National University of Computer and Emerging Sciences



Name: Muhammad Suleman

Roll #: 22F-3350

Lab#: 04

Section: BCS-4B

# Problem 01

#include <iostream>

using namespace std;

class Node

{

private:

public:

int data;

Node \*next;

Node \*prev;

Node()

: data(0), next(nullptr), prev(nullptr) {}

Node(int data)

: data(data), next(nullptr), prev(nullptr) {}

};

class LinkedList

{

private:

Node \*Head;

Node \*Tail;

public:

LinkedList()

: Head(nullptr), Tail(nullptr) {}

int getHead() { return Head->data; }

int getTail() { return Tail->data; }

// methods

void insertAtBegin(int data);

void insertAtEnd(int data);

void inerstAfter(int data, int position);

bool isEmpty();

void deleteAtBegin();

void deleteAtEnd();

void deleteBefore(int position);

void printList();

};

int main()

{

LinkedList list;

list.insertAtBegin(10);

list.insertAtBegin(20);

list.insertAtBegin(30);

list.insertAtBegin(40);

list.insertAtBegin(50);

list.insertAtEnd(100);

list.inerstAfter(60, 10);

list.printList();

cout << "Head: " << list.getHead() << "\nTail: " << list.getTail() << endl;

list.deleteAtBegin();

list.printList();

cout << "Head: " << list.getHead() << "\nTail: " << list.getTail() << endl;

list.deleteAtEnd();

list.printList();

list.deleteBefore(2);

list.printList();

cout << "Head: " << list.getHead() << "\nTail: " << list.getTail() << endl;

list.deleteBefore(5);

list.printList();

cout << "Head: " << list.getHead() << "\nTail: " << list.getTail() << endl;

list.deleteBefore(4);

list.printList();

cout << "Head: " << list.getHead() << "\nTail: " << list.getTail() << endl;

/\*int choice, key, data;

do

{

cout << "\nMenu Driven Doubly Linked List Implementation" << endl;

cout << "1) Insert at Beginning" << endl;

cout << "2) Insert After a Node" << endl;

cout << "3) Insert at End" << endl;

cout << "4) Delete at Beginning" << endl;

cout << "5) Delete Before a Node" << endl;

cout << "6) Delete at End" << endl;

cout << "7) Check if Empty" << endl;

cout << "8) Display List" << endl;

cout << "9) Exit" << endl;

cout << "Enter your choice: ";

cin >> choice;

switch (choice)

{

case 1:

cout << "Enter data to insert at beginning: ";

cin >> data;

list.insertAtBegin(data);

break;

case 2:

cout << "Enter key value after which to insert: ";

cin >> key;

cout << "Enter data to insert: ";

cin >> data;

list.inerstAfter(key, data);

break;

case 3:

cout << "Enter data to insert at end: ";

cin >> data;

list.insertAtEnd(data);

break;

case 4:

if (list.isEmpty())

{

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

}

else

{

list.deleteAtBegin();

}

break;

case 5:

if (list.isEmpty())

{

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

}

else

{

cout << "Enter key value before which to delete: ";

cin >> key;

list.deleteBefore(key);

}

break;

case 6:

if (list.isEmpty())

{

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

}

else

{

list.deleteAtEnd();

}

break;

case 7:

if (list.isEmpty())

{

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

}

else

{

cout << "List is not empty" << endl;

}

break;

case 8:

if (list.isEmpty())

{

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

}

else

{

list.printList();

}

break;

case 9:

cout << "Program Ended" << endl;

break;

default:

cout << "Invalid choice!" << endl;

}

} while (choice != 9);\*/

return 0;

}

void LinkedList::insertAtBegin(int data)

{

if (Head == nullptr)

{

Node \*newNode = new Node(data);

Head = newNode;

Tail = newNode;

}

else

{

Node \*newNode = new Node(data);

newNode->next = Head;

Head->prev = newNode;

Head = newNode;

}

}

void LinkedList::insertAtEnd(int data)

{

Node \*newNode = new Node(data);

if (Head == nullptr)

{

Head = newNode;

Tail = newNode;

}

else

{

newNode->prev = Tail;

Tail->next = newNode;

Tail = newNode;

}

}

void LinkedList::inerstAfter(int data, int position)

{

if (position <= 0)

{

cout << "Invalid position" << endl;

return;

}

Node \*curr = Head;

int count = 0;

while (curr != nullptr && count < position)

{

curr = curr->next;

count++;

}

if (curr == nullptr)

{

cout << "Invalid position" << endl;

return;

}

else

{

Node \*newNode = new Node(data);

newNode->next = curr->next;

newNode->prev = curr;

curr->next = newNode;

if (newNode->next == nullptr)

{

Tail = newNode;

}

}

}

void LinkedList::printList()

{

if (Head == nullptr)

{

cout << "Empty list" << endl;

return;

}

Node \*curr = Head;

while (curr != nullptr)

{

cout << curr->data << " -> ";

curr = curr->next;

}

cout << "NULL" << endl;

}

bool LinkedList::isEmpty()

{

if (Head == nullptr)

{

return true;

}

else

{

return false;

}

}

void LinkedList::deleteAtBegin()

{

// empty list case

if (Head == nullptr)

{

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

return;

}

// single node case

if (Head->next == nullptr)

{

delete Head;

Head = nullptr;

Tail = nullptr;

}

else

{

// multiple node case

Node \*temp = Head->next;

Head->next = nullptr;

Head = temp;

}

}

void LinkedList::deleteAtEnd()

{

// empty list case

if (Head == nullptr)

{

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

return;

}

// single node case

if (Head->next == nullptr)

{

delete Head;

Head = nullptr;

Tail = nullptr;

}

else

{

// multiple node case

Node \*temp = Tail->prev;

Tail->prev = nullptr;

delete Tail;

Tail = temp;

Tail->next = nullptr;

}

}

void LinkedList::deleteBefore(int position)

{

if (position <= 1)

{

cout << "Invalid position" << endl;

return;

}

// empty list case

if (Head == nullptr)

{

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

return;

}

Node \*prev = nullptr;

Node \*curr = Head;

int count = 0;

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

{

prev = curr;

curr = curr->next;

count++;

}

// if head to delete

if (prev == Head)

{

deleteAtBegin();

return;

}

// if tail to delete

if (curr)

{

if (curr->next == nullptr)

{

deleteAtEnd();

return;

}

else

{

// in between head and tail

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

curr->prev = prev->prev;

prev->prev = nullptr;

prev->next = nullptr;

delete prev;

}

}

else

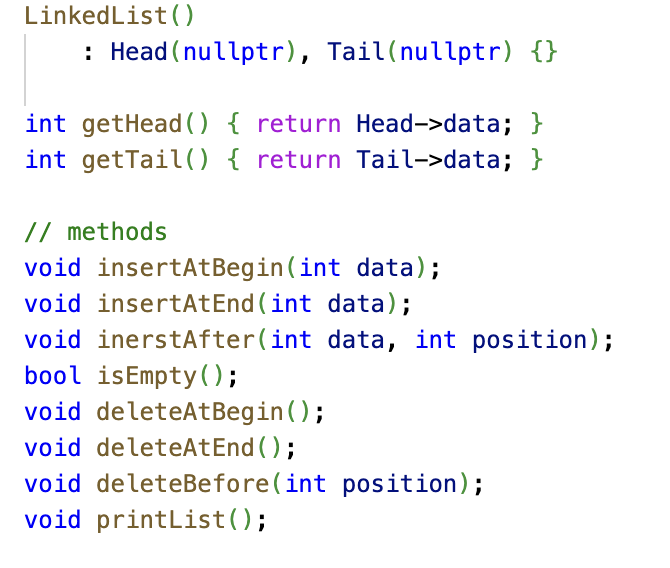
{

cout << "Invalid position" << endl;

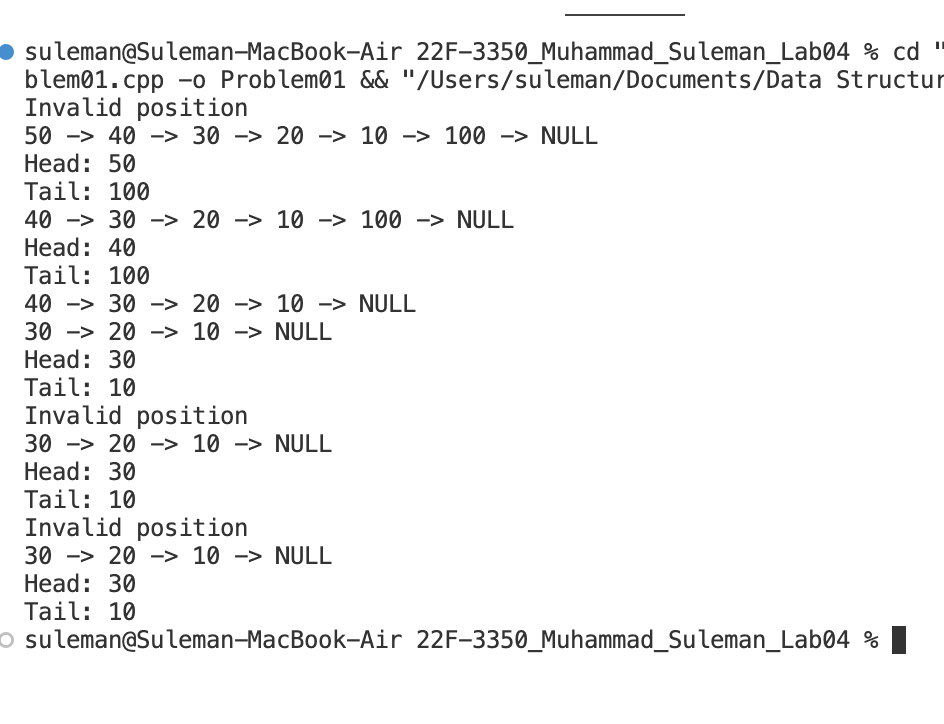
}

}

## Output









# Problem 02

#include <iostream>

using namespace std;

class Node

{

private:

public:

int data;

Node \*next;

Node \*prev;

Node()

: data(0), next(nullptr), prev(nullptr) {}

Node(int data)

: data(data), next(nullptr), prev(nullptr) {}

};

class LinkedList

{

private:

Node \*Head;

Node \*Tail;

public:

LinkedList()

: Head(nullptr), Tail(nullptr) {}

int getHead() { return Head->data; }

int getTail() { return Tail->data; }

// methods

void insertAtBegin(int data);

void insertAtEnd(int data);

void inerstAfter(int data, int position);

bool isEmpty();

void deleteAtBegin();

void deleteAtEnd();

void deleteBefore(int position);

void printList();

void sortList();

};

int main()

{

LinkedList list;

list.insertAtBegin(10);

list.insertAtBegin(20);

list.insertAtBegin(30);

list.insertAtBegin(40);

list.insertAtBegin(50);

list.insertAtEnd(100);

list.inerstAfter(60, 10);

cout << "Original List: ";

list.printList();

list.sortList();

cout << "Sorted List: ";

list.printList();

return 0;

}

void LinkedList::insertAtBegin(int data)

{

if (Head == nullptr)

{

Node \*newNode = new Node(data);

Head = newNode;

Tail = newNode;

}

else

{

Node \*newNode = new Node(data);

newNode->next = Head;

Head->prev = newNode;

Head = newNode;

}

}

void LinkedList::insertAtEnd(int data)

{

Node \*newNode = new Node(data);

if (Head == nullptr)

{

Head = newNode;

Tail = newNode;

}

else

{

newNode->prev = Tail;

Tail->next = newNode;

Tail = newNode;

}

}

void LinkedList::inerstAfter(int data, int position)

{

if (position <= 0)

{

cout << "Invalid position" << endl;

return;

}

Node \*curr = Head;

int count = 0;

while (curr != nullptr && count < position)

{

curr = curr->next;

count++;

}

if (curr == nullptr)

{

cout << "Invalid position" << endl;

return;

}

else

{

Node \*newNode = new Node(data);

newNode->next = curr->next;

newNode->prev = curr;

curr->next = newNode;

if (newNode->next == nullptr)

{

Tail = newNode;

}

}

}

void LinkedList::printList()

{

if (Head == nullptr)

{

cout << "Empty list" << endl;

return;

}

Node \*curr = Head;

while (curr != nullptr)

{

cout << curr->data << " -> ";

curr = curr->next;

}

cout << "NULL" << endl;

}

bool LinkedList::isEmpty()

{

if (Head == nullptr)

{

return true;

}

else

{

return false;

}

}

void LinkedList::deleteAtBegin()

{

// empty list case

if (Head == nullptr)

{

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

return;

}

// single node case

if (Head->next == nullptr)

{

delete Head;

Head = nullptr;

Tail = nullptr;

}

else

{

// multiple node case

Node \*temp = Head->next;

Head->next = nullptr;

Head = temp;

}

}

void LinkedList::deleteAtEnd()

{

// empty list case

if (Head == nullptr)

{

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

return;

}

// single node case

if (Head->next == nullptr)

{

delete Head;

Head = nullptr;

Tail = nullptr;

}

else

{

// multiple node case

Node \*temp = Tail->prev;

Tail->prev = nullptr;

delete Tail;

Tail = temp;

Tail->next = nullptr;

}

}

void LinkedList::deleteBefore(int position)

{

if (position <= 1)

{

cout << "Invalid position" << endl;

return;

}

// empty list case

if (Head == nullptr)

{

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

return;

}

Node \*prev = nullptr;

Node \*curr = Head;

int count = 0;

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

{

prev = curr;

curr = curr->next;

count++;

}

// if head to delete

if (prev == Head)

{

deleteAtBegin();

return;

}

// if tail to delete

if (curr)

{

if (curr->next == nullptr)

{

deleteAtEnd();

return;

}

else

{

// in between head and tail

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

curr->prev = prev->prev;

prev->prev = nullptr;

prev->next = nullptr;

delete prev;

}

}

else

{

cout << "Invalid position" << endl;

}

}

void LinkedList::sortList()

{

if (Head == nullptr || Head->next == nullptr)

{

return;

}

bool swapped;

Node \*curr;

do

{

swapped = false;

curr = Head;

while (curr->next != nullptr)

{

if (curr->data > curr->next->data)

{

// swap links of adjacent nodes

if (curr->prev != nullptr)

{

curr->prev->next = curr->next;

}

else

{

// if the current node is the head node

Head = curr->next;

}

curr->next->prev = curr->prev;

curr->prev = curr->next;

curr->next = curr->next->next;

curr->prev->next = curr;

if (curr->next != nullptr)

{

curr->next->prev = curr;

}

if (curr->next == nullptr)

{

// if the current node is the tail node

Tail = curr;

}

swapped = true;

}

else

{

curr = curr->next;

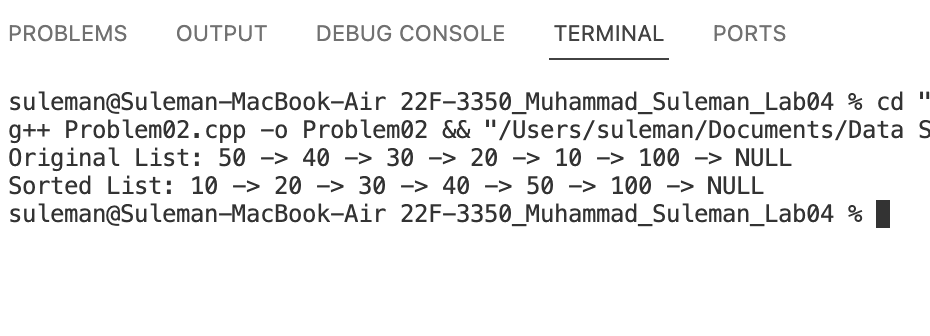
}

}

} while (swapped);

}

## Output



# Problem 03

#include <iostream>

using namespace std;

class Node

{

private:

public:

int data;

Node \*next;

Node \*prev;

Node()

: data(0), next(nullptr), prev(nullptr) {}

Node(int data)

: data(data), next(nullptr), prev(nullptr) {}

};

class LinkedList

{

private:

Node \*Head;

Node \*Tail;

public:

LinkedList()

: Head(nullptr), Tail(nullptr) {}

// copy constructor to store list in reverse order

LinkedList(const LinkedList &list)

{

Node \*current = list.Tail;

while (current != nullptr)

{

this->insertAtEnd(current->data);

current = current->prev;

}

}

int getHead() { return Head->data; }

int getTail() { return Tail->data; }

// methods

void insertAtBegin(int data);

void insertAtEnd(int data);

void inerstAfter(int data, int position);

bool isEmpty();

void deleteAtBegin();

void deleteAtEnd();

void deleteBefore(int position);

void printList();

};

int main()

{

LinkedList list1;

list1.insertAtBegin(10);

list1.insertAtBegin(20);

list1.insertAtBegin(30);

list1.insertAtBegin(40);

list1.insertAtBegin(50);

list1.insertAtBegin(60);

list1.insertAtBegin(70);

list1.insertAtBegin(80);

list1.insertAtBegin(90);

list1.insertAtBegin(100);

cout << "List 1" << endl;

cout << "Head: " << list1.getHead() << "\nTail: " << list1.getTail() << endl;

list1.printList();

LinkedList list2(list1);

cout << "List 2" << endl;

cout << "Head: " << list2.getHead() << "\nTail: " << list2.getTail() << endl;

list2.printList();

system("pause");

return 0;

}

void LinkedList::insertAtBegin(int data)

{

if (Head == nullptr)

{

Node \*newNode = new Node(data);

Head = newNode;

Tail = newNode;

}

else

{

Node \*newNode = new Node(data);

newNode->next = Head;

Head->prev = newNode;

Head = newNode;

}

}

void LinkedList::insertAtEnd(int data)

{

Node \*newNode = new Node(data);

if (Head == nullptr)

{

Head = newNode;

Tail = newNode;

}

else

{

newNode->prev = Tail;

Tail->next = newNode;

Tail = newNode;

}

}

void LinkedList::inerstAfter(int data, int position)

{

if (position <= 0)

{

cout << "Invalid position" << endl;

return;

}

Node \*curr = Head;

int count = 0;

while (curr != nullptr && count < position)

{

curr = curr->next;

count++;

}

if (curr == nullptr)

{

cout << "Invalid position" << endl;

return;

}

else

{

Node \*newNode = new Node(data);

newNode->next = curr->next;

newNode->prev = curr;

curr->next = newNode;

if (newNode->next == nullptr)

{

Tail = newNode;

}

}

}

void LinkedList::printList()

{

if (Head == nullptr)

{

cout << "Empty list" << endl;

return;

}

Node \*curr = Head;

while (curr != nullptr)

{

cout << curr->data << " -> ";

curr = curr->next;

}

cout << "NULL" << endl;

}

bool LinkedList::isEmpty()

{

if (Head == nullptr)

{

return true;

}

else

{

return false;

}

}

void LinkedList::deleteAtBegin()

{

// empty list case

if (Head == nullptr)

{

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

return;

}

// single node case

if (Head->next == nullptr)

{

delete Head;

Head = nullptr;

Tail = nullptr;

}

else

{

// multiple node case

Node \*temp = Head->next;

Head->next = nullptr;

Head = temp;

}

}

void LinkedList::deleteAtEnd()

{

// empty list case

if (Head == nullptr)

{

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

return;

}

// single node case

if (Head->next == nullptr)

{

delete Head;

Head = nullptr;

Tail = nullptr;

}

else

{

// multiple node case

Node \*temp = Tail->prev;

Tail->prev = nullptr;

delete Tail;

Tail = temp;

Tail->next = nullptr;

}

}

void LinkedList::deleteBefore(int position)

{

if (position <= 1)

{

cout << "Invalid position" << endl;

return;

}

// empty list case

if (Head == nullptr)

{

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

return;

}

Node \*prev = nullptr;

Node \*curr = Head;

int count = 0;

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

{

prev = curr;

curr = curr->next;

count++;

}

// if head to delete

if (prev == Head)

{

deleteAtBegin();

return;

}

// if tail to delete

if (curr)

{

if (curr->next == nullptr)

{

deleteAtEnd();

return;

}

else

{

// in between head and tail

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

curr->prev = prev->prev;

prev->prev = nullptr;

prev->next = nullptr;

delete prev;

}

}

else

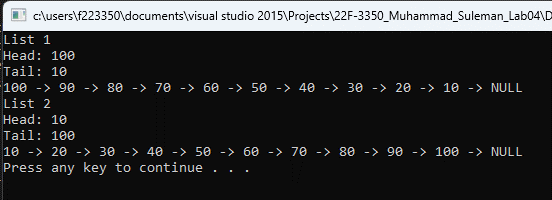
{

cout << "Invalid position" << endl;

}

}

## Output



# Problem 04

#include <iostream>

using namespace std;

class Node

{

private:

public:

int data;

Node \*next;

Node \*prev;

Node()

: data(0), next(nullptr), prev(nullptr) {}

Node(int data)

: data(data), next(nullptr), prev(nullptr) {}

};

class LinkedList

{

private:

Node \*Head;

Node \*Tail;

public:

LinkedList()

: Head(nullptr), Tail(nullptr) {}

int getHead() { return Head->data; }

int getTail() { return Tail->data; }

// methods

void insertAtBegin(int data);

void insertAtEnd(int data);

void inerstAfter(int data, int position);

bool isEmpty();

void deleteAtBegin();

void deleteAtEnd();

void deleteBefore(int position);

void printList();

void convertToCircular();

void printListCircular();

};

int main()

{

LinkedList list;

list.insertAtBegin(10);

list.insertAtBegin(20);

list.insertAtBegin(30);

list.insertAtBegin(40);

list.insertAtBegin(50);

list.insertAtEnd(100);

list.inerstAfter(60, 10);

list.printList();

cout << "Head: " << list.getHead() << "\nTail: " << list.getTail() << endl;

list.convertToCircular();

cout << "Circular List: ";

list.printListCircular();

return 0;

}

void LinkedList::insertAtBegin(int data)

{

if (Head == nullptr)

{

Node \*newNode = new Node(data);

Head = newNode;

Tail = newNode;

}

else

{

Node \*newNode = new Node(data);

newNode->next = Head;

Head->prev = newNode;

Head = newNode;

}

}

void LinkedList::insertAtEnd(int data)

{

Node \*newNode = new Node(data);

if (Head == nullptr)

{

Head = newNode;

Tail = newNode;

}

else

{

newNode->prev = Tail;

Tail->next = newNode;

Tail = newNode;

}

}

void LinkedList::inerstAfter(int data, int position)

{

if (position <= 0)

{

cout << "Invalid position" << endl;

return;

}

Node \*curr = Head;

int count = 0;

while (curr != nullptr && count < position)

{

curr = curr->next;

count++;

}

if (curr == nullptr)

{

cout << "Invalid position" << endl;

return;

}

else

{

Node \*newNode = new Node(data);

newNode->next = curr->next;

newNode->prev = curr;

curr->next = newNode;

if (newNode->next == nullptr)

{

Tail = newNode;

}

}

}

void LinkedList::printList()

{

if (Head == nullptr)

{

cout << "Empty list" << endl;

return;

}

Node \*curr = Head;

while (curr != nullptr)

{

cout << curr->data << " -> ";

curr = curr->next;

}

cout << "NULL" << endl;

}

void LinkedList::printListCircular()

{

if (Head == nullptr)

{

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

return;

}

Node \*temp = Head;

while (temp->next != Head)

{

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

temp = temp->next;

}

cout << "Head" << endl;

}

bool LinkedList::isEmpty()

{

if (Head == nullptr)

{

return true;

}

else

{

return false;

}

}

void LinkedList::deleteAtBegin()

{

// empty list case

if (Head == nullptr)

{

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

return;

}

// single node case

if (Head->next == nullptr)

{

delete Head;

Head = nullptr;

Tail = nullptr;

}

else

{

// multiple node case

Node \*temp = Head->next;

Head->next = nullptr;

Head = temp;

}

}

void LinkedList::deleteAtEnd()

{

// empty list case

if (Head == nullptr)

{

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

return;

}

// single node case

if (Head->next == nullptr)

{

delete Head;

Head = nullptr;

Tail = nullptr;

}

else

{

// multiple node case

Node \*temp = Tail->prev;

Tail->prev = nullptr;

delete Tail;

Tail = temp;

Tail->next = nullptr;

}

}

void LinkedList::deleteBefore(int position)

{

if (position <= 1)

{

cout << "Invalid position" << endl;

return;

}

// empty list case

if (Head == nullptr)

{

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

return;

}

Node \*prev = nullptr;

Node \*curr = Head;

int count = 0;

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

{

prev = curr;

curr = curr->next;

count++;

}

// if head to delete

if (prev == Head)

{

deleteAtBegin();

return;

}

// if tail to delete

if (curr)

{

if (curr->next == nullptr)

{

deleteAtEnd();

return;

}

else

{

// in between head and tail

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

curr->prev = prev->prev;

prev->prev = nullptr;

prev->next = nullptr;

delete prev;

}

}

else

{

cout << "Invalid position" << endl;

}

}

void LinkedList::convertToCircular()

{

if (Head == nullptr)

{

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

return;

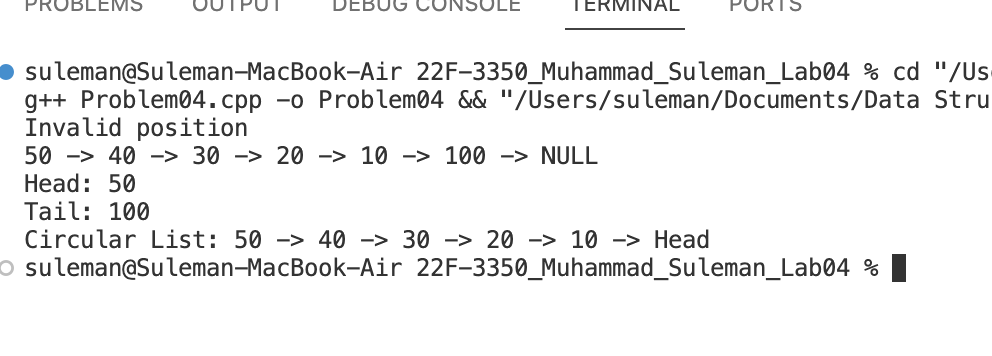
}

Tail->next = Head;

Head->prev = Tail;

}

## Output



# Problem 05

#include <iostream>

#include <string>

using namespace std;

class Node

{

private:

public:

int data;

Node \*next;

Node()

: data(0), next(nullptr) {}

Node(int data)

: data(data), next(nullptr) {}

};

class CircularLinkedList

{

Node \*Head;

public:

CircularLinkedList()

: Head(nullptr) {}

int getHead() { return Head->data; }

void insertAtBegin(int data);

void deleteAtEnd();

void printList();

void menu()

{

cout << "CircularLinkedList" << endl;

cout << "1) Insert at beginnig" << endl;

cout << "2) Delete at end" << endl;

cout << "Any other key to exit" << endl;

}

};

int main()

{

CircularLinkedList list;

string choice;

do

{

cout << "Menu:" << endl;

cout << "1) Insert at the beginning" << endl;

cout << "2) Delete at the end" << endl;

cout << "3) Exit" << endl;

cout << "Enter choice: ";

getline(cin, choice);

if (choice == "1")

{

int data;

cout << "Enter data: ";

cin >> data;

cin.ignore();

list.insertAtBegin(data);

list.printList();

}

else if (choice == "2")

{

list.deleteAtEnd();

list.printList();

}

else if (choice != "3")

{

cout << "Invalid choice" << endl;

cout << "Please enter a valid option" << endl;

}

} while (choice != "3");

cout << "Exiting program" << endl;

return 0;

}

void CircularLinkedList::insertAtBegin(int data)

{

if (Head == nullptr)

{

Node \*newNode = new Node(data);

newNode->next = newNode;

Head = newNode;

}

else

{

Node \*newNode = new Node(data);

Node \*temp = Head;

while (temp->next != Head)

{

temp = temp->next;

}

newNode->next = Head;

temp->next = newNode;

Head = newNode;

}

}

void CircularLinkedList::deleteAtEnd()

{

if (Head == nullptr)

{

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

return;

}

if (Head->next == Head)

{

Head->next = nullptr;

delete Head;

Head = nullptr;

}

else

{

Node \*prev = nullptr;

Node \*curr = Head;

while (curr->next != Head)

{

prev = curr;

curr = curr->next;

}

prev->next = curr->next;

curr->next = nullptr;

delete curr;

}

}

void CircularLinkedList::printList()

{

if (Head == nullptr)

{

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

return;

}

else

{

Node \*temp = Head;

do

{

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

temp = temp->next;

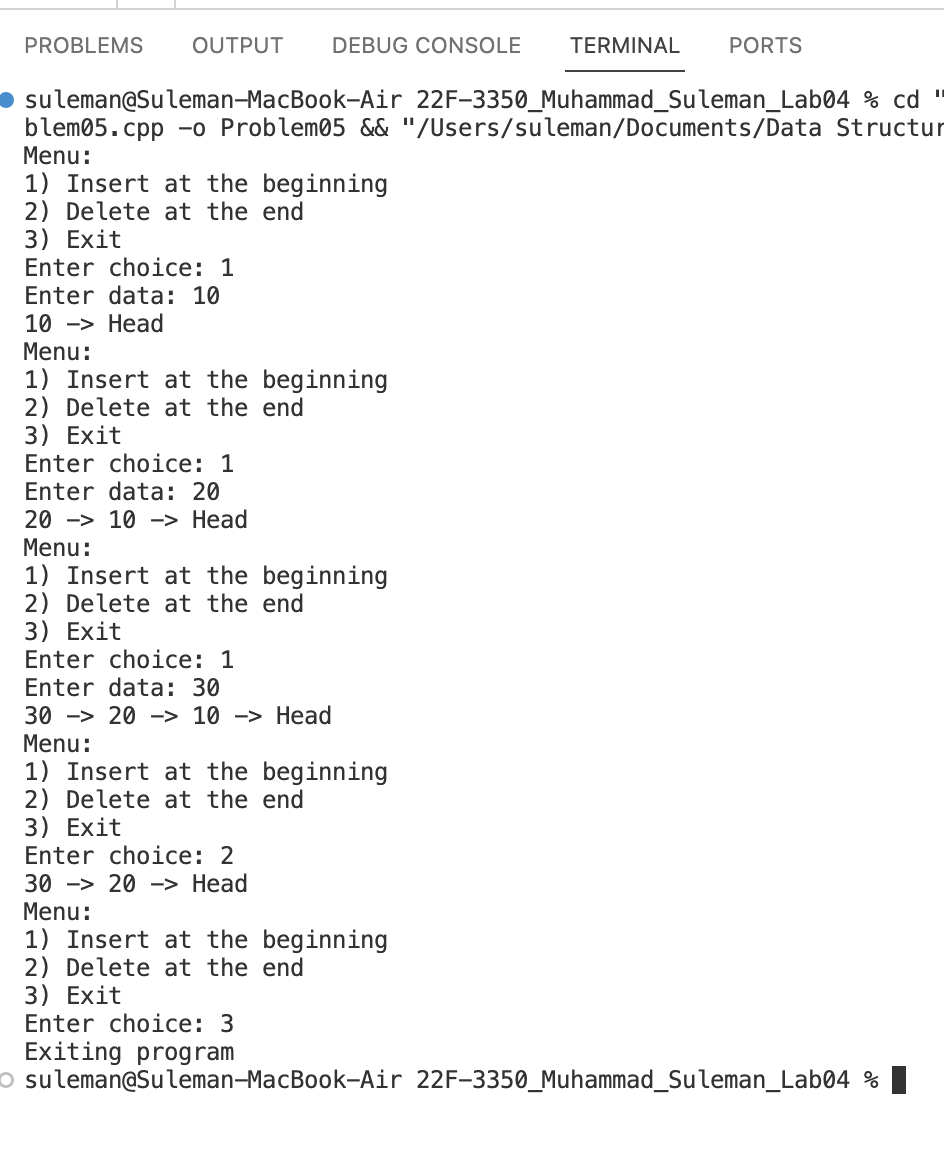
} while (temp != Head);

cout << "Head" << endl;

}

}

## Output



# Problem 06

#include <iostream>

using namespace std;

class Node

{

private:

public:

int data;

Node \*next;

Node \*prev;

Node()

: data(0), next(nullptr), prev(nullptr) {}

Node(int data)

: data(data), next(nullptr), prev(nullptr) {}

};

class LinkedList

{

private:

Node \*Head;

Node \*Tail;

public:

LinkedList()

: Head(nullptr), Tail(nullptr) {}

// copy constructor to store list in reverse order

LinkedList(const LinkedList &list)

{

Node \*current = list.Tail;

while (current != nullptr)

{

this->insertAtEnd(current->data);

current = current->prev;

}

}

int getHead() { return Head->data; }

int getTail() { return Tail->data; }

// methods

void insertAtBegin(int data);

void insertAtEnd(int data);

void inerstAfter(int data, int position);

bool isEmpty();

void deleteAtBegin();

void deleteAtEnd();

void deleteBefore(int position);

void stroeEvenOdd(LinkedList &even, LinkedList &odd);

void printList();

};

int main()

{

LinkedList dataList;

dataList.insertAtBegin(1);

dataList.insertAtBegin(2);

dataList.insertAtBegin(3);

dataList.insertAtBegin(4);

dataList.insertAtBegin(5);

dataList.insertAtBegin(6);

dataList.insertAtBegin(7);

dataList.insertAtBegin(8);

dataList.insertAtBegin(9);

dataList.insertAtBegin(10);

cout << "Data List" << endl;

cout << "Head: " << dataList.getHead() << endl;

cout << "Tail: " << dataList.getTail() << endl;

dataList.printList();

LinkedList evenList;

LinkedList oddList;

dataList.stroeEvenOdd(evenList, oddList);

cout << "\nEven List" << endl;

cout << "Head: " << evenList.getHead() << endl;

cout << "Tail: " << evenList.getTail() << endl;

evenList.printList();

cout << "\nOdd List" << endl;

cout << "Head: " << oddList.getHead() << endl;

cout << "Tail: " << oddList.getTail() << endl;

oddList.printList();

system("pause");

return 0;

}

void LinkedList::insertAtBegin(int data)

{

if (Head == nullptr)

{

Node \*newNode = new Node(data);

Head = newNode;

Tail = newNode;

}

else

{

Node \*newNode = new Node(data);

newNode->next = Head;

Head->prev = newNode;

Head = newNode;

}

}

void LinkedList::insertAtEnd(int data)

{

Node \*newNode = new Node(data);

if (Head == nullptr)

{

Head = newNode;

Tail = newNode;

}

else

{

newNode->prev = Tail;

Tail->next = newNode;

Tail = newNode;

}

}

void LinkedList::inerstAfter(int data, int position)

{

if (position <= 0)

{

cout << "Invalid position" << endl;

return;

}

Node \*curr = Head;

int count = 0;

while (curr != nullptr && count < position)

{

curr = curr->next;

count++;

}

if (curr == nullptr)

{

cout << "Invalid position" << endl;

return;

}

else

{

Node \*newNode = new Node(data);

newNode->next = curr->next;

newNode->prev = curr;

curr->next = newNode;

if (newNode->next == nullptr)

{

Tail = newNode;

}

}

}

void LinkedList::printList()

{

if (Head == nullptr)

{

cout << "Empty list" << endl;

return;

}

Node \*curr = Head;

while (curr != nullptr)

{

cout << curr->data << " -> ";

curr = curr->next;

}

cout << "NULL" << endl;

}

bool LinkedList::isEmpty()

{

if (Head == nullptr)

{

return true;

}

else

{

return false;

}

}

void LinkedList::deleteAtBegin()

{

// empty list case

if (Head == nullptr)

{

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

return;

}

// single node case

if (Head->next == nullptr)

{

delete Head;

Head = nullptr;

Tail = nullptr;

}

else

{

// multiple node case

Node \*temp = Head->next;

Head->next = nullptr;

Head = temp;

}

}

void LinkedList::deleteAtEnd()

{

// empty list case

if (Head == nullptr)

{

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

return;

}

// single node case

if (Head->next == nullptr)

{

delete Head;

Head = nullptr;

Tail = nullptr;

}

else

{

// multiple node case

Node \*temp = Tail->prev;

Tail->prev = nullptr;

delete Tail;

Tail = temp;

Tail->next = nullptr;

}

}

void LinkedList::deleteBefore(int position)

{

if (position <= 1)

{

cout << "Invalid position" << endl;

return;

}

// empty list case

if (Head == nullptr)

{

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

return;

}

Node \*prev = nullptr;

Node \*curr = Head;

int count = 0;

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

{

prev = curr;

curr = curr->next;

count++;

}

// if head to delete

if (prev == Head)

{

deleteAtBegin();

return;

}

// if tail to delete

if (curr)

{

if (curr->next == nullptr)

{

deleteAtEnd();

return;

}

else

{

// in between head and tail

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

curr->prev = prev->prev;

prev->prev = nullptr;

prev->next = nullptr;

delete prev;

}

}

else

{

cout << "Invalid position" << endl;

}

}

void LinkedList::stroeEvenOdd(LinkedList &even, LinkedList &odd)

{

Node \*temp = Head;

while (temp != nullptr)

{

if (temp->data % 2 == 0)

{

even.insertAtEnd(temp->data);

}

else

{

odd.insertAtEnd(temp->data);

}

temp = temp->next;

}

}

## Output

