

**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: Кольцевой буфер**

Студент гр. 4342

Кринкин Н.К.

Преподаватель

Иванов Д.В.

Санкт-Петербург

2025

### **Цель работы.**

Реализовать кольцевой буфер (а именно кольцевую деку) для целых чисел на основе статического массива длины  $N$ .

### **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 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 амортизировано):

`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.

В классе требуется реализовать следующие методы:

- `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.

### **Основные теоретические положения.**

В рамках данной работы основной задачей была реализация структуры данных “Кольцевой буфер”. Кольцевой буфер, она же кольцевая дека, представляет из себя реализацию двусвязной очереди (деки) на основе массива фиксированного размера с “соединенными” концами, то есть за последним элементом следует первый, а перед первым – последний.

Дека, как абстрактный тип данных, подразумевает наличие следующих методов:

- вставка в начало
- вставка в конец
- удаление из начала
- удаление из конца

Согласно задания, кольцевую деку требуется реализовать на основе массива. По этой причине алгоритмическая сложность каждого из перечисленных выше методов будет составлять  $O(1)$ , то есть выполняться за константное время, поскольку доступ к элементу массива осуществляется по указателю и не требует перебора элементов для доступа к нему по индексу.

Массив имеет фиксированный размер, который определяется в момент инициализации и зависит от контекста решаемой задачи, никакой дополнительной памяти в процессе работы не выделяется.

## Выполнение работы.

Кольцевая дека была реализована в виде класса на языке Python. Для начала определим поля класса, для этого обратимся к методу `__init__`. Инициализация экземпляра класса требует следующих параметров:

- `n: int` – количество элементов в деке (массиве)
- `push_force: bool` – возможность перезаписи данных в деку

В процессе инициализации в экземпляре определяются следующие поля:

- `self.deq_size = n` – размер деки
- `self.push_force = push_force`
- `self.vals: list[int | None]` – непосредственно массив, куда будут записаны данные
- `self.push_front_idx: int | None` – индекс начала деки
- `self.push_back_idx: int | None` – индекс конца деки

Изначально атрибуты `push_front_idx` и `push_back_idx` имеют значения `None`, конкретные цифровые индексы приобретаются позднее – после первой вставки в деку. Подробнее об этом будет сказано позднее.

Немного отойдем от программной реализации и опишем алгоритм вставки элемента в деку. Рассмотрим основной случай добавления элемента. Большой разницы между добавлением в начало и в конце в контексте логики выполнения операций нет, поэтому будем рассматривать случай с добавлением в начало деки.

Добавление элемента подразумевает следующие шаги: 1) смещение “указателя” (индекса) на нужную позицию; 2) добавление элемента (если `push_force` выставлен `True`); 3) Проверка “коллапса” коллизий.

На рисунке 1 представлен “стандартный” случай добавления элемента, когда не произошло “коллапса коллизий”.

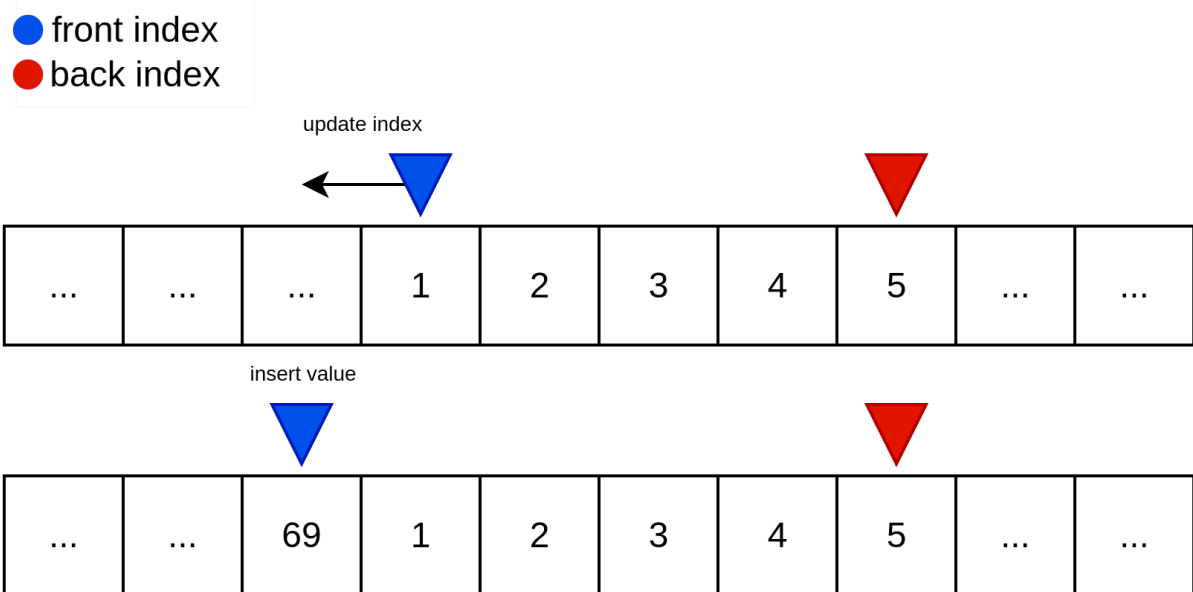


Рис.1 – стандартный случай

Прежде чем перейти к “нестандартному” случаю, требуется сказать немного относительно метода обновления индекса. Метод обновления индекса (`update_front_index` и `update_back_index` соответственно) принимает на вход значение *step* равное числу сдвига индекса. Обновление индекса осуществляется при добавлении элемента в начале или в конец (поэтому стандартные значения параметра *step* – +1 и -1 соответственно). После изменения значения индекса осуществляется проверка выхода за границу слева и справа. Если происходит выход за границу “слева”, значения индекса меняется на значение индекса последнего элемента. В случае выхода за границу “справа”, значение индекса меняется на 0.

Перейдем к “нестандартному” случаю и рассмотрим ситуация “коллапса” коллизий. Под “коллапсом” коллизий подразумевается ситуация, когда в результате push-а (вставки) “указатели” (индексы) указывают на один и тот же элемент, то есть когда требуется осуществить запись элемента в ячейку, которая одновременно является и началом и концом деки. Проверка “коллапса” коллизий осуществляется специальным методом `check_collision`, принимающим на вход строку `atr` – значение атрибута, который вызвал проверку коллизий (“front” или “back”). Если индексы деки совпадают, необходимый индекс

смещается в зависимости от значения параметра `atr`. Подобный “нестандартный” случай показан на рисунке 2.

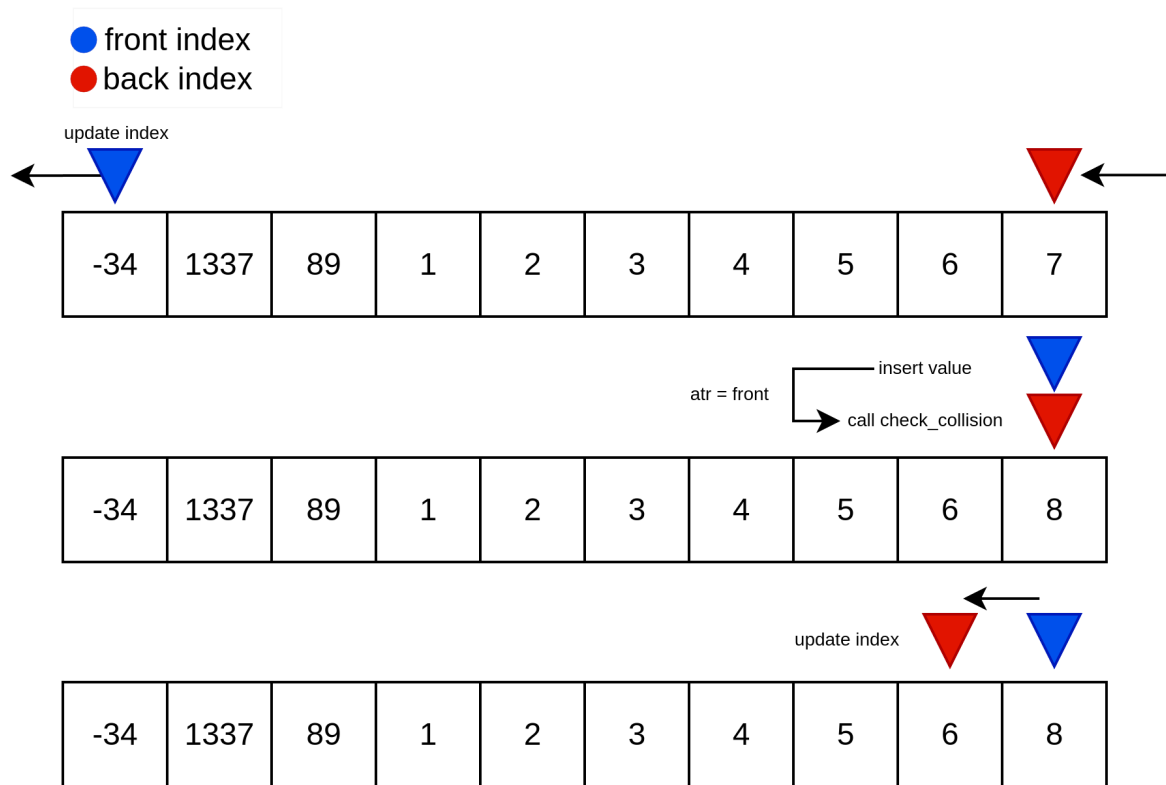


Рис. 2 – “Нестандартный” случай

Таким образом, наличие и реализация методов `push-a` и проверки коллизий вынуждают выставлять значения атрибутов `push_front_idx` и `push_back_idx` равными `None`, дабы облегчить логику программы. Для выставления корректных индексов в процессе использования методы вставки осуществляют проверку заполненности массива, если массив полон – элемент помещается в ячейку с индексом 0, а атрибутам `push_front_idx` и `push_back_idx` присваиваются значения 0.

Удаление (то бишь методы `pop_front` и `pop_back`) работают в обратной относительно методов вставки последовательности, а именно: 1) удаление элемента по нужному индексу; 2) смещение индекса (вызов `update_front_index` или `update_back_index` со значением 1 или -1 соответственно). Методы удаления также осуществляют проверку заполненности массива. Если массив пуст –

индексы устанавливаются в изначальное состояние (то есть приобретают значение None).

Среди не упомянутых и не описанных методов внимания заслуживает только `resize` метод, принимающий на вход значение `new_cap` – новый размер деки. Метод выполняется по следующему алгоритму:

1. инициализация нового массива длины `new_cap`
2. циклический вызов метода `pop_front` до опустошения деки или до заполнения нового массива, что обеспечивает запись значений деки от начала согласно заданию
3. смена полей `self.vals`, `self.deq_size`, а также “обнуление” фронтового и тылового индексов (присваивается значение None)

Разработанный программный код см. в приложении А.

### **Исследование.**

Согласно задания, методы `push_front` и `push_back` должны выполняться с алгоритмической сложностью  $O(1)$ . Чтобы в этом убедиться, были проведены тесты быстродействия методов. Диапазон значений: 10, 100, 1000, от 10 тыс. до 100 тыс., от 100 тыс. до 1.4 миллионов. Полученные результаты представлены на рисунке 3.

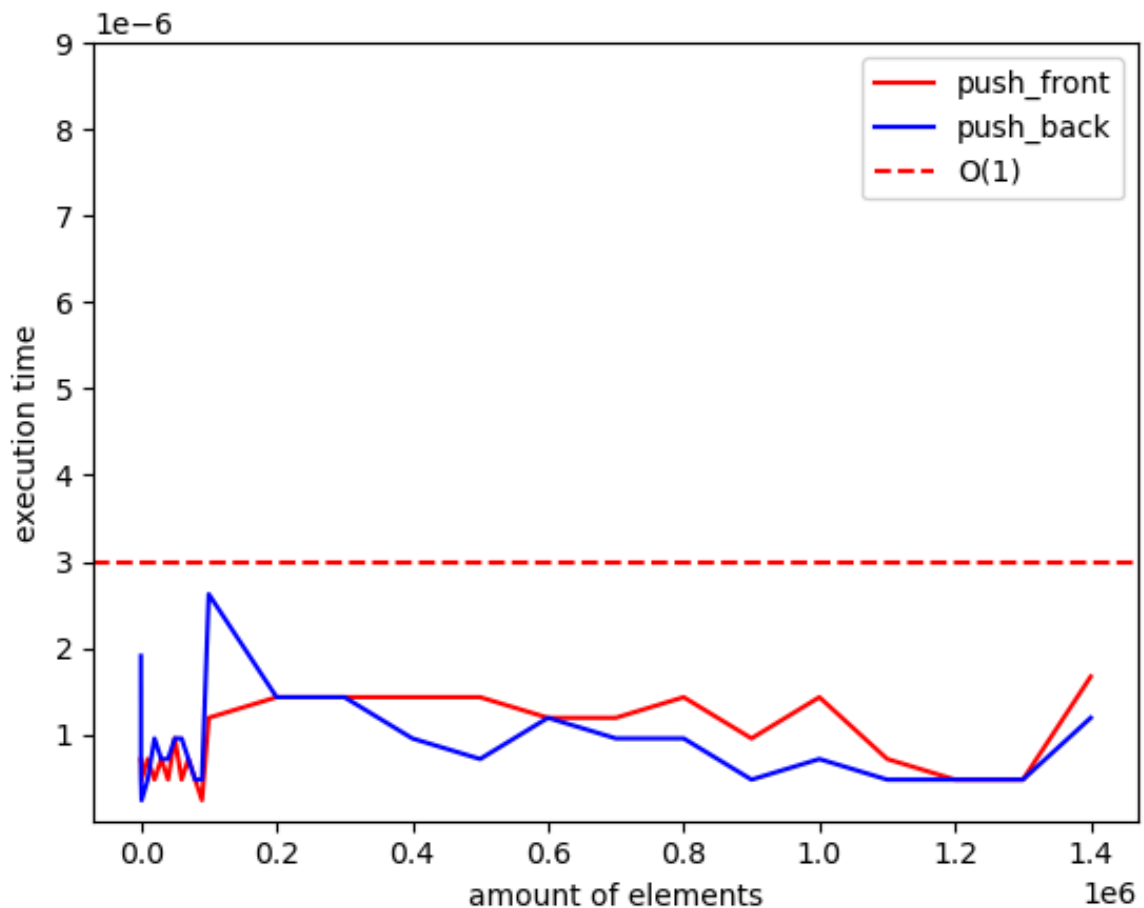


Рис. 3 – результаты

Как можно понять по графику, скорость выполнения операции вставки не превышает 3 микросекунд, а также не увеличивается с ростом данных. На основании полученных результатов можно сделать вывод, что реализованные методы соответствуют требованию задания и исполняются со сложностью  $O(1)$ .

Код исследования см. Приложение А: файл reSearch.ipynb

### Тестирование.

Реализованный класс был спроектирован весьма стрессоустойчиво. Согласно задания, при невозможности выполнения операции требуется вызвать исключение, что и было сделано.

Результаты тестирования представлены в таблице 1.

Код тестов см. Приложение А: файл tests.py



Таблица 1 – Результаты тестирования

№ п/п	Тест	Результат исполнения
1.	Проверка инициализации	Тест пройден
2.	Инит с размером -3	AssertionError: Invalid deque size
3.	Инит с не численным значением размера	AssertionError: Deque only supports integers
4.	Инит с не булевым значением <i>push_force</i>	AssertionError: push_force should be bool!
5.	Проверка метода <i>push_front</i> в стандартных условиях	Тест пройден
6.	Проверка метода <i>push_back</i> в стандартных условиях	Тест пройден
7.	Попытка пуша не числового значения	AssertionError: Deque only supports integers
8.	Попытка пуша при переполнении массива и значением <i>push_force</i> – False	IndexError: Unable to push: place is busy!
9.	Попытка удаления элемента из пустой деки	IndexError: Unable to front -- deque is empty!
10.	Попытка <i>resize</i> с не численным значением <i>new_cap</i>	AssertionError: New_cap should be integer!
11.	Попытка <i>resize</i> с некорректным значением размера	AssertionError: Invalid deque size

### Выводы.

В результате работы была реализована кольцевая дека на основе массива с использованием языка программирования Python. Были реализованы все требуемые методы, проведено исследование, а также осуществлено тестирование программы.

Реализованные методы *push\_front* и *push\_back* соответствуют требованиям быстродействия, что подтверждается проведенным исследованием.

## ПРИЛОЖЕНИЕ А

### ИСХОДНЫЙ КОД ПРОГРАММЫ

Файл lab.py:

```
class CircularDeque:
    def __init__(self, n: int, push_force: bool):
        assert type(n) is int, "Deque only supports integers"
        assert type(push_force) is bool, "push_force should be
bool!"

        assert n > 0, "Invalid deque size"
        self.deq_size = n
        self.push_force = push_force
        self.vals: list[int | None] = [None for _ in range(n)]
        # front and back 'pointers'
        self.push_front_idx: int | None = None
        self.push_back_idx: int | None = None

    def update_front_index(self, step=-1):
        self.push_front_idx += step
        if self.push_front_idx >= self.deq_size:
            self.push_front_idx = 0
        elif self.push_front_idx < 0:
            self.push_front_idx = self.deq_size - 1

    def update_back_index(self, step=1):
        self.push_back_idx += step
        if self.push_back_idx >= self.deq_size:
            self.push_back_idx = 0
        elif self.push_back_idx < 0:
            self.push_back_idx = self.deq_size - 1

    def check_collision(self, atr: str):
        if atr != "front" and atr != "back":
            raise KeyError("Only front and back are available")
        if self.push_back_idx == self.push_front_idx:
            if atr == "front":
                self.update_back_index(-1)
            elif atr == "back":
                self.update_front_index(1)

    def first_push(self, x: int):
        self.vals[0] = x
        self.push_front_idx = 0
        self.push_back_idx = 0

    def push_front(self, x: int):
        assert type(x) is int, "Deque only supports integers"
        if self.push_front_idx is None:
            self.first_push(x)
        else:
            self.update_front_index()
            if self.vals[self.push_front_idx] is None:
                self.vals[self.push_front_idx] = x
            elif self.push_force:
                self.vals[self.push_front_idx] = x
            else:
                raise IndexError("Unable to push: place is busy!")
```

```

        self.check_collision("front")

def push_back(self, x: int):
    assert type(x) is int, "Deque only supports integers"
    if self.push_back_idx is None:
        self.first_push(x)
    else:
        self.update_back_index()
        if self.vals[self.push_back_idx] is None:
            self.vals[self.push_back_idx] = x
        elif self.push_force:
            self.vals[self.push_back_idx] = x
        else:
            raise IndexError("Unable to push: place is busy!")
        self.check_collision("back")

def pop_front(self) -> int:
    if self.empty():
        raise IndexError("Unable to pop -- deque is empty!")
    val = self.vals[self.push_front_idx]
    if val is None:
        raise KeyError
    self.vals[self.push_front_idx] = None
    self.update_front_index(1)
    if self.empty():
        self.push_front_idx: int | None = None
        self.push_back_idx: int | None = None
    return val

def pop_back(self) -> int:
    if self.empty():
        raise IndexError("Unable to pop -- deque is empty!")
    val = self.vals[self.push_back_idx]
    if val is None:
        raise KeyError
    self.vals[self.push_back_idx] = None
    self.update_back_index(-1)
    if self.empty():
        self.push_front_idx: int | None = None
        self.push_back_idx: int | None = None
    return val

def front(self) -> int:
    if self.empty():
        raise IndexError("Unable to front -- deque is empty!")
    val = self.vals[self.push_front_idx]
    if val is None:
        raise KeyError
    return val

def back(self) -> int:
    if self.empty():
        raise IndexError("Unable to back -- deque is empty!")
    val = self.vals[self.push_back_idx]
    if val is None:
        raise KeyError
    return val

```

```

def size(self) -> int:
    return self.deq_size - self.vals.count(None)

def empty(self) -> bool:
    return self.vals.count(None) == self.deq_size

def full(self):
    return self.vals.count(None) == 0

def resize(self, new_cap: int):
    assert new_cap > 0, "Invalid deque size"
    assert new_cap is int, "New_cap should be integer!"
    if new_cap < 1:
        raise AttributeError("New capacity should be greater
that zero!")
    new_vals: list[int | None] = [None for _ in
range(new_cap)]
    new_idx = 0
    while not self.empty() and new_vals.count(None) != 0:
        new_vals[new_idx] = self.pop_front()
        new_idx += 1
    self.vals = new_vals
    self.deq_size = new_cap
    self.push_front_idx = 0
    self.push_back_idx = new_idx - 1

```

#### Файл tests.py:

```

import pytest
from lab import CircularDeque

def test_init():
    dq = CircularDeque(3, True)

def test_init_er_1():
    dq = CircularDeque(-3, True)

def test_init_er_2():
    dq = CircularDeque('123', True)

def test_init_er_3():
    dq = CircularDeque(12, 12)

def test_push_front():
    dq = CircularDeque(3, True)
    dq.push_front(14)

def test_push_back():
    dq = CircularDeque(3, True)
    dq.push_back(14)

def test_push_front_er_1():
    dq = CircularDeque(3, True)
    dq.push_front('sas')

def test_push_front_er_2():
    dq = CircularDeque(3, False)

```

```
    dq.push_front(12)
    dq.push_front(13)
    dq.push_front(15)
    dq.push_front(14)

def test_pop_front():
    dq = CircularDeque(3, False)
    dq.front()

def test_resize():
    dq = CircularDeque(3, False)
    dq.push_front(12)
    dq.push_front(13)
    dq.push_front(15)
    dq.resize(5)

def test_resize_er_1():
    dq = CircularDeque(3, False)
    dq.push_front(12)
    dq.push_front(13)
    dq.push_front(15)
    dq.resize(-5)

Файл reSearch.ipynb:
????
```