

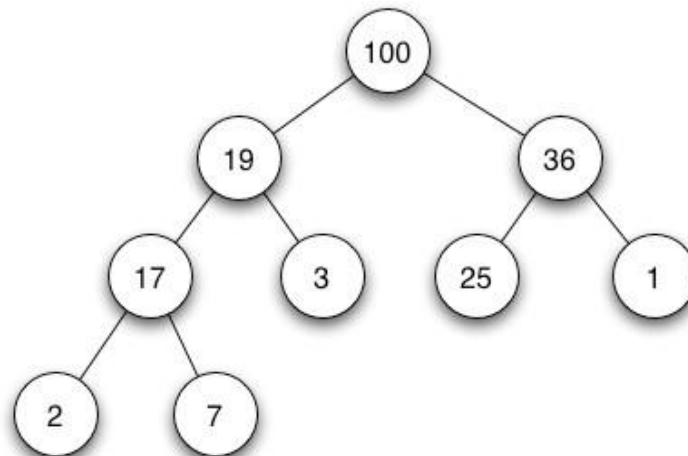
Двоичная куча

Качанов Станислав

Двоичная куча – пирамида (binary heap)

- Двоичная куча (пирамида, сортирующее дерево, binary heap) – это двоичное дерево, удовлетворяющее следующим условиям:

- Приоритет любой вершины не меньше (\geq), приоритета ее потомков
- Дерево является *полным двоичным деревом* (complete binary tree) – все уровни заполнены слева направо (возможно за исключением последнего)



История появления

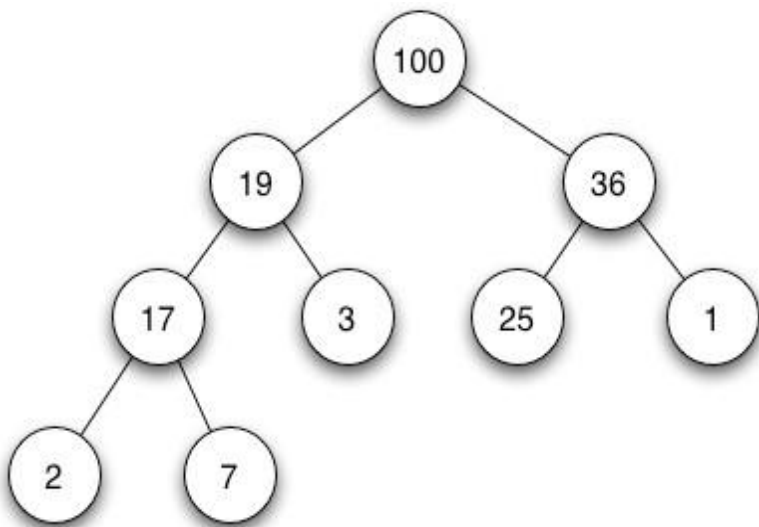
Бинарная куча была Джоном Уильямом Джозефом Уильямсом в 1964 году как структура данных для heapsort(пирамидальной сортовки).

Но наибольшего применения достигла лишь в 1990-х годах, в эпоху повсеместного использования компьютеров. В том числе двоичную кучу существенно популяризировал Чарльз Лейзерсон, который также использовал её в разработке собственного языка программирования Clik.

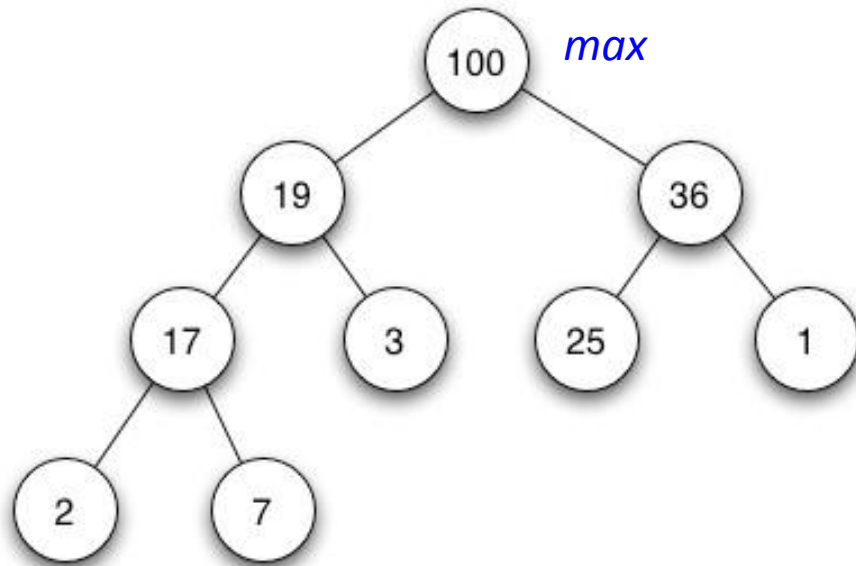


Двоичная куча

- ❑ Приоритет в любой вершине не меньше, чем приоритет её потомков
- ❑ Глубина всех листьев (расстояние до корня) отличается не более чем на 1 слой.
- ❑ Последний слой заполняется слева направо без «дырок».

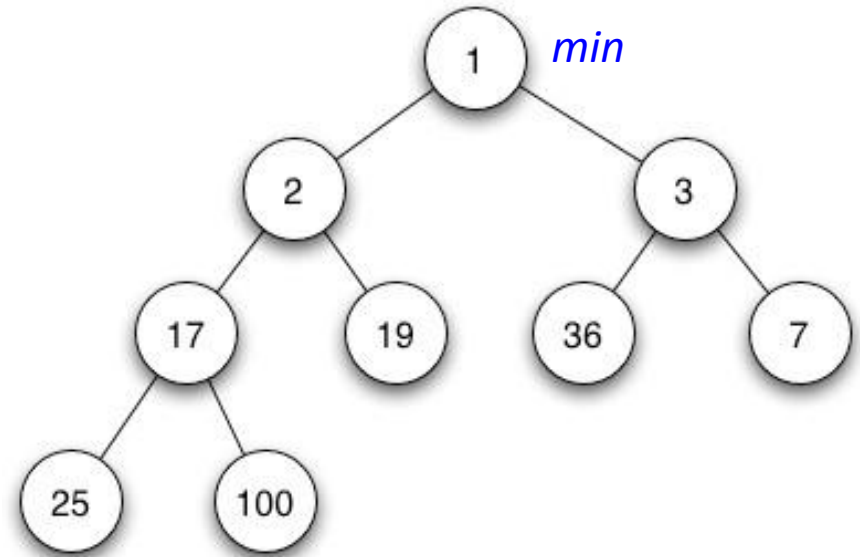


Бинарная куча – пирамида (binary heap)



Невозрастающая пирамида max-heap

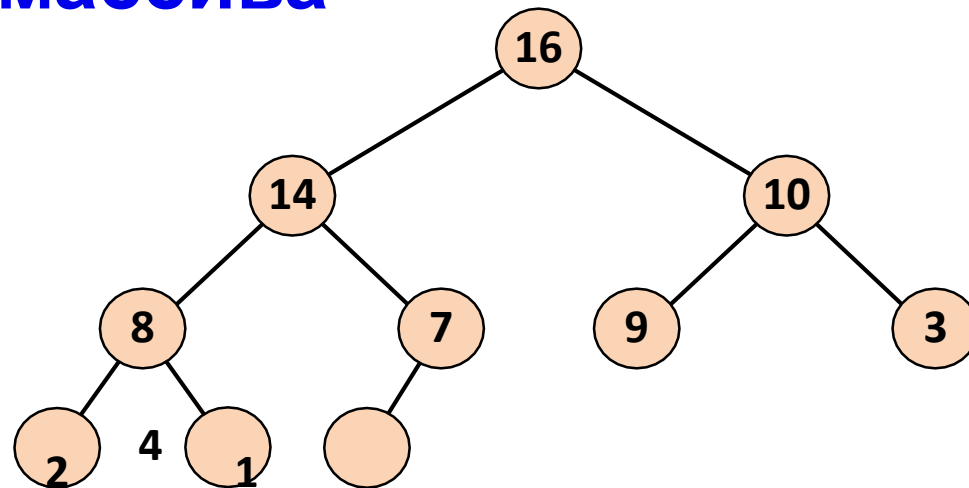
Приоритет любой вершины **не меньше (\geq)**, приоритета потомков



Неубывающая пирамида min-heap

Приоритет любой вершины **не больше (\leq)**, приоритета потомков

Реализация бинарной кучи на основе массива



max-heap (10
элементов)

Массив $H[1..14]$ приоритетов

(ключей):

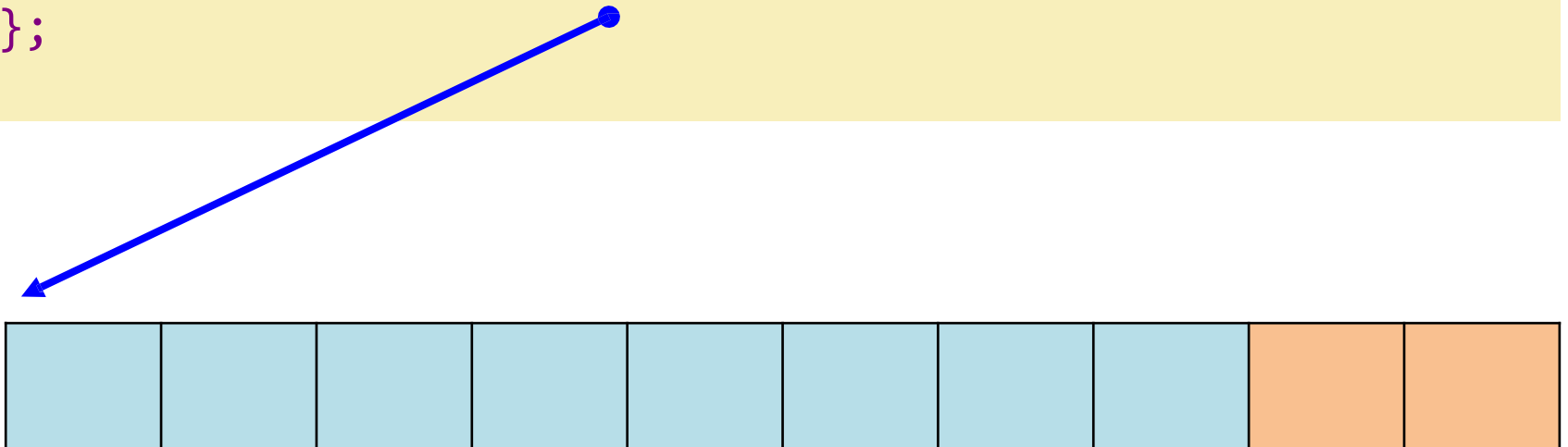
16	14	10	8	7	9	3	2	4	1				
----	----	----	---	---	---	---	---	---	---	--	--	--	--

- Корень дерева храниться 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: $Right(i) = 2i + 1$

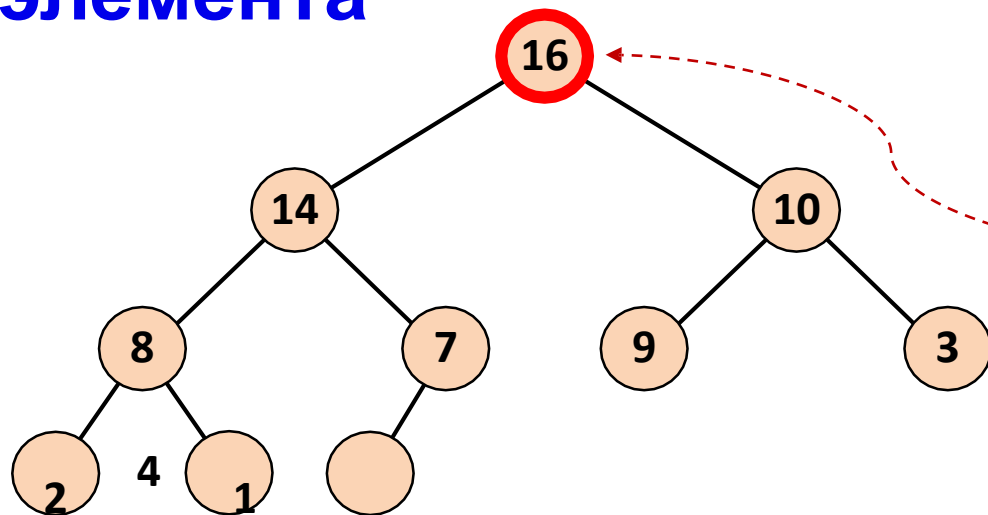
Реализация бинарной кучи на основе

массива

```
struct heapnode {  
    int key;           /* Priority (key) */  
    char *value;      /* Data */  
};  
  
struct heap {  
    int maxsize;      /* Размер массива*/  
    int nnodes;       /* Номер ключа*/  
    struct heapnode *nodes; /* Nodes: [0..maxsize] */  
};
```



Поиск максимального элемента



max-heap (10 элементов)
 Максимальный элемент хранится в корне max-heap

Массив $H[1..14]$ приоритетов

(ключей):

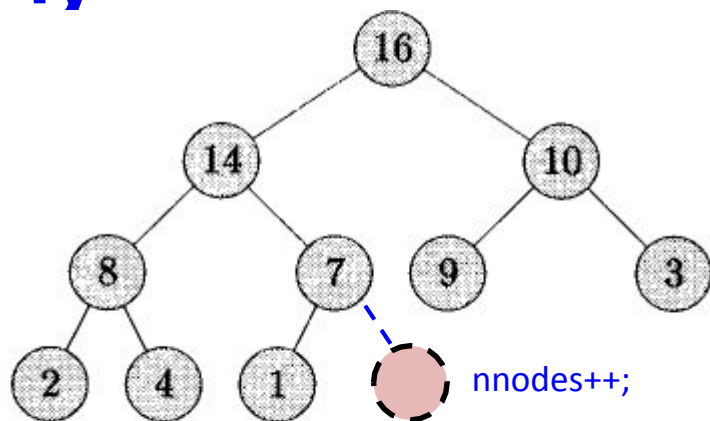
16	14	10	8	7	9	3	2	4	1				
----	----	----	---	---	---	---	---	---	---	--	--	--	--

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

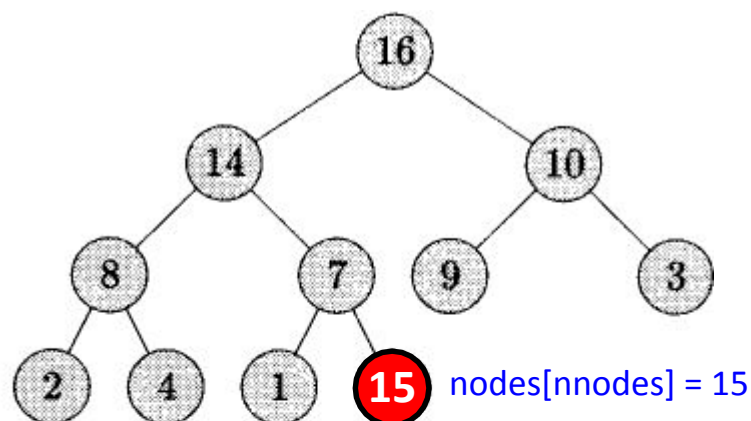
$$H[Parent(i)] > H[i]$$

Вставка элемента в бинарную

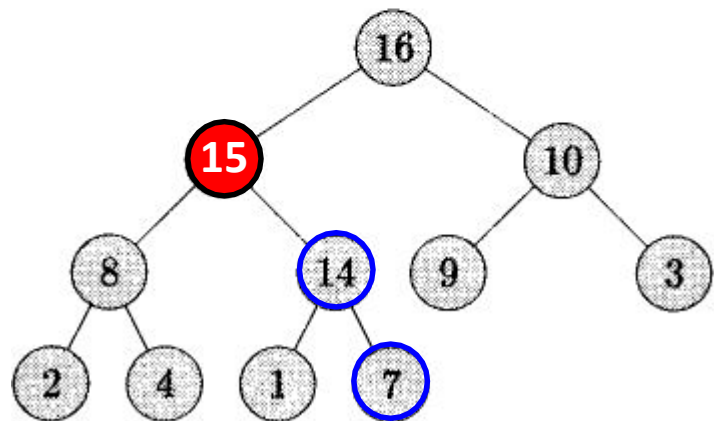
кучу



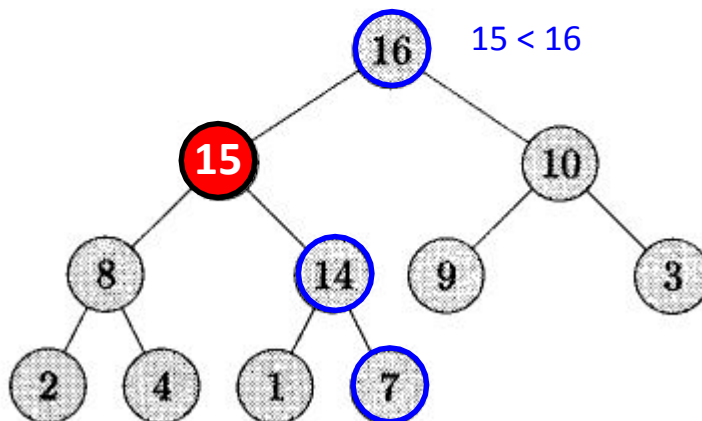
(a)



(б)



(в) `15 > 7 => swap(15, 7)`
`15 > 14 => swap(15, 14)`



(г)

Вставка элемента с приоритетом 15

Вставка элемента в бинарную

vvvv

```
void addelem(int n) {
    int i, parent;
    i = HeapSize;
    h[i] = n;
    parent = (i-1)/2;
    while(parent >= 0 && i > 0) {
        if(h[i] > h[parent]) {
            int temp = h[i];
            h[i] = h[parent];
            h[parent] = temp;
        }
        i = parent;
        parent = (i-1)/2;
    }
    HeapSize++;
}
```

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

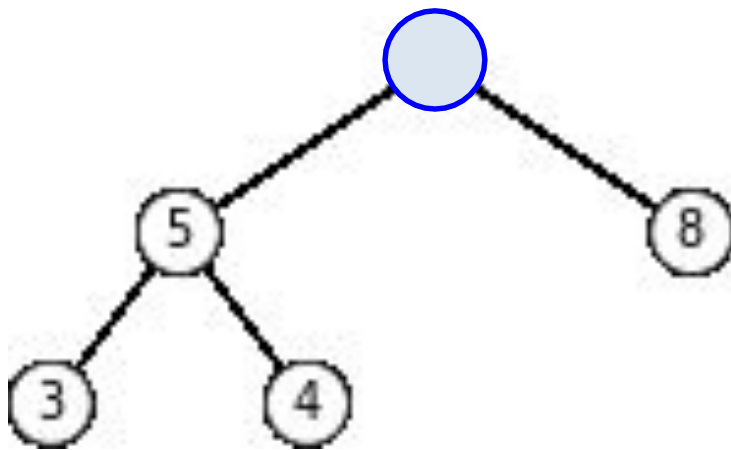
Поиск максимального

элемента

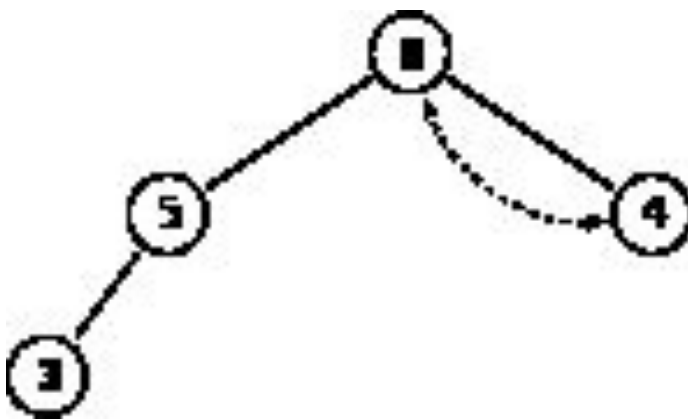
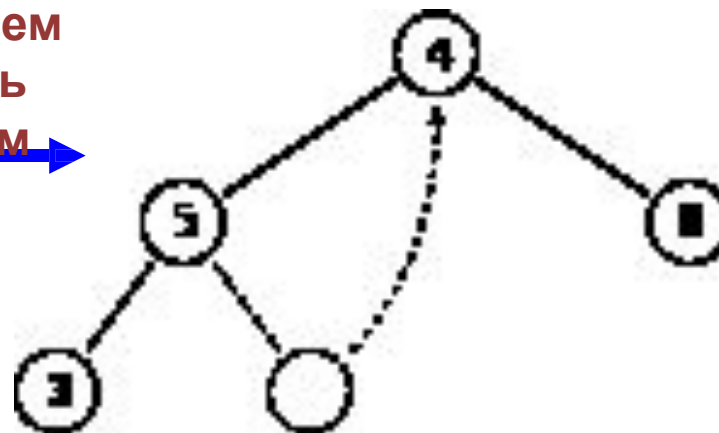
```
struct heapnode *heap_max(struct heap *h)
{
    if (h->nnodes == 0)
        return NULL;
    return &h->nodes[1];
}
```

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

Удаление максимального элемента



Заменяем
корень
листом



Восстанавливаем свойства
кучи
`heap_heapify(N, 1)`

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];
    h->nnodes--;
    heap_heapify(h, 1);

    return maxnode;
}
```

Удаление максимального элемента

```
int getmax() {  
    int x;  
    x = h[0];  
    h[0] = h[HeapSize-1];  
    HeapSize--;  
    heapify(0);  
    return(x);  
}  
};
```

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;
        /* Heap nodes [0, 1, ..., maxsize] */
        h->nodes = malloc(sizeof(*h->nodes) * (maxsize +
1));
        if (h->nodes == NULL) {
            free(h);
            return NULL;
        }
    }
    return h;
}
```

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

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;
}
```


Восстановление свойств кучи

(max-heap)

```
void heap_heapify(struct heap *h, int index)
{
    for (;;) {
        int left = 2 * index;
        int right = 2 * index + 1;

        // Find largest key: A[index], A[left] and
        // A[right]
        int largest = index;
        if (left <= h->nnodes &&
            h->nodes[left].key > h->nodes[index].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;
    }
}
```

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

Работа с бинарной

кучей

```
int main()
{
    struct heap *h;
    struct heapnode node;

    h = heap_create(100);
    heap_insert(h, 16, "16");
    heap_insert(h, 14, "14");
    heap_insert(h, 10, "10");
    heap_insert(h, 8, "8");
    heap_insert(h, 7, "7");
    heap_insert(h, 9, "9");
    heap_insert(h, 3, "3");
    heap_insert(h, 2, "2");
    heap_insert(h, 4, "4");
    heap_insert(h, 1, "1");

    node = heap_extract_max(h);
    printf("Item: %d\n", node.key);
    return 0;
}
```

Изменение кучи

```
void heapify(int i) {
    int left, right;
    int temp;
    left = 2 * i + 1;
    right = 2 * i + 2;
    if(left < HeapSize) {
        if(h[i] < h[left]) {
            temp = h[i];
            h[i] = h[left];
            h[left] = temp;
            heapify(left);
        }
    }
}
```

Увеличение

ключа

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

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

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

Построение бинарной

кучи

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

Построение бинарной кучи

(v1)

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

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

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

Построение бинарной кучи

(v2)

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

```
function MaxHeapify(A[1:n],
i) left = 2 * i           // left sub heap
   right = 2 * i +       // right sub heap
   1 largest = i
   if left <= n and A[left] > A[i] then
       largest = left
   if right <= n and A[right] > A[largest]
   then
       largest = right
   if largest != i then
       heap_swap(A[i],
       A[largest]) MaxHeapify(A,
       largest)
   end if
end function

function BuildMaxHeap(A[1:n])
h = CreateBinaryHeap(n)           // O(1)
for i = [n / 2] downto 1 do
    MaxHeapify(A, i)             // O(h)
end function
```

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

Использование двоичной кучи

Очередь с приоритетом (priority queue)

- **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** – объединяет две очереди в одну

Значение (value)	Приоритет (priority)
Слон	3
Кит	1
Лев	15

Сравнение оценки алгоритмов

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

Операция	Binary heap	Binomial heap	Fibonacci heap	Pairing heap	Brodal heap
FindMin	$\Theta(1)$	$O(\log n)$	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
DeleteMin	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)^*$	$O(\log n)^*$	$O(\log n)$
Insert	$\Theta(\log n)$	$O(\log n)$	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
DecreaseKey	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)^*$	$O(\log n)^*$	$\Theta(1)$
Merge/Union	$\Theta(n)$	$\Omega(\log n)$	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$

Алгоритм Дейкстры

Обозначим через n количество вершин, а через m – количество рёбер в графе G .

В обычном простейшем случае получаем $O(n^2)$.

С помощью двоичной кучи сложность алгоритма получается:

$O(\log n * (n + m))$.

Так как время удаления вершины из кучи станет $\log n$ при том, что время модификации тоже возрастёт до $\log n$.

Так как цикл выполняется порядка n раз, а количество релаксаций (смен меток) не больше m , $O(\log n * (n + m))$.

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 $O(n \log n)$ в худшем случае

Как ?

```
function HeapSort(v[1:n])
```

```
  h = CreateBinaryHeap(n)
```

```
  for i = 1 to n do
```

```
    HeapInsert(h, v[i], v[i])
```


```
  end for
```

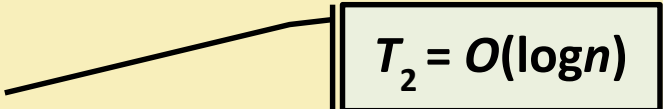
```
  for i = 1 to n do
```


```
    v[i] = HeapRemoveMax(h)
```

```
  end for
```

```
end function
```


$$T_1 = O(1)$$


$$T_2 = O(\log n)$$


$$T_3 = O(\log n)$$

Сортировка на базе бинарной

кучи

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

Как ?

```
function HeapSort(v[1:n])
```

```
  h = CreateBinaryHeap(n)
```

```
  for i = 1 to n do
```

```
    HeapInsert(h, v[i], v[i])
```

```
  end for
```

```
  for i = 1 to n do
```

```
    v[i] = HeapRemoveMax(h)
```

```
  end for
```

```
end function
```

$$T_1 = O(1)$$

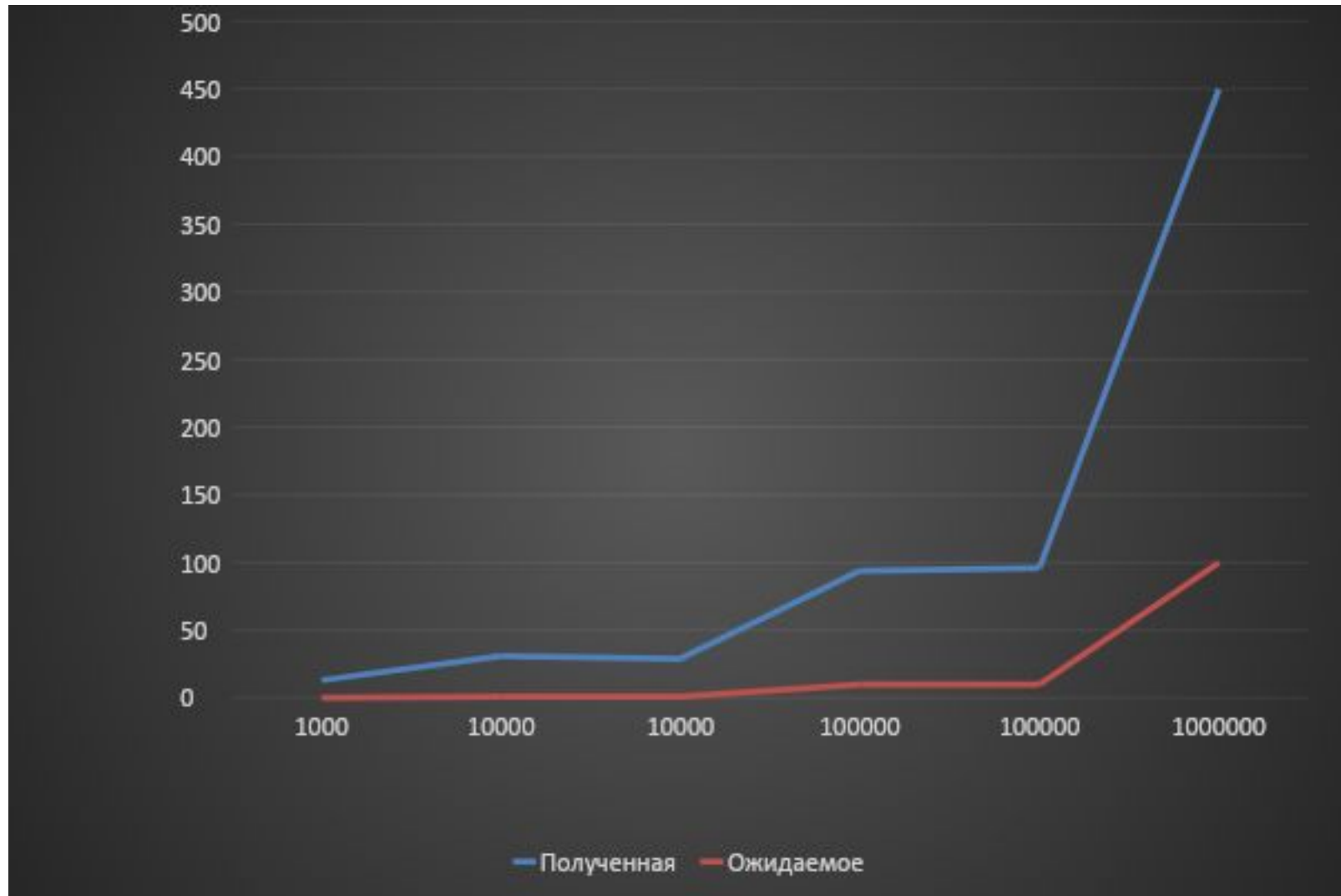
$$T_2 = O(\log n)$$

$$T_3 = O(\log n)$$

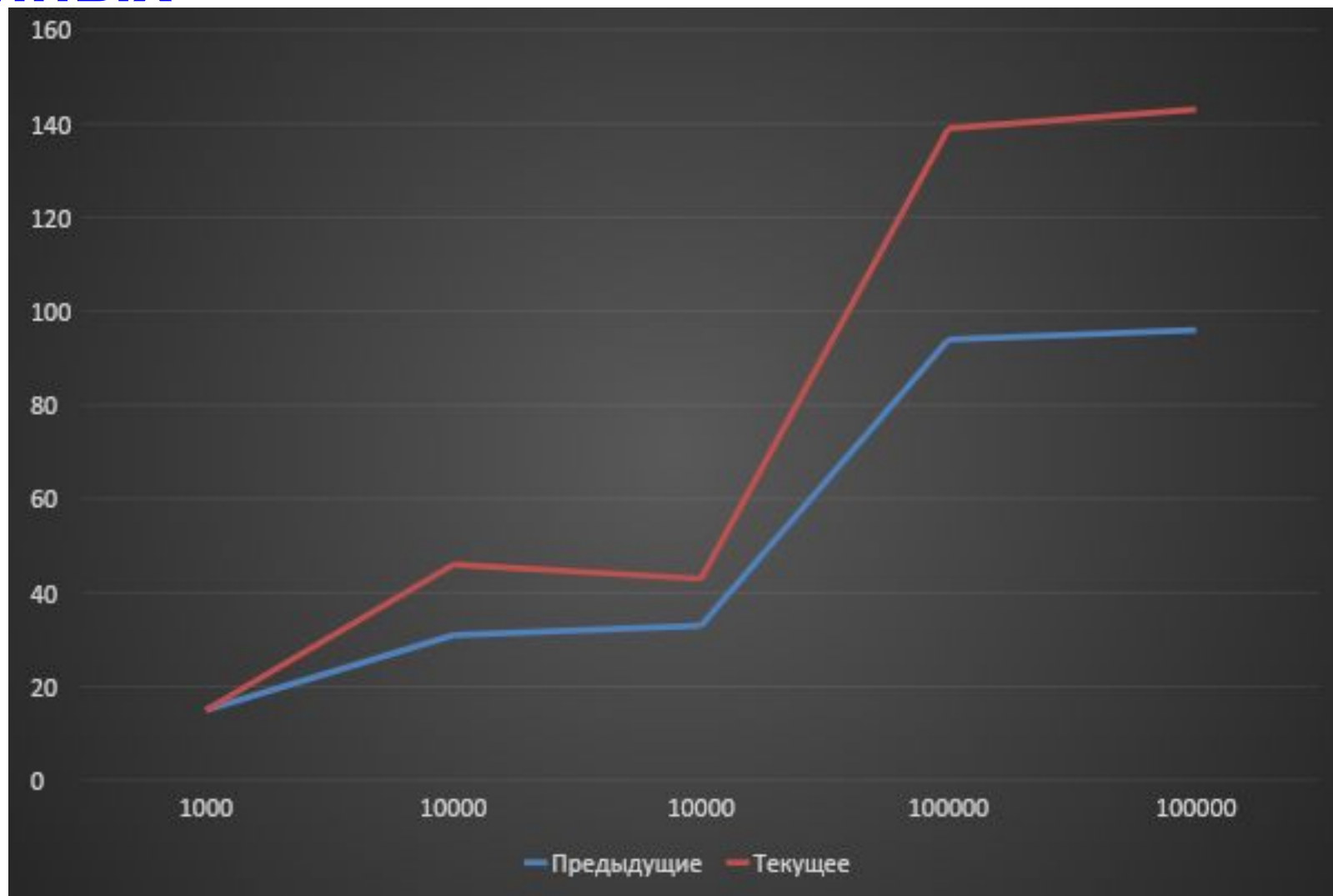
$$T_{\text{HeapSort}} = 1 + n \log n + n \log n = O(n \log n)$$

Оценки работы алгоритма

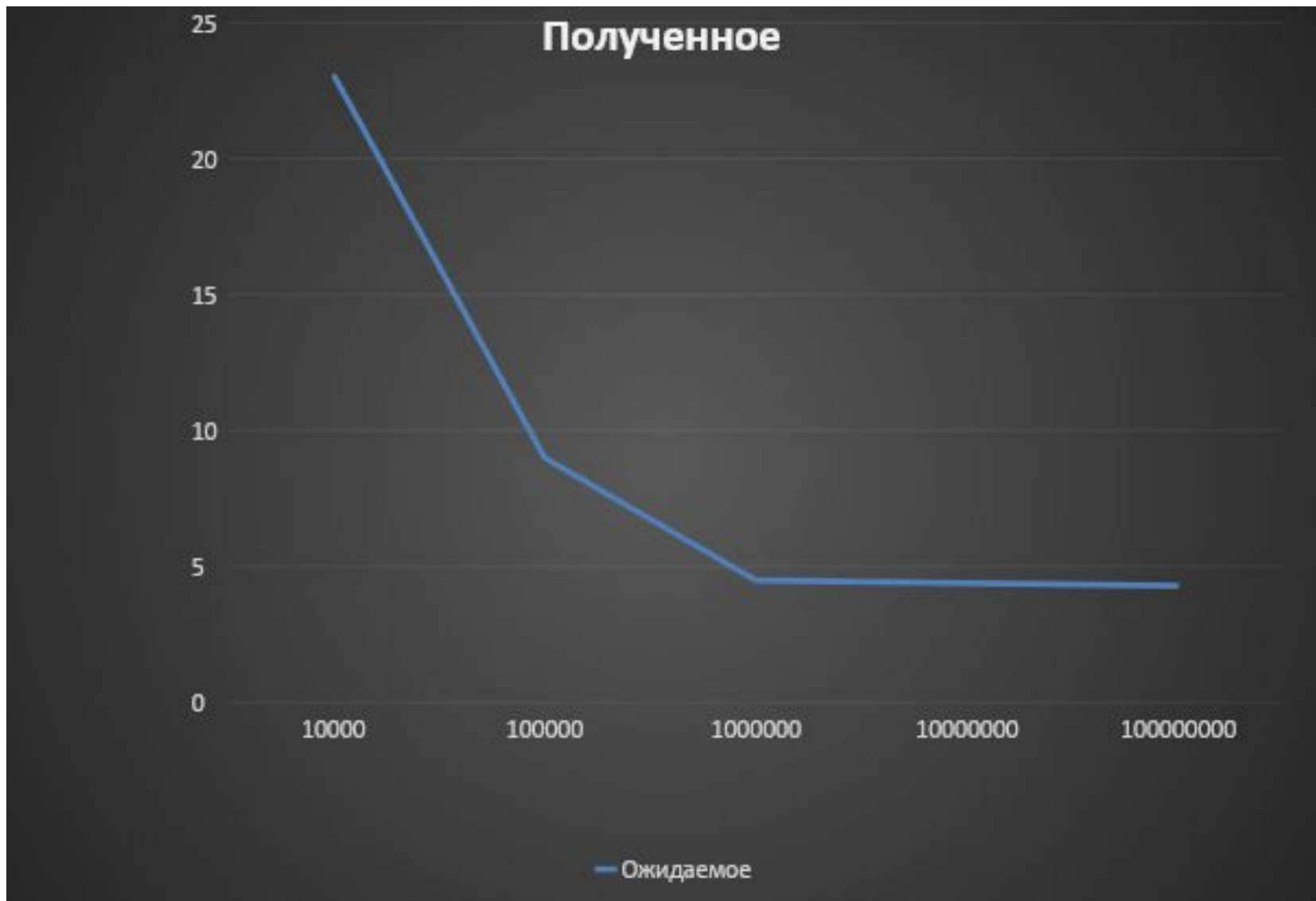
Скорость работы программы



Скорость работы программы с выводом данных

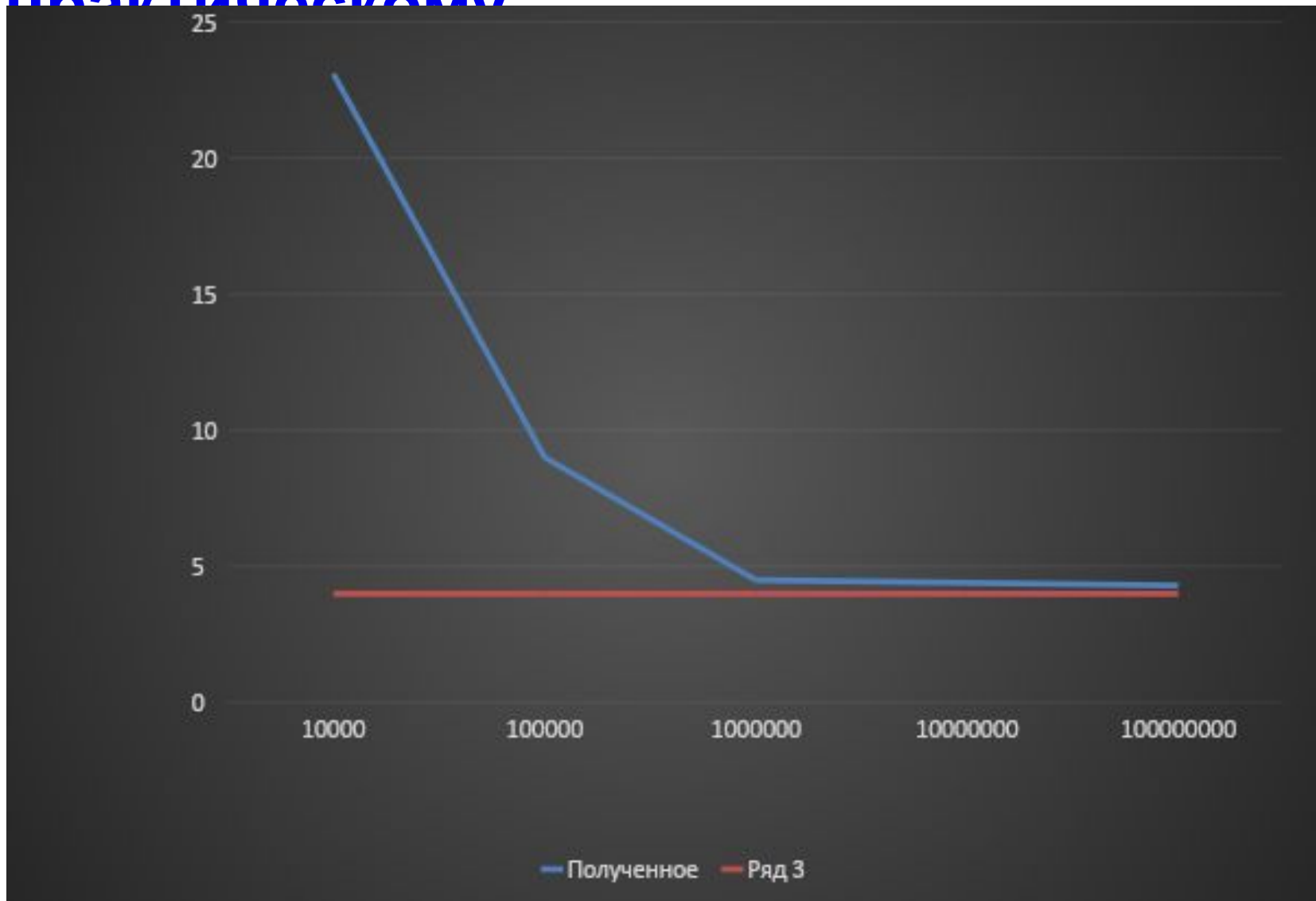


Отношение



Отношение теоретического к

практическому



Спасибо за внимание!