**Final Assignment**

**Double linked list**

**Program 1**

#include <iostream>

using namespace std;

// Define a structure for the doubly linked list node

struct Node {

int data; // The data part of the node

Node\* next; // Pointer to the next node

Node\* prev; // Pointer to the previous node

// Constructor to create a new node

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

};

// Class to represent the Doubly Linked List

class DoublyLinkedList {

private:

Node\* head; // Pointer to the first node in the list

public:

// Constructor to initialize the list

DoublyLinkedList() : head(nullptr) {}

// Function to add a node at the end of the list

void append(int data) {

Node\* newNode = new Node(data);

if (head == nullptr) {

head = newNode;

} else {

Node\* temp = head;

while (temp->next != nullptr) {

temp = temp->next;

}

temp->next = newNode;

newNode->prev = temp;

}

}

// Function to delete the first node

void deleteFirst() {

if (head == nullptr) {

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

return;

}

Node\* temp = head;

head = head->next; // Move the head to the next node

if (head != nullptr) {

head->prev = nullptr; // Update the previous pointer of the new head

}

delete temp; // Free the memory of the deleted node

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

}

// Function to display the list

void display() {

if (head == nullptr) {

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

return;

}

Node\* temp = head;

while (temp != nullptr) {

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

temp = temp->next;

}

cout << "NULL" << endl;

}

// Destructor to free up the list memory

~DoublyLinkedList() {

while (head != nullptr) {

Node\* temp = head;

head = head->next;

delete temp;

}

}

};

int main() {

DoublyLinkedList list;

// Add some nodes to the list

list.append(10);

list.append(20);

list.append(30);

list.append(40);

// Display the list before deletion

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

list.display();

// Delete the first node

list.deleteFirst();

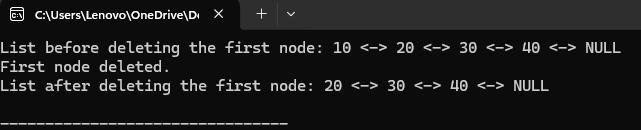
// Display the list after deletion

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

list.display();

return 0;

}



**Program 2**

#include <iostream>

using namespace std;

// Define a structure for the doubly linked list node

struct Node {

int data; // The data part of the node

Node\* next; // Pointer to the next node

Node\* prev; // Pointer to the previous node

// Constructor to create a new node

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

};

// Class to represent the Doubly Linked List

class DoublyLinkedList {

private:

Node\* head; // Pointer to the first node in the list

public:

// Constructor to initialize the list

DoublyLinkedList() : head(nullptr) {}

// Function to add a node at the end of the list

void append(int data) {

Node\* newNode = new Node(data);

if (head == nullptr) {

head = newNode;

} else {

Node\* temp = head;

while (temp->next != nullptr) {

temp = temp->next;

}

temp->next = newNode;

newNode->prev = temp;

}

}

// Function to delete the last node

void deleteLast() {

if (head == nullptr) {

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

return;

}

// If the list has only one node

if (head->next == nullptr) {

delete head;

head = nullptr;

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

return;

}

// Traverse to the last node

Node\* temp = head;

while (temp->next != nullptr) {

temp = temp->next;

}

// Remove the last node

Node\* lastNode = temp;

temp->prev->next = nullptr; // Update the second last node's next pointer to null

delete lastNode; // Free memory for the last node

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

}

// Function to display the list

void display() {

if (head == nullptr) {

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

return;

}

Node\* temp = head;

while (temp != nullptr) {

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

temp = temp->next;

}

cout << "NULL" << endl;

}

// Destructor to free up the list memory

~DoublyLinkedList() {

while (head != nullptr) {

Node\* temp = head;

head = head->next;

delete temp;

}

}

};

int main() {

DoublyLinkedList list;

// Add some nodes to the list

list.append(10);

list.append(20);

list.append(30);

list.append(40);

// Display the list before deletion

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

list.display();

// Delete the last node

list.deleteLast();

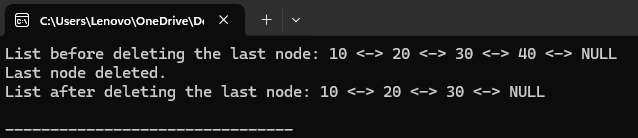
// Display the list after deletion

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

list.display();

return 0;

}



**Program 3**

#include <iostream>

using namespace std;

// Define a structure for the doubly linked list node

struct Node {

int data; // The data part of the node

Node\* next; // Pointer to the next node

Node\* prev; // Pointer to the previous node

// Constructor to create a new node

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

};

// Class to represent the Doubly Linked List

class DoublyLinkedList {

private:

Node\* head; // Pointer to the first node in the list

public:

// Constructor to initialize the list

DoublyLinkedList() : head(nullptr) {}

// Function to add a node at the end of the list

void append(int data) {

Node\* newNode = new Node(data);

if (head == nullptr) {

head = newNode;

} else {

Node\* temp = head;

while (temp->next != nullptr) {

temp = temp->next;

}

temp->next = newNode;

newNode->prev = temp;

}

}

// Function to delete a node by its value

void deleteByValue(int value) {

if (head == nullptr) {

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

return;

}

Node\* temp = head;

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

if (temp->data == value) {

head = temp->next;

if (head != nullptr) {

head->prev = nullptr;

}

delete temp;

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

return;

}

// Traverse the list to find the node with the given value

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

temp = temp->next;

}

// If node with the value is not found

if (temp == nullptr) {

cout << "Node with value " << value << " not found." << endl;

return;

}

// If the node is found, update the previous and next pointers

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." << endl;

}

// Function to display the list

void display() {

if (head == nullptr) {

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

return;

}

Node\* temp = head;

while (temp != nullptr) {

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

temp = temp->next;

}

cout << "NULL" << endl;

}

// Destructor to free up the list memory

~DoublyLinkedList() {

while (head != nullptr) {

Node\* temp = head;

head = head->next;

delete temp;

}

}

};

int main() {

DoublyLinkedList list;

// Add some nodes to the list

list.append(10);

list.append(20);

list.append(30);

list.append(40);

// Display the list before deletion

cout << "List before deleting a node by value: ";

list.display();

// Delete a node with a specific value

list.deleteByValue(30);

// Display the list after deletion

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

list.display();

// Try to delete a non-existing value

list.deleteByValue(50);

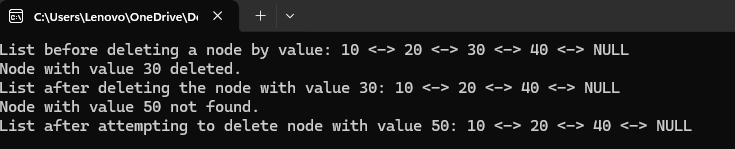
// Display the list after attempting to delete a non-existing value

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

list.display();

return 0;

}



**Program 4**

#include <iostream>

using namespace std;

// Define a structure for the doubly linked list node

struct Node {

int data; // The data part of the node

Node\* next; // Pointer to the next node

Node\* prev; // Pointer to the previous node

// Constructor to create a new node

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

};

// Class to represent the Doubly Linked List

class DoublyLinkedList {

private:

Node\* head; // Pointer to the first node in the list

public:

// Constructor to initialize the list

DoublyLinkedList() : head(nullptr) {}

// Function to add a node at the end of the list

void append(int data) {

Node\* newNode = new Node(data);

if (head == nullptr) {

head = newNode;

} else {

Node\* temp = head;

while (temp->next != nullptr) {

temp = temp->next;

}

temp->next = newNode;

newNode->prev = temp;

}

}

// Function to delete a node at a specific position (1-based index)

void deleteAtPosition(int position) {

if (head == nullptr) {

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

return;

}

Node\* temp = head;

int count = 1;

// If position is 1, we delete the head node

if (position == 1) {

head = temp->next;

if (head != nullptr) {

head->prev = nullptr;

}

delete temp;

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

return;

}

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

while (temp != nullptr && count < position) {

temp = temp->next;

count++;

}

// If the position is greater than the number of nodes

if (temp == nullptr) {

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

return;

}

// If the node is not the last one

if (temp->next != nullptr) {

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

}

// If the node is not the first one

if (temp->prev != nullptr) {

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

}

// Delete the node

delete temp;

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

}

// Function to display the list

void display() {

if (head == nullptr) {

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

return;

}

Node\* temp = head;

while (temp != nullptr) {

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

temp = temp->next;

}

cout << "NULL" << endl;

}

// Destructor to free up the list memory

~DoublyLinkedList() {

while (head != nullptr) {

Node\* temp = head;

head = head->next;

delete temp;

}

}

};

int main() {

DoublyLinkedList list;

// Add some nodes to the list

list.append(10);

list.append(20);

list.append(30);

list.append(40);

// Display the list before deletion

cout << "List before deleting a node at position: ";

list.display();

// Delete the node at position 3

list.deleteAtPosition(3);

// Display the list after deletion

cout << "List after deleting the node at position 3: ";

list.display();

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

list.deleteAtPosition(10);

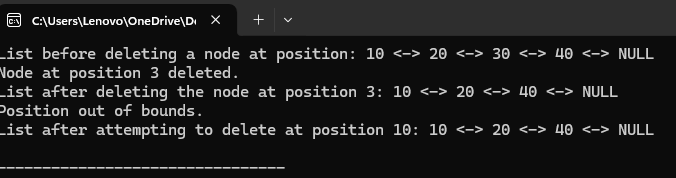
// Display the list after attempting to delete at an out-of-bounds position

cout << "List after attempting to delete at position 10: ";

list.display();

return 0;

}



**Program 5**

#include <iostream>

using namespace std;

// Define a structure for the doubly linked list node

struct Node {

int data; // The data part of the node

Node\* next; // Pointer to the next node

Node\* prev; // Pointer to the previous node

// Constructor to create a new node

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

};

// Class to represent the Doubly Linked List

class DoublyLinkedList {

private:

Node\* head; // Pointer to the first node in the list

Node\* tail; // Pointer to the last node in the list

public:

// Constructor to initialize the list

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

// Function to add a node at the end of the list

void append(int data) {

Node\* newNode = new Node(data);

if (head == nullptr) {

head = tail = newNode;

} else {

tail->next = newNode;

newNode->prev = tail;

tail = newNode;

}

}

// Function to delete a node at a specific position (1-based index)

void deleteAtPosition(int position) {

if (head == nullptr) {

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

return;

}

Node\* temp = head;

int count = 1;

// If position is 1, we delete the head node

if (position == 1) {

head = temp->next;

if (head != nullptr) {

head->prev = nullptr;

}

delete temp;

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

return;

}

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

while (temp != nullptr && count < position) {

temp = temp->next;

count++;

}

// If the position is greater than the number of nodes

if (temp == nullptr) {

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

return;

}

// If the node is not the last one

if (temp->next != nullptr) {

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

}

// If the node is not the first one

if (temp->prev != nullptr) {

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

}

// If it is the last node

if (temp == tail) {

tail = temp->prev;

}

// Delete the node

delete temp;

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

}

// Function for forward traversal (head to tail)

void forwardTraversal() {

if (head == nullptr) {

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

return;

}

Node\* temp = head;

while (temp != nullptr) {

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

temp = temp->next;

}

cout << "NULL" << endl;

}

// Function for reverse traversal (tail to head)

void reverseTraversal() {

if (tail == nullptr) {

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

return;

}

Node\* temp = tail;

while (temp != nullptr) {

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

temp = temp->prev;

}

cout << "NULL" << endl;

}

// Destructor to free up the list memory

~DoublyLinkedList() {

while (head != nullptr) {

Node\* temp = head;

head = head->next;

delete temp;

}

}

};

int main() {

DoublyLinkedList list;

// Add some nodes to the list

list.append(10);

list.append(20);

list.append(30);

list.append(40);

// Display the list before deletion

cout << "List before deletion: " << endl;

list.forwardTraversal();

list.reverseTraversal();

// Delete the node at position 3

list.deleteAtPosition(3);

// Display the list after deletion

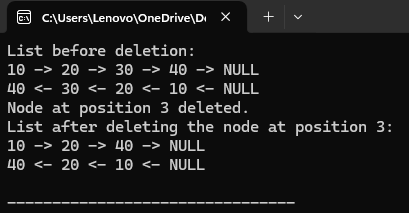
cout << "List after deleting the node at position 3: " << endl;

list.forwardTraversal();

list.reverseTraversal();

return 0;

}



**Circular linked list**

**Program 1**

#include <iostream>

using namespace std;

// Define a structure for the circular linked list node

struct Node {

int data; // The data part of the node

Node\* next; // Pointer to the next node

// Constructor to create a new node

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

};

// Class to represent the Circular Linked List

class CircularLinkedList {

private:

Node\* head; // Pointer to the first node in the list

public:

// Constructor to initialize the list

CircularLinkedList() : head(nullptr) {}

// Function to add a node at the end of the list (to form the circular nature)

void append(int data) {

Node\* newNode = new Node(data);

if (head == nullptr) {

head = newNode;

newNode->next = head; // The last node points to the head to form a circle

} else {

Node\* temp = head;

while (temp->next != head) {

temp = temp->next;

}

temp->next = newNode;

newNode->next = head; // Make the list circular

}

}

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

void deleteFirst() {

if (head == nullptr) {

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

return;

}

// If there's only one node in the list

if (head->next == head) {

delete head;

head = nullptr;

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

return;

}

// If there are multiple nodes

Node\* temp = head;

Node\* last = head;

// Find the last node (the one whose next points to the head)

while (last->next != head) {

last = last->next;

}

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

last->next = head->next;

// Delete the head node

head = head->next;

delete temp;

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

}

// Function to display the list

void display() {

if (head == nullptr) {

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

return;

}

Node\* temp = head;

do {

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

temp = temp->next;

} while (temp != head);

cout << "(head)" << endl;

}

// Destructor to free up the list memory

~CircularLinkedList() {

if (head == nullptr) return;

Node\* temp = head;

do {

Node\* nextNode = temp->next;

delete temp;

temp = nextNode;

} while (temp != head);

}

};

int main() {

CircularLinkedList list;

// Add some nodes to the circular linked list

list.append(10);

list.append(20);

list.append(30);

list.append(40);

// Display the list before deletion

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

list.display();

// Delete the first node

list.deleteFirst();

// Display the list after deletion

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

list.display();

// Delete the first node again to test the list behavior

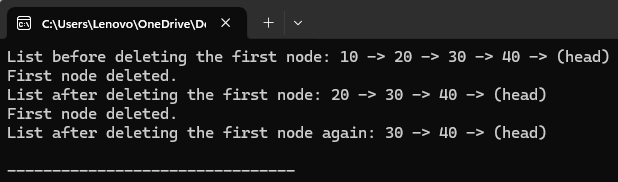
list.deleteFirst();

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

list.display();

return 0;

}



**Program 2**

#include <iostream>

using namespace std;

// Define a structure for the circular linked list node

struct Node {

int data; // The data part of the node

Node\* next; // Pointer to the next node

// Constructor to create a new node

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

};

// Class to represent the Circular Linked List

class CircularLinkedList {

private:

Node\* head; // Pointer to the first node in the list

public:

// Constructor to initialize the list

CircularLinkedList() : head(nullptr) {}

// Function to add a node at the end of the list (to form the circular nature)

void append(int data) {

Node\* newNode = new Node(data);

if (head == nullptr) {

head = newNode;

newNode->next = head; // The last node points to the head to form a circle

} else {

Node\* temp = head;

while (temp->next != head) {

temp = temp->next;

}

temp->next = newNode;

newNode->next = head; // Make the list circular

}

}

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

void deleteLast() {

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. List is now empty." << endl;

return;

}

// If there are multiple nodes

Node\* temp = head;

// Traverse the list to find the second-to-last node

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

temp = temp->next;

}

// temp now points to the second-to-last node

Node\* lastNode = temp->next; // The last node to delete

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

temp->next = head;

// Delete the last node

delete lastNode;

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

}

// Function to display the list

void display() {

if (head == nullptr) {

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

return;

}

Node\* temp = head;

do {

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

temp = temp->next;

} while (temp != head);

cout << "(head)" << endl;

}

// Destructor to free up the list memory

~CircularLinkedList() {

if (head == nullptr) return;

Node\* temp = head;

do {

Node\* nextNode = temp->next;

delete temp;

temp = nextNode;

} while (temp != head);

}

};

int main() {

CircularLinkedList list;

// Add some nodes to the circular linked list

list.append(10);

list.append(20);

list.append(30);

list.append(40);

// Display the list before deletion

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

list.display();

// Delete the last node

list.deleteLast();

// Display the list after deletion

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

list.display();

// Delete the last node again to test the list behavior

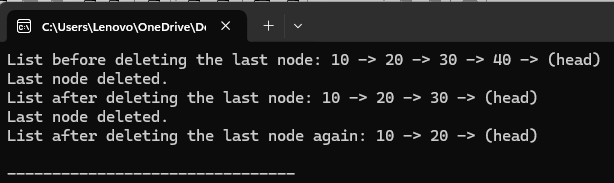
list.deleteLast();

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

list.display();

return 0;

}



**Program 3**

#include <iostream>

using namespace std;

// Define a structure for the circular linked list node

struct Node {

int data; // The data part of the node

Node\* next; // Pointer to the next node

// Constructor to create a new node

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

};

// Class to represent the Circular Linked List

class CircularLinkedList {

private:

Node\* head; // Pointer to the first node in the list

public:

// Constructor to initialize the list

CircularLinkedList() : head(nullptr) {}

// Function to add a node at the end of the list (to form the circular nature)

void append(int data) {

Node\* newNode = new Node(data);

if (head == nullptr) {

head = newNode;

newNode->next = head; // The last node points to the head to form a circle

} else {

Node\* temp = head;

while (temp->next != head) {

temp = temp->next;

}

temp->next = newNode;

newNode->next = head; // Make the list circular

}

}

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

void deleteByValue(int value) {

if (head == nullptr) {

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

return;

}

// If the node to delete is the head node

if (head->data == value) {

// If there's only one node

if (head->next == head) {

delete head;

head = nullptr;

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

return;

}

// If there are multiple nodes, update the last node's next pointer

Node\* temp = head;

while (temp->next != head) {

temp = temp->next;

}

// Temp now points to the last node

temp->next = head->next;

Node\* oldHead = head;

head = head->next;

delete oldHead;

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

return;

}

// If the node to delete is not the head node

Node\* current = head;

Node\* previous = nullptr;

// Traverse the list to find the node

do {

previous = current;

current = current->next;

if (current->data == value) {

// Remove the node from the list

previous->next = current->next;

delete current;

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

return;

}

} while (current != head);

// If the value is not found in the list

cout << "Node with value " << value << " not found." << endl;

}

// Function to display the list

void display() {

if (head == nullptr) {

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

return;

}

Node\* temp = head;

do {

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

temp = temp->next;

} while (temp != head);

cout << "(head)" << endl;

}

// Destructor to free up the list memory

~CircularLinkedList() {

if (head == nullptr) return;

Node\* temp = head;

do {

Node\* nextNode = temp->next;

delete temp;

temp = nextNode;

} while (temp != head);

}

};

int main() {

CircularLinkedList list;

// Add some nodes to the circular linked list

list.append(10);

list.append(20);

list.append(30);

list.append(40);

// Display the list before deletion

cout << "List before deleting a node with value 20: ";

list.display();

// Delete the node with value 20

list.deleteByValue(20);

// Display the list after deletion

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

list.display();

// Try deleting a node that does not exist

list.deleteByValue(50);

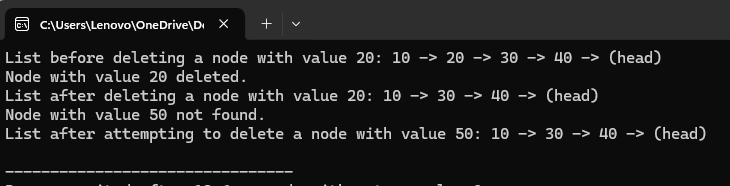
// Display the list again

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

list.display();

return 0;

}



**Program 4**

#include <iostream>

using namespace std;

// Define a structure for the circular linked list node

struct Node {

int data; // The data part of the node

Node\* next; // Pointer to the next node

// Constructor to create a new node

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

};

// Class to represent the Circular Linked List

class CircularLinkedList {

private:

Node\* head; // Pointer to the first node in the list

public:

// Constructor to initialize the list

CircularLinkedList() : head(nullptr) {}

// Function to add a node at the end of the list (to form the circular nature)

void append(int data) {

Node\* newNode = new Node(data);

if (head == nullptr) {

head = newNode;

newNode->next = head; // The last node points to the head to form a circle

} else {

Node\* temp = head;

while (temp->next != head) {

temp = temp->next;

}

temp->next = newNode;

newNode->next = head; // Make the list circular

}

}

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

void deleteAtPosition(int position) {

if (head == nullptr) {

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

return;

}

Node\* temp = head;

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

if (position == 0) {

// If there is only one node

if (head->next == head) {

delete head;

head = nullptr;

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

return;

}

// Traverse to the last node

while (temp->next != head) {

temp = temp->next;

}

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

temp->next = head->next;

Node\* oldHead = head;

head = head->next;

delete oldHead;

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

return;

}

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

Node\* prev = nullptr;

int count = 0;

while (count < position) {

prev = temp;

temp = temp->next;

count++;

// If we reach the end of the list (i.e., the position is out of bounds)

if (temp == head) {

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

return;

}

}

// Remove the node at the specified position

prev->next = temp->next;

delete temp;

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

}

// Function to display the list

void display() {

if (head == nullptr) {

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

return;

}

Node\* temp = head;

do {

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

temp = temp->next;

} while (temp != head);

cout << "(head)" << endl;

}

// Destructor to free up the list memory

~CircularLinkedList() {

if (head == nullptr) return;

Node\* temp = head;

do {

Node\* nextNode = temp->next;

delete temp;

temp = nextNode;

} while (temp != head);

}

};

int main() {

CircularLinkedList list;

// Add some nodes to the circular linked list

list.append(10);

list.append(20);

list.append(30);

list.append(40);

// Display the list before deletion

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

list.display();

// Delete the node at position 2 (0-based index)

list.deleteAtPosition(2);

// Display the list after deletion

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

list.display();

// Try deleting a node at an invalid position

list.deleteAtPosition(5); // Out of bounds

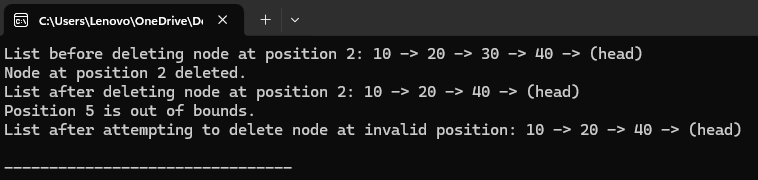
// Display the list again

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

list.display();

return 0;

}



**Program 5**

#include <iostream>

using namespace std;

// Define a structure for the circular linked list node

struct Node {

int data; // The data part of the node

Node\* next; // Pointer to the next node

// Constructor to create a new node

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

};

// Class to represent the Circular Linked List

class CircularLinkedList {

private:

Node\* head; // Pointer to the first node in the list

public:

// Constructor to initialize the list

CircularLinkedList() : head(nullptr) {}

// Function to add a node at the end of the list (to form the circular nature)

void append(int data) {

Node\* newNode = new Node(data);

if (head == nullptr) {

head = newNode;

newNode->next = head; // The last node points to the head to form a circle

} else {

Node\* temp = head;

while (temp->next != head) {

temp = temp->next;

}

temp->next = newNode;

newNode->next = head; // Make the list circular

}

}

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

void deleteByValue(int value) {

if (head == nullptr) {

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

return;

}

Node\* temp = head;

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

if (head->data == value) {

// If there's only one node

if (head->next == head) {

delete head;

head = nullptr;

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

return;

}

// Traverse to the last node

while (temp->next != head) {

temp = temp->next;

}

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

temp->next = head->next;

Node\* oldHead = head;

head = head->next;

delete oldHead;

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

return;

}

// Traverse the list to find the node to delete

Node\* prev = nullptr;

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

prev = temp;

temp = temp->next;

}

// If the node to delete is found

if (temp->data == value) {

prev->next = temp->next;

delete temp;

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

} else {

cout << "Node with value " << value << " not found." << endl;

}

}

// Function to display the list (forward traversal)

void display() {

if (head == nullptr) {

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

return;

}

Node\* temp = head;

do {

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

temp = temp->next;

} while (temp != head);

cout << "(head)" << endl;

}

// Destructor to free up the list memory

~CircularLinkedList() {

if (head == nullptr) return;

Node\* temp = head;

do {

Node\* nextNode = temp->next;

delete temp;

temp = nextNode;

} while (temp != head);

}

};

int main() {

CircularLinkedList list;

// Add some nodes to the circular linked list

list.append(10);

list.append(20);

list.append(30);

list.append(40);

// Display the list before deletion

cout << "List before deleting a node with value 20: ";

list.display();

// Delete the node with value 20

list.deleteByValue(20);

// Display the list after deletion

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

list.display();

// Delete the node with value 10 (head node)

list.deleteByValue(10);

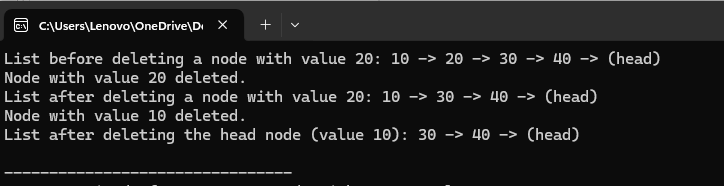
// Display the list after deletion of the head node

cout << "List after deleting the head node (value 10): ";

list.display();

return 0;

}



**Binary tree search**

**Program 1**

#include <iostream>

using namespace std;

// Structure for the Node of the Binary Search Tree

struct Node {

int data; // Data of the node

Node\* left; // Pointer to the left child

Node\* right; // Pointer to the right child

// Constructor to create a new node

Node(int value) {

data = value;

left = right = nullptr;

}

};

// Class for Binary Search Tree

class BST {

private:

Node\* root; // Root node of the tree

// Helper function to count nodes recursively

int countNodesHelper(Node\* node) {

if (node == nullptr) {

return 0; // Base case: if the node is NULL, return 0

}

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

return 1 + countNodesHelper(node->left) + countNodesHelper(node->right);

}

public:

// Constructor to initialize the tree

BST() {

root = nullptr;

}

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

void insert(int value) {

root = insertHelper(root, value);

}

// Helper function to insert a node

Node\* insertHelper(Node\* node, int value) {

if (node == nullptr) {

return new Node(value); // Create a new node if the position is empty

}

if (value < node->data) {

node->left = insertHelper(node->left, value); // Insert in the left subtree

} else {

node->right = insertHelper(node->right, value); // Insert in the right subtree

}

return node;

}

// Public function to count all nodes in the tree

int countNodes() {

return countNodesHelper(root);

}

// Function to display the tree (In-order traversal for visualization)

void inorder() {

inorderHelper(root);

cout << endl;

}

// Helper function for In-order traversal

void inorderHelper(Node\* node) {

if (node != nullptr) {

inorderHelper(node->left);

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

inorderHelper(node->right);

}

}

};

int main() {

BST tree;

// Inserting nodes into the BST

tree.insert(50);

tree.insert(30);

tree.insert(20);

tree.insert(40);

tree.insert(70);

tree.insert(60);

tree.insert(80);

// Displaying the tree using In-order traversal

cout << "In-order traversal of the BST: ";

tree.inorder();

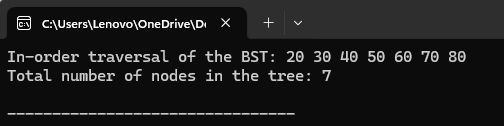
// Counting the nodes in the BST

int nodeCount = tree.countNodes();

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

return 0;

}



**Program 2**

#include <iostream>

using namespace std;

// Structure for the Node of the Binary Search Tree

struct Node {

int data; // Data of the node

Node\* left; // Pointer to the left child

Node\* right; // Pointer to the right child

// Constructor to create a new node

Node(int value) {

data = value;

left = right = nullptr;

}

};

// Class for Binary Search Tree

class BST {

private:

Node\* root; // Root node of the tree

// Helper function to insert a node into the Binary Search Tree

Node\* insertHelper(Node\* node, int value) {

if (node == nullptr) {

return new Node(value); // Create a new node if the position is empty

}

if (value < node->data) {

node->left = insertHelper(node->left, value); // Insert in the left subtree

} else {

node->right = insertHelper(node->right, value); // Insert in the right subtree

}

return node;

}

// Helper function to search for a value in the Binary Search Tree

Node\* searchHelper(Node\* node, int value) {

if (node == nullptr) {

return nullptr; // Base case: the value is not found

}

if (node->data == value) {

return node; // The value is found

}

if (value < node->data) {

return searchHelper(node->left, value); // Search in the left subtree

} else {

return searchHelper(node->right, value); // Search in the right subtree

}

}

public:

// Constructor to initialize the tree

BST() {

root = nullptr;

}

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

void insert(int value) {

root = insertHelper(root, value);

}

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

bool search(int value) {

Node\* result = searchHelper(root, value);

return result != nullptr; // If the result is not null, the value was found

}

// Function to display the tree (In-order traversal for visualization)

void inorder() {

inorderHelper(root);

cout << endl;

}

// Helper function for In-order traversal

void inorderHelper(Node\* node) {

if (node != nullptr) {

inorderHelper(node->left);

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

inorderHelper(node->right);

}

}

};

int main() {

BST tree;

// Inserting nodes into the BST

tree.insert(50);

tree.insert(30);

tree.insert(20);

tree.insert(40);

tree.insert(70);

tree.insert(60);

tree.insert(80);

// Displaying the tree using In-order traversal

cout << "In-order traversal of the BST: ";

tree.inorder();

// Searching for a specific value in the BST

int valueToSearch = 40;

if (tree.search(valueToSearch)) {

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

} else {

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

}

// Searching for another value in the BST

valueToSearch = 25;

if (tree.search(valueToSearch)) {

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

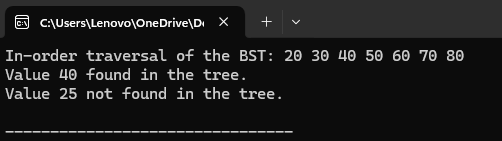
} else {

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

}

return 0;

}



**Program 3**

#include <iostream>

using namespace std;

// Structure for the Node of the Binary Search Tree

struct Node {

int data; // Data of the node

Node\* left; // Pointer to the left child

Node\* right; // Pointer to the right child

// Constructor to create a new node

Node(int value) {

data = value;

left = right = nullptr;

}

};

// Class for Binary Search Tree

class BST {

private:

Node\* root; // Root node of the tree

// Helper function to insert a node into the Binary Search Tree

Node\* insertHelper(Node\* node, int value) {

if (node == nullptr) {

return new Node(value); // Create a new node if the position is empty

}

if (value < node->data) {

node->left = insertHelper(node->left, value); // Insert in the left subtree

} else {

node->right = insertHelper(node->right, value); // Insert in the right subtree

}

return node;

}

// Helper function for In-order traversal

void inorderHelper(Node\* node) {

if (node != nullptr) {

inorderHelper(node->left); // Traverse left subtree

cout << node->data << " "; // Visit root

inorderHelper(node->right); // Traverse right subtree

}

}

// Helper function for Pre-order traversal

void preorderHelper(Node\* node) {

if (node != nullptr) {

cout << node->data << " "; // Visit root

preorderHelper(node->left); // Traverse left subtree

preorderHelper(node->right); // Traverse right subtree

}

}

// Helper function for Post-order traversal

void postorderHelper(Node\* node) {

if (node != nullptr) {

postorderHelper(node->left); // Traverse left subtree

postorderHelper(node->right); // Traverse right subtree

cout << node->data << " "; // Visit root

}

}

public:

// Constructor to initialize the tree

BST() {

root = nullptr;

}

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

void insert(int value) {

root = insertHelper(root, value);

}

// Public function to perform In-order traversal

void inorder() {

cout << "In-order Traversal: ";

inorderHelper(root);

cout << endl;

}

// Public function to perform Pre-order traversal

void preorder() {

cout << "Pre-order Traversal: ";

preorderHelper(root);

cout << endl;

}

// Public function to perform Post-order traversal

void postorder() {

cout << "Post-order Traversal: ";

postorderHelper(root);

cout << endl;

}

};

int main() {

BST tree;

// Inserting nodes into the BST

tree.insert(50);

tree.insert(30);

tree.insert(20);

tree.insert(40);

tree.insert(70);

tree.insert(60);

tree.insert(80);

// Perform different types of traversals

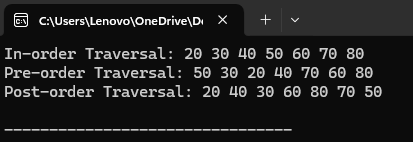
tree.inorder(); // In-order traversal

tree.preorder(); // Pre-order traversal

tree.postorder(); // Post-order traversal

return 0;

}



**Program 4**

#include <iostream>

using namespace std;

// Structure for the Node of the Binary Search Tree

struct Node {

int data; // Data of the node

Node\* left; // Pointer to the left child

Node\* right; // Pointer to the right child

// Constructor to create a new node

Node(int value) {

data = value;

left = right = nullptr;

}

};

// Class for Binary Search Tree

class BST {

private:

Node\* root; // Root node of the tree

// Helper function to insert a node into the Binary Search Tree

Node\* insertHelper(Node\* node, int value) {

if (node == nullptr) {

return new Node(value); // Create a new node if the position is empty

}

if (value < node->data) {

node->left = insertHelper(node->left, value); // Insert in the left subtree

} else {

node->right = insertHelper(node->right, value); // Insert in the right subtree

}

return node;

}

// Helper function for Reverse In-order traversal

void reverseInorderHelper(Node\* node) {

if (node != nullptr) {

reverseInorderHelper(node->right); // Traverse right subtree

cout << node->data << " "; // Visit root

reverseInorderHelper(node->left); // Traverse left subtree

}

}

public:

// Constructor to initialize the tree

BST() {

root = nullptr;

}

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

void insert(int value) {

root = insertHelper(root, value);

}

// Public function to perform Reverse In-order traversal

void reverseInorder() {

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

reverseInorderHelper(root);

cout << endl;

}

};

int main() {

BST tree;

// Inserting nodes into the BST

tree.insert(50);

tree.insert(30);

tree.insert(20);

tree.insert(40);

tree.insert(70);

tree.insert(60);

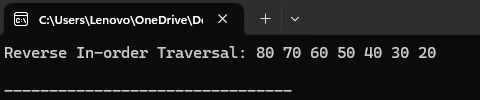
tree.insert(80);

// Perform Reverse In-order traversal

tree.reverseInorder(); // Reverse In-order traversal

return 0;

}



**Program 5**

#include <iostream>

#include <unordered\_set>

using namespace std;

// Structure for the Node of the Binary Search Tree

struct Node {

int data; // Data of the node

Node\* left; // Pointer to the left child

Node\* right; // Pointer to the right child

// Constructor to create a new node

Node(int value) {

data = value;

left = right = nullptr;

}

};

// Class for Binary Search Tree

class BST {

private:

Node\* root; // Root node of the tree

// Helper function to insert a node into the Binary Search Tree

Node\* insertHelper(Node\* node, int value) {

if (node == nullptr) {

return new Node(value); // Create a new node if the position is empty

}

if (value < node->data) {

node->left = insertHelper(node->left, value); // Insert in the left subtree

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

node->right = insertHelper(node->right, value); // Insert in the right subtree

}

// If value is equal to node's data, it means it's a duplicate in the BST

return node;

}

// Helper function to perform In-order traversal and check for duplicates

bool inorderHelper(Node\* node, unordered\_set<int>& visited) {

if (node == nullptr) {

return true;

}

// Traverse left subtree

if (!inorderHelper(node->left, visited)) {

return false;

}

// Check if the current node's data is already in the set

if (visited.find(node->data) != visited.end()) {

return false; // Duplicate found

}

// Add the current node's data to the set

visited.insert(node->data);

// Traverse right subtree

return inorderHelper(node->right, visited);

}

public:

// Constructor to initialize the tree

BST() {

root = nullptr;

}

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

void insert(int value) {

root = insertHelper(root, value);

}

// Function to check if the BST contains duplicates

bool containsDuplicates() {

unordered\_set<int> visited;

return inorderHelper(root, visited);

}

};

int main() {

BST tree;

// Inserting nodes into the BST

tree.insert(50);

tree.insert(30);

tree.insert(20);

tree.insert(40);

tree.insert(70);

tree.insert(60);

tree.insert(80);

// Check if the tree contains duplicates

if (tree.containsDuplicates()) {

cout << "The tree does not contain duplicates." << endl;

} else {

cout << "The tree contains duplicates." << endl;

}

// Insert a duplicate value into the tree

tree.insert(40);

// Check again if the tree contains duplicates

if (tree.containsDuplicates()) {

cout << "The tree does not contain duplicates." << endl;

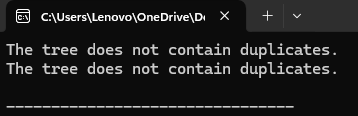
} else {

cout << "The tree contains duplicates." << endl;

}

return 0;

}



**Program 6**

#include <iostream>

using namespace std;

// Structure for the Node of the Binary Search Tree

struct Node {

int data;

Node\* left;

Node\* right;

Node(int value) {

data = value;

left = right = nullptr;

}

};

// Class for Binary Search Tree

class BST {

private:

Node\* root;

// Helper function to find the minimum value node in a subtree

Node\* findMin(Node\* node) {

while (node->left != nullptr) {

node = node->left;

}

return node;

}

// Helper function to delete a node from the BST

Node\* deleteNode(Node\* node, int value) {

if (node == nullptr) {

return node; // Node not found

}

// If the value to be deleted is smaller than the node's data,

// then it lies in the left subtree

if (value < node->data) {

node->left = deleteNode(node->left, value);

}

// If the value to be deleted is larger than the node's data,

// then it lies in the right subtree

else if (value > node->data) {

node->right = deleteNode(node->right, value);

}

// If the value is the same as the node's data, then this is the node to be deleted

else {

// Case 1: Node with only one child or no child

if (node->left == nullptr) {

Node\* temp = node->right;

delete node;

return temp; // Return the non-null child (or null if no children)

} else if (node->right == nullptr) {

Node\* temp = node->left;

delete node;

return temp; // Return the non-null child (or null if no children)

}

// Case 2: Node with two children

Node\* temp = findMin(node->right); // Get the in-order successor (smallest in the right subtree)

node->data = temp->data; // Copy the in-order successor's value to this node

node->right = deleteNode(node->right, temp->data); // Delete the in-order successor

}

return node;

}

// Helper function for In-order traversal (for printing the tree)

void inorderHelper(Node\* node) {

if (node != nullptr) {

inorderHelper(node->left);

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

inorderHelper(node->right);

}

}

public:

BST() {

root = nullptr;

}

// Public function to insert a node into the BST

void insert(int value) {

Node\* newNode = new Node(value);

if (root == nullptr) {

root = newNode;

} else {

Node\* current = root;

Node\* parent = nullptr;

while (current != nullptr) {

parent = current;

if (value < current->data) {

current = current->left;

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

current = current->right;

} else {

return; // Value already exists (BST doesn't allow duplicates)

}

}

if (value < parent->data) {

parent->left = newNode;

} else {

parent->right = newNode;

}

}

}

// Public function to delete a node from the BST

void deleteNode(int value) {

root = deleteNode(root, value);

}

// Public function to print the tree using In-order traversal

void inorder() {

inorderHelper(root);

cout << endl;

}

};

int main() {

BST tree;

// Insert some nodes

tree.insert(50);

tree.insert(30);

tree.insert(20);

tree.insert(40);

tree.insert(70);

tree.insert(60);

tree.insert(80);

// Print the tree before deletion

cout << "In-order traversal before deletion: ";

tree.inorder();

// Delete a leaf node

tree.deleteNode(20); // Deleting a leaf node

cout << "In-order traversal after deleting leaf node (20): ";

tree.inorder();

// Delete a node with one child

tree.deleteNode(30); // Deleting node with one child

cout << "In-order traversal after deleting node with one child (30): ";

tree.inorder();

// Delete a node with two children

tree.deleteNode(70); // Deleting node with two children

cout << "In-order traversal after deleting node with two children (70): ";

tree.inorder();

return 0;

}

