Лекция 4. Амортизированный анализ. Стек, очередь, дек.

1. Стек (Stack)

Стек – это абстрактный тип данных (АТД), представляющий собой коллекцию элементов, организованную по принципу **LIFO (Last In, First Out)** – последним пришёл, первым ушёл. Его можно представить как стопку тарелок: новую тарелку можно положить только сверху, и взять можно только верхнюю.

Основные операции:

1. push(x) – добавить элемент x на вершину стека.
2. pop() – удалить и вернуть элемент с вершины стека.
3. top() или peek() – вернуть элемент с вершины стека, не удаляя его (часто называется peek).
4. is_empty() – проверить, пуст ли стек.

Простая реализация на списке:

Давайте реализуем стек, используя список. Мы будем хранить элементы в списке и использовать переменную n (или просто полагаться на len(list)) для отслеживания текущего размера.

```python
class Stack:
    def __init__(self):
        self._data = []  # Пустой список для хранения элементов

    def push(self, x: int) -> None:
        """Добавляет элемент x на вершину стека."""
        self._data.append(x)  # O(1) амортизированно

    def pop(self) -> int:
        """Удаляет и возвращает элемент с вершины стека."""
        if self.is_empty():
            raise Exception("Стек пуст")
        return self._data.pop()  # O(1) амортизированно

    def top(self) -> int:
        """Возвращает элемент с вершины стека, не удаляя его."""
        if self.is_empty():
            raise Exception("Стек пуст")
        return self._data[-1]  # O(1)

    def is_empty(self) -> bool:
        """Проверяет, пуст ли стек."""
        return len(self._data) == 0


# Пример использования:
s = Stack()
s.push(1)
s.push(2)
s.push(3)
print(s.pop())  # 3
```

```
print(s.top())  # 2
print(s.pop())  # 2
```

В этой реализации операции push и pop в среднем выполняются за O(1). Однако, как мы увидим далее, это O(1) является **амортизированной** сложностью.

Применение стека:

Стек – фундаментальная структура данных с огромным количеством применений:

1. **Управление вызовами функций и рекурсия.** Когда вы вызываете функцию, её контекст (адрес возврата, локальные переменные) помещается в **стек вызовов**. При возврате из функции контекст снимается с вершины стека. Рекурсия – это просто частный случай использования стека вызовов.

2. **Анализ скобочных последовательностей.** Задача: проверить, правильно ли расставлены скобки (((())())(), и, например, для каждой открывающей скобки найти парную ей закрывающую.

Алгоритм: проходим по строке. При встрече открывающей скобки помещаем её индекс в стек. При встрече закрывающей скобки извлекаем индекс из стека – это и есть индекс парной открывающей скобки. Если стек пуст к концу обхода и не было попыток извлечь из пустого стека – последовательность правильная.

3. Алгоритм сортировки станционной железной дороги.

4. Обход деревьев и графов в глубину (DFS).

2. Очередь (Queue)

Очередь – это АТД, работающий по принципу **FIFO (First In, First Out)** – первым пришёл, первым ушёл. Как очередь в магазине.

Основные операции:

1. enqueue(x) или add(x) – добавить элемент x в конец очереди.
2. dequeue() или remove() – удалить и вернуть элемент из начала очереди.
3. peek() – вернуть элемент из начала очереди, не удаляя его.
4. is_empty() – проверить, пуста ли очередь.

Наивная реализация на массиве (проблематичная):

Попробуем реализовать очередь на списке, храня индексы начала (head) и конца (tail).

```
class NaiveQueue:
    def __init__(self, initial_capacity: int = 10):
        self._data = [None] * initial_capacity
        self._head = 0  # Индекс первого элемента
        self._tail = 0  # Индекс следующего свободного места
        self._size = 0  # Количество элементов

    def enqueue(self, x: int) -> None:
        """Добавляет элемент в конец очереди."""
        if self._tail == len(self._data):
            # Если массив заполнен до конца, нужно его расширить.
```

```
            # Пока просто оставим так, это создаёт проблему.
            self._resize(2 * len(self._data))
        self._data[self._tail] = x
        self._tail += 1
        self._size += 1

    def dequeue(self) -> int:
        """Удаляет и возвращает элемент из начала очереди."""
        if self.is_empty():
            raise Exception("Очередь пуста")
        x = self._data[self._head]
        self._head += 1
        self._size -= 1
        return x

    def _resize(self, new_capacity: int) -> None:
        """Создаёт новый массив большего размера и копирует в него
элементы."""
        old_data = self._data
        self._data = [None] * new_capacity
        # Копируем элементы от _head до _tail в начало нового массива
        for i in range(self._size):
            self._data[i] = old_data[self._head + i]
        self._head = 0
        self._tail = self._size

    def is_empty(self) -> bool:
        return self._size == 0


# Пример использования:
q = NaiveQueue()
q.enqueue(1)
q.enqueue(2)
q.enqueue(3)
print(q.dequeue())  # 1
print(q.dequeue())  # 2
q.enqueue(4)
print(q.dequeue())  # 3
print(q.dequeue())  # 4
```

Проблема. В этой реализации при многократных операциях enqueue и dequeue индекс _head будет постоянно увеличиваться. Мы будем постоянно перевыделять память и копировать данные в функцию _resize, даже если в начале массива есть свободное место (после dequeue). Фактически, наша очередь будет «ползти» вправо по памяти, пока не упрётся в конец выделенного массива.

Решение: кольцевой буфер (Circular Buffer)

Идея в том, чтобы считать массив замкнутым в кольцо. Когда tail доходит до конца массива, он переходит в его начало (если там есть свободное место после операций dequeue).

```python
class Queue:
    def __init__(self, initial_capacity: int = 10):
        self._data = [None] * initial_capacity
        self._head = 0
        self._tail = 0
        self._size = 0

    def enqueue(self, x: int) -> None:
        """Добавляет элемент в конец очереди."""
        if self._size == len(self._data):
            # Массив полон, нужно увеличить его в 2 раза
            self._resize(2 * len(self._data))

        self._data[self._tail] = x
        self._tail = (self._tail + 1) % len(self._data)  # Кольцевой переход
        self._size += 1

    def dequeue(self) -> int:
        """Удаляет и возвращает элемент из начала очереди."""
        if self.is_empty():
            raise Exception("Очередь пуста")
        x = self._data[self._head]
        self._head = (self._head + 1) % len(self._data)  # Кольцевой переход
        self._size -= 1
        return x

    def _resize(self, new_capacity: int) -> None:
        """Создаёт новый массив и копирует в него элементы, начиная с
_head."""
        old_data = self._data
        old_len = len(old_data)
        self._data = [None] * new_capacity
        # Копируем элементы, учитывая кольцевую структуру
        for i in range(self._size):
            self._data[i] = old_data[(self._head + i) % old_len]
        self._head = 0
        self._tail = self._size

    def is_empty(self) -> bool:
        return self._size == 0

    def peek(self) -> int:
        if self.is_empty():
            raise Exception("Очередь пуста")
        return self._data[self._head]
```

В этой реализации операции enqueue и dequeue работают за O(1) **амортизированно**, так как дорогая операция _resize происходит не при каждой вставке.

3. Дек (Deque, Double-Ended Queue)

Дек – это обобщение стека и очереди, позволяющее добавлять и удалять элементы с **обоих концов**.

Основные операции:

1. add_first(x) – добавить x в начало дека.
2. add_last(x) – добавить x в конец дека.
3. remove_first() – удалить и вернуть элемент из начала дека.
4. remove_last() – удалить и вернуть элемент из конца дека.

Реализация дека на кольцевом буфере аналогична очереди, но требует более аккуратной работы с индексами head и tail. Часто для реализации дека используют двусвязный список, где каждая операция выполняется за O(1) в худшем случае без необходимости перевыделения памяти.

4. Амортизированный анализ

Мы несколько раз упомянули **амортизированную сложность**. Давайте разберёмся, что это такое.

Амортизированный анализ – это метод анализа алгоритмов, который показывает среднее время выполнения операции в **худшем случае** в течение последовательности операций.

Зачем это нужно? Некоторые операции структуры данных могут быть дорогими (например, _resize в стеке), но если они происходят редко, то средняя стоимость всех операций в последовательности может быть небольшой.

Определение. Пусть есть структура данных и последовательность из n операций $op_1, op_2, \dots, op_n$. Пусть $T(op_i)$ – реальное время выполнения i-ой операции. Тогда **амортизированная стоимость** операции $A(op_i)$ – это такая величина, что:

$$\sum_{i=1}^{n} T(op_i) \leq \sum_{i=1}^{n} A(op_i)$$

Иными словами, суммарное реальное время выполнения всех операций не превышает суммы их амортизированных стоимостей.

Существует три основных метода амортизированного анализа:

1. Метод агрегирования (усреднения).

2. Метод бухгалтерского учёта.

3. Метод потенциалов.

Проанализируем операцию push в нашем первоначальном стеке (на списке), которая иногда приводит к дорогому копированию массива.

4.1. Метод агрегирования

Рассмотрим последовательность из n операций push. Пусть начальный размер массива равен 1.

- Первый push: стоимость 1 (записали элемент) + 0 (копирования не было) = 1.
- Второй push: массив заполнен (размер=1). Стоимость: 1 (запись) + 1 (копирование старого массива в новый размером 2) = 2.

• Третий push: массив заполнен (размер=2). Стоимость: 1 + 2 (копирование 2 элементов в массив размером 4) = 3.

• Четвёртый push: стоимость 1.

• Пятый push: массив заполнен (размер=4). Стоимость: 1 + 4 (копирование 4 элементов в массив размером 8) = 5.

Видна закономерность: дорогие операции, связанные с удвоением массива, происходят при push с номерами, равными степеням двойки: 1, 2, 4, 8, 16, ….

Общая стоимость n операций push:

$$T(n) = n + (1 + 2 + 4 + 8 + \ldots + 2^k),$$

где $2^k < n \leq 2^{k+1}$.

Сумма в скобках – это сумма геометрической прогрессии, она меньше $2n$.

Таким образом, $T(n) < n + 2n = 3n$.

Средняя стоимость одной операции: $\frac{T(n)}{n} < 3$, то есть O(1).

4.2. Метод бухгалтерского учёта

В этом методе мы назначаем каждой операции её **амортизированную стоимость** $A(op)$. Если реальная стоимость операции $T(op)$ меньше $A(op)$, разница зачисляется на счёт операции как «кредит» (учётная монетка). Если $T(op) > A(op)$, разница покрывается за счёт ранее накопленных кредитов.

Для операции push в стеке назначим амортизированную стоимость A(push) = 3 (монетки).

• Реальная стоимость T(push) = 1 (простая вставка).

• Из амортизированной стоимости 3 тратится 1 монетка на саму вставку. Оставшиеся 2 монетки остаются как кредит «прикреплённый» к вставленному элементу. Одна монетка будет потрачена на его будущее копирование при расширении массива, а вторая – на копирование какого-то старого элемента, который уже был в массиве.

Когда происходит дорогое расширение массива и копирование k элементов, на это уже заготовлено k * 2 кредитов (по 2 на каждый элемент). Этого хватает, так как реальная стоимость копирования k элементов равна k.

Поскольку для каждой операции push мы взимаем 3 монетки и никогда не уходим в минус, суммарная амортизированная стоимость n операций равна 3n, а значит, реальная стоимость O(n), а средняя – O(1).

4.3. Метод потенциалов

В этом методе мы вводим функцию **потенциала** $\Phi(D_i)$, которая отображает состояние структуры данных D после i-ой операции в вещественное число.

Амортизированная стоимость операции определяется как:

$$A(op_i) = T(op_i) + \Phi(D_i) - \Phi(D_{i-1}).$$

Суммируя амортизированные стоимости всех операций, получаем:

$$\sum A(op_i) = \sum T(op_i) + \Phi(D_n) - \Phi(D_0).$$

Мы хотим, чтобы $\sum T(op_i) \leq \sum A(op_i)$. Для этого нужно, чтобы $\Phi(D_n) \geq \Phi(D_0)$. Обычно выбирают $\Phi(D_0) = 0$ и $\Phi(D_n) \geq 0$ для всех i.

Для стека определим потенциал как Φ = 2 * (number_of_items_in_array) - (array_capacity)

Или более популярный вариант: Φ = 2 * (size) - capacity (где size – количество элементов, capacity – размер массива).

1. $\Phi(D_0) = 2 \cdot 0 - 0 = 0$.

2. Рассмотрим операцию push:

• Если расширения не происходит:

$$T(push) = 1,$$
$$\Delta\Phi = \Phi_i - \Phi_{i-1} = (2 \cdot (s+1) - c) - (2 \cdot s - c) = 2,$$
$$A(push) = 1 + 2 = 3.$$

• Если расширение происходит (т.е. s == c):

$T(push) = 1 + s$ (1 для вставки + s для копирования)

Старая ёмкость c, новая $2c$.

$$\Delta\Phi = (2 \cdot (s+1) - 2c) - (2s - c) = (2s + 2 - 2c) - (2s - c) = 2 - c,$$
$$A(push) = (1 + s) + (2 - c) = 3 + s - c.$$

Но в момент расширения $s = c$, следовательно:

$$A(push) = 3 + c - c = 3.$$

В обоих случаях амортизированная стоимость $A(push) \leq 3$.

Потенциал всегда неотрицателен? После расширения $\Phi = 2 \cdot (c+1) - 2c = 2 > 0$. После серии добавлений без расширения потенциал растёт, что и «оплачивает» будущее расширение.

Итог: $\sum T(op_i) \leq \sum A(op_i) = 3n$, следовательно, средняя стоимость O(1).

5. Очередь на двух стеках

Реализуем очередь, используя два стека. Эта реализация также демонстрирует идею амортизированного анализа.

Идея. Один стек (stack_in) будем использовать для добавления элементов (enqueue). Другой стек (stack_out) – для извлечения (dequeue). Когда stack_out становится пустым, мы «переливаем» все элементы из stack_in в stack_out (при этом элементы в stack_out окажутся в обратном порядке, что и нужно для операции dequeue – первый вошедший элемент окажется на вершине stack_out).

```
class QueueTwoStacks:
    def __init__(self):
        self._stack_in = []  # Стек для добавления
        self._stack_out = []  # Стек для извлечения

    def enqueue(self, x: int) -> None:
        """Добавляет элемент в конец очереди. Сложность: O(1)."""
```

```
        self._stack_in.append(x)

    def dequeue(self) -> int:
        """Удаляет и возвращает элемент из начала очереди. Амортизированная
сложность: O(1)."""
        if self.is_empty():
            raise Exception("Очередь пуста")
        if not self._stack_out:
            # Если stack_out пуст, переливаем в него всё из stack_in
            while self._stack_in:
                self._stack_out.append(self._stack_in.pop())
        return self._stack_out.pop()

    def is_empty(self) -> bool:
        return not (self._stack_in or self._stack_out)


# Пример использования:
q = QueueTwoStacks()
q.enqueue(1)
q.enqueue(2)
q.enqueue(3)
print(q.dequeue())  # 1
q.enqueue(4)
print(q.dequeue())  # 2
print(q.dequeue())  # 3
print(q.dequeue())  # 4
```

Амортизированный анализ операции dequeue:

Реальная стоимость операции dequeue:

1. Если stack_out не пуст: $T(dequeue) = 1$ (pop из stack_out).

2. Если stack_out пуст: $T(dequeue) = 1 + |stack\_in|$ (потребовалось перелить k элементов из stack_in, сделав k операций pop из stack_in и k операций push в stack_out, и затем один pop из stack_out).

Используя метод учёта, назначим амортизированную стоимость $A(enqueue) = 2, A(dequeue) = 2$.

1. При enqueue:

• $T(enqueue) = 1$ (простой push в stack_in).

• Мы взяли 2 монетки. Одну тратим на саму операцию. Вторую оставляем «прикреплённой» к добавленному элементу как кредит на его будущее перемещение из stack_in в stack_out.

2. При dequeue:

• Если stack_out не пуст: $T(dequeue) = 1$. Мы платим 2 монетки. Одну тратим на операцию pop, вторую просто «уничтожаем».

• Если stack_out пуст: $T(dequeue) = 1 + k$. На перемещение k элементов нужно k кредитов (по 1 на каждый элемент, ведь их перемещение – это два pop/push). У нас как раз есть k кредитов, оставленных этими элементами при enqueue. Дополнительно мы платим 2 монетки за сам dequeue: одну за финальный pop из stack_out, вторую так же «уничтожаем». Таким образом, общая стоимость покрывается.

Итог: амортизированная стоимость обеих операций равна O(1).