

**Курс «Алгоритмы и алгоритмические языки»  
1 семестр 2018/2019**

**Лекция 24**

## 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  
 $a[i/2] \leq a[i]$ 
  - ◆ 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)

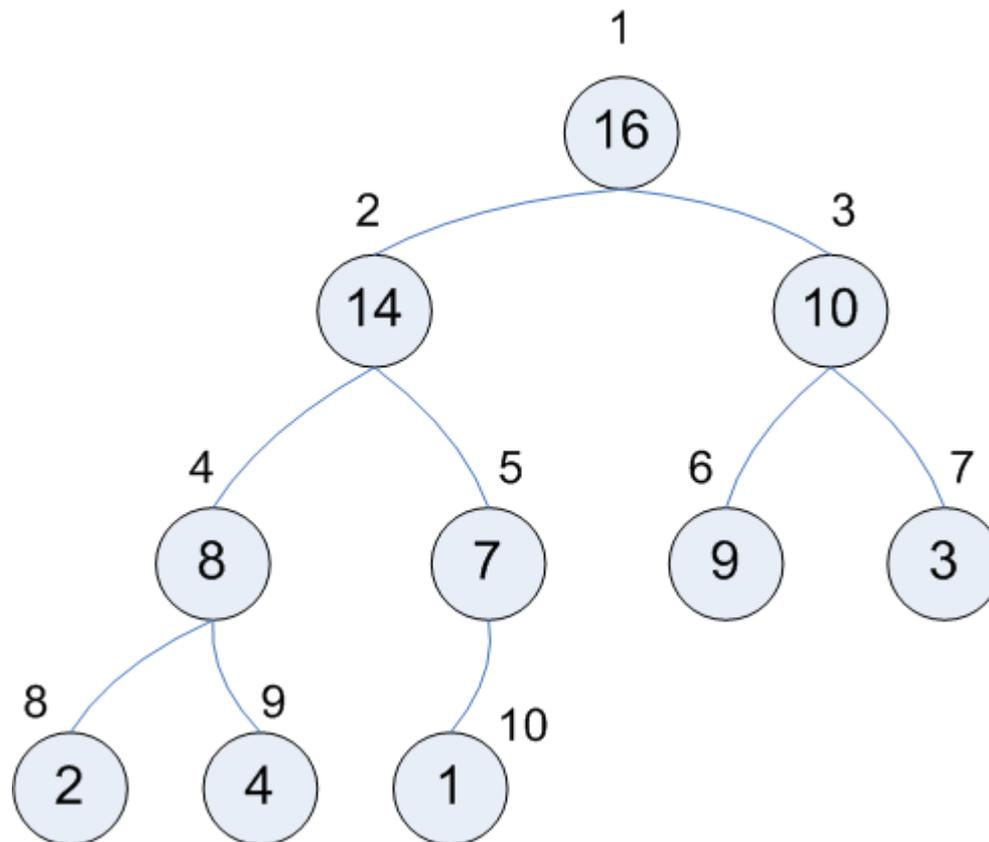
◇ Для всех элементов пирамиды выполняется соотношение:

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

или

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

◆ 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

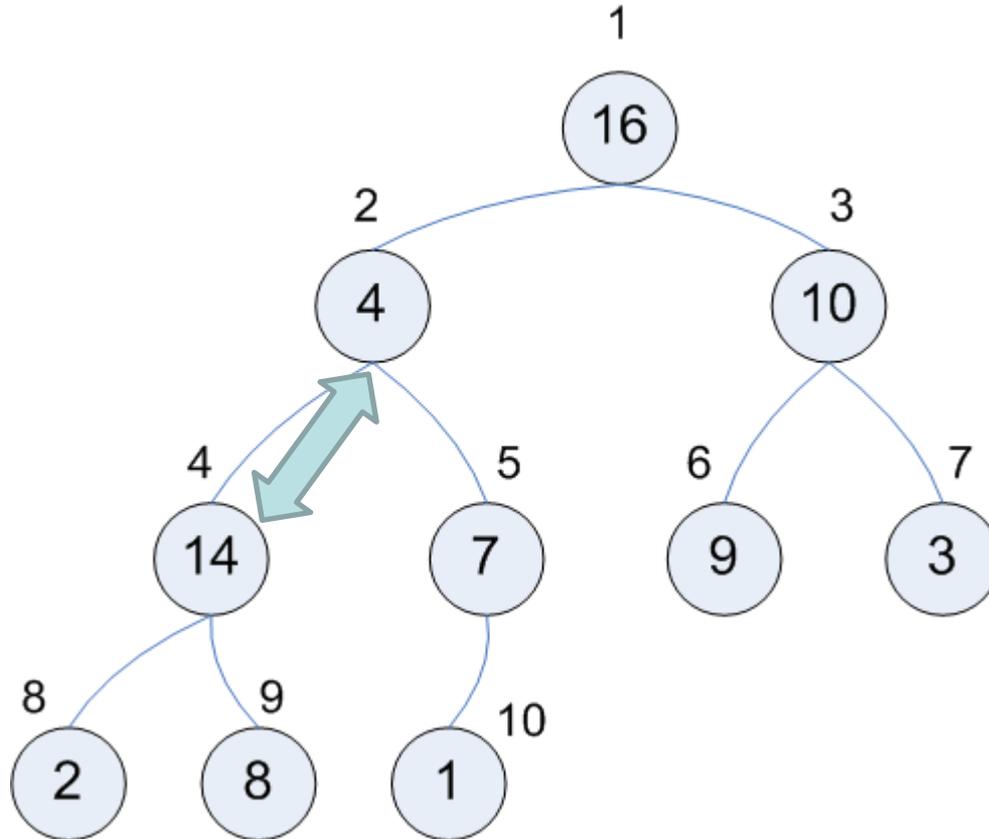
## Пирамидальная сортировка: просеивание элемента

```
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++;
    }
}
```

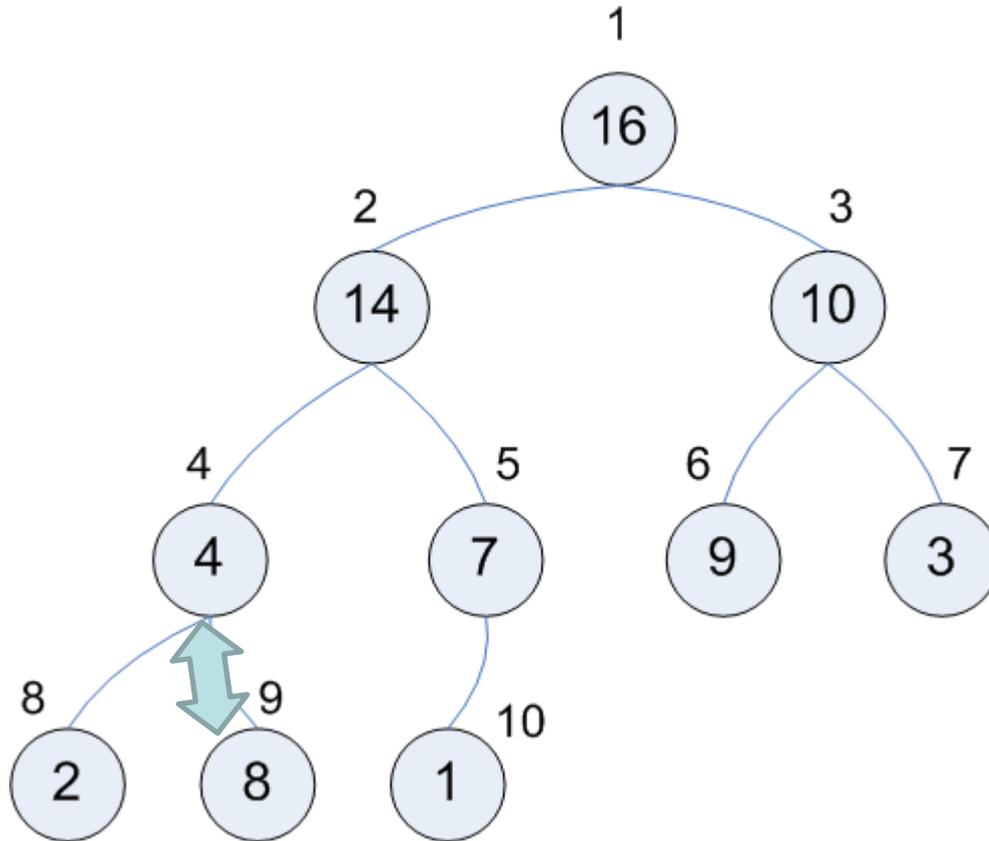
# Пирамидальная сортировка: просеивание элемента

◇ Вызов sift (2, 10) для левого поддерева



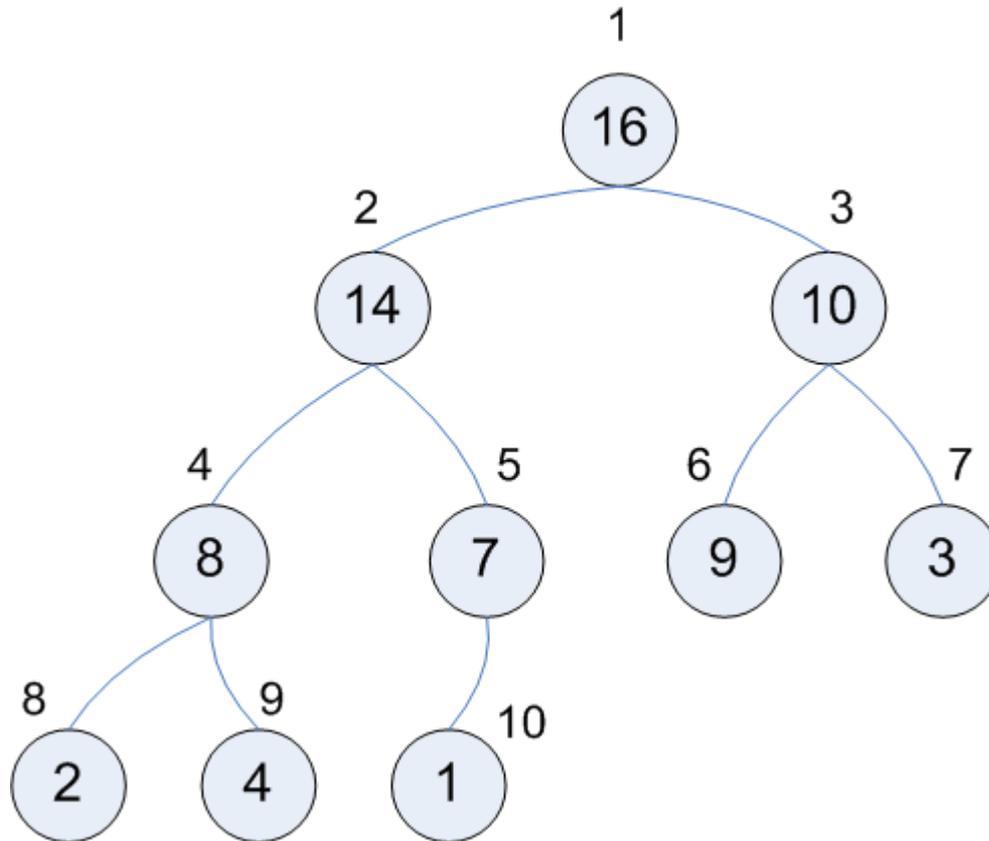
# Пирамидальная сортировка: просеивание элемента

◇ Вызов sift (2, 10) для левого поддерева



# Пирамидальная сортировка: просеивание элемента

◇ Вызов sift (2, 10) для левого поддерева



## ***Пирамидальная сортировка: алгоритм***

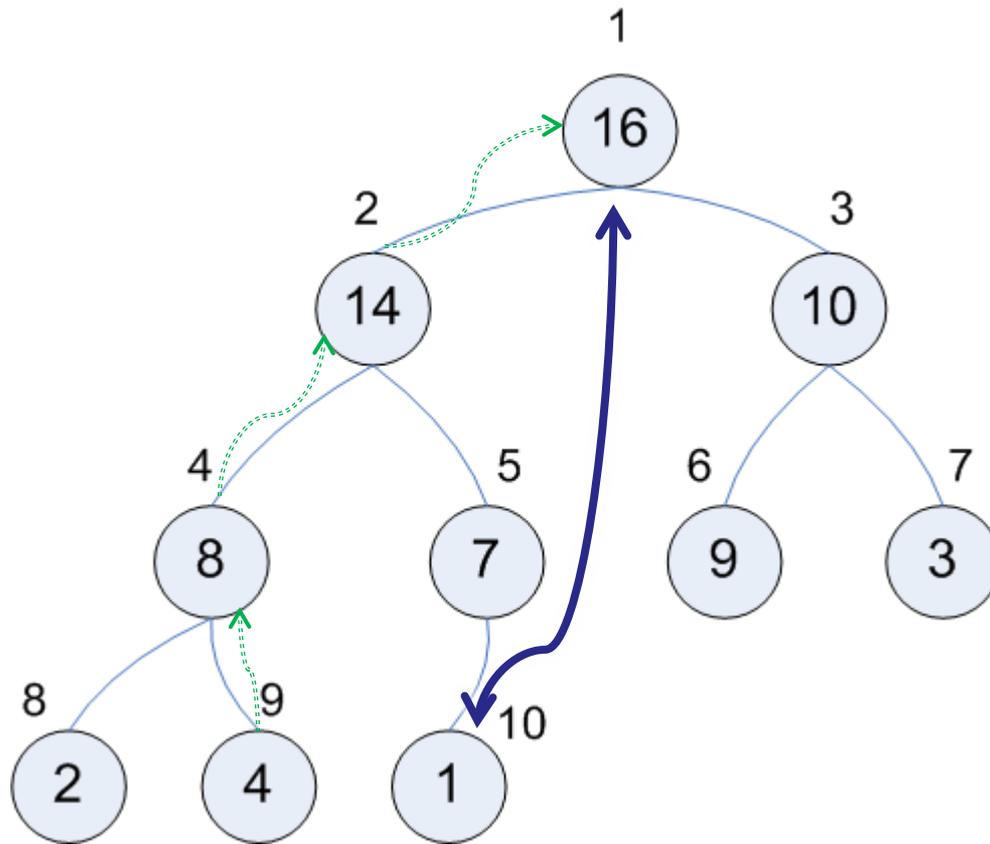
- ◆ (1) Построим пирамиду по сортируемому массиву
  - ◆ Элементы массива от  $n/2$  до  $n$  являются листьями дерева, а следовательно, правильными пирамидами из одного элемента
  - ◆ Для остальных элементов в порядке уменьшения индекса просеиваем их через правую часть массива
- ◆ (2) Отсортируем массив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

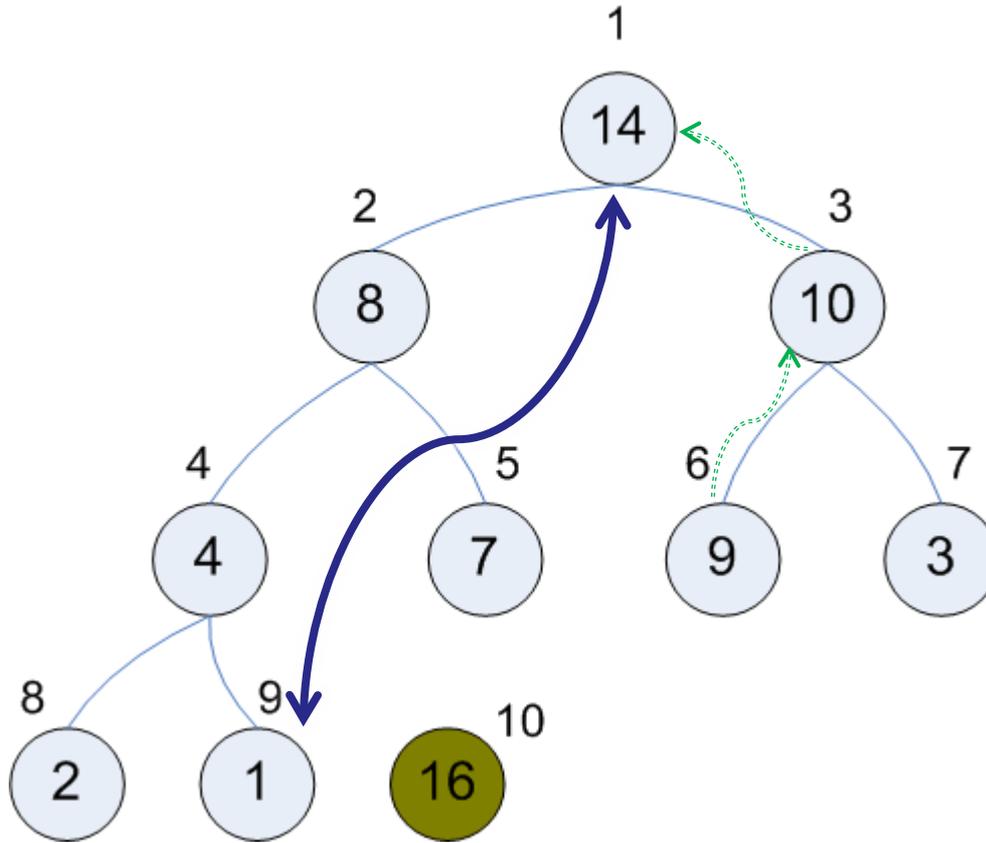
```
void heapsort (int *a, int n) {
    int i, x;

    /* 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);
    }
}
```

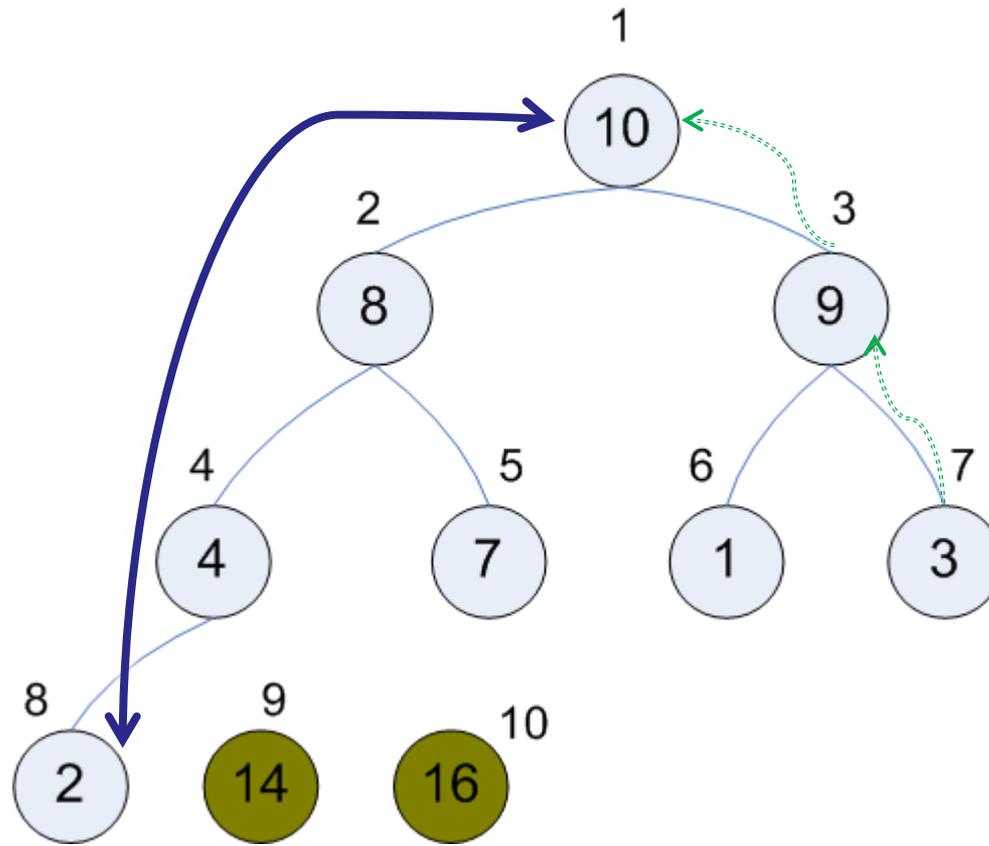
# Пирамидальная сортировка: пример



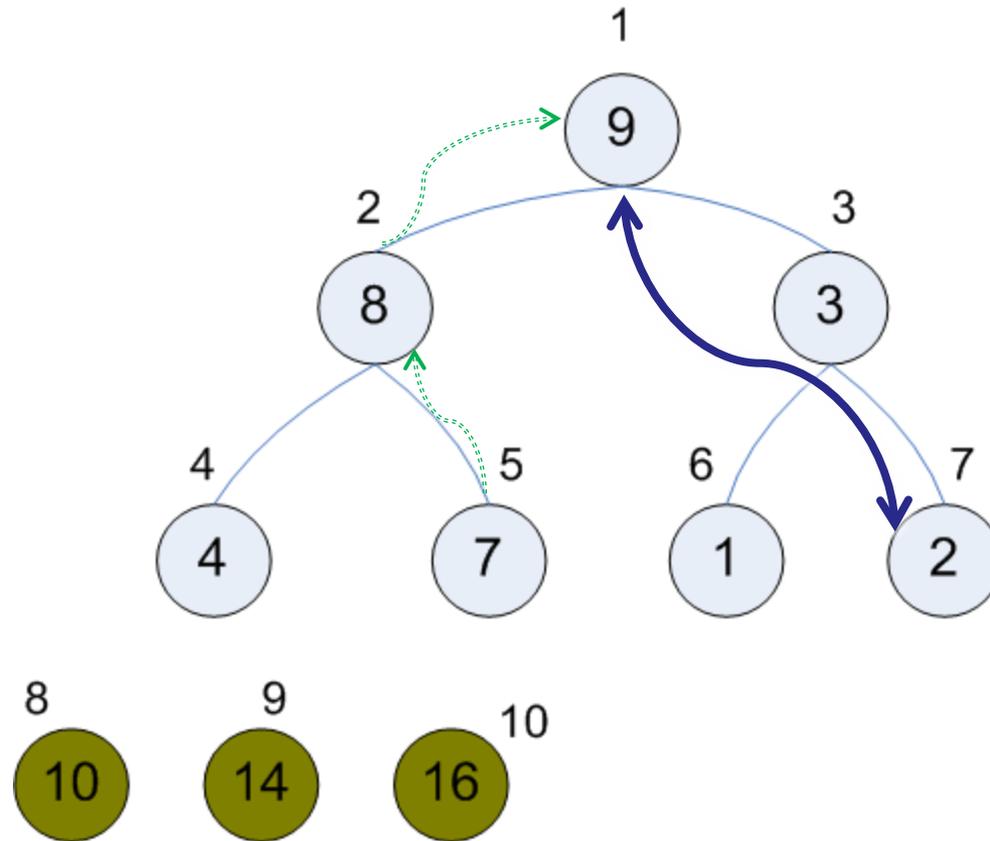
# Пирамидальная сортировка: пример



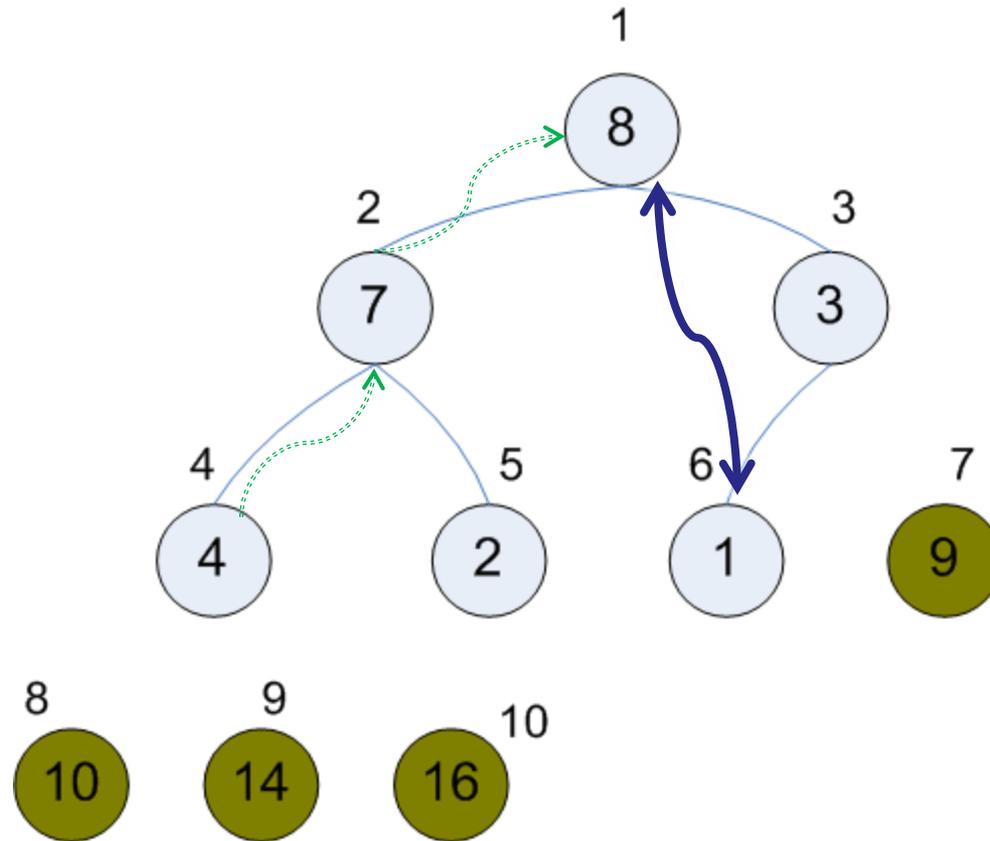
# Пирамидальная сортировка: пример



# 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 –  $n/2 * \log n$

# Хеш-таблицы

- ◆ Словарные операции: *добавление*, *поиск* и *удаление* элементов по их ключам.
- ◆ Организуется таблица ключей: массив **Index[m]** длины **m**, элементы которого содержат значение ключа и указатель на данные (информацию), соответствующие этому ключу.
  - ◆ **Прямая адресация.** Применяется, когда количество возможных ключей невелико: например, ключи перенумерованы целыми числами из множества  $U = \{0, 1, 2, \dots, m - 1\}$ , где  $m$  не очень большое число.
  - ◆ В случае прямой адресации ключ с номером  $k$  соответствует элементу **Index[k]**. Этот ключ обычно не записывается в элемент массива, т.к. совпадает с индексом.
  - ◆ Все три словарные операции выполняются за время порядка  $O(1)$ .
  - ◆ Основной недостаток прямой адресации – таблица *Index* занимает слишком много места, если множество всевозможных ключей  $U$  достаточно велико ( $m$  большое целое число).

# Хеш-таблицы

- ◇ **Хеширование** тоже позволяет обеспечить среднее время операций с данными  $T_{\text{cp}}(n) = O(1)$  и тоже за счет использования таблицы *Index*.
- ◇ Хеш-таблица использует память объемом  $\Theta(|K|)$ , где  $|K|$  – мощность множества использованных ключей (правда, это оценка в среднем, а не в худшем случае, да и то при определенных предположениях).
- ◇ В случае хеш-адресации элементу с ключом *key* отводится строка таблицы с номером  $\text{hash}(\text{key})$ , где  $\text{hash}: U \rightarrow \{0, 1, 2, \dots, m - 1\}$  – хеш-функция. Число  $\text{hash}(\text{key})$  называется *хеш-значением* ключа *key*.
- ◇ Если хеш-значения ключей  $\text{key}_1$  и  $\text{key}_2$  совпадают ( $\text{hash}(\text{key}_1) == \text{hash}(\text{key}_2)$ ), говорят, что случилась *коллизия*. Выбрать хеш-функцию, для которой коллизии исключены, возможно лишь тогда, когда все возможные значения ключей заранее известны. В общем же случае коллизии неизбежны, так как  $|U| > m$ .

# Хеш-таблицы

- ◇ Простейший способ обработки коллизий – сцепление элементов с одинаковыми значениями хеш-функции: все такие элементы сцепляются в список, а в хеш-таблицу помещается указатель на первый элемент этого списка. В пределах каждого такого списка осуществляется последовательный поиск.
- ◇ В случае использования двусвязного списка среднее время выполнения каждой из трех словарных операций будет иметь порядок  $O(1)$ . Основная трудность – в поиске по списку, но коллизий не очень много и  $hash(key)$  можно выбрать так, чтобы списки были достаточно короткими.
- ◇ Примером хеш-таблицы с цепочками является записная книжка с алфавитом.

# Хеш-таблицы

- ◆ Устройство простой хеш-таблицы (реализация хеширования с цепочками).
  - ◆ Задается некоторое фиксированное число  $m$  (типичные значения  $m$  от 100 до 1,000,000).
  - ◆ Создается массив **Index[m]** указателей начал двусвязных списков (цепочек), который называется *индексом* хеш-таблицы. В начале работы все указатели имеют значения *NULL*.
  - ◆ Задается хеш-функция  $hash()$ , которая получает на вход ключи и выдает значение от 0 до  $m - 1$ .
  - ◆ При добавлении пары ( $key, value$ ) вычисляется  $h = hash(key)$  и пара добавляется в список **Index[h]**.
  - ◆ При удалении либо поиске пары ( $key, value$ ) вычисляется  $h = hash(key)$  и происходит удаление либо поиск пары ( $key, value$ ) в списке **Index[h]**.

# Хеш-таблицы



Анализ хеширования с цепочками.

- ◆ Пусть **Index[m]** – хеш-таблица с  $m$  позициями, в которую занесено  $n$  пар (*key*, *value*). Отношение  $\alpha = n/m$  называется *коэффициентом заполнения* хеш-таблицы.

- ◆ Коэффициент заполнения  $\alpha$  позволяет судить о качестве хеш-функции:

пусть  $M = \frac{1}{m} \sum_{i=0}^{m-1} |Index[i]|$  – средняя длина списков;

если  $hash(key)$  – «хорошая» хеш-функция, то

дисперсия  $D = \frac{1}{m} \sum_{i=0}^{m-1} (M - |Index[i]|)^2 \leq \alpha$ .

- ◆ Это условие исключает наихудший случай, когда хеш-значения всех ключей одинаковы, заполнен только один список и поиск в этом списке из  $n$  элементов имеет среднее время  $\Theta(n)$ .

# Хеш-таблицы

- ◆ Анализ хеширования с цепочками.
  - ◆ *Равномерное хеширование*: хеш-функция подобрана таким образом, что каждый данный элемент может попасть в любую из  $t$  позиций хеш-таблицы с равной вероятностью, независимо от того, куда попали другие элементы.
  - ◆ Условие из предыдущего слайда выполняется и *средняя длина каждого из  $t$  списков хеш-таблицы с коэффициентом заполнения  $\alpha$  равна  $\alpha$* .
  - ◆ Среднее время поиска элемента, отсутствующего в таблице, пропорционально средней длине списка  $\alpha$ , так как поиск сводится к просмотру одного из списков.
  - ◆ Поскольку среднее время вычисления хеш-функции равно  $\Theta(1)$ , то среднее время выполнения каждой из словарных операций с учетом вычисления хеш-функции равно  $\Theta(1 + \alpha)$ .

# Хеш-таблицы

- ◇ **Теорема.** Пусть  $T$  – хеш-таблица с цепочками, имеющая коэффициент заполнения  $\alpha$ , причем хеширование равномерно. Тогда при поиске элемента, **отсутствующего** в таблице, будет просмотрено в среднем  $\alpha$  элементов таблицы, а время поиска, включая время на вычисление хеш-функции, будет равно  $\Theta(1 + \alpha)$ .
- ◇ **Теорема.** При равномерном хешировании среднее время **успешного** поиска в хеш-таблице с коэффициентом заполнения  $\alpha$  есть  $\Theta(1 + \alpha)$ .
  - ◆ **Замечание.** Теорема не сводится к предыдущей, так как в предыдущей теореме оценивалось среднее число действий, необходимых для поиска случайного элемента, равновероятно попадающего в любую из ячеек таблицы.
  - ◆ В этой теореме сначала рассматривается случайно выбранная последовательность элементов, добавляемых в таблицу (на каждом шаге все значения ключа равновероятны и шаги независимы); потом в полученной таблице выбираем элемент для поиска, считая, что все ее элементы равновероятны.
- ◇ Из теорем следует, что в случае равномерного хеширования среднее время выполнения любой словарной операции есть  $O(1)$ .

# Методы построения хеш-функций

- ◆ Построение хеш-функции **методом деления с остатком**.
  - ◆ Хеш-функция  $hash(key)$  определяется соотношением  **$hash(key) = key \% m$** .
  - ◆ При правильном выборе  $m$  такая хеш-функция обеспечивает распределение, близкое к равномерному.
  - ◆ Правильный выбор  $m$ : в качестве  $m$  выбирается достаточно большое простое число, далеко отстоящее от степеней двойки.
  - ◆ Например, если устраивает средняя длина списков 3, а число записей, доступ к которым нужно обеспечить с помощью хеш-таблицы  $\approx 2000$ , то можно взять  $m = 2000/3 \approx 701$ . Тогда  $hash(key) = key \% 701$ .
  - ◆ Недостаток: в качестве  $m$  нельзя брать степень двойки, так как если  $m = 2^p$ , то  $hash(key)$  – это просто  $p$  младших битов числа  $key$ .

# Методы построения хеш-функций

- ◆ Построение хеш-функции **методом умножения**.
  - ◆ Пусть количество хеш-значений равно  $m$ . Выберем и зафиксируем вещественную константу  $v$ ,  $0 < v < 1$ ; положим  $hash(key) = \lfloor m \cdot frac(key \cdot v) \rfloor$   
 $frac(key \cdot v)$  – дробная часть числа  $key \cdot v$ .
  - ◆ Достоинство метода умножения в том, что качество хеш-функции слабо зависит от выбора  $m$ . Обычно в качестве  $m$  выбирают степень двойки, так как в этом случае умножение на  $m$  сводится к сдвигу.
  - ◆ **Пример.** Пусть в используемом компьютере длина слова равна  $w$  битам и ключ  $key$  помещается в одно слово.
  - ◆ Если  $m = 2^p$ , то вычисление  $hash(key)$  можно выполнить следующим образом: умножим  $key$  на  $w$ -битовое целое число  $v \cdot 2^w$ ; получится  $2w$ -битовое число  $r_0$ .  
В качестве значения  $hash(key)$  возьмем старшие  $p$  битов “дробной” части числа  $r_0 / 2^w$  ( $r_0 \% 2^w$  или обнуление  $w$  старших разрядов, потом умножение на  $m = 2^p$ ).
  - ◆ Согласно Д. Кнуту выбор  $v = (\sqrt{5} - 1) / 2 = 0.6180339887\dots$  является удачным.

## ***Хеш-функции: программы***

```
#define MAX 701    /* размер хеш-таблицы */
struct htype {
    int key;        /* ключ */
    int val;        /* значение элемента данных */
    struct htype *next; /* указатель на следующий элемент
                        цепочки */
    struct htype *prvs; /* указатель на предыдущий элемент
                        цепочки */
};
struct htype *index[MAX];
```

## ***Хеш-функции: программы***

```
#define MAX 701    /* размер хеш-таблицы */
static inline int hash (int key) {
    return key % MAX;
}
/* инициализация хеш-таблицы */
void init (void) {
    int i;
    for (i = 0; i < MAX; i++)
        index[i] = NULL; /* массив начал цепочек */
}
```

## Хеш-функции: программы

```
/* Вычисление хеш-адреса и поиск по ключу k:  
   если элемент с ключом k найден, возвращаем указатель  
   на него, если нет, возвращаем NULL */
```

```
struct htype *search (int k) {  
    /* вычисление хеш-адреса */  
    int h = hash (k);  
    /* поиск ключа k */  
    if (index[h]) {  
        struct htype *p = index[h];  
        do {  
            if (p->key == k)  
                return p;  
            else  
                p = p->next;  
        } while (p);  
    }  
    return NULL;  
}
```

## ***Хеш-функции: программы***

```
/* Порождение нового элемента цепочки и возврат указателя  
на него */
```

```
struct htype *new (void) {  
    struct htype *p;  
    p = malloc (sizeof (struct htype));  
    if (!p)  
        abort ();  
    p->key = -1;  
    p->val = 0;  
    p->next = NULL;  
    p->prvs = NULL;  
    return p;  
}
```

## Хеш-функции: программы

```
/* Вычисление хеш-адреса и поиск по ключу k: если элемент с ключом k найден, возвращаем значение true и указатель на найденный элемент; если элемент не найден, возвращаем значение false и указатель на последний элемент либо NULL, если цепочка пустая */
```

```
static bool search_internal (int k, struct htype **r) {  
    struct htype *p, *q;  
  
    if ((p = index[hash (k)]) != NULL) {  
        do {  
            if (p->key == k) {  
                *r = p;  
                return true;  
            }  
            else  
                q = p, p = p->next;  
        } while (p);  
        *r = q;  
    } else  
        *r = NULL;  
    return false;  
}
```

## Хеш-функции: программы

```
/* Добавление новой пары (key, value) */
void insert (int k, int v) {
    struct htype *p, *q;
    /* Если элемент с ключом k уже имеется в цепочке,
       изменяем его значение на v */
    if (search_internal (k, &p))
        p->val = v;
    else {
        /* Если элемента с ключом k в цепочке нет */
        /* порождение и инициализация нового элемента цепочки */
        q = new ();
        q->key = k;
        q->val = v;
        /* Включение порожденного элемента в цепочку */
        if (p) {
            p->next = q;
            q->prvs = p;
        } else
            index[hash (k)] = q;
    }
}
```

## Хеш-функции: программы

```
/* Исключение пары (key, value) */  
void delete (int k, int v) {  
    struct htype *p;  
    if (search_internal (k, &p)) {  
        if (p->prvs)  
            p->prvs->next = p->next;  
        else  
            index[hash (k)] = p->next;  
        if (p->next)  
            p->next->prvs = p->prvs;  
        free (p);  
    }  
    /* иначе ничего не нашли, удалять не нужно */  
}  
  
// Дома. Сделайте так, чтобы хеш-функция не вычислялась  
    дважды (внутри search_internal и внутри insert/delete).
```

## Хеширование с открытой адресацией

- ◇ Все записи хранятся в самой хеш-таблице: каждая ячейка таблицы (массива длины  $m$ ) содержит либо хранимый элемент, либо `NULL`. Указатели вообще не используются, что приводит к сохранению места и ускорению поиска.
- ◇ Таким образом, коэффициент заполнения  $\alpha = n/m$  не больше 1.
- ◇ **Поиск (search)**: мы определенным образом просматриваем элементы таблицы, пока не найдем искомый или не убедимся, что искомый элемент отсутствует.
- ◇ Просматриваются не все элементы (иначе это был бы последовательный поиск), а только некоторые согласно значению хеш-функции, которая в этом случае имеет два аргумента – ключ и «номер попытки»:

$$\text{hash}: U \times \{0, 1, \dots, m - 1\} \rightarrow \{0, 1, \dots, m - 1\}.$$

- ◇ Функцию *hash* нужно выбрать такой, чтобы в последовательности проб  $\langle \text{hash}(k, 0), \text{hash}(k, 1), \dots, \text{hash}(k, m - 1) \rangle$  каждый номер ячейки  $0, 1, \dots, m - 1$  встретился только один раз.
- ◇ Если при поиске мы добираемся до ячейки, содержащей `NULL`, можно быть уверенным, что элемент с данным ключом отсутствует (иначе он попал бы в эту ячейку).

# Хеширование с открытой адресацией: программы

```
#define m 1999
struct htype {
    int key;           /* ключ */
    int val;          /* значение элемента данных */
} *index[m];

/* Поиск элемента */
struct htype *search (int k) {
    int i = 0, j;

    do {
        j = hash (k, i);
        if (index[j] && index[j]->key == k)
            return index[j];
    } while (index[j] && ++i < m);
    return NULL;
}
```

# Хеширование с открытой адресацией: программы

```
/* Добавление элемента */
int insert (int k, int v) {
    int i = 0, j;

    do {
        j = hash (k, i);
        if (index[j] && index[j]->key == k) {
            index[j]->val = v;
            return j;
        }
    } while (index[j] && ++i < m);
    /* Таблица может оказаться заполненной */
    if (i == m)
        return -1; /* Или расширим index */
    index[j] = new ();
    index[j]->key = k, index[j]->val = v;
    return j;
}
```

# Хеширование с открытой адресацией: программы

```
/* Внутренний поиск: вернем индекс массива */
```

```
static int search_internal (int k) {  
    int i = 0, j;  
  
    do {  
        j = hash (k, i);  
        if (index[j] && index[j]->key == k)  
            return j;  
    } while (index[j] && ++i < m);  
    return -1;  
}
```

```
/* Внешний поиск легко реализуется через внутренний */
```

```
struct htype *search (int k) {  
    int j = search_internal (k);  
    return j >= 0 ? index[j] : NULL;  
}
```

# Хеширование с открытой адресацией: программы

```
/* Удаление элемента */
```

```
void delete (int k) {
```

```
    int j;
```

```
    j = search_internal (k);
```

```
    if (j < 0)
```

```
        return;
```

```
    /* Нельзя писать index[j] = NULL!
```

```
    Будут потеряны ключи, возможно, находящиеся  
    за удаляемым ключом (с тем же хешем). */
```

```
    ???
```

```
}
```

# Хеширование с открытой адресацией: программы

```
#define SHADOW ((void *) (intptr_t) 1)

/* Удаление элемента */
void delete (int k) {
    int j;

    j = search_internal (k);
    if (j < 0)
        return;

    /* Нельзя писать index[j] = NULL! */
    free (index[j]);
    index[j] = SHADOW;
}
```

# Хеширование с открытой адресацией: программы

```
#define SHADOW ((void *) (intptr_t) 1)
#define ISEMPTY(el) ((!el) || (el) == SHADOW)

static int search_internal (int k) {
    int i = 0, j;

    do {
        j = hash (k, i);
        if (!ISEMPTY (index[j]) && index[j]->key == k)
            return j;
    } while (index[j] && ++i < m);
    return -1;
}
```

# Хеширование с открытой адресацией: программы

```
#define SHADOW ((void *) (intptr_t) 1)
#define ISEMPTY(e1) ((!e1) || (e1) == SHADOW)

/* Добавление элемента */
int insert (int k, int v) {
    int i = 0, j;

    do {
        j = hash (k, i);
        if (! ISEMPTY (index[j]) && index[j]->key == k) {
            index[j]->val = v;
            return j;
        }
    } while (index[j] && ++i < m);

    /* Таблица может оказаться заполненной (много вставок/удалений) */
    if (i == m)
        return -1; /* Или расширим index */
    index[j] = new (); /* можно запомнить индекс последнего shadow
                        элемента и вставить вместо него */
    index[j]->key = k, index[j]->val = v;
    return j;
}
```