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

Отчет по лабораторной работе №4
по курсу «Алгоритмы и структуры данных»
Стек, очередь, связанный список
Вариант 15

Выполнил:
Левахин Лев Александрович
К3140

Проверил:
--

Санкт-Петербург
2024 г.

Содержание отчета

Задания по варианту: 1, 4, 5, 9

Задача 1 - Стек

Задача 4 - Скобочная последовательность

Задача 5 - Стек с максимумом

Задача 9 - Поликлиника

Задания по выбору: 7, 13

Задача 7 - Максимум в движущемся списке

Задача 13 - Стек, очередь, связанные списки

Задачи по варианту

Задача №1. Стек

Листинг кода.

```
"""
Элементарная структура - Стек.
Стек - динамическое множество
top - индекс последнего добавленного элемента
"""

from lab4 import utils
import os


CURRENT_SCRIPT_DIR_PATH = os.path.dirname(os.path.abspath(__file__))

class Node:
    """Содержит текущий элемент и ссылку на следующий"""

    def __init__(self, value):
        self.value = value
        self.next = None

class Stack:

    def __init__(self):
        self.head = Node("head")
        self.size = 0

    def __str__(self):
        cur_el = self.head.next
        result = ""
        while cur_el:
            result += str(cur_el.value) + ","
            cur_el = cur_el.next

        return result[:-1]

    def is_empty(self):
        """
        Проверка стека на пустоту
         - Если стек пуст: False
         - Если нет: True
        """
        return self.size == 0

    def size(self):
        """Возвращает размер стека"""
        return self.size

    def push(self, element):
        """Добавляет элемент в стек"""
        node = Node(element) # Узел следующего элемента
        node.next = self.head.next # Указываем следующему узлу элемента
следующий элемент головы
        self.head.next = node # Присваиваем нашей голове следующий элемент
        self.size += 1

    def pop(self):
        """Удаляет элемент из стека и возвращает его"""
        if self.is_empty():
            raise Exception("Нельзя удалить элемент из пустого списка!")
```

```
        else:
            remove_element = self.head.next
            self.head.next = remove_element.next
            self.size -=1

            return remove_element.value

    def top(self):
        """Возвращает последний элемент стека"""
        if self.is_empty():
            return None

        return self.head.next.value


class Utils:
    """Считывает и обрабатывает данные для стека"""
    def __init__(self):
        self.commands_list = []
        self.commands_cnt = None
        self.commands_name_list = []

    def read_stack_data(self):
        """Считывает данные для стека"""
        data = utils.read_file(CURRENT_SCRIPT_DIR_PATH)
        self.commands_cnt = int(data[0])
        self.commands_list = data[1:]

    def fill_commands_list(self):
        """Заполняет список, в котором операция добавления или исключения
элемента из стека
        соответствует команде"""
        for el in self.commands_list:
            if "+" in el:
                self.commands_name_list.append(["push", el[1:] ])
            if "-" in el:
                self.commands_name_list.append(["pop"])

    def fill_stack(self, stack):
        """
        Заполняет передаваемый стек элементами из файла
        Возвращает: (заполненный стек, элементы, которые были удалены)
        """
        pop_el_list = []
        for command in self.commands_name_list:
            if command[0] == "push":
                stack.push(command[-1])
            elif command[0] == "pop":
                pop_el_list.append(stack.pop())
        return stack, pop_el_list

def input_data():
    """Возвращает входные данные"""
    input_data = utils.read_file(CURRENT_SCRIPT_DIR_PATH)
    return input_data

def main():
    my_stack = Stack()  # Создаём стек
    stack_utils = Utils()  # Создаём утилс для считывания из файла
    stack_utils.read_stack_data()  # Считываем данные из файла
    stack_utils.fill_commands_list()  # Переделываем операции в команды
    my_stack, result = stack_utils.fill_stack(my_stack)  # Заполняем стек и
получаем список с удалёнными элементами
```

```
    return result

if __name__ == "__main__":
    result = main()
    utils.write_file(CURRENT_SCRIPT_DIR_PATH, result)
```

Текстовое объяснение решения.

Результат работы кода на примерах из текста задачи:
(скрины input output файлов)

Тесты к задаче:
Листинг кода:

```
import unittest
from lab4.task1.src.task1 import Stack, Utils
import tracemalloc
import datetime


class TaskTest1(unittest.TestCase):

    def test_stack_performance(self):
        """Тест на время и память"""
        # given
        my_stack = Stack()
        stack_utils = Utils()
        stack_utils.read_stack_data()
        max_allowed_time = datetime.timedelta(seconds=2) # Задаю
ограничение по времени

        # when
        tracemalloc.start() # Запускаем счётчик памяти
        start_time = datetime.datetime.now() # Запускаем счётчик времени

        stack_utils.fill_commands_list() # Переделываем операции в команды
        my_stack, result = stack_utils.fill_stack(my_stack) # Заполняем
стек и получаем список с удалёнными элементами

        finish_time = datetime.datetime.now()
        spent_time = finish_time - start_time # Итоговое время

        current, peak = tracemalloc.get_traced_memory()
        memory_used = current / 10 ** 6

        # then
        self.assertEqual(my_stack.__str__(),"2,1" )
        self.assertLessEqual(spent_time, max_allowed_time)
        self.assertLessEqual(memory_used, 256)

    def test_stack_correctly(self):
        """Тест на корректность работы стека"""
        # given
        my_stack = Stack()
        empty_stack = Stack()

        # when
        my_stack.push(4)
        my_stack.push(8)
```

```
        my_stack.push(15)
        my_stack.push(16)

        # then
        self.assertEqual(my_stack.__str__(), "16,15,8,4")
        self.assertEqual(empty_stack.__str__(), "")

    def test_stack_is_empty(self):
        """Тест на пустоту стека"""
        # given
        my_stack = Stack()
        empty_stack = Stack()

        # when
        my_stack.push(4)
        my_stack.push(8)
        my_stack.push(15)
        my_stack.push(16)

        # then
        self.assertEqual(my_stack.is_empty(), False)
        self.assertEqual(empty_stack.is_empty(), True)

    def test_stack_size(self):
        """Тест на размер стека"""
        # given
        my_stack = Stack()
        empty_stack = Stack()

        # when
        my_stack.push(4)
        my_stack.push(8)
        my_stack.push(15)
        my_stack.push(16)

        # then
        self.assertEqual(my_stack.size, 4)
        self.assertEqual(empty_stack.size, 0)

    def test_stack_top(self):
        """Тест на верхний элемент стека"""
        # given
        my_stack = Stack()
        empty_stack = Stack()

        # when
        my_stack.push(4)
        my_stack.push(8)
        my_stack.push(15)
        my_stack.push(16)

        # then
        self.assertEqual(my_stack.top(), 16)


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

Скрины работы тестов:

Задача №4. Скобочная последовательность

Листинг кода.

```
"""
Скобочная последовательность. Версия 2
"""

from lab4 import utils
import os


CURRENT_SCRIPT_DIR_PATH = os.path.dirname(os.path.abspath(__file__))


class BracketChecker:
    """Проверяет, нет ли ошибок в скобочной последовательности"""

    def __init__(self):
        # Словарь для соответствия открывающих и закрывающих скобок
        self.pairs = {'(': ')', '[': ']', '{': '}'}

    def check(self, row):
        """
        Проверяет на отсутствие закрывающих и открывающих скобок
         - Если таких нет: Success
         - Если есть: Выводит индекс такой (начиная с 1)
        """
        stack = []
        index_stack = []

        for index, char in enumerate(row):
            if char in self.pairs: # Для открывающей скобки
                stack.append(char)
                index_stack.append(index + 1)

            elif char in self.pairs.values(): # Для закрывающей скобки
                # Проверяем, есть ли соответствующая открывающая скобка
                if stack and self.pairs[stack[-1]] == char:
                    stack.pop()  # Убираем соответствующую открывающую
скобку
                    index_stack.pop()  # Убираем соответствующий индекс
                else:
                    return index + 1  # Возвращаем индекс закрывающей
скобки

        if index_stack:
            return index_stack[0]  # Возвращаем индекс первой открывающей
скобки без закрывающей

        return "Success"

def input_data():
    """Возвращает входные данные"""
    input_data = utils.read_file(CURRENT_SCRIPT_DIR_PATH)
    return input_data

def main():
    bracket_checker = BracketChecker()
    lines_lst = utils.read_file(CURRENT_SCRIPT_DIR_PATH)
    result = []
    for line in lines_lst:
        result.append(bracket_checker.check(line))

    return result
```

```
if __name__ == "__main__":
    result = main()
    utils.write_file(CURRENT_SCRIPT_DIR_PATH, result)
```

Текстовое объяснение решения.

Результат работы кода на примерах из текста задачи:
(скрины input output файлов)

Тесты к задаче:
Листинг кода:

```
from lab4.task4.src.task4 import BracketChecker
from lab4 import utils
import unittest
import tracemalloc
import datetime
import os


CURRENT_SCRIPT_DIR_PATH = os.path.dirname(os.path.abspath(__file__))

class TaskTest4(unittest.TestCase):

    def test_sort(self):
        """Тест на данных из примера"""
        # given
        bracket_checker = BracketChecker()
        lines_lst = utils.read_file(CURRENT_SCRIPT_DIR_PATH)
        max_allowed_time = datetime.timedelta(seconds=5)  # Задаю
ограничение по времени
        result = []

        # when
        tracemalloc.start()  # Запускаем счётчик памяти
        start_time = datetime.datetime.now()  # Запускаем счётчик времени

        for line in lines_lst:
            result.append(bracket_checker.check(line))

        finish_time = datetime.datetime.now()  # Измеряем время конца
работы
        spent_time = finish_time - start_time  # Итоговое время

        current, peak = tracemalloc.get_traced_memory()
        memory_used = current / 10 ** 6

        # then
        self.assertEqual(result, ['Success', 'Success', 'Success',
'Success', 1, 3, 'Success', 10])
        self.assertLessEqual(spent_time, max_allowed_time)
        self.assertLessEqual(memory_used, 256)
```

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

Скрины работы тестов:

Задача 5. Стек с максимумом

Листинг кода.

```
"""
Стек с максимумом
"""

from lab4 import utils
from lab4.task1.src.task1 import Stack, Utils
import os


CURRENT_SCRIPT_DIR_PATH = os.path.dirname(os.path.abspath(__file__))

class StackMax(Stack):
    def max(self):
        cur_el = self.head.next
        max_el = -10**9
        while cur_el:
            max_el = max(int(cur_el.value), max_el)
            cur_el = cur_el.next

        return max_el

class UtilsMax(Utils):

    def read_stack_data(self, input_file_name = "input.txt"):
        """Считывает данные для стека"""
        with open(os.path.join(CURRENT_SCRIPT_DIR_PATH, "../txtfiles/",
input_file_name), "r") as file:
            self.commands_cnt = int(file.readline())
            for i in range(self.commands_cnt):
                self.commands_list.append(file.readline().strip())

    def fill_stack(self, stack):
        max_el_list = []
        for command in self.commands_list:
            command = command.split()
            if command[0] == "push":
                stack.push(command[-1])
            elif command[0] == "pop":
                stack.pop()
            elif command[0] == "max":
                max_el_list.append(stack.max())
        return stack, max_el_list


def input_data():
    """Возвращает входные данные"""
    input_data = utils.read_file(CURRENT_SCRIPT_DIR_PATH)
    return input_data

def main():
    result = []
```

```
    my_stack1 = StackMax()  # Создаём стек
    stack_utils1 = UtilsMax()  # Создаём утилс для считывания из файла
    stack_utils1.read_stack_data()  # Считываем данные из файла
    my_stack1, result1 = stack_utils1.fill_stack(my_stack1)  # Заполняем
стек и получаем список с максимальными элементами

    my_stack2 = StackMax()  # Создаём стек
    stack_utils2 = UtilsMax()  # Создаём утилс для считывания из файла
    stack_utils2.read_stack_data("input2.txt")  # Считываем данные из файла
    my_stack2, result2 = stack_utils2.fill_stack(my_stack2)  # Заполняем
стек и получаем список с максимальными элементами

    my_stack3 = StackMax()  # Создаём стек
    stack_utils3 = UtilsMax()  # Создаём утилс для считывания из файла
    stack_utils3.read_stack_data("input3.txt")  # Считываем данные из файла
    my_stack3, result3 = stack_utils3.fill_stack(my_stack3)  # Заполняем
стек и получаем список с максимальными элементами

    return ["input 1:", result1, "\ninput 2:", result2, "\ninput 3:",
result3]


if __name__ == "__main__":
    result = main()
    utils.write_file(CURRENT_SCRIPT_DIR_PATH, result)
```

Текстовое объяснение решения.

Результат работы кода на примерах из текста задачи:
(скрины input output файлов)

Тесты к задаче:
Листинг кода:

```
from lab4 import utils
import unittest
from lab4.task5.src.task5 import StackMax, UtilsMax
import tracemalloc
import datetime


class TaskTest5(unittest.TestCase):

    def test_stack_performance(self):
        """Тест на время и память"""
        # given
        my_stack = StackMax()
        stack_utils = UtilsMax()
        stack_utils.read_stack_data()
        max_allowed_time = datetime.timedelta(seconds=5) # Задаю
ограничение по времени

        # when
        tracemalloc.start()  # Запускаем счётчик памяти
```

```
        start_time = datetime.datetime.now()  # Запускаем счётчик времени

        my_stack, result = stack_utils.fill_stack(my_stack)  # Заполняем
стек и получаем список с максимальными элементами

        finish_time = datetime.datetime.now()
        spent_time = finish_time - start_time # Затраченное время

        current, peak = tracemalloc.get_traced_memory()
        memory_used = current / 10 ** 6 # Затраченная память

        # then
        self.assertEqual(my_stack.__str__(), "2")
        self.assertEqual(result, [2, 2])
        self.assertLessEqual(spent_time, max_allowed_time)
        self.assertLessEqual(memory_used, 256)


    def test_stack_correctly(self):
        """Тест на корректность работы стека"""
        # given
        my_stack2 = StackMax()
        stack_utils2 = UtilsMax()
        stack_utils2.read_stack_data("input2.txt")

        my_stack3 = StackMax()
        stack_utils3 = UtilsMax()
        stack_utils3.read_stack_data("input3.txt")

        # when
        my_stack2, result2 = stack_utils2.fill_stack(my_stack2)
        my_stack3, result3 = stack_utils3.fill_stack(my_stack3)

        # then
        self.assertEqual(my_stack2.__str__(), "1")
        self.assertEqual(result2, [2, 1])
        self.assertEqual(my_stack3.__str__(), "1")
        self.assertEqual(result3, [])


    def test_stack_is_empty(self):
        """Тест на пустоту стека"""
        # given
        my_stack2 = StackMax()
        stack_utils2 = UtilsMax()
        stack_utils2.read_stack_data("input2.txt")

        my_stack3 = StackMax()
        stack_utils3 = UtilsMax()
        stack_utils3.read_stack_data("input3.txt")

        # when
        my_stack2, result2 = stack_utils2.fill_stack(my_stack2)
        my_stack3, result3 = stack_utils3.fill_stack(my_stack3)

        # then
        self.assertEqual(my_stack2.is_empty(), False)
        self.assertEqual(my_stack3.is_empty(), False)

    def test_stack_size(self):
        """Тест на размер стека"""
        # given
        my_stack2 = StackMax()
        stack_utils2 = UtilsMax()
```

```
        stack_utils2.read_stack_data("input2.txt")

        my_stack3 = StackMax()
        stack_utils3 = UtilsMax()
        stack_utils3.read_stack_data("input3.txt")

        # when
        my_stack2, result2 = stack_utils2.fill_stack(my_stack2)
        my_stack3, result3 = stack_utils3.fill_stack(my_stack3)

        # then
        self.assertEqual(my_stack2.size, 1)
        self.assertEqual(my_stack3.size, 1)


    def test_stack_top(self):
        """Тест на верхний элемент стека"""
        # given
        my_stack2 = StackMax()
        stack_utils2 = UtilsMax()
        stack_utils2.read_stack_data("input2.txt")

        # when
        my_stack2, result2 = stack_utils2.fill_stack(my_stack2)

        # then
        self.assertEqual(my_stack2.top(), "1")


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

Скрины работы тестов:

Задача №9. Поликлиника

Листинг кода.

```
"""
Сортировка целых чисел
"""

from lab4 import utils
import os


CURRENT_SCRIPT_DIR_PATH = os.path.dirname(os.path.abspath(__file__))

class PatientQueue:
    """Очередь пациентов на основе списка"""

    def __init__(self):
        self.queue = []
        self.length = 0

    def __str__(self):
        return " ".join(self.queue)

    def length(self):
        """Возвращает длину очереди"""
```

```
        return self.length

    def is_empty(self):
        """
        Проверка стека на пустоту
         - Если стек пуст: False
         - Если нет: True
        """
        return self.length == 0

    def enqueue_to_end(self, patient_number):
        """Включить элемент в очередь"""
        self.queue.append(patient_number)
        self.length += 1

    def enqueue_to_middle(self, patient_number):
        """Включить элемент в очередь"""
        middle_index = (self.length + 1)//2
        self.queue.insert(middle_index, patient_number)
        self.length += 1

    def dequeue(self):
        """Исключить первый элемент из очереди"""
        self.length -= 1
        return self.queue.pop(0)


class ReadPatientData:
    """Считывает и обрабатывает данные для очереди пациентов"""

    def __init__(self):
        self.patients_list = []
        self.commands_cnt = None

    def read_utils_data(self, input_file_name = "input.txt"):
        """Считывает данные для стека"""
        with open(os.path.join(CURRENT_SCRIPT_DIR_PATH, "../txtfiles/",
input_file_name), "r") as file:
            self.commands_cnt = int(file.readline())
            for i in range(self.commands_cnt):
                self.patients_list.append(file.readline().strip())

    def fill_queue(self, queue):
        """
        Заполняет передаваемую очередь элементами из файла
        Возвращает: заполненную очередь, список пациентов, зашедших к
врачу(исключенные из очереди)
        """
        pop_patients_list = []
        for patient in self.patients_list:
            if patient[0] == "+":
                queue.enqueue_to_end(patient[1])
            elif patient[0] == "*":
                queue.enqueue_to_middle(patient[1])
            elif patient[0] == "-":
                pop_patients_list.append(queue.dequeue())

        return queue, pop_patients_list

def input_data():
    """Возвращает входные данные"""
    input_data = utils.read_file(CURRENT_SCRIPT_DIR_PATH)
    return input_data
```

```
def main():
    # Для первого инпута
    queue = PatientQueue()
    queue_utils = ReadPatientData()
    queue_utils.read_utils_data()
    queue, result1 = queue_utils.fill_queue(queue)

    # Для второго инпута
    queue2 = PatientQueue()
    queue_utils2 = ReadPatientData()
    queue_utils2.read_utils_data("input2.txt")
    queue2, result2 = queue_utils2.fill_queue(queue2)

    result = ["input1"] + result1 + ["\ninput2"] + result2
    return result

if __name__ == "__main__":
    result = main()
    utils.write_file(CURRENT_SCRIPT_DIR_PATH, result)
```

Текстовое объяснение решения.

Результат работы кода на примерах из текста задачи:
(скрины input output файлов)

Тесты к задаче:

Листинг кода:

```
from lab4 import utils
import unittest
from lab4.task9.src.task9 import PatientQueue, ReadPatientData
import tracemalloc
import datetime


class TaskTest9(unittest.TestCase):

    def test_queue_performance(self):
        """Тест на время и память"""
        # given
        queue = PatientQueue()
        queue_utils = ReadPatientData()
        queue_utils.read_utils_data()

        # when
        tracemalloc.start()  # Запускаем счётчик памяти
        start_time = datetime.datetime.now()  # Запускаем счётчик времени

        queue, result1 = queue_utils.fill_queue(queue)

        finish_time = datetime.datetime.now()
        spent_time = finish_time - start_time # Итоговое время

        current, peak = tracemalloc.get_traced_memory()
```

```
        memory_used = current / 10 ** 6  # Итоговая память

        # then
        self.assertEqual(queue.__str__(), "4")
        self.assertEqual(result1, ["1", "2", "3"])

    def test_queue_correctly(self):
        """Тест на корректность работы очереди"""
        # given
        queue2 = PatientQueue()
        queue_utils2 = ReadPatientData()
        queue_utils2.read_utils_data("input2.txt")

        # when
        queue2, result2 = queue_utils2.fill_queue(queue2)

        # then
        self.assertEqual(queue2.__str__(), "")
        self.assertEqual(result2, ["1", "3", "2", "5", "4"])


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

Скрины работы:

Дополнительные задачи

Задача №7. Максимум в движущемся списке

Листинг кода.

```
"""
Поиск максимума в движущемся массиве
"""

from lab4 import utils
from collections import deque
import os


CURRENT_SCRIPT_DIR_PATH = os.path.dirname(os.path.abspath(__file__))


def find_sliding_max(lst: list, n: int, m: int) -> list:
    """
    Находит максимум в движущемся списке
     - Принимает: список, длину списка, число окон
     - Возвращает: список с максимумами от 1 до n-m + 1
    """
    res = []
    d_queue = deque()  # Двусторонняя очередь

    for i in range(n):
        if d_queue and d_queue[0] < i - m + 1:  # Удаляем элементы за
пределами окна
            d_queue.popleft()

        while d_queue and lst[d_queue[-1]] < lst[i]:  # Удаляем элементы,
которые меньше текущего
            d_queue.pop()

        d_queue.append(i)  # Добавляем текущий элемент в очередь

        if i >= m - 1:
            res.append(lst[d_queue[0]])  # Если достигли размера окна,
добавляем максимальное значение в результат

    return res


def read_input_file() -> (list, int, int):
    """Считываем специфический инпут"""
    data = utils.read_file(CURRENT_SCRIPT_DIR_PATH)
    n = int(data[0])  # Длина списка
    lst = utils.str_to_list(data[1])
    m = int(data[-1])  # Ширина окна

    return lst, n, m


def input_data():
    """Возвращает входные данные"""
    input_data = utils.read_file(CURRENT_SCRIPT_DIR_PATH)
    return input_data

def main():
    lst, n, m = read_input_file()
    return find_sliding_max(lst, n, m)
```

```
if __name__ == "__main__":
    result = main()
    utils.write_file(CURRENT_SCRIPT_DIR_PATH, [result])
```

Текстовое объяснение решения.

Результат работы кода на примерах из текста задачи:
(скрины input output файлов)

Тесты к задаче:
Листинг кода:

```
from lab4 import utils
import unittest
from lab4.task7.src.task7 import find_sliding_max, read_input_file
import tracemalloc
import datetime


class TaskTest7(unittest.TestCase):

    def test_stack(self):
        """Тест стека"""
        # given
        lst, n, m = read_input_file()
        max_allowed_time = datetime.timedelta(seconds=2)  # Задаю
ограничение по времени

        # when
        tracemalloc.start()
        start_time = datetime.datetime.now()

        find_sliding_max(lst, n, m)

        finish_time = datetime.datetime.now()
        spent_time = finish_time - start_time # Итоговое время

        current, peak = tracemalloc.get_traced_memory()
        memory_used = current / 10 ** 6 # Итоговая память

        # then
        self.assertEqual(find_sliding_max(lst, n, m), [7, 7, 5, 6, 6])
        self.assertLessEqual(spent_time, max_allowed_time)
        self.assertLessEqual(memory_used, 256)


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

Скрины работы тестов:

Задача №13. Стек и очередь

Листинг кода.

```
"""Стек на основе связанного списка"""

from lab4 import utils
import os


CURRENT_SCRIPT_DIR_PATH = os.path.dirname(os.path.abspath(__file__))

class Node:
    """Содержит текущий элемент и ссылку на следующий"""
    def __init__(self, value):
        self.value = value
        self.next = None

class Stack:
    """Стек на основе связанного списка"""
    def __init__(self):
        self.head = Node("head")
        self.size = 0

    def __str__(self):
        cur_el = self.head.next
        result = ""
        while cur_el:
            result += str(cur_el.value) + ","
            cur_el = cur_el.next

        return result[:-1]

    def is_empty(self):
        """
        Проверка стека на пустоту
         - Если стек пуст: False
         - Если нет: True
        """
        return self.size == 0

    def size(self):
        """Возвращает размер стека"""
        return self.size

    def push(self, element):
        """Добавляет элемент в стек"""
        node = Node(element) # Узел следующего элемента
        node.next = self.head.next # Указываем следующему узлу элемента
следующий элемент головы
        self.head.next = node # Присваиваем нашей голове следующий элемент
        self.size += 1

    def pop(self):
        """Удаляет элемент из стека и возвращает его"""
        if self.is_empty():
            raise Exception("Нельзя удалить элемент из пустого списка!")
        else:
            remove_element = self.head.next
            self.head.next = remove_element.next
            self.size -=1

            return remove_element.value

    def top(self):
```

```
        """Возвращает последний элемент стека"""
        if self.is_empty():
            return None

        return self.head.next.value


class Utils:
    """Считывает и обрабатывает данные для стека"""
    def __init__(self):
        self.commands_list = []
        self.commands_cnt = None
        self.commands_name_list = []

    def read_stack_data(self):
        """Считывает данные для стека"""
        data = utils.read_file(CURRENT_SCRIPT_DIR_PATH)
        self.commands_cnt = int(data[0])
        self.commands_list = data[1:]

    def fill_commands_list(self):
        """Заполняет список, в котором операция добавления или исключения
элемента из стека
        соответствует команде"""
        for el in self.commands_list:
            if "+" in el:
                self.commands_name_list.append(["push", el[1:]])
            if "-" in el:
                self.commands_name_list.append(["pop"])

    def fill_stack(self, stack):
        """
        Заполняет передаваемый стек элементами из файла
        Возвращает: (заполненный стек, элементы, которые были удалены)
        """
        pop_el_list = []
        for command in self.commands_name_list:
            if command[0] == "push":
                stack.push(command[-1])
            elif command[0] == "pop":
                pop_el_list.append(stack.pop())
        return stack, pop_el_list


def input_data():
    """Возвращает входные данные"""
    input_data = utils.read_file(CURRENT_SCRIPT_DIR_PATH)
    return input_data


def main():
    my_stack = Stack()  # Создаём стек
    stack_utils = Utils()  # Создаём утилс для считывания из файла
    stack_utils.read_stack_data()  # Считываем данные из файла
    stack_utils.fill_commands_list()  # Переделываем операции в команды
    my_stack, result = stack_utils.fill_stack(my_stack)  # Заполняем стек и
получаем список с удалёнными элементами

    return result


if __name__ == "__main__":
    result = main()
    utils.write_file(CURRENT_SCRIPT_DIR_PATH, result)
```

Текстовое объяснение решения.

Результат работы кода на примерах из текста задачи:
(скрины input output файлов)

Тесты к задаче:
Листинг кода:

```
from lab4 import utils
import unittest
from lab4.task13.src.task13_2 import Queue
import tracemalloc
import datetime


class TaskTest13Queue(unittest.TestCase):

    def test_queue(self):
        """Тест стека на основе связанного списка"""
        # given
        queue = Queue()

        # when
        tracemalloc.start()  # Запускаем счётчик памяти
        start_time = datetime.datetime.now()  # Запускаем счётчик времени

        for i in [4, 8, 15, 16, 23, 42]:
            queue.enqueue(i)

        for i in range(2):
            queue.dequeue()

        finish_time = datetime.datetime.now()
        spent_time = finish_time - start_time  # Итоговое время

        current, peak = tracemalloc.get_traced_memory()
        memory_used = current / 10 ** 6

        # then
        self.assertEqual(queue.__str__(),"15,16,23,42")

    def test_queue_correctly(self):
        """Тест на корректность работы"""
        # given
        queue = Queue()
        empty_queue = Queue()

        # when
        for i in [4, 8, 15, 16, 23, 42]:
            queue.enqueue(i)
        for i in range(2):
            queue.dequeue()

        # then
        self.assertEqual(empty_queue.__str__(), "")

    def test_stack_is_empty(self):
        """Тест на пустоту очереди"""
```

```
        # given
        queue = Queue()
        empty_queue = Queue()

        # when
        for i in [4, 8, 15, 16, 23, 42]:
            queue.enqueue(i)
        for i in range(2):
            queue.dequeue()

        # then
        self.assertEqual(queue.is_empty(), False)
        self.assertEqual(empty_queue.is_empty(), True)

    def test_stack_length(self):
        """Тест на длину очереди"""
        # given
        queue = Queue()
        empty_queue = Queue()

        # when
        for i in [4, 8, 15, 16, 23, 42]:
            queue.enqueue(i)
        for i in range(2):
            queue.dequeue()

        # then
        self.assertEqual(queue.length(), 4)
        self.assertEqual(empty_queue.length(), 0)


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

```
import unittest
from lab4.task13.src.task13_1 import Stack, Utils
import tracemalloc
import datetime


class TaskTest13Stack(unittest.TestCase):

    def test_stack_performance(self):
        """Тест на время и память"""
        # given
        my_stack = Stack()
        stack_utils = Utils()
        stack_utils.read_stack_data()

        # when
        tracemalloc.start() # Запускаем счётчик памяти
        start_time = datetime.datetime.now() # Запускаем счётчик времени

        stack_utils.fill_commands_list() # Переделываем операции в команды
        my_stack, result = stack_utils.fill_stack(my_stack) # Заполняем
стек и получаем список с удалёнными элементами

        finish_time = datetime.datetime.now()
        spent_time = finish_time - start_time # Итоговое время

        current, peak = tracemalloc.get_traced_memory()
        memory_used = current / 10 ** 6

        # then
```

```
        self.assertEqual(my_stack.__str__(),"2,1" )

    def test_stack_correctly(self):
        """Тест на корректность работы стека"""
        # given
        my_stack = Stack()
        empty_stack = Stack()

        # when
        my_stack.push(4)
        my_stack.push(8)
        my_stack.push(15)
        my_stack.push(16)

        # then
        self.assertEqual(my_stack.__str__(), "16,15,8,4")
        self.assertEqual(empty_stack.__str__(), "")

    def test_stack_is_empty(self):
        """Тест на пустоту стека"""
        # given
        my_stack = Stack()
        empty_stack = Stack()

        # when
        my_stack.push(4)
        my_stack.push(8)
        my_stack.push(15)
        my_stack.push(16)

        # then
        self.assertEqual(my_stack.is_empty(), False)
        self.assertEqual(empty_stack.is_empty(), True)

    def test_stack_size(self):
        """Тест на размер стека"""
        # given
        my_stack = Stack()
        empty_stack = Stack()

        # when
        my_stack.push(4)
        my_stack.push(8)
        my_stack.push(15)
        my_stack.push(16)

        # then
        self.assertEqual(my_stack.size, 4)
        self.assertEqual(empty_stack.size, 0)

    def test_stack_top(self):
        """Тест на верхний элемент стека"""
        # given
        my_stack = Stack()

        # when
        my_stack.push(4)
        my_stack.push(8)
        my_stack.push(15)
        my_stack.push(16)

        # then
        self.assertEqual(my_stack.top(), 16)
```

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

Скрины работы тестов:

Вывод

Стек и очередь распространённые алгебраические структуры данных, которые довольно удобны в использовании. В питон довольно просто реализовать такие структуры с помощью обычных списков.