Курс «Алгоритмы и алгоритмические языки»
1 семестр 2016/2017

Лекция 21

Самоперестраивающиеся деревья (splay trees)

- Двоичное дерево поиска, не содержащее дополнительных служебных полей в структуре данных
  (нет баланса, цвета и т.п.)
- Гарантируется не логарифмическая сложность в худшем случае, а *амортизированная* логарифмическая сложность:
  - Любая последовательность из m словарных операций (поиска, вставки, удаления) над n элементами, *начиная с пустого дерева*, имеет сложность $O(m \log n)$
  - Средняя сложность одной операции $O(\log n)$
  - Некоторые операции могут иметь сложность $\Theta(n)$
  - Не делается предположений о распределении вероятностей ключей дерева и словарных операций (т.е. что некоторые операции выполнялись чаще других)
- Хорошее описание в:

  Harry R. Lewis, Larry Denenberg. Data Structures and Their Algorithms. HarperCollins, 1991. Глава 7.3.

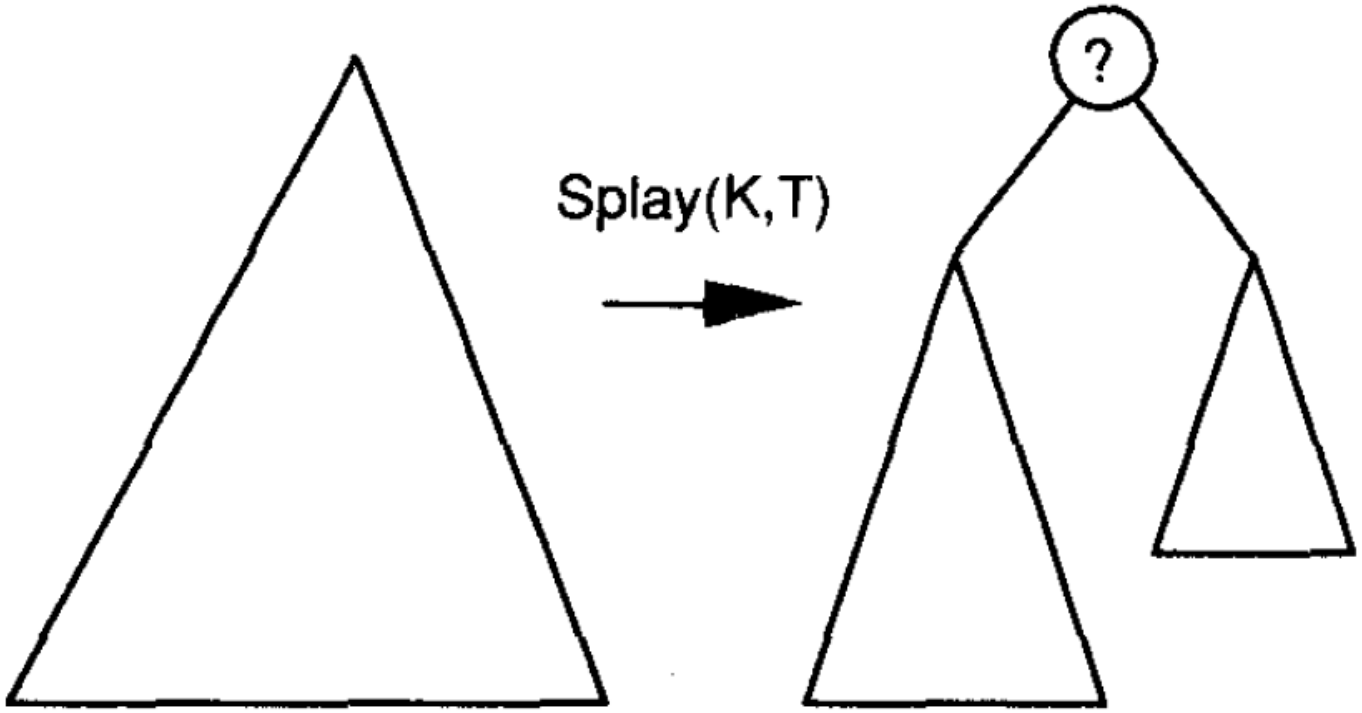
  http://www.amazon.com/Structures-Their-Algorithms-Harry-Lewis/dp/067339736X

Самоперестраивающиеся деревья (splay trees)

- Идея: эвристика Move-to-Front
  - Список: давайте при поиске элемента в списке перемещать найденный элемент в начало списка
  - Если он потребуется снова в обозримом будущем, он найдется быстрее
- Move-to-Front для двоичного дерева поиска: операция $Splay(K, T)$ (подравнивание, перемешивание, расширение)
  - После выполнения операции $Splay$ дерево T перестраивается (оставаясь деревом поиска) так, что:
  - Если ключ K есть в дереве, то он становится корнем
  - Если ключа K нет в дереве, то в корне оказывается его предшественник или последователь в симметричном порядке обхода

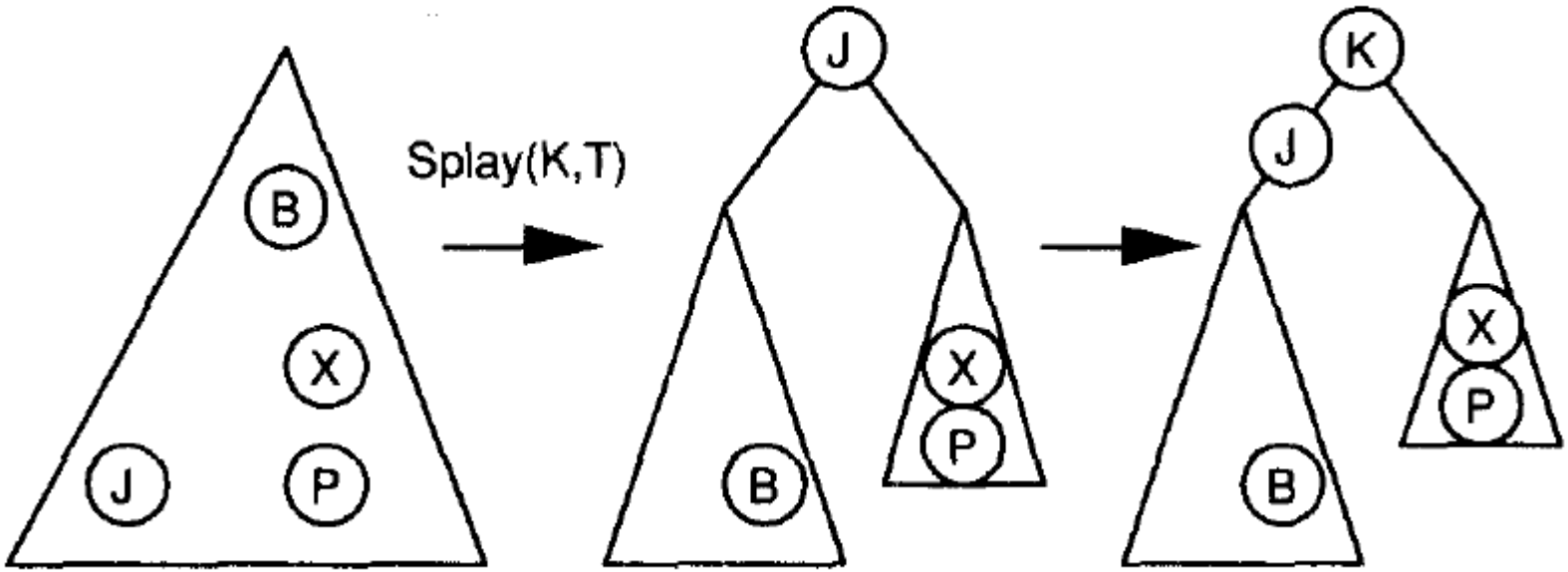
Реализация словарных операций через splay

- Поиск (LookUp): выполним операцию $Splay(K, T)$ и проверим значение ключа в корне:
  - если значение равно K, то ключ найден



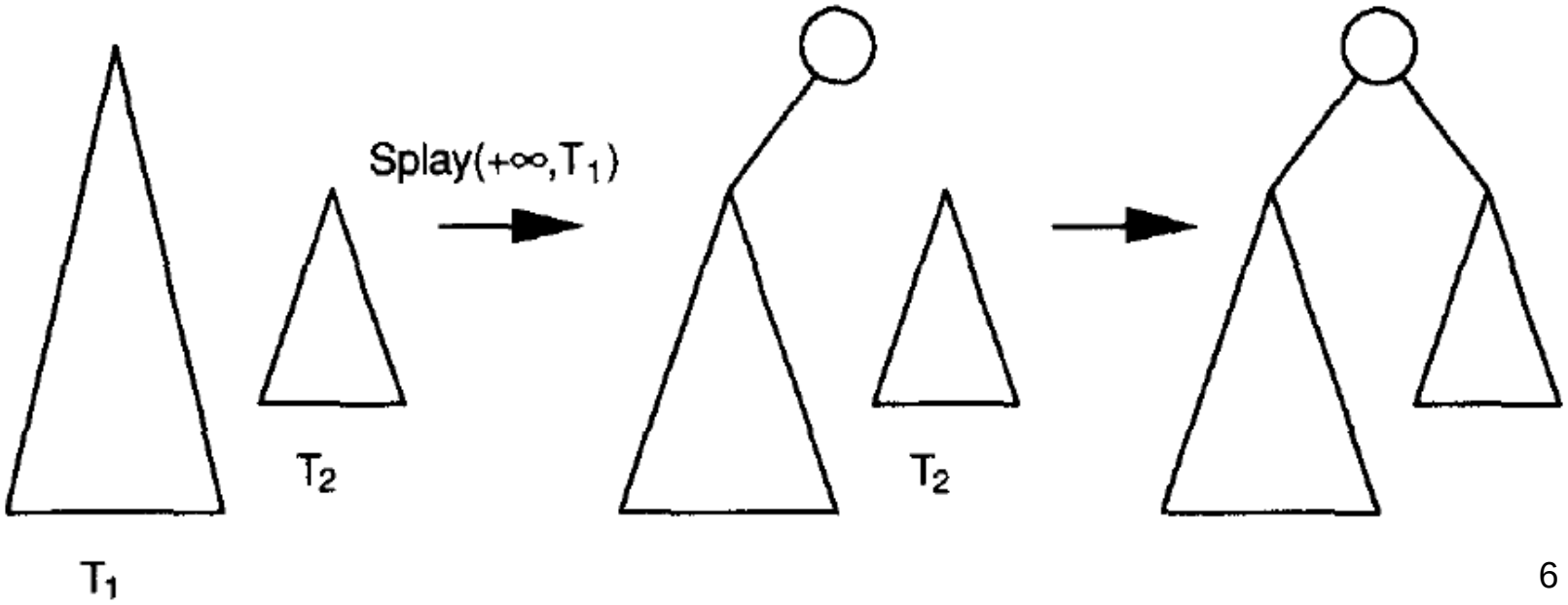
Реализация словарных операций через splay

- Вставка (Insert): выполним операцию $Splay(K, T)$ и проверим значение ключа в корне:
  - если значение уже равно K, то обновим данные ключа
  - если значение другое, то вставим новый корень K и поместим старый корень J слева или справа (в зависимости от значения J)



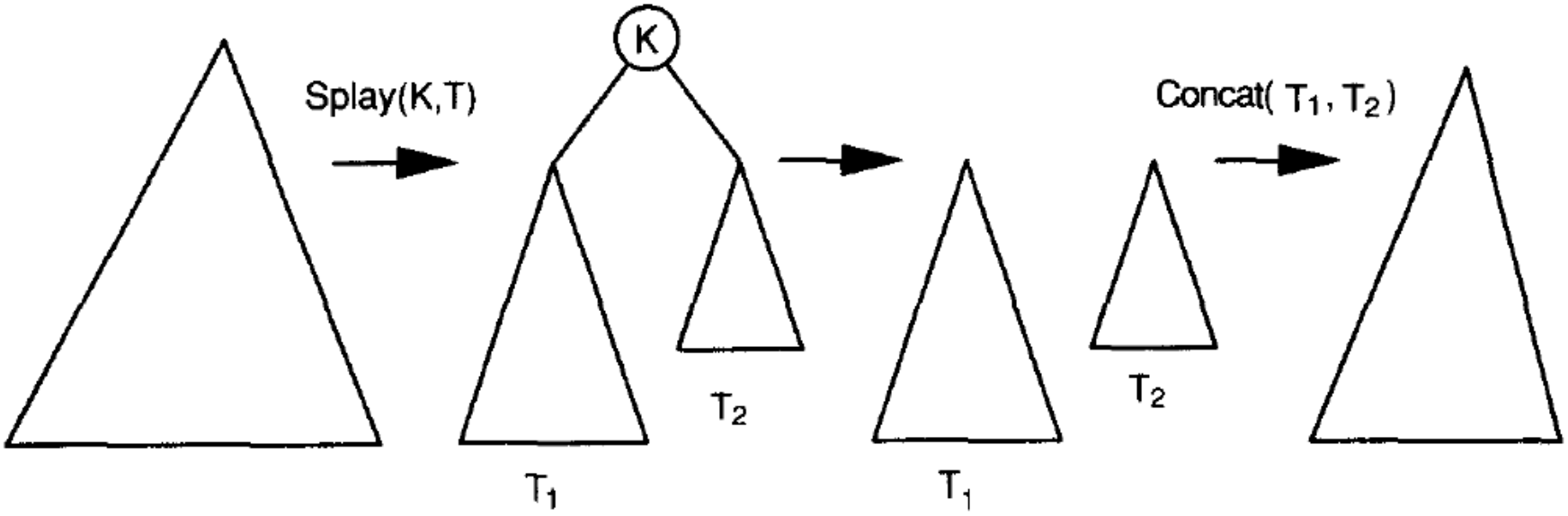
Реализация словарных операций через splay

- Операция $Concat\ (T_1, T_2)$ – слияние деревьев поиска T_1 и T_2 таких, что **все** ключи в дереве T_1 **меньше**, чем **все** ключи в дереве T_2, в одно дерево поиска
- Слияние (Concat): выполним операцию $Splay(+\infty, T_1)$ со значением ключа, заведомо больше любого другого в T_1
  - После $Splay(+\infty, T_1)$ у корня дерева T_1 нет правого сына
  - Присоединим дерево T_2 как правый сын корня T_1



Реализация словарных операций через splay

- Удаление (Delete): выполним операцию $Splay(K, T)$ и проверим значение ключа в корне:
  - если значение **не равно** K, то ключа в дереве нет и удалять нам нечего
  - иначе (ключ был найден) выполним операцию *Concat* над левым и правым сыновьями корня, а корень удалим

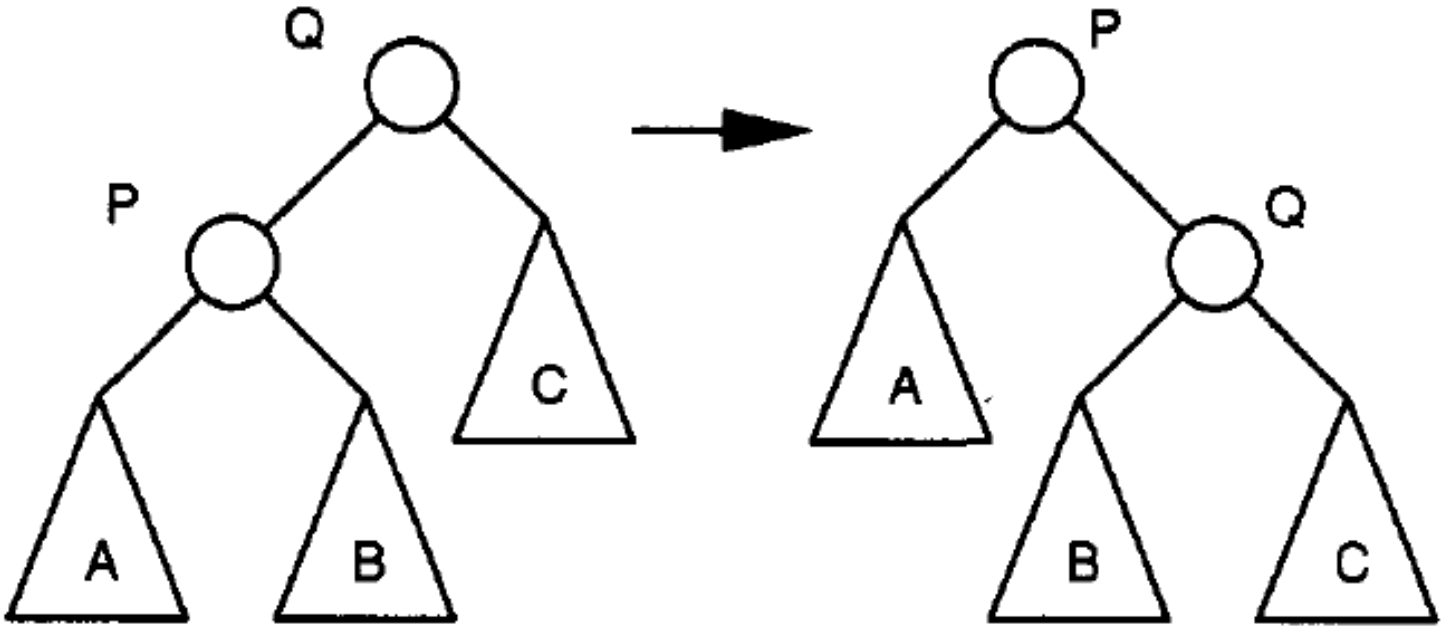


Реализация операции splay

- Шаг 1: ищем ключ K в дереве обычным способом, запоминая пройденный путь по дереву
  - Может потребоваться память, линейная от количества узлов дерева
  - Для уменьшения количества памяти можно воспользоваться *инверсией ссылок* (link inversion)
    - перенаправление указателей на сына назад на родителя вдоль пути по дереву плюс 1 бит на обозначение направления
- Шаг 2: получаем указатель P на узел дерева либо с ключом K, либо с его соседом в симметричном порядке обхода, на котором закончился поиск (сосед имеет единственного сына)
- Шаг 3: возвращаемся назад вдоль запомненного пути, перемещая узел P к корню (узел P будет новым корнем)

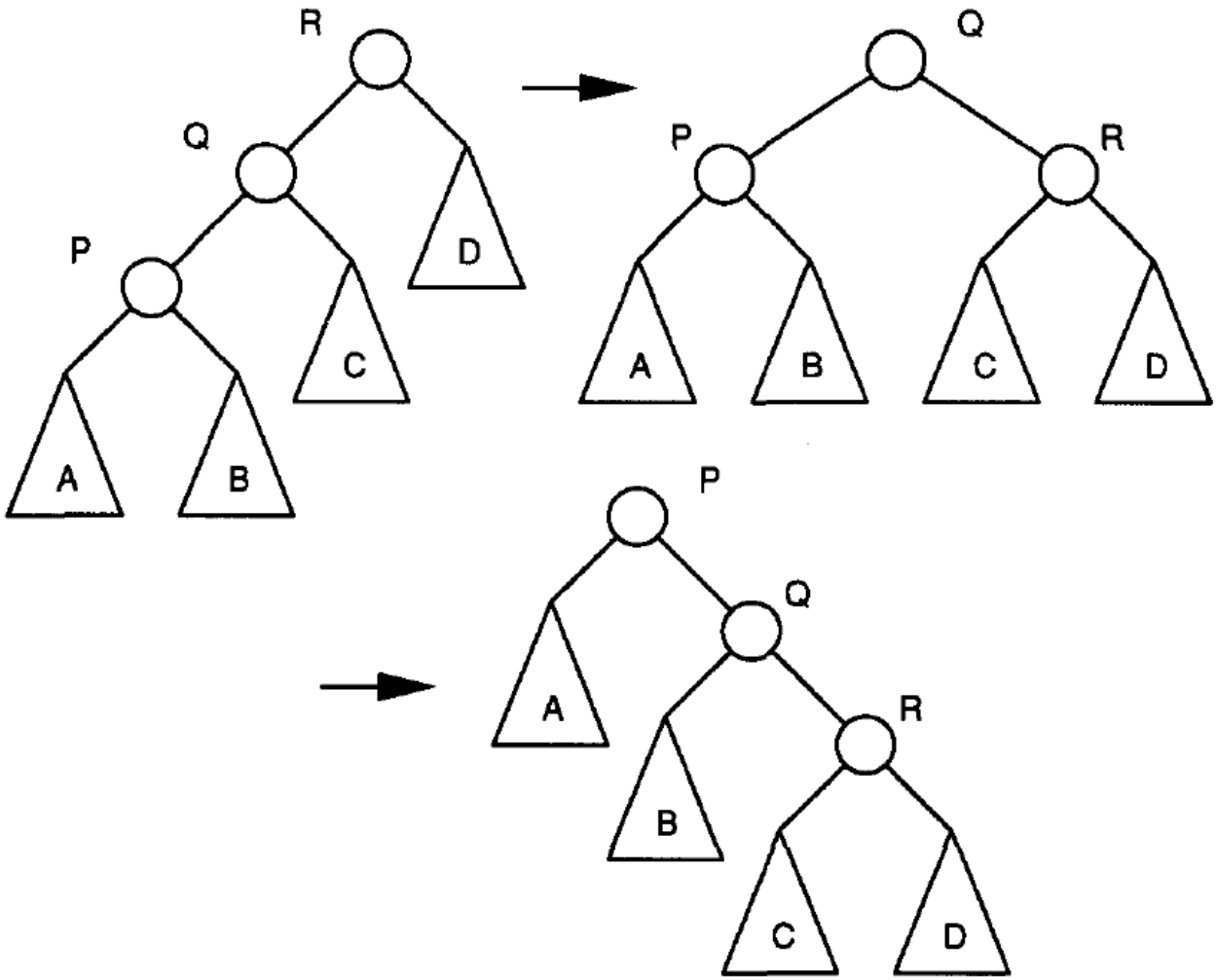
Реализация операции splay

- Шаг 3а): отец узла P – корень дерева (или у P нет деда)
  - выполняем однократный поворот налево или направо



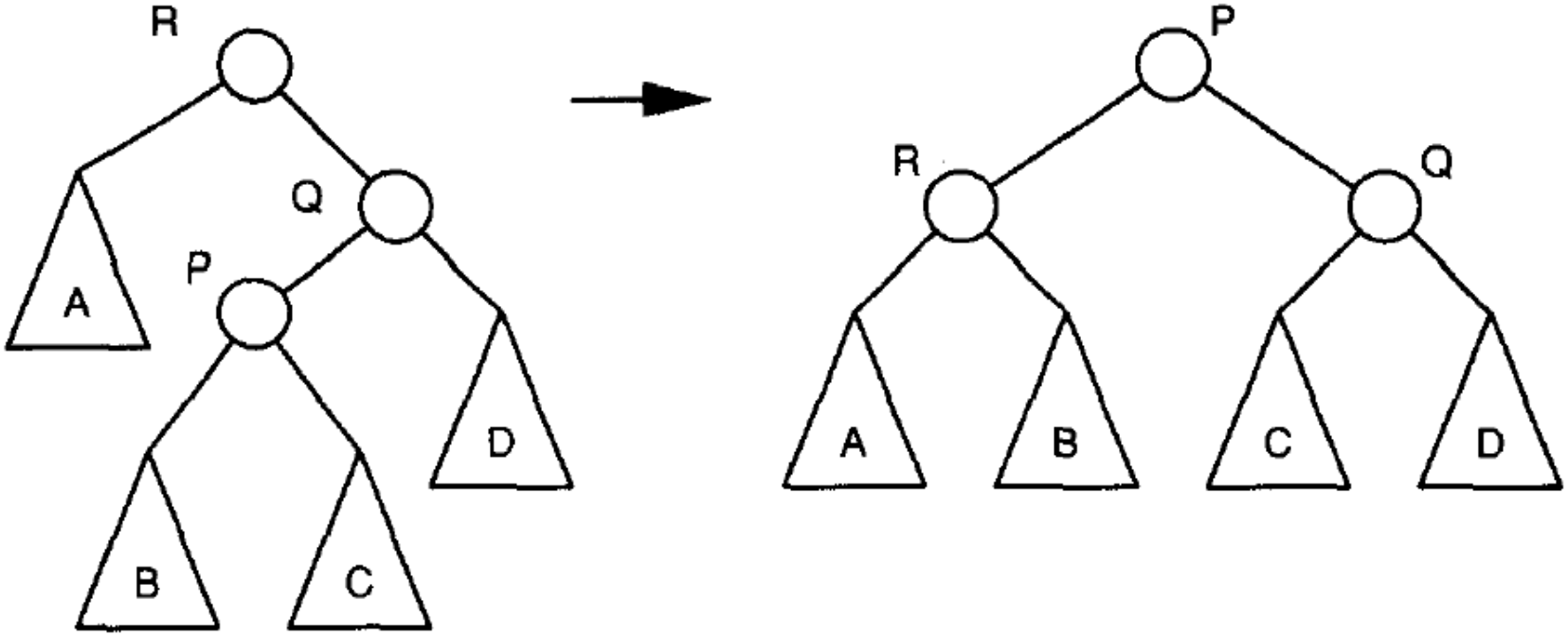
Реализация операции splay

- Шаг 3б): узел P и отец узла P – оба левые или правые дети
  - выполняем два однократных поворота направо (налево), сначала вокруг деда P, потом вокруг отца P



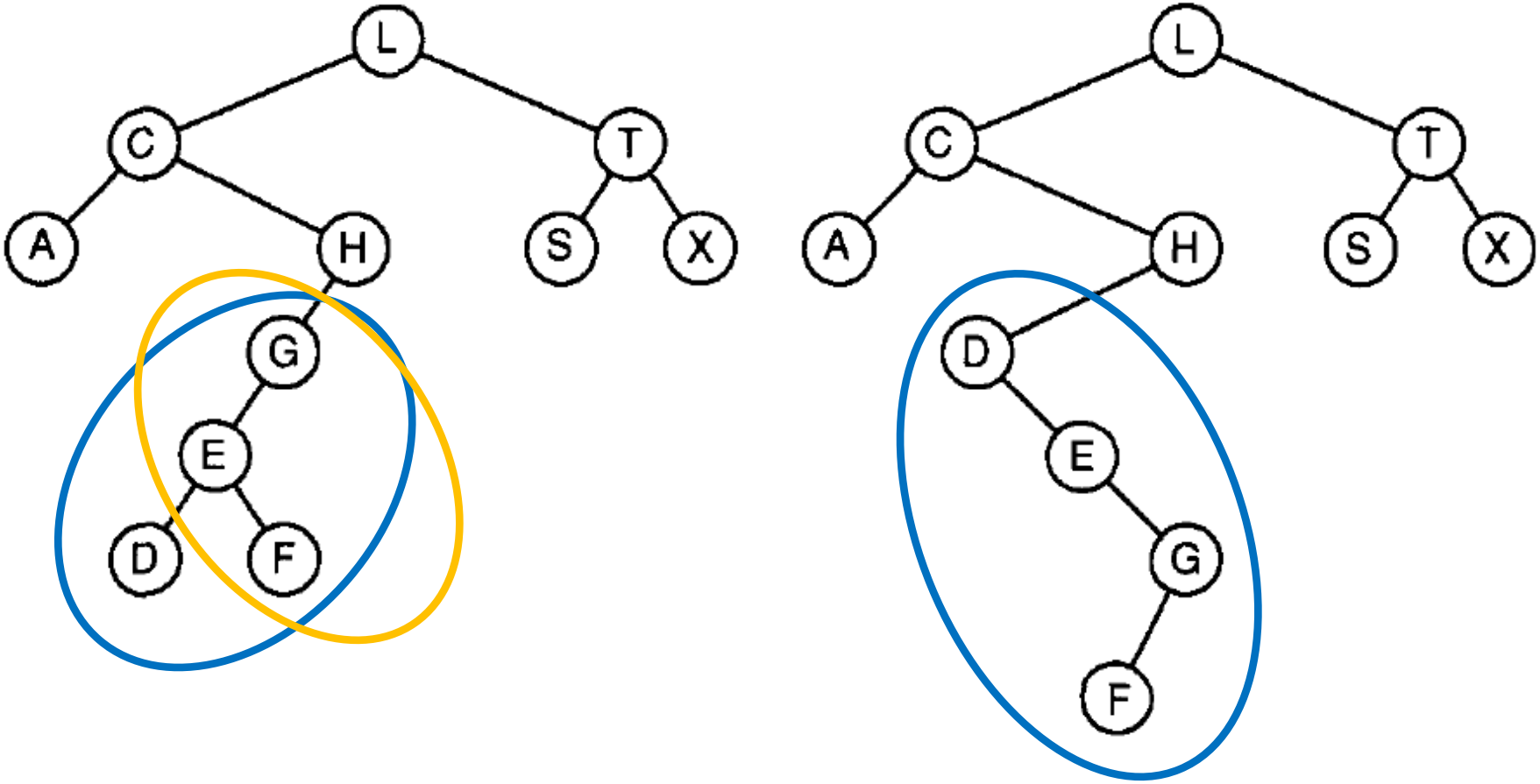
Реализация операции splay

- Шаг 3в): отец узла P – правый сын, а P – левый сын (или наоборот)
  - выполняем два однократных поворота в противоположных направлениях (сначала вокруг отца P направо, потом вокруг деда P налево)



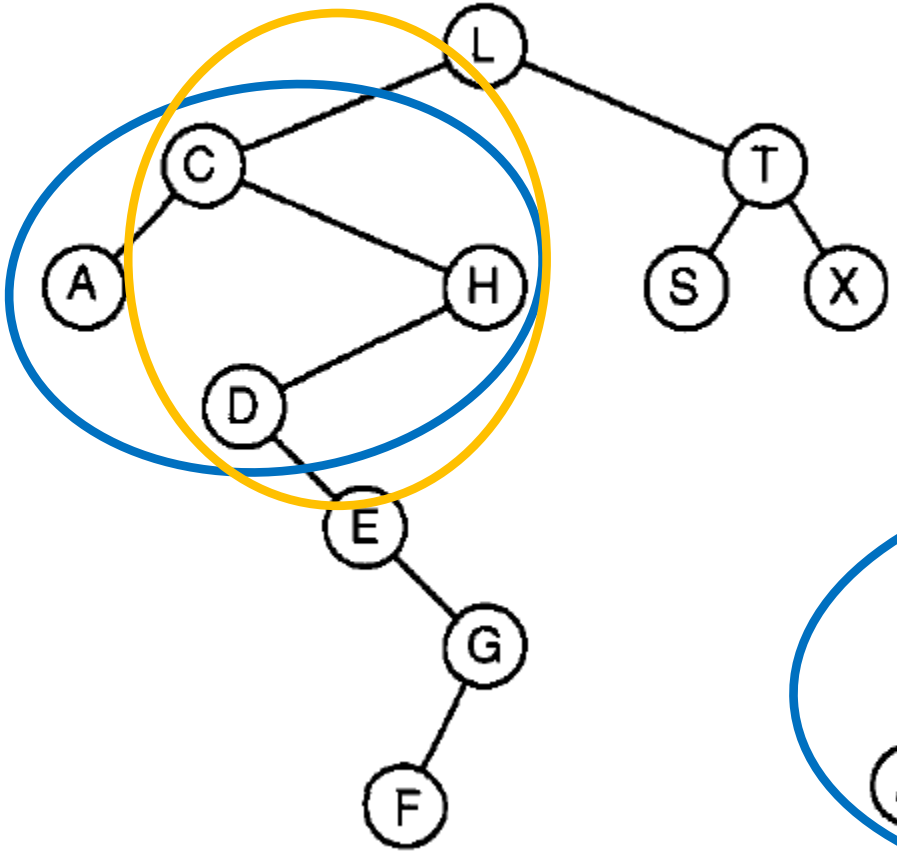
Пример операции splay над узлом D

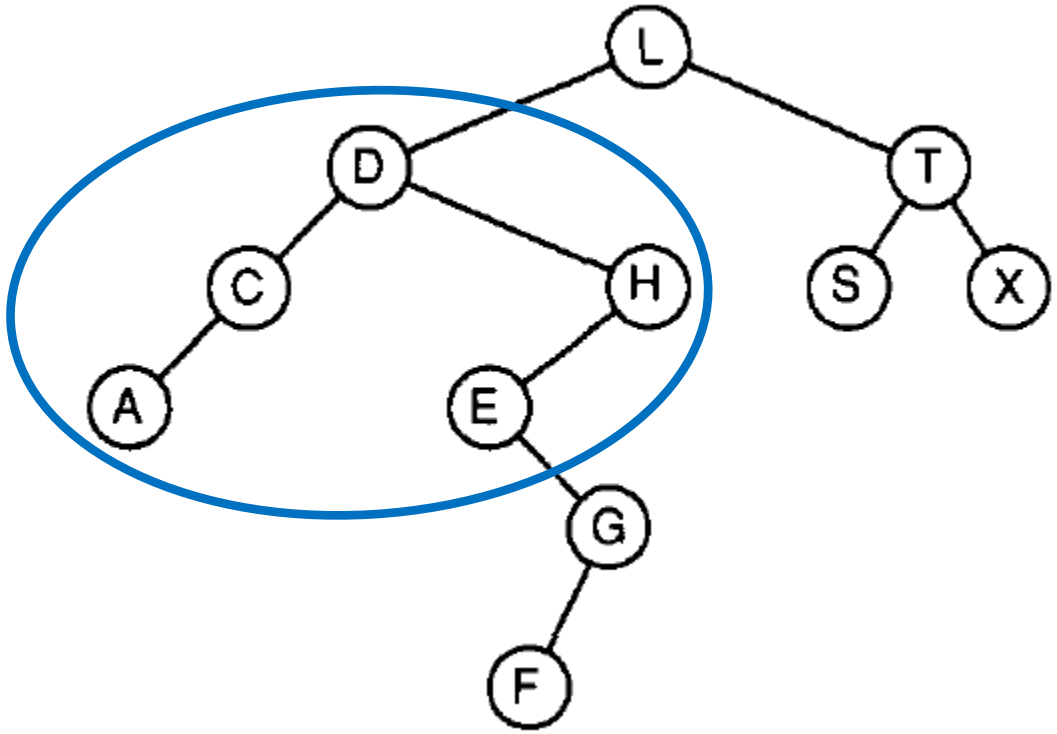
- Случай б): отец узла D (E) и сам узел D – оба левые сыновья



Пример операции splay над узлом D

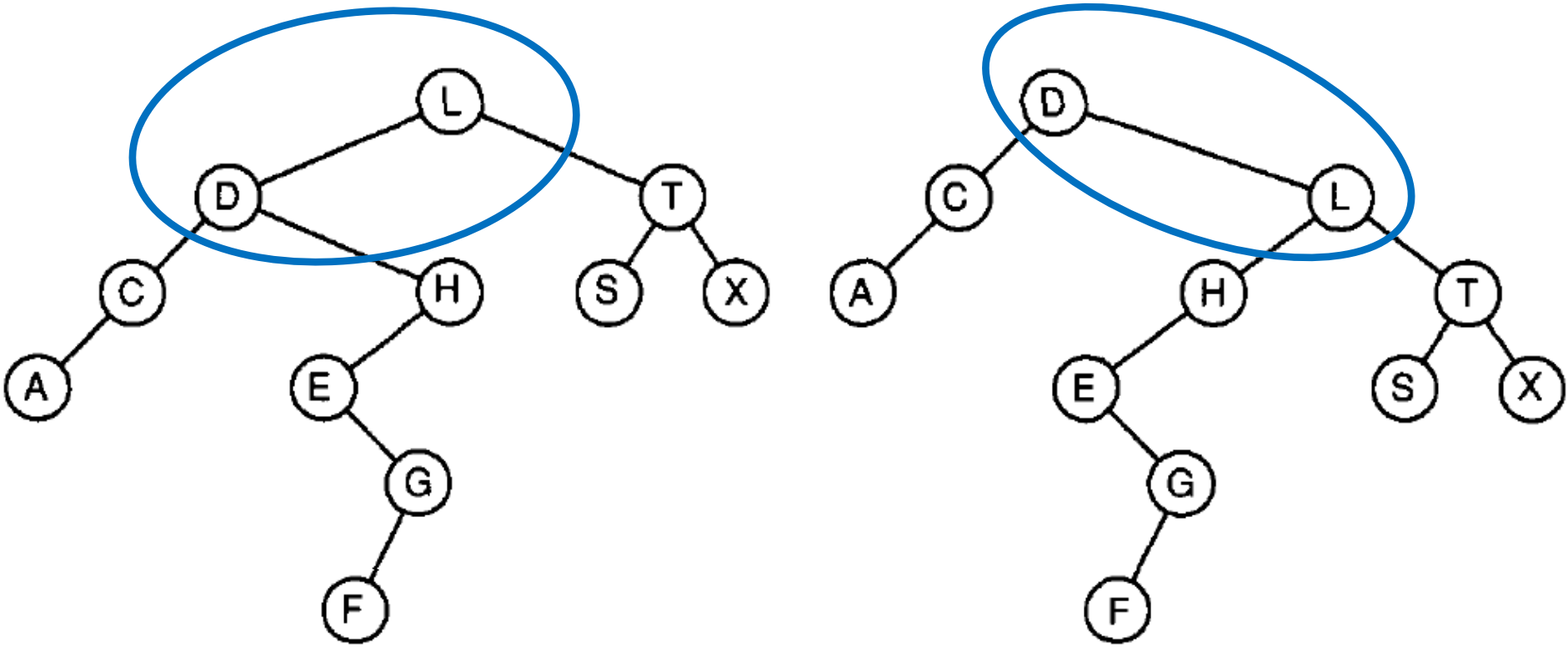
- Случай в): отец узла D (H) – правый сын, а сам узел D – левый сын





Пример операции splay над узлом D

- Случай а): отец узла D (L) – корень дерева



Сложность операции splay

- Пусть каждый узел дерева содержит некоторую сумму денег.
  - *Весом* узла является количество ее потомков, включая сам узел
  - *Рангом* узла $r(N)$ называется логарифм ее веса
  - *Денежный инвариант*: во время всех операций с деревом каждый узел содержит $r(N)$ рублей
  - Каждая операция с деревом стоит фиксированную сумму за единицу времени
- Лемма. Операция $splay$ требует *инвестирования* не более чем в $3\lfloor \lg n \rfloor + 1$ рублей с **сохранением** денежного инварианта.
- Теорема. Любая последовательность из m словарных операций на самоперестраивающемся дереве, которое было изначально пусто и на каждом шаге содержало не более n узлов, занимает не более $O(m \log n)$ времени.
  - Каждая операция требует не более $O(\log n)$ инвестиций, при этом может использовать деньги узла
  - По лемме инвестируется всего не более $m(3\lfloor \lg n \rfloor + 1)$ рублей, сначала дерево содержит 0 рублей, в конце содержит ≥0 рублей – $O(m \log n)$ хватает на все операции.

Сбалансированные деревья: *обобщение через ранги*

- Haeupler, Sen, Tarjan. Rank-balanced trees. ACM Transactions on Algorithms, 2015.

- Обобщение разных видов сбалансированных деревьев через понятие ранга (rank) и ранговой разницы (rank difference)
  - АВЛ, красно-черные деревья, 2-3 деревья, B-деревья

- Новый вид деревьев: слабые АВЛ-деревья (weak AVL)

- Анализ слабых АВЛ-деревьев, анализ потенциалов

Сбалансированные деревья: *понятие ранга*

- Ранг (rank) вершины r(x): неотрицательное целое число
  - Ранг отсутствующей (null) вершины равен -1
- Ранг дерева: ранг корня дерева
- Ранговая разница (rank difference): если у вершины x есть родитель p(x), то это число r(p(x)) – r(x).
  - У корня дерева нет ранговой разницы
- *i*-сын: вершина с ранговой разницей, равной i.
- *i*,*j*-вершина: вершина, у которой левый сын – это *i*-сын, а правый сын – это *j*-сын. Один или оба сына могут отсутствовать. *i*,*j*- и *j*,*i*-вершины не различаются.

Сбалансированные деревья: *ранговый формализм*

- Конкретный вид сбалансированного дерева определяется *рангом* и *ранговым правилом*.

- Ранговое правило должно гарантировать:
  - Высота дерева (h) превосходит его ранг не более чем в константное количество раз (плюс, возможно, O(1))
  - Ранг вершины (k) превосходит *логарифм* ее размера (n) не более чем в константное количество раз (плюс, возможно, O(1))

    Размер вершины – число ее потомков, включая себя, т.е. размер поддерева с корнем в этой вершине
  - T.e. $h = O(k)$, $k = O(\log n) \rightarrow h = O(\log n)$

- Совершенное дерево:

  ранг дерева – его высота; все вершины – 1,1.

Сбалансированные деревья: *ранговые правила*

- *АВЛ*-правило: каждая вершина – 1,1 или 1,2.
  - Ранг: высота дерева.

  (или: все ранги положительны, каждая вершина имеет хотя бы одного 1-сына)
  - Можно хранить один бит, указывающий на ранговую разницу вершины
- *Красно-черное* правило: ранговая разница любой вершины равна 0 или 1, при этом родитель 0-сына не может быть 0-сыном.
  - 0-сын – красная вершина, 1-сын – черная вершина
  - Ранг: черная высота
  - Корень не имеет цвета (т.к. не имеет ранговой разницы!)
- *Слабое* АВЛ-правило: ранговая разница любой вершины равна 1 или 2; все листья имеют ранг 0.
  - Вдобавок к АВЛ-деревьям разрешаются 2,2-вершины
  - Бит на узел для ранговой разницы или ее *четности*
  - Балансировка: не более двух поворотов и O(log n) изменений ранга для вставки/удаления, при этом амортизированно – лишь O(1) изменений.
  - Слабое АВЛ-дерево является красно-черным деревом

Пирамидальная сортировка (heapsort)

- Можно использовать дерево поиска для сортировки
- Например, последовательный поиск минимального элемента, удаление его и вставка в отсортированный массив
  - Сложность такого алгоритма есть $O(nh)$, где h – высота дерева
- Недостатки:
  - Требуется дополнительная память для дерева
  - Требуется построить само дерево (с минимальной высотой)
- Можно ли построить похожий алгоритм без требований к дополнительной памяти?

Пирамидальная сортировка: пирамида (двоичная куча)

- Рассматриваем массив **a** как двоичное дерево:
  - Элемент **a[i]** является узлом дерева
  - Элемент **a[i/2]** является родителем узла **a[i]**
  - Элементы **a[2*i]** и **a[2*i+1]** являются детьми узла **a[i]**

- Для всех элементов пирамиды выполняется соотношение (основное свойство кучи):
  a[i] >= a[2*i] и **a[i] >= a[2*i+1]**
  или
  a[i/2] <= a[i]
  - Сравнение может быть как в большую, так и в меньшую сторону

- **Замечание**. Определение предполагает нумерацию элементов массива от 1 до n
  - Для нумерации от 0 до n-1:
  a[i] >= a[2*i+1] и **a[i] >= a[2*i+2]**

Пирамидальная сортировка: пирамида (двоичная куча)

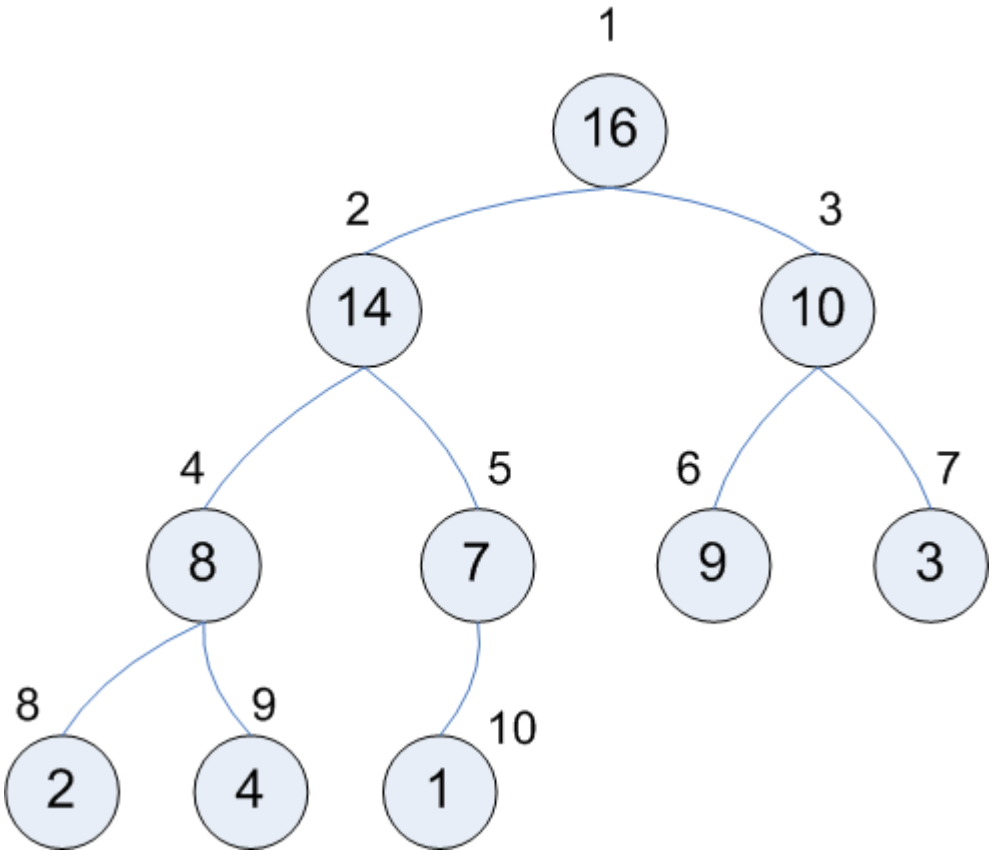
- Для всех элементов пирамиды выполняется соотношение:

  `a[i] >= a[2*i]` и **`a[i] >= a[2*i+1]`**

  или

  `a[i/2] <= a[i]`

  - Сравнение может быть как в большую, так и в меньшую сторону



Пирамидальная сортировка: просеивание элемента

- Как добавить элемент в уже существующую пирамиду?
- Алгоритм:
  - Поместим новый элемент в корень пирамиды
  - Если этот элемент меньше одного из сыновей:
    - Элемент меньше наибольшего сына
    - Обменяем элемент с наибольшим сыном (это позволит сохранить свойство пирамиды для другого сына)
    - Повторим процедуру для обмененного сына

Пирамидальная сортировка: просеивание элемента

```
static void sift (int *a, int l, int r) {
  int i, j, x;

  i = l; j = 2*l; x = a[l];
  /* j указывает на наибольшего сына */
  if (j < r && a[j] < a[j + 1])
    j++;
  /* i указывает на отца */
  while (j <= r && x < a[j]) {
    /* обмен с наибольшим сыном: a[i] == x */
    a[i] = a[j]; a[j] = x;
    /* продвижение индексов к следующему сыну */
    i = j; j = 2*j;
    /* выбор наибольшего сына */
    if (j < r && a[j] < a[j + 1])
      j++;
  }
}
```

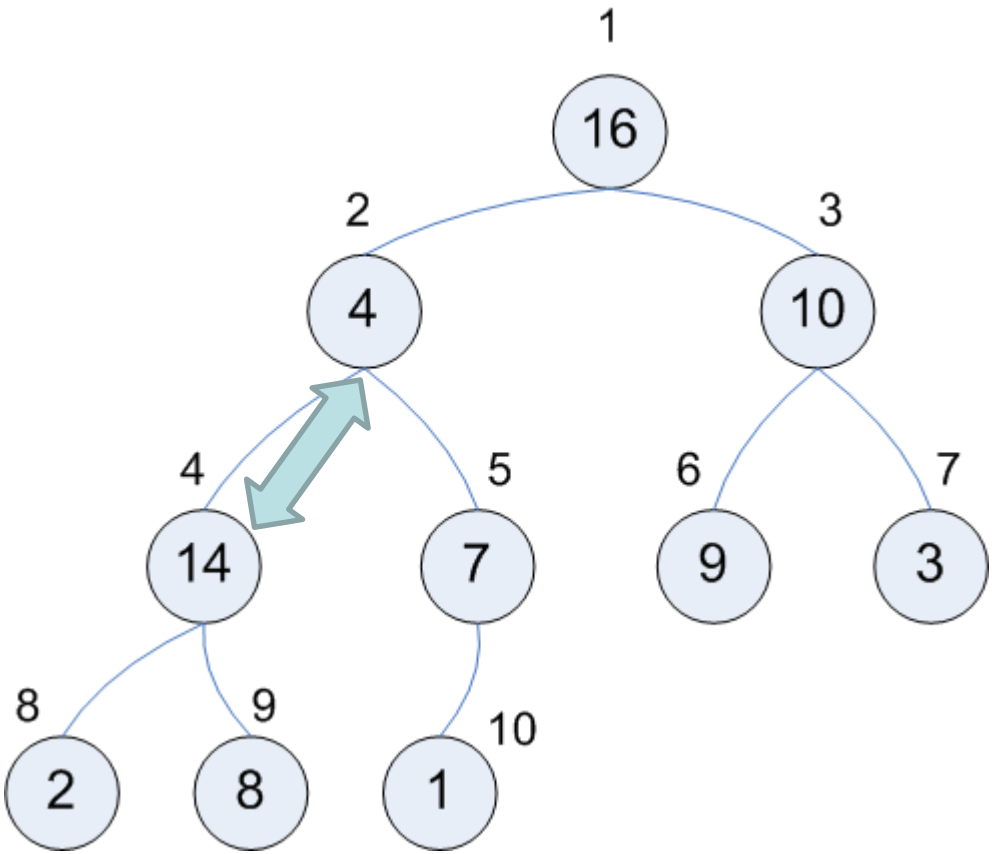
Пирамидальная сортировка: просеивание элемента

```
/* l, r - от 0 до n-1 */
static void sift (int *a, int l, int r) {
  int i, j, x;

  /* Теперь l, r, i, j от 1 до n, а индексы массива
     уменьшаются на 1 при доступе */
  l++, r++;
  i = l; j = 2*l; x = a[l-1];
  /* j указывает на наибольшего сына */
  if (j < r && a[j-1] < a[j])
    j++;
  /* i указывает на отца */
  while (j <= r && x < a[j-1]) {
    /* обмен с наибольшим сыном: a[i-1] == x */
    a[i-1] = a[j-1]; a[j-1] = x;
    /* продвижение индексов к следующему сыну */
    i = j; j = 2*j;
    /* выбор наибольшего сына */
    if (j < r && a[j-1] < a[j])
      j++;
  }
}
```

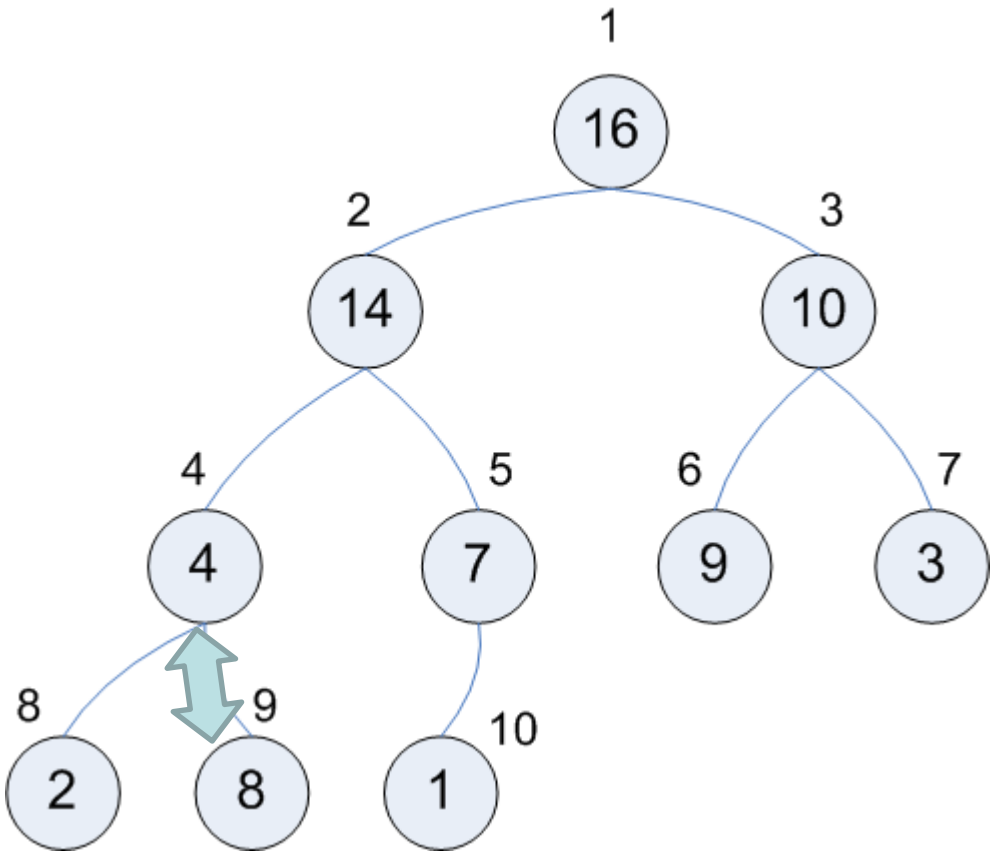
Пирамидальная сортировка: просеивание элемента

- Вызов sift (2, 10) для левого поддерева



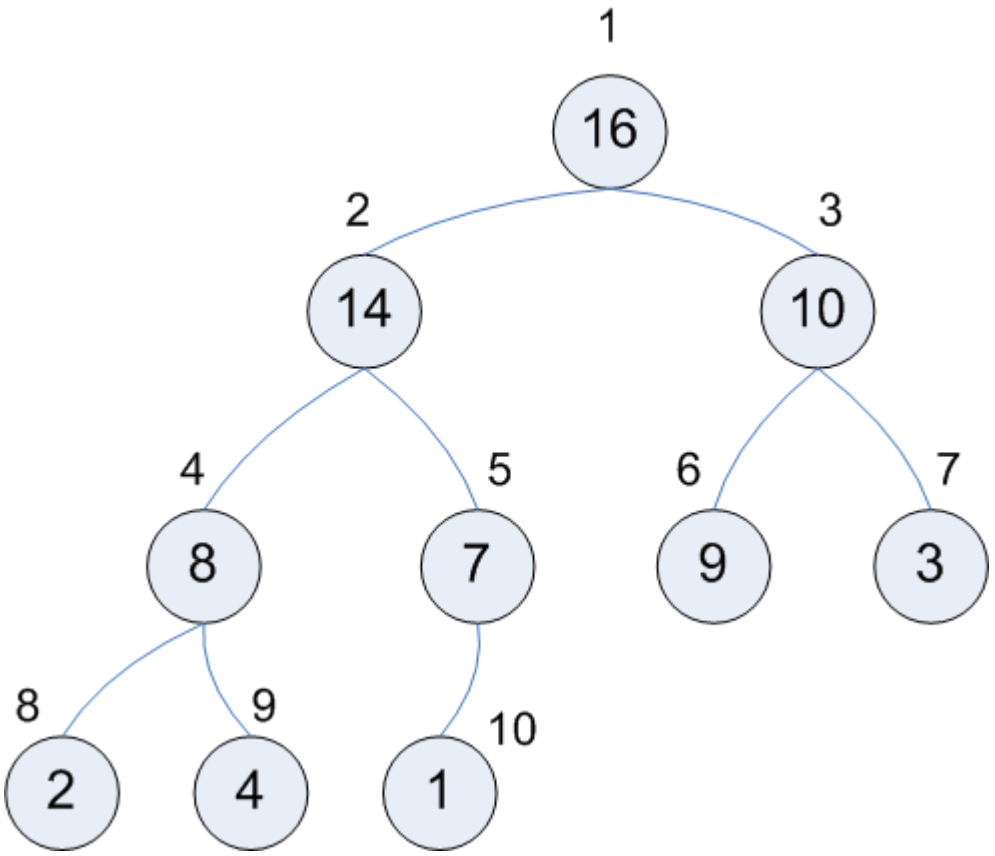
Пирамидальная сортировка: просеивание элемента

- Вызов sift (2, 10) для левого поддерева



Пирамидальная сортировка: просеивание элемента

- Вызов sift (2, 10) для левого поддерева



Пирамидальная сортировка: алгоритм

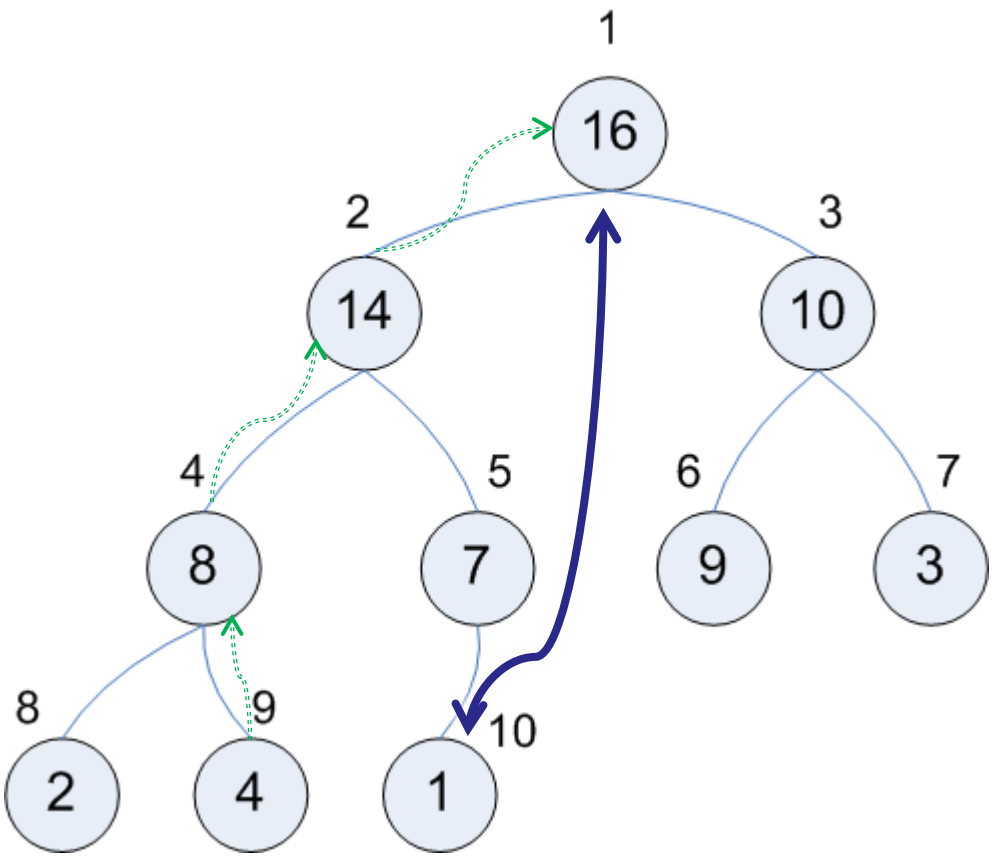
- (1) Построим пирамиду по сортируемому массиву
  - Элементы массива от n/2 до n являются листьями дерева, а следовательно, правильными пирамидами из одного элемента
  - Для остальных элементов в порядке уменьшения индекса просеиваем их через правую часть массива
- (2) Отсортируем массив по пирамиде
  - Первый элемент массива максимален (корень пирамиды)
  - Поменяем первый элемент с последним (таким образом, последний элемент отсортирован)
  - Теперь для первого элемента свойство кучи нарушено: повторим просеивание первого элемента в пирамиде от первого до предпоследнего
  - Снова поменяем первый и предпоследний элемент и т.п.

Пирамидальная сортировка: программа

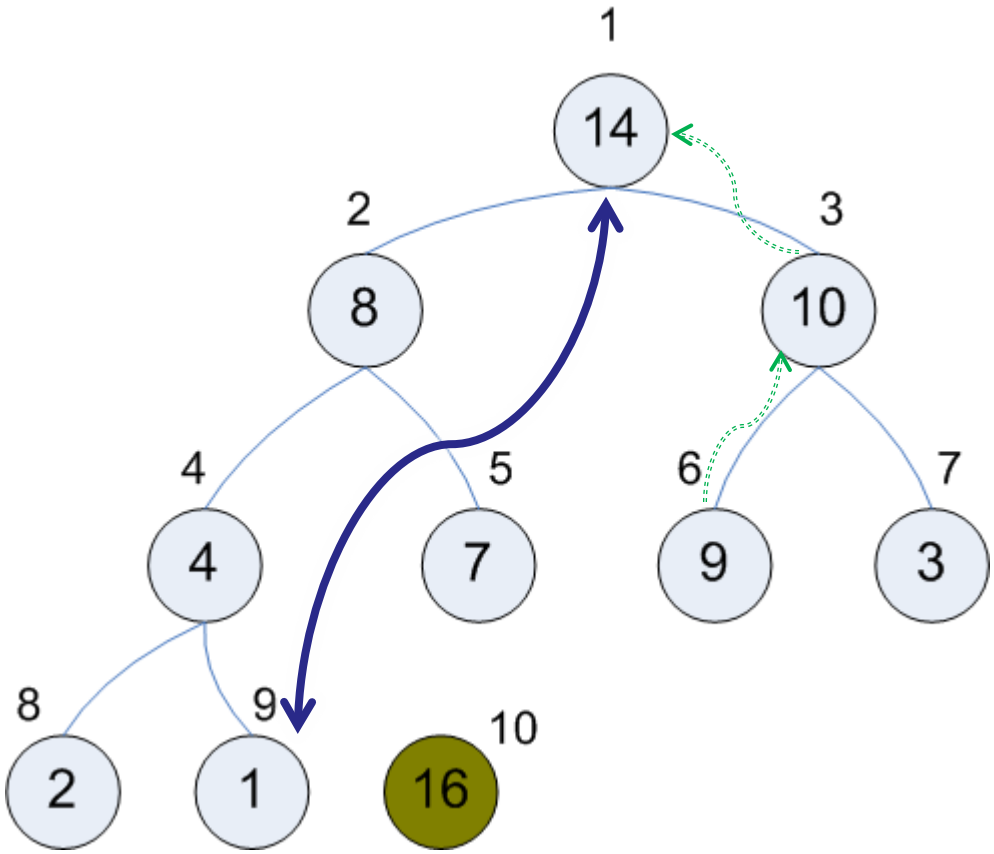
```
void heapsort (int *a, int n) {
  int i, x;

  /* Построим пирамиду по сортируемому массиву */
  /* Элементы нумеруются с 0 -> идем от n/2-1 */
  for (i = n/2 - 1; i >= 0; i--)
    sift (a, i, n - 1);
  for (i = n – 1; i > 0; i--) {
    /* Текущий максимальный элемент в конец */
    x = a[0]; a[0] = a[i]; a[i] = x;
    /* Восстановим пирамиду в оставшемся массиве */
    sift (a, 0, i – 1);
  }
}
```

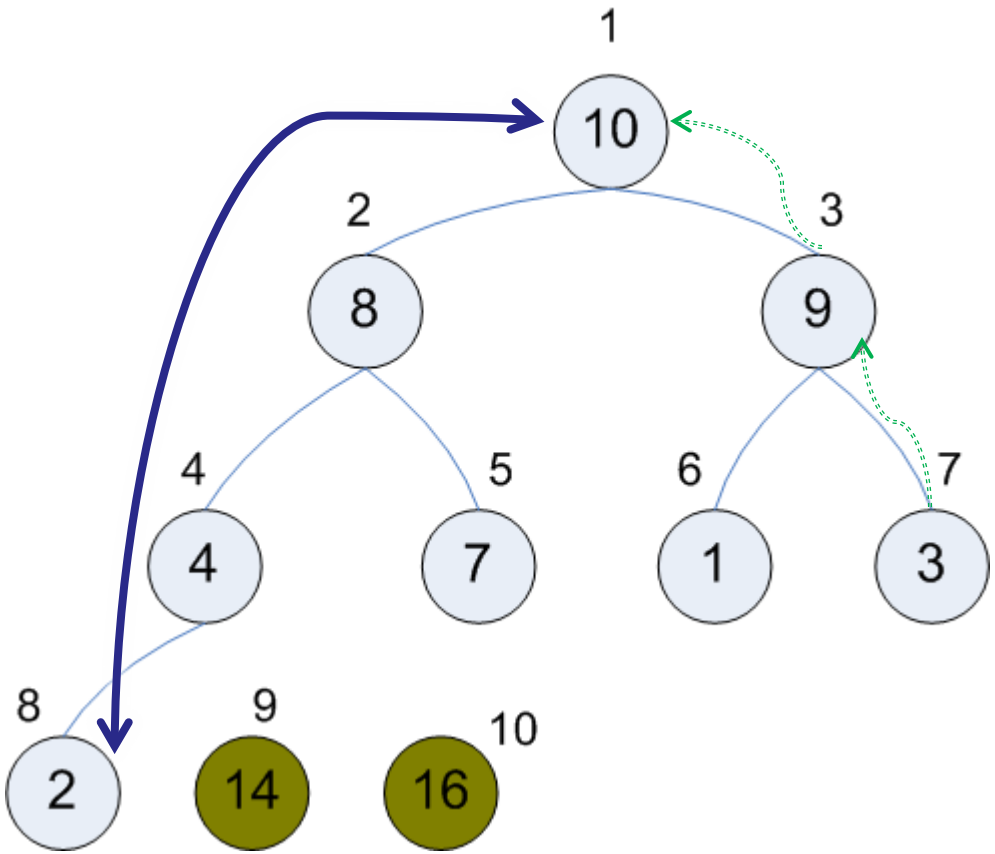
Пирамидальная сортировка: пример



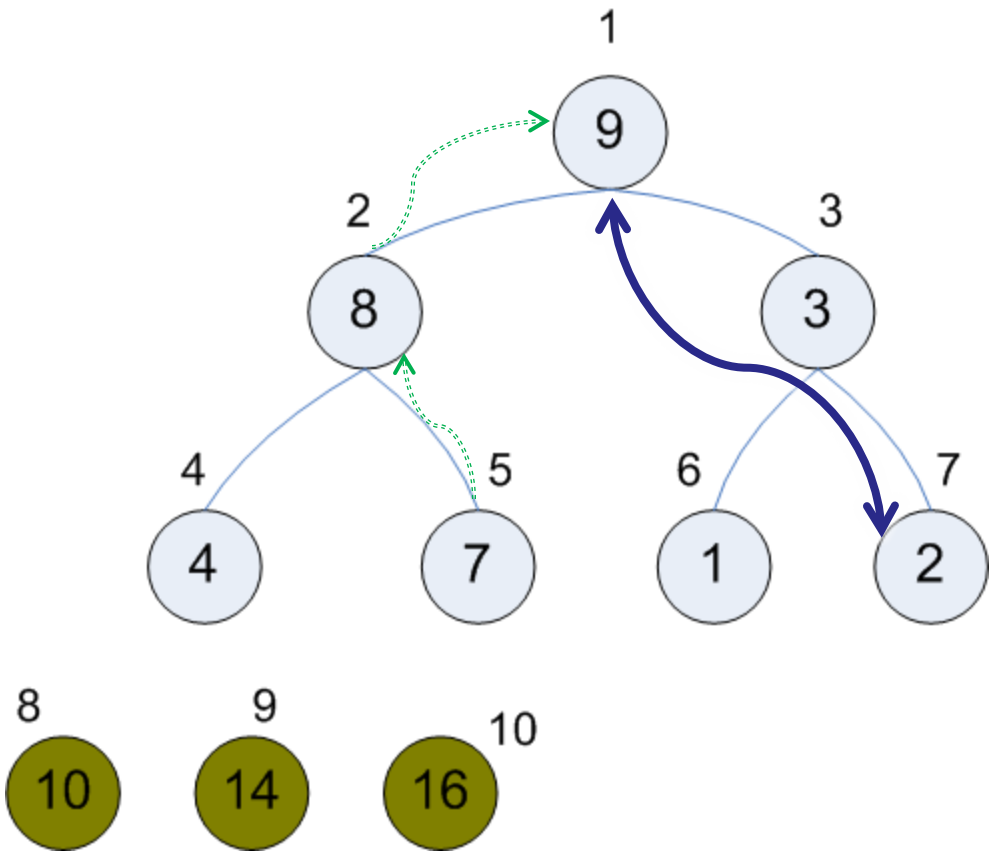
Пирамидальная сортировка: пример



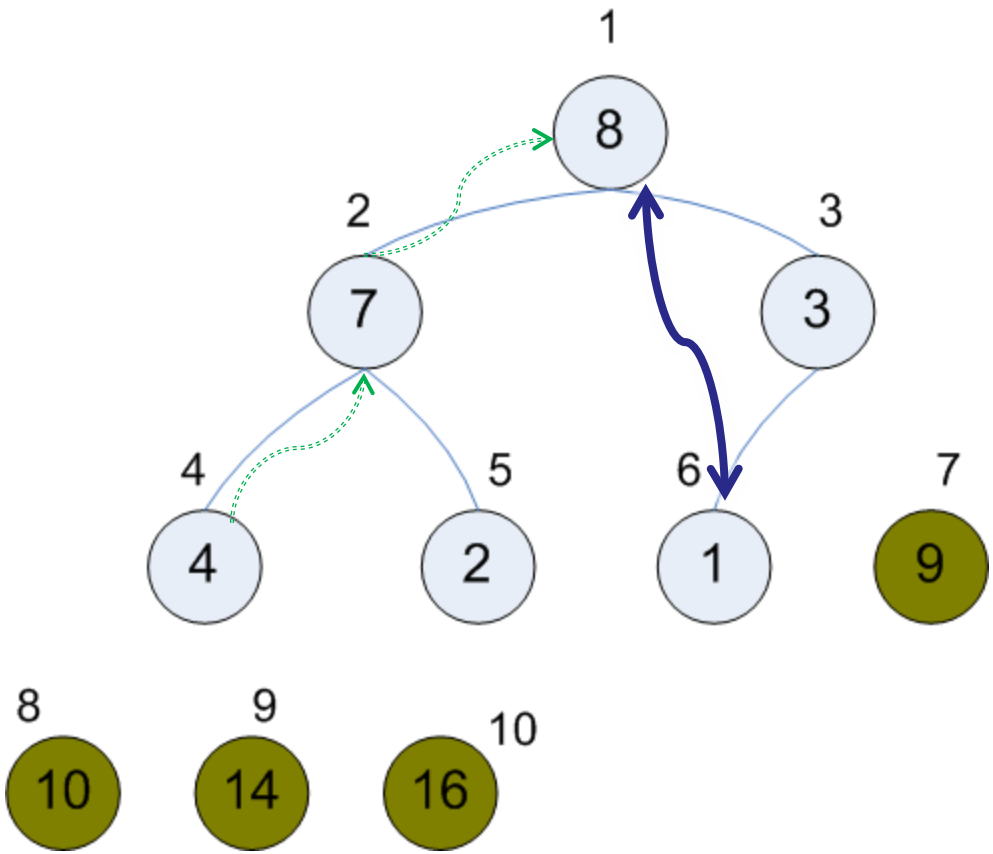
Пирамидальная сортировка: пример



Пирамидальная сортировка: пример



Пирамидальная сортировка: пример



Пирамидальная сортировка: сложность алгоритма

- (1) Построим пирамиду по сортируемому массиву
  - Элементы массива от n/2 до n являются листьями дерева, а следовательно, правильными пирамидами из 1 элемента
  - Для остальных элементов в порядке уменьшения индекса просеиваем их через правую часть массива
- (2) Отсортируем массив по пирамиде
  - Первый элемент массива максимален (корень пирамиды)
  - Поменяем первый элемент с последним (таким образом, последний элемент отсортирован)
  - Теперь для первого элемента свойство кучи нарушено: повторим просеивание первого элемента в пирамиде от первого до предпоследнего
  - Снова поменяем первый и предпоследний элемент и т.п.

- Сложность этапа построения пирамиды есть $O(n)$
- Сложность этапа сортировки есть $O(n \log n)$
- Сложность в худшем случае также $O(n \log n)$
- Среднее количество обменов – $n/2 * \log n$