

**Курс «Алгоритмы и алгоритмические языки»
1 семестр 2017/2018**

Лекция 21

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):
 $a[i] \geq a[2*i]$ и $a[i] \geq a[2*i+1]$
или
 $a[i/2] \geq a[i]$
 - ◆ 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 $n-1$:
 $a[i] \geq a[2*i+1]$ и $a[i] \geq a[2*i+2]$

Пирамидальная сортировка: пирамида (двоичная куча)

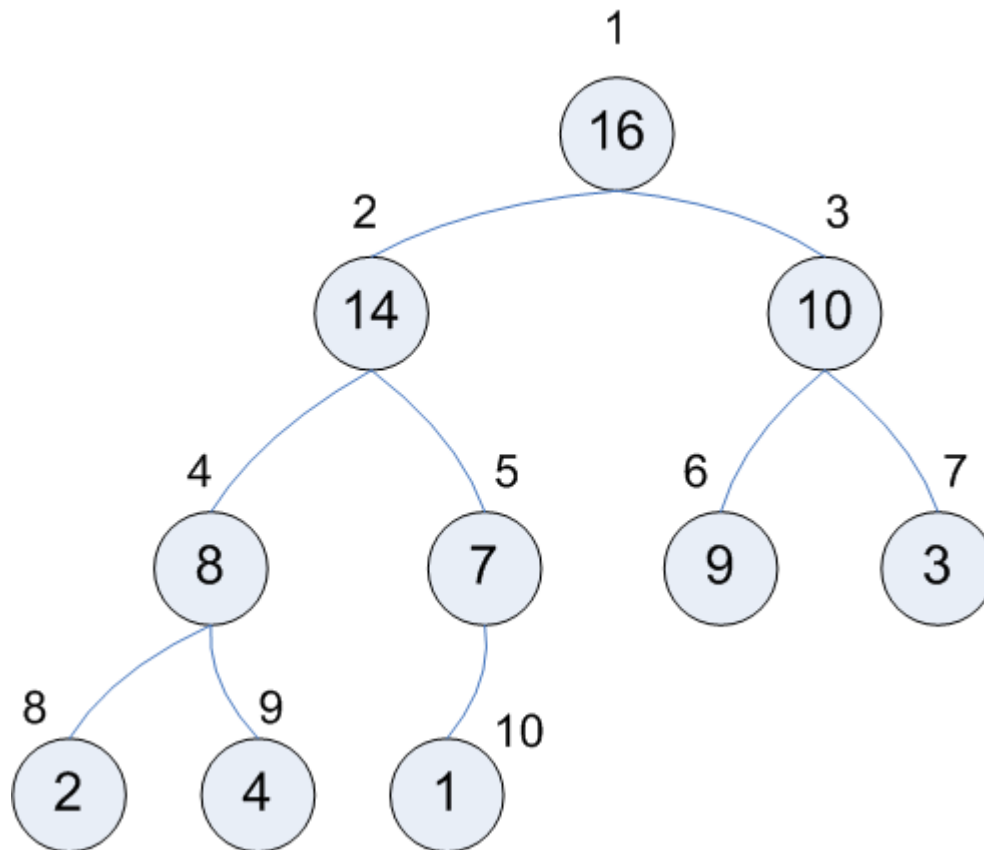
◇ Для всех элементов пирамиды выполняется соотношение:

$$a[i] \geq a[2*i] \text{ и } a[i] \geq a[2*i+1]$$

или

$$a[i/2] \geq a[i]$$

◆ 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++;
    }
}
```

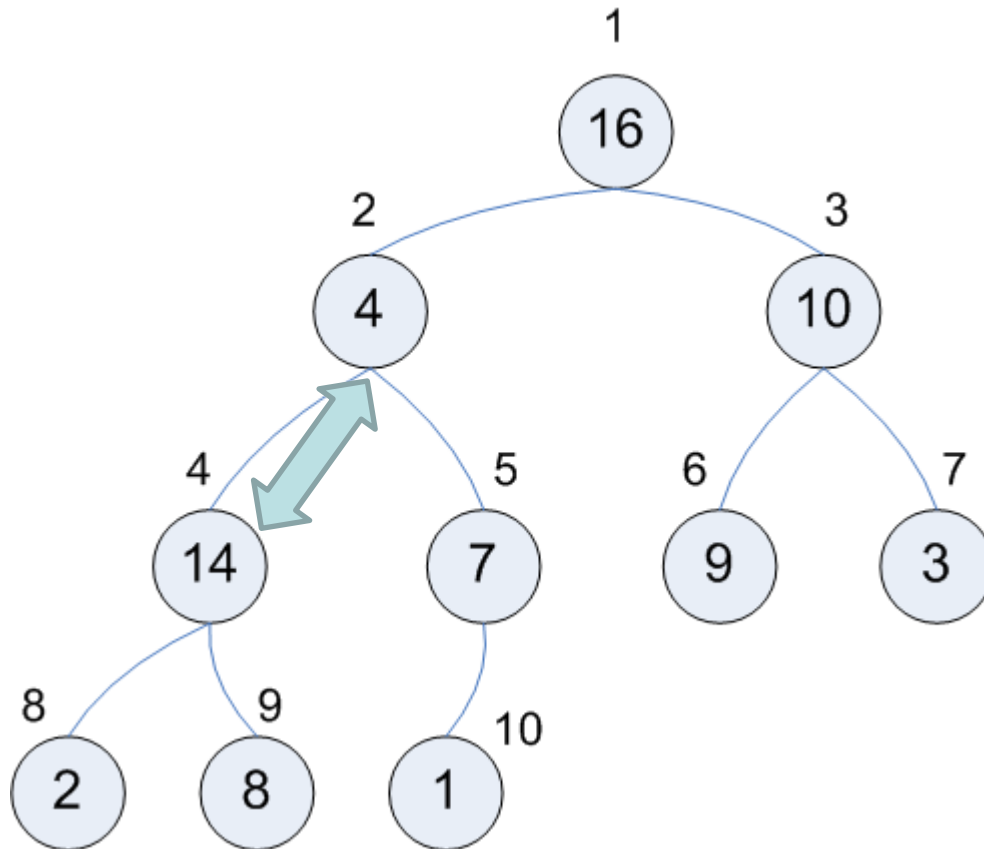
Пирамидальная сортировка: просеивание элемента

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

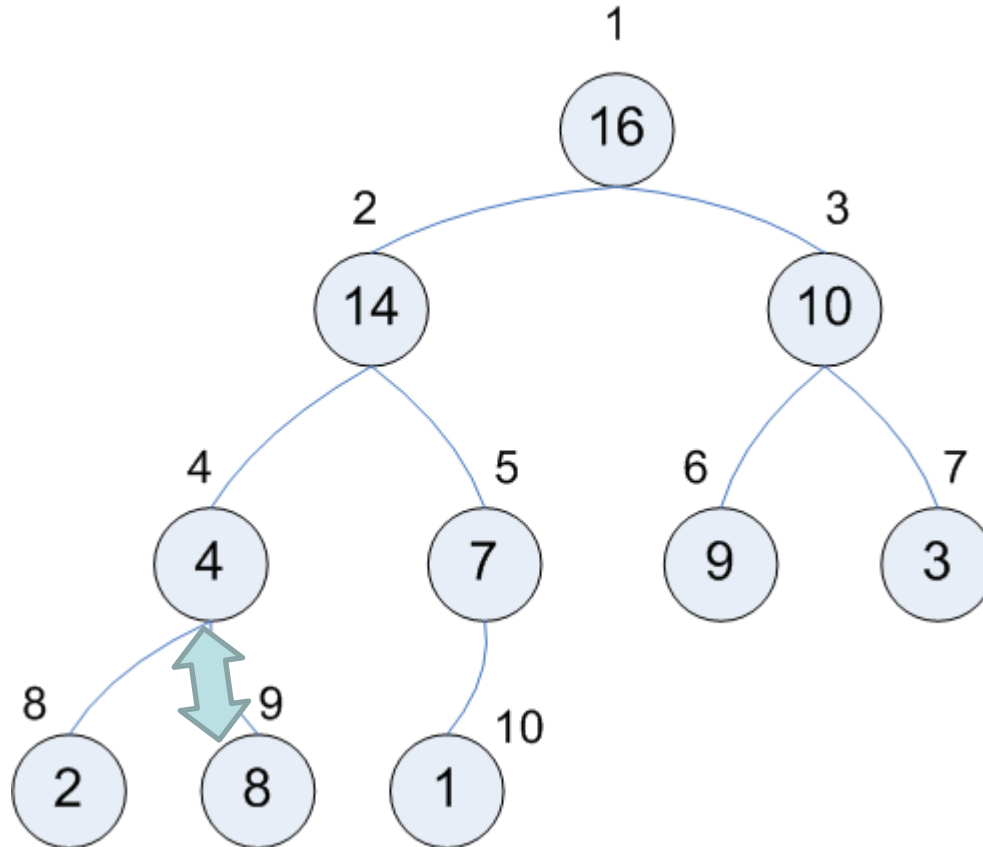
Пирамидальная сортировка: просеивание элемента

◇ Вызов sift (2, 10) для левого поддерева



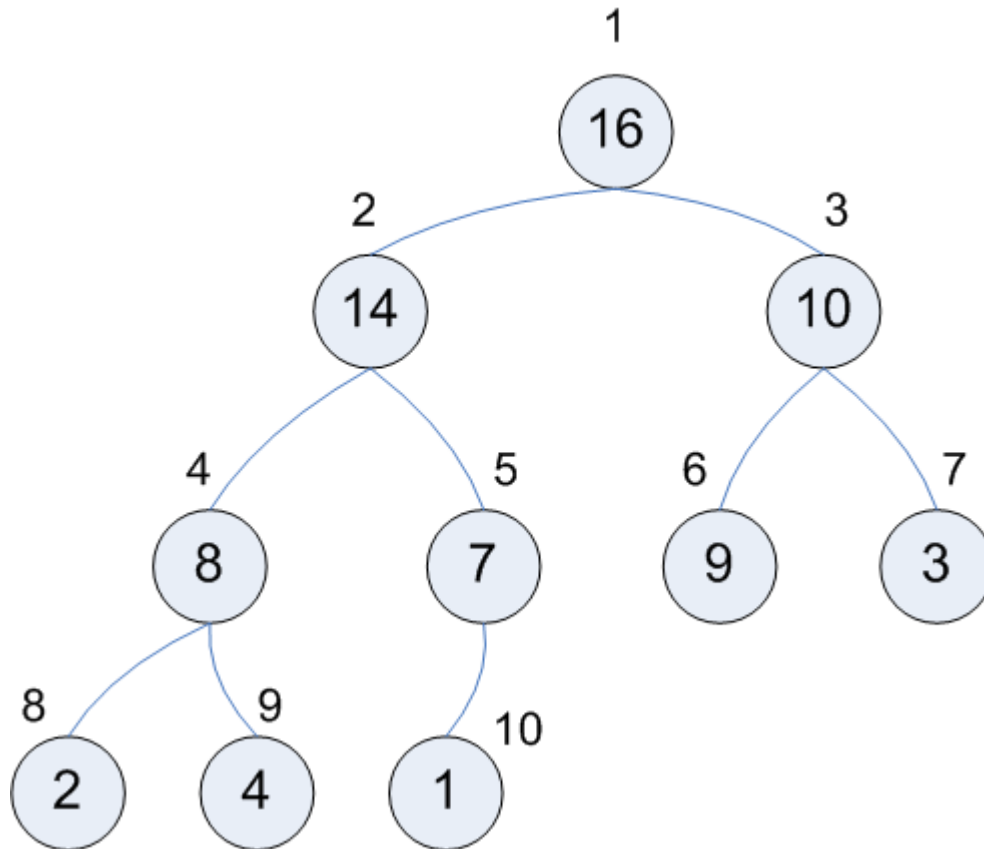
Пирамидальная сортировка: просеивание элемента

◇ Вызов sift (2, 10) для левого поддерева



Пирамидальная сортировка: просеивание элемента

◇ Вызов sift (2, 10) для левого поддерева



Пирамидальная сортировка: алгоритм

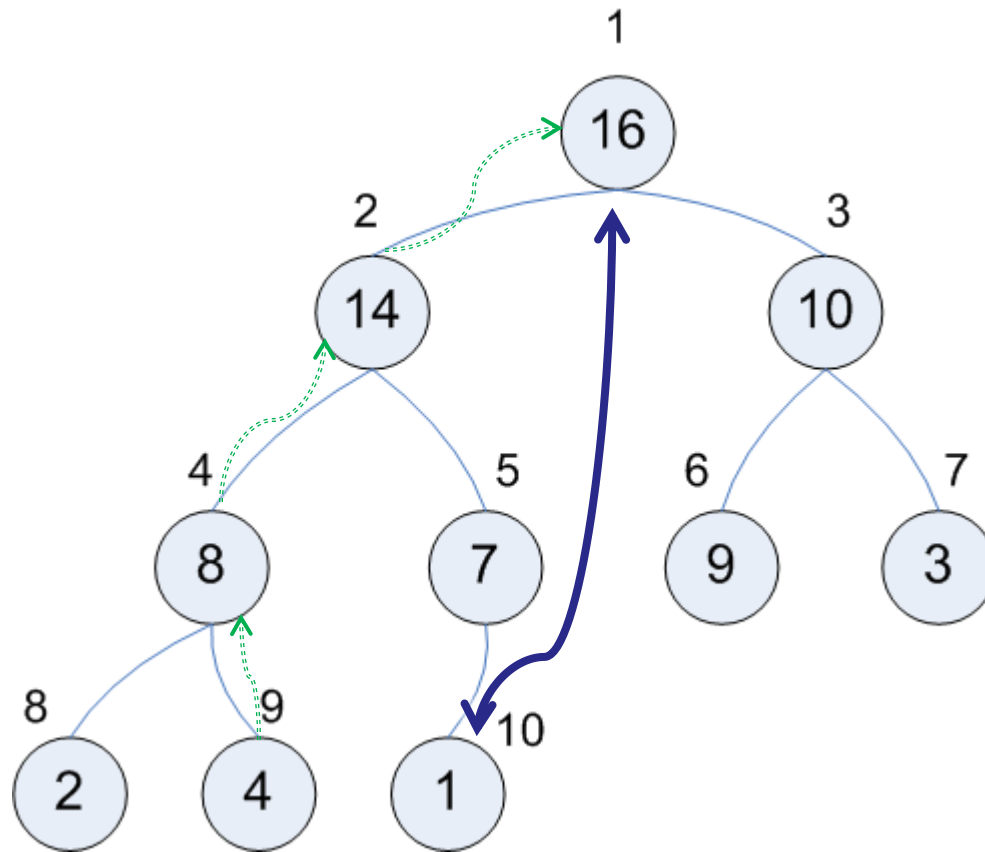
- ◆ (1) Построим пирамиду по сортируемому массиву
 - ◆ Элементы массива от $n/2$ до n являются листьями дерева, а следовательно, правильными пирамидами из одного элемента
 - ◆ Для остальных элементов в порядке уменьшения индекса просеиваем их через правую часть массива
- ◆ (2) Отсортируем массив по пирамиде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

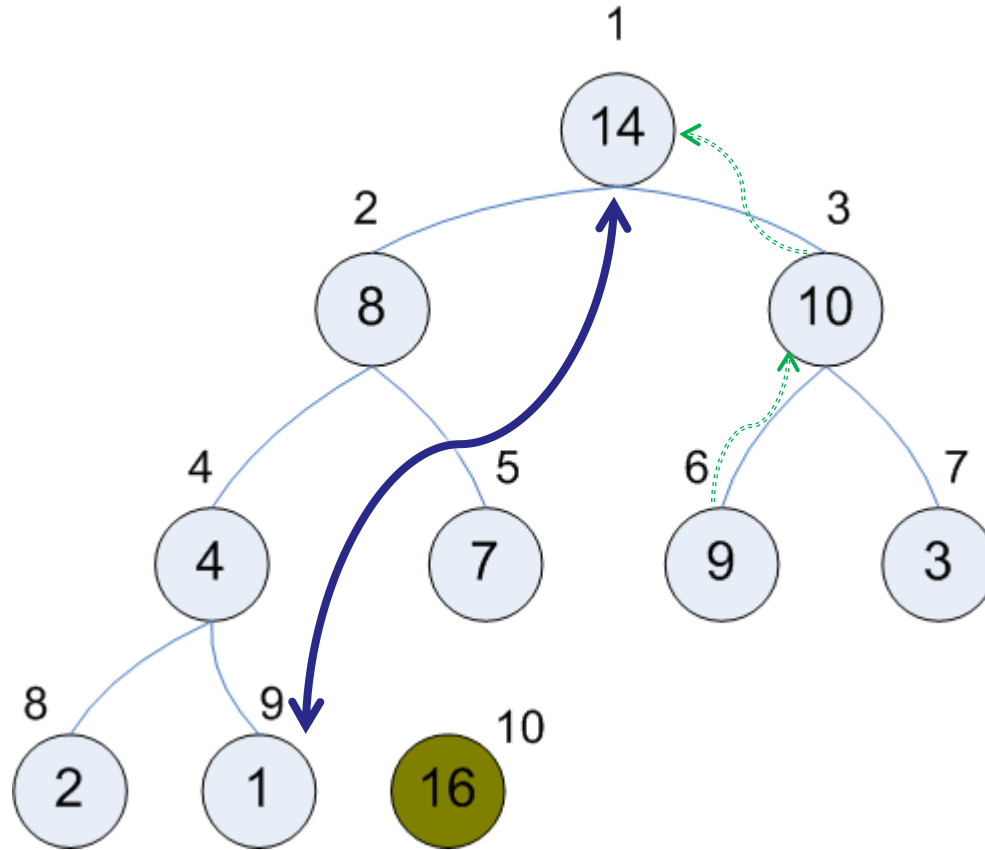
```
void heapsort (int *a, int n) {
    int i, x;

    /* 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);
    }
}
```

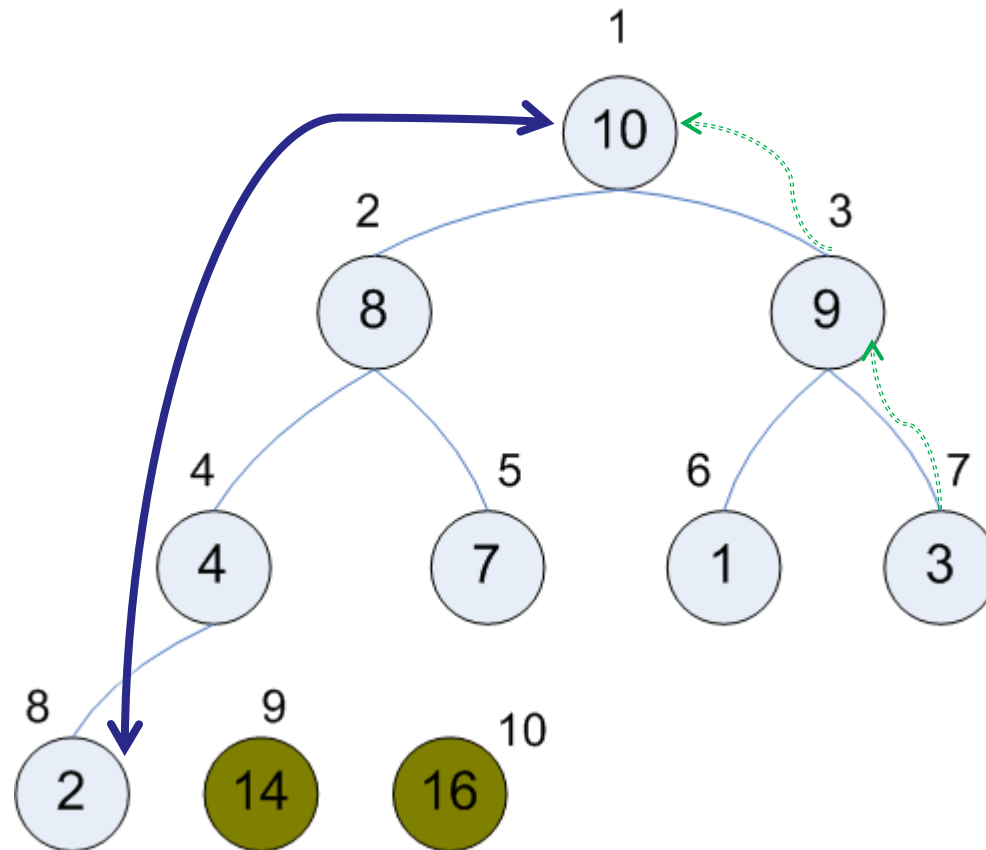
Пирамидальная сортировка: пример



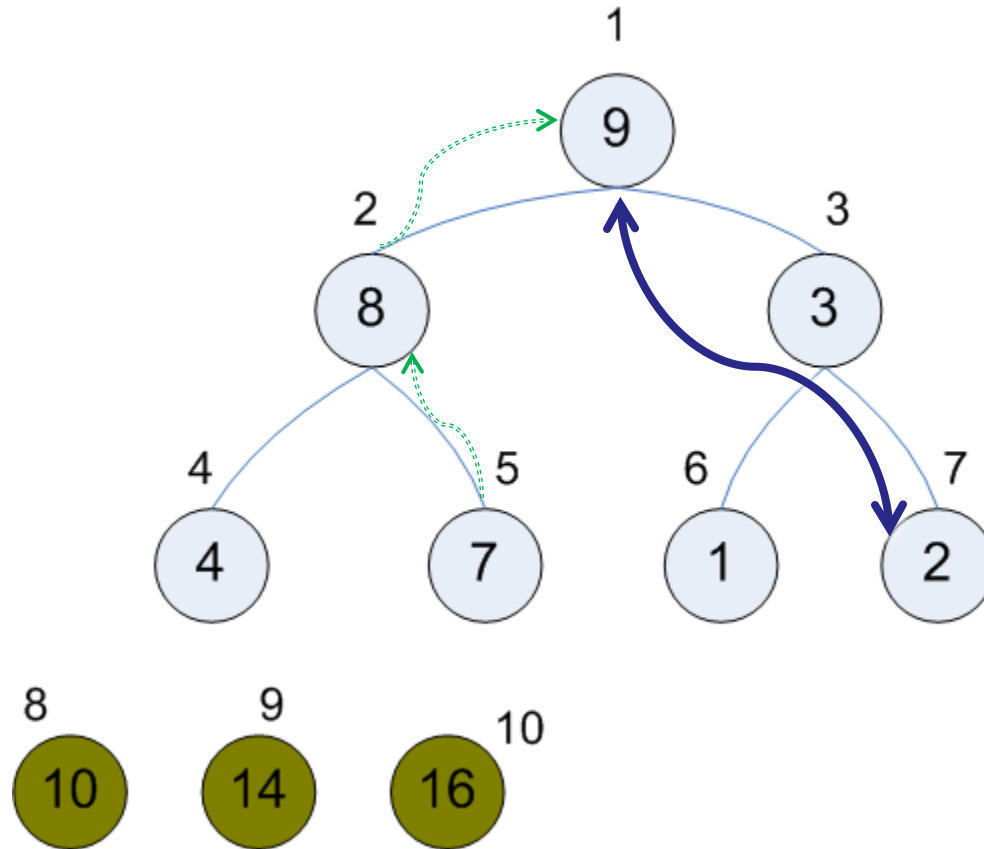
Пирамидальная сортировка: пример



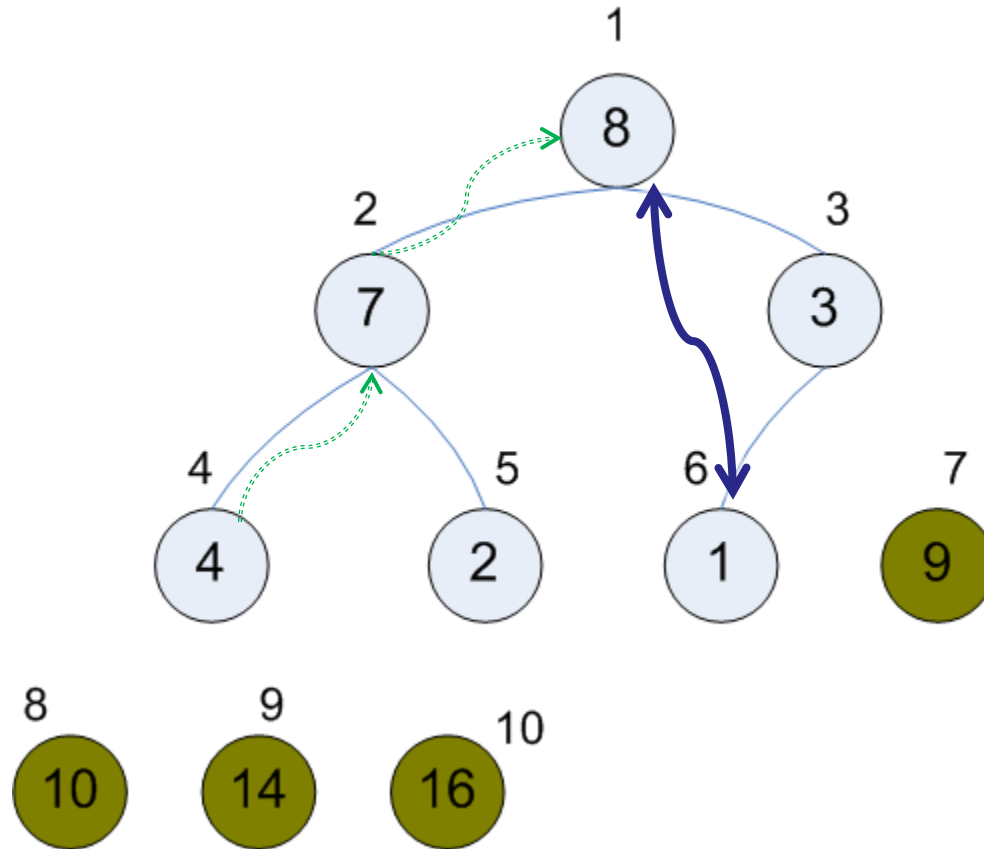
Пирамидальная сортировка: пример



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 – $n/2 * \log n$

Цифровой поиск

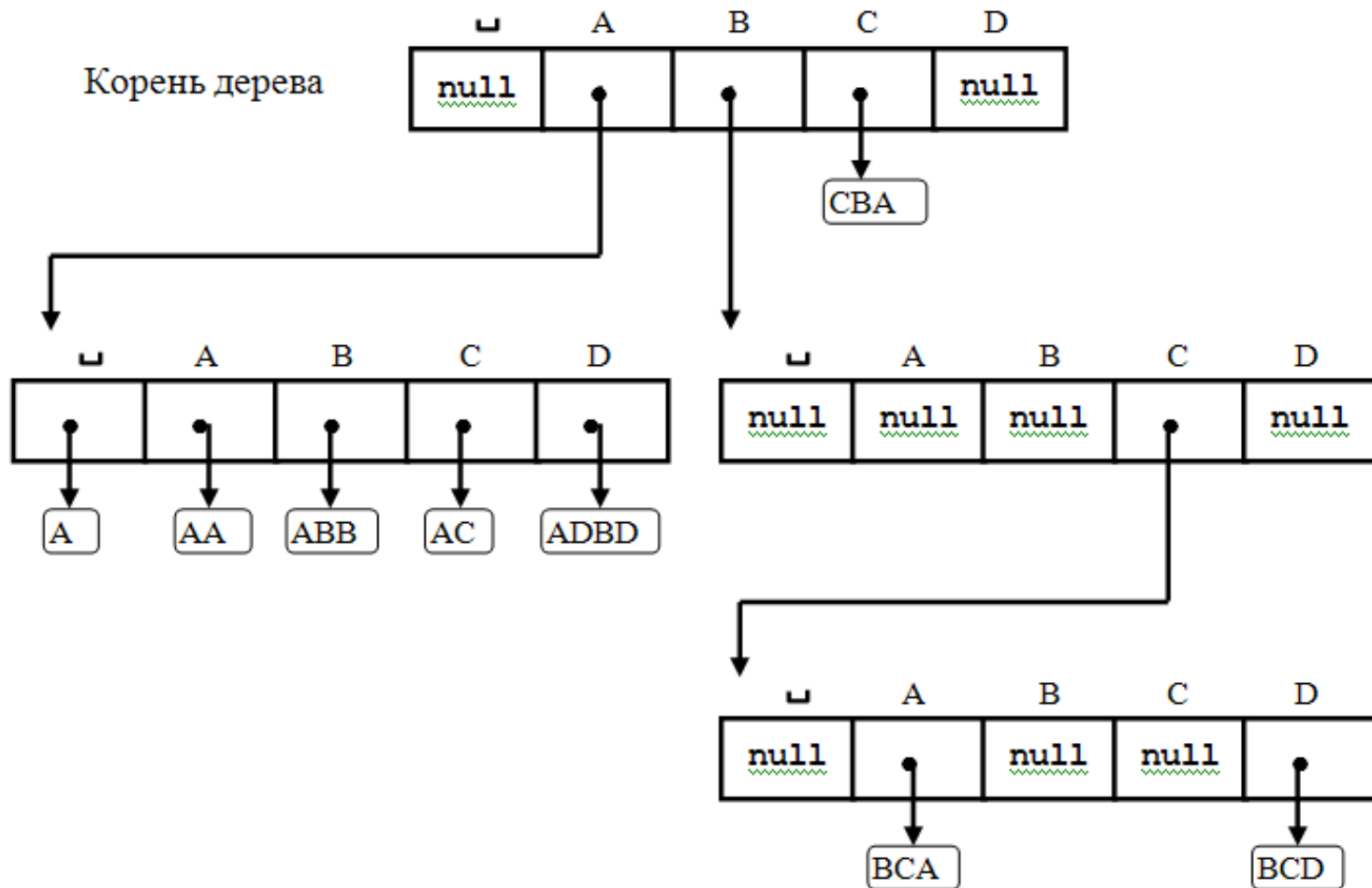
- ◇ *Цифровой поиск* – частный случай поиска заданной подстроки (*образца*) в длинной строке (*тексте*).
- ◇ *Примеры цифрового поиска*: поиск в словаре, в библиотечном каталоге и т.п., когда делается поиск по образцу в нескольких текстах (названиях книг, фамилиях авторов, текстах на вызванных сайтах и т.п.).
- ◇ Хороший пример – *словарь с высечками*, т.е. словарь, в котором обеспечен быстрый доступ к некоторым страницам (например, начальным страницам списков слов, начинающихся на очередную букву алфавита). Иногда используются *многоуровневые высечки*.
- ◇ При цифровом поиске ключи рассматриваются как последовательности символов рассматриваемого алфавита (в частности, цифр или букв). Ожидаемое число сравнений порядка $O(\log_m N)$, где m - число различных букв, используемых в словаре, N – мощность словаря. В худшем случае дерево содержит k уровней, где k – длина максимального слова.

Цифровой поиск

- ◇ *Пример.* Пусть множество используемых букв (алфавит) $\{A, B, C, D\}$. Мы добавим к алфавиту еще одну букву $_$ (пробел). По определению слова AA , $AA_$, $AA__$ совпадают. Пусть $\{A, AA, ABB, AC, ADBD, BSA, BCD, CBA\}$ – словарь (множество ключей).
- ◇ Построим m -ичное дерево, где $m = 5 = |_ , A, B, C, D|$. Следующая небольшая хитрость позволит иногда сократить поиск: если в словаре есть слово $a_1a_2a_3\dots a_k$ и первые i его букв ($i < k$) задают уникальное значение: комбинация $a_1\dots a_i$ встречается в словаре только один раз, то не нужно строить дерево для $j > i$, так как слово можно идентифицировать по первым i буквам.
- ◇ Очень важное обобщение цифрового поиска: таким же образом можно обрабатывать любые ключи, не привязываясь к байту (8 битам), который обычно используется для кодирования символов алфавита. Мы можем отсекать от ключа первые m бит, использовать 2^m -ичное разветвление, т.е. строить 2^m -ичное дерево поиска (на двоичных деревьях для разветвления берется один бит: $m = 1$).

Цифровой поиск

- ♦ Прямоугольниками изображены вершины дерева, в овалах – значения слов (ключей) и связанная с ним информация. Тем самым любая вершина дерева – массив из m элементов. Каждый элемент вершины содержит либо ссылку на другую вершину m -ичного дерева, либо на овал (ключ).



Цифровой поиск

- ◇ Иногда используют комбинации нескольких методов: цифровой поиск вначале, а затем переключение на поиск в последовательных таблицах.
 - ◆ Именно так мы и работаем со словарем с высечками: вначале на высечку, а затем либо последовательный поиск, либо дихотомический.
- ◇ Обычно предлагается пользоваться цифровым поиском, пока количество различных слов не меньше некоторого k , а затем переключаться на последовательные таблицы.
- ◇ Обобщения: поиск по неполным ключам, поиск по образцу.
- ◇ Варианты:
 - ◆ Не строить промежуточных узлов из одного разветвления, вместо этого хранить индекс следующего символа с нетривиальным разветвлением
 - ◆ Писать символы ключа на ребрах (“бор/сжатый бор”)

Цифровой поиск

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define M 5

typedef enum {word, node} tag_t;
struct record {
    char *key;
    int value;
};

struct tree {
    tag_t tag;
    union {
        struct record *r;
        struct tree *nodes[M+1];
    }; /* анонимное объединение */
};
```

Цифровой поиск: поиск элемента

```
static inline int ord (char c) {
    return c ? c - 'A' + 1 : 0; // ASCII-only
}

struct record *find (struct tree *t, char *key) {
    int i = 0;

    while (t) {
        switch (t->tag) {
            case word:
                for (; key[i]; i++)
                    if (key[i] != t->r->key[i])
                        return NULL;
                return t->r->key[i] ? NULL : t->r;
            case node:
                t = t->nodes[ord(key[i])];
                if (key[i])
                    i++;
        }
    }
    return NULL;
}
```


Цифровой поиск: вставка – вспомогательные функции

```
struct record *make_record (char *key, int value) {
    struct record *r = malloc (sizeof (struct record));
    r->key = strdup (key);
    r->value = value;
    return r;
}
```

```
struct tree *make_word (char *key, int value) {
    struct tree *t = malloc (sizeof (struct tree));
    t->tag = word;
    t->r = make_record (key, value);
    return t;
}
```

```
struct tree *make_node (void) {
    struct tree *t = calloc (1, sizeof (struct tree));
    t->tag = node;
    return t;
}
```

```
struct tree *make_from_record (struct record *r) {
    struct tree *t = malloc (sizeof (struct tree));
    t->tag = word;
    t->r = r;
    return t;
}
```

Цифровой поиск: вставка элемента

```
struct tree *insert (struct tree *t, char *key, int value) {
    if (!t)
        return make_word (key, value);

    int i = 0;
    struct tree *root = t;

    /* skip all nodes */
    while (t->tag == node) {
        struct tree **p = &t->nodes[ord(key[i++])];
        if (!*p) {
            *p = make_word (key, value);
            return root;
        }
        t = *p;
    }

    /* all word skipped -- key exists, update value */
    if (i && !key[i - 1]) {
        t->r->value = value;
        return root;
    }
}
```

Цифровой поиск: вставка элемента

```
/* compare the remaining part */
int j = i;
for (; key[i]; i++)
    if (key[i] != t->r->key[i])
        break;

/* key already exists -- update value */
if (!key[i] && !t->r->key[i]) {
    t->r->value = value;
    return root;
}

/* turn t into a node */
struct record *other = t->r;
t->tag = node;
memset (t->nodes, 0, sizeof (t->nodes));
```

Цифровой поиск: вставка элемента

```
/* make new nodes for remaining common prefix */
for (; j < i; j++) {
    struct tree *p = make_node ();
    t->nodes[ord(key[j])] = p;
    t = p;
} // Дома. Реализуйте поиск и вставку для варианта
// дерева, в котором нет тривиальных разветвлений
// (вместо цикла выше добавляется один узел
// на всю оставшуюся длину общего префикса)

/* accommodate both other and new record */
t->nodes[ord(other->key[i])]
    = make_from_record (other);
t->nodes[ord(key[i])] = make_word (key, value);
return root;
}
```

Цифровой поиск: печать элементов

```
void print (struct tree *t, char c) {
    static int level = 0;
    if (!t) {
        printf ("empty\n");
        return;
    }
    for (int i = 0; i < level; i++)
        putchar (' ');
    if (level)
        printf ("%c: ", chr (c));
    if (t->tag == word) {
        printf ("word: %s %d\n", t->r->key, t->r->value);
    } else {
        printf ("node: ");
        for (int i = 0; i < M + 1; i++)
            if (t->nodes[i])
                printf ("%c ", chr(i));
        putchar ('\n');
        level++;
        for (int i = 0; i < M + 1; i++)
            if (t->nodes[i])
                print (t->nodes[i], i);
        level--;
    }
}
```