**Final Assingment**

**Double link list**

**Program:**

#include <iostream>

using namespace std;

struct Node {

int data;

Node\* next;

Node\* prev;

Node(int value) : data(value), next(nullptr), prev(nullptr) {}

};

class DoublyLinkedList {

private:

Node\* head;

public:

DoublyLinkedList() : head(nullptr) {}

void append(int value) {

Node\* newNode = new Node(value);

if (!head) {

head = newNode;

return;

}

Node\* temp = head;

while (temp->next) {

temp = temp->next;

}

temp->next = newNode;

newNode->prev = temp;

}

void deleteFirstNode() {

if (head == nullptr) {

cout << "List is empty, no node to delete." << endl;

return;

}

Node\* temp = head;

head = head->next;

if (head != nullptr) {

head->prev = nullptr;

}

delete temp;

cout << "First node deleted." << endl;

}

void display() {

if (!head) {

cout << "List is empty." << endl;

return;

}

Node\* temp = head;

while (temp) {

cout << temp->data << " ";

temp = temp->next;

}

cout << endl;

}

~DoublyLinkedList() {

while (head) {

deleteFirstNode();

}

}

};

int main() {

DoublyLinkedList list;

list.append(10);

list.append(20);

list.append(30);

list.append(40);

cout << "Original List: ";

list.display();

list.deleteFirstNode();

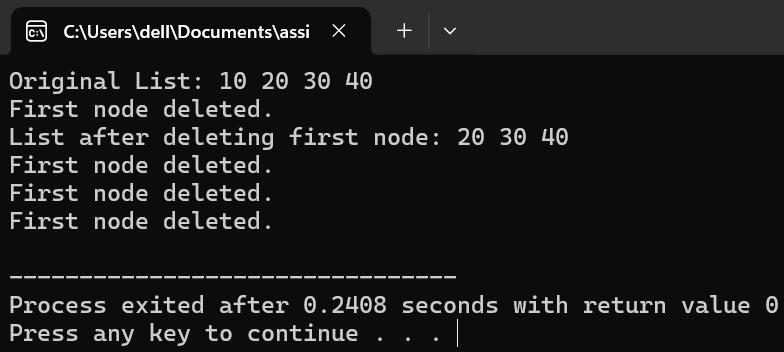
cout << "List after deleting first node: ";

list.display();

return 0;

}

**Output:**

****

**Program #03:**

#include <iostream>

struct Node {

int data;

Node\* next;

Node\* prev;

Node(int value) : data(value), next(nullptr), prev(nullptr) {}

};

class DoublyLinkedList {

public:

Node\* head;

Node\* tail;

DoublyLinkedList() : head(nullptr), tail(nullptr) {}

void insertAtEnd(int value) {

Node\* newNode = new Node(value);

if (!head) { // List is empty

head = tail = newNode;

} else {

tail->next = newNode;

newNode->prev = tail;

tail = newNode;

}

}

void deleteLastNode() {

if (!tail) { // List is empty

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

return;

}

if (head == tail) {

delete tail;

head = tail = nullptr;

} else {

Node\* temp = tail;

tail = tail->prev;

tail->next = nullptr;

delete temp;

}

}

void display() {

Node\* temp = head;

while (temp) {

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

temp = temp->next;

}

std::cout << std::endl;

}

};

int main() {

DoublyLinkedList list;

list.insertAtEnd(10);

list.insertAtEnd(20);

list.insertAtEnd(30);

list.insertAtEnd(40);

std::cout << "Original List: ";

list.display();

list.deleteLastNode();

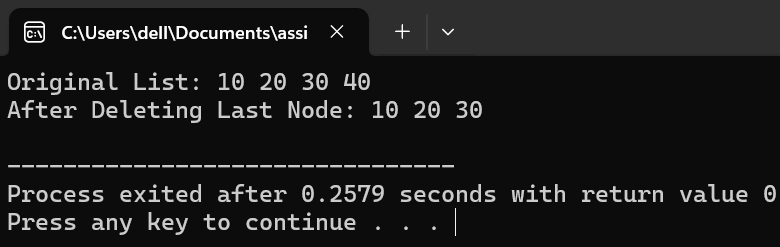
std::cout << "After Deleting Last Node: ";

list.display();

return 0;

}

**Output:**

****

**Program #03:**

#include <iostream>

struct Node {

int data;

Node\* next;

Node\* prev;

Node(int value) : data(value), next(nullptr), prev(nullptr) {}

};

class DoublyLinkedList {

public:

Node\* head;

Node\* tail;

DoublyLinkedList() : head(nullptr), tail(nullptr) {}

void insertAtEnd(int value) {

Node\* newNode = new Node(value);

if (!head) {

head = tail = newNode;

} else {

tail->next = newNode;

newNode->prev = tail;

tail = newNode;

}

}

void deleteByValue(int value) {

if (!head) {

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

return;

}

Node\* temp = head;

while (temp) {

if (temp->data == value) {

if (temp == head) {

head = head->next;

if (head) {

head->prev = nullptr;

} else {

tail = nullptr;

}

}

else if (temp == tail) {

tail = tail->prev;

tail->next = nullptr;

}

else {

temp->prev->next = temp->next;

temp->next->prev = temp->prev;

}

delete temp;

std::cout << "Node with value " << value << " deleted.\n";

return;

}

temp = temp->next;

}

std::cout << "Node with value " << value << " not found.\n";

}

void display() {

Node\* temp = head;

while (temp) {

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

temp = temp->next;

}

std::cout << std::endl;

}

};

int main() {

DoublyLinkedList list;

list.insertAtEnd(10);

list.insertAtEnd(20);

list.insertAtEnd(30);

list.insertAtEnd(40);

std::cout << "Original List: ";

list.display();

list.deleteByValue(20);

std::cout << "After Deleting Node with value 20: ";

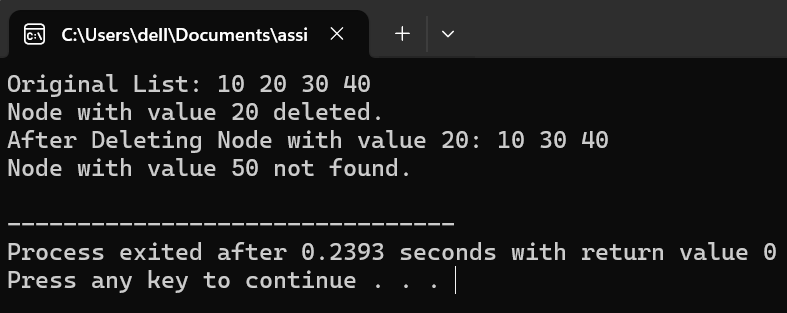
list.display();

list.deleteByValue(50);

return 0;

}

**Output:**

****

**Program #04:**

#include <iostream>

using namespace std;

struct Node {

int data;

Node\* next;

Node\* prev;

Node(int value) : data(value), next(nullptr), prev(nullptr) {}

};

class DoublyLinkedList {

public:

Node\* head;

Node\* tail;

DoublyLinkedList() : head(nullptr), tail(nullptr) {}

void insertAtEnd(int value) {

Node\* newNode = new Node(value);

if (!head) {

head = tail = newNode;

} else {

tail->next = newNode;

newNode->prev = tail;

tail = newNode;

}

}

void deleteAtPosition(int position) {

if (!head) {

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

return;

}

Node\* temp = head;

int currentPosition = 0;

while (temp && currentPosition < position) {

temp = temp->next;

currentPosition++;

}

if (!temp) {

cout << "Position out of range.\n";

return;

}

if (temp == head) {

head = head->next;

if (head) {

head->prev = nullptr;

} else {

tail = nullptr;

}

}

else if (temp == tail) {

tail = tail->prev;

tail->next = nullptr;

}

else {

temp->prev->next = temp->next;

temp->next->prev = temp->prev;

}

delete temp;

cout << "Node at position " << position << " deleted.\n";

}

void display() {

Node\* temp = head;

while (temp) {

cout << temp->data << " ";

temp = temp->next;

}

cout << endl;

}

};

int main() {

DoublyLinkedList list;

list.insertAtEnd(10);

list.insertAtEnd(20);

list.insertAtEnd(30);

list.insertAtEnd(40);

cout << "Original List: ";

list.display();

list.deleteAtPosition(2);

cout << "After Deleting Node at Position 2: ";

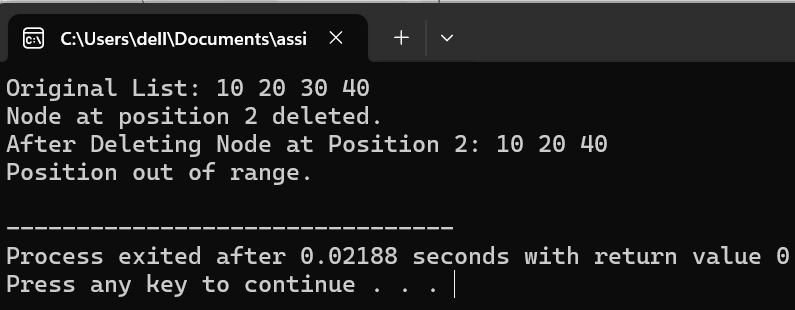
list.display();

list.deleteAtPosition(5);

return 0;

}

**Output:**

****

**Program #05:**

#include <iostream>

using namespace std:

struct Node {

int data;

Node\* next;

Node\* prev;

Node(int value) : data(value), next(nullptr), prev(nullptr) {}

};

class DoublyLinkedList {

public:

Node\* head;

Node\* tail;

DoublyLinkedList() : head(nullptr), tail(nullptr) {}

void insertAtEnd(int value) {

Node\* newNode = new Node(value);

if (!head) {

head = tail = newNode;

} else {

tail->next = newNode;

newNode->prev = tail;

tail = newNode;

}

}

void deleteAtPosition(int position) {

if (!head) {

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

return;

}

Node\* temp = head;

int currentPosition = 0;

while (temp && currentPosition < position) {

temp = temp->next;

currentPosition++;

}

if (!temp) {

cout << "Position out of range.\n";

return;

}

if (temp == head) {

head = head->next;

if (head) {

head->prev = nullptr;

} else {

tail = nullptr;

}

}

else if (temp == tail) {

tail = tail->prev;

tail->next = nullptr;

}

else {

temp->prev->next = temp->next;

temp->next->prev = temp->prev;

}

delete temp; // Free the memory

cout << "Node at position " << position << " deleted.\n";

}

void forwardTraversal() {

if (!head) {

cout << "List is empty.\n";

return;

}

Node\* temp = head;

cout << "Forward Traversal: ";

while (temp) {

cout << temp->data << " ";

temp = temp->next;

}

cout << endl;

}

void reverseTraversal() {

if (!tail) {

cout << "List is empty.\n";

return;

}

Node\* temp = tail;

cout << "Reverse Traversal: ";

while (temp) {

cout << temp->data << " ";

temp = temp->prev;

}

cout << std::endl;

}

};

int main() {

DoublyLinkedList list;

list.insertAtEnd(10);

list.insertAtEnd(20);

list.insertAtEnd(30);

list.insertAtEnd(40);

cout << "Original List: ";

list.forwardTraversal();

list.deleteAtPosition(2);

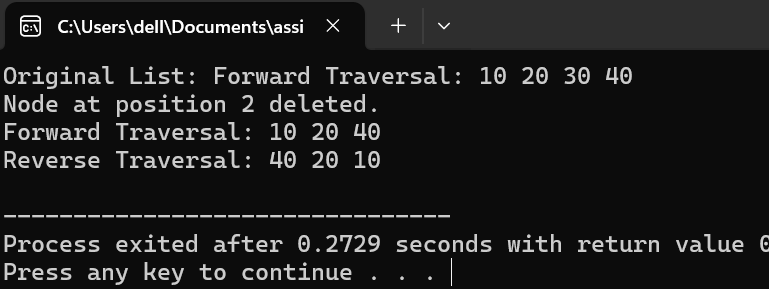
list.forwardTraversal();

list.reverseTraversal();

return 0;

}

**Output:**

****

**Circular Linked List**

**Program #01:**

#include <iostream>

using namespace std;

struct Node {

int data;

Node\* next;

Node(int value) : data(value), next(nullptr) {}

};

class CircularLinkedList {

public:

Node\* head;

CircularLinkedList() : head(nullptr) {}

void insertAtEnd(int value) {

Node\* newNode = new Node(value);

if (!head) {

head = newNode;

newNode->next = head;

} else {

Node\* temp = head;

while (temp->next != head) {

temp = temp->next;

}

temp->next = newNode;

newNode->next = head;

}

}

void deleteFirstNode() {

if (!head) {

cout << "List is empty. Nothing to delete.\n";

return;

}

if (head->next == head) {

delete head;

head = nullptr;

} else {

Node\* temp = head;

while (temp->next != head) {

temp = temp->next;

}

temp->next = head->next;

Node\* nodeToDelete = head;

head = head->next;

delete nodeToDelete;

}

cout << "First node deleted.\n";

}

void display() {

if (!head) {

cout << "List is empty.\n";

return;

}

Node\* temp = head;

do {

cout << temp->data << " ";

temp = temp->next;

} while (temp != head);

cout <<endl;

}

};

int main() {

CircularLinkedList list;

list.insertAtEnd(10);

list.insertAtEnd(20);

list.insertAtEnd(30);

list.insertAtEnd(40);

cout << "Original Circular Linked List: ";

list.display();

list.deleteFirstNode();

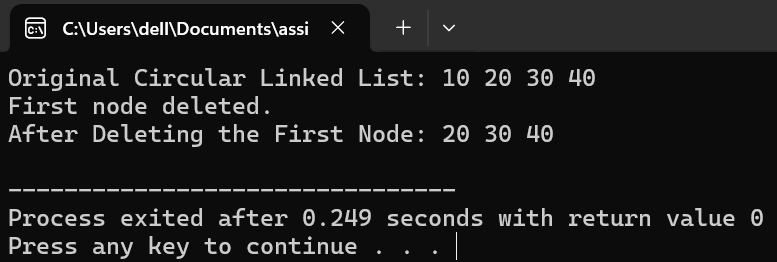
cout << "After Deleting the First Node: ";

list.display();

return 0;

}

**Output:**

****

**Program #02:**

#include <iostream>

using namespace std;

struct Node {

int data;

Node\* next;

Node(int value) : data(value), next(nullptr) {}

};

class CircularLinkedList {

public:

Node\* head;

CircularLinkedList() : head(nullptr) {}

void insertAtEnd(int value) {

Node\* newNode = new Node(value);

if (!head) {

head = newNode;

newNode->next = head;

} else {

Node\* temp = head;

while (temp->next != head) {

temp = temp->next;

}

temp->next = newNode;

newNode->next = head;

}

}

void deleteLastNode() {

if (!head) {

cout << "List is empty. Nothing to delete.\n";

return;

}

if (head->next == head) {

delete head;

head = nullptr;

} else {

Node\* temp = head;

while (temp->next->next != head) {

temp = temp->next;

}

Node\* lastNode = temp->next;

temp->next = head;

delete lastNode;

}

cout << "Last node deleted.\n";

}

void display() {

if (!head) {

cout << "List is empty.\n";

return;

}

Node\* temp = head;

do {

cout << temp->data << " ";

temp = temp->next;

} while (temp != head);

out << endl;

}

};

int main() {

CircularLinkedList list;

list.insertAtEnd(10);

list.insertAtEnd(20);

list.insertAtEnd(30);

list.insertAtEnd(40);

cout << "Original Circular Linked List: ";

list.display();

list.deleteLastNode();

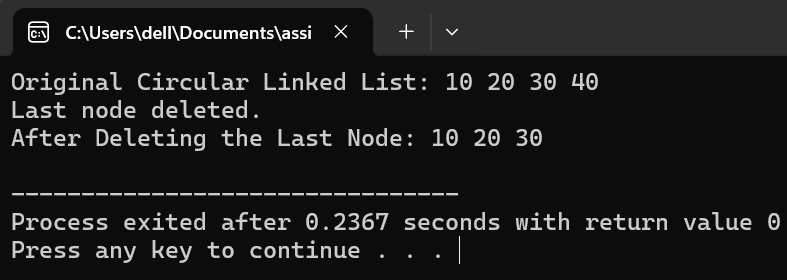
cout << "After Deleting the Last Node: ";

list.display();

return 0;

}

**Output:**

****

**Program #03:**

#include <iostream>

using namespace std;

struct Node {

int data;

Node\* next;

Node(int value) : data(value), next(nullptr) {}

};

class CircularLinkedList {

public:

Node\* head;

CircularLinkedList() : head(nullptr) {}

void insertAtEnd(int value) {

Node\* newNode = new Node(value);

if (!head) {

head = newNode;

newNode->next = head;

} else {

Node\* temp = head;

while (temp->next != head) {

temp = temp->next;

}

temp->next = newNode;

newNode->next = head;

}

}

void deleteByValue(int value) {

if (!head) { // List is empty

cout << "List is empty. Nothing to delete.\n";

return;

}

Node\* temp = head;

Node\* prev = nullptr;

do {

if (temp->data == value) {

if (temp == head) {

if (head->next == head) {

delete head;

head = nullptr;

} else {

Node\* last = head;

while (last->next != head) {

last = last->next;

}

head = head->next;

last->next = head;

delete temp;

}

}

else {

prev->next = temp->next;

delete temp;

}

cout << "Node with value " << value << " deleted.\n";

return;

}

prev = temp;

temp = temp->next;

} while (temp != head);

cout << "Node with value " << value << " not found.\n";

}

void display() {

if (!head) {

cout << "List is empty.\n";

return;

}

Node\* temp = head;

do {

cout << temp->data << " ";

temp = temp->next;

} while (temp != head);

cout << endl;

}

};

int main() {

CircularLinkedList list;

list.insertAtEnd(10);

list.insertAtEnd(20);

list.insertAtEnd(30);

list.insertAtEnd(40);

cout << "Original Circular Linked List: ";

list.display();

list.deleteByValue(20);

cout << "After Deleting Node with value 20: ";

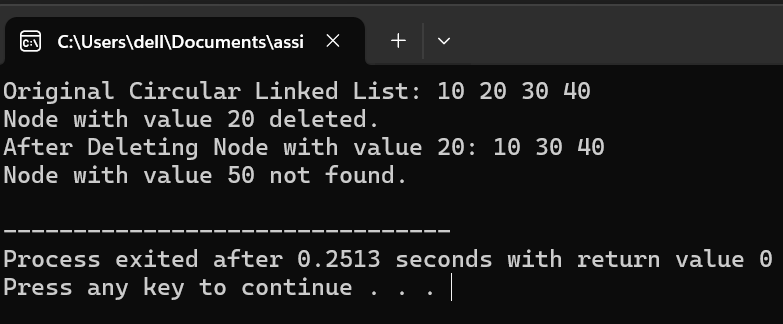
list.display();

list.deleteByValue(50);

return 0;

}

**Output:**



**Programe #04:**

#include <iostream>

using namespace std;

struct Node {

int data;

Node\* next;

};

void insertNode(Node\*& head, int data) {

Node\* newNode = new Node{data, nullptr};

if (!head) {

head = newNode;

head->next = head;

} else {

Node\* temp = head;

while (temp->next != head) {

temp = temp->next;

}

temp->next = newNode;

newNode->next = head;

}

}

void display(Node\* head) {

if (!head) {

cout << "List is empty." << endl;

return;

}

Node\* temp = head;

do {

cout << temp->data << " ";

temp = temp->next;

} while (temp != head);

cout << endl;

}

void deleteNodeAtPosition(Node\*& head, int position) {

if (!head) {

cout << "List is empty. Cannot delete node." << endl;

return;

}

int count = 1;

Node\* current = head;

Node\* prev = nullptr;

if (position == 1) {

if (head->next == head) {

delete head;

head = nullptr;

return;

}

while (current->next != head) {

current = current->next;

}

Node\* temp = head;

head = head->next;

current->next = head;

delete temp;

return;

}

current = head;

for (int i = 1; i < position; ++i) {

prev = current;

current = current->next;

if (current == head) {

cout << "Invalid position." << endl;

return;

}

}

prev->next = current->next;

delete current;

}

int main() {

Node\* head = nullptr;

insertNode(head, 10);

insertNode(head, 20);

insertNode(head, 30);

insertNode(head, 40);

cout << "Original List: ";

display(head);

deleteNodeAtPosition(head, 2);

cout << "After deleting node at position 2: ";

display(head);

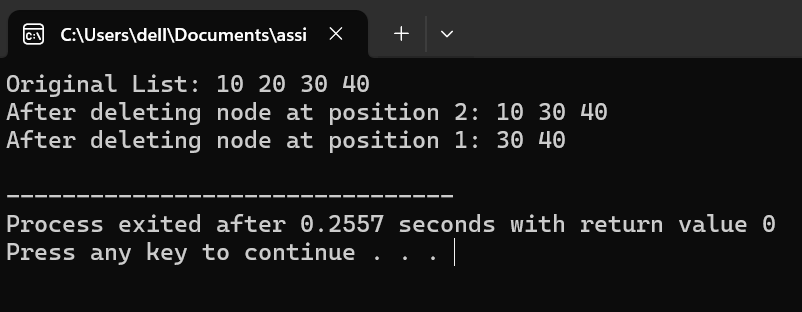
deleteNodeAtPosition(head, 1);

cout << "After deleting node at position 1: ";

display(head);

return 0;

**output:**



**Binary Search Tree**

**Program #01:**

#include <iostream>

using namespace std;

// Node structure for Binary Search Tree

struct Node {

int data;

Node\* left;

Node\* right;

Node(int value) {

data = value;

left = right = nullptr;

}

};

// Function to insert a node in the Binary Search Tree

Node\* insert(Node\* root, int value) {

if (root == nullptr) {

return new Node(value);

}

if (value < root->data) {

root->left = insert(root->left, value);

} else if (value > root->data) {

root->right = insert(root->right, value);

}

return root;

}

// Function to count the nodes in the Binary Search Tree

int countNodes(Node\* root) {

if (root == nullptr) {

return 0;

}

// Recursively count nodes in the left and right subtrees, and add 1 for the current node

return 1 + countNodes(root->left) + countNodes(root->right);

}

// Function to perform an in-order traversal and print the tree

void inorderTraversal(Node\* root) {

if (root != nullptr) {

inorderTraversal(root->left);

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

inorderTraversal(root->right);

}

}

// Main function

int main() {

Node\* root = nullptr;

// Insert nodes into the BST

root = insert(root, 50);

root = insert(root, 30);

root = insert(root, 20);

root = insert(root, 40);

root = insert(root, 70);

root = insert(root, 60);

root = insert(root, 80);

cout << "In-order traversal of the Binary Search Tree: ";

inorderTraversal(root);

cout << endl;

// Count the nodes in the BST

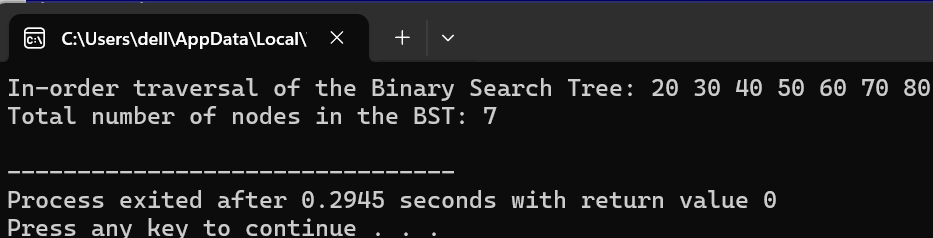
int nodeCount = countNodes(root);

cout << "Total number of nodes in the BST: " << nodeCount << endl;

return 0;

}

**Output:**

****

**Program #02:**

#include <iostream>

using namespace std;

// Node structure

struct Node {

int data;

Node\* left;

Node\* right;

Node(int val) {

data = val;

left = NULL;

right = NULL;

}

};

// Function to find the minimum value node

Node\* findMin(Node\* root) {

while (root && root->left != NULL) {

root = root->left;

}

return root;

}

// Function to delete a node

Node\* deleteNode(Node\* root, int key) {

if (root == NULL) return root;

if (key < root->data) {

root->left = deleteNode(root->left, key);

} else if (key > root->data) {

root->right = deleteNode(root->right, key);

} else {

if (root->left == NULL) {

Node\* temp = root->right;

delete root;

return temp;

} else if (root->right == NULL) {

Node\* temp = root->left;

delete root;

return temp;

}

Node\* temp = findMin(root->right);

root->data = temp->data;

root->right = deleteNode(root->right, temp->data);

}

return root;

}

// Main function to test deletion

int main() {

Node\* root = new Node(10);

root->left = new Node(5);

root->right = new Node(15);

root->left->left = new Node(3);

root->left->right = new Node(7);

root->right->left = new Node(12);

root->right->right = new Node(18);

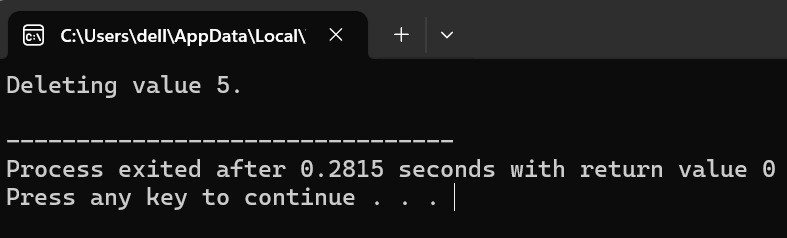
cout << "Deleting value 5." << endl;

root = deleteNode(root, 5);

return 0;

}

**Output:**

****

**Program#03:**

#include <iostream>

using namespace std;

// Node structure

struct Node {

int data;

Node\* left;

Node\* right;

Node(int val) {

data = val;

left = NULL;

right = NULL;

}

};

// Function to insert a value (with duplicate check)

Node\* insert(Node\* root, int val) {

if (root == NULL) {

return new Node(val);

}

if (val < root->data) {

root->left = insert(root->left, val);

} else if (val > root->data) {

root->right = insert(root->right, val);

} else {

cout << "Duplicate value " << val << " not allowed." << endl;

}

return root;

}

// Main function to test duplication handling

int main() {

Node\* root = NULL;

root = insert(root, 10);

root = insert(root, 5);

root = insert(root, 15);

root = insert(root, 3);

root = insert(root, 7);

root = insert(root, 12);

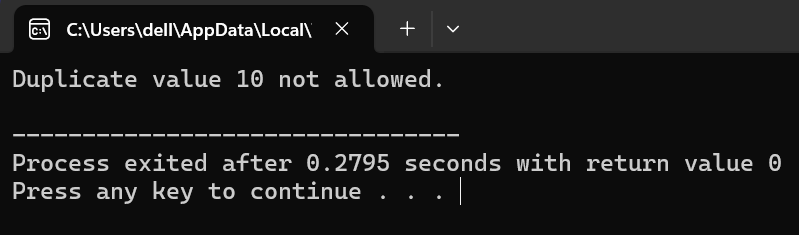
root = insert(root, 10); // Duplicate Values

root = insert(root, 18);

return 0;

}

**Output:**

****

**Program #04:**

#include <iostream>

using namespace std;

// Node structure

struct Node {

int data;

Node\* left;

Node\* right;

Node(int val) {

data = val;

left = NULL;

right = NULL;

}

};

// Function to search for a value

bool search(Node\* root, int key) {

if (root == NULL) return false;

if (root->data == key) return true;

if (key < root->data) return search(root->left, key);

return search(root->right, key);

}

// Main function to test searching

int main() {

Node\* root = new Node(10);

root->left = new Node(5);

root->right = new Node(15);

int searchKey = 5;

if (search(root, searchKey)) {

cout << "Value " << searchKey << " found in the tree." << endl;

} else {

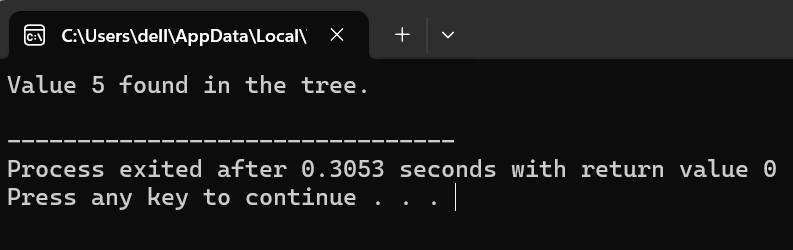
cout << "Value " << searchKey << " not found in the tree." << endl;

}

return 0;

}

**Output:**

****

**Program:**

#include <iostream>

using namespace std;

// Node structure

struct Node {

int data;

Node\* left;

Node\* right;

Node(int val) {

data = val;

left = NULL;

right = NULL;

}

};

// Traversal functions

void inorder(Node\* root) {

if (root == NULL) return;

inorder(root->left);

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

inorder(root->right);

}

void preorder(Node\* root) {

if (root == NULL) return;

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

preorder(root->left);

preorder(root->right);

}

void postorder(Node\* root) {

if (root == NULL) return;

postorder(root->left);

postorder(root->right);

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

}

// Main function to test traversals

int main() {

Node\* root = new Node(10);

root->left = new Node(5);

root->right = new Node(15);

root->left->left = new Node(3);

root->left->right = new Node(7);

root->right->left = new Node(12);

root->right->right = new Node(18);

cout << "Inorder: ";

inorder(root);

cout << endl;

cout << "Preorder: ";

preorder(root);

cout << endl;

cout << "Postorder: ";

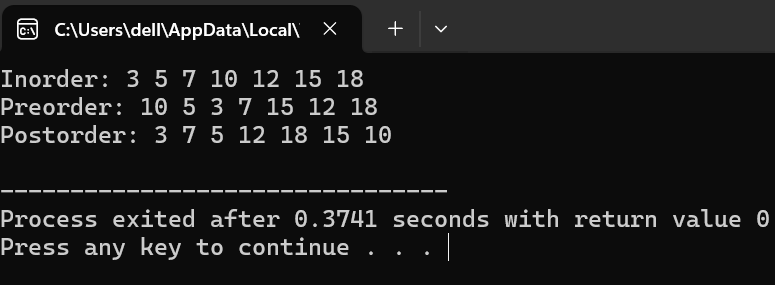
postorder(root);

cout << endl;

return 0;

}

**Output:**

****