Министерство науки и высшего образования Российской Федерации

Пензенский государственный университет

Кафедра «Вычислительная техника»

**ОТЧЕТ**

по лабораторной работе №2

по курсу «Логика и основы алгоритмизации

в инженерных задачах»

на тему «Динамические списки»

Выполнил: ст. гр. 24ВВВ1

Будников А.С.

Принял:

к.т.н, доцент Юрова О. В.

к.т.н., доцент Деев М.В.

Пенза

2025

**Цель работы:**

Познакомиться с динамическими списками и реализовать предложенные типы данных.

**Лабораторное задание:**

1. Реализовать приоритетную очередь, путём добавления элемента в список в

соответствии с приоритетом объекта (т.е. объект с большим приоритетом

становится перед объектом с меньшим приоритетом).

2. \*На основе приведенного кода реализуйте структуру данных Очередь.

3. \*На основе приведенного кода реализуйте структуру данных Стек.

**Ход работы:**

1. **Приоритетная очередь** представляет собой абстрактную структуру данных, где порядок обработки элементов определяется не временем добавления, а их «важностью», то есть приоритетом. У приоритетной очереди, как и у обычной, имеется начало и конец, а элементы извлекаются от начала. Но у приоритетной очереди элементы упорядочиваются по ключу, так что элемент с наименьшим (в некоторых реализациях — наибольшим) значением ключа всегда находится в начале. Новые элементы вставляются в позициях, сохраняющих порядок сортировки.

Для реализации было объявлено два класса: *Node* – элемент очереди, хранящий данные и свой приоритет, и *PriorityQueue* – непосредственно реализация приоритетной очереди, хранящая максимальную вместимость (*capacity*), текущий размер очереди (*size*), имитацию двоичной кучи (*heap*) для упрощенного доступа к элементам и методы для работы с ней. Было реализовано 11 методов:

1) void up(size\_t index) - поднимает новый элемент «вверх» по очереди, если его приоритет больше, чем у вышестоящих;

2) void down(size\_t index) – производит сортировку элементов очереди после удаления самого приоритетного элемента;

3) bool is\_empty() const – возвращает *True*, если очередь не имеет ни одного элемента, и *False* в противном случае;

4) bool is\_full() const – метод обратный *is\_empty*;

5) size\_t get\_size() const – возвращает текущий размер очереди;

6) size\_t get\_capacity() const – возвращает вместимость очереди;

7) const E& top() const – возвращает данные, хранящиеся в наиболее приоритетном элементе очереди, не извлекая сам элемент;

8) int top\_priority() const – возвращает приоритет «верхнего» элемента очереди;

9) void push(E&& value, int priority) – осуществляет вставку элемента в очередь с последующей сортировкой по приоритету;

10) E pop() – осуществляет извлечение элемента, находящегося на верхней границе очереди;

11) void resize(const size\_t newCapacity) – изменяет вместимость структуры данных в зависимости от текущего размера.

Код реализации приоритетной очереди представлен в листинге.

2. **Очередь** представляет собой структуру данных, добавление и удаление элементов из которой осуществляется по принципу «первым вошел — первым вышел» (англ. *first-in, first-out — FIFO*). У очереди имеется голова и хвост. Когда элемент ставится в очередь, он занимает место в её хвосте. Из очереди всегда выводится элемент, который находится в ее голове.

Для реализации был объявлен класс *Queue*, хранящий данные о вместимости очереди (*capacity*), текущем размере очереди (*size*), голове и хвосте очереди (в данном случае, *front* и *rear* соответственно) и массив элементов (*heap*). Было реализовано 9 методов, исключая перегрузку *push*:

1) bool is\_empty() const – возвращает *True*, если очередь не имеет ни одного элемента, и *False* в противном случае;

2) bool is\_full() const – метод обратный *is\_empty*;

3) size\_t get\_size() const – возвращает текущий размер очереди;

4) size\_t get\_capacity() const – возвращает вместимость очереди;

5) const E& peek\_back() const – возвращает последний добавленный элемент очереди;

6) const E& peek\_front() const – возвращает первый добавленный элемент очереди;

7) void push(…) - осуществляет вставку элемента в очередь;

8) E pop() – осуществляет извлечение элемента, находящегося на верхней границе очереди;

9) void resize(const size\_t newCapacity) – изменяет вместимость структуры данных в зависимости от текущего размера.

Код реализации очереди представлен в листинге.

3. **Стек** представляет собой упорядоченный набор элементов, в котором добавление новых элементов и удаление существующих производится с одного конца, называемого вершиной стека. Притом первым из стека удаляется элемент, который был помещен туда последним, то есть в стеке реализуется стратегия «последним вошел — первым вышел» (last-in, first-out — LIFO).

Для реализации был объявлен класс *Stack*, хранящий данные о вместимости стека (*capacity*), текущем размере (*size*), верхнем элементе (*top*) и массив элементов (*heap*).Было реализовано 8 методов, исключая перегрузку *push*:

1) bool is\_empty() const – возвращает *True*, если стек не имеет ни одного элемента, и *False* в противном случае;

2) bool is\_full() const – метод обратный *is\_empty*;

3) size\_t get\_size() const – возвращает текущий размер стека;

4) size\_t get\_capacity() const – возвращает вместимость стека;

5) E peek() const – возвращает «верхушку», т.е. последний добавленный элемент стека;

6) void push(…) - осуществляет вставку элемента в стек;

7) E pop() – осуществляет извлечение верхнего элемента стека;

8) void resize(const size\_t newCapacity) – изменяет вместимость структуры данных в зависимости от текущего размера.

Код реализации представлен в листинге.

**Вывод:** Реализованы три ключевые структуры данных с разными принципами доступа: стек (LIFO), очередь (FIFO) и приоритетная очередь. Цель работы достигнута, принципы работы структур изучены.

**Листинг программы:**

**Файл *priority\_q.h***

#ifndef PRIORITY\_Q\_H  
#define PRIORITY\_Q\_H  
#include <stdexcept>  
  
template<typename E>  
class Node {  
 public:  
 E data;  
 int priority{};  
  
 // ==Constructor==  
 Node() = default;  
 Node(const E& d, const int p) : data(d), priority(p) {}  
 // ==Destructor==  
 ~Node() = default;  
};  
  
template<typename E>  
class PriorityQueue {  
 private:  
 size\_t capacity;  
 size\_t size;  
 Node<E>\* heap;  
  
 // == Utils methods ==  
 /\*  
 \* We use binary heap there  
 \* to optimize push algorithm  
 \*/  
 void up(size\_t index) {  
 while (index > 0) {  
 size\_t parent = (index - 1) >> 2;  
 if (heap[index].priority > heap[parent].priority) {  
 std::swap(heap[index], heap[parent]);  
 index = parent;  
 } else {  
 break;  
 }  
 }  
 }  
  
 void down(size\_t index) {  
 while (true) {  
 size\_t left = 2 \* index + 1;  
 size\_t right = 2 \* index + 2;  
 size\_t largest = index;  
  
 if (left < size && heap[left].priority > heap[largest].priority) {  
 largest = left;  
 }  
 if (right < size && heap[right].priority > heap[largest].priority) {  
 largest = right;  
 }  
 if (largest != index) {  
 std::swap(heap[index], heap[largest]);  
 index = largest;  
 } else {  
 break;  
 }  
 }  
 }  
  
 void resize(const size\_t newCap) {  
 auto\* newHeap = new Node<E>[newCap];  
 for (size\_t i = 0; i < size; i++) {  
 newHeap[i] = std::move(heap[i]);  
 }  
 delete[] heap;  
 heap = newHeap;  
 capacity = newCap;  
  
 }  
  
 public:  
 // ==Constructor==  
 explicit PriorityQueue(const size\_t cap)  
 : capacity(cap),  
 size(0),  
 heap(new Node<E>[cap]) {}  
 // ==Destructor==  
 ~PriorityQueue() {  
 delete[] heap;  
 }  
  
 // ==Prohibit assignment==  
 PriorityQueue& operator=(const PriorityQueue&) = delete;  
 PriorityQueue(const PriorityQueue&) = delete;  
  
 // ==Prohibit movement==  
 PriorityQueue(PriorityQueue&&) = delete;  
 PriorityQueue& operator=(PriorityQueue&&) = delete;  
  
 // ==Basic operations==  
 [[nodiscard]] bool is\_empty() const { return size == 0; }  
  
 [[nodiscard]] bool is\_full() const { return size == capacity; }  
  
 [[nodiscard]] size\_t get\_size() const { return size; }  
  
 [[nodiscard]] size\_t get\_capacity() const { return capacity; }  
  
 [[maybe\_unused]] const E& top() const {  
 if (is\_empty()) throw std::out\_of\_range("Priority queue is empty");  
 return heap[0].data;  
 }  
  
 [[nodiscard]] int top\_priority() const {  
 if (is\_empty()) throw std::out\_of\_range("Priority queue is empty");  
 return heap[0].priority;  
 }  
  
 void push(E&& value, int priority) {  
 if (is\_full()) resize(2 \* capacity);  
 heap[size] = Node<E>(std::move(value), priority);  
 up(size);  
 ++size;  
 }  
  
 void push(const E& value, int priority) {  
 if (is\_full()) resize(2 \* capacity);  
 heap[size] = Node<E>(value, priority);  
 up(size);  
 ++size;  
 }  
  
 E pop() {  
 if (is\_empty()) throw std::out\_of\_range("Priority queue is empty");  
  
 E res = std::move(heap[0].data);  
  
 if (size > 1) {  
 heap[0] = std::move(heap[size - 1]);  
 --size;  
 down(0);  
 } else {  
 size = 0;  
 }  
  
 if (size < capacity / 4 && capacity > 10) resize(std::max(static\_cast<size\_t>(10), capacity / 2));  
  
 return res;  
 }  
};  
  
#endif

**Файл *test\_priority\_q.h***

//  
// Created by IWOFLEUR on 19.09.2025.  
//  
  
#ifndef TEST\_PRIORITY\_Q\_H  
#define TEST\_PRIORITY\_Q\_H  
  
void run\_tests\_priority\_q();  
void run\_demo\_priority\_q();  
  
#endif //TEST\_PRIORITY\_Q\_H

**Файл *test\_priority\_q.cpp***

//  
// Created by IWOFLEUR on 19.09.2025.  
//  
#include <cassert>  
#include <iostream>  
#include <string>  
#include "priority\_q/priority\_q.h"  
#include "test\_priority\_q.h"  
  
void run\_demo\_priority\_q() {  
 std::cout << "\n=== Priority Queue Demo ====" << std::endl;  
  
 PriorityQueue<int> pq(10);  
  
 // Add elements with different priorities  
 pq.push(5, 2);  
 pq.push(10, 5);  
 pq.push(3, 1);  
 pq.push(7, 3);  
 pq.push(15, 8);  
  
 std::cout << "Queue size: " << pq.get\_size() << std::endl;  
 std::cout << "Queue capacity: " << pq.get\_capacity() << std::endl;  
 std::cout << "Top element: " << pq.top() << " (priority: " << pq.top\_priority() << ")" << std::endl;  
  
 std::cout << "\nExtracting all elements (highest priority first):" << std::endl;  
 while (!pq.is\_empty()) {  
 std::cout << "Popped: " << pq.pop() << std::endl;  
 }  
}  
  
void run\_tests\_priority\_q() {  
 std::cout << "=== Running Priority Queue Tests ===" << std::endl;  
  
 // Test 1: Constructor and basic properties  
 std::cout << "Test 1: Constructor and basic properties... ";  
 PriorityQueue<int> pq(5);  
 assert(pq.is\_empty());  
 assert(!pq.is\_full());  
 assert(pq.get\_size() == 0);  
 assert(pq.get\_capacity() == 5);  
 std::cout << "PASSED" << std::endl;  
  
 // Test 2: Push and check top element  
 std::cout << "Test 2: Push and Top... ";  
 pq.push(10, 2);  
 assert(!pq.is\_empty());  
 assert(pq.get\_size() == 1);  
 assert(pq.top() == 10);  
 assert(pq.top\_priority() == 2);  
  
 pq.push(20, 5); // Higher priority  
 assert(pq.top() == 20); // Should be new top  
 assert(pq.top\_priority() == 5);  
 assert(pq.get\_size() == 2);  
 std::cout << "PASSED" << std::endl;  
  
 // Test 3: Pop elements  
 std::cout << "Test 3: Pop... ";  
 assert(pq.pop() == 20); // Highest priority first  
 assert(pq.get\_size() == 1);  
 assert(pq.top() == 10); // Now 10 should be top  
 assert(pq.top\_priority() == 2);  
  
 assert(pq.pop() == 10);  
 assert(pq.is\_empty());  
 std::cout << "PASSED" << std::endl;  
  
 // Test 4: Priority ordering (max-heap behavior)  
 std::cout << "Test 4: Priority ordering... ";  
 PriorityQueue<int> pq2(10);  
  
 // Insert in random order  
 pq2.push(1, 1);  
 pq2.push(9, 9);  
 pq2.push(5, 5);  
 pq2.push(3, 3);  
 pq2.push(7, 7);  
  
 // Should extract in descending priority order  
 assert(pq2.pop() == 9);  
 assert(pq2.pop() == 7);  
 assert(pq2.pop() == 5);  
 assert(pq2.pop() == 3);  
 assert(pq2.pop() == 1);  
 assert(pq2.is\_empty());  
 std::cout << "PASSED" << std::endl;  
  
 // Test 5: Edge cases and exceptions  
 std::cout << "Test 5: Edge cases and exceptions... ";  
 PriorityQueue<int> pq3(3);  
  
 // Test empty queue exceptions  
 try {  
 pq3.top();  
 assert(false); // Should not reach here  
 } catch (const std::out\_of\_range& e) {  
 assert(std::string(e.what()) == "Priority queue is empty");  
 }  
  
 try {  
 pq3.pop();  
 assert(false); // Should not reach here  
 } catch (const std::out\_of\_range& e) {  
 assert(std::string(e.what()) == "Priority queue is empty");  
 }  
  
 // Test with full queue (should resize automatically)  
 pq3.push(1, 1);  
 pq3.push(2, 2);  
 pq3.push(3, 3);  
 pq3.push(4, 4); // This should trigger resize  
  
 assert(pq3.get\_size() == 4);  
 assert(pq3.get\_capacity() > 3);  
 std::cout << "PASSED" << std::endl;  
  
 // Test 6: Move semantics  
 std::cout << "Test 6: Move semantics... ";  
 PriorityQueue<std::string> pq4(5);  
  
 std::string str = "Hello";  
 pq4.push(str, 1); // Copy  
 pq4.push(std::move(str), 2); // Move  
 pq4.push("World", 3); // Temporary  
  
 assert(pq4.pop() == "World"); // Highest priority first  
 assert(pq4.pop() == "Hello"); // Moved version  
 assert(pq4.pop() == "Hello"); // Copied version  
 std::cout << "PASSED" << std::endl;  
  
 // Test 7: Complex priority scenario  
 std::cout << "Test 7: Complex priority scenario... ";  
 PriorityQueue<int> pq5(10);  
  
 // Add elements with same priority (should maintain order for same priority)  
 for (int i = 1; i <= 5; i++) {  
 pq5.push(i \* 10, i); // Priorities 1-5  
 }  
  
 // Add elements with duplicate priorities  
 pq5.push(100, 5);  
 pq5.push(200, 3);  
  
 // Should pop in priority order (5, 5, 4, 3, 3, 2, 1)  
 // For same priority, order is not guaranteed but both 5s should come first  
 int last\_priority = 10;  
 while (!pq5.is\_empty()) {  
 int current\_priority = pq5.top\_priority();  
 assert(current\_priority <= last\_priority); // Priority should not increase  
 last\_priority = current\_priority;  
 pq5.pop();  
 }  
 std::cout << "PASSED" << std::endl;  
  
 // Test 8: Dynamic resizing  
 std::cout << "Test 8: Dynamic resizing... ";  
 PriorityQueue<int> pq6(2);  
  
 pq6.push(1, 1);  
 pq6.push(2, 2);  
 assert(pq6.get\_capacity() == 2);  
  
 pq6.push(3, 3); // Trigger resize  
 assert(pq6.get\_capacity() > 2);  
 assert(pq6.get\_size() == 3);  
  
 // Verify data integrity after resize  
 assert(pq6.pop() == 3);  
 assert(pq6.pop() == 2);  
 assert(pq6.pop() == 1);  
 assert(pq6.is\_empty());  
 std::cout << "PASSED" << std::endl;  
  
 std::cout << "\n=== All Priority Queue tests PASSED! ===" << std::endl;  
}

**Файл *queue.h***

//  
// Created by IWOFLEUR on 20.09.2025.  
//  
  
#ifndef QUEUE\_H  
#define QUEUE\_H  
#include <stdexcept>  
  
template<typename E>  
class Queue {  
 private:  
 size\_t capacity;  
 size\_t size;  
 /\*  
 \* Save rear and front elements  
 \* of queue array to properly  
 \* push and pop  
 \*/  
 size\_t rear;  
 size\_t front;  
  
 E\* heap;  
  
 void resize(const size\_t newCapacity) {  
 auto\* newHeap = new E[newCapacity];  
  
 if (front < rear) {  
 for (size\_t i = 0; i < size; i++) {  
 newHeap[i] = std::move(heap[(front + i)]);  
 }  
 } else {  
 const size\_t firstPart = capacity - front;  
 for (size\_t i = 0; i < firstPart; i++) {  
 newHeap[i] = std::move(heap[(front + i)]);  
 }  
 for (size\_t i = 0; i < rear; i++) {  
 newHeap[firstPart + i] = std::move(heap[i]);  
 }  
 }  
 delete[] heap;  
 heap = newHeap;  
 capacity = newCapacity;  
 front = 0;  
 rear = size;  
 }  
  
 public:  
 // Constructor  
 explicit Queue(const size\_t cap)  
 : capacity(cap),  
 size(0),  
 rear(0),  
 front(0),  
 heap(new E[capacity]) {}  
 // Destructor  
 ~Queue() {  
 delete[] heap;  
 }  
 // Prohibit assignment and movement  
 Queue& operator=(const Queue&) = delete;  
 Queue(const Queue&) = delete;  
 Queue& operator=(Queue&&) = delete;  
 Queue(Queue&&) = delete;  
  
 // == Basic operations ==  
 [[nodiscard]] bool is\_empty() const { return size == 0; }  
  
 [[nodiscard]] bool is\_full() const { return size == capacity; }  
  
 [[nodiscard]] size\_t get\_size() const { return size; }  
  
 [[nodiscard]] size\_t get\_capacity() const { return capacity; }  
  
 [[maybe\_unused]] const E& peek\_back() const {  
 if (is\_empty()) throw std::out\_of\_range("Queue is empty");  
 return heap[(rear == 0) ? capacity - 1 : rear - 1];  
 }  
  
 [[maybe\_unused]] const E& peek\_front() const {  
 if (is\_empty()) throw std::out\_of\_range("Queue is empty");  
 return heap[front];  
 }  
  
 void push(E&& e) {  
 if (is\_full()) resize(2 \* capacity);  
 heap[rear] = std::move(e);  
 rear = (rear + 1) % capacity;  
 ++size;  
 }  
  
 void push(const E& e) {  
 if (is\_full()) resize(2 \* capacity);  
 heap[rear] = e;  
 rear = (rear + 1) % capacity;  
 ++size;  
 }  
  
 [[maybe\_unused]] E pop() {  
 if (is\_empty()) throw std::out\_of\_range("Queue is empty");  
  
 E res = std::move(heap[front]);  
 front = (front + 1) % capacity;  
 --size;  
  
 if (size < capacity / 4 && capacity > 10) resize(std::max(static\_cast<size\_t>(10), capacity / 2));  
  
 return res;  
 }  
};  
  
#endif //QUEUE\_H

**Файл *test\_queue.h***

//  
// Created by IWOFLEUR on 20.09.2025.  
//  
  
#ifndef TEST\_QUEUE\_H  
#define TEST\_QUEUE\_H  
  
void run\_tests\_queue();  
void run\_demo\_queue();  
  
#endif //TEST\_QUEUE\_H

**Файл *test\_queue.cpp***

//  
// Created by IWOFLEUR on 20.09.2025.  
//  
#include "test\_queue.h"  
#include <cassert>  
#include <iostream>  
#include <string>  
#include "queue/queue.h"  
  
void run\_demo\_queue() {  
 std::cout << "\n=== Queue Demo ====" << std::endl;  
  
 // Create queue with initial capacity of 10  
 Queue<int> q(10);  
  
 // Add elements to the queue  
 for (int i = 10; i <= 100; i += 10) {  
 q.push(i);  
 }  
  
 std::cout << "Queue size: " << q.get\_size() << std::endl;  
 std::cout << "Capacity: " << q.get\_capacity() << std::endl;  
 std::cout << "Front element: " << q.peek\_front() << std::endl;  
 std::cout << "Back element: " << q.peek\_back() << std::endl;  
  
 std::cout << "\nExtracting all elements (FIFO order):" << std::endl;  
 while (!q.is\_empty()) {  
 std::cout << "Popped: " << q.pop() << std::endl;  
 }  
}  
  
void run\_tests\_queue() {  
 std::cout << "=== Running Queue Tests ===" << std::endl;  
  
 // Test 1: Constructor and basic properties  
 std::cout << "Test 1: Constructor and basic properties... ";  
 Queue<int> q(5);  
 assert(q.is\_empty());  
 assert(!q.is\_full());  
 assert(q.get\_size() == 0);  
 assert(q.get\_capacity() == 5);  
 std::cout << "PASSED" << std::endl;  
  
 // Test 2: Push elements and peek at front and back  
 std::cout << "Test 2: Push and Peek... ";  
 q.push(10);  
 assert(!q.is\_empty());  
 assert(q.get\_size() == 1);  
 assert(q.peek\_front() == 10);  
 assert(q.peek\_back() == 10); // Only one element, so front and back are same  
  
 q.push(20);  
 assert(q.peek\_front() == 10); // Queue is FIFO, so front should be first pushed element  
 assert(q.peek\_back() == 20); // Back should be last pushed element  
 assert(q.get\_size() == 2);  
 std::cout << "PASSED" << std::endl;  
  
 // Test 3: Pop elements from the queue  
 std::cout << "Test 3: Pop... ";  
 assert(q.pop() == 10);  
 assert(q.get\_size() == 1);  
 assert(q.peek\_front() == 20); // After pop, front should be next element  
 assert(q.peek\_back() == 20); // Only one element left  
  
 assert(q.pop() == 20);  
 assert(q.is\_empty());  
 std::cout << "PASSED" << std::endl;  
  
 // Test 4: Move semantics with string objects  
 std::cout << "Test 4: Move semantics... ";  
 Queue<std::string> q2(3);  
 std::string str = "Hello";  
 q2.push(str); // Copy constructor  
 q2.push(std::move(str)); // Move constructor  
 q2.push("World"); // Temporary object (move)  
  
 assert(q2.pop() == "Hello"); // FIFO: first in, first out  
 assert(q2.pop() == "Hello"); // Moved version (original string was moved)  
 assert(q2.pop() == "World");  
 std::cout << "PASSED" << std::endl;  
  
 // Test 5: Edge cases and exception handling  
 std::cout << "Test 5: Edge cases and exceptions... ";  
 Queue<int> q3(3);  
  
 // Test empty queue exceptions  
 try {  
 q3.peek\_front();  
 assert(false); // Should not reach this point  
 } catch (const std::out\_of\_range& e) {  
 assert(std::string(e.what()) == "Queue is empty");  
 }  
  
 try {  
 q3.peek\_back();  
 assert(false); // Should not reach this point  
 } catch (const std::out\_of\_range& e) {  
 assert(std::string(e.what()) == "Queue is empty");  
 }  
  
 try {  
 q3.pop();  
 assert(false); // Should not reach this point  
 } catch (const std::out\_of\_range& e) {  
 assert(std::string(e.what()) == "Queue is empty");  
 }  
 std::cout << "PASSED" << std::endl;  
  
 // Test 6: Specific FIFO behavior example  
 std::cout << "Test 6: Specific FIFO example... ";  
 Queue<int> q4(10);  
  
 // Fill queue with numbers 1 to 10  
 for (int i = 1; i <= 10; i++) {  
 q4.push(i);  
 }  
  
 assert(q4.get\_capacity() == 10);  
 assert(q4.get\_size() == 10);  
 assert(q4.peek\_front() == 1); // First pushed element should be at front  
 assert(q4.peek\_back() == 10); // Last pushed element should be at back  
  
 // Verify FIFO order (First In, First Out)  
 for (int i = 1; i <= 10; i++) {  
 assert(q4.pop() == i);  
 }  
  
 assert(q4.is\_empty());  
 assert(q4.get\_size() == 0);  
 std::cout << "PASSED" << std::endl;  
  
 // Test 7: Dynamic resizing functionality  
 std::cout << "Test 7: Dynamic resizing... ";  
 Queue<int> q5(2); // Small initial capacity to trigger resizing  
  
 q5.push(1);  
 q5.push(2);  
 assert(q5.get\_capacity() == 2);  
  
 // This should trigger automatic resizing  
 q5.push(3);  
 assert(q5.get\_capacity() > 2);  
 assert(q5.get\_size() == 3);  
  
 // Verify data integrity after resize operation (FIFO order maintained)  
 assert(q5.pop() == 1);  
 assert(q5.pop() == 2);  
 assert(q5.pop() == 3);  
 assert(q5.is\_empty());  
 std::cout << "PASSED" << std::endl;  
  
 // Test 8: Queue with different data types  
 std::cout << "Test 8: Different data types... ";  
 Queue<double> doubleQueue(5);  
 doubleQueue.push(3.14);  
 doubleQueue.push(2.71);  
 assert(doubleQueue.pop() == 3.14);  
 assert(doubleQueue.pop() == 2.71);  
  
 Queue<char> charQueue(3);  
 charQueue.push('A');  
 charQueue.push('B');  
 assert(charQueue.pop() == 'A');  
 assert(charQueue.pop() == 'B');  
 std::cout << "PASSED" << std::endl;  
  
 std::cout << "\n=== All Queue tests PASSED! ===" << std::endl;  
}

**Файл *stack.h***

//  
// Created by IWOFLEUR on 20.09.2025.  
//  
  
#ifndef STACK\_H  
#define STACK\_H  
#include <stdexcept>  
  
template<typename E>  
class Stack {  
 private:  
 size\_t capacity; // Max size of our stack  
 size\_t size;  
 size\_t top; // Top of the stack  
 E\* heap;  
  
 void resize(const size\_t newCapacity) {  
 E\* newHeap = new E[newCapacity];  
 for (size\_t i = 0; i <= top; i++) {  
 newHeap[i] = std::move(heap[i]);  
 }  
 delete[] heap;  
 heap = newHeap;  
 capacity = newCapacity;  
 }  
  
 public:  
 // Constructor  
 explicit Stack(const size\_t cap) : capacity(cap), size(0), top(-1), heap(new E[cap]) {}  
 // Destructor  
 ~Stack() {  
 delete[] heap;  
 }  
 // Prohibit assignment and movement  
 Stack(const Stack&) = delete;  
 Stack& operator=(const Stack&) = delete;  
 Stack(Stack&&) = delete;  
 Stack& operator=(Stack&&) = delete;  
  
 // === Basic operations ===  
 // True if stack is empty  
 [[nodiscard]] bool is\_empty() const { return top == -1; }  
 // True if stack is full  
 [[nodiscard]] bool is\_full() const { return top == capacity - 1; }  
  
 [[nodiscard]] size\_t get\_size() const { return size; }  
  
 [[nodiscard]] size\_t get\_capacity() const { return capacity; }  
 // Copy existing object  
 void push(const E& e) {  
 if (is\_full()) resize(2 \* capacity);  
 heap[++top] = e;  
 ++size;  
 }  
 // Move object or move temporary  
 void push(E&& e) {  
 if (is\_full()) resize(2 \* capacity);  
 heap[++top] = std::move(e);  
 ++size;  
 }  
  
 [[maybe\_unused]] E pop() {  
 if (is\_empty()) throw std::out\_of\_range("Stack is empty");  
 E result = std::move(heap[top--]);  
 --size;  
  
 if (size < capacity / 4 && capacity > 10) resize(std::max(static\_cast<size\_t>(10), capacity / 2));  
  
 return result;  
 }  
  
 [[maybe\_unused]] E peek() const {  
 if (is\_empty()) throw std::out\_of\_range("Stack is empty");  
 return heap[top];  
 }  
};  
  
#endif //STACK\_H

**Файл *test\_stack.h***

//  
// Created by IWOFLEUR on 21.09.2025.  
//  
  
#ifndef TEST\_STACK\_H  
#define TEST\_STACK\_H  
  
void run\_tests\_stack();  
void run\_demo\_stack();  
  
#endif //TEST\_STACK\_H

**Файл *test\_stack.cpp***

//  
// Created by IWOFLEUR on 20.09.2025.  
//  
#include "test\_stack.h"  
#include <cassert>  
#include <iostream>  
#include <string>  
#include "stack/stack.h"  
  
void run\_demo\_stack() {  
 std::cout << "\n=== Stack Demo ====" << std::endl;  
  
 // Create stack with initial capacity of 10  
 Stack<int> s(10);  
  
 // Add elements to the stack  
 for (int i = 1; i <= 10; i++) {  
 s.push(i);  
 }  
  
 std::cout << "Stack size: " << s.get\_size() << std::endl;  
 std::cout << "Capacity: " << s.get\_capacity() << std::endl;  
 std::cout << "Top element: " << s.peek() << std::endl;  
  
 std::cout << "\nExtracting all elements (LIFO order):" << std::endl;  
 while (!s.is\_empty()) {  
 std::cout << "Popped: " << s.pop() << std::endl;  
 }  
}  
  
void run\_tests\_stack() {  
 std::cout << "=== Running Stack Tests ===" << std::endl;  
  
 // Test 1: Constructor and basic properties  
 std::cout << "Test 1: Constructor and basic properties... ";  
 Stack<int> s(5);  
 assert(s.is\_empty());  
 assert(!s.is\_full());  
 assert(s.get\_size() == 0);  
 assert(s.get\_capacity() == 5);  
 std::cout << "PASSED" << std::endl;  
  
 // Test 2: Push elements and peek at the top  
 std::cout << "Test 2: Push and Peek... ";  
 s.push(10);  
 assert(!s.is\_empty());  
 assert(s.get\_size() == 1);  
 assert(s.peek() == 10);  
  
 s.push(20);  
 assert(s.peek() == 20); // Stack is LIFO, so top should be the last pushed element  
 assert(s.get\_size() == 2);  
 std::cout << "PASSED" << std::endl;  
  
 // Test 3: Pop elements from the stack  
 std::cout << "Test 3: Pop... ";  
 assert(s.pop() == 20);  
 assert(s.get\_size() == 1);  
 assert(s.pop() == 10);  
 assert(s.is\_empty());  
 std::cout << "PASSED" << std::endl;  
  
 // Test 4: Move semantics with string objects  
 std::cout << "Test 4: Move semantics... ";  
 Stack<std::string> s2(3);  
 std::string str = "Hello";  
 s2.push(str); // Copy constructor  
 s2.push(std::move(str)); // Move constructor  
 s2.push("World"); // Temporary object (move)  
  
 assert(s2.pop() == "World");  
 assert(s2.pop() == "Hello"); // Moved version  
 assert(s2.pop() == "Hello"); // Copied version  
 std::cout << "PASSED" << std::endl;  
  
 // Test 5: Edge cases and exception handling  
 std::cout << "Test 5: Edge cases and exceptions... ";  
 Stack<int> s3(3);  
  
 // Test empty stack exceptions  
 try {  
 s3.peek();  
 assert(false); // Should not reach this point  
 } catch (const std::out\_of\_range& e) {  
 assert(std::string(e.what()) == "Stack is empty");  
 }  
  
 try {  
 s3.pop();  
 assert(false); // Should not reach this point  
 } catch (const std::out\_of\_range& e) {  
 assert(std::string(e.what()) == "Stack is empty");  
 }  
 std::cout << "PASSED" << std::endl;  
  
 // Test 6: Specific LIFO behavior example  
 std::cout << "Test 6: Specific LIFO example... ";  
 Stack<int> s4(10);  
  
 // Fill stack with numbers 1 to 10  
 for (int i = 1; i <= 10; i++) {  
 s4.push(i);  
 }  
  
 assert(s4.get\_capacity() == 10);  
 assert(s4.get\_size() == 10);  
 assert(s4.peek() == 10); // Last pushed element should be on top  
  
 // Verify LIFO order (Last In, First Out)  
 for (int i = 10; i >= 1; i--) {  
 assert(s4.pop() == i);  
 }  
  
 assert(s4.is\_empty());  
 assert(s4.get\_size() == 0);  
 std::cout << "PASSED" << std::endl;  
  
 // Test 7: Dynamic resizing functionality  
 std::cout << "Test 7: Dynamic resizing... ";  
 Stack<int> s5(2); // Small initial capacity to trigger resizing  
  
 s5.push(1);  
 s5.push(2);  
 assert(s5.get\_capacity() == 2);  
  
 // This should trigger automatic resizing  
 s5.push(3);  
 assert(s5.get\_capacity() > 2);  
 assert(s5.get\_size() == 3);  
  
 // Verify data integrity after resize operation  
 assert(s5.pop() == 3);  
 assert(s5.pop() == 2);  
 assert(s5.pop() == 1);  
 assert(s5.is\_empty());  
 std::cout << "PASSED" << std::endl;  
  
 // Test 8: Stack with different data types  
 std::cout << "Test 8: Different data types... ";  
 Stack<double> doubleStack(5);  
 doubleStack.push(3.14);  
 doubleStack.push(2.71);  
 assert(doubleStack.pop() == 2.71);  
 assert(doubleStack.pop() == 3.14);  
  
 Stack<char> charStack(3);  
 charStack.push('A');  
 charStack.push('B');  
 assert(charStack.pop() == 'B');  
 assert(charStack.pop() == 'A');  
 std::cout << "PASSED" << std::endl;  
  
 std::cout << "\n=== All Stack tests PASSED! ===" << std::endl;  
}

**Файл *utils.h***

//  
// Created by IWOFLEUR on 24.09.2025.  
//  
  
#ifndef UTILS\_H  
#define UTILS\_H  
#include <string>  
  
namespace Utils {  
 void clear\_input\_buffer();  
 std::string to\_lower(const std::string& str);  
 bool is\_valid\_mode(const std::string& str);  
 std::string get\_valid\_mode();  
 bool get\_confirm(const std::string& msg);  
 void run\_free\_mode();  
}  
  
#endif //UTILS\_H

**Файл *utils.cpp***

//  
// Created by IWOFLEUR on 24.09.2025.  
//  
  
#include "utils.h"  
  
#include <iostream>  
#include <limits>  
  
namespace Utils {  
 void clear\_input\_buffer() {  
 std::cin.clear();  
 std::cin.ignore(std::numeric\_limits<std::streamsize>::*max*(), '\n');  
 }  
  
 std::string to\_lower(const std::string &str) {  
 std::string result = str;  
 for (char& c : result) {  
 c = std::tolower(static\_cast<unsigned char>(c));  
 }  
 return result;  
 }  
  
 bool is\_valid\_mode(const std::string &str) {  
 const std::string lower = to\_lower(str);  
 return lower == "demo" || lower == "test" || lower == "free" || lower == "exit";  
 }  
  
 std::string get\_valid\_mode() {  
 std::string mode;  
  
 while (true) {  
 std::cout << "\nSelect mode:" << std::endl;  
 std::cout << "1. demo - automatic push and pop elements" << std::endl;  
 std::cout << "2. test - run all tests" << std::endl;  
 std::cout << "3. free - run your own code" << std::endl;  
 std::cout << "4. exit - stop the program" << std::endl;  
 std::cout << "Enter 'demo', 'test', 'free' or 'exit': ";  
  
 std::cin >> mode;  
  
 if (is\_valid\_mode(mode)) {  
 return to\_lower(mode);  
 }  
 std::cout << "Invalid input '" << mode << "'. Please try again." << std::endl;  
 clear\_input\_buffer();  
 }  
 }  
  
  
 bool get\_confirm(const std::string &msg) {  
 std::string input;  
  
 while (true) {  
 std::cout << msg << " (y/n): ";  
 std::cin >> input;  
 input = to\_lower(input);  
  
 if (input == "y" || input == "yes") {  
 clear\_input\_buffer();  
 return true;  
 }  
 if (input == "n" || input == "no") {  
 clear\_input\_buffer();  
 return false;  
 }  
 std::cout << "Please enter 'y' or 'n'." << std::endl;  
 clear\_input\_buffer();  
 }  
 }  
}

**Файл *main.cpp***

#include <iostream>  
#include <string>  
  
#include "priority\_q\_tests/test\_priority\_q.h"  
#include "queue\_tests/test\_queue.h"  
#include "stack\_tests/test\_stack.h"  
#include "utils/utils.h"  
  
int main() {  
 /\*  
 \* There`s three modes:  
 \* 1. demo - automatically push and pop elements  
 \* 2. test - do every sort of tests  
 \* 3. free - write your own code and have fun  
 \*/  
 while (true) {  
 std::string mode = Utils::get\_valid\_mode();  
  
 try {  
 if (mode == "test") {  
 std::cout << "=== Running Tests ===" << std::endl;  
 run\_tests\_priority\_q();  
 run\_tests\_queue();  
 run\_tests\_stack();  
 std::cout << "\n=== All Tests Completed ===" << std::endl;  
 }  
 if (mode == "demo") {  
 std::cout << "\n=== Running Demos ===" << std::endl;  
 run\_demo\_priority\_q();  
 run\_demo\_queue();  
 run\_demo\_stack();  
 }  
 if (mode == "free") {  
 std::cout << "\n=== Running Free Mode ===" << std::endl;  
 Utils::run\_free\_mode(); // <- write your code here  
 std::cout << "\n=== Free mode is completed ===" << std::endl;  
 }  
 if (mode == "exit") {  
 if (Utils::get\_confirm("Are you sure?")) {  
 std::cout << "Exiting program. Goodbye!" << std::endl;  
 break;  
 }  
 }  
 } catch (const std::exception& e) {  
 std::cerr << "Error: " << e.what() << std::endl;  
 return 1;  
 }  
 }  
  
 return 0;  
}  
  
void Utils::run\_free\_mode() {  
 /\* \*/  
 std::cout << "Empty!" << std::endl;  
}