Лекция 2. Бинарная куча. Очередь с приоритетом. Пирамидальная сортировка.

1. Зачем нужны структуры данных?

В предыдущей лекции мы обсуждали RAM-модель, где память представляет собой бесконечный массив ячеек с константным временем доступа. Возникает закономерный вопрос: если данные можно просто положить в этот массив и читать их оттуда за O(1), зачем тогда изобретать какие-то сложные **структуры данных**?

Ответ заключается в том, что данные редко просто «лежат». Мы хотим выполнять над ними определённый набор **операций** (например, добавить новый элемент, найти минимальный, удалить максимальный, найти все элементы в заданном диапазоне и т.д.), и делать эти операции **максимально быстро**.

1. **Данные + операции = абстрактный тип данных (АТД).**
2. **Структура данных** – это конкретная реализация АТД в памяти машины (чаще всего на основе массивов или указателей).

Эффективность структуры данных измеряется временем работы каждой из предоставляемых ею операций. Не существует «идеальной» структуры данных, которая все операции делает за O(1). Выбор структуры данных – это всегда компромисс, зависящий от того, какие операции в вашей задаче выполняются чаще, а какие реже.

Очередь с приоритетом – яркий пример такого АТД. Она хранит мультимножество элементов.

Мультимножество – это множество, в котором элементы могут повторяться. То есть нас интересует не только факт наличия элемента, но и его кратность (сколько раз он встретился).

Элементы должны быть сравнимы между собой (должна существовать операция сравнения a < b или задан компаратор). Очередь с приоритетом поддерживает три основные операции:

1. insert(x) // $A = A \cup \{x\}$ (добавить новый элемент x в наше мультимножество A).
2. get_min() // $\min A$ (вернуть значение минимального элемента; иногда эту операцию называют find_min или peek).
3. remove_min() // $A = A \setminus \{\min A\}$ (удалить **один** минимальный элемент из множества; иногда называется extract_min).

Важно: get_min и remove_min – это разные операции. Часто реализации объединяют их в одну extract_min, которая возвращает минимум и сразу же его удаляет.

2. Наивные реализации очереди с приоритетом

Давайте попробуем реализовать этот АТД, просто храня данные в массиве, и проанализируем время работы операций.

Реализация v.0.1 «Наивный массив»

Храним элементы в массиве a[] размера n. Новые элементы добавляем в конец.

```python
def naive_insert(a, x):
    """Добавляем элемент в конец массива."""
    a.append(x)  # O(1) амортизированная сложность


def naive_get_min(a):
    """Чтобы найти минимум, необходимо просканировать весь массив."""
    if not a:
        raise Exception("Очередь пуста")
    min_val = a[0]
    for num in a:
        if num < min_val:
            min_val = num
    return min_val  # O(n)


def naive_remove_min(a):
    """Чтобы удалить минимум, необходимо найти его индекс, затем удалить."""
    if not a:
        raise Exception("Очередь пуста")
    min_idx = 0
    for i in range(1, len(a)):
        if a[i] < a[min_idx]:
            min_idx = i
    # Меняем найденный минимум с последним элементом и удаляем с конца
    min_val = a[min_idx]
    a[min_idx] = a[-1]
    a.pop()
    return min_val  # O(n)
```

Анализ:

1. insert(x): **O(1)** (амортизированно, так как list.append в Python иногда требует копирования массива, но в среднем это O(1)).
2. get_min(): **O(n)**.
3. remove_min(): **O(n)**.

Эта реализация плоха, если операции get_min или remove_min вызываются часто.

Реализация v.0.1.1 «Массив с запоминанием минимума»

Попробуем оптимизировать, сохраняя дополнительное состояние – индекс текущего минимума.

```python
def cached_min_insert(state, x):
    a, min_idx = state
    a.append(x)
    if min_idx == -1 or x < a[min_idx]:
        min_idx = len(a) - 1  # Новый элемент стал минимумом
    return a, min_idx  # O(1)


def cached_min_get_min(state):
    a, min_idx = state
```

```python
    if min_idx == -1:
        raise Exception("Очередь пуста")
    return a[min_idx]  # O(1)


def cached_min_remove_min(state):
    a, min_idx = state
    if min_idx == -1:
        raise Exception("Очередь пуста")
    min_val = a[min_idx]
    last_val = a.pop()
    if a:  # Если массив не пуст после удаления
        if min_idx < len(a):  # Если минимум был не последним
            a[min_idx] = last_val
        # После удаления и перестановки минимум мог испортиться
        min_idx = 0
        for i in range(1, len(a)):
            if a[i] < a[min_idx]:
                min_idx = i
    else:
        min_idx = -1
    return min_val, (a, min_idx)  # O(n)
```

Анализ:

1. insert(x): **O(1)**.
2. get_min(): **O(1)**.
3. remove_min(): **O(n)** (так как после удаления нам приходится заново искать минимум во всём массиве).

Мы улучшили get_min, но remove_min всё ещё остаётся O(n).

Реализация v.0.2 «Отсортированный массив»

Будем хранить массив отсортированным в порядке убывания (тогда минимум будет в конце).

```python
def sorted_insert(a, x):
    """Чтобы вставить элемент, необходимо найти ему правильную позицию."""
    # Ищем позицию для вставки с конца
    i = len(a)
    a.append(x)
    # Сдвигаем элементы вправо, пока не найдём место для x
    while i > 0 and a[i - 1] < x:  # Ищем первый элемент, который >= x
        a[i] = a[i - 1]
        i -= 1
    a[i] = x  # Вставляем x на найденное место
    return a  # В худшем случае: O(n)


def sorted_get_min(a):
    if not a:
        raise Exception("Очередь пуста")
    return a[-1]  # Минимум в конце. O(1)
```

```python
def sorted_remove_min(a):
    if not a:
        raise Exception("Очередь пуста")
    return a.pop(), a  # Удаляем последний элемент. O(1)
```

Анализ:

1. insert(x): **O(n)** (линейный поиск места для вставки + сдвиг).
2. get_min(): **O(1)**.
3. remove_min(): **O(1)**.

Теперь мы улучшили удаление, но ухудшили вставку.

Вывод. Во всех наивных реализациях хотя бы одна из ключевых операций работает за линейное время O(n). Если мы планируем выполнить последовательность из k операций (например, k вставок и k удалений), общее время работы будет $O(k * n) = O(k^2)$, что неприемлемо для больших k.

Нам нужна структура данных, которая гарантирует, что **все три основные операции** выполняются за **логарифмическое** время O(log n). Этой структурой является **бинарная куча**.

3. Бинарная куча (Binary Heap)

Бинарная куча – это структура данных, реализующая очередь с приоритетом. Она представляет собой **полное бинарное дерево** (все уровни заполнены, кроме, возможно, последнего, который заполняется слева направо), для которого выполняется **основное свойство кучи**:

1. **Свойство кучи (min-heap).** Значение в каждом узле меньше или равно значениям в его дочерних узлах.
2. Следствие. В корне дерева находится минимальный элемент.
3. Существует также max-heap, где значение в узле больше или равно значениям в потомках.

Ключевая идея: вместо хранения дерева в виде связной структуры с узлами, мы будем хранить его **в массиве**. Это эффективно с точки зрения памяти и скорости.

Схема хранения:

1. Корень дерева хранится в ячейке a[0].
2. Для узла с индексом i: **левый потомок** имеет индекс left(i) = 2*i + 1, **правый потомок** – right(i) = 2*i + 2, а **родитель** вычисляется по формуле parent(i) = (i - 1) // 2 (с использованием целочисленного деления).

Рассмотрим представление массива [2, 5, 7, 8, 6, 10, 42, 11, 15, 28, 9, 13] в виде двоичного дерева, где элемент с индексом 0 (значение 2) является корнем. Для этого дерева проверяются его свойства: узел с индексом 1 (родитель – 0) и узел с индексом 2 (родитель – 0) являются дочерними по отношению к корню. Далее, у узла с индексом 3 (значение 8) левый потомок имеет индекс 7 (значение 11), а правый – индекс 8 (значение 15). Аналогично, у узла с индексом 0 (корень) левый потомок – это узел 1 (значение 5), а правый – узел 2 (значение 7).

Реализация операций:

get_min(): Так как минимальный элемент всегда в корне, операция тривиальна.

```
def heap_get_min(a):
    if not a:
        raise Exception("Куча пуста")
    return a[0]  # O(1)
```

insert(x): Алгоритм состоит из двух шагов:

1. Добавляем новый элемент x в конец массива (в следующую свободную позицию в последнем уровне дерева).

2. Выполняем операцию sift_up (**просеивание вверх**) для нового элемента, чтобы восстановить свойство кучи. Мы сравниваем добавленный элемент с его родителем. Если он меньше родителя, мы меняем их местами. Процесс повторяется до тех пор, пока элемент не станет больше или равен родителю либо не достигнет корня.

```
def heap_insert(a, x):
    """Добавляем элемент в кучу."""
    a.append(x)  # Добавляем в конец
    sift_up(a, len(a) - 1)  # Просеиваем вверх
    return a


def sift_up(a, idx):
    """Поднимает элемент с индексом idx вверх, пока не восстановится куча."""
    while idx > 0:
        parent_idx = (idx - 1) // 2
        if a[idx] < a[parent_idx]:
            # Нарушен порядок, меняем местами
            a[idx], a[parent_idx] = a[parent_idx], a[idx]
            idx = parent_idx
        else:
            # Порядок восстановлен
            break
    return a
```

Время работы: O(log n), так как высота дерева ~ $\log_2 n$.

remove_min(): Алгоритм состоит из трёх шагов:

1. Запоминаем значение корня (минимум) для возврата.

2. Перемещаем **последний** элемент массива в корень.

3. Выполняем операцию sift_down (**просеивание вниз**) для нового корневого элемента, чтобы восстановить свойство кучи. Мы сравниваем этот элемент с его детьми. Если он больше кого-то из детей, мы меняем его местами с **наименьшим** из детей. Процесс повторяется до тех пор, пока элемент не станет меньше или равен обоим детям либо не станет листом.

```
def heap_remove_min(a):
    """Удаляем минимальный элемент из кучи."""
    if not a:
        raise Exception("Куча пуста")
    min_val = a[0]
    last_val = a.pop()
```

```python
    if a:
        a[0] = last_val  # Перемещаем последний элемент в корень
        sift_down(a, 0)  # Просеиваем вниз
    return min_val, a


def sift_down(a, idx):
    """Опускает элемент с индексом idx вниз, пока не восстановится куча."""
    n = len(a)
    while idx * 2 + 1 < n:  # Пока есть хотя бы один ребенок
        left_idx = 2 * idx + 1
        right_idx = 2 * idx + 2
        min_child_idx = left_idx

        if right_idx < n and a[right_idx] < a[left_idx]:
            min_child_idx = right_idx

        if a[idx] <= a[min_child_idx]:
            break
        else:
            a[idx], a[min_child_idx] = a[min_child_idx], a[idx]
            idx = min_child_idx
    return a
```

Время работы: O(log n).

Полная реализация бинарной кучи (min-heap):

```python
def heap_get_min(a):
    if not a:
        raise Exception("Куча пуста")
    return a[0]  # O(1)


def heap_insert(a, x):
    """Добавляем элемент в кучу."""
    a.append(x)  # Добавляем в конец
    sift_up(a, len(a) - 1)  # Просеиваем вверх
    return a


def sift_up(a, idx):
    """Поднимает элемент с индексом idx вверх, пока не восстановится куча."""
    while idx > 0:
        parent_idx = (idx - 1) // 2
        if a[idx] < a[parent_idx]:
            # Нарушен порядок, меняем местами
            a[idx], a[parent_idx] = a[parent_idx], a[idx]
            idx = parent_idx
        else:
            # Порядок восстановлен
            break
    return a
```

```python
def heap_remove_min(a):
    """Удаляем минимальный элемент из кучи."""
    if not a:
        raise Exception("Куча пуста")
    min_val = a[0]
    last_val = a.pop()
    if a:
        a[0] = last_val  # Перемещаем последний элемент в корень
        sift_down(a, 0)  # Просеиваем вниз
    return min_val, a


def sift_down(a, idx):
    """Опускает элемент с индексом idx вниз, пока не восстановится куча."""
    n = len(a)
    while idx * 2 + 1 < n:  # Пока есть хотя бы один ребенок
        left_idx = 2 * idx + 1
        right_idx = 2 * idx + 2
        min_child_idx = left_idx

        if right_idx < n and a[right_idx] < a[left_idx]:
            min_child_idx = right_idx

        if a[idx] <= a[min_child_idx]:
            break
        else:
            a[idx], a[min_child_idx] = a[min_child_idx], a[idx]
            idx = min_child_idx
    return a
```

4. Пирамидальная сортировка (Heapsort)

Бинарная куча позволяет очень элегантно реализовать алгоритм сортировки, называемый **пирамидальной сортировкой** или **сортировкой кучей**.

Идея алгоритма:

1. **Построение кучи.** Превратим исходный массив в max-heap (кучу, где корень – максимум). Для этого можно использовать процедуру heapify.

2. **Извлечение сортировки.** Будем последовательно извлекать максимум из кучи и перемещать его в конец массива, уменьшая размер «кучи» на единицу. После каждого извлечения восстанавливаем свойства кучи на уменьшенном массиве.

Шаг 1. Построение кучи (Heapify).

Есть два способа построить кучу из неупорядоченного массива:

1. **Последовательная вставка (O(n log n)).** Вызывать insert для каждого из n элементов.

2. **Линейный алгоритм (O(n)).** Идти снизу вверх и вызывать sift_down для каждого узла, начиная с последнего нелистового узла. Этот метод более эффективен.

Почему линейный heapify работает за O(n)?

Рассмотрим полное бинарное дерево высоты h.

1. На нижнем уровне (листья) находится ~ n/2 узлов. Высота листьев равна 0, для них sift_down выполняется 0 операций.
2. На уровне выше находится ~ n/4 узлов. Высота этих узлов равна 1, для них sift_down выполняется не более 1 операции.
3. На уровне выше находится ~ n/8 узлов. Высота равна 2, не более 2 операций.
4. …
5. В корне (1 узел) высота h, не более h операций.

Общее время:

$T(n) = 0 * (n/2) + 1 * (n/4) + 2 * (n/8) + 3 * (n/16) + ... + h * 1$

Можно показать, что эта сумма ограничена сверху числом 2n. Таким образом, $T(n) = O(n)$.

Реализация heapify и самой сортировки:

```python
def heapify(arr):
    """Превращает массив в max-heap."""
    n = len(arr)
    start_idx = n // 2 - 1  # Последний нелистовой узел
    for i in range(start_idx, -1, -1):
        sift_down_max(arr, i, n)

def sift_down_max(arr, idx, size):
    """Просеивание вниз для max-heap."""
    while idx * 2 + 1 < size:
        left_idx = 2*idx + 1
        right_idx = 2*idx + 2
        max_child_idx = left_idx

        if right_idx < size and arr[right_idx] > arr[left_idx]:
            max_child_idx = right_idx

        if arr[idx] >= arr[max_child_idx]:
            break
        else:
            arr[idx], arr[max_child_idx] = arr[max_child_idx], arr[idx]
            idx = max_child_idx

def heapsort(arr):
    """Пирамидальная сортировка."""
    n = len(arr)
    # Построение max-heap
    heapify(arr)  # O(n)
```

```python
    # Извлечение элементов
    for i in range(n-1, 0, -1):
        arr[0], arr[i] = arr[i], arr[0]  # Перемещаем максимум в конец
        sift_down_max(arr, 0, i)  # Восстанавливаем кучу
```

Анализ времени работы:

1. Шаг 1 (heapify): **O(n)**.
2. Шаг 2: Мы выполняем n-1 раз операцию swap (O(1)) и sift_down_max (O(log i)). Сумма log 1 + log 2 + … + log (n-1) ~ log(n!) = **O(n log n)**.
3. **Итого:** O(n) + O(n log n) = **O(n log n)**.

Сравнение с предыдущими сортировками:

В отличие от сортировки вставками (O(n^2) в худшем случае) и сортировки слиянием (O(n log n) всегда, но требует O(n) доп. памяти), пирамидальная сортировка работает за O(n log n) в худшем случае и использует O(1) дополнительной памяти (сортировка на месте).