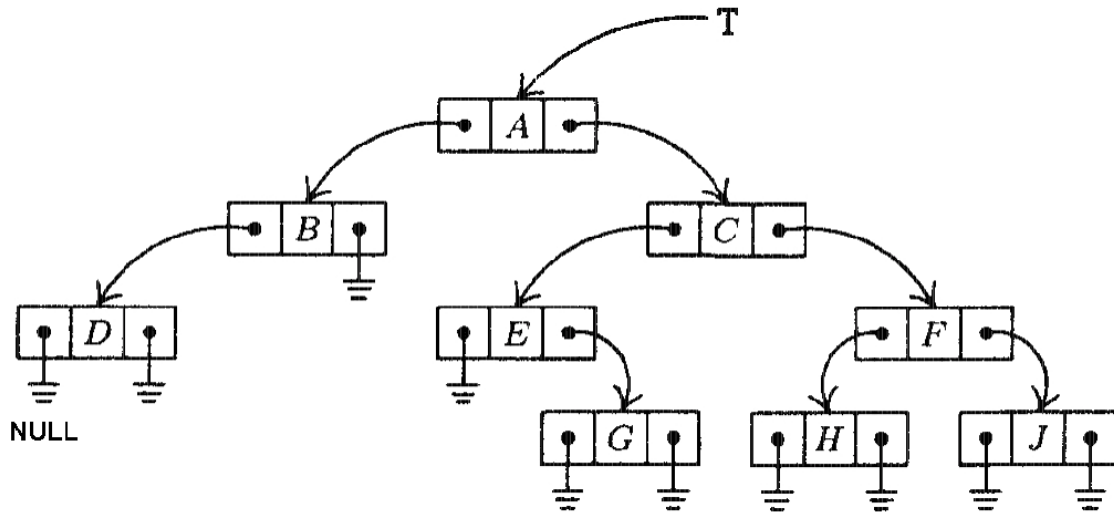
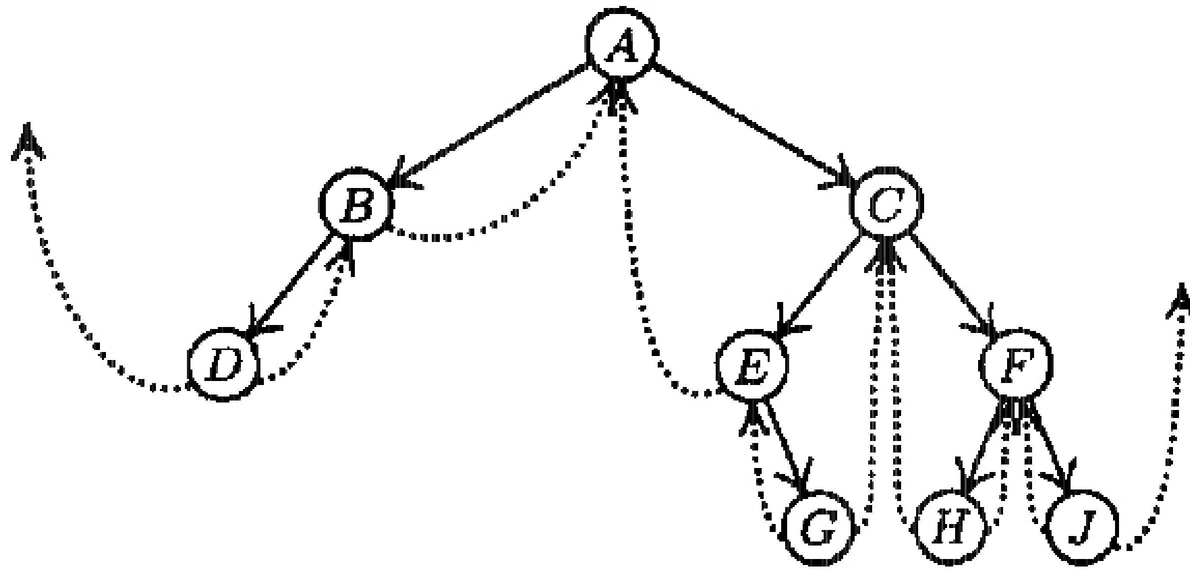
Курс «Алгоритмы и алгоритмические языки»

Лекция 17

Двоичное дерево

- ***Прошитое двоичное дерево***





Рассмотрим двоичное дерево, на верхнем рисунке.
У этого дерева нулевых указателей, больше, чем ненулевых:
10 против 8
Это – типичный случай.

Будем записывать вместо нулевых указателей указатели на родителей (или более далеких предков) соответствующих узлов (такие указатели называются *нитями*).
Это позволит при обходе дерева не использовать стек.

Двоичное дерево

- ***Прошитое двоичное дерево***
- Описание узла прошитого двоичного дерева

```
typedef struct bin_tree {
  char info;
  int left_tag;
  struct bin_tree *left;
  int right_tag;
  struct bin_tree *right;
} threaded_node;
```

Нити устанавливаются таким образом, чтобы указывать на предшественников (левые нити) или последователей (правые нити) текущего узла при соответствующем обходе дерева. Например, в случае симметричного обхода

Обычное дерево	*Прошитое дерево*
`P->left == NULL`	`P->left_tag == 1, P->left == P_pred_in`
`P->left == Q`	`P->left_tag == 0, P->left == Q`
`P->right == NULL`	`P->right_tag == 1, P->right == P_next_in`
`P->right == Q`	`P->right_tag == 0, P->right == Q`

Двоичное дерево

- ***Прошитое двоичное дерево***
- Нити существенно упрощают алгоритмы обхода двоичных деревьев. Например, для вычисления для каждого узла **`P`** указатель узла **`P_next_in`** можно использовать следующий простой алгоритм:

```
threaded_node * Next_in (threaded_node *P) {
  threaded_node *Q = P->right;
  if (P->right_tag == 1)
    return Q;
  while (Q->left_tag == 0)
    Q = Q->left;
  return Q;
}
```

- Функция **`Next_in`** фактически реализует симметричный обход дерева, так как позволяет для произвольного узла дерева **`P`** найти **`P_next_in`**, т.е. применяя эту функцию несколько раз, можно вычислить топологический порядок узлов двоичного дерева, соответствующий симметричному обходу.

Двоичное дерево

- ***Прошитое двоичное дерево***
- Аналогичным образом можно вычислить **`P_next_pred`** и **`P_next_post`**.

  Применяя функции **`P_next_pred`** (либо **`P_next_post`**), можно вычислить топологический порядок узлов, соответствующий прямому (либо обратному) обходу.
- ***Замечания***

  (1) С помощью обычного представления невозможно для произвольного узла **`P`** вычислить **`P_next_in`**, не вычисляя всей последовательности узлов.

  (2) Функции **`Next_in`** не требуется стек ни в явной, ни в неявной (рекурсия) форме.

Двоичное дерево

- ***Прошитое двоичное дерево***
- Сравнение функций **inorder()** и **Next_in()** позволяет сделать следующие выводы:
  - Если **P** – произвольно выбранный узел дерева,то следующий фрагмент функции **Next_in()**:

```
Q = P->right;
if (P->right_tag == 1)
  return Q;
```

    выполняется только один раз.
  - Обход прошитого дерева выполняется быстрее, так как для него не нужны операции со стеком.
  - Для **inorder()** требуется больше памяти, чем для **Next_in()**, из-за массива **Stack[D]**
    D обычно стараются взять не очень большим, но **D** не может быть меньше высоты двоичного дерева.
    Нельзя допускать переполнение стека деревьев.
  - Можно доказать, что функция **inorder()** работает примерно в два раза дольше, чем функция **Next_in()**.
  - Алгоритм, реализуемый функцией **Next_in()**, более общий, чем алгоритм, реализуемый функцией **inorder()**: он позволяет перейти от узла **P** к узлу **P_next_in**, не выполняя обхода соответствующего двоичного дерева.

Двоичное дерево

◊ *Прошитое двоичное дерево*

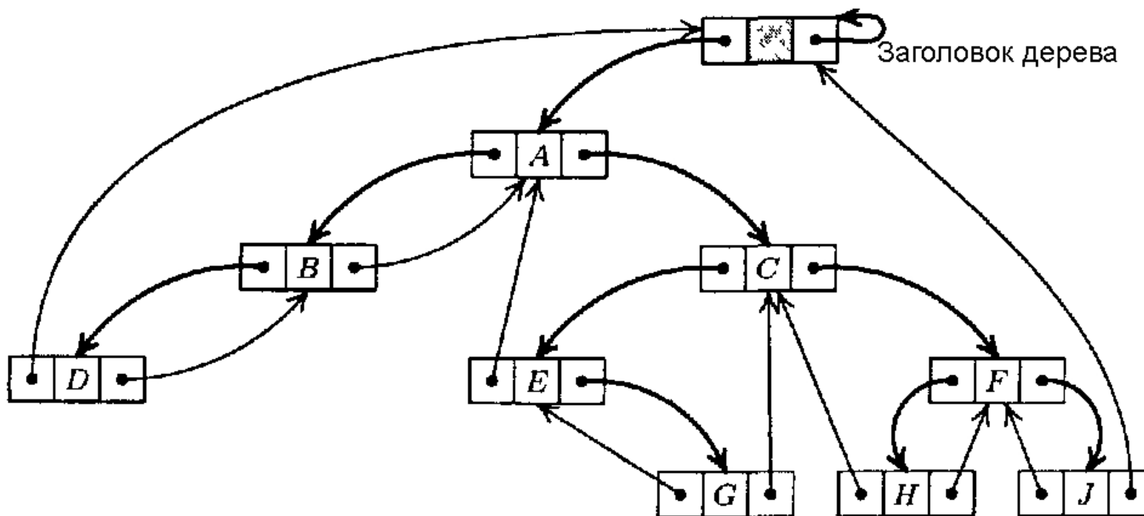
В функции **inorder()** используется указатель **r** на корень двоичного дерева. Желательно, применив функцию **Next_in()** к корню **r**, получить указатель узла дерева, следующего за корнем для выбранного порядка обхода. Для этого к дереву добавляется еще один узел – заголовок дерева.

поля структуры

```
struct bin_tree {
    char info;
   int left_tag;
   struct bin_tree *left;
   int right_tag;
   struct bin_tree *right;
} *header;
```

заполняются в заголовке следующим образом

```
header->left_tag = 0;
header->right_tag = 0;
header->left = r;
header->right = header;
```



На рисунке дуги дерева показаны более жирными линиями, чем нити.

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)

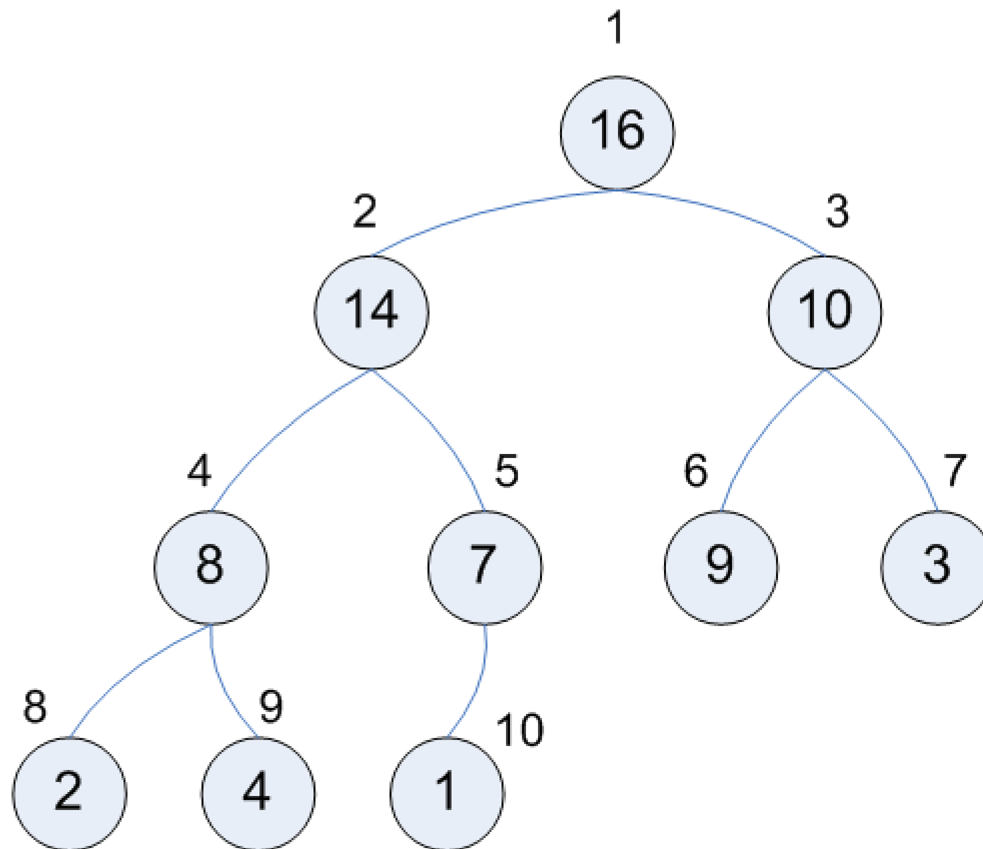
- Для всех элементов пирамиды выполняется соотношение:

  `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++;
  }
}
```

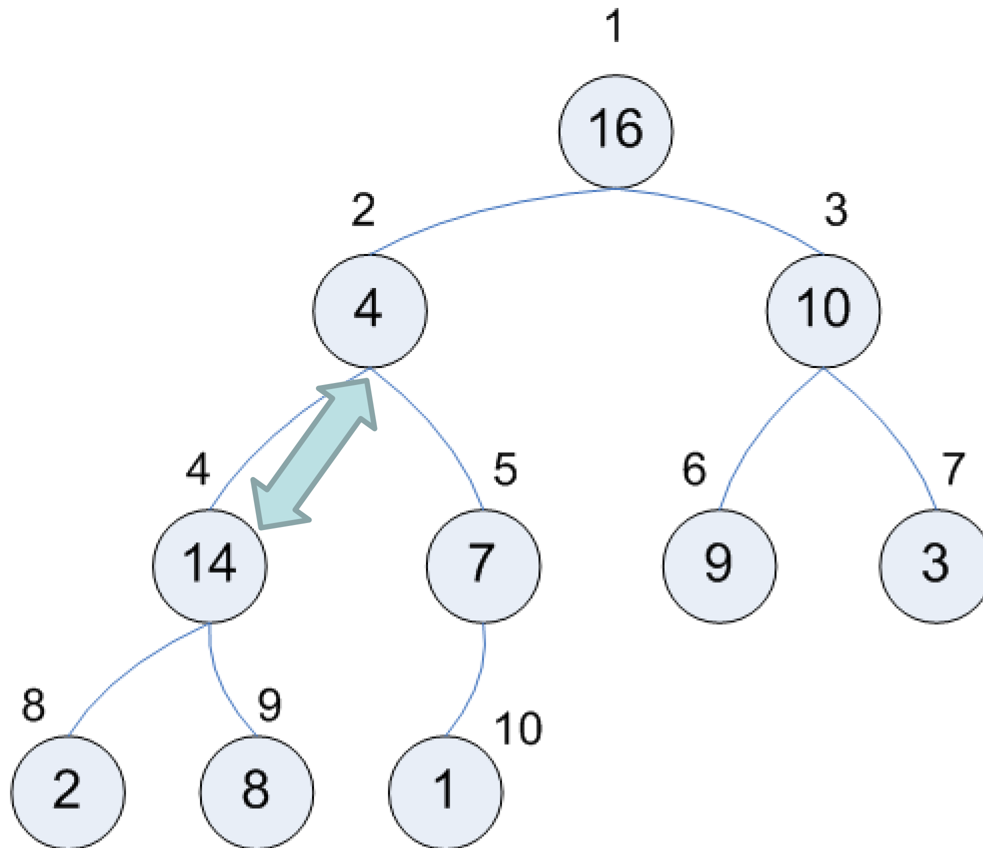
Пирамидальная сортировка: просеивание элемента

```
/* 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++;
  }
}
```

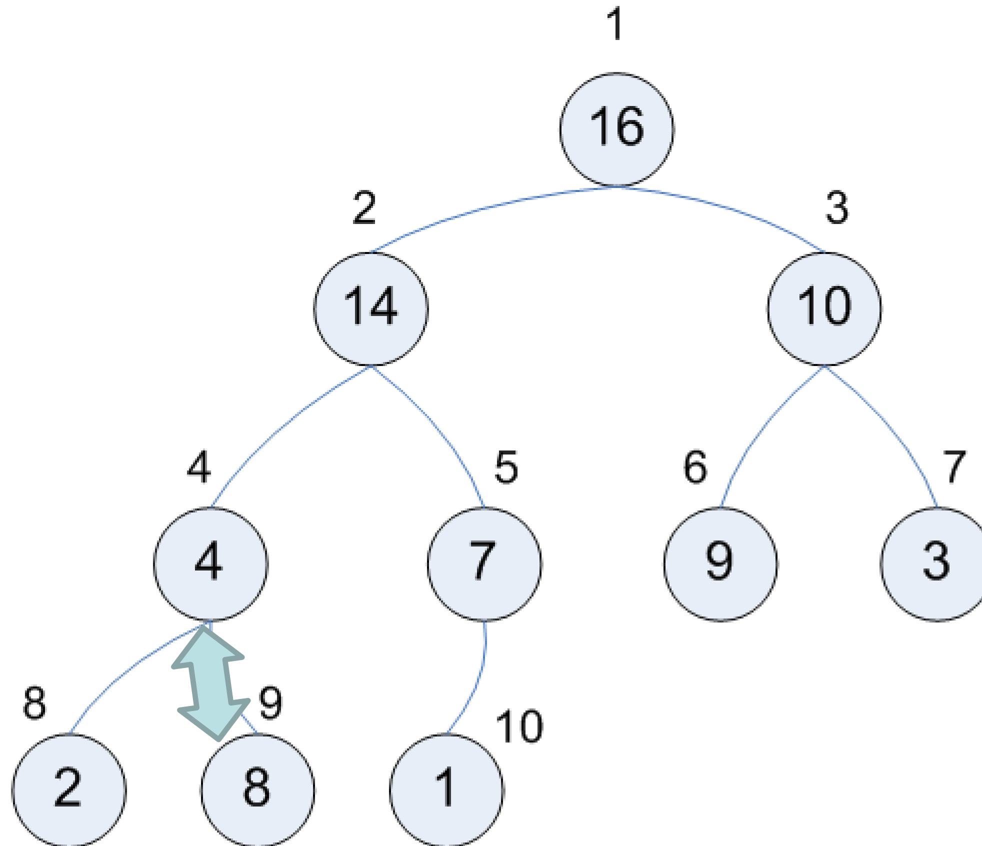
Пирамидальная сортировка: просеивание элемента

- Вызов sift (2, 10) для левого поддерева



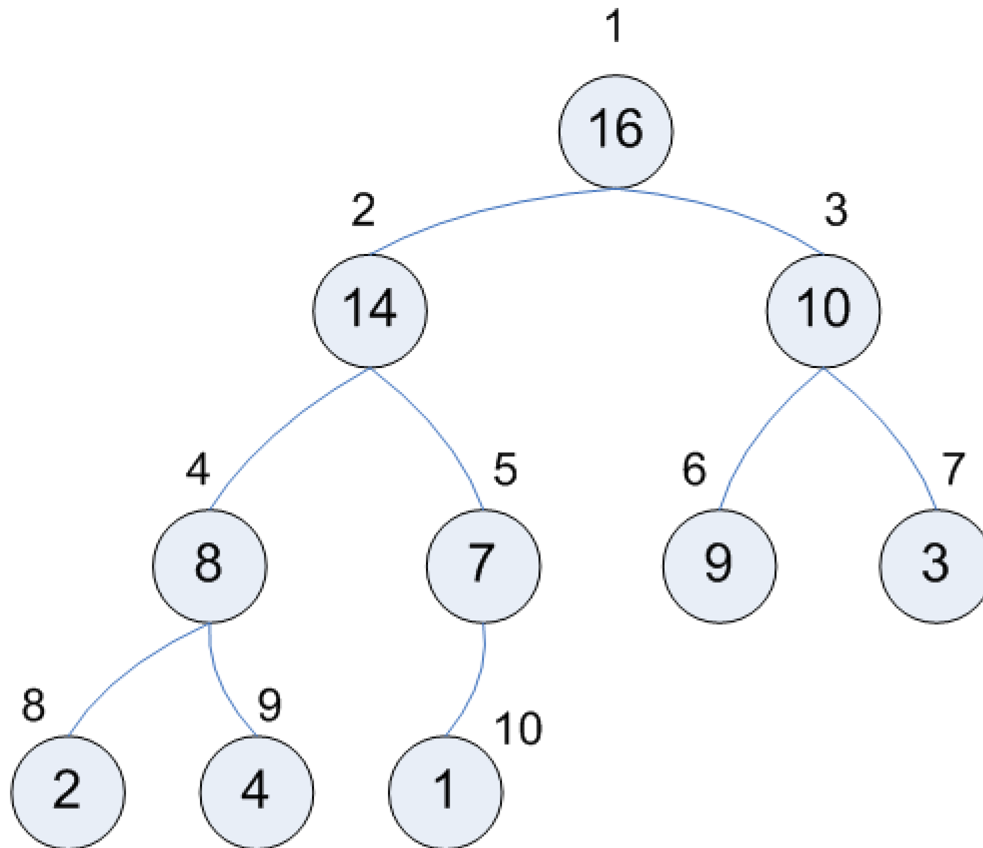
Пирамидальная сортировка: просеивание элемента

- Вызов sift (2, 10) для левого поддерева



Пирамидальная сортировка: просеивание элемента

- Вызов sift (2, 10) для левого поддерева



Пирамидальная сортировка: алгоритм

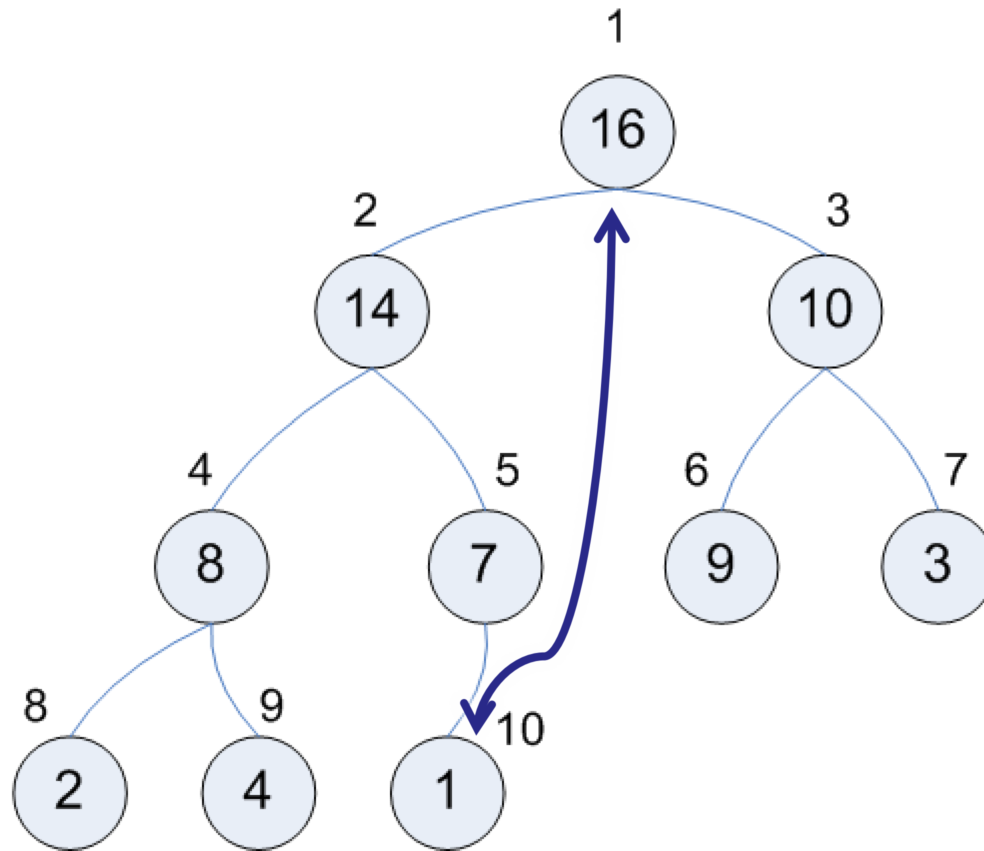
- (1) Построим пирамиду по сортируемому массиву
  - Элементы массива от n/2 до n являются листьями дерева, а следовательно, правильными пирамидами
  - Для остальных элементов в порядке уменьшения индекса просеиваем их через правую часть массива
- (2) Отсортируем массив по пирамиде
  - Первый элемент 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

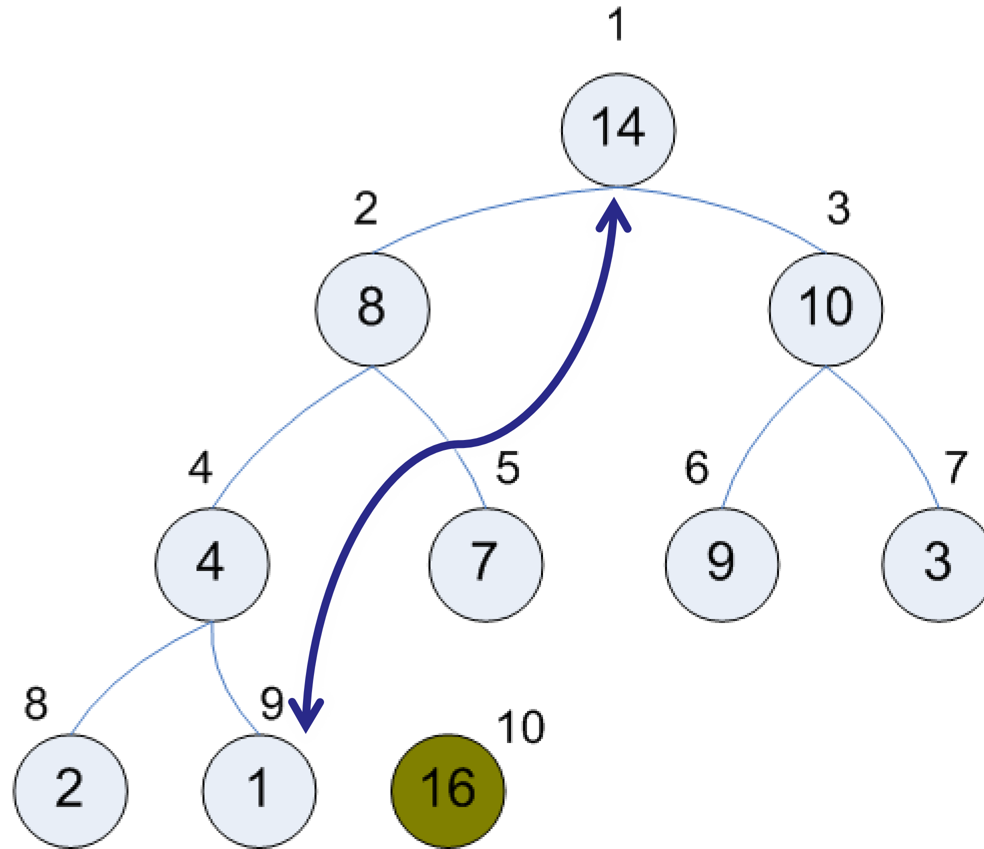
```
void heapsort (int *a, int n) {
  int i, x;

  /* 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);
  }
}
```

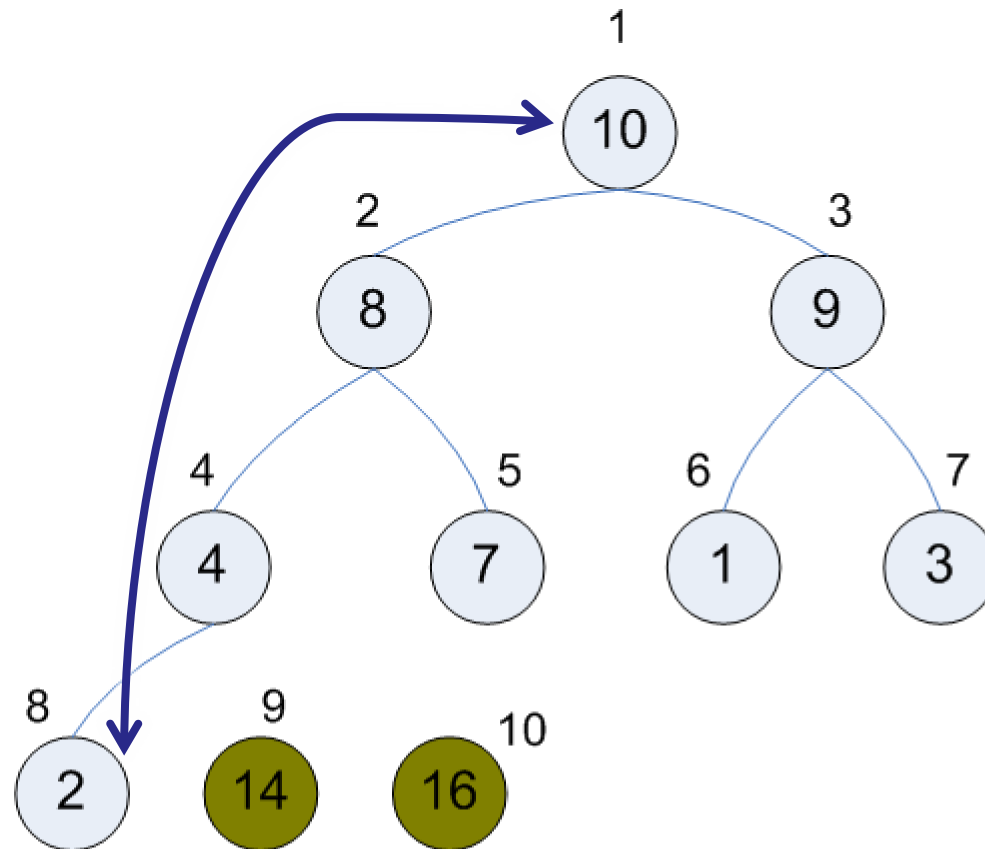
Пирамидальная сортировка: пример



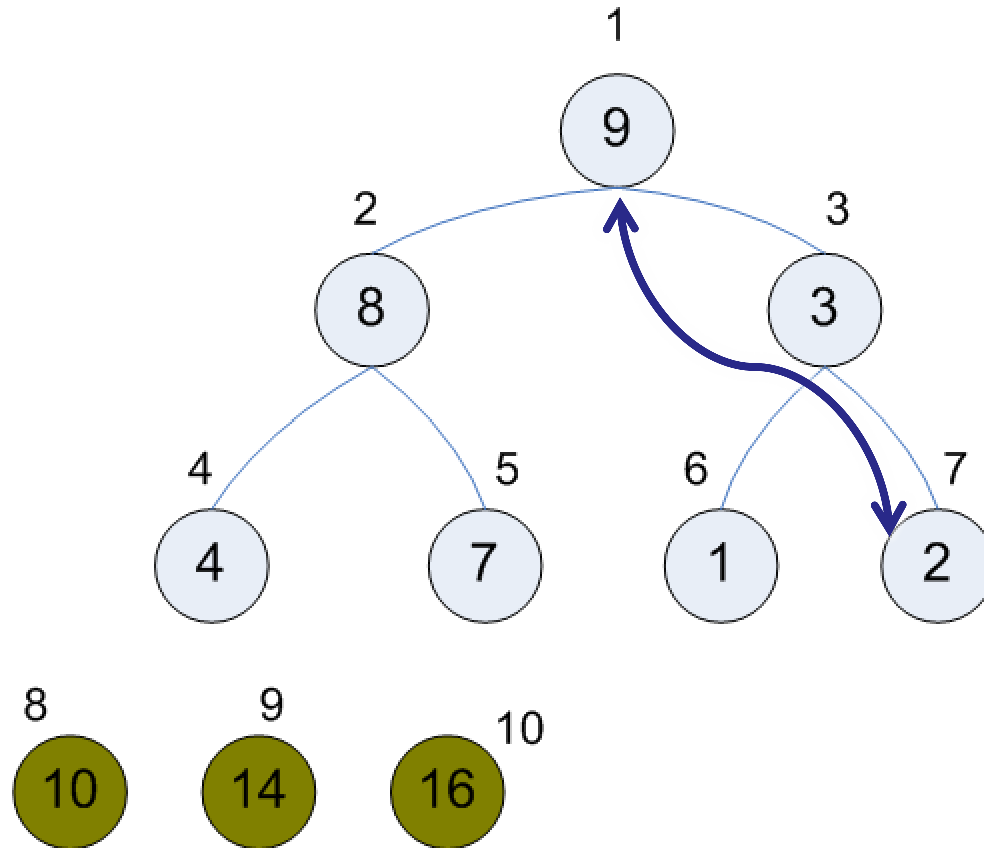
Пирамидальная сортировка: пример



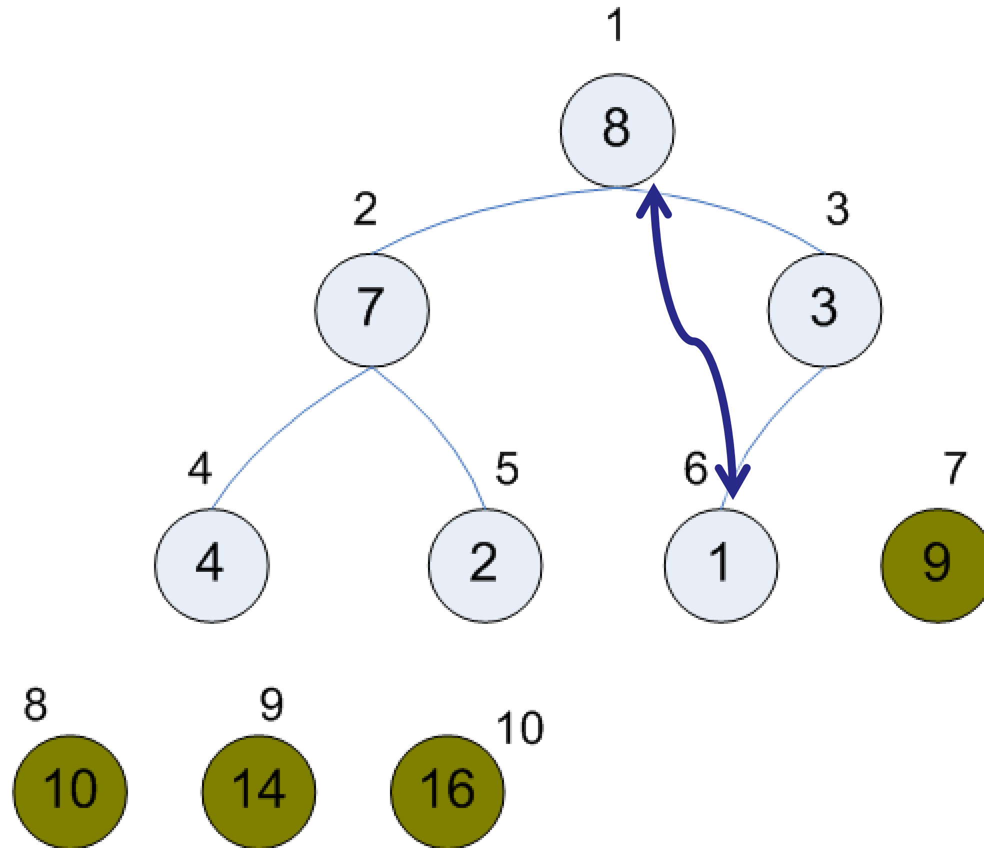
Пирамидальная сортировка: пример



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
  - 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 – $n/2 * \log n$

Построение двоичного дерева поиска.

- **Постановка задачи.** Пусть имеется множество K из m ключей:

$$K = \{k_0, k_1, \ldots, k_{m-1}\}$$

Разбиение K на три подмножества K_1, K_2, K_3:

- $|K_2| = 1, |K_1| \geq 0, |K_3| \geq 0.$
- $K_2 = \{k\} \Rightarrow \forall l \in K_1: l < k$ и $\forall r \in K_3: r \geq k$

Далее по рекурсии: разбиваем K_1 на K_{11}, K_{12}, K_{13}
K_3 на K_{31}, K_{32}, K_{33}

и т.д. пока ключи не кончатся

- **Пример**: $K = \{15,10,1,3,8,12,4\}$.
Первое разбиение: $\{1,3,4\}, \{8\}, \{15,10,12\}$;
второе разбиение: $\{\{1\}\{3\}\{4\}\}\{8\}\{\{10\}\{12\}\{15\}\}$.

Получилось полностью сбалансированное двоичное дерево.

- **Определение**. Дерево называется *полностью сбалансированным* (*совершенным*), если длина пути от корня до любой листовой вершины одинакова.

Построение двоичного дерева поиска.

- Пусть h – высота полностью сбалансированного двоичного дерева. Тогда число вершин m должно быть равно:

$$m = 1 + 2 + 2^2 + \ldots + 2^h = 2^h - 1$$

откуда $h = \log_2(m + 1)$.

- Если все m ключей известны заранее, их можно отсортировать за $O(m \cdot \log_2 m)$, после чего построение сбалансированного дерева будет тривиальной задачей.
- Если дерево строится по мере поступления ключей, то все может быть: от линейного дерева с высотой $O(m)$ до полностью сбалансированного дерева с высотой $O(\log_2 m)$.
- Пусть $T = \{root, left, right\}$ – двоичное дерево; тогда $h_T = \max(h_{left}, h_{right}) + 1$.

Деревья Фибоначчи

- **Числа Фибоначчи** возникли в решении задачи о кроликах, предложенном в XIII веке Леонардо из Пизы, известным как Фибоначчи.

Задача о кроликах: пара новорожденных кроликов помещена на остров. Каждый месяц любая пара дает приплод – также пару кроликов.

Пара начинает давать приплод в возрасте двух месяцев.

Сколько кроликов будет на острове в конце n-го месяца?

В конце первого и второго месяцев на острове будет одна пара кроликов:

$$f_1 = 1, f_2 = 1.$$

В конце третьего месяца родится новая пара, так что

$$f_3 = f_2 + f_1 = 2.$$

По индукции можно доказать, что для $n \geq 3$

$$f_n = f_{n-1} + f_{n-2}.$$

Деревья Фибоначчи

- n-е число Фибоначчи вычисляет следующая функция:

```
int Fbn (int n) {
      if (n == 1 || n == 2)
            return 1;
      else {
            int g, h, k, Fb;
            g = h = 1;
            for (k = 2; k < n; k++) {
              Fb = g + h;
              h = g;
              g = Fb;
            }
            return Fb;
      }
}
```