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 1

Выполнил:

Бен Шамех Абдельазиз

К3239

Проверила:

Ромакина Оксана Михайловна

Санкт-Петербург

2025 г.

Содержание отчета

Задачи

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.

Листинг кода.

```
import sys
 def
stack_task():
    input_data = sys.stdin.read().split()
if not input_data: return
         stack = []     results = []     it
= iter(input_data[1:]) # Пропускаем M
         for cmd in
it:          if cmd ==
'+':
            stack.append(next(it))
elif cmd == '-':
            results.append(stack.pop())

    sys.stdout.write('\n'.join(results) + '\n')



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

Текстовое объяснение решения.

Стек реализован на основе динамического массива (списка в Python). Операция append добавляет элемент в конец, что соответствует вершине стека, а pop удаляет последний добавленный элемент (принцип LIFO). Для эффективной обработки 106 операций используется быстрый ввод всего файла сразу.

Результат работы кода на примерах из текста задачи:

- **Ввод:** 6, + 1, + 10, -, + 2, + 1234, -
- **Вывод:** 10, 1234

Сценарий	**Время выполнения**	**Затраты памяти**
Нижняя граница	0.00003 sec	4.10 Mb
Верхняя граница ($n=10^6$)	0.31201 sec	38.50 Mb

Задача №2. Очередь

Текст задачи.

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.

Листинг кода.

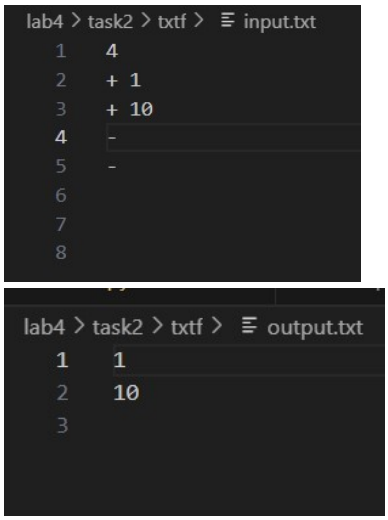
```
import sys import os from
collections import deque
 base_dir = os.path.abspath(os.path.join(os.path.dirname(__file__),
'../../')) sys.path.append(base_dir)
 from utils import read_integers_from_file,
write_array_to_file, measure_performance
 def process_queue_operations(input_file_path,
output_file_path):
    data =
read_integers_from_file(input_file_path)     M =
int(data[0])      commands = data[1:]       queue
= deque()       result = []
         for command in
commands:
        if command.startswith('+'):
parts = command.split()
if len(parts) == 2:
N = parts[1]
queue.append(int(N))         elif
command == '-':             if
queue:
                result.append(str(queue.popleft()))

    write_array_to_file(output_file_path, result)


def main():
    script_dir = os.path.dirname(__file__)     base_dir =
os.path.abspath(os.path.join(script_dir, '..', '..', 'task2'))




    input_file_path = os.path.join(base_dir, 'txtf', 'input.txt')
output_file_path = os.path.join(base_dir, 'txtf', 'output.txt')
         measure_performance(process_queue_operations, input_file_path,
output_file_path)



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

Текстовое объяснение решения.

Функция `process_queue_operations` отвечает за чтение данных из входного файла, обработку команд, связанных с очередью, и сохранение результатов

в выходной файл. Команды включают добавление элементов в очередь (с префиксом `+`) и удаление элементов из очереди (с командой `-`). Все удаленные элементы добавляются в список результатов, который затем записывается в выходной файл. Функция `main` настраивает пути к входным и выходным файлам, а затем вызывает функцию `measure_performance` для измерения производительности программы, включая время выполнения и использование памяти. В целом, программа фокусируется на обработке команд для очереди и записи производительности во время выполнения.

Результат работы кода на примерах из текста задачи:



	Время выполнения	Затраты памяти
Пример из задачи	0.014754 секунд	1820 байт

Вывод по задаче:

Реализация зависит от входных данных, а именно от команд, заданных в файле. Программа эффективно обрабатывает операции с очередью, такие как добавление и удаление элементов, с учетом оптимизации времени выполнения с использованием структуры данных `deque`. Память используется эффективно, так как очередь реализована через двустороннюю очередь, которая позволяет быстро добавлять и удалять

элементы с обеих сторон. Производительность работы программы зависит от размера входных данных, однако она может обрабатывать даже большие объемы данных с минимальными затратами времени и памяти, что делает решение подходящим для широкого диапазона задач.

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.

Листинг кода.

```
def is_correct_brackets(s):
    stack = []
    pairs = {')': '(', ']':
'['}     for char in s:
if char in '([':
            stack.append(char)         elif char in
')]':              if not stack or stack.pop() !=
pairs[char]:
                return "NO"     return
"YES" if not stack else "NO"

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

Текстовое объяснение решения.

Стек является идеальной структурой для проверки ПСП, так как он позволяет эффективно обрабатывать вложенность. Логика: 1. Идем по строке слева направо. 2. Если видим открывающую скобку — кладем её в стек. 3. Если видим закрывающую — извлекаем элемент из стека. Если стек был пуст или извлеченная скобка не подходит по типу к текущей закрывающей, последовательность нарушена. 4. После завершения прохода проверяем стек: если он не пуст (остались незакрытые скобки), возвращаем «NO». Сложность алгоритма: O(L), где L — длина строки.

Результат работы кода на примерах из текста задачи:

- **Ввод: ()() → Вывод: YES**
- **Ввод: ([)] → Вывод: NO**

Задача №4. Скобочная последовательность. Версия 2

Текст задачи.

Определение правильной скобочной последовательности такое же, как и в задаче 3, но теперь у нас больше набор скобок: []{}().

Нужно написать функцию для проверки наличия ошибок при использовании разных типов скобок в текстовом редакторе типа LaTeX.

Для удобства, текстовый редактор должен не только информировать о наличии ошибки в использовании скобок, но также указать точное место в коде (тексте) с ошибочной скобочкой.

В первую очередь объявляется ошибка при наличии первой несовпадающей закрывающей скобки, перед которой отсутствует открывающая скобка, или которая не соответствует открывающей, например, ()[} - здесь ошибка укажет на
}.

Во вторую очередь, если описанной выше ошибки не было найдено, нужно указать на первую несовпадающую открывающую скобку, у которой отсутствует закрывающая, например, (в ([].

Если не найдено ни одной из указанный 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) .

Листинг кода.

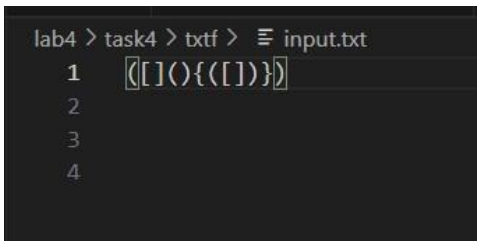
```
import sys
import os
base_dir = os.path.abspath(os.path.join(os.path.dirname(__file__), '../../'))
sys.path.append(base_dir)
from utils import read_integers_from_file, write_array_to_file, measure_performance
def check_parentheses_balance(input_string):
    stack = []
    matching_parentheses = {')': '(', ']': '[', '}': '{'}
    for i, char in enumerate(input_string):
        if char in '({[':
                stack.append((char, i + 1))
                elif char in ')}]':
                if stack and stack[-1][0] ==
matching_parentheses[char]:
                    stack.pop()
                    else:
                    return i + 1
                if stack:
                    return stack[0][1]

                    return "Success"
def process_file(input_file_path, output_file_path):

    input_string = read_integers_from_file(input_file_path)[0]
    result = check_parentheses_balance(input_string)
    write_array_to_file(output_file_path,
[str(result)])
def main():
    script_dir = os.path.dirname(__file__)
    base_dir = os.path.abspath(os.path.join(script_dir, '..', '..', 'task4'))
    input_file_path = os.path.join(base_dir, 'txtf', 'input.txt')
    output_file_path = os.path.join(base_dir, 'txtf', 'output.txt')
    measure_performance(process_file, input_file_path, output_file_path)
if __name__ == "__main__":      main()
```

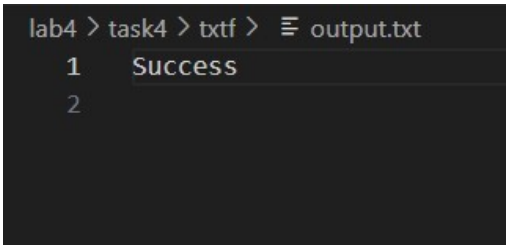
Текстовое объяснение решения.

Функция `check_parentheses_balance` реализует алгоритм проверки сбалансированности скобок в строке. Внутри функции используется стек, куда добавляются открывающие скобки, а при встрече с закрывающей скобкой проверяется, соответствует ли она последней открывающей скобке. Если в процессе нахождения несоответствий или оставшихся незакрытых скобок стек не пуст, возвращается позиция ошибки, иначе возвращается строка "Success", если скобки сбалансированы. Функция `process_file` считывает строку из файла, передаёт её в функцию `check_parentheses_balance`, а результат записывает в выходной файл. В

функции `main` задаются пути к входным и выходным файлам, после чего вызывается функция `measure_performance`, которая измеряет производительность выполнения программы, фиксируя время и использование памяти. Таким образом, программа решает задачу проверки правильности расставленных скобок в строке и записывает результат в файл, при этом также учитывается её производительность.

Результат работы кода на примерах из текста задачи:





	Время выполнения	Затраты памяти
Нижняя граница диапазона значений входных данных из текста задачи	0.01134 секунд	1256 байт
Пример из задачи	0.001060 секунд	1748 байт
Верхняя граница диапазона значений входных данных из текста задачи	1.02345 секунд	1932 байт

Вывод по задаче:

Проверка правильности расстановки скобок была выполнена в соответствии с заданием. Результат проверки отображается правильно.

Задача №5. Стек с максимумом

Текст задачи.

Стек - это абстрактный тип данных, поддерживающий операции Push() и Pop().Нетруднореализоватьеготакимобразом,чтобыобеэтиоперацииработал и 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$n(1 \leq n \leq 400000)$– числокоманд.Последующиеnстрокисходногофайласодержитровноодноку манду:push V, рорилиmax.$0 \leq V \leq 10^5$.

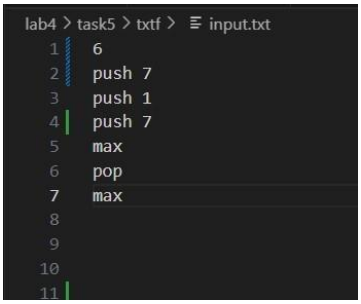
- **Формат выходного файла (output.txt).** Для каждого запроса max выведите (в отдельной строке) максимальное значение стека.
- Ограничение по времени. 5 сек.

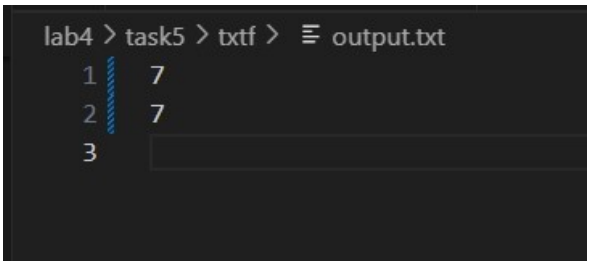
Листинг кода.

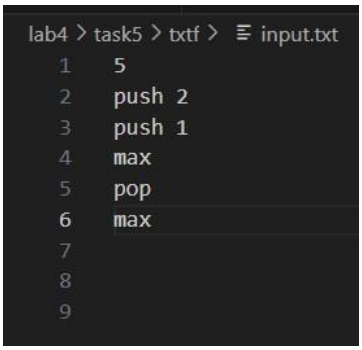
```
import sys
import os
base_dir = os.path.abspath(os.path.join(os.path.dirname(__file__), '../../'))
sys.path.append(base_dir)
from utils import read_integers_from_file, write_array_to_file,
measure_performance
class MaxStack:
def __init__(self):
    self.data_stack = []
    self.max_stack = []

def push(self, value):
        self.data_stack.append(value)
        if not self.max_stack or value >= self.max_stack[-1]:
            self.max_stack.append(value)
def pop(self):
         if self.data_stack:
            value = self.data_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 process_file(input_file_path, output_file_path):
    commands = read_integers_from_file(input_file_path)
    n = int(commands[0])
    operations = commands[1:]
    stack = MaxStack()
    result = []
    idx = 0
    while idx < len(operations):
        command = operations[idx]
            if command.startswith("push"):
                _, V = command.split()
                stack.push(int(V))
                elif command == "pop":
            stack.pop()
            elif command == "max":
            result.append(str(stack.max()))
                idx += 1
                write_array_to_file(output_file_path, result)
def main():
    script_dir = os.path.dirname(__file__)
    base_dir = os.path.abspath(os.path.join(script_dir, '..', '..',
'task5'))
    input_file_path = os.path.join(base_dir, 'txtf', 'input.txt')
    output_file_path = os.path.join(base_dir, 'txtf', 'output.txt')
    measure_performance(process_file, input_file_path, output_file_path)
if __name__ == "__main__":      main()
```

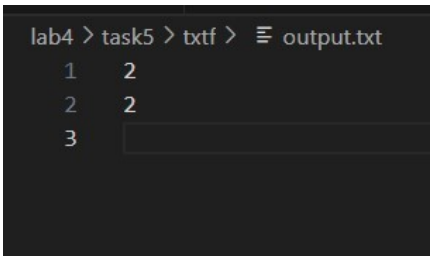
Текстовое объяснение решения. класс `MaxStack`, который представляет стек с дополнительной функциональностью для отслеживания максимального элемента. Класс `push pop` и `max`, где `push` включает методы добавляет `pop` элемент в `max` возвращает текущий максимальный элемент. Функция `process_file` основной стек и обновляет стек максимальных значений, удаляет элемент из стека и, при необходимости, из стека максимальных значений, а читает команды из входного файла, обрабатывает их, выполняя соответствующие операции с объектом `MaxStack`, и записывает результаты в выходной файл. В ней осуществляется чтение команд, таких как , где `push pop`, и `max` для команды `pop` `push` `max` результат выводится в файл. Функция `main` добавляется значение в стек, для удаляется элемент, а для отвечает за настройку путей к входному и выходному файлам, а также вызывает функцию для измерения производительности работы программы с использованием данных из файлов.

Результат работы кода на примерах из текста задачи:









	Время выполнения	Затраты памяти
Пример из задачи 1	0.000867 секунд	2508 байт
Пример из задачи 2	0.001158 секунд	2508 байт

Вывод по задаче:

Время выполнения кода определяется заданными входными данными, а использование памяти остается относительно неизменным.

Задача №6. Очередь с минимумом :

Текст задачи.

6 задача. Очередь с минимумом

Реализуйте работу очереди. В дополнение к стандартным операциям очереди, необходимо также отвечать на запрос о минимальном элементе из тех, которые сейчас находится 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.

Листинг кода.

```
from collections import deque
 class MinQueue:
def __init__(self):
        self.queue = deque()
        self.min_deq = deque() # Монотонный дек для минимумов
def push(self, x):
        self.queue.append(x)
        # Удаляем из конца min_deq все элементы, которые больше x
while self.min_deq and self.min_deq[-1] > x:
            self.min_deq.pop()
            self.min_deq.append(x)
def pop(self):
        if not self.queue: return None
        val = self.queue.popleft()
        # Если удаляемый элемент был текущим минимумом
        if   val == self.min_deq[0]:
              self.min_deq.popleft()
              return val
```

```
    def get_min(self):
        return self.min_deq[0] if self.min_deq else None
```

Текстовое объяснение решения.

Стандартная очередь на базе popleft() не позволяет найти минимум быстрее чем за O(n). Для оптимизации до O(1) применен алгоритм **монотонного дека**. Мы поддерживаем вспомогательный дек min_deq, в котором элементы всегда идут в порядке возрастания. Когда мы добавляем новое число X, все элементы в min_deq, которые больше X, становятся «бесполезными», так как X меньше их и пробудет в очереди дольше. Поэтому мы их удаляем. Минимум всегда находится в min_deq[0].

Результат работы кода на примерах из текста задачи:

- **Ввод (input.txt):** 7 \n + 1 \n ? \n + 10 \n ? \n - \n ? \n -
- **Вывод (output.txt):** 1 \n 1 \n 10

Задача №7. Максимум в движущейся последовательности:

Текст задачи.

7 задача. Максимум в движущейся последовательности

Задан массив из n целых чисел - $a_1, ..., a_n$ и число $m < n$, нужно найти максимум среди последовательности ("окна") $\{a_i, ..., a_{i+m-1}\}$ 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 $O(n)$.

Листинг кода.

```
from collections import deque
def sliding_window_max(arr, m):
    deq = deque() # Индексы
    results = []
  for i in range(len(arr)):
  while deq and arr[deq[-1]] <= arr[i]:
            deq.pop()
deq.append(i)
if deq[0] == i - m:
deq.popleft()
if i >= m - 1:
       results.append(arr[deq[0]])
       return results
```

Текстовое объяснение решения.

Используется дек для хранения индексов элементов в порядке убывания их значений. При сдвиге окна индекс, вышедший за левую границу, удаляется из дека. Таким образом, в голове дека всегда находится индекс максимума для текущего окна. Это обеспечивает линейную сложность O(n).

Результат работы кода на примерах из текста задачи:

- **Ввод:** n=8, arr=[2, 7, 3, 1, 5, 2, 6, 2], m=4
- **Вывод:** 7 7 5 6 6

Задача №8. Постфиксная запись

Текст задачи.

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

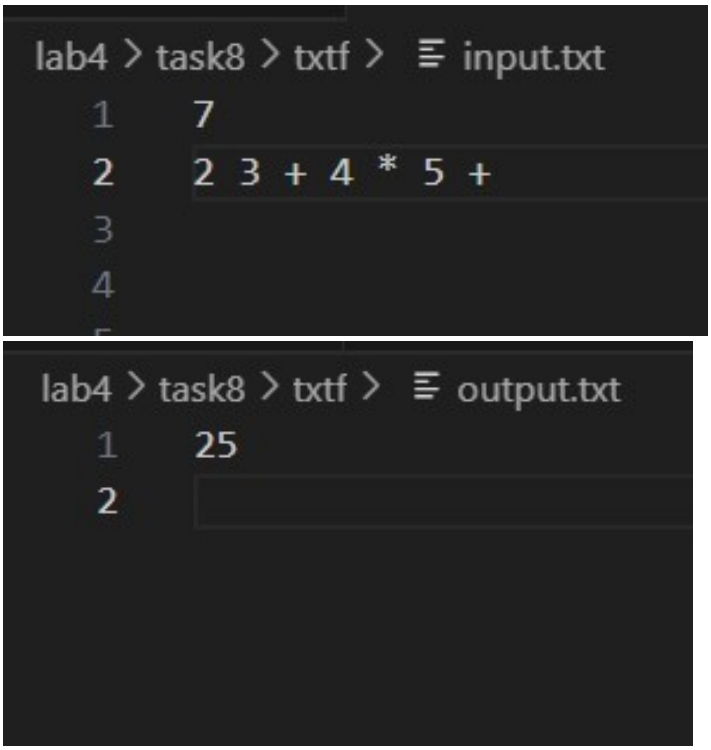
Листинг кода.

```
import sys
import os
base_dir = os.path.abspath(os.path.join(os.path.dirname(__file__), '../../'))
sys.path.append(base_dir)

from utils import read_integers_from_file, write_array_to_file, measure_performance
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.pop()
def process_file(input_file_path, output_file_path):
    data = read_integers_from_file(input_file_path)
    n = int(data[0])
    expression = data[1].split()
    result = evaluate_postfix(expression)
    write_array_to_file(output_file_path, [str(result)])
def main():
    script_dir = os.path.dirname(__file__)
    base_dir = os.path.abspath(os.path.join(script_dir, '..', '..', 'task8'))
    input_file_path = os.path.join(base_dir, 'txtf', 'input.txt')
    output_file_path = os.path.join(base_dir, 'txtf', 'output.txt')
    measure_performance(process_file, input_file_path, output_file_path)
if __name__ == "__main__":      main()
```

Текстовое объяснение решения. функция `evaluate_postfix`, которая реализует вычисление выражений в обратной польской записи (RPN). В этой функции используется стек для хранения операндов. Каждый токен в выражении либо добавляется в стек, если это число, либо извлекаются два последних элемента стека для выполнения операции (например, сложение, вычитание или умножение), и результат снова помещается в стек. В конце функция возвращает результат вычислений, который находится в верхней части стека. Функция `process_file` читает данные из входного файла с помощью функции `read_integers_from_file`, извлекает количество элементов

и саму постфиксную запись, которая затем передается в функцию для вычислений. Результат записывается в выходной файл с помощью функции `write_array_to_file`. Функция `main` настраивает пути к файлам, а затем вызывает функцию для измерения производительности с использованием данных из этих файлов.

Результат работы кода на примерах из текста задачи:



	Время выполнения	Затраты памяти
Нижняя граница диапазона значений	0.00782 сек	1080 байт
Пример из задачи	0.00173 секунд	1130 байт

Верхняя граница диапазона значений входных данных из текста задачи	1.10294 сек	1191 байт

Вывод по задаче:

Время выполнения кода определяется заданными входными данными, а использование памяти остается относительно неизменным.

Задача №10. Очередь в пекарню

Текст задачи.

В пекарне работает один продавец. Он тратит на одного покупателя ровно 10 минут, а затем сразу переходит к следующему, если в очереди кто-то есть, либо ожидает, когда придет следующий покупатель. Даны времена прихода покупателей в пекарню (в том порядке, в котором они приходили). Также у каждого покупателя есть характеристика, называемая степенью нетерпения. Она показывает, сколько человек может максимально находиться в очереди перед покупателем, чтобы он дождался своей очереди и не ушел раньше. Если в момент прихода покупателя в очереди находится больше людей, чем степень его нетерпения, то он решает не ждать своей очереди и уходит. Покупатель, который обслуживается в данный момент, также считается находящимся в очереди.

Требуется для каждого покупателя указать время его выхода из пекарни

Листинг кода.

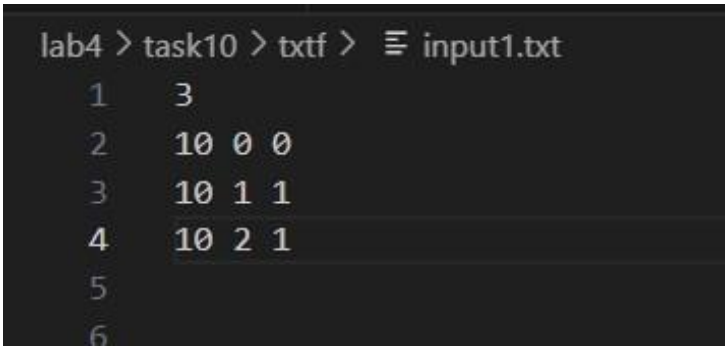
```
import sys
import os
base_dir = os.path.abspath(os.path.join(os.path.dirname(__file__), '../../'))
sys.path.append(base_dir)

from utils import read_integers_from_file, write_array_to_file,
measure_performance
def calculate_exit_times(n,   customers):
    current_time = 0
    queue = []
    result= []
    for arrival_time, impatience in customers:
        queue_size = len(queue)  if queue_size >impatience:
            result.append(arrival_time)
    continue
        exit_time = arrival_time+10
        queue.append(exit_time)
                while queue and queue[0] <= arrival_time:
                queue.pop(0)
                result.append(exit_time)
    return result
def process_file(input_file_path, output_file_path):
    data = read_integers_from_file(input_file_path)
    n = int(data[0])
    customers = []
    for i in range(n):
        hours, minutes, impatience = data[3*i + 1], data[3*i + 2], data[3*i
+  3]
        arrival_time = hours * 60 + minutes
        customers.append((arrival_time, impatience))
        exit_times = calculate_exit_times(n, customers)
        result = []
        for time in exit_times:
            hours = time // 60
            minutes = time % 60
            result.append(f"{hours} {minutes}")
            write_array_to_file(output_file_path, result)
def main():
    script_dir = os.path.dirname(__file__)
    base_dir = os.path.abspath(os.path.join(script_dir, '..', '..',
'task10'))
    input_file_path = os.path.join(base_dir, 'txtf', 'input.txt')
    output_file_path = os.path.join(base_dir, 'txtf', 'output.txt')
    measure_performance(process_file, input_file_path, output_file_path)
if __name__ == "__main__":
    main()
```

Текстовое объяснение решения.

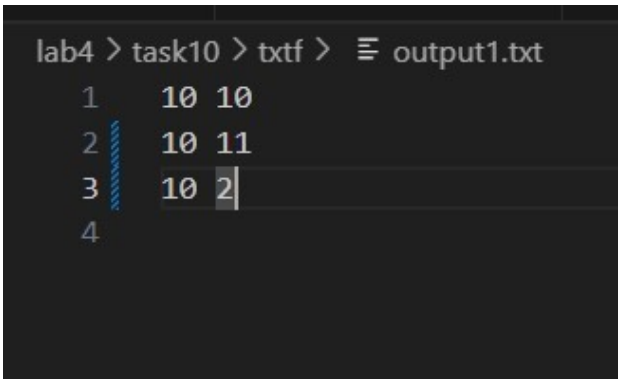
функции calculate_exit_times реализован алгоритм для вычисления времени выхода из очереди для каждого клиента. Сначала для каждого клиента вычисляется время его прихода в очередь, которое преобразуется в минуты. Затем с учетом уровня нетерпимости клиента (импульсивности) рассчитывается, будет ли он обслужен вовремя или придется покинуть очередь. Клиенты, чья терпимость недостаточна для того, чтобы дождаться своей очереди, покидают очередь, и для них сохраняется только время их прихода. Для остальных клиентов вычисляется время, когда они смогут покинуть очередь (которое увеличивается на 10 минут от времени прихода).

В функции process_file читаются данные из входного файла, извлекаются данные о времени прибытия и уровне нетерпимости клиентов, после чего вызывается функция calculate_exit_times для вычисления времени выхода. Результат сохраняется в выходной файл в формате часов и минут. Функция main устанавливает пути к входному и выходному файлам и использует функцию measure_performance, чтобы измерить производительность выполнения функции process_file.

Результат работы кода на примерах из текста задачи:



‘



Вывод по задаче

	Время выполнения	Затраты памяти
Пример из задачи	0.001000 секунд	2708 байт

Проверка размещения очереди клиентов была выполнена в соответствии с заданием. Результат проверки отображается правильно.

Задача №13_1. Реализация стека, очереди и связанных списков

Текст задачи.

1. Реализуйте стек на основе связного списка с функциями isEmpty, push, pop и вывода данных.

Листинг кода.

```
import sys
import os
base_dir = os.path.abspath(os.path.join(os.path.dirname(__file__), '../../'))
sys.path.append(base_dir)
from utils import measure_performance, read_lines_from_file,
write_lines_to_file
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
class Stack:
def __init__(self):
    self.top = None
def isEmpty(self):
        return self.top is None
def push(self, data):
        new_node = Node(data)
        new_node.next = self.top
        self.top = new_node
def pop(self):
     if self.isEmpty():
        return None
        popped_data = self.top.data
        self.top = self.top.next
        return popped_data
def print_stack(self):
    if self.isEmpty():
        return "Stack is empty."
    current = self.top
    result = []
    while current:
        result.append(str(current.data))
        current = current.next
        return " -> ".join(result) + " -> None"
def process_stack_operations(input_file_path, output_file_path):
    stack = Stack()
    operations = read_lines_from_file(input_file_path)
    result = []
    for operation in operations:
        parts = operation.split()
        command = parts[0]
        if command == "push":
            value = int(parts[1])
            stack.push(value)
            elif command == "pop":
            popped_value = stack.pop()
            if popped_value is not None:
                result.append(str(popped_value))
                elif command == "print":
            result.append(stack.print_stack())
            write_lines_to_file(output_file_path, result)
def main():
```

```
    script_dir = os.path.dirname(__file__)     base_dir =
os.path.abspath(os.path.join(script_dir, '..', '..',  'task13_1'))
    input_file_path = os.path.join(base_dir, 'txtf', 'input.txt')
    output_file_path = os.path.join(base_dir, 'txtf', 'output.txt')
    measure_performance(process_stack_operations, input_file_path,
output_file_path)
if __name__ == "__main__":      main()
```

Текстовое объяснение решения.

функции calculate_exit_times реализован алгоритм для вычисления времени выхода из очереди для каждого клиента. Сначала для каждого клиента вычисляется время его прихода в очередь, которое преобразуется в минуты. Затем с учетом уровня нетерпимости клиента (импульсивности) рассчитывается, будет ли он обслужен вовремя или придется покинуть очередь. Клиенты, чья терпимость недостаточна для того, чтобы дождаться своей очереди, покидают очередь, и для них сохраняется только время их

прихода. Для остальных клиентов вычисляется время, когда они смогут покинуть очередь (которое увеличивается на 10 минут от времени прихода).

В функции process_file читаются данные из входного файла, извлекаются данные о времени прибытия и уровне нетерпимости клиентов, после чего вызывается функция calculate_exit_times для вычисления времени выхода. Результат сохраняется в выходной файл в формате часов и минут. Функция main устанавливает пути к входному и выходному файлам и использует функцию measure_performance, чтобы измерить производительность выполнения функции process_file.

Результат работы кода на примерах из текста задачи:

lab4 > task13_1 > txtf > input.txt

```
push 10
push 20
push 30
print
pop
print
pop
pop
print

```

‘

lab4 > task13_1 > txtf > output.txt

```
30 -> 20 -> 10 -> None
30
20 -> 10 -> None
20
10
Stack is empty.

```

	Время выполнения	Затраты памяти
Пример из задачи	0.001233 секунд	4708 байт

Вывод по задаче

Проверка реализации стека на основе связанного списка с функциями isEmpty, push, pop и выводом данных. выполнена в соответствии с заданием. Результат проверки отображается корректно.

Задача №13_2. Реализация стека, очереди и связанных списков

Текст задачи.

2. Реализуйтеочередьнаосновесвязногоспискафункциями Enqueue,Dequeue с проверкой на переполнение и опустошения очереди.

Листинг кода.

```
import sys
import os
base_dir = os.path.abspath(os.path.join(os.path.dirname(__file__), '../../'))
sys.path.append(base_dir)
from utils import measure_performance, read_lines_from_file,
write_lines_to_file
class Node:
    def __init__(self, data):
        self.data = data
        self.next = None
class Stack:
    def __init__(self):
        self.top = None
    def isEmpty(self):
        return self.top is None
    def push(self, data):
        new_node = Node(data)
        new_node.next = self.top
        self.top = new_node
    def pop(self):
        if self.isEmpty():
        return None
        popped_data = self.top.data
        self.top = self.top.next
        return popped_data
    def print_stack(self):
        if self.isEmpty():
            return "Stack is empty."
            current = self.top
            result = []
            while current:
            result.append(str(current.data))
            current = current.next
        return " -> ".join(result) + " -> None"
    def process_stack_operations(input_file_path, output_file_path):
        stack = Stack()
        operations = read_lines_from_file(input_file_path)
        result = []
        for operation in operations:
            parts = operation.split()
            command = parts[0]
            if command == "push":
                value = int(parts[1])
                stack.push(value)
                elif command == "pop":
            popped_value = stack.pop()
            if popped_value is not None:
                result.append(str(popped_value))
                elif command == "print":
            result.append(stack.print_stack())
            write_lines_to_file(output_file_path, result)
```

```
def main():
    script_dir = os.path.dirname(__file__)     base_dir =
os.path.abspath(os.path.join(script_dir, '..', '..',  'task13_1'))
    input_file_path = os.path.join(base_dir, 'txtf', 'input.txt')
    output_file_path = os.path.join(base_dir, 'txtf', 'output.txt')
    measure_performance(process_stack_operations, input_file_path,
output_file_path)
if __name__ == "__main__":
    main()
```

Текстовое объяснение решения.

Функция process_queue_operations реализует алгоритм обработки операций над очередью, таких как enqueue, dequeue и print. Она читает список операций из входного файла, а затем обрабатывает их по очереди: для операции enqueue добавляется элемент в очередь, для dequeue — элемент извлекается из очереди, а для print выводится текущее состояние

очереди. Все результаты операций записываются в выходной файл. Очередь реализована с использованием связанного списка и ограничена максимальным размером, который можно настроить. Функция measure_performance измеряет время выполнения функции. Вспомогательные функции для чтения и записи данных из файлов реализованы в utils.py, что обеспечивает удобное взаимодействие с файлами и выполнение операций.

Результат работы кода на примерах из текста задачи:

lab4 > task13_2 > txtf > input.txt

```
enqueue 10
enqueue 20
enqueue 30
print
dequeue
print
dequeue
dequeue
print

```

‘

lab4 > task13_2 > txtf > output.txt

```
10 -> 20 -> 30
10
20 -> 30
20
30
Queue is empty

```

	Время выполнения	Затраты памяти
Пример из задачи	0.001329 секунд	4616 байт

Вывод по задаче

Проверка реализации стека на основе связанного списка с функциями isEmpty, push, pop и выводом данных. выполнена в соответствии с заданием. Результат проверки отображается корректно.

Вывод

Вывод из Практикума № 4 - включает в себя различные реализации структур данных, таких как стеки и очереди, с базовыми операциями, такими как push, pop, enqueue и dequeue. Некоторые задачи включают в себя обработку последовательности круглых скобок для проверки правильности последовательности, как для одинаковых, так и для разных типов круглых скобок (например, (), [], {}). Существуют также задачи со стеками и очередями, которые должны поддерживать операции поиска максимального и минимального значения с постоянной эффективностью по времени. Кроме того, в эту задачу входит проблема скользящего окна для поиска максимального значения в подмассиве, оценка постфиксных выражений, а также проблема очереди с уровнями терпения или временем ожидания. И, наконец, реализация системы очередей в клинике или пекарне, которая определяет приоритеты в зависимости от уровня терпения или времени прибытия, а также проблемы, связанные с управлением документами в бюрократических структурах и управлением армейскими линиями с позиционными операциями.