

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 5

Выполнил:
Артемов И. В.
К3141

Проверил:
Афанасьев А. В.

Санкт-Петербург
2024 г.

Содержание отчета

| | |
|---|----|
| Содержание отчета | 2 |
| Задачи по варианту | 3 |
| Задача №1. Стек | 3 |
| Задача №3. Скобочная последовательность. Версия 1 | 8 |
| Задача №5. Стек с максимумом | 14 |
| Задача №11. Бюрократия | 19 |
| Дополнительные задачи | 23 |
| Задача №8. Постфиксная запись | 23 |
| Задача №13. Реализация стека, очереди и связанных списков | 28 |
| Вывод | 34 |

Задачи по варианту

Задача №1. Стек

Текст задачи.

1 задача. 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 | |
| - | |

Листинг кода.

```
# Импортируем библиотеки для отслеживания памяти и времени выполнения программы
import tracemalloc
import time
from lab4.utils import *

# Запускаем таймер для измерения времени работы программы
t_start = time.perf_counter()

# Включаем отслеживание памяти
tracemalloc.start()

# Пути к входному и выходному файлам
current_dir = os.path.dirname(os.path.abspath( file )) # Директория
```

```

task1/src
txtf_dir = os.path.join(os.path.dirname(current_dir), "txtf") # Директория
task1/txtf
input_path = os.path.join(txtf_dir, "input.txt")
output_path = os.path.join(txtf_dir, "output.txt")

def process_stack(commands):
    """
    Обработывает команды работы со стеком.
    :param commands: Список строк, содержащих команды (+ N или -)
    :return: Список чисел, удаленных из стека
    """
    stack = []
    results = []

    for command in commands:
        if command.startswith("+"):
            _, number = command.split()
            stack.append(int(number))
        elif command == "-":
            results.append(stack.pop())

    return results

# Основной блок программы
if __name__ == "__main__":
    # Читаем данные из файла input.txt
    lines = open_file(input_path)

    # Удаляем символы новой строки у каждой команды
    commands = [cmd.strip() for cmd in lines[1:]] # Убираем \n

    # Проверка корректности входных данных
    if 1 <= len(commands) <= 10 ** 6 and all(
        (cmd.startswith("+") and len(cmd.split()) == 2 and
         abs(int(cmd.split()[1])) <= 10 ** 9) or cmd == "-" for
        cmd in commands
    ):
        print(f"\nTask 1\nInput:\n{len(commands)}\n{commands}")
        delete_prev_values(1)

        # Обработываем команды и записываем результат
        results = process_stack(commands)
        write_file("\n".join(map(str, results)), output_path)
        print_output_file(1)
    else:
        # Выводим сообщение об ошибке, если данные некорректны
        print("Введите корректные данные")

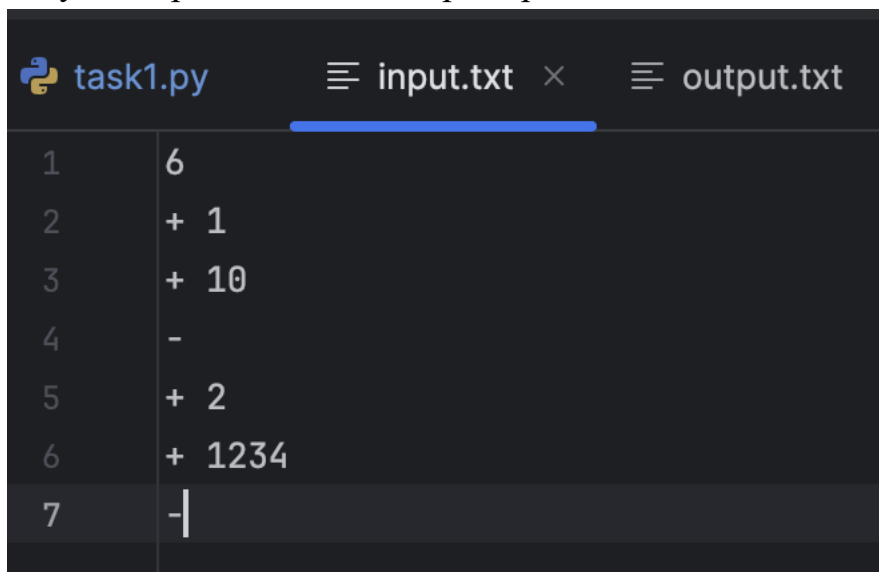
    # Выводим время работы программы
    print("Время работы: %s секунд" % (time.perf_counter() - t_start))
    # Выводим количество памяти, затраченной на выполнение программы
    print("Затрачено памяти:", tracemalloc.get_traced_memory()[1], "байт")

    # Останавливаем отслеживание памяти
    tracemalloc.stop()

```

Текстовое объяснение решения.

1. Импортируем библиотеки для отслеживания времени и памяти: ``tracemalloc`` и ``time``.
 2. Запускаем таймер для измерения времени работы программы.
 3. Включаем отслеживание памяти с помощью ``tracemalloc.start()``.
 4. Определяем пути к входным и выходным файлам.
 5. Создаём функцию ``process_stack``, которая обрабатывает команды для работы со стеком.
 6. В основном блоке программы открываем файл ``input.txt`` для чтения команд.
 7. Убираем символы новой строки у каждой команды.
 8. Проверяем корректность входных данных (длина команд и их формат).
 9. Если данные корректны, обрабатываем команды и записываем результат в ``output.txt``.
 10. Если данные некорректны, выводим сообщение об ошибке.
 11. Выводим время работы программы.
 12. Выводим количество памяти, затраченной на выполнение программы.
 13. Останавливаем отслеживание памяти с помощью ``tracemalloc.stop()``.
- Результат работы кода на примере из задачи:



```
task1.py  input.txt  output.txt
1      6
2      + 1
3      + 10
4      -
5      + 2
6      + 1234
7      -|
```

| task1.py | | input.txt | output.txt |
|----------|------|-----------|------------|
| 1 | 10 | | |
| 2 | 1234 | | |
| | | | |

Результат работы кода на максимальных и минимальных значениях:

| task1.py | | input.txt | output.txt |
|----------|-----|-----------|------------|
| 1 | 2 | | |
| 2 | + 1 | | |
| 3 | - | | |
| | | | |

| task1.py | | input.txt | output.txt |
|----------|---|-----------|------------|
| 1 | 1 | | |
| | | | |

```
task1.py  input.txt  output.txt
1 4
2 + 1000000000
3 + -1000000000
4 -
5 -
```

```
task1.py  input.txt  output.txt
1 -1000000000
2 1000000000
```

| | Время выполнения | Затраты памяти |
|--|------------------------------|----------------|
| Нижняя граница диапазона значений входных данных из текста задачи | 0.0005106249882373959 секунд | 15477 байт |
| Пример из задачи | 0.0005482909909915179 секунд | 15659 байт |
| Верхняя граница диапазона значений входных данных из текста задачи | 0.0005373329913709313 секунд | 1064396 байт |

Вывод по задаче: мною был изучен алгоритм реализации стека.

Задача №3. Скобочная последовательность. Версия 1

Текст задачи.

3 задача. Скобочная последовательность. Версия 1

Последовательность A , состоящую из символов из множества «(», «)», «[» и «]», 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 |
|)() | |

Листинг кода.

```
# Импортируем библиотеки для отслеживания памяти и времени выполнения
программы
import tracemalloc
import time
from lab4.utils import *

# Запускаем таймер для измерения времени работы программы
t_start = time.perf_counter()

# Включаем отслеживание памяти
tracemalloc.start()

current_dir = os.path.dirname(os.path.abspath(__file__)) # Директория
task/src
txtf_dir = os.path.join(os.path.dirname(current_dir), "txtf") # Директория
task/txtf
input_path = os.path.join(txtf_dir, "input.txt")

def is_valid_bracket_sequence(sequence):
    """
    Проверяет, является ли скобочная последовательность правильной.
    :param sequence: строка со скобочной последовательностью
    :return: True, если последовательность правильная, иначе False
    """
    stack = []
    matching_brackets = {'(': ')', '[': ']', '{': '}'

    for char in sequence:
        if char in "([{" : # Если открывающая скобка
            stack.append(char)
        elif char in ")]}" : # Если закрывающая скобка
            if stack and stack[-1] == matching_brackets[char]:
                stack.pop()
            else:
                return False # Если не соответствует, последовательность
# неправильная
    return not stack # Если стек пуст, последовательность правильная

# Основной блок программы
if __name__ == "__main__":
    # Читаем данные из файла input.txt с помощью функции open_file
    lines = open_file(input_path)
    n = int(lines[0]) # Количество строк
    sequences = lines[1:] # Последовательности для проверки

    # Проверка корректности входных данных
    if 1 <= n <= 500 and all(1 <= len(seq) <= 10**4 for seq in sequences):
        print(f"\nTask 3\nInput:\n{n}\n{sequences}")
        delete_prev_values(3)

        # Проверяем каждую последовательность
        results = []
        for seq in sequences:
            results.append("YES" if is_valid_bracket_sequence(seq) else
"NO")

        output_path = get_output_path(3)
        # Записываем результаты проверки в файл output.txt
        write_file("\n".join(results), output_path)
        print_output_file(3)
    else:
```

```

# Выводим сообщение об ошибке, если данные некорректны
print('Введите корректные данные')

# Выводим время работы программы
print("Время работы: %s секунд" % (time.perf_counter() - t_start))
# Выводим количество памяти, затраченной на выполнение программы
print("Затрачено памяти:", tracemalloc.get_traced_memory()[1], "байт")

# Останавливаем отслеживание памяти
tracemalloc.stop()

```

Текстовое объяснение решения.

1. Импортируем библиотеки для отслеживания времени и памяти: `tracemalloc` и `time`.
 2. Запускаем таймер для измерения времени работы программы.
 3. Включаем отслеживание памяти с помощью `tracemalloc.start()`.
 4. Определяем пути к директориям и входному файлу.
 5. Создаём функцию `is_valid_bracket_sequence`, которая проверяет правильность скобочной последовательности.
 6. В основном блоке программы открываем файл `input.txt` для чтения данных.
 7. Извлекаем количество строк и последовательности для проверки.
 8. Проверяем корректность входных данных (длина последовательностей и их количество).
 9. Если данные корректны, проверяем каждую последовательность на правильность.
 10. Если данные некорректны, выводим сообщение об ошибке.
 11. Записываем результаты проверки в файл `output.txt`.
 12. Выводим время работы программы.
 13. Выводим количество памяти, затраченной на выполнение программы.
 14. Останавливаем отслеживание памяти с помощью `tracemalloc.stop()`.
- Результат работы кода на примере из текста задачи:

```
task3.py  input.txt  output.txt
1  5
2  ()()
3  ([)]
4  ([)]
5  (([]
6  )()
```

```
task3.py  input.txt  output.txt
1  YES
2  YES
3  NO
4  NO
5  NO
```

Результат работы кода на максимальных и минимальных значениях:

```
task3.py  input.txt  output.txt  test_task3.py
1 1
2 (
3
```

```
task3.py  input.txt  output.txt  test_task3.py
1 NO
```

| | Время выполнения | Затраты памяти |
|---|------------------------------|----------------|
| Нижняя граница диапазона значений входных данных из текста задачи | 0.0006272080063354224 секунд | 15335 байт |
| Пример из задачи | 0.0005470830074045807 секунд | 15551 байт |
| Верхняя граница | 0.22705929099993227 | 430401 байт |

| | | |
|--|--------|--|
| диапазона значений входных данных из текста задачи | секунд | |
|--|--------|--|

Вывод по задаче: мною был изучен алгоритм определения правильности скобочной последовательности.

Задача №5. Стек с максимумом

Текст задачи.

5 задача. 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 содержит 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 | | | |

Листинг кода.

```
import tracemalloc
import time
from lab4.utils import *

# Запускаем таймер для измерения времени работы программы
t_start = time.perf_counter()

# Включаем отслеживание памяти
tracemalloc.start()

current_dir = os.path.dirname(os.path.abspath(__file__)) # Директория
task1/src
txtf_dir = os.path.join(os.path.dirname(current_dir), "txtf") # Директория
task1/txtf
input_path = os.path.join(txtf_dir, "input.txt")

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)
        else:
            self.max_stack.append(self.max_stack[-1])

    def pop(self):
        if self.stack:
            self.stack.pop()
            self.max_stack.pop()

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

# Основной блок программы
if __name__ == "__main__":
    # Читаем данные из файла input.txt с помощью функции open_file
    lines = open_file(input_path)
    n = int(lines[0].strip()) # Количество команд
    commands = [line.strip() for line in lines[1:]] # Список команд

    max_stack = MaxStack()
    output = []

    # Проверка корректности входных данных
    if 1 <= n <= 400000 and all(
        command.startswith("push") and 0 <= int(command.split()[1]) <=
100000 if command.startswith("push") else True for command in commands):
        print(f"\nTask 5\nInput:\n{n}\n\n{commands}")
        delete_prev_values(5)

        for command in commands:
            if command.startswith("push"):
                _, value = command.split()
                value = int(value)
                max_stack.push(value)
            elif command == "pop":
                max_stack.pop()
            elif command == "max":
```

```

        output.append(str(max_stack.max()))

    # Записываем результаты в файл output.txt
    output_path = get_output_path(5)
    write_file("\n".join(output), output_path)
    print_output_file(5)
else:
    print('Введите корректные данные')

# Выводим время работы программы
print("Время работы: %s секунд" % (time.perf_counter() - t_start))
# Выводим количество памяти, затраченной на выполнение программы
print("Затрачено памяти:", tracemalloc.get_traced_memory()[1], "байт")

# Останавливаем отслеживание памяти
tracemalloc.stop()

```

Текстовое объяснение решения.

1. Импортируем библиотеки для отслеживания времени и памяти: `tracemalloc` и `time`.
 2. Запускаем таймер для измерения времени работы программы.
 3. Включаем отслеживание памяти с помощью `tracemalloc.start()`.
 4. Определяем пути к директориям и входному файлу.
 5. Создаём класс `MaxStack`, который реализует стек с поддержкой операции получения максимального элемента.
 6. В классе `MaxStack` реализуем методы:
 - `push` для добавления элемента в стек и обновления максимального элемента.
 - `pop` для удаления элемента из стека.
 - `max` для получения текущего максимального элемента в стеке.
 7. В основном блоке программы открываем файл `input.txt` для чтения данных.
 8. Извлекаем количество команд и сам список команд.
 9. Проверяем корректность входных данных (формат команд и допустимые значения).
 10. Если данные корректны, обрабатываем команды:
 - Для `push` добавляем значение в стек.
 - Для `pop` удаляем элемент из стека.
 - Для `max` записываем текущий максимальный элемент.
 11. Если данные некорректны, выводим сообщение об ошибке.
 12. Записываем результаты выполнения команд в файл `output.txt`.
 13. Выводим время работы программы.
 14. Выводим количество памяти, затраченной на выполнение программы.
 15. Останавливаем отслеживание памяти с помощью `tracemalloc.stop()`.
- Результат работы кода на примере из задачи:


```
task5.py test_task5.py input.txt x output.txt
1 5
2 push 2
3 push 1
4 max
5 pop
6 max
```

```
task5.py test_task5.py input.txt output.txt x
1 2
2 2
```

| | Время выполнения | Затраты памяти |
|---|-------------------------------|----------------|
| Нижняя граница диапазона значений входных данных из текста задачи | 0.00027195800794288516 секунд | 18226 байт |
| Пример из задачи | 0.0007647079764865339 секунд | 18226 байт |
| Верхняя граница диапазона значений | 0.09492137498455122 секунд | 2127532 байт |

| | | |
|------------------------------------|--|--|
| ВХОДНЫХ ДАННЫХ ИЗ ТЕКСТА ЗАДАЧИ | | |
|------------------------------------|--|--|

Вывод по задаче: мною был изучен алгоритм реализации стека с максимумом.

Задача №11. Бюрократия

Текст задачи.

11 задача. Бюрократия

В министерстве бюрократии одно окно для приема граждан. Утром в очередь встают n человек, i -й посетитель хочет получить a_i справок. За один прием можно получить только одну справку, поэтому если после приема посетителю нужны еще справки, он встает в конец очереди. За время приема министерство успевает выдать m справок. Остальным придется ждать следующего приемного дня. Ваша задача - сказать, сколько еще справок хочет получить каждый из оставшихся в очереди посетитель в тот момент, когда прием закончится. Если все к этому моменту разойдутся, выведите -1.

- **Формат входного файла (input.txt).** В первой строке - количество посетителей n ($1 \leq n \leq 10^5$) и количество справок m ($0 \leq m \leq 10^9$). Во второй строке для каждого посетителя в порядке очереди указано количество справок a_i ($1 \leq a_i \leq 10^6$), которое он рассчитывает получить. Номером посетителя называется его место в исходной очереди.
- **Формат выходного файла (output.txt).** В первой строке выведите, сколько посетителей останется в очереди, когда прием закончится. Во второй строке выведите состояние очереди на тот момент, когда прием закончится: для всех посетителей по порядку выведите по одному числу через пробел - количество справок, которое он хочет еще получить. В случае, если в очереди никого не останется выведите одно число: -1
- Ограничение по времени. Оцените время работы и используемую память при заданных максимальных значениях.
- Пример:

| input1.txt | output1.txt | input2.txt | output2.txt |
|------------|-------------|------------|-------------|
| 3 2 | 2 | 4 5 | 3 |
| 1 2 3 | 3 1 | 2 5 2 3 | 4 1 2 |

Листинг кода.

```
# Импортируем библиотеки для отслеживания памяти и времени выполнения программы
import tracemalloc
import time
from collections import deque
from lab4.utils import *

# Запускаем таймер для измерения времени работы программы
t_start = time.perf_counter()

# Включаем отслеживание памяти
tracemalloc.start()

# Устанавливаем пути к входному и выходному файлам
current_dir = os.path.dirname(os.path.abspath(__file__)) # Директория task/src
txtf_dir = os.path.join(os.path.dirname(current_dir), "txtf") # Директория
```

```

task/txtf
input_path = os.path.join(txtf_dir, "input.txt")

def process_queue(n, m, a):
    """
    Функция вычисляет состояние очереди после завершения выдачи m справок.

    :param n: Количество посетителей в очереди.
    :param m: Общее количество справок, которое министерство может выдать.
    :param a: Список чисел, где a[i] – количество справок, которые нужны i-
му посетителю.
    :return: Количество оставшихся в очереди посетителей и их оставшиеся
справки.
    """
    queue = deque((i, a[i]) for i in range(n))
    remaining_docs = m # Сколько справок осталось выдать

    while queue and remaining_docs > 0:
        idx, docs_needed = queue.popleft()
        if docs_needed > 1:
            queue.append((idx, docs_needed - 1)) # Возвращаем в конец
очереди с уменьшенным числом справок
            remaining_docs -= 1

    if not queue:
        return -1, []
    remaining_people = len(queue)
    remaining_a = [docs for _, docs in queue]

    return remaining_people, remaining_a

# Основной блок программы
if __name__ == "__main__":
    # Читаем данные из файла input.txt с помощью функции open_file
    first_line, second_line = open_file(input_path)
    first_line = first_line.strip()
    n = int(first_line[0])
    m = int(first_line[2])
    a = []
    i = 0
    while len(a) != n:
        a.append(int(second_line[i]))
        i += 2

    # Проверка корректности входных данных
    if (1 <= n <= 10 ** 5) and (0 <= m <= 10 ** 9) and all(1 <= ai <= 10 **
6 for ai in a):
        print(f"\nTask 11\nInput:\n{n} {m}\n{a}")
        delete_prev_values(11)

        # Обрабатываем очередь
        remaining_people, remaining_a = process_queue(n, m, a)

        output_path = get_output_path(11)
        # Записываем результат в файл output.txt
        if remaining_people == -1:
            write_file("-1", output_path)
        else:
            write_file(f"{remaining_people}\n" + " ".join(map(str,
remaining_a)), output_path)

        print_output_file(11)

```

```

else:
    print("Введите корректные данные")

# Выводим время работы программы
print("Время работы: %s секунд" % (time.perf_counter() - t_start))
# Выводим количество памяти, затраченной на выполнение программы
print("Затрачено памяти:", tracemalloc.get_traced_memory()[1], "байт")

# Останавливаем отслеживание памяти
tracemalloc.stop()

```

Текстовое объяснение решения.

1. Импортируем библиотеки для отслеживания времени и памяти: `tracemalloc`, `time` и `deque` из `collections`.
2. Запускаем таймер для измерения времени работы программы.
3. Включаем отслеживание памяти с помощью `tracemalloc.start()`.
4. Определяем пути к директориям и входному файлу.
5. Создаём функцию `process_queue`, которая:
 - Принимает количество посетителей `n`, количество справок `m` и список чисел `a`, где каждое число — это количество справок, нужных i-му посетителю.
 - Использует очередь (`deque`), чтобы обрабатывать выдачу справок.
 - На каждом шаге удаляет одного посетителя, выдаёт ему справку, а если ему нужно больше одной справки, возвращает его в очередь с уменьшенным числом справок.
 - Возвращает количество оставшихся посетителей и их оставшиеся справки, или -1, если все посетители получили нужные справки.
6. В основном блоке программы открываем файл `input.txt` для чтения данных.
7. Извлекаем количество посетителей `n`, количество справок `m` и список необходимых справок `a` для каждого посетителя.
8. Проверяем корректность входных данных (ограничения на n, m и элементы списка).
9. Если данные корректны, обрабатываем очередь с помощью функции `process_queue`.
10. Записываем результат в файл `output.txt`: если очередь пуста, записываем "-1", иначе — количество оставшихся посетителей и их оставшиеся справки.
11. Если данные некорректны, выводим сообщение об ошибке.
12. Выводим время работы программы.
13. Выводим количество памяти, затраченной на выполнение программы.
14. Останавливаем отслеживание памяти с помощью `tracemalloc.stop()`.

Результат работы кода на примере из текста задачи:

The first screenshot shows the 'input.txt' file with the following content:

```
1 3 2
2 1 2 3
```

The second screenshot shows the 'output.txt' file with the following content:

```
1 2
2 3 1
```

Результат работы кода на максимальных и минимальных значениях:

The first screenshot shows the 'input.txt' file with the following content:

```
1 1 0
2 1
```

The second screenshot shows the 'output.txt' file with the following content:

```
1 1
2 1
```

| | Время выполнения | Затраты памяти |
|--|------------------------------|----------------|
| Нижняя граница диапазона значений входных данных из текста задачи | 0.0004885839880444109 секунд | 15329 байт |
| Пример из задачи | 0.001447082991944626 секунд | 15226 байт |
| Верхняя граница диапазона значений входных данных из текста задачи | 30.278248833999896 секунд | 15810163 байт |

Вывод по задаче: мною был изучен алгоритм того, сколько человек останется в очереди, когда приём закончится.

Дополнительные задачи

Задача №8. Постфиксная запись

Текст задачи.

8 задача. 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 - * | |

Листинг кода.

```
# Импортируем библиотеки для отслеживания памяти и времени выполнения
программы
import tracemalloc
import time
from lab4.utils import *

# Запускаем таймер для измерения времени работы программы
t_start = time.perf_counter()

# Включаем отслеживание памяти
tracemalloc.start()

# Устанавливаем пути для файлов
current_dir = os.path.dirname(os.path.abspath(__file__)) # Директория
task1/src
txtf_dir = os.path.join(os.path.dirname(current_dir), "txtf") # Директория
task1/txtf
input_path = os.path.join(txtf_dir, "input.txt")
```

```

# Функция для вычисления значения выражения в постфиксной записи
def evaluate_postfix(expression):
    """
    Вычисляет значение выражения в постфиксной записи.

    :param expression: Список строк, представляющих постфиксное выражение.
    :return: Результат вычисления.
    """
    stack = [] # Инициализируем стек для промежуточных значений

    for token in expression:
        if token.isdigit():
            # Если токен - число, помещаем его в стек
            num = int(token)
            if abs(num) >= 2 ** 31:
                raise ValueError("Найдено число, выходящее за пределы
|2^31|")
            stack.append(num)
        else:
            # Если токен - оператор, выполняем соответствующую операцию
            b = stack.pop() # Второй операнд
            a = stack.pop() # Первый операнд

            # Гарантируем, что промежуточные результаты также находятся в
пределах |2^31|
            if token == '+':
                result = a + b
            elif token == '-':
                result = a - b
            elif token == '*':
                result = a * b
            else:
                raise ValueError(f"Неизвестный оператор: {token}")

            if abs(result) >= 2 ** 31:
                raise ValueError("Промежуточный результат превышает предел
|2^31|")
            stack.append(result)

    # Результат вычислений остаётся единственным элементом в стеке
    return stack.pop()

# Основной блок программы
if __name__ == "__main__":
    # Читаем данные из файла input.txt с помощью функции open_file
    lines = open_file(input_path)
    n = int(lines[0].strip()) # Количество элементов в выражении
    expr = lines[1].strip().split() # Постфиксное выражение как список
    строк

    # Проверка корректности входных данных
    if 1 <= n <= 10 ** 6 and len(expr) == n:
        print(f"\nTask 8\nInput:\n{n}\n{' '.join(expr)}")
        delete_prev_values(8)

    # Вычисляем значение выражения
    try:
        result = evaluate_postfix(expr)
    except ValueError as e:
        print(f"Ошибка: {e}")
        result = None

```



```

    if result is not None:
        # Записываем результат в файл output.txt
        output_path = get_output_path(8)
        write_file(str(result), output_path)
        print_output_file(8)
    else:
        # Выводим сообщение об ошибке, если данные некорректны
        print('Введите корректные данные')

# Выводим время работы программы
print("Время работы: %s секунд" % (time.perf_counter() - t_start))
# Выводим количество памяти, затраченной на выполнение программы
print("Затрачено памяти:", tracemalloc.get_traced_memory()[1], "байт")

# Останавливаем отслеживание памяти
tracemalloc.stop()

```

Текстовое объяснение решения.

1. Импортируем библиотеки для отслеживания времени и памяти: `tracemalloc` и `time`.
2. Запускаем таймер для измерения времени работы программы.
3. Включаем отслеживание памяти с помощью `tracemalloc.start()`.
4. Определяем пути к директориям и входному файлу.
5. Создаём функцию `evaluate_postfix`, которая:
 - Принимает список строк (постфиксное выражение).
 - Использует стек для вычисления выражения.
 - Для чисел добавляет их в стек.
 - Для операторов выполняет соответствующие операции с последними двумя значениями из стека.
 - Проверяет, чтобы результаты и числа не выходили за пределы $|2^{31}|$.
 - Возвращает результат вычисления выражения.
6. В основном блоке программы открываем файл `input.txt` для чтения данных.
7. Извлекаем количество элементов в выражении и само постфиксное выражение.
8. Проверяем корректность входных данных (число элементов и длина выражения).
9. Если данные корректны, вычисляем значение выражения с помощью функции `evaluate_postfix`.
10. Если при вычислении возникает ошибка, выводим сообщение об ошибке.
11. Записываем результат вычисления в файл `output.txt`, если результат корректен.
12. Если данные некорректны, выводим сообщение об ошибке.
13. Выводим время работы программы.

14. Выводим количество памяти, затраченной на выполнение программы.

15. Останавливаем отслеживание памяти с помощью `tracemalloc.stop()`.

Результат работы кода на примере из текста задачи:

```
task8.py test_task8.py input.txt x output.txt
1 7
2 8 9 + 1 7 - *
```

```
task8.py test_task8.py input.txt output.txt x
1 -102
```

Результат работы кода на максимальных и минимальных значениях:

```
task8.py test_task8.py input.txt x output.txt
1 5
2 0 0 + 0 *
```

```
task8.py test_task8.py input.txt output.txt x
1 0
```

| | Время выполнения | Затраты памяти |
|---|-------------------------------|----------------|
| Нижняя граница диапазона значений входных данных из текста задачи | 0.00047524998080916703 секунд | 15374 байт |

| | | |
|---|---------------------------------|--------------|
| Пример из задачи | 0.0006085000059101731 секунд | 15378 байт |
| Верхняя граница диапазона значений входных данных из текста задачи | 0.22705929099993227 секунд | 1064396 байт |

Вывод по задаче: мною был изучен алгоритм нахождения значения выражения, исходя из построенной постфиксной записи.

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 13_1.

```
import tracemalloc
import time
# Запускаем таймер для измерения времени работы программы
t_start = time.perf_counter()

# Включаем отслеживание памяти
tracemalloc.start()

class Node:
    """Класс для создания узлов связанного списка."""
    def __init__(self, data):
        self.data = data # Данные, которые хранит узел
        self.next = None # Ссылка на следующий узел (изначально None)

class Stack:
    """Класс для стека на основе связанного списка."""
    def __init__(self):
        self.top = None # В стеке нет элементов, top указывает на None

    def isEmpty(self):
        """Проверка, пуст ли стек."""
        return self.top is None # Если top равен None, стек пуст

    def push(self, data):
        """Добавление элемента в стек."""
        new_node = Node(data) # Создаем новый узел с переданными данными
        new_node.next = self.top # Новый узел указывает на текущий top
        self.top = new_node # Теперь новый узел стал верхом стека

    def pop(self):
        """Удаление элемента из стека."""
        if self.isEmpty():
            print("Стек пуст. Невозможно удалить элемент.")
            return None # Если стек пуст, возвращаем None
        popped_node = self.top # Сохраняем верхний узел
        self.top = self.top.next # Теперь top указывает на следующий узел
        return popped_node.data # Возвращаем данные удаленного элемента

    def print_stack(self):
        """Вывод элементов стека."""
        current = self.top # Начинаем с верхнего элемента
        if self.isEmpty():
            print("Стек пуст.")
        else:
            while current: # Пока текущий элемент не равен None
                print(current.data, end=" -> ") # Выводим данные текущего
                current = current.next # Переходим к следующему элементу
            print("None") # Конец стека
```

узла

```

print("\nTask 13_1")
if __name__ == "__main__":
    # Пример использования стека
    stack = Stack() # Создаем пустой стек
    stack.push(10)  # Добавляем элемент 10
    stack.push(20)  # Добавляем элемент 20
    stack.push(30)  # Добавляем элемент 30

    print("Стек после добавления элементов:")
    stack.print_stack() # Выводим стек

    print("\nУдаленный элемент:", stack.pop()) # Удаляем верхний элемент
(30)
    print("Стек после удаления элемента:")
    stack.print_stack() # Выводим стек после удаления элемента
    # Выводим время работы программы
    print("Время работы: %s секунд" % (time.perf_counter() - t_start))
    # Выводим количество памяти, затраченной на выполнение программы
    print("Затрачено памяти:", tracemalloc.get_traced_memory()[1], "байт")

    # Останавливаем отслеживание памяти
    tracemalloc.stop()

```

Текстовое объяснение решения.

1. Импортируем библиотеки для отслеживания времени и памяти: `tracemalloc` для отслеживания памяти и `time` для измерения времени работы программы.
2. Запускаем таймер с помощью `time.perf_counter()` для вычисления времени выполнения программы.
3. Включаем отслеживание памяти с помощью `tracemalloc.start()`.
4. Создаём класс `Node`:
 - Этот класс представляет узел связного списка.
 - Каждый узел хранит данные (`data`) и ссылку на следующий узел (`next`), которая изначально равна `None`.
5. Создаём класс `Stack`:
 - Этот класс представляет стек, реализованный с использованием связного списка.
 - В нем определены несколько методов:
 - `isEmpty()`: Проверяет, пуст ли стек (если `top` равен `None`, то стек пуст).
 - `push(data)`: Добавляет новый элемент в стек. Создается новый узел, который указывает на текущий верхний элемент стека, а `top` обновляется на новый узел.
 - `pop()`: Удаляет верхний элемент стека. Если стек пуст, выводится сообщение об ошибке.
 - `print_stack()`: Выводит все элементы стека, начиная с верхнего. Если стек пуст, выводится сообщение "Стек пуст".

6. Основной блок программы:

- Создаем объект стека `stack`.
- Добавляем в стек элементы 10, 20 и 30 с помощью метода `push`.
- После добавления элементов выводим текущий состав стека с помощью метода `print_stack`.
- Удаляем верхний элемент (30) с помощью метода `pop` и выводим результат.

7. Выводим время работы программы с помощью `time.perf_counter()`.

8. Выводим количество памяти, использованной программой, с помощью `tracemalloc.get_traced_memory()[1]`.

9. Останавливаем отслеживание памяти с помощью `tracemalloc.stop()`.

Результат работы кода:

```
Task 13_1
Стек после добавления элементов:
30 -> 20 -> 10 -> None

Удаленный элемент: 30
Стек после удаления элемента:
20 -> 10 -> None
Время работы: 0.00012904198956675828 секунд
Затрачено памяти: 7163 байт
```

Листинг кода 13_2.

```
import tracemalloc
import time
# Запускаем таймер для измерения времени работы программы
t_start = time.perf_counter()

# Включаем отслеживание памяти
tracemalloc.start()
class Node:
    """
    Класс для узла связного списка.
    Каждый узел будет хранить значение и ссылку на следующий узел.
    """

    def __init__(self, value):
        self.value = value # Значение в узле
        self.next = None # Ссылка на следующий узел

class Queue:
    """
```

```

Класс для очереди, реализованной на основе связного списка.
"""

def __init__(self, max_size=None):
    self.front = None # Указатель на начало очереди (первый элемент)
    self.rear = None # Указатель на конец очереди (последний элемент)
    self.size = 0 # Текущий размер очереди
    self.max_size = max_size # Максимальный размер очереди (None -
неограниченная очередь)

def is_empty(self):
    """
    Проверка на пустоту очереди.
    """
    return self.size == 0

def is_full(self):
    """
    Проверка на переполнение очереди.
    Возвращает True, если очередь переполнена.
    """
    if self.max_size is None:
        return False # Если max_size не задан, очередь не может
переполниться
    return self.size == self.max_size

def enqueue(self, value):
    """
    Добавление элемента в очередь (в конец очереди).
    Если очередь переполнена, выводится ошибка.
    """
    if self.is_full():
        raise OverflowError("Очередь переполнена. Невозможно добавить
элемент.")

    new_node = Node(value)

    if self.is_empty():
        self.front = self.rear = new_node # Если очередь пустая, новый
узел будет и первым, и последним
    else:
        self.rear.next = new_node # Добавляем новый элемент в конец
        self.rear = new_node # Обновляем указатель на конец очереди

    self.size += 1 # Увеличиваем размер очереди

def dequeue(self):
    """
    Удаление элемента из очереди (с начала очереди).
    Если очередь пуста, выводится ошибка.
    """
    if self.is_empty():
        raise IndexError("Очередь пуста. Невозможно удалить элемент.")

    dequeued_value = self.front.value # Сохраняем значение, которое
будем удалять
    self.front = self.front.next # Сдвигаем указатель на начало
очереди

    if self.front is None: # Если после удаления очередь пуста,
обновляем указатель на конец
        self.rear = None

    self.size -= 1 # Уменьшаем размер очереди

```

```

        return dequeued_value # Возвращаем значение удалённого элемента

    def peek(self):
        """
        Возвращает элемент, который сейчас в начале очереди, но не удаляет
        его.

        Если очередь пуста, выводится ошибка.
        """
        if self.is_empty():
            raise IndexError("Очередь пуста. Нет элементов для просмотра.")
        return self.front.value

print("\nTask 13_2")
# Пример использования очереди
if __name__ == "__main__":
    queue = Queue(max_size=3) # Создаём очередь с максимальным размером 3

    try:
        queue.enqueue(10)
        queue.enqueue(20)
        queue.enqueue(30)

        print(f"Первый элемент в очереди: {queue.peek()}") # Просмотр
        первого элемента (10)
        print(f"Удалён элемент: {queue.dequeue()}") # Удаляем 10
        print(f"Первый элемент в очереди после удаления: {queue.peek()}")
        # Просмотр первого элемента (20)

        queue.enqueue(40) # Добавляем 40
        print(f"Первый элемент в очереди после добавления: {queue.peek()}")
        # Просмотр первого элемента (20)

        queue.enqueue(50) # Попытка добавить 50 в переполненную очередь
        # Выводим время работы программы
        print("Время работы: %s секунд" % (time.perf_counter() - t_start))
        # Выводим количество памяти, затраченной на выполнение программы
        print("Затрачено памяти:", tracemalloc.get_traced_memory()[1],
        "байт")

        # Останавливаем отслеживание памяти
        tracemalloc.stop()
    except Exception as e:
        print(f"Ошибка: {e}")
        # Выводим время работы программы
        print("Время работы: %s секунд" % (time.perf_counter() - t_start))
        # Выводим количество памяти, затраченной на выполнение программы
        print("Затрачено памяти:", tracemalloc.get_traced_memory()[1],
        "байт")

    # Останавливаем отслеживание памяти
    tracemalloc.stop()

```

Текстовое объяснение решения.

1 Импортируем библиотеки:

- `tracemalloc` используется для отслеживания памяти, а `time` — для измерения времени работы программы.

2 Запускаем таймер с помощью `time.perf_counter()` для вычисления времени работы программы.

3 Включаем отслеживание памяти с помощью `tracemalloc.start()`.

4 Создаём класс Node:

- Этот класс представляет узел в связном списке.
- Каждый узел хранит значение (value) и ссылку на следующий узел (next), которая изначально равна None.

5 Создаём класс Queue:

- Это очередь, реализованная на основе связного списка.
- В классе определены следующие методы:
 - `is_empty()`: Проверяет, пуста ли очередь. Возвращает True, если размер очереди равен нулю.
 - `is_full()`: Проверяет, заполнена ли очередь. Если задан максимальный размер очереди (`max_size`), то проверяется, достигла ли очередь этого размера.
 - `enqueue(value)`: Добавляет новый элемент в конец очереди. Если очередь переполнена, выбрасывает ошибку `OverflowError`.
 - `dequeue()`: Удаляет элемент с начала очереди. Если очередь пуста, выбрасывает ошибку `IndexError`.
 - `peek()`: Возвращает элемент в начале очереди, но не удаляет его. Если очередь пуста, выбрасывает ошибку `IndexError`.

Результат работы кода

```
Task 13_2
Первый элемент в очереди: 10
Удалён элемент: 10
Первый элемент в очереди после удаления: 20
Первый элемент в очереди после добавления: 20
Ошибка: Очередь переполнена. Невозможно добавить элемент.
Время работы: 0.0001228329783771187 секунд
Затрачено памяти: 8587 байт
```

Вывод по задаче: мною был изучен алгоритм реализации стека на основе связного списка с функциями `isEmpty`, `push`, `pop` и вывода данных и реализации очереди на основе связного списка функциями `Enqueue`, `Dequeue` с проверкой на переполнение и опустошения очереди.

Вывод

В ходе лабораторной работы были изучены стеки, очереди и связанные списки.