**M. Ali. Arslan**

**19F-0348**

**Lab-06**

**Task # 01**

#include <iostream>

using namespace std;

class Node

{

public:

int key;

Node\* leftChild;

Node\* rightChild;

Node(int data)

{

key = data;

leftChild = NULL;

rightChild = NULL;

}

void display()

{

cout << "key is: " << key << endl;

}

};

class TREE

{

public:

Node\* root;

Node\* root2;

TREE()

{

root = NULL; root2 = NULL;

}

bool empty()

{

if (root == NULL)

return true;

else

return false;

}

void insert(Node\* i)

{

if (root == NULL)

root = i;

else

{

Node\* curr = root;

Node\* parent;

while (1)

{

parent = curr;

if (i->key < curr->key)

{

curr = curr->leftChild;

if (curr == NULL)

{

parent->leftChild = i;

return;

}

}

else

{

curr = curr->rightChild;

if (curr == NULL)

{

parent->rightChild = i;

return;

}

}

}

}

}

void Deletion\_in\_BST(int key)

{

if (root == NULL)

{

cout << " TREE IS EMPTY!" << endl;

return;

}

else

{

Node\* curr = root;

Node\* parent = root;

bool Go\_Right = true;

while (curr->key != key)

{

parent = curr;

if (key < curr->key)

{

curr = curr->leftChild;

Go\_Right = false;

}

else

{

curr = curr->rightChild;

Go\_Right = true;

}

if (curr == NULL)

{

cout << "\n KEY IS NOT FOUND: " << endl;

return;

}

}

if (curr->leftChild == NULL && curr->rightChild == NULL)

{

if (curr == root)

{

root = NULL;

}

else if (Go\_Right == false)

{

parent->leftChild = NULL;

delete curr;

}

else

{

parent->rightChild = NULL;

delete curr;

}

}

else if (curr->rightChild == NULL)

{

if (curr == root)

{

root = curr->leftChild;

}

else if (Go\_Right == false)

{

parent->leftChild = curr->leftChild;

delete curr;

}

else

{

parent->rightChild = curr->leftChild;

delete curr;

}

}

else if (curr->leftChild == NULL)

{

if (curr == root)

{

root = curr->rightChild;

}

else if (Go\_Right == false)

{

parent->leftChild = curr->rightChild;

delete curr;

}

else

{

parent->rightChild = curr->rightChild;

delete curr;

}

}

else

{

Node\* successor = getSuccessor(curr);

if (curr == root)

{

root = successor;

}

else if (Go\_Right == false)

{

parent->leftChild = successor;

}

else

{

parent->rightChild = successor;

}

successor->leftChild = curr->leftChild;

delete curr;

}

}

}

Node\* getSuccessor(Node\* curr)

{

Node\* Psuccessor = curr;

Node\* successor = curr;

curr = curr->rightChild;

if (curr != NULL)

{

Psuccessor = successor;

successor = curr;

curr = curr->leftChild;

}

return successor;

}

void InOrderTraversing(Node\* localRoot)

{

if (localRoot != NULL)

{

InOrderTraversing(localRoot->leftChild);

cout << localRoot->key << "\t";

InOrderTraversing(localRoot->rightChild);

}

}

void PreOrderTraversing(Node\* localRoot)

{

if (localRoot != NULL)

{

cout << localRoot->key << "\t";

InOrderTraversing(localRoot->leftChild);

InOrderTraversing(localRoot->rightChild);

}

}

void PostOrderTraversing(Node\* localRoot)

{

if (localRoot != NULL)

{

InOrderTraversing(localRoot->leftChild);

InOrderTraversing(localRoot->rightChild);

cout << localRoot->key << "\t";

}

}

};

int main() {

TREE\* t = new TREE();

t->insert(new Node(50));

t->insert(new Node(40));

t->insert(new Node(60));

t->insert(new Node(30));

t->insert(new Node(55));

t->insert(new Node(44));

t->insert(new Node(52));

t->insert(new Node(67));

t->insert(new Node(69));

cout << " BEFORE DELETION \n\n";

cout << " InOrderTraversing \n";

t->InOrderTraversing(t->root);

cout << "\n Pre-OrderTraversing \n";

t->PreOrderTraversing(t->root);

cout << "\n Post-OrderTraversing \n";

t->PostOrderTraversing(t->root);

int key;

cout << "\n\nEnter which node you want to delete: ";

cin >> key;

t->Deletion\_in\_BST(key);

cout << "\n\n AFTER DELETION \n\n";

cout << " InOrderTraversing \n";

t->InOrderTraversing(t->root);

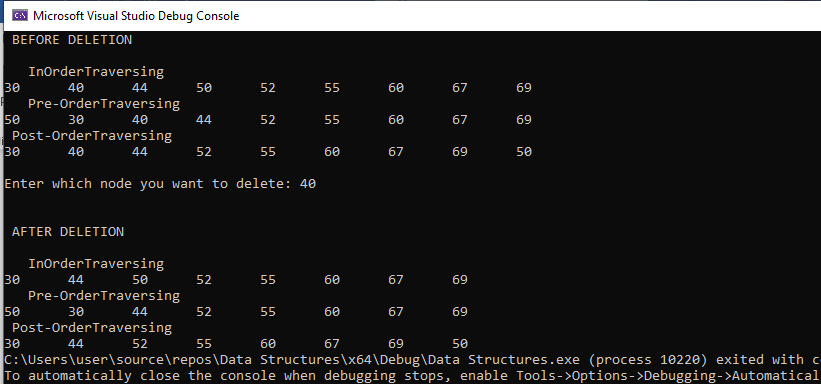
cout << "\n Pre-OrderTraversing \n";

t->PreOrderTraversing(t->root);

cout << "\n Post-OrderTraversing \n";

t->PostOrderTraversing(t->root);

}



**Task # 02**

#include <iostream>

using namespace std;

class Node

{

public:

int key;

Node\* leftChild;

Node\* rightChild;

Node(int data)

{

key = data;

leftChild = NULL;

rightChild = NULL;

}

void display()

{

cout << "key is: " << key << endl;

}

};

class TREE

{

public:

Node\* root;

Node\* root2;

TREE()

{

root = NULL; root2 = NULL;

}

bool empty()

{

if (root == NULL)

return true;

else

return false;

}

void insert(Node\* i)

{

if (root == NULL)

root = i;

else

{

Node\* curr = root;

Node\* parent;

while (1)

{

parent = curr;

if (i->key < curr->key)

{

curr = curr->leftChild;

if (curr == NULL)

{

parent->leftChild = i;

return;

}

}

else

{

curr = curr->rightChild;

if (curr == NULL)

{

parent->rightChild = i;

return;

}

}

}

}

}

void Deletion\_in\_BST(int key)

{

if (root == NULL)

{

cout << " TREE IS EMPTY!" << endl;

return;

}

else

{

Node\* curr = root;

Node\* parent = root;

bool Go\_Right = true;

while (curr->key != key)

{

parent = curr;

if (key < curr->key)

{

curr = curr->leftChild;

Go\_Right = false;

}

else

{

curr = curr->rightChild;

Go\_Right = true;

}

if (curr == NULL)

{

cout << "\n KEY IS NOT FOUND: " << endl;

return;

}

}

if (curr->leftChild == NULL && curr->rightChild == NULL)

{

if (curr == root)

{

root = NULL;

}

else if (Go\_Right == false)

{

parent->leftChild = NULL;

delete curr;

}

else

{

parent->rightChild = NULL;

delete curr;

}

}

else if (curr->rightChild == NULL)

{

if (curr == root)

{

root = curr->leftChild;

}

else if (Go\_Right == false)

{

parent->leftChild = curr->leftChild;

delete curr;

}

else

{

parent->rightChild = curr->leftChild;

delete curr;

}

}

else if (curr->leftChild == NULL)

{

if (curr == root)

{

root = curr->rightChild;

}

else if (Go\_Right == false)

{

parent->leftChild = curr->rightChild;

delete curr;

}

else

{

parent->rightChild = curr->rightChild;

delete curr;

}

}

else

{

Node\* successor = getSuccessor(curr);

if (curr == root)

{

root = successor;

}

else if (Go\_Right == false)

{

parent->leftChild = successor;

}

else

{

parent->rightChild = successor;

}

successor->leftChild = curr->leftChild;

delete curr;

}

}

}

Node\* getSuccessor(Node\* curr)

{

Node\* Psuccessor = curr;

Node\* successor = curr;

curr = curr->rightChild;

if (curr != NULL)

{

Psuccessor = successor;

successor = curr;

curr = curr->leftChild;

}

return successor;

}

void InOrderTraversing(Node\* localRoot)

{

if (localRoot != NULL)

{

InOrderTraversing(localRoot->leftChild);

cout << localRoot->key << "\t";

InOrderTraversing(localRoot->rightChild);

}

}

void PreOrderTraversing(Node\* localRoot)

{

if (localRoot != NULL)

{

cout << localRoot->key << "\t";

InOrderTraversing(localRoot->leftChild);

InOrderTraversing(localRoot->rightChild);

}

}

void PostOrderTraversing(Node\* localRoot)

{

if (localRoot != NULL)

{

InOrderTraversing(localRoot->leftChild);

InOrderTraversing(localRoot->rightChild);

cout << localRoot->key << "\t";

}

}

int countNode(Node\* root)

{

if (root == NULL)

{

return 0;

}

return 1 + countNode(root->leftChild) + countNode(root->rightChild);

}

};

int main() {

TREE\* t = new TREE();

t->insert(new Node(50));

t->insert(new Node(40));

t->insert(new Node(60));

t->insert(new Node(30));

t->insert(new Node(55));

t->insert(new Node(44));

t->insert(new Node(52));

t->insert(new Node(67));

t->insert(new Node(69));

cout << " BEFORE DELETION \n\n";

cout << " InOrderTraversing \n";

t->InOrderTraversing(t->root);

cout << "\n Pre-OrderTraversing \n";

t->PreOrderTraversing(t->root);

cout << "\n Post-OrderTraversing \n";

t->PostOrderTraversing(t->root);

int key;

cout << "\n\nEnter which node you want to delete: ";

cin >> key;

t->Deletion\_in\_BST(key);

cout << "\n\n AFTER DELETION \n\n";

cout << " InOrderTraversing \n";

t->InOrderTraversing(t->root);

cout << "\n Pre-OrderTraversing \n";

t->PreOrderTraversing(t->root);

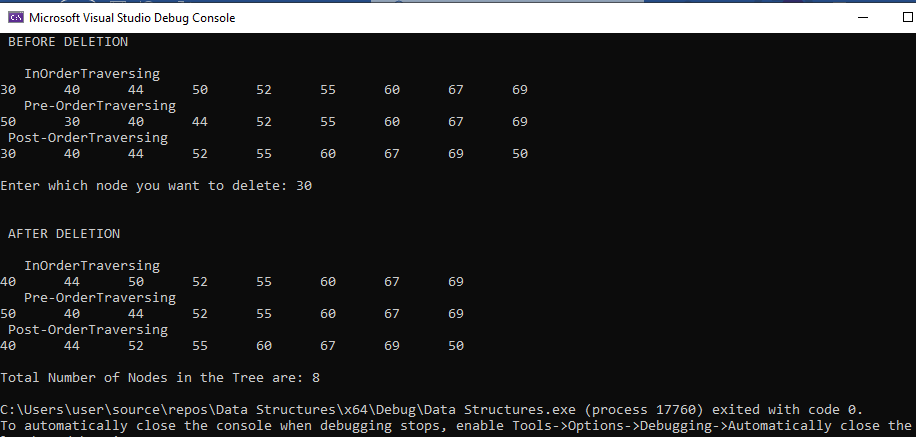
cout << "\n Post-OrderTraversing \n";

t->PostOrderTraversing(t->root);

cout << "\n\nTotal Number of Nodes in the Tree are: " << t->countNode(t->root);

cout << endl;

}



**Task # 03**

#include <iostream>

using namespace std;

class Node

{

public:

int key;

Node\* leftChild;

Node\* rightChild;

Node(int data)

{

key = data;

leftChild = NULL;

rightChild = NULL;

}

void display()

{

cout << "key is: " << key << endl;

}

};

class TREE

{

public:

Node\* root;

Node\* root2;

TREE()

{

root = NULL; root2 = NULL;

}

bool empty()

{

if (root == NULL)

return true;

else

return false;

}

void insert(Node\* i)

{

if (root == NULL)

root = i;

else

{

Node\* curr = root;

Node\* parent;

while (1)

{

parent = curr;

if (i->key < curr->key)

{

curr = curr->leftChild;

if (curr == NULL)

{

parent->leftChild = i;

return;

}

}

else

{

curr = curr->rightChild;

if (curr == NULL)

{

parent->rightChild = i;

return;

}

}

}

}

}

void Deletion\_in\_BST(int key)

{

if (root == NULL)

{

cout << " TREE IS EMPTY!" << endl;

return;

}

else

{

Node\* curr = root;

Node\* parent = root;

bool Go\_Right = true;

while (curr->key != key)

{

parent = curr;

if (key < curr->key)

{

curr = curr->leftChild;

Go\_Right = false;

}

else

{

curr = curr->rightChild;

Go\_Right = true;

}

if (curr == NULL)

{

cout << "\n KEY IS NOT FOUND: " << endl;

return;

}

}

if (curr->leftChild == NULL && curr->rightChild == NULL)

{

if (curr == root)

{

root = NULL;

}

else if (Go\_Right == false)

{

parent->leftChild = NULL;

delete curr;

}

else

{

parent->rightChild = NULL;

delete curr;

}

}

else if (curr->rightChild == NULL)

{

if (curr == root)

{

root = curr->leftChild;

}

else if (Go\_Right == false)

{

parent->leftChild = curr->leftChild;

delete curr;

}

else

{

parent->rightChild = curr->leftChild;

delete curr;

}

}

else if (curr->leftChild == NULL)

{

if (curr == root)

{

root = curr->rightChild;

}

else if (Go\_Right == false)

{

parent->leftChild = curr->rightChild;

delete curr;

}

else

{

parent->rightChild = curr->rightChild;

delete curr;

}

}

else

{

Node\* successor = getSuccessor(curr);

if (curr == root)

{

root = successor;

}

else if (Go\_Right == false)

{

parent->leftChild = successor;

}

else

{

parent->rightChild = successor;

}

successor->leftChild = curr->leftChild;

delete curr;

}

}

}

Node\* getSuccessor(Node\* curr)

{

Node\* Psuccessor = curr;

Node\* successor = curr;

curr = curr->rightChild;

if (curr != NULL)

{

Psuccessor = successor;

successor = curr;

curr = curr->leftChild;

}

return successor;

}

void InOrderTraversing(Node\* localRoot)

{

if (localRoot != NULL)

{

InOrderTraversing(localRoot->leftChild);

cout << localRoot->key << "\t";

InOrderTraversing(localRoot->rightChild);

}

}

void PreOrderTraversing(Node\* localRoot)

{

if (localRoot != NULL)

{

cout << localRoot->key << "\t";

InOrderTraversing(localRoot->leftChild);

InOrderTraversing(localRoot->rightChild);

}

}

void PostOrderTraversing(Node\* localRoot)

{

if (localRoot != NULL)

{

InOrderTraversing(localRoot->leftChild);

InOrderTraversing(localRoot->rightChild);

cout << localRoot->key << "\t";

}

}

int Count\_Leaves\_Node(Node\* node)

{

if (node == NULL)

{

return 0;

}

if (node->leftChild == NULL && node->rightChild == NULL)

{

return 1;

}

else

{

return Count\_Leaves\_Node(node->leftChild) + Count\_Leaves\_Node(node->rightChild);

}

}

};

int main() {

TREE\* t = new TREE();

t->insert(new Node(50));

t->insert(new Node(40));

t->insert(new Node(60));

t->insert(new Node(30));

t->insert(new Node(55));

t->insert(new Node(44));

t->insert(new Node(52));

t->insert(new Node(67));

t->insert(new Node(69));

cout << " BEFORE DELETION \n\n";

cout << " InOrderTraversing \n";

t->InOrderTraversing(t->root);

cout << "\n Pre-OrderTraversing \n";

t->PreOrderTraversing(t->root);

cout << "\n Post-OrderTraversing \n";

t->PostOrderTraversing(t->root);

int key;

cout << "\n\nEnter which node you want to delete: ";

cin >> key;

t->Deletion\_in\_BST(key);

cout << "\n\n AFTER DELETION \n\n";

cout << " InOrderTraversing \n";

t->InOrderTraversing(t->root);

cout << "\n Pre-OrderTraversing \n";

t->PreOrderTraversing(t->root);

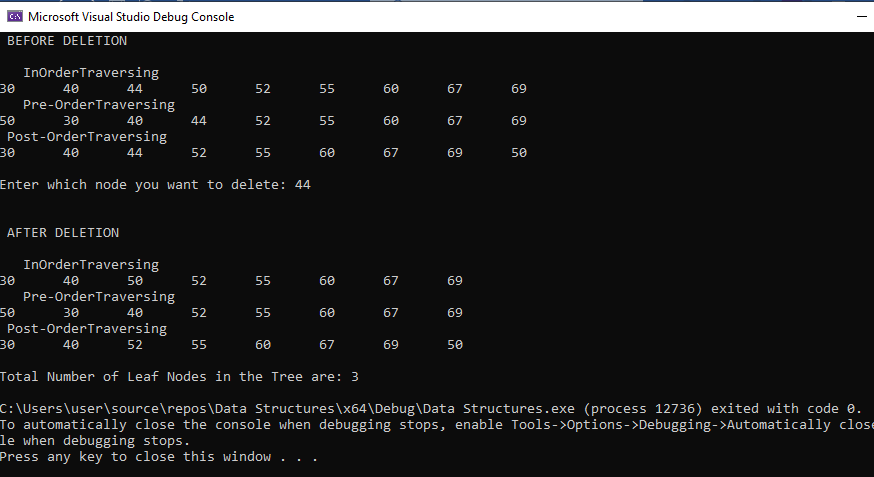
cout << "\n Post-OrderTraversing \n";

t->PostOrderTraversing(t->root);

cout << "\n\nTotal Number of Leaf Nodes in the Tree are: " << t->Count\_Leaves\_Node(t->root);

cout << endl;

}



**Task # 04**

#include <iostream>

using namespace std;

class Node

{

public:

int key;

Node\* leftChild;

Node\* rightChild;

Node(int data)

{

key = data;

leftChild = NULL;

rightChild = NULL;

}

void display()

{

cout << "key is: " << key << endl;

}

};

class TREE

{

public:

Node\* root;

Node\* root2;

TREE()

{

root = NULL; root2 = NULL;

}

bool empty()

{

if (root == NULL)

return true;

else

return false;

}

void insert(Node\* i)

{

if (root == NULL)

root = i;

else

{

Node\* curr = root;

Node\* parent;

while (1)

{

parent = curr;

if (i->key < curr->key)

{

curr = curr->leftChild;

if (curr == NULL)

{

parent->leftChild = i;

return;

}

}

else

{

curr = curr->rightChild;

if (curr == NULL)

{

parent->rightChild = i;

return;

}

}

}

}

}

void Deletion\_in\_BST(int key)

{

if (root == NULL)

{

cout << " TREE IS EMPTY!" << endl;

return;

}

else

{

Node\* curr = root;

Node\* parent = root;

bool Go\_Right = true;

while (curr->key != key)

{

parent = curr;

if (key < curr->key)

{

curr = curr->leftChild;

Go\_Right = false;

}

else

{

curr = curr->rightChild;

Go\_Right = true;

}

if (curr == NULL)

{

cout << "\n KEY IS NOT FOUND: " << endl;

return;

}

}

if (curr->leftChild == NULL && curr->rightChild == NULL)

{

if (curr == root)

{

root = NULL;

}

else if (Go\_Right == false)

{

parent->leftChild = NULL;

delete curr;

}

else

{

parent->rightChild = NULL;

delete curr;

}

}

else if (curr->rightChild == NULL)

{

if (curr == root)

{

root = curr->leftChild;

}

else if (Go\_Right == false)

{

parent->leftChild = curr->leftChild;

delete curr;

}

else

{

parent->rightChild = curr->leftChild;

delete curr;

}

}

else if (curr->leftChild == NULL)

{

if (curr == root)

{

root = curr->rightChild;

}

else if (Go\_Right == false)

{

parent->leftChild = curr->rightChild;

delete curr;

}

else

{

parent->rightChild = curr->rightChild;

delete curr;

}

}

else

{

Node\* successor = getSuccessor(curr);

if (curr == root)

{

root = successor;

}

else if (Go\_Right == false)

{

parent->leftChild = successor;

}

else

{

parent->rightChild = successor;

}

successor->leftChild = curr->leftChild;

delete curr;

}

}

}

Node\* getSuccessor(Node\* curr)

{

Node\* Psuccessor = curr;

Node\* successor = curr;

curr = curr->rightChild;

if (curr != NULL)

{

Psuccessor = successor;

successor = curr;

curr = curr->leftChild;

}

return successor;

}

void InOrderTraversing(Node\* localRoot)

{

if (localRoot != NULL)

{

InOrderTraversing(localRoot->leftChild);

cout << localRoot->key << "\t";

InOrderTraversing(localRoot->rightChild);

}

}

void PreOrderTraversing(Node\* localRoot)

{

if (localRoot != NULL)

{

cout << localRoot->key << "\t";

InOrderTraversing(localRoot->leftChild);

InOrderTraversing(localRoot->rightChild);

}

}

void PostOrderTraversing(Node\* localRoot)

{

if (localRoot != NULL)

{

InOrderTraversing(localRoot->leftChild);

InOrderTraversing(localRoot->rightChild);

cout << localRoot->key << "\t";

}

}

};

int main() {

TREE\* t = new TREE();

t->insert(new Node(50));

t->insert(new Node(40));

t->insert(new Node(60));

t->insert(new Node(30));

t->insert(new Node(55));

t->insert(new Node(44));

t->insert(new Node(52));

t->insert(new Node(67));

t->insert(new Node(69));

cout << " BEFORE DELETION \n\n";

cout << " InOrderTraversing \n";

t->InOrderTraversing(t->root);

cout << "\n Pre-OrderTraversing \n";

t->PreOrderTraversing(t->root);

cout << "\n Post-OrderTraversing \n";

t->PostOrderTraversing(t->root);

int key;

cout << "\n\nEnter which node you want to delete: ";

cin >> key;

t->Deletion\_in\_BST(key);

cout << "\n\n AFTER DELETION \n\n";

cout << " InOrderTraversing \n";

t->InOrderTraversing(t->root);

cout << "\n Pre-OrderTraversing \n";

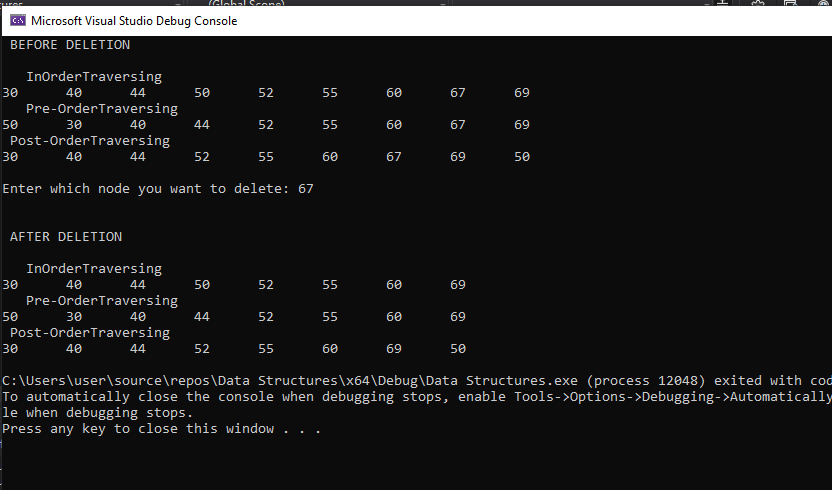
t->PreOrderTraversing(t->root);

cout << "\n Post-OrderTraversing \n";

t->PostOrderTraversing(t->root);

cout << endl;

}



**Task # 05**

#include <iostream>

using namespace std;

class Node

{

public:

int key;

Node\* leftChild;

Node\* rightChild;

Node(int data)

{

key = data;

leftChild = NULL;

rightChild = NULL;

}

void display()

{

cout << "key is: " << key << endl;

}

};

class TREE

{

public:

Node\* root;

Node\* root2;

TREE()

{

root = NULL; root2 = NULL;

}

bool empty()

{

if (root == NULL)

return true;

else

return false;

}

void insert(Node\* i)

{

if (root == NULL)

root = i;

else

{

Node\* curr = root;

Node\* parent;

while (1)

{

parent = curr;

if (i->key < curr->key)

{

curr = curr->leftChild;

if (curr == NULL)

{

parent->leftChild = i;

return;

}

}

else

{

curr = curr->rightChild;

if (curr == NULL)

{

parent->rightChild = i;

return;

}

}

}

}

}

void insertM(Node\* i)

{

if (root2 == NULL)

root2 = i;

else

{

Node\* curr = root2;

Node\* parent;

while (1)

{

parent = curr;

if (i->key < curr->key)

{

curr = curr->leftChild;

if (curr == NULL)

{

parent->leftChild = i;

return;

}

}

else

{

curr = curr->rightChild;

if (curr == NULL)

{

parent->rightChild = i;

return;

}

}

}

}

}

void Deletion\_in\_BST(int key)

{

if (root == NULL)

{

cout << " TREE IS EMPTY!" << endl;

return;

}

else

{

Node\* curr = root;

Node\* parent = root;

bool Go\_Right = true;

while (curr->key != key)

{

parent = curr;

if (key < curr->key)

{

curr = curr->leftChild;

Go\_Right = false;

}

else

{

curr = curr->rightChild;

Go\_Right = true;

}

if (curr == NULL)

{

cout << "\n KEY IS NOT FOUND: " << endl;

return;

}

}

if (curr->leftChild == NULL && curr->rightChild == NULL)

{

if (curr == root)

{

root = NULL;

}

else if (Go\_Right == false)

{

parent->leftChild = NULL;

delete curr;

}

else

{

parent->rightChild = NULL;

delete curr;

}

}

else if (curr->rightChild == NULL)

{

if (curr == root)

{

root = curr->leftChild;

}

else if (Go\_Right == false)

{

parent->leftChild = curr->leftChild;

delete curr;

}

else

{

parent->rightChild = curr->leftChild;

delete curr;

}

}

else if (curr->leftChild == NULL)

{

if (curr == root)

{

root = curr->rightChild;

}

else if (Go\_Right == false)

{

parent->leftChild = curr->rightChild;

delete curr;

}

else

{

parent->rightChild = curr->rightChild;

delete curr;

}

}

else

{

Node\* successor = getSuccessor(curr);

if (curr == root)

{

root = successor;

}

else if (Go\_Right == false)

{

parent->leftChild = successor;

}

else

{

parent->rightChild = successor;

}

successor->leftChild = curr->leftChild;

delete curr;

}

}

}

Node\* getSuccessor(Node\* curr)

{

Node\* Psuccessor = curr;

Node\* successor = curr;

curr = curr->rightChild;

if (curr != NULL)

{

Psuccessor = successor;

successor = curr;

curr = curr->leftChild;

}

return successor;

}

void InOrderTraversing(Node\* localRoot)

{

if (localRoot != NULL)

{

InOrderTraversing(localRoot->leftChild);

cout << localRoot->key << "\t";

InOrderTraversing(localRoot->rightChild);

}

}

void PreOrderTraversing(Node\* localRoot)

{

if (localRoot != NULL)

{

cout << localRoot->key << "\t";

InOrderTraversing(localRoot->leftChild);

InOrderTraversing(localRoot->rightChild);

}

}

void PostOrderTraversing(Node\* localRoot)

{

if (localRoot != NULL)

{

InOrderTraversing(localRoot->leftChild);

InOrderTraversing(localRoot->rightChild);

cout << localRoot->key << "\t";

}

}

};

int main() {

TREE\* t = new TREE();

t->insertM(new Node(69));

t->insertM(new Node(67));

t->insertM(new Node(52));

t->insertM(new Node(64));

t->insertM(new Node(44));

t->insertM(new Node(55));

t->insertM(new Node(30));

t->insertM(new Node(60));

t->insertM(new Node(40));

t->insertM(new Node(50));

cout << "\n\n AFTER MODIFY THE INSERT METHOD \n\n";

cout << " InOrderTraversing \n";

t->InOrderTraversing(t->root2);

cout << "\n Pre-OrderTraversing \n";

t->PreOrderTraversing(t->root2);

cout << "\n Post-OrderTraversing \n";

t->PostOrderTraversing(t->root2);

cout << endl;

}

