**PUNE INSTITUTE OF COMPUTER TECHNOLOGY DHANKAWADI, PUNE**

**Data Structures And Algorithms(DSA)**

**Assignment No. 05**

**Title : Binary Search Tree**

**SE-IT-10**  **ACADEMIC YEAR :- 2020-2021**

**Name :- Diptesh Ravindra Varule Roll No :- 23277**

**Source Code :**

//============================================================================

// Name : dsa\_Assignment5.cpp

// Author : Diptesh Varule

// Version : Updating…..

// Copyright : The Cartel

// Description : Hello World in C++, Ansi-style

//============================================================================

**CPP File:**

#include <iostream>

#include"bstree.h"

using namespace std;

int main() {

bstree t;

bnode\* temp=NULL;

int ch;

do{

cout<<"\*\*MENU\*\*\*"<<endl;

cout<<"1)Rec-Insert\n2)Insert\n3)Inorder traversal\n4)Search\n5)mirror\n6)copy\n7)Height\n8)Delete\n9)exit"<<endl;

cin>>ch;

switch(ch){

case 1:int x1;

cout<<"Enter element to be inserted"<<endl;

cin>>x1;

temp=t.rec\_insert(temp, x1);

break;

case 2:int x2;

cout<<"Enter element to be inserted"<<endl;

cin>>x2;

temp=t.insert(temp, x2);

break;

case 3:t.inorder(temp);

cout<<"\n";

break;

case 4:int x3;

cout<<"Enter element to be searched"<<endl;

cin>>x3;

bnode\* key;

key=t.search(temp, x3);

cout<<key->data<<endl;

break;

case 5:bnode\* mir;

mir=t.mirror(temp);

t.inorder(mir);

break;

case 6:bnode\* cpy;

cpy=t.copy(temp);

cout<<"Copied tree:"<<endl;

t.inorder(cpy);

break;

case 7:cout<<"Height of bst: "<<t.height(temp)<<endl;

break;

case 8:int x4;

cout<<"Enter element to be deleted"<<endl;

cin>>x4;

temp=t.deleteNode(temp, x4);

t.inorder(temp);

break;

case 9:return 0;

default:cout<<"invalid choice"<<endl;

break;

}

}while(true);

return 0;

}

**Header file:**

#ifndef BSTREE\_H\_INCLUDED

#define BSTREE\_H\_INCLUDED

#include<iostream>

using namespace std;

struct bnode{

int data;

bnode\* lchild;

bnode\* rchild;

};

class bstree{

bnode\* root;

//bnode\* p,q;

public:

bstree();

bnode\* search(bnode\*,int);

bnode\* rec\_insert(bnode\* p,int x);

bnode\* insert(bnode\* root,int x);

bnode\* getnode(int);

bnode\* mirror(bnode\*);

void inorder(bnode\*);

bnode\* copy(bnode\*);

int height(bnode\*);

bnode\* deleteNode(bnode\*,int);

bnode\* maxOf(bnode\*);

};

bstree::bstree(){

root=NULL;

}

bnode\* bstree::maxOf(bnode\* temp) {

while(temp->rchild!=NULL)

temp=temp->rchild;

return temp;

}

bnode\* bstree::deleteNode(bnode\* temp,int key) {

if(temp==NULL) return NULL;

if(key<temp->data) //traversing the left child

temp->lchild=deleteNode(temp->lchild,key);

else if(key>temp->data) //traversing the right child

temp->rchild=deleteNode(temp->rchild,key);

else{ //element found

bnode\* reference=temp;

if(temp->lchild==NULL and temp->rchild==NULL){ //no children

temp=NULL;

delete reference;

}

else if(temp->lchild==NULL) { //only right child

temp=temp->rchild;

delete reference;

}

else if(temp->rchild==NULL) { //only left child

temp=temp->lchild;

delete reference;

}

else { //both child are present

bnode\* max=maxOf(temp->lchild); //getting the max value left subTree of target tree

int maxVal=max->data; //storing the max Val

temp=deleteNode(temp,max->data); //deleting the node with max value

temp->data=maxVal; //assigning the max value to the current node

delete max; //deleting the copy of max node

}

}

return temp; //returning the updated node

}

void bstree::inorder(bnode\* temp){

if(temp!=NULL){

inorder(temp->lchild);

cout<<temp->data;

inorder(temp->rchild);

}

}

bnode\* bstree::getnode(int x){

bnode \*node=new bnode;

node->data=x;

node->lchild=NULL;

node->rchild=NULL;

return node;

}

bnode\* bstree::rec\_insert(bnode\* temp,int x){

if(temp==NULL){

temp=getnode(x);

return temp;

}

else if(x<temp->data)

temp->lchild=rec\_insert(temp->lchild,x);

else if(x>temp->data)

temp->rchild=rec\_insert(temp->rchild,x);

else if(x==temp->data){

cout<<"duplicate not allowed"<<endl;

return temp;

}

return temp;

}

bnode\* bstree::insert(bnode\* root,int x){

if(root==NULL){

root=getnode(x);

return(root);

}

bnode\* p;

bnode\* q;

p=root;

q=root;

while(q!=NULL and x!=p->data){

p=q;

if(x<p->data)

q=p->lchild;

else if(x>p->data)

q=p->rchild;

}

if(x==(p->data)){

cout<<"Duplicate not allowed"<<endl;

return root;

}

if(x<p->data)

{

p->lchild=getnode(x);

}

else{

p->rchild=getnode(x);

}

return root;

}

bnode\* bstree::search(bnode\* root,int key){

if(root==NULL){

cout<<"Empty tree"<<endl;

}

else

{ bnode\* p=root;

if(key<p->data){

p=search(p->lchild,key);

}

else if(key>p->data)

p=search(p->rchild,key);

else

return p;

}

}

bnode\* bstree::mirror(bnode\* root){

if(root){

bnode\*temp=new(bnode);

temp=root->rchild;

root->rchild=root->lchild;

root->lchild=temp;

mirror(root->lchild);

mirror(root->rchild);

}

return root;

}

bnode\* bstree::copy(bnode\* root){

if(root == NULL)

return NULL;

// create a copy of root node

bnode\* new\_node = getnode(root->data) // Recursively create clone of left and right sub tree

new\_node->lchild = copy(root->lchild);

new\_node->rchild = copy(root->rchild);

return new\_node; // Return root of cloned tree

}

int bstree::height(bnode \*node){

if (node == NULL)

return 0;

else{

int l\_subtree\_depth = height(node->lchild);

int r\_subtree\_depth = height(node->rchild);

if (l\_subtree\_depth > r\_subtree\_depth)

return(l\_subtree\_depth + 1);

else return(r\_subtree\_depth + 1);

}

}

#endif // BSTREE\_H\_INCLUDED

**Output:**

\*MENU\*\*

1)Rec-Insert

2)Insert

3)Inorder traversal

4)Search

5)mirror

6)copy

7)Height

8)Delete

9)exit

1

Enter element to be inserted

1

\*MENU\*\*

1)Rec-Insert

2)Insert

3)Inorder traversal

4)Search

5)mirror

6)copy

7)Height

8)Delete

9)exit

1

Enter element to be inserted

2

\*MENU\*\*

1)Rec-Insert

2)Insert

3)Inorder traversal

4)Search

5)mirror

6)copy

7)Height

8)Delete

9)exit

2

Enter element to be inserted

3

\*MENU\*\*

1)Rec-Insert

2)Insert

3)Inorder traversal

4)Search

5)mirror

6)copy

7)Height

8)Delete

9)exit

2

Enter element to be inserted

4

\*MENU\*\*

1)Rec-Insert

2)Insert

3)Inorder traversal

4)Search

5)mirror

6)copy

7)Height

8)Delete

9)exit

3

1234

\*MENU\*\*

1)Rec-Insert

2)Insert

3)Inorder traversal

4)Search

5)mirror

6)copy

7)Height

8)Delete

9)exit

4

Enter element to be searched

4

4

\*MENU\*\*

1)Rec-Insert

2)Insert

3)Inorder traversal

4)Search

5)mirror

6)copy

7)Height

8)Delete

9)exit

5

4321\*MENU\*\*

1)Rec-Insert

2)Insert

3)Inorder traversal

4)Search

5)mirror

6)copy

7)Height

8)Delete

9)exit

6

Copied tree:

4321\*MENU\*\*

1)Rec-Insert

2)Insert

3)Inorder traversal

4)Search

5)mirror

6)copy

7)Height

8)Delete

9)exit

5

1234\*MENU\*\*

1)Rec-Insert

2)Insert

3)Inorder traversal

4)Search

5)mirror

6)copy

7)Height

8)Delete

9)exit

7

Height of bst: 4

\*MENU\*\*

1)Rec-Insert

2)Insert

3)Inorder traversal

4)Search

5)mirror

6)copy

7)Height

8)Delete

9)exit

8

Enter element to be deleted

3

124\*MENU\*\*

1)Rec-Insert

2)Insert

3)Inorder traversal

4)Search

5)mirror

6)copy

7)Height

8)Delete

9)exit

3

124

\*MENU\*\*

1)Rec-Insert

2)Insert

3)Inorder traversal

4)Search

5)mirror

6)copy

7)Height

8)Delete

9)exit