///////DYNAMIC ALLOCATION OF MATRICES

#include <iostream>

using namespace std;

int main() {

int rows, cols;

//get the number of rows and columns for the matrix from the user

cout << "Enter the number of rows for the matrix: ";

cin >> rows;

cout << "Enter the number of columns for the matrix: ";

cin >> cols;

//dynamically allocate memory for the matrix

int \*\*matrix = new int\*[rows];

for (int i = 0; i < rows; i++) {

matrix[i] = new int[cols];

}

//fill the matrix with data

for (int i = 0; i < rows; i++) {

for (int j = 0; j < cols; j++) {

cout << "Enter the value for element (" << i + 1 << "," << j + 1 << "): ";

cin >> matrix[i][j];

}

}

//print the matrix

cout << "The matrix is: " << endl;

for (int i = 0; i < rows; i++) {

for (int j = 0; j < cols; j++) {

cout << matrix[i][j] << " ";

}

cout << endl;

}

//free the dynamically allocated memory

for (int i = 0; i < rows; i++) {

delete[] matrix[i];

}

delete[] matrix;

system("pause");

return 0;

}

////////Singly Linked List

#include <iostream>

using namespace std;

class Node

{

public:

int data;

Node\* next;

Node(int data) {

this->data = data;

next = nullptr;

}

};

class SinglyLinkedList

{

private:

Node\* head;

public:

SinglyLinkedList() {

head = nullptr;

}

void insert(int data)

{

Node\* newNode = new Node(data);

if (head == nullptr)

{

head = newNode;

return;

}

Node\* current = head;

while (current->next != nullptr)

{

current = current->next;

}

current->next = newNode;

}

void display() {

Node\* current = head;

while (current != nullptr)

{

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

current = current->next;

}

cout << endl;

}

~SinglyLinkedList()

{

Node\* current = head;

while (current != nullptr)

{

Node\* next = current->next;

delete current;

current = next;

}

head = nullptr;

}

};

int main()

{

SinglyLinkedList list;

list.insert(1);

list.insert(2);

list.insert(3);

list.display();

system("pause");

return 0;

}

///////SINGLY LINKED LIST - 2

#include <iostream>

using namespace std;

class Node

{

public:

int data;

Node\* next;

};

class SinglyLinkedList

{

public:

SinglyLinkedList() {

head = NULL;

}

void insert(int value)

{

Node\* newNode = new Node();

newNode->data = value;

newNode->next = head;

head = newNode;

}

void remove(int value)

{

Node\* current = head;

Node\* previous = NULL;

while (current != NULL)

{

if (current->data == value)

{

if (previous == NULL)

{

head = current->next;

}

else

{

previous->next = current->next;

}

delete current;

return;

}

previous = current;

current = current->next;

}

}

void print()

{

Node\* current = head;

if (head == NULL)

{

cout << "Empty List" << endl;

return;

}

while (current != NULL)

{

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

current = current->next;

}

cout << endl;

}

private:

Node\* head;

};

int main() {

SinglyLinkedList myList;

myList.insert(5);

myList.insert(10);

myList.insert(15);

myList.insert(20);

myList.print(); // output: 20 15 10 5

myList.remove(15);

myList.print(); // output: 20 10 5

myList.remove(10);

myList.remove(5);

myList.print(); // output: 20

myList.remove(20);

myList.print(); // output: Empty List

system("pause");

return 0;

}

///////DOUBLY LINKED LIST

#include <iostream>

using namespace std;

class Node {

public:

int data;

Node\* prev;

Node\* next;

Node(int d) {

data = d;

prev = nullptr;

next = nullptr;

} };

class DoublyLinkedList {

private:

Node\* head;

Node\* tail;

public:

DoublyLinkedList() {

head = nullptr;

tail = nullptr;

}

bool isEmpty()

{

return head == nullptr;

}

void addNode(int d) {

Node\* newNode = new Node(d);

if (isEmpty()) {

head = newNode;

tail = newNode;

}

else {

tail->next = newNode;

newNode->prev = tail;

tail = newNode;

}

}

void deleteNode(int d) {

Node\* curr = head;

while (curr != nullptr)

{

if (curr->data == d)

{

if (curr == head)

{

head = curr->next;

if (head != nullptr)

head->prev = nullptr;

}

else if (curr == tail)

{

tail = curr->prev;

tail->next = nullptr;

}

else {

//for example our list is : d e f

//to delete e from d to f

//current=e

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

// d->next will point f

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

//f->prev will point d

}

delete curr;

return;

}

curr = curr->next;

}

}

void printList() {

Node\* curr = head;

while (curr != nullptr) {

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

curr = curr->next;

}

cout << endl;

}

};

int main() {

DoublyLinkedList list;

list.addNode(1);

list.addNode(2);

list.addNode(3);

list.addNode(4);

list.printList();

list.deleteNode(2);

/\*list.addNode(5);

list.deleteNode(10);\*/

list.printList();

system("pause");

return 0;

}

////DOUBLY LINKED LIST WITH EXCEPTION

#include <iostream>

#include <stdexcept>

using namespace std;

class Node {

public:

int data;

Node\* prev;

Node\* next;

Node(int d) {

data = d;

prev = nullptr;

next = nullptr;

}

};

class DoublyLinkedList {

private:

Node\* head;

Node\* tail;

public:

DoublyLinkedList() {

head = nullptr;

tail = nullptr;

}

bool isEmpty()

{

return head == nullptr;

}

void addNode(int d) {

Node\* newNode = new Node(d);

if (isEmpty()) {

head = newNode;

tail = newNode;

}

else {

tail->next = newNode;

newNode->prev = tail;

tail = newNode;

}

}

void deleteNode(int d)

{

if (isEmpty()) {

throw out\_of\_range("List is empty");

}

Node\* curr = head;

while (curr != nullptr)

{

if (curr->data == d)

{

if (curr == head) {

head = curr->next;

if (head != nullptr)

head->prev = nullptr;

}

else if (curr == tail) {

tail = curr->prev;

tail->next = nullptr; }

else

{

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

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

}

delete curr;

return; }

curr = curr->next;

}

throw out\_of\_range("Element not found in list");

}

void printList() {

Node\* curr = head;

while (curr != nullptr) {

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

curr = curr->next;

}

cout << endl;

}

};

int main()

{

DoublyLinkedList list;

try {

list.deleteNode(1); // throws out\_of\_range exception

}

catch (const out\_of\_range& e) {

cout << "Exception: " << e.what() << endl;

}

list.addNode(1);

list.addNode(2);

list.addNode(3);

list.addNode(4);

list.printList();

try {

list.deleteNode(2);

list.deleteNode(5);

// throws out\_of\_range exception

}

catch (const out\_of\_range& e) {

cout << "Exception: " << e.what() << endl;

}

list.printList();

system("pause");

return 0;

}