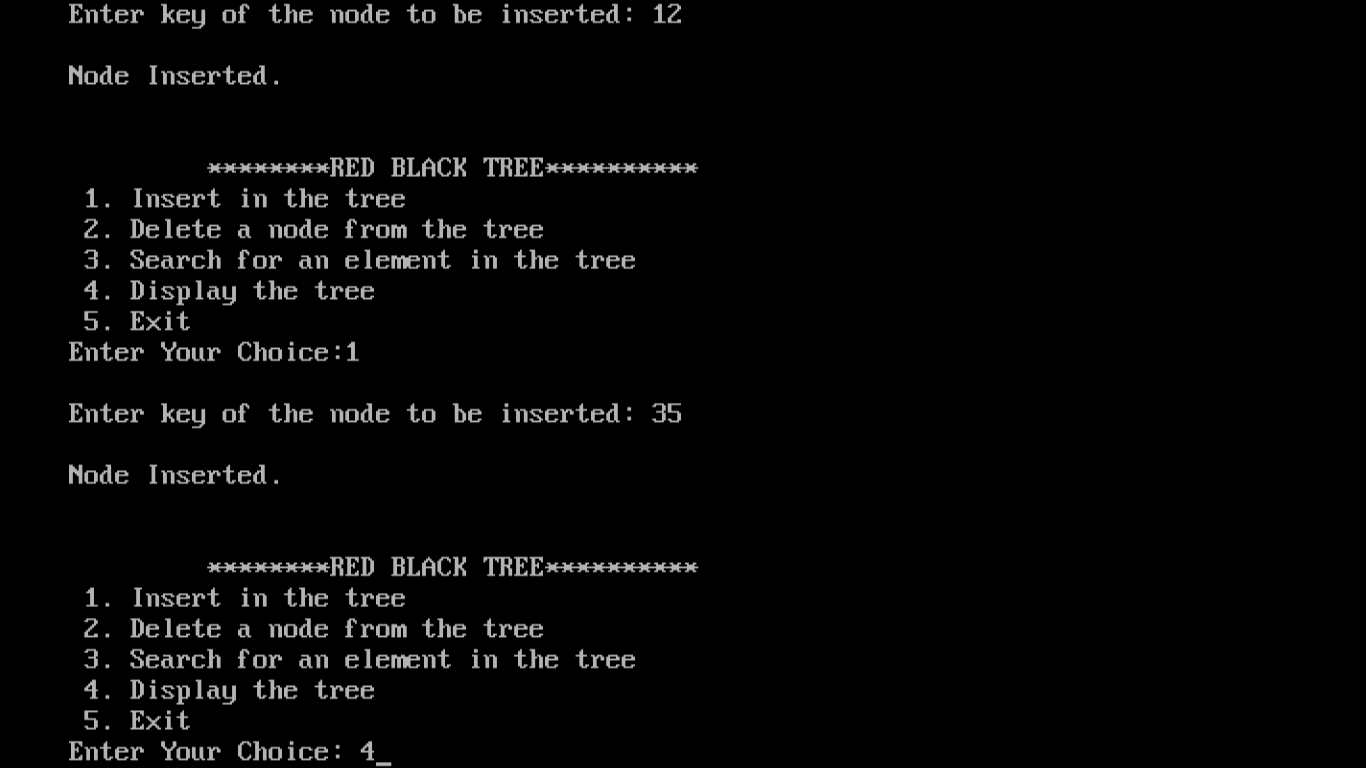
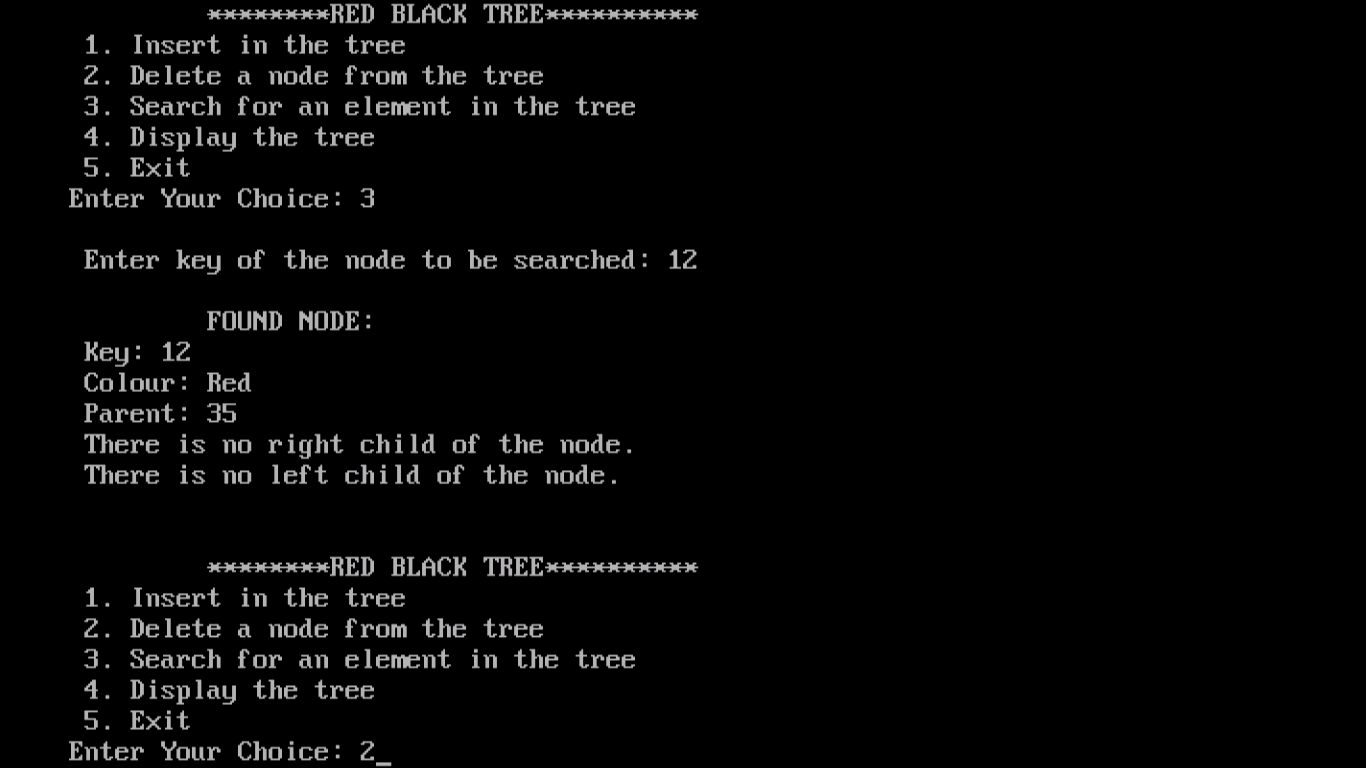
}while(y!=1);

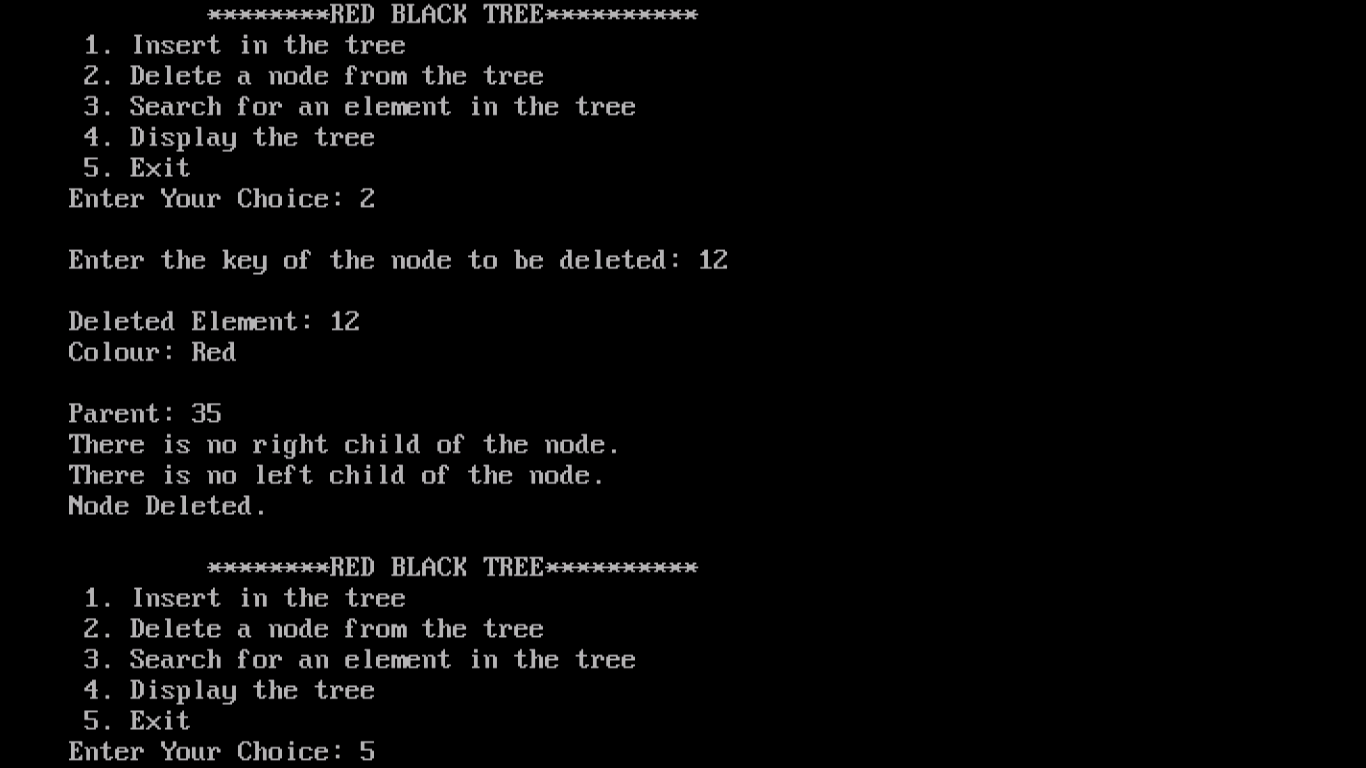
getch();

}

**Output**



**6. Write a program to determine the LCS of two given sequences**

**Source code**

#include<iostream.h>

#include<conio.h>

int max(int a,int b)

{

if(a>b)

return a;

else

return b;

}

void main()

{

clrscr();

int m,n;

char X[20],Y[20];

int lcs[21][21];

cout<<"Enter the number of elements of sequence X"<<endl;

cin>>m;

cout<<"Enter the elements of sequence X"<<endl;

for(int i=0;i<m;i++)

cin>>X[i];

cout<<"Enter the number of elements of sequence Y"<<endl;

cin>>n;

cout<<"Enter the elements of sequence Y"<<endl;

for(i=0;i<n;i++)

cin>>Y[i];

char l[100];

for(i=0;i<=m;i++)

{

for(int j=0;j<=n;j++)

{

if(i==0 || j==0)

lcs[i][j]=0;

if(X[i-1]==Y[j-1])

{

lcs[i][j]=lcs[i-1][j-1]+1;

}

else

lcs[i][j]=max(lcs[i-1][j],lcs[i][j-1]);

}

}

int index=lcs[m][n];

i=m;

int j=n;

l[index]='\0';

while(i>0 && j>0)

{

if(X[i-1]==Y[j-1])

{

l[index-1]=X[i-1];

i--;

j--;

index--;

}

else if(lcs[i-1][j]>lcs[i][j-1])

i--;

else

j--;

}

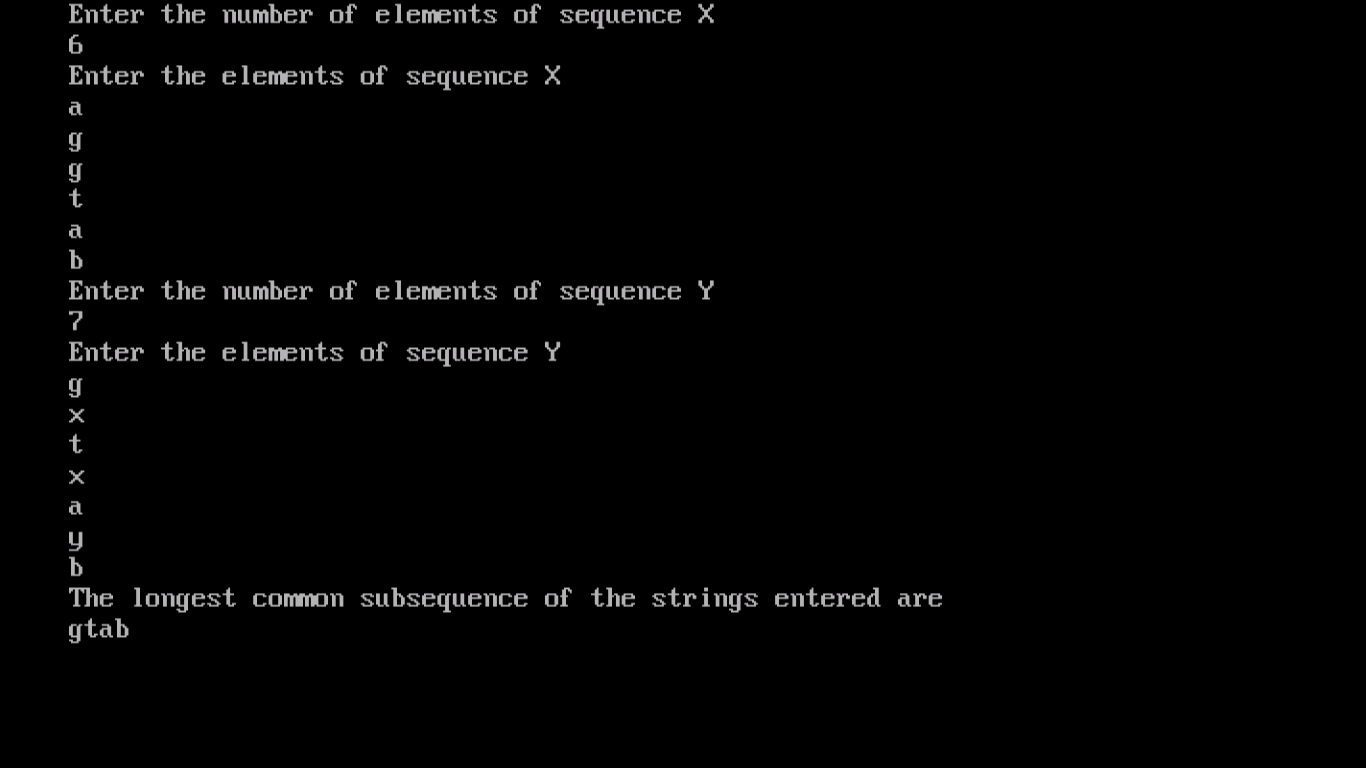
cout<<"The longest common subsequence of the strings entered are"<<endl;

cout<<l;

getch();

}

**Output**



**7. Implement Breadth-First Search in a graph**

**Source code**

#include<iostream.h>

#include<conio.h>

#define max 20;

typedef struct q

{

int f,r;

int data[20];

}q1;

typedef struct node

{

struct node\* next;

int vertex;

}node;

node\* G[20];

int visited[20];

void insert(int vi,int vj)

{

node \*p,\*q;

q=new node;

q->vertex=vj;

q->next=NULL;

if(G[vi]==NULL)

G[vi]=q;

else

{

p=G[vi];

while(p->next!=NULL)

p=p->next;

p->next=q;

}

}

int readgraph()

{

int n;

int i,vi,vj,edges;

cout<<"Enter the number of vertices"<<endl;

cin>>n;

cout<<"Enter the number of edges"<<endl;

cin>>edges;

for(i=0;i<edges;i++)

{

cout<<"Enter an edge ( u, v ) by entering values of just u and v "<<endl;

cin>>vi;

cin>>vj;

insert(vi,vj);

insert(vj,vi);

}

return n;

}

void BFS(int i,int n);

void main()

{

clrscr();

int i;

int n=readgraph();

cout<<"Enter starting node"<<endl;

cin>>i;

cout<<"The BFS traversal is"<<endl;

BFS(i,n);

getch();

}

int full(q \*p)

{

if(p->r==19)

return 1;

return 0;

}

int empty(q \*p)

{

if(p->r==-1)

return 1;

return 0;

}

void enqueue(q \*p,int x)

{

if(p->r==-1)

{

p->r=p->f=0;

p->data[p->f]=x;

}

else

{

p->r=(p->r)+1;

p->data[p->r]=x;

}

}

int dequeue(q \*p)

{

int x;

x=p->data[p->f];

if(p->r==p->f)

{

p->r=-1;

p->f=-1;

}

else

p->f=(p->f)+1;

return x;

}

void BFS(int v,int n)

{

int w,i,visited[20];

q q1;

node \*p;

q1.r=q1.f=-1;

for(i=0;i<=n;i++)

{

visited[i]=0;

}

enqueue(&q1,v);

visited[v]=1;

while(!empty(&q1))

{

v=dequeue(&q1);

cout<<v;

for(p=G[v];p!=NULL;p=p->next)

{

w=p->vertex;

if(visited[w]==0)

{

enqueue(&q1,w);

visited[w]=1;

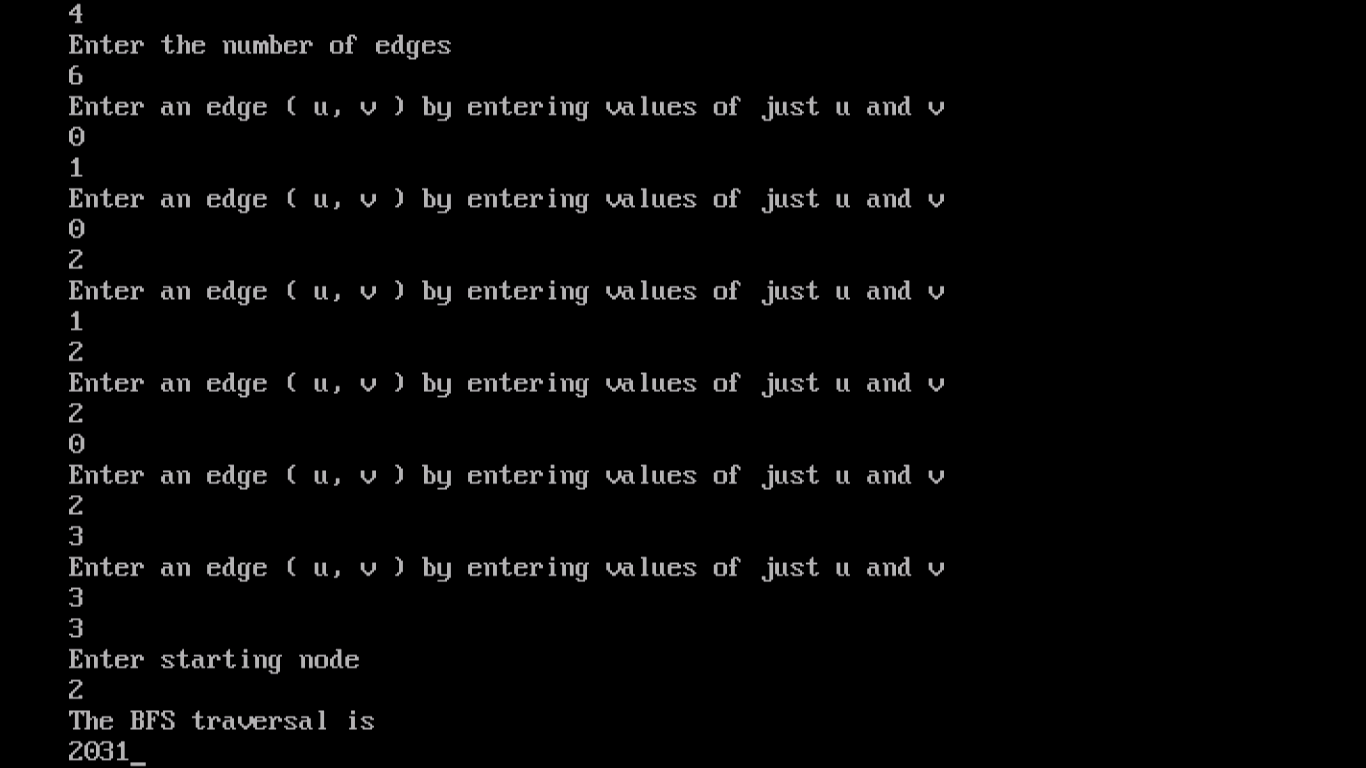
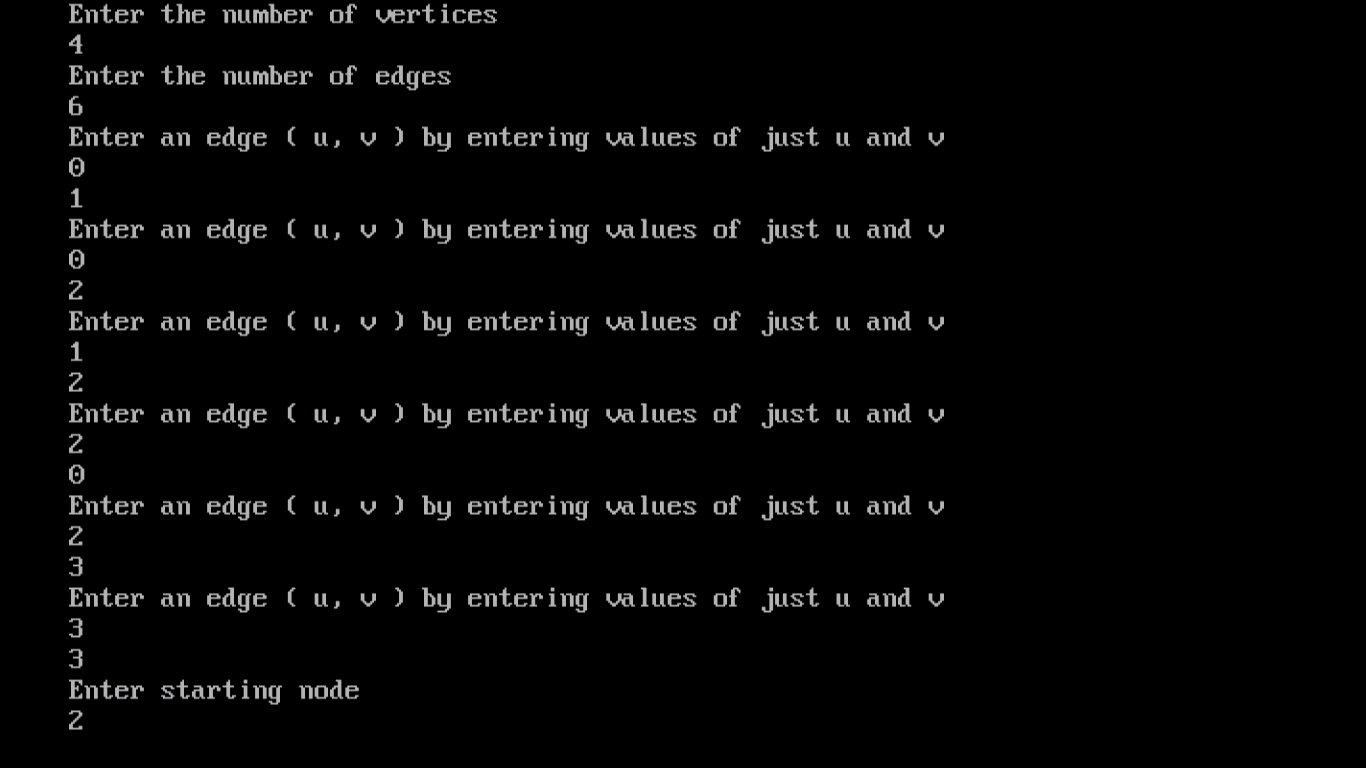
}

}

}

}

**Output**



**8. Implement Depth-First Search in a graph**

**Source code**

#include<iostream.h>

#include<conio.h>

typedef struct node

{

struct node\* next;

int vertex;

}node;

node\* G[20];

int visited[20];

void insert(int vi,int vj)

{

node \*p,\*q;

q=new node;

q->vertex=vj;

q->next=NULL;

if(G[vi]==NULL)

G[vi]=q;

else

{

p=G[vi];

while(p->next!=NULL)

p=p->next;

p->next=q;

}

}

void readgraph()

{

int n;

int i,vi,vj,edges;

cout<<"Enter the number of vertices"<<endl;

cin>>n;

cout<<"Enter the number of edges"<<endl;

cin>>edges;

for(i=0;i<edges;i++)

{

cout<<"Enter an edge ( u, v ) by entering values of just u

and v "<<endl;

cin>>vi;

cin>>vj;

insert(vi,vj);

insert(vj,vi);

}

}

void DFS(int i)

{

node \*p;

cout<<i<<endl;

p=G[i];

visited[i]=1;

while(p!=NULL)

{

i=p->vertex;

if(!visited[i])

DFS(i);

p=p->next;

}

}

void main()

{

clrscr();

int i;

readgraph();

cout<<"Enter starting node"<<endl;

cin>>i;

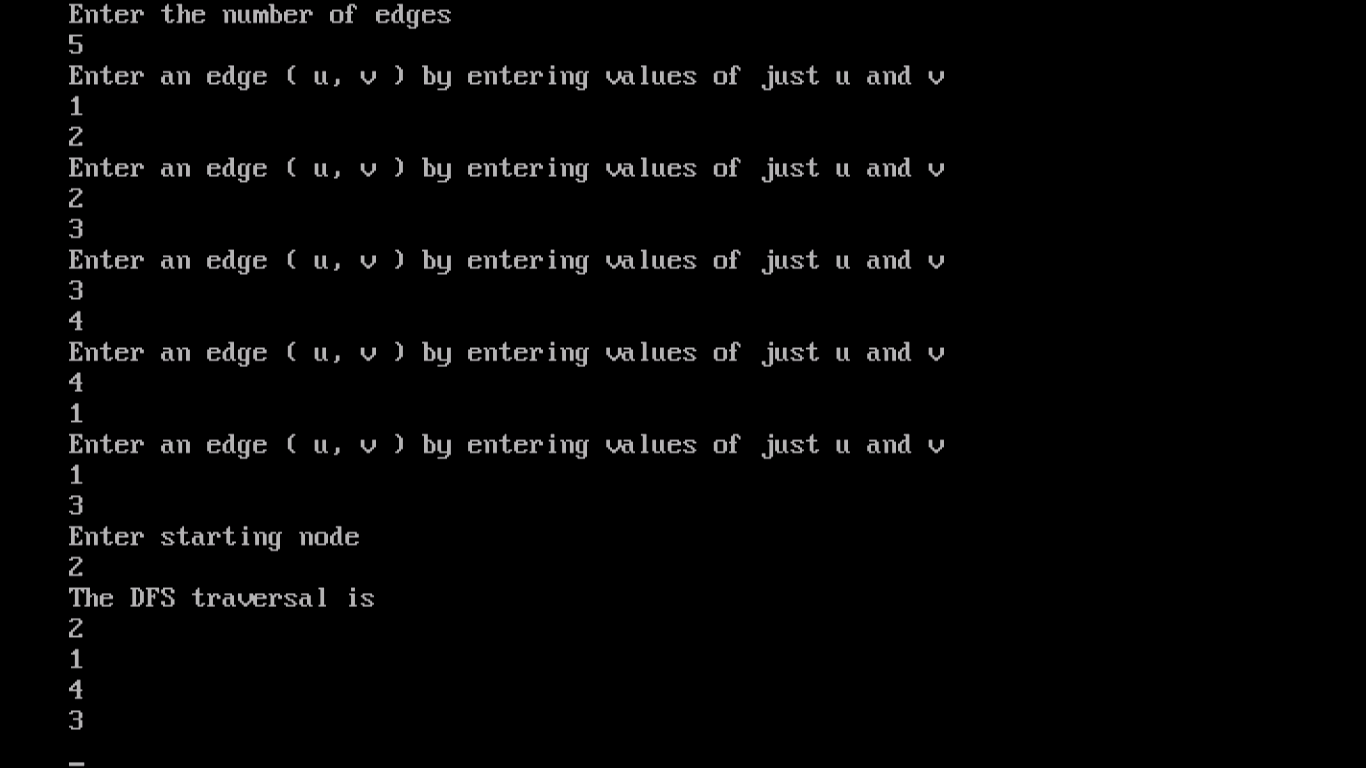
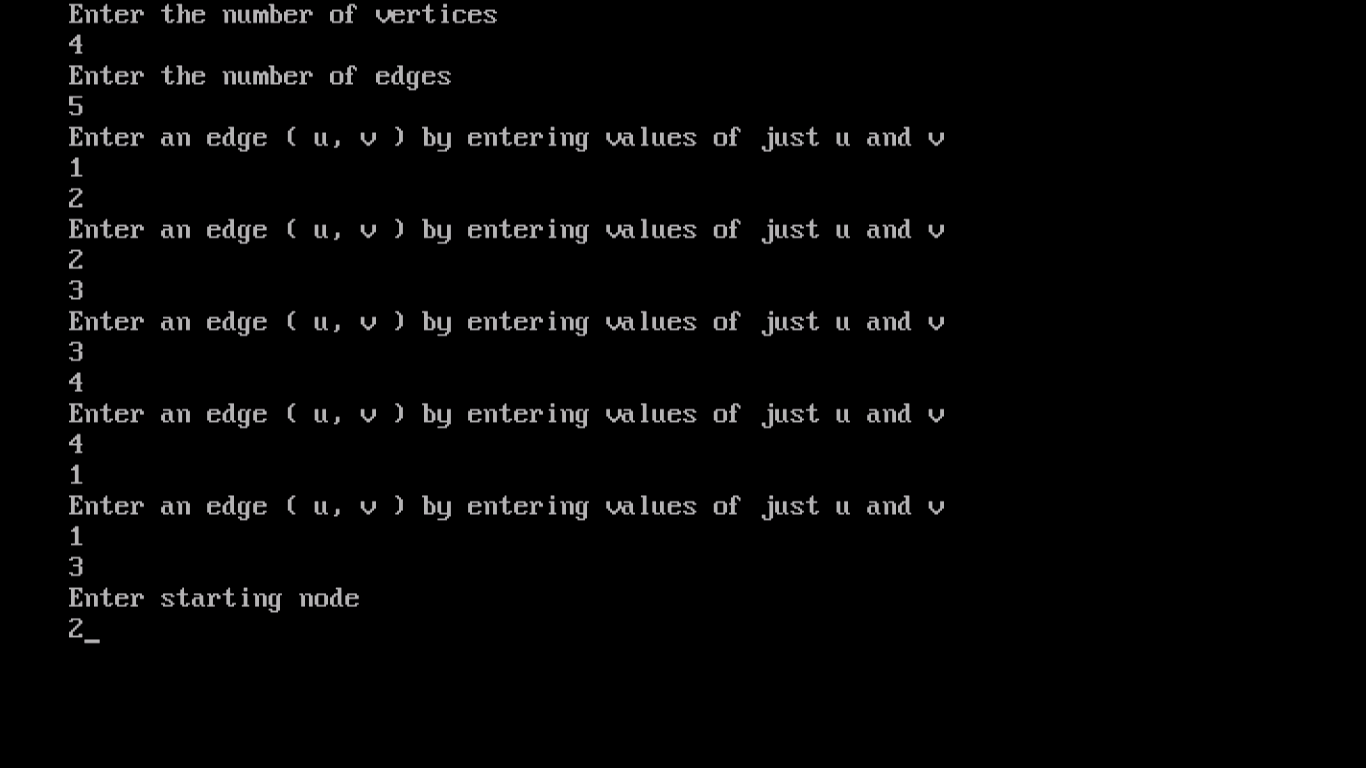
cout<<"The DFS traversal is"<<endl;

DFS(i);

getch();

}

**Output:**



**9. Write a program to determine the minimum spanning tree of a graph.**

**Source code**

#include<iostream.h>

#include<stdio.h>

#include<conio.h>

#include<stdlib.h>

int i,j,k,a,b,u,v,n,ne=1;

int min,mincost=0,cost[9][9],parent[9];

int find(int);

int uni(int,int);

void main()

{

clrscr();

cout<<"\nImplementation of Kruskal's algorithm";

cout<<"\nEnter the no. of vertices: ";

cin>>n;

cout<<"\nEnter the cost adjacency matrix: ";

for(i=1;i<=n;i++)

{

for(j=1;j<=n;j++)

{

cin>>cost[i][j];

if(cost[i][j]==0)

cost[i][j]=999;

}

}

cout<<"\nThe edges of Minimum Cost Spanning Tree are: ";

while(ne<n)

{

for(i=1,min=999;i<=n;i++)

{

for(j=1;j<=n;j++)

{

if(cost[i][j]<min)

{

min=cost[i][j];

a=u=i;

b=v=j;

}

}

}

u=find(u);

v=find(v);

if(uni(u,v))

{

cout<<"\n "<<ne++<<" edge ("<<a<<","<<b<<") ="<<min;

mincost +=min;

}

cost[a][b]=cost[b][a]=999;

}

cout<<"\nMinimum cost = "<<mincost;

getch();

}

int find(int i)

{

while(parent[i])

i=parent[i];

return i;

}

int uni(int i,int j)

{

if(i!=j)

{

parent[j]=i;

return 1;

}

return 0;

}

**Output**

