GURNOOR SINGH VIRDI

190911076

IT-2

**L5R1:**

**Node.hpp and Node.cpp contain the Linked List implementation and are used by main**

**Node.hpp**

#pragma once

class Node

{

private:

Node\* next = nullptr;

int data = 0;

private:

void insert\_node(Node\* pointer, int data);

public:

Node();

Node(int data);

void insert\_head(Node\*& head, int data);

void insert\_tail(Node\*& head, int data);

void insert\_before(Node\*& head, int data, int before);

void insert\_after(Node\*& head, int data, int after);

void delete\_element(Node\*& head, int data);

void display(Node\*& head);

};

**Node.cpp**

#include <iostream>

#include "Node.hpp"

//inserts node after the pointer provided

void Node::insert\_node(Node\* pointer, int data)

{

Node\* next\_node = pointer->next;

pointer->next = new Node(data);

pointer->next->next = next\_node;

}

Node::Node()

{}

Node::Node(int data) : data(data), next(nullptr)

{

}

void Node::insert\_head(Node\*& head, int data)

{

if (!head) {

head = new Node(data);

return;

}

Node\* curr = head;

head = new Node(data);

head->next = curr;

}

void Node::insert\_tail(Node\*& head, int data)

{

if (!head) {

head = new Node(data);

return;

}

Node\* curr = head;

while (curr->next) {

curr = curr->next;

}

curr->next = new Node(data);

}

void Node::insert\_before(Node\*& head, int data, int before)

{

if (!head) {

head = new Node(data);

return;

}

if (head->data == before) {

insert\_head(head, data);

return;

}

Node\* curr = head;

while (curr->next) {

if (curr->next->data == before) {

insert\_node(curr, data);

return;

}

curr = curr->next;

}

std::cout << "Element not found!\n";

}

void Node::insert\_after(Node\*& head, int data, int after)

{

if (!head) {

head = new Node(data);

return;

}

Node\* curr = head;

while (curr) {

if (curr->data == after) {

insert\_node(curr, data);

curr = curr->next;

return;

}

curr = curr->next;

}

std::cout << "Element not found\n";

}

void Node::delete\_element(Node\*& head, int data)

{

if (!head) {

std::cout << "List is empty!\n";

return;

}

if (head->data == data) {

Node\* temp = head;

head = head->next;

delete temp;

temp = nullptr;

return;

}

Node\* curr = head;

while (curr->next) {

if (curr->next->data == data) {

Node\* next\_node = curr->next->next;

delete curr->next;

curr->next = next\_node;

return;

}

curr = curr->next;

}

std::cout << "Node not found\n";

}

void Node::display(Node\*& head)

{

if (!head) {

std::cout << "List is empty\n";

return;

}

Node\* curr = head;

while (curr) {

std::cout << curr->data << " ";

curr = curr->next;

}

std::cout << std::endl;

}

**Main.cpp**

#include <iostream>

#include "Node.hpp"

int main()

{

Node\* head = nullptr;

Node list;

int meth = 0;

std::cout << "1. AddHead 2. AddTail 3. InsertBefore 4. InsertTail 5. DeleteElement 6.Display\n";

while (true) {

std::cin >> meth;

switch (meth) {

case 1: {

std::cout << "Enter element to push:\n";

int x = 0;

std::cin >> x;

list.insert\_head(head, x);

break;

}

case 2: {

std::cout << "Enter element to push:\n";

int x = 0;

std::cin >> x;

list.insert\_tail(head, x);

break;

}

case 3: {

std::cout << "Enter element to push:\n";

int x = 0;

std::cin >> x;

std::cout << "Enter element to search for:\n";

int y = 0;

std::cin >> y;

list.insert\_before(head, x, y);

break;

}

case 4: {

std::cout << "Enter element to push:\n";

int x = 0;

std::cin >> x;

std::cout << "Enter element to search for:\n";

int y = 0;

std::cin >> y;

list.insert\_after(head, x, y);

break;

}

case 5: {

std::cout << "Enter element to delete:\n";

int x = 0;

std::cin >> x;

list.delete\_element(head, x);

break;

}

case 6: {

std::cout << "LIST: \n";

list.display(head);

break;

}

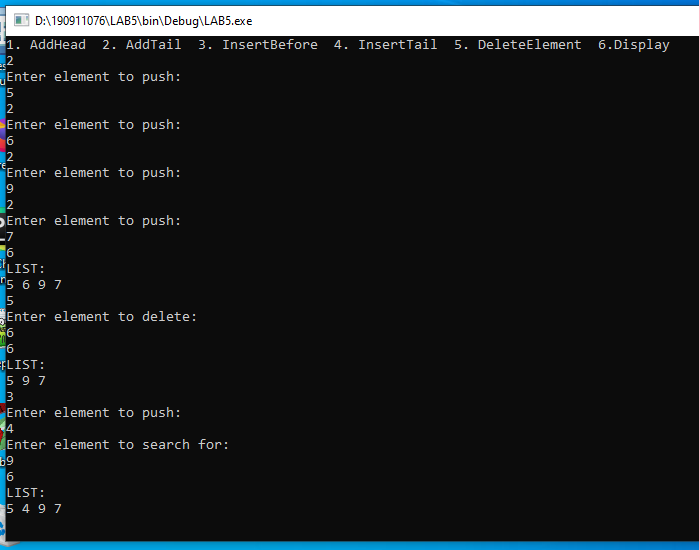
default: std::cout << "Exiting\n"; return 0;

}

}

return 0;

}



**L6R1:**

**DNode.hpp and cpp contain the doubly linked list implementation and main.cpp uses them**

**DNode.hpp**

#pragma once

class DNode

{

private:

DNode\* next = nullptr;

DNode\* prev = nullptr;

int data = 0;

private:

void insert\_node(DNode\*& pointer, int data);

public:

DNode();

DNode(int data);

void insert\_head(DNode\*& head, int data);

void insert\_tail(DNode\*& head, int data);

void delete\_tail(DNode\*& head);

void insert\_index(DNode\*& head, int data, int i);

void delete\_index(DNode\*& head, int i);

void insert\_after(DNode\*& head, int data, int after);

void insert\_before(DNode\*& head, int data, int before);

void display(DNode\*& head);

};

**Dnode.cpp**

#include <iostream>

#include "DNode.hpp"

void DNode::insert\_node(DNode\*& pointer, int data)

{

DNode\* next\_node = pointer->next;

pointer->next = new DNode(data);

pointer->next->next = next\_node;

pointer->next->prev = pointer;

next\_node->prev = pointer->next;

}

DNode::DNode()

{

}

DNode::DNode(int data) : data(data)

{

}

void DNode::insert\_head(DNode\*& head, int data)

{

if (!head) {

head = new DNode(data);

return;

}

DNode\* next\_node = head;

head = new DNode(data);

head->next = next\_node;

next\_node->prev = head;

}

void DNode::insert\_tail(DNode\*& head, int data)

{

if (!head) {

head = new DNode(data);

return;

}

DNode\* curr = head;

while (curr->next) {

curr = curr->next;

}

curr->next = new DNode(data);

curr->next->prev = curr;

}

void DNode::delete\_tail(DNode\*& head)

{

if (!head) {

std::cout << "List is empty" << std::endl;

return;

}

if (!head->next) {

delete head;

head = nullptr;

return;

}

DNode\* curr = head;

while (curr->next) {

curr = curr->next;

}

curr->prev->next = nullptr;

delete curr;

curr = nullptr;

}

void DNode::insert\_index(DNode\*& head, int data, int i)

{

if (!head) {

head = new DNode(data);

return;

}

if (i == 0) {

DNode\* next\_node = head;

head = new DNode(data);

head->next = next\_node;

next\_node->prev = head;

return;

}

i--;

DNode\* curr = head;

while (i > 0 && curr->next) {

curr = curr->next;

i--;

}

if (i != 0) {

std::cout << "Invalid index\n";

return;

}

if (!curr->next) {

insert\_tail(head, data);

return;

}

insert\_node(curr, data);

}

void DNode::delete\_index(DNode\*& head, int i)

{

if (!head) {

std::cout << "List is Empty\n";

return;

}

if (i == 0) {

DNode\* temp = head;

head = head->next;

head->prev = nullptr;

delete temp;

return;

}

if (!head->next) {

delete head;

head = nullptr;

return;

}

i--;

DNode\* curr = head->next;

while (i > 0 && curr->next) {

curr = curr->next;

i--;

}

if (i != 0) {

std::cout << "Invalid Index\n";

return;

}

if (!curr->next) {

delete\_tail(head);

return;

}

curr->prev->next = curr->next;

curr->next->prev = curr->prev;

delete curr;

}

void DNode::insert\_after(DNode\*& head, int data, int after)

{

if (!head) {

std::cout << "List is Empty\n";

return;

}

DNode\* curr = head;

while (curr->next) {

if (curr->data == after) {

insert\_node(curr, data);

return;

}

curr = curr->next;

}

if (curr->data) {

insert\_tail(head, data);

return;

}

std::cout << "Element not found\n";

}

void DNode::insert\_before(DNode\*& head, int data, int before)

{

if (!head) {

std::cout << "List is Empty\n";

return;

}

if (head->data == before) {

DNode\* temp = head;

head = new DNode(data);

head->next = temp;

temp->prev = head;

return;

}

DNode\* curr = head;

while (curr->next) {

if (curr->next->data == before) {

insert\_node(curr, data);

return;

}

curr = curr->next;

}

std::cout << "Element not found\n";

}

void DNode::display(DNode\*& head)

{

if (!head) {

std::cout << "List is empty\n";

return;

}

DNode\* curr = head;

while (curr) {

std::cout << curr->data << " ";

curr = curr->next;

}

std::cout << std::endl;

}

**Main.cpp**

#include <iostream>

#include "DNode.hpp"

int main()

{

DNode\* head = nullptr;

DNode list;

int meth = 0;

std::cout << "1. InsertTail 2. DeleteTail 3. InsertIndex 4. DeleteIndex 5. InsertAfter 6. InsertBefore 7. Display:\n";

while (true) {

std::cin >> meth;

switch (meth) {

case 1: {

std::cout << "Enter element to push:\n";

int x = 0;

std::cin >> x;

list.insert\_tail(head, x);

std::cout << "Tail inserted\n";

break;

}

case 2: {

list.delete\_tail(head);

std::cout << "Tail deleted\n";

break;

}

case 3: {

std::cout << "Enter element to push:\n";

int x = 0;

std::cin >> x;

std::cout << "Enter index: ";

int i = 0;

std::cin >> i;

list.insert\_index(head, x, i);

break;

}

case 4: {

std::cout << "Enter index:\n";

int y = 0;

std::cin >> y;

list.delete\_index(head, y);

break;

}

case 5: {

std::cout << "Enter element to push:\n";

int x = 0;

std::cin >> x;

std::cout << "Enter element to search for:\n";

int y = 0;

std::cin >> y;

list.insert\_after(head, x, y);

list.display(head);

break;

}

case 6: {

std::cout << "Enter element to push:\n";

int x = 0;

std::cin >> x;

std::cout << "Enter element to search for:\n";

int y = 0;

std::cin >> y;

list.insert\_before(head, x, y);

list.display(head);

break;

}

case 7: {

std::cout << "LIST: \n";

list.display(head);

break;

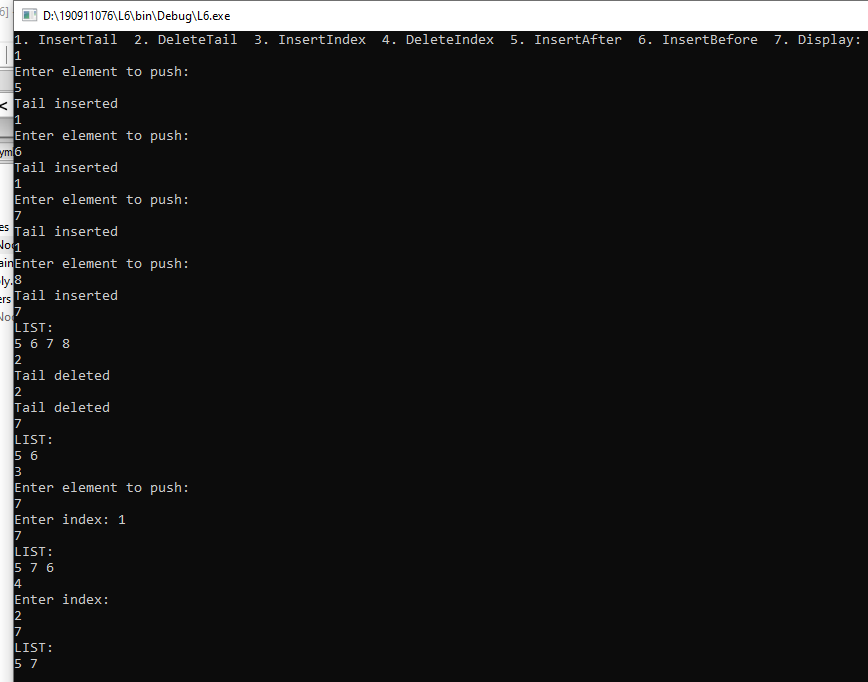
}

default: std::cout << "Exiting\n"; return 0;

}

}

}



**L6R2:**

**DNode.hpp and cpp has the Doubly Linked List for polynomials and main.cpp uses them**

**DNode.hpp**

#pragma once

class PolyNode

{

private:

PolyNode\* next = nullptr;

PolyNode\* prev = nullptr;

int coef = 0;

int power = 0;

public:

PolyNode();

PolyNode(int coef, int power);

void read(PolyNode\*& head);

void add(PolyNode\*& p1head, PolyNode\*& p2head);

void insert\_tail(PolyNode\*& head, int coef, int power);

void display(PolyNode\*& head);

};

**DNode.cpp**

#include <iostream>

#include "DNode.hpp"

PolyNode::PolyNode()

{

}

PolyNode::PolyNode(int coef, int power) : coef(coef), power(power)

{

}

void PolyNode::read(PolyNode\*& head)

{

int c = 0, p = 0;

while (true) {

std::cout << "Enter coef and power of term: ";

std::cin >> c >> p;

if (c == 0) {

continue;

}

else if (c == -999) {

return;

}

insert\_tail(head, c, p);

std::cout << "Term added\n";

}

}

void PolyNode::add(PolyNode\*& p1head, PolyNode\*& p2head)

{

PolyNode\* p1 = p1head;

PolyNode\* p2 = p2head;

if (!p2) {

std::cout << "Addition: ";

display(p1);

return;

}

else if (!p1) {

std::cout << "Addition: ";

display(p2);

return;

}

PolyNode\* p3 = nullptr;

int meth = -1;

while (p1 && p2) {

if (p1->power == p2->power) {

meth = 0;

}

else if (p1->power > p2->power) {

meth = 1;

}

else {

meth = 2;

}

switch (meth) {

case 0: {

insert\_tail(p3, p1->coef + p2->coef, p1->power);

p1 = p1->next;

p2 = p2->next;

break;

}

case 1: {

insert\_tail(p3, p1->coef, p1->power);

p1 = p1->next;

break;

}

case 2: {

insert\_tail(p3, p2->coef, p2->power);

p2 = p2->next;

break;

}

default: break;

}

}

while (p1) {

insert\_tail(p3, p1->coef, p1->power);

p1 = p1->next;

break;

}

while (p2) {

insert\_tail(p3, p2->coef, p2->power);

p2 = p2->next;

break;

}

std::cout << "Addition: ";

display(p3);

}

void PolyNode::insert\_tail(PolyNode\*& head, int coef, int power)

{

if (!head) {

head = new PolyNode(coef, power);

return;

}

PolyNode\* curr = head;

while (curr->next) {

curr = curr->next;

}

curr->next = new PolyNode(coef, power);

curr->next->prev = curr;

}

void PolyNode::display(PolyNode\*& head)

{

if (!head) {

return;

}

PolyNode\* curr = head;

while (curr->next) {

if (curr->coef == 0) {

curr = curr->next;

continue;

}

std::cout << curr->coef << "x^" << curr->power;

curr = curr->next;

if (curr->coef > 0) {

std::cout << " + ";

}

else {

std::cout << " ";

}

}

std::cout << curr->coef << "x^" << curr->power << '\n';

}

**Main.cpp**

int main()

{

PolyNode\* p1 = nullptr;

PolyNode\* p2 = nullptr;

PolyNode poly\_list;

poly\_list.read(p1);

std::cout << "polynomial 1 has been read\n";

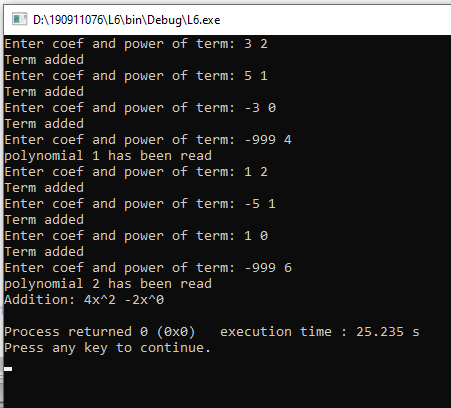
poly\_list.read(p2);

std::cout << "polynomial 2 has been read\n";

poly\_list.add(p1, p2);

}

}



**L7R1:**

**BinaryTree.hpp and cpp contain binary tree implementation and main.cpp uses them.**

**We also need a Stack of Node Pointers for some functions which is implemented in Stack.hpp and cpp.**

**Stack.hpp**

class NodePointerStack

{

private:

int top = -1;

Node\* arr[MAX];

public:

NodePointerStack();

void push(Node\* n);

Node\* pop();

Node\* peek() const;

bool empty() const;

void display();

};

**Stack.cpp**

NodePointerStack::NodePointerStack()

{

}

void NodePointerStack::push(Node\* n)

{

if (top == MAX - 1) {

return;

}

arr[++top] = n;

}

Node\* NodePointerStack::pop()

{

if (top == -1) {

std::cout << "Empty!\n";

return 0;

}

return arr[top--];

}

Node\* NodePointerStack::peek() const

{

if (empty()) {

return 0;

}

return arr[top];

}

bool NodePointerStack::empty() const

{

if (top == -1) {

return true;

}

else return false;

}

void NodePointerStack::display()

{

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

std::cout << arr[i]->get\_data() << " ";

}

std::cout << "\n";

}

**BinaryTree.hpp**

#pragma once

class Node {

private:

int data = 0;

Node\* lc = nullptr;

Node\* rc = nullptr;

public:

Node();

Node(int data);

void create\_BT(Node\*& root, int data, char const path[]);

void inorder\_iter(Node\*& root);

void postorder\_iter(Node\*& root);

void preorder\_iter(Node\*& root);

int count\_leaf\_nodes(Node\*& root);

int get\_data() { return data; }

};

**BinaryTree.cpp**

#include <iostream>

#include <cstring>

#include "Stack.hpp"

#include "BinaryTree.hpp"

Node::Node() {}

Node::Node(int data) : data(data) {}

void Node::create\_BT(Node\*& root, int data, char const path[]) {

if (!root) {

root = new Node(data);

return;

}

Node\* prev = nullptr;

Node\* curr = root;

int i = 0;

for (i = 0; i < strlen(path); i++) {

if (!curr) {

break;

}

prev = curr;

if (path[i] == 'L') {

curr = curr->lc;

}

else if (path[i] == 'R') {

curr = curr->rc;

}

}

if (curr != nullptr || i != strlen(path)) {

std::cout << "Impossible\n";

return;

}

Node\* temp = new Node(data);

if (path[i - 1] == 'L') {

prev->lc = temp;

}

else if (path[i - 1] == 'R') {

prev->rc = temp;

}

return;

}

void Node::inorder\_iter(Node\*& root) {

if (!root) {

return;

}

NodePointerStack s;

Node\* curr = root;

while (curr || !s.empty()) {

while (curr) {

s.push(curr);

curr = curr->lc;

}

curr = s.pop();

std::cout << curr->data << " ";

curr = curr->rc;

}

std::cout << std::endl;

}

void Node::postorder\_iter(Node\*& root) {

if (!root) {

return;

}

NodePointerStack s;

Node\* curr = root;

do

{

while (curr)

{

if (curr->rc) {

s.push(curr->rc);

}

s.push(curr);

curr = curr->lc;

}

curr = s.pop();

if (curr->rc && s.peek() == curr->rc)

{

s.pop();

s.push(curr);

curr = curr->rc;

}

else

{

std::cout << curr->data << " ";

curr = nullptr;

}

} while (!s.empty());

std::cout << std::endl;

}

void Node::preorder\_iter(Node\*& root) {

if (root == NULL)

return;

Node\* curr = root;

NodePointerStack nodeStack;

nodeStack.push(curr);

while (!nodeStack.empty()) {

Node\* node = nodeStack.pop();

std::cout << node->data << " ";

if (node->rc)

nodeStack.push(node->rc);

if (node->lc)

nodeStack.push(node->lc);

}

std::cout << std::endl;

}

int Node::count\_leaf\_nodes(Node\*& root) {

if(!root) {

return 0;

}

Node\* node = root;

if(node->lc == nullptr && node->rc == nullptr) {

return 1;

}

else {

return count\_leaf\_nodes(node->lc) + count\_leaf\_nodes(node->rc);

}

}

**Main.cpp**

#include <iostream>

#include "BinaryTree.hpp"

int main()

{

Node n;

Node\* root = nullptr;

n.create\_BT(root, 5, "L");

n.create\_BT(root, 10, "L");

n.create\_BT(root, 15, "LL");

n.create\_BT(root, 2, "R");

n.create\_BT(root, 7, "LR");

n.create\_BT(root, 45, "RR");

n.create\_BT(root, 789, "LLR");

std::cout << "Inorder:\n";

n.inorder\_iter(root);

std::cout << "Postorder:\n";

n.postorder\_iter(root);

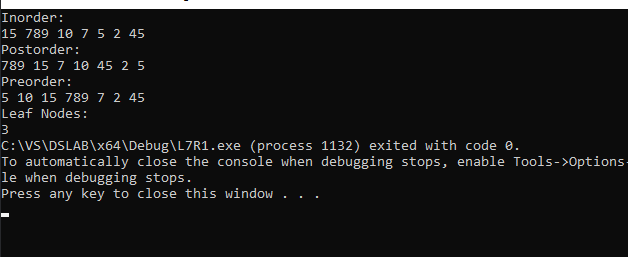
std::cout << "Preorder:\n";

n.preorder\_iter(root);

std::cout << "Leaf Nodes:\n";

std::cout << n.count\_leaf\_nodes(root);

}



**L7R2R3R4:**

**The 3 questions have been done together.**

**Node.hpp**

#pragma once

class NodeBST

{

private:

int data = 0;

NodeBST\* lc = nullptr;

NodeBST\* rc = nullptr;

public:

NodeBST();

NodeBST(int data);

void recursive\_inorder(NodeBST\* root);

void insert\_element(NodeBST\*& root, int data);

void delete\_element(NodeBST\*& root, int data);

void search\_element(NodeBST\*& root, int data);

};

**Node.cpp**

#include <iostream>

#include "NodeBST.hpp"

NodeBST::NodeBST()

{

}

NodeBST::NodeBST(int data) : data(data)

{

}

void NodeBST::recursive\_inorder(NodeBST\* root)

{

if (root) {

recursive\_inorder(root->lc);

std::cout << root->data << " ";

recursive\_inorder(root->rc);

}

}

void NodeBST::insert\_element(NodeBST\*& root, int data)

{

if (!root) {

root = new NodeBST(data);

return;

}

NodeBST\* prev = nullptr;

NodeBST\* curr = root;

while (curr) {

prev = curr;

if (data < curr->data) {

curr = curr->lc;

}

else curr = curr->rc;

}

if (data < prev->data) {

prev->lc = new NodeBST(data);

}

else prev->rc = new NodeBST(data);

}

void NodeBST::delete\_element(NodeBST\*& root, int data)

{

if (!root) {

std::cout << "Empty tree\n";

return;

}

NodeBST\* prev = nullptr;

NodeBST\* curr = root;

while (curr && curr->data != data) {

prev = curr;

if (data < curr->data)

curr = curr->lc;

else

curr = curr->rc;

}

if (curr == nullptr) {

std::cout << "Key " << data << " not found\n";

return;

}

if (curr->lc == nullptr || curr->rc == nullptr) {

NodeBST\* newCurr = nullptr;

if (curr->lc == nullptr)

newCurr = curr->rc;

else

newCurr = curr->lc;

if (prev == nullptr) {

return;

}

if (curr == prev->lc)

prev->lc = newCurr;

else

prev->rc = newCurr;

delete curr;

}

else {

NodeBST\* par = nullptr;

NodeBST\* temp = nullptr;

temp = curr->rc;

while (temp->lc) {

par = temp;

temp = temp->lc;

}

if (par)

par->lc = temp->rc;

else

curr->rc = temp->rc;

curr->data = temp->data;

delete temp;

}

}

void NodeBST::search\_element(NodeBST\*& root, int data)

{

if (!root) {

std::cout << "List is empty\n";

return;

}

NodeBST\* curr = root;

while (curr) {

if (curr->data == data) {

std::cout << "Element found\n";

return;

}

if (data < curr->data) {

curr = curr->lc;

}

else curr = curr->rc;

}

std::cout << "Element not found\n";

}

**Main.cpp**

#include <iostream>

#include "NodeBST.hpp"

int main()

{

NodeBST\* root = nullptr;

NodeBST n;

n.insert\_element(root, 5);

n.insert\_element(root, 10);

n.insert\_element(root, 30);

n.insert\_element(root, 3);

n.insert\_element(root, 34);

n.insert\_element(root, 78);

n.insert\_element(root, 9);

n.recursive\_inorder(root);

std::cout << std::endl;

n.delete\_element(root, 30);

n.delete\_element(root, 5);

n.recursive\_inorder(root);

std::cout << std::endl;

n.search\_element(root, 5);

n.search\_element(root, 10);

}

