Kashan Baig

CT-24178

LAB 02

Q1.

#include <iostream>

using namespace std;

class Node{

    public:

    int data;

    Node \*next;

    Node(int *val*){ data = *val*; next = nullptr; }

};

class List{

    public:

    Node\* head;

    Node\* tail;

    List(){

        head = tail = nullptr;

    }

    void push(int *val*){

        Node\* newnode = new Node(*val*);

        if(head==nullptr){

            head = tail = newnode;

        }

        else{

            newnode->next = head;

            head = newnode;

        }

    }

     void display(Node\* *node*){

        Node\* temp = *node*;

        while (temp!=nullptr){

            cout << temp->data ;

            temp = temp->next;

            if(temp) cout << " -> ";

        }

        cout << " -> NULL" << endl;

    }

    Node\* mergelsit(Node\* *list1*,Node\* *list2*){

        Node\* dummy = new Node(0);

        tail = dummy;

        while(*list1*!=nullptr && *list2*!=nullptr){

            if(*list1*->data <= *list2*->data){

                tail->next = *list1*;

*list1* = *list1*->next;

            }

            else{

            tail->next = *list2*;

*list2* = *list2*->next;

        }

        tail = tail->next;

        }

        if (*list1*) tail->next = *list1*;

        if (*list2*) tail->next = *list2*;

        return dummy->next;

    }

};

int main(){

     List l1, l2, mergedList;

*// Create first sorted list: 1 -> 2 -> 4*

    l1.push(4);

    l1.push(2);

    l1.push(1);

*// Create second sorted list: 1 -> 3 -> 4*

    l2.push(4);

    l2.push(3);

    l2.push(1);

    cout << "List 1: ";

    l1.display(l1.head);

    cout << "List 2: ";

    l2.display(l2.head);

*// Merge lists*

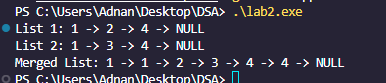
    Node\* mergedHead = mergedList.mergelsit(l1.head, l2.head);

    cout << "Merged List: ";

    mergedList.display(mergedHead);

    return 0;

}



Q2.

 void deleteduplicate(Node\* *head*){

        Node\* temp = *head*;

        Node\*current = *head*;

        while(current!=nullptr){

            if(current->data == temp->data){

                Node\* dup = current->next;

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

                delete dup;

*// delete data*

            }

            else{

                temp = temp->next;

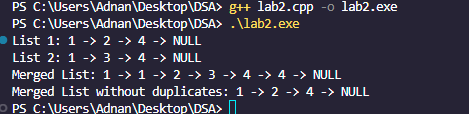
                continue;

            }

            current = current->next;

        }

    }



Q3.

    Node\* sortList(Node\* *head*) {

    if (*head* == nullptr || *head*->next == nullptr)

        return *head*;

    Node\* slow = *head*;

    Node\* fast = *head*->next;

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

        slow = slow->next;

        fast = fast->next->next;

    }

    Node\* rightHalf = slow->next;

    slow->next = nullptr;

    Node\* leftHalf = *head*;

    Node\* leftSorted = sortList(leftHalf);

    Node\* rightSorted = sortList(rightHalf);

    return mergelsit(leftSorted, rightSorted);

}



Q4.

    Node\* reverseList(Node\* *head*) {

        Node\* prev = nullptr;

        while (*head* != nullptr) {

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

*head*->next = prev;

            prev = *head*;

*head* = next;

        }

        return prev;

    }

    bool isPalindrome(Node\* *head*) {

        if (*head* == nullptr) return true;

        Node\* slow = *head*;

        Node\* fast = *head*;

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

            slow = slow->next;

            fast = fast->next->next;

        }

        Node\* secondHalfStart = reverseList(slow->next);

        Node\* firstPointer = *head*;

        Node\* secondPointer = secondHalfStart;

        bool result = true;

        while (secondPointer != nullptr) {

            if (firstPointer->data != secondPointer->data) {

                result = false;

                break;

            }

            firstPointer = firstPointer->next;

            secondPointer = secondPointer->next;

        }

        slow->next = reverseList(secondHalfStart);

        return result;

    }



Q5.

#include <iostream>

using namespace std;

class StackNode {

public:

    int data;

    StackNode\* next;

    StackNode(int *val*) : data(*val*), next(nullptr) {}

};

class Stack {

    StackNode\* head;

public:

    Stack() : head(nullptr) {}

    void push(int *val*) {

        StackNode\* node = new StackNode(*val*);

        node->next = head;

        head = node;

    }

    int pop() {

        if (!head) throw runtime\_error("Stack is empty");

        int val = head->data;

        StackNode\* temp = head;

        head = head->next;

        delete temp;

        return val;

    }

    int peek() {

        if (!head) throw runtime\_error("Stack is empty");

        return head->data;

    }

    bool isEmpty() { return head == nullptr; }

    ~Stack() {

        while (head) {

            StackNode\* temp = head;

            head = head->next;

            delete temp;

        }

    }

};

void stackDemo() {

    Stack s;

    s.push(10);

    s.push(20);

    s.push(30);

    cout << "Stack pop: " << s.pop() << endl;

    cout << "Stack peek: " << s.peek() << endl;

    cout << "Stack pop: " << s.pop() << endl;

    cout << "Stack pop: " << s.pop() << endl;

    cout << "Stack is empty: " << (s.isEmpty() ? "true" : "false") << endl;

}

class QueueNode {

public:

    int data;

    QueueNode\* next;

    QueueNode(int *val*) : data(*val*), next(nullptr) {}

};

class CircularQueue {

    QueueNode\* front;

    QueueNode\* rear;

public:

    CircularQueue() : front(nullptr), rear(nullptr) {}

    void enqueue(int *val*) {

        QueueNode\* node = new QueueNode(*val*);

        if (!front) {

            front = rear = node;

            rear->next = front;

        } else {

            rear->next = node;

            rear = node;

            rear->next = front;

        }

    }

    int dequeue() {

        if (!front) throw runtime\_error("Queue is empty");

        int val = front->data;

        if (front == rear) {

            delete front;

            front = rear = nullptr;

        } else {

            QueueNode\* temp = front;

            front = front->next;

            rear->next = front;

            delete temp;

        }

        return val;

    }

    int peek() {

        if (!front) throw runtime\_error("Queue is empty");

        return front->data;

    }

    bool isEmpty() { return front == nullptr; }

    ~CircularQueue() {

        while (front) {

            dequeue();

        }

    }

};

void queueDemo() {

    CircularQueue q;

    q.enqueue(1);

    q.enqueue(2);

    q.enqueue(3);

    cout << "Queue dequeue: " << q.dequeue() << endl;

    cout << "Queue peek: " << q.peek() << endl;

    cout << "Queue dequeue: " << q.dequeue() << endl;

    cout << "Queue dequeue: " << q.dequeue() << endl;

    cout << "Queue is empty: " << (q.isEmpty() ? "true" : "false") << endl;

}

int main() {

    cout << "--- Stack Demo ---" << endl;

    stackDemo();

    cout << endl;

    cout << "--- Circular Queue Demo ---" << endl;

    queueDemo();

    return 0;

}

