Очереди с приоритетом. Бинарные кучи. Реализация бинарной кучи на основе массива. Построение бинарной кучи за время O(n).

Очередь с приоритетом

- Очередь с приоритетом *(priority queue)* – очередь, в которой элементы имеют приоритет *(вес)*
- Первым извлекается элемент с наибольшим приоритетом (ключом).
- Поддерживаемые операции:
  - **Insert** – добавление элемента в очередь.
  - **Max** – возвращает элемент с максимальным приоритетом.
  - **ExtractMax** – удаляет из очереди элемент с максимальным приоритетом.
  - **IncreaseKey** – изменяет значение приоритета заданного элемента.
  - **Merge** – сливает две очереди в одну.

Значение (value)	Приоритет (priority)
Кот	34
Волк	45
Рысь	21

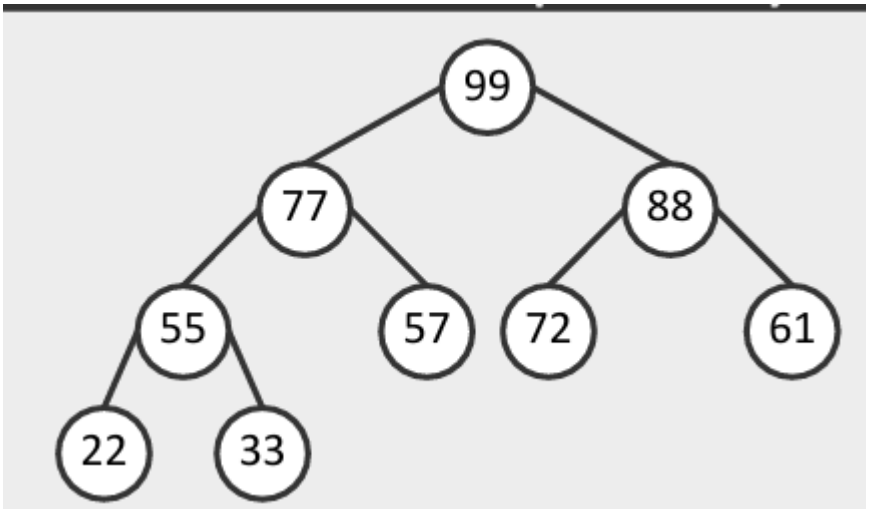
Бинарная куча *(binary heap)*

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

Есть два вида бинарной кучи:

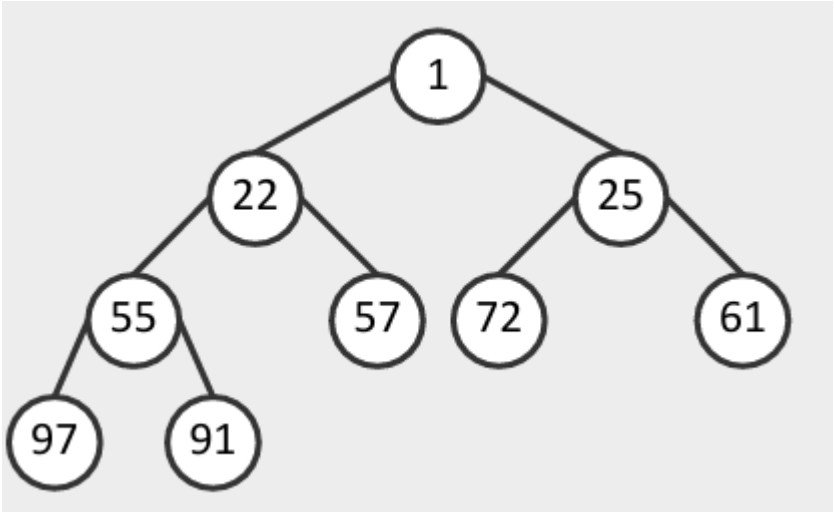
Невозрастающая куча (*max-heap*):

- Приоритет любой вершины **не меньше** *(≥)* приоритета потомков

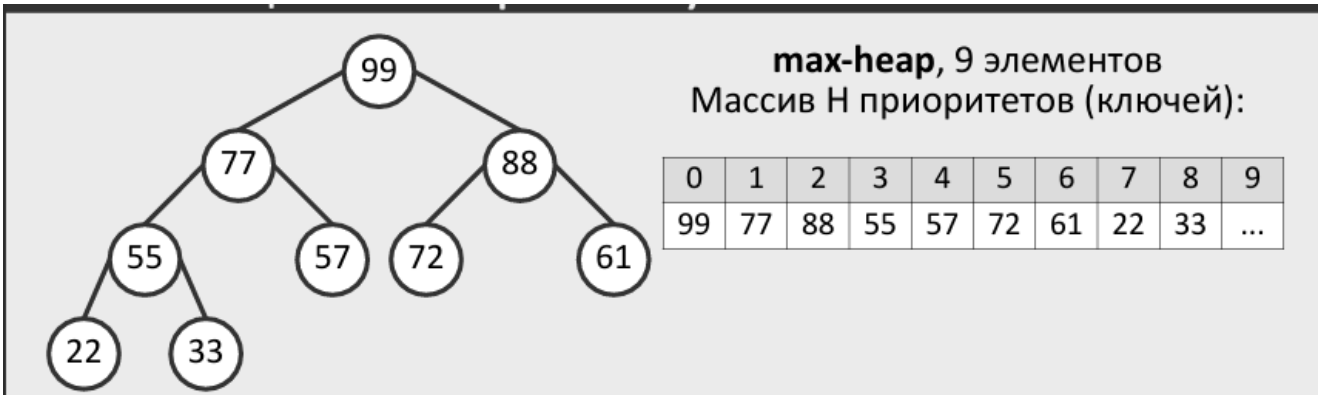


Неубывающая куча (*min-heap*)

- Приоритет любой вершины **не больше** *(≤)* приоритета потомков



Реализация бинарной кучи на основе массива



- Корень дерева хранится в ячейке H[0] – максимальный элемент
- Индекс родителя узла i: **Parent**(i) = [(i – 1) / 2]
- Индекс левого дочернего узла: **Left**(i) = 2i + 1
- Индекс правого дочернего узла: **Right**(i) = 2i + 2
- H[Parent(i)] ≥ H[i]

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

struct heap {
    int maxsize; /* Максимальный размер кучи */
    int nnodes; /* Число элементов */
    struct heapnode *nodes; /* Массив узлов [1...maxsize] */
};
```

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;
        /* Последний индекс – maxsize */
        h->nodes = malloc(sizeof(*h->nodes) * (maxsize + 1));
        if (h->nodes == NULL) {
            free(h);
            return NULL;
        }
    }
    return h;
}
```

Удаление кучи

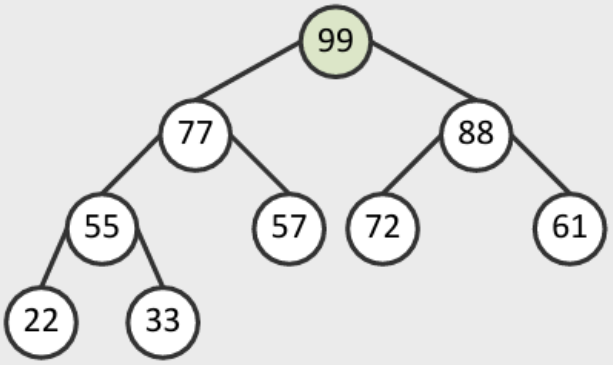
```
void heap_free(struct heap *h)
{
    free(h->nodes);
    free(h);
}

void heap_swap(struct heapnode *a, struct heapnode *b)
```

```
{
    struct heapnode temp;
    temp = *a;
    *a = *b;
    *b = temp;
}
```

Поиск максимального элемента

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

Поиск максимального элемента

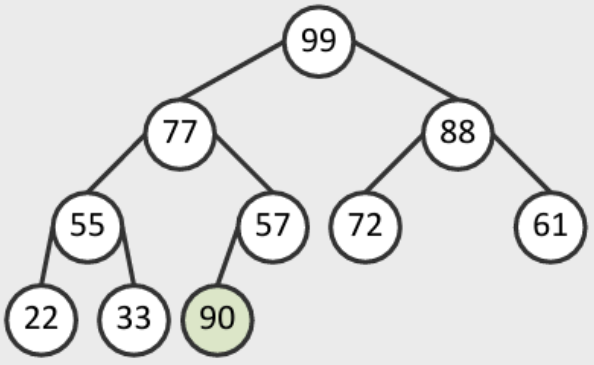


max-heap, 9 элементов
Массив H приоритетов (ключей):

0	1	2	3	4	5	6	7	8	9
99	77	88	55	57	72	61	22	33	...

- Корень дерева хранится в ячейке H[0] – максимальный элемент

Вставка элемента в бинарную кучу



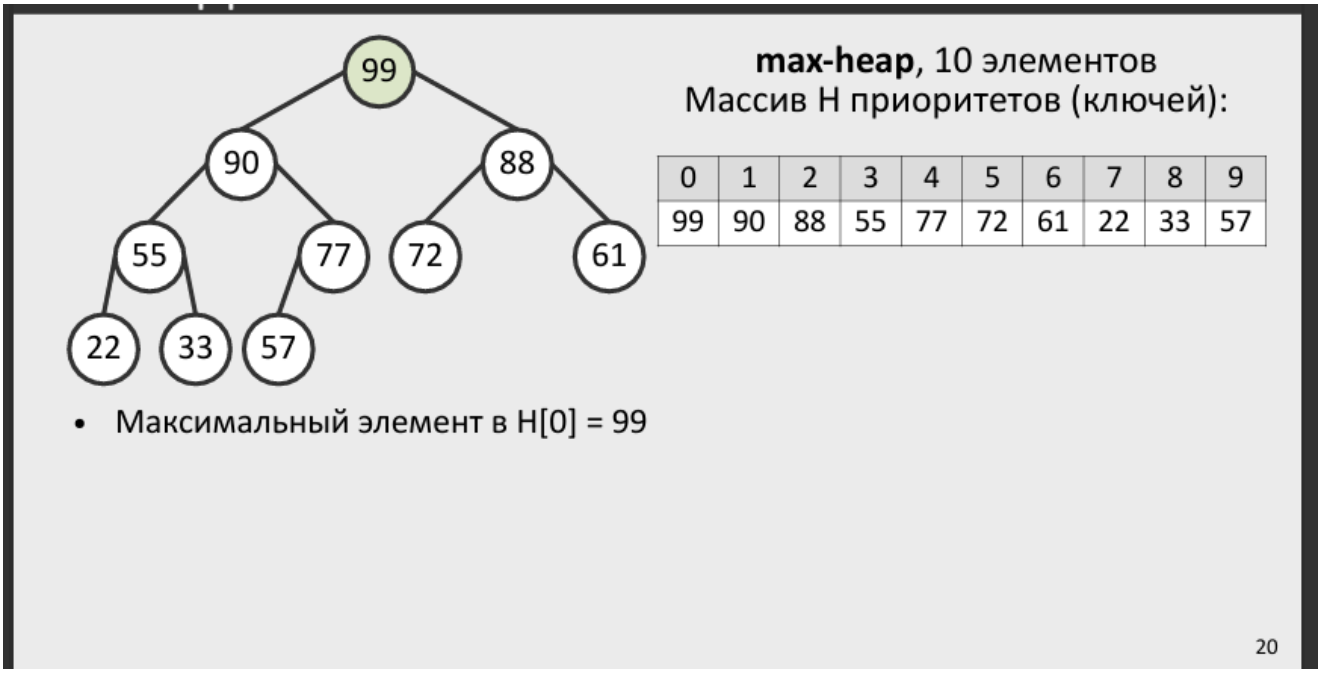
max-heap, 9 элементов
Массив H приоритетов (ключей):

0	1	2	3	4	5	6	7	8	9
99	77	88	55	57	72	61	22	33	90

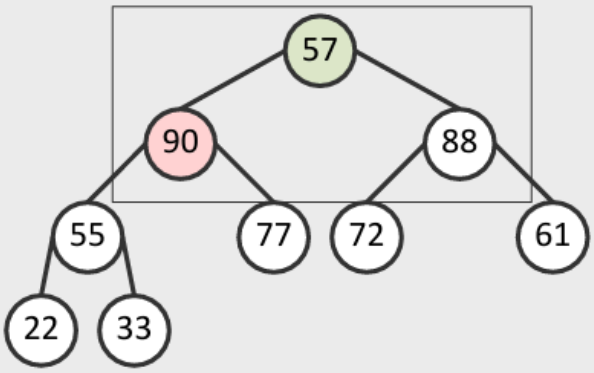
- Вставка элемента с приоритетом 90
- Добавляется на **текущий уровень** или на новый, если текущий заполнен
- **HeapifyUp** (двигаем элемент вверх по куче, если свойства нарушены)

Вставка элемента в бинарную кучу

```c
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;
}
```



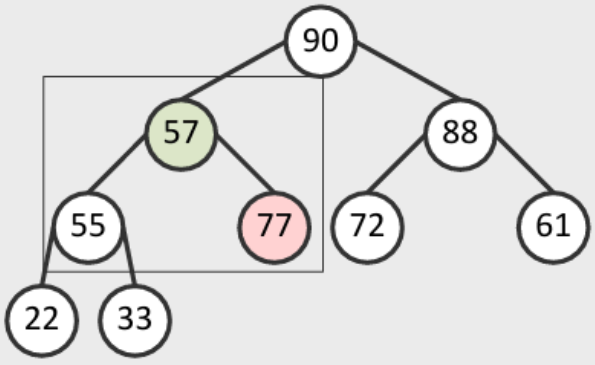
Удаление максимального элемента



max-heap, 10 элементов
Массив H приоритетов (ключей):

0	1	2	3	4	5	6	7	8	9
57	90	88	55	77	72	61	22	33	...

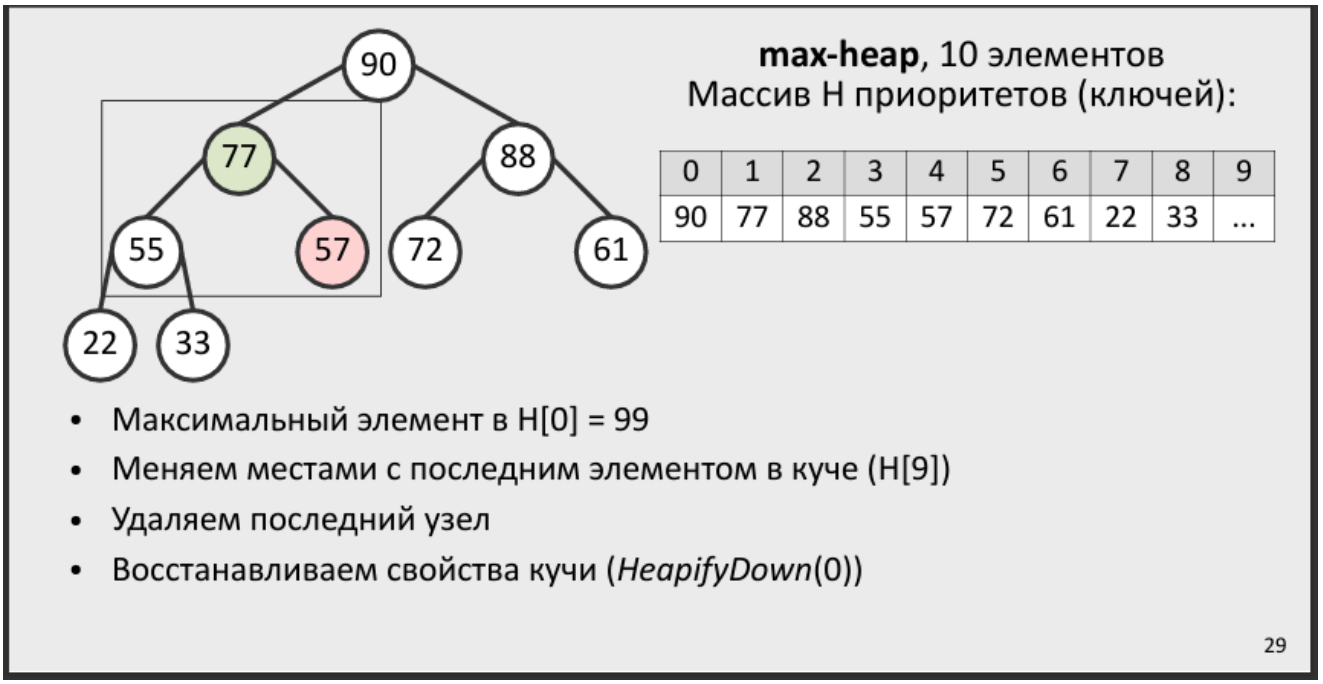
- Максимальный элемент в H[0] = 99
- Меняем местами с последним элементом в куче (H[9])
- Удаляем последний узел
- Восстанавливаем свойства кучи (*HeapifyDown*(0))



max-heap, 10 элементов
Массив H приоритетов (ключей):

0	1	2	3	4	5	6	7	8	9
90	57	88	55	77	72	61	22	33	...

- Максимальный элемент в H[0] = 99
- Меняем местами с последним элементом в куче (H[9])
- Удаляем последний узел
- Восстанавливаем свойства кучи (*HeapifyDown*(0))



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

```
/**
 * Функция "просеивания вниз" (heapify) для поддержки свойств max-кучи
 *
 * @param h     - указатель на структуру кучи
 * @param index - индекс узла, с которого начинается просеивание
 */
void heap_heapify(struct heap *h, int index)
{
    // Бесконечный цикл, выход через break при выполнении условия
    while (1) {
        // Вычисляем индексы левого и правого потомков
        int left = 2 * index;      // Левый потомок
        int right = 2 * index + 1; // Правый потомок
        int largest = index;       // Изначально считаем текущий узел
наибольшим

        // Сравниваем с левым потомком (если он существует)
```

```
        if (left <= h->nnodes &&
            h->nodes[left].key > h->nodes[largest].key)
            largest = left; // Левый потомок становится новым кандидатом

        // Сравниваем с правым потомком (если он существует)
        if (right <= h->nnodes &&
            h->nodes[right].key > h->nodes[largest].key)
            largest = right; // Правый потомок становится новым кандидатом

        // Если текущий узел больше обоих потомков - куча корректна
        if (largest == index)
            break; // Выход из цикла

        // Меняем местами текущий узел с наибольшим потомком
        heap_swap(&h->nodes[index], &h->nodes[largest]);

        // Продолжаем просеивание с новой позиции
        index = largest;
    }
}
```

Увеличение приоритета элемента

```
/**
 * Увеличивает значение ключа элемента в куче и восстанавливает свойства
кучи
 *
 * @param h       - указатель на структуру кучи (max-куча)
 * @param index   - индекс изменяемого элемента (1-based)
 * @param newkey  - новое значение ключа
 * @return        - новый индекс элемента или -1 при ошибке
 */
int heap_increase_key(struct heap *h, int index, int newkey)
{
    // Проверка, что новое значение действительно больше текущего
    if (h->nodes[index].key >= newkey)
        return -1; // Нельзя уменьшать ключ в max-куче

    // Устанавливаем новое значение ключа
    h->nodes[index].key = newkey;

    // Просеивание вверх для восстановления свойств кучи
    while (index > 1 &&                          // Пока не дошли до корня
           h->nodes[index].key > h->nodes[index / 2].key)  // И текущий узел
больше родителя
    {
        // Меняем местами с родителем
        heap_swap(&h->nodes[index], &h->nodes[index / 2]);
```

```
        // Переходим к рассмотрению родительского узла
        index /= 2;
    }

    // Возвращаем новый индекс элемента
    return index;
}
```

Построение бинарной кучи за время O(n).

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

```
void *build_max_heap(int *arr, int n)
{
    for (int i = floor(n / 2) - 1; i > -1; i--)
        heapify_down(arr, n, i);
}
```

```
/**
 * Функция "просеивания вниз" (heapify down) для поддержания свойств max-
кучи
 *
 * @param arr - указатель на массив, представляющий кучу
 * @param n   - размер кучи (количество элементов в массиве)
 * @param i   - индекс элемента, с которого начинается просеивание (0-based)
 */
void heapify_down(int *arr, int n, int i)
{
    // Цикл продолжается, пока текущий узел находится в пределах кучи
    while (i < n) {
        // Вычисляем индексы левого и правого потомков (0-based индексация)
        int left = 2 * i + 1;   // Левый потомок
        int right = 2 * i + 2;  // Правый потомок
        int largest = i;        // Изначально считаем текущий узел
наибольшим

        // Сравниваем с левым потомком, если он существует
        if (left < n && arr[left] > arr[largest]) {
            largest = left;  // Левый потомок больше текущего узла
        }

        // Сравниваем с правым потомком, если он существует
        if (right < n && arr[right] > arr[largest]) {
            largest = right; // Правый потомок больше текущего наибольшего
```

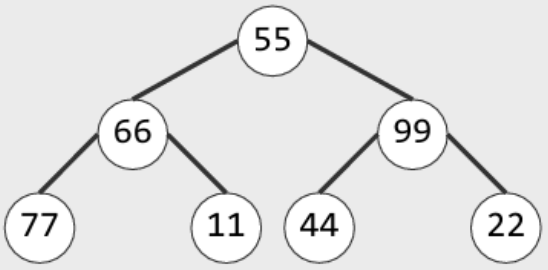
```
        }

        // Если найден потомок больше текущего узла
        if (largest != i) {
            // Меняем местами текущий узел с наибольшим потомком
            swap(arr[i], arr[largest]);
            // Продолжаем просеивание с новой позиции
            i = largest;
        } else {
            // Если оба потомка не больше текущего узла - свойство кучи
восстановлено
            break;
        }
    }
}
```

Построение бинарной кучи за O(n)

- Дан неупорядоченный массив A длины $n = 7$
- Требуется построить из его элементов бинарную кучу

```
void *build_max_heap(int *arr,
                     int n)
{
    for (int i = floor(n / 2) - 1;
         i > -1; i--) {
        heapify_down(arr, n, i);
    }
}
```

$T_{BuildHeap} = O(n)$

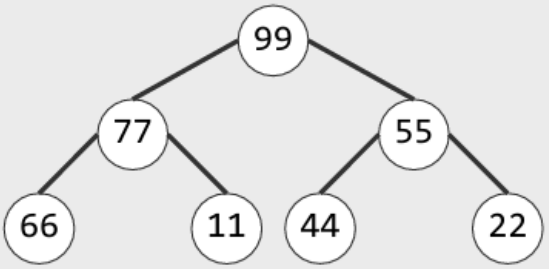


0	1	2	3	4	5	6
55	66	99	77	11	44	22

- Дан неупорядоченный массив A длины $n = 7$
- Требуется построить из его элементов бинарную кучу

```
void *build_max_heap(int *arr,
                     int n)
{
    for (int i = floor(n / 2) - 1;
         i > -1; i--) {
        heapify_down(arr, n, i);
    }
}
```

$T_{BuildHeap} = O(n)$



0	1	2	3	4	5	6
99	77	55	66	11	44	22

55

(Последовательно проходимся по элементам и строим кучу не нарушая её свойств, более подробно есть в презентации , ~20 слайдов не стал сюда добавлять)

Очередь с приоритетом (priority queue)

- В таблице приведены трудоемкости операций очереди с приоритетом (в худшем случае, worst case)
- Символом «*» отмечена амортизированная сложность операций

Операция	Binary heap	Binomial heap	Fibonacci heap	Pairing heap	Brodal heap
FindMin	Θ(1)	O(log*n*)	Θ(1)	Θ(1)*	Θ(1)
DeleteMin	Θ(log*n*)	Θ(log*n*)	O(log*n*)*	O(log*n*)*	O(log*n*)
Insert	Θ(log*n*)	O(log*n*)	Θ(1)	Θ(1)*	Θ(1)
DecreaseKey	Θ(log*n*)	Θ(log*n*)	Θ(1)*	O(log*n*)*	Θ(1)
Merge/Union	Θ(*n*)	Ω(log*n*)	Θ(1)	Θ(1)*	Θ(1)

82