

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 9

Выполнила:

Левчук С.А.

К3141

Проверил:

Афанасьев А. В.

Санкт-Петербург

2024 г.

Содержание отчета

Содержание отчета	2
Задачи по варианту	3
Задание №1. Стек	3
Задание №3. Скобочная последовательность. Версия 1	6
Задание №6. Очередь с минимумом	10
Задание №9. Поликлиника	14
Дополнительные задачи	20
Задание №8. Постфиксная запись	20
Задание №13. Реализация стека, очереди и связанных списков	24
Вывод	30

Задачи по варианту

Задание №1. Стек

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	
-	

Решение:

```
from utils import read, write
from lab4.task13.src.stack import Stack

def main():
    stack = Stack()
    write(end='')
    for line in read(type_convert=str):
        if line[0] == '+':
            stack.push(line[1])
        elif line[0] == '-':
            write(stack.pop(), to_end=True)
        else:
            raise ValueError()

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

Решение основано на уже реализованном в задаче 13 (пункт 1) классе Stack.

В зависимости от обрабатываемой строки входных данных:

1. если в начале строки стоит '+', то к стеку добавляется с помощью метода push следующее после него число,
2. если в начале строки стоит '-', то из стека извлекается последний добавленный элемент с помощью метода pop и передается в функцию write для вывода,
3. если ни одно из условий не выполняется, вызывается исключение ValueError, указывающее на ошибку в формате команды.

Результат работы программы:

Входные данные:

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

Выходные данные:

```
10
1234
```

Тесты:

```
import unittest
from utils import memory_data, time_data
from lab4.task1.src.stack import main

class TestStack(unittest.TestCase):

    def test_should_check_time_data(self):
        # given
        expected_time = 2

        # when
        time = time_data(main)

        # then
        self.assertTrue(time < expected_time)

    def test_should_check_memory_data(self):
        # given
        expected_memory = 256

        # when
        current, peak = memory_data(main)

        # then
```

```
self.assertTrue(current < expected_memory)
self.assertTrue(peak < expected_memory)

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

Вывод по задаче:

В ходе решения данной задачи мы реализовали работу стека.

Задание №3. Скобочная последовательность. Версия 1

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
)()	

Решение:

```
from utils import read, write
from lab4.task13.src.stack import Stack

def brackets_check(brackets):
    stack = Stack()
    for item in brackets:
        if item in ')]':
            if stack.is_empty():
                return False
            last_item = stack.pop()
```

```

        if (item + last_item == ']( ' or (item + last_item == ')[ '):
            return False
        else:
            stack.push(item)
    return stack.is_empty()

def main():
    brackets_list = read(type_convert=str)
    write(end='')
    for brackets, in brackets_list:
        write('YES' if brackets_check(brackets) else 'NO', to_end=True)

if __name__ == '__main__':
    main()

```

Решение основано на уже реализованном в задаче 13 (пункт 1) классе Stack.

Проходим циклом по символам строки:

1. Если текущий символ является закрывающей скобкой (] или)), то:
 - 1) проверяется, пуст ли стек. Если да, значит открывающая скобка отсутствует, и функция сразу возвращает False,
 - 2) иначе извлекается последний элемент из стека с помощью метода pop() и сравнивается с текущим символом. Если пара не соответствует правилам ([) или (]), возвращается False.
2. Если же текущий символ — это открывающая скобка ((или [), она добавляется в стек методом push().

Проверка после цикла: После того как все символы обработаны, проверяется, что стек пуст. Это означает, что все открывающие скобки были закрыты соответствующими закрывающими скобками. Если стек пуст, функция возвращает True, иначе False.

Результат работы программы:

Входные данные:

```

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

```

Выходные данные:

```

YES
YES
NO
NO
NO

```

Тесты:

```
import unittest
from utils import memory_data, time_data
from lab4.task3.src.brackets import main, brackets_check

class TestBrackets(unittest.TestCase):

    def test_should_check_correct_data(self):
        # given
        data = ['()()', '([])', '[([]())]', '']

        # when
        res0 = brackets_check(data[0])
        res1 = brackets_check(data[1])
        res2 = brackets_check(data[2])
        res3 = brackets_check(data[3])

        # then
        self.assertIs(res0, True)
        self.assertIs(res1, True)
        self.assertIs(res2, True)
        self.assertIs(res3, True)

    def test_should_check_incorrect_data(self):
        # given
        data = ['([])', '([)]', ')((']

        # when
        res0 = brackets_check(data[0])
        res1 = brackets_check(data[1])
        res2 = brackets_check(data[2])

        # then
        self.assertIs(res0, False)
        self.assertIs(res1, False)
        self.assertIs(res2, False)

    def test_should_check_time_data(self):
        # given
        expected_time = 2

        # when
        time = time_data(main)

        # then
        self.assertTrue(time < expected_time)

    def test_should_check_memory_data(self):
        # given
        expected_memory = 256

        # when
        current, peak = memory_data(main)
```



```
        # then
        self.assertTrue(current < expected_memory)
        self.assertTrue(peak < expected_memory)

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

Вывод по задаче:

В ходе решения данной задачи мы реализовали алгоритм проверки правильности расстановки скобок в последовательности.

Задание №6. Очередь с минимумом

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 находятся в очереди. Для каждой операции запроса минимального элемента выведите ее результат.

На вход программе подаются строки, содержащие команды. Каждая строка содержит одну команду. Команда – это либо «+ N », либо «-», либо «?». Команда «+ N » означает добавление в очередь числа N , по модулю не превышающего 10^9 . Команда «-» означает изъятие элемента из очереди. Команда «?» означает запрос на поиск минимального элемента в очереди.

- **Формат входного файла (input.txt).** В первой строке содержится M ($1 \leq M \leq 10^6$) – число команд. В последующих строках содержатся команды, по одной в каждой строке.
- **Формат выходного файла (output.txt).** Для каждой операции поиска минимума в очереди выведите её результат. Результаты должны быть выведены в том порядке, в котором эти операции встречаются во входном файле. Гарантируется, что операций извлечения или поиска минимума для пустой очереди не производится.
- Ограничение по времени. 2 сек.
- Ограничение по памяти. 256 мб.
- Пример:

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

Решение:

```
from utils import write, read

class QueueWithMin:
    def __init__(self):
        self.queue = []
        self.minimum = float('+inf')

    def push(self, value):
        self.queue.append(int(value))
        if int(value) < self.minimum:
            self.minimum = int(value)

    def pop(self):
        if not self.queue:
            raise IndexError()
        del_value = self.queue.pop(0)
```

```

        if del_value == self.minimum:
            self.minimum = float('+inf')
            for value in self.queue:
                if int(value) < self.minimum:
                    self.minimum = int(value)

    def get_min(self):
        if not self.queue:
            raise IndexError()
        return self.minimum

def main():
    queue = QueueWithMin()
    write(end='')
    for line in read(type_convert=str):
        if line[0] == '+':
            value = line[1]
            queue.push(value)
        elif line[0] == '-':
            queue.pop()
        else:
            write(queue.get_min(), sep='\n', to_end=True)

if __name__ == '__main__':
    main()

```

1. Определение класса QueueWithMin, который представляет собой очередь с возможностью находить минимальный элемент.
2. Метод push используется для добавления нового элемента в очередь.
3. Метод pop удаляет первый элемент из очереди.
4. Метод get_min возвращает текущее минимальное значение в очереди. Если очередь пуста, генерируется исключение IndexError.
5. Основная программа построчно считывает и обрабатывает входные данные:
 - 1) Если строка начинается с +, то вызывается метод push для добавления нового элемента в очередь.
 - 2) Если строка начинается с -, то вызывается метод pop для удаления первого элемента из очереди.
 - 3) В остальных случаях вызывается метод get_min для получения текущего минимального значения, которое затем выводится с помощью write.

Результат работы программы:

Входные данные:

```
+ 1
```

```
?  
+ 10  
?  
-  
?  
-
```

Выходные данные:

```
1  
1  
10
```

Тесты:

```
import unittest  
from utils import memory_data, time_data  
from lab4.task6.src.queue_with_min import main, QueueWithMin  
  
class TestQueueWithMin(unittest.TestCase):  
    def setUp(self) -> None:  
        self.queue = QueueWithMin()  
  
    def test_should_check_get_min(self):  
        # given  
        self.queue.push(10)  
        expected_result = 10  
  
        # when  
        result = self.queue.get_min()  
  
        # then  
        self.assertEqual(result, expected_result)  
  
    def test_should_check_get_min_and_pop(self):  
        # given  
        self.queue.push(10)  
        self.queue.push(20)  
  
        # when (удалили число 10 - текущий минимум)  
        self.queue.pop()  
  
        # then  
        self.assertEqual(self.queue.get_min(), 20)  
  
    def test_should_check_time_data(self):  
        # given  
        expected_time = 2  
  
        # when  
        time = time_data(main)  
  
        # then  
        self.assertTrue(time < expected_time)  
  
    def test_should_check_memory_data(self):  
        # given
```

```
expected_memory = 256

# when
current, peak = memory_data(main)

# then
self.assertTrue(current < expected_memory)
self.assertTrue(peak < expected_memory)

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

Вывод по задаче:

В ходе решения данной задачи мы реализовали очередь с дополнительной функцией поиска и вывода минимального элемента.

Задание №9. Поликлиника

9 задача. Поликлиника

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

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

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

Решение:

```
from utils import write, read

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

class Queue:
    def __init__(self):
        self.head = None
        self.middle = None
        self.end = None
```

```

        self.__length = 0

def __len__(self):
    return self.__length

def push_to_end(self, item):
    node = Node(item)
    node.previous = self.end
    if node.previous is not None:
        node.previous.next = node
        if self.__length % 2 == 0:
            self.middle = self.middle.next
    else:
        self.head = self.middle = node
    self.end = node
    self.__length += 1

def push_to_middle(self, item):
    node = Node(item)
    node.previous = self.middle
    if self.middle is not None:
        node.next = self.middle.next
        self.middle.next = node
        if node.next is not None:
            node.next.previous = node
        else:
            self.end = node
        if self.__length % 2 == 0:
            self.middle = self.middle.next
    else:
        self.head = self.middle = self.end = node
    self.__length += 1

def pop_from_head(self):
    if self.__length == 0:
        raise IndexError
    if self.__length % 2 == 0:
        self.middle = self.middle.next
    pop_node = self.head
    self.head = self.head.next
    self.__length -= 1
    if self.__length == 0:
        self.middle = self.end = None
    return pop_node.value

def print(self):
    node = self.head
    while node is not None:
        print(node, end=' ')
        node = node.next
    print()

def main():
    queue = Queue()
    write(end='')

```

```

for line in read(type_convert=str):
    if line[0] == '+':
        queue.push_to_end(line[1])
    elif line[0] == '*':
        queue.push_to_middle(line[1])
    else:
        write(queue.pop_from_head(), sep='\n', to_end=True)

if __name__ == '__main__':
    main()

```

1. Создание класса Queue - очереди, реализованной на базе двусвязного списка.
2. Метод `__len__` позволяет использовать встроенную функцию `len()` для определения длины очереди.
3. Метод `push_to_end` добавляет новый элемент в конец очереди.
4. Метод `push_to_middle` добавляет новый элемент в середину очереди.
5. Метод `pop_from_head` удаляет элемент из начала очереди и возвращает его значение.
6. Метод `print` печатает содержимое очереди. Он проходит по всем узлам от головы до конца и выводит каждое значение.
7. Основная программа построчно считывает и обрабатывает входные данные:
 - 1) Если в начале строки "+", то элемент добавляется в конец очереди.
 - 2) Если в начале строки "*", то элемент добавляется в середину очереди.
 - 3) В остальных случаях производится извлечение элемента из начала очереди и его вывод.

Результат работы программы:

Входные данные:

```

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

```

Выходные данные:

1
3
2
5
4

Тесты:

```
import unittest
from utils import memory_data, time_data
from lab4.task9.src.polyclinic import main, Queue

class TestPolyclinic(unittest.TestCase):

    def setUp(self) -> None:
        self.queue = Queue()

    def test_should_check_length_initial(self):
        # given
        expected_len = 0

        # when
        queue_len = len(self.queue)

        # then
        self.assertEqual(queue_len, expected_len)

    def test_should_check_push_to_end(self):
        # when (step 1)
        self.queue.push_to_end(10)

        # then
        self.assertEqual(len(self.queue), 1)
        self.assertEqual(self.queue.head.value, 10)
        self.assertEqual(self.queue.middle.value, 10)
        self.assertEqual(self.queue.end.value, 10)

        # when (step 2)
        self.queue.push_to_end(20)

        # then
        self.assertEqual(len(self.queue), 2)
        self.assertEqual(self.queue.head.value, 10)
        self.assertEqual(self.queue.middle.value, 10)
        self.assertEqual(self.queue.end.value, 20)

        # when (step 3)
        self.queue.push_to_end(30)

        # then
        self.assertEqual(len(self.queue), 3)
        self.assertEqual(self.queue.head.value, 10)
        self.assertEqual(self.queue.middle.value, 20)
        self.assertEqual(self.queue.end.value, 30)
```

```

def test_should_check_push_to_middle(self):
    # given
    self.queue.push_to_end(10)
    self.queue.push_to_end(20)
    self.queue.push_to_end(40)
    self.queue.push_to_end(50)

    # when (step 1)
    self.queue.push_to_middle(30)

    # then
    self.assertEqual(len(self.queue), 5)
    self.assertEqual(self.queue.head.value, 10)
    self.assertEqual(self.queue.middle.value, 30)
    self.assertEqual(self.queue.end.value, 50)

    # when (step 2)
    self.queue.push_to_middle(31)

    # then
    self.assertEqual(len(self.queue), 6)
    self.assertEqual(self.queue.head.value, 10)
    self.assertEqual(self.queue.middle.value, 30)
    self.assertEqual(self.queue.middle.next.value, 31)
    self.assertEqual(self.queue.end.value, 50)

def test_should_check_pop_from_head(self):
    # given
    self.queue.push_to_end(10)
    self.queue.push_to_end(20)
    self.queue.push_to_end(30)

    # when (step 1)
    self.assertEqual(self.queue.pop_from_head(), 10)

    # then
    self.assertEqual(len(self.queue), 2)
    self.assertEqual(self.queue.head.value, 20)
    self.assertEqual(self.queue.middle.value, 20)
    self.assertEqual(self.queue.end.value, 30)

    # when (step 2)
    self.assertEqual(self.queue.pop_from_head(), 20)

    # then
    self.assertEqual(len(self.queue), 1)
    self.assertEqual(self.queue.head.value, 30)
    self.assertEqual(self.queue.middle.value, 30)
    self.assertEqual(self.queue.end.value, 30)

    # when (step 3)
    self.assertEqual(self.queue.pop_from_head(), 30)

    # then
    self.assertEqual(len(self.queue), 0)
    self.assertIsNone(self.queue.head)

```

```

        self.assertIsNone(self.queue.middle)
        self.assertIsNone(self.queue.end)

    def test_should_check_pop_from_empty_queue(self):
        with self.assertRaises(IndexError):
            self.queue.pop_from_head()

    def test_should_check_time_data(self):
        # given
        expected_time = 2

        # when
        time = time_data(main)

        # then
        self.assertTrue(time < expected_time)

    def test_should_check_memory_data(self):
        # given
        expected_memory = 256

        # when
        current, peak = memory_data(main)

        # then
        self.assertTrue(current < expected_memory)
        self.assertTrue(peak < expected_memory)

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

```

Вывод по задаче:

В ходе решения данной задачи мы реализовали очередь с функциями вставки элемента в середину, печати содержимого очереди.

Дополнительные задачи

Задание №8. Постфиксная запись

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 - *	

Решение:

```
from lab4.task13.src.stack import Stack
from utils import read, write

def addition(x, y): return x + y
def subtraction(x, y): return x - y
def multiplication(x, y): return x * y

operators = {'+': addition, '-': subtraction, '*': multiplication}

def postfix_notation(expression):
    stack = Stack()
    for operator in expression:
        if operator in operators:
            y, x = stack.pop(), stack.pop()
            stack.push(operators[operator](x, y))
        else:
            try:
```

```

        stack.push(int(operator))
    except ValueError:
        raise KeyError()
    return stack.pop()

def main():
    expression, = read(type_convert=str)
    write(postfix_notation(expression))

if __name__ == '__main__':
    main()

```

1. Определяются три простые функции для выполнения базовых арифметических операций: сложения, вычитания и умножения.
2. Создан словарь `operators`, где ключами являются символы операторов (+, -, *), а значениями — соответствующие им функции.
3. Создаётся пустой стек `stack` для хранения промежуточных значений (на основе реализованного в задаче 13 класса `Stack`).
4. Проходим циклом по символам выражения:
 - 1) Если символ является оператором (+, -, *), то из стека извлекаются два последних числа (они будут операндами), и к ним применяется соответствующая оператору функция. Результат снова помещается в стек.
 - 2) Если символ не является оператором, предполагается, что это цифра. Пробуем преобразовать её в целое число и поместить в стек. Если преобразование невозможно (например, если встретилась недопустимая символ), выводим ошибку `KeyError`.

Результат работы программы:

Входные данные:

```
8 9 + 1 7 - *
```

Выходные данные:

```
-102
```

Тесты:

```

import unittest
from utils import memory_data, time_data
from lab4.task8.src.postfix_notation import main, postfix_notation

class TestPostfixNotation(unittest.TestCase):
    def test_should_check_addition(self):
        # given
        expression = ['18', '21', '+']

```

```

        expected_result = 39

        # when
        result = postfix_notation(expression)

        # then
        self.assertEqual(result, expected_result)

    def test_should_check_multiplication(self):
        # given
        expression = ['10', '2', '*']
        expected_result = 20

        # when
        result = postfix_notation(expression)

        # then
        self.assertEqual(result, expected_result)

    def test_should_check_subtraction(self):
        # given
        expression = ['57', '14', '-']
        expected_result = 43

        # when
        result = postfix_notation(expression)

        # then
        self.assertEqual(result, expected_result)

    def test_should_check_complex_expression(self):
        # given
        expression = ['8', '9', '+', '1', '7', '-', '*']
        expected_result = -102

        # when
        result = postfix_notation(expression)

        # then
        self.assertEqual(result, expected_result)

    def test_should_check_time_data(self):
        # given
        expected_time = 2

        # when
        time = time_data(main)

        # then
        self.assertTrue(time < expected_time)

    def test_should_check_memory_data(self):
        # given
        expected_memory = 256

        # when

```

```
        current, peak = memory_data(main)

        # then
        self.assertTrue(current < expected_memory)
        self.assertTrue(peak < expected_memory)

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

Вывод по задаче:

В ходе решения данной задачи мы реализовали алгоритм вычисления математического выражения в постфиксной записи.

Задание №13. Реализация стека, очереди и связанных списков

13 задача★. Реализация стека, очереди и связанных списков

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.
3. Реализуйте односвязный список с функциями вывода содержимого списка, добавления элемента в начало списка, удаления элемента с начала списка, добавления и удаления элемента *после* заданного элемента (key); поиска элемента в списке.
4. Реализуйте двусвязный список с функциями вывода содержимого списка, добавления и удаления элемента с начала списка, добавления и удаления элемента с конца списка, добавления и удаления элемента *до* заданного элемента (key); поиска элемента в списке.

В рамках данной задачи необходимо реализовать минимум два пункта из четырёх. Здесь приведены решения для пунктов 1 и 3.

Решение:

Реализация стека:

```
from lab4.task13.src.singly_linked_list import SinglyLinkedList
from typing import TypeVar
T = TypeVar('T')

class Stack:
    def __init__(self):
        self.stack = SinglyLinkedList()

    def is_empty(self):
        return len(self.stack) == 0

    def push(self, item: T):
        self.stack.push(item)

    def pop(self):
        return self.stack.pop()
```

1. Класс Stack инициализируется созданием объекта SinglyLinkedList (пункт 3 данной задачи), который будет служить основой для реализации стека.
2. Метод is_empty проверяет, пуст ли стек. Для этого он вызывает функцию len для связанного списка и проверяет, равно ли его длина нулю. Если да, то стек считается пустым.
3. Метод push добавляет элемент в стек. Для этого он просто передает элемент методу push связанного списка.

4. Метод pop удаляет верхний элемент из стека и возвращает его.

Реализация односвязного списка:

```
from typing import TypeVar
T = TypeVar('T')

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

class SinglyLinkedList:
    def __init__(self):
        self.head = None
        self.__length = 0

    def __len__(self):
        return self.__length

    def push(self, value: T):
        node = SinglyLinkedListNode(value)
        node.next = self.head
        self.head = node
        self.__length += 1

    def pop(self):
        if self.__length == 0:
            raise IndexError()
        pop_node = self.head
        self.head = self.head.next
        self.__length -= 1
        return pop_node.value

    def find(self, value: T):
        current_node = self.head
        while current_node is not None:
            if current_node.value == value:
                return current_node
            current_node = current_node.next
        raise ValueError()

    def remove_after(self, key: T):
        current_node = self.find(key)
        if current_node.next is None:
            raise IndexError()
        removed_node = current_node.next
        current_node.next = current_node.next.next
        self.__length -= 1
        return removed_node.value

    def print_stack(self):
        current_node = self.head
        while current_node is not None:
```

```
print(current_node.value, end=' ')
current_node = current_node.next
print()
```

1. Класс `SinglyLinkedListNode` представляет собой узел односвязного списка.
2. Класс `SinglyLinkedList` представляет собой сам односвязный список.
3. Метод `__len__` позволяет использовать встроенную функцию `len()` для получения текущей длины списка.
4. Метод `push` добавляет новый узел в начало списка.
5. Метод `pop` удаляет и возвращает значение головного узла.
6. Метод `find` ищет узел с указанным значением в списке.
7. Метод `remove_after` удаляет узел, следующий за узлом с указанным ключом.
8. Метод `print_stack` выводит все значения узлов списка.

Тесты:

Для стека:

```
import unittest
from lab4.task13.src.stack import Stack

class TestStack(unittest.TestCase):

    def setUp(self):
        self.stack = Stack()

    def test_should_check_push(self):
        # given
        expected_len = 3

        # when
        self.stack.push(1)
        self.stack.push(2)
        self.stack.push(3)

        result = len(self.stack.stack)

        # then
        self.assertEqual(result, expected_len)

    def test_should_check_is_empty(self):
        self.assertTrue(self.stack.is_empty())

        # when
        self.stack.push(1)

        # then
        self.assertFalse(self.stack.is_empty())
```

```

def test_should_check_pop(self):
    # given
    self.stack.push(1)
    self.stack.push(2)

    # when
    self.assertEqual(self.stack.pop(), 2)
    self.assertEqual(self.stack.pop(), 1)

    # then
    self.assertTrue(self.stack.is_empty())

def test_should_check_pop_from_empty_stack(self):
    with self.assertRaises(IndexError):
        self.stack.pop()

def test_should_check_len(self):
    self.assertTrue(self.stack.is_empty())

    # when (step 1)
    self.stack.push(5)

    # then
    self.assertEqual(len(self.stack.stack), 1)

    # when (step 2)
    self.stack.pop()

    # then
    self.assertTrue(self.stack.is_empty())

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

```

Для односвязного списка:

```

import unittest
from lab4.task13.src.singly_linked_list import SinglyLinkedList

class TestSinglyLinkedList(unittest.TestCase):
    def setUp(self):
        self.linked_list = SinglyLinkedList()

    def test_should_check_initialization(self):
        self.assertEqual(len(self.linked_list), 0)
        self.assertIsNone(self.linked_list.head)

    def test_should_check_push(self):
        # when (step 1)
        self.linked_list.push(10)

        # then
        self.assertEqual(len(self.linked_list), 1)

```

```

self.assertEqual(self.linked_list.head.value, 10)

# when (step 2)
self.linked_list.push(20)

# then
self.assertEqual(len(self.linked_list), 2)
self.assertEqual(self.linked_list.head.value, 20)

def test_should_check_pop(self):
    # given
    with self.assertRaises(IndexError):
        self.linked_list.pop()

    self.linked_list.push(10)
    self.linked_list.push(20)
    self.linked_list.push(30)

    # when (step 1)
    self.assertEqual(self.linked_list.pop(), 30)

    # then
    self.assertEqual(len(self.linked_list), 2)

    # when (step 2)
    self.assertEqual(self.linked_list.pop(), 20)

    # then
    self.assertEqual(len(self.linked_list), 1)

    # when (step 3)
    self.assertEqual(self.linked_list.pop(), 10)

    # then
    self.assertEqual(len(self.linked_list), 0)

def test_should_check_find(self):
    # given
    self.linked_list.push(10)
    self.linked_list.push(20)
    self.linked_list.push(30)

    # when
    result = self.linked_list.find(20)

    # then
    self.assertIsNotNone(result)
    self.assertEqual(self.linked_list.find(20).value, 20)

def test_should_check_remove_after(self):
    # given
    self.linked_list.push(10)
    self.linked_list.push(20)
    self.linked_list.push(30)

    # when

```

```
        result = self.linked_list.remove_after(30)

        # then
        self.assertEqual(result, 20)

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

Вывод по задаче:

В ходе решения данной задачи мы реализовали стек на основе односвязного списка. Стек поддерживает основные операции: проверка на пустоту, добавление и удаление элементов. Также реализован односвязный список с основными операциями: добавление узла в начало, удаление узла из начала, поиск узла по значению, удаление узла после указанного ключа и вывод списка.

Вывод

В ходе выполнения лабораторной работы №4 мы изучили и реализовали такие структуры данных, как стек, очередь и связанный список.