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 21

Выполнил:

Ступичев М. Н.

К3139

Проверила:

…

Санкт-Петербург

2024 г.

# Содержание отчета

[Содержание отчета 2](#_Toc184924841)

[Задачи по варианту 3](#_Toc184924842)

[Задача №1. Сортировка слиянием. 3](#_Toc184924843)

[Задача №3. Скобочная последовательность. Версия 1. 4](#_Toc184924844)

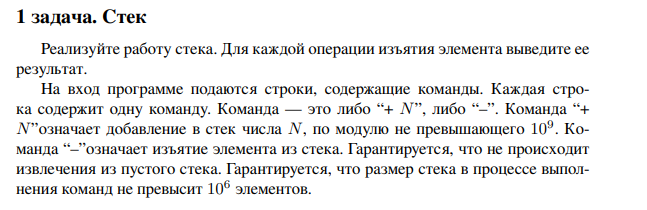
[Задача №5. Стек с максимумом. 6](#_Toc184924845)

[Задача №8. Постфиксная запись. 7](#_Toc184924846)

[Вывод 9](#_Toc184924847)

# Задачи по варианту

## Задача №1. Сортировка слиянием.



Решение:

def main():

   commands = read(type\_convert=str)

   write("", end="")

   stack = deque()

   for command in commands:

       if command[0] == '+':

           stack.append(command[1])

       elif command[0] == '-':

           write(stack.pop(), to\_end=True)

​Текстовое объяснение решения.

Этот код реализует простой стековый автомат, обрабатывающий команды из входного потока. Разберем его по частям:

1. **commands = read(type\_convert=str)**: Эта строка считывает информацию из заданного файла,  type\_convert=str указывает, что считанные данные должны быть преобразованы в строки. Результат — список команд (строк).
2. **write("", end="")**: Эта строка вызывает функцию write, вероятно, для вывода в выходной поток (файл или стандартный вывод). end="" указывает, что после вывода не должно быть символа новой строки. Этот вызов, скорее всего, очищает выходной поток.
3. **stack = deque()**: Создается стек (используется deque из модуля collections, что эффективно для операций добавления и удаления элементов с обоих концов).
4. **for command in commands**: Цикл проходит по каждой команде из списка commands.
5. **if command[0] == '+':**: Если первая буква команды — ‘+’, то это команда добавления. command[1] (второй символ команды) добавляется в стек.
6. **elif command[0] == '-':**: Если первая буква команды — ‘-‘, то это команда удаления. stack.pop() извлекает и возвращает последний элемент из стека, write(stack.pop(), to\_end=True) выводит этот элемент в выходной поток. to\_end=True предполагает, что вывод происходит в конец выходного потока.

**Асимптотическая сложность:**

Асимптотическая сложность определяется количеством команд. Каждый элемент обрабатывается за O(1) времени (постоянное время), поэтому общая сложность — **O(n)**, где n — количество команд во входном потоке. Пространственная сложность также **O(n)** в худшем случае (если все команды — добавление).

**Пример:**

Если commands = [‘+a’, ‘+b’, ‘-‘,’+c’, ‘-‘,’-‘], то:

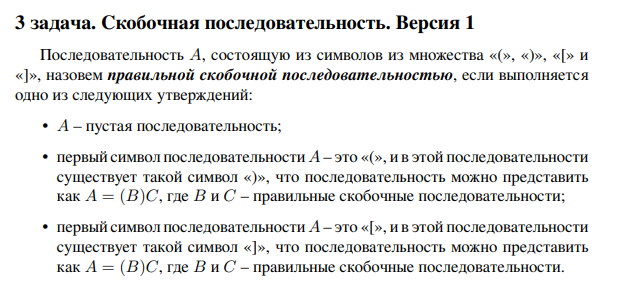
1. ‘+a’: стек: [‘a’]
2. ‘+b’: стек: [‘a’, ‘b’]
3. ’-‘: вывод: ‘b’, стек: [‘a’]
4. ‘+c’: стек: [‘a’, ‘c’]
5. ’-‘: вывод: ‘c’, стек: [‘a’]
6. ’-‘: вывод: ‘a’, стек: []

Выходной поток: ‘bca’

Вывод по задаче:

Реализована работа стека.

## Задача №3. Скобочная последовательность. Версия 1.



Решение:

def check\_right\_brackets\_sequence(brackets: str):

   stack = deque()

   for bracket in brackets:

       if bracket in ')]':

           if len(stack) == 0:

               return False

​

           last = stack.pop()

           if bracket == ')' and last == '[' or bracket == ']' and last == '(':

               return False

       else:

           stack.append(bracket)

​

   if len(stack) != 0:

       return False

   return True​​

​

Текстовое объяснение решения.

Функция check\_right\_brackets\_sequence проверяет, является ли данная последовательность скобок правильной. Она использует стек для отслеживания открытых скобок.

Разберем её работу:

1. **stack = deque()**: Создается пустой стек для хранения открытых скобок. Использование deque обеспечивает эффективное добавление и удаление элементов.
2. **for bracket in brackets**: Цикл перебирает каждый символ в строке brackets.
3. **if bracket in ')]'**: Если символ является закрывающей скобкой () или ]), то:
   * **if len(stack) == 0**: Проверяется, пуст ли стек. Если пуст, значит, встретилась закрывающая скобка без соответствующей открывающей, функция возвращает False.
   * **last = stack.pop()**: Извлекается последняя открывающая скобка из стека.
   * **if bracket == ')' and last == '[' or bracket == ']' and last == '('**: Проверяется соответствие открывающей и закрывающей скобок. Если тип скобок не совпадает (например, ) и [ ), функция возвращает False.
4. **else**: Если символ является открывающей скобкой (( или [ ), то он добавляется в стек: **stack.append(bracket)**.
5. **if len(stack) != 0**: После обработки всей строки, если стек не пуст, значит, есть несоответствие открывающих и закрывающих скобок, функция возвращает False.
6. **return True**: Если стек пуст и все скобки соответствуют друг другу, функция возвращает True.

**Асимптотическая сложность:**

Функция проходит по каждому символу в строке один раз. Все операции со стеком выполняются за O(1) времени. Поэтому общая асимптотическая сложность — **O(n)**, где n — длина строки brackets. Пространственная сложность также O(n) в худшем случае (если все скобки - открывающие).

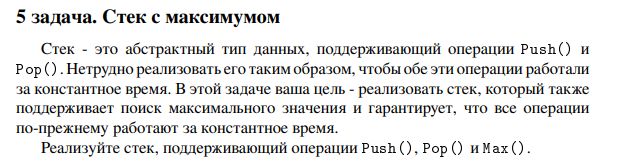
**Пример:**

* check\_right\_brackets\_sequence("([])") вернет True.
* check\_right\_brackets\_sequence("([)]") вернет False.
* check\_right\_brackets\_sequence("((()") вернет False.
* check\_right\_brackets\_sequence(")") вернет False.

Вывод по задаче:

Реализован алгоритм проверки скобочной последовательности на правильность.

## Задача №5. Стек с максимумом.



Решение:

class MaxStack:

   def \_\_init\_\_(self):

       self.stack = deque()

       self.cur\_max = float('-inf')

​

   def push(self, value: int):

       self.stack.append((value, self.cur\_max))

       if self.cur\_max < value:

           self.cur\_max = value

​

   def pop(self):

       last, last\_max = self.stack.pop()

​

       if last == self.cur\_max:

           self.cur\_max = last\_max

       return last

​

   def max(self):

       if len(self.stack) == 0:

           raise IndexError

       return self.cur\_max​​

Текстовое объяснение решения.

Класс MaxStack реализует стек, который позволяет за O(1) времени получать максимальное значение среди элементов в стеке. Это достигается за счет хранения дополнительной информации вместе с каждым элементом.

Разберем методы:

* **\_\_init\_\_(self)**: Конструктор класса. Инициализируется стек stack (используется deque для эффективности) и переменная cur\_max, которая хранит текущий максимальный элемент стека. Изначально cur\_max устанавливается в отрицательную бесконечность.
* **push(self, value: int)**: Добавляет элемент value в стек. В стек добавляется кортеж (value, self.cur\_max), где self.cur\_max — текущий максимум *до* добавления нового элемента. Если value больше текущего максимума, то cur\_max обновляется.
* **pop(self)**: Извлекает и возвращает последний добавленный элемент из стека. При извлечении, если извлеченный элемент был равен текущему максимуму, то cur\_max обновляется до значения максимума из предыдущего элемента стека (хранящегося во втором элементе кортежа).
* **max(self)**: Возвращает текущий максимальный элемент стека. Если стек пуст, выбрасывается исключение IndexError.

**Асимптотическая сложность:**

* **push**: O(1)
* **pop**: O(1)
* **max**: O(1)

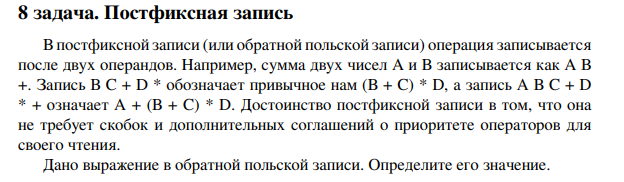
**Преимущества:**

* Получение максимального элемента за O(1) время. Это значительно быстрее, чем проход по всему стеку (O(n)).
* Эффективное использование памяти, так как хранится только дополнительная информация о максимальном элементе, а не вся история максимумов.

Вывод по задаче:

Реализован класс стека с возможностью вывода максимального элемента за константу.

## Задача №8. Постфиксная запись.



Решение:

def calculate\_polish\_notation(expression: list) -> int:

   stack = deque()

   for element in expression:

       if element in "+-\*":

           num2 = stack.pop()

           num1 = stack.pop()

           result = None

           if element == '+':

               result = num1 + num2

           elif element == '-':

               result = num1 - num2

           elif element == '\*':

               result = num1 \* num2

           stack.append(result)

       else:

           stack.append(int(element))

   return stack.pop()​​

Текстовое объяснение решения.

Функция calculate\_polish\_notation вычисляет значение выражения, записанного в обратной польской нотации (ОПН, или постфиксной нотации).

Разберем ее работу пошагово:

1. **stack = deque()**: Создается пустой стек для хранения операндов. Используется deque для эффективности операций добавления и удаления.
2. **for element in expression**: Цикл проходит по каждому элементу выражения (в ОПН).
3. **if element in "+-\*"**: Если элемент является оператором (+, -, \*), то:
   * **num2 = stack.pop()**: Извлекается второй операнд из стека.
   * **num1 = stack.pop()**: Извлекается первый операнд из стека.
   * Вычисляется результат операции (result) в зависимости от оператора.
   * **stack.append(result)**: Результат операции добавляется в стек.
4. **else**: Если элемент не является оператором, то это операнд.
   * **stack.append(int(element))**: Элемент преобразуется в целое число и добавляется в стек.
5. **return stack.pop()**: После обработки всех элементов в стеке останется только один элемент — результат вычисления выражения. Этот элемент извлекается и возвращается.

**Пример:**

Пусть expression = [‘3’, ‘4’, ‘+’, ‘2’, ‘\*’].

1. ‘3’: стек: [3]
2. ‘4’: стек: [3, 4]
3. ’+’: стек: [7] (3 + 4)
4. ‘2’: стек: [7, 2]
5. ’\*’: стек: [14] (7 \* 2)

Результат: 14

**Асимптотическая сложность:**

Функция проходит по каждому элементу выражения один раз. Все операции со стеком (добавление, удаление) выполняются за O(1) времени. Поэтому общая асимптотическая сложность — **O(n)**, где n — длина выражения. Пространственная сложность также **O(n)** в худшем случае (если все операнды).

Вывод по задаче:

Реализован алгоритм подсчета значения выражения в постфиксной форме.

# Вывод

Решены 3 задачи из 21 варианта и 8 задача вместо 9.