Лекция 9.
Бинарные кучи

Даниил Михайлович Берлизов
Старший преподаватель Кафедры вычислительных систем СибГУТИ
E-mail: sillyhat34@gmail.com

Курс «Структуры и алгоритмы обработки данных»
Весенний семестр, 2021 г.

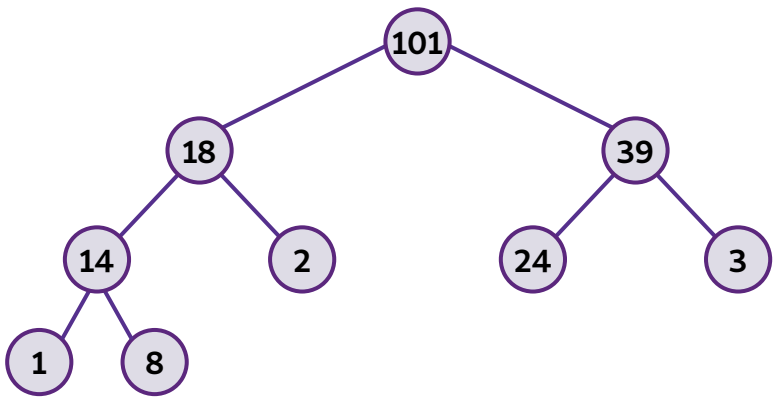
АТД «Очередь с приоритетом» (priority queue)

- **Очередь с приоритетом** (priority queue) — очередь, в которой элементы имеют приоритет (вес)
- Первым извлекается элемент с наибольшим приоритетом

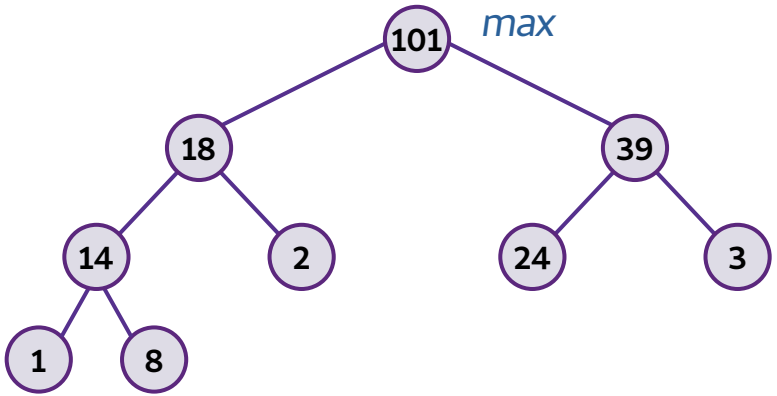
- **Поддерживаемые операции:**
  - **Insert** — добавление элемента в очередь
  - **Max** — поиск элемента с максимальным приоритетом
  - **ExtractMax** — удаление из очереди элемента с максимальным приоритетом
  - **IncreaseKey** — изменение приоритета заданного элемента
  - **Merge** — слияние двух очередей в одну

Бинарная куча (binary heap)

- **Бинарная куча** (пирамида, сортирующее дерево, binary heap) — это двоичное дерево, удовлетворяющее следующим условиям:
  - Приоритет любой вершины не меньше (≥) приоритета её потомков
  - Дерево является *полным бинарным деревом* (complete binary tree) — все уровни заполнены слева направо (возможно, за исключением последнего)

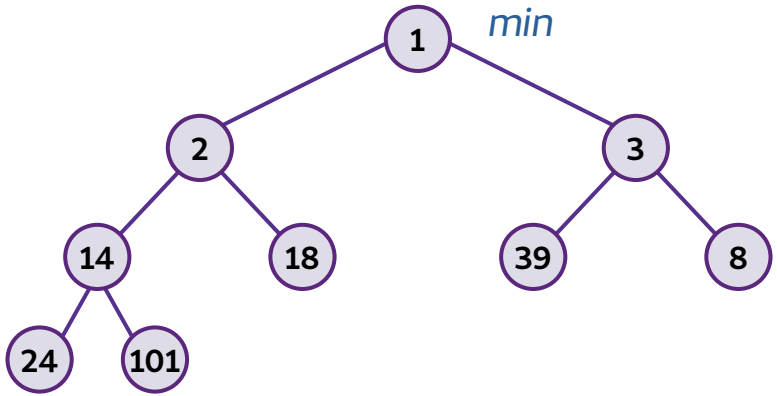


Бинарная куча (binary heap)



Невозрастающая куча
max-heap

Приоритет любой вершины
не меньше (≥) приоритета потомков

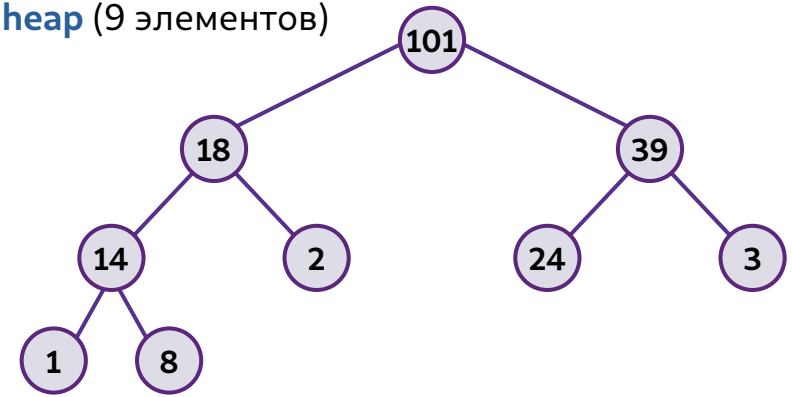


Неубывающая куча
min-heap

Приоритет любой вершины
не больше (≤) приоритета потомков

Реализация бинарной кучи на основе массива

max-heap (9 элементов)



- Корень дерева хранится в ячейке $H[1]$ — максимальный элемент
- Индекс родителя узла i: $Parent(i) = \lfloor i / 2 \rfloor$
- Индекс левого дочернего узла: $Left(i) = 2i$
- Индекс правого дочернего узла: $Right(i) = 2i + 1$

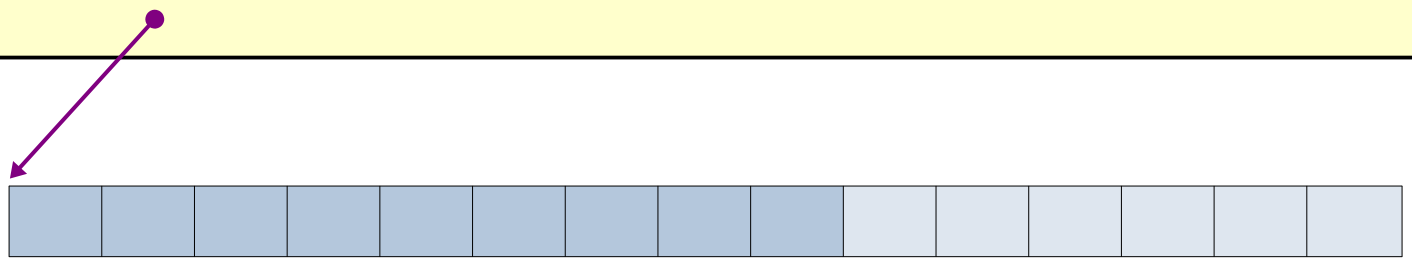
$$H[Parent(i)] \geq H[i]$$

Массив $H[1..15]$ приоритетов (ключей):

101	18	39	14	2	24	3	1	8						

Реализация бинарной кучи на основе массива

```
struct heapnode {
    int key;         /* Приоритет (ключ) */
    char *value;     /* Значение */
};

struct heap {
    int maxsize;              /* Максимальный размер кучи */
    int nnodes;               /* Число элементов */
    struct heapnode *nodes;   /* Массив элементов. Для удобства реализации элементы нумеруются с 1 */
};
```



Создание пустой кучи

```
struct heap *heap_create(int maxsize)
{
    struct heap *h;
    h = malloc(sizeof(*h));
    if (h != NULL) {
        h->maxsize = maxsize;
        h->nnodes = 0;
        h->nodes = malloc(sizeof(*h->nodes) * (maxsize + 1));  /* Последний индекс - maxsize */
        if (h->nodes == NULL) {
            free(h);
            return NULL;
        }
    }
    return h;
}
```

$T_{Create} = O(1)$

Удаление кучи. Обмен узлов кучи

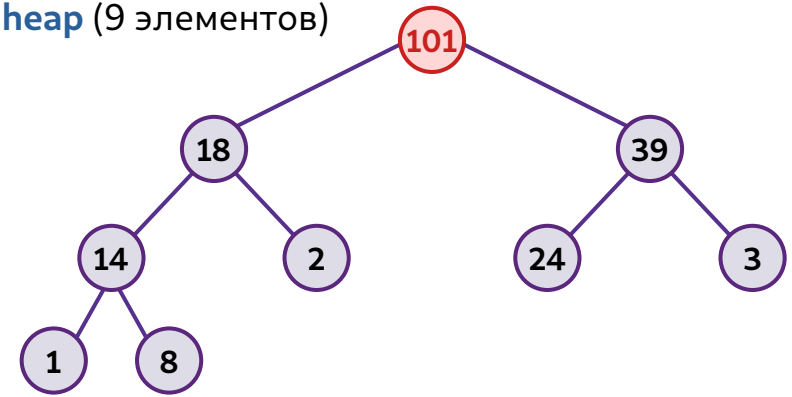
```
void heap_free(struct heap *h)
{
    free(h->nodes);
    free(h);
}

void heap_swap(struct heapnode *a, struct heapnode *b)
{
    struct heapnode temp = *a;
    *a = *b;
    *b = temp;
}
```

$T_{Free} = O(1)$

$T_{Swap} = O(1)$

Поиск максимального элемента

max-heap (9 элементов)



- Корень дерева хранится в ячейке $H[1]$ — **максимальный элемент**
- Индекс родителя узла i: $Parent(i) = \lfloor i / 2 \rfloor$
- Индекс левого дочернего узла: $Left(i) = 2i$
- Индекс правого дочернего узла: $Right(i) = 2i + 1$

$$H[Parent(i)] \geq H[i]$$

Массив $H[1..15]$ приоритетов (ключей):

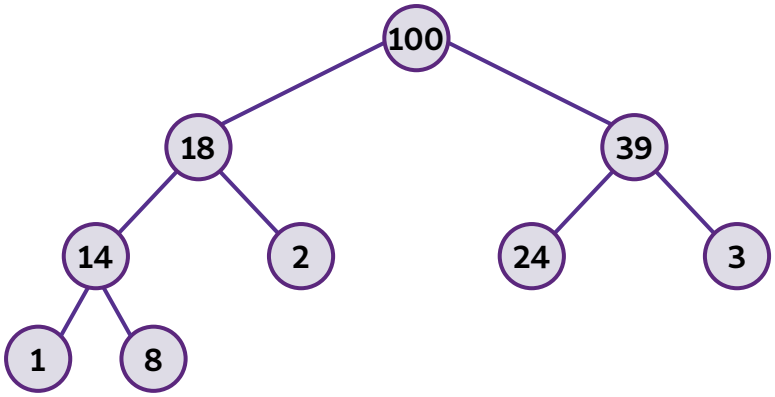
101	18	39	14	2	24	3	1	8						

Поиск максимального элемента

```
struct heapnode *heap_max(struct heap *h)
{
    if (h->nnodes == 0)
        return NULL;
    return &h->nodes[1];
}
```

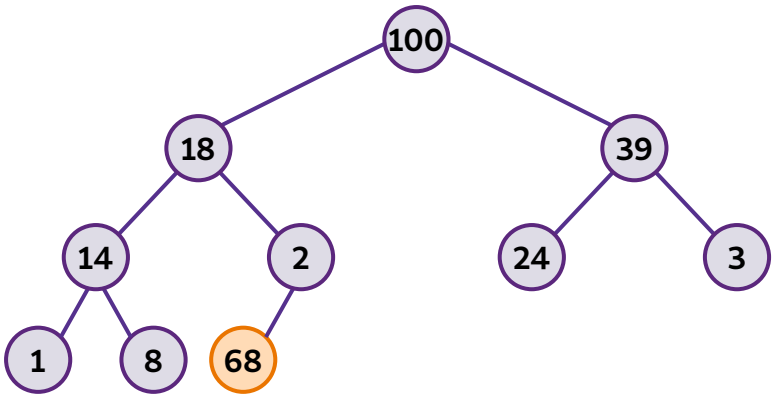
$T_{Max} = O(1)$

Вставка элемента в бинарную кучу



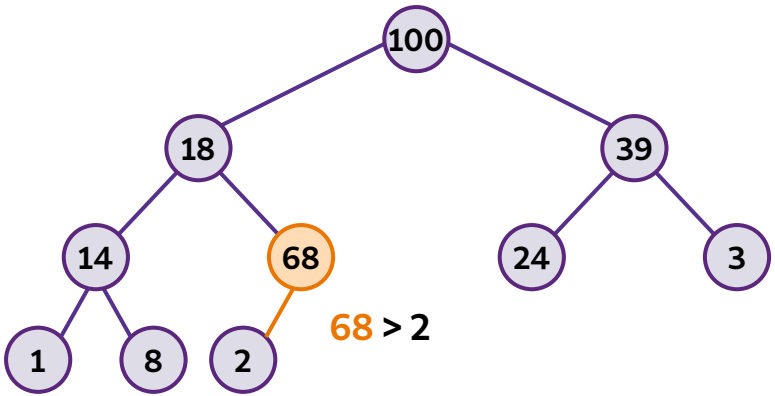
Вставка элемента с приоритетом 68

Вставка элемента в бинарную кучу



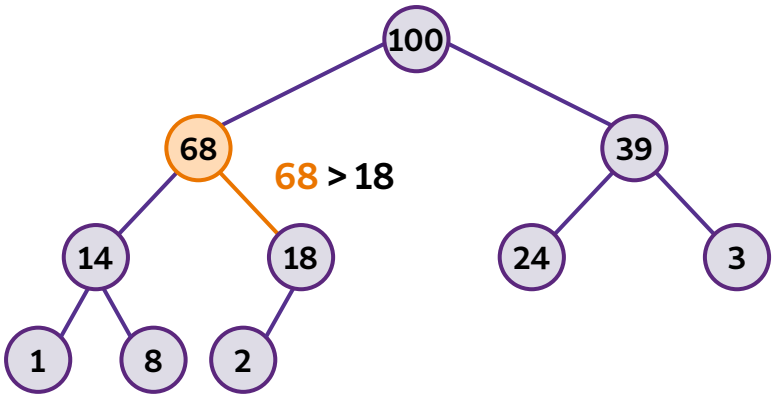
Вставка элемента с приоритетом 68

Вставка элемента в бинарную кучу



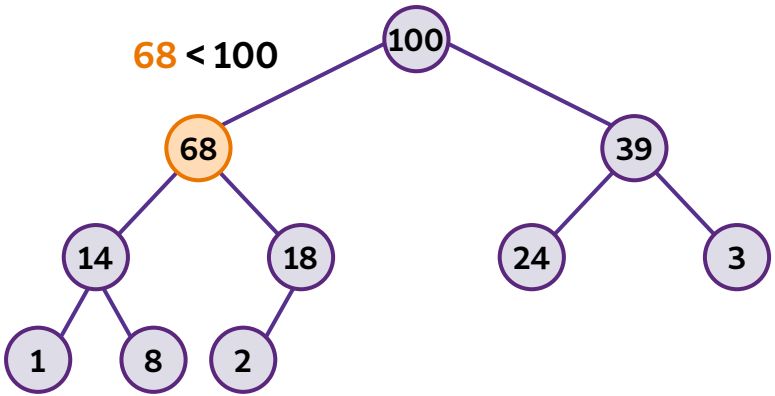
Вставка элемента с приоритетом 68

Вставка элемента в бинарную кучу



Вставка элемента с приоритетом 68

Вставка элемента в бинарную кучу



Вставка элемента с приоритетом 68

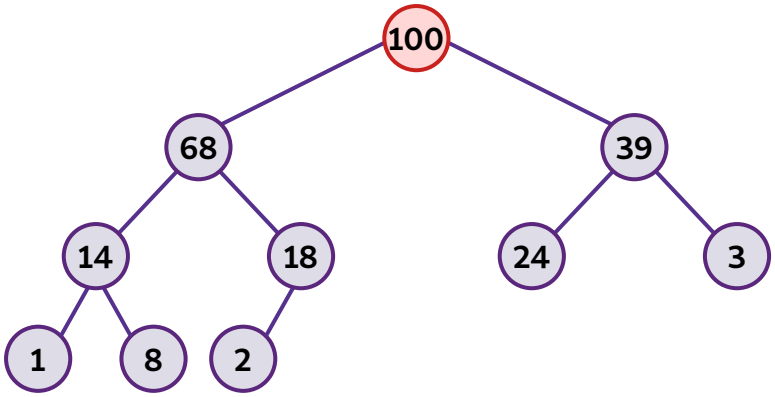
$$T_{Insert} = O(\log n)$$

Вставка элемента в бинарную кучу

```
int heap_insert(struct heap *h, int key, char *value)
{
    if (h->nnodes >= h->maxsize)    /* Переполнение кучи */
        return -1;

    h->nnodes++;
    h->nodes[h->nnodes].key = key;
    h->nodes[h->nnodes].value = value;

    /* HeapifyUp */
    for (int i = h->nnodes; i > 1 && h->nodes[i].key > h->nodes[i / 2].key; i = i / 2)
        heap_swap(&h->nodes[i], &h->nodes[i / 2];
    return 0;
}
```

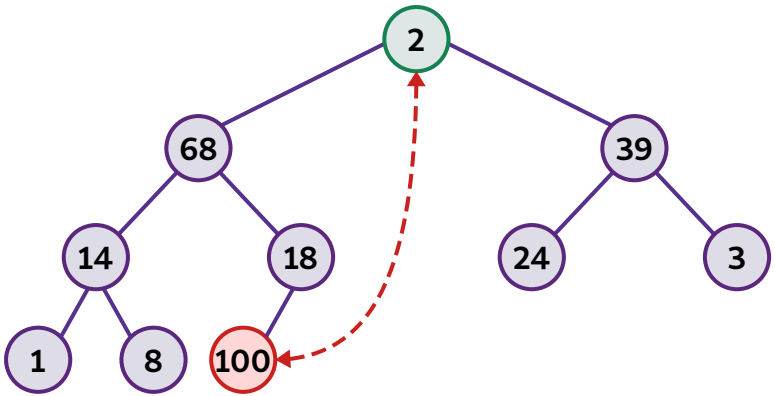
$T_{Insert} = O(\log n)$

Удаление максимального элемента



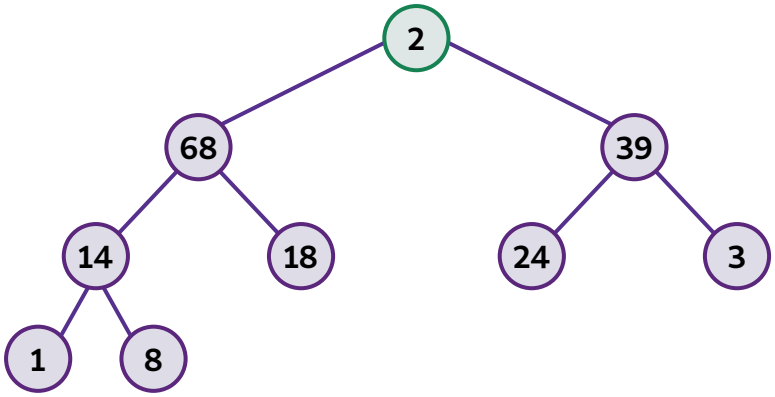
Максимальный элемент — 100

Удаление максимального элемента



1. Замена корня *H*[1] листом *H*[*n*]

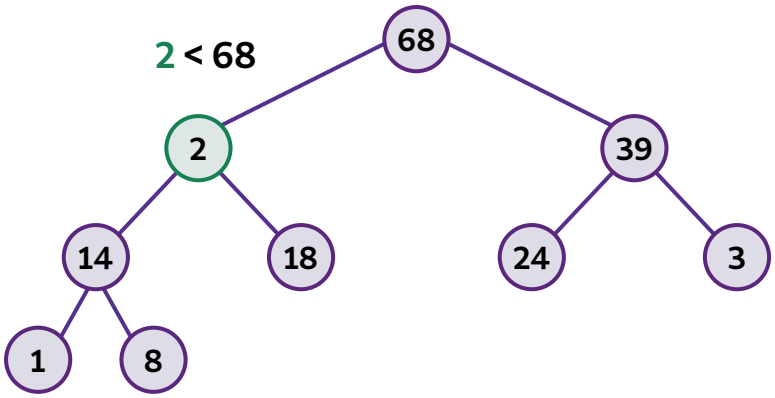
Удаление максимального элемента



2. Удаление последнего узла

$n = n - 1$

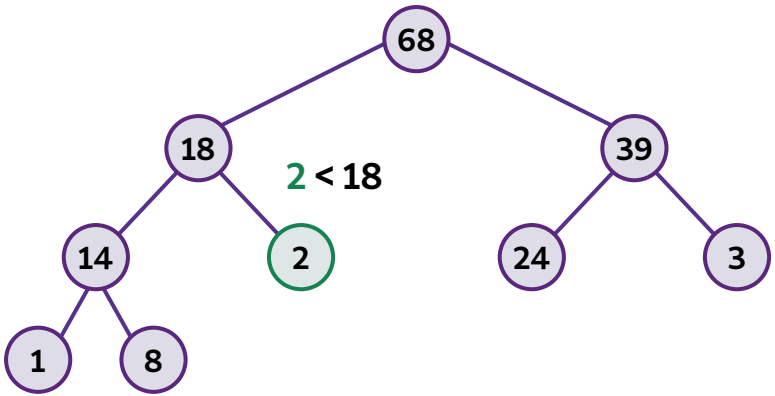
Удаление максимального элемента



3. Восстановление бинарной кучи

HeapifyDown(1)

Удаление максимального элемента



3. Восстановление бинарной кучи

HeapifyDown(1)

$$T_{DeleteMax} = O(\log n)$$

Удаление максимального элемента

```
struct heapnode heap_extract_max(struct heap *h)
{
    if (h->nnodes == 0)
        return (struct heapnode){0, NULL};

    struct heapnode maxnode = h->nodes[1];
    h->nodes[1] = h->nodes[h->nnodes--];
    heap_heapify(h, 1)

    return maxnode;
}
```

Восстановление свойств кучи

```
void heap_heapify(struct heap *h, int index)
{
    while (1) {
        int left = 2 * index;
        int right = 2 * index + 1;
        int largest = index;
        if (left <= h->nnodes && h->nodes[left].key > h->nodes[largest].key)
            largest = left;
        if (right <= h->nnodes && h->nodes[right].key > h->nodes[largest].key)
            largest = right;
        if (largest == index)
            break;
        heap_swap(&h->nodes[index], &h->nodes[largest]);
        index = largest;
    }
}
```

$T_{Heapify} = O(\log n)$

Увеличение приоритета элемента

```
int heap_increase_key(struct heap *h, int index, int newkey)
{
    if (h->nodes[index].key >= newkey)
        return -1;

    h->nodes[index].key = newkey;
    while (index > 1 && h->nodes[index].key > h->nodes[index / 2].key) {
        heap_swap(&h->nodes[index], &h->nodes[index / 2]);
        index /= 2;
    }
    return index;
}
```

$T_{IncreaseKey} = O(\log n)$

Построение бинарной кучи

- Дан неупорядоченный массив A длины n
- Требуется построить из его элементов бинарную кучу

Построение бинарной кучи

- Дан неупорядоченный массив *A* длины *n*
- Требуется построить из его элементов бинарную кучу

```
function CreateHeap(A[1..n])
    h = CreateBinaryHeap(n)        /* O(1) */
    for i = 1 to n do
        HeapInsert(h, A[i])        /* O(logn) */
    end for
end function
```

$T_{BuildHeap} = O(n \log n)$

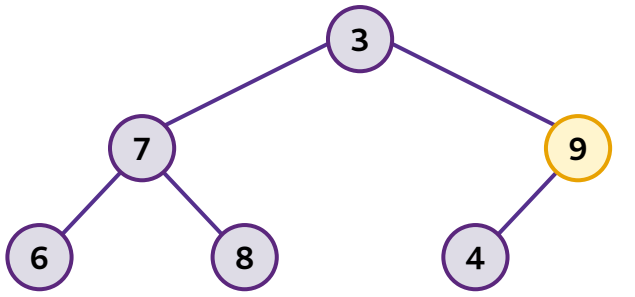
Построение бинарной кучи за время $O(n)$

- Задан массив $A[1..n]$ элементов
- Требуется построить бинарную кучу

```
function BuildMaxHeap(A[1..n], n)
    i = ⌊n / 2⌋
    while i ≥ 1 do
        HeapifyDown(A, i)
        i = i - 1
    end while
end function
```

Построение кучи из массива за время $O(n)$

```
function BuildMaxHeap(A[1..n], n)
    i = ⌊n / 2⌋
    while i ≥ 1 do
        HeapifyDown(A, i)
        i = i - 1
    end while
end function
```

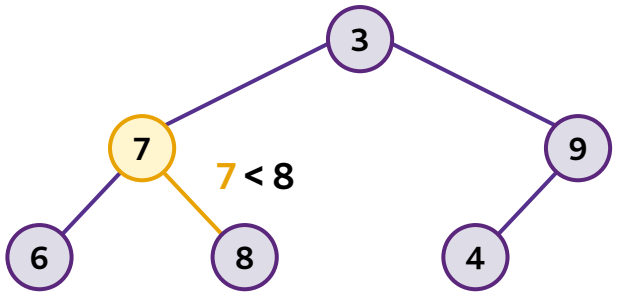


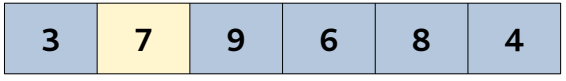
$i = \lfloor n / 2 \rfloor = 3$

HeapifyDown(3)

Построение кучи из массива за время $O(n)$

```
function BuildMaxHeap(A[1..n], n)
    i = ⌊n / 2⌋
    while i ≥ 1 do
        HeapifyDown(A, i)
        i = i - 1
    end while
end function
```



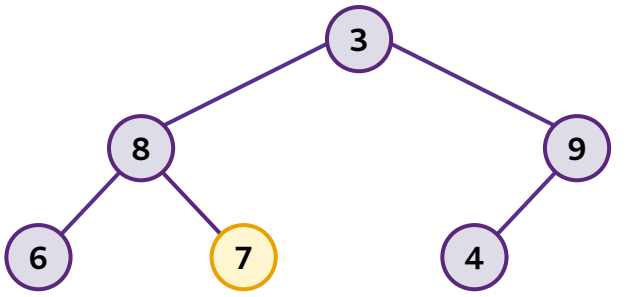


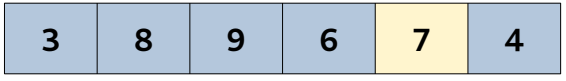
i = 2

HeapifyDown(2)

Построение кучи из массива за время $O(n)$

```
function BuildMaxHeap(A[1..n], n)
    i = ⌊n / 2⌋
    while i ≥ 1 do
        HeapifyDown(A, i)
        i = i - 1
    end while
end function
```



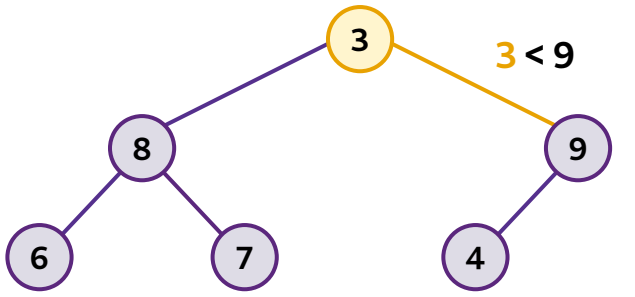


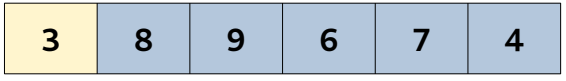
i = 2

HeapifyDown(2)

Построение кучи из массива за время $O(n)$

```
function BuildMaxHeap(A[1..n], n)
    i = ⌊n / 2⌋
    while i ≥ 1 do
        HeapifyDown(A, i)
        i = i - 1
    end while
end function
```



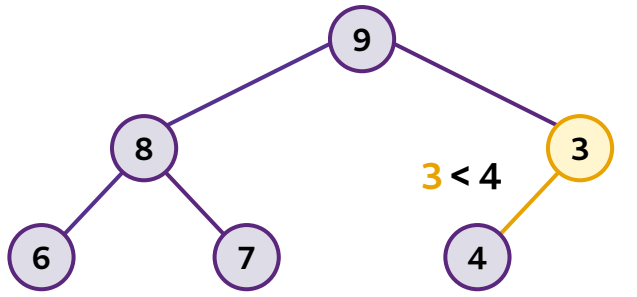


i = 1

HeapifyDown(1)

Построение кучи из массива за время $O(n)$

```
function BuildMaxHeap(A[1..n], n)
    i = ⌊n / 2⌋
    while i ≥ 1 do
        HeapifyDown(A, i)
        i = i - 1
    end while
end function
```

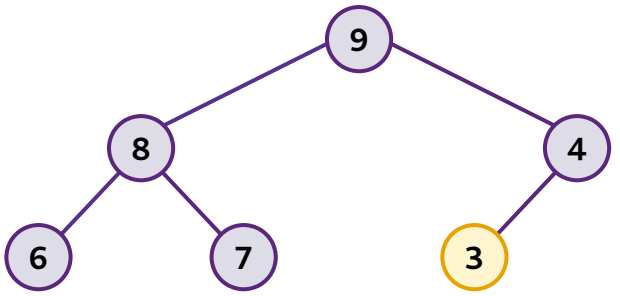


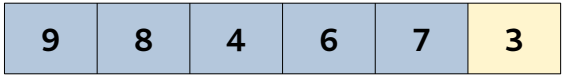
i = 1

HeapifyDown(1)

Построение кучи из массива за время $O(n)$

```
function BuildMaxHeap(A[1..n], n)
    i = ⌊n / 2⌋
    while i ≥ 1 do
        HeapifyDown(A, i)
        i = i - 1
    end while
end function
```





i = 1

HeapifyDown(1)

Построение кучи из массива за время $O(n)$

```
function BuildMaxHeap(A[1..n], n)
    i = ⌊n / 2⌋
    while i ≥ 1 do
        HeapifyDown(A, i)
        i = i - 1
    end while
end function
```

$$T(n)=\sum_{h=1}^{\lfloor \log_2 n \rfloor}\left\lceil \frac{n}{2^{h+1}} \right\rceil O(h)=O\left(n\sum_{h=1}^{\lfloor \log_2 n \rfloor}\frac{h}{2^h}\right);$$

$$\sum_{h=1}^{\lfloor \log_2 n \rfloor} h\left(\frac{1}{2}\right)^h < \sum_{h=0}^{\infty} h\left(\frac{1}{2}\right)^h = \frac{1/2}{(1-1/2)^2};$$

$$T(n)=O\left(n\sum_{h=1}^{\lfloor \log_2 n \rfloor}\frac{h}{2^h}\right)=O(n).$$

Пример использования бинарной кучи

```
int main()
{
    struct heap *h;
    struct heapnode *node;
    h = heap_create(32);
    heap_insert(h, 9, “9”);
    heap_insert(h, 7, “7”);
    heap_insert(h, 4, “4”);
    /* ... */
    node = heap_extract_max(h);
    printf(“Item: %d\n”, node.key);

    int i = heap_increase_key(h, 4, 14);
    heap_free(h);
    return 0;
}
```

Сортировка на базе бинарной кучи

- На основе бинарной кучи можно реализовать алгоритм сортировки с вычислительной сложностью $O(n \log n)$ в худшем случае

Как?

Сортировка на базе бинарной кучи

- На основе бинарной кучи можно реализовать алгоритм сортировки с вычислительной сложностью ***O(nlogn)*** в худшем случае

Как?

```
function HeapSort(A[1..n], n)
    h = CreateBinaryHeap(n)          /* O(1) */
    for i = 1 to n do
        HeapInsert(h, A[i])          /* O(logn) */
    end for
    for i = n to 1 do
        A[i] = HeapExtractMax(h)     /* O(logn) */
    end for
end function
```

$$T_{HeapSort} = 1 + n\log n + n\log n = O(n\log n)$$

Сортировка на базе бинарной кучи

- На основе бинарной кучи можно реализовать алгоритм сортировки с вычислительной сложностью ***O(nlogn)*** в худшем случае

Как?

```
function HeapSort(A[1..n], n)
    BuildMaxHeap(A, n)
    i = n
    while i ≥ 2 do
        Swap(A[1], A[i])
        i = i - 1
        HeapifyDown(A, 1)
    end while
end function
```

$$T_{HeapSort} = O(n) + O((n-1)\log n) = O(n\log n)$$

Очереди с приоритетом (priority queues)

- В таблице приведены трудоёмкости операций очереди с приоритетом в худшем случае
- Символом «*» отмечена амортизированная сложность операций

Операция	Binary heap	Binomial heap	Fibonacci heap	Pairing heap	Brodal heap
FindMax	$\Theta(1)$	$O(\log n)$	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
DeleteMax	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)^*$	$O(\log n)^*$	$O(\log n)$
Insert	$\Theta(\log n)$	$O(\log n)$	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
IncreaseKey	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)^*$	$O(\log n)^*$	$\Theta(1)$
Merge/Union	$\Theta(n)$	$\Omega(\log n)$	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$

Домашнее чтение

- **[DSABook] Глава 14. «Бинарные кучи»**
- [CLRS, Глава 6] Анализ вычислительной сложности алгоритма построения бинарной кучи (BuildMaxHeap) за время $O(n)$

ご清聴ありがとうございました!

Даниил Михайлович Берлизов
Старший преподаватель Кафедры вычислительных систем СибГУТИ
E-mail: sillyhat34@gmail.com

Курс «Структуры и алгоритмы обработки данных»
Весенний семестр, 2021 г.