****

**Talha Abdullah Bangyal**

**22F-3194**

**BS (AI) - 3A**

**LAB 3**

**Task 1**

#include<iostream>

using namespace std;

struct Node {

int data;

Node\* next;

};

class SinglyLinkedList {

public:

SinglyLinkedList();

void Insert\_at\_front(int val);

void Insert\_at\_end(int val);

void Insert\_at\_index(int index, int val);

void Delete\_at\_front();

void Delete\_at\_end();

void Delete\_at\_index(int index);

bool Isempty();

void Print();

~SinglyLinkedList();

private:

Node\* head;

};

SinglyLinkedList::SinglyLinkedList() {

head = NULL;

}

bool SinglyLinkedList::Isempty() {

return head == NULL;

}

void SinglyLinkedList::Insert\_at\_front(int val) {

if (Isempty()) {

head = new Node;

head->data = val;

head->next = NULL;

}

else {

Node\* temp = new Node;

temp->data = val;

temp->next = head;

head = temp;

}

}

void SinglyLinkedList::Insert\_at\_end(int val) {

if (Isempty()) {

head = new Node;

head->data = val;

head->next = NULL;

}

else {

Node\* current = head;

while (current->next != NULL) {

current = current->next;

}

Node\* temp = new Node;

temp->data = val;

temp->next = NULL;

current->next = temp;

}

}

void SinglyLinkedList::Insert\_at\_index(int index, int val) {

if (Isempty()) {

if (index == 0) {

head = new Node;

head->data = val;

head->next = NULL;

}

else {

cout << "Index out of bounds" << endl;

return;

}

}

else {

Node\* current = head;

int currentIndex = 0;

while (current->next != NULL && currentIndex < index - 1) {

current = current->next;

currentIndex++;

}

if (currentIndex == index - 1) {

Node\* newNode = new Node;

newNode->data = val;

newNode->next = current->next;

current->next = newNode;

}

else {

cout << "Index out of bounds" << endl;

}

}

}

void SinglyLinkedList::Delete\_at\_front() {

if (Isempty()) {

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

return;

}

else {

Node\* temp = head;

head = head->next;

delete temp;

}

}

void SinglyLinkedList::Delete\_at\_end() {

if (Isempty()) {

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

return;

}

else if (head->next == NULL) {

delete head;

head = NULL;

}

else {

Node\* current = head;

while (current->next->next != NULL) {

current = current->next;

}

delete current->next;

current->next = NULL;

}

}

void SinglyLinkedList::Delete\_at\_index(int index) {

if (Isempty()) {

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

return;

}

else if (index == 0) {

Node\* temp = head;

head = head->next;

delete temp;

}

else {

Node\* current = head;

int currentIndex = 0;

while (current->next != NULL && currentIndex < index - 1) {

current = current->next;

currentIndex++;

}

if (currentIndex == index - 1) {

Node\* temp = current->next;

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

delete temp;

}

else {

cout << "Index out of bounds" << endl;

}

}

}

void SinglyLinkedList::Print() {

Node\* current = head;

while (current != NULL) {

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

current = current->next;

}

cout << endl;

}

SinglyLinkedList::~SinglyLinkedList() {

Node\* current = head;

while (current != NULL) {

Node\* next = current->next;

delete current;

current = next;

}

}

int main() {

SinglyLinkedList SSL;

int choice, val, index;

while (true) {

cout << "------------------------Menu--------------------------" << endl;

cout << "1. Insert at front" << endl;

cout << "2. Insert at end" << endl;

cout << "3. Insert at index" << endl;

cout << "4. Delete at front" << endl;

cout << "5. Delete at end" << endl;

cout << "6. Delete at index" << endl;

cout << "7. Print list" << endl;

cout << "0. Exit" << endl;

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1:

cout << "Enter value to insert at front: ";

cin >> val;

SSL.Insert\_at\_front(val);

break;

case 2:

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

cin >> val;

SSL.Insert\_at\_end(val);

break;

case 3:

cout << "Enter index: ";

cin >> index;

cout << "Enter value to insert: ";

cin >> val;

SSL.Insert\_at\_index(index, val);

break;

case 4:

SSL.Delete\_at\_front();

break;

case 5:

SSL.Delete\_at\_end();

break;

case 6:

cout << "Enter index: ";

cin >> index;

SSL.Delete\_at\_index(index);

break;

case 7:

SSL.Print();

break;

case 0:

return 0;

default:

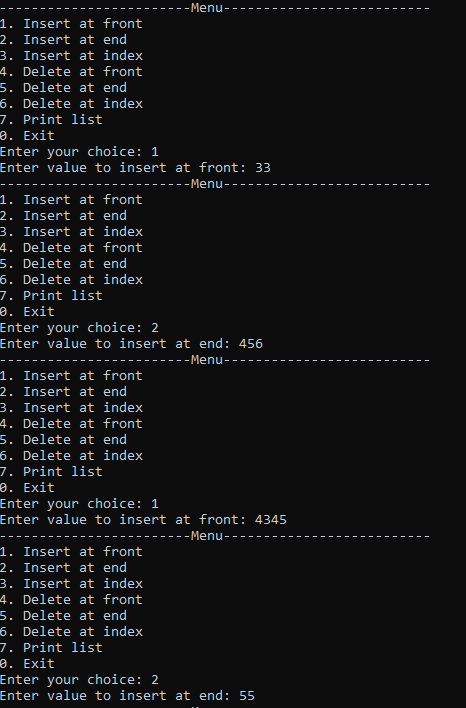
cout << "Invalid choice. Try again." << endl;

}

}

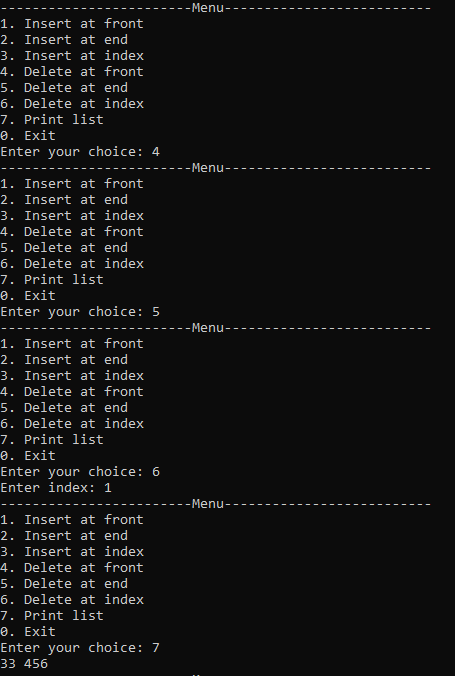
return 0;

}



A screenshot of a computer program

Description automatically generated



**Task 2**

#include<iostream>

using namespace std;

struct Node {

int data;

Node\* next;

Node\* prev;

};

class DoublyLinkedList {

public:

DoublyLinkedList();

void Insert\_at\_front(int val);

void Insert\_at\_end(int val);

void Insert\_at\_index(int index, int val);

void Delete\_at\_front();

void Delete\_at\_end();

void Delete\_at\_index(int index);

bool Isempty();

void Print();

~DoublyLinkedList();

private:

Node\* head;

Node\* tail;

};

DoublyLinkedList::DoublyLinkedList() {

head = NULL;

tail = NULL;

}

bool DoublyLinkedList::Isempty() {

return head == NULL;

}

void DoublyLinkedList::Insert\_at\_front(int val) {

if (Isempty()) {

head = new Node;

head->data = val;

head->next = NULL;

head->prev = NULL;

tail = head;

}

else {

Node\* temp = new Node;

temp->data = val;

temp->next = head;

head->prev = temp;

temp->prev = NULL;

head = temp;

}

}

void DoublyLinkedList::Insert\_at\_end(int val) {

if (Isempty()) {

head = new Node;

head->data = val;

head->next = NULL;

head->prev = NULL;

tail = head;

}

else {

Node\* temp = new Node;

temp->data = val;

temp->next = NULL;

temp->prev = tail;

tail->next = temp;

tail = temp;

}

}

void DoublyLinkedList::Insert\_at\_index(int index, int val) {

if (Isempty()) {

if (index == 0) {

head = new Node;

head->data = val;

head->next = NULL;

head->prev = NULL;

tail = head;

}

else {

cout << "Index out of bounds" << endl;

return;

}

}

else {

Node\* current = head;

int currentIndex = 0;

while (current->next != NULL && currentIndex < index - 1) {

current = current->next;

currentIndex++;

}

if (currentIndex == index - 1) {

Node\* newNode = new Node;

newNode->data = val;

newNode->next = current->next;

newNode->prev = current;

current->next = newNode;

if (newNode->next != NULL) {

newNode->next->prev = newNode;

}

else {

tail = newNode;

}

}

else {

cout << "Index out of bounds" << endl;

}

}

}

void DoublyLinkedList::Delete\_at\_front() {

if (Isempty()) {

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

return;

}

else {

Node\* temp = head;

head = head->next;

if (head != NULL) {

head->prev = NULL;

}

else {

tail = NULL;

}

delete temp;

}

}

void DoublyLinkedList::Delete\_at\_end() {

if (Isempty()) {

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

return;

}

else if (head->next == NULL) {

delete head;

head = NULL;

tail = NULL;

}

else {

Node\* current = head;

while (current->next->next != NULL) {

current = current->next;

}

Node\* temp = current->next;

current->next = NULL;

tail = current;

delete temp;

}

}

void DoublyLinkedList::Delete\_at\_index(int index) {

if (Isempty()) {

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

return;

}

if (index < 0) {

cout << "Index cannot be negative" << endl;

return;

}

if (index == 0) {

Node\* temp = head;

head = head->next;

if (head != nullptr) {

head->prev = nullptr;

}

else {

tail = nullptr;

}

delete temp;

}

else {

Node\* current = head;

int currentIndex = 0;

while (current != nullptr && currentIndex < index - 1) {

current = current->next;

currentIndex++;

}

if (current == nullptr || current->next == nullptr) {

cout << "Index out of bounds" << endl;

return;

}

Node\* temp = current->next;

current->next = temp->next;

if (temp->next != nullptr) {

temp->next->prev = current;

}

else {

tail = current;

}

delete temp;

}

}

void DoublyLinkedList::Print() {

Node\* current = head;

while (current != NULL) {

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

current = current->next;

}

cout << endl;

}

DoublyLinkedList::~DoublyLinkedList() {

Node\* current = head;

while (current != NULL) {

Node\* next = current->next;

delete current;

current = next;

}

}

int main() {

DoublyLinkedList DLL;

int choice, val, index;

while (true) {

cout << "------------------------Menu--------------------------" << endl;

cout << "1. Insert at front" << endl;

cout << "2. Insert at end" << endl;

cout << "3. Insert at index" << endl;

cout << "4. Delete at front" << endl;

cout << "5. Delete at end" << endl;

cout << "6. Delete at index" << endl;

cout << "7. Print list" << endl;

cout << "0. Exit" << endl;

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1:

cout << "Enter value to insert at front: ";

cin >> val;

DLL.Insert\_at\_front(val);

break;

case 2:

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

cin >> val;

DLL.Insert\_at\_end(val);

break;

case 3:

cout << "Enter index: ";

cin >> index;

cout << "Enter value to insert: ";

cin >> val;

DLL.Insert\_at\_index(index, val);

break;

case 4:

DLL.Delete\_at\_front();

break;

case 5:

DLL.Delete\_at\_end();

break;

case 6:

cout << "Enter index: ";

cin >> index;

DLL.Delete\_at\_index(index);

break;

case 7:

DLL.Print();

break;

case 0:

return 0;

default:

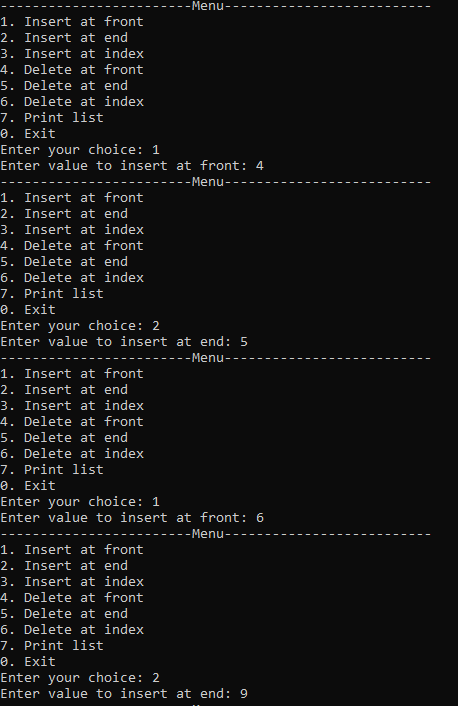
cout << "Invalid choice. Try again." << endl;

}

}

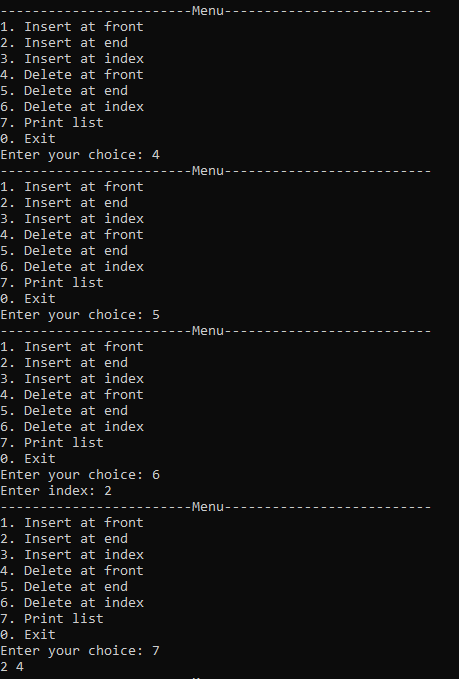
return 0;

}



A screenshot of a computer program

Description automatically generated



**Task 3**

#include<iostream>

using namespace std;

struct Node {

int data;

Node\* next;

Node\* prev;

};

class DoublyLinkedList {

public:

DoublyLinkedList();

void Insert\_at\_front(int val);

void Insert\_at\_end(int val);

void Insert\_at\_index(int index, int val);

void Delete\_at\_front();

void Delete\_at\_end();

void Delete\_at\_index(int index);

void reverseDoubly();

bool Isempty();

void Print();

~DoublyLinkedList();

private:

Node\* head;

Node\* tail;

};

DoublyLinkedList::DoublyLinkedList() {

head = NULL;

tail = NULL;

}

bool DoublyLinkedList::Isempty() {

return head == NULL;

}

void DoublyLinkedList::Insert\_at\_front(int val) {

if (Isempty()) {

head = new Node;

head->data = val;

head->next = NULL;

head->prev = NULL;

tail = head;

}

else {

Node\* temp = new Node;

temp->data = val;

temp->next = head;

head->prev = temp;

temp->prev = NULL;

head = temp;

}

}

void DoublyLinkedList::Insert\_at\_end(int val) {

if (Isempty()) {

head = new Node;

head->data = val;

head->next = NULL;

head->prev = NULL;

tail = head;

}

else {

Node\* temp = new Node;

temp->data = val;

temp->next = NULL;

temp->prev = tail;

tail->next = temp;

tail = temp;

}

}

void DoublyLinkedList::Insert\_at\_index(int index, int val) {

if (Isempty()) {

if (index == 0) {

head = new Node;

head->data = val;

head->next = NULL;

head->prev = NULL;

tail = head;

}

else {

cout << "Index out of bounds" << endl;

return;

}

}

else {

Node\* current = head;

int currentIndex = 0;

while (current->next != NULL && currentIndex < index - 1) {

current = current->next;

currentIndex++;

}

if (currentIndex == index - 1) {

Node\* newNode = new Node;

newNode->data = val;

newNode->next = current->next;

newNode->prev = current;

current->next = newNode;

if (newNode->next != NULL) {

newNode->next->prev = newNode;

}

else {

tail = newNode;

}

}

else {

cout << "Index out of bounds" << endl;

}

}

}

void DoublyLinkedList::Delete\_at\_front() {

if (Isempty()) {

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

return;

}

else {

Node\* temp = head;

head = head->next;

if (head != NULL) {

head->prev = NULL;

}

else {

tail = NULL;

}

delete temp;

}

}

void DoublyLinkedList::Delete\_at\_end() {

if (Isempty()) {

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

return;

}

else if (head->next == NULL) {

delete head;

head = NULL;

tail = NULL;

}

else {

Node\* current = head;

while (current->next->next != NULL) {

current = current->next;

}

Node\* temp = current->next;

current->next = NULL;

tail = current;

delete temp;

}

}

void DoublyLinkedList::Delete\_at\_index(int index) {

if (Isempty()) {

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

return;

}

if (index < 0) {

cout << "Index cannot be negative" << endl;

return;

}

if (index == 0) {

Node\* temp = head;

head = head->next;

if (head != nullptr) {

head->prev = nullptr;

}

else {

tail = nullptr;

}

delete temp;

}

else {

Node\* current = head;

int currentIndex = 0;

while (current != nullptr && currentIndex < index - 1) {

current = current->next;

currentIndex++;

}

if (current == nullptr || current->next == nullptr) {

cout << "Index out of bounds" << endl;

return;

}

Node\* temp = current->next;

current->next = temp->next;

if (temp->next != nullptr) {

temp->next->prev = current;

}

else {

tail = current;

}

delete temp;

}

}

void DoublyLinkedList::reverseDoubly() {

Node\* current = head;

Node\* temp = NULL;

while (current != NULL) {

temp = current->next;

current->next = current->prev;

current->prev = temp;

current = temp;

}

temp = head;

head = tail;

tail = temp;

}

void DoublyLinkedList::Print() {

Node\* current = head;

while (current != NULL) {

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

current = current->next;

}

cout << endl;

}

DoublyLinkedList::~DoublyLinkedList() {

Node\* current = head;

while (current != NULL) {

Node\* next = current->next;

delete current;

current = next;

}

}

int main() {

DoublyLinkedList DLL;

int choice, val, index;

while (true) {

cout << "------------------------Menu--------------------------" << endl;

cout << "1. Insert at front" << endl;

cout << "2. Insert at end" << endl;

cout << "3. Insert at index" << endl;

cout << "4. Delete at front" << endl;

cout << "5. Delete at end" << endl;

cout << "6. Delete at index" << endl;

cout << "7. reverse list" << endl;

cout << "8. Print list" << endl;

cout << "0. Exit" << endl;

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1:

cout << "Enter value to insert at front: ";

cin >> val;

DLL.Insert\_at\_front(val);

break;

case 2:

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

cin >> val;

DLL.Insert\_at\_end(val);

break;

case 3:

cout << "Enter index: ";

cin >> index;

cout << "Enter value to insert: ";

cin >> val;

DLL.Insert\_at\_index(index, val);

break;

case 4:

DLL.Delete\_at\_front();

break;

case 5:

DLL.Delete\_at\_end();

break;

case 6:

cout << "Enter index: ";

cin >> index;

DLL.Delete\_at\_index(index);

break;

case 7:

DLL.reverseDoubly();

break;

case 8:

DLL.Print();

break;

case 0:

return 0;

default:

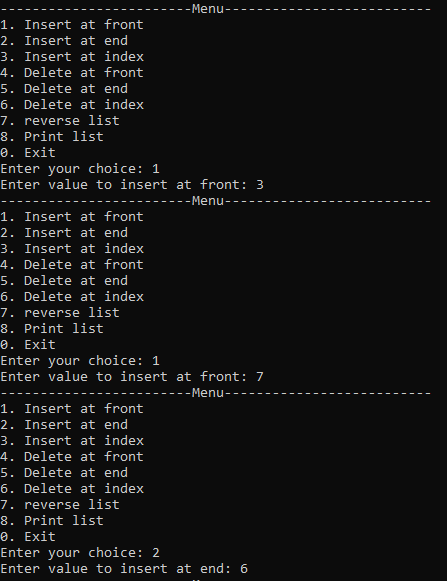
cout << "Invalid choice. Try again." << endl;

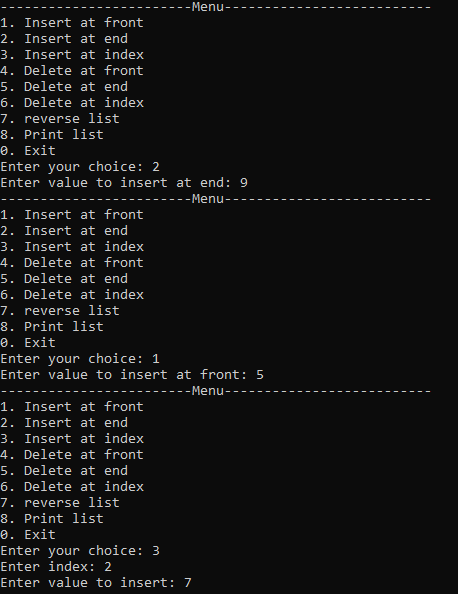
}

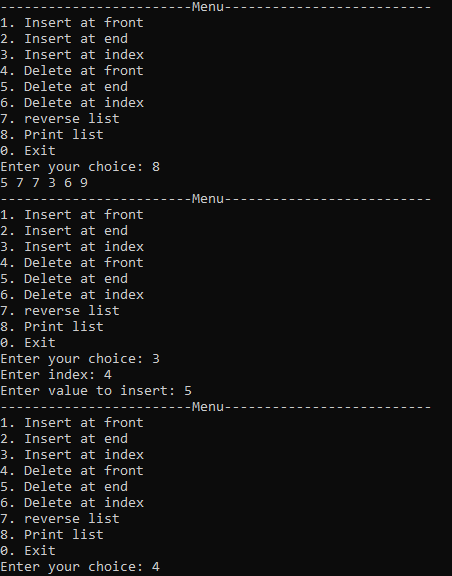
}

return 0;

}







A screenshot of a computer program

Description automatically generated

A screen shot of a computer

Description automatically generated

**Task 4**

#include<iostream>

using namespace std;

struct Node {

int data;

Node\* next;

};

class SinglyLinkedList {

public:

SinglyLinkedList();

void Insert\_at\_front(int val);

void Insert\_at\_end(int val);

void Insert\_at\_index(int index, int val);

void Delete\_at\_front();

void Delete\_at\_end();

void Delete\_at\_index(int index);

void rotateSingly(int K);

bool Isempty();

void Print();

~SinglyLinkedList();

private:

Node\* head;

};

SinglyLinkedList::SinglyLinkedList() {

head = NULL;

}

bool SinglyLinkedList::Isempty() {

return head == NULL;

}

void SinglyLinkedList::Insert\_at\_front(int val) {

if (Isempty()) {

head = new Node;

head->data = val;

head->next = NULL;

}

else {

Node\* temp = new Node;

temp->data = val;

temp->next = head;

head = temp;

}

}

void SinglyLinkedList::Insert\_at\_end(int val) {

if (Isempty()) {

head = new Node;

head->data = val;

head->next = NULL;

}

else {

Node\* current = head;

while (current->next != NULL) {

current = current->next;

}

Node\* temp = new Node;

temp->data = val;

temp->next = NULL;

current->next = temp;

}

}

void SinglyLinkedList::Insert\_at\_index(int index, int val) {

if (Isempty()) {

if (index == 0) {

head = new Node;

head->data = val;

head->next = NULL;

}

else {

cout << "Index out of bounds" << endl;

return;

}

}

else {

Node\* current = head;

int currentIndex = 0;

while (current->next != NULL && currentIndex < index - 1) {

current = current->next;

currentIndex++;

}

if (currentIndex == index - 1) {

Node\* newNode = new Node;

newNode->data = val;

newNode->next = current->next;

current->next = newNode;

}

else {

cout << "Index out of bounds" << endl;

}

}

}

void SinglyLinkedList::Delete\_at\_front() {

if (Isempty()) {

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

return;

}

else {

Node\* temp = head;

head = head->next;

delete temp;

}

}

void SinglyLinkedList::Delete\_at\_end() {

if (Isempty()) {

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

return;

}

else if (head->next == NULL) {

delete head;

head = NULL;

}

else {

Node\* current = head;

while (current->next->next != NULL) {

current = current->next;

}

delete current->next;

current->next = NULL;

}

}

void SinglyLinkedList::Delete\_at\_index(int index) {

if (Isempty()) {

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

return;

}

else if (index == 0) {

Node\* temp = head;

head = head->next;

delete temp;

}

else {

Node\* current = head;

int currentIndex = 0;

while (current->next != NULL && currentIndex < index - 1) {

current = current->next;

currentIndex++;

}

if (currentIndex == index - 1) {

Node\* temp = current->next;

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

delete temp;

}

else {

cout << "Index out of bounds" << endl;

}

}

}

void SinglyLinkedList::rotateSingly(int K) {

if (Isempty()) {

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

return;

}

int length = 0;

Node\* current = head;

while (current != NULL) {

length++;

current = current->next;

}

K = K % length; // handle cases where K is greater than the length of the list

if (K == 0) {

return; // no rotation needed

}

Node\* newTail = head;

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

newTail = newTail->next;

}

Node\* newHead = newTail->next;

newTail->next = NULL;

Node\* current2 = newHead;

while (current2->next != NULL) {

current2 = current2->next;

}

current2->next = head;

head = newHead;

}

void SinglyLinkedList::Print() {

Node\* current = head;

while (current != NULL) {

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

current = current->next;

}

cout << endl;

}

SinglyLinkedList::~SinglyLinkedList() {

Node\* current = head;

while (current != NULL) {

Node\* next = current->next;

delete current;

current = next;

}

}

int main() {

SinglyLinkedList SSL;

int choice, val, index, K;

while (true) {

cout << "------------------------Menu--------------------------" << endl;

cout << "1. Insert at front" << endl;

cout << "2. Insert at end" << endl;

cout << "3. Insert at index" << endl;

cout << "4. Delete at front" << endl;

cout << "5. Delete at end" << endl;

cout << "6. Delete at index" << endl;

cout << "7. Rotate list" << endl;

cout << "8. Print list" << endl;

cout << "0. Exit" << endl;

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1:

cout << "Enter value to insert at front: ";

cin >> val;

SSL.Insert\_at\_front(val);

break;

case 2:

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

cin >> val;

SSL.Insert\_at\_end(val);

break;

case 3:

cout << "Enter index: ";

cin >> index;

cout << "Enter value to insert: ";

cin >> val;

SSL.Insert\_at\_index(index, val);

break;

case 4:

SSL.Delete\_at\_front();

break;

case 5:

SSL.Delete\_at\_end();

break;

case 6:

cout << "Enter index: ";

cin >> index;

SSL.Delete\_at\_index(index);

break;

case 7:

cout << "Enter Value of to rotate Singly linked List: ";

cin >> K;

SSL.rotateSingly(K);

break;

case 8:

SSL.Print();

break;

case 0:

return 0;

default:

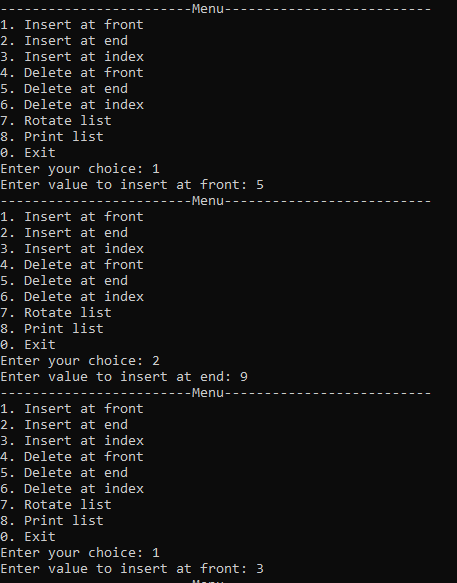
cout << "Invalid choice. Try again." << endl;

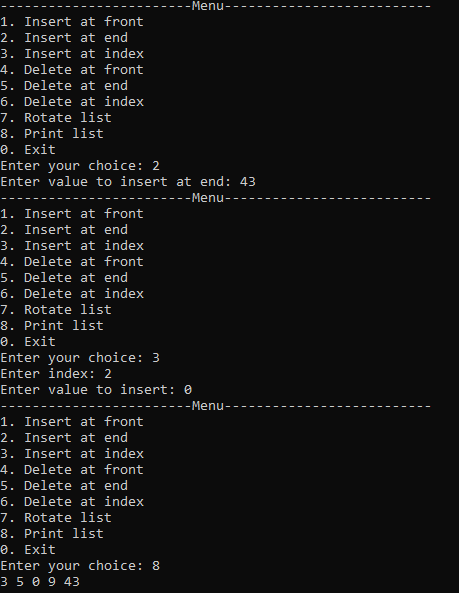
}

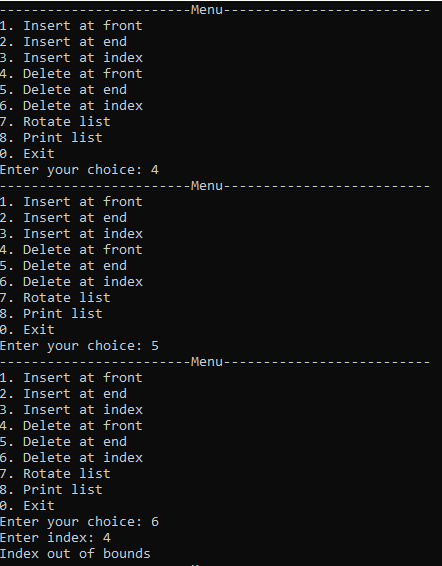
}

return 0;

}

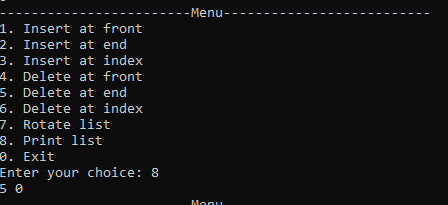






A screenshot of a computer program

Description automatically generated



**Task 5**

**#include<iostream>**

**using namespace std;**

**struct Node {**

**int data;**

**Node\* next;**

**Node\* prev;**

**};**

**class DoublyLinkedList {**

**public:**

**DoublyLinkedList();**

**void Insert\_at\_front(int val);**

**void Insert\_at\_end(int val);**

**void Insert\_at\_index(int index, int val);**

**void Delete\_at\_front();**

**void Delete\_at\_end();**

**void Delete\_at\_index(int index);**

**int middleDoubly();**

**bool Isempty();**

**void Print();**

**~DoublyLinkedList();**

**private:**

**Node\* head;**

**Node\* tail;**

**};**

**DoublyLinkedList::DoublyLinkedList() {**

**head = NULL;**

**tail = NULL;**

**}**

**bool DoublyLinkedList::Isempty() {**

**return head == NULL;**

**}**

**void DoublyLinkedList::Insert\_at\_front(int val) {**

**if (Isempty()) {**

**head = new Node;**

**head->data = val;**

**head->next = NULL;**

**head->prev = NULL;**

**tail = head;**

**}**

**else {**

**Node\* temp = new Node;**

**temp->data = val;**

**temp->next = head;**

**head->prev = temp;**

**temp->prev = NULL;**

**head = temp;**

**}**

**}**

**void DoublyLinkedList::Insert\_at\_end(int val) {**

**if (Isempty()) {**

**head = new Node;**

**head->data = val;**

**head->next = NULL;**

**head->prev = NULL;**

**tail = head;**

**}**

**else {**

**Node\* temp = new Node;**

**temp->data = val;**

**temp->next = NULL;**

**temp->prev = tail;**

**tail->next = temp;**

**tail = temp;**

**}**

**}**

**void DoublyLinkedList::Insert\_at\_index(int index, int val) {**

**if (Isempty()) {**

**if (index == 0) {**

**head = new Node;**

**head->data = val;**

**head->next = NULL;**

**head->prev = NULL;**

**tail = head;**

**}**

**else {**

**cout << "Index out of bounds" << endl;**

**return;**

**}**

**}**

**else {**

**Node\* current = head;**

**int currentIndex = 0;**

**while (current->next != NULL && currentIndex < index - 1) {**

**current = current->next;**

**currentIndex++;**

**}**

**if (currentIndex == index - 1) {**

**Node\* newNode = new Node;**

**newNode->data = val;**

**newNode->next = current->next;**

**newNode->prev = current;**

**current->next = newNode;**

**if (newNode->next != NULL) {**

**newNode->next->prev = newNode;**

**}**

**else {**

**tail = newNode;**

**}**

**}**

**else {**

**cout << "Index out of bounds" << endl;**

**}**

**}**

**}**

**void DoublyLinkedList::Delete\_at\_front() {**

**if (Isempty()) {**

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

**return;**

**}**

**else {**

**Node\* temp = head;**

**head = head->next;**

**if (head != NULL) {**

**head->prev = NULL;**

**}**

**else {**

**tail = NULL;**

**}**

**delete temp;**

**}**

**}**

**void DoublyLinkedList::Delete\_at\_end() {**

**if (Isempty()) {**

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

**return;**

**}**

**else if (head->next == NULL) {**

**delete head;**

**head = NULL;**

**tail = NULL;**

**}**

**else {**

**Node\* current = head;**

**while (current->next->next != NULL) {**

**current = current->next;**

**}**

**Node\* temp = current->next;**

**current->next = NULL;**

**tail = current;**

**delete temp;**

**}**

**}**

**void DoublyLinkedList::Delete\_at\_index(int index) {**

**if (Isempty()) {**

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

**return;**

**}**

**else if (index == 0) {**

**Node\* temp = head;**

**head = head->next;**

**if (head != NULL) {**

**head->prev = NULL;**

**}**

**else {**

**tail = NULL;**

**}**

**delete temp;**

**}**

**else {**

**Node\* current = head;**

**int currentIndex = 0;**

**while (current->next != NULL && currentIndex < index - 1) {**

**current = current->next;**

**currentIndex++;**

**}**

**if (currentIndex == index - 1) {**

**Node\* temp = current->next;**

**current->next = current->next->next;**

**if (current->next != NULL) {**

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

**}**

**else {**

**tail = current;**

**}**

**delete temp;**

**}**

**else {**

**cout << "Index out of bounds" << endl;**

**}**

**}**

**}**

**void DoublyLinkedList::Print() {**

**Node\* current = head;**

**while (current != NULL) {**

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

**current = current->next;**

**}**

**cout << endl;**

**}**

**DoublyLinkedList::~DoublyLinkedList() {**

**Node\* current = head;**

**while (current != NULL) {**

**Node\* next = current->next;**

**delete current;**

**current = next;**

**}**

**}**

**int DoublyLinkedList::middleDoubly() {**

**if (Isempty()) {**

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

**return -1;**

**}**

**Node\* slow = head;**

**Node\* fast = head;**

**while (fast != NULL && fast->next != NULL) {**

**slow = slow->next;**

**fast = fast->next->next;**

**}**

**return slow->data;**

**}**

**int main() {**

**DoublyLinkedList DLL;**

**int choice, val, index;**

**while (true) {**

**cout << "------------------------Menu--------------------------" << endl;**

**cout << "Press 1 to Insert at front" << endl;**

**cout << "Press 2 to Insert at end" << endl;**

**cout << "Press 3 to Insert at index" << endl;**

**cout << "Press 4 to Delete at front" << endl;**

**cout << "Press 5 to Delete at end" << endl;**

**cout << "Press 6 to Delete at index" << endl;**

**cout << "Press 7 to Print list" << endl;**

**cout << "Press 8 to Find middle element" << endl; // New option**

**cout << "Press 0 to Exit" << endl;**

**cout << "Enter your choice: ";**

**cin >> choice;**

**switch (choice) {**

**case 1:**

**cout << "Enter value to insert at front: ";**

**cin >> val;**

**DLL.Insert\_at\_front(val);**

**break;**

**case 2:**

**cout << "Enter value to insert at end: ";**

**cin >> val;**

**DLL.Insert\_at\_end(val);**

**break;**

**case 3:**

**cout << "Enter index: ";**

**cin >> index;**

**cout << "Enter value to insert: ";**

**cin >> val;**

**DLL.Insert\_at\_index(index, val);**

**break;**

**case 4:**

**DLL.Delete\_at\_front();**

**break;**

**case 5:**

**DLL.Delete\_at\_end();**

**break;**

**case 6:**

**cout << "Enter index: ";**

**cin >> index;**

**DLL.Delete\_at\_index(index);**

**break;**

**case 7:**

**DLL.Print();**

**break;**

**case 8:**

**int middle = DLL.middleDoubly();**

**if (middle != -1) {**

**cout << "The middle element is: " << middle << endl;**

**}**

**break;**

**case 0:**

**return 0;**

**default:**

**cout << "Invalid choice. Try again." << endl;**

**}**

**}**

**return 0;**

**}**

**Task 6**

#include<iostream>

using namespace std;

struct Node {

int data;

Node\* next;

Node\* prev;

};

class DoublyLinkedList {

public:

DoublyLinkedList();

void Insert\_at\_front(int val);

void Insert\_at\_end(int val);

void Insert\_at\_index(int index, int val);

void Delete\_at\_front();

void Delete\_at\_end();

void Delete\_at\_index(int index);

bool Isempty();

void Print();

bool palindromeDoubly();

~DoublyLinkedList();

private:

Node\* head;

Node\* tail;

};

DoublyLinkedList::DoublyLinkedList() {

head = NULL;

tail = NULL;

}

bool DoublyLinkedList::Isempty() {

return head == NULL;

}

void DoublyLinkedList::Insert\_at\_front(int val) {

if (Isempty()) {

head = new Node;

head->data = val;

head->next = NULL;

head->prev = NULL;

tail = head;

}

else {

Node\* temp = new Node;

temp->data = val;

temp->next = head;

head->prev = temp;

temp->prev = NULL;

head = temp;

}

}

void DoublyLinkedList::Insert\_at\_end(int val) {

if (Isempty()) {

head = new Node;

head->data = val;

head->next = NULL;

head->prev = NULL;

tail = head;

}

else {

Node\* temp = new Node;

temp->data = val;

temp->next = NULL;

temp->prev = tail;

tail->next = temp;

tail = temp;

}

}

void DoublyLinkedList::Insert\_at\_index(int index, int val) {

if (Isempty()) {

if (index == 0) {

head = new Node;

head->data = val;

head->next = NULL;

head->prev = NULL;

tail = head;

}

else {

cout << "Index out of bounds" << endl;

return;

}

}

else {

Node\* current = head;

int currentIndex = 0;

while (current->next != NULL && currentIndex < index - 1) {

current = current->next;

currentIndex++;

}

if (currentIndex == index - 1) {

Node\* newNode = new Node;

newNode->data = val;

newNode->next = current->next;

newNode->prev = current;

current->next = newNode;

if (newNode->next != NULL) {

newNode->next->prev = newNode;

}

else {

tail = newNode;

}

}

else {

cout << "Index out of bounds" << endl;

}

}

}

void DoublyLinkedList::Delete\_at\_front() {

if (Isempty()) {

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

return;

}

else {

Node\* temp = head;

head = head->next;

if (head != NULL) {

head->prev = NULL;

}

else {

tail = NULL;

}

delete temp;

}

}

void DoublyLinkedList::Delete\_at\_end() {

if (Isempty()) {

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

return;

}

else if (head->next == NULL) {

delete head;

head = NULL;

tail = NULL;

}

else {

Node\* current = head;

while (current->next->next != NULL) {

current = current->next;

}

Node\* temp = current->next;

current->next = NULL;

tail = current;

delete temp;

}

}

void DoublyLinkedList::Delete\_at\_index(int index) {

if (Isempty()) {

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

return;

}

if (index < 0) {

cout << "Index cannot be negative" << endl;

return;

}

if (index == 0) {

Node\* temp = head;

head = head->next;

if (head != nullptr) {

head->prev = nullptr;

}

else {

tail = nullptr;

}

delete temp;

}

else {

Node\* current = head;

int currentIndex = 0;

while (current != nullptr && currentIndex < index - 1) {

current = current->next;

currentIndex++;

}

if (current == nullptr || current->next == nullptr) {

cout << "Index out of bounds" << endl;

return;

}

Node\* temp = current->next;

current->next = temp->next;

if (temp->next != nullptr) {

temp->next->prev = current;

}

else {

tail = current;

}

delete temp;

}

}

void DoublyLinkedList::Print() {

Node\* current = head;

while (current != NULL) {

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

current = current->next;

}

cout << endl;

}

bool DoublyLinkedList::palindromeDoubly() {

if (Isempty() || head->next == nullptr) {

return true;

}

Node\* slow = head;

Node\* fast = head;

while (fast != nullptr && fast->next != nullptr) {

slow = slow->next;

fast = fast->next->next;

}

Node\* secondHalf = slow;

Node\* prev = nullptr;

Node\* current = secondHalf;

Node\* next = nullptr;

while (current != nullptr) {

next = current->next;

current->next = prev;

current->prev = next;

prev = current;

current = next;

}

secondHalf = prev;

Node\* firstHalf = head;

while (secondHalf != nullptr) {

if (firstHalf->data != secondHalf->data) {

return false;

}

firstHalf = firstHalf->next;

secondHalf = secondHalf->next;

}

return true;

}

DoublyLinkedList::~DoublyLinkedList() {

Node\* current = head;

while (current != NULL) {

Node\* next = current->next;

delete current;

current = next;

}

}

int main() {

DoublyLinkedList DLL;

int choice, val, index;

while (true) {

cout << "------------------------Menu--------------------------" << endl;

cout << "1. Insert at front" << endl;

cout << "2. Insert at end" << endl;

cout << "3. Insert at index" << endl;

cout << "4. Delete at front" << endl;

cout << "5. Delete at end" << endl;

cout << "6. Delete at index" << endl;

cout << "7. Check if list is palindrome" << endl;

cout << "8. Print list" << endl;

cout << "0. Exit" << endl;

cout << "Enter your choice: ";

cin >> choice;

switch (choice) {

case 1:

cout << "Enter value to insert at front: ";

cin >> val;

DLL.Insert\_at\_front(val);

break;

case 2:

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

cin >> val;

DLL.Insert\_at\_end(val);

break;

case 3:

cout << "Enter index: ";

cin >> index;

cout << "Enter value to insert: ";

cin >> val;

DLL.Insert\_at\_index(index, val);

break;

case 4:

DLL.Delete\_at\_front();

break;

case 5:

DLL.Delete\_at\_end();

break;

case 6:

cout << "Enter index: ";

cin >> index;

DLL.Delete\_at\_index(index);

break;

case 7:

if (DLL.palindromeDoubly()) {

cout << "The list is a palindrome." << endl;

}

else {

cout << "The list is not a palindrome." << endl;

}

break;

case 8:

DLL.Print();

break;

case 0:

return 0;

default:

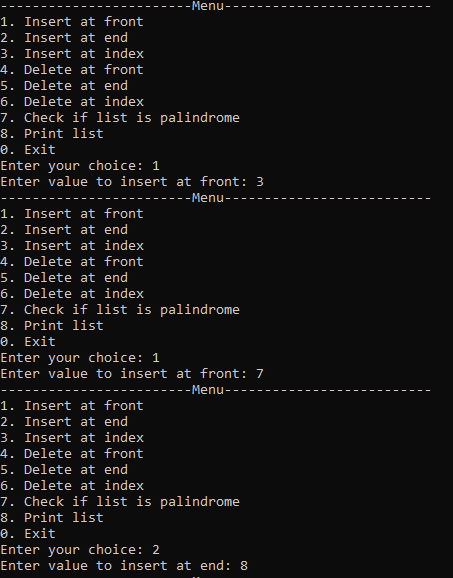
cout << "Invalid choice. Try again." << endl;

}

}

return 0;

}



A screenshot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

A screenshot of a computer program

Description automatically generated

**Task 7**