



Алгоритмы и структуры данных

Лекция 2. Алгоритмы сортировки.

(с) Глухих Михаил Игоревич, glukhikh@mail.ru

Основные характеристики алгоритмов сортировки

- ▶ Трудоёмкость $T=O(\dots)$ – в среднем и худшем случае
- ▶ Ресурсоёмкость $R=O(\dots)$
 - ▶ Для $R=O(1)$ говорят «сортировка на месте»

Основные характеристики алгоритмов сортировки

- ▶ Трудоёмкость $T=O(\dots)$ – в среднем и худшем случае
- ▶ Ресурсоёмкость $R=O(\dots)$
 - ▶ Для $R=O(1)$ говорят «сортировка на месте»
- ▶ Устойчивость – изменяется ли порядок равных элементов в списке
 - ▶ Бывает важна в некоторых случаях, например, при индексации баз данных
 - ▶ Точно неважна, если элемент – это только ключ сравнения
- ▶ Операции сравнения – используются или нет

Алгоритмы сортировки

- ▶ Простые

- ▶ $O(N^2)$, где N – число элементов

- ▶ Сложные

- ▶ $O(N \log_2 N)$, где N – число элементов; для больших значений N существенно быстрее простых методов

Алгоритмы сортировки

➤ Сложные

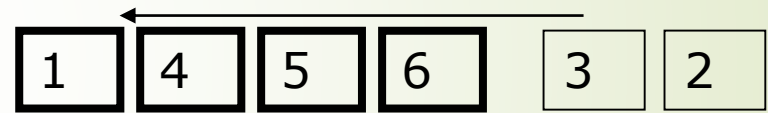
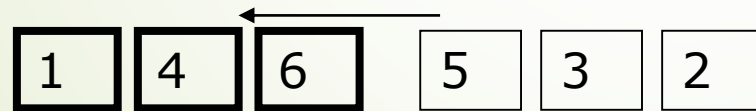
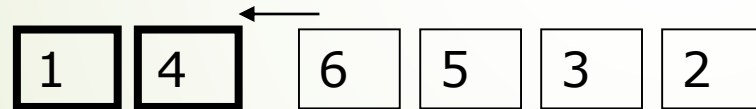
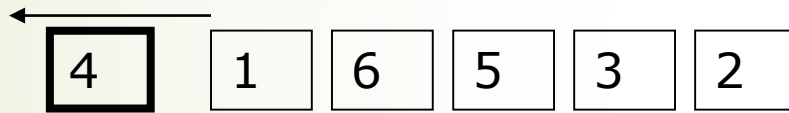
- слияниями
- Хоара, или быстрая сортировка
- двоичной кучей, или пирамидальная сортировка

➤ Простые

- включениями (вставками)
- выбором
- обменом (пузырьком)

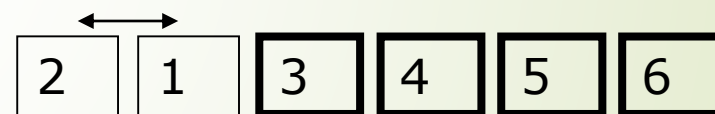
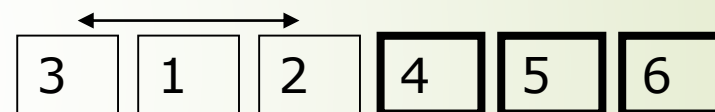
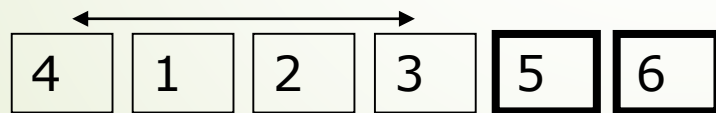
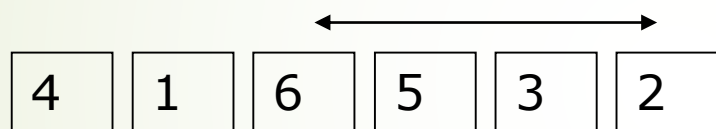
Простые алгоритмы сортировки

- Сортировка **включениями** – на каждой итерации вставляем очередной элемент в отсортированную часть массива



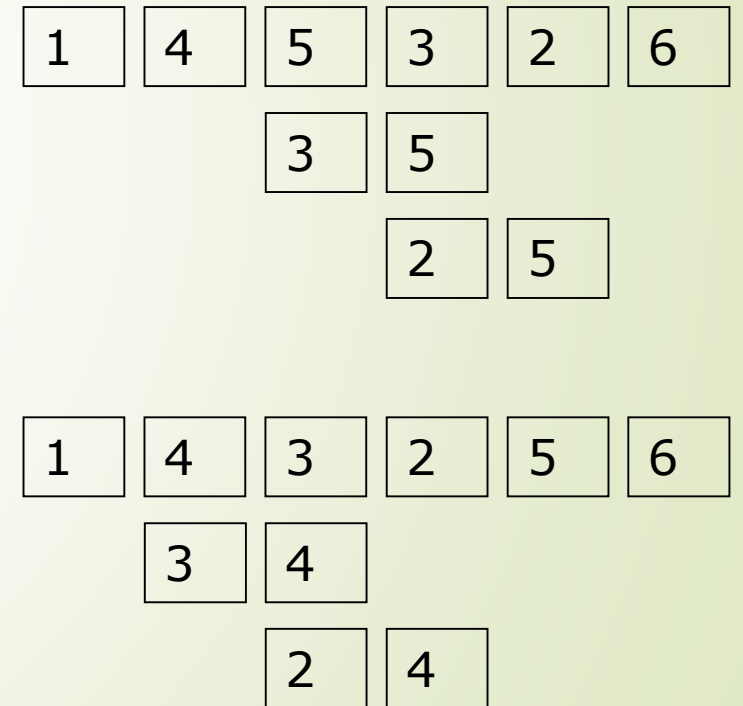
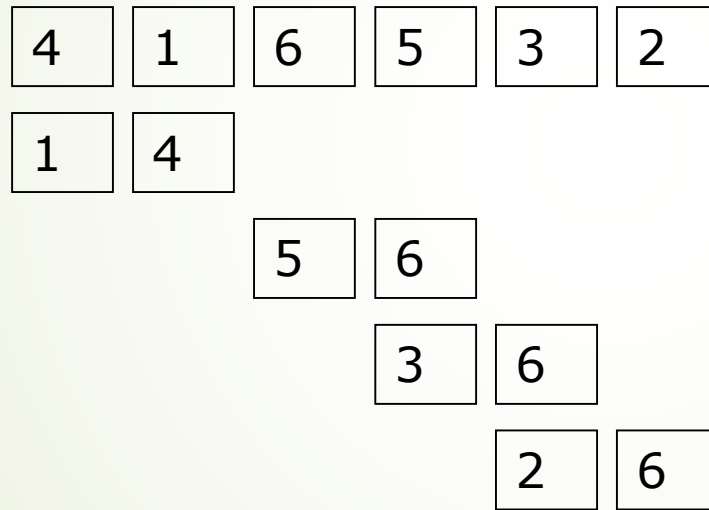
Простые алгоритмы сортировки

- Сортировка **выбором** – на каждой итерации находим максимальный элемент и меняем с последним



Простые алгоритмы сортировки

- Сортировка **обменом** – последовательно меняем местами элементы в паре, если их порядок неверен



Устойчивые сортировки

- ▶ Пузырьком $T=O(N^2)$, $R=O(1)$
- ▶ Вставками $T=O(N^2)$, $R=O(1)$
- ▶ Слиянием $T=O(N \log N)$, $R=O(N)$

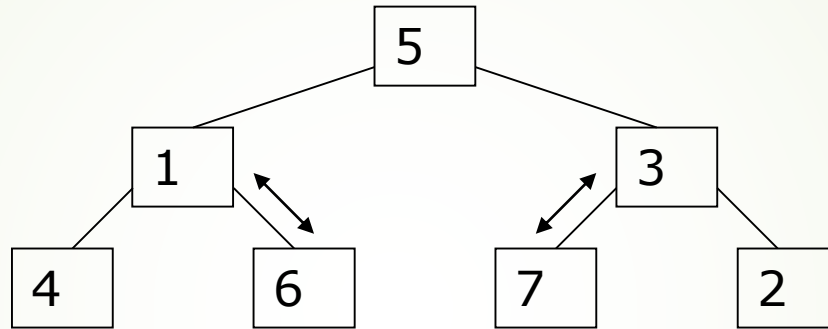
Неустойчивые сортировки

- ▶ Выбором $T=O(N^2)$, $R=O(1)$
 - ▶ Этот алгоритм уже на первом шаге меняет местами минимальный элемент в списке с первым, что и приводит к неустойчивости
 - ▶ Пример: $(2A, 2B, 1) \rightarrow (1, 2B, 2A)$ (считаем $2A == 2B$)

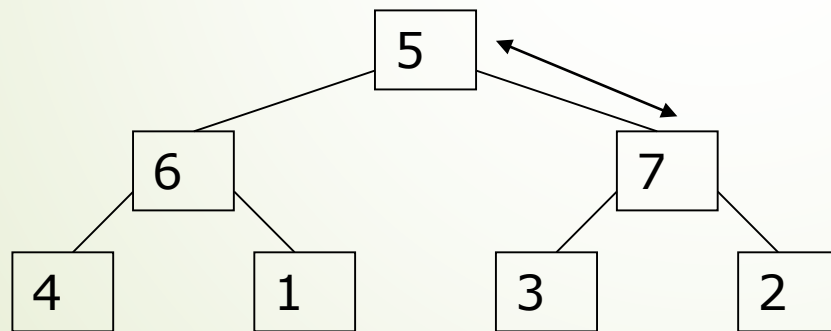
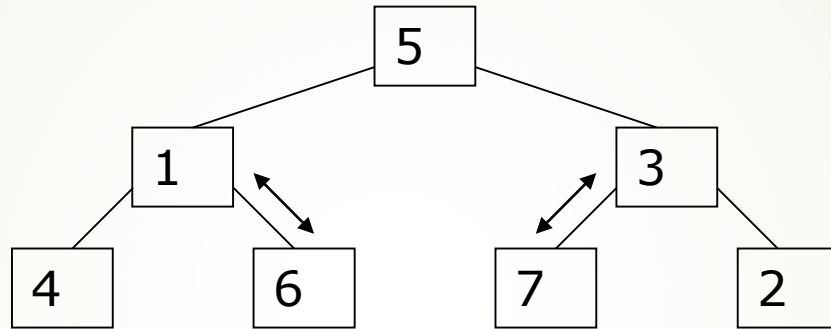
Сортировка двоичной кучей (вид бинарного дерева)

1. Подготовка (просеивание) – вершина дерева должна быть больше любого элемента в поддеревьях
2. Выбор – выкидываем вершину
3. Повтор – переходим к 1 с меньшим количеством вершин

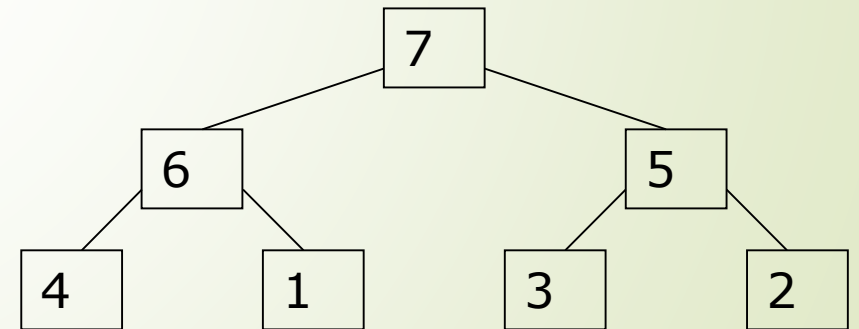
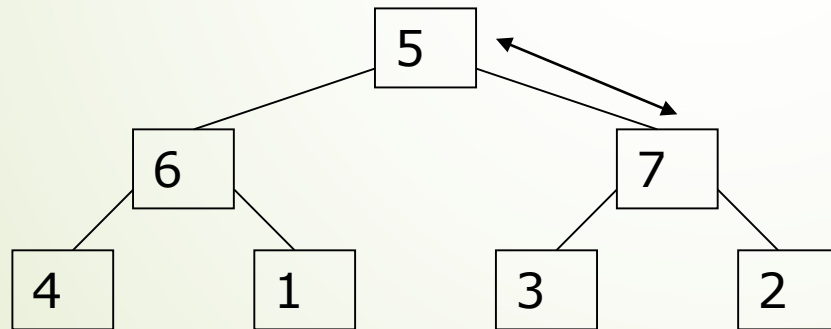
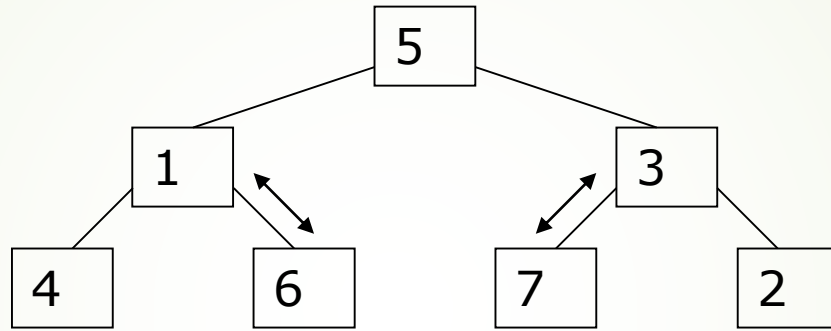
Сортировка двоичной кучей – подготовка



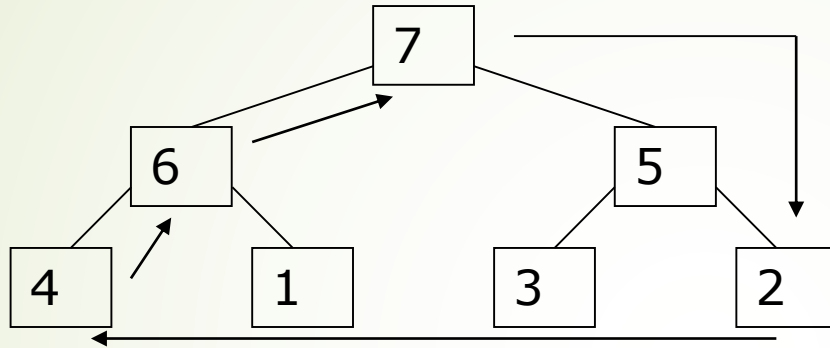
Сортировка двоичной кучей – подготовка



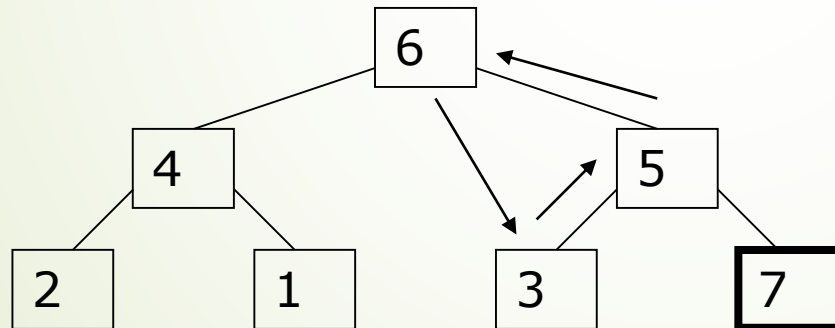
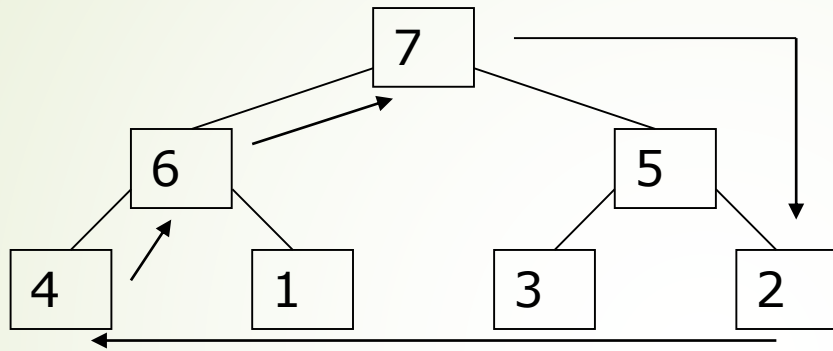
Сортировка двоичной кучей – подготовка



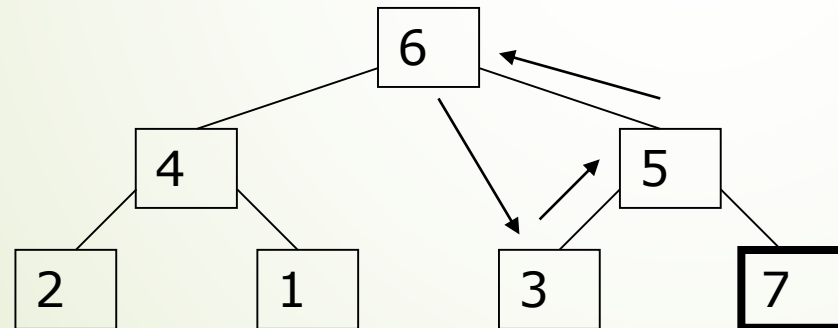
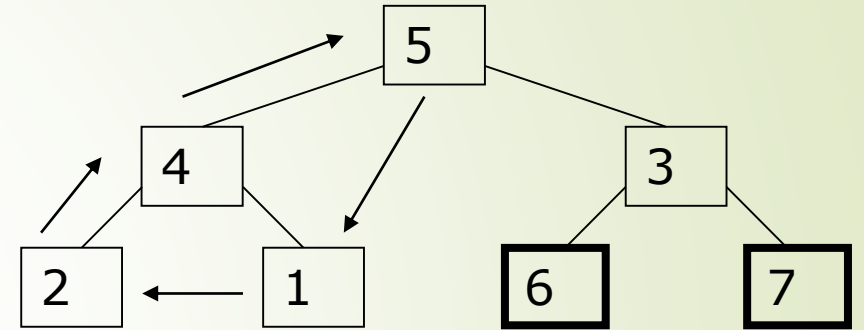
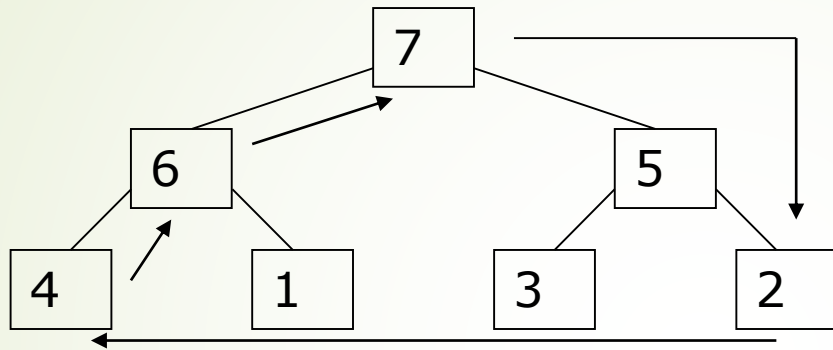
Сортировка двоичной кучей – выбор



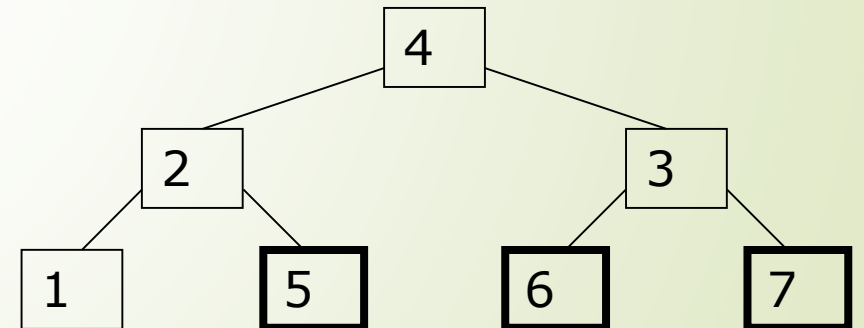
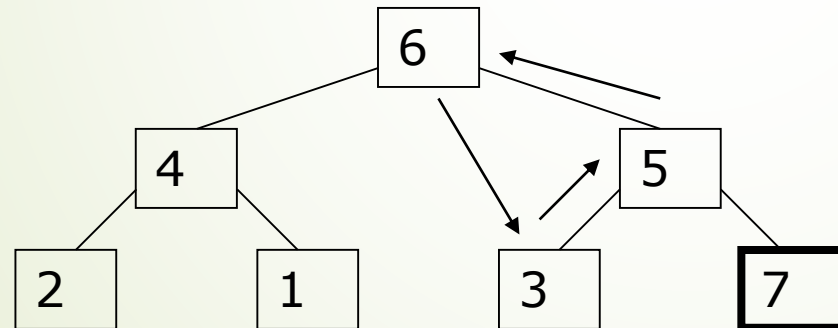
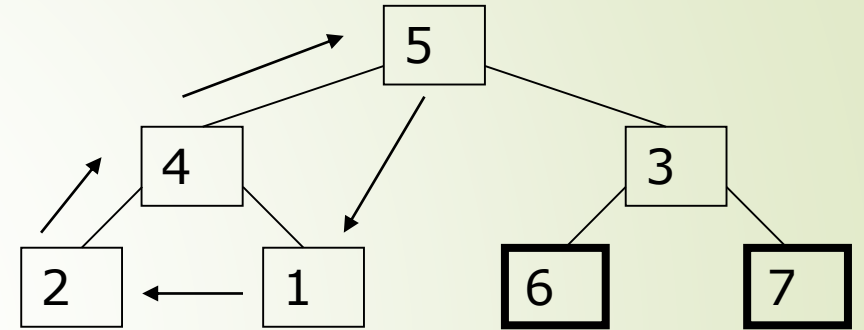
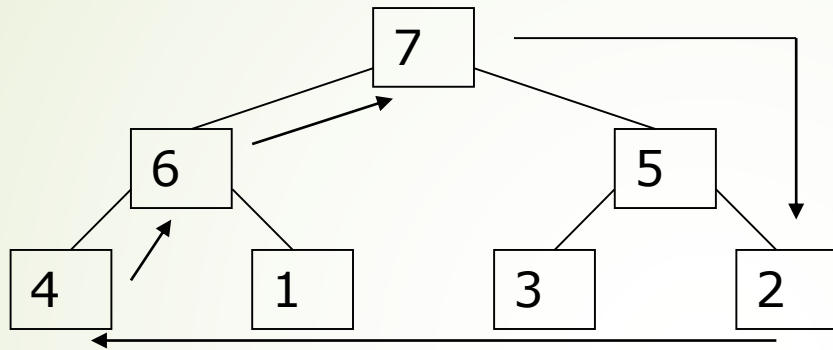
Сортировка двоичной кучей – выбор



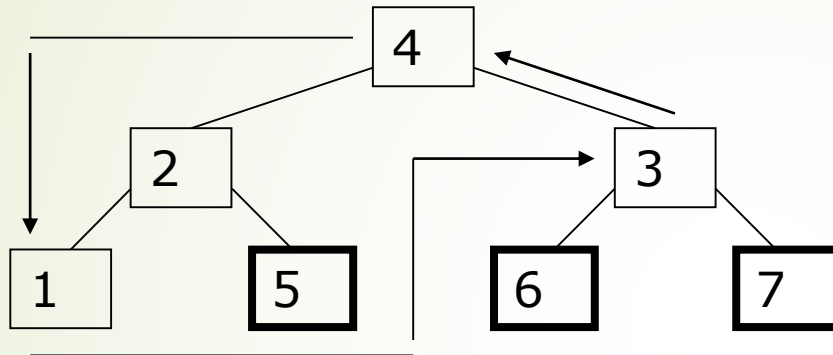
Сортировка двоичной кучей – выбор



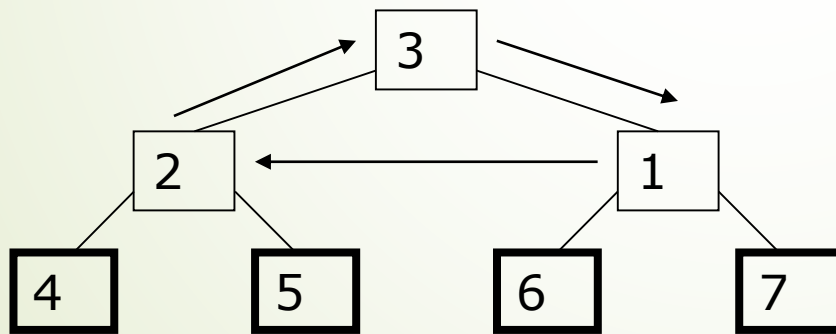
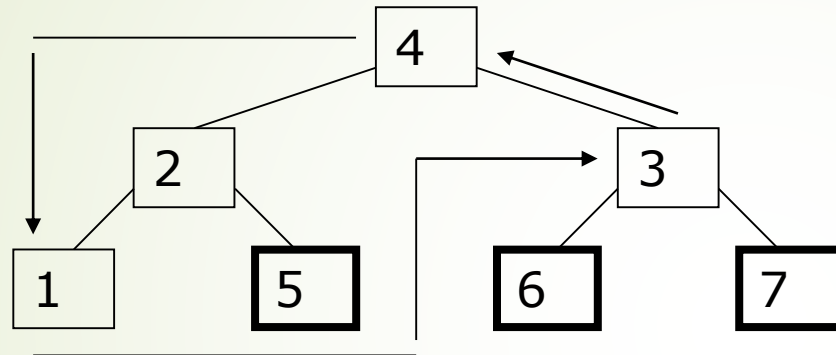
Сортировка двоичной кучей – выбор



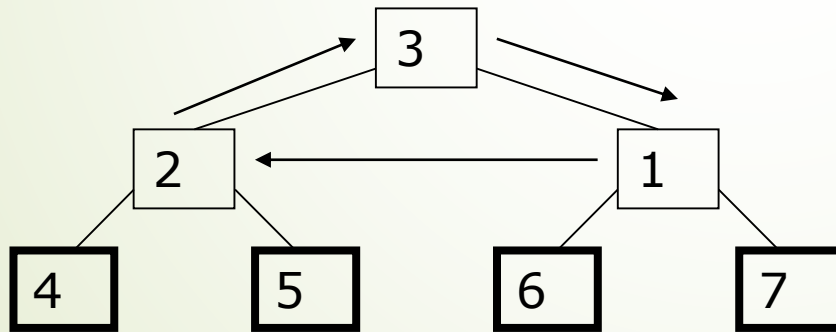
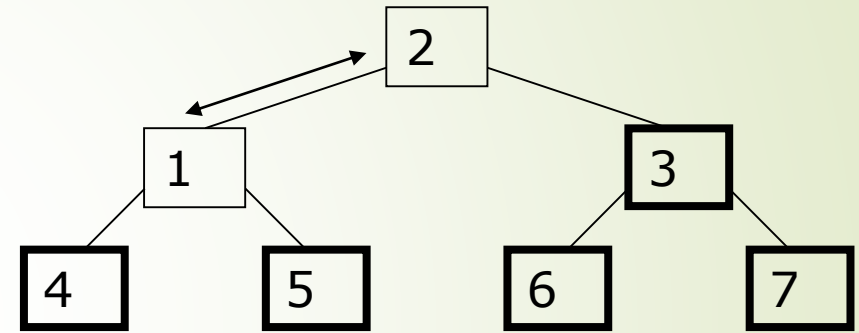
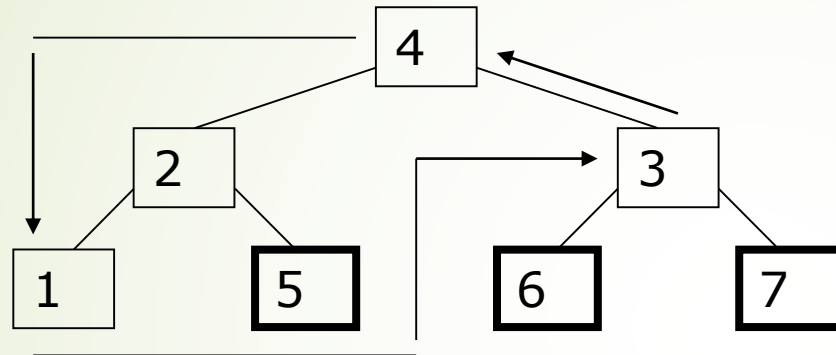
Сортировка двоичной кучей – выбор



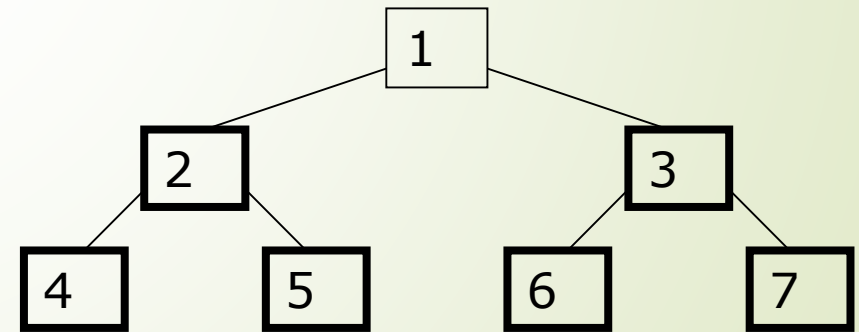
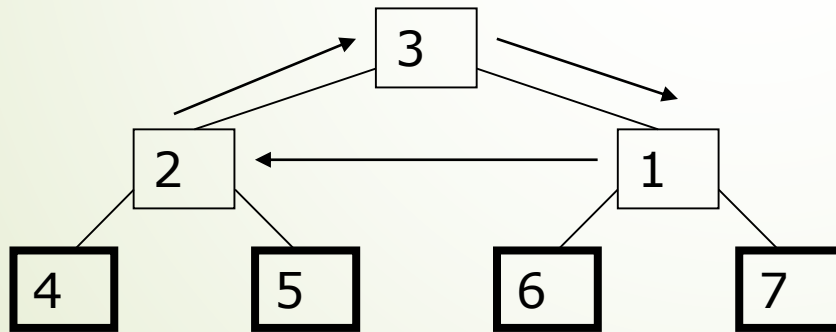
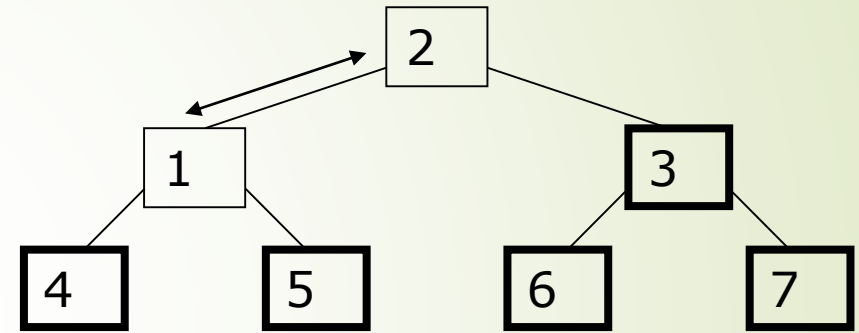
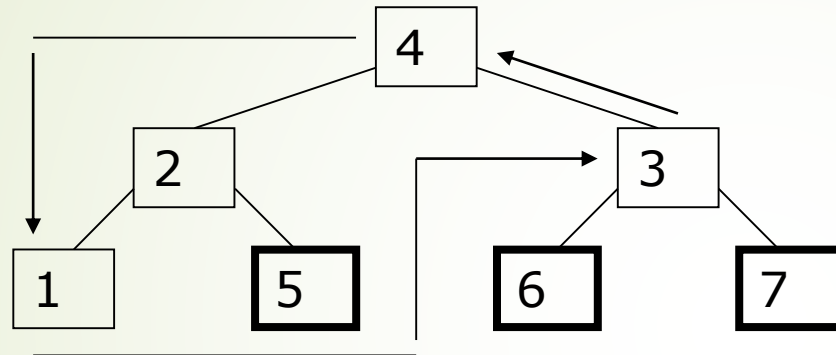
Сортировка двоичной кучей – выбор



Сортировка двоичной кучей – выбор

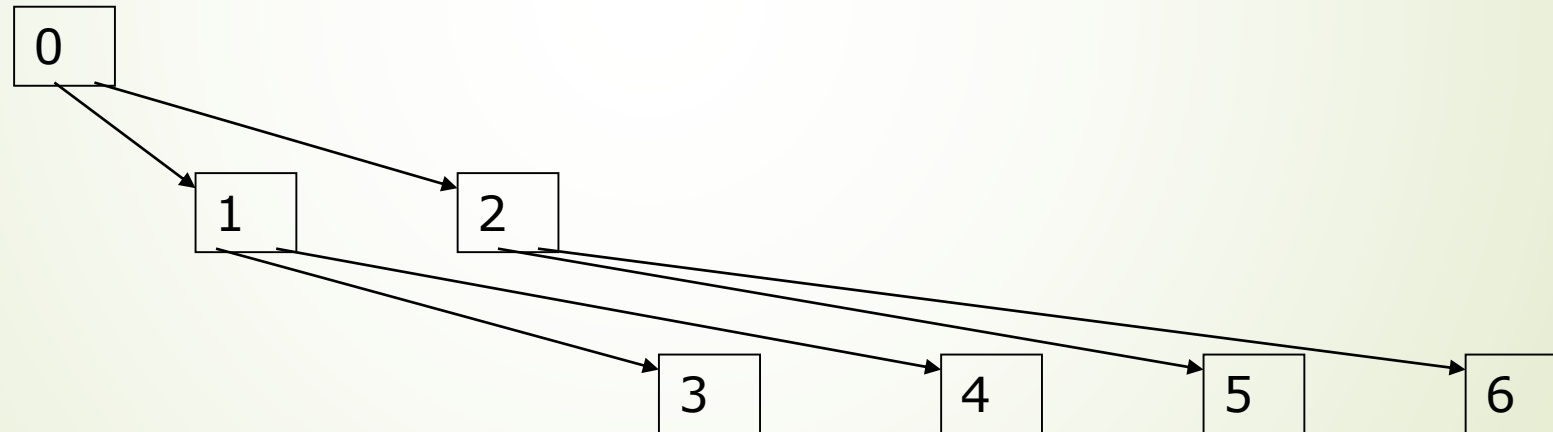


Сортировка двоичной кучей – выбор



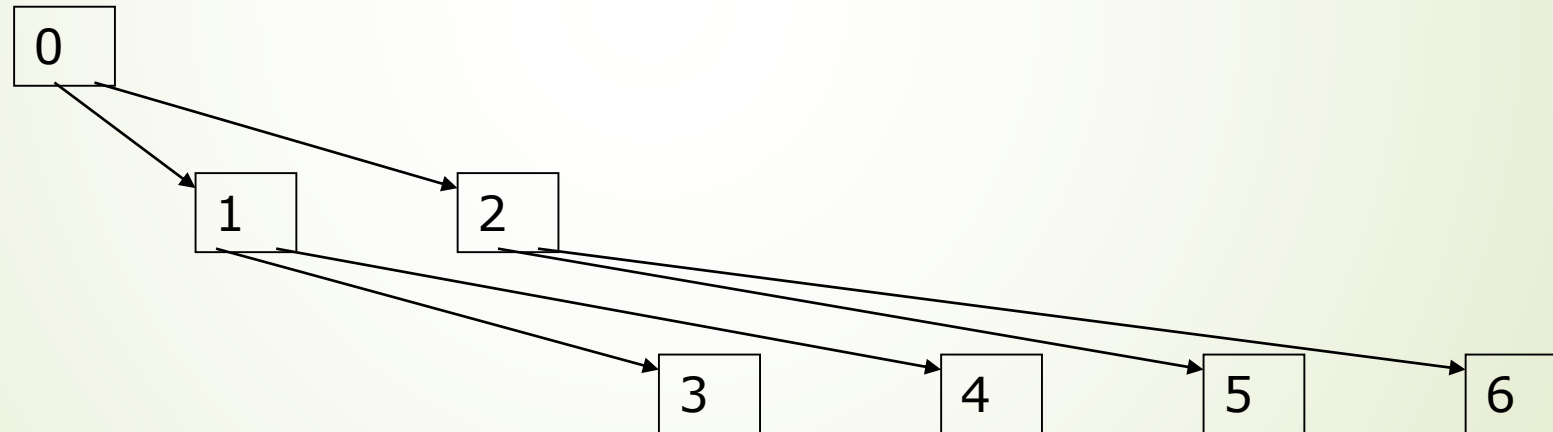
Организация двоичной кучи

- Корень бинарного дерева расположен в массиве по 0-му индексу, корни его поддеревьев – по 1-му и 2-му индексу, и так далее



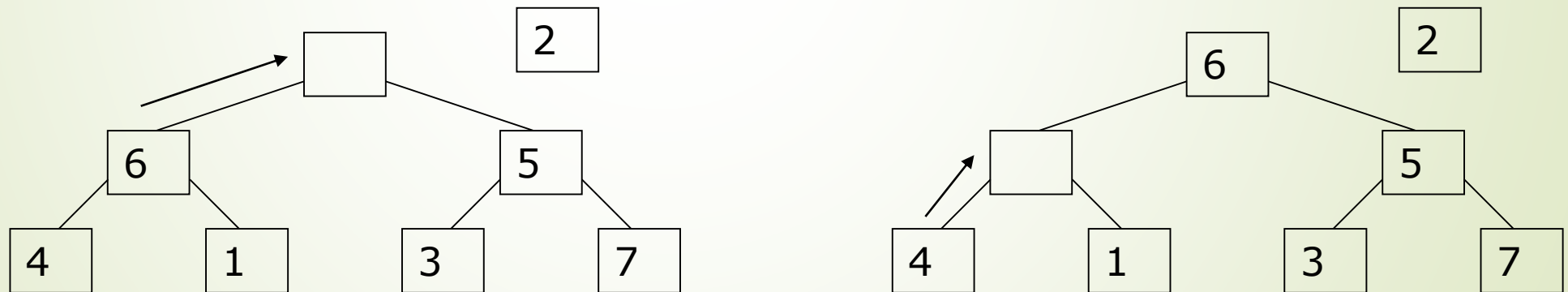
Организация двоичной кучи

- Если корень какого-либо поддерева имеет индекс N , то поддеревья нижнего уровня имеют индексы $2N+1$ и $2N+2$



Организация просеивания

- Корень запоминается
- Сравнивается с корнями поддеревьев; если меньше, на его место ставится один из корней поддеревьев
- Процедура повторяется для одного из поддеревьев



Псевдокод: просеивание

```
MAX-HEAPIFY (A, J, S) :  
    L = 2*J+1  
    R = 2*J+2  
    Max = J  
    if L < S and A[L] > A[Max]:  
        Max = L  
    if R < S and A[R] > A[Max]:  
        Max = R  
    if Max != J:  
        Swap A[J] with A[Max]  
        MAX-HEAPIFY(A, Max, S)
```

Псевдокод: подготовка кучи

BUILD-MAX-HEAP (A) :

```
for J = A.length / 2 - 1 downto 0:
```

```
    MAX-HEAPIFY (A, J, A.length)
```

- ▶ Трудоемкость: $O(N)$ просеиваний, каждое из которых имеет трудоемкость $O(\log N)$
- ▶ Корректность
 - ▶ Инвариант: перед каждой итерацией цикла узлы с номером больше J являются корнями бинарной пирамиды

Псевдокод: пирамидальная сортировка

HEAP-SORT (A) :

BUILD-MAX-HEAP (A)

for J = A.length - 1 downto 1

 Swap A[0] with A[J]

 MAX-HEAPIFY (A, 0, J)

- Трудоемкость: $O(N)$ просеиваний, каждое из которых имеет трудоемкость $O(\log N)$
- Корректность: следует из того, что после Swap только корень пирамиды нарушает её свойство, и из инварианта

Неустойчивые сортировки

- ▶ Пирамидальная сортировка (сортировка двоичной кучей, Heap Sort) $T=O(N \log N)$, $R=O(1)$
 - ▶ Неустойчива примерно по тем же причинам – вершина в процессе сортировки «уезжает» в некоторое место кучи

Быстрая сортировка

- ▶ Используется принцип декомпозиции «Разделяй и Властвуй»
- ▶ На каждом шаге массив $A[\text{Min} \dots \text{Max}]$ путём перестановки элементов разбивается на два подмассива $A[\text{Min} \dots R]$ и $A[R+1 \dots \text{Max}]$, таких, что
 - ▶ $A[J] \leq A[R]$ для $J \leq R$
 - ▶ $A[J] \geq A[R]$ для $J > R$
- ▶ Каждый из подмассивов сортируется рекурсивно

Сортировка Хоара

- ▶ На каждом шаге алгоритма выбирается **разделяющий элемент** массива (в идеале – его **медиана**), после чего элементы переставляются так, чтобы меньшие оказались слева, а большие – справа

4	1	6	5	7	3	2
---	---	---	---	---	---	---

4	1	2	5	7	3	6
---	---	---	---	---	---	---

4	1	2	3	7	5	6
---	---	---	---	---	---	---

Сортировка Хоара

- ▶ На каждом шаге алгоритма выбирается **разделяющий элемент** массива (в идеале – его **медиана**), после чего элементы переставляются так, чтобы меньшие оказались слева, а большие – справа

4	1	6	5	7	3	2
----------	---	---	---	---	---	---

2	1	6	5	7	3	4
---	---	----------	---	---	----------	----------

2	1	3		5	7	6	4
---	---	----------	--	----------	---	---	----------

Псевдокод: разбиение

```
PARTITION(A, Min, Max) :  
  X = A[RANDOM(Min, Max)]  
  L=Min, R=Max  
  while L <= R:  
    while A[L] < X:  
      L++  
    while A[R] > X:  
      R--  
    if (L <= R):  
      Swap A[L] with A[R]  
      L++, R--
```

- **Инвариант:** в начале каждой итерации элементы #Min...#L-1 $\leq X$, #R+1...#Max $\geq X$

Псевдокод: быстрая сортировка

```
QUICK-SORT (A, Min, Max) :  
  if (Min < Max) :  
    R = PARTITION (A, Min, Max)  
    QUICK-SORT (A, Min, R)  
    QUICK-SORT (A, R+1, MAX)
```

Производительность быстрой сортировки

- ▶ Наихудший случай: $R == \text{Min}$ или $R == \text{Max}$
 - ▶ $T(N) = T(N-1) + O(N)$
- ▶ Наилучший случай: $R == (\text{Min} + \text{Max}) / 2$
 - ▶ $T(N) = 2T(N/2) + O(N)$
- ▶ 20 / 80: $R = 0.2\text{Min} + 0.8\text{Max}$
 - ▶ $T(N) = T(0.8N) + T(0.2N) + O(N)$
- ▶ «Средний» ~ чередование наилучшего и наихудшего

Сортировки сравнениями

- ▶ Трудоёмкость в худшем случае $O(N \log N)$ или хуже
- ▶ Бинарное дерево решений:
 - ▶ Внутренние узлы – сравнения между двумя элементами
 - ▶ Листья – всевозможные перестановки списка (их $N!$)
 - ▶ Отсюда минимальное число сравнений $\lg(N!) = O(N \log N)$
- ▶ Тем не менее, существуют сортировки за линейное время...

Сортировки за линейное время

- Все сортировки за линейное время основаны не на сравнениях и предполагают какие-то дополнительные требования к исходным данным
 - Сортировка подсчётом
 - Поразрядная сортировка
 - Карманная сортировка

Сортировка подсчётом

- ▶ Работает для целых чисел в интервале от 0 до K , где $K = O(N)$
 - ▶ Также работает для данных, сводимых к таким целым числам, например ...

Сортировка подсчётом

- ▶ Работает для целых чисел в интервале от 0 до K , где $K = O(N)$
 - ▶ Также работает для данных, сводимых к таким целым числам, например для элементов перечислений
- ▶ Трудоёмкость $O(N)$, ресурсоёмкость $O(N)$, устойчива

Сортировка подсчётом

- ▶ Работает для целых чисел в интервале от 0 до K , где $K = O(N)$
 - ▶ Также работает для данных, сводимых к таким целым числам, например для элементов перечислений
- ▶ Трудоёмкость $O(N)$, ресурсоёмкость $O(N)$, устойчива
- ▶ Идея
 - ▶ Вначале подсчитать, сколько в списке целых чисел, равных J (для всех J от 0 до K): $\text{EqCount}(J)$

Сортировка подсчётом

- ▶ Работает для целых чисел в интервале от 0 до K , где $K = O(N)$
 - ▶ Также работает для данных, сводимых к таким целым числам, например для элементов перечислений
- ▶ Трудоёмкость $O(N)$, ресурсоёмкость $O(N)$, устойчива
- ▶ Идея
 - ▶ Вначале подсчитать, сколько в списке целых чисел, равных J (для всех J от 0 до K): $EqCount(J)$
 - ▶ Потом подсчитать, сколько в списке целых чисел, меньших J (опять-таки для всех J): $LessCount(J)$

Сортировка подсчётом

- ▶ Работает для целых чисел в интервале от 0 до K , где $K = O(N)$
 - ▶ Также работает для данных, сводимых к таким целым числам, например для элементов перечислений
- ▶ Трудоёмкость $O(N)$, ресурсоёмкость $O(N)$, устойчива
- ▶ Идея
 - ▶ Вначале подсчитать, сколько в списке целых чисел, равных J (для всех J от 0 до K): $EqCount(J)$
 - ▶ Потом подсчитать, сколько в списке целых чисел, меньших J (опять-таки для всех J): $LessCount(J)$
 - ▶ Затем мы размещаем числа, равные J , по индексу $LessCount(J)$ и далее

Сортировка подсчётом

```
COUNTING-SORT(In, Out, K):  
  for J = 0 to K:                // clear  
    Count[J] = 0  
  for J = 0 to In.length - 1: // Count equals  
    Count[In[J]] ++  
  for J = 1 to K:                // Count less or equals  
    Count[J] += Count[J-1]  
  for J = In.length - 1 downto 0:  
    Out[Count[In[J]] - 1] = In[J]  
    Count[In[J]]--
```

Карманная сортировка

- Она же – корзинная (bucket sort)
- Предполагает, что мы имеем числа, распределенные равномерно в некотором интервале
- Трудоёмкость $O(N)$, ресурсоёмкость $O(N)$
- Идея
 - Разбить интервал на N карманов равного размера
 - Распределить числа по карманам в соответствии с их значениями, получив $O(1)$ чисел в каждом кармане
 - Отсортировать числа в каждом кармане отдельно любым простым способом, например, сортировкой вставками
 - Соединить карманы

Карманная сортировка

```
N = In.length
let B: array of lists
for J = 0 to N - 1:
    B[J] = emptyList()
for J = 0 to N - 1:
    K = floor(N*In[J])
    B[K] += In[J]
Out = emptyList()
for J = 0 to N - 1:
    SORT(B[K])
    Out += B[K]
```

Итоги

- Рассмотрены
 - Простые и сложные сортировки
 - Характеристики сортировок: трудоёмкость, ресурсоёмкость, устойчивость
 - Показана нижняя граница трудоёмкости для сортировок, основанных на сравнениях
 - Сортировки за линейное время
- Далее
 - Простые структуры данных