**Doubly Linked List**

including insertion, deletion, search, update, and reversal:

#include <iostream>

using namespace std;

class Node {

public:

int data;

Node\* next;

Node\* prev;

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

};

class DoublyLinkedList {

private:

Node\* head;

public:

DoublyLinkedList() : head(nullptr) {}

// Insert at the end

void insertEnd(int data) {

Node\* newNode = new Node(data);

if (!head) {

head = newNode;

} else {

Node\* temp = head;

while (temp->next) {

temp = temp->next;

}

temp->next = newNode;

newNode->prev = temp;

}

cout << "Inserted " << data << " at the end.\n";

}

// Insert at the beginning

void insertBeginning(int data) {

Node\* newNode = new Node(data);

if (!head) {

head = newNode;

} else {

newNode->next = head;

head->prev = newNode;

head = newNode;

}

cout << "Inserted " << data << " at the beginning.\n";

}

// Insert at a certain position

void insertAtPosition(int data, int position) {

if (position <= 0) {

cout << "Invalid position\n";

return;

}

Node\* newNode = new Node(data);

if (position == 1) {

insertBeginning(data);

return;

}

Node\* temp = head;

int count = 1;

while (temp && count < position - 1) {

temp = temp->next;

count++;

}

if (!temp) {

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

return;

}

newNode->next = temp->next;

if (temp->next) {

temp->next->prev = newNode;

}

temp->next = newNode;

newNode->prev = temp;

cout << "Inserted " << data << " at position " << position << ".\n";

}

// Delete from the end

void deleteEnd() {

if (!head) {

cout << "List is empty\n";

return;

}

if (!head->next) {

delete head;

head = nullptr;

} else {

Node\* temp = head;

while (temp->next) {

temp = temp->next;

}

temp->prev->next = nullptr;

delete temp;

}

cout << "Deleted node from the end.\n";

}

// Delete from the beginning

void deleteBeginning() {

if (!head) {

cout << "List is empty\n";

return;

}

Node\* temp = head;

head = head->next;

if (head) {

head->prev = nullptr;

}

delete temp;

cout << "Deleted node from the beginning.\n";

}

// Delete from a certain position

void deleteAtPosition(int position) {

if (position <= 0 || !head) {

cout << "Invalid position or list is empty\n";

return;

}

if (position == 1) {

deleteBeginning();

return;

}

Node\* temp = head;

int count = 1;

while (temp && count < position) {

temp = temp->next;

count++;

}

if (!temp) {

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

return;

}

if (temp->next) {

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

}

if (temp->prev) {

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

}

delete temp;

cout << "Deleted node at position " << position << ".\n";

}

// Search for an element

void search(int data) {

Node\* temp = head;

int pos = 1;

while (temp) {

if (temp->data == data) {

cout << "Found " << data << " at position " << pos << ".\n";

return;

}

temp = temp->next;

pos++;

}

cout << data << " not found in the list.\n";

}

// Display the list

void display() {

if (!head) {

cout << "List is empty\n";

return;

}

Node\* temp = head;

while (temp) {

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

temp = temp->next;

}

cout << "\n";

}

// Update a node's data

void update(int oldData, int newData) {

Node\* temp = head;

while (temp) {

if (temp->data == oldData) {

temp->data = newData;

cout << "Updated " << oldData << " to " << newData << ".\n";

return;

}

temp = temp->next;

}

cout << oldData << " not found in the list.\n";

}

// Reverse the list

void reverse() {

if (!head) {

cout << "List is empty\n";

return;

}

Node\* temp = nullptr;

Node\* current = head;

while (current) {

temp = current->prev;

current->prev = current->next;

current->next = temp;

current = current->prev;

}

if (temp) {

head = temp->prev;

}

cout << "List reversed.\n";

}

};

int main() {

DoublyLinkedList list;

int choice, data, position;

while (true) {

cout << "\n1. Insert at End\n";

cout << "2. Insert at Beginning\n";

cout << "3. Insert at Position\n";

cout << "4. Delete from End\n";

cout << "5. Delete from Beginning\n";

cout << "6. Delete at Position\n";

cout << "7. Search\n";

cout << "8. Display\n";

cout << "9. Update\n";

cout << "10. Reverse\n";

cout << "11. Exit\n";

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1:

cout << "Enter data: ";

cin >> data;

list.insertEnd(data);

break;

case 2:

cout << "Enter data: ";

cin >> data;

list.insertBeginning(data);

break;

case 3:

cout << "Enter data: ";

cin >> data;

cout << "Enter position: ";

cin >> position;

list.insertAtPosition(data, position);

break;

case 4:

list.deleteEnd();

break;

case 5:

list.deleteBeginning();

break;

case 6:

cout << "Enter position: ";

cin >> position;

list.deleteAtPosition(position);

break;

case 7:

cout << "Enter data to search: ";

cin >> data;

list.search(data);

break;

case 8:

list.display();

break;

case 9:

cout << "Enter old data: ";

cin >> data;

cout << "Enter new data: ";

cin >> position;

list.update(data, position);

break;

case 10:

list.reverse();

break;

case 11:

return 0;

default:

cout << "Invalid choice. Try again.\n";

}

}

return 0;

}

**C++ Implementation for Doubly Linked List Using Arrays:**

#include <iostream>

using namespace std;

#define MAX\_SIZE 100 // Maximum size of the list

class DoublyLinkedListArray {

private:

int data[MAX\_SIZE];

int next[MAX\_SIZE];

int prev[MAX\_SIZE];

int head;

int free;

public:

DoublyLinkedListArray() {

head = -1; // Initially, the list is empty

free = 0; // The first free position is at index 0

// Initialize the free list

for (int i = 0; i < MAX\_SIZE - 1; i++) {

next[i] = i + 1;

}

next[MAX\_SIZE - 1] = -1;

}

// Function to get a new free node index

int getFreeNode() {

if (free == -1) {

cout << "List is full, cannot allocate more nodes.\n";

return -1;

}

int newNode = free;

free = next[free]; // Move the free pointer to the next available index

return newNode;

}

// Insert at the end

void insertEnd(int val) {

int newNode = getFreeNode();

if (newNode == -1) return;

data[newNode] = val;

next[newNode] = -1;

prev[newNode] = -1;

if (head == -1) {

head = newNode; // If the list is empty, make the new node the head

} else {

int temp = head;

while (next[temp] != -1) {

temp = next[temp]; // Traverse to the last node

}

next[temp] = newNode;

prev[newNode] = temp;

}

cout << "Inserted " << val << " at the end.\n";

}

// Insert at the beginning

void insertBeginning(int val) {

int newNode = getFreeNode();

if (newNode == -1) return;

data[newNode] = val;

next[newNode] = head;

prev[newNode] = -1;

if (head != -1) {

prev[head] = newNode;

}

head = newNode;

cout << "Inserted " << val << " at the beginning.\n";

}

// Insert at a certain position

void insertAtPosition(int val, int position) {

if (position <= 0) {

cout << "Invalid position\n";

return;

}

int newNode = getFreeNode();

if (newNode == -1) return;

data[newNode] = val;

if (position == 1) {

insertBeginning(val);

return;

}

int temp = head;

int count = 1;

while (temp != -1 && count < position - 1) {

temp = next[temp];

count++;

}

if (temp == -1) {

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

return;

}

next[newNode] = next[temp];

prev[newNode] = temp;

if (next[temp] != -1) {

prev[next[temp]] = newNode;

}

next[temp] = newNode;

cout << "Inserted " << val << " at position " << position << ".\n";

}

// Delete from the end

void deleteEnd() {

if (head == -1) {

cout << "List is empty\n";

return;

}

int temp = head;

while (next[temp] != -1) {

temp = next[temp]; // Traverse to the last node

}

if (prev[temp] != -1) {

next[prev[temp]] = -1; // Remove the last node

} else {

head = -1; // If there's only one node

}

next[temp] = free;

free = temp; // Return the deleted node to the free list

cout << "Deleted node from the end.\n";

}

// Delete from the beginning

void deleteBeginning() {

if (head == -1) {

cout << "List is empty\n";

return;

}

int temp = head;

head = next[head];

if (head != -1) {

prev[head] = -1;

}

next[temp] = free;

free = temp; // Return the deleted node to the free list

cout << "Deleted node from the beginning.\n";

}

// Delete at a certain position

void deleteAtPosition(int position) {

if (position <= 0 || head == -1) {

cout << "Invalid position or list is empty\n";

return;

}

if (position == 1) {

deleteBeginning();

return;

}

int temp = head;

int count = 1;

while (temp != -1 && count < position) {

temp = next[temp];

count++;

}

if (temp == -1) {

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

return;

}

if (prev[temp] != -1) {

next[prev[temp]] = next[temp];

}

if (next[temp] != -1) {

prev[next[temp]] = prev[temp];

}

next[temp] = free;

free = temp; // Return the deleted node to the free list

cout << "Deleted node at position " << position << ".\n";

}

// Search for an element

void search(int val) {

int temp = head;

int pos = 1;

while (temp != -1) {

if (data[temp] == val) {

cout << "Found " << val << " at position " << pos << ".\n";

return;

}

temp = next[temp];

pos++;

}

cout << val << " not found in the list.\n";

}

// Display the list

void display() {

if (head == -1) {

cout << "List is empty\n";

return;

}

int temp = head;

while (temp != -1) {

cout << data[temp] << " ";

temp = next[temp];

}

cout << "\n";

}

// Update a node's data

void update(int oldData, int newData) {

int temp = head;

while (temp != -1) {

if (data[temp] == oldData) {

data[temp] = newData;

cout << "Updated " << oldData << " to " << newData << ".\n";

return;

}

temp = next[temp];

}

cout << oldData << " not found in the list.\n";

}

// Reverse the list

void reverse() {

if (head == -1) {

cout << "List is empty\n";

return;

}

int temp = -1;

int current = head;

while (current != -1) {

int nextNode = next[current];

next[current] = prev[current];

prev[current] = nextNode;

temp = current;

current = nextNode;

}

head = temp;

cout << "List reversed.\n";

}

};

int main() {

DoublyLinkedListArray list;

int choice, data, position;

while (true) {

cout << "\n1. Insert at End\n";

cout << "2. Insert at Beginning\n";

cout << "3. Insert at Position\n";

cout << "4. Delete from End\n";

cout << "5. Delete from Beginning\n";

cout << "6. Delete at Position\n";

cout << "7. Search\n";

cout << "8. Display\n";

cout << "9. Update\n";

cout << "10. Reverse\n";

cout << "11. Exit\n";

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1:

cout << "Enter data: ";

cin >> data;

list.insertEnd(data);

break;

case 2:

cout << "Enter data: ";

cin >> data;

list.insertBeginning(data);

break;

case 3:

cout << "Enter data: ";

cin >> data;

cout << "Enter position: ";

cin >> position;

list.insertAtPosition(data, position);

break;

case 4:

list.deleteEnd();

break;

case 5:

list.deleteBeginning();

break;

case 6:

cout << "Enter position: ";

cin >> position;

list.deleteAtPosition(position);

break;

case 7:

cout << "Enter data to search: ";

cin >> data;

list.search(data);

break;

case 8:

list.display();

break;

case 9:

cout << "Enter old data: ";

cin >> data;

cout << "Enter new data: ";

cin >> position;

list.update(data, position);

break;

case 10:

list.reverse();

break;

case 11:

return 0;

default:

cout << "Invalid choice. Try again.\n";

}

}

return 0;

}

**C++ Program for Doubly Linked List Using Pointers**

#include <iostream>

using namespace std;

// Define a structure for a node in the doubly linked list

struct Node {

int data;

Node\* next;

Node\* prev;

};

// Class for the Doubly Linked List

class DoublyLinkedList {

private:

Node\* head;

public:

// Constructor to initialize the list as empty

DoublyLinkedList() {

head = nullptr;

}

// Insert at the end

void insertEnd(int value) {

Node\* newNode = new Node();

newNode->data = value;

newNode->next = nullptr;

if (head == nullptr) { // If the list is empty, make the new node the head

newNode->prev = nullptr;

head = newNode;

} else {

Node\* temp = head;

while (temp->next != nullptr) {

temp = temp->next;

}

temp->next = newNode;

newNode->prev = temp;

}

cout << "Inserted " << value << " at the end.\n";

}

// Insert at the beginning

void insertBeginning(int value) {

Node\* newNode = new Node();

newNode->data = value;

newNode->prev = nullptr;

if (head == nullptr) { // If the list is empty, make the new node the head

newNode->next = nullptr;

head = newNode;

} else {

newNode->next = head;

head->prev = newNode;

head = newNode;

}

cout << "Inserted " << value << " at the beginning.\n";

}

// Insert at a certain position

void insertAtPosition(int value, int position) {

if (position <= 0) {

cout << "Invalid position\n";

return;

}

Node\* newNode = new Node();

newNode->data = value;

if (position == 1) {

insertBeginning(value);

return;

}

Node\* temp = head;

int count = 1;

while (temp != nullptr && count < position - 1) {

temp = temp->next;

count++;

}

if (temp == nullptr) {

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

return;

}

newNode->next = temp->next;

if (temp->next != nullptr) {

temp->next->prev = newNode;

}

temp->next = newNode;

newNode->prev = temp;

cout << "Inserted " << value << " at position " << position << ".\n";

}

// Delete from the end

void deleteEnd() {

if (head == nullptr) {

cout << "List is empty\n";

return;

}

Node\* temp = head;

while (temp->next != nullptr) {

temp = temp->next;

}

if (temp->prev != nullptr) {

temp->prev->next = nullptr;

} else {

head = nullptr;

}

delete temp;

cout << "Deleted node from the end.\n";

}

// Delete from the beginning

void deleteBeginning() {

if (head == nullptr) {

cout << "List is empty\n";

return;

}

Node\* temp = head;

head = head->next;

if (head != nullptr) {

head->prev = nullptr;

}

delete temp;

cout << "Deleted node from the beginning.\n";

}

// Delete from a certain position

void deleteAtPosition(int position) {

if (position <= 0 || head == nullptr) {

cout << "Invalid position or list is empty\n";

return;

}

if (position == 1) {

deleteBeginning();

return;

}

Node\* temp = head;

int count = 1;

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

temp = temp->next;

count++;

}

if (temp == nullptr) {

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

return;

}

if (temp->next != nullptr) {

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

}

if (temp->prev != nullptr) {

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

}

delete temp;

cout << "Deleted node at position " << position << ".\n";

}

// Search for an element

void search(int value) {

Node\* temp = head;

int position = 1;

while (temp != nullptr) {

if (temp->data == value) {

cout << "Found " << value << " at position " << position << ".\n";

return;

}

temp = temp->next;

position++;

}

cout << value << " not found in the list.\n";

}

// Display the list

void display() {

if (head == nullptr) {

cout << "List is empty\n";

return;

}

Node\* temp = head;

while (temp != nullptr) {

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

temp = temp->next;

}

cout << "\n";

}

// Update a node's data

void update(int oldData, int newData) {

Node\* temp = head;

while (temp != nullptr) {

if (temp->data == oldData) {

temp->data = newData;

cout << "Updated " << oldData << " to " << newData << ".\n";

return;

}

temp = temp->next;

}

cout << oldData << " not found in the list.\n";

}

// Reverse the list

void reverse() {

if (head == nullptr) {

cout << "List is empty\n";

return;

}

Node\* temp = nullptr;

Node\* current = head;

while (current != nullptr) {

temp = current->prev;

current->prev = current->next;

current->next = temp;

current = current->prev;

}

if (temp != nullptr) {

head = temp->prev;

}

cout << "List reversed.\n";

}

};

// Main function to demonstrate the operations

int main() {

DoublyLinkedList list;

int choice, data, position;

while (true) {

cout << "\n1. Insert at End\n";

cout << "2. Insert at Beginning\n";

cout << "3. Insert at Position\n";

cout << "4. Delete from End\n";

cout << "5. Delete from Beginning\n";

cout << "6. Delete at Position\n";

cout << "7. Search\n";

cout << "8. Display\n";

cout << "9. Update\n";

cout << "10. Reverse\n";

cout << "11. Exit\n";

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1:

cout << "Enter data: ";

cin >> data;

list.insertEnd(data);

break;

case 2:

cout << "Enter data: ";

cin >> data;

list.insertBeginning(data);

break;

case 3:

cout << "Enter data: ";

cin >> data;

cout << "Enter position: ";

cin >> position;

list.insertAtPosition(data, position);

break;

case 4:

list.deleteEnd();

break;

case 5:

list.deleteBeginning();

break;

case 6:

cout << "Enter position: ";

cin >> position;

list.deleteAtPosition(position);

break;

case 7:

cout << "Enter data to search: ";

cin >> data;

list.search(data);

break;

case 8:

list.display();

break;

case 9:

cout << "Enter old data: ";

cin >> data;

cout << "Enter new data: ";

cin >> position;

list.update(data, position);

break;

case 10:

list.reverse();

break;

case 11:

return 0;

default:

cout << "Invalid choice. Try again.\n";

}

}

return 0;

}

Top of Form

Bottom of Form