**QANDEEL ASIM**

**BS AI 037**

**DATA STRUCTURE**

**FINAL ASSIGNMENT**

**Doubly Linked List**

**• Write a program to delete the first node in a doubly linked list.**

#include <iostream>

using namespace std;

struct Node {

int data;

Node\* next;

Node\* prev;

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

};

void deleteFirstNode(Node\*& head) {

if (head == nullptr) {

cout << "List is empty. Nothing to delete." << endl;

return;

}

Node\* temp = head;

head = head->next;

if (head != nullptr) {

head->prev = nullptr;

}

delete temp;

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

}

void printList(Node\* head) {

Node\* temp = head;

while (temp != nullptr) {

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

temp = temp->next;

}

cout << endl;

}

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

Node\* newNode = new Node(data);

if (head == nullptr) {

head = newNode;

return;

}

Node\* temp = head;

while (temp->next != nullptr) {

temp = temp->next;

}

temp->next = newNode;

newNode->prev = temp;

}

int main() {

Node\* head = nullptr;

appendNode(head, 10);

appendNode(head, 20);

appendNode(head, 30);

cout << "Original List: ";

printList(head);

deleteFirstNode(head);

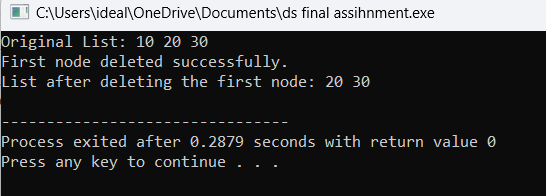
cout << "List after deleting the first node: ";

printList(head);

return 0;

}

**Output:**



• **How can you delete the last node in a doubly linked list? Write the code.**

#include <iostream>

using namespace std;

struct Node {

int data;

Node\* next;

Node\* prev;

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

};

void deleteLastNode(Node\*& head) {

if (head == nullptr) {

cout << "List is empty. Nothing to delete." << endl;

return;

}

if (head->next == nullptr) {

delete head;

head = nullptr;

cout << "Last node deleted successfully." << endl;

return;

}

Node\* temp = head;

while (temp->next != nullptr) {

temp = temp->next;

}

temp->prev->next = nullptr;

delete temp;

cout << "Last node deleted successfully." << endl;

}

void printList(Node\* head) {

Node\* temp = head;

while (temp != nullptr) {

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

temp = temp->next;

}

cout << endl;

}

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

Node\* newNode = new Node(data);

if (head == nullptr) {

head = newNode;

return;

}

Node\* temp = head;

while (temp->next != nullptr) {

temp = temp->next;

}

temp->next = newNode;

newNode->prev = temp;

}

int main() {

Node\* head = nullptr;

appendNode(head, 10);

appendNode(head, 20);

appendNode(head, 30);

cout << "Original List: ";

printList(head);

deleteLastNode(head);

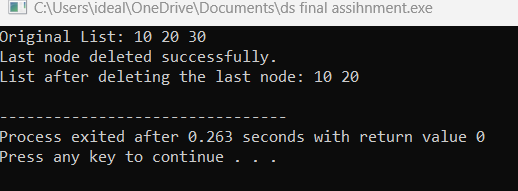
cout << "List after deleting the last node: ";

printList(head);

return 0;

}

Output:



**Write code to delete a node by its value in a doubly linked list.**

#include <iostream>

using namespace std;

struct Node {

int data;

Node\* next;

Node\* prev;

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

};

void deleteNodeByValue(Node\*& head, int value) {

if (head == nullptr) {

cout << "List is empty. Nothing to delete." << endl;

return;

}

Node\* temp = head;

while (temp != nullptr && temp->data != value) {

temp = temp->next;

}

if (temp == nullptr) {

cout << "Value " << value << " not found in the list." << endl;

return;

}

if (temp == head) {

head = head->next;

if (head != nullptr) {

head->prev = nullptr;

}

}

else {

if (temp->next != nullptr) {

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

}

if (temp->prev != nullptr) {

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

}

}

delete temp;

cout << "Node with value " << value << " deleted successfully." << endl;

}

void printList(Node\* head) {

Node\* temp = head;

while (temp != nullptr) {

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

temp = temp->next;

}

cout << endl;

}

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

Node\* newNode = new Node(data);

if (head == nullptr) {

head = newNode;

return;

}

Node\* temp = head;

while (temp->next != nullptr) {

temp = temp->next;

}

temp->next = newNode;

newNode->prev = temp;

}

int main() {

Node\* head = nullptr;

appendNode(head, 10);

appendNode(head, 20);

appendNode(head, 30);

appendNode(head, 40);

cout << "Original List: ";

printList(head);

deleteNodeByValue(head, 20);

cout << "List after deleting node with value 20: ";

printList(head);

deleteNodeByValue(head, 50);

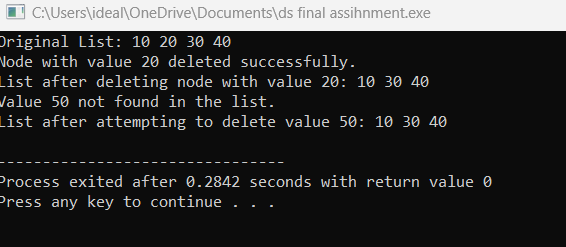
cout << "List after attempting to delete value 50: ";

printList(head);

return 0;

}

Output:



**How would you delete a node at a specific position in a doubly linked list? Show it in code.**

#include <iostream>

using namespace std;

// Node structure for the doubly linked list

struct Node {

int data;

Node\* next;

Node\* prev;

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

};

// Function to delete a node at a specific position

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

if (head == nullptr) {

cout << "List is empty. Nothing to delete." << endl;

return;

}

Node\* temp = head;

// If the position is 0, delete the head node

if (position == 0) {

head = head->next;

if (head != nullptr) {

head->prev = nullptr;

}

delete temp;

cout << "Node at position " << position << " deleted successfully." << endl;

return;

}

// Traverse to the node at the given position

for (int i = 0; i < position && temp != nullptr; i++) {

temp = temp->next;

}

// If the position is invalid (greater than the length of the list)

if (temp == nullptr) {

cout << "Position " << position << " is out of bounds." << endl;

return;

}

// Update pointers to bypass the node

if (temp->next != nullptr) {

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

}

if (temp->prev != nullptr) {

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

}

// Delete the node

delete temp;

cout << "Node at position " << position << " deleted successfully." << endl;

}

// Function to print the doubly linked list

void printList(Node\* head) {

Node\* temp = head;

while (temp != nullptr) {

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

temp = temp->next;

}

cout << endl;

}

// Helper function to add a new node at the end of the list

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

Node\* newNode = new Node(data);

if (head == nullptr) {

head = newNode;

return;

}

Node\* temp = head;

while (temp->next != nullptr) {

temp = temp->next;

}

temp->next = newNode;

newNode->prev = temp;

}

int main() {

Node\* head = nullptr;

// Adding some nodes to the list

appendNode(head, 10);

appendNode(head, 20);

appendNode(head, 30);

appendNode(head, 40);

cout << "Original List: ";

printList(head);

// Deleting a node at position 2

deleteNodeAtPosition(head, 2);

cout << "List after deleting node at position 2: ";

printList(head);

// Deleting the head node (position 0)

deleteNodeAtPosition(head, 0);

cout << "List after deleting node at position 0: ";

printList(head);

// Attempt to delete a node at an out-of-bounds position

deleteNodeAtPosition(head, 10);

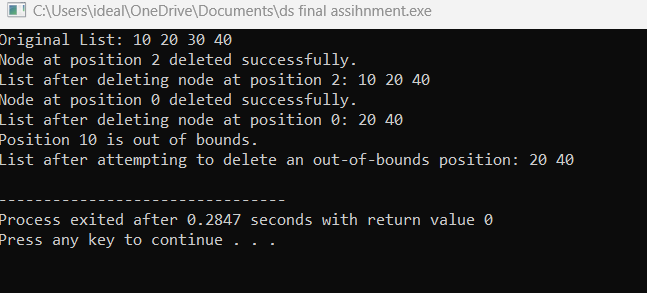
cout << "List after attempting to delete an out-of-bounds position: ";

printList(head);

return 0;

}

Output:



**After deleting a node, how will you write the forward and reverse traversal functions?**

#include <iostream>

using namespace std;

struct Node {

int data;

Node\* next;

Node\* prev;

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

};

void forwardTraversal(Node\* head) {

cout << "Forward Traversal: ";

Node\* temp = head;

while (temp != nullptr) {

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

temp = temp->next;

}

cout << endl;

}

void reverseTraversal(Node\* head) {

if (head == nullptr) {

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

return;

}

Node\* temp = head;

while (temp->next != nullptr) {

temp = temp->next;

}

cout << "Reverse Traversal: ";

// Traverse backward using the `prev` pointer

while (temp != nullptr) {

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

temp = temp->prev;

}

cout << endl;

}

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

Node\* newNode = new Node(data);

if (head == nullptr) {

head = newNode;

return;

}

Node\* temp = head;

while (temp->next != nullptr) {

temp = temp->next;

}

temp->next = newNode;

newNode->prev = temp;

}

void deleteNodeByValue(Node\*& head, int value) {

if (head == nullptr) {

cout << "List is empty. Nothing to delete." << endl;

return;

}

Node\* temp = head;

while (temp != nullptr && temp->data != value) {

temp = temp->next;

}

if (temp == nullptr) {

cout << "Value " << value << " not found in the list." << endl;

return;

}

if (temp == head) {

head = head->next;

if (head != nullptr) {

head->prev = nullptr;

}

}

else {

if (temp->next != nullptr) {

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

}

if (temp->prev != nullptr) {

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

}

}

delete temp;

cout << "Node with value " << value << " deleted successfully." << endl;

}

int main() {

Node\* head = nullptr;

appendNode(head, 10);

appendNode(head, 20);

appendNode(head, 30);

appendNode(head, 40);

forwardTraversal(head);

reverseTraversal(head);

deleteNodeByValue(head, 20);

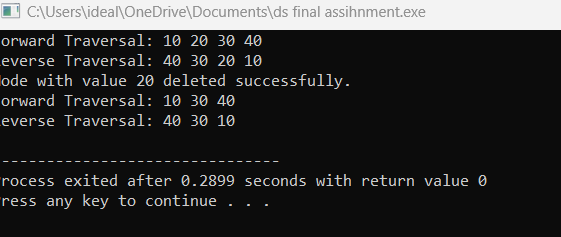
forwardTraversal(head);

reverseTraversal(head);

return 0;

}

Output:



**Circular Linked List**

**• Write a program to delete the first node in a circular linked list.**

#include <iostream>

using namespace std;

// Node structure for the circular linked list

struct Node {

int data;

Node\* next;

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

};

// Function to delete the first node in a circular linked list

void deleteFirstNode(Node\*& head) {

if (head == nullptr) {

cout << "List is empty. Nothing to delete." << endl;

return;

}

Node\* temp = head;

// If there is only one node in the list

if (head->next == head) {

delete head;

head = nullptr;

cout << "First node deleted successfully. List is now empty." << endl;

return;

}

// Find the last node

Node\* last = head;

while (last->next != head) {

last = last->next;

}

// Update the last node's next pointer to the second node

last->next = head->next;

// Update the head pointer to the second node

head = head->next;

// Delete the original first node

delete temp;

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

}

// Function to print the circular linked list

void printList(Node\* head) {

if (head == nullptr) {

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

return;

}

Node\* temp = head;

do {

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

temp = temp->next;

} while (temp != head);

cout << endl;

}

// Helper function to append a node to the circular linked list

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

Node\* newNode = new Node(data);

if (head == nullptr) {

head = newNode;

newNode->next = head; // Point to itself to make it circular

return;

}

Node\* temp = head;

while (temp->next != head) {

temp = temp->next;

}

temp->next = newNode;

newNode->next = head;

}

int main() {

Node\* head = nullptr;

// Adding nodes to the circular linked list

appendNode(head, 10);

appendNode(head, 20);

appendNode(head, 30);

appendNode(head, 40);

cout << "Original List: ";

printList(head);

// Deleting the first node

deleteFirstNode(head);

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

printList(head);

// Deleting the first node again

deleteFirstNode(head);

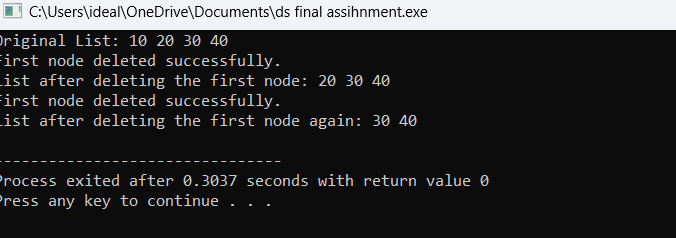
cout << "List after deleting the first node again: ";

printList(head);

return 0;

}

Output:



**How can you delete the last node in a circular linked list? Write the code.**

#include <iostream>

using namespace std;

// Node structure for the circular linked list

struct Node {

int data;

Node\* next;

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

};

// Function to delete the last node in a circular linked list

void deleteLastNode(Node\*& head) {

if (head == nullptr) {

cout << "List is empty. Nothing to delete." << endl;

return;

}

// If there is only one node in the list

if (head->next == head) {

delete head;

head = nullptr;

cout << "Last node deleted successfully. List is now empty." << endl;

return;

}

Node\* temp = head;

Node\* prev = nullptr;

// Traverse to the second-to-last node

while (temp->next != head) {

prev = temp;

temp = temp->next;

}

// Update the second-to-last node's next pointer to head

prev->next = head;

// Delete the last node

delete temp;

cout << "Last node deleted successfully." << endl;

}

// Function to print the circular linked list

void printList(Node\* head) {

if (head == nullptr) {

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

return;

}

Node\* temp = head;

do {

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

temp = temp->next;

} while (temp != head);

cout << endl;

}

// Helper function to append a node to the circular linked list

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

Node\* newNode = new Node(data);

if (head == nullptr) {

head = newNode;

newNode->next = head; // Point to itself to make it circular

return;

}

Node\* temp = head;

while (temp->next != head) {

temp = temp->next;

}

temp->next = newNode;

newNode->next = head;

}

int main() {

Node\* head = nullptr;

// Adding nodes to the circular linked list

appendNode(head, 10);

appendNode(head, 20);

appendNode(head, 30);

appendNode(head, 40);

cout << "Original List: ";

printList(head);

// Deleting the last node

deleteLastNode(head);

cout << "List after deleting the last node: ";

printList(head);

// Deleting the last node again

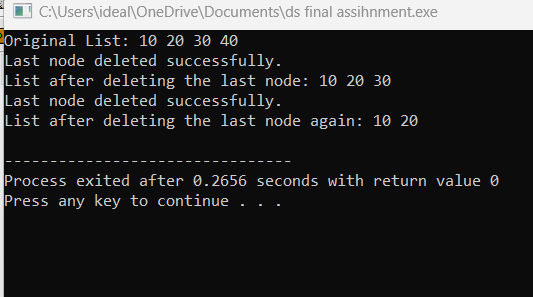
deleteLastNode(head);

cout << "List after deleting the last node again: ";

printList(head);

return 0;

}

Output:  


**Write a function to delete a node by its value in a circular linked list.**

#include <iostream>

using namespace std;

// Node structure for the circular linked list

struct Node {

int data;

Node\* next;

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

};

// Function to delete a node by its value in a circular linked list

void deleteNodeByValue(Node\*& head, int value) {

if (head == nullptr) {

cout << "List is empty. Nothing to delete." << endl;

return;

}

Node\* temp = head;

Node\* prev = nullptr;

// If the list contains only one node and it's the node to be deleted

if (head->next == head && head->data == value) {

delete head;

head = nullptr;

cout << "Node with value " << value << " deleted successfully. List is now empty." << endl;

return;

}

// Traverse the list to find the node to delete

do {

if (temp->data == value) {

// If the node to be deleted is the head node

if (temp == head) {

// Find the last node to update its next pointer

Node\* last = head;

while (last->next != head) {

last = last->next;

}

// Update the head and the last node's next pointer

head = head->next;

last->next = head;

} else {

// If the node to delete is not the head, update previous node's next pointer

prev->next = temp->next;

}

delete temp;

cout << "Node with value " << value << " deleted successfully." << endl;

return;

}

prev = temp;

temp = temp->next;

} while (temp != head);

// If the node is not found in the list

cout << "Value " << value << " not found in the list." << endl;

}

// Function to print the circular linked list

void printList(Node\* head) {

if (head == nullptr) {

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

return;

}

Node\* temp = head;

do {

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

temp = temp->next;

} while (temp != head);

cout << endl;

}

// Helper function to append a node to the circular linked list

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

Node\* newNode = new Node(data);

if (head == nullptr) {

head = newNode;

newNode->next = head; // Point to itself to make it circular

return;

}

Node\* temp = head;

while (temp->next != head) {

temp = temp->next;

}

temp->next = newNode;

newNode->next = head;

}

int main() {

Node\* head = nullptr;

// Adding nodes to the circular linked list

appendNode(head, 10);

appendNode(head, 20);

appendNode(head, 30);

appendNode(head, 40);

cout << "Original List: ";

printList(head);

// Deleting a node by value

deleteNodeByValue(head, 30);

cout << "List after deleting node with value 30: ";

printList(head);

// Attempt to delete a node that doesn't exist

deleteNodeByValue(head, 50);

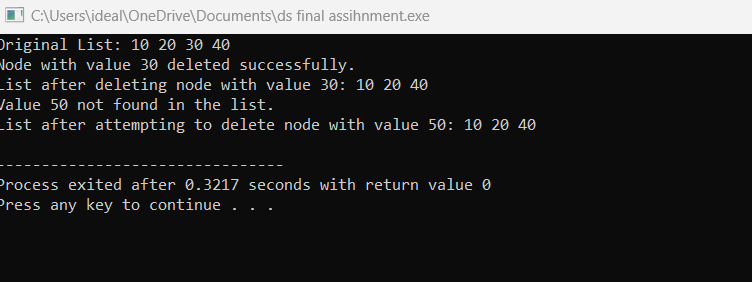
cout << "List after attempting to delete node with value 50: ";

printList(head);

return 0;

}

Output:



• **How will you delete a node at a specific position in a circular linked list? Write code for it**

#include <iostream>

using namespace std;

// Node structure for the circular linked list

struct Node {

int data;

Node\* next;

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

};

// Function to delete a node at a specific position in a circular linked list

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

if (head == nullptr) {

cout << "List is empty. Nothing to delete." << endl;

return;

}

Node\* temp = head;

// If the list contains only one node and the position is 0

if (head->next == head && position == 0) {

delete head;

head = nullptr;

cout << "Node at position " << position << " deleted successfully. List is now empty." << endl;

return;

}

// If position is 0, we need to delete the head node

if (position == 0) {

// Find the last node to update its next pointer

Node\* last = head;

while (last->next != head) {

last = last->next;

}

// Update head to the next node

head = head->next;

// Update the last node's next pointer

last->next = head;

// Delete the old head node

delete temp;

cout << "Node at position " << position << " (head node) deleted successfully." << endl;

return;

}

// Traverse the list to find the node at the given position

for (int i = 0; i < position - 1; i++) {

temp = temp->next;

if (temp->next == head) {

cout << "Position " << position << " is out of bounds." << endl;

return;

}

}

// temp now points to the node before the node to be deleted

Node\* nodeToDelete = temp->next;

temp->next = nodeToDelete->next;

// Delete the node

delete nodeToDelete;

cout << "Node at position " << position << " deleted successfully." << endl;

}

// Function to print the circular linked list

void printList(Node\* head) {

if (head == nullptr) {

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

return;

}

Node\* temp = head;

do {

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

temp = temp->next;

} while (temp != head);

cout << endl;

}

// Helper function to append a node to the circular linked list

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

Node\* newNode = new Node(data);

if (head == nullptr) {

head = newNode;

newNode->next = head; // Point to itself to make it circular

return;

}

Node\* temp = head;

while (temp->next != head) {

temp = temp->next;

}

temp->next = newNode;

newNode->next = head;

}

int main() {

Node\* head = nullptr;

// Adding nodes to the circular linked list

appendNode(head, 10);

appendNode(head, 20);

appendNode(head, 30);

appendNode(head, 40);

cout << "Original List: ";

printList(head);

// Deleting node at position 2

deleteNodeAtPosition(head, 2);

cout << "List after deleting node at position 2: ";

printList(head);

// Deleting node at position 0 (head)

deleteNodeAtPosition(head, 0);

cout << "List after deleting node at position 0: ";

printList(head);

// Attempting to delete a node at an out-of-bounds position

deleteNodeAtPosition(head, 5);

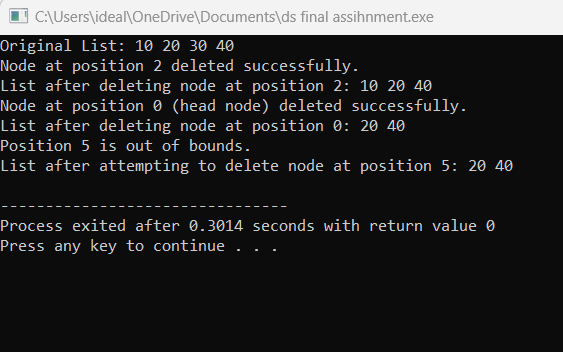
cout << "List after attempting to delete node at position 5: ";

printList(head);

return 0;

}

Output:



**Write a program to show forward traversal after deleting a node in a circular linked list**.

#include <iostream>

using namespace std;

// Node structure for the circular linked list

struct Node {

int data;

Node\* next;

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

};

// Function to delete a node by its value in a circular linked list

void deleteNodeByValue(Node\*& head, int value) {

if (head == nullptr) {

cout << "List is empty. Nothing to delete." << endl;

return;

}

Node\* temp = head;

Node\* prev = nullptr;

// If the list contains only one node and it's the node to be deleted

if (head->next == head && head->data == value) {

delete head;

head = nullptr;

cout << "Node with value " << value << " deleted successfully. List is now empty." << endl;

return;

}

// Traverse the list to find the node to delete

do {

if (temp->data == value) {

// If the node to be deleted is the head node

if (temp == head) {

// Find the last node to update its next pointer

Node\* last = head;

while (last->next != head) {

last = last->next;

}

// Update head to the next node

head = head->next;

// Update the last node's next pointer

last->next = head;

} else {

// If the node to delete is not the head, update previous node's next pointer

prev->next = temp->next;

}

delete temp;

cout << "Node with value " << value << " deleted successfully." << endl;

return;

}

prev = temp;

temp = temp->next;

} while (temp != head);

// If the node is not found in the list

cout << "Value " << value << " not found in the list." << endl;

}

// Function to print the circular linked list (forward traversal)

void printList(Node\* head) {

if (head == nullptr) {

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

return;

}

Node\* temp = head;

do {

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

temp = temp->next;

} while (temp != head);

cout << endl;

}

// Helper function to append a node to the circular linked list

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

Node\* newNode = new Node(data);

if (head == nullptr) {

head = newNode;

newNode->next = head; // Point to itself to make it circular

return;

}

Node\* temp = head;

while (temp->next != head) {

temp = temp->next;

}

temp->next = newNode;

newNode->next = head;

}

int main() {

Node\* head = nullptr;

// Adding nodes to the circular linked list

appendNode(head, 10);

appendNode(head, 20);

appendNode(head, 30);

appendNode(head, 40);

cout << "Original List: ";

printList(head);

// Deleting a node by value (e.g., deleting 30)

deleteNodeByValue(head, 30);

cout << "List after deleting node with value 30: ";

printList(head);

// Deleting the head node (e.g., deleting 10)

deleteNodeByValue(head, 10);

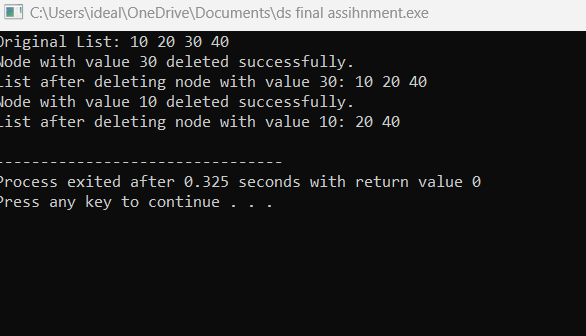
cout << "List after deleting node with value 10: ";

printList(head);

return 0;

}

Output:



**Binary Search Tree**

**• Write a program to count all the nodes in a binary search tree.**

#include <iostream>

using namespace std;

// Define a structure for the node of the Binary Search Tree

struct Node {

int data;

Node\* left;

Node\* right;

Node(int val) : data(val), left(nullptr), right(nullptr) {}

};

// Function to insert a new node into the Binary Search Tree

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

if (root == nullptr) {

return new Node(data);

}

if (data < root->data) {

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

} else {

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

}

return root;

}

// Function to count the total number of nodes in the Binary Search Tree

int countNodes(Node\* root) {

// Base case: If the tree is empty

if (root == nullptr) {

return 0;

}

// Recursive case: Count the node and its left and right subtrees

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

}

// Main function

int main() {

Node\* root = nullptr;

// Inserting nodes into the BST

root = insert(root, 50);

insert(root, 30);

insert(root, 20);

insert(root, 40);

insert(root, 70);

insert(root, 60);

insert(root, 80);

// Counting the number of nodes in the BST

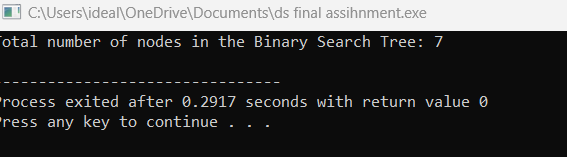
int totalNodes = countNodes(root);

cout << "Total number of nodes in the Binary Search Tree: " << totalNodes << endl;

return 0;

}

Output:



**How can you search for a specific value in a binary search tree? Write the code.**

#include <iostream>

using namespace std;

// Define a structure for the node of the Binary Search Tree

struct Node {

int data;

Node\* left;

Node\* right;

Node(int val) : data(val), left(nullptr), right(nullptr) {}

};

// Function to insert a new node into the Binary Search Tree

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

if (root == nullptr) {

return new Node(data);

}

if (data < root->data) {

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

} else {

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

}

return root;

}

// Function to search for a specific value in the Binary Search Tree

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

// Base case: If the tree is empty or value is not found

if (root == nullptr) {

return false;

}

// If the value matches the current node, return true

if (root->data == value) {

return true;

}

// If the value is less, search in the left subtree

if (value < root->data) {

return search(root->left, value);

}

// If the value is greater, search in the right subtree

return search(root->right, value);

}

// Main function

int main() {

Node\* root = nullptr;

// Inserting nodes into the BST

root = insert(root, 50);

insert(root, 30);

insert(root, 20);

insert(root, 40);

insert(root, 70);

insert(root, 60);

insert(root, 80);

// Searching for a specific value in the BST

int value = 40;

if (search(root, value)) {

cout << "Value " << value << " found in the Binary Search Tree." << endl;

} else {

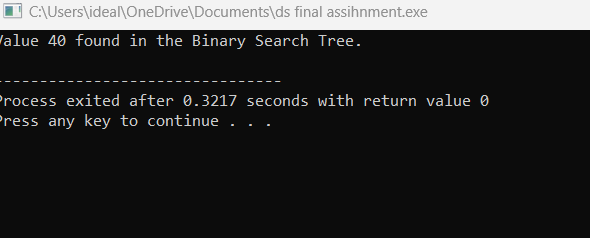
cout << "Value " << value << " not found in the Binary Search Tree." << endl;

}

return 0;

}

Output:



**Write code to traverse a binary search tree in in-order, pre-order, and postorder**.

#include <iostream>

using namespace std;

// Define a structure for the node of the Binary Search Tree

struct Node {

int data;

Node\* left;

Node\* right;

Node(int val) : data(val), left(nullptr), right(nullptr) {}

};

// Function to insert a new node into the Binary Search Tree

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

if (root == nullptr) {

return new Node(data);

}

if (data < root->data) {

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

} else {

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

}

return root;

}

// In-order Traversal (Left, Root, Right)

void inOrderTraversal(Node\* root) {

if (root == nullptr) {

return;

}

// Traverse the left subtree

inOrderTraversal(root->left);

// Visit the root node

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

// Traverse the right subtree

inOrderTraversal(root->right);

}

// Pre-order Traversal (Root, Left, Right)

void preOrderTraversal(Node\* root) {

if (root == nullptr) {

return;

}

// Visit the root node

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

// Traverse the left subtree

preOrderTraversal(root->left);

// Traverse the right subtree

preOrderTraversal(root->right);

}

// Post-order Traversal (Left, Right, Root)

void postOrderTraversal(Node\* root) {

if (root == nullptr) {

return;

}

// Traverse the left subtree

postOrderTraversal(root->left);

// Traverse the right subtree

postOrderTraversal(root->right);

// Visit the root node

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

}

// Main function

int main() {

Node\* root = nullptr;

// Inserting nodes into the BST

root = insert(root, 50);

insert(root, 30);

insert(root, 20);

insert(root, 40);

insert(root, 70);

insert(root, 60);

insert(root, 80);

// Traversing the tree in different orders

cout << "In-order Traversal: ";

inOrderTraversal(root);

cout << endl;

cout << "Pre-order Traversal: ";

preOrderTraversal(root);

cout << endl;

cout << "Post-order Traversal: ";

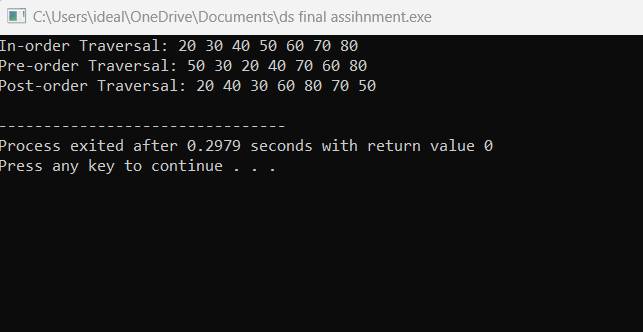
postOrderTraversal(root);

cout << endl;

return 0;

}

Output:



• **How will you write reverse in-order traversal for a binary search tree? Show it in code.**

#include <iostream>

using namespace std;

// Define a structure for the node of the Binary Search Tree

struct Node {

int data;

Node\* left;

Node\* right;

Node(int val) : data(val), left(nullptr), right(nullptr) {}

};

// Function to insert a new node into the Binary Search Tree

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

if (root == nullptr) {

return new Node(data);

}

if (data < root->data) {

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

} else {

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

}

return root;

}

// Reverse In-order Traversal (Right, Root, Left)

void reverseInOrderTraversal(Node\* root) {

if (root == nullptr) {

return;

}

// Traverse the right subtree

reverseInOrderTraversal(root->right);

// Visit the root node

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

// Traverse the left subtree

reverseInOrderTraversal(root->left);

}

// Main function

int main() {

Node\* root = nullptr;

// Inserting nodes into the BST

root = insert(root, 50);

insert(root, 30);

insert(root, 20);

insert(root, 40);

insert(root, 70);

insert(root, 60);

insert(root, 80);

// Reverse In-order Traversal

cout << "Reverse In-order Traversal: ";

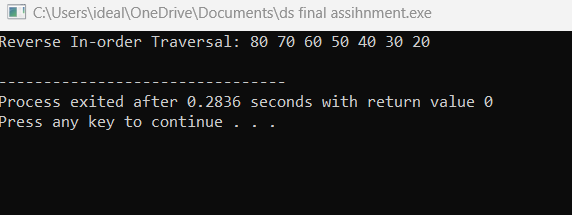
reverseInOrderTraversal(root);

cout << endl;

return 0;

}

Output:



**Write a program to check if there are duplicate values in a binary search tree.**

#include <iostream>

using namespace std;

// Define a structure for the node of the Binary Search Tree

struct Node {

int data;

Node\* left;

Node\* right;

Node(int val) : data(val), left(nullptr), right(nullptr) {}

};

// Function to insert a new node into the Binary Search Tree

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

if (root == nullptr) {

return new Node(data);

}

if (data < root->data) {

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

} else {

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

}

return root;

}

// Function to check for duplicate values during in-order traversal

bool checkForDuplicates(Node\* root, int& prevValue) {

if (root == nullptr) {

return false; // No duplicates in an empty tree

}

// Traverse the left subtree

if (checkForDuplicates(root->left, prevValue)) {

return true; // Duplicate found in the left subtree

}

// Check if the current node's value is the same as the previous node's value

if (root->data == prevValue) {

return true; // Duplicate found

}

// Update the previous value to the current node's value

prevValue = root->data;

// Traverse the right subtree

return checkForDuplicates(root->right, prevValue);

}

// Main function to check if the BST has duplicates

bool hasDuplicates(Node\* root) {

int prevValue = -1; // Initialize with a value that cannot be in the BST

return checkForDuplicates(root, prevValue);

}

// Main function

int main() {

Node\* root = nullptr;

// Inserting nodes into the BST

root = insert(root, 50);

insert(root, 30);

insert(root, 20);

insert(root, 40);

insert(root, 70);

insert(root, 60);

insert(root, 80);

// Insert a duplicate value (e.g., 40)

insert(root, 40);

// Check if the BST has duplicate values

if (hasDuplicates(root)) {

cout << "The Binary Search Tree contains duplicate values." << endl;

} else {

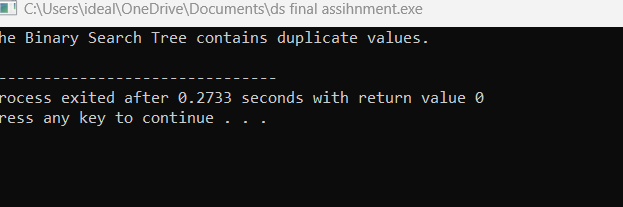
cout << "The Binary Search Tree does not contain duplicate values." << endl;

}

return 0;

}

Output:



**How can you delete a node from a binary search tree? Write code for deleting a leaf, a node with one child, and a node with two children.**

#include <iostream>

using namespace std;

// Define a structure for the node of the Binary Search Tree

struct Node {

int data;

Node\* left;

Node\* right;

Node(int val) : data(val), left(nullptr), right(nullptr) {}

};

// Function to insert a new node into the Binary Search Tree

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

if (root == nullptr) {

return new Node(data);

}

if (data < root->data) {

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

} else {

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

}

return root;

}

// Function to find the minimum node in the BST (used for in-order successor)

Node\* findMin(Node\* root) {

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

root = root->left;

}

return root;

}

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

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

if (root == nullptr) {

return root; // Node not found

}

// If the key to be deleted is smaller than the root's key, go to the left subtree

if (key < root->data) {

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

}

// If the key to be deleted is larger than the root's key, go to the right subtree

else if (key > root->data) {

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

}

// If the key is the same as the root's key, this is the node to be deleted

else {

// Case 1: Node has no children (leaf node)

if (root->left == nullptr && root->right == nullptr) {

delete root;

root = nullptr;

}

// Case 2: Node has one child

else if (root->left == nullptr) {

Node\* temp = root;

root = root->right;

delete temp;

}

else if (root->right == nullptr) {

Node\* temp = root;

root = root->left;

delete temp;

}

// Case 3: Node has two children

else {

// Find the in-order successor (smallest in the right subtree)

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

// Copy the in-order successor's value to this node

root->data = temp->data;

// Delete the in-order successor

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

}

}

return root;

}

// Function for in-order traversal

void inOrderTraversal(Node\* root) {

if (root == nullptr) {

return;

}

inOrderTraversal(root->left);

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

inOrderTraversal(root->right);

}

// Main function

int main() {

Node\* root = nullptr;

// Inserting nodes into the BST

root = insert(root, 50);

insert(root, 30);

insert(root, 20);

insert(root, 40);

insert(root, 70);

insert(root, 60);

insert(root, 80);

cout << "Original Tree (In-order Traversal): ";

inOrderTraversal(root);

cout << endl;

// Deleting a node with no children (leaf)

cout << "Deleting leaf node 20:" << endl;

root = deleteNode(root, 20);

inOrderTraversal(root);

cout << endl;

// Deleting a node with one child

cout << "Deleting node with one child 30:" << endl;

root = deleteNode(root, 30);

inOrderTraversal(root);

cout << endl;

// Deleting a node with two children

cout << "Deleting node with two children 70:" << endl;

root = deleteNode(root, 70);

inOrderTraversal(root);

cout << endl;

return 0;

}

**Output:**

