Алгоритмы
и
структуры данных

СОРТИРОВКА. ЧАСТЬ 2

УЛУЧШЕННЫЕ СОРТИРОВКИ. ВНЕШНИЕ СОРТИРОВКИ

Сортировка - процесс перегруппировки элементов в определенном порядке.

Цель - облегчить последующий поиск элементов в отсортированном множестве.

Выбор алгоритма зависит от структуры обрабатываемых данных:

- Внутренняя – для массивов (быстрая память, прямой доступ к элементам)
- Внешняя – для файлов (более медленная, но и более емкая внешняя память)

Понятия и обозначения:

Если есть элементы

a[1], a[2], a[3] . . . a[N],

то сортировка есть перестановка этих элементов в массив, так что

a[k1], a[k2], a[k3] . . . a[kN],

где при некоторой упорядочивающей функции f выполняются отношения

f (a[k1]) ≤ f (a[k2]) ≤ f (a[k3]) ≤. . . ≤ f (a[kN])

.

Обычно,

Ключ каждого элемента не вычисляется, а хранится как явная компонента (поле)

Для простоты будем учитывать только на ключ, сопутствующую информацию будем опускать

Сортировка массивов

Одно из основных условий – экономное использование доступной памяти,

перестановки, приводящие элементы в порядок, должны выполняться на том же месте.

Метод сортировки будем называть устойчивым,

если в процессе сортировки относительное расположение элементов с равными ключами не изменяется.

(4,a) (3,b) (4,c) (5,d)

(3,b) (4,a) (4,c) (5,d) - устойчивый

(3,b) (4,c) (4,a) (5,d)

Обозначения:

N – число сортируемых элементов

C – число необходимых сравнений

M – число перестановок (пересылок) элементов

.

Сортировка

1. Прямые (сортировка по методу вставок, по методу выбора, простыми обменами) O(n*n)

2. Особые (сортировка подсчетом, поразрядная сортировка, карманная сортировка) O(n)

3. Улучшенные сортировки

  3.1 Сортировка Шелла

  3.2 Пирамидальная сортировка

  3.3 Быстрая сортировка

  3.4 Сортировка Introsort

  3.4 Сортировка слиянием

  3.5 Сортировка деревом

  3.6 Сортировка Timsort

4. Внешние сортировки

  4.1 Сортировка простым слиянием

  4.2 Сортировка естественным слиянием

Улучшенные методы (n * log n):

Пирамидальная сортировка

Быстрая сортировка

Сортировка Шелла

Эффективность метода Шелла объясняется тем, что сдвигаемые элементы быстро попадают на нужные места.

Среднее время для сортировки Шелла равняется $O(n^{1.25})$,

для худшего случая оценкой является $O(n^{1.5})$.

3.2 Пирамидальная сортировка

3.2.1 Пирамида (binary heap)

3.2.2 Поддержка свойств пирамиды

3.2.3 Построение пирамиды

3.2.4 Пирамидальная сортировка

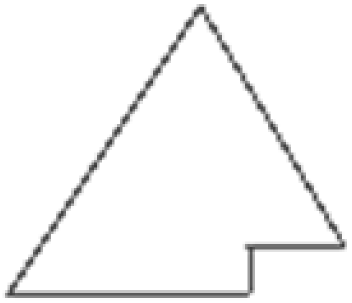
3.2.5 Очереди с приоритетами

3.2.1 Пирамида

Пирамида(heap) - бинарное дерево высоты k, в котором

- все узлы имеют глубину k или k-1 - дерево сбалансированное.
- при этом уровень k-1 полностью заполнен, а уровень k заполнен слева направо,
- выполняется "*свойство пирамиды*“

Форма пирамиды имеет приблизительно такой вид:



Два вида бинарных пирамид: невозрастающие и неубывающие

Свойство *невозрастающих пирамид (max-heap property):*

Для каждого узла (кроме корня) значение узла не больше значения родительского узла =>

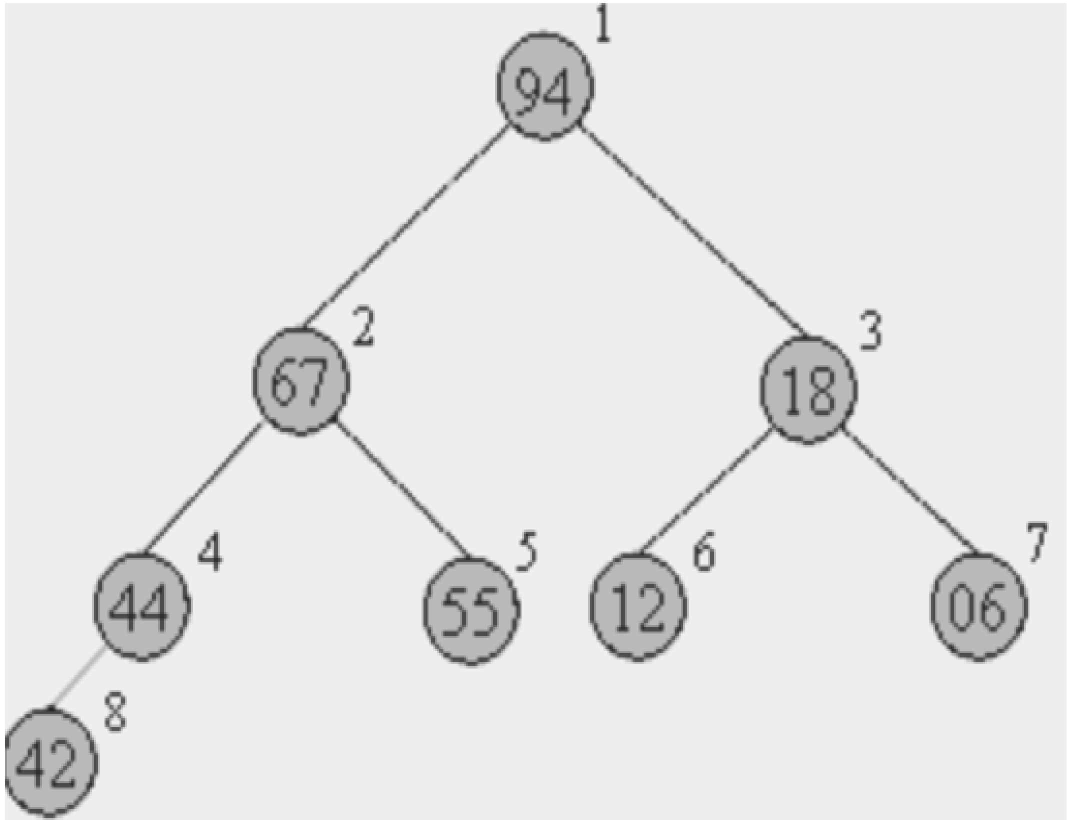
Корень – наибольший элемент

Свойство неубывающих пирамид (min-heap property):

Для каждого узла (кроме корня) значение узла не меньше значения родительского узла =>

Корень – наименьший элемент

Невозрастающая пирамида (max-heap) , представленная в виде бинарного дерева:



Как хранить пирамиду? Поместим ее в массив - A [1 . . *length*[A]]

A[1] – корень дерева

A[2] – левый дочерний узел корня

A[3] – правый дочерний узел корня

Если i – индекс узла,

Parent (i) // индекс родительского узла

return i / 2

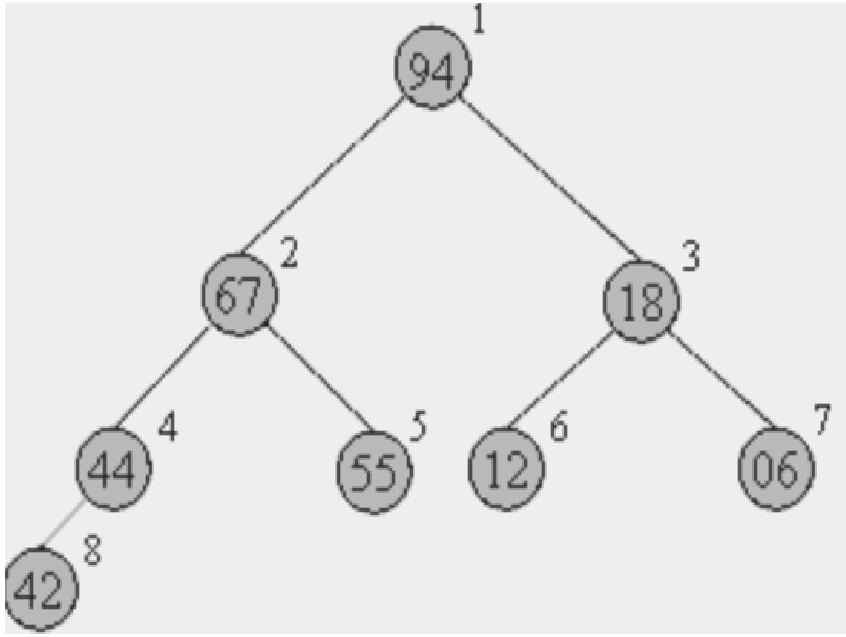
Left (i) // индекс левого дочернего узла

return 2 * i

Right (i) // индекс правого дочернего узла

return 2 * i +1

Невозрастающая пирамида (*max-heap*), представленная в виде бинарного дерева:



Пирамида (*max-heap*), представленная в виде массива:

94 67 18 44 55 12 06 42

Плюсы хранения пирамиды в массиве:

- никаких дополнительных переменных, нужно лишь понимать схему
- узлы хранятся от вершины и далее вниз, уровень за уровнем
- узлы одного уровня хранятся в массиве слева направо

Для записи в виде массива пирамиду надо обойти дерево по уровням сверху-вниз.

Восстановить пирамиду из массива как геометрический объект легко - достаточно вспомнить схему хранения и нарисовать, начиная от корня

Пирамида – объект-массив,

который рассматривается как почти полное двоичное дерево.

Рассматриваем только ***невозрастающие пирамиды (max-heap):***

Для каждого узла (кроме корня) с индексом i выполняется:

A[Parent(i)] >= A[i]

Значение узла не превосходит значение родительского узла =>

Корень – наибольший элемент

Пирамида – объект-массив:

Высота пирамиды определяется как высота двоичного дерева (количество рёбер в самом длинном простом пути, соединяющем корень пирамиды с одним из её листьев).

Пирамида - Двоичная куча - Сортирующее дерево

Биномиальная куча, Фибоначчиева куча

3.2.2 Поддержка свойств пирамиды (**невозрастающей**)

Процедура *MAX_HEAPIFY*:

Вход – массив А,   индекс i

Предполагается,

что деревья с корнями A[Left (i)] и A[Right (i)] – уже *max-heap,*

НО A[i] может быть меньше своих дочерних элементов.

Процедура восстанавливает свойство упорядоченности во всём поддереве, корнем которого является элемент A[i].

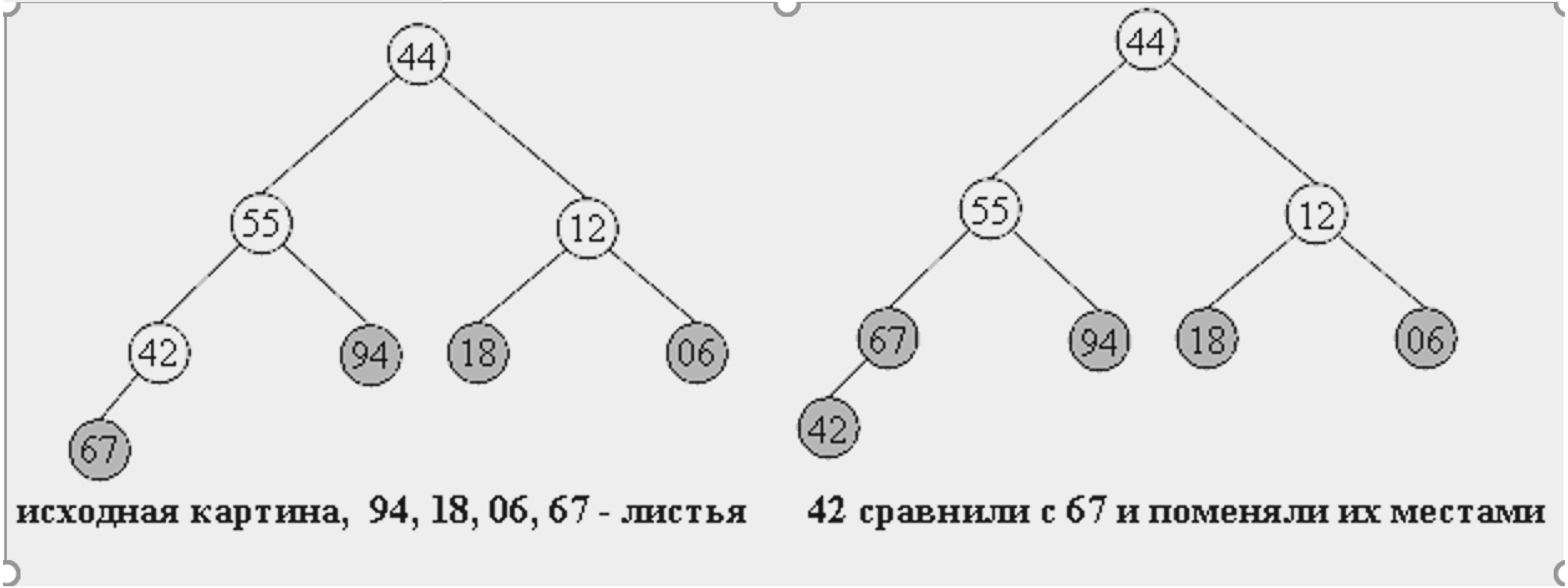
Время работы – O(ln n)

Процесс построения пирамиды

Начало массива (неготовая часть пирамиды) окрашена в белый цвет,

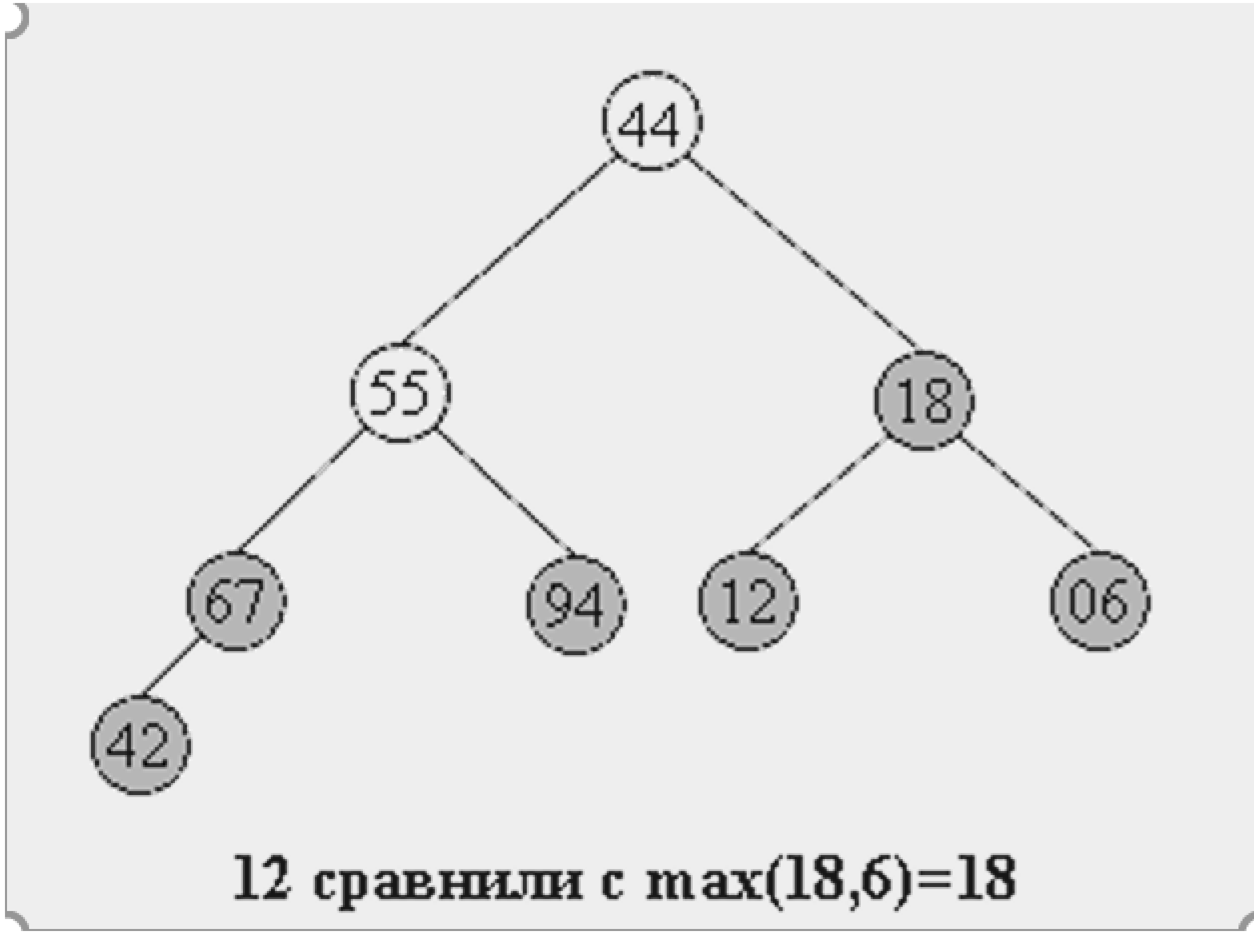
конец массива, удовлетворяющий свойству пирамиды, - в темный.

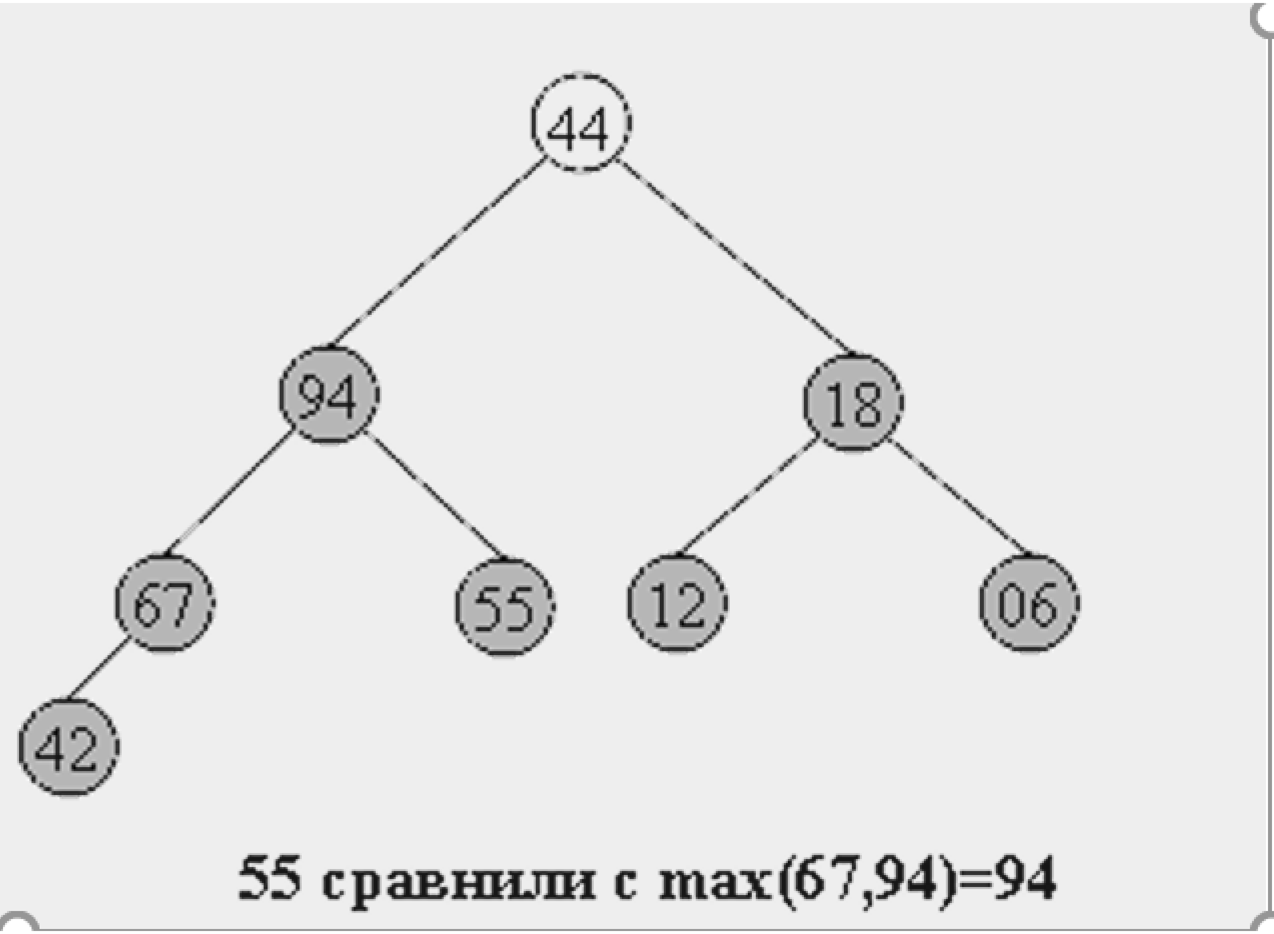
44 55 12 42 // 94 18 06 67



44 55 12 42 **94 18 06 67**

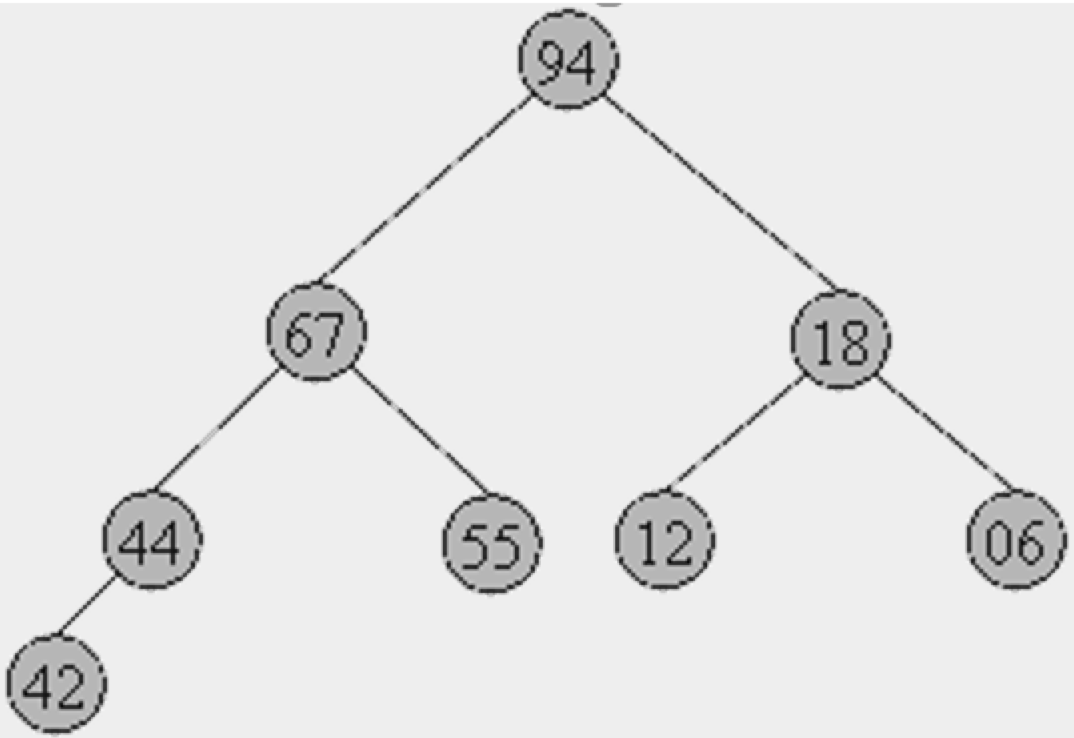
44 55 12 **67 94 18 06 42**





44 55 **18 67 94 12 06 42**

44 **94 18 67 67 55 12 06 42**



44 просеяли сквозь 94, 67, остановились на 42

94 67 18 44 55 12 06 42

Восстановление свойства упорядоченности во всём поддереве, корнем которого является элемент A[i]:

Если i-й элемент больше, чем его дочерние, всё поддерево уже является пирамидой

(делать ничего не надо).

В противном случае меняем местами i-й элемент с наибольшим из его дочерних, после чего выполняем *Max_Heapify* для этого дочернего.

MAX_HEAPIFY(A, i) ► *heap_size - количество элементов в куче*

```
left ← 2i
right ← 2i+1
largest ← i
 if left ≤ heap_size [A] and A[left] > A[largest] then
     largest ← left
 end if
if right ≤ heap_size [A] and  A[right] > A[largest]  then
    largest ← right
end if
 if largest ≠ i   then
     Обменять A[i] ↔ A[largest]
     MAX_HEAPIFY (A, largest)
end if
```

3.2.3 Построение пирамиды

Создание пирамиды из неупорядоченного массива входных данных

Если выполнить MAX_HEAPIFY для всех элементов массива A, начиная с последнего и кончая первым, он станет пирамидой.

По индукции:

к моменту выполнения MAX_HEAPIFY *(A, i)* все поддеревья, чьи корни имеют индекс больше i,- пирамиды.

→ после выполнения MAX_HEAPIFY *(A, i)* пирамидой будут все поддеревья, чьи корни имеют индекс, не меньший i.

MAX_HEAPIFY*(A,i)* не делает ничего, если i>n/2 (при нумерации с первого элемента), где n — количество элементов массива. (у таких элементов нет потомков => соответствующие поддеревья уже являются пирамидами, так как содержат всего один элемент).

→достаточно вызвать MAX_HEAPIFY для всех элементов массива A, начиная (при нумерации с первого элемента) с n/2 -го и кончая первым.

BUILD_MAX_HEAP(A)

```
heap_size [A] ← length [A]
for i ← ⌊length [A] /2⌋ downto 1  do
     MAX_HEAPIFY (A, i)
 end for
```

Здесь происходит n/2 вызовов функции MAX_HEAPIFY со сложностью

O(ln n) => время работы алгоритма - O(n ln n)

НО чаще всего функция MAX_HEAPIFY вызывается для деревьев маленькой глубины, можно показать, что время работы BUILD_MAX_HEAP равно O(n)

3.2.4 Пирамидальная сортировка

Процедура HEAPSORT сортирует массив без привлечения дополнительной памяти за время O(n ln n).

Метод *не является устойчивым*: по ходу работы массив так "перетряхивается", что исходный порядок элементов может измениться случайным образом.

Поведение *неестественно*: частичная упорядоченность массива никак не учитывается

Обозначения:

A – массив-пирамида

A [1 . . *length*[A]]

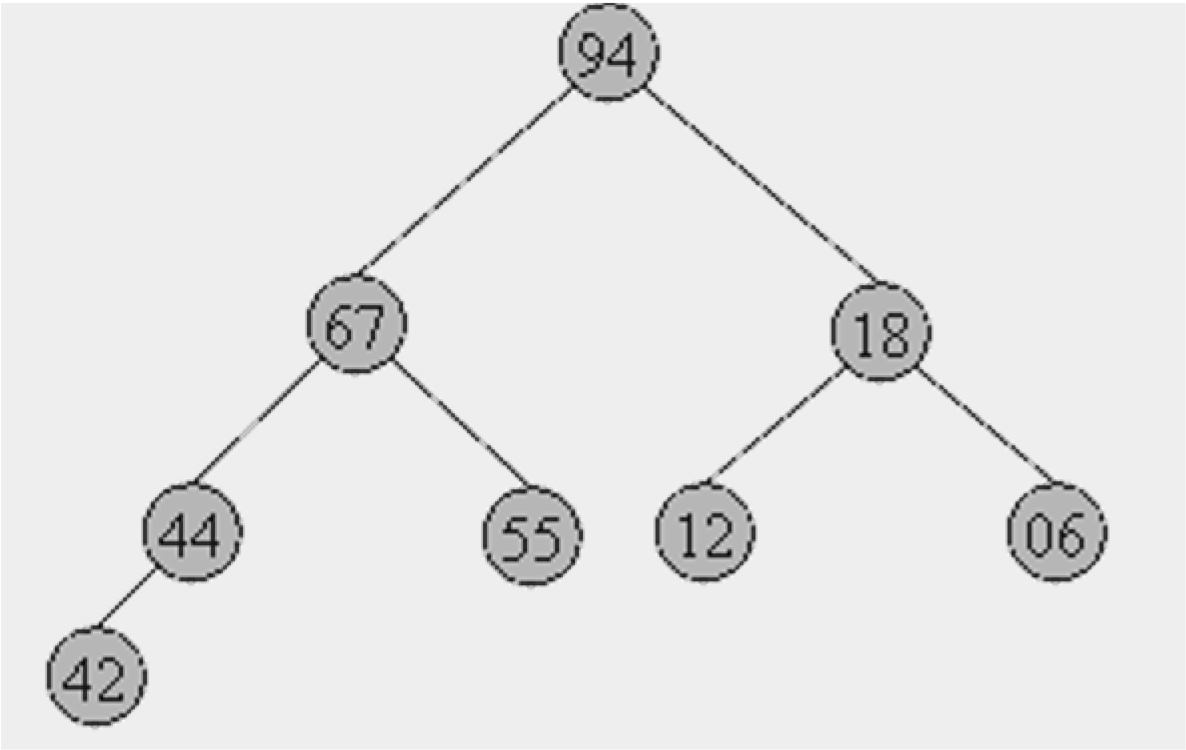
length[A] – количество элементов массива

heap_size[A] – количество элементов пирамиды, содержащихся в массиве

A [1 . . *heap_size*[A]] - элементы пирамиды

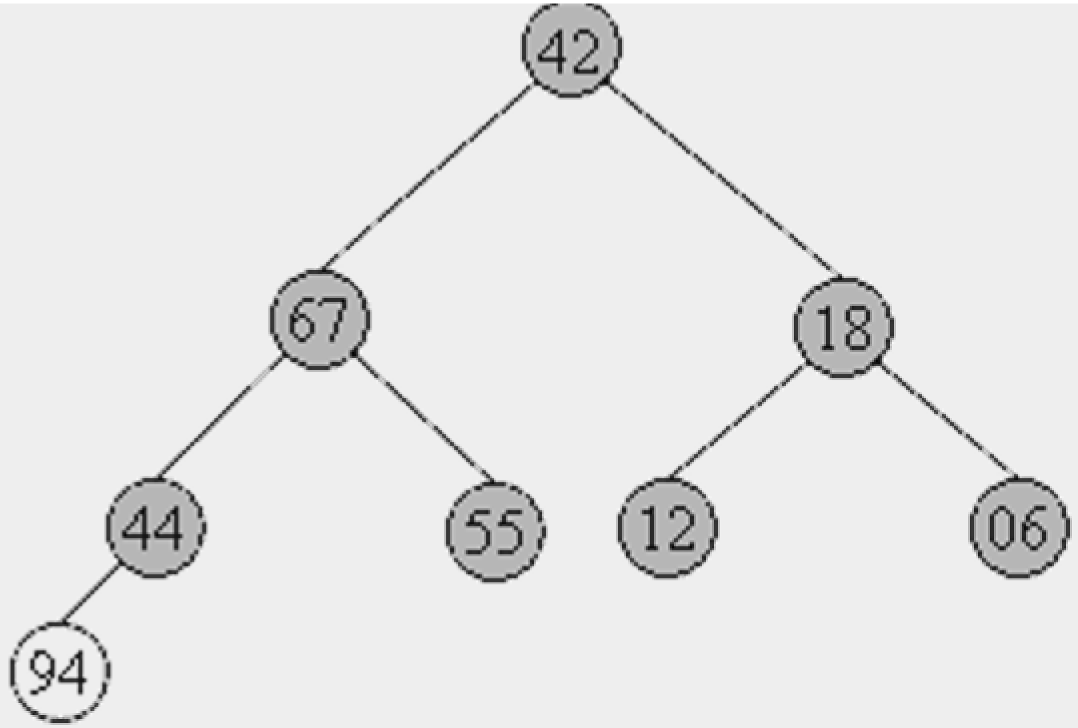
A [*heap_size*[A]+1 . . *length*[A]] - не элементы пирамиды

Пирамида построена:

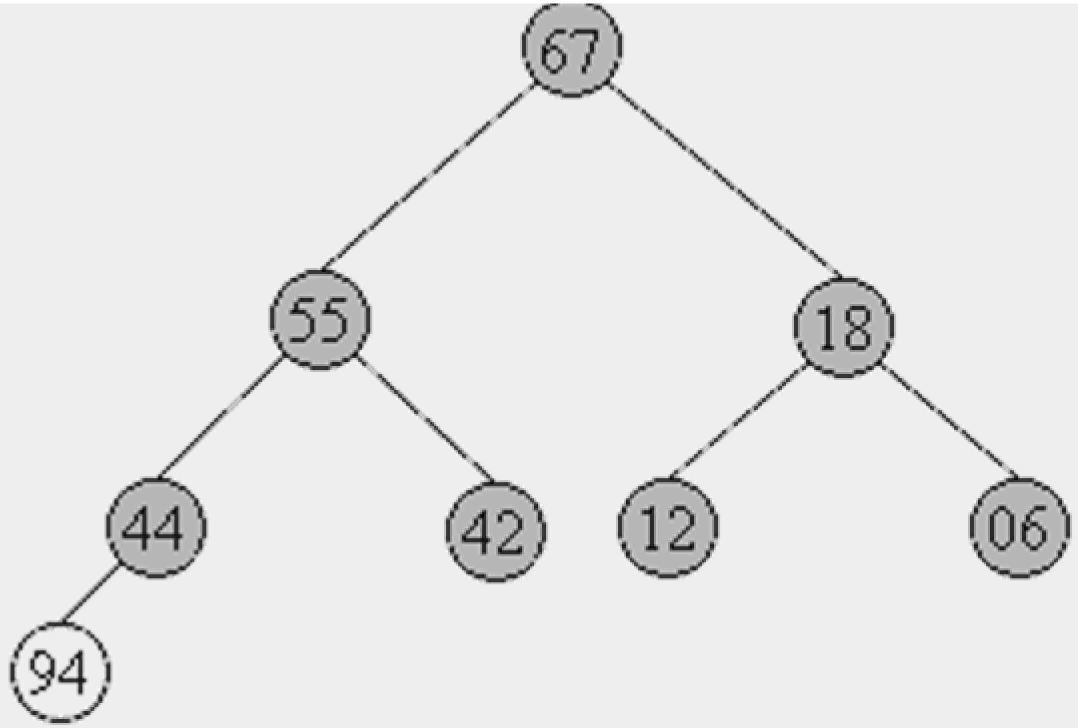


Пирамида построена (первый раз): *heap_size[A] = length[A]*

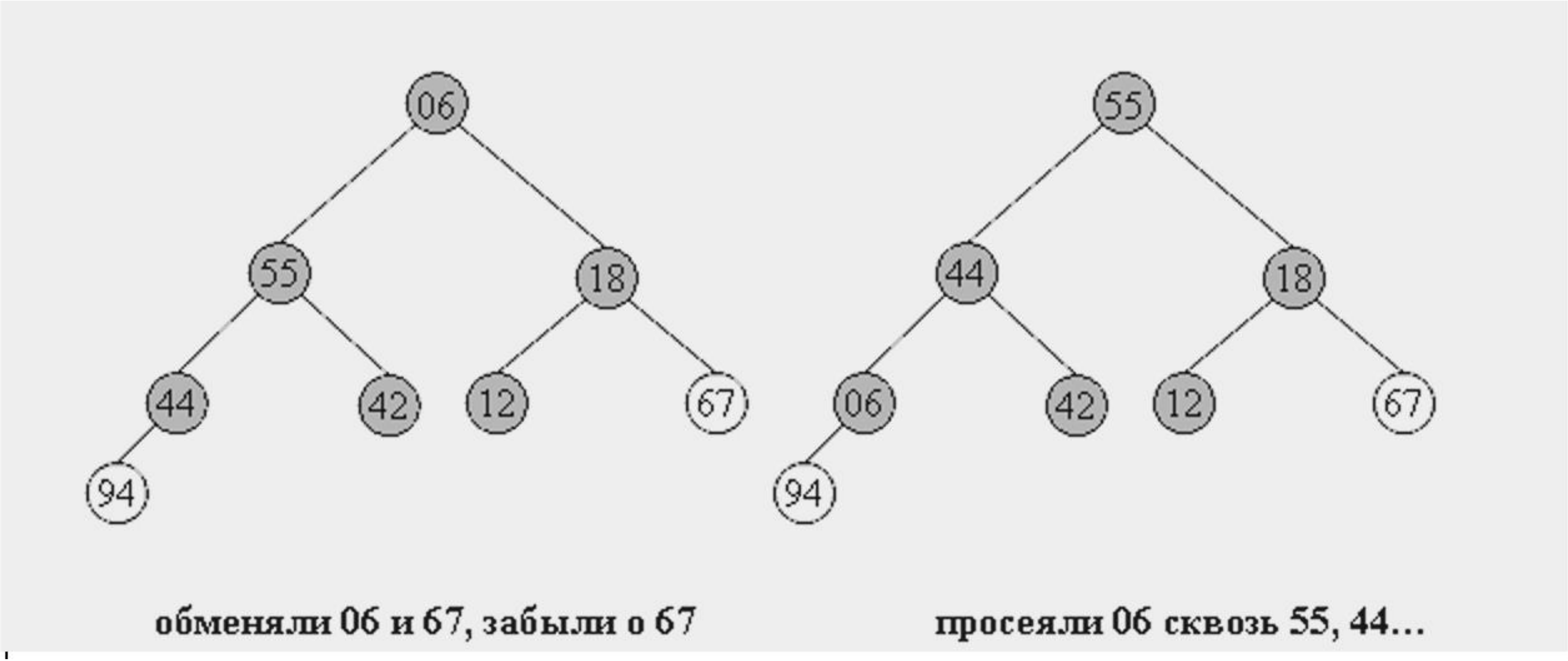
1. Поменяем первый элемент (то есть корень) с последним элементом ==> последний элемент станет самым большим.
2. Исключим последний элемент из пирамиды (то есть формально уменьшим её длину *heap_size[A]* на 1),
3. Первые *heap_size[A]* элементов будут удовлетворять условиям пирамиды все, за исключением, может быть, корня.
4. Вызовем Heapify, первые *heap_size[A]* элементов станут пирамидой.
5. Повторяя эти действия *length[A] -1* раз, мы отсортируем массив.



обменяли 94 и 42, забыли о 94



просеяли 42 сквозь 67, 55


06
55
18
44
42
12
67
94
обменяли 06 и 67, забыли о 67
55
44
18
06
42
12
67
94
просеяли 06 сквозь 55, 44…

HEAPSORT(A)

1. BUILD_MAX_HEAP(A)
2. **for** i ← length[A] **downto** 1 **do**
3. Обменять A[1] ↔ A [i]
4. heap_size [A] ← heap_size [A]-1
5. MAX_HEAPIFY(A,1)
6. **end for**

Процедура HEAPSORT сортирует массив без привлечения дополнительной памяти

за время O(n ln n), т.к.

Вызов BUILD_MAX_HEAP*(A) – O(n),*

Каждый из n вызовов MAX_HEAPIFY– *O(ln n).*

3.2.5 Очереди с приоритетами

Два типа: невозрастающие и неубывающие.

Реализация невозрастающих очередей с приоритетами на основе невозрастающих пирамид.

Очередь с приоритетами (priority queue) – структура данных, предназначенная для обслуживание множества S, с каждым элементов которого связано определенное значение, называемое *ключом* (key).

Операции невозрастающей очереди с приоритетами:

INSERT (S,x) - вставить элемент x в множество S

MAXIMUM(S) - возвратить элемент множества S с наибольшим ключом

EXTRACT_MAX(S) - возвратить элемент множества S с наибольшим ключом и удалить его из множества

INCREASE_KEY(S,x,k) - увеличить значения ключа элемента x, путем замены его на ключ k (величина k не меньше текущего значения ключа x)

► возвратить элемент множества S с наибольшим ключом

HEAP_ MAXIMUM (A)

1. **if** heap_size[A] < 1 **then**
2. error “очередь пуста”
3. **else**
4. return A[1]
5. **end if**

► возвратить элемент множества S с наибольшим ключом и удалить его

HEAP_ EXTRACT_MAX (A)

1. **if** heap_size[A] < 1 **then**
2. error “очередь пуста” ►выход из функции
3. max ← A[1]
4. A[1] ← A[heap_size[A]]
6. heap_size[A] ← heap_size[A] - 1
7. MAX_HEAPIFY(A,1)
8. **return** max

► увеличить значения ключа элемента i, путем замены его на ключ key

► величина key не меньше текущего значения ключа записи i A[i]

HEAP_ INCREASE_KEY(A, i, key)

1. **If** key < A[i] **then**
2. **error** “новый ключ меньше текущего” ►выход из функции
3. A[i] $\leftarrow$ key
4. **while** i > 1 **and** A[Parent(i)] < A[i] **do**
5. обменять A[i] $\leftrightarrow$ A[Parent(i)]
6. **i** $\leftarrow$ Parent(i)
7. **end do**

► вставить элемент с ключом key в множество

MAX_HEAP_INSERT (A, key)

heap_size[A] ← heap_size[A] + 1

A[heap_size[A]] ← - ∞

HEAP_ INCREASE_KEY(A, heap_size[A] , key)

Операция	Время выполнения
HEAP_ MAXIMUM (A)	ϴ (1)
HEAP_ EXTRACT_MAX (A)	O(ln n)
HEAP_ INCREASE_KEY (A,i,key)	O(ln n)
MAX_HEAP_INSERT (A,key)	O(ln n)

STL C++:

контейнер priority_queue,

шаблоны функций для управления кучей make_heap, push_heap и pop_heap (бинарные кучи).

3.3 Быстрая сортировка.
Метод декомпозиции («разделяй и властвуй»)

Быстрая сортировка", разработана более 40 лет назад, является наиболее широко применяемым и одним их самых эффективных алгоритмов.

Рассмотрим сортировку подмассива A[p..r].

Подход «разделяй и властвуй» - решение будет состоять из следующих этапов:

- *Разделение.*
- *Покорение.*
- *Комбинирование*

Разделение.

Массив A[p..r] разбивается на два (возможно, пустых) подмассива A[p..q-1] и A[q+1..r]. Каждый элемент подмассива A[p..q-1] не превышает элемент A[q], а каждый элемент подмассива A[q+1..r] не меньше A[q]. Индекс q вычисляется в ходе выполнения процедуры.

Покорение.

Подмассивы A[p..q-1] и A[q+1..r] сортируются путем рекурсивного вызова процедуры быстрой сортировки.

Комбинирование.

Поскольку подмассивы сортируются на месте для их объединения не нужны никакие действия, весь массив A[p..r] оказывается отсортирован.

QUICKSORT(A, p, r)

1. **if** p < r **then**

2. q ← PARTITION(A, p, r)

3. QUICKSORT(A, p, q-1)

4. QUICKSORT(A, q+1, r)

5. **end if**

Для сортировки всего массива A, следует выполнить QUICKSORT(A, 1, length[A]).

- Процедура PARTITION изменяет порядок следования элементов подмассива A[p..r] без использования дополнительной памяти.
- Каждое разделение требует Θ (n) операций.
- Количество шагов деления (глубина рекурсии) составляет приблизительно log n, если массив делится на более-менее равные части.

Таким образом, общее быстродействие: O (n log n), что и имеет место на практике.

Сортировка использует дополнительную память, так как приблизительная глубина рекурсии составляет O(log n), а данные о рекурсивных под вызовах каждый раз добавляются в стек.

PARTITION(A, p, r)

```
x ← A[r]
i ← p – 1
for j ← p to r – 1 do
   if A[j] ≤  x then
       i ← i + 1
       обменять  A[i] ↔  A[j]
   end if
обменять A[i+1 ] ↔  A[r]
return i + 1
```

Анализ.

Худший случай: набор входных данных, на которых алгоритм будет работать за $O(n^2)$.

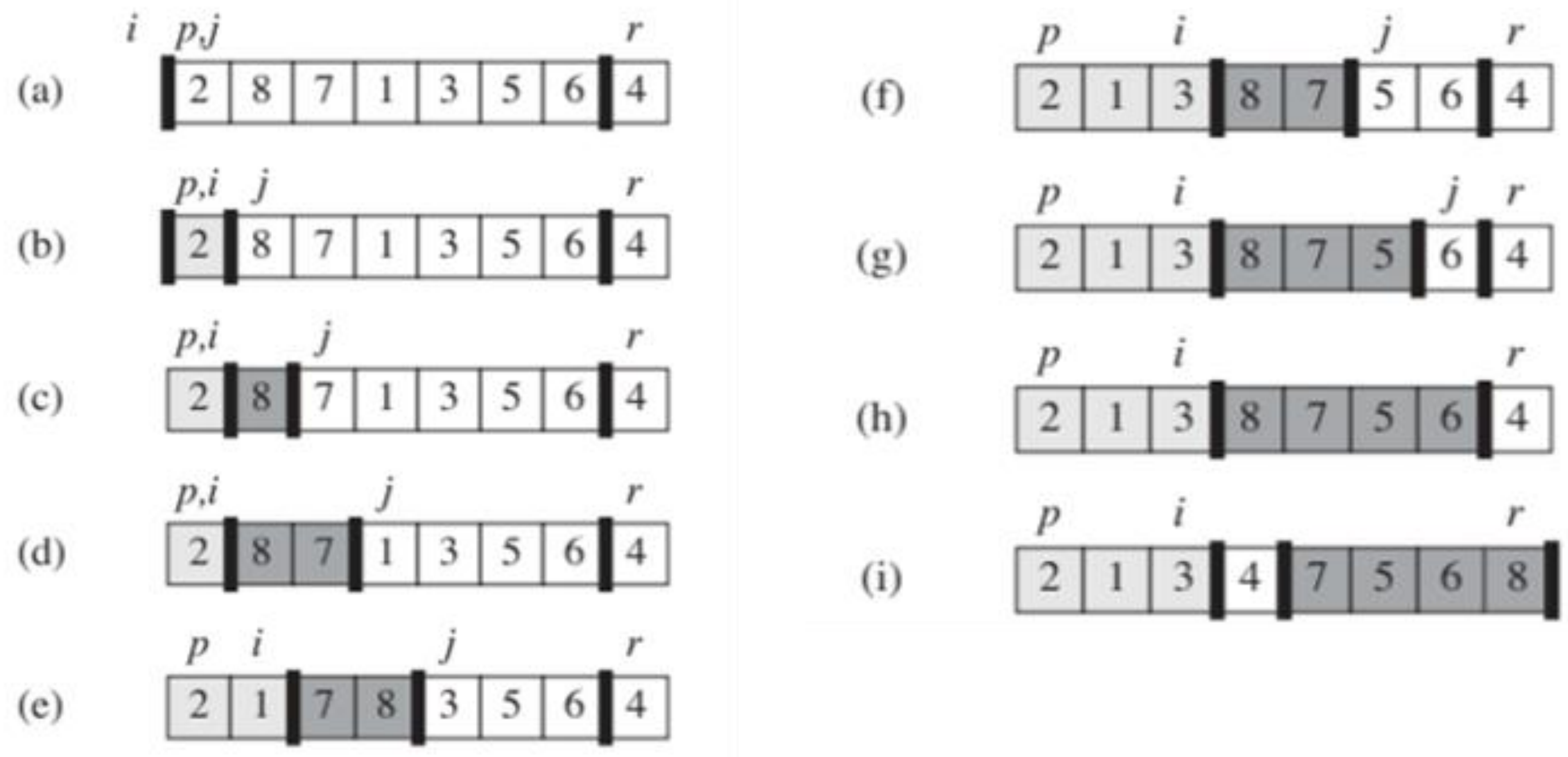
Если каждый раз в качестве центрального элемента выбирается **максимум или минимум** входной последовательности.

Если данные взяты случайно, вероятность этого равна $\mathbf{2^n}$. И эта вероятность должна реализовываться на каждом шаге. → малореальная ситуация.

Метод неустойчив.

Поведение довольно естественно, если учесть, что при частичной упорядоченности повышаются шансы разделения массива на более равные части.

Разбиение массива. Пример действия процедуры PARTITION на массив, *опорный элемент r*



Общая схема быстрой сортировки:

1. из массива выбирается некоторый опорный элемент, например, x = a[r],

2. запускается процедура разделения массива, которая перемещает все ключи, меньшие, либо равные x, влево от него, а все ключи, большие, либо равные x - вправо,

3. теперь массив состоит из двух подмножеств, причем левое содержит элементы меньшие, либо равные, элементов из правого,

A[p.q-1] x = A[q] A[q+1..r].

4. для обоих подмассивов: если в подмассиве более двух элементов, рекурсивно запускаем для него ту же процедуру.

В конце получится полностью отсортированная последовательность.

3.4 *Сортировка Introsort или интроспективная сортировка*

алгоритм сортировки, предложен Дэвидом Мюссером (англ.)рус. в 1997 году.

Использует быструю сортировку и переключается на пирамидальную сортировку, когда глубина рекурсии превысит некоторый заранее установленный уровень (например, логарифм от числа сортируемых элементов).

Подход сочетает в себе достоинства обоих методов с худшим случаем O(nlogn). и быстродействием, сравнимым с быстрой сортировкой.

Оба алгоритма используют сравнения, этот алгоритм также принадлежит классу сортировок на основе сравнений.

В быстрой сортировке одна из критичных операций — выбор опоры (элемент, относительно которого разбивается массив).

Простейший алгоритм выбора опоры — взятие первого или последнего элемента массива за опору - плохим поведение на отсортированных или почти отсортированных данных.

Никлаус Вирт - предложил серединный элемент для предотвращения этого случая, деградирующего до $O(n^2)$ при неудачных входных данных.

Алгоритм выбора опоры «медиана трёх» выбирает опорой средний из первого, среднего и последнего элементов массива.

Это хорошо на большинстве входных данных, но можно найти такие входные данные, которые сильно замедлят этот алгоритм сортировки.

Злоумышленники: могут посылать такой массив Веб-серверу, добиваясь отказа в обслуживании.

В быстрой сортировке одна из критичных операций — выбор опоры (элемент, относительно которого разбивается массив).

Простейший алгоритм выбора опоры — взятие первого или последнего элемента массива за опору - плохим поведение на отсортированных или почти отсортированных данных.

Никлаус Вирт - предложил серединный элемент для предотвращения этого случая, деградирующего до $O(n^2)$ при неудачных входных данных.

Алгоритм выбора опоры «медиана трёх» выбирает опорой средний из первого, среднего и последнего элементов массива. Это хорошо на большинстве входных данных, но можно найти такие входные данные, которые сильно замедлят этот алгоритм сортировки.

Злоумышленники: могут посылать такой массив Веб-серверу, добиваясь отказа в обслуживании.

3.5 *Сортировка слиянием.*

Метод декомпозиции («разделяй и властвуй»)

Идея метода:

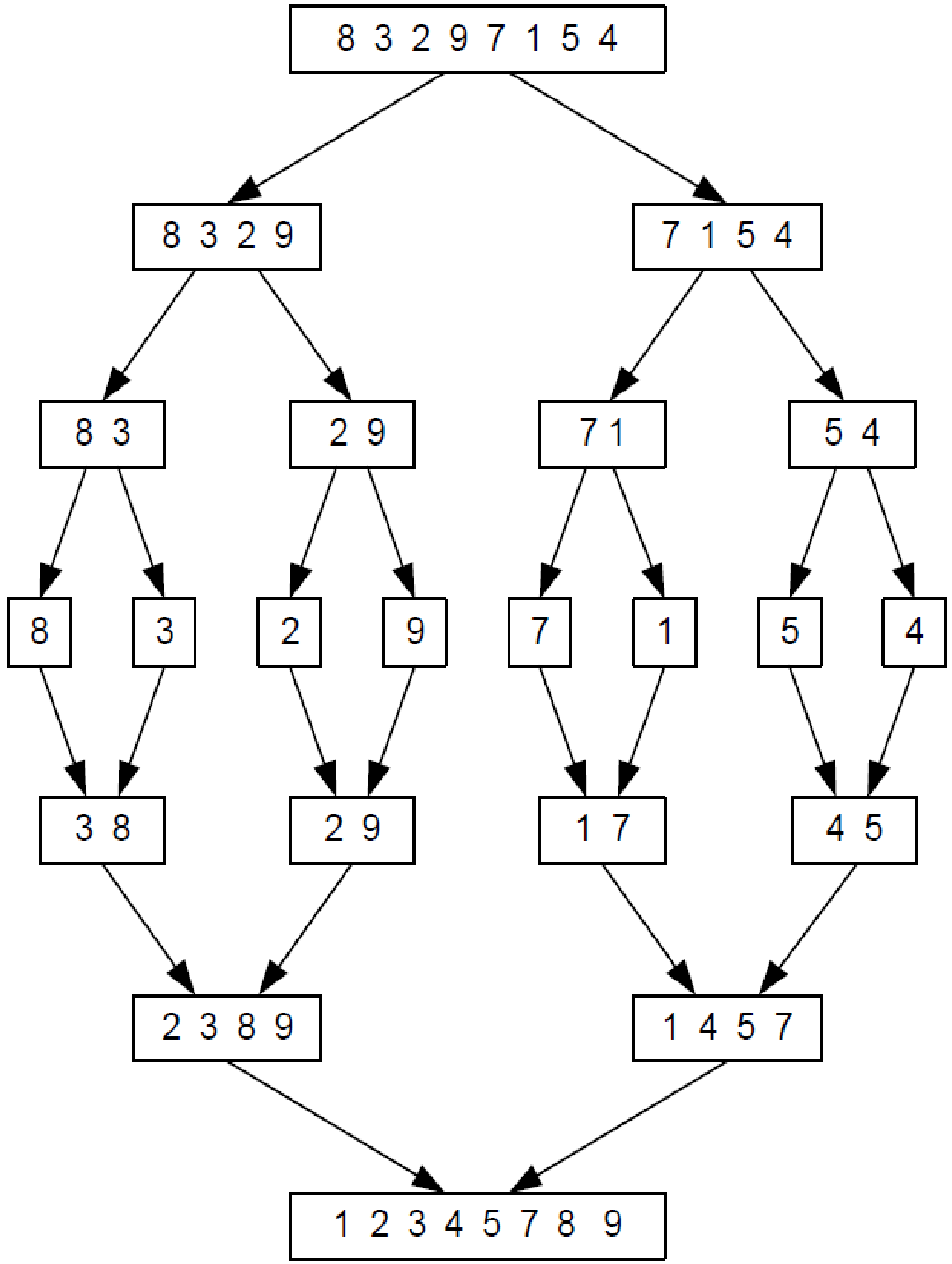
Разделим массив пополам,

Рекурсивно отсортируем части,

После чего выполним процедуру слияния: поддерживаем два указателя, один на текущий элемент первой части, второй – на текущий элемент второй части. Из этих двух элементов выбираем минимальный, вставляем в ответ и сдвигаем указатель…

Слияние - O(n), уровней - logn $\rightarrow$ O(nlogn).

3.5 Сортировка слиянием



3.6 *Сортировка деревом*

Идея метода:

1. Из сортируемых элементов строим двоичное дерево поиска.

2. Выполняем инфиксный обход дерева и получаем отсортированный массив.

Если строить сбалансированное дерево (красно-черное дерево),

То в худшем, среднем и лучшем случае: $O(n \log n)$

3.7 *Сортировка Timsort*

Timsort — разработан в 2002 году Тимом Петерсом.

Стандартный алгоритмом сортировки в Python.

Часто сортируемый массив данных часто содержат в себе упорядоченные (не важно, по возрастанию или по убыванию) подмассивы. На таких данных Timsort эффективен.

Сложность алгоритма – O(nlogn).

Дополнительная память - O(n)

Timsort — это эффективная комбинация нескольких других алгоритмов и собственных идей.

Суть алгоритма:

1. По специальному алгоритму разделяем входной массив на подмассивы.
2. Сортируем каждый подмассив обычной сортировкой вставками.
3. Собираем отсортированные подмассивы в единый массив с помощью модифицированной сортировки слиянием.

4. *Внешняя сортировка*

Применяются к данным, хранящимся

- во внешней памяти
- на внешних *устройствах последовательного доступа*
- в каждый момент времени доступен только один компонент данных

Внешняя сортировка - это *сортировка* данных, которые расположены на *внешних устройствах* и не вмещающихся в *оперативную память*.

Наиболее известны алгоритмы:

- сортировки слиянием (простое слияние и естественное слияние);
- улучшенные сортировки (многофазная сортировка и каскадная сортировка).

Серия (упорядоченный отрезок) – последовательность элементов, которая упорядочена по ключу.

Длина серии - количество элементов в серии.

- Серия, состоящая из одного элемента - упорядочена.
- Последняя серия может иметь длину меньшую, чем остальные серии.
- Максимальное количество серий в файле - N (все элементы не упорядочены).
- Минимальное количество серий - 1 (все элементы упорядочены).

Распределение – процесс разделения упорядоченных серий на два и несколько вспомогательных файлов.

Фаза – действия по однократной обработке всей последовательности элементов.

Двухфазная сортировка – сортировка, в которой отдельно реализуется две фазы: распределение и слияние.

Однофазная сортировка – сортировка, в которой объединены фазы распределения и слияния в одну.

Двухпутевое слияние - сортировка, в которой данные распределяются на два вспомогательных файла.

Многопутевое слияние - сортировка, в которой данные распределяются на N (N > 2) вспомогательных файлов.

Основные характеристики сортировки слиянием:

- количество фаз в реализации сортировки;
- количество вспомогательных файлов, на которые распределяются серии.

Сортировка простым слиянием –

Основана на процедуре *слияния серией*.

Длина серий фиксируется на каждом шаге.

В исходном файле все серии имеют длину 1,

после первого шага она равна 2,

после второго – 4, . . .

после **k** -го шага – 2^k.

Сортировка двухпутевым (f1, f2) двухфазным (распределение, слияние) простым слиянием

Исходный файл **f: 3 2 4 5 6 9 8 7 1**

	Распределение	*Слияние*
1 проход	f1: 3 4 6 8 1 f2: 2 5 9 7	f: 2 3 4 5 6 9 7 8 1
2 проход	f1: 2 3 6 9 1 f2: 4 5 7 8	f: 2 3 4 5 6 7 8 9 1
3 проход	f1: 2 3 4 5 1 f2: 6 7 8 9	f: 2 3 4 5 6 7 8 9 1
4 проход	f1: 2 3 4 5 6 7 8 9 f2: 1	f: 1 2 3 4 5 6 7 8 9

Алгоритм сортировки простым слиянием

Шаг 1. Исходный файл f разбивается на два вспомогательных файла f1 и f2.

Шаг 2. Вспомогательные файлы f1 и f2 сливаются в файл f, при этом одиночные элементы образуют упорядоченные пары.

Шаг 3. Полученный файл f вновь обрабатывается, как указано в шагах 1 и 2. При этом упорядоченные пары переходят в упорядоченные четверки.

Шаг 4. Повторяя шаги, сливаем четверки в восьмерки и т.д., каждый раз удваивая длину слитых последовательностей до тех пор, пока не будет упорядочен целиком весь файл

После выполнения i проходов получаем два файла, состоящих из серий длины 2i. Окончание процесса происходит при выполнении условия 2i>=n.

Процесс сортировки простым слиянием требует порядка O(log n) проходов по данным.

Признаками конца сортировки простым слиянием являются следующие условия:

- длина серии не меньше количества элементов в файле (определяется после фазы слияния);
- количество серий равно 1 (определяется на фазе слияния).
- при однофазной сортировке второй по счету вспомогательный файл после распределения серий остался пустым.

Сортировка естественным слиянием –

Основана на процедуре *слияния серий*.

При естественном слиянии длина серий не ограничивается, а определяется количеством элементов в уже упорядоченных подпоследовательностях, выделяемых на каждом проходе.

Сортировка двухпутевым (f1, f2) двухфазным (распределение, слияние)

естественным слиянием

Исходный файл **f: 8 9 3 4 5 2 1 6 7**

	Распределение	*Слияние*
1 проход	f1: 8 9 '2 '6 7 f2: 3 4 5 '1	f: 3 4 5 8 9 ' 1 2 ' 6 7
2 проход	f1: 3 4 5 8 9 ' 6 7 f2: 1 2	f: 1 2 3 4 5 8 9 ' 67
3 проход	f1: 1 2 3 4 5 8 9 f2: 6 7	f:1 2 3 4 5 6 7 8 9

Алгоритм сортировки естественным слиянием

Шаг 1. Исходный файл `f` разбивается на два вспомогательных файла `f1` и `f2`. Распределение происходит следующим образом: поочередно считываются записи a_i исходной последовательности (неупорядоченной) таким образом, что если значения ключей соседних записей удовлетворяют условию $f(a_i)<=f(a_{i+1})$, то они записываются в первый вспомогательный файл `f1`. Как только встречаются $f(a_i)>f(a_{i+1})$, то записи a_{i+1} копируются во второй вспомогательный файл `f2`. Процедура повторяется до тех пор, пока все записи исходной последовательности не будут распределены по файлам.

Шаг 2. Вспомогательные файлы `f1` и `f2` сливаются в файл `f`, при этом серии образуют упорядоченные последовательности.

Шаг 3. Полученный файл `f` вновь обрабатывается, как указано в шагах 1 и 2.

Шаг 4. Повторяя шаги, сливаем упорядоченные серии до тех пор, пока не будет упорядочен целиком весь файл.

Символ "`" обозначает признак конца серии.

Признаками конца сортировки естественным слиянием являются следующие условия:

- количество серий равно 1 (определяется на фазе слияния).
- при однофазной сортировке второй по счету вспомогательный файл после распределения серий остался пустым.

Естественное слияние, у которого после фазы распределения количество серий во вспомогательных файлах отличается друг от друга не более чем на единицу, называется *сбалансированным слиянием*, в противном случае – *несбалансированное слияние.*

Число чтений или перезаписей файлов при использовании метода естественного слияния будет не хуже, чем при применении метода простого слияния, а в среднем – даже лучше.

НО

- увеличивается число сравнений за счет тех, которые требуются для распознавания концов серий.
- максимальный размер вспомогательных файлов может быть близок к размеру исходного файла, так как длина серий может быть произвольной