Сортировка пузырьком (bubble sort)

- Алгоритм состоит из повторяющихся проходов по сортируемому массиву. За каждый проход элементы последовательно сравниваются попарно и, если порядок в паре неверный, выполняется обмен элементов. Проходы по массиву повторяются N-1 раз или до тех пор, пока на очередном проходе не окажется, что обмены больше не нужны, что означает — массив отсортирован. При каждом проходе алгоритма по внутреннему циклу, очередной наибольший элемент массива ставится на своё место в конце массива рядом с предыдущим «наибольшим элементом», а наименьший элемент перемещается на одну позицию к началу массива («всплывает» до нужной позиции как пузырёк в воде, отсюда и название алгоритма).

Сортировка пузырьком (bubble sort)

Сортировка пузырьком

- Асимптотическая сложность - $O(n^2)$
- Время работы в худшем и среднем случае - $O(n^2)$
- Дополнительная память - $O(1)$

```
1. function bubbleSort(a):
2.   for i = 0 to n - 2
3.     for j = 0 to n – i - 2
4.       if a[j] > a[j + 1]
5.         swap(a[j], a[j + 1])
```

Пример

i	*j*		*1*	*2*	*3*	*4*	*5*	*6*
1	1		4	5	9	1	3	6
	2		4	5	9	1	3	6
	3	обмен	4	5	9	1	3	6
	4	обмен	4	5	1	9	3	6
	5	обмен	4	5	1	3	9	6
2	1		4	5	1	3	6	**9**
	2	обмен	4	5	1	3	6	**9**
	3	обмен	4	1	5	3	6	**9**
	4		4	1	3	5	6	**9**
3	1	обмен	4	1	3	5	**6**	**9**
	2	обмен	1	4	3	5	**6**	**9**
	3		1	3	4	5	**6**	**9**
4	1		1	3	4	**5**	**6**	**9**
	2		1	3	4	**5**	**6**	**9**
5	1		1	3	**4**	**5**	**6**	**9**

Оптимизация (условие Айверсона 1)

- Если после выполнения внутреннего цикла не произошло ни одного обмена, то массив уже отсортирован, и продолжать что-то делать бессмысленно.

```
1. function bubbleSort(a):
2.   i = 0
3.   t = true
4.   while t
5.     t = false
6.     for j = 0 to n - i - 2
7.       if a[j] > a[j + 1]
8.         swap(a[j], a[j + 1])
9.         t = true
10.    i = i + 1
```

Анализ алгоритма

```
1. function bubbleSort(a):
2.   i = 0
3.   t = true
4.   while t
5.     t = false
6.     for j = 0 to n - i - 2
7.       if a[j] > a[j + 1]
8.         swap(a[j], a[j + 1])
9.         t = true
10.    i = i + 1
```

Макконнелл Дж. Основы современных алгоритмов = Analysis of Algorithms: An Active Learning Approach / Под ред. С. К. Ландо. — М.: Техносфера, 2004. — С. 72—76. — ISBN 5-94836-005-9.

Анализ алгоритма

```
1. function bubbleSort(a):
2.   i = 0
3.   t = true
4.   while t
5.     t = false
6.     for j = 0 to n - i - 2
7.       if a[j] > a[j + 1]
8.         swap(a[j], a[j + 1])
9.         t = true
10.    i = i + 1
```

Макконнелл Дж. Основы современных алгоритмов = Analysis of Algorithms: An Active Learning Approach / Под ред. С. К. Ландо. — М.: Техносфера, 2004. — С. 72—76. — ISBN 5-94836-005-9.

Анализ алгоритма

```
1. function bubbleSort(a):
2.   i = 0
3.   t = true
4.   while t
5.     t = false
6.     for j = 0 to n - i - 2
7.       if a[j] > a[j + 1]
8.         swap(a[j], a[j + 1])
9.         t = true
10.    i = i + 1
```

Макконнелл Дж. Основы современных алгоритмов = Analysis of Algorithms: An Active Learning Approach / Под ред. С. К. Ландо. — М.: Техносфера, 2004. — С. 72—76. — ISBN 5-94836-005-9.

Оптимизация (условие Айверсона 1)

- Асимптотическая сложность - $O(n^2)$
- Время работы в худшем и среднем случае - $O(n^2)$
- Время работы в лучшем случае - $O(n)$
- Дополнительная память - $O(1)$

```
1. function bubbleSort(a):
2.   i = 0
3.   t = true
4.   while t
5.     t = false
6.     for j = 0 to n - i - 2
7.       if a[j] > a[j + 1]
8.         swap(a[j], a[j + 1])
9.         t = true
10.    i = i + 1
```

Оптимизация (условие Айверсона 2)

- Запоминаем, в какой позиции t был последний обмен на предыдущей итерации внешнего цикла.
- Это – верхняя граница просмотра массива Bound на следующей итерации
- Если t = 0 после выполнения внутреннего цикла, значит, обменов не было, алгоритм заканчивает работу.
- Основная идея – **уменьшаем количество проходов внутреннего цикла**

Быстрая сортировка

- Алгоритм сортировки, разработанный английским информатиком Тони Хоаром во время его работы в МГУ в 1960 году.
- Один из самых быстрых известных универсальных алгоритмов сортировки массивов.
- Алгоритм основан на принципе «разделяй и властвуй».

Быстрая сортировка. Алгоритм

1) В исходном несортированном массиве некоторым образом выбирается разделительный элемент x (барьерный элемент, опорный элемент, pivot).

2) Массив разбивается на две части. Элементы массива переставляются таким образом, чтобы

- в левой части массива оказались элементы <=x,
- в правой – элементы массива, большие или равные >=x.

В итоге все элементы левой части меньше любого элементов правой части, за исключением элементов, равных барьерному (они могут быть как слева, так и справа).

3) Рекурсивно обрабатываются левый и правый подмассивы

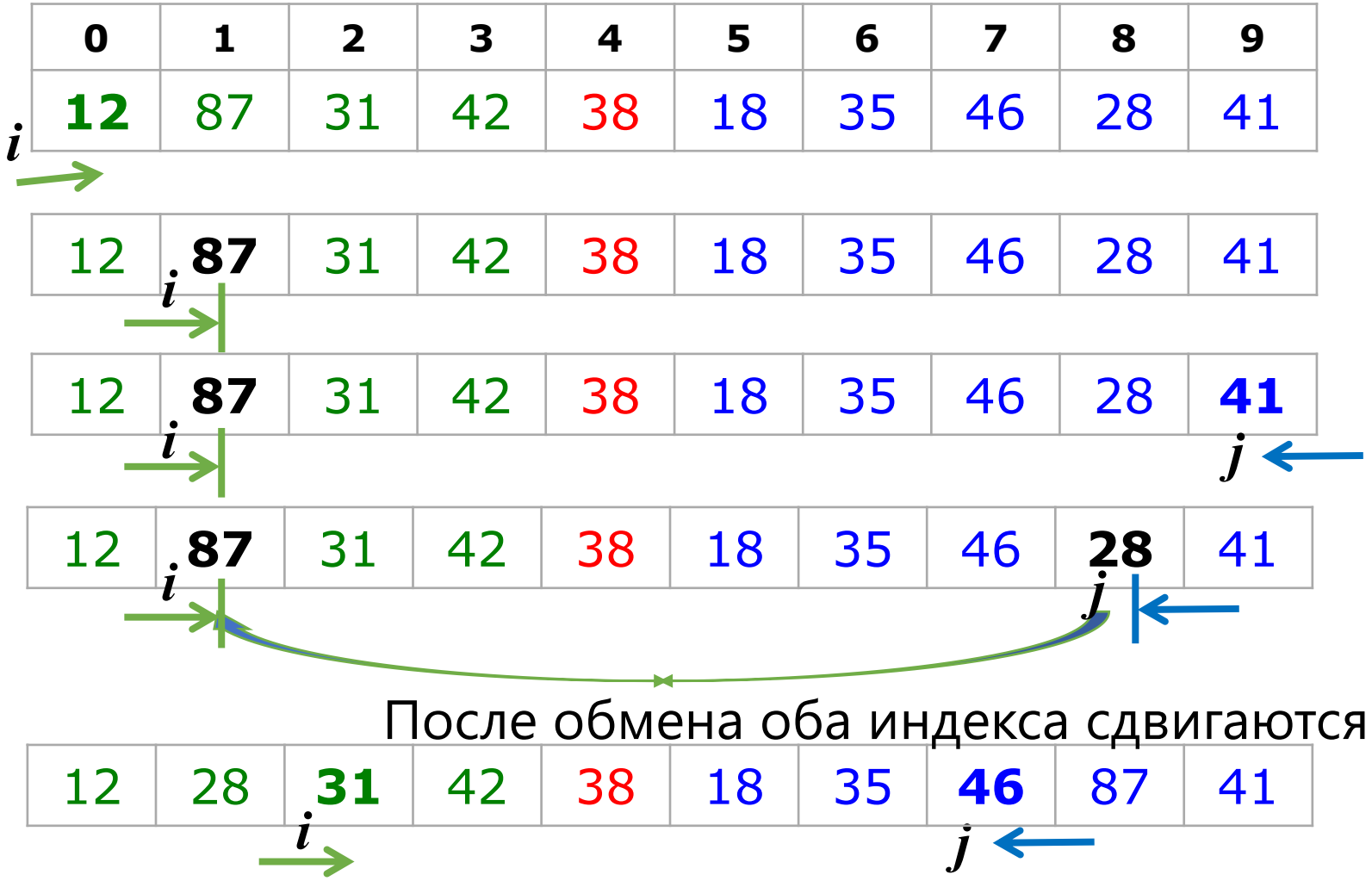
Разделительный элемент

1. Первый (последний) элемент рассматриваемой части массива (разбиение Ломуто).
2. Второй (предпоследний) элемент рассматриваемой части массива.
3. Элемент, находящийся в середине рассматриваемой части массива.
4. Среднее арифметическое всех элементов рассматриваемой части массива.
5. Среднее арифметическое из трех элементов в начале, в конце и в середине рассматриваемой части массива.
6. Медиана трех элементов в начале, в конце и в середине рассматриваемой части массива.
7. Медиана подмассива.
8. Случайным образом.

Разбиение Хоара

```
algorithm quicksort(A, lo, hi) is
   if lo < hi then
       p:= partition(A, lo, hi)
       quicksort(A, lo, p)
       quicksort(A, p + 1, hi)
algorithm partition(A, low, high) is
   pivot:= A[(low + high) / 2]
   i:= low
   j:= high
   loop forever

       while A[i] < pivot
              i:= i + 1
       while A[j] > pivot
              j:= j - 1
       if i >= j then
           return j
       swap A[i++] with A[j--]
```

Пример алгоритма с разбиением Хоара



- $k = (0 + 9) / 2 = 9 / 2 = 4$, X=38

Разделительный элемент выделен красным цветом,
Элементы, подлежащие **обмену**, – черным,
Левая часть массива – зеленым,
Правая – синим

Разбиение Ломуто

```
algorithm quicksort(A, lo, hi) is
   if lo < hi then
       p:= partition(A, lo, hi)
       quicksort(A, lo, p)
       quicksort(A, p + 1, hi)
```

```
algorithm partition(A, low, high) is
    pivot := A[high]
    i := low
    for j := low to high - 1 do
        if A[j] ≤ pivot then
            swap A[i] with A[j]
            i := i + 1
    swap A[i] with A[high]
    return i
```

Пример разбиения Ломуто

j	0	1	2	3	4	5	6	7	8	9
0	12	87	31	42	38	18	35	46	28	**41**
1		87	31							
2		31	87							
3			87		38					
4			38	42	87	18				
5				18	87	42	35			
6					35	42	87			
7						42				
8	12	31	38	18	35	28	87	46	42	**41**
							41			87
	12	31	38	18	35	28	**41**	46	42	87

```
algorithm partition(A, low, high) is
    pivot := A[high]
    i := low
    for j := low to high – 1 do
        if A[j] ≤ pivot then
            swap A[i] with A[j]
            i := i + 1
    swap A[i] with A[high]
    return i
```

Разбиение для повторяющихся элементов

```
algorithm quicksort(A, low, high) is
    if low < high then
        p := pivot(A, low, high)
        left, right := partition(A, p, low, high)
        quicksort(A, low, left)
        quicksort(A, right, high)
```

Достоинства и недостатки

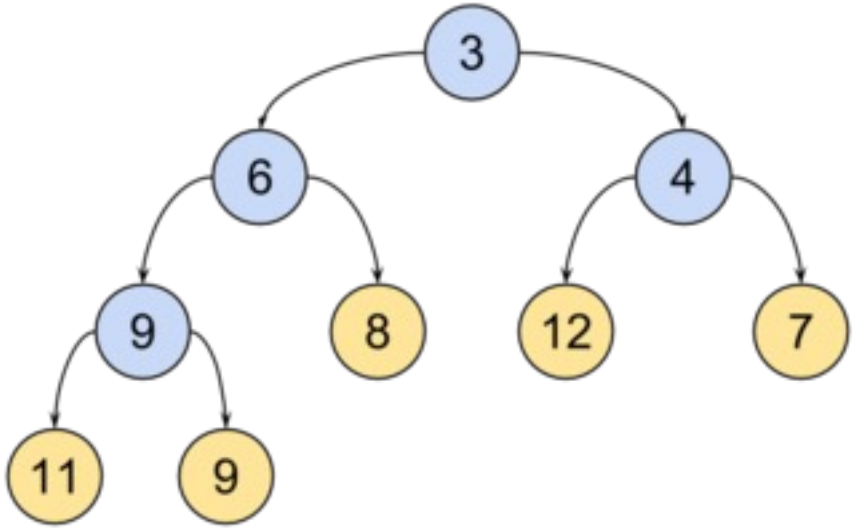
- Прост в реализации
- Возможно распараллеливание
- Один из самых быстрых на практике
- Асимптотическая сложность - $O(n \log n)$ в среднем случае
- Деградация до квадратичной сложности в худшем случае
- Рекурсивная реализация может привести к переполнению стека
- Не устойчивая

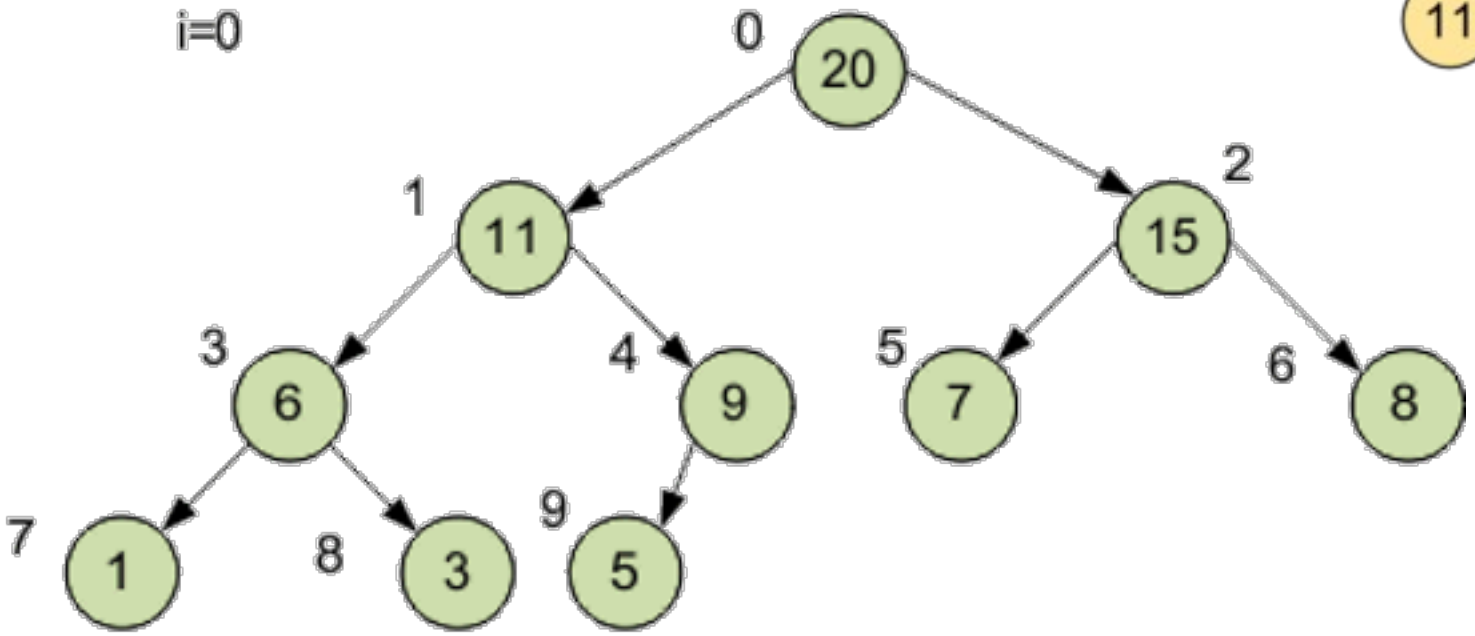
Двоичная куча

- Двоичная куча представляет собой полное бинарное дерево, для которого выполняется основное свойство кучи: приоритет каждой вершины больше приоритетов её потомков.
- В простейшем случае приоритет каждой вершины можно считать равным её значению. В таком случае структура называется ***max-куча***, поскольку корень поддерева является максимумом из значений элементов поддерева.
- В качестве альтернативы, если сравнение перевернуть, то наименьший элемент будет всегда корневым узлом, такие кучи называют ***min-кучами***.

Двоичная куча

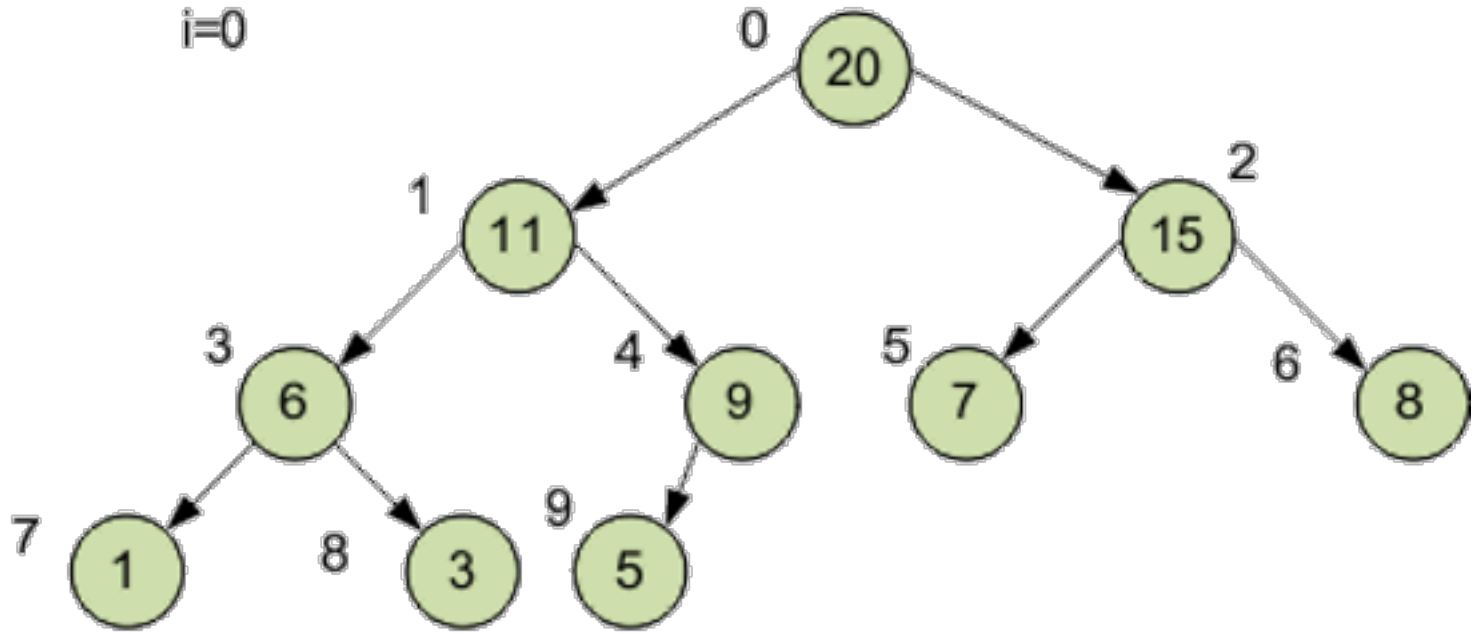
- **a[i] >= a[2i+1];**
- **a[i] >= a[2i+2].**



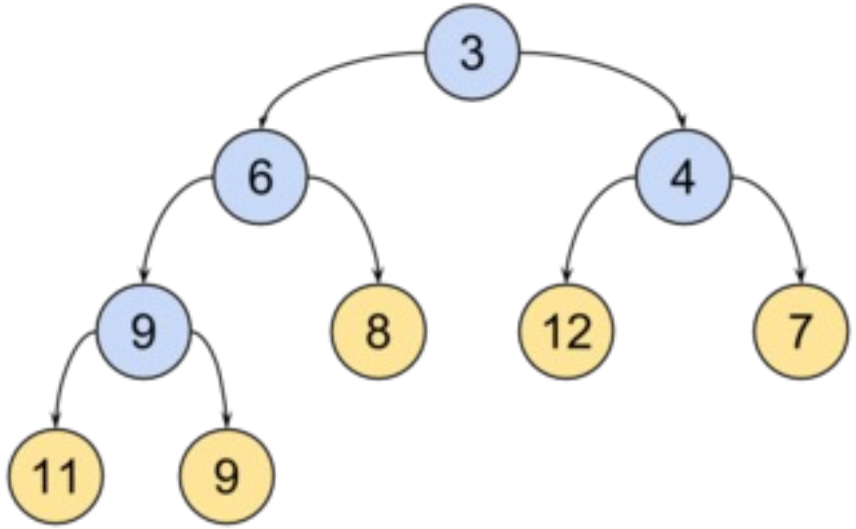


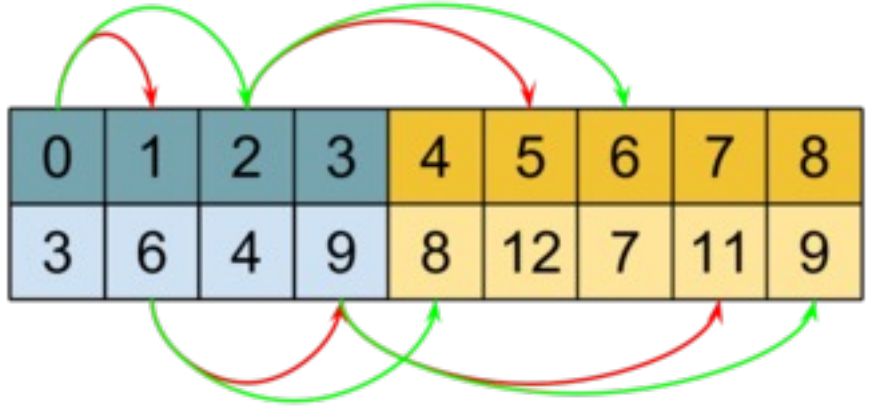
Двоичная куча

- левый потомок вершины с индексом i имеет индекс 2*i+1,
- правый потомок вершины с индексом i имеет индекс 2*i+2,
- Высота двоичной кучи равна высоте дерева, то есть $\mathbf{\log_2 (N+1)\uparrow}$,



Хранение кучи в массиве



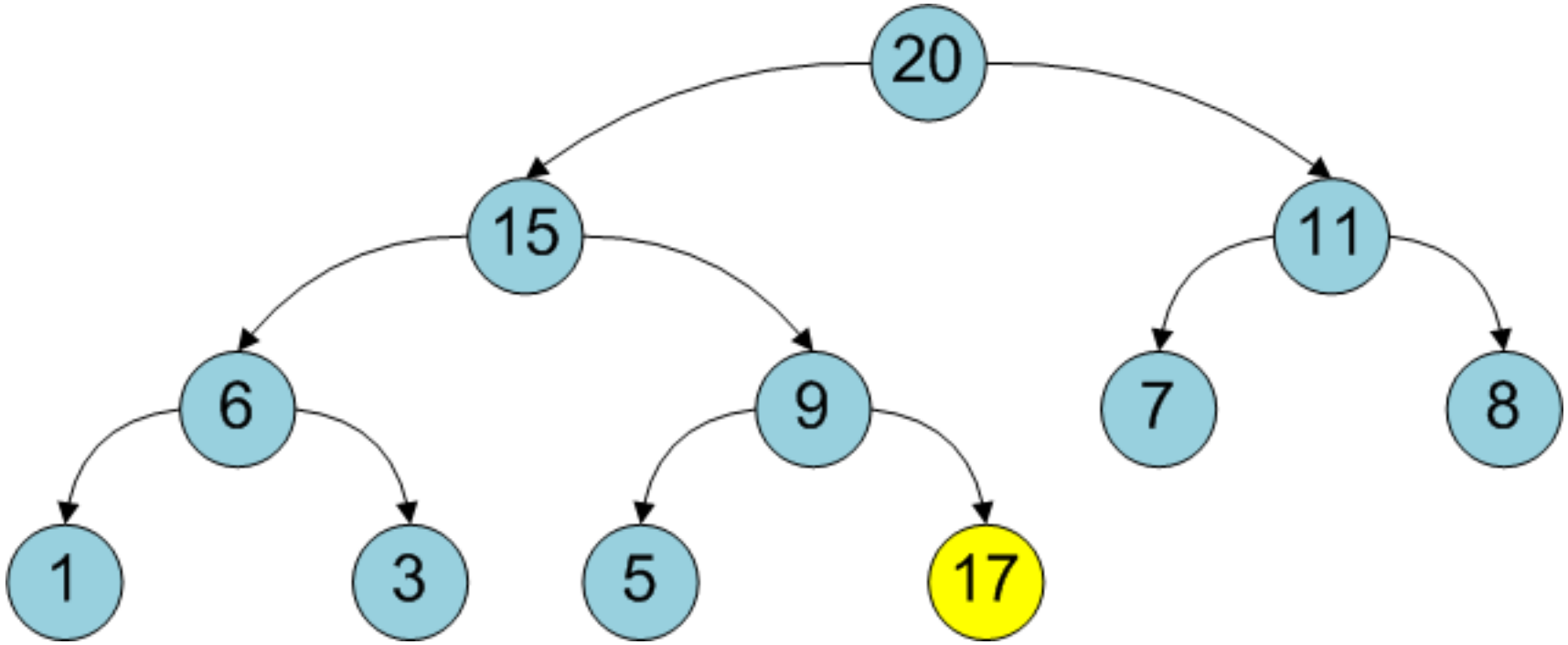


Реализация BinaryHeap

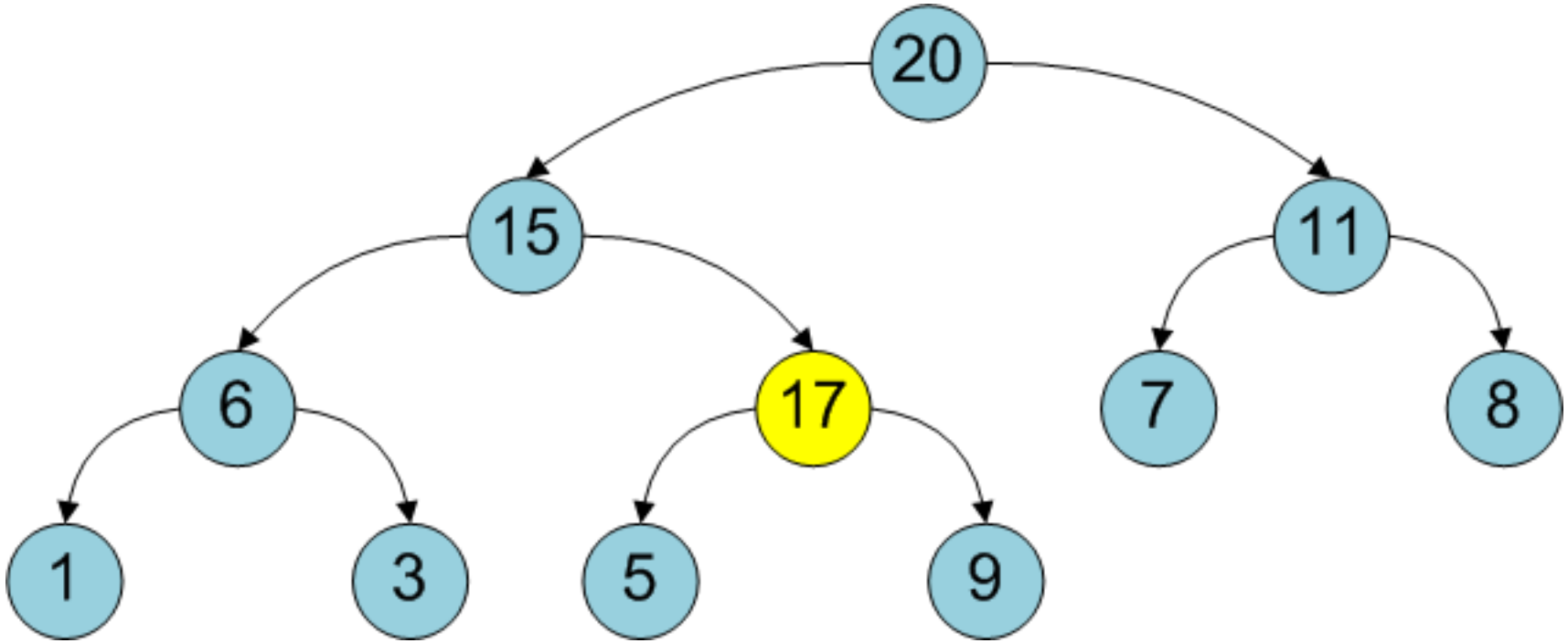
```
public class BinaryHeap
{
    private List<int> list;

    public int heapSize
    {
        get
        {
            return this.list.Count();
        }
    }
}
```

https://habr.com/ru/post/112222/

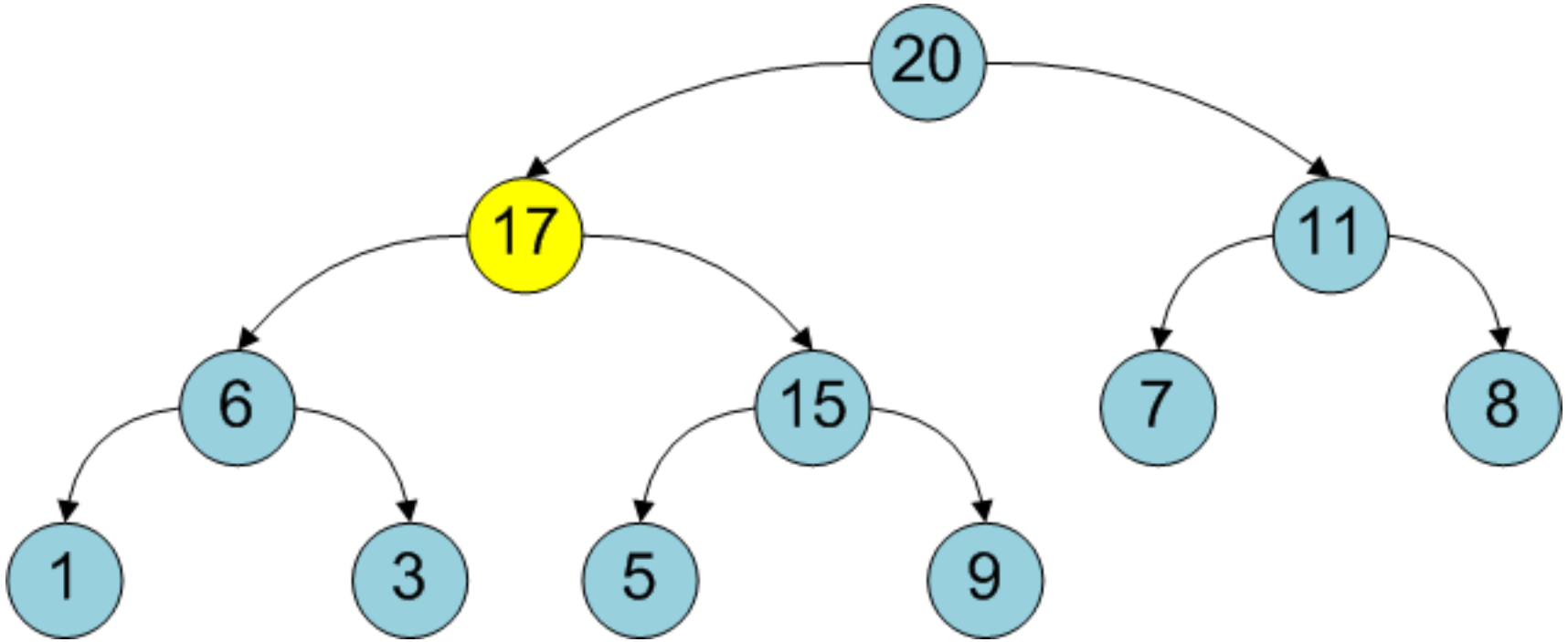
Добавление элемента



Добавление элемента



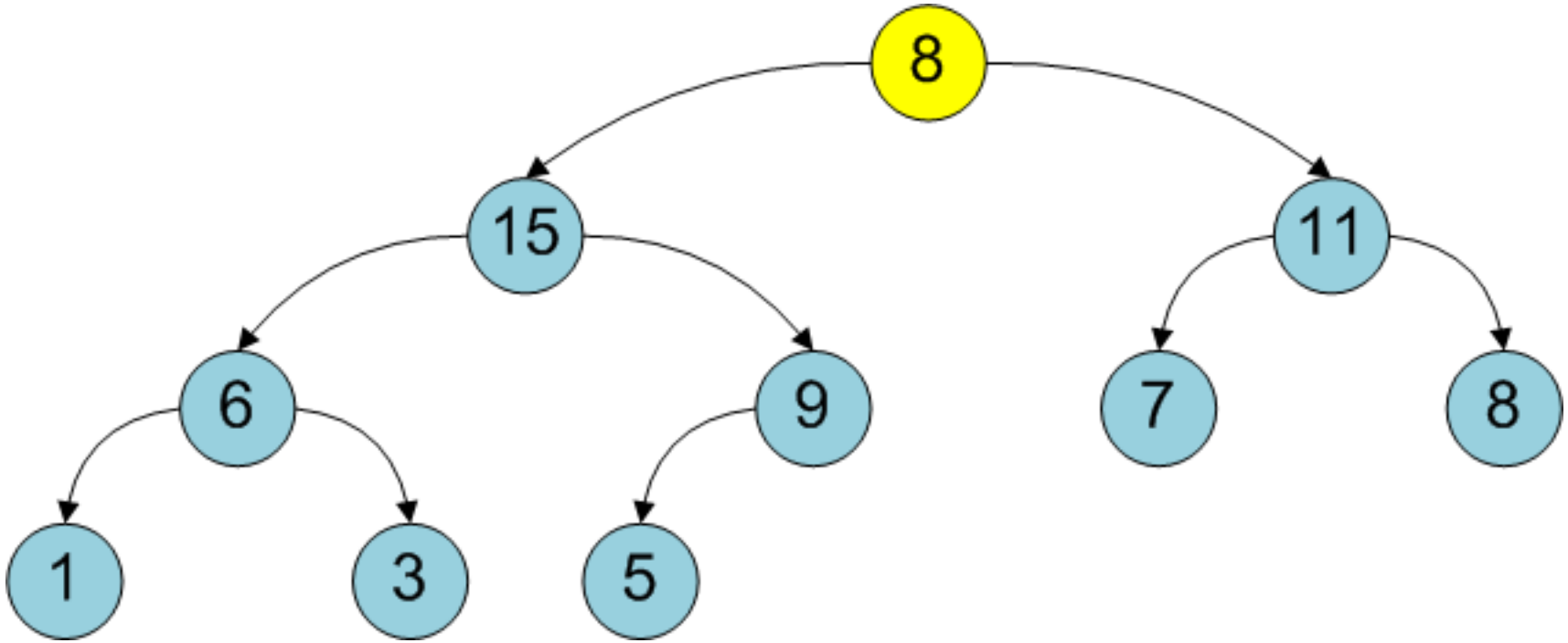
Добавление элемента



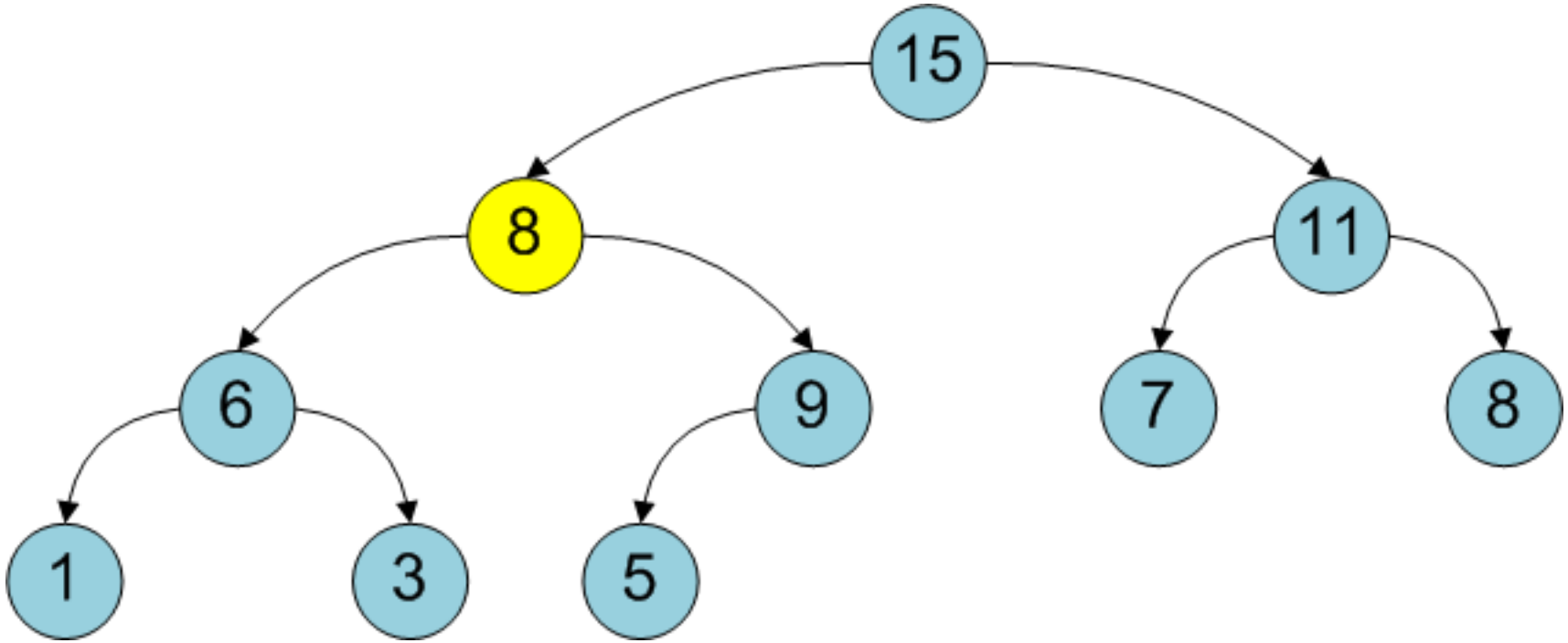
Добавление элемента

```
public void add(int value)
{
    list.Add(value);
    int i = heapSize - 1;
    int parent = (i - 1) / 2;

    while (i > 0 && list[parent] < list[i])
    {
        int temp = list[i];
        list[i] = list[parent];
        list[parent] = temp;

        i = parent;
        parent = (i - 1) / 2;
    }
}
```

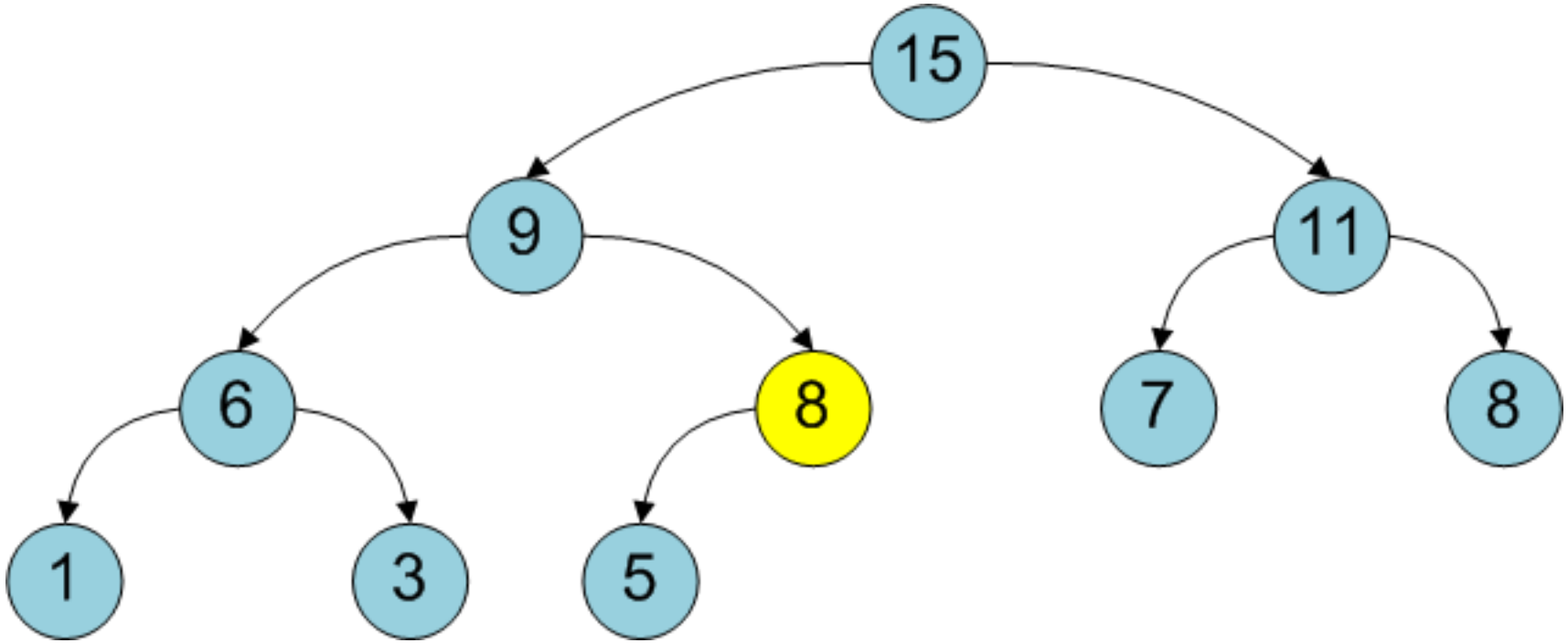
Упорядочение двоичной кучи



Упорядочение двоичной кучи



Упорядочение двоичной кучи



```
public void heapify(int i)
{
    int leftChild;
    int rightChild;
    int largestChild;

    for (; ; )
    {
        leftChild = 2 * i + 1;
        rightChild = 2 * i + 2;
        largestChild = i;

        if (leftChild < heapSize && list[leftChild] > list[largestChild])
        {
            largestChild = leftChild;
        }
        if (rightChild < heapSize && list[rightChild] > list[largestChild])
        {
            largestChild = rightChild;
        }

        if (largestChild == i)
        {
            break;
        }

        int temp = list[i];
        list[i] = list[largestChild];
        list[largestChild] = temp;
        i = largestChild;
    }
}
```

Построение двоичной кучи

```
public void buildHeap(int[] sourceArray)
{
    list = sourceArray.ToList();
    for (int i = heapSize / 2; i >= 0; i--)
    {
        heapify(i);
    }
}
```

Извлечение (удаление) максимального элемента

```
public int getMax()
{
    int result = list[0];
    list[0] = list[heapSize - 1];
    list.RemoveAt(heapSize - 1);
    return result;
}
```

Сортировка с помощью двоичной кучи

- Пирамидальная сортировка (англ. Heapsort, «Сортировка кучей») — алгоритм сортировки, работающий в худшем, в среднем и в лучшем случае (то есть гарантированно) за O(n log n) операций при сортировке n элементов.
- Количество применяемой дополнительной памяти не зависит от размера массива (то есть, O(1)).
- Может рассматриваться как усовершенствованная сортировка пузырьком.

Пирамидальная сортировка

- ***1 этап*** Построение пирамиды.
- ***2 этап*** Сортировка на построенной пирамиде. Берем последний элемент массива в качестве текущего. Меняем верхний (наименьший) элемент массива и текущий местами. Текущий элемент (он теперь верхний) просеиваем сквозь n-1 элементную пирамиду. Затем берем предпоследний элемент и т.д.

Пирамидальная сортировка. Алгоритм

```
public void heapSort(int[] array)
{
    buildHeap(array);
    for (int i = array.Length - 1; i >= 0; i--)
    {
        array[i] = getMax();
        heapify(0);
    }
}
```

Достоинства и недостатки

- Имеет доказанную оценку худшего случая O(n logn).
- Сортирует на месте, то есть требует всего $O(1)$ дополнительной памяти
- Неустойчив
- На почти отсортированных массивах работает столь же долго, как и на хаотических данных.
- На одном шаге выборку приходится делать хаотично по всей длине массива — поэтому алгоритм плохо сочетается с кэшированием и подкачкой памяти.
- Методу требуется «мгновенный» прямой доступ; не работает на связанных списках и других структурах памяти последовательного доступа.
- Не распараллеливается.