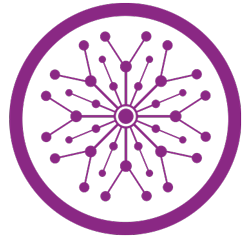
**LAB:12**



**Submitted by:** Abdulah Iqbal

**Roll No:**SU92-BSSE-S24-070

**Software Engineering**

**Section:** BSSE-3A

**Submitted to: Sir Rasikh Ali**

**QUESTION - 01:**

1. **Insert and Traverse for BST**

#include <iostream>

using namespace std;

struct Node

{

int key;

Node \*left, \*right;

Node(int k= 0);

~Node(){

delete left , right;

left = nullptr;

right = nullptr;

}

};

Node::Node(int k )

{

{

key = k;

left = right = nullptr;

}

}

class Binary\_search\_tree

{

private:

Node \*root;

Node\* deleteNode(Node\* rootNode , int k){

/\*

7

5 8

4 6 90

\*/

if(!rootNode ) return nullptr; //Base Case

//search for Node

if(k <rootNode->key){

rootNode->left = deleteNode(rootNode->left,k);

}else if(k > rootNode->key){

rootNode->right = deleteNode(rootNode->right,k);

}else {

//Node found -now delete it

//case 1 : No Child or one child

if(!rootNode->left){

Node\* temp = rootNode->right;

delete rootNode;

return temp;

}else if(!rootNode->right) {

Node\* temp = rootNode->left;

delete rootNode;

return temp;

}

//case 3: Two Children

Node\* successor = findMin(rootNode->right);

rootNode->key = successor->key;

rootNode->right = deleteNode(rootNode->right,successor->key);

}

return rootNode;

}

public:

Binary\_search\_tree(){

root = nullptr;

}

~Binary\_search\_tree(){};

void insert(int k)

{

Node \*newNode = new Node(k); // 10

if (root != nullptr)

{

Node \*temp = root;

while (true)

{

if (k <temp->key)

{

if(temp->left == nullptr){

temp->left = newNode;

break;

}

// Left side

temp = temp->left;

}

else

{

// RIght side

if(temp->right == nullptr){

temp->right = newNode;

break;

}

temp = temp->right;

}

}

// cout<<"Added Successfully.\n";

/\*

BST have two rules ...the Next newNode will be linked if its samller than the root Node it will linked on left side and if its bigger than the root Node it will linked on right side

BST Have

1--- insert ,

3--- search

4--- update

\*/

}

else

{

root = newNode;

}

}

void delete\_Node(int k)

{

root = deleteNode(root,k);

}

Node\* findMin(Node\* node ){

while (node->left != nullptr)

{

node = node->left;

}

return node;

}

void search(int k)

{

if (root != nullptr)

{

Node \*temp = root;

while (temp != nullptr && temp->key != k )

{

if (k > temp->key)

{

temp = temp->right;

}else if(temp == nullptr){

break;

}

else

{

temp = temp->left;

}

}

if (temp->key == k && temp != nullptr)

{

cout << "\nFound Out. \n Address : " << temp;

}

else

{

cout << "Not Found Out.\n";

}

}

else

{

cout << "Tree is Empty.\n";

}

}

void update(int d, int newValue)

{

Node \*temp = root;

while (temp->key != d)

{

if (d > temp->key)

{

temp = temp->right;

}

else

{

temp = temp->left;

}

}

temp->key = newValue;

if (temp->key == newValue)

{

cout << "Value Updated Successfully.\n";

}

else

{

cout << "Error While Updating the Value.\n";

}

}

void inorder(Node\* root)

{

if(root == nullptr) {

return ;

}else {

inorder(root->left);

cout<<root->key<<" -> ";

inorder(root->right);

}

}

void preOrder(Node\* root){

if(root == nullptr){

return ;

}else{

preOrder(root->left);

postOder(root->right);

cout<<root->key<<" -> ";

}

}

void postOder(Node\* root){

if(root == nullptr){

return ;

}else {

cout<<root->key <<" -> ";

postOder(root->left);

postOder(root->right);

}

}

Node\* getRoot(){

return root;

}

};

int main()

{

int keys[5] = { 90,2 ,8,7,91};

Binary\_search\_tree bst;

for(int key : keys){

bst.insert(key);

}

cout<<"Pre-order Traverse : ";

bst.preOrder(bst.getRoot());

cout<<endl;

bst.delete\_Node(8); //Deleting 8 key

cout<<endl;

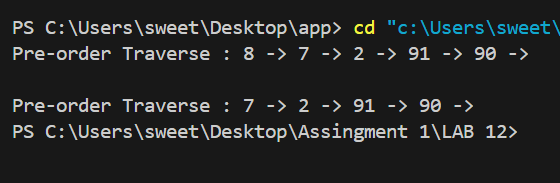
cout<<"Pre-order Traverse : ";

bst.preOrder(bst.getRoot());

cout<<endl;

}

**OUTPUT**



**QUESTION NO: 02**

1. **Insert and Traverse for AVL**

#include<iostream>

using namespace std;

struct Node {

int key, height;

Node \*left, \*right;

Node(int k = 0) {

key = k;

left = NULL;

right = NULL;

height = 1;

}

};

class Avl {

Node\* root;

int max(int a, int b) {

return (a > b) ? a : b;

}

int height(Node\* N) {

if (N == NULL) return 0;

return N->height;

}

Node\* rightRotate(Node\* y) {

Node\* x = y->left;

Node\* t2 = x->right;

x->right = y;

y->left = t2;

y->height = 1 + max(height(y->left), height(y->right));

x->height = 1 + max(height(x->left), height(x->right));

return x;

}

Node\* leftRotate(Node\* x) {

Node\* y = x->right;

Node\* t2 = y->left;

y->left = x;

x->right = t2;

x->height = 1 + max(height(x->left), height(x->right));

y->height = 1 + max(height(y->left), height(y->right));

return y;

}

Node\* minValueNode(Node\* node) {

Node\* current = node;

while (current && current->left != NULL)

current = current->left;

return current;

}

Node\* deleteNodeHelper(Node\* node, int key) {

if (node == NULL) return node;

if (key < node->key)

node->left = deleteNodeHelper(node->left, key);

else if (key > node->key)

node->right = deleteNodeHelper(node->right, key);

else {

if (node->left == NULL || node->right == NULL) {

Node\* temp = node->left ? node->left : node->right;

if (temp == NULL) {

temp = node;

node = NULL;

}

else {

\*node = \*temp;

}

delete temp;

}

else {

Node\* temp = minValueNode(node->right);

node->key = temp->key;

node->right = deleteNodeHelper(node->right, temp->key);

}

}

if (node == NULL) return node;

node->height = 1 + max(height(node->left), height(node->right));

int balance = getBalancedFactor(node);

if (balance > 1 && getBalancedFactor(node->left) >= 0)

return rightRotate(node);

if (balance > 1 && getBalancedFactor(node->left) < 0) {

node->left = leftRotate(node->left);

return rightRotate(node);

}

if (balance < -1 && getBalancedFactor(node->right) <= 0)

return leftRotate(node);

if (balance < -1 && getBalancedFactor(node->right) > 0) {

node->right = rightRotate(node->right);

return leftRotate(node);

}

return node;

}

Node\* insertNode(Node\* node, int key) {

if (node == NULL)

return new Node(key);

if (key < node->key)

node->left = insertNode(node->left, key);

else if (key > node->key)

node->right = insertNode(node->right, key);

else

return node;

node->height = 1 + max(height(node->left), height(node->right));

int balance = getBalancedFactor(node);

if (balance > 1 && key < node->left->key)

return rightRotate(node);

if (balance < -1 && key > node->right->key)

return leftRotate(node);

if (balance > 1 && key > node->left->key) {

node->left = leftRotate(node->left);

return rightRotate(node);

}

if (balance < -1 && key < node->right->key) {

node->right = rightRotate(node->right);

return leftRotate(node);

}

return node;

}

public:

Avl() {

root = NULL;

}

int getBalancedFactor(Node\* node) {

if (node == NULL) return 0;

return height(node->left) - height(node->right);

}

void insert(int key) {

root = insertNode(root, key);

}

void deleteNode(int key) {

root = deleteNodeHelper(root, key);

}

void inorder(Node\* node) {

if (node == NULL) return;

inorder(node->left);

cout << node->key << " -> ";

inorder(node->right);

}

void preorder(Node\* node) {

if (node != NULL) {

cout << node->key << " ";

preorder(node->left);

preorder(node->right);

}

}

Node\* getRoot() const { return root; }

};

int main() {

int nodes[] = {10, 20, 30, 40, 50, 25};

Avl avl;

for (int key : nodes) {

avl.insert(key);

}

cout << "Preorder traversal before deletion: ";

avl.preorder(avl.getRoot());

cout << endl;

avl.deleteNode(30);

cout << "Preorder traversal after deletion: ";

avl.preorder(avl.getRoot());

cout << endl;

return 0;

}

**OUTPUT**

