Куча

Denis Bakin

Зачем нужна очередь с приоритетом?

Нужна структура, которая умеет быстро:

- находить минимальный элемент
- удалять минимальный элемент
- добавлять новый элемент

Зачем нужна очередь с приоритетом?

Нужна структура, которая умеет быстро:

- находить минимальный элемент
- удалять минимальный элемент
- добавлять новый элемент

Такие операции нужны, например:

- в алгоритме Дейкстры
- в алгоритме Прима
- в задачах планирования и обработки событий

Что хотим от структуры?

В этой лекции рассматриваем **мин-кучу**:

- в корне хранится минимальный элемент
- чем меньше значение, тем выше приоритет

Что хотим от структуры?

В этой лекции рассматриваем **мин-кучу**:

- в корне хранится минимальный элемент
- чем меньше значение, тем выше приоритет

Хотелось бы получить:

- `min` за $O(1)$
- `push` за $O(\log n)$
- `pop` за $O(\log n)$

Идея двоичной кучи

Двоичная куча — это дерево, для которого выполнены три свойства:

1. родитель не больше своих детей
2. у каждой вершины не более двух детей
3. дерево заполнено по слоям

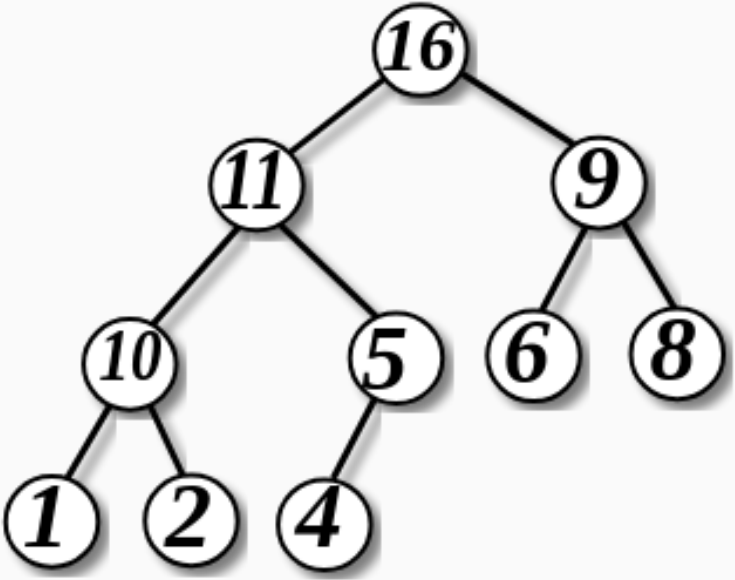
Идея двоичной кучи

Двоичная куча — это дерево, для которого выполнены три свойства:

1. родитель не больше своих детей
2. у каждой вершины не более двух детей
3. дерево заполнено по слоям

Важно: куча задает не полный порядок элементов, а только локальное условие «родитель не больше детей»

Как выглядит куча?



Почему это быстро?

Если в куче n элементов, то её высота равна:

Почему это быстро?

Если в куче n элементов, то её высота равна:

$$\log_2 n$$

Почему это быстро?

Если в куче n элементов, то её высота равна:

$$\log_2 n$$

Операция	Что происходит	Сложность
Найти минимум	смотрим в корень	$O(1)$
Добавить элемент	поднимаемся по одному пути вверх	$O(\log n)$
Удалить минимум	спускаемся по одному пути вниз	$O(\log n)$

Куча хранится в массиве

Если вершина хранится в `t[k]`, то:

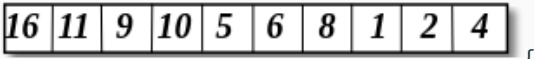
- левый сын: `t[2k]`
- правый сын: `t[2k + 1]`
- родитель: `t[k / 2]`

Куча хранится в массиве

Если вершина хранится в `t[k]`, то:

- левый сын: `t[2k]`
- правый сын: `t[2k + 1]`
- родитель: `t[k / 2]`

`t[1]` — корень



Ключевая идея

После изменения одного элемента куча может «сломаться» только:

- на пути от этой вершины к корню
- или на пути от неё вниз к листьям

Ключевая идея

После изменения одного элемента куча может «сломаться» только:

- на пути от этой вершины к корню
- или на пути от неё вниз к листьям

Поэтому достаточно уметь делать две операции:

- `sift_up`
- `sift_down`

Просеивание вверх: `sift_up`

Когда применяется?

- когда значение вершины **уменьшилось**
- или когда мы **добавили** новый элемент в конец массива

Просеивание вверх: `sift_up`

Когда применяется?

- когда значение вершины **уменьшилось**
- или когда мы **добавили** новый элемент в конец массива

Что делаем:

1. сравниваем вершину с родителем
2. если вершина меньше родителя, меняем их местами
3. продолжаем, пока не дойдем до корня или пока свойство кучи не восстановится

Живой пример: вставка в кучу

Добавим 3 в мин-кучу

```
[2, 5, 4, 8, 7, 9, 6]
```

Просеивание вниз: `sift_down`

Когда применяется?

- когда значение вершины **увеличилось**
- или когда мы поставили в корень «чужой» элемент после удаления минимума

Просеивание вниз: `sift_down`

Когда применяется?

- когда значение вершины **увеличилось**
- или когда мы поставили в корень «чужой» элемент после удаления минимума

Что делаем:

1. выбираем меньшего из детей
2. если текущая вершина больше него, меняем их местами
3. продолжаем движение вниз, пока свойство кучи не восстановится

Три основные операции

1. **Найти минимум** просто вернуть `t[1]`
2. **Удалить минимум** перенести последний элемент в корень и вызвать `sift_down(1)`
3. **Добавить элемент** положить его в конец массива и вызвать `sift_up`

Построение кучи

Кучу из массива можно построить двумя способами:

- добавлять элементы по одному: $O(n \log n)$
- взять готовый массив и превратить его в кучу снизу вверх: $O(n)$. Как?

Построение кучи

Кучу из массива можно построить двумя способами:

- добавлять элементы по одному: $O(n \log n)$
- взять готовый массив и превратить его в кучу снизу вверх: $O(n)$. Как?

Идея линейного построения:

- листья уже являются кучами
- достаточно пройти по внутренним вершинам справа налево
- в каждой вершине вызвать `sift_down`

build_heap

```
void build_heap() {
    for (int i = n / 2; i >= 1; --i) {
        sift_down(i);
    }
}
```

Почему начинаем с `n / 2`?

build_heap

```
void build_heap() {
    for (int i = n / 2; i >= 1; --i) {
        sift_down(i);
    }
}
```

Почему начинаем с `n / 2`?

- все вершины с индексами больше `n / 2` — листья
- листья уже удовлетворяют свойству кучи

Почему это линейно?

- Почему это работает за $O(n)$? Оценим сверху суммарное количество просеиваний вниз
- Для каждой высоты i будет не больше 2^{h-i} вершин на ней:

$$\sum_{i=1}^{h} 2^{h-i} \cdot i = 2^{h+1} - h - 2 \leq 2n - h - 2 \leq 2n = O(n)$$

Первое равенство несложно доказывается индукцией по h.

STL: алгоритмы над массивом

В стандартной библиотеке есть функции:

- `make_heap`
- `push_heap`
- `pop_heap`
- `sort_heap`

STL: алгоритмы над массивом

В стандартной библиотеке есть функции:

- `make_heap`
- `push_heap`
- `pop_heap`
- `sort_heap`

Важно: **по умолчанию STL строит максимальную кучу**

STL: алгоритмы над массивом

В стандартной библиотеке есть функции:

- `make_heap`
- `push_heap`
- `pop_heap`
- `sort_heap`

Важно: **по умолчанию STL строит максимальную кучу**

То есть после `make_heap(v.begin(), v.end())` в `v.front()` будет лежать **максимальный**, а не минимальный элемент

STL: priority_queue

По умолчанию контейнер

```
std::priority_queue<int> q;
```

тоже является **максимальной** кучей

STL: priority_queue

По умолчанию контейнер

```
std::priority_queue<int> q;
```

тоже является **максимальной** кучей

Если нужна минимальная куча, пишем:

```
std::priority_queue<int, std::vector<int>, std::greater<int>> q;
```

Когда полезен `std::set`?

Иногда в качестве очереди с приоритетом используют `std::set`

Когда полезен `std::set`?

Иногда в качестве очереди с приоритетом используют `std::set`

Это удобно, если нужно:

- удалять не только минимум
- явно обновлять значения
- хранить элементы в отсортированном порядке

Итоги

- Куча — реализация очереди с приоритетами
- Двоичная куча хранит элементы в почти полном двоичном дереве
- Свойство кучи восстанавливают две операции: `sift_up` и `sift_down`
- `min` работает за $O(1)$, `push` и `pop` — за $O(\log n)$
- `build_heap` можно сделать за $O(n)$
- В STL по умолчанию используется **максимальная** куча