Куча

Рассмотрим всем знакомую структуру данных - очередь. Это линейная последовательность данных, к которой применимы только две операции:

- добавить элемент в конец очереди,
- извлечь элемент из начала очереди.

Тут мы имеем фиксированный порядок элементов. Если элемент A попал в очередь раньше, чем элемент B, то и покинет очередь он раньше. Поменяем задачу: теперь мы каждому элементу сопоставим число – приоритет. Соответственно, добавляя элемент в очередь, мы протолкнем его через все элементы, приоритет которых ниже. А покидать очередь будет тот элемент, приоритет которого выше всех. Описанная выше абстрактная структура данных - очередь с приоритетами. Одной из ее реализаций является двоичная куча или пирамида (англ. Binary heap). Она представляет собой двоичное дерево, для которого выполняются три условия (рис. 1):

- зачение в любой вершине не меньше (не больше), чем значения ее потомков;
- на i-ом (нумерация с 0) слое 2^i вершин, кроме последнего;
- последний слой заполняется слева направо без пропусков.

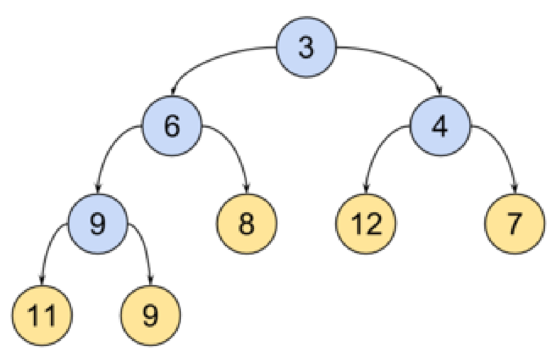

Рис. 1: Пример кучи для минимума

Используя второе свойство, оценим количество слоев в куче. Пусть в ней n элементов, тогда они распределены по слоям следующим образом (пока предположим, что n элементов хватит, чтобы заполнить нижний слой до конца): $1, 2, 4, ..., 2^i, ..., 2^h$, где h – искомая высота. По формуле суммы первых m элементов геометрической прогрессии $S_m = \frac{b_1 \cdot 1 - q^m}{1-q}$ вычислим $n = 2^{h+1} - 1$ (не забываем про 0-ой слой). Тогда $\log_2 n = \lfloor \log_2 (2^{h+1} - 1) \rfloor = h$. Отсюда получаем, что для произвольного n высота кучи оценивается как $O(\log n)$.

Самый удобный вариант хранения кучи - одномерный массив. Заметим следующую зависимость: для элемента в ячейке с индексом i потомки будут храниться в ячейках с индексами $2i+1$ и $2i+2$, а его предок будет храниться в ячейке с индексом $\lfloor (i-1)/2 \rfloor$.

Как и очередь куча поддерживает две операции:

- добавить элемент в кучу,
- извлечь из кучи максимум (минимум).

Для конкретности считаем, что на вершине куче минимум. Добавление работает следующим образом: добавим в конец кучи новый элемент x. Однако первое свойство могло быть нарушено. Для этого посмотрим на предка нового элемента. Если он больше, чем x, то поменяем их местами. Теперь посмотрим на нового предка x. Если первое свойство все еще нарушается, то поменяем их местами. Это будет продолжаться, пока мы не достигнем верха или пока очередной предок будет больше x. Заметим, что операция проталкивания элемента вверх выполнится не больше, чем высота кучи (так как при обмене элементов x перемещается ровно на один уровень вверх). Таким образом время работы операции добавления $O(\log n)$.

Для того, чтобы выполнить опреацию извелечения минимума, необходимо последний элемент x кучи поместить на самый верх, удалив минимум. После этого надо восстановить первое свойство кучи. Для этого посмотрим на двух непосредственных потомков элемента x. Если они оба больше x, то свойство кучи восстановлено. Если хотя бы один меньше x, то выберем минимального потомка и обменяем его с x. И так далее. При каждом обмене x опускается ровно на один уровень вниз, а значит что таких обменов не больше, чем высота кучи. Время работы этой операции также $O(\log n)$.

Ниже приведен пример реализации кучи для минимума. Программа ожидает на вход одну из комманд:

- add value - добавить в кучу число value;
- min - извлечь минимум из кучи и напечатать его;
- print - распечатать кучу в виде дерева;
- exit - завершить работу программы.

```python
from math import log2


def sift_up(heap, i):
    while i > 0 and heap[(i - 1) // 2] > heap[i]:
        heap[i], heap[(i - 1) // 2] = heap[(i - 1) // 2], heap[i]
        i = (i - 1) // 2


def sift_down(heap, i):
    n = len(heap)
    while i * 2 + 1 < n:
        j = i
        if heap[i] > heap[i * 2 + 1]:
            j = i * 2 + 1
        if i * 2 + 2 < n and heap[j] > heap[i * 2 + 2]:
            j = i * 2 + 2
        if i == j:
            break
        heap[i], heap[j] = heap[j], heap[i]
        i = j


def add(heap, x):
    heap.append(x)
    sift_up(heap, len(heap) - 1)


def extract_min(heap):
    x = heap[0]
    heap[0] = heap.pop()
    sift_down(heap, 0)
    return x


def pretty_print(heap):
    height = int(log2(len(heap)))
    node_width = len(str(max(heap)))
    str_width = (2 ** (height + 1) - 1) * node_width
    interval = node_width
    result = []
    for i in range(height, -1, -1):
        start = 2 ** i - 1
        nums_in_line = 2 ** i
        args = heap[start:start + nums_in_line]
        line = (" " * interval).join(["{:" + "{}".format(node_width) + "}"] *
```

```
                                        min(nums_in_line, len(args))).format(*args)
        if i != height:
            line = " " * ((str_width - len(line)) // 2) + line
        result.append(line)
        interval = interval * 2 + node_width
    print("\n".join(reversed(result)))


if __name__ == "__main__":
    heap = []
    while True:
        cmd = input().strip()
        if cmd.find(' ') != -1:
            cmd, v = cmd.split()
            v = int(v)
        if cmd == "exit":
            break
        elif cmd == "add":
            add(heap, v)
        elif cmd == "min":
            print(extract_min(heap))
        elif cmd == "print":
            pretty_print(heap)
        else:
            print("incorrect command")
```

Осталось рассмотреть процедуру построения кучи из неотсортированного массива данных. Тут есть два подхода. Первый подход предполагает поочередное добавление в кучу элементов массива. Каждое добавление займет $O(\log n)$ времени, всего добавлений n. Итого, операция построения займет $O(n \log n)$. Второй подход более хитрый. Надо просто пройти по массиву от $\lceil \frac{n}{2} \rceil$ до 0 и выполнить операцию просеивания вниз. Начнем с того, почему нужна толкьо половина массива. Легко показать, что элемент $\lceil \frac{n}{2} \rceil + 1$ не будет иметь ни одного потомка, т.к. индекс левого потомка будет первышать n. Соответственно каждый из этих элементов сам по себе является корректной кучей. Тогда для остальных элементов это будет значить, что оба поддерева уже являются кучами, а этот элемент просто нарушает первое свойство. Тогда нужно просто выполнить операцию просеивания вниз. Когда мы будем обрабатывать элемент i, все элементы со старшими индексами уже должны быть обработаны.

Оценим время работы такого алгоритма.Число вершин на высоте h в куче из n элементов не превосходит $\lceil \frac{n}{2^h} \rceil$. Высота кучи не превосходит $\log_2 n$. Обозначим за H высоту дерева, тогда время построения не превосходит

$$\sum_{h=1}^{H} \frac{n}{2^h} \cdot h \cdot 2 = 2 \cdot n \cdot \sum_{h=1}^{H} \frac{h}{2^h}. \tag{1}$$

Докажем вспомогательную лемму о сумме ряда.

$$\sum_{h=1}^{\infty} \frac{h}{d^h} = \frac{d}{(d-1)^2}.$$

Обозначим за s сумму ряда. Заметим, что

$$\frac{h}{d^h} = \frac{1}{d} \cdot \frac{h-1}{d^{h-1}} + \frac{1}{d^h}. \tag{2}$$

$\sum_{h=1}^{\infty} \frac{1}{d^h}$ – это сумма бесконечной убывающей геометрической прогрессии, и она равна $\frac{1}{d-1}$.

Получаем

$$s = \frac{1}{d} \cdot s + \frac{1}{d-1}. \tag{3}$$

Откуда $s = \frac{d}{(d-1)^2}$.

Подставляя в формулу 1 результат леммы, получаем

$$2 \cdot n \cdot \frac{2}{(2-1)^2} = 4 \cdot n = O(n). \tag{4}$$

```python
from math import ceil

def build_heap(heap):
    for i in range(ceil(len(heap) / 2), -1, -1):
        sift_down(heap, i)
```