## 5. Стек

Стек -- это простой тип данных, который работает по принципу "последний вошёл -- первый вышел". Стек поддерживает всего две операции -- push (втолкнуть) и pop (вытолкнуть) и представляет собой линейное хранилище элементов, в котором есть только один вход. Операция push помещает элемент в этот вход -- говорят, на самый верх стека. Операция pop извлекает последний втолкнутый в стек элемент. Кроме того, стеки обычно поддерживают для удобства операцию peek() -- получить верхний элемент стека, но не удалять его.

Говорят ещё, что стек работает по схемам

LIFO = Last in First out (последний вошёл, первый вышел)

FILO = First in Last out (первый вошёл, последний вышел)

Например, если мы делаем последовательно push(1), push(2) и push(3) для исходно пустого стека, то вызовы pop() вернут 3, потом 2, и потом 1. Сам стек в итоге снова окажется пустым.

Где востребованы LIFO/FILO? Во множестве самых разных задач, начиная от бэктрекинга/отката (когда например мы храним в стеке операции, выполненные в текстовом редакторе, чтобы их можно было корректно откатывать) до рекурсивного разбора абстрактных синтаксических деревьев.

Реализация стека проста и во многом интуитивно понятна. Мы можем использовать для хранения стека самые разные структуры данных, обычно уже существующие в языке: как правило, это динамический список, который расширяется новым элементом простой операцией добавления "в хвост". Мы условимся верхушкой стека считать последний элемент в списке. Соответственно, операция pop() должна выдавать этот последний элемент, удаляя его из списка.

stack = Stack()

stack.push(**1**)

stack.push("2")

stack.push(**3.14**)

**while** stack.size() > **0**:

stack.pop()

**Амортизационный анализ**

**Банковский метод** (или метод предоплаты), с которым мы знакомились при оценке реаллокации динамического массива из предыдущего занятия, относится к так называемому **амортизационному анализу (amortized analysis)**. Этот анализ занимается изучением подобных оценок для конкретного типа, конкретной структуры данных (точнее даже, для конкретной её реализации). Он специализируется на ситуациях, не когда все операции выполняются примерно одинаково, а когда наоборот, изредка возникают нагрузочные операции, но часто выполняются дешёвые в плане ресурсов действия.

В процессе амортизационного анализа мы каждой операции присваиваем некоторую **учётную стоимость (amortized cost)**, которая может отличаться от реальной, истинной. В методе предоплаты мы оценивали операции вставки в три раза больше, нежели они реально требуют, это и есть учётная стоимость. И хотя эта стоимость для разных операций может отличаться от реальной как в большую сторону, так и в меньшую, важно, чтобы истинная итоговая стоимость (как правило, длительность) серии операций над структурой данных не превышала учётной итоговой стоимости. Тогда считается, что учётные стоимости мы подобрали корректно.

Всего в амортизационном анализ применяются три метода. Банковский метод мы рассмотрели в динамическом массиве. Более простой **метод группировки** подразумевает, что если удобнее оценить не одну операцию, а их некоторую последовательность, в наихудшем случае, то мы получаем учётную стоимость простым делением суммарного значения на количество операций.

Если мы реализуем стек на основе связанного списка, где и добавление, и удаление элементов требуют O(1), то n операций над стеком потребуют в n раз больше времени. Метод группировки как раз исходит из общей оценки всех операций. Если например мы определили O(n\*n) (когда push и pop реализуются на основе более медленных структур, требующих O(n) на каждую операцию), то учётная стоимость будет n.

И третий метод -- это **метод потенциалов**. Он похож на банковский метод, только мы не подсчитываем стоимости операций дискретно, поштучно, а используем функцию так называемой "потенциальной энергии" (потенциал).

Стоимость, которую мы накапливали пошагово путём переоценки истинной стоимости каждой операции в более высокую сторону (учётная стоимость), теперь вычисляется потенциалом автоматически -- как функция от текущего состояния структуры данных. Учётная стоимость i-й операции теперь рассчитывается как истинная стоимость этой операции плюс потенциал i-й операции минус потенциал (i-1)-й операции (разность потенциалов). Когда разность потенциалов положительна, она отправляется в резерв, а когда отрицательна, то компенсируется из резерва.

Стандартная функция потенциала для стека -- это количество элементов в стеке. Тогда для i-й операции push её учётная стоимость будет равна:

1 (истинная стоимость) + i - (i-1) = 2

Для i-й операции pop, так как в результате количество элементов не увеличивается, а уменьшается, получаем

1 (истинная стоимость) + (i-1) - i = 0

Таким образом, наихудшая оценка одной операции O(1) -- не зависит от количества элементов, и получаем в итоге для n операций оценку O(n).

**Задания.**

1. Подберите подходящую динамическую структуру данных для хранения стека. Реализуйте методы size(), pop(), push() и peek().   
Добавьте тесты для каждого из этих четырёх методов.   
Оцените меру сложности для операций pop и push.

2. Переделайте реализацию стека так, чтобы она работала не с хвостом списка как с верхушкой стека, а с его головой.

3. Не запуская программу, скажите, как отработает такой цикл?

**while** stack.size() > **0**:

print(stack.pop())

print(stack.pop())

4. Оцените меру сложности для операций pop и push.

5. Напишите функцию, которая получает на вход строку, состоящую из открывающих и закрывающих скобок (например, "(()((())()))" или "(()()(()") и, используя только стек и оператор цикла, определите, сбалансированы ли скобки в этой строке. Сбалансированной считается последовательность, в которой каждой открывающей обязательно соответствует закрывающая, а каждой закрывающей -- открывающая скобки, то есть последовательности "())(" , "))((" или "((())" будут несбалансированы.

5 (необязательное). **бонус +300 золотых**  
Постфиксная запись выражения -- это запись, в которой порядок вычислений определяется не скобками и приоритетами, а только позицией элемента в выражении. Например, в выражениях разрешено использовать целые числа и операции + и \* . Тогда выражение

(1 + 2) \* 3

запишется как

1 2 + 3 \* (верхушка стека слева)

Такой стек обрабатывается следующим образом: берём с верхушки объект, если это число, сохраняем во втором стеке, а если операция, выполняем её над двумя верхними элементами второго стека и возвращаем её обратно во второй стек.

В нашем случае:

S1: 1 2 + 3 \*

S2:

S1: 2 + 3 \*

S2: 1

S1: + 3 \*

S2: 2 1

Берём операцию + и применяем её к содержимому S2:

S1: 3 \*

S2: 3

S1: \*

S2: 3 3

S1:

S2: 9

Можно ещё добавить операцию = , которая выдаёт содержимое второго стека как результат. Напишите функцию, которая с помощью двух стеков реализует вычисление подобных постфиксных выражений.

Рассчитайте с её помощью например такое выражение:

8 2 + 5 \* 9 + =

## заготовка класса для автоматической проверки

**class** **Stack**:

**def** **\_\_init\_\_**(self):

self.stack = []

**def** **size**(self):

**return** len(self.stack)

**def** **pop**(self):

# ваш код

**return** **None** # если стек пустой

**def** **push**(self, value):

# ваш код

**def** **peek**(self):

# ваш код

**return** **None** # если стек пустой