

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

Выполнил:  
Нгуен Хыу Жанг  
K3140

Проверила:  
Афанасьев А.В

Санкт-Петербург  
2024 г

# Содержание

Содержание .....	2
Задание 1 : Стек .....	3
Задание 2 : Очередь .....	7
Задание 3 : Скобочная последовательность. Версия 1 .....	11
Задание 4 : Скобочная последовательность. Версия 2 .....	17
Задание 5 : Стек с максимумом .....	21
Задание 6 : Очередь с минимумом .....	27
Задание 7 : Максимум в движущейся последовательности .....	33
Задание 8 : Постфиксная запись .....	37
Задание 9 : Поликлиника .....	41

## Задачи по варианту

### Задание 1 : Стек

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  $10^9$ . Команда “-” означает изъятие элемента из стека. Гарантируется, что не происходит извлечения из пустого стека. Гарантируется, что размер стека в процессе выполнения команд не превысит  $10^6$  элементов.

- **Формат входного файла (input.txt).** В первой строке входного файла содержится  $M$  ( $1 \leq M \leq 10^6$ ) — 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.
- Пример:

input.txt	output.txt
6	10
+ 1	1234
+ 10	
-	
+ 2	
+ 1234	
-	

```
import os

def process_stack_commands(input_file, output_file):
    stack = []
    output = []

    with open(input_file, 'r', encoding='utf-8') as infile:
        M = int(infile.readline().strip())

        for _ in range(M):
            command = infile.readline().strip()
            if command.startswith('+'):
                _, number = command.split()
                stack.append(int(number))
            elif command == '-':
                output.append(stack.pop())

    with open(output_file, 'w', encoding='utf-8') as outfile:
        for value in output:
            outfile.write(f"{value}\n")
```

```
def main():
    txtf_dir = os.path.join(os.path.dirname(os.path.abspath(__file__)), 'txtf')
    input_file_path = os.path.join( '..', 'txtf', 'input.txt')
    output_file_path = os.path.join( '..', 'txtf', 'output.txt')

    process_stack_commands(input_file_path, output_file_path)

if __name__ == "__main__":
    main()
```

input.txt :

1	6
2	+ 1
3	+ 10
4	-
5	+ 2
6	+ 1234
7	-

output.txt:

1	10
2	1234
3	

## 1. Определение функции `process_stack_commands`

Функция принимает два аргумента:

- `input_file`: путь к файлу с входными данными.
- `output_file`: путь к файлу, куда будет записан результат.

Логика функции:

### 1. Инициализация:

- `stack = []`: пустой список, который используется как стек.
- `output = []`: список для хранения удалённых элементов стека.

### 2. Чтение данных:

- Файл с входными данными открывается для чтения.
- Первая строка файла содержит число команд `M`.

### 3. Обработка команд:

- Для каждой команды:
  - Если команда начинается с `+` (например, `+ N`), то число `N` извлекается из команды и добавляется в стек.
  - Если команда равна `-`, то выполняется операция удаления элемента из стека с помощью `stack.pop()`, а удалённое значение добавляется в список `output`.

### 4. Запись результата:

- Файл с выходными данными открывается для записи.
- Каждое значение из `output` записывается в файл в новой строке.

## 2. Функция `main`

Основная функция программы:

1. Определяет пути к входному (input.txt) и выходному (output.txt) файлам.
2. Вызывает функцию process\_stack\_commands с этими путями.

### Unittest для задание 1:

```
import unittest
import sys
import os

sys.path.insert(0, os.path.abspath(os.path.join(os.path.dirname(__file__), '..', 'src')))
from bl import process_stack_commands

class TestProcessStackCommands(unittest.TestCase):

    def setUp(self):
        # given
        self.test_dir = 'test_txtf'
        os.makedirs(self.test_dir, exist_ok=True)

        # when
        self.input_file = os.path.join(self.test_dir, 'input.txt')
        self.output_file = os.path.join(self.test_dir, 'output.txt')

        # then
        with open(self.input_file, 'w', encoding='utf-8') as f:
            f.write("6\n+ 1\n+ 10\n-\n+ 2\n+ 1234\n-\n")

    def tearDown(self):
        # given
        if os.path.exists(self.test_dir):
            for file in os.listdir(self.test_dir):
                os.remove(os.path.join(self.test_dir, file))
            os.rmdir(self.test_dir)

    def test_process_stack_commands(self):
        # given
        process_stack_commands(self.input_file, self.output_file)

        # when
        with open(self.output_file, 'r', encoding='utf-8') as f:
            output = f.read().strip().split('\n')

        # then
        expected_output = ['10', '1234']
        self.assertEqual(output, expected_output)

if __name__ == '__main__':
    unittest.main()
```

### **\* Результат :**

✓ Test Results 16 ms	✓ Tests passed: 1 of 1 test - 16 ms
✓ test 1 16 ms	collecting ... collected 1 item
✓ TestProcessStack 16 ms	test 1.py::TestProcessStackCommands::test_process_stack_commands PASSED [100%]
✓ test_process_st 16 ms	===== 1 passed, 1 warning in 0.06s =====
	Process finished with exit code 0

## Импортируемые модули

1. **unittest** — стандартная библиотека Python для написания модульных тестов.
2. **sys** и **os** — используются для работы с путями файлов и каталогов.
3. **b1** — импортируется тестируемая функция `process_stack_commands` из модуля `b1`.

## Описание классов и методов

### 1. Класс `TestProcessStackCommands`

Этот класс наследует `unittest.TestCase` и содержит тесты для функции `process_stack_commands`.

### 2. Метод `setUp`

Этот метод выполняется перед каждым тестом.

- **Что делает:**

1. Создает временную директорию `test_txtf` для входных и выходных файлов.
2. Создает входной файл `input.txt` с тестовыми данными:

```
6
+ 1
+ 10
-
+ 2
+ 1234
-
```

  - Эти команды означают:
    - Добавить в стек 1.
    - Добавить в стек 10.
    - Удалить элемент из стека (ожидается 10).
    - Добавить в стек 2.
    - Добавить в стек 1234.
    - Удалить элемент из стека (ожидается 1234).

### 3. Метод `tearDown`

Этот метод выполняется после каждого теста.

- **Что делает:**

1. Удаляет все файлы и директорию `test_txtf`, созданные во время теста.
2. Это помогает обеспечить "чистую" среду для следующего теста.

### 4. Метод `test_process_stack_commands`

Содержит сам тест:

- **Шаги:**

1. Вызывает функцию `process_stack_commands`, передавая входной и выходной файлы.
2. Читает содержимое выходного файла `output.txt`.
3. Сравнивает содержимое выходного файла (`['10', '1234']`) с ожидаемым результатом.

## 5. Запуск тестов

- `unittest.main()` — запускает тесты, если скрипт выполняется напрямую.

## Задание 2 : Очередь

Реализуйте работу очереди. 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 очередь числа  $N$ , по модулю не превышающего  $10^9$ . Команда «-» означает изъятие элемента из очереди. Гарантируется, что размер очереди в процессе выполнения команд не превысит  $10^6$  элементов.

- **Формат входного файла (input.txt).** В первой строке содержится  $M$  ( $1 \leq M \leq 10^6$ ) — число команд. В последующих строках содержатся команды, по одной в каждой строке.
- **Формат выходного файла (output.txt).** Выведите числа, которые удаляются из очереди с помощью команды «-», по одному в каждой строке. Числа нужно выводить в том порядке, в котором они были извлечены из очереди. Гарантируется, что извлечения из пустой очереди не производится.
- Ограничение по времени. 2 сек.
- Ограничение по памяти. 256 мб.
- Пример:

input.txt	output.txt
4	1
+ 1	10
+ 10	
-	
-	

```
import os
from collections import deque

def process_queue_commands(input_file, output_file):
    queue = deque()
    output = []

    with open(input_file, 'r', encoding='utf-8') as infile:
        M = int(infile.readline().strip())

        for _ in range(M):
            command = infile.readline().strip()
            if command.startswith('+'):
                _, number = command.split()
                queue.append(int(number))
            elif command == '-':
                output.append(queue.popleft())

    with open(output_file, 'w', encoding='utf-8') as outfile:
        for value in output:
            outfile.write(f"{value}\n")

def main():
    txtf_dir = os.path.join(os.path.dirname(__file__), '..', 'txtf')
    os.makedirs(txtf_dir, exist_ok=True)

    input_file_path = os.path.join(txtf_dir, 'input.txt')
    output_file_path = os.path.join(txtf_dir, 'output.txt')
```

```

process_queue_commands(input_file_path, output_file_path)

if __name__ == "__main__":
    main()

```

\* Результат :

input.txt :

1	4
2	+ 1
3	+ 10
4	-
5	-

output.txt :

1
10

## 1. Импорт библиотек:

```

import os
from collections import deque

```

- os используется для работы с путями файлов.
- deque из модуля collections предоставляет двустороннюю очередь (дека), которая оптимизирована для операций добавления/удаления с обоих концов.

## 2. Функция process\_queue\_commands:

```

def process_queue_commands(input_file, output_file):
    queue = deque()
    output = []

```

```

    with open(input_file, 'r', encoding='utf-8') as infile:
        M = int(infile.readline().strip())

```

```

    for _ in range(M):
        command = infile.readline().strip()
        if command.startswith('+'):
            _, number = command.split()
            queue.append(int(number))
        elif command == '-':
            output.append(queue.popleft())

```

```

    with open(output_file, 'w', encoding='utf-8') as outfile:
        for value in output:
            outfile.write(f"{value}\n")

```

### • Параметры:

- input\_file: путь к входному файлу с командами.
- output\_file: путь к выходному файлу для результатов.

### • Логика работы:

1. Инициализируется пустая очередь queue и список для результатов output.



2. Читается количество команд  $M$  из первой строки входного файла.
3. Для каждой команды:
  - Если команда начинается с  $+$ , число  $N$  добавляется в очередь с помощью метода `append()`.
  - Если команда равна  $-$ , первый элемент удаляется из очереди методом `popleft()` и добавляется в список `output`.
4. После обработки всех команд числа из списка `output` записываются в выходной файл, каждое на новой строке.

### 3. Функция `main`:

```
def main():
    txtf_dir = os.path.join(os.path.dirname(__file__), '..',
                             'txtf')
    os.makedirs(txtf_dir, exist_ok=True)

    input_file_path = os.path.join(txtf_dir, 'input.txt')
    output_file_path = os.path.join(txtf_dir, 'output.txt')

    process_queue_commands(input_file_path, output_file_path)
```

- Определяет пути к входному и выходному файлам.
- Вызывает функцию `process_queue_commands` для обработки очереди.

### 4. Точка входа:

```
if __name__ == "__main__":
    main()
```

- Гарантирует, что функция `main()` выполнится, только если файл запускается как основная программа.

### Unittest для задание 2:

```
import os
import sys
import unittest

sys.path.insert(0, os.path.abspath(os.path.join(os.path.dirname(__file__), '..', 'src')))
from b2 import process_queue_commands

class TestProcessQueueCommands(unittest.TestCase):

    def setUp(self):
        # given
        self.test_dir = 'txtf'
        os.makedirs(self.test_dir, exist_ok=True)

        # when
        self.input_file = os.path.join(self.test_dir, 'input.txt')
        self.output_file = os.path.join(self.test_dir, 'output.txt')

        # then
        with open(self.input_file, 'w', encoding='utf-8') as f:
            f.write("4\n+ 1\n+ 10\n-\n-\n")

    def tearDown(self):
        # given
```

```

    if os.path.exists(self.test_dir):
        for file in os.listdir(self.test_dir):
            os.remove(os.path.join(self.test_dir, file))
        os.rmdir(self.test_dir)

    def test_process_queue_commands(self):
        # given
        process_queue_commands(self.input_file, self.output_file)

        # when
        with open(self.output_file, 'r', encoding='utf-8') as f:
            output = f.read().strip().split('\n')

        # then
        expected_output = ['1', '10']
        self.assertEqual(output, expected_output)

if __name__ == '__main__':
    unittest.main()

```

\* Результат :

## 1. Импортирование библиотек:

- `import os`: Модуль для работы с операционной системой (например, создание и удаление файлов).
- `import sys`: Модуль для работы с системой Python (например, для изменения путей поиска модулей).
- `import unittest`: Модуль для написания и запуска тестов (модуль юнит-тестирования).

## 2. Настройка пути к исходному коду:

```

sys.path.insert(0,
os.path.abspath(os.path.join(os.path.dirname(__file__), '..',
'src'))))

```

Эта строка добавляет путь к директории `src`, которая находится на уровень выше от текущего файла, в системный путь поиска модулей. Это нужно, чтобы импортировать функцию `process_queue_commands` из каталога `src`.

## 3. Создание класса тестов `TestProcessQueueCommands`:

Этот класс наследует `unittest.TestCase` и описывает тесты для функции `process_queue_commands`.

- **Метод `setUp()`**: Это метод, который вызывается перед каждым тестом. Он создает директорию для тестов (`txtf`), а затем записывает команды в файл `input.txt`.  
`self.input_file = os.path.join(self.test_dir, 'input.txt')`

```
self.output_file = os.path.join(self.test_dir,
'output.txt')
with open(self.input_file, 'w', encoding='utf-8') as f:
    f.write("4\n+ 1\n+ 10\n-\n-\n")
```

Здесь создаются два файла:

- `input.txt` с командами:

```
4
+ 1
+ 10
-
-
```

- `output.txt` будет использоваться для записи результатов.

- **Метод `tearDown()`**: Этот метод вызывается после каждого теста и удаляет созданные файлы и директорию:

```
if os.path.exists(self.test_dir):
    for file in os.listdir(self.test_dir):
        os.remove(os.path.join(self.test_dir, file))
    os.rmdir(self.test_dir)
```

#### 4. Тестирование функции `process_queue_commands()`:

```
def test_process_queue_commands(self):
    process_queue_commands(self.input_file, self.output_file)
    with open(self.output_file, 'r', encoding='utf-8') as f:
        output = f.read().strip().split('\n')
    expected_output = ['1', '10']
    self.assertEqual(output, expected_output)
```

- Вызов функции `process_queue_commands`, которая должна обработать команды из `input.txt` и записать результаты в `output.txt`.
- После выполнения функции результаты читаются из `output.txt` и сравниваются с ожидаемыми значениями:  
`expected_output = ['1', '10']`
- Метод `assertEqual()` проверяет, что фактический вывод из файла совпадает с ожидаемым результатом.

#### 5. Точка входа:

```
if __name__ == '__main__':
    unittest.main()
```

Это условие проверяет, что код выполняется как основная программа (не импортирован в другой модуль) и запускает все тесты с помощью `unittest.main()`.

### Задание 3 : Скобочная последовательность. Версия 1

Последовательность  $A$ , состоящую из символов из множества  $\langle \langle \rangle, \langle \rangle \rangle, \langle [ \rangle$  и  $\langle ] \rangle$ , назовем **правильной скобочной последовательностью**, если выполняется одно из следующих утверждений:

- $A$  – пустая последовательность;

- первый символ последовательности  $A$  – это «(», и в этой последовательности существует такой символ «)», что последовательность можно представить как  $A = (B)C$ , где  $B$  и  $C$  – правильные скобочные последовательности;
- первый символ последовательности  $A$  – это «[», и в этой последовательности существует такой символ «]», что последовательность можно представить как  $A = (B)C$ , где  $B$  и  $C$  – правильные скобочные последовательности.

Так, например, последовательности «(())» и «()[]» являются правильными скобочными последовательностями, а последовательности «[]» и «((» таковыми не являются.

Входной файл содержит несколько строк, каждая из которых содержит последовательность символов «(», «)», «[» и «]». Для каждой из этих строк выясните, является ли она правильной скобочной последовательностью.

- **Формат входного файла (input.txt).** Первая строка входного файла содержит число  $N$  ( $1 \leq N \leq 500$ ) – число скобочных последовательностей, которые необходимо проверить. Каждая из следующих  $N$  строк содержит скобочную последовательность длиной от 1 до  $10^4$  включительно. В каждой из последовательностей присутствуют только скобки указанных выше видов.
- **Формат выходного файла (output.txt).** Для каждой строки входного файла (кроме первой, в которой записано число таких строк) выведите в выходной файл «YES», если соответствующая последовательность является правильной скобочной последовательностью, или «NO», если не является.
- Ограничение по времени. 2 сек.
- Ограничение по памяти. 256 мб.
- Пример:

input.txt	output.txt
5	YES
()	YES
()	NO
()	NO
()	NO
()	

```
import os

def is_valid_bracket_sequence(sequence):
    stack = []
    bracket_map = {'(': ')', '[': ']'}

    for char in sequence:
        if char in bracket_map.values():
            stack.append(char)
        elif char in bracket_map.keys():
            if not stack or stack.pop() != bracket_map[char]:
                return False
    return not stack

def process_bracket_sequences(input_file, output_file):
    with open(input_file, 'r', encoding='utf-8') as infile:
        N = int(infile.readline().strip())
        results = []

        for _ in range(N):
```

```

sequence = infile.readline().strip()
if is_valid_bracket_sequence(sequence):
    results.append("YES")
else:
    results.append("NO")

with open(output_file, 'w', encoding='utf-8') as outfile:
    for result in results:
        outfile.write(result + "\n")

def main():
    input_file_path = os.path.join '..', 'txtf', 'input.txt'
    output_file_path = os.path.join '..', 'txtf', 'output.txt'

    process_bracket_sequences(input_file_path, output_file_path)

if __name__ == "__main__":
    main()

```

\* Результат :

input.txt :

```

5
()(
([)]
([)]
([)]
)(

```

output.txt:

```

YES
YES
NO
NO
NO

```

## 1. Функция `is_valid_bracket_sequence(sequence)`

Эта функция проверяет, является ли переданная строка корректной скобочной последовательностью.

### 1. Объявление переменных:

- `stack`: список, используемый как стек для проверки последовательности.
- `bracket_map`: словарь, который связывает закрывающие скобки с соответствующими открывающими ( ' ) ' : ' ( ' и ' ] ' : ' [ ' ).

### 2. Проход по символам строки:

- Если символ является открывающей скобкой ( ( или [ ), он добавляется в стек.
- Если символ является закрывающей скобкой ) или ] ), выполняются проверки:
  - Если стек пустой, это означает, что закрывающая скобка не имеет пары — возвращается `False`.
  - Если верхний элемент стека не соответствует открывающей скобке для текущей закрывающей, возвращается `False`.

### 3. Финальная проверка:

- Если стек пустой после обработки всех символов, это значит, что все открывающие скобки были корректно закрыты. Возвращается `True`.
- В противном случае — `False`.

## 2. Функция `process_bracket_sequences(input_file, output_file)`

Эта функция обрабатывает файл со скобочными последовательностями, проверяет каждую строку и записывает результат в файл.

### 1. Чтение данных из файла:

- Открывается входной файл.
- Первая строка содержит число последовательностей `N`.
- Каждая из следующих строк — это отдельная скобочная последовательность, которую нужно проверить.

### 2. Проверка каждой последовательности:

- Для каждой строки вызывается функция `is_valid_bracket_sequence(sequence)`.
- Если строка корректна, в список результатов добавляется `"YES"`, иначе — `"NO"`.

### 3. Запись результатов в файл:

- Открывается выходной файл.
- Каждое значение из списка результатов записывается в файл с новой строки.

## 3. Функция `main()`

1. Определяет пути к входному и выходному файлам.
2. Вызывает функцию `process_bracket_sequences()` для выполнения основной работы.

## Unittest для задание 3:

```
import os
import unittest
import sys

sys.path.insert(0, os.path.abspath(os.path.join(os.path.dirname(__file__), '..', 'src')))
from b3 import is_valid_bracket_sequence, process_bracket_sequences

class TestBracketSequence(unittest.TestCase):

    def setUp(self):
        # given
        self.test_dir = 'txtf'
        os.makedirs(self.test_dir, exist_ok=True)

        # when
        self.input_file = os.path.join(self.test_dir, 'input.txt')
        self.output_file = os.path.join(self.test_dir, 'output.txt')

        # then
        with open(self.input_file, 'w', encoding='utf-8') as f:
            f.write("5\n() ()\n[]\n[][]\n([])\n()\n\n")

    def tearDown(self):
        # given
        if os.path.exists(self.test_dir):
```

```

        for file in os.listdir(self.test_dir):
            os.remove(os.path.join(self.test_dir, file))
        os.rmdir(self.test_dir)

    def test_is_valid_bracket_sequence(self):
        # given
        self.assertTrue(is_valid_bracket_sequence("()()"))
        self.assertTrue(is_valid_bracket_sequence("([])"))
        self.assertFalse(is_valid_bracket_sequence("([)]"))
        self.assertFalse(is_valid_bracket_sequence("( [ ] )"))
        self.assertFalse(is_valid_bracket_sequence(" ( ) "))

    def test_process_bracket_sequences(self):
        # given
        process_bracket_sequences(self.input_file, self.output_file)

        # when
        with open(self.output_file, 'r', encoding='utf-8') as f:
            output = f.read().strip().split('\n')

        # then
        expected_output = ['YES', 'YES', 'NO', 'NO', 'NO'] # Kết quả mong đợi
        self.assertEqual(output, expected_output)

if __name__ == '__main__':
    unittest.main()

```

## \* Результат :

<div> <div>✓ Test Results17 ms</div> <div> <div>✓ test 317 ms</div> <div> <div>✓ TestBracketSequer17 ms</div> <div> <div>✓ test_is_valid_brac1ms</div> <div>✓ test_process_br16 ms</div> </div> </div> </div> </div>	<div> <div>✓ Tests passed: 2 of 2 tests – 17 ms</div> <div> <div>===== test session starts =====</div> <div>collecting ... collected 2 items</div> <div> <div>test 3.py::TestBracketSequence::test_is_valid_bracket_sequence PASSED [ 50%]</div> <div>test 3.py::TestBracketSequence::test_process_bracket_sequences PASSED [100%]</div> </div> <div>===== 2 passed, 2 warnings in 0.07s =====</div> </div> </div>
----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

## 1. Импорты и настройка

```

import os
import unittest
import sys

```

```

sys.path.insert(0,
os.path.abspath(os.path.join(os.path.dirname(__file__), '..',
'src')))
from b3 import is_valid_bracket_sequence,
process_bracket_sequences

```

- **os, unittest, sys:** Используются для работы с файловой системой, создания тестов и настройки пути для импорта функций из модуля b3.
- **sys.path.insert:** Добавляет путь к папке src, чтобы тесты могли импортировать функции `is_valid_bracket_sequence` и `process_bracket_sequences`.

## 2. Класс тестирования

Класс `TestBracketSequence` расширяет `unittest.TestCase` и содержит два теста.

### 3. Метод setUp

```
def setUp(self):
    self.test_dir = 'txtf'
    os.makedirs(self.test_dir, exist_ok=True)
    self.input_file = os.path.join(self.test_dir, 'input.txt')
    self.output_file = os.path.join(self.test_dir, 'output.txt')

    with open(self.input_file, 'w', encoding='utf-8') as f:
        f.write("5\n()()\n([\ ])\n([\ ])\n([\ ])\n(\n")
```

- **Создание тестовой директории `txtf`:** Метод запускается перед каждым тестом.
- **Файлы `input.txt` и `output.txt`:** Входной файл содержит 5 строк с разными скобочными последовательностями. Файл создается автоматически.

- Содержимое:

5  
( ) ( )  
( [ ] )  
( [ ) ]  
( ( ] ]  
) (

## 4. Метод `tearDown`

```
def tearDown(self):
    if os.path.exists(self.test_dir):
        for file in os.listdir(self.test_dir):
            os.remove(os.path.join(self.test_dir, file))
        os.rmdir(self.test_dir)
```

- **Очистка после теста:** Удаляет все файлы и директорию `txtf`, созданные в процессе тестирования.

## 5. Tect test is valid bracket sequence

```
def test_is_valid_bracket_sequence(self):
    self.assertTrue(is_valid_bracket_sequence("()()"))
    self.assertTrue(is_valid_bracket_sequence("([])"))
    self.assertFalse(is_valid_bracket_sequence("([)]"))
    self.assertFalse(is_valid_bracket_sequence("( [])"))
    self.assertFalse(is_valid_bracket_sequence(") (")
```

- Проверяет функцию `is_valid_bracket_sequence`, которая оценивает корректность последовательности:
  - `"()()"` и `"([])"` корректны  $\rightarrow$  возвращают `True`.
  - `"([)]"`, `"([)]"`, `"("` некорректны  $\rightarrow$  возвращают `False`.

## 6. Tect test process bracket sequences

```
def test_process_bracket_sequences(self):
    process_bracket_sequences(self.input_file, self.output_file)
```



```
with open(self.output_file, 'r', encoding='utf-8') as f:
    output = f.read().strip().split('\n')
```

```
expected_output = ['YES', 'YES', 'NO', 'NO', 'NO']
self.assertEqual(output, expected_output)
```

- **Функция process\_bracket\_sequences:**

- Читает входные данные из input\_file.
- Проверяет каждую строку через is\_valid\_bracket\_sequence.
- Записывает результаты в output\_file:
  - YES, если последовательность корректна.
  - NO, если некорректна.
- Результат для тестового ввода:  
['YES', 'YES', 'NO', 'NO', 'NO']

## 7. Запуск тестов

```
if __name__ == '__main__':
    unittest.main()
```

- Запускает тесты при выполнении файла.

## Задание 4 : Скобочная последовательность. Версия 2

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 ) .

- **Формат входного файла (input.txt).** Входные данные содержат одну строку  $S$ , состоящую из больших и маленьких латинских букв, цифр, знаков препинания и скобок из набора []{}(). Длина строки  $S - 1 \leq S \leq 10^5$ .
- **Формат выходного файла (output.txt).** Если в строке  $S$  скобки используются правильно, выведите «Success» (без кавычек). В противном случае выведите отсчитываемый от 1 индекс первой несовпадающей закрывающей скобки, а если нет несовпадающих закрывающих скобок, выведите отсчитываемый от 1 индекс первой открывающей скобки, не имеющей закрывающей.

- Ограничение по времени. 5 сек.
- Ограничение по памяти. 256 мб.
- Примеры:

input.txt	output.txt
[]	Success
{[]}[]	Success
[0]	Success
(0)	Success
{	1
{[]}	3
foo(bar);	Success
foo(bar[i];	10

```
import os

def check_brackets(sequence):
    stack = []
    bracket_map = {
        ')': '(',
        ']': '[',
        '}': '{'
    }

    for index, char in enumerate(sequence):
        if char in bracket_map.values():
            stack.append((char, index + 1))
        elif char in bracket_map.keys():
            if not stack:
                return index + 1
            top_char, top_index = stack.pop()
            if top_char != bracket_map[char]:
                return index + 1

    if stack:
        return stack[-1][1]

    return "Success"

def main():
    input_file_path = os.path.join '..', 'txtf', 'input.txt'
    output_file_path = os.path.join '..', 'txtf', 'output.txt'

    results = []

    with open(input_file_path, 'r', encoding='utf-8') as infile:
        for line in infile:
            sequence = line.strip()
            result = check_brackets(sequence)
            results.append(str(result))

    with open(output_file_path, 'w', encoding='utf-8') as outfile:
```

```
outfile.write("\n".join(results) + "\n")

if __name__ == '__main__':
    main()
```

\* Результат :

input.txt:

```
[]
{}[]
[()]
(( ))
{
{[]
foo(bar);
foo(bar[i];
```

output.txt:

```
Success
Success
Success
Success
1
3
Success
10
```

## 1. Функция `check_brackets`:

- **Аргумент:** принимает строку `sequence`, содержащую текст со скобками, буквами, цифрами и другими символами.
- **Цель:** проверить, правильно ли расставлены скобки в строке, и в случае ошибки вернуть индекс первой проблемной скобки (индексы считаются от 1).
- **Алгоритм:**
  1. Создается пустой стек `stack` для хранения открывающих скобок и их позиций.
  2. Создается словарь `bracket_map`, в котором каждой закрывающей скобке сопоставляется соответствующая открывающая.
  3. Перебираются все символы строки:
    - Если символ — открывающая скобка (`{`, `[`, `(`), он добавляется в стек вместе с его индексом.
    - Если символ — закрывающая скобка (`}`, `]`, `)`):
      - Если стек пуст (нет соответствующей открывающей скобки), возвращается текущий индекс.
      - Если верхняя скобка в стеке не соответствует текущей закрывающей, возвращается текущий индекс.
      - Если скобка совпала, она удаляется из стека.
  4. После обработки строки:

- Если стек не пуст (остались несоответствующие открывающие скобки), возвращается индекс первой из них.
- Если ошибок не обнаружено, возвращается "Success".

## 2. Функция main:

- Открывает входной файл, считывает строки с последовательностями скобок.
- Для каждой строки вызывает функцию `check_brackets` и записывает результат в список `results`.
- Результаты записываются в выходной файл построчно.

## Unittest для задание 4:

```
import os
import unittest
import sys

sys.path.insert(0, os.path.abspath(os.path.join(os.path.dirname(__file__), '..', 'src')))
from b4 import check_brackets

class TestCheckBrackets(unittest.TestCase):

    def test_success_cases(self):
        # given
        self.assertEqual(check_brackets("[]"), "Success")
        self.assertEqual(check_brackets("{}[]"), "Success")
        self.assertEqual(check_brackets("[()]"), "Success")
        self.assertEqual(check_brackets("()"), "Success")

    def test_first_unmatched_closing(self):
        # given
        self.assertEqual(check_brackets("{}"), 3)
        self.assertEqual(check_brackets("foo(bar[i]);"), 10)

    def test_first_unmatched_opening(self):
        # given
        self.assertEqual(check_brackets("("), 1)
        self.assertEqual(check_brackets "["), 1)
        self.assertEqual(check_brackets("foo(bar)"), "Success")

if __name__ == '__main__':
    unittest.main()
```

## \* Результат :

<div> <div>Test Results</div> <div>0 ms</div> </div> <div> <div>test 4</div> <div>0 ms</div> </div> <div> <div>TestCheckBrackets</div> <div>0 ms</div> </div> <div> <div>test_first_unmatc</div> <div>0 ms</div> </div> <div> <div>test_first_unmatc</div> <div>0 ms</div> </div> <div> <div>test_success_ca</div> <div>0 ms</div> </div>	<div> <div>Tests passed: 3 of 3 tests – 0 ms</div> </div> <div> <div>test session starts</div> <div>collecting ... collected 3 items</div> </div> <div> <div>test 4.py::TestCheckBrackets::test_first_unmatched_closing PASSED [ 33%]</div> <div>test 4.py::TestCheckBrackets::test_first_unmatched_opening PASSED [ 66%]</div> <div>test 4.py::TestCheckBrackets::test_success_cases PASSED [100%]</div> </div> <div> <div>3 passed, 3 warnings in 0.06s</div> </div>
-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

## 1. Импорт модулей

- **os, unittest, sys:** стандартные библиотеки Python:
  - `os` и `sys` используются для настройки пути импорта.

- `unittest` — библиотека для написания и запуска модульных тестов.
- `sys.path.insert(...)`: добавляет в путь поиска модулей (`sys.path`) директорию, содержащую тестируемую функцию `check_brackets`.

## 2. Класс `TestCheckBrackets`

Это класс с тестами, унаследованный от `unittest.TestCase`. Он содержит методы для проверки различных сценариев работы функции `check_brackets`.

## 3. Тесты

### a. `test_success_cases`

- Проверяет корректную работу функции на правильных строках:
  - `"[]"`, `"{} []"`, `"[()]"`, `"(())"` — все строки правильно сбалансированы.
- Ожидаемый результат: строка `"Success"`.

### b. `test_first_unmatched_closing`

- Проверяет случаи, где первая проблема возникает из-за лишней или неверной закрывающей скобки:
  - В строке `"{ []}"` на позиции 3 обнаружена закрывающая `}`, не соответствующая ожидаемой `]`.
  - В строке `"foo(bar[i]);"` на позиции 10 обнаружена лишняя закрывающая `)`.

### c. `test_first_unmatched_opening`

- Проверяет случаи, где строка содержит лишнюю открывающую скобку:
  - В строках `" ("` и `"["` лишние скобки находятся на 1-й позиции.
  - В строке `"foo(bar)"` скобки корректно сбалансированы, ожидаемый результат: `"Success"`.

## 4. Точка входа

- Если файл запускается напрямую, выполняется `unittest.main()`. Этот вызов запускает все тесты, определенные в классе `TestCheckBrackets`.

## Задание 5 : Стек с максимумом

Стек - это абстрактный тип данных, поддерживающий операции `Push()` и `Pop()`. Нетрудно реализовать его таким образом, чтобы обе эти операции работали за константное время. В этой задаче ваша цель - реализовать стек, который также поддерживает поиск максимального значения и гарантирует, что все операции по-прежнему работают за константное время.

Реализуйте стек, поддерживающий операции `Push()`, `Pop()` и `Max()`.

- **Формат входного файла (`input.txt`).** В первой строке входного файла содержится  $n$  ( $1 \leq n \leq 400000$ ) – число команд. Последующие  $n$  строк исходного файла содержат ровно одну команду: `push V`, `pop` или `max`.  $0 \leq V \leq 10^5$ .
- **Формат выходного файла (`output.txt`).** Для каждого запроса `max` выведите (в отдельной строке) максимальное значение стека.
- Ограничение по времени. 5 сек.
- Ограничение по памяти. 512 мб.
- Пример:

input.txt	output.txt	input.txt	output.txt	input.txt	output.txt
5	2	5	2	3	
push 2	2	push 1	1	push 1	
push 1		push 2		push 7	
max		max		pop	
pop		pop			
max		max			

input.txt	output.txt	input.txt	output.txt
10	9	6	7
push 2	9	push 7	7
push 3	9	push 1	
push 9	9	push 7	
push 7		max	
push 2		pop	
max		max	
max			
max			
pop			
max			

```

class MaxStack:
    def __init__(self):
        self.stack = []
        self.max_stack = []

    def push(self, value):
        self.stack.append(value)
        if not self.max_stack or value >= self.max_stack[-1]:
            self.max_stack.append(value)

    def pop(self):
        if self.stack:
            value = self.stack.pop()
            if value == self.max_stack[-1]:
                self.max_stack.pop()

    def max(self):
        if self.max_stack:
            return self.max_stack[-1]
        return None

def main():
    input_file_path = '../txtf/input.txt'
    output_file_path = '../txtf/output.txt'

    max_stack = MaxStack()
    results = []

    with open(input_file_path, 'r', encoding='utf-8') as infile:
        n = int(infile.readline().strip())
        for _ in range(n):
            command = infile.readline().strip().split()
            if command[0] == 'push':
                value = int(command[1])
                max_stack.push(value)

```

```

elif command[0] == 'pop':
    max_stack.pop()
elif command[0] == 'max':
    results.append(max_stack.max())

with open(output_file_path, 'w', encoding='utf-8') as outfile:
    for result in results:
        outfile.write(str(result) + '\n')

if __name__ == '__main__':
    main()

```

\* Результат :

input.txt:

```

5
push 2
push 1
max
pop
max

```

output.txt:

```

2
2

```

## Основная структура

### 1. Класс MaxStack:

- Хранит два списка:
  - stack: основной стек, в котором хранятся добавляемые элементы.
  - max\_stack: дополнительный стек для отслеживания текущего максимального значения.

### 2. Методы:

- \_\_init\_\_: инициализирует оба стека как пустые.
- push(value):
  - Добавляет value в основной стек.
  - Если max\_stack пуст или value больше/равно текущему максимуму, добавляет value в max\_stack.
- pop():
  - Удаляет верхний элемент из основного стека.
  - Если этот элемент равен текущему максимуму (max\_stack[-1]), то он также удаляется из max\_stack.
- max():
  - Возвращает верхний элемент из max\_stack, что является текущим максимумом стека. Если max\_stack пуст, возвращает None.

## Разбор основного кода

### 1. Чтение входных данных:

- Из файла input.txt читается количество команд n и последовательно выполняются команды push, pop или max.

### 2. Обработка команд:

- push V: вызывает метод push() с заданным значением V.
- pop: вызывает метод pop() для удаления верхнего элемента.
- max: вызывает метод max() и сохраняет результат в списке results.

### 3. Вывод результатов:

- После обработки всех команд результаты запросов max записываются в файл output.txt, по одному значению на строку.

### Unittest для задание 5:

```
import os
import unittest
import sys

sys.path.insert(0, os.path.abspath(os.path.join(os.path.dirname(__file__), '..', 'src')))
from b5 import MaxStack

class TestMaxStack(unittest.TestCase):

    def setUp(self):
        self.max_stack = MaxStack()

    def test_push_and_max(self):
        # given
        self.max_stack.push(2)
        self.assertEqual(self.max_stack.max(), 2)

        # when
        self.max_stack.push(1)
        self.assertEqual(self.max_stack.max(), 2)

        # then
        self.max_stack.push(3)
        self.assertEqual(self.max_stack.max(), 3)

    def test_pop(self):
        # given
        self.max_stack.push(1)
        self.max_stack.push(2)
        self.max_stack.push(3)

        # when
        self.max_stack.pop()
        self.assertEqual(self.max_stack.max(), 2)

        # then
        self.max_stack.pop()
        self.assertEqual(self.max_stack.max(), 1)

    def test_multiple_max(self):
        # given
        self.max_stack.push(5)
        self.max_stack.push(3)
        self.max_stack.push(8)

        # when
        self.assertEqual(self.max_stack.max(), 8)
        self.max_stack.pop()

        # then
        self.assertEqual(self.max_stack.max(), 5)
        self.max_stack.pop()
```



```

        self.assertEqual(self.max_stack.max(), 5)

if __name__ == '__main__':
    unittest.main()

```

## \* Результат :

```

Test Results 0 ms
└─ test 5 0 ms
    └─ TestMaxStack 0 ms
        └─ test_multiple_max 0 ms
        └─ test_pop 0 ms
        └─ test_push_and_max 0 ms

Tests passed: 3 of 3 tests - 0 ms

===== test session starts =====
collecting ... collected 3 items

test 5.py::TestMaxStack::test_multiple_max PASSED [ 33%]
test 5.py::TestMaxStack::test_pop PASSED [ 66%]
test 5.py::TestMaxStack::test_push_and_max PASSED [100%]

===== 3 passed, 3 warnings in 0.04s =====

```

## 1. Импорты и настройка пути

```

import os
import unittest
import sys

```

```

sys.path.insert(0,
os.path.abspath(os.path.join(os.path.dirname(__file__), '..',
'src'))))
from b5 import MaxStack

```

### • Импорты:

- `os` и `sys`: используются для работы с путями файлов.
- `unittest`: стандартная библиотека Python для написания и выполнения тестов.

### • Настройка пути:

- Добавляется путь к каталогу `src`, чтобы Python мог найти модуль `MaxStack` в файле `b5.py`.

## 2. Класс тестирования **TestMaxStack**

```

class TestMaxStack(unittest.TestCase):

```

- Наследует `unittest.TestCase`, чтобы определить тестовые методы.

### Метод **setUp**

```

def setUp(self):
    self.max_stack = MaxStack()

```

- Выполняется перед каждым тестом.
- Создает новый экземпляр класса `MaxStack`, чтобы тесты были независимыми друг от друга.

## 3. Тесты

### Тест **test\_push\_and\_max**

```

def test_push_and_max(self):
    # given
    self.max_stack.push(2)
    self.assertEqual(self.max_stack.max(), 2)

```

```
# when
self.max_stack.push(1)
self.assertEqual(self.max_stack.max(), 2)

# then
self.max_stack.push(3)
self.assertEqual(self.max_stack.max(), 3)
```

- Проверяет работу операций push и max.
  1. Добавляет 2 в стек и проверяет, что максимум равен 2.
  2. Добавляет 1, но максимум остается 2.
  3. Добавляет 3, теперь максимум становится 3.

### Тест **test\_pop**

```
def test_pop(self):
    # given
    self.max_stack.push(1)
    self.max_stack.push(2)
    self.max_stack.push(3)

    # when
    self.max_stack.pop()
    self.assertEqual(self.max_stack.max(), 2)

    # then
    self.max_stack.pop()
    self.assertEqual(self.max_stack.max(), 1)
```

- Проверяет, что после удаления элемента с вершины стека (pop) максимум обновляется корректно:
  1. Добавляет в стек элементы 1, 2, 3.
  2. Удаляет 3 и проверяет, что максимум стал 2.
  3. Удаляет 2 и проверяет, что максимум стал 1.

### Тест **test\_multiple\_max**

```
def test_multiple_max(self):
    # given
    self.max_stack.push(5)
    self.max_stack.push(3)
    self.max_stack.push(8)

    # when
    self.assertEqual(self.max_stack.max(), 8)
    self.max_stack.pop()

    # then
    self.assertEqual(self.max_stack.max(), 5)
    self.max_stack.pop()
```

```
self.assertEqual(self.max_stack.max(), 5)
```

- Проверяет работу стека, когда максимальный элемент изменяется неоднократно:
  1. Добавляет 5, 3, 8. Текущий максимум — 8.
  2. Удаляет 8, максимум становится 5.
  3. Удаляет 3, максимум остается 5.

#### 4. Запуск тестов

```
if __name__ == '__main__':  
    unittest.main()
```

- Запускает все тесты, определенные в классе TestMaxStack.

### Задание 6 : Очередь с минимумом

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  $10^9$ . Команда «-» означает изъятие элемента из очереди. Команда «?» означает запрос на поиск минимального элемента в очереди.

- **Формат входного файла (input.txt).** В первой строке содержится  $M$  ( $1 \leq M \leq 10^6$ ) – 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. 2 сек.
- Ограничение по памяти. 256 мб.
- Пример:

input.txt	output.txt
7	1
+ 1	1
?	10
+ 10	
?	
-	
?	
-	

- Вам может помочь идея, изложенная во второй части [вот этой страницы](#).

```
from collections import deque  
  
class MinQueue:  
    def __init__(self):  
        self.queue = deque()  
        self.min_queue = deque()
```

```

def add(self, value):
    self.queue.append(value)
    while self.min_queue and self.min_queue[-1] > value:
        self.min_queue.pop()
    self.min_queue.append(value)

def remove(self):
    if not self.queue:
        raise IndexError("remove from empty queue")
    value = self.queue.popleft()
    if value == self.min_queue[0]:
        self.min_queue.popleft()

def get_min(self):
    if not self.min_queue:
        raise IndexError("get_min from empty queue")
    return self.min_queue[0]

def main():
    input_file_path = '../txtf/input.txt'
    output_file_path = '../txtf/output.txt'

    min_queue = MinQueue()
    results = []

    with open(input_file_path, 'r', encoding='utf-8') as infile:
        m = int(infile.readline().strip())
        for _ in range(m):
            command = infile.readline().strip().split()
            if command[0] == '+':
                value = int(command[1])
                min_queue.add(value)
            elif command[0] == '-':
                min_queue.remove()
            elif command[0] == '?':
                results.append(min_queue.get_min())

    with open(output_file_path, 'w', encoding='utf-8') as outfile:
        for result in results:
            outfile.write(str(result) + '\n')

if __name__ == '__main__':
    main()

```

**\* Результат :**

**input.txt:**

```

7
+ 1
?
+ 10
?
-
?
-

```

```
1
1
10
```

output.txt:

## 1. Класс `MinQueue`

Класс реализует очередь с поддержкой операций:

- Добавление элемента (`add`)
- Удаление элемента (`remove`)
- Запрос минимального элемента (`get_min`)

Поля:

- **`self.queue`**: основная очередь, реализованная с помощью `deque`.
- **`self.min_queue`**: вспомогательная очередь, которая хранит минимальные элементы в порядке возрастания.

Методы:

- **`add(self, value)`**:
  - Добавляет элемент в основную очередь.
  - Удаляет из `min_queue` все элементы, которые больше добавляемого значения (они не могут быть минимальными после добавления нового значения).
  - Добавляет новый элемент в `min_queue`.
- **`remove(self)`**:
  - Удаляет элемент из начала основной очереди.
  - Если удаляемый элемент совпадает с текущим минимальным (первым в `min_queue`), то удаляет его также из `min_queue`.
- **`get_min(self)`**:
  - Возвращает первый элемент из `min_queue`, который является минимальным в текущей очереди.

## 2. Функция `main()`

Обрабатывает команды из входного файла и записывает результаты в выходной файл.

Этапы:

### 1. Чтение данных:

- Открывает входной файл, читает количество команд `M`.
- Обрабатывает каждую команду:
  - Команда `+` `N`: добавляет число `N` в очередь.
  - Команда `-`: удаляет первый элемент из очереди.
  - Команда `?`: добавляет минимальный элемент в список результатов.

### 2. Запись результатов:

- Открывает выходной файл.
- Записывает все результаты операций запроса минимального элемента (?) построчно.

## 3. Особенности

- **Очередь минимальных элементов (`min_queue`)**:
  - Обеспечивает быстрый доступ к минимальному элементу за  $O(1)$ .

- Хранит только те значения, которые могут быть минимальными на текущий момент.
- **Гарантии:**
  - Не допускаются операции удаления (-) или запроса минимума (?) для пустой очереди, что исключает необходимость дополнительных проверок.

## Unittest для задание 6:

```
import os
import unittest
import sys

sys.path.insert(0, os.path.abspath(os.path.join(os.path.dirname(__file__), '..', 'src')))
from b6 import MinQueue

class TestMinQueue(unittest.TestCase):

    def setUp(self):
        self.min_queue = MinQueue()

    def test_add_and_min(self):
        # given
        self.min_queue.add(5)
        self.assertEqual(self.min_queue.get_min(), 5)

        # when
        self.min_queue.add(3)
        self.assertEqual(self.min_queue.get_min(), 3)

        # when
        self.min_queue.add(4)
        self.assertEqual(self.min_queue.get_min(), 3)

        # then
        self.min_queue.add(1)
        self.assertEqual(self.min_queue.get_min(), 1)

    def test_remove(self):
        # given
        self.min_queue.add(5)
        self.min_queue.add(3)
        self.min_queue.add(1)
        self.min_queue.remove()

        # when
        self.assertEqual(self.min_queue.get_min(), 1)
        self.min_queue.remove()
        self.assertEqual(self.min_queue.get_min(), 1)
        self.min_queue.remove()

        # then
        with self.assertRaises(IndexError):
            self.min_queue.get_min()

    def test_multiple_min_queries(self):
        self.min_queue.add(10)
        self.min_queue.add(20)
        self.min_queue.add(5)
        self.assertEqual(self.min_queue.get_min(), 5)
        self.min_queue.remove()
        self.assertEqual(self.min_queue.get_min(), 5)
```

```

        self.min_queue.remove()
        self.assertEqual(self.min_queue.get_min(), 5)

    def test_empty_queue(self):
        with self.assertRaises(IndexError):
            self.min_queue.remove()
        self.min_queue.add(10)
        self.min_queue.remove()
        with self.assertRaises(IndexError):
            self.min_queue.get_min()

if __name__ == '__main__':
    unittest.main()

```

\* Результат :

The screenshot shows a test runner interface. On the left, a tree view shows the test hierarchy: 'Test Results' (0 ms) contains 'test 6' (0 ms), which contains 'TestMinQueue' (0 ms). Under 'TestMinQueue', four sub-items are listed, all with 0 ms: 'test\_add\_and\_min', 'test\_empty\_queue', 'test\_multiple\_min\_queries', and 'test\_remove'. On the right, the test output is displayed. It starts with '==== test session starts ===', followed by 'collecting ... collected 4 items'. Then, four test results are shown: 'test 6.py::TestMinQueue::test\_add\_and\_min PASSED [ 25%]', 'test 6.py::TestMinQueue::test\_empty\_queue PASSED [ 50%]', 'test 6.py::TestMinQueue::test\_multiple\_min\_queries PASSED [ 75%]', and 'test 6.py::TestMinQueue::test\_remove PASSED [100%]'. The session ends with '==== 4 passed, 4 warnings in 0.03s ====='.

## 1. Импорты и настройка пути

```

import os
import unittest
import sys
sys.path.insert(0,
os.path.abspath(os.path.join(os.path.dirname(__file__), '..',
'src'))))
from b6 import MinQueue

```

- **os** и **sys**: используются для настройки пути к модулю MinQueue.
- **sys.path.insert**: добавляет каталог ../src в sys.path, чтобы модуль MinQueue был доступен для импорта.

## 2. Класс TestMinQueue

Класс определяет тесты для проверки функциональности MinQueue. Он наследует unittest.TestCase, что позволяет использовать встроенные функции тестирования.

### Метод setUp

```

def setUp(self):
    self.min_queue = MinQueue()

```

- Этот метод вызывается перед каждым тестом.
- Создает экземпляр MinQueue, который используется в тестах.

## 3. Тесты

### test\_add\_and\_min

```

def test_add_and_min(self):
    # given
    self.min_queue.add(5)
    self.assertEqual(self.min_queue.get_min(), 5)

```

```
# when
self.min_queue.add(3)
self.assertEqual(self.min_queue.get_min(), 3)
```

```
# when
self.min_queue.add(4)
self.assertEqual(self.min_queue.get_min(), 3)
```

```
# then
self.min_queue.add(1)
self.assertEqual(self.min_queue.get_min(), 1)
```

- Тест проверяет корректность добавления элементов и получения минимального элемента.
- Используется `assertEqual` для проверки, что минимальный элемент соответствует ожидаемому значению.

### **test\_remove**

```
def test_remove(self):
    # given
    self.min_queue.add(5)
    self.min_queue.add(3)
    self.min_queue.add(1)
    self.min_queue.remove()

    # when
    self.assertEqual(self.min_queue.get_min(), 1)
    self.min_queue.remove()
    self.assertEqual(self.min_queue.get_min(), 1)
    self.min_queue.remove()
```

```
# then
with self.assertRaises(IndexError):
    self.min_queue.get_min()
```

- Проверяет корректность удаления элементов.
- Удаляет элементы и проверяет обновление минимального значения.
- Проверяет, что при попытке получить минимальный элемент из пустой очереди возникает `IndexError`.

### **test\_multiple\_min\_queries**

```
def test_multiple_min_queries(self):
    self.min_queue.add(10)
    self.min_queue.add(20)
    self.min_queue.add(5)
    self.assertEqual(self.min_queue.get_min(), 5)
    self.min_queue.remove()
    self.assertEqual(self.min_queue.get_min(), 5)
```



```
self.min_queue.remove()
self.assertEqual(self.min_queue.get_min(), 5)
```

- Проверяет, что `get_min` возвращает корректное значение после нескольких запросов.
- Убеждается, что удаление не нарушает логику работы очереди минимальных элементов.

### **test\_empty\_queue**

```
def test_empty_queue(self):
    with self.assertRaises(IndexError):
        self.min_queue.remove()
    self.min_queue.add(10)
    self.min_queue.remove()
    with self.assertRaises(IndexError):
        self.min_queue.get_min()
```

- Проверяет поведение очереди при попытке выполнить операции с пустой очередью.
- Убеждается, что удаление и запрос минимума из пустой очереди вызывают `IndexError`.

## **4. Запуск тестов**

```
if __name__ == '__main__':
    unittest.main()
```

- Запускает тесты при выполнении файла как основного модуля.

## **Задание 7 : Максимум в движущейся последовательности**

Задан массив из  $n$  целых чисел -  $a_1, \dots, a_n$  и число  $m < n$ , нужно найти максимум среди последовательности ("окна")  $\{a_i, \dots, a_{i+m-1}\}$  для каждого значения  $1 \leq i \leq n - m + 1$ . Простой алгоритм решения этой задачи за  $O(nm)$  сканирует каждое "окно" отдельно.

Ваша цель - алгоритм за  $O(n)$ .

- **Формат входного файла (input.txt).** В первой строке содержится целое число  $n$  ( $1 \leq n \leq 10^5$ ) – количество чисел в исходном массиве, вторая строка содержит  $n$  целых чисел  $a_1, \dots, a_n$  этого массива, разделенных пробелом ( $0 \leq a_i \leq 10^5$ ). В третьей строке - целое число  $m$  - ширина "окна" ( $1 \leq m \leq n$ ).
- **Формат выходного файла (output.txt).** Нужно вывести  $\max a_i, \dots, a_{i+m-1}$  для каждого  $1 \leq i \leq n - m + 1$ .
- Ограничение по времени. 5 сек.
- Ограничение по памяти. 512 мб.
- Пример:

input.txt	output.txt
8	7 7 5 6 6
2 7 3 1 5 2 6 2	
4	

Есть несколько решений этой задачи. Например:

- использование очереди на основе двух стеков;
- использование Dequeue.

```
from collections import deque

def max_in_sliding_window(n, arr, m):
    result = []
    deq = deque()

    for i in range(n):
        if deq and deq[0] < i - m + 1:
            deq.popleft()

        while deq and arr[deq[-1]] < arr[i]:
            deq.pop()

        deq.append(i)

        if i >= m - 1:
            result.append(arr[deq[0]])

    return result

def main():
    input_file_path = '../txtf/input.txt'
    output_file_path = '../txtf/output.txt'

    with open(input_file_path, 'r', encoding='utf-8') as infile:
        n = int(infile.readline().strip())
        arr = list(map(int, infile.readline().strip().split()))
        m = int(infile.readline().strip())

    result = max_in_sliding_window(n, arr, m)

    with open(output_file_path, 'w', encoding='utf-8') as outfile:
        outfile.write(' '.join(map(str, result)) + '\n')

if __name__ == '__main__':
    main()
```

**\*Результат :**

input.txt:

```
8
2 7 3 1 5 2 6 2
4
```

output.txt:

```
7 7 5 6 6
```

## 1. Функция `max_in_sliding_window(n, arr, m)`

### 1. Инициализация:

- `result = []`: список для хранения результатов (максимальных значений в каждом окне).
- `deq = deque()`: двусторонняя очередь для индексов элементов массива, поддерживающая быструю вставку и удаление с обоих концов.

### 2. Итерация по массиву:

- Цикл проходит по всем индексам  $i$  от 0 до  $n-1$ .

### 3. Удаление устаревших элементов из очереди:

- `if deq and deq[0] < i - m + 1`: Проверяем, не вышел ли индекс, находящийся в начале очереди, за пределы текущего окна размера  $m$ . Если да — удаляем его из очереди.

### 4. Удаление элементов, меньших текущего:

- `while deq and arr[deq[-1]] < arr[i]`: Удаляем из конца очереди индексы элементов, которые меньше текущего элемента `arr[i]`. Это обеспечивает, что в начале очереди всегда будет индекс наибольшего элемента.

### 5. Добавление текущего индекса:

- `deq.append(i)`: Добавляем текущий индекс в очередь.

### 6. Добавление максимума текущего окна в результат:

- `if i >= m - 1`: После того как обработаны индексы, достаточные для формирования первого окна, добавляем максимальный элемент текущего окна (`arr[deq[0]]`) в `result`.

### 7. Возврат результата:

- Возвращаем список `result`, содержащий максимумы всех окон.

## 2. Функция `main()`

### 1. Чтение входных данных:

- Считываются три строки из файла:
  - `n`: количество элементов массива.
  - `arr`: массив целых чисел.
  - `m`: ширина окна.

### 2. Вызов функции обработки:

- `max_in_sliding_window` вызывается с параметрами `n, arr, m`.

### 3. Запись результата:

- Результаты сохраняются в выходной файл `output.txt` в виде строки чисел, разделенных пробелами.

## Unittest для задание 7:

```
import os
import unittest
import sys
from collections import deque

sys.path.insert(0, os.path.abspath(os.path.join(os.path.dirname(__file__), '..', 'src')))
from b7 import max_in_sliding_window

class TestMaxInSlidingWindow(unittest.TestCase):

    def test_basic_cases(self):
        # given
        self.assertEqual(max_in_sliding_window(8, [2, 7, 3, 1, 5, 2, 6, 2], 4), [7, 7, 5, 6, 6])
        self.assertEqual(max_in_sliding_window(8, [1, 3, -1, -3, 5, 3, 6, 7], 3), [3, 3, 5, 5, 6, 7])
        self.assertEqual(max_in_sliding_window(1, [1], 1), [1])
```

```

        self.assertEqual(max_in_sliding_window(5, [1, 2, 3, 4, 5], 2), [2, 3, 4, 5])

    def test_edge_cases(self):
        # given
        self.assertEqual(max_in_sliding_window(5, [5, 5, 5, 5, 5], 5), [5])
        self.assertEqual(max_in_sliding_window(3, [1, 2, 3], 1), [1, 2, 3])

    def test_large_input(self):
        # given
        large_input = list(range(100000))

        # when
        expected_output = [i + 999 for i in range(99001)]

        # then
        self.assertEqual(max_in_sliding_window(100000, large_input, 1000),
expected_output)

if __name__ == '__main__':
    unittest.main()

```

## \* Результат:

The screenshot shows the output of a test runner. On the left, a tree view shows the test results: 'Test Results' (30 ms) is expanded, showing 'test 7' (30 ms) which is also expanded to show three sub-tests: 'TestMaxInSlidingW' (30 ms), 'test\_basic\_cases' (0 ms), 'test\_edge\_cases' (0 ms), and 'test\_large\_input' (30 ms). On the right, the test session log shows: 'Tests passed: 3 of 3 tests - 30 ms', 'test session starts', 'collecting ... collected 3 items', the names of the three tests, '3 passed, 3 warnings in 0.06s', a progress bar showing 33%, 66%, and 100% completion, and 'Process finished with exit code 0'.

## 1. Импорты

1. **os и sys:** Используются для манипуляций с путями файловой системы и добавления пути до исходного кода в `sys.path`.
2. **unittest:** Библиотека для написания и выполнения модульных тестов.
3. **deque из collections:** Хотя в этом коде он не используется, deque применяется в реализации функции `max_in_sliding_window`.

## 2. Добавление пути

```

sys.path.insert(0,
os.path.abspath(os.path.join(os.path.dirname(__file__), '..',
'src'))))

```

Эта строка добавляет каталог `../src` к пути импорта модулей. Это нужно, чтобы импортировать функцию `max_in_sliding_window` из модуля `b7`.

## 3. Класс `TestMaxInSlidingWindow`

Создается класс, наследующийся от `unittest.TestCase`, содержащий методы-тесты.

### 1. Метод `test_basic_cases`

Проверяет базовые случаи:

- Тестируются массивы разного размера и окна с различной длиной.
- Проверяется корректность результатов для обычных входных данных:

```
self.assertEqual(max_in_sliding_window(8, [2, 7, 3, 1, 5, 2, 6, 2], 4), [7, 7, 5, 6, 6])
```

## 2. Метод `test_edge_cases`

Тестирует крайние случаи:

- Массив состоит из одинаковых элементов:  

```
self.assertEqual(max_in_sliding_window(5, [5, 5, 5, 5, 5], 5), [5])
```
- Длина окна равна 1, то есть максимум на каждом шаге равен самому элементу.

## 3. Метод `test_large_input`

Тестирует работу функции на больших объемах данных:

- Генерируется большой массив `large_input` из 100,000 элементов.
- Ожидаемый результат вычисляется заранее:  

```
expected_output = [i + 999 for i in range(99001)]
```
- Тест проверяет, справляется ли функция с большими входными данными без ошибок.

## 4. Запуск тестов

```
if __name__ == '__main__':  
    unittest.main()
```

Эта конструкция гарантирует, что тесты будут запущены только при прямом запуске файла.

## Задание 8 : Постфиксная запись

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  $N$  ( $1 \leq n \leq 10^6$ ) – 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. 2 сек.
- Ограничение по памяти. 256 мб.
- Пример:

input.txt	output.txt
7	-102
8 9 + 1 7 - *	

```

import os

def evaluate_postfix(expression):
    stack = []
    for token in expression:
        if token.isdigit():
            stack.append(int(token))
        else:
            b = stack.pop()
            a = stack.pop()
            if token == '+':
                stack.append(a + b)
            elif token == '-':
                stack.append(a - b)
            elif token == '*':
                stack.append(a * b)
    return stack[0]

def main():
    input_path = os.path.join '..', 'txtf', 'input.txt'
    output_path = os.path.join '..', 'txtf', 'output.txt'

    with open(input_path, 'r') as input_file:
        n = int(input_file.readline().strip())
        expression = input_file.readline().strip().split()

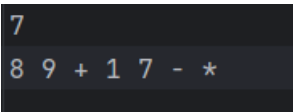
    result = evaluate_postfix(expression)

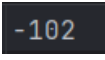
    with open(output_path, 'w') as output_file:
        output_file.write(str(result))

if __name__ == '__main__':
    main()

```

\* Результат:

input.txt: 

output.txt: 

## 1. Функция `evaluate_postfix`

### 1. Аргумент:

- `expression`: список строк, каждая из которых представляет либо операнд (число), либо оператор (+, -, \*).

### 2. Логика функции:

- Создаем пустой стек `stack`, который будет использоваться для хранения чисел (операндов).
- Проходим по каждому элементу выражения:
  - Если элемент — число (выявляется с помощью `isdigit()`), преобразуем его в `int` и добавляем в стек.
  - Если элемент — оператор (+, -, \*), выполняем следующие действия:
    - Извлекаем два верхних элемента стека: сначала `b`, затем `a`.
    - Применяем оператор к этим числам.

- Результат операции помещаем обратно в стек.
- В конце работы функции в стеке останется единственное число — результат вычисления выражения.

### 3. Возврат результата:

- Возвращаем единственное число из стека (`stack[0]`), которое является результатом вычислений.

## 2. Функция `main`

### 1. Пути к файлам:

- `input_path`: путь к входному файлу, где записано выражение.
- `output_path`: путь к выходному файлу, в который записывается результат.

### 2. Чтение данных:

- Открываем входной файл в режиме чтения.
- Считываем количество элементов выражения (`n`) из первой строки (оно не используется далее напрямую).
- Считываем само выражение из второй строки и разделяем его на элементы с помощью метода `.split()`.

### 3. Вычисление выражения:

- Вызываем функцию `evaluate_postfix`, передавая ей список элементов выражения.
- Сохраняем результат вычислений в переменной `result`.

### 4. Запись результата:

- Открываем выходной файл в режиме записи.
- Записываем результат в файл в виде строки.

## 3. Работа программы

- Если запускать скрипт как отдельную программу (через `__main__`), то он прочитает входные данные, вычислит результат и запишет его в файл `output.txt`.

## Unittest для задание 8:

```
import os
import unittest
import sys

sys.path.insert(0, os.path.abspath(os.path.join(os.path.dirname(__file__), '..', 'src')))
from b8 import evaluate_postfix

class TestPostfixEvaluator(unittest.TestCase):

    def test_basic_operations(self):
        # given
        self.assertEqual(evaluate_postfix(['2', '3', '+']), 5)
        self.assertEqual(evaluate_postfix(['5', '1', '2', '+', '4', '*', '+']), 17)
        self.assertEqual(evaluate_postfix(['4', '2', '-']), 2)
        self.assertEqual(evaluate_postfix(['0', '0', '+']), 0)

    def test_complex_expression(self):
        # given
        self.assertEqual(evaluate_postfix(['3', '4', '+', '2', '*', '7', '-']), 7)
        self.assertEqual(evaluate_postfix(['8', '9', '+', '1', '7', '-', '*']), -102)

    def test_edge_cases(self):
```

```
# given
self.assertEqual(evaluate_postfix(['1']), 1)

if __name__ == '__main__':
    unittest.main()
```

\* Результат:

```

✓ Test Results 0 ms
  ✓ test 8 0 ms
    ✓ TestPostfixEvaluator 0 ms
      ✓ test_basic_operations 0 ms
      ✓ test_complex_expression 0 ms
      ✓ test_edge_cases 0 ms

✓ Tests passed: 3 of 3 tests - 0 ms

===== test session starts =====
collecting ... collected 3 items

test 8.py::TestPostfixEvaluator::test_basic_operations PASSED [ 33%]
test 8.py::TestPostfixEvaluator::test_complex_expression PASSED [ 66%]
test 8.py::TestPostfixEvaluator::test_edge_cases PASSED [100%]

===== 3 passed, 3 warnings in 0.03s =====

```

## 1. Подключение модулей

- **os** и **sys**: Используются для изменения пути поиска модулей Python. Это позволяет импортировать функцию `evaluate_postfix` из директории `src`, которая находится на уровень выше текущего файла тестов.  
`sys.path.insert(0, os.path.abspath(os.path.join(os.path.dirname(__file__), '..', 'src')))`  
 Это удобно при организации проектов, где исходный код и тесты хранятся в отдельных папках.
- **unittest**: Модуль для создания и запуска тестов.

## 2. Класс `TestPostfixEvaluator`

Класс, наследующийся от `unittest.TestCase`, содержит тесты для функции `evaluate_postfix`.

### Метод `test_basic_operations`

Тестирует базовые операции:

- `['2', '3', '+']`: Результат  $2 + 3 = 5$ .
- `['5', '1', '2', '+', '4', '*', '+']`: Выражение эквивалентно  $5 + (1 + 2) * 4 = 17$ .
- `['4', '2', '-']`: Результат  $4 - 2 = 2$ .
- `['0', '0', '+']`: Результат  $0 + 0 = 0$ .

```
self.assertEqual(evaluate_postfix(['2', '3', '+']), 5)
```

### Метод `test_complex_expression`

Тестирует более сложные выражения:

- `['3', '4', '+', '2', '*', '7', '-']`:
  - Сначала  $3 + 4 = 7$ .
  - Затем  $7 * 2 = 14$ .
  - Наконец,  $14 - 7 = 7$ .
- `['8', '9', '+', '1', '7', '-', '*']`:
  - Сначала  $8 + 9 = 17$ .



- Затем  $1 - 7 = -6$ .
- Наконец,  $17 * -6 = -102$ .

```
self.assertEqual(evaluate_postfix(['8', '9', '+', '1', '7', '-', '*']), -102)
```

### Метод `test_edge_cases`

Тестирует крайний случай, когда выражение состоит из одного числа:

- `['1']`: Результат 1.

```
self.assertEqual(evaluate_postfix(['1']), 1)
```

### 3. Запуск тестов

В блоке `if __name__ == '__main__'` вызывается `unittest.main()`, который запускает все тесты в классе.

```
if __name__ == '__main__':
    unittest.main()
```

## Задание 9 : Поликлиника

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.

- **Формат входного файла (input.txt).** В первой строке записано одно целое число  $n$  ( $1 \leq n \leq 10^5$ ) - число запросов к вашей программе. В следующих  $n$  строках заданы описания запросов в следующем формате:
  - «+  $i$ » – к очереди присоединяется пациент  $i$  ( $1 \leq i \leq N$ ) и встает в ее конец;
  - «\*  $i$ » – пациент  $i$  встает в середину очереди ( $1 \leq i \leq N$ );
  - «-» – первый пациент в очереди заходит к врачу. Гарантируется, что на момент каждого такого запроса очередь будет не пуста.
- **Формат выходного файла (output.txt).** Для каждого запроса третьего типа в отдельной строке выведите номер пациента, который должен зайти к шаманам.
- Ограничение по времени. Оцените время работы и используемую память при заданных максимальных значениях.
- Пример:

input.txt	output.txt	input.txt	output.txt
7	1	10	1
+ 1	2	+ 1	3
+ 2	3	+ 2	2
-		* 3	5
+ 3		-	4
+ 4		+ 4	
-		* 5	
-		-	
		-	
		-	
		-	

```

import os
from collections import deque

def process_queue():
    input_path = os.path.join '..', 'txtf', 'input.txt'
    output_path = os.path.join '..', 'txtf', 'output.txt'

    queue = deque()

    with open(output_path, 'w') as output_file:
        pass

    with open(input_path, 'r') as input_file:
        n = int(input_file.readline().strip())
        for _ in range(n):
            line = input_file.readline().strip()
            if not line:
                continue

            command = line.split()
            action = command[0]

            if action == '+':
                patient_id = int(command[1])
                queue.append(patient_id)
            elif action == '*':
                patient_id = int(command[1])

                mid_index = len(queue) // 2
                if len(queue) % 2 == 0:
                    mid_index -= 1
                queue.insert(mid_index + 1, patient_id)
            elif action == '-':
                with open(output_path, 'a') as output_file:
                    output_file.write(str(queue.popleft()) + '\n')

if __name__ == '__main__':
    process_queue()

```

\* Результат :

input.txt:

```

10
+ 1
+ 2
* 3
-
+ 4
* 5
-
-
-
-

```

output.txt:

## 1. Импорт библиотек

```
import os
from collections import deque
```

- **os**: используется для работы с файловой системой.
- **deque**: двусторонняя очередь из модуля `collections`, которая обеспечивает эффективное добавление и удаление элементов с обеих сторон.

## 2. Определение функции `process_queue`

```
def process_queue():
    input_path = os.path.join('..', 'txtf', 'input.txt')
    output_path = os.path.join('..', 'txtf', 'output.txt')
    queue = deque()
```

- **input\_path** и **output\_path**: пути к входному и выходному файлам.
- **queue**: создается пустая очередь для обработки пациентов.

## 3. Очистка выходного файла

```
with open(output_path, 'w') as output_file:
    pass
```

- Перед началом работы очищается файл `output.txt`, чтобы не было старых данных.

## 4. Открытие входного файла и обработка команд

```
with open(input_path, 'r') as input_file:
    n = int(input_file.readline().strip())
```

- Открывается входной файл, читается количество запросов **n**.

## Чтение и выполнение запросов

```
for _ in range(n):
    line = input_file.readline().strip()
    if not line:
        continue
    command = line.split()
    action = command[0]
```

- Каждая строка с запросом считывается.
- Запрос разбивается на **action** (тип действия) и дополнительные данные, если они есть.

## 5. Обработка команд

### Добавление в конец очереди

```
if action == '+':
    patient_id = int(command[1])
    queue.append(patient_id)
```

- Если команда + i, то пациент с идентификатором i добавляется в конец очереди с помощью `queue.append()`.

#### Добавление в середину очереди

```
elif action == '*':
    patient_id = int(command[1])
    mid_index = len(queue) // 2
    if len(queue) % 2 == 0:
        mid_index -= 1
    queue.insert(mid_index + 1, patient_id)
```

- Если команда \* i, пациент добавляется в середину очереди:
  1. `mid_index = len(queue) // 2`: вычисляется индекс середины.
  2. Если длина очереди чётная, `mid_index -= 1`, чтобы пациент вставал сразу за "центром".
  3. `queue.insert(mid_index + 1, patient_id)`: пациент вставляется в нужное место.

#### Удаление первого пациента

```
elif action == '-':
    with open(output_path, 'a') as output_file:
        output_file.write(str(queue.popleft()) +
```

'\n')

- Если команда -, первый пациент удаляется из очереди с помощью `queue.popleft()`.
- Его номер записывается в выходной файл `output.txt`.

#### 6. Главная часть программы

```
if __name__ == '__main__':
    process_queue()
```

- Запускается функция `process_queue`.

#### Unittest для задание 9:

```
import os
import unittest
from collections import deque

def process_queue(commands):
    queue = deque()
    output = []

    for command in commands:
        action = command[0]
        if action == '+':
            patient_id = command[1]
            queue.append(patient_id)
        elif action == '*':
            patient_id = command[1]
            mid_index = len(queue) // 2
            if len(queue) % 2 == 0:
                mid_index -= 1
            queue.insert(mid_index + 1, patient_id)
        elif action == '-':
            output.append(queue.popleft())
```



Функция `process_queue` принимает список команд и обрабатывает их с использованием очереди. Вот основные действия, которые она выполняет:

#### Аргументы:

- **commands:** список команд. Каждая команда представлена кортежем, где первый элемент — это действие ('+', '-', или '\*'), а второй элемент — значение (например, ID пациента).

#### Логика обработки:

##### 1. Создание очереди:

```
queue = deque()
output = []
```

Используется `deque` из модуля `collections` для работы с очередью.

##### 2. Обработка команд:

```
for command in commands:
    action = command[0]
```

Перебираются команды, и каждая команда обрабатывается в зависимости от значения `action`.

###### ◦ Добавление в конец очереди ('+');

```
if action == '+':
    patient_id = command[1]
    queue.append(patient_id)
```

Число добавляется в конец очереди с помощью метода `append`.

###### ◦ Добавление в середину ('\*');

```
elif action == '*':
    patient_id = command[1]
    mid_index = len(queue) // 2
    if len(queue) % 2 == 0:
        mid_index -= 1
    queue.insert(mid_index + 1, patient_id)
```

Число добавляется в середину очереди. Сначала вычисляется индекс середины:

- Для четной длины середина смещается левее (`mid_index -= 1`).
- Затем элемент вставляется после середины.

###### ◦ Удаление из начала очереди ('-');

```
elif action == '-':
    output.append(queue.popleft())
```

Удаление элемента из начала очереди с использованием `popleft`, а удалённое значение добавляется в список `output`.

##### 3. Результат:

```
return output
```

Возвращается список удалённых элементов.

## 2. Тесты

Класс `TestClinicQueue` содержит два теста для проверки корректности функции `process_queue`.

### Тест 1:

```
commands = [  
    ('+', 1),  
    ('+', 2),  
    ('-', None),  
    ('+', 3),  
    ('-', None),  
    ('-', None)  
]  
expected_output = [1, 2, 3]
```

Проверяет простые команды: добавление (+) и удаление (-).

### Тест 2:

```
commands = [  
    ('+', 1),  
    ('+', 2),  
    ('*', 3),  
    ('-', None),  
    ('+', 4),  
    ('*', 5),  
    ('-', None),  
    ('-', None),  
    ('-', None),  
    ('-', None)  
]  
expected_output = [1, 3, 2, 5, 4]
```

Проверяет все типы команд, включая вставку в середину ('\*').

### 3. Запуск программы

```
if __name__ == '__main__':  
    unittest.main()
```

Используется модуль unittest, который автоматически запускает тесты.