Двоичная куча

Содержание

Определение

Определение:

Двоичная куча или **пирамида** (англ. *Binary heap*) — такое двоичное подвешенное дерево, для которого выполнены следующие три условия:

- Значение в любой вершине не меньше (если куча для максимума), чем значения её потомков.
- На i-ом слое 2^i вершин, кроме последнего. Слои нумеруются с нуля.
- Последний слой заполнен слева направо (как показано на рисунке)

Удобнее всего двоичную кучу хранить в виде массива $a[0..n-1]$, у которого нулевой элемент, $a[0]$ — элемент в корне, а потомками элемента $a[i]$ являются $a[2i+1]$ и $a[2i+2]$. Высота кучи определяется как высота двоичного дерева. То есть она равна количеству рёбер в самом длинном простом пути, соединяющем корень кучи с одним из её листьев. Высота кучи есть $O(\log n)$, где n — количество узлов дерева.

Чаще всего используют кучи для минимума (когда предок не больше детей) и для максимума (когда предок не меньше детей).

Двоичные кучи используют, например, для того, чтобы извлекать минимум из набора чисел за $O(\log n)$. Они являются частным случаем приоритетных очередей.

Базовые процедуры

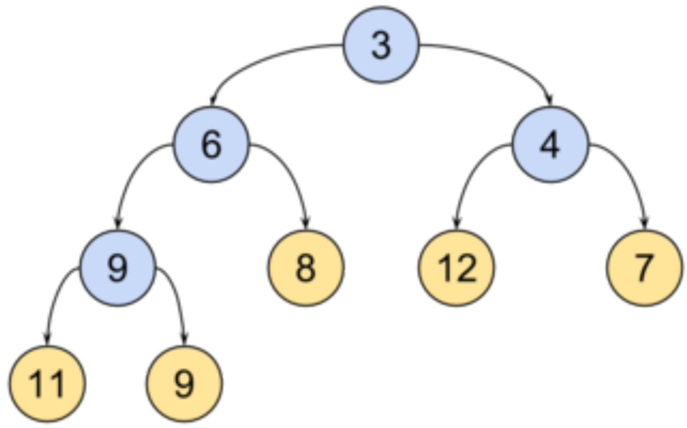
Восстановление свойств кучи

Если в куче изменяется один из элементов, то она может перестать удовлетворять свойству упорядоченности. Для восстановления этого свойства служат процедуры $\mathrm{siftDown}$ (просеивание вниз) и siftUp (просеивание вверх).

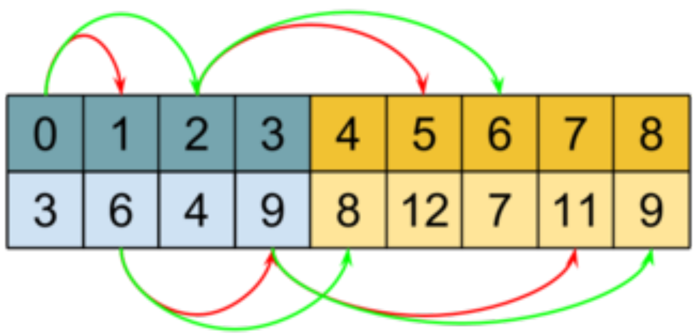
siftDown

Если значение измененного элемента увеличивается, то свойства кучи восстанавливаются функцией $\mathrm{siftDown}$.

Работа процедуры: если i-й элемент меньше, чем его сыновья, всё поддерево уже является кучей, и делать ничего не надо. В противном случае меняем местами i-й элемент с наименьшим из его сыновей, после чего выполняем $\mathrm{siftDown}$ для этого сына. Процедура выполняется за время $O(\log n)$.



Пример кучи для минимума



Хранение кучи в массиве, красная стрелка — левый сын, зеленая — правый

```
function siftDown(i : int):
    while 2 * i + 1 < a.heapSize     // heapSize — количество элементов в куче
        left = 2 * i + 1             // left — левый сын
        right = 2 * i + 2            // right — правый сын
        j = left
        if right < a.heapSize and a[right] < a[left]
            j = right
        if a[i] <= a[j]
            break
        swap(a[i], a[j])
        i = j
```

siftUp

Если значение измененного элемента уменьшается, то свойства кучи восстанавливаются функцией siftUp.

Работа процедуры: если элемент больше своего отца, условие 1 соблюдено для всего дерева, и больше ничего делать не нужно. Иначе, мы меняем местами его с отцом. После чего выполняем siftUp для этого отца. Иными словами, слишком маленький элемент всплывает наверх. Процедура выполняется за время $O(\log n)$.

```
function siftUp(i : int):
    while a[i] < a[(i - 1) / 2]     // i   0 — мы в корне
        swap(a[i], a[(i - 1) / 2])
        i = (i - 1) / 2
```

Извлечение минимального элемента

Выполняет извлечение минимального элемента из кучи за время $O(\log n)$. Извлечение выполняется в четыре этапа:

1. Значение корневого элемента (он и является минимальным) сохраняется для последующего возврата.
2. Последний элемент копируется в корень, после чего удаляется из кучи.
3. Вызывается $\mathrm{siftDown}$ для корня.
4. Сохранённый элемент возвращается.

```
int extractMin():
    int min = a[0]
    a[0] = a[a.heapSize - 1]
    a.heapSize = a.heapSize - 1
    siftDown(0)
    return min
```

Добавление нового элемента

Выполняет добавление элемента в кучу за время $O(\log n)$. Добавление произвольного элемента в конец кучи, и восстановление свойства упорядоченности с помощью процедуры siftUp.

```
function insert(key : int):
    a.heapSize = a.heapSize + 1
    a[a.heapSize - 1] = key
    siftUp(a.heapSize - 1)
```

Построение кучи за O(n)

Определение:

D**-куча** — это куча, в которой у каждого элемента, кроме, возможно, элементов на последнем уровне, ровно d потомков.

Дан массив $a[0..n-1]$. Требуется построить d-кучу с минимумом в корне. Наиболее очевидный способ построить такую кучу из неупорядоченного массива — сделать нулевой элемент массива корнем, а дальше по очереди добавить все его элементы в конец кучи и запускать от каждого добавленного элемента siftUp. Временная оценка такого алгоритма $O(n\log n)$. Однако можно построить кучу еще быстрее — за $O(n)$.

Представим, что в массиве хранится дерево ($a[0]-$ корень, а потомками элемента $a[i]$ являются $a[di+1]...a[di+d]$). Сделаем $\mathrm{siftDown}$ для вершин, имеющих хотя бы одного потомка: от $\dfrac{n}{d}$ до 0,— так как поддеревья, состоящие из одной вершины без потомков, уже упорядочены.

Лемма:

На выходе получим искомую кучу.

Доказательство:

▷

До вызова $\mathrm{siftDown}$ для вершины, ее поддеревья являются кучами. После выполнения $\mathrm{siftDown}$ эта вершина с ее поддеревьями будут также являться кучей. Значит, после выполнения всех $\mathrm{siftDown}$ получится куча.

◁

Лемма:

Время работы этого алгоритма $O(n)$.

Доказательство:

▷

Число вершин на высоте h в куче из n элементов не превосходит $\left\lceil \frac{n}{d^h} \right\rceil$. Высота кучи не превосходит $\log_d n$. Обозначим за H высоту дерева, тогда время построения не превосходит

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

Докажем вспомогательную лемму о сумме ряда.

Лемма:

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

Доказательство:

▷

Обозначим за s сумму ряда. Заметим, что $\frac{n}{d^n} = \frac{1}{d} \cdot \frac{n-1}{d^{n-1}} + \frac{1}{d^n}$.

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

Получаем $s = \frac{1}{d} \cdot s + \frac{1}{d-1}$. Откуда $s = \frac{d}{(d-1)^2}$.

◁

Подставляя в нашу формулу результат леммы, получаем $n \cdot (\frac{d}{d-1})^2 \leqslant 4 \cdot n = O(n)$.

◁

Псевдокод алгоритма:

```
function buldHeap():
    for i = a.heapSize / 2 downto 0
        siftDown(i)
```

Слияние двух куч

Даны две кучи a и b, размерами n и m, требуется объединить эти две кучи.

Наивная реализация

Поочередно добавим все элементы из b в a. Время работы — $O(m\log(n+m))$.

```
function merge(a, b : Heap):
    while b.heapSize > 0
        a.insert(b.extractMin())
```

Реализация с помощью построения кучи

Добавим все элементы кучи b в конец массива a, после чего вызовем функцию построения кучи. Процедура выполняется за время $O(n+m)$.

```
function merge(a, b : Heap):
    for i = 0 to b.heapSize - 1
        a.heapSize = a.heapSize + 1
        a[a.heapSize - 1] = b[i]
    a.heapify()
```

Поиск k-ого элемента

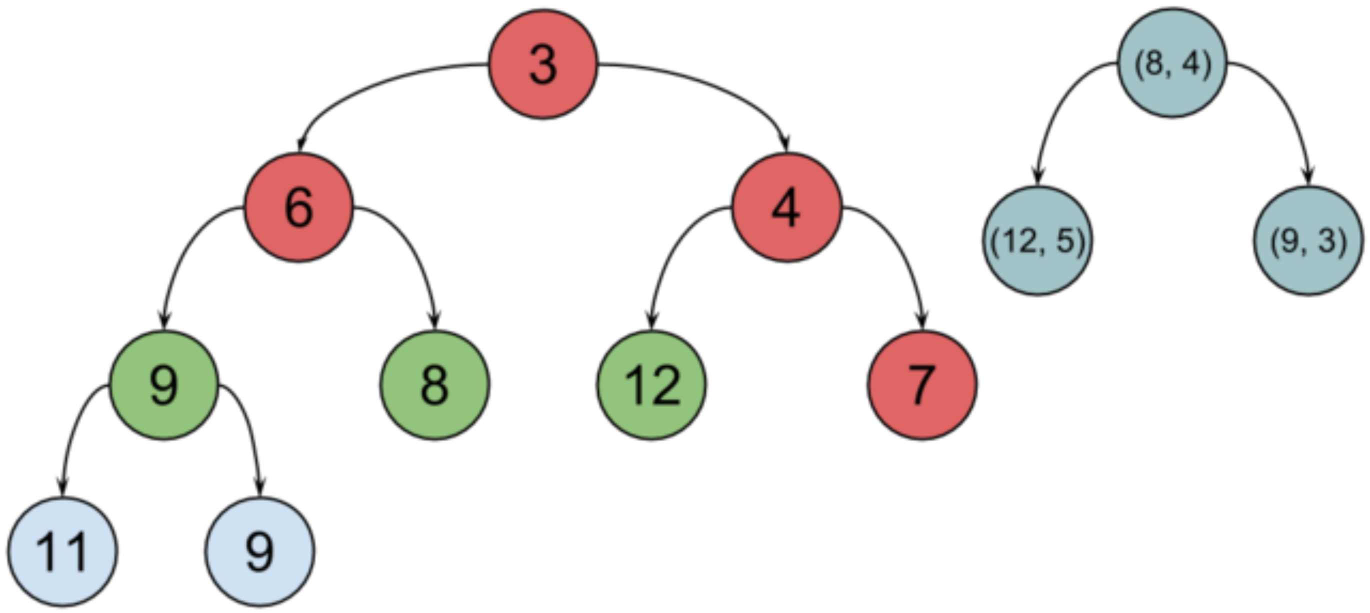
Требуется найти k-ый по величине элемент в куче.

1. Создаем новую кучу, в которой будем хранить пару $\langle \mathtt{value}, \mathtt{index} \rangle$, где $\mathtt{value}$ — значение элемента, а $\mathtt{index}$ — индекс элемента в основном массиве, и добавляем в нее корень кучи.
2. Возьмем корень новой кучи и добавим её детей из основной кучи, после чего удалим корень. Проделаем этот шаг $k-1$ раз.

3. В корне новой кучи будет находиться ответ.

Время работы алгоритма — $O(k \log k)$.

При $n \lessapprox k \log k$ выгоднее запускать поиск k-ой порядковой статистики.



Пример при $k = 5$, красные — уже удаленные из кучи элементы, зеленые находятся в куче, а голубые — еще не рассмотрены.

См. также

- Биномиальная куча
- Фибоначчиева куча
- Левосторонняя куча

Источники информации

- Википедия — Двоичная куча
- Википедия — Очередь с приоритетом
- Wikipedia — Binary heap
- Wikipedia — Priority queue

Источник — «http://neerc.ifmo.ru/wiki/index.php?title=Двоичная_куча&oldid=68231»

- Эта страница последний раз была отредактирована 9 января 2019 в 23:22.