TASK – 1

VECTOR

BEGIN

#include <iostream>

#include <vector>

using namespace std;

int main() {

vector<char> v{'a', 'e', 'i', 'o', 'u'};

vector<char>::iterator itr;

// Use a for loop to iterate through the vector

for (itr = v.begin(); itr != v.end(); ++itr) {

cout << \*itr << " "; // Print each element followed by a space

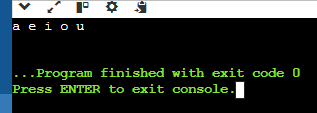
}

cout << endl; // Print a newline at the end

return 0;

}

OUTPUT:



RBEGIN ()

#include <iostream>

#include <vector>

using namespace std;

int main() {

vector<char> v{'a', 'e', 'i', 'o', 'u'};

vector<char>::reverse\_iterator rit;

// Use a for loop to iterate through the vector in reverse

for (rit = v.rbegin(); rit != v.rend(); ++rit) {

cout << \*rit << " "; // Print each element followed by a space

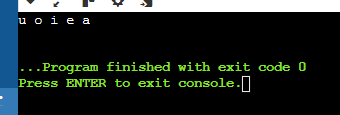
}

cout << endl; // Print a newline at the end

return 0;

}

OUTPUT:



CUSTOM VECTOR

#include <iostream>

#include <vector>

#include <algorithm> // Include the algorithm header for std::remove

using namespace std;

template <typename T>

class CustomVector {

private:

vector<T> data;

public:

void add(const T& element) {

data.push\_back(element);

}

void remove(const T& element) {

auto it = std::remove(data.begin(), data.end(), element);

data.erase(it, data.end());

}

T get(size\_t index) const {

if (index >= data.size()) {

throw out\_of\_range("Index out of range");

}

return data[index];

}

size\_t size() const {

return data.size();

}

void print() const {

for (const auto& element : data) {

cout << element << " ";

}

cout << endl;

}

};

int main() {

CustomVector<int> myVector;

myVector.add(1);

myVector.add(2);

myVector.add(3);

cout << "Vector elements: ";

myVector.print();

cout << "Element at index 1: " << myVector.get(1) << endl;

myVector.remove(2);

cout << "Vector elements after removal: ";

myVector.print();

return 0;

}

OUTPUT:

Problem 1: Reverse a Queue

Description: Implement a function to reverse the elements of a queue using a stack.

#include <iostream>

#include <queue>

#include <stack>

using namespace std; // Function to reverse a queue using a stack

void reverseQueue(queue<int>& q) {

stack<int> s;

while (!q.empty()) { // Transfer elements from queue to stack

s.push(q.front());

q.pop();

}

// Transfer elements from stack to queue (reversed order)

while (!s.empty()) {

q.push(s.top());

s.pop();

}

}

// Function to print elements of a queue

void printQueue(queue<int> q) {

while (!q.empty()) {

cout << q.front() << " ";

q.pop();

}

cout << endl;

}

int main() {

queue<int> myqueue;

// Add elements to the queue

myqueue.push(1);

myqueue.push(2);

myqueue.push(3);

myqueue.push(4);

myqueue.push(5);

cout << "Original Queue: ";

printQueue(myqueue); // Output: Original Queue: 1 2 3 4 5

// Reverse the queue using the stack

reverseQueue(myqueue);

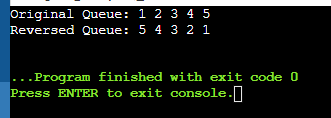
cout << "Reversed Queue: ";

printQueue(myqueue); // Output: Reversed Queue: 5 4 3 2 1

return 0;

}

OUTPUT:



2. Maximum Element in Stack

Description: Design a stack that supports push, pop, and retrieving the maximum element in constant time.

A: #include <iostream>

#include <stack>

using namespace std;

class MaxStack {

private:

stack<pair<int, int>> s;

public:

void push(int x) {

int maxVal = s.empty() ? x : max(x, s.top().second);

s.push({x, maxVal});

} void pop() {

if (s.empty()) {

cout << "Stack is empty.\n";

return;

}

s.pop();

} int top() {

if (s.empty()) {

cout << "Stack is empty.\n";

return -1; // Returning -1 to indicate an error

}

return s.top().first;

}

int getMax() {

if (s.empty()) {

cout << "Stack is empty.\n";

return -1; // Returning -1 to indicate an error

}

return s.top().second; }

bool empty() {

return s.empty(); }

int size() {

return s.size(); }

};

int main() {

MaxStack s;

s.push(5);

s.push(1);

s.push(3);

s.push(9);

s.push(7);

cout << "Current max: " << s.getMax() << '\n';

s.pop();

cout << "Current max after one pop: " << s.getMax() << '\n';

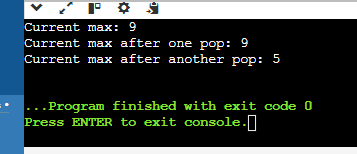
s.pop();

cout << "Current max after another pop: " << s.getMax() << '\n';

return 0;

}

OUTPUT:



3. Circular Queue Implementation

Description:

Implement a circular queue using an array. The queue should support enqueue, dequeue, and front operations.

#include <iostream>

using namespace std;

class CircularQueue {

private:

int \*arr;

int front;

int rear;

int size;

int capacity;

public:

CircularQueue(int capacity) {

this->capacity = capacity;

arr = new int[capacity];

front = -1;

rear = -1;

size = 0;

}

~CircularQueue() {

delete[] arr; }

bool enqueue(int value) {

if (isFull()) {

cout << "Queue is full\n";

return false; }

if (isEmpty()) {

front = rear = 0;

} else {

rear = (rear + 1) % capacity; }

arr[rear] = value;

size++;

return true; }

bool dequeue() {

if (isEmpty()) {

cout << "Queue is empty\n";

return false; }

if (front == rear) {

front = rear = -1;

} else {

front = (front + 1) % capacity; }

size--;

return true; }

int frontElement() {

if (isEmpty()) { cout << "Queue is empty\n";

return -1; // Indicate error

}

return arr[front]; }

bool isEmpty() {

return size == 0; }

bool isFull() {

return size == capacity; }

int getSize() {

return size; }

};

int main() {

CircularQueue q(5);

q.enqueue(1);

q.enqueue(2);

q.enqueue(3);

q.enqueue(4);

q.enqueue(5);

cout << "Front element: " << q.frontElement() << endl;

q.dequeue();

cout << "Front element after dequeue: " << q.frontElement() << endl;

q.enqueue(6);

cout << "Front element after enqueue: " << q.frontElement() << endl;

while (!q.isEmpty()) {

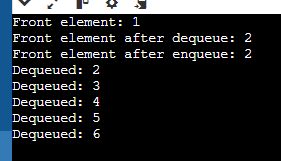
cout << "Dequeued: " << q.frontElement() << endl;

q.dequeue(); }

return 0;

}

OUTPUT:



4. Sort a Stack

Description:

Write a function to sort a stack such that the smallest items are on the top.

#include <iostream>

#include <stack>

using namespace std;

void sortStack(stack<int>& s) {

stack<int> tempStack;

while (!s.empty()) {

int current = s.top();

s.pop();

// Move elements from tempStack back to s until we find the correct position for current

while (!tempStack.empty() && tempStack.top() > current) {

s.push(tempStack.top());

tempStack.pop();

}

// Place current element in the correct sorted position in tempStack

tempStack.push(current); }

// Move sorted elements from tempStack back to s

while (!tempStack.empty()) {

s.push(tempStack.top());

tempStack.pop();

}

}

int main() {

stack<int> s;

s.push(5);

s.push(2);

s.push(8);

s.push(1);

s.push(3);

cout << "Original Stack: ";

stack<int> original = s;

while (!original.empty()) {

cout << original.top() << " ";

original.pop();

}

cout << endl;

sortStack(s);

cout << "Sorted Stack: ";

while (!s.empty()) {

cout << s.top() << " ";

s.pop();

}

cout << endl;

return 0;

}

OUTPUT:

