Алгоритмы и структуры данных

Лекция 4. Алгоритмы сортировки (прочие).

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

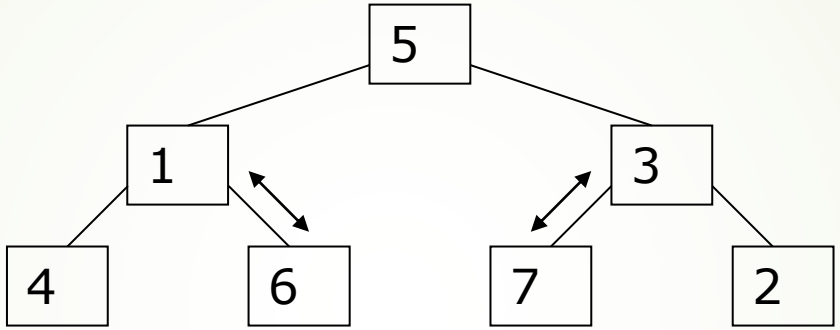
Сложные сортировки

- Сортировка слиянием
- Быстрая сортировка (сортировка Хоара)
- **Сортировка двоичной кучей (пирамидальная)**

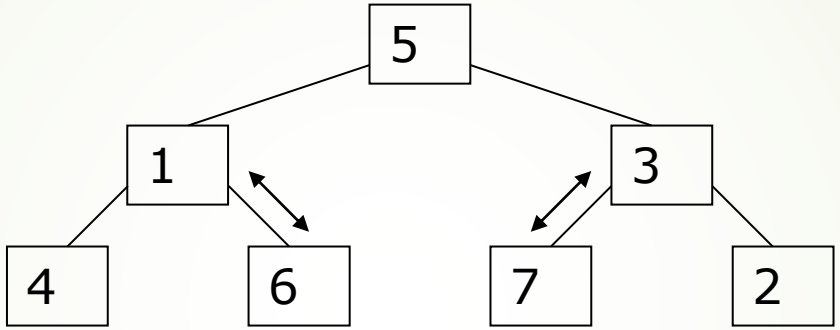
Сортировка двоичной кучей (вид бинарного дерева)

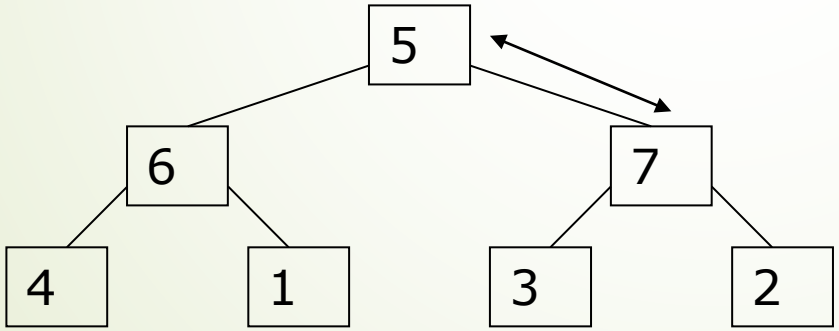
1. Подготовка (просеивание) – вершина дерева должна быть больше любого элемента в поддеревьях
2. Выбор – выкидываем вершину
3. Повтор – переходим к 1 с меньшим количеством вершин

Сортировка двоичной кучей – подготовка

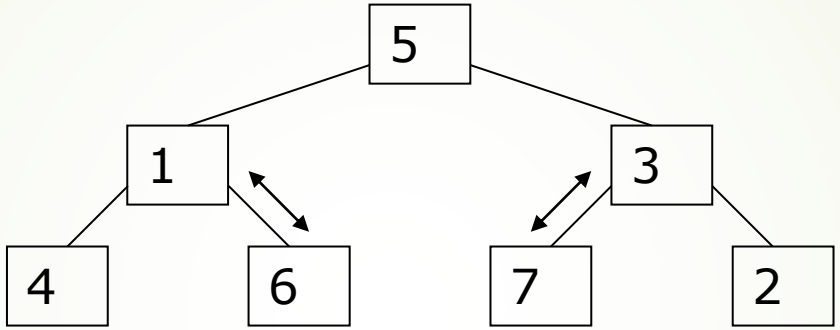


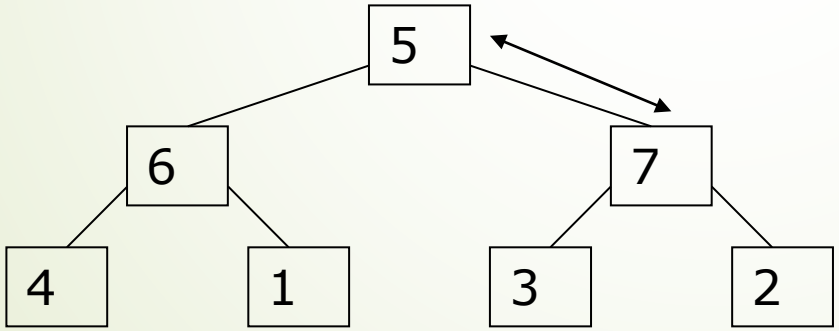
Сортировка двоичной кучей – подготовка

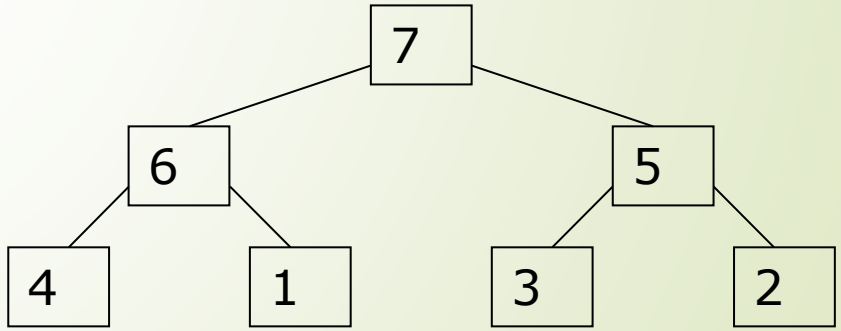




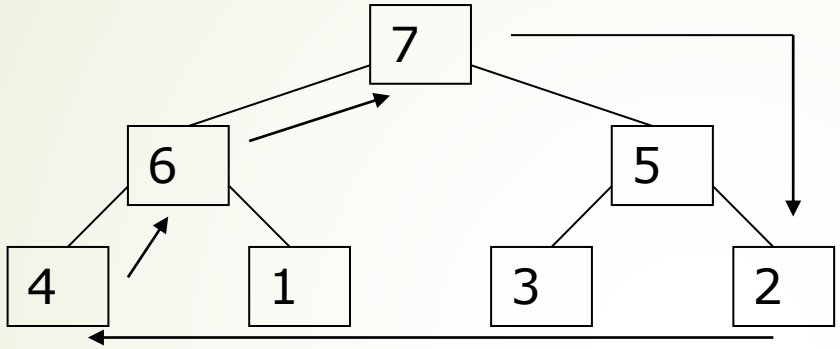
Сортировка двоичной кучей – подготовка



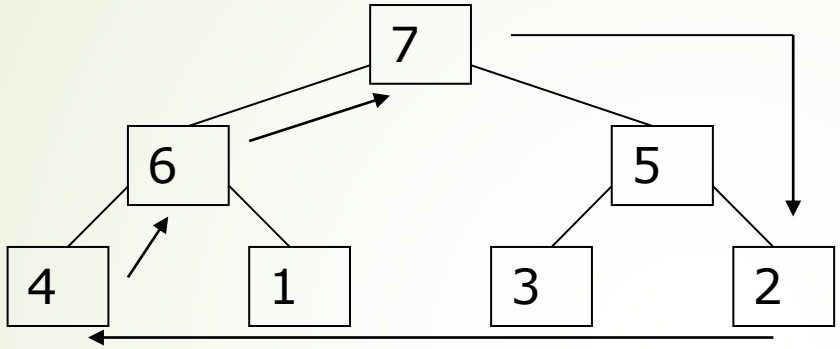


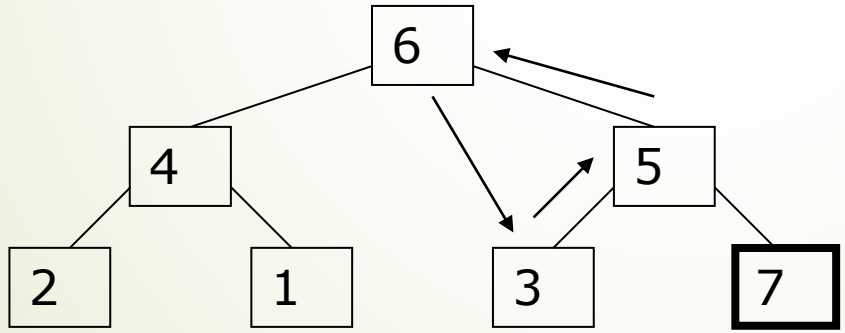


Сортировка двоичной кучей – выбор

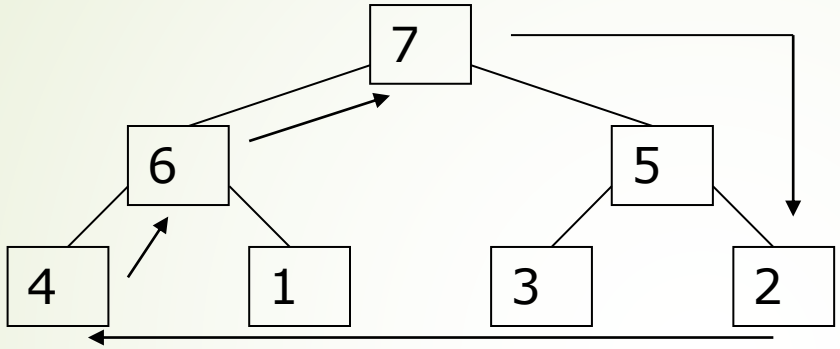


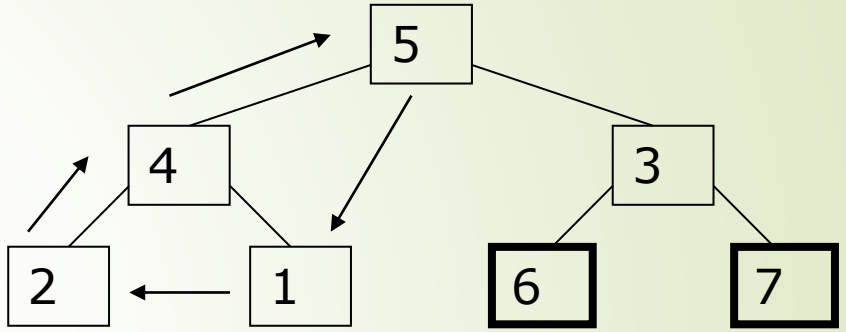
Сортировка двоичной кучей – выбор

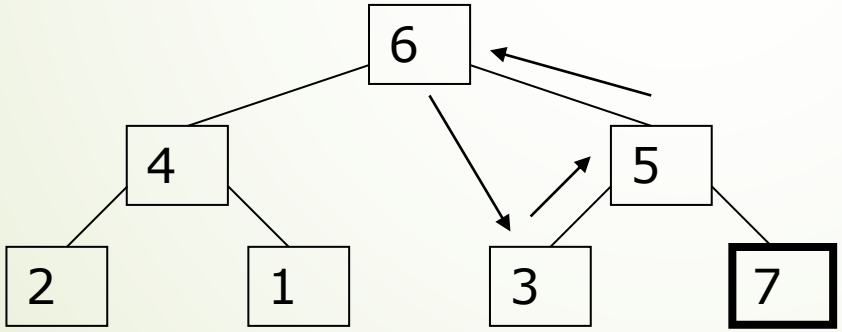




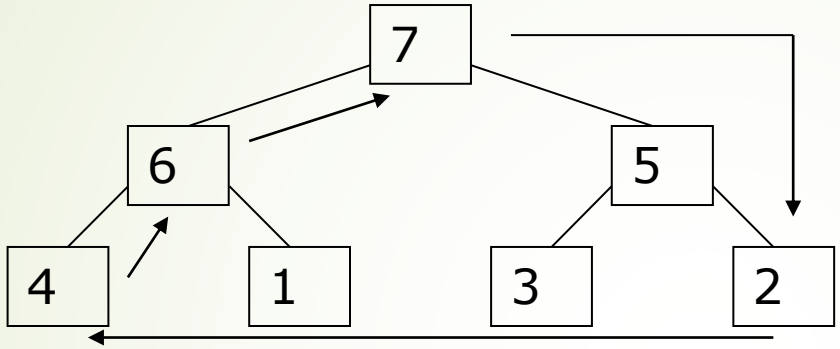
Сортировка двоичной кучей – выбор

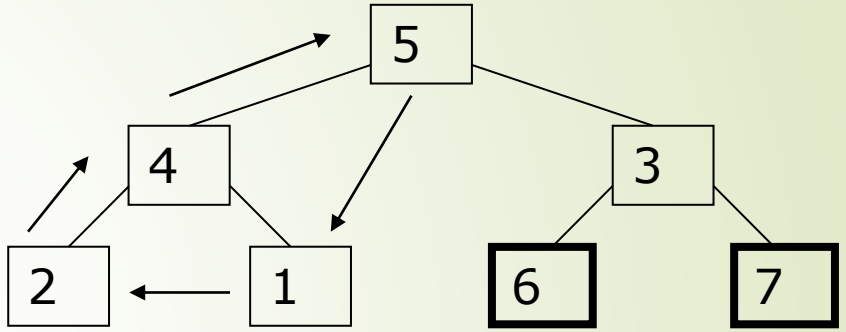


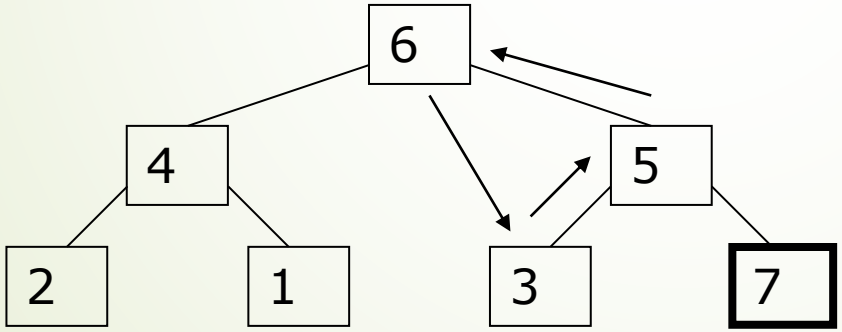


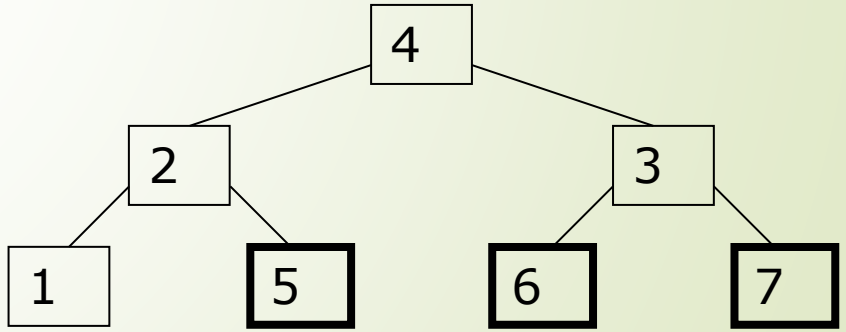


Сортировка двоичной кучей – выбор

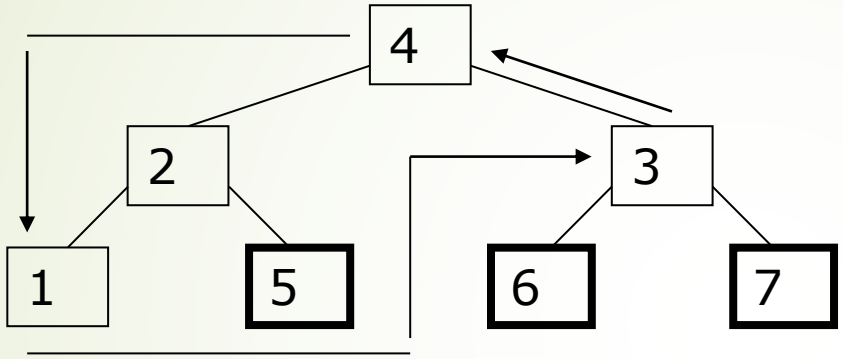




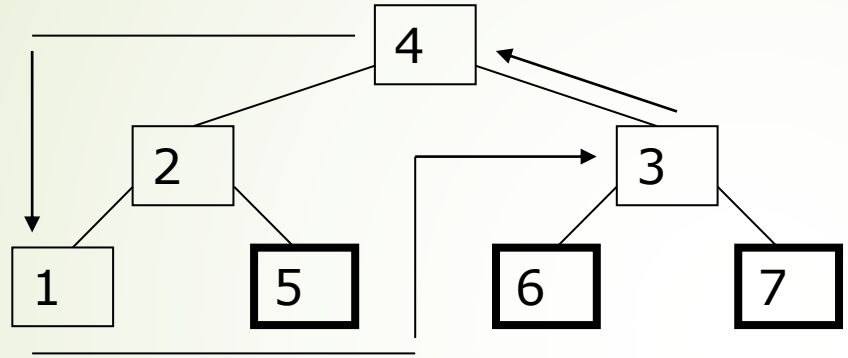


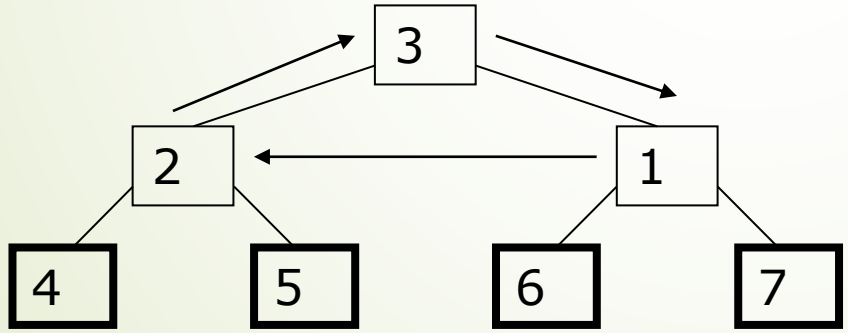


Сортировка двоичной кучей – выбор

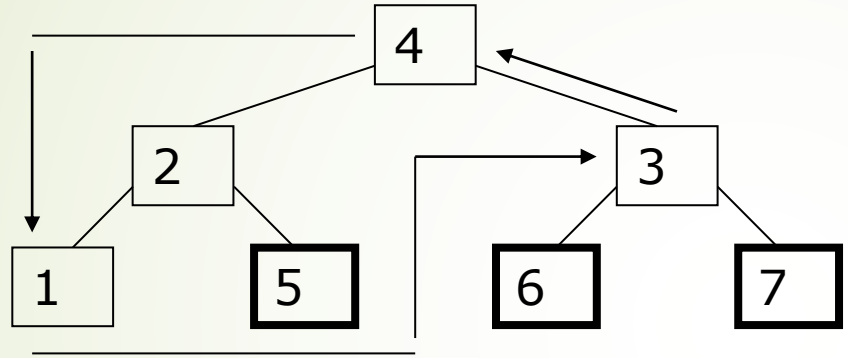


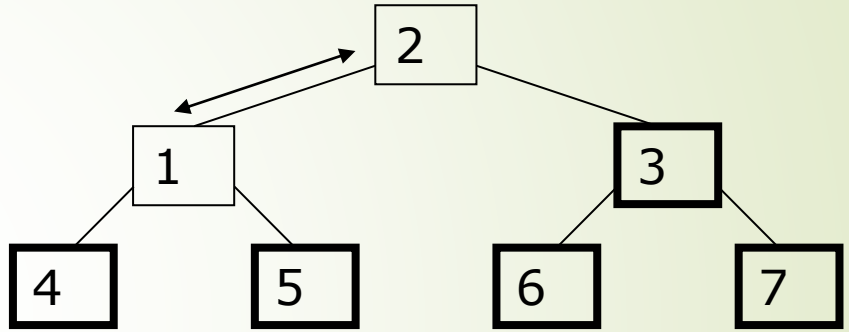
Сортировка двоичной кучей – выбор

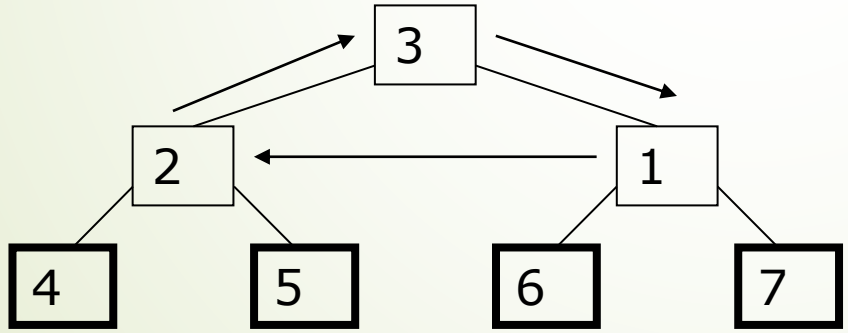




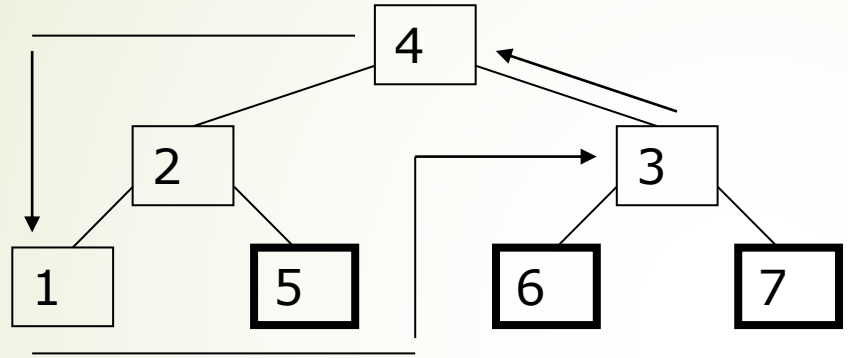
Сортировка двоичной кучей – выбор

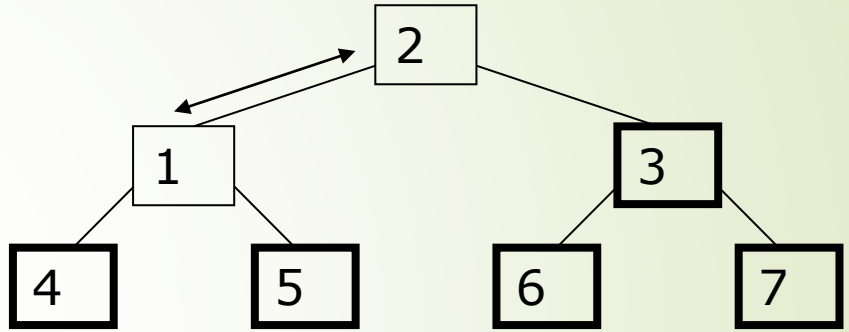


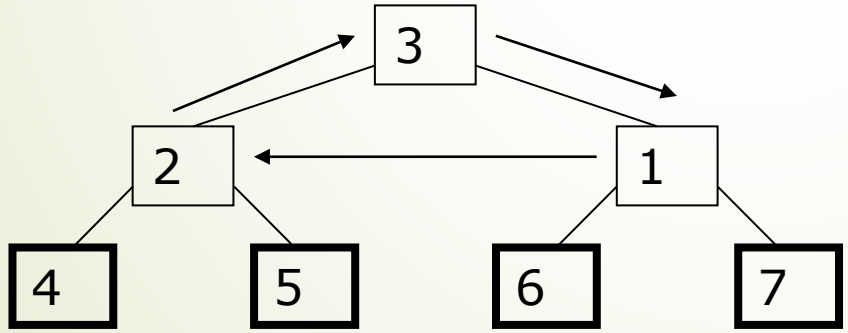


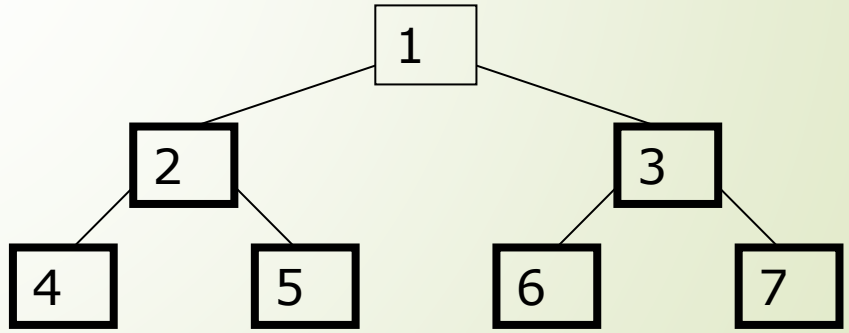


Сортировка двоичной кучей – выбор



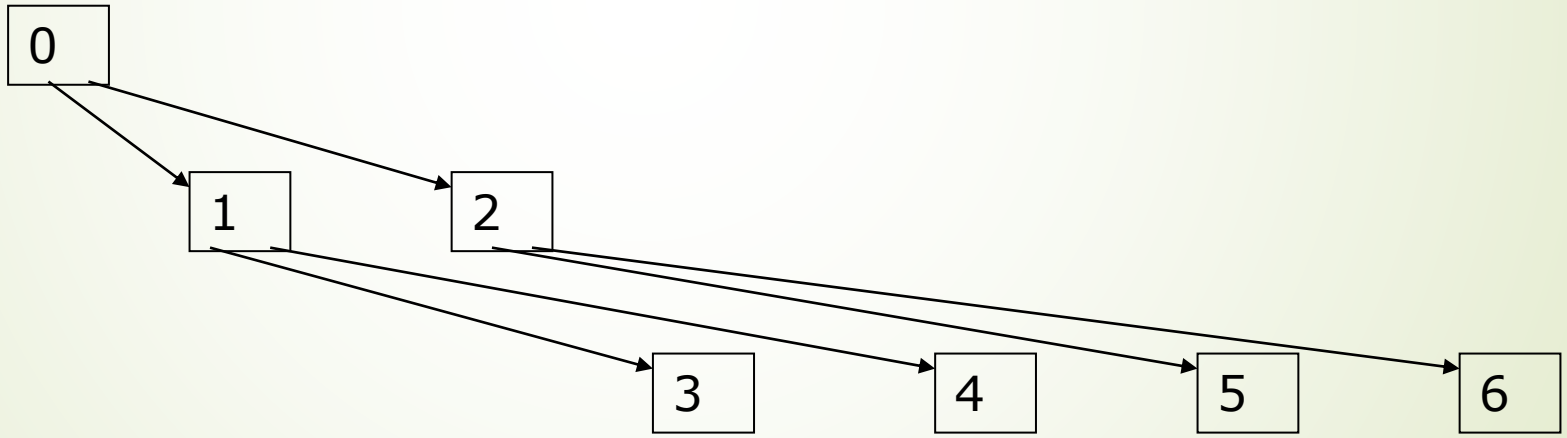






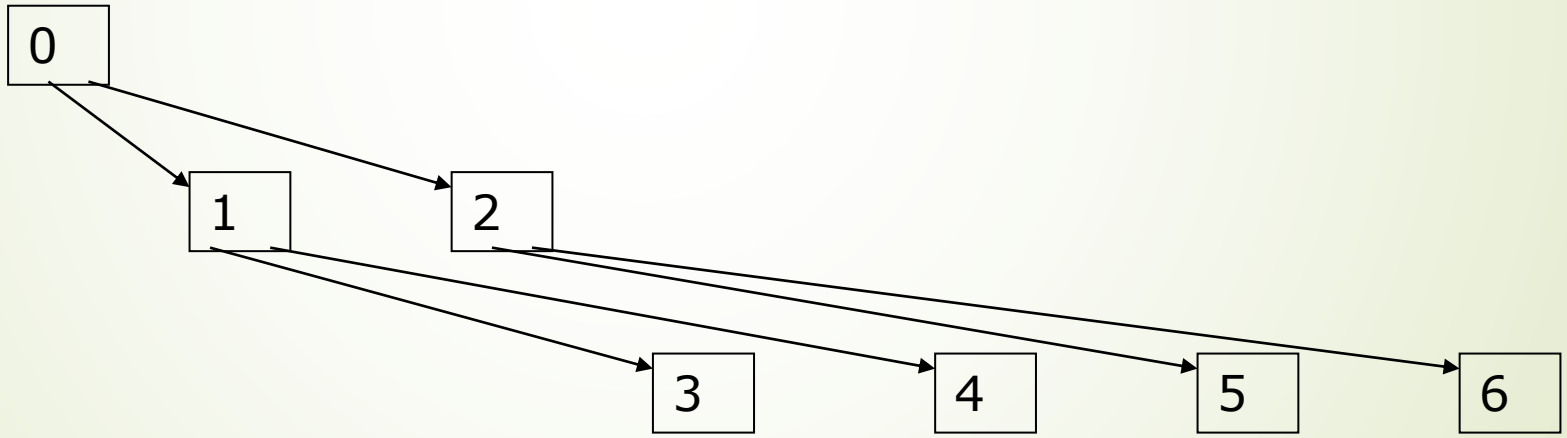
Организация двоичной кучи

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



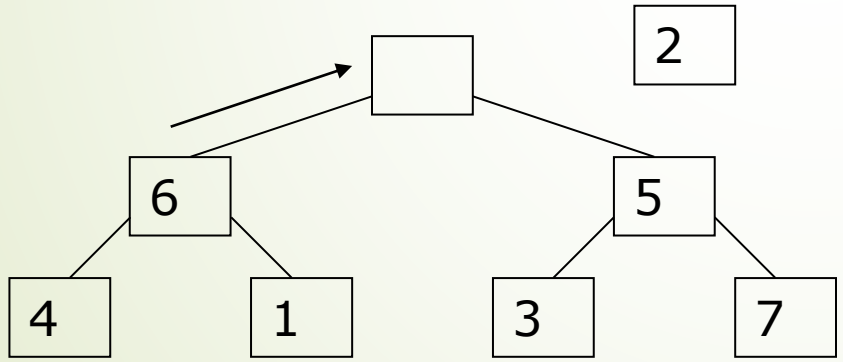
Организация двоичной кучи

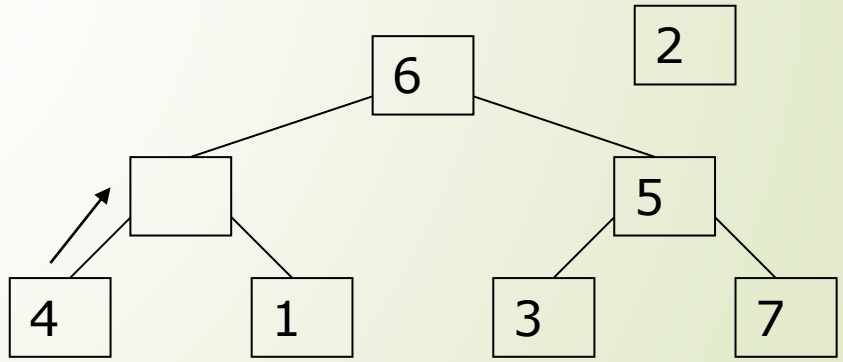
- Если корень какого-либо поддерева имеет индекс N, то поддеревья нижнего уровня имеют индексы 2N+1 и 2N+2



Организация просеивания

- Корень запоминается
- Сравнивается с корнями поддеревьев; если меньше, на его место ставится больший из корней поддеревьев
- Процедура повторяется для поддерева с большим корнем





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)
```

- Трудоёмкость: O(N) просеиваний, каждое из которых имеет трудоёмкость O(logN)
- 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)
```

- Трудоёмкость: O(N) просеиваний, каждое из которых имеет трудоёмкость O(logN)
- 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 logN), R=O(1)
  - Неустойчива примерно по тем же причинам – вершина в процессе сортировки «уезжает» в некоторое место кучи

Сортировки сравнениями

- Трудоёмкость в худшем случае O(N logN) 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 logN)
- Тем не менее, существуют сортировки за линейное время…

Отступление: готовые функции

- Java
  - Collections.sort(someList)

Отступление: готовые функции

- Java
  - Collections.sort(someList)
- Kotlin
  - mutableList.sort()
  - list.sorted() // не на месте

Отступление: готовые функции

- Java
  - Collections.sort(someList)
  - Arrays.sort(someArray)
- Kotlin
  - mutableList.sort()
  - list.sorted() // не на месте
  - array.sort()

Отступление: готовые функции

- Java
  - Collections.sort(someList)
  - Arrays.sort(someArray)
- Kotlin
  - mutableList.sort()
  - list.sorted() // не на месте
  - array.sort()
- Реализация (во всех случаях)
  - На основе сортировки слияниями (merge sort)

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): 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