//TBT

#include<iostream>

using namespace std;

class TreeNode

{

public:

TreeNode \*lchild, \*rchild;

int data;

bool lbit, rbit;

TreeNode()

{

lchild=rchild=NULL;

}

};

class TBT

{

TreeNode\* Root;

public:

TreeNode\* Insert(TreeNode\*, int);

TreeNode\* Successor(TreeNode\*);

void Inorder(TreeNode\*);

TBT()

{

Root=NULL;

}

};

TreeNode\* TBT :: Insert(TreeNode\* Root, int Key)

{

TreeNode\* ptr = Root;

TreeNode\*par=NULL;

while(ptr!=NULL)

{

if(ptr->data==Key)

{

return Root;

}

par=ptr;

if (Key < ptr->data)

{

if (ptr -> lbit == false)

ptr = ptr -> lchild;

else

break;

}

else

{

if (ptr->rbit == false)

ptr = ptr -> rchild;

else

break;

}

}

TreeNode \*Temp = new TreeNode;

Temp -> data = Key;

Temp -> lbit = true;

Temp -> rbit = true;

if (par == NULL)

{

Root = Temp;

Temp -> lchild = NULL;

Temp -> rchild = NULL;

}

else if (Key < (par -> data))

{

Temp -> lchild = par -> lchild;

Temp -> rchild = par;

par -> lbit = false;

par -> lchild = Temp;

}

else

{

Temp -> lchild = par;

Temp -> rchild = par -> rchild;

par -> rbit = false;

par -> rchild = Temp;

}

return Root;

}

TreeNode\* TBT :: Successor(TreeNode\* ptr)

{

if(ptr->rbit == true)

return ptr->rchild;

ptr= ptr->rchild;

while(ptr->lbit == false)

ptr = ptr->lchild;

return ptr;

}

void TBT :: Inorder(TreeNode\* Root)

{

if (Root == NULL)

cout<<"Tree is empty";

TreeNode \*ptr = Root;

while (ptr -> lbit == false)

ptr = ptr -> lchild;

//print successors one by one

while (ptr != NULL)

{

cout<<" "<<ptr -> data;

ptr =Successor(ptr);

}

}

int main()

{

TBT a;

TreeNode\* Root = NULL;

Root = a.Insert(Root, 90);

Root = a.Insert(Root, 56);

Root = a.Insert(Root, 600);

Root = a.Insert(Root, 9);

Root = a.Insert(Root, 10);

a.Inorder(Root);

return 0;

}

//AVL Tree

#include<iostream>

#include<algorithm>

using namespace std;

class avl {

public:

int d;

struct avl \*l;

struct avl \*r;

}\*root;

class avl\_tree {

public:

int height(avl \*);

int difference(avl \*);

avl \*rr\_rotat(avl \*);

avl \*ll\_rotat(avl \*);

avl \*lr\_rotat(avl \*);

avl \*rl\_rotat(avl \*);

avl \* balance(avl \*);

avl \* insert(avl\*, int);

void inorder(avl \*);

void preorder(avl \*);

void search(int);

avl\_tree() {

root = NULL;

}

};

int avl\_tree::height(avl \*t) {

int h = 0;

if (t != NULL) {

int l\_height = height(t->l);

int r\_height = height(t->r);

int max\_height = max(l\_height, r\_height);

h = max\_height + 1;

}

return h;

}

int avl\_tree::difference(avl \*t) {

int l\_height = height(t->l);

int r\_height = height(t->r);

int b\_factor = l\_height - r\_height;

return b\_factor;

}

avl \*avl\_tree::rr\_rotat(avl \*parent) {

avl \*t;

t = parent->r;

parent->r = t->l;

t->l = parent;

cout<<"\nRight-Right Rotation\n";

return t;

}

avl \*avl\_tree::ll\_rotat(avl \*parent) {

avl \*t;

t = parent->l;

parent->l = t->r;

t->r = parent;

cout<<"\nLeft-Left Rotation\n";

return t;

}

avl \*avl\_tree::lr\_rotat(avl \*parent) {

avl \*t;

t = parent->l;

parent->l = rr\_rotat(t);

cout<<"\nLeft-Right Rotation\n";

return ll\_rotat(parent);

}

avl \*avl\_tree::rl\_rotat(avl \*parent) {

avl \*t;

t = parent->r;

parent->r = ll\_rotat(t);

cout<<"\nRight-Left Rotation\n";

return rr\_rotat(parent);

}

avl \*avl\_tree::balance(avl \*t) {

int bal\_factor = difference(t);

if (bal\_factor > 1) {

if (difference(t->l) > 0)

t = ll\_rotat(t);

else

t = lr\_rotat(t);

} else if (bal\_factor < -1) {

if (difference(t->r) > 0)

t = rl\_rotat(t);

else

t = rr\_rotat(t);

}

return t;

}

avl \*avl\_tree::insert(avl \*r, int v) {

if (r == NULL) {

r = new avl;

r->d = v;

r->l = NULL;

r->r = NULL;

return r;

} else if (v< r->d) {

r->l = insert(r->l, v);

r = balance(r);

} else if (v >= r->d) {

r->r = insert(r->r, v);

r = balance(r);

} return r;

}

void avl\_tree::inorder(avl \*t) {

if (t == NULL)

return;

inorder(t->l);

cout << t->d << " ";

inorder(t->r);

}

void avl\_tree::preorder(avl \*t) {

if (t == NULL)

return;

cout << t->d << " ";

preorder(t->l);

preorder(t->r);

}

void avl\_tree:: search(int val)

{

int flag=0;

avl \*temp;

temp=root;

while(temp!=NULL)

{

if(temp->d == val)

{

flag=1;

break;

}

else if(temp->d < val)

{

temp=temp->r;

}

else if(temp->d > val)

{

temp=temp->l;

}

}

if(flag==1)

cout<<"Key found!!"<<endl;

else

cout<<"Key not found!!"<<endl;

}

int main() {

int c, z;

avl\_tree avl;

while (1) {

cout << "1.Insert Element into the tree" << endl;

cout<< "2.Search element "<<endl;

cout << "3.InOrder traversal" << endl;

cout << "4.PreOrder traversal" << endl;

cout << "6.Exit" << endl;

cout << "Enter your Choice: ";

cin >> c;

switch (c) {

case 1:

int k;

cout<<"how many values to insert? ";

cin>>k;

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

cout << "Enter value to be inserted: ";

cin >> z;

root = avl.insert(root, z);

}

break;

case 2:

cout<<"enter value to search: ";

int m;

cin>>m;

avl.search(m);

break;

case 3:

cout << "Inorder Traversal:" << endl;

avl.inorder(root);

cout << endl;

break;

case 4:

cout << "Preorder Traversal:" << endl;

avl.preorder(root);

cout << endl;

break;

case 6:

return 0;

break;

default:

cout << "Wrong Choice" << endl;

}

}

return 0;

}

//DFS BFS

#include<iostream>

#include<queue>

using namespace std;

#define max 5

class node

{

public:

int vertex;

node \*next;

node()

{

vertex=0;

next=NULL;

}

};

class graph

{

node \*heads[max];

int n, visited[max], visited2[max];

public:

graph()

{

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

{

heads[i]=NULL;

visited[i]=0;

visited2[i]=0;

}

}

void create();

void BFS(int);

void DFS(int);

};

void graph::create()

{

node \*curr, \*prev;

int n1,i,j,vertex;

bool done=false;

cout<<"\nEnter number of vertices : ";

cin>>n;

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

{

if(!(heads[i]=new node))

{

cout<<"\nMemory not allocated!\n";

exit;

}

heads[i]->vertex=i+1;

cout<<"\nEnter number of vertices connected to "<<i+1<<" : ";

cin>>n1;

prev=heads[i];

for(int j=0;j<n1;j++)

{

if(!(curr=new node))

{

cout<<"\nMemory not allocated!\n";

exit;

}

done=false;

do

{

cout<<"\nEnter vertex number of connected vertex : ";

cin>>vertex;

if(vertex > n && vertex < 1)

{

cout<<"\nVertex out of range!";

}

else

{

curr->vertex=vertex;

prev->next=curr;

prev=curr;

done=true;

}

}

while(!done);

}

if(n1==0)

prev->next=NULL;

}

return;

}

void graph::DFS(int v)

{

node \*curr;

int w;

curr=heads[v];

cout<<"\t"<<curr->vertex;

visited2[v]=true;

curr=curr->next;

while(curr!=NULL)

{

if(!visited2[w=(curr->vertex - 1)])

DFS(w);

curr=curr->next;

}

return;

}

void graph::BFS(int v)

{

queue<node\*>q;

node \*curr;

visited[v]=true;

cout<<"\t"<<heads[v]->vertex;

q.push(heads[v]);

while(!q.empty())

{

curr=q.front();

q.pop();

curr=curr->next;

while(curr!=NULL)

{

if(!visited[curr->vertex -1])

{

q.push(heads[curr->vertex -1]);

cout<<"\t"<<curr->vertex;

visited[curr->vertex -1]=true;

}

curr=curr->next;

}

}

return;

}

int main()

{

graph obj;

int choice;

while(1)

{

cout<<"1.Create\n2.Display DFS\n3.Display BFS\n0.Exit";

cout<<"\nEnter choice : ";

cin>>choice;

switch(choice)

{

case 1:

obj.create();

cout<<"\n";

break;

case 2:

obj.DFS(1);

cout<<"\n";

break;

case 3:

obj.BFS(1);

cout<<"\n";

break;

case 0:

return 0;

break;

default:

cout<<"Enter valid choice!\n";

break;

}

}

return 0;

}

//Prims

#include <iostream>

#include <conio.h>

using namespace std;

struct node

{

int fr, to, cost;

}p[6];

int c = 0, temp1 = 0, temp = 0;

void prims(int \*a, int b[][7], int i, int j)

{

a[i] = 1;

while (c < 6)

{

int min = 999;

for (int i = 0; i < 7; i++)

{

if (a[i] == 1)

{

for (int j = 0; j < 7; )

{

if (b[i][j] >= min || b[i][j] == 0)

{

j++;

}

else if (b[i][j] < min)

{

min = b[i][j];

temp = i;

temp1 = j;

}

}

}

}

a[temp1] = 1;

p[c].fr = temp;

p[c].to = temp1;

p[c].cost = min;

c++;

b[temp][temp1] = b[temp1][temp]=1000;

}

for (int k = 0; k < 6; k++)

{

cout<<"source node:"<<p[k].fr<<endl;

cout<<"destination node:"<<p[k].to<<endl;

cout<<"weight of node"<<p[k].cost<<endl;

}

}

int main()

{

int a[7];

for (int i = 0; i < 7; i++)

{

a[i] = 0;

}

int b[7][7];

for (int i = 0; i < 7; i++)

{

cout<<"enter values for "<<(i+1)<<" row"<<endl;

for (int j = 0; j < 7; j++)

{

cin>>b[i][j];

}

}

prims(a, b, 0, 0);

getch();

}

//Merge Sort

#include<iostream>  
using namespace std;  
  
class Employee  
{  
public:  
string name;  
int id;  
int salary;  
  
friend void merge\_sort(Employee \*, int, int);  
friend void merge(Employee \*, int, int, int);  
};  
  
const int maxsize = 10;  
Employee sorted[maxsize];  
  
void merge\_sort(Employee \*emp, int lowb, int uppb)  
{  
int mid;  
  
if(lowb<uppb)  
{  
mid = (lowb+uppb)/2;  
merge\_sort(emp, lowb, mid);  
merge\_sort(emp, mid+1, uppb);  
merge(emp, lowb, mid, uppb);  
}  
}  
  
void merge(Employee \*emp, int lowb, int mid, int uppb)  
{  
int i = lowb, j = mid+1, k = lowb;  
  
while(i<=mid && j<=uppb)  
{  
if(emp[i].salary >= emp[j].salary)  
{  
sorted[k] = emp[i];  
++k;  
++i;  
}  
else  
{  
sorted[k] =  emp[j];  
++k;  
++j;  
}  
}  
if(i>mid)  
{  
while(j<=uppb)  
{  
sorted[k] = emp[j];  
++k;  
++j;  
}  
}  
  
else if(j>uppb)  
{  
while(i<=mid)  
{  
sorted[k] = emp[i];  
++k;  
++i;  
}  
  
}  
  
for(int i = lowb; i<=uppb; ++i)  
emp[i] = sorted[i];  
}  
  
int main()  
{  
int size;  
cout<<"\nEnter the number of entries:";  
cin>>size;  
  
Employee emp[size];  
  
for(int i=0; i<size; i++)  
{  
cout<<"\nEnter the ID:";  
cin>>emp[i].id;  
cout<<"\nEnter Name of Employee:";  
cin>>emp[i].name;  
cout<<"\nEnter Salary of Employee:";  
cin>>emp[i].salary;  
}  
  
int lowb = 0, uppb = size-1;  
merge\_sort(emp, lowb, uppb);  
cout<<"\nThe top 5 employees with Highest Salaries are:";  
  
for(int i=0; i<5; i++)  
{  
cout<<"\nID:"<<sorted[i].id;  
cout<<"\nName:"<<sorted[i].name;  
cout<<"\nSalary:"<<sorted[i].salary;  
cout<<endl;  
}  
  
return 0;  
  
}

//BST

#include<iostream>  
using namespace std;  
  
class BSTnode  
{  
    public:  
    int data;  
    BSTnode \*rchild, \*lchild;  
};  
  
class BST  
{  
    public:  
     
    BSTnode \*root, \*temp;  
    BST()  
    {  
        root = NULL;  
    }  
    void Insert(int key);  
    void Display();  
    void Delete(int key);  
    void Search(int key);  
    //bool IsEmpty();  
};  
  
class stack\_node  
{  
    public:  
    BSTnode \* data;  
    stack\_node \*next;  
};  
  
class Stack  
{  
    public:  
    stack\_node \*top;  
    int Is\_Empty();  
    BSTnode\* Pop();  
    void Push(BSTnode \* temp);  
  
  
    Stack()  
    {  
        top=NULL;  
  
    }  
};  
  
int Stack :: Is\_Empty()  
{  
if(top==NULL)  
return 1;  
else  
return 0;  
}  
  
void Stack :: Push(BSTnode \* temp)  
{  
    stack\_node \*newnode;  
    newnode = new stack\_node;  
    newnode->data=temp;  
    newnode->next=top;  
    top=newnode;  
}  
  
BSTnode\* Stack :: Pop()  
{  
    stack\_node \*temp=top;  
    BSTnode \* value;  
    value=top->data;  
  
    if(Is\_Empty())  
    return 0;  
  
    else  
    {  
    top=top->next;  
    delete temp;  
    return (value);  
    }  
}  
  
//stack2 for postorder  
class stack2\_node  
{  
    public:  
    char data;  
    stack2\_node \*next;  
};  
  
class Stack2  
{  
    public:  
    stack2\_node \*top;  
    int Is\_Empty();  
    char Pop();  
    void Push(char temp);  
  
  
    Stack2()  
    {  
        top=NULL;  
  
    }  
};  
  
int Stack2 :: Is\_Empty()  
{  
if(top==NULL)  
return 1;  
else  
return 0;  
}  
  
void Stack2 :: Push(char temp)  
{  
    stack2\_node \*newnode;  
    newnode = new stack2\_node;  
    newnode->data=temp;  
    newnode->next=top;  
    top=newnode;  
}  
  
char Stack2 :: Pop()  
{  
    stack2\_node \*temp=top;  
    char value;  
    value=top->data;  
  
    if(Is\_Empty())  
    return 0;  
  
    else  
    {  
    top=top->next;  
    delete temp;  
    return (value);  
    }  
}  
  
void BST :: Insert(int key)  
{  
    BSTnode \*newnode;  
    newnode = new BSTnode ;  
    newnode->data=key;  
    newnode->rchild=NULL;  
    newnode->lchild=NULL;  
     
    if(root==NULL)  
    {  
        root=newnode;    
         
    }  
     
    else  
    {  
        temp=root;  
        while(temp != NULL)  
        {  
            if(key < temp->data )  
            {  
                if(temp->lchild == NULL)  
                {  
                    temp->lchild = newnode;  
                    break;  
                }  
                temp= temp->lchild;  
            }  
            else  
            {  
                if(temp->rchild == NULL)  
                {  
                    temp->rchild = newnode;  
                    break;  
                }  
             
                temp = temp->rchild;  
            }  
         
    }  
}  
}  
  
void BST :: Display()  
{  
    //inorder  
    cout<<"\nInorder: ";  
    Stack S1;  
    temp = root;  
    while(1)  
    {  
        while(temp != NULL)  
        {  
            S1.Push(temp);  
            temp = temp->lchild;  
        }  
        if(S1.Is\_Empty())  
        break;  
  
        temp = S1.Pop();  
        cout << temp->data<<" ";  
        temp = temp->rchild;  
    }  
     
    //preorder  
    cout<<"\nPreorder: ";  
    Stack S2;  
    temp = root;  
    while(1)  
    {  
        while(temp != NULL)  
        {  
            S2.Push(temp);  
            cout << temp->data<<" ";  
            temp = temp->lchild;  
        }  
        if(S2.Is\_Empty())  
        break;  
  
        temp = S2.Pop();      
        temp = temp->rchild;  
    }  
     
    //postorder  
    cout<<"\nPostorder: ";  
    Stack S3;  
    Stack2 S;  
    temp=root;  
    char flag;  
    // stack S3 stores the node and S stores the flag ‘L’ or ‘R’  
    while(1)  
    {  
    while(temp != NULL)  
    {  
        S3.Push(temp);  
        S.Push('L');  
        temp = temp->lchild;  
    }  
    if(S3.Is\_Empty())  
        break;  
    else  
    {  
        temp = S3.Pop();  
        flag = S.Pop();  
        if(flag == 'R')  
        {  
            cout << temp->data<<" ";  
            temp = NULL;  
        }  
        else  
        {  
            S3.Push( temp);  
            S.Push('R');  
            temp = temp->rchild;  
        }  
    }  
    }  
}  
     
  
void BST :: Delete(int key)  
{  
    int flag=0, count=0;  
    temp=root;  
    BSTnode \*parent, \*curr;  
    if(root == NULL)  
    {  
        cout<<"\n BST is empty";  
    }  
    else  
    {  
        parent=temp;  
        while(temp)  
        {  
            if(temp->data==key)  
                break;  
                 
            else if(key<temp->data)  
            {  
                parent=temp;  
                temp=temp->lchild;  
            }  
            else  
            {  
                parent=temp;  
                temp=temp->rchild;    
            }  
        }  
        if(temp==NULL)  
            cout<<"\nkey to be deleted not found!";  
        else  
        {  
            //case of leaf node  
            if(temp->rchild == NULL && temp->lchild == NULL)  
            {  
                if(temp==parent->rchild)  
                    parent->rchild=NULL;  
                else if(temp==parent->lchild)  
                    parent->lchild=NULL;  
                     
                delete temp;  
            }  
             
            //case of BST node having right children  
            else if(temp->rchild != NULL && temp->lchild == NULL)  
            {  
                 
                curr = temp->rchild;  
                 
                if(temp==parent->rchild)  
                {  
                    parent->rchild=curr;  
                }  
                else if(temp==parent->lchild)  
                {  
                    parent->lchild=curr;  
                }  
                 
                delete temp;  
            }  
             
            //case of BST node having left children  
            else if(temp->lchild!=NULL && temp->rchild == NULL)  
            {  
                curr = temp->lchild;      
                if(temp==parent->rchild)  
                {  
                    parent->rchild=curr;  
                }  
                else if(temp==parent->lchild)  
                {  
                    parent->lchild=curr;  
                }  
                 
                delete temp;  
            }  
    }  
     
}  
}  
  
void BST :: Search(int key)  
{  
    temp=root;  
    int flag=0;  
    while(temp)  
    {  
        if(temp->data==key)  
        {  
            cout<<"\nKey found";  
            flag=1;  
            break;  
        }  
         
        else if(key<temp->data)  
            temp=temp->lchild;  
        else  
            temp=temp->rchild;          
    }  
    if(flag==0)  
        cout<<"\nkey not found!";  
     
}  
  
int main()  
{  
    int ch,key;  
    BST obj;  
     
    do  
    {  
        cout<<"\nMenu:\n1.insert\n2.delete\n3.display\n4.search\n5.exit\n";  
        cin>>ch;  
        switch(ch)  
        {  
            case 1:  
                    cout<<"\nEnter key to be inserted: ";  
                    cin>>key;  
                    obj.Insert(key);  
                    break;  
                     
            case 2:  
                    cout<<"\nEnter key to be deleted: ";  
                    cin>>key;  
                    obj.Delete(key);  
                    obj.Display();  
                    break;  
                     
            case 3:  
                    obj.Display();  
                    break;  
                     
            case 4:  
                    cout<<"\nEnter key to be searched: ";  
                    cin>>key;  
                    obj.Search(key);  
                    break;  
        }  
                 
         
    }while(ch!=5);  
     
  
    return 0;  
}

//Quick Sort

#include<iostream>

using namespace std;

int partition(int salary[], int lb, int ub)

{

int pivot = salary[lb], count = 0;

for (int i = lb + 1; i <= ub; i++)

{

if (salary[i] <= pivot)

count++;

}

int pindex = lb + count;

swap(salary[pindex], salary[lb]);

int start = lb, end = ub;

while (start < pindex && end > pindex)

{

while (salary[start] <= pivot)

start++;

while (salary[end] > pivot)

end--;

if (start < pindex && end > pindex)

swap(salary[start++], salary[end--]);

}

return pindex;

}

void qcksort(int salary[], int lb, int ub)

{

if (lb < ub)

{

int loc = partition(salary, lb, ub);

qcksort(salary, lb, loc - 1);

qcksort(salary, loc + 1, ub);

}

else

return;

}

int main()

{

int salary[10], lb, ub, n, i;

cout << "\nEnter number of employees : ";

cin >> n;

cout << "\nEnter the Salaries of Employees :\n";

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

cin >> salary[i];

qcksort(salary, 0, n - 1);

cout<<"\nTop 5 highest salaries are :\n";

for (i = n-1; i >=n-5; i--)

cout << salary[i] << " ";

return 0;

}