**Name : Advait Katikar**

**Roll No : 2334**

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\***Assignment No : 5** \*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

**Title : Implement binary search tree and perform following operations:**

**a. Insert**

**b. Delete**

**c. Search**

**d. Mirror image**

**e. Display**

**f. Display level wise**

.

\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*\*

#include<iostream>

#include<stdlib.h>

using namespace std;

struct node //Node declaration

{

int data;

node \*left,\*right,\*next;

}\*root;

class BST //class declaration

{

//Declared variable as public

public:

void find(int, node \*\*, node \*\*); //Function to find the node

int search(node \*, int ); //Function to search the node

void mirror(node \*); //Function to create a copy of that node

node \*getroot();

void level\_BST();

void insert(node \*,node \*); //Function to insert new node

void del(int); // Function to delete given node

void case\_a(node \*,node \*);

void case\_b(node \*,node \*);

void case\_c(node \*,node \*);

void inorder(node \*); //Function to perform inorder traversal

void display(node \*, int); //Function to display the tree

BST()

{

root = NULL; //Initialize root as NULL

}

};

class queue

{

node \*data[30];

int rear,front;

public : //Declare variable as public

queue() //Initialize rear and front variable of queue as -1

{

rear=front=-1;

}

void init() // Initialize rear and front variable of queue as -1

{

rear=front=-1;

}

void insert(node \*); //Recursive function call to insert new node

node \*del(); //Recursive function call to delete the node

int qempty();

};

void queue::insert(node \*p)

{

if(qempty()) //Checking queue is empty or not

rear=front=0;

else

rear=rear+1;

data[rear]=p;

}

node \*queue::del()

{

node \*p=data[front];

if(rear==front) //Compare the rare and front variable of queue

rear=front=-1;

else

front=front+1;

return(p);

}

int queue::qempty()

{

if(rear==-1)

return 1;

return 0;

}

/\* Main Contains Menu \*/

int main()

{

int choice, num,x;

BST bst;

node \*temp;

while (1)

{

cout<<"\n\n\t-----------------";

cout<<"\n\n\tOperations on BST";

cout<<"\n\n\t-----------------";

cout<<"\n\t1.Insert Element\n\t2.Delete Element\n\t3.Search\n\t4.Display of

BST\n\t5.Mirror Image\n\t6.Display level wise\n\t7.Quit\n\n\tEnter your choice : ";

cin>>choice;

switch(choice)

{

case 1:

temp = new node;

cout<<"\n\n\tEnter the number to be inserted : ";

cin>>temp->data;

bst.insert(root, temp);

break;

case 2:

if (root == NULL)

{

cout<<"\n\n\tTree is empty, nothing to delete";

continue;

}

cout<<"\n\n\tEnter the number to be deleted : ";

cin>>num;

bst.del(num);

break;

case 3:

cout<<"\n\nEnter the no to serached:";

cin>>num;

x=bst.search(root,num);

if(x==0)

cout<<"\n\tEelemt Not Found ";

else

cout<<"\n\tElement Found";

break;

case 4:

cout<<"\n\n\tDislpay of BST:";

bst.inorder(root);

break;

case 5:

cout<<"\n\tMirror image :";

bst.mirror(bst.getroot());

bst.insert(root,temp);

break;

case 6:

bst.level\_BST();

break;

case 7:

exit(1);

default:

cout<<"\n\tInvalid choice";

}

}

}

node \*BST::getroot() // Function Get the root node

{

return root;

}

/\* Find Element in the Tree \*/

void BST::find(int item, node \*\*par, node \*\*loc)

{

node \*ptr, \*ptrsave;

if (root == NULL) //Compare root as NULL

{

\*loc = NULL;

\*par = NULL;

return;

}

if (item == root->data) //Compare the item which has to be find with the

element of node

{

\*loc = root;

\*par = NULL;

return;

}

if (item < root->data) /\*Compare weather item which has to be find

with the element of root \*/

ptr = root->left;

else

ptr = root->right;

ptrsave = root; //Move the root node into ptrsave variable

while (ptr != NULL) //Loop will execute till ptr is not NULL

{

if (item == ptr->data) /\*Compare the item which has to be find with

the element of node \*/

{

\*loc = ptr;

\*par = ptrsave;

return;

}

ptrsave = ptr;

if (item < ptr->data) /\*Compare weather item which has to be find

with the element of root \*/

ptr = ptr->left;

else

ptr = ptr->right;

}

\*loc = NULL;

\*par = ptrsave;

}

int BST::search(node \*tree , int n)

{

int Flag=0;

while(tree!=NULL) //Loop will execute till ptr is not NULL

{

if(n > tree->data) /\*Compare weather the element which is to be

find is greater than element of tree not\*/

tree=tree->right;

else if(n < tree->data) /\*Compare weather the element which is to be

find is greater than element of tree not\*/

tree=tree->left;

else

{

Flag=1;

break;

}

}

return Flag;

}

void BST::level\_BST()

{

queue q1,q2;

node \*p1,\*p2;

node \*T=root;

if(T==NULL) //Function to check weather T is NULL or not

return;

q1.insert(T); //Insert node T into Queue

cout<<T->data;

while(!q1.qempty()) //Loop will execute till queue will not empty

{

q2.init(); //Initialize queue

while(!q1.qempty()) //Loop will execute till queue will not empty

{

p1=q1.del(); //Deleted element from queue store into p1

variable

if(p1->left!=NULL) //check weather left child of p1 is NULL or not

{

q2.insert(p1->left); //insert left child of p1 into queue

cout<<" "<<p1->left->data;

}

if(p1->right!=NULL) //check weather right child of p1 is NULL or not

{

q2.insert(p1->right); //insert right child of p1 into queue

cout<<" "<<p1->right->data;

}

}

q1=q2;

}

}

void BST::mirror(node \*root)

{

node \*T=root,\*temp;

if(T!=NULL) //Compare the node T is NULL or not

{

temp = T->left;

T->left = T->right;

T->right = temp;

mirror(T->left); //Recursive function call to mirror the left child

mirror(T->right); //Recursive function call to mirror the right child

cout<<T->data<<"\t";

}

}

/\* Inserting Element into the Tree \*/

void BST::insert(node \*tree, node \*newnode)

{

if (root == NULL) //Check weather root is NULL or not

{

root = new node;

root->data = newnode->data;

root->left = NULL;

root->right = NULL;

cout<<"\n\tRoot Node is Added \n";

return;

}

if (tree->data == newnode->data)

{

cout<<"\n\n\tElement already in the tree";

return;

}

if (tree->data > newnode->data)

{

if (tree->left != NULL)

insert(tree->left, newnode); //Recursive function call to insert the new node else

{

tree->left = newnode;

(tree->left)->left = NULL;

(tree->left)->right = NULL;

cout<<"\n\n\tNode Added To Left";

return;

}

}

else

{

if (tree->right != NULL)

insert(tree->right, newnode); //Recursive function call to insert the new node

else

{

tree->right = newnode;

(tree->right)->left = NULL;

(tree->right)->right = NULL;

cout<<"\n\n\tNode Added To Right"<<endl;

return;

}

}

}

/\* Delete Element from the tree \*/

void BST::del(int item)

{

node \*parent, \*location;

if (root == NULL)

{

cout<<"\n\n\tTree empty";

return;

}

find(item, &parent, &location); //Recursive function call to find the element

if (location == NULL)

{

cout<<"\n\n\tItem not present in tree";

return;

}

if (location->left == NULL && location->right == NULL)

case\_a(parent, location); //Recursive Function call to case a

if (location->left != NULL && location->right == NULL)

case\_b(parent, location); //Recursive Function call to case b

if (location->left == NULL && location->right != NULL)

case\_b(parent, location); //Recursive Function call to case c

if (location->left != NULL && location->right != NULL)

case\_c(parent, location); //Recursive Function call to case d

free(location); //Function to free the location

}

/\* Case A\*/

void BST::case\_a(node \*par, node \*loc )

{

if (par == NULL)

root = NULL;

else

{

if (loc == par->left)

par->left = NULL;

else

par->right = NULL;

}

}

/\*Case B \*/

void BST::case\_b(node \*par, node \*loc)

{

node \*child;

if (loc->left != NULL)

child = loc->left;

else

child = loc->right;

if (par == NULL)

root = child;

else

{

if (loc == par->left)

par->left = child;

else

par->right = child;

}

}

/\* Case C \*/

void BST::case\_c(node \*par, node \*loc)

{

node \*ptr, \*ptrsave, \*suc, \*parsuc;

ptrsave = loc;

ptr = loc->right;

while (ptr->left != NULL) //Loop will execute till left child of node is NULL

{

ptrsave = ptr;

ptr = ptr->left;

}

suc = ptr;

parsuc = ptrsave;

if (suc->left == NULL && suc->right == NULL)

case\_a(parsuc, suc);

else

case\_b(parsuc, suc);

if (par == NULL)

root = suc;

else

{

if (loc == par->left)

par->left = suc;

else

par->right = suc;

}

suc->left = loc->left;

suc->right = loc->right;

}

/\* In Order Traversal \*/

void BST::inorder(node \*ptr)

{

if (root == NULL)

{

cout<<"\n\n\tTree is empty";

return;

}

if (ptr != NULL)

{

inorder(ptr->left);

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

inorder(ptr->right);

}

}

**Output:**

advait@ubuntu:~/advait/DSF$ g++ BST1.cpp

advait@ubuntu:~/advait/DSF$ ./a.out

-----------------

Operations on BST

-----------------

1.Insert Element

2.Delete Element

3.Search

4.Display of BST

5.Mirror Image

6.Display level wise

7.Quit

Enter your choice : 1

Enter the number to be inserted : 1

Root Node is Added

-----------------

Operations on BST

-----------------

1.Insert Element

2.Delete Element

3.Search

4.Display of BST

5.Mirror Image

6.Display level wise

7.Quit

Enter your choice : 1

Enter the number to be inserted : 2

Node Added To Right

-----------------

Operations on BST

-----------------

1.Insert Element

2.Delete Element

3.Search

4.Display of BST

5.Mirror Image

6.Display level wise

7.Quit

Enter your choice : 1

Enter the number to be inserted : 3

Node Added To Right

-----------------

Operations on BST

-----------------

1.Insert Element

2.Delete Element

3.Search

4.Display of BST

5.Mirror Image

6.Display level wise

7.Quit

Enter your choice : 1

Enter the number to be inserted : 4

Node Added To Right

-----------------

Operations on BST

-----------------

1.Insert Element

2.Delete Element

3.Search

4.Display of BST

5.Mirror Image

6.Display level wise

7.Quit

Enter your choice : 2

Enter the number to be deleted : 2

-----------------

Operations on BST

-----------------

1.Insert Element

2.Delete Element

3.Search

4.Display of BST

5.Mirror Image

6.Display level wise

7.Quit

Enter your choice : 4

Dislpay of BST:1 3 4

-----------------

Operations on BST

-----------------

1.Insert Element

2.Delete Element

3.Search

4.Display of BST

5.Mirror Image

6.Display level wise

7.Quit

Enter your choice : 3

Enter the no to serached:3

Element Found

-----------------

Operations on BST

-----------------

1.Insert Element

2.Delete Element

3.Search

4.Display of BST

5.Mirror Image

6.Display level wise

7.Quit

Enter your choice : 5

Mirror image :4 3 1

Node Added To Right

-----------------

Operations on BST

-----------------

1.Insert Element

2.Delete Element

3.Search

4.Display of BST

5.Mirror Image

6.Display level wise

7.Quit

Enter your choice : 6

1 3 4 4

-----------------

Operations on BST

-----------------

1.Insert Element

2.Delete Element

3.Search

4.Display of BST

5.Mirror Image

6.Display level wise

7.Quit

Enter your choice : 7