

САНКТ-ПЕТЕРБУРГСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ  
УНИВЕРСИТЕТ  
ИНФОРМАЦИОННЫХ ТЕХНОЛОГИЙ, МЕХАНИКИ И ОПТИКИ  
ФАКУЛЬТЕТ ИНФОКОММУНИКАЦИОННЫХ ТЕХНОЛОГИЙ

Отчет по лабораторной работе №4  
по курсу «Алгоритмы и структуры данных»  
Тема: Стек, очередь, связанный список.  
Вариант 3

Выполнила:  
Блинова П. В.  
К3139 (номер группы)

Проверил:  
Афанасьев А. В.

Санкт-Петербург  
2024 г.

## Содержание отчета

Содержание отчета	2
Задачи по варианту	3
Задача №1. Стек	3
Задача №4. Скобочная последовательность. Версия 2	4
Задача №6. Очередь с минимумом	6
Задача №9. Поликлиника	8
Дополнительные задачи	11
Задача №8. Постфиксная запись	11
Задача №13. Реализация стека, очереди и связанных списков	12
Вывод	15

## Задачи по варианту

### Задача №1. Стек

Текст задачи.

Реализуйте работу стека. Для каждой операции изъятия элемента выведите ее результат. На вход программе подаются строки, содержащие команды. Каждая строка содержит одну команду. Команда — это либо “+ N”, либо “-”. Команда “+ N” означает добавление в стек числа N, по модулю не превышающего 109. Команда “-” означает изъятие элемента из стека. Гарантируется, что не происходит извлечения из пустого стека. Гарантируется, что размер стека в процессе выполнения команд не превысит 106 элементов.

- Формат входного файла (input.txt). В первой строке входного файла содержится M ( $1 \leq M \leq 106$ ) – число команд. Каждая последующая строка исходного файла содержит ровно одну команду.
- Формат выходного файла (output.txt). Выведите числа, которые удаляются из стека с помощью команды “-”, по одному в каждой строке. Числа нужно выводить в том порядке, в котором они были извлечены из стека. Гарантируется, что изъятий из пустого стека не производится.
- Ограничение по времени. 2 сек.
- Ограничение по памяти. 256 мб.

Листинг кода.

```
def stack_commands(commands):  
    stack = list()  
    deleted_el = list()  
    for command in commands:  
        if command[0] == '+':  
            stack.append(int(command.split('+')[1]))  
        elif command == '-':  
            element = stack.pop()  
            deleted_el.append(element)  
    return deleted_el
```

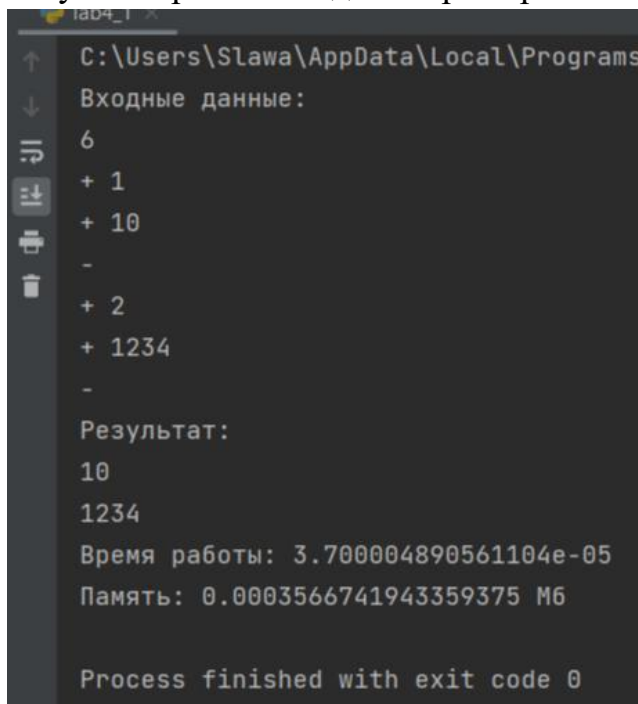
Текстовое объяснение решения.

Функция `stack_commands` моделирует работу стека, обрабатывая последовательность команд. Она принимает на вход список команд (`commands`), где каждая команда представляет собой строку. Команды могут быть двух типов:

- Добавление элемента: Команда начинается с символа '+', за которым следует целое число, например, "+12". В этом случае целое число добавляется в стек.
- Удаление элемента: Команда представляет собой

одионый символ '-'. В этом случае, если стек не пуст, верхний элемент извлекается из стека и добавляется в список `deleted_el`. Функция использует два списка: • `stack`: список, реализующий стек (LIFO - Last-In, First-Out). • `deleted_el`: список, в который добавляются элементы, извлеченные из стека командой '- '. Функция итерирует по списку команд. Если команда начинается с '+', она извлекает число из команды и добавляет его в стек. Если команда равна '-', она извлекает элемент из стека (если стек не пуст) и добавляет его в `deleted_el`. Наконец, функция возвращает список `deleted_el`, содержащий все удаленные элементы.

Результат работы кода на примерах из текста задачи:



```
C:\Users\Slawa\AppData\Local\Programs
Входные данные:
6
+ 1
+ 10
-
+ 2
+ 1234
-
Результат:
10
1234
Время работы: 3.700004890561104e-05
Память: 0.0003566741943359375 МБ
Process finished with exit code 0
```

Вывод по задаче:

В ходе работы над задачей мной был изучен стек.

## Задача №4. Скобочная последовательность. Версия 2

Текст задачи.

Определение правильной скобочной последовательности такое же, как и в задаче 3, но теперь у нас больше набор скобок: `[]{}()`. Нужно написать функцию для проверки наличия ошибок при использовании разных типов

скобок в текстовом редакторе типа LaTeX. Для удобства, текстовый редактор должен не только информировать о наличии ошибки в использовании скобок, но также указать точное место в коде (тексте) с ошибочной скобочкой. В первую очередь объявляется ошибка при наличии первой несовпадающей закрывающей скобки, перед которой отсутствует открывающая скобка, или которая не соответствует открывающей, например, `()[]` - здесь ошибка укажет на `]`. Во вторую очередь, если описанной выше ошибки не было найдено, нужно указать на первую несовпадающую открывающую скобку, у которой отсутствует закрывающая, например, `(` в `([]`. Если не найдено ни одной из указанных выше ошибок, нужно сообщить, что использование скобок корректно. Помимо скобок, код может содержать большие и маленькие латинские буквы, цифры и знаки препинания. Формально, все скобки в коде (тексте) должны быть разделены на пары совпадающих скобок, так что в каждой паре открывающая скобка идет перед закрывающей скобкой, а для любых двух пар скобок одна из них вложена внутри другой, как в `(foo[bar])` или они разделены, как в `f(a,b)-g[c]`. Скобка `[` соответствует скобке `]`, соответствует и `(` соответствует `)`.

- Ограничение по времени. 2сек.
- Ограничение по памяти. 256 мб.

Листинг кода.

```
def check_brackets(data):
    stack = []
    bracket_pairs = {
        ')': '(',
        ']': '[',
        '}': '{'
    }
    for i, symb in enumerate(data, start=1):
        if symb in "([{":
            stack.append((symb, i))
        elif symb in ")]}":
            if not stack or stack[-1][0] != bracket_pairs[symb]:
                return i
            stack.pop()

    if stack:
        return stack[0][1]

    return "Success"
```

Текстовое объяснение решения.

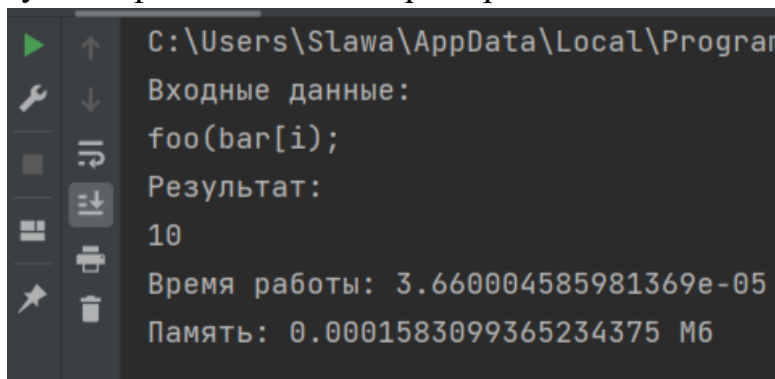
Алгоритм использует стек `stack` для хранения открывающих скобок. Словарь `bracket_pairs` хранит соответствия между закрывающими и открывающими скобками. Алгоритм проходит по входной строке `data` символ за символом.

- Если символ – открывающая скобка: Он добавляется в стек вместе с его индексом в строке.
- Если символ – закрывающая скобка:
  - \* Если стек пуст: Значит, закрывающая скобка не соответствует ни одной открывающей, и алгоритм возвращает индекс этой закрывающей скобки.
  - \* Если верхний элемент стека не соответствует текущей закрывающей скобке: Скобки не сбалансированы, и алгоритм возвращает индекс текущей закрывающей скобки.
  - \* Иначе: Верхний элемент стека (соответствующая открывающая скобка) удаляется.

После обработки всей строки:

- Если стек не пуст: Остались непарные открывающие скобки. Алгоритм возвращает индекс первой из них (верхний элемент стека).
- Если стек пуст: Скобки сбалансированы, и алгоритм возвращает строку "Success".

Результат работы кода на примерах из текста задачи.



The screenshot shows a debugger window with the following text:

```
C:\Users\Slawa\AppData\Local\Program
Входные данные:
foo(bar[i];
Результат:
10
Время работы: 3.660004585981369e-05
Память: 0.0001583099365234375 M6
```

Вывод по задаче:

В ходе работы над задачей мной был изучен способ проверки скобочной последовательности.

## Задача №6. Очередь с минимумом

Текст задачи.

Реализуйте работу очереди. В дополнение к стандартным операциям очереди, необходимо также отвечать на запрос о минимальном элементе из тех, которые сейчас находятся в очереди. Для каждой операции запроса минимального элемента выведите ее результат. На вход программе подаются

строки, содержащие команды. Каждая строка содержит одну команду. Команда – это либо «+ N», либо «-», либо «?». Команда «+ N» означает добавление в очередь числа N, по модулю не превышающего 109. Команда «-» означает изъятие элемента из очереди. Команда «?» означает запрос на поиск минимального элемента в очереди.

- Формат входного файла (input.txt). В первой строке содержится M ( $1 \leq M \leq 106$ ) – число команд. В последующих строках содержатся команды, по одной в каждой строке.

- Формат выходного файла (output.txt). Для каждой операции поиска минимума в очереди выведите её результат. Результаты должны быть выведены в том порядке, в котором эти операции встречаются во входном файле. Гарантируется, что операций извлечения или поиска минимума для пустой очереди не производится.

- Ограничение по времени. 2сек.
- Ограничение по памяти. 256 мб

Листинг кода.

```
def queue_min(commands):
    queue = list()
    result = list()
    for command in commands:
        if command[0] == "+":
            queue.append(int(command.split()[1]))
        elif command[0] == "-":
            queue.pop(0)
        elif command[0] == "?":
            result.append(min(queue))
    return result
```

Текстовое объяснение решения.

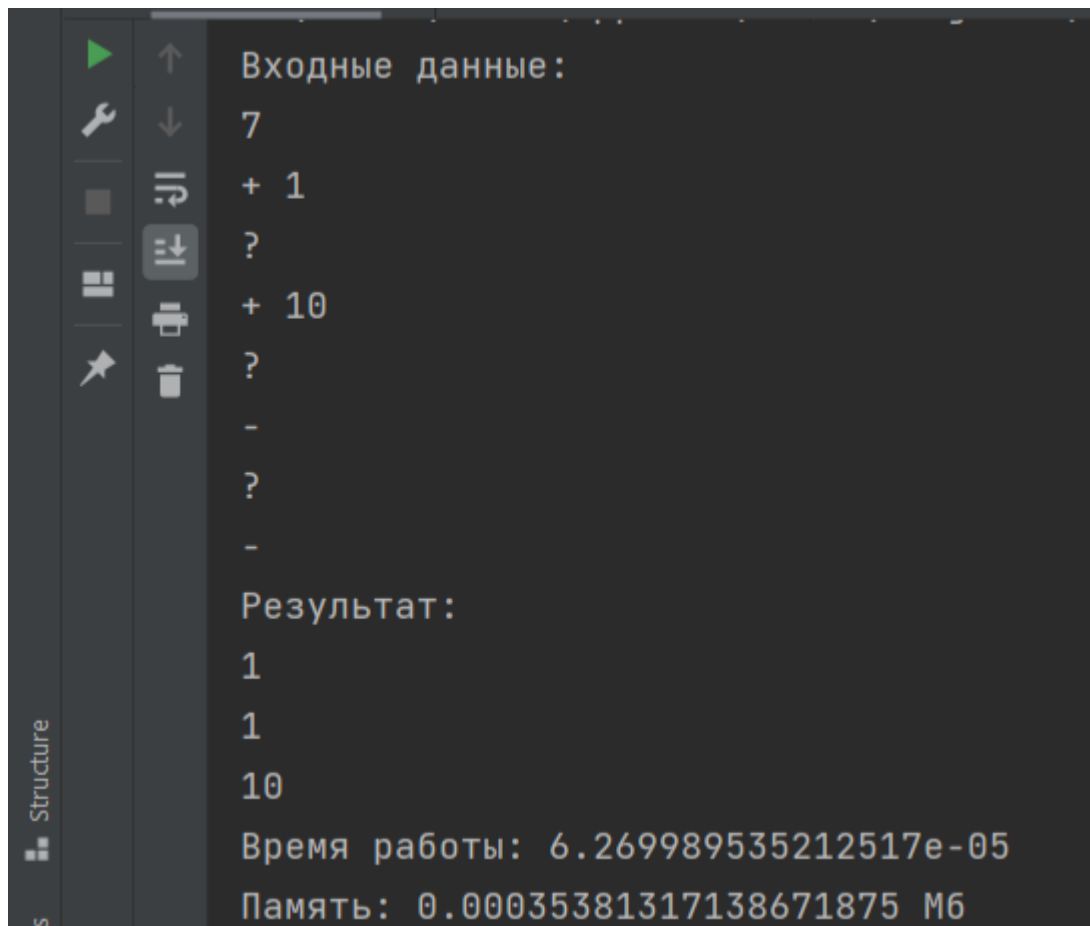
Алгоритм использует список queue для реализации очереди и список result для хранения результатов запросов на поиск минимального элемента. Входные данные представляют собой список команд commands, где каждая команда имеет один из следующих форматов:

- "+ value": Добавить значение value в конец очереди.
- "-": Удалить элемент из начала очереди.
- "?": Найти и вернуть минимальное значение в очереди.

Алгоритм итерирует по списку команд:

- Команда "+ value": Значение value преобразуется в целое число и добавляется в конец списка queue.
- Команда "-": Первый элемент списка queue удаляется с помощью метода pop(0).
- Команда "?": Минимальное значение в списке queue находится с помощью функции min() и добавляется в список result.

Результат работы кода на примерах из текста задачи.



The screenshot shows a Python IDE with a dark theme. On the left, there is a sidebar with icons for running, debugging, and other IDE functions. The main area displays the following text:

```
Входные данные:
7
+ 1
?
+ 10
?
-
?
-
Результат:
1
1
10
Время работы: 6.269989535212517e-05
Память: 0.00035381317138671875 Мб
```

Вывод по задаче:

В ходе работы над задачей мной был получен способ получения очереди минимумом.

## Задача №9. Поликлиника

Текст задачи.

Очередь в поликлинике работает по сложным правилам. Обычные пациенты при посещении должны вставать в конец очереди. Пациенты, которым "только справку забрать" встают ровно в ее середину, причем при нечетной длине очереди они встают сразу за центром. Напишите программу, которая отслеживает порядок пациентов в очереди.

- Ограничение по времени. 2сек.
- Ограничение по памяти. 256 мб

Листинг кода.

```
class Queue:
    def __init__(self):
```



```

        self.queue = []

    def add(self, value):
        self.queue.append(value)

    def add_mid(self, value):
        if len(self.queue) % 2 == 0:
            middle = len(self.queue) // 2
        else:
            middle = len(self.queue) // 2 + 1

        self.queue.insert(middle, value)

    def remove(self):
        del_value = self.queue[0]
        self.queue.pop(0)
        return del_value

def queue_min(commands):
    queue = Queue()
    result = list()
    for command in commands:
        if command[0] == "+":
            queue.add(int(command.split()[1]))
        elif command[0] == "*":
            queue.add_mid(int(command.split()[1]))
        elif command[0] == "-":
            result.append(queue.remove())
    return result

```

Текстовое объяснение решения.

Класс Queue реализует очередь с помощью списка Python. Он предоставляет следующие методы:

- `add(value)`: Добавляет элемент `value` в конец очереди.
- `add_mid(value)`: Добавляет элемент `value` в середину очереди. Если длина очереди чётная, элемент добавляется в позицию `len(queue) // 2`; если нечётная – в позицию `len(queue) // 2 + 1`.
- `remove()`: Удаляет и возвращает первый элемент очереди.

Описание алгоритма обработки команд `queue_min`: Функция `queue_min` принимает список команд `commands` и использует объект класса `Queue` для обработки этих команд. Каждая команда имеет один из следующих форматов:

- `" + value "`: Добавить значение `value` в конец очереди.
- `" * value "`: Добавить значение `value` в середину очереди.
- `" - "`: Удалить и вернуть первый элемент очереди.

Алгоритм итерирует по списку команд и выполняет соответствующие действия с помощью методов класса `Queue`. Результатом работы функции является список элементов, удалённых из очереди командами `" - "`.

Результат работы кода на примерах из текста задачи.

Входные данные:

7

+ 1

+ 2

-

+ 3

+ 4

-

-

Результат:

1

2

3

Время работы: 5.3299940191209316e-05

Память: 0.000335693359375 Мб

Вывод по задаче:

В ходе работы над задачей мной был изучен способ бинарного поиска. Была изучена эффективность этого метода.

## Дополнительные задачи

### Задача №8. Постфиксная запись

Текст задачи.

В постфиксной записи (или обратной польской записи) операция записывается после двух операндов. Например, сумма двух чисел  $A$  и  $B$  записывается как  $A B +$ . Запись  $B C + D *$  обозначает привычное нам  $(B + C) * D$ , а запись  $A B C + D * +$  означает  $A + (B + C) * D$ . Достоинство постфиксной записи в том, что она не требует скобок и дополнительных соглашений о приоритете операторов для своего чтения. Дано выражение в обратной польской записи. Определите его значение.

- Формат входного файла (input.txt). В первой строке входного файла дано число  $N$  ( $1 \leq n \leq 106$ ) – число элементов выражения. Во второй строке содержится выражение в постфиксной записи, состоящее из  $N$  элементов. В выражении могут содержаться неотрицательные однозначные числа и операции  $+$ ,  $-$ ,  $*$ . Каждые два соседних элемента выражения разделены ровно одним пробелом.
- Формат выходного файла (output.txt). Необходимо вывести значение записанного выражения. Гарантируется, что результат выражения, а также результаты всех промежуточных вычислений, по модулю будут меньше, чем  $2^{31}$ .

- Ограничение по времени. 2сек.
- Ограничение по памяти. 256 мб

Листинг кода.

```
def postfix_calculate(expression):
    stack = []
    actions = ['+', '-', '*', '/']
    for symbol in expression.split():
        if symbol not in actions:
            stack.append(int(symbol))
        else:
            b = stack.pop()
            a = stack.pop()
            if symbol == '+':
                stack.append(a + b)
            elif symbol == '-':
                stack.append(a - b)
            elif symbol == '*':
                stack.append(a * b)
            elif symbol == '/':
                stack.append(int(a / b))
    return stack[0]
```

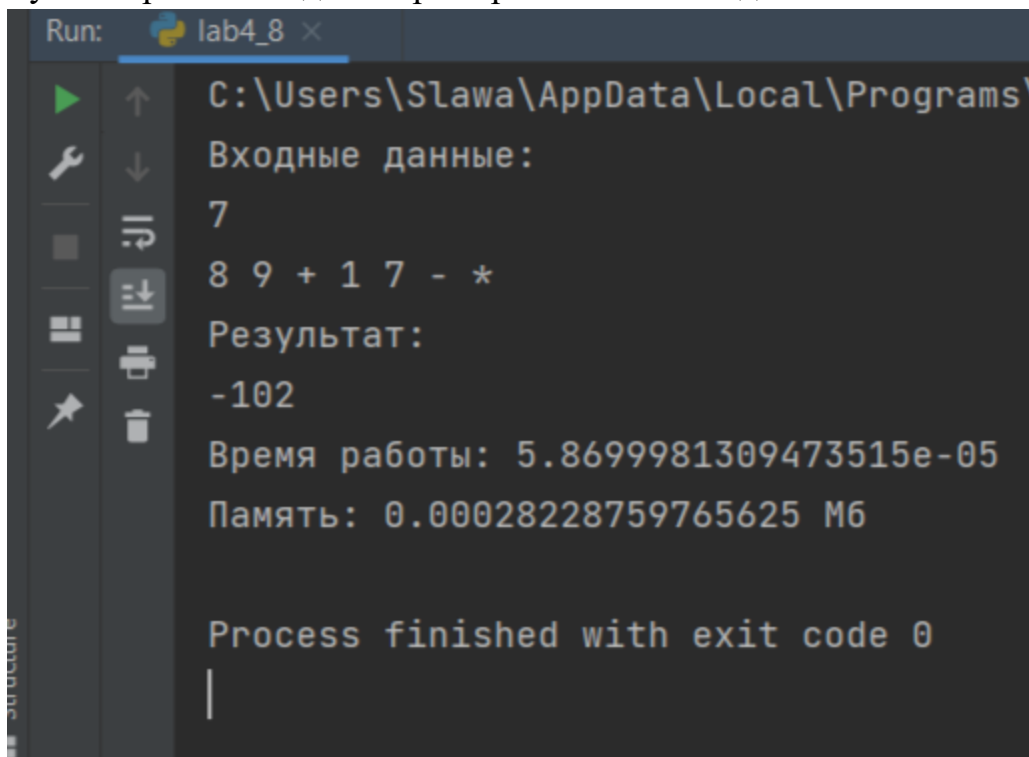
Текстовое объяснение решения.

Алгоритм использует стек `stack` для хранения операндов. Входное выражение `expression` предполагается разделенным пробелами на отдельные токены (операнды и операторы). Алгоритм проходит по токенам выражения:

- Если токен – число: Он преобразуется в целое число и помещается на вершину стека.
- Если токен – оператор: \* Из стека извлекаются два последних операнда (`b` и `a`, где `b` – верхний элемент). \* Выполняется соответствующая арифметическая операция (+, -, \*, /) над `a` и `b`. \* Результат операции помещается на вершину стека.

После обработки всех токенов, единственный элемент, оставшийся в стеке, будет являться результатом вычисления выражения.

Результат работы кода на примерах из текста задачи.



```
Run: lab4_8 x
C:\Users\Slawa\AppData\Local\Programs\
Входные данные:
7
8 9 + 1 7 - *
Результат:
-102
Время работы: 5.8699981309473515e-05
Память: 0.00028228759765625 M6

Process finished with exit code 0
```

Вывод по задаче:

В ходе работы над задачей мной был изучен способ бинарного поиска. Была изучена эффективность этого метода.

### Задача №13. Реализация стека, очереди и связанных списков

Текст задачи.

1. Реализуйте стек на основе связанного списка с функциями isEmpty, push, pop и вывода данных.
2. Реализуйте очередь на основе связанного списка функциями Enqueue, Dequeue с проверкой на переполнение и опустошения очереди.

Листинг кода.

```
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None

class Stack:
    def __init__(self):
        self.top = None

    def isEmpty(self):
        return self.top is None

    def push(self, data):
        new_node = Node(data)
        new_node.next = self.top
        self.top = new_node

    def pop(self):
        if self.isEmpty():
            return None
        popped_node = self.top
        self.top = self.top.next
        return popped_node.data

    def display(self):
        if self.isEmpty():
            return
        current = self.top
        elements = list()
        while current:
            elements.append(current.data)
            current = current.next
        return ", ".join(map(str, elements))

def result_f(result, nums):
    stack = Stack()
    result += f'Стек пуст: {stack.isEmpty()}\n'
    for num in nums:
        stack.push(num)
        result += stack.display() + '\n'
    result += f'Извлечён элемент: {stack.pop()}\n'
    result += stack.display() + '\n'
    result += f'Стек пуст: {stack.isEmpty()}\n'
    return result
```

```
class Node:
    def __init__(self, value):
```

```

        self.value = value
        self.next = None

class Queue:
    def __init__(self, max_size):
        self.first = None
        self.last = None
        self.size = 0
        self.max_size = max_size

    def isEmpty(self):
        return self.size == 0

    def isFull(self):
        return self.size == self.max_size

    def enqueue(self, value):
        if self.isFull():
            return 'Очередь переполнена'
        new_node = Node(value)
        if self.last is None:
            self.first = self.last = new_node
        else:
            self.last.next = new_node
            self.last = new_node
        self.size += 1

    def dequeue(self):
        if self.isEmpty():
            return 'Очередь пуста'
        dequeued_value = self.first.value
        self.first = self.first.next
        if self.first is None:
            self.last = None
        self.size -= 1
        return dequeued_value

    def peek(self):
        if self.isEmpty():
            return None
        return self.first.value

    def queue_size(self):
        return self.size

def result_f(result, max_size, nums):
    queue = Queue(max_size)
    result += f'Очередь пуста: {queue.isEmpty()}\n'
    result += f'Добавление элементов в очередь...\n'
    for num in nums:
        queue.enqueue(num)
        result += f'Было добавлено: {num}\n'
        result += f'Последний элемент очереди: {queue.peek()}\n\n'

    result += f'Добавление элементов закончено\n'
    result += f'Размер очереди: {queue.queue_size()}\n'

```

```
queue.dequeue()
result += f"Был извлечён последний элемент\n"
result += f"Размер очереди: {queue.queue_size()}\n"
result += f"Последний элемент очереди: {queue.peek()}\n\n"
return result
```

Текстовое объяснение решения.

Класс Stack реализует стек на основе связанного списка. Он содержит указатель на вершину стека (top). Методы класса: • isEmpty(): Проверяет, пуст ли стек. • push(data): Добавляет элемент data на вершину стека. • pop(): Извлекает и возвращает элемент с вершины стека. Возвращает None, если стек пуст. • display(): Возвращает строковое представление элементов стека, разделённых запятыми.

Класс Queue реализует очередь с ограниченным размером (max\_size) на основе связанного списка. Он содержит указатели на первый (first) и последний (last) элементы очереди, а также переменные для отслеживания размера очереди (size) и максимального размера (max\_size). Методы класса: • isEmpty(): Проверяет, пуста ли очередь. • isFull(): Проверяет, заполнена ли очередь. • enqueue(value): Добавляет значение value в конец очереди. Возвращает сообщение об ошибке, если очередь переполнена. • dequeue(): Извлекает и возвращает значение из начала очереди. Возвращает сообщение об ошибке, если очередь пуста. • peek(): Возвращает значение первого элемента очереди без его удаления. Возвращает None, если очередь пуста. • queue\_size(): Возвращает текущий размер очереди.

Вывод по задаче:

В ходе работы над задачей мной был изучен способ подсчёта инверсий. Была изучена эффективность этого метода.

## Вывод

В ходе лабораторной работы был изучен метод сортировки слиянием и метод декомпозиции («Разделяй и властвуй»). Был проведён анализ работы алгоритмов на максимальных и минимальных значениях. Был изучен алгоритм бинарного поиска.

