Create a class with two different vectors ?

#include <iostream>

#include <vector>

class VectorPair {

private:

std::vector<int> vector1;

std::vector<int> vector2;

public:

// Constructor to initialize the vectors with given sizes and values

VectorPair(const std::vector<int>& v1, const std::vector<int>& v2) : vector1(v1), vector2(v2) {}

// Method to add elements to the vectors

void addElementsToVector1(int value) {

vector1.push\_back(value);

}

void addElementsToVector2(int value) {

vector2.push\_back(value);

}

// Method to display the vectors

void displayVectors() const {

std::cout << "Vector 1: ";

for (const auto& elem : vector1) {

std::cout << elem << " ";

}

std::cout << std::endl;

std::cout << "Vector 2: ";

for (const auto& elem : vector2) {

std::cout << elem << " ";

}

std::cout << std::endl;

}

// Method to get the size of the vectors

size\_t getVector1Size() const {

return vector1.size();

}

size\_t getVector2Size() const {

return vector2.size();

}

// Method to clear the vectors

void clearVectors() {

vector1.clear();

vector2.clear();

}

};

int main() {

std::vector<int> v1 = {1, 2, 3};

std::vector<int> v2 = {4, 5, 6};

VectorPair vp(v1, v2);

vp.displayVectors();

vp.addElementsToVector1(7);

vp.addElementsToVector2(8);

vp.displayVectors();

std::cout << "Size of Vector 1: " << vp.getVector1Size() << std::endl;

std::cout << "Size of Vector 2: " << vp.getVector2Size() << std::endl;

vp.clearVectors();

vp.displayVectors();

return 0;

}

QUEUE : PUSH:

#include<iostream>

#include<queue>

using namespace std;

int main(){

queue<int> myqueue;

myqueue.push(0);

myqueue.push(1);

myqueue.push(2);

while(!myqueue.empty()){

cout <<'h'<< myqueue.front();

myqueue.pop();

}

}

QUEUE:POP:

#include<iostream>

#include<queue>

using namespace std;

int main(){

queue<int> myqueue;

myqueue.push(0);

myqueue.push(1);

myqueue.push(2);

myqueue.pop();

myqueue.pop();

while(!myqueue.empty()){

cout <<' '<< myqueue.front();

myqueue.pop();

}

}

STACK:

#include<bits/stdc++.h>

using namespace std;

void showstack(stack <int> s)

{

while(!s.empty())

{

cout<<'\t'<< s.top();

s.pop();

}

cout<<'\n';

}

int main(){

stack<int> s;

s.push(10);

s.push(30);

s.push(20);

s.push(5);

s.push(1);

cout<<"The stack is:";

showstack(s);

cout<<"\ns.size() : "<<s.size();

cout<<"\ns.top() : "<<s.top();

cout<< "\ns.pop() :";

s.pop();

showstack(s);

return 0;

}

4. Problem 4: 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;

// Move all elements from the queue to the stack

while (!q.empty()) {

s.push(q.front()); // Push the front element of the queue to the stack

q.pop(); // Remove the front element from the queue

}

// Move all elements back from the stack to the queue

while (!s.empty()) {

q.push(s.top()); // Push the top element of the stack to the queue

s.pop(); // Remove the top element from the stack

}

}

int main() {

queue<int> q;

// Adding elements to the queue

q.push(1);

q.push(2);

q.push(3);

q.push(4);

q.push(5);

// Display the original queue

cout << "Original Queue: ";

queue<int> temp = q; // Copying queue to display the original queue

while (!temp.empty()) {

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

temp.pop();

}

cout << endl;

// Reverse the queue

reverseQueue(q);

// Display the reversed queue

cout << "Reversed Queue: ";

while (!q.empty()) {

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

q.pop();

}

cout << endl;

return 0;

}

Implement Queue Using Stacks

Maximum Element in Stack

Description:

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

#include <iostream>

#include <stack>

using namespace std;

class MaxStack {

private:

stack<int> mainStack; // Main stack to store elements

stack<int> maxStack; // Auxiliary stack to store maximum elements

public:

// Push an element onto the stack

void push(int x) {

mainStack.push(x);

if (maxStack.empty() || x >= maxStack.top()) {

maxStack.push(x);

}

}

// Pop an element from the stack

void pop() {

if (!mainStack.empty()) {

if (mainStack.top() == maxStack.top()) {

maxStack.pop();

}

mainStack.pop();

}

}

// Retrieve the top element of the stack

int top() {

if (!mainStack.empty()) {

return mainStack.top();

}

return -1; // Return -1 if the stack is empty

}

// Retrieve the maximum element in the stack

int getMax() {

if (!maxStack.empty()) {

return maxStack.top();

}

return -1; // Return -1 if the stack is empty

}

};

int main() {

MaxStack s;

s.push(3);

s.push(1);

s.push(5);

s.push(2);

cout << "Max element: " << s.getMax() << endl; // Output: 5

s.pop();

cout << "Max element after one pop: " << s.getMax() << endl; // Output: 5

s.pop();

cout << "Max element after two pops: " << s.getMax() << endl; // Output: 3

return 0;

}

2. 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; // Array to store queue elements

int size; // Maximum size of the queue

int front; // Front points to the front element in the queue

int rear; // Rear points to the last element in the queue

int count; // Current number of elements in the queue

public:

// Constructor to initialize the queue

CircularQueue(int size) {

this->size = size;

arr = new int[size];

front = -1;

rear = -1;

count = 0;

}

// Destructor to clean up allocated memory

~CircularQueue() {

delete[] arr;

}

// Enqueue operation to add an element to the queue

bool enqueue(int value) {

if (isFull()) {

cout << "Queue is full" << endl;

return false;

}

if (isEmpty()) {

front = 0;

}

rear = (rear + 1) % size;

arr[rear] = value;

count++;

return true;

}

// Dequeue operation to remove an element from the queue

bool dequeue() {

if (isEmpty()) {

cout << "Queue is empty" << endl;

return false;

}

if (front == rear) {

front = -1;

rear = -1;

} else {

front = (front + 1) % size;

}

count--;

return true;

}

// Front operation to get the front element of the queue

int getFront() {

if (isEmpty()) {

cout << "Queue is empty" << endl;

return -1;

}

return arr[front];

}

// Check if the queue is empty

bool isEmpty() {

return (count == 0);

}

// Check if the queue is full

bool isFull() {

return (count == 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.getFront() << endl; // Output: 1

q.dequeue();

cout << "Front element after deq

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;

// Function to insert an element into a sorted stack

void sortedInsert(stack<int>& s, int element) {

// Base case: If stack is empty or element is greater than top element

if (s.empty() || element > s.top()) {

s.push(element);

return;

}

// Remove the top element

int temp = s.top();

s.pop();

// Recursively call sortedInsert to find the correct position for the element

sortedInsert(s, element);

// Put the removed element back

s.push(temp);

}

// Function to sort the stack

void sortStack(stack<int>& s) {

// Base case: If stack is not empty

if (!s.empty()) {

// Remove the top element

int temp = s.top();

s.pop();

// Recursively sort the remaining stack

sortStack(s);

// Insert the removed element into the sorted stack

sortedInsert(s, temp);

}

}

// Function to print the stack

void printStack(stack<int> s) {

while (!s.empty()) {

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

s.pop();

}

cout << endl;

}

int main() {

stack<int> s;

s.push(34);

s.push(3);

s.push(31);

s.push(98);

s.push(92);

s.push(23);

cout << "Original Stack: ";

printStack(s);

// Sorting the stack

sortStack(s);

cout << "Sorted Stack: ";

printStack(s);

return 0;

}

RANGE BASED FOR LOOP:

#include <iostream>

#include <list>

int main() {

// Create a list

std::list<int> myList;

// Insert elements at the end

myList.push\_back(10);

myList.push\_back(20);

myList.push\_back(30);

// Insert elements at the front

myList.push\_front(5);

myList.push\_front(1);

// Display elements

std::cout << "List after push\_back and push\_front: ";

for (int val : myList) {

std::cout << val << " ";

}

std::cout << std::endl;

// Insert element at a specific position

auto it = myList.begin();

std::advance(it, 2);

myList.insert(it, 15);

std::cout << "List after insert: ";

for (int val : myList) {

std::cout << val << " ";

}

std::cout << std::endl;

// Erase element at a specific position

it = myList.begin();

std::advance(it, 3);

myList.erase(it);

std::cout << "List after erase: ";

for (int val : myList) {

std::cout << val << " ";

}

std::cout << std::endl;

// Remove elements by value

myList.remove(10);

std::cout << "List after remove: ";

for (int val : myList) {

std::cout << val << " ";

}

std::cout << std::endl;

// Remove elements based on a condition

myList.remove\_if([](int n) { return n < 10; });

std::cout << "List after remove\_if: ";

for (int val : myList) {

std::cout << val << " ";

}

std::cout << std::endl;

// Sorting the list

myList.sort();

std::cout << "List after sort: ";

for (int val : myList) {

std::cout << val << " ";

}

std::cout << std::endl;

// Reversing the list

myList.reverse();

std::cout << "List after reverse: ";

for (int val : myList) {

std::cout << val << " ";

}

std::cout << std::endl;

// Merging two lists

std::list<int> otherList = {40, 50, 60};

myList.merge(otherList);

std::cout << "List after merge: ";

for (int val : myList) {

std::cout << val << " ";

}

std::cout << std::endl;

// Clearing the list

myList.clear();

std::cout << "List after clear: ";

for (int val : myList) {

std::cout << val << " ";

}

std::cout << std::endl;

// Checking if the list is empty

if (myList.empty()) {

std::cout << "List is empty." << std::endl;

}

// Adding elements again

myList.push\_back(100);

myList.push\_back(200);

// Accessing front and back elements

std::cout << "Front element: " << myList.front() << std::endl;

std::cout << "Back element: " << myList.back() << std::endl;

return 0;

}