

Разработка и анализ алгоритмов

Лекция 6

Кучи

Сергей Леонидович Бабичев

Приоритетная очередь

- В алгоритма Хаффмена из прошлой лекции требовалось:
 1. В некотором множестве значений найти два минимальных элемента.
 2. Удалить эти элементы из множества.
 3. Сложить их значения и снова добавить в множество.
 4. Прodelывать эту операцию до тех пор, пока в множестве не останется ровно один элемент.
- Наивная реализация алгоритма: массив.
- Сложность формирования множества тогда $O(N \log N)$.
- Сложность поиска наименьшего $O(1)$.
- Сложность удаления наименьшего $O(1)$.
- Сложность добавления $O(N)$.
- Введём абстракцию *приоритетная очередь* для таких запросов.

Абстракция *приоритетная очередь*

Приоритетная очередь

- Приоритетная очередь (priority queue) — структура данных, элементы которой сравнимы и имеют приоритет.
- Первым извлекается наиболее приоритетный элемент (максимум или минимум) (*голова очереди*).
- Любая операция вставки оставляет неизменным инвариант: в голове очереди всегда находится самый приоритетный элемент.

Интерфейс абстракции *приоритетная очередь*

- $insert(q, x)$ — добавление элемента x в очередь;
- $x = getMin(q)$ — получает самый приоритетный элемент.
- $extractMin(q)$ — извлекает самый приоритетный элемент и удаляет его;
- $decreaseKey(q, p, d)$ — сделать элемент, идентифицированный как p , более приоритетным, уменьшить его приоритет на $d > 0$. Элемент, возможно, станет ближе к голове очереди.
- $size(q)$ — получить количество элементов в очереди.

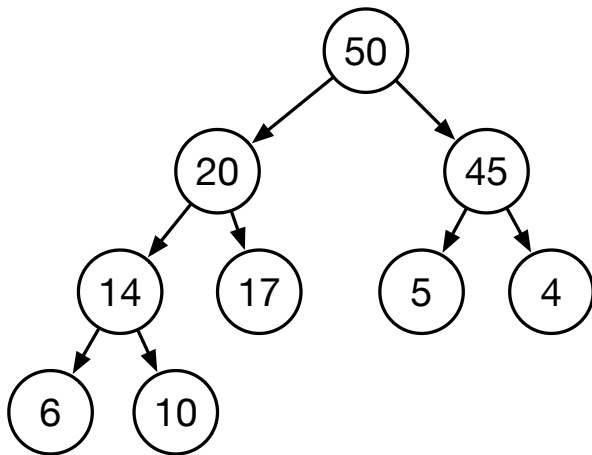
Бинарная куча

Бинарная куча

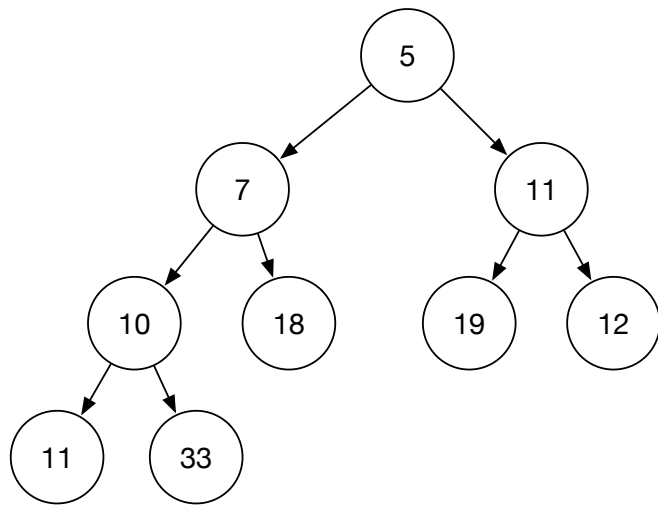
- *Бинарная куча* — бинарное дерево, удовлетворяющее условиям:
 - ▶ Для любой вершины её приоритет не меньше приоритета потомков.
 - ▶ Дерево является правильным подмножеством полного бинарного — *инвариант по структуре*.

Другое название — пирамида (heap).

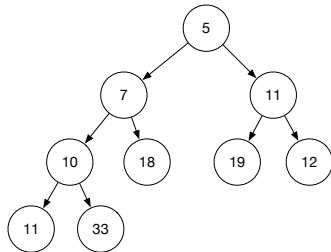
Приоритетная очередь: невозрастающая пирамида



Приоритетная очередь: неубывающая пирамида



Приоритетная очередь: реализация в виде массива



Хранение в виде массива с индексами от 1 до N :

5	7	11	10	18	19	12	11	33
---	---	----	----	----	----	----	----	----

- Индекс корня дерева всегда равен 1 — максимальный (минимальный) элемент
- Индекс родителя узла i равен $\lfloor \frac{i}{2} \rfloor$
- Индекс левого потомка узла i равен $2i$
- Индекс правого потомка узла i равен $2i + 1$
- Инвариант по данным:

$$\forall a_i \begin{cases} a_i \leq a_{2i}, & \text{если } 2i \leq N; \\ a_i \leq a_{2i+1}, & \text{если } 2i + 1 \leq N; \end{cases}$$

Приоритетная очередь: реализация на базе массива.

```
typedef int T;  
  
typedef struct binary_heap_s {  
    T      *body;  
    size_t  allocated;  
    size_t  nodes;  
} bh;
```

Бинарная куча: создание

```
bh* bh_create(size_t maxsize) {
    bh *t = malloc(sizeof(bh));
    t->body = malloc(sizeof(T) * (maxsize+1));
    t->allocated = maxsize;
    t->nodes = 0;
    return t;
}

void bh_destroy(bh *t) {
    free(t->body); free(t);
}

void bh_swap(bh *t, int a, int b) {
    T tmp = t->body[a]; t->body[b] = t->body[a]; t->body[a] = tmp;
}
```

$$T_{create} = O(N)$$

Бинарная куча: поиск минимума

- Соблюдается инвариант по данным.
- Операция \leq транзитивна.
- Самый приоритетный элемент — в вершине кучи.

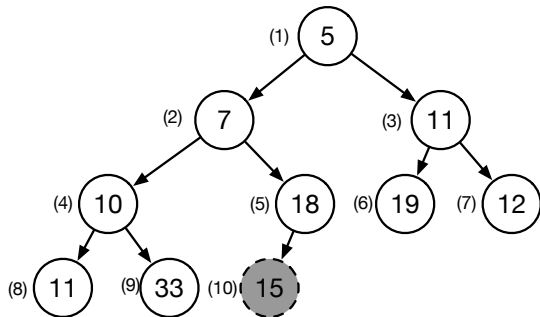
```
T bh_getMin(bh *t)
    assert(t->nodes > 0);
    return t->body[1];
}
```

$$T_{fetchMin} = O(1)$$

Бинарная куча: вставка элемента

- Этап 1. Вставка в конец кучи.

Вставка элемента 15



Отлично! Структура кучи не испортилась!

5	7	11	10	18	19	12	11	33	15
---	---	----	----	----	----	----	----	----	----

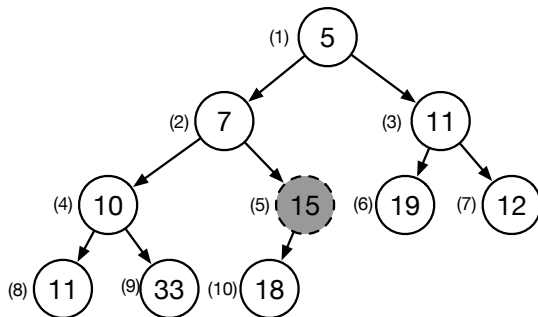
Бинарная куча: подъём элемента

- Требуется, не меняя структурного инварианта, добиться инварианта по данным.
- Если для элемента не выполняется инвариант по данным — два случая.
 1. Элемент находится ниже допустимого — поднимаем его (*siftUp*).
 2. Элемент находится выше допустимого — опускаем его (*siftDown*).

Бинарная куча: вставка элемента: *siftUp*

- Этап 2. Корректировка значений.

Вставка элемента 15



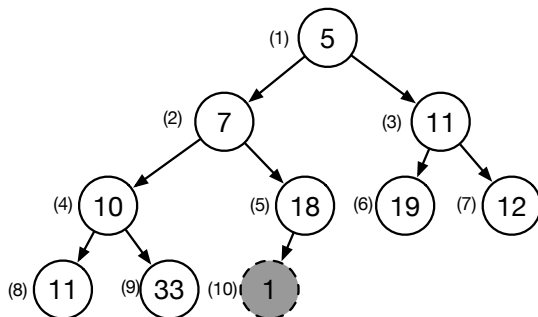
Куча удовлетворяет всем условиям.

5	7	11	10	15	19	12	11	33	18
---	---	----	----	----	----	----	----	----	----

Бинарная куча: вставка элемента: *siftUp*

- Попытаемся вставить минимальный элемент.

Вставка элемента 1

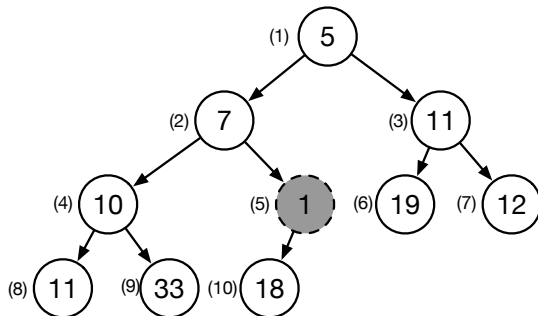


5	7	11	10	18	19	12	11	33	1
---	---	----	----	----	----	----	----	----	---

Бинарная куча: вставка элемента: *siftUp*

- Минимальный элемент ползёт вверх по дереву.

Вставка элемента 1

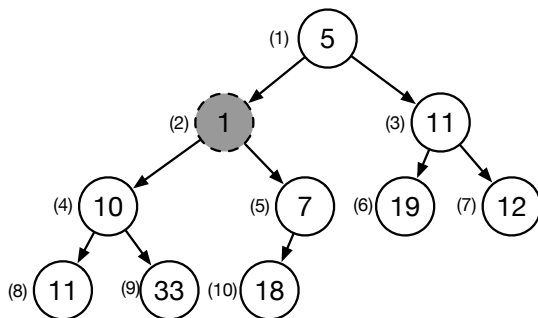


5	7	11	10	1	19	12	11	33	18
---	---	----	----	---	----	----	----	----	----

Бинарная куча: вставка элемента: *siftUp*

- Минимальный элемент ползёт вверх по дереву.

Вставка элемента 1

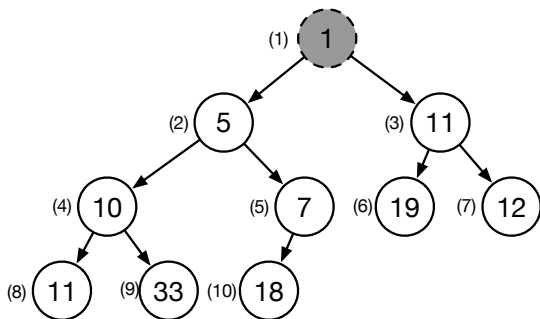


5	1	11	10	7	19	12	11	33	18
---	---	----	----	---	----	----	----	----	----

Бинарная куча: вставка элемента

- Минимальный элемент ползёт вверх по дереву.

Вставка элемента 1



1	5	11	10	7	19	12	11	33	18
---	---	----	----	---	----	----	----	----	----

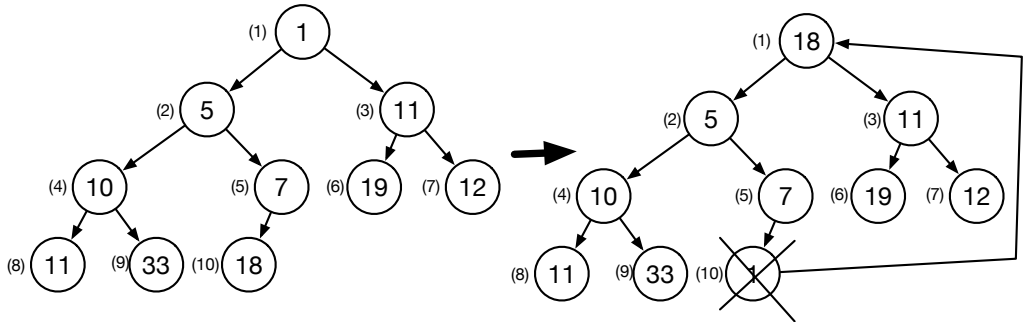
Бинарная куча: вставка элемента: реализация *siftUp*

```
void bh_siftUp(bh *t, size_t index) {
    for (size_t i = index;
         i > 1 && t->body[i] < t->body[i/2];
         i /= 2) {
        bh_swap(t, i, i/2);
    }
}
```

```
void bh_insert(bh *t, T node) {
    assert(nodes < allocated);
    t->body[++nodes] = node;
    bh_siftUp(nodes);
}
```

$$T_{Insert} = O(\log N)$$

Бинарная куча: удаление максимального (минимального)



Свойства кучи нарушены. Требуется восстановление.

Бинарная куча: восстановление свойств

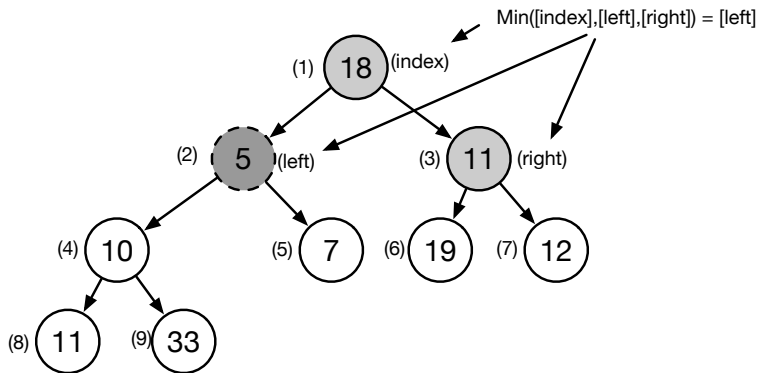
- Для восстановления свойств применяем функцию *siftDown*.

```
void bh_siftDown(bh *t, size_t index) {
    for (;;) {
        size_t left = index + index, right = left + 1;
        // Who is greater, [index], [left], [right]?
        size_t smallest = index;
        if (left <= t->nodes && t->body[left] < t->body[index])
            smallest = left;
        if (right <= t->nodes && t->body[right] < t->body[smallest])
            smallest = right;
        if (smallest == index) break;
        bh_swap(t, index, smallest);
        index = smallest;
    }
}
```

$$T_{siftDown} = O(\log N)$$

Бинарная куча: восстановление свойств

- Восстановление индекса (1)

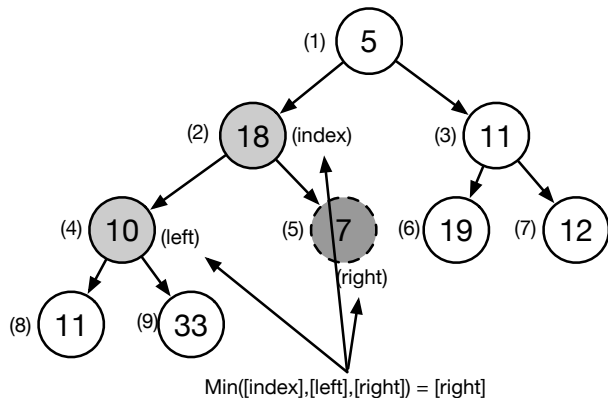


18	5	11	10	7	19	12	11	33
----	---	----	----	---	----	----	----	----

Новый индекс для восстановления (2)

Бинарная куча: восстановление свойств

- Восстановление индекса (2)

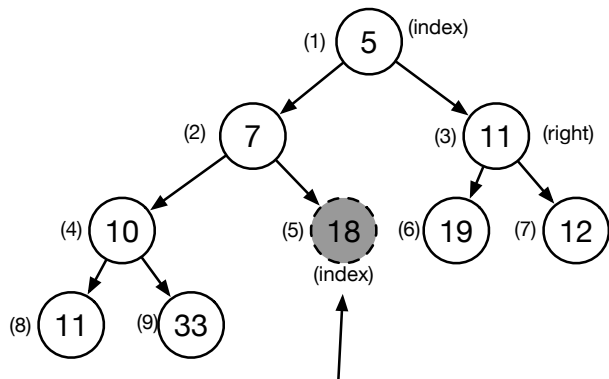


50	14	20	17	45	10	6	5	4
----	----	----	----	----	----	---	---	---

Новый индекс для восстановления (5)

Бинарная куча: восстановление свойств

- Восстановление индекса (5)



$\text{Min}([\text{index}], [\text{left}], [\text{right}]) = [\text{right}]$

5	18	11	10	7	19	12	11	33
---	----	----	----	---	----	----	----	----

Восстановление закончено.

Лемма

Лемма (об операциях *siftUp* и *siftDown*)

Пусть имеется корректная куча, представленная $a_i, i = 1 \dots N$. Пусть пришёл запрос об изменении элемента a_k на значение a'_k . Тогда:

- 1) если $a'_k < a_k$, то *siftUp*(k) восстановит свойства кучи;
- 2) если $a'_k > a_k$, то *siftDown*(k) восстановит свойства кучи.

Доказательство.

1) Индукция по k . База: $k = 1$.

Индуктивный переход. $k > 1$. Алгоритм сравнивает a'_k с $a_{k/2}$. При $a'_k \geq a_{k/2}$ куча остаётся корректной, а алгоритм ничего не исполняет. При $a'_k < a_{k/2}$ операция *swap* приводит к уменьшению значения в вершине $a_{k/2}$ и инвариант сохраняется. □

Лемма об операциях *siftUp* и *siftDown*

Доказательство.

2) Индукция по k . База: a_k — лист.

Индуктивный переход. a_k — не лист. Тогда существует a_{2k} , и, возможно, парный элемент a_{2k+1} . Пусть $a_m = \min(a_{2k}, a_{2k+1})$.

При $a'_k \leq a_m$ куча остаётся корректной, а алгоритм ничего не исполняет. При $a'_k > a_m$ операция *swap* приводит к увеличению значения в вершине a_m и уменьшению в вершине a_k . □

Операция *decreaseKey*

- Доказанная лемма даёт реализацию:

```
void bh_decreaseKey(bh *t, size_t k, T delta) {  
    assert(k < t->nodes);  
    assert(delta > 0);  
    t->body[k] -= delta;  
    pq_siftUp(t, k);  
}
```

Бинарная куча: сложность операций

- **insert** — $O(\log N)$
- **extractMin** — $O(\log N)$
- **getMin** — $O(1)$
- **decreaseKey** — $O(\log N)$

HeapSort

HeapSort

- На основе бинарной кучи можно реализовать алгоритм сортировки со сложностью $O(N \log N)$ в худшем случае.
- Как?
- Является ли отсортированным массив, являющийся представлением бинарной кучи?

HeapSort

- На основе бинарной кучи можно реализовать алгоритм сортировки со сложностью $O(N \log N)$ в худшем случае.
- Как?
- Является ли отсортированным массив, являющийся представлением бинарной кучи?
- Нет.
- Кто виноват? Что делать?

HeapSort

- Можно скомбинировать методы бинарной кучи.
 - 1 Создать бинарную кучу.
 - 2 Вставить в неё элементы массива
 - 3 Извлекать из неё максимальный (минимальный) элемент с удалением.

HeapSort

- Примерный код наивной сортировки HeapSort

```
void heapsort(int v[], size_t vsize) {
    bh *h = binary_heap_create(vsize);
    for (size_t i = 0; i < vsize; i++) {
        h->insert(v[i]);
    }
    for (size_t i = 0; i < vsize; i++) {
        v[i] = h->extractMin();
    }
    binary_heap_destroy(h);
}
```

HeapSort

- Сложность алгоритма:

- 1 Создание бинарной кучи — $T_1 = O(1)$.
- 2 Вставка N элементов — $T_2 = O(N \log N)$.
- 3 Извлечение удалением N элементов — $T_3 = O(N \log N)$.

$$T_{heapsort} = O(1) + O(N \log N) + O(N \log N) = O(N \log N)$$

HeapSort

- Реализация не особенно хороша: требуется $O(N)$ добавочной памяти на бинарную кучу.
- Небольшая хитрость — и добавочной памяти можно избежать.

HeapSort

Модифицируем `siftDown`, построив функцию `heapify`, создающую кучу с максимумом в корне в заданном массиве. `size` — размер кучи.

```
void heapify(T v[], size_t idx, size_t size)
{
    size_t curr = v[idx];
    size_t index = idx;
    for (;;) {
        size_t left = index + index + 1, right = left + 1;
        if ( left < size && v[left] > curr)
            index = left;
        if ( right < size && v[right] > a[index])
            index = right;
        if (index == idx ) break;
        v[idx] = v[index];
        v[index] = curr;
        idx = index;
    }
}
```

HeapSort

- Создаём бинарную кучу размером $size$ на месте исходного массива, переставляя его элементы.
- Затем на шаге i мы обмениваем самый приоритетный элемент кучи из позиции 0 с элементом под номером $size - i - 1$.
- Размер кучи при этом уменьшается на единицу, а самый приоритетный элемент занимает теперь положенное ему по рангу место.

```
void sort_heap(T v[], size_t size) {
    for(ssize_t i = size/2-1; i >= 0; i--) {
        heapify(a, i, size);
    }
    while( size > 1 ) {
        size--;
        swap(a[0],a[size]);
        heapify(a, 0, size);
    }
}
```

HeapSort vs QuickSort

- HeapSort гарантирует сложность $O(N \log N)$ даже в худшем случае.
- QuickSort такой сложности не гарантирует.
- Почему не забыть о QuickSort в пользу HeapSort?

HeapSort vs QuickSort

- HeapSort гарантирует сложность $O(N \log N)$ даже в худшем случае.
- QuickSort такой сложности не гарантирует.
- Почему не забыть о QuickSort в пользу HeapSort?
- Причина 1: в быстрой сортировке используется меньшее количество операций обмена с памятью.
- Причина 2: N обращений к последовательным ячейкам памяти исполняются до 10-15 раз быстрее, чем столько же обращений к случайным ячейкам памяти из-за организации кэш-памяти.