**МИНОБРНАУКИ РОССИИ**

**САНКТ-ПЕТЕРБУРГСКИЙ ГОСУДАРСТВЕННЫЙ**

**ЭЛЕКТРОТЕХНИЧЕСКИЙ УНИВЕРСИТЕТ**

**«ЛЭТИ» ИМ. В.И. УЛЬЯНОВА (ЛЕНИНА)**

**Кафедра МО ЭВМ**

**ОТЧЕТ**

**по лабораторной работе №1**

**по дисциплине «Алгоритмы и структуры данных»**

**Тема: Реализация и исследование кольцевого буфера**

|  |  |  |
| --- | --- | --- |
| Студент гр. 4344 |  | Байдаков Г.И. |
| Преподаватель |  | Корытов П.В. |

Санкт-Петербург

2025

## Цель работы.

Разработать и реализовать класс `CircularDeque` на основе статического массива, обеспечивающий эффективное управление кольцевой декой целых чисел. Класс должен поддерживать операции добавления, удаления и доступа к элементам с обоих концов деки за константное время, а также гибкое изменение размера буфера с сохранением порядка элементов. Особое внимание уделить корректной обработке ситуаций переполнения буфера в зависимости от заданного флага `push\_force`.

## Задание.

Требуется реализовать кольцевой буфер (а именно кольцевую деку) для целых чисел на основе статического массива длины N. Реализация должна представлять из себя класс *CircularDeque*. В поле предварительного ответа предложены сигнатуры класса для python и c++.

В конструкторе класс должен принимать количество элементов *N* в массиве, а также значение *bool push\_force*, отражающее, будут ли при переполнении буфера элементы перезаписывать старые.

Класс должен поддерживать следующие методы (каждый за O(1) в худшем случае или амортизированно):

* *void push\_front(x)* – добавить элемент *x* в начало деки;
* *void push\_back(x)* – добавить элемент *x* в конец деки;

Если дека заполнена, а *push\_force* указан как *false,* при попытке добавить элементы в деку должно выбрасываться исключение; если же *push\_force* указан как *true*, то при добавлении элемента в конец деки он должен записаться поверх элемента с начала деки, а при добавлении элемента в начало - поверх элемента с конца. При этом начало/конец деки должно сместиться на последующий/предыдущий элемент.

*Пример:*

Пусть дека вместимости 3 заполнена:

[1, 2, 3], front = 1, back = 3, push\_force = true.

При выполнении push\_back(4) элемент 4 запишется поверх начала деки - элемента 1, а само начало деки сдвинется на следующий элемент - 2:

[4, 2, 3], front = 2, back = 4.

* *int pop\_front()* – удалить и вернуть элемент из начала деки;
* *int pop\_back()* – удалить и вернуть элемент из конца деки;
* *int front()* – вернуть (не удаляя сам элемент) значение первого элемента в деке;
* *int back()* – вернуть значение последнего элемента;
* *int size()* – вернуть текущее число элементов в деке;
* *bool empty()* – проверить деку на пустоту;
* *bool full()* – проверить деку на заполненность (если массив полон).
* *void resize(new\_capacity) –* изменить размер статического массива и перезаписать элементы в массив нового размера. При перезаписи необходимо перемещать элементы в правильном порядке - от начала деки к её концу. Если не все элементы старого массива помещаются в новый, необходимо оставить лишь *new\_capacity* первых (с начала деки) элементов массива.

Если выполнить какую-либо операцию невозможно, необходимо выбросить исключение.

## Основные теоретические положения.

Дека (Deque), или двусторонняя очередь (Double-Ended Queue), представляет собой абстрактный тип данных, который является обобщением очереди и стека. Отличительной особенностью деки является то, что добавление и удаление элементов возможно как с начала, так и с конца структуры.

Кольцевая Дека – это реализация деки, использующая кольцевой буфер для хранения элементов. Основная идея кольцевого буфера заключается в том, что когда указатель достигает конца физического массива, он "переносится" в его начало, образуя таким образом логически непрерывную структуру. Это позволяет эффективно использовать память, выделенную под массив, и избегать дорогостоящих операций сдвига элементов при добавлении или удалении.

## Выполнение работы.

Решение поставленной задачи по реализации кольцевой деки (класса CircularDeque) для целых чисел основано на использовании статического массива и двух указателей (индексов) для отслеживания начала (head) и конца (tail) деки.

**Описание класса CircularDeque**

Класс CircularDeque инкапсулирует логику работы с кольцевой декой и содержит следующие поля и методы:

**Поля класса:**

* int \*buffer: Указатель статический массив для хранения элементов деки.
* size\_t capacity: Переменная типа size\_t, хранящая максимальное количество элементов, которое может вместить дека. Задается при инициализации.
* bool is\_push\_force: Переменная типа bool, флаг, который определяет поведение операций push\_back и push\_front при переполнении деки. Если true, то элементы будут перезаписываться при переполнении, иначе будет выброшено исключение.
* size\_t head: Переменная типа size\_t, являющаяся индексом первого элемента в деке. Указывает на начало деки.
* size\_t tail: Переменная типа size\_t, являющаяся индексом следующей свободной позиции для добавления элемента в конец деки. Указывает на конец деки.
* size\_t count: Переменная типа size\_t, хранящая текущее количество элементов, находящихся в деке.

**Конструктор класса:**

* CircularDeque(size\_t N, bool push\_force): Конструктор, принимающий два аргумента.
  + N (тип size\_t): Задает максимальную вместимость деки.
  + push\_force (тип bool): Инициализирует флаг is\_push\_force.  
    В конструкторе инициализируются все поля класса: capacity устанавливается в N, head, tail, current\_size инициализируются нулями is\_push\_force получает значение push\_force. Выделяется память под массив buffer для N элементов типа int.

**Деструктор класса:**

* ~CircularDeque(): Деструктор, очищает выделенную память массива buffer.

Описание методов класса

Все методы, за исключением resize, реализованы таким образом, чтобы выполняться за амортизированное O(1) время, что соответствует требованиям задачи.

1. **size\_t nextIdx(size\_t idx)**
   * Описание: Вычисляет следующий индекс.
   * Аргументы: size\_t idx – добавляемый элемент.
   * Возвращаемое значение: size\_t– следующий индекс.
   * Логика:
     + Возвращает значение следующего за idx индекса по формуле (idx + 1) % capacity, таким образом происходит «связывание» массива в кольцо.
2. **size\_t prevIdx(size\_t idx)**
   * Описание: Вычисляет предыдущий индекс.
   * Аргументы: size\_t idx – добавляемый элемент.
   * Возвращаемое значение: size\_t– предыдущий индекс.
   * Логика:
     + Возвращает значение предыдущего индекса по формуле (idx + capacity - 1) % capacity, таким образом происходит «связывание» массива в кольцо.
3. **void push\_front(int x)**
   * Описание: Добавляет элемент x в начало деки.
   * Аргументы: int x – добавляемый элемент.
   * Возвращаемое значение:void – отсутствует.
   * Логика:
     + Проверяется состояние деки на полноту с помощью full().
     + Если дека полна и is\_push\_force равно false, выбрасывается исключение std::overflow\_error.
     + Если дека полна и is\_push\_force равно true:
       - Индекс head сдвигается на предыдущую позицию методом prevIdx
       - Элемент x записывается в позицию buffer[head].
       - Индекс tail сдвигается на предыдущую позицию методом prevIdx, чтобы сохранить корректную позицию для следующего добавления в конец
     + Если дека не полна:
       - Индекс head сдвигается на предыдущую позицию методом prevIdx
       - Элемент x записывается в позицию buffer[head].
       - Увеличивается count на единицу.
4. **void push\_back(int x)**
   * Описание: Добавляет элемент x в конец деки.
   * Аргументы: int x – добавляемый элемент.
   * Возвращаемое значение:void – отсутствует.
   * Логика:
     + Проверяется состояние деки на полноту с помощью full().
     + Если дека полна и is\_push\_force равно false, выбрасывается исключение std::overflow\_error.
     + Если дека заполнена и is\_push\_force = true:
       - Элемент x записывается в позицию buffer[tail].
       - Индекс head сдвигается на следующую позицию методом nextIdx.
       - Индекс tail сдвигается на следующую позицию методом nextIdx.
     + Если дека не полна:
       - Элемент x записывается в позицию buffer[tail].
       - Индекс tail сдвигается на следующую позицию методом nextIdx.
       - Увеличивается count на единицу.
5. **int pop\_front()**
   * Описание: Удаляет и возвращает элемент из начала деки.
   * Аргументы: Отсутствуют.
   * Возвращаемое значение:int val – удаленный элемент.
   * Логика:
     + Проверяется состояние деки на пустоту с помощью empty(). Если дека пуста, выбрасывается исключение std::underflow\_error.
     + Значение элемента по индексу head сохраняется в локальной переменной val.
     + Индекс head сдвигается на следующую позицию методом nextIdx.
     + Уменьшается count на единицу.
     + Возвращается val.
6. **int pop\_back()**
   * Описание: Удаляет и возвращает элемент из конца деки.
   * Аргументы: Отсутствуют.
   * Возвращаемое значение: int – удаленный элемент.
   * Логика:
     + Проверяется состояние деки на пустоту с помощью empty(). Если дека пуста, выбрасывается исключение std::underflow\_error.
     + Индекс tail сдвигается на предыдущую позицию методом prevIdx.
     + Уменьшается count на единицу.
     + Возвращается buffer[tail], так как tail указывает на последний элемент.
7. **int& front()**
   * Описание: Возвращает ссылку на первый элемент в деке (без удаления).
   * Аргументы: Отсутствуют.
   * Возвращаемое значение: int& - Ссылка на первый элемент.
   * Логика:
     + Возвращается ссылка на элемент buffer[head].
8. **int& back()**
   * Описание: Возвращает ссылку на последний элемент в деке (без удаления).
   * Аргументы: Отсутствуют.
   * Возвращаемое значение: int& - Ссылка на последний элемент.
   * Логика:
     + Возвращается ссылка на элемент, находящийся по предыдущему индексу перед tail, вычисленному с помощью метода prevIdx(tail)
9. **int size() const**
   * Описание: Возвращает текущее количество элементов в деке.
   * Аргументы: Отсутствуют.
   * Возвращаемое значение: size\_t – count.
   * Логика: Возвращает значение поля count.
10. **bool empty() const**
    * Описание: Проверяет, пуста ли дека.
    * Аргументы: Отсутствуют.
    * Возвращаемое значение: bool - true, если дека пуста, иначе false.
    * Логика: Возвращает результат сравнения count == 0.
11. **bool full() const**
    * Описание: Проверяет, заполнена ли дека.
    * Аргументы: Отсутствуют.
    * Возвращаемое значение: bool - true, если дека заполнена, иначе false.
    * Логика: Возвращает результат сравнения count == capacity.
12. **void resize(size\_t new\_cap)**
    * Описание: Изменяет размер внутреннего статического массива деки и перезаписывает элементы.
    * Аргументы: size\_t new\_cap – новая вместимость деки.
    * Возвращаемое значение: void - Отсутствует.
    * Логика:
      + Если new\_cap == 0, возбуждается исключение std::invalid\_argument
      + Создается новый временный вектор new\_buffer размером new\_cap.
      + Определяется new\_count – количество элементов, которое будет скопировано. Это минимальное значение из count и new\_cap, чтобы избежать выхода за границы нового массива или копирования несуществующих элементов.
      + В цикле for элементы из старого data копируются в new\_data в правильном порядке (от head к tail по логическому порядку деки). Для этого используется (head + i) % capacity для доступа к элементам в старом кольцевом буфере.
      + Старый массив buffer удаляется, а новый new\_buffer присваивается полю buffer, тем самым заменяя массив.
      + Обновляются поля capacity = new\_cap, head = 0, current\_size = new\_count и tail = new\_count % new\_cap.

Разработанный программный код см. в приложении А.

## Pipeline данных.

Представим CircularDeque deque(4); (емкость 4). “\_” означает неинициализированные данные.

Начальное состояние:

* data: [ \_, \_, \_, \_ ]
* head: 0
* tail: 0
* current\_size: 0

1. deque.push\_back(10);

* Дек неполон. Значение 10 записывается в data[tail], то есть data[0].
* tail сдвигается на 1. current\_size увеличивается.
* Состояние:
  + data: [ 10, \_, \_, \_ ]
  + head: 0, tail: 1, current\_size: 1

2. deque.push\_back(20);

* Значение 20 записывается в data[tail], то есть data[1].
* tail сдвигается на 2. current\_size увеличивается.
* Состояние:
  + data: [ 10, 20, \_, \_ ]
  + head: 0, tail: 2, current\_size: 2

3. deque.push\_front(5);

* Дек неполон. head сдвигается назад: (0 - 1 + 4) % 4 становится 3.
* Значение 5 записывается в data[head], то есть data[3].
* current\_size увеличивается.
* Состояние:
  + data: [ 10, 20, \_, 5 ]
  + head: 3, tail: 2, current\_size: 3
  + Логически дек сейчас выглядит как [5, 10, 20].

4. deque.pop\_back();

* tail сдвигается назад: (2 - 1 + 4) % 4 становится 1.
* Возвращается значение из data[1], то есть 20.
* current\_size уменьшается.
* Состояние:
  + data: [ 10, 20, \_, 5 ] (значение 20 все еще там, но недоступно)
  + head: 3, tail: 1, current\_size: 2
  + Логически дек теперь [5, 10].

5. deque.push\_back(30);

* Значение 30 записывается в data[tail], то есть data[1].
* tail сдвигается на 2.
* Состояние:
  + data: [ 10, 30, \_, 5 ]
  + head: 3, tail: 2, current\_size: 3

6. deque.push\_back(40);

* Значение 40 записывается в data[tail], то есть data[2].
* tail сдвигается на 3. Дек теперь полон.
* Состояние:
  + data: [ 10, 30, 40, 5 ]
  + head: 3, tail: 3, current\_size: 4
  + Логически дек [5, 10, 30, 40].

7. deque.push\_back(50); (с push\_force\_enabled = true)

* Дек полон. push\_force сработает.
* 50 записывается в data[tail], то есть data[3]. Перезаписывается 5.
* head сдвигается вперед: (3 + 1) % 4 становится 0.
* tail сдвигается вперед: (3 + 1) % 4 становится 0.
* Состояние:
  + data: [ 10, 30, 40, 50 ]
  + head: 0, tail: 0, current\_size: 4
  + Логически дек теперь [10, 30, 40, 50].

## Исследование.

Для подтверждения корректности работы и оценки эффективности класса CircularDeque проводилось тестирование его базовых операций. Центральной задачей стало экспериментальное доказательство постоянного времени выполнения операций добавления элементов — push\_back и push\_front, что соответствует сложности O(1).

Процедура тестирования строилась вокруг измерения скорости вставки при различных начальных условиях. С целью исключения влияния операций переаллокации, требующих линейного времени, создавалась дека с значительным запасом памяти. На каждом этапе испытаний структура предварительно заполнялась переменным числом элементов — от нуля до десяти миллионов.

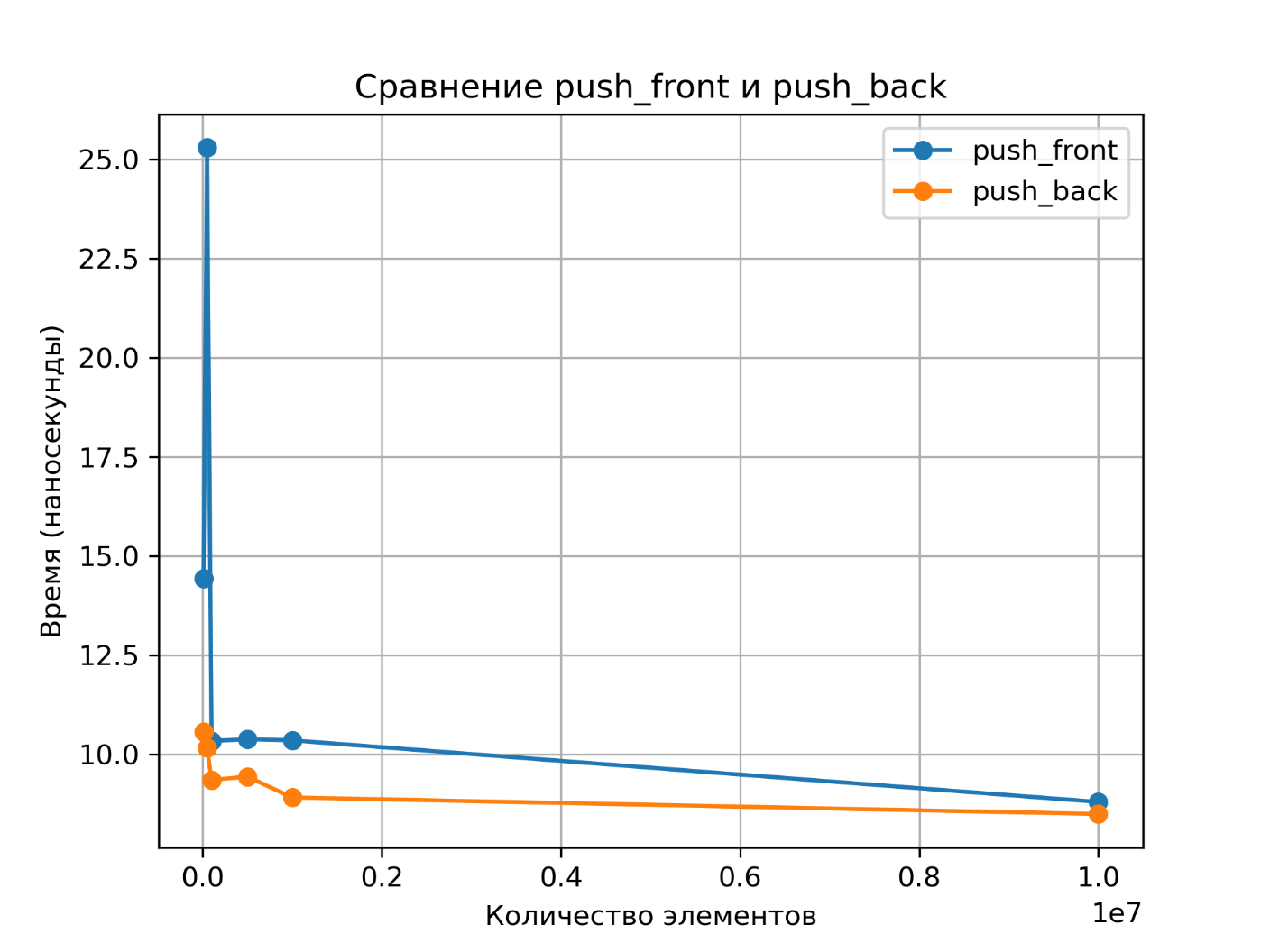
Для всех состояний дека выполнялись многочисленные циклы операций вставки, при этом фиксировалась общая продолжительность их выполнения. Полученные временные показатели делились на количество операций, что позволяло вычислить среднюю длительность одной вставки.

Рисунок 1 – График зависимости среднего времени выполнения операций вставки от начального размера деки

Отсутствие восходящего тренда наглядно подтверждает, что сложность этих операций действительно является константной, или O(1), как и требовалось в задании.

## Выводы.

В рамках проекта успешно создана и проанализирована программная реализация кольцевой двусторонней очереди на базе статического массива. Разработанный класс инкапсулирует всю функциональность кольцевого буфера для работы с целочисленными значениями, предоставляя полный спектр операций управления декой. В качестве основы для реализации был применен массив, а контроль границ данных осуществлялся через индексы head и tail. Циклическая организация буфера достигалась за счет использования модульной арифметики, гарантируя постоянную скорость выполнения базовых операций.

Обработка ошибочных сценариев, включая попытки извлечения из пустой структуры или добавления в переполненную дека, была организована через механизм исключений. Реализация режима принудительной записи потребовала введения дополнительных проверок состояния буфера. Операция изменения размера была выполнена через создание нового массива с последующим переносом элементов с сохранением их логической последовательности.

Практические испытания производительности убедительно продемонстрировали постоянное время выполнения операций вставки, что подтвердило соответствие реализации теоретическим ожиданиям. Полученные результаты свидетельствуют о создании эффективного и надежного инструмента для задач, требующих использования двусторонней очереди с предсказуемыми характеристиками производительности.

# **ПРИЛОЖЕНИЕ А ИСХОДНЫЙ КОД ПРОГРАММЫ**

Сначала указываем имя файла, в котором код лежит в репозитории:

Название файла: circular-buffer.cpp

#include "circular-buffer.hpp"

CircularDeque::CircularDeque(size\_t N, bool push\_force):

capacity(N), is\_push\_force(push\_force), head(0), tail(0), count(0)

{

buffer = new int[N];

}

CircularDeque::~CircularDeque()

{

delete[] buffer;

}

size\_t CircularDeque::nextIdx(size\_t idx)

{

return (idx + 1) % capacity;

}

size\_t CircularDeque::prevIdx(size\_t idx)

{

return (idx + capacity - 1) % capacity;

}

void CircularDeque::push\_front(int x)

{

if (full()) {

if (!is\_push\_force) throw std::overflow\_error("Deque is full");

head = prevIdx(head);

buffer[head] = x;

tail = prevIdx(tail);

} else {

head = prevIdx(head);

buffer[head] = x;

count++;

}

}

void CircularDeque::push\_back(int x)

{

if (full()) {

if (!is\_push\_force) throw std::overflow\_error("Deque is full");

buffer[tail] = x;

tail = nextIdx(tail);

head = nextIdx(head);

} else {

buffer[tail] = x;

tail = nextIdx(tail);

count++;

}

}

int CircularDeque::pop\_front()

{

if (empty()) throw std::underflow\_error("Deque is empty");

int val = buffer[head];

head = nextIdx(head);

count--;

return val;

}

int CircularDeque::pop\_back()

{

if (empty()) throw std::underflow\_error("Deque is empty");

tail = prevIdx(tail);

count--;

return buffer[tail];

}

int& CircularDeque::front() {

return buffer[head];

}

int& CircularDeque::back() {

return buffer[prevIdx(tail)];

}

int CircularDeque::size() const

{

return count;

}

bool CircularDeque::empty() const

{

return count == 0;

}

bool CircularDeque::full() const

{

return count == capacity;

}

void CircularDeque::resize(size\_t new\_cap)

{

if (new\_cap == 0) throw std::invalid\_argument("Capacity must be positive");

int \*new\_buffer = new int[new\_cap];

size\_t new\_count = std::min(count, new\_cap);

for (int i = 0; i < new\_count; i++) {

new\_buffer[i] = buffer[(head + i) % capacity];

}

delete[] buffer;

buffer = new\_buffer;

capacity = new\_cap;

head = 0;

tail = new\_count % new\_cap;

count = new\_count;

}

Название файла: circular-buffer.hpp

#include <cstddef>

#include <stdexcept>

class CircularDeque

{

int \*buffer;

size\_t capacity;

bool is\_push\_force;

size\_t head;

size\_t tail;

size\_t count;

public:

explicit CircularDeque(size\_t N, bool push\_force = false);

void push\_front(int x);

void push\_back(int x);

int pop\_front();

int pop\_back();

int& front();

int& back();

int size() const;

bool empty() const;

bool full() const;

void resize(size\_t new\_cap);

~CircularDeque();

private:

size\_t nextIdx(size\_t idx);

size\_t prevIdx(size\_t idx);

};

Название файла: tests.cpp

#include <gtest/gtest.h>

#include "circular-buffer.hpp"

using namespace std;

TEST(CircularDequeTest, Initially) {

CircularDeque deq(3, false);

EXPECT\_TRUE(deq.empty());

EXPECT\_FALSE(deq.full());

EXPECT\_EQ(deq.size(), 0);

}

TEST(CircularDequeTest, PushBack) {

CircularDeque deq(3, false);

deq.push\_back(3);

deq.push\_back(2);

deq.push\_back(1);

EXPECT\_TRUE(deq.full());

EXPECT\_EQ(deq.size(), 3);

EXPECT\_EQ(deq.front(), 3);

EXPECT\_EQ(deq.back(), 1);

}

TEST(CircularDequeTest, PushFront) {

CircularDeque deq(3, false);

deq.push\_front(3);

deq.push\_front(2);

deq.push\_front(1);

EXPECT\_TRUE(deq.full());

EXPECT\_EQ(deq.size(), 3);

EXPECT\_EQ(deq.front(), 1);

EXPECT\_EQ(deq.back(), 3);

}

TEST(CircularDequeTest, PushBackOverflow) {

CircularDeque dq(2, false);

dq.push\_back(1);

dq.push\_back(2);

EXPECT\_THROW(dq.push\_back(3), std::overflow\_error);

}

TEST(CircularDequeTest, PushForce) {

CircularDeque dq(3, true);

dq.push\_back(1);

dq.push\_back(2);

dq.push\_back(3);

dq.push\_back(4);

EXPECT\_EQ(dq.front(), 2);

EXPECT\_EQ(dq.back(), 4);

EXPECT\_EQ(dq.size(), 3);

dq.push\_front(5);

EXPECT\_EQ(dq.front(), 5);

EXPECT\_EQ(dq.back(), 3);

}

TEST(CircularDequeTest, ResizeUp) {

CircularDeque dq(2, false);

dq.push\_back(1);

dq.push\_back(2);

dq.resize(4);

EXPECT\_EQ(dq.size(), 2);

EXPECT\_EQ(dq.front(), 1);

dq.push\_back(3);

dq.push\_back(4);

EXPECT\_EQ(dq.back(), 4);

EXPECT\_TRUE(dq.full());

}

TEST(CircularDequeTest, ResizeDown) {

CircularDeque dq(5, false);

dq.push\_back(1);

dq.push\_back(2);

dq.push\_back(3);

dq.push\_back(4);

dq.push\_back(5);

dq.resize(3);

EXPECT\_EQ(dq.size(), 3);

EXPECT\_EQ(dq.front(), 1);

EXPECT\_EQ(dq.back(), 3);

}

int main(int argc, char \*\*argv) {

::testing::InitGoogleTest(&argc, argv);

return RUN\_ALL\_TESTS();

}

Название файла: execution-time-test.cpp

#include <exception>

#include <iostream>

#include <string>

#include "circular-buffer.hpp"

#include <chrono>

using namespace std;

using namespace std::chrono;

int main(int argc, char \*\*argv) {

if (argc < 2) {

cout << "Usage: " << argv[0] << " <type> <number\_of\_elements>" << endl;

return 1;

}

try {

int type = std::stoi(argv[1]);

int n = std::stoi(argv[2]);

CircularDeque cd = CircularDeque(n);

auto start = high\_resolution\_clock::now();

switch (type) {

case 0:

{

for (int i = 0; i < n; i++)

cd.push\_back(42);

break;

}

case 1:

{

for (int i = 0; i < n; i++)

cd.push\_front(42);

break;

}

default:

{

cout << "Error: type is not correct. It should be 0 for push\_back or 1 for push\_front.";

return 1;

}

}

auto end = high\_resolution\_clock::now();

duration<double> elapsed = end - start;

cout << (elapsed.count() / n) << endl;

} catch (const std::exception& e) {

cout << "Error: params are not a number: " << e.what() << endl;

return 1;

}

}

Название файла: execution-time-graphic.py

import subprocess

import matplotlib.pyplot as plt

def run\_test(push\_type: int, n: int) -> float:

"""

Запускает ./exectest push\_type n и возвращает время работы в секундах.

Ожидается, что программа печатает только число (время).

"""

result = subprocess.run(

["./exectest", str(push\_type), str(n)],

capture\_output=True,

text=True,

check=True

)

return float(result.stdout.strip())

def main():

sizes = [10\_000, 50\_000, 100\_000, 500\_000, 1\_000\_000, 10\_000\_000]

times\_front = []

times\_back = []

for n in sizes:

t\_front = run\_test(1, n) \* 1e9 # сек -> наносекунды

t\_back = run\_test(0, n) \* 1e9 # сек -> наносекунды

times\_front.append(t\_front)

times\_back.append(t\_back)

print(f"n={n}: push\_front={t\_front:.0f} ns, push\_back={t\_back:.0f} ns")

# Строим график

plt.plot(sizes, times\_front, marker="o", label="push\_front")

plt.plot(sizes, times\_back, marker="o", label="push\_back")

plt.xlabel("Количество элементов")

plt.ylabel("Время (наносекунды)")

plt.title("Сравнение push\_front и push\_back")

plt.legend()

plt.grid(True)

# Сохраняем график в файл

plt.savefig("results.png", dpi=300)

print("График сохранён в results.png")

if \_\_name\_\_ == "\_\_main\_\_":

main()

Название файла: makefile

GXX=g++

exectest=exectest

tests=run

libs=-lgtest

sources=./circular-buffer.cpp \

./circular-buffer.hpp \

.PHONY: clean exec-test

all:

$(GXX) $(libs) $(sources) ./tests.cpp -o $(tests)

exec-test:

$(GXX) $(sources) ./execution-time-test.cpp -o $(exectest)

clean:

rm -rf $(exectest) $(tests)