

# Бинарная куча. Основные операции. Построение бинарной кучи за $\mathcal{O}(n)$

Соль Михаил

## 1 Бинарная куча

**Определение 1.1.** Бинарной кучей назовём структуру данных, представляющую собой подвешенное бинарное дерево, где для каждого нелистового узла верно, что его дети не меньше (не больше) его самого, а также, что на  $n$ -ом уровне, кроме листового,  $2^n$  узлов.

**Лемма 1.1.** Число ярусов (высота) бинарной кучи из  $n$  элементов можно оценить, как  $\mathcal{O}(\log n)$ .

*Доказательство.* Рассмотрим все ярусы кучи, без последнего. Тогда у нас имеется полная бинарная куча в которой  $k < n$  элементов и ее высота на 1 меньше. У такой кучи каждый следующий ярус содержит в два раза больше элементов. Пусть высота этой кучи равна  $h$ . Тогда:

$$k = \sum_{i=0}^{h-1} 2^i = 2^h - 1$$

Из этого следует, что  $h = \log_2(k+1) \leq \log_2 n = \mathcal{O}(\log n)$ . Поскольку исходная куча содержит на 1 ярус больше, ее высоту можно оценить как:

$$\mathcal{O}(\log n)$$

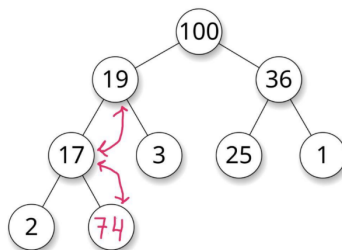
□

Хранение в памяти:

- Хранить кучу удобно в массиве (далее будем использовать 0-индексацию)
- Пусть мы рассматриваем элемент с индексом  $i$
- Индекс левого ребенка:  $2i + 1$
- Индекс правого ребенка:  $2i + 2$
- Элементы, которые являются листьями, имеют индексы  $\lfloor \frac{n}{2} \rfloor \dots (n - 1)$

## 2 Sift Up (Просеивание вверх)

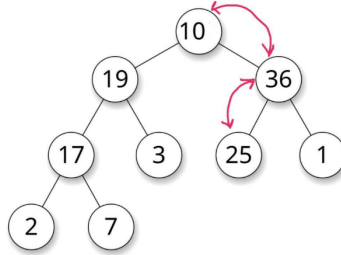
- Сравним элемент с родительским элементом, если он меньше — поменяем их местами
- Продолжим выполнение операции до тех пор, пока родитель не окажется больше или равным значению нашего элемента, либо пока элемент не станет корнем



См. реализацию в конце конспекта

### 3 Sift Down (Просеивание вниз)

- Сравним элемент с дочерними элементами, если один из них больше, чем этот элемент, поменяем местами с большим из дочерних элементов
- Продолжим выполнение операции до тех пор, пока элемент не окажется больше всех детей или не окажется листом



См. реализацию в конце конспекта

### 4 Вставка

Имеется куча (возможно, пустая), мы хотим добавить в неё элемент:

- Свойства кучи позволяют только добавить его на последний ярус
- Такое добавление может нарушить свойства кучи: добавленный элемент может оказаться больше родителя
- Исправить проблему возможно, если выполнить операцию SiftUp

Операция будет работать за  $\mathcal{O}(\log n)$ , поскольку в худшем случае число обменов равно высоте кучи.

### 5 ExtractMax (ExtractMin)

Имеется непустая куча, хотим извлечь максимальный (минимальный) элемент (после извлечения элемент больше не должен находиться в куче):

- Исходя из свойств кучи максимум (минимум) всегда находится в корне
- Также согласно свойствам кучи (все ярусы кроме последнего должны быть заполнены), на место извлекаемого элемента необходимо что-то поставить
- Для этого подойдет последний элемент последнего яруса, но он может оказаться меньше (больше) какого-либо из детей корня
- Исправить проблему возможно, если выполнить операцию Sift Down

Операция будет работать за  $\mathcal{O}(\log n)$ , поскольку в худшем случае число обменов равно высоте кучи.

### 6 Increase Key и Decrease Key

Имеется непустая куча, хотим заменить элемент по ключу:

Increase Key:

- Имея ключ, увеличиваем его значение на  $\Delta > 0$
- Для восстановления свойства кучи выполняем Sift Up (для max-кучи) или Sift Down (для min-кучи)

Decrease Key:

- Имея ключ, уменьшаем его значение на  $\Delta > 0$
- Для восстановления свойства кучи выполняем Sift Up (для min-кучи) или Sift Down (для max-кучи)

Операции будут работать за  $\mathcal{O}(\log n)$ , поскольку они основаны на операциях Sift Down и Sift Up.

## 7 Удаление элемента

Имеется непустая куча, хотим удалить элемент по ключу:

- Имея ключ, делаем его значение условно  $+\infty$  ( $-\infty$ )
- Выполняем операцию Sift Up (элемент окажется в корне)
- Выполняем операцию ExtractMax (ExtractMin)

**Замечание 7.1.** Такая последовательность действий подходит для Increase/Decrease Key, но тогда после неё необходимо вставить новый элемент, равный значению удалённого  $\pm \Delta$ , и просеять его вверх (Sift Up).

## 8 Построение

1. Можно построить за  $\mathcal{O}(n \log n)$ , просто сделав  $n$  операций вставки
2. Можно построить за  $\mathcal{O}(n)$ . Давайте поочередно от элемента с индексом  $\lfloor \frac{n}{2} \rfloor$  до начала массива вызывать операцию SiftDown

**Замечание 8.1.** Это будет работать, так как кучи из 1 вершины корректны, а далее мы восстанавливаем кучи снизу вверх.

*Доказательство.* Докажем, что построение 2 работает за  $\mathcal{O}(n)$ . Заметим, что:

- На высоте  $i$  (листья имеют высоту 1) число вершин не превосходит  $\lceil \frac{n}{2^i} \rceil$
- Одна операция SiftDown делает  $2i$  сравнений, а также высота кучи не превосходит  $\lceil \log n \rceil$

Тогда:

$$T(n) \leq \sum_{i=1}^{\lceil \log n \rceil} \frac{n}{2^i} \cdot 2i = 2n \sum_{i=1}^{\lceil \log n \rceil} \frac{i}{2^i}$$

Пусть

$$S = \sum_{i=1}^{\lceil \log n \rceil} \frac{i}{2^i} = \sum_{i=1}^{\lceil \log n \rceil} \left( \frac{1}{2} \frac{i-1}{2^{i-1}} + \frac{1}{2^i} \right) = \frac{1}{2} \sum_{i=1}^{\lceil \log n \rceil} \frac{i-1}{2^{i-1}} + \sum_{i=1}^{\lceil \log n \rceil} \frac{1}{2^i}$$

Оценим первое слагаемое:

$$\frac{1}{2} \sum_{i=1}^{\lceil \log n \rceil} \frac{i-1}{2^{i-1}} < \frac{1}{2} \sum_{i=1}^{\lceil \log n \rceil} \frac{i}{2^i} = \frac{1}{2} S$$

Оценим второе слагаемое как бесконечно убывающую геометрическую прогрессию:

$$\sum_{i=1}^{\lceil \log n \rceil} \frac{1}{2^i} < 1$$

Следовательно  $S < \frac{1}{2}S + 1 \iff S < 2$

Тогда:

$$T(n) \leq 2n \sum_{i=1}^{\lceil \log n \rceil} \frac{i}{2^i} < 4n = \mathcal{O}(n)$$

□

**Замечание 8.2.** Для слияния двух куч сложим их элементы в один массив и построим кучу. Будет работать за  $\mathcal{O}(n)$ .

```

1 public class Heap {
2     private Object[] elements;
3     private int size;
4     private final Comparator<Object> comparator;
5     public Heap(Comparator<Object> cmp) {
6         this.comparator = cmp;
7         this.elements = new Object[10];
8         this.size = 0;
9     }
10    protected void siftUp(int index) {
11        assert elements != null;
12        while (index != 0) {
13            int parent = (index - 1) / 2;
14            if (comparator.compare(elements[parent], elements[index]) < 0) {
15                swap(parent, index);
16                index = parent;
17            } else {
18                return;
19            }
20        }
21    }
22    protected void siftDown(int index) {
23        if (index >= size) return;
24
25        int leftChild = index * 2 + 1;
26        int rightChild = index * 2 + 2;
27        int iMin = index;
28        if (leftChild < size &&
29            comparator.compare(elements[iMin], elements[leftChild]) < 0) {
30            iMin = leftChild;
31        }
32        if (rightChild < size &&
33            comparator.compare(elements[iMin], elements[rightChild]) < 0) {
34            iMin = rightChild;
35        }
36        if (iMin != index) {
37            swap(index, iMin);
38            siftDown(iMin);
39        }
40    }
41    private void swap(int i, int j) {
42        Object temp = elements[i];
43        elements[i] = elements[j];
44        elements[j] = temp;
45    }
46 }

```