МГТУ им. Н.Э. Баумана

ТЕХНОПАРК

Mail.Ru Group

Лекция 3. Двоичная куча. Сортировки (часть 1).

Алгоритмы и структуры данных

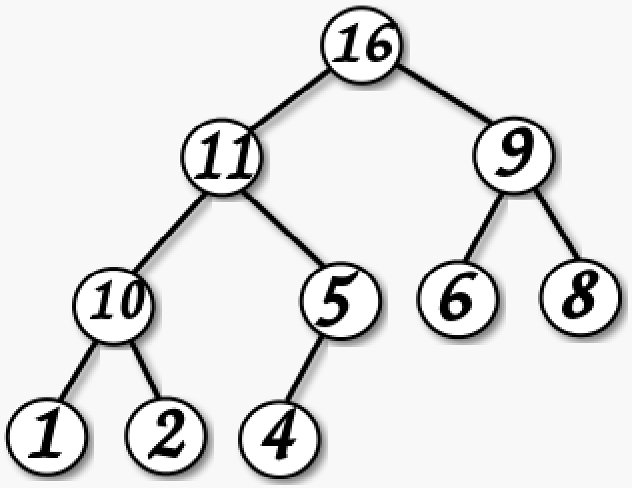
Мацкевич С.Е.

СД «Двоичная куча»

__Определение.__ **Двоичная куча, пирамида**, или **сортирующее дерево** — такое почти полное двоичное дерево, для которого выполнены три условия:

1) Значение в любой вершине не меньше, чем значения её потомков.
2) Глубина листьев (расстояние до корня) отличается не более чем на один.
3) Последний слой заполняется слева направо.

Глубина кучи = $O(\log n)$, где n – количество элементов.



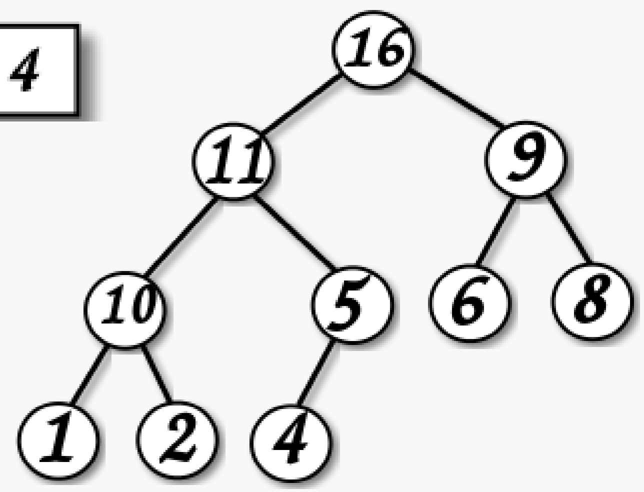
СД «Двоичная куча»

Удобный способ хранения для двоичной кучи — массив.

Последовательно храним все элементы кучи «по слоям».

Корень — первый элемент массива, второй и третий элемент — дочерние элементы и так далее.

16	11	9	10	5	6	8	1	2	4



СД «Двоичная куча»

Такой способ хранения элементов в массиве позволяет быстро получать дочерние и родительские элементы.

1) Если индексация элементов массива начинается с 1.
   - $A[1]$ – элемент в корне,
   - потомки элемента $A[i]$ – элементы $A[2i]$ и $A[2i + 1]$.
   - предок элемента $A[i]$ – элемент $A[i/2]$.
2) Если индексация элементов массива начинается с 0.
   - $A[0]$ – элемент в корне,
   - потомки элемента $A[i]$ – элементы $A[2i + 1]$ и $A[2i + 2]$.
   - предок элемента $A[i]$ – элемент $A[(i – 1)/2]$.

СД «Двоичная куча»

Восстановление свойств кучи

Если в куче изменяется один из элементов, то она может перестать удовлетворять свойству упорядоченности.

Для восстановления этого свойства служат две процедуры
Sift Up и **Sift Down**.

Sift Down спускает элемент, который меньше дочерних.

Sift Up поднимает элемент, который больше родительского.

СД «Двоичная куча». SiftDown.

Восстановление свойств кучи

Sift Down (также используется название Heapify)

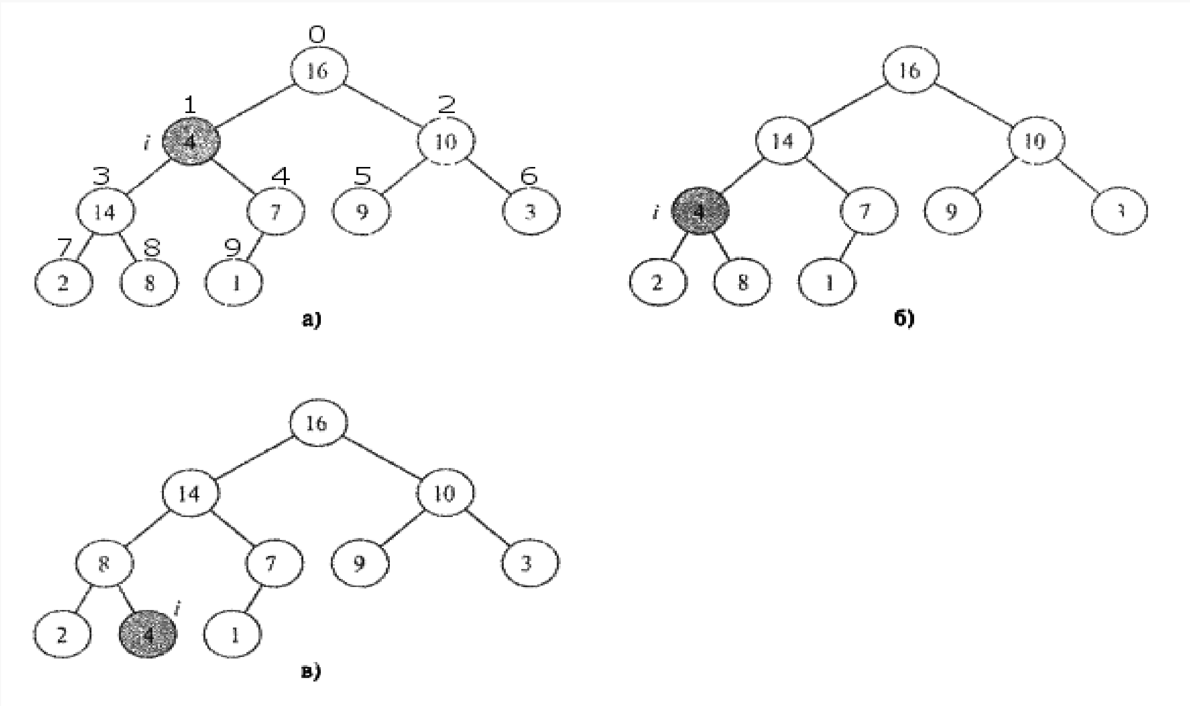
Если i-й элемент больше, чем его сыновья, всё поддерево уже является кучей, и делать ничего не надо. В противном случае меняем местами i-й элемент с наибольшим из его сыновей, после чего выполняем **Sift Down** для этого сына.

Функция выполняется за время $O(\log n)$.

СД «Двоичная куча». SiftDown.

```
// Проталкивание элемента вниз. CArray – целочисленный массив.
void SiftDown( CArray& arr, int i )
{
    int left = 2 * i + 1;
    int right = 2 * i + 2;
    // Ищем большего сына, если такой есть.
    int largest = i;
    if( left < arr.Size() && arr[left] > arr[i] )
        largest = left;
    if( right < arr.Size() && arr[right] > arr[largest] )
        largest = right;
    // Если больший сын есть, то проталкиваем корень в него.
    if( largest != i ) {
        std::swap( arr[i], arr[largest] );
        SiftDown( arr, largest );
    }
}
```

СД «Двоичная куча». SiftDown.



СД «Двоичная куча». Построение кучи.

Задача. Создать кучу из неупорядоченного массива входных данных.

Если выполнить **Sift Down** для всех элементов массива A, начиная с последнего и кончая первым, он станет кучей.

SiftDown(A, i) не делает ничего, если $i \geq n/2$.

Достаточно вызвать SiftDown для всех элементов массива A с $([n/2] - 1)$-го по 1-ый.

Функция выполняется за время $O(n)$.

СД «Двоичная куча». Построение кучи.

```
// Построение кучи.
void BuildHeap( CArray& arr, int i )
{
    for( int i = arr.Size() / 2 - 1; i >= 0; --i ) {
        SiftDown( arr, i );
    }
}
```

СД «Двоичная куча». Построение кучи.

Утверждение. Время работы BuildHeap = O(n).

Доказательство. Время работы SiftDown для работы с узлом, который находится на высоте h (снизу), равно $C \cdot h$.

На уровне h, содержится не более $\lceil n/2^{h+1} \rceil$ узлов.
Общее время работы:

$$T(n) = \sum_{h=0}^{\log n} \left\lceil \frac{n}{2^{h+1}} \right\rceil C \cdot h = O\left(n \sum_{h=0}^{\log n} \frac{h}{2^h} \right).$$

Воспользуемся формулой $\sum_{h=0}^{\infty} \frac{h}{2^h} = \frac{1/2}{(1-1/2)^2} = 2.$

Таким образом, $T(n) = O\left(n \sum_{h=0}^{\infty} \frac{h}{2^h}\right) = O(n).$

СД «Двоичная куча». SiftUp.

Восстанавливает свойство упорядоченности, проталкивая элемент наверх.

Если элемент больше отца, меняет местами его с отцом.

Если после этого отец больше деда, меняет местами отца с дедом, и так далее.

Время работы – $O(\log n)$.

СД «Двоичная куча». SiftUp.

```
// Проталкивание элемента наверх.
void SiftUp( CArray& arr, int index )
{
    while( index > 0 ) {
        int parent = ( index - 1 ) / 2;
        if( arr[index] <= arr[parent] )
            return;
        std::swap( arr[index], arr[parent] );
        index = parent;
    }
}
```

СД «Двоичная куча». Добавление элемента.

1. Добавим элемент в конец кучи.
2. Восстановим свойство упорядоченности, проталкивая элемент наверх с помощью SiftUp.

Время работы – $O(\log n)$, если буфер для кучи позволяет добавить элемент без переаллокации.

```
// Добавление элемента.
void Add( CArray& arr, int element )
{
    arr.Add( element );
    SiftUp( arr, arr.Size() - 1 );
}
```

СД «Двоичная куча». Извлечение максимума.

Максимальный элемент располагается в корне. Для его извлечения:

1. Сохраним значение корневого элемента для возврата.
2. Скопируем последний элемент в корень, удалим последний элемент.
3. Вызовем SiftDown для корня.
4. Возвратим сохраненный корневой элемент.

Время работы – $O(\log n)$.

СД «Двоичная куча». Извлечение максимума.

```
// Извлечение максимального элемента.
int ExtractMax( CArray& arr )
{
    assert( !arr.IsEmpty() );
    // Запоминаем значение корня.
    int result = arr[0];
    // Переносим последний элемент в корень.
    arr[0] = arr.Last();
    arr.DeleteLast();
    // Вызываем SiftDown для корня.
    if( !arr.IsEmpty() ) {
        SiftDown( arr, 0 );
    }
    return result;
}
```

АТД «Очередь с приоритетом»

<u>**Определение.**</u> **Очередь с приоритетом —** абстрактный тип данных, поддерживающий три операции:

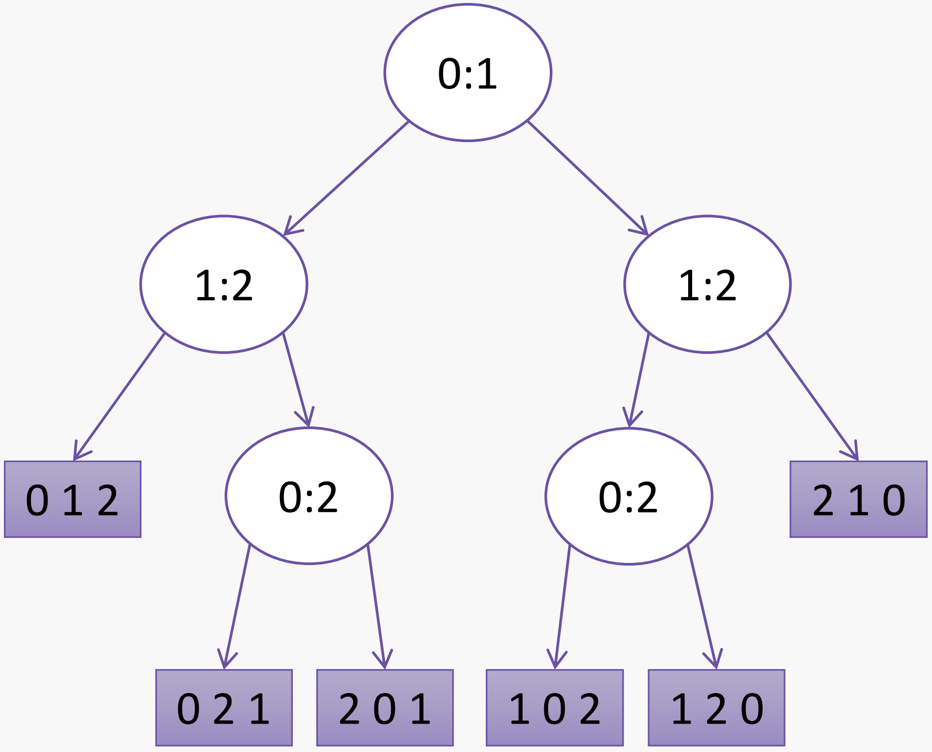
1. **InsertWithPriority —** добавить в очередь элемент с назначенным приоритетом.
2. **GetNext —** извлечь из очереди и вернуть элемент с наивысшим приоритетом. Другие названия: «PopElement», «GetMaximum».
3. **PeekAtNext** (необязательная операция): просмотреть элемент с наивысшим приоритетом без извлечения.

Сортировка

Сортировка – процесс упорядочивания элементов массива.

Пример. Сортировка трех.

```
void Sort3( int* a ) {
    if( a[0] <= a[1] ) {
        if( a[1] <= a[2] ) {
            // 0 1 2
        } else {
            if( a[0] <= a[2] )
                // 0 2 1
            else
                // 2 0 1
        }
    } else {
        if( a[1] <= a[2] ) {
            if( a[0] <= a[2] )
                // 1 0 2
            else
                // 1 2 0
        } else {
            // 2 1 0
        }
    }
}
```



Квадратичные сортировки

- Сортировка выбором,
- Сортировка вставками,
- Пузырьковая сортировка.

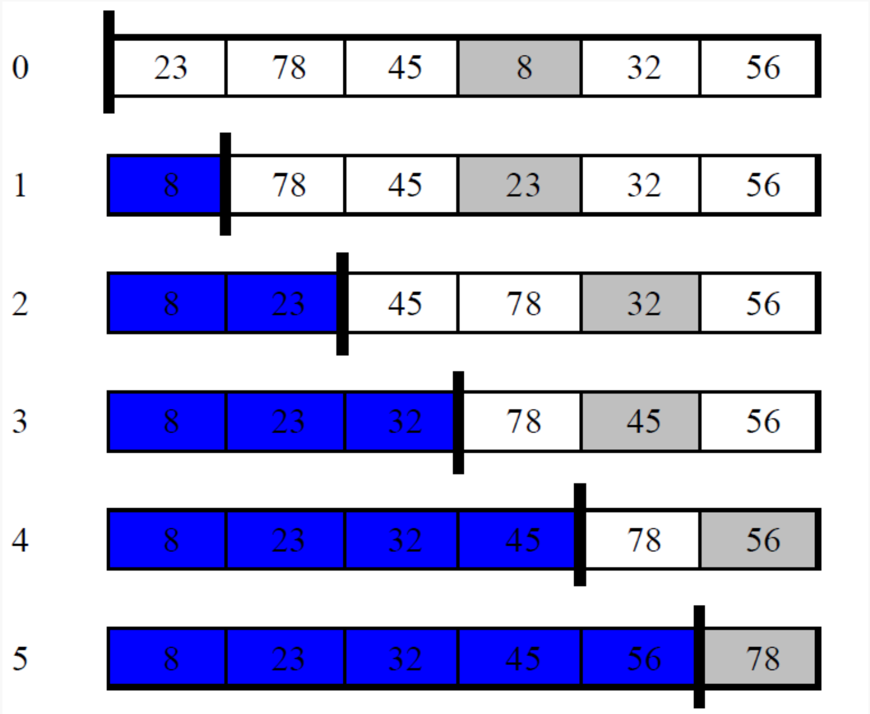
Сортировка выбором

Во время работы алгоритма:

Массив разделен на 2 части:
левая – готова, правая – нет.

На одном шаге:
1) ищем минимум в правой части,
2) меняем его с первым элементом правой части,
3) сдвигаем границу разделения на 1 вправо.

Сортировка выбором



Сортировка выбором

```
void SelectionSort( int* a, int n ) {
    for( int i = 0; i < n - 1; ++i ) {
        // i – индекс начала правой части.
        int minIndex = i;
        for( int j = i + 1; j < n; ++j ) {
            if( a[j] < a[minIndex] )
                minIndex = j;
        }
        swap( a[i], a[minIndex] );
    }
}
```

$\frac{n(n-1)}{2}$ сравнений, $n-1$ перемещений. $T(n) = \Theta(n^2)$.

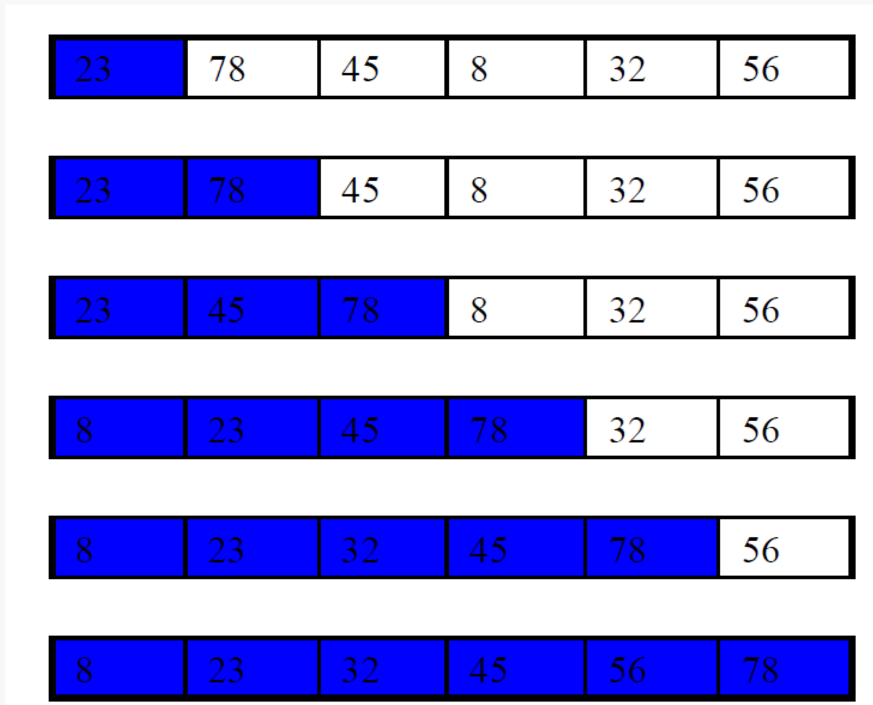
Сортировка вставками

Простой алгоритм, часто применяемый на малых объемах.

Массив разделен на 2 части:
левая – упорядочена, правая – нет.

На одном шаге:
1) берем первый элемент правой части,
2) вставляем его на подходящее место в левой части.

Сортировка вставками



Сортировка вставками

```
void InsertionSort( int* a, int n ) {
    for( int i = 1; i < n; ++i ) {
        int tmp = a[i]; // Запомним, т.к. может перезаписаться.
        int j = i - 1;
        for( ; j >= 0 && tmp < a[j]; --j ) {
            a[j + 1] = a[j];
        }
        a[j + 1] = tmp;
    }
}
```

Сортировка вставками. Анализ.

- Лучший случай: $O(n)$
  - Массив упорядочен по возрастанию.
  - $2 \cdot (n-1)$ копирований,
  - $(n-1)$ сравнений.
- Худший случай: $O(n^2)$
  - Массив упорядочен по убыванию.
  - $2 \cdot (n-1) + \frac{n(n-1)}{2}$ копирований,
  - $\frac{n(n-1)}{2}$ сравнений.
- В среднем: $O(n^2)$

Сортировка вставками. Оптимизации.

- Используем бинарный поиск места вставки в левой части,
- Используем memmove, чтобы эффективно сдвинуть часть элементов левой части вправо на 1 позицию.

$O(n \log n)$ сравнений,

$O(n^2)$ для копирования элементов (с маленькой константой).

Пузырьковая сортировка

Частный случай сортировки выбором.

- Массив разделен на 2 части:
  правая – готова, левая – нет.
- На одном шаге проходим левую часть слева направо:
  1) сравниваем элемент с соседом справа,
  2) меняем с правым соседом, если он меньше.
  3) двигаем границу между частями на 1 влево.

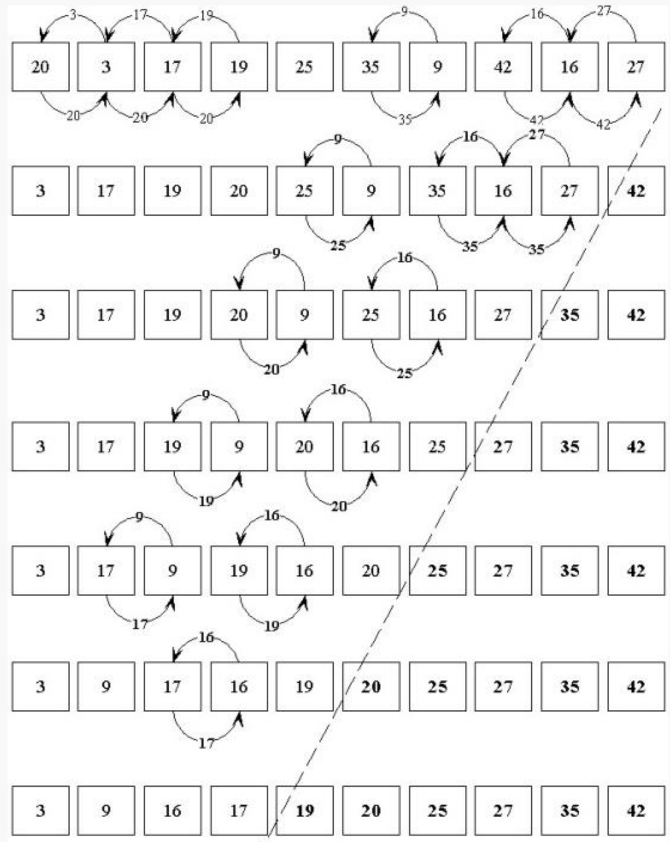
Наибольший элемент всплывет к границе.

Останавливаемся, если не было обменов.

Лучший случай: 1 проход. $T(n) = O(n)$.

Худший случай: убывающий массив. $T(n) = O(n^2)$.

Пузырьковая сортировка



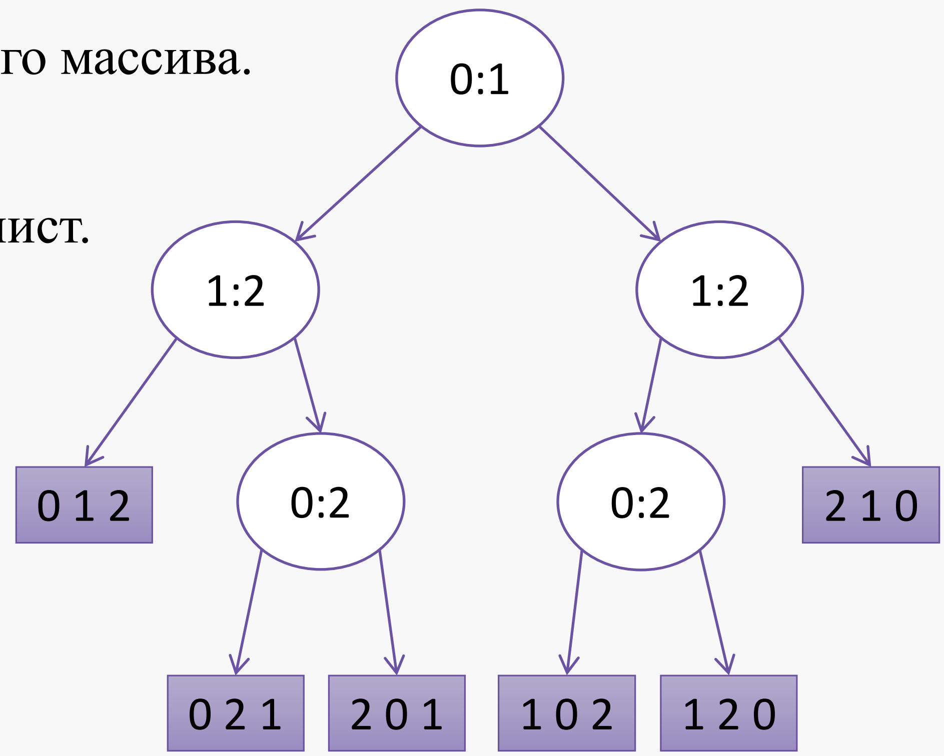
Оценка сложности снизу

В процессе работы алгоритма сравниваются элементы исходного массива.

Ветвление = дерево.

Окончание работы алгоритма – лист.

Лист = перестановка.



Оценка сложности снизу

Утверждение. Время работы любого алгоритма сортировки, использующего сравнение, $\Omega(N \log N)$.

Доказательство.

Всего листьев в дереве решения не меньше N!

Высота дерева не меньше

$$\log(N!) \cong CN \log N \, .$$

Следовательно, существует перестановка, на которой алгоритм делает не менее $CN \log N$ сравнений.

“Хорошие” сортировки

- Пирамидальная сортировка – Heap Sort.
- Сортировка слиянием – Merge Sort.
- Быстрая сортировка = Сортировка Хоара – Quick Sort.

Пирамидальная сортировка

1. Строим кучу на исходном массиве.
2. $N - 1$ раз достаем максимальный элемент, кладем его на освободившееся место в правой части.

Пирамидальная сортировка

МГТУ им. Н.Э. Баумана ТЕХНОПАРК Mail.Ru Group

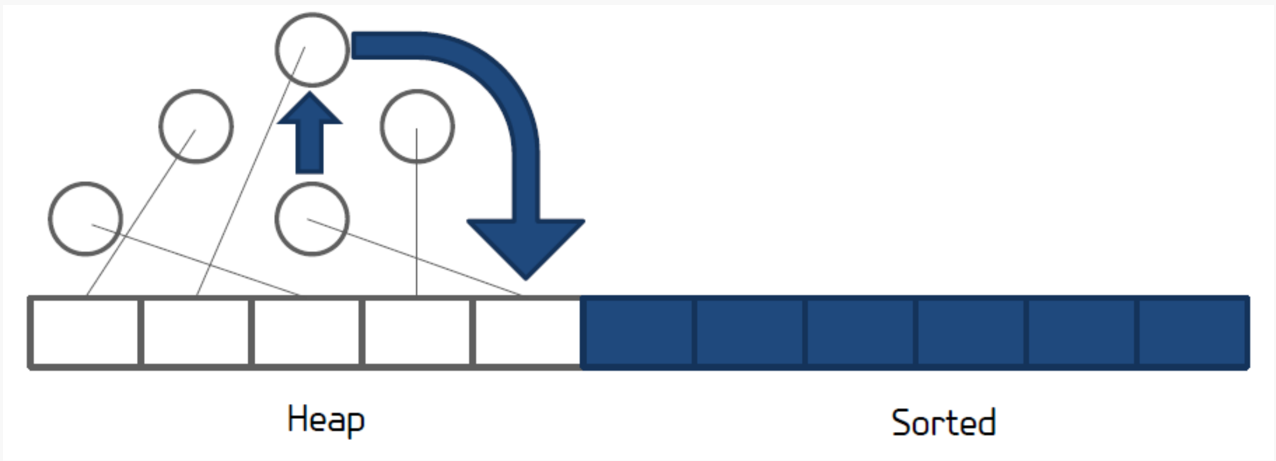
```
void HeapSort( int* a, int n ) {
    int heapSize = n;
    BuildHeap( a, heapSize );
    while( heapSize > 1 ) {
        // Немного переписанный ExtractMax.
        swap( a[0], a[heapSize - 1] );
        --heapSize;
        SiftDown( a, heapSize, 0 );
    }
}
```

$T(n) = O(n \log n)$.

Пирамидальная сортировка

Аналогия с сортировкой выбором:

Берем максимум из левой части, кладем в конец левой части.



Сортировка слиянием

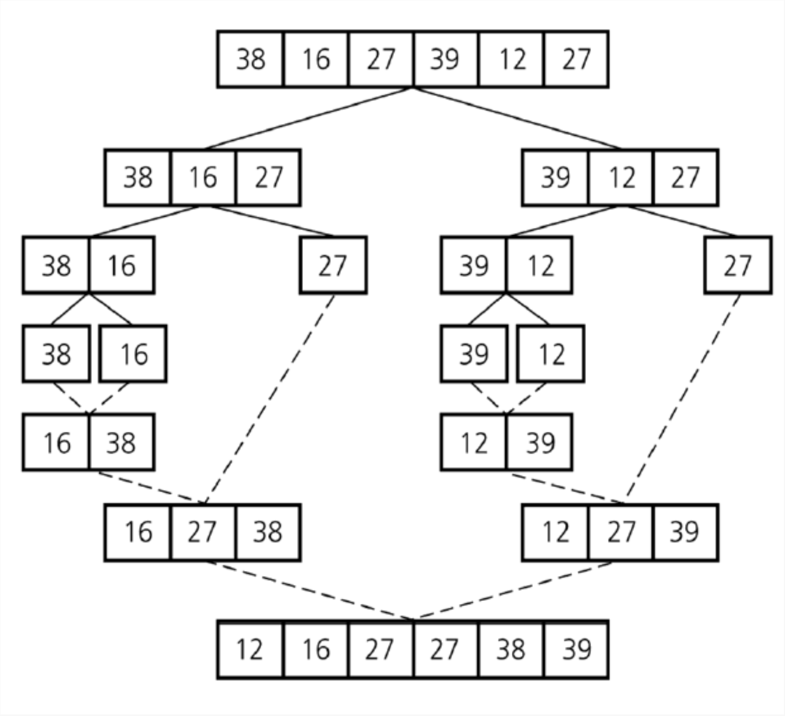
Алгоритм:

1. Разбить массив на два.
2. Отсортировать каждый (рекурсивно).
3. Слить отсортированные в один.

Вариант без рекурсии:

1. Разбить на 2^k подмассива, $2^k < n$.
2. Отсортировать каждый.
3. – Слить 1 и 2, 3 и 4, 5 и 6,…, $2^k - 1$ и 2^k,
   – Слить 12 и 34, 56 и 78,...,
   …
   – Слить $123 \ldots 2^{k-1}$ и $2^{k-1} + 1 \ldots 2^k$.

Сортировка слиянием



Слияние двух отсортированных массивов

Слияние двух отсортированных массивов:

- Выберем массив, крайний элемент которого меньше,
- Извлечем этот элемент в массив-результат,
- Продолжим, пока один из массивов не опустеет,
- Копируем остаток второго массива в конец массива-результата.

5	7	15	20

9	30	45	90

5	7	9					

Слияние двух отсортированных массивов

```
void Merge( const int* a, int aLen, const int* b, int bLen, int* c ) {
    int i = 0, j = 0;
    while( ; i < aLen && j < bLen; ) {
        if( a[i] <= b[j] ) {
            c[i + j] = a[i];
            ++i;
        } else {
            c[i + j] = b[j];
            ++j;
        }
    }
    // Обработаем остаток.
    if( i == aLen ) {
        for( ; j < bLen; ++j )
            c[i + j] = b[j];
    } else {
        for( ; i < aLen; ++i )
            c[i + j] = a[j];
    }
}
```

Слияние двух отсортированных массивов

- Сложность: $T(n, m) = O(n + m)$.
- Количество сравнений:
  - В лучшем случае $\min(n, m)$.
  - В худшем случае $n + m - 1$.

Сортировка слиянием

```
void MergeSort( int* a, int aLen ) {
    if( aLen <= 1 ) {
        return;
    }
    int firstLen = aLen / 2;
    int secondLen = aLen - firstLen;
    MergeSort( a, firstLen );
    MergeSort( a + firstLen, secondLen );
    int* c = new int[aLen];
    Merge( a, firstLen, a + firstLen, secondLen, c );
    memcpy( a, c, sizeof( int ) * aLen );
    delete[] c;
}
```

Сортировка слиянием

Утверждение. Время работы сортировки слиянием = $O(n \log n)$.
Доказательство.
Рекуррентное соотношение

$$T(n) \leq 2T\left(\frac{n}{2}\right) + c \cdot n,$$

разложим дальше

$$T(n) \leq 2T\left(\frac{n}{2}\right) + c \cdot n \leq 4\mathrm{T}\left(\frac{n}{4}\right) + 2c \cdot n \leq \cdots \leq 2^k T(1) + k \cdot c \cdot n.$$

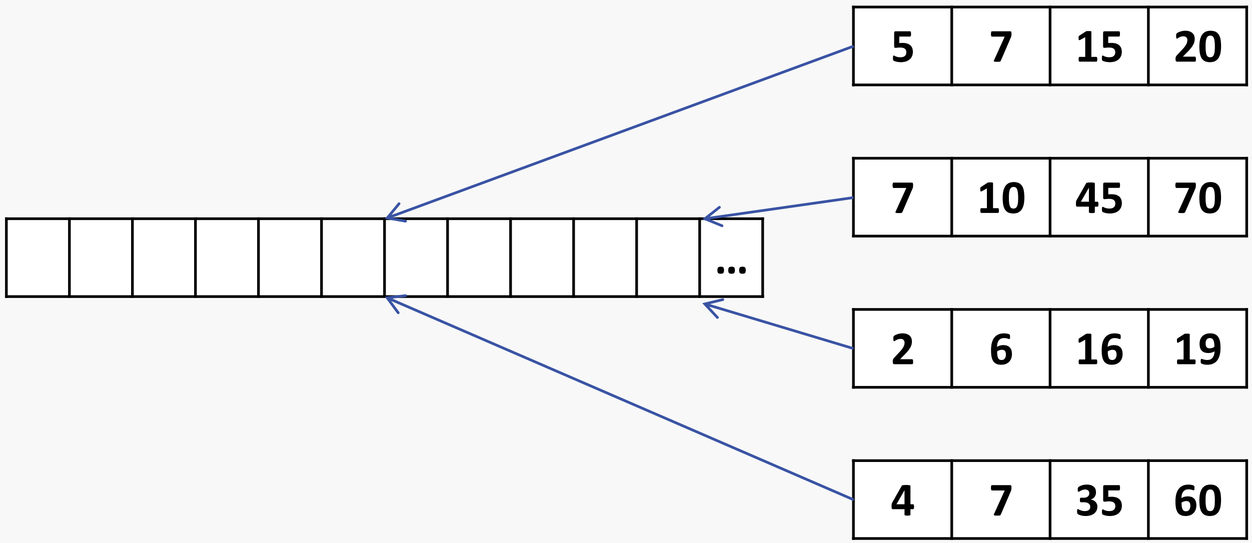
$k = \log n$, следовательно,

$$T(n) = O(n \log n).$$

Используется доп. память $M(n) = O(n)$.

k-путевое слияние

Задача. Дано k отсортированных массивов $A_1, A_2, \ldots, A_k$.
Слить в один.



k-путевое слияние

Решение.

- Построим min-heap из первых элементов $A_1[0], A_2[0], \ldots, A_k[0]$.
- Пока куча не пуста:
  - Скопируем минимум из кучи в результат.
  - Если минимальный элемент – последний в своем массиве,
    - то извлечем элемент из кучи,
    - иначе заменим его следующим из того же массива.
  - Восстановим свойство кучи – SiftDown(0).

Чтобы определять индекс массива по элементу в куче, будем хранить в куче не сам элемент, а пару <индекс массива, индекс элемента>.

$T(n,k) = O(k + n \log k)$, $M(n,k) = O(k)$.