1. Singly Linked List

#include <iostream>

using namespace std;

class Node {

public:

int data;

Node\* next;

};

class LinkedList {

public:

LinkedList() {

head = nullptr;

}

void add(int value) {

Node\* new\_node = new Node();

new\_node->data = value;

new\_node->next = nullptr;

if (head == nullptr) {

head = new\_node;

} else {

Node\* current = head;

while (current->next != nullptr) {

current = current->next;

}

current->next = new\_node;

}

}

void display() {

Node\* current = head;

while (current != nullptr) {

cout << current->data << " ";

current = current->next;

}

cout << endl;

}

private:

Node\* head;

};

int main() {

LinkedList list;

list.add(1);

list.add(2);

list.add(3);

list.display();

return 0;

}

1-2

#include <iostream>

// Node structure

struct Node {

int data;

Node\* next;

};

class LinkedList {

private:

Node\* head;

public:

LinkedList() : head(nullptr) {}

// Insert at the beginning

void insertAtBeginning(int value) {

Node\* newNode = new Node;

newNode->data = value;

newNode->next = head;

head = newNode;

}

// Insert at the end

void insertAtEnd(int value) {

Node\* newNode = new Node;

newNode->data = value;

newNode->next = nullptr;

if (head == nullptr) {

head = newNode;

} else {

Node\* temp = head;

while (temp->next != nullptr) {

temp = temp->next;

}

temp->next = newNode;

}

}

// Delete first occurrence of a value

void deleteValue(int value) {

if (head == nullptr) return;

if (head->data == value) {

Node\* temp = head;

head = head->next;

delete temp;

return;

}

Node\* current = head;

Node\* previous = nullptr;

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

previous = current;

current = current->next;

}

if (current != nullptr) {

previous->next = current->next;

delete current;

}

}

// Display the list

void display() {

Node\* temp = head;

while (temp != nullptr) {

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

temp = temp->next;

}

std::cout << "nullptr" << std::endl;

}

// Destructor to free memory

~LinkedList() {

Node\* temp;

while (head != nullptr) {

temp = head;

head = head->next;

delete temp;

}

}

};

int main() {

LinkedList list;

// Insert at the beginning

list.insertAtBeginning(3);

list.insertAtBeginning(2);

list.insertAtBeginning(1);

std::cout << "Insert at the beginning: ";

list.display();

// Insert at the end

list.insertAtEnd(4);

list.insertAtEnd(5);

list.insertAtEnd(6);

std::cout << "Insert at the end: ";

list.display();

// Delete value

list.deleteValue(1);

list.deleteValue(4);

list.deleteValue(6);

std::cout << "Delete values (1, 4, 6): ";

list.display();

return 0;

}

1-3

#include <iostream>

template <typename T>

struct Node {

T data;

Node\* next;

};

template <typename T>

class LinkedList {

private:

Node<T>\* head;

public:

LinkedList() : head(nullptr) {}

void pushFront(T value) {

Node<T>\* newNode = new Node<T>;

newNode->data = value;

newNode->next = head;

head = newNode;

}

void pushBack(T value) {

Node<T>\* newNode = new Node<T>;

newNode->data = value;

newNode->next = nullptr;

if (head == nullptr) {

head = newNode;

} else {

Node<T>\* temp = head;

while (temp->next != nullptr) {

temp = temp->next;

}

temp->next = newNode;

}

}

void deleteValue(T value) {

if (head == nullptr) return;

if (head->data == value) {

Node<T>\* temp = head;

head = head->next;

delete temp;

return;

}

Node<T>\* current = head;

Node<T>\* previous = nullptr;

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

previous = current;

current = current->next;

}

if (current != nullptr) {

previous->next = current->next;

delete current;

}

}

void display() {

Node<T>\* temp = head;

while (temp != nullptr) {

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

temp = temp->next;

}

std::cout << "nullptr" << std::endl;

}

~LinkedList() {

Node<T>\* temp;

while (head != nullptr) {

temp = head;

head = head->next;

delete temp;

}

}

};

int main() {

LinkedList<int> list;

// Push front

list.pushFront(3);

list.pushFront(2);

list.pushFront(1);

std::cout << "Push front: ";

list.display();

// Push back

list.pushBack(4);

list.pushBack(5);

list.pushBack(6);

std::cout << "Push back: ";

list.display();

// Delete value

list.deleteValue(1);

list.deleteValue(4);

list.deleteValue(6);

std::cout << "Delete values (1, 4, 6): ";

list.display();

return 0;

}

1. Doubly Linked List

#include <iostream>

using namespace std;

class Node {

public:

int data;

Node\* prev;

Node\* next;

};

class LinkedList {

public:

LinkedList() {

head = nullptr;

tail = nullptr;

}

void add(int value) {

Node\* new\_node = new Node();

new\_node->data = value;

new\_node->prev = nullptr;

new\_node->next = nullptr;

if (head == nullptr) {

head = new\_node;

tail = new\_node;

} else {

tail->next = new\_node;

new\_node->prev = tail;

tail = new\_node;

}

}

void display() {

Node\* current = head;

while (current != nullptr) {

cout << current->data << " ";

current = current->next;

}

cout << endl;

}

private:

Node\* head;

Node\* tail;

};

int main() {

LinkedList list;

list.add(1);

list.add(2);

list.add(3);

list.display();

return 0;

}

2-1

#include <iostream>

template <typename T>

struct Node {

T data;

Node\* prev;

Node\* next;

};

template <typename T>

class DoublyLinkedList {

private:

Node<T>\* head;

public:

DoublyLinkedList() : head(nullptr) {}

void pushFront(T value) {

Node<T>\* newNode = new Node<T>;

newNode->data = value;

newNode->prev = nullptr;

newNode->next = head;

if (head != nullptr) {

head->prev = newNode;

}

head = newNode;

}

void pushBack(T value) {

Node<T>\* newNode = new Node<T>;

newNode->data = value;

newNode->next = nullptr;

if (head == nullptr) {

newNode->prev = nullptr;

head = newNode;

} else {

Node<T>\* temp = head;

while (temp->next != nullptr) {

temp = temp->next;

}

temp->next = newNode;

newNode->prev = temp;

}

}

void deleteValue(T value) {

if (head == nullptr) return;

Node<T>\* current = head;

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

current = current->next;

}

if (current == nullptr) return;

if (current->prev != nullptr) {

current->prev->next = current->next;

} else {

head = current->next;

}

if (current->next != nullptr) {

current->next->prev = current->prev;

}

delete current;

}

void display() {

Node<T>\* temp = head;

while (temp != nullptr) {

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

temp = temp->next;

}

std::cout << "nullptr" << std::endl;

}

~DoublyLinkedList() {

Node<T>\* temp;

while (head != nullptr) {

temp = head;

head = head->next;

delete temp;

}

}

};

int main() {

DoublyLinkedList<int> list;

// Push front

list.pushFront(3);

list.pushFront(2);

list.pushFront(1);

std::cout << "Push front: ";

list.display();

// Push back

list.pushBack(4);

list.pushBack(5);

list.pushBack(6);

std::cout << "Push back: ";

list.display();

// Delete value

list.deleteValue(1);

list.deleteValue(4);

list.deleteValue(6);

std::cout << "Delete values (1, 4, 6): ";

list.display();

return 0;

}

1. Circularly singly Link

#include <iostream>

using namespace std;

// Node class for the linked list

class Node {

public:

int data;

Node\* next;

Node(int d) {

data = d;

next = nullptr;

}

};

// Circularly Singly Linked List class

class CircularSinglyLinkedList {

private:

Node\* head;

Node\* tail;

public:

CircularSinglyLinkedList() {

head = nullptr;

tail = nullptr;

}

// Insert a new node at the end of the linked list

void insert(int data) {

Node\* new\_node = new Node(data);

if (head == nullptr) {

head = new\_node;

tail = new\_node;

} else {

tail->next = new\_node;

tail = new\_node;

}

tail->next = head;

}

// Traverse and print the linked list

void print() {

if (head == nullptr) {

cout << "Empty list" << endl;

} else {

Node\* current = head;

cout << "Circularly Singly Linked List: ";

do {

cout << current->data << " ";

current = current->next;

} while (current != head);

cout << endl;

}

}

};

// Main function to test the Circularly Singly Linked List

int main() {

CircularSinglyLinkedList list;

// Insert some nodes into the linked list

list.insert(10);

list.insert(20);

list.insert(30);

// Print the linked list

list.print();

return 0;

}

* 1. Circularly doubly Link List

#include <iostream>

using namespace std;

// Node class for the linked list

class Node {

public:

int data;

Node\* next;

Node\* prev;

Node(int d) {

data = d;

next = nullptr;

prev = nullptr;

}

};

// Circularly Doubly Linked List class

class CircularDoublyLinkedList {

private:

Node\* head;

Node\* tail;

public:

CircularDoublyLinkedList() {

head = nullptr;

tail = nullptr;

}

// Insert a new node at the end of the linked list

void insert(int data) {

Node\* new\_node = new Node(data);

if (head == nullptr) {

head = new\_node;

tail = new\_node;

} else {

tail->next = new\_node;

new\_node->prev = tail;

tail = new\_node;

}

tail->next = head;

head->prev = tail;

}

// Traverse and print the linked list

void print() {

if (head == nullptr) {

cout << "Empty list" << endl;

} else {

Node\* current = head;

cout << "Circularly Doubly Linked List: ";

do {

cout << current->data << " ";

current = current->next;

} while (current != head);

cout << endl;

}

}

};

// Main function to test the Circularly Doubly Linked List

int main() {

CircularDoublyLinkedList list;

// Insert some nodes into the linked list

list.insert(10);

list.insert(20);

list.insert(30);

// Print the linked list

list.print();

return 0;

}