

Материалы презентации предназначены для размещения только для использования студентами кафедры «Компьютерные системы и технологии» НИЯУ МИФИ дневного и вечернего отделений, изучающими курс «Программирование (Алгоритмы и структуры данных)».

Публикация (размещение) данных материалов полностью или частично в электронном или печатном виде в любых других открытых или закрытых изданиях (ресурсах), а также использование их для целей, не связанных с учебным процессом в рамках курса «Программирование (Алгоритмы и структуры данных)» кафедры «КСиТ» НИЯУ МИФИ, без письменного разрешения автора запрещена.

6. Куча

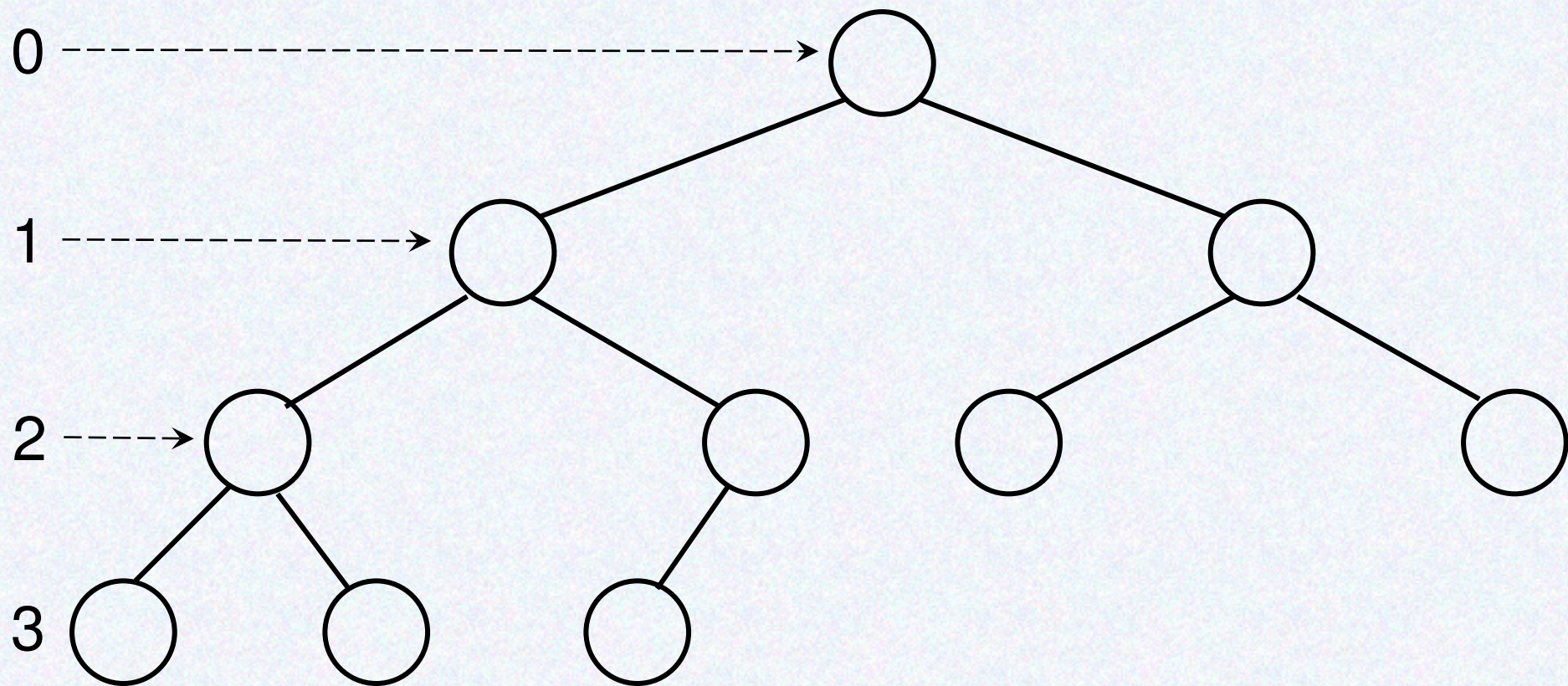
6.1

Пирамида

Пирамида – *binary heap* – объект-массив;
почти полное бинарное дерево:

- узел дерева – элемент массива
- на всех уровнях (кроме последнего) дерево заполнено полностью
- последний уровень заполняется слева направо

Пирамида



6.3

Пирамида

Атрибуты массива – пирамиды A :

- $length(A)$ – количество элементов массива
- $heap_size(A)$ – количество элементов пирамиды, содержащихся в массиве

$$heap_size(A) \leq length(A)$$

6.4

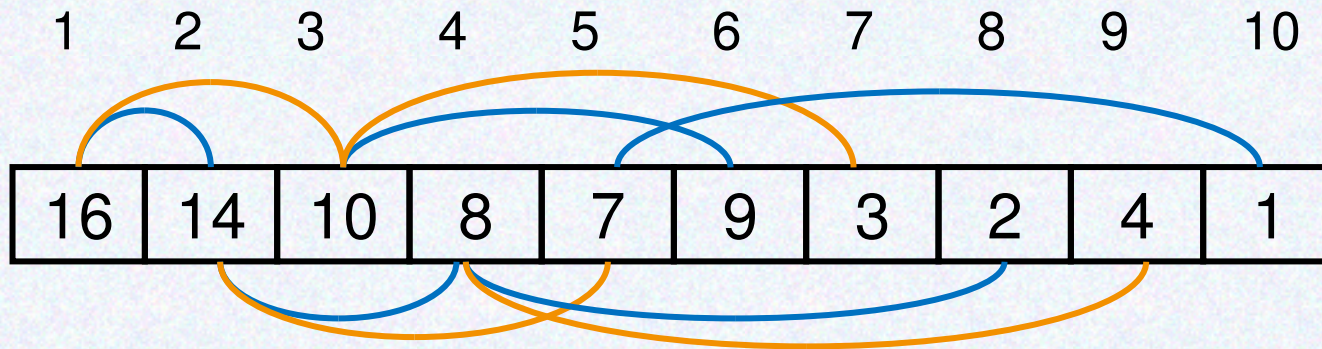
Пирамида

- Элементы массива A_1, A_2, \dots
- A_1 – корень дерева
- Если A_i – промежуточный узел дерева, то для него:
 - индекс родительского узла $parent(i) = i / 2$
 - индекс левого дочернего узла $left(i) = 2i$
 - индекс правого дочернего узла $right(i) = 2i + 1$

Пример пирамиды

$$\text{left}(i) = 2i$$

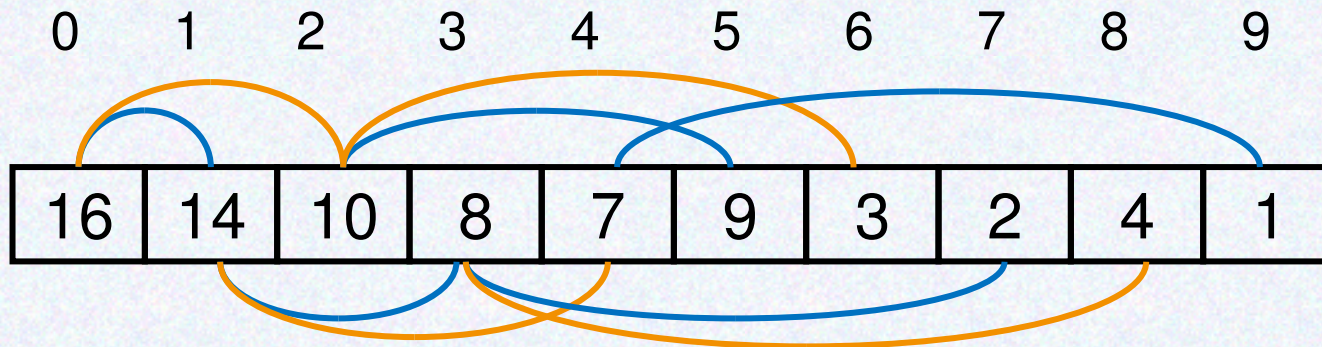
$$\text{right}(i) = 2i + 1$$



Пример пирамиды

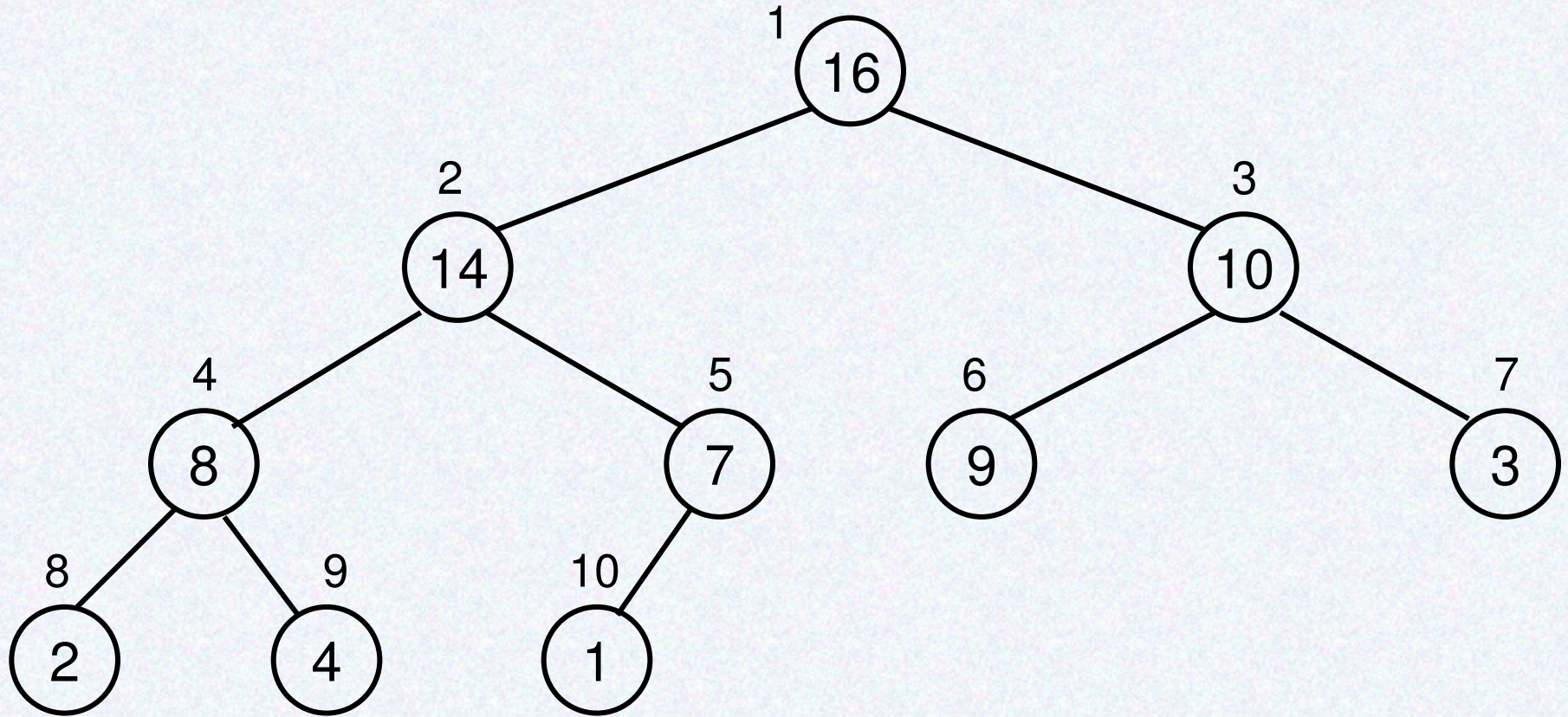
$$\text{left}(i) = 2i + 1$$

$$\text{right}(i) = 2i + 2$$



Пример пирамиды

Представление в виде дерева



6.8

Свойства пирамид

Два вида пирамид:

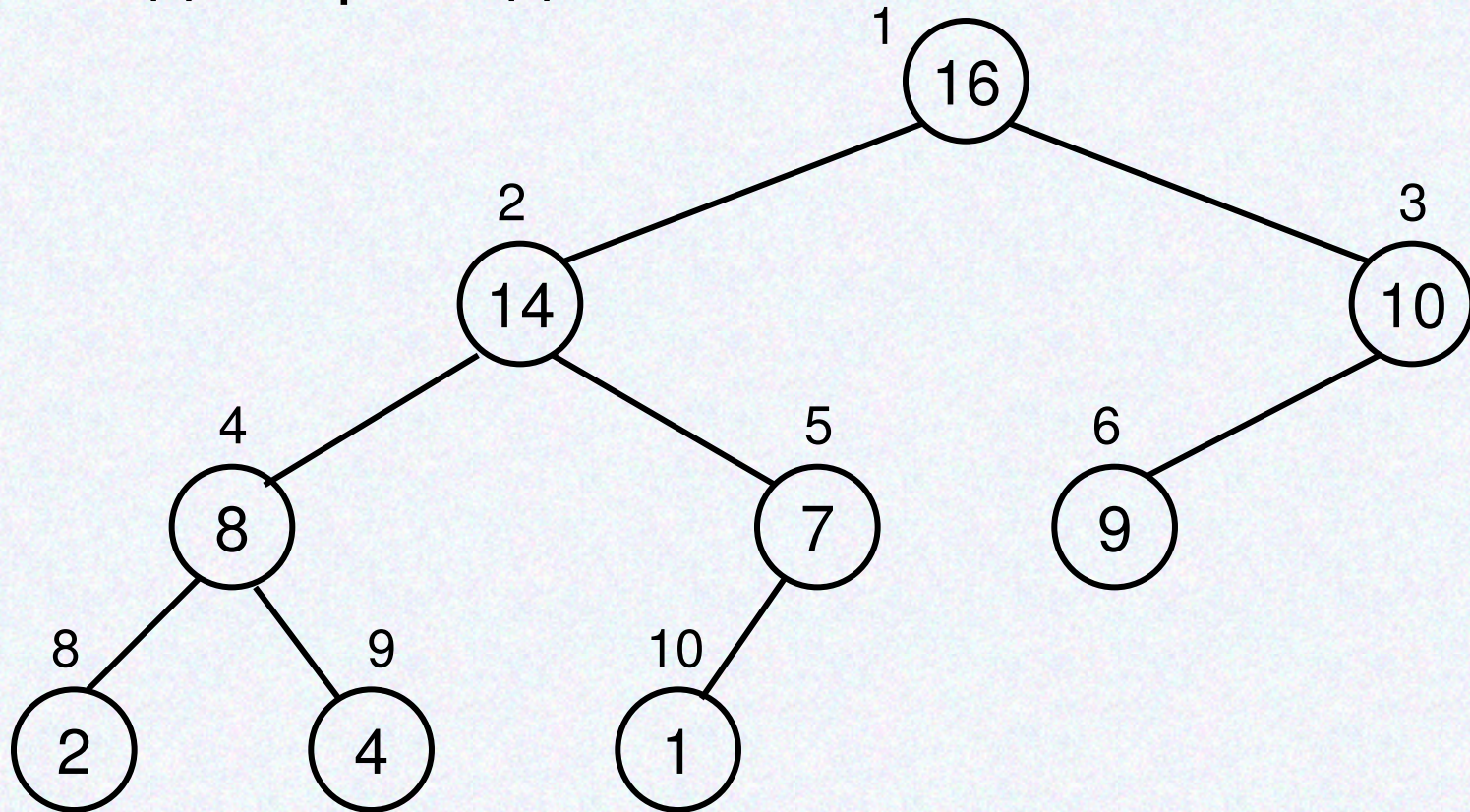
- *невозрастающие* – max-heap property:

$$A_{parent(i)} \geq A_i$$

Пирамидальные сортировки

Свойства пирамид

Два вида пирамид:



6.10

Свойства пирамид

Два вида пирамид:

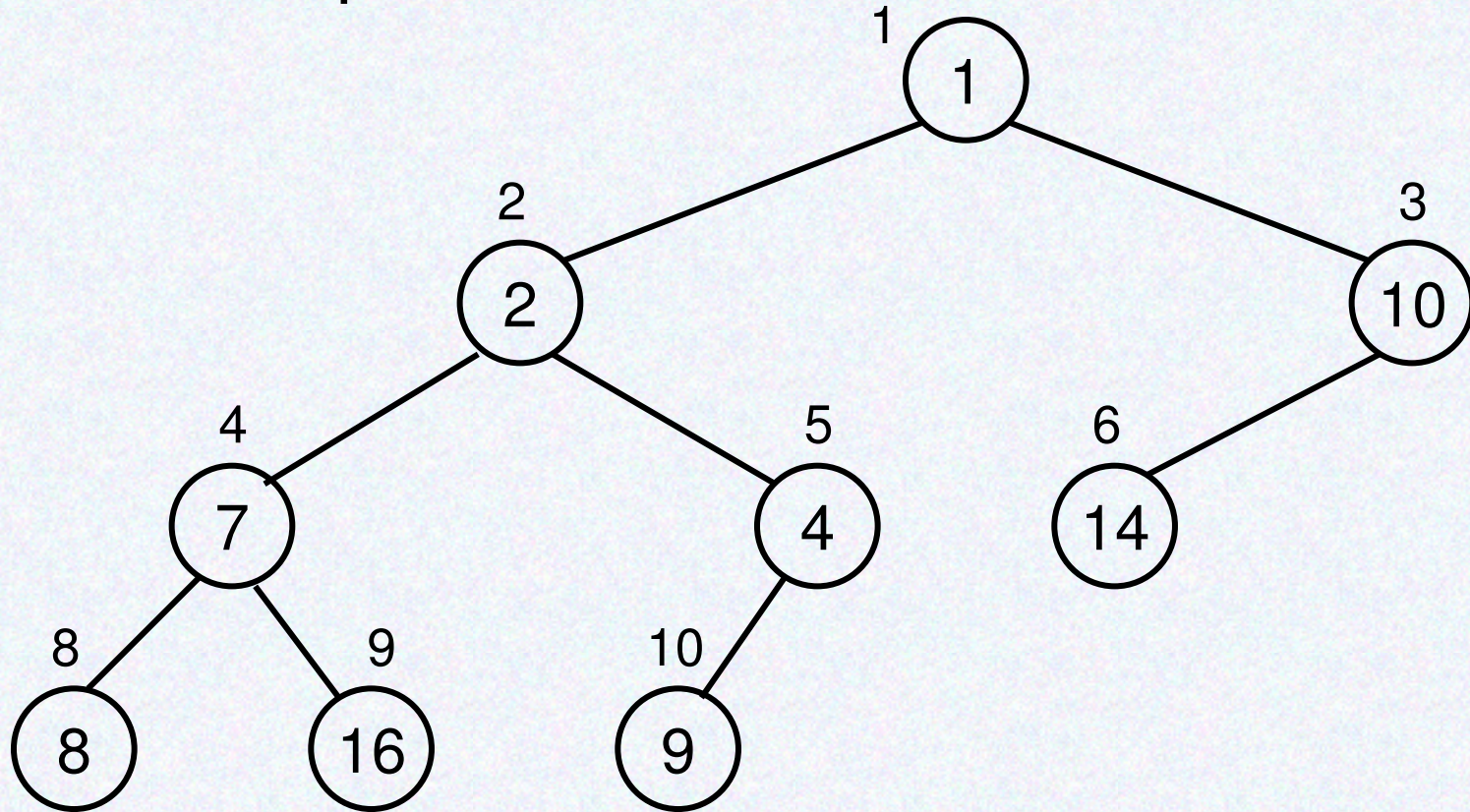
- *неубывающие* – min-heap property:

$$A_{parent(i)} \leq A_i$$

Очереди с приоритетами

Свойства пирамид

Два вида пирамид:



Max_Heapify(A, i)

Поддержка свойства пирамиды

Предположения:

- бинарные деревья с корнями *left(i)* и *right(i)* – невозрастающие пирамиды
- элемент A_i может нарушить свойство пирамиды

Время работы с узлом, расположенным на высоте h – $O(h)$

Max_Heapify(A, i)

l = *left(i)*

r = *right(i)*

largest = индекс максимального из трех элементов: A_i , A_l , A_r , при условии, что $l \leq \text{heap_size}(A)$ и $r \leq \text{heap_size}(A)$

if *largest* $\neq i$ {

 поменять местами A_i и $A_{largest}$

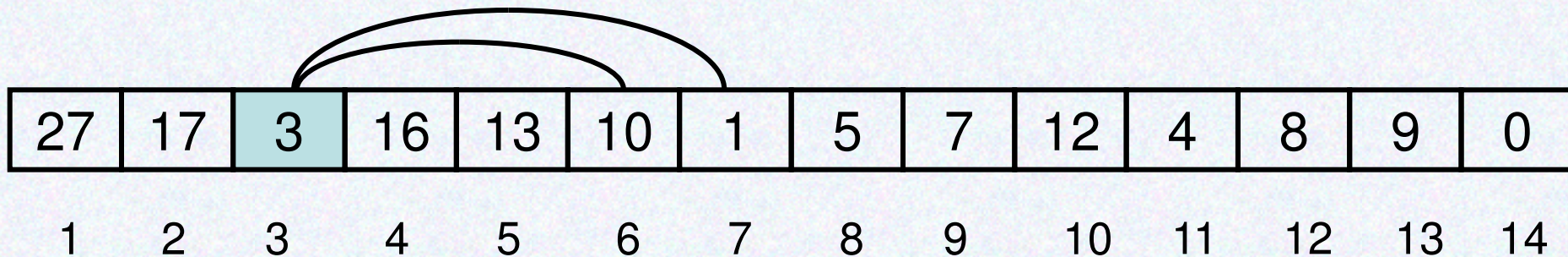
Max_Heapify(*A*, *largest*)

}

Пример

$A = \langle 27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0 \rangle$

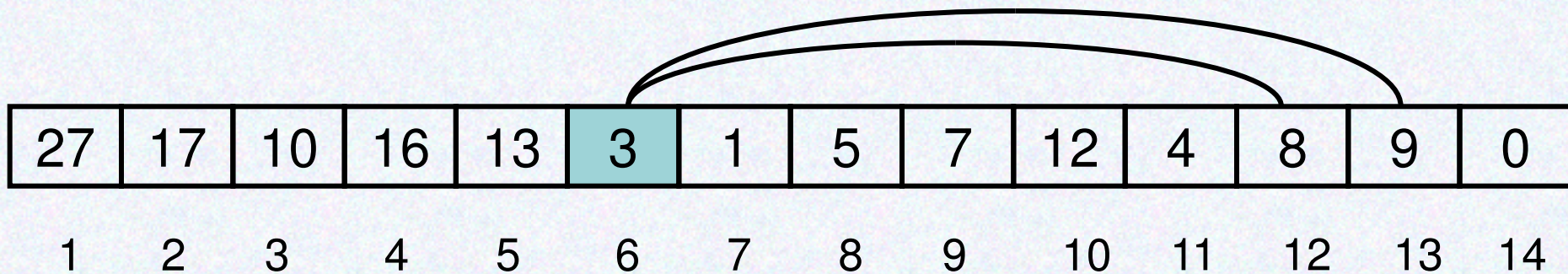
$Max_Heapify(A, 3)$



Пример

$A = \langle 27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0 \rangle$

$Max_Heapify(A, 3)$



Пример

$A = \langle 27, 17, 3, 16, 13, 10, 1, 5, 7, 12, 4, 8, 9, 0 \rangle$

$Max_Heapify(A, 3)$

27	17	10	16	13	9	1	5	7	12	4	8	3	0
1	2	3	4	5	6	7	8	9	10	11	12	13	14

6.17

Создание пирамиды

Дан массив $A = \langle A_1, A_2, \dots, A_n \rangle$,
 $n = \text{length}(A)$

Элементы массива $A_{j+1}, A_{j+2}, \dots, A_n$, где
 $j = n / 2$ – листья

Построение пирамиды – от листьев к корню

Build_Max_Heap(A)

heap_size(A) = length(A)

i = n / 2

while i > 0 {

Max_Heapify(A, i)


i = i - 1

}

Время работы алгоритма – $O(n \log n)$; более точная оценка – $O(n)$

Иллюстрация

$$i = 5$$

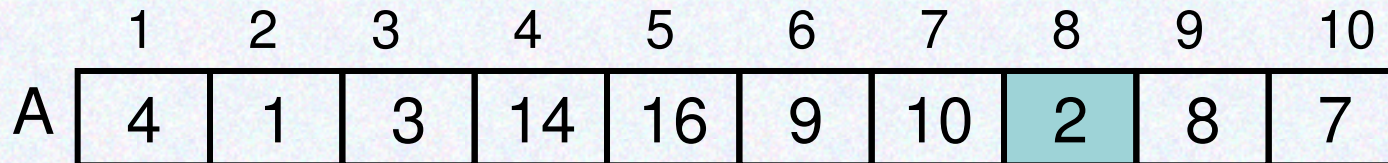
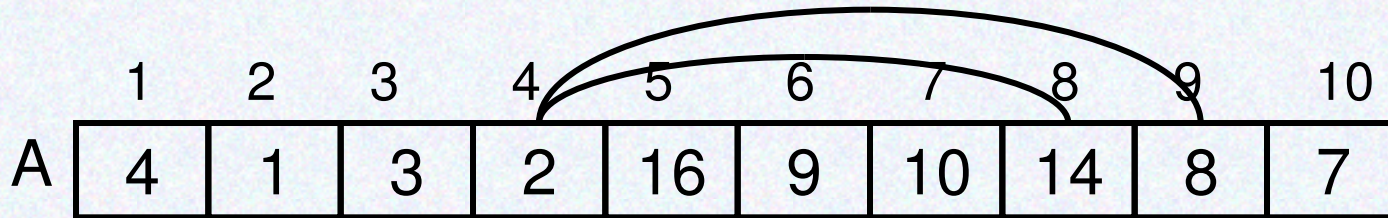


	1	2	3	4	5	6	7	8	9	10
A	4	1	3	2	16	9	10	14	8	7

	1	2	3	4	5	6	7	8	9	10
A	4	1	3	2	16	9	10	14	8	7

Иллюстрация

$$i = 4$$



Иллюстрация

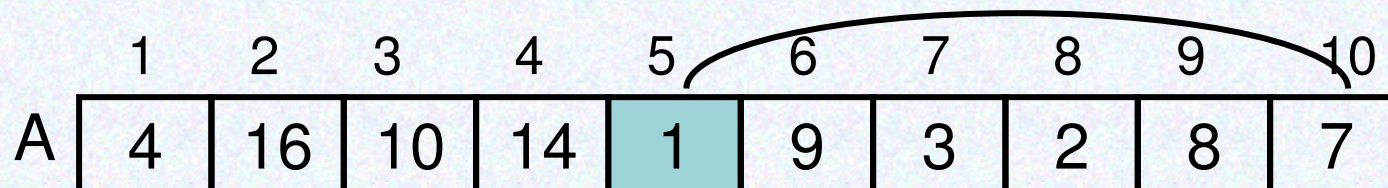
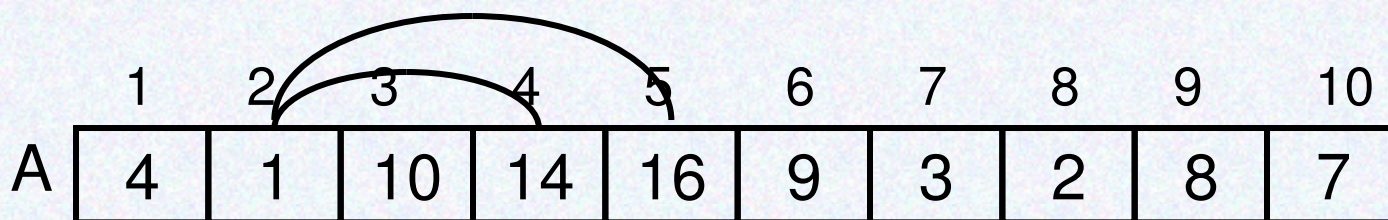
$$i = 3$$

	1	2	3	4	5	6	7	8	9	10
A	4	1	3	14	16	9	10	2	8	7

	1	2	3	4	5	6	7	8	9	10
A	4	1	10	14	16	9	3	2	8	7

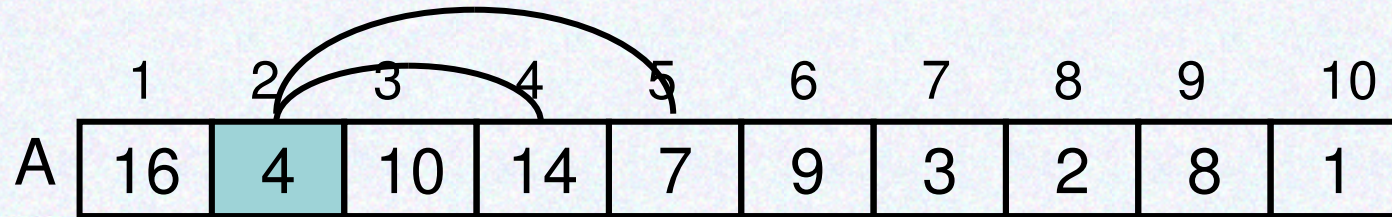
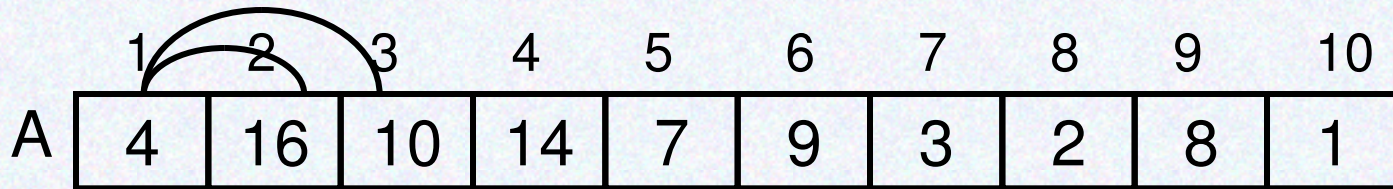
Иллюстрация

$$i = 2$$

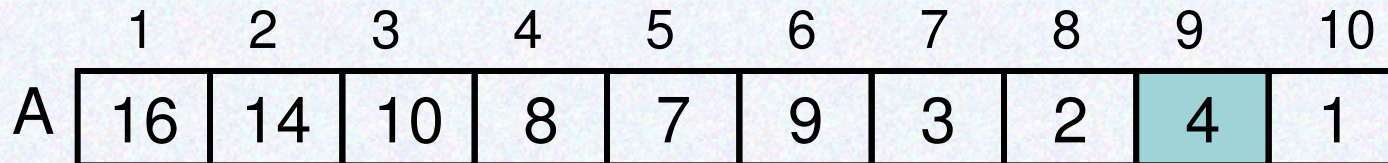
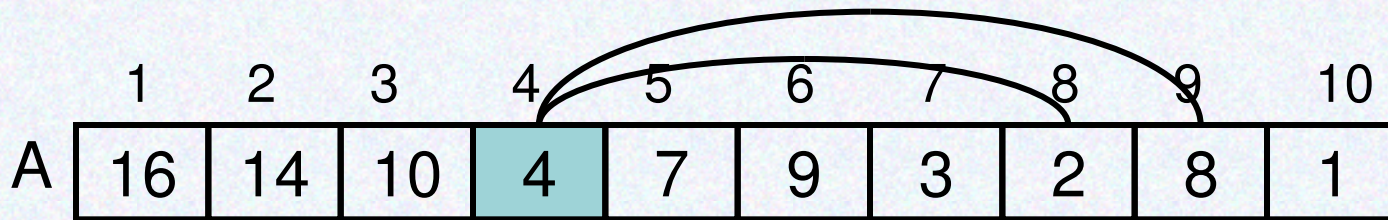


Иллюстрация

$$i = 1$$



Иллюстрация

 $i = 1$ 

6.25

Пирамидальная сортировка

Шаг 1: построить невозрастающую пирамиду

Шаг 2: сортировка

- поменять местами первый и последний элементы пирамиды
- уменьшить размер пирамиды
- скорректировать свойство пирамиды для первого узла

Heap_Sort(A)

Build_Max_Heap(A)

$i = \text{length}(A)$

while $i \geq 2$ {

 поменять местами A_1 и A_i

$\text{heap_size}(A) = \text{heap_size}(A) - 1$

$\text{Max_Heapify}(A, 1)$

$i = i - 1$

}

Иллюстрация сортировки

Построение пирамиды

1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

1	2	3	4	5	6	7	8	9	10
1	14	10	8	7	9	3	2	4	16

Иллюстрация сортировки

Коррекция свойств пирамиды

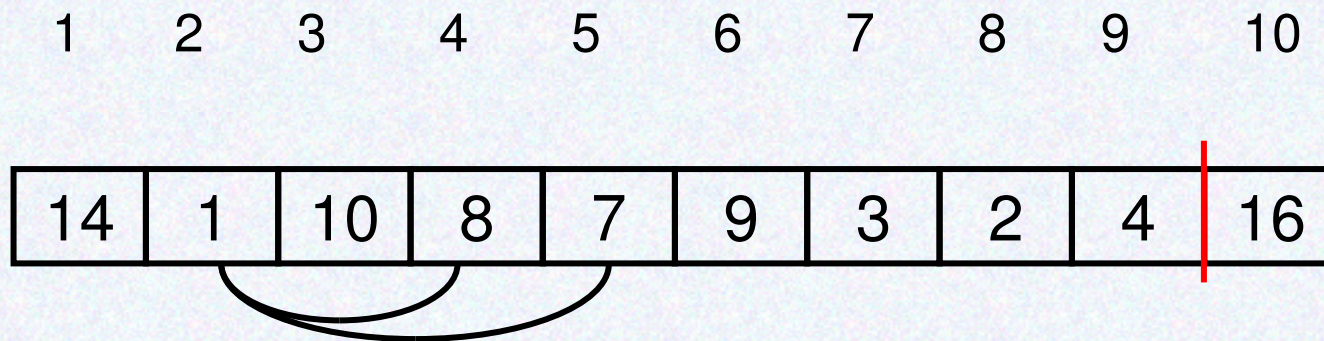
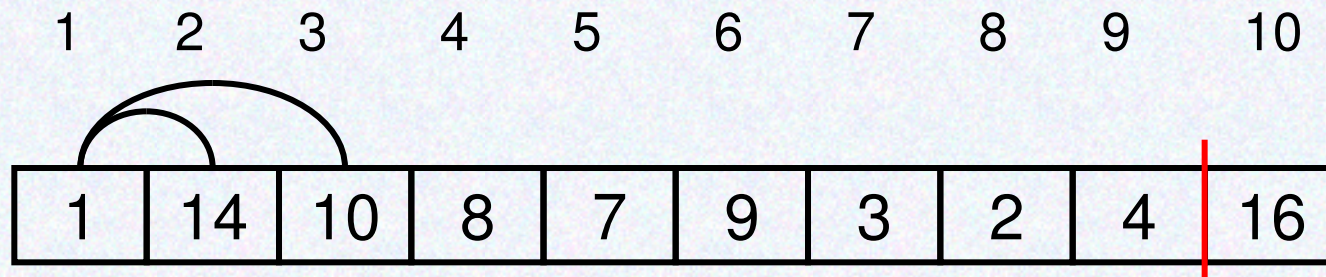


Иллюстрация сортировки

Коррекция свойств пирамиды

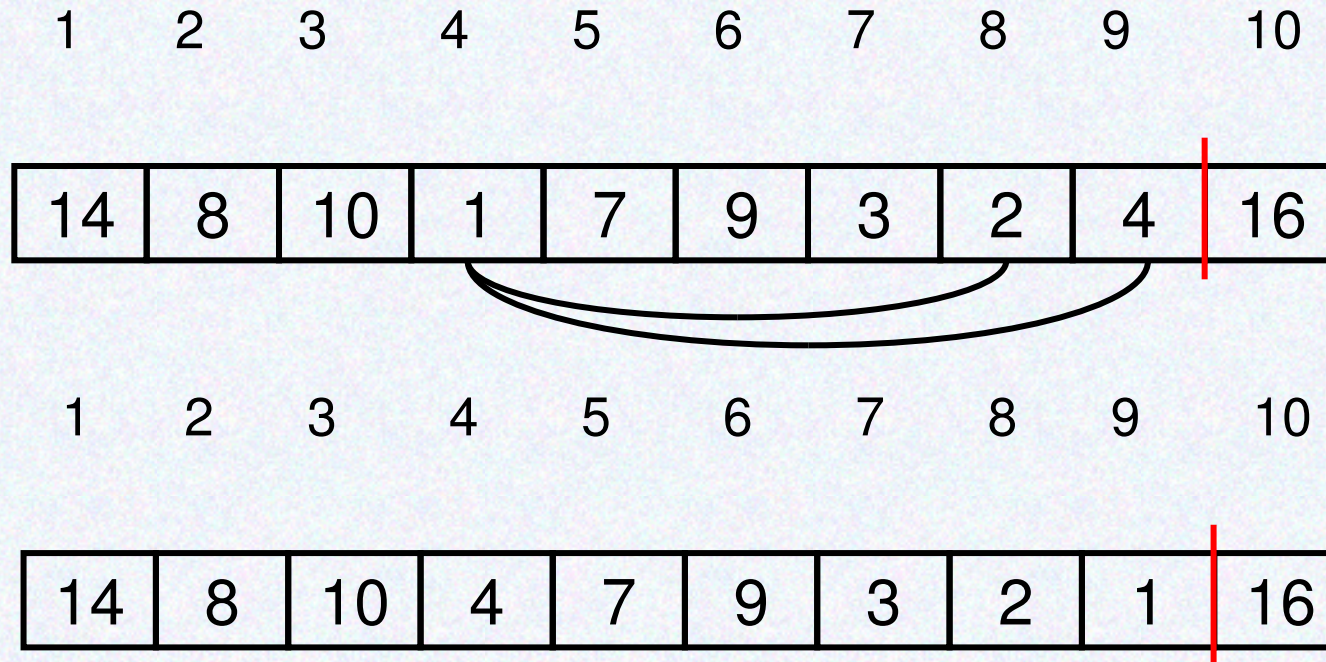


Иллюстрация сортировки

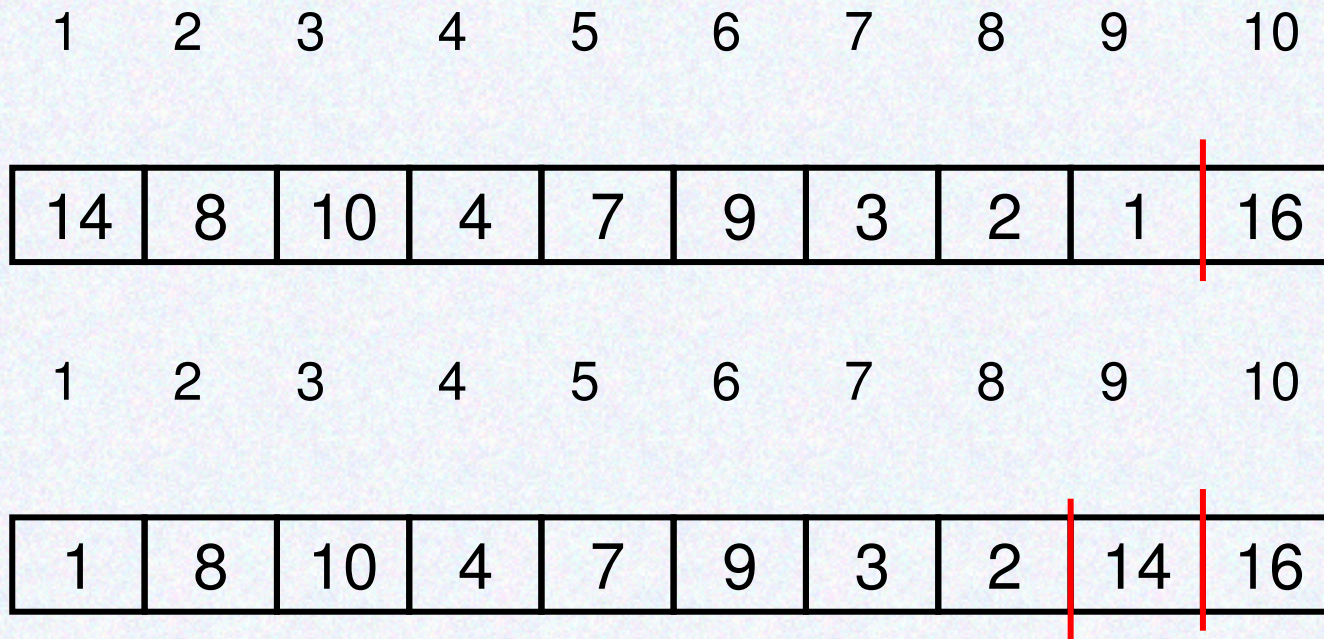


Иллюстрация сортировки

Коррекция свойств пирамиды

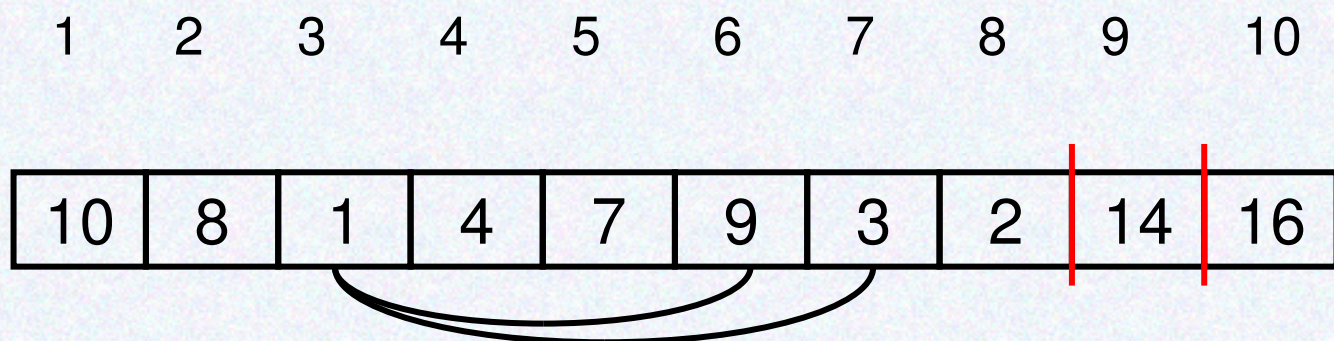
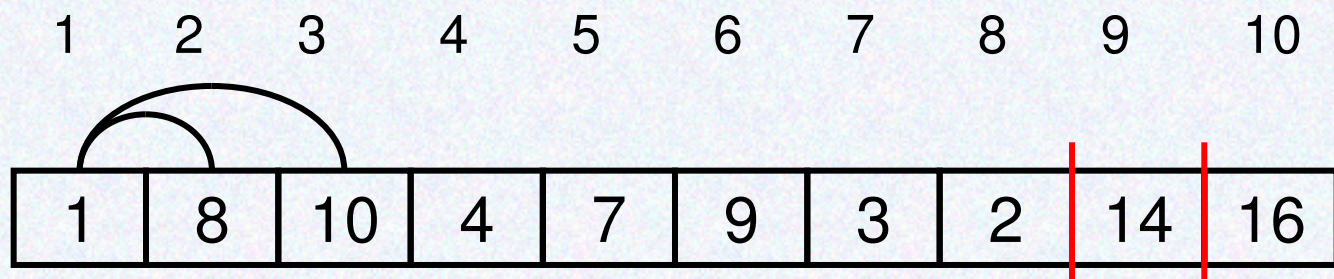


Иллюстрация сортировки

Коррекция свойств пирамиды

1	2	3	4	5	6	7	8	9	10
10	8	9	4	7	1	3	2	14	16

Очереди с приоритетами

Два вида: *невозрастающие* и *неубывающие*

Очередь с приоритетами (*priority queue*) – структура данных, предназначенная для обслуживания множества S , с каждым элементом которого связано определенное значение, называемое ключом (key), или приоритетом

Операции

Для невозрастающей очереди с приоритетами:

- *Insert*(S, x) – вставить элемент x в множество S
- *Maximum*(S) – вернуть элемент множества S с наибольшим ключом
- *Extract_Max*(S) – вернуть элемент множества S с наибольшим ключом, удалив его из множества

Операции

- *Increase_Key*(S, x, k) – увеличить значение ключа, соответствующего элементу x , путем его замены ключом со значением k (предполагается, что величина k не меньше текущего ключа элемента x)

6.36

*Maximum(A)**Heap_Maximum(A)*Возвращает A_1 Время работы – $\Theta(1)$

Extract_Maximum(A)

if очередь пуста
отказ

Извлечь из пирамиды A_1 : $res = A_1$

Поместить в первую позицию пирамиды
ее последний элемент

Уменьшить размер пирамиды на 1

Восстановить свойства пирамиды для первого
узла

Время работы $O(\log n)$

6.38

Increase_Key(A, i, key)

if $key < A_i$

отказ: новый ключ меньше текущего

Обновить ключ: $A_i = key$

Вставить обновленный элемент в
соответствующую позицию:

while $i > 1$ и $A_{parent(i)} < A_i$ {

 обменять значения A_i и $A_{parent(i)}$

$i = parent(i)$

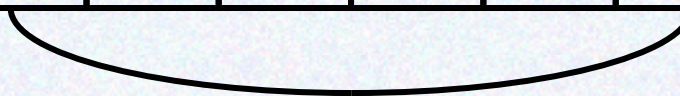
}

Пример

Изменить ключ элемента с индексом 9 на 15

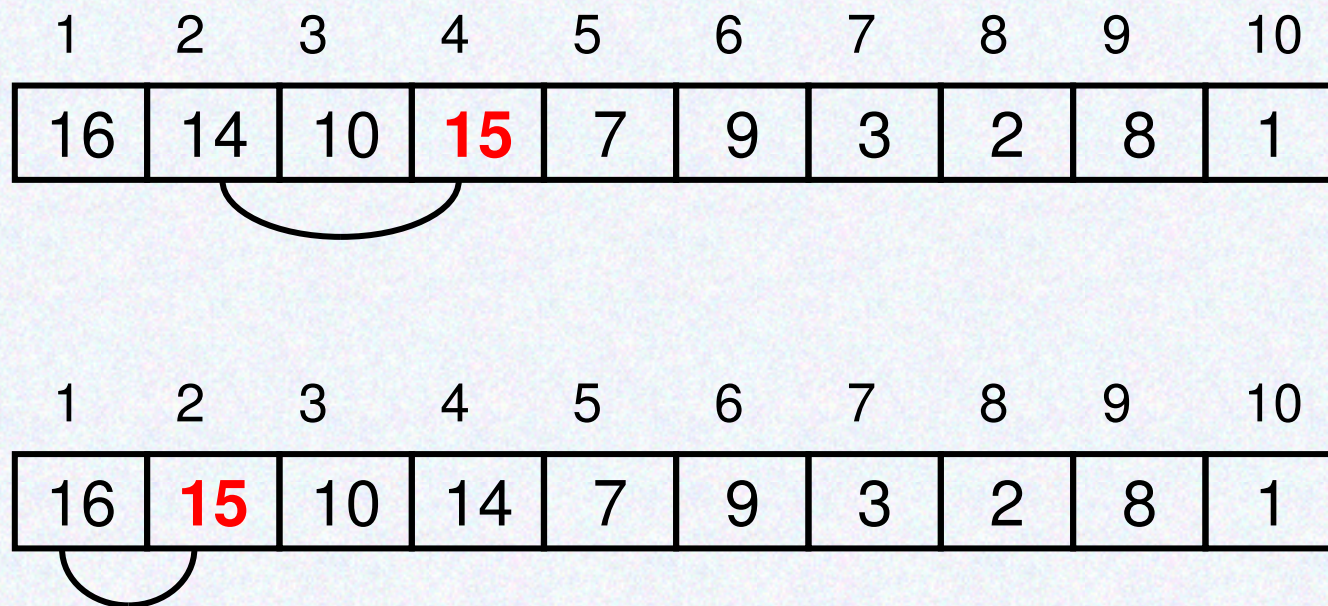
1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	15	1



Пример

Изменить ключ элемента с индексом 9 на 15



Время работы – $O(\log n)$

Insert(A, key)

Вставить новый лист с ключом $-\infty$:

$$\text{heap_size}(A) = \text{heap_size}(A) + 1$$

$$A_{\text{heap_size}(A)} = -\infty$$

Увеличить значение ключа:

$$\text{Increase_Key}(A, \text{heap_size}(A), \text{key})$$

Время вставки – $O(\log n)$

7. Списки с пропусками

7.1

Skip Lists

Skip Lists – списки с пропусками: вероятностная структура данных

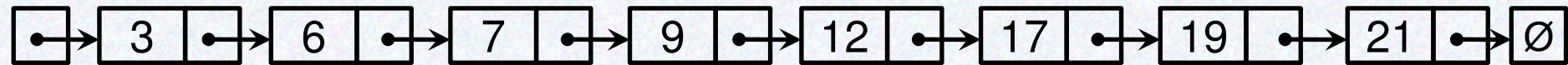
Альтернатива сбалансированным деревьям

Предложены William Pugh (1990 г.)

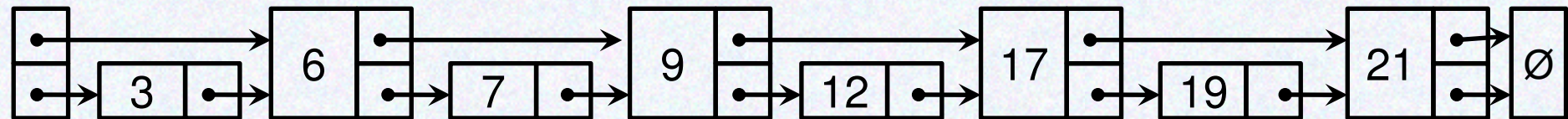
В основе – обычный упорядоченный односвязный список

Общий подход

Обычный упорядоченный список

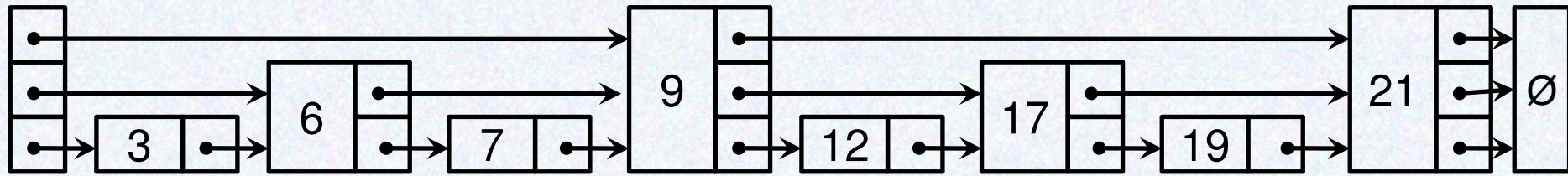


Дополнительные указатели:

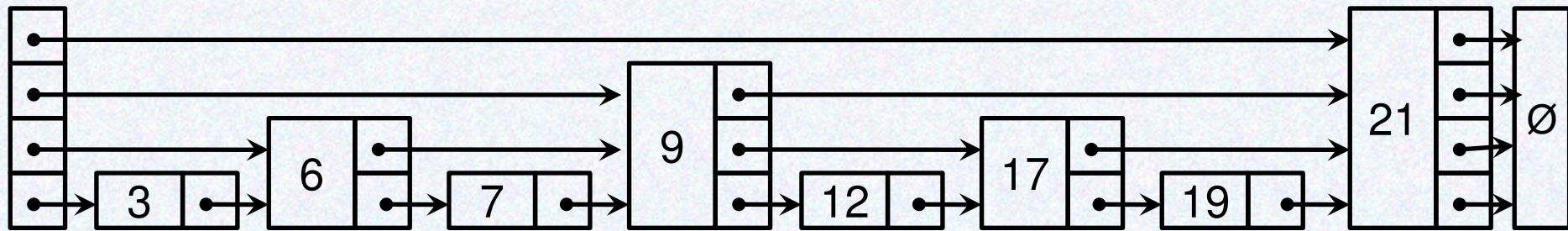


Поиск — $\lceil n/2 \rceil + 1$ элемент

Общий подход



Поиск — $\lceil n/4 \rceil + 2$ элемента



Поиск — $\lceil \log_2 n \rceil$

Определения

Элемент списка, имеющий k указателей –
элемент уровня k

Если каждый 2^i -ый элемент имеет 2^i указателей:

50% элементов – элементы уровня 1 (n_1)

25% элементов – элементы уровня 2
($n_2 = n_1 / 2$)

12,5% элементов – элементы уровня 3
($n_3 = n_2 / 2$)

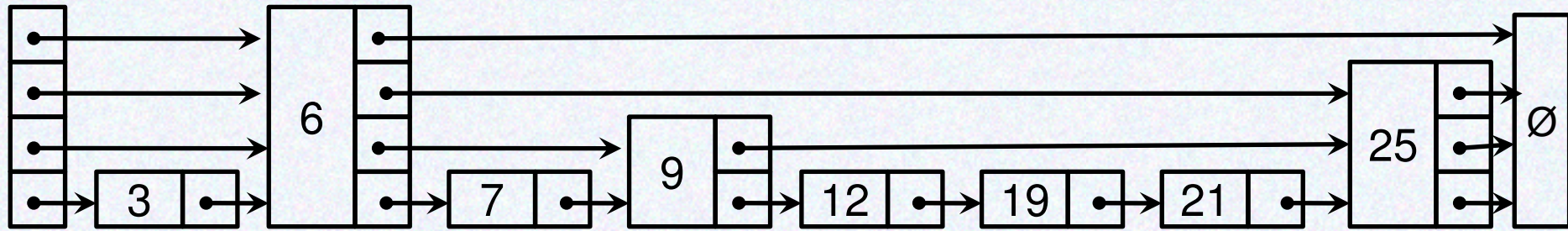
Skip Lists

Уровни элементов списка выбираются случайным образом

Элемент списка уровня j указывает на следующий элемент этого же уровня (не обязательно это 2^j -ый элемент списка)

Уровень элемента списка выбирается случайно при вставке элемента и не изменяется в дальнейшем

Skip Lists



Операции вставки и удаления потребуют только локальные модификации (переопределение указателей)

Предположения

В общем случае p – доля элементов i -го уровня, определяющая элементы $(i + 1)$ -го уровня

Обычно $p = 1/2$ или $p = 1/4$

Уровень списка – максимальный уровень вершины в списке

Уровень элемента списка ограничен некоторым значением *MaxLevel*

Предположения

Заголовок Skip Lists – содержит уровень списка и массив из *MaxLevel* указателей

Указатели заголовка списка для уровней, превышающих текущий уровень списка, указывают на специальный элемент *EList*

Уровень элемента генерируется случайным образом независимо от количества элементов в Skip List

7.9

Предположения

Определение уровня элемента

В идеале начать поиск на уровне L , где ожидается $1/p$ элементов:

$$L(n) = \log_{1/p} n$$

Предположения

Определение *MaxLevel*

$MaxLevel = L(N)$, N – максимально допустимое количество элементов в Skip Lists

Для $p = 1/2$ выбор $MaxLevel = 16$ позволяет разместить в Skip Lists до 2^{16} элементов

7.11 Структура элемента списка

key – ключ элемента

сопутствующая информация

level – количество указателей

forwards[] – массив из *MaxLevel* указателей

Элемент *EList* – специальный элемент списка
с максимальным значением ключа

7.12

Структура элемента списка

```
#include <limits.h>
```

```
struct Item {
```

```
    int key;
```

```
    Type info;
```

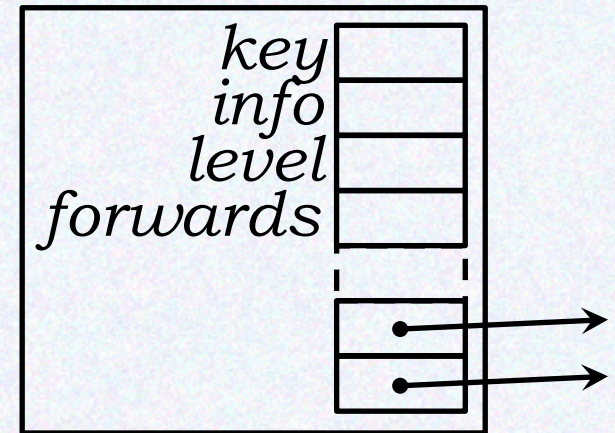
```
    int level;
```

```
    struct Item *forwards[MaxLevel];
```

```
};
```

```
struct Item EList = {INT_MAX};
```

Item



7.13

Задание Skip Lists

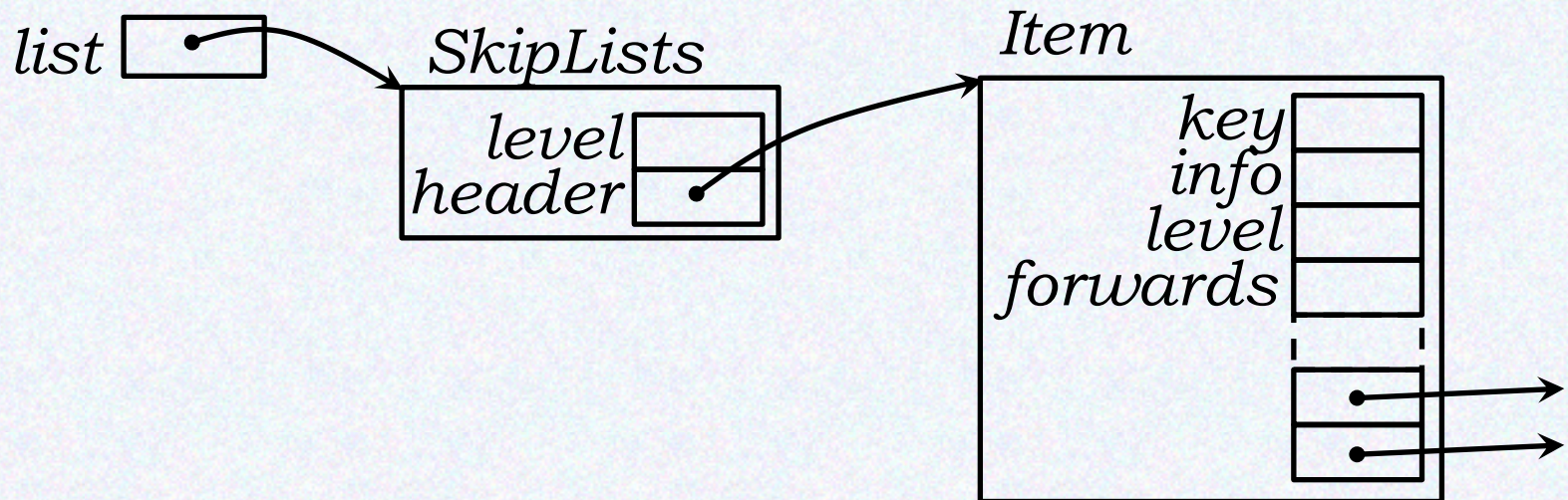
```
struct SkipLists {
```

```
    int level; – текущий уровень списка
```

```
    struct Item *header; – указатель на головной элемент
```

```
};
```

```
struct SkipLists *list; – указатель на Skip Lists
```



Генерация уровня

level = 1

while random() < p u level < MaxLevel

level = level + 1

random() – возвращает случайное число из $[0 - 1)$:

((double) rand() - 1) / RAND_MAX

7.15

Алгоритм поиска

list – указатель на Skip Lists

k – ИСКОМЫЙ КЛЮЧ

x = list->header – указатель на головной элемент

ЦИКЛ ПО *i* ОТ *list->level - 1* ДО 0 {

while x->forward[i]->key < k

x = x->forward[i]

}

7.16

Алгоритм поиска

Вышли на список элементов 1-го уровня

$x = x \rightarrow forward[0]$

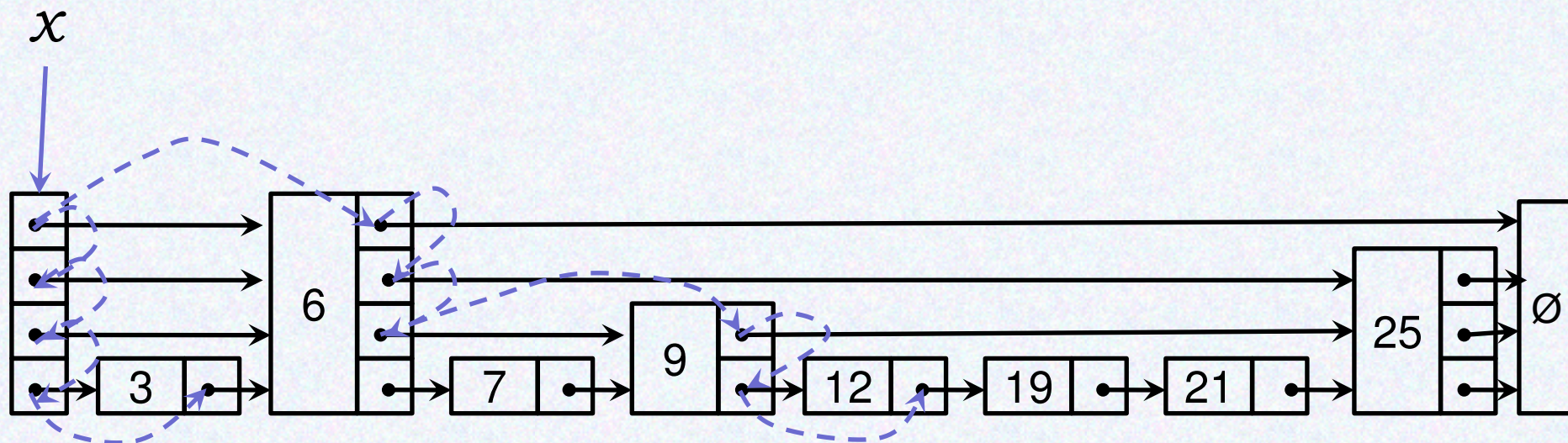
if $x \rightarrow key == k$

успех: x

else

отказ

Пример поиска



Найти 19

Найти 5

7.18

Алгоритм вставки

list – указатель на Skip Lists

k – новое значение

update[] – локальный массив из *MaxLevel*
указателей

$x = list \rightarrow header$

Алгоритм вставки

1. Поиск позиции в *Skip Lists* для вставки нового элемента

цикл по i от $list \rightarrow level - 1$ до 0 {

$while\ x \rightarrow forward[i] \rightarrow key < k$

$x = x \rightarrow forward[i]$

$update[i] = x$ – позиция предшествующего элемента i -го уровня

}

$x = x \rightarrow forward[0]$ – элемент, перед которым нужно вставить новый

Алгоритм вставки

2. Формирование нового элемента

if $x \rightarrow key == k$

отказ

level = уровень нового элемента (случайный)

if $list \rightarrow level < level \{$

добавить в элементы $update[i]$ значения

$list \rightarrow header$

$list \rightarrow level = level$

$\}$

x = новый элемент списка уровня *level*

Алгоритм вставки

3. Модификация указателей

для всех i от 0 до $level - 1$ {

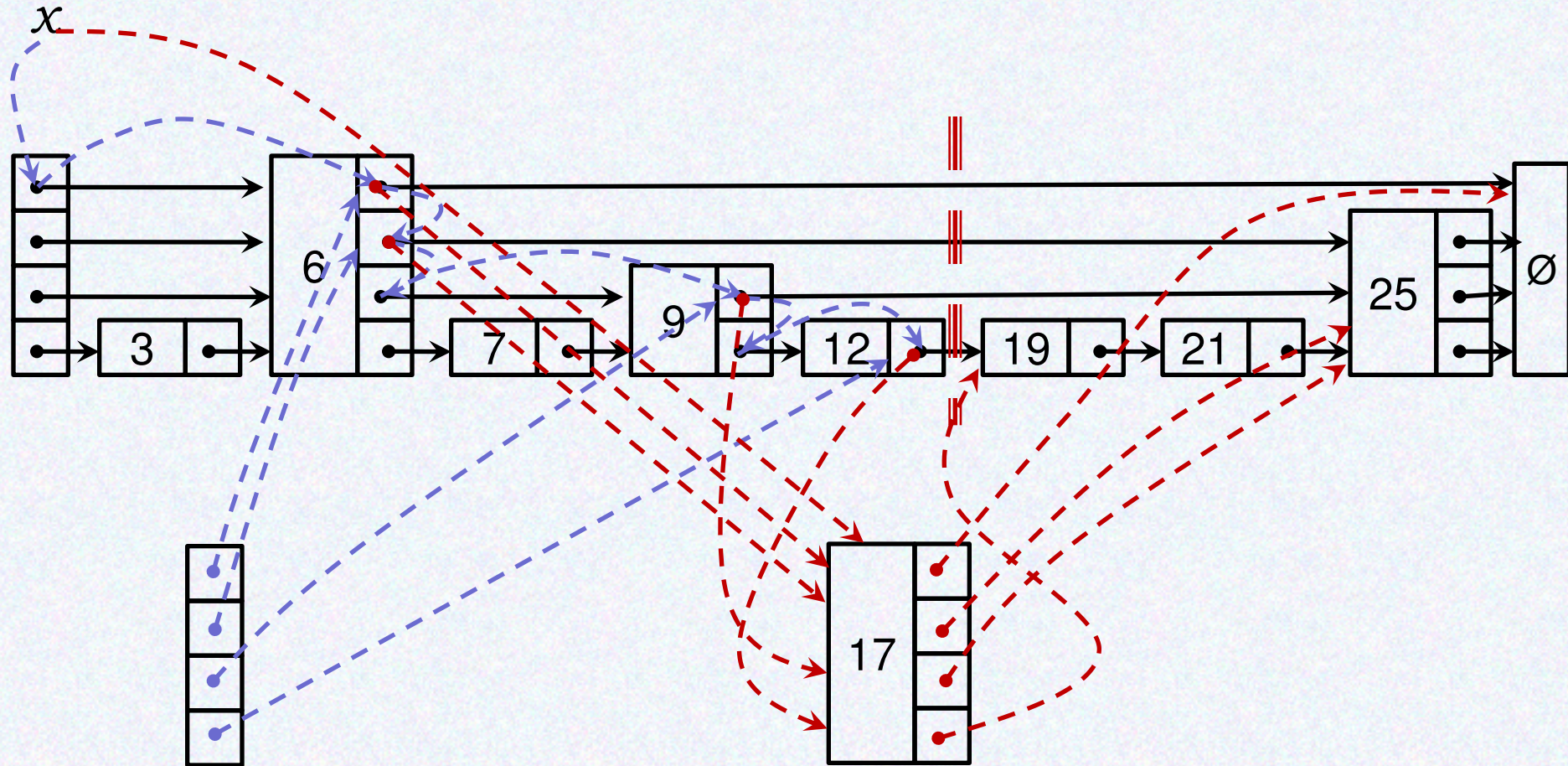
$x \rightarrow forward[i] = update[i] \rightarrow forward[i]$

$update[i] \rightarrow forward[i] = x$

}

Пример вставки

Вставить элемент 17



update

7.23

Алгоритм удаления

list – указатель на Skip Lists

k – удаляемое значение

update[] – локальный массив из *MaxLevel*
указателей

$x = list \rightarrow header$

Алгоритм удаления

Поиск в Skip Lists удаляемого элемента

цикл по i от $list \rightarrow level - 1$ до 0 {

while $x \rightarrow forward[i] \rightarrow key < k$

$x = x \rightarrow forward[i]$

*$update[i] = x$ – позиция предшествующего
элемента i -го уровня*

}

$x = x \rightarrow forward[0]$ – удаляемый элемент

Алгоритм удаления

if $x \rightarrow key = k$ {

 модифицируем указатели:

$i = 0$

while $i < level$ *u* $update[i] \rightarrow forward[i] = x$ {

$update[i] \rightarrow forward[i] = x \rightarrow forward[i]$

$i = i + 1$

 }

удалить x

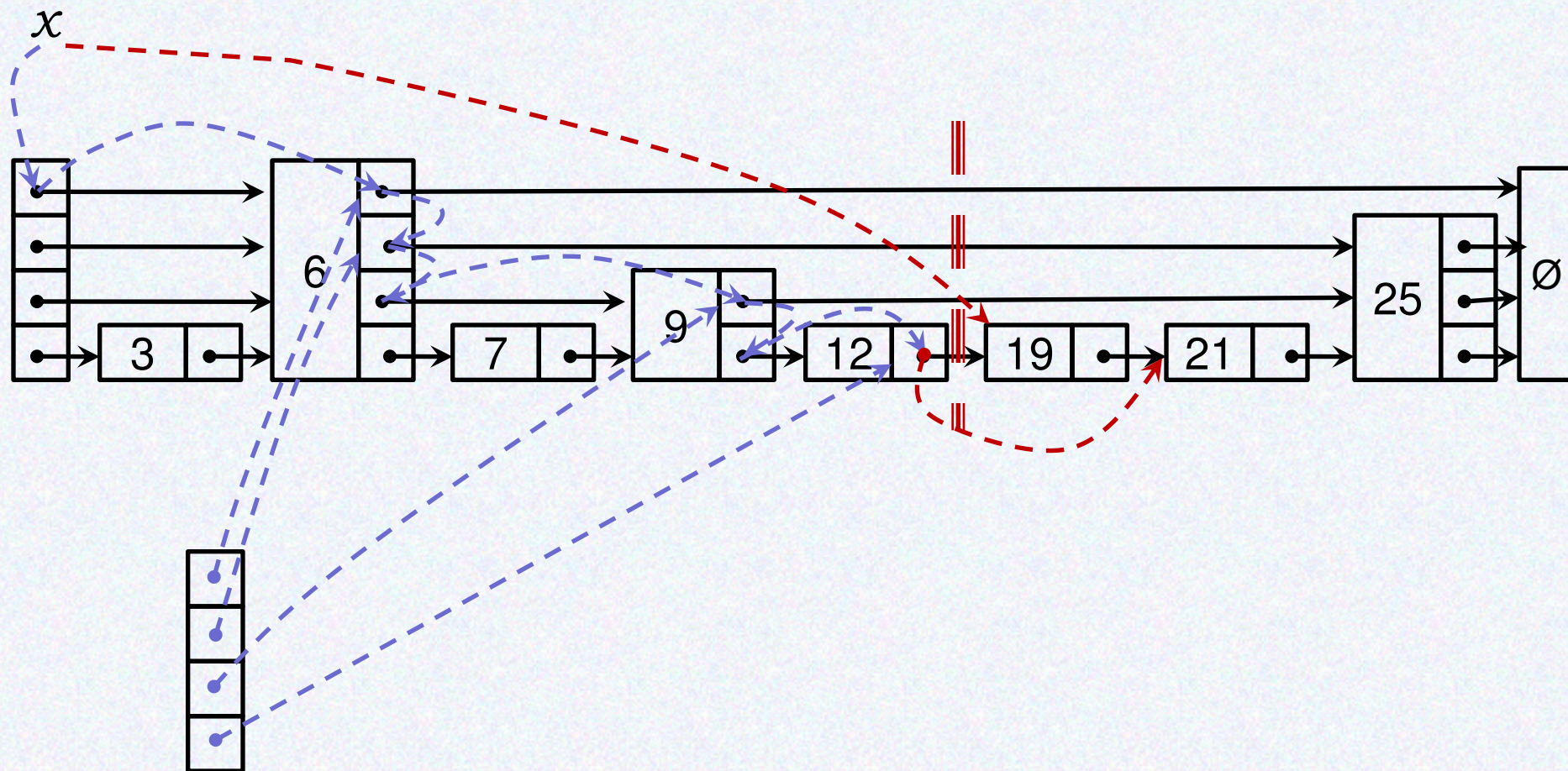
7.26

Алгоритм удаления

уменьшить, при необходимости, уровень списка:

```
while list->level > 1 и  
list->header->forwards[list->level] = EList  
list->level = list->level - 1
```

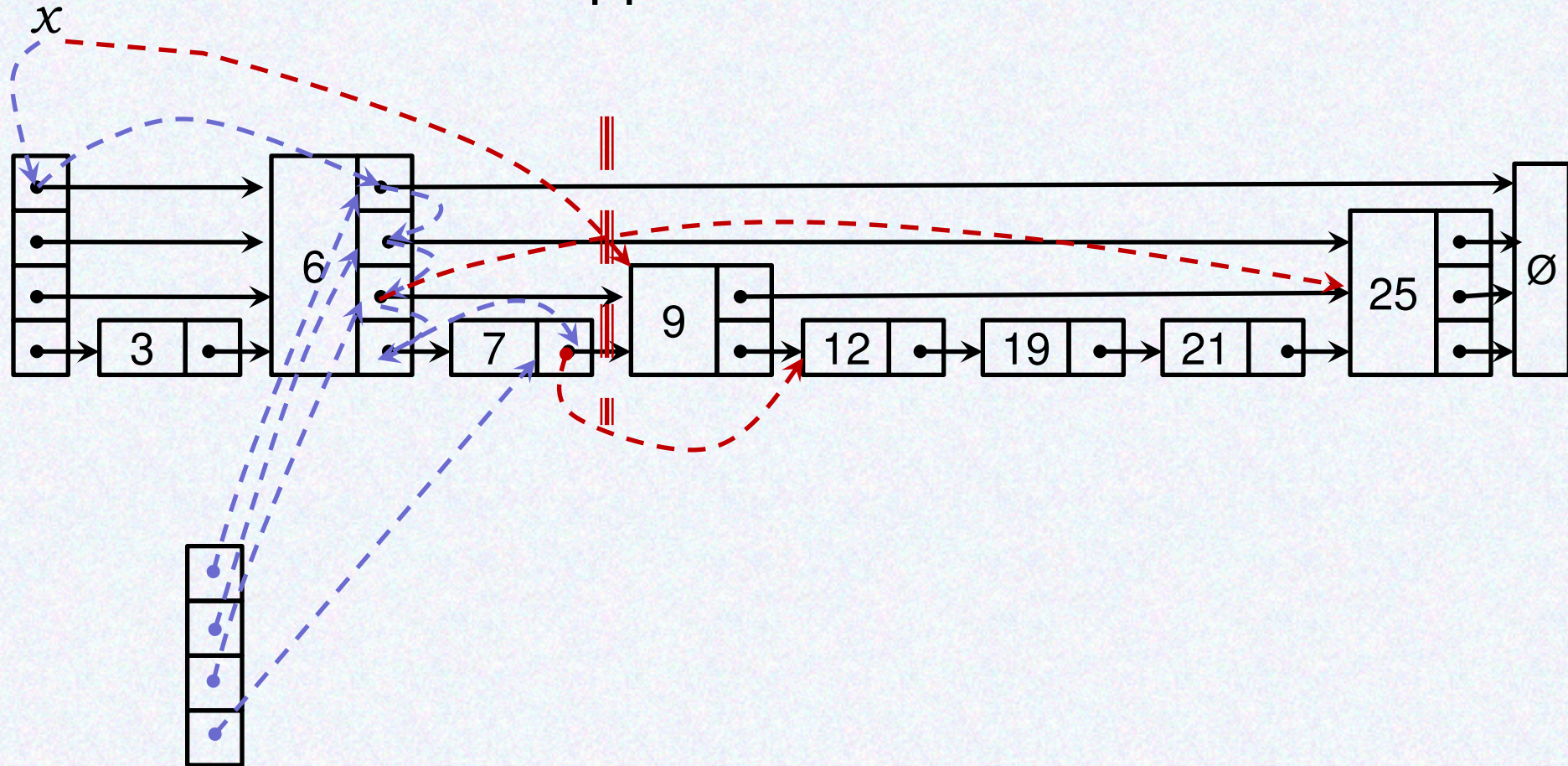
Удалить элемент 19



update

Пример удаления

Удалить элемент 9



update