

Лекция 10

Биномиальные кучи (Binomial heaps)

Курносов Михаил Георгиевич

E-mail: mkurnosov@gmail.com

WWW: www.mkurnosov.net

Курс «Алгоритмы и структуры данных»

Сибирский государственный университет телекоммуникаций и информатики (Новосибирск)

Осенний семестр, 2015

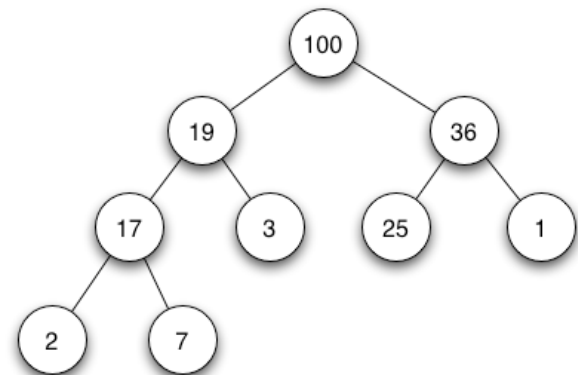
Очередь с приоритетом (Priority queue)

- **Очередь с приоритетом (Priority queue)** – это очередь, в которой элементы имеют приоритет (вес)
- **Поддерживаемые операции:**
 - **Insert(*key*, *value*)** – добавляет в очередь значение *value* с приоритетом (весом, ключом) *key*
 - **DeleteMin/DeleteMax** – удаляет из очереди элемент с мин./макс. приоритетом (ключом)
 - **Min/Max** – возвращает элемент с мин./макс. ключом
 - **DecreaseKey** – изменяет значение ключа заданного элемента
 - **Merge(q_1 , q_2)** – сливает две очереди в одну

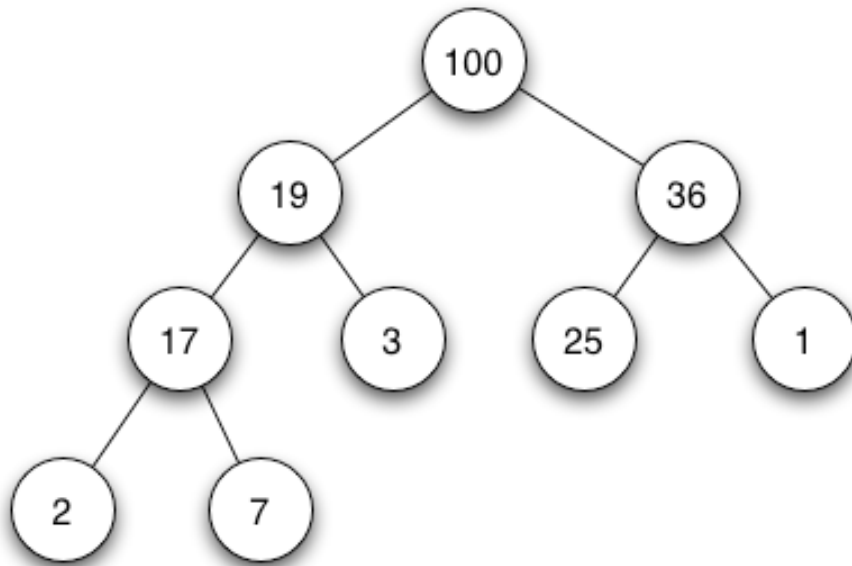
Значение (Value)	Приоритет (Priority)
Слон	3
Кит	1
Лев	15

Бинарная куча (binary heap)

- **Бинарная куча (пирамида, сортирующее дерево, binary heap)** – это бинарное дерево, удовлетворяющее следующим условиям:
 - а) приоритет (ключ) любой вершины не меньше (\geq), приоритета её потомков
 - б) дерево является завершённым бинарным деревом (complete binary tree) – все уровни заполнены слева направо, возможно за исключением последнего

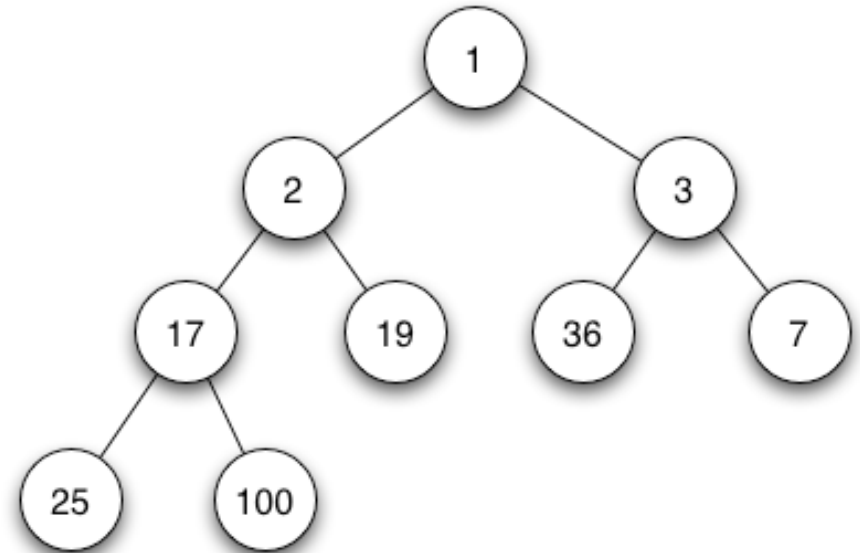


Двоичная куча (Binary heap)



max-heap

Приоритет любой вершины
не меньше (\geq),
приоритета потомков



min-heap

Приоритет любой вершины
не больше (\leq),
приоритета потомков

Очередь с приоритетом (Priority queue)

- В таблице приведены трудоемкости операций очереди с приоритетом (в худшем случае, worst case)
- Символом '*' отмечена амортизированная сложность операций

Операция	Binary heap	Binomial heap	Fibonacci heap	Pairing heap	Brodal heap
FindMin	$\Theta(1)$	$O(\log n)$	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
DeleteMin	$\Theta(\log n)$	$\Theta(\log n)$	$O(\log n)^*$	$O(\log n)^*$	$O(\log n)$
Insert	$\Theta(\log n)$	$O(\log n)$	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$
DecreaseKey	$\Theta(\log n)$	$\Theta(\log n)$	$\Theta(1)^*$	$O(\log n)^*$	$\Theta(1)$
Merge/Union	$\Theta(n)$	$\Omega(\log n)$	$\Theta(1)$	$\Theta(1)^*$	$\Theta(1)$

Биномиальные кучи (Binomial heaps)

- **Биномиальная куча** (binomial heap, пирамида) – это эффективно *сливаемая куча* (mergeable heap)
- В биномиальной куче слияние (merge, union) двух куч выполняется за время $O(\log n)$
- Биномиальные кучи формируются на основе *биномиальных деревьев* (binomial trees)

Биномиальное дерево (Binomial tree)

- **Биномиальное дерево B_k** (Binomial tree) – это рекурсивно определяемое дерево высоты k , в котором:

- ☐ количество узлов равно 2^k

- ☐ количество узлов на уровне $i = 0, 1, \dots, k$

$$\binom{k}{i} = \frac{k!}{i! (k - i)!}$$

(количество сочетаний из k по i)

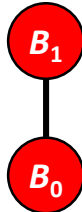
- ☐ корень имеет k дочерних узлов:
 - первый дочерний узел (самый левый) – дерево B_{k-1}
 - второй дочерний узел (второй слева) – дерево B_{k-2}
 - ...
 - k -й дочерний узел (самый правый) – дерево B_0

Биномиальное дерево (Binomial tree)

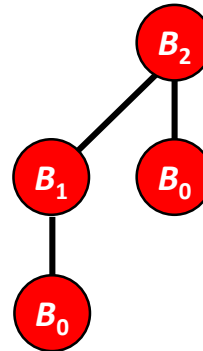
Биномиальное
дерево B_0
($n = 2^0 = 1$)



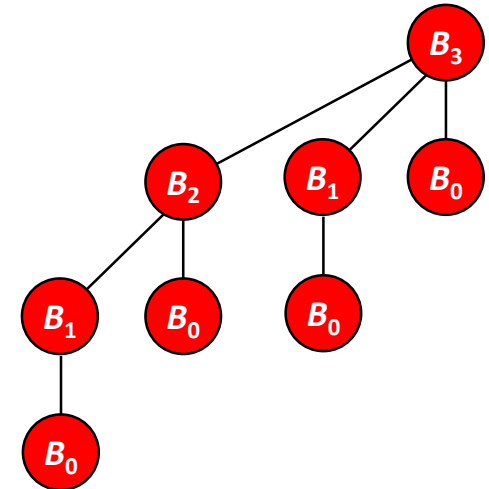
Биномиальное
дерево B_1
($n = 2^1 = 2$)



Биномиальное
дерево B_2
($n = 2^2 = 4$)



Биномиальное
дерево B_3
($n = 2^3 = 8$)



- Максимальная степень узла
в биномиальном дереве с n вершинами равна $O(\log n)$

Биномиальная куча (Binomial heap)

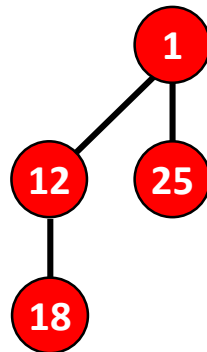
- **Биномиальная куча (Binomial heap)** – это множество биномиальных деревьев, которые удовлетворяют свойствам биномиальных куч:
 - 1) каждое *биномиальное дерево упорядочено (ordered)* в соответствии со свойствами неубывающей или невозрастающей кучи (min-heap/max-heap):
ключ узла не меньше/не больше ключа его родителя
 - 2) для любого целого $k \geq 0$ в куче имеется не более одного биномиального дерева, чей корень имеет степень k
- Биномиальная куча содержащая n узлов состоит не более чем из $\lfloor \log(n) \rfloor + 1$ биномиальных деревьев

Биномиальная куча (Binomial heap)

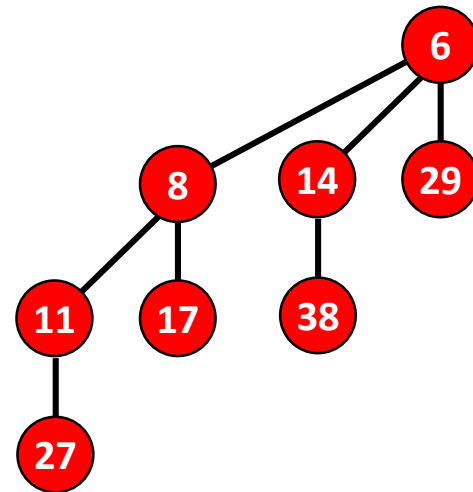
Биномиальное
дерево B_0
($n = 2^0 = 1$)



Биномиальное
дерево B_2
($n = 2^2 = 4$)



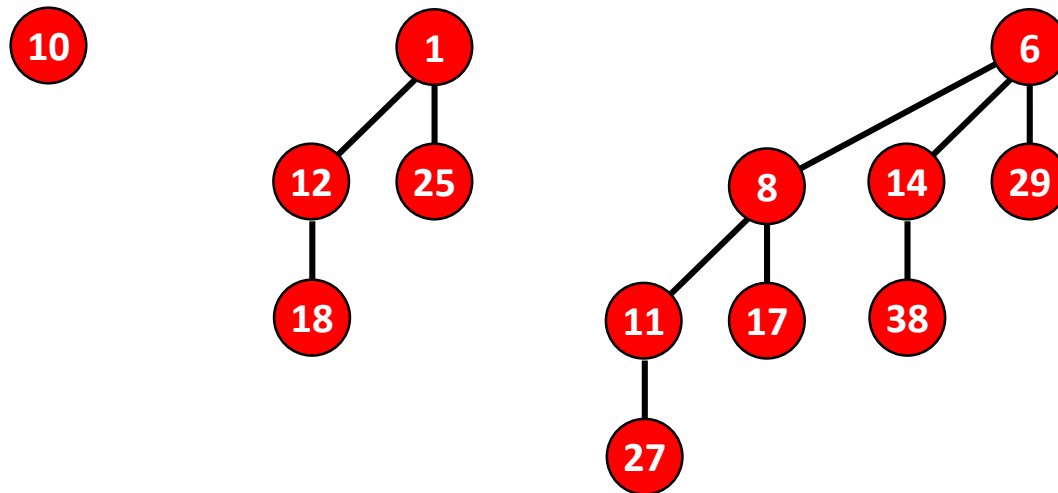
Биномиальное
дерево B_3
($n = 2^3 = 8$)



Биномиальная куча из 13 узлов
(упорядоченные биномиальные деревья B_0 , B_2 и B_3)

Биномиальное дерево (Binomial tree)

- Пусть биномиальная куча содержит n узлов
- Если записать n в двоичной системе исчисления, то номера ненулевых битов будут соответствовать степеням биномиальных деревьев, образующих кучу

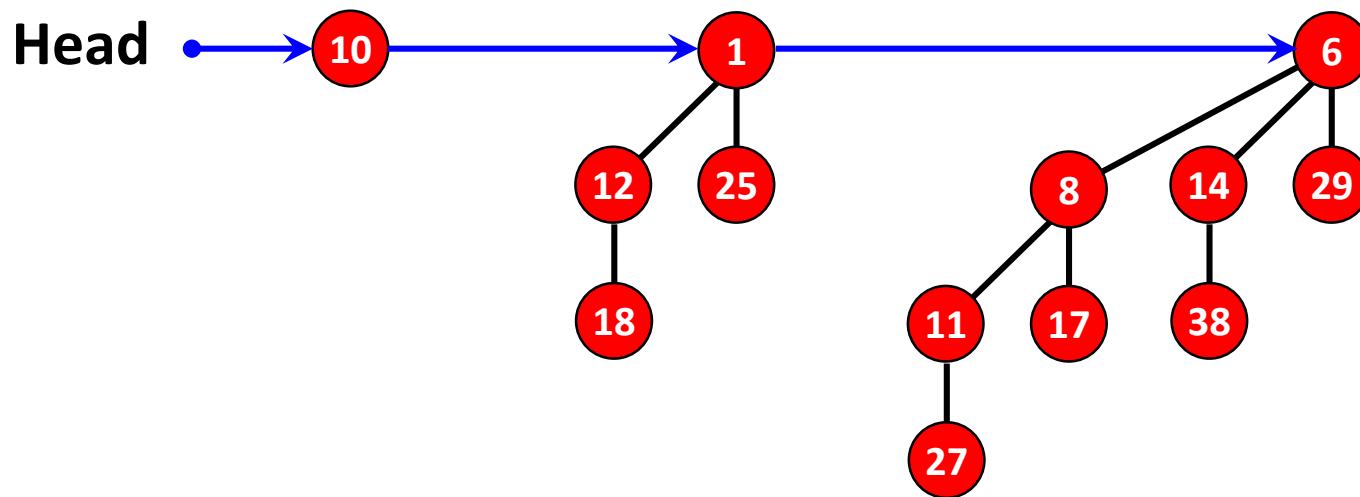


Биномиальная куча из 13 узлов (деревья B_0 , B_2 и B_3)

$$13_{10} = 1101_2$$

Биномиальное дерево (Binomial tree)

- Корни биномиальных деревьев биномиальной кучи хранятся в односвязном списке – *списке корней* (root list)
- В списке корней *узлы упорядочены* по возрастанию их степеней (степеней корней биномиальных деревьев кучи)

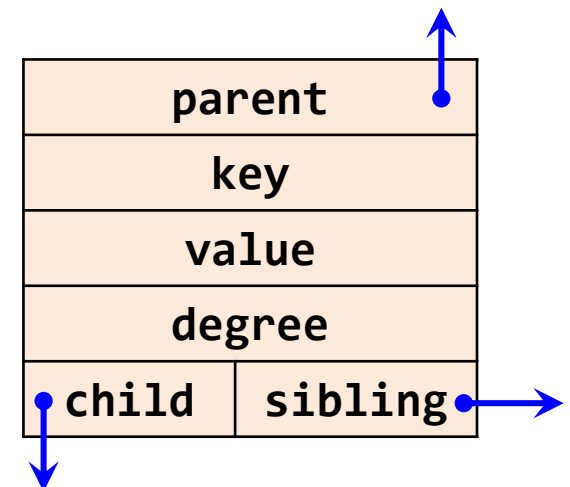


Биномиальная куча из 13 узлов (деревья B_0 , B_2 и B_3)

$$13_{10} = 1101_2$$

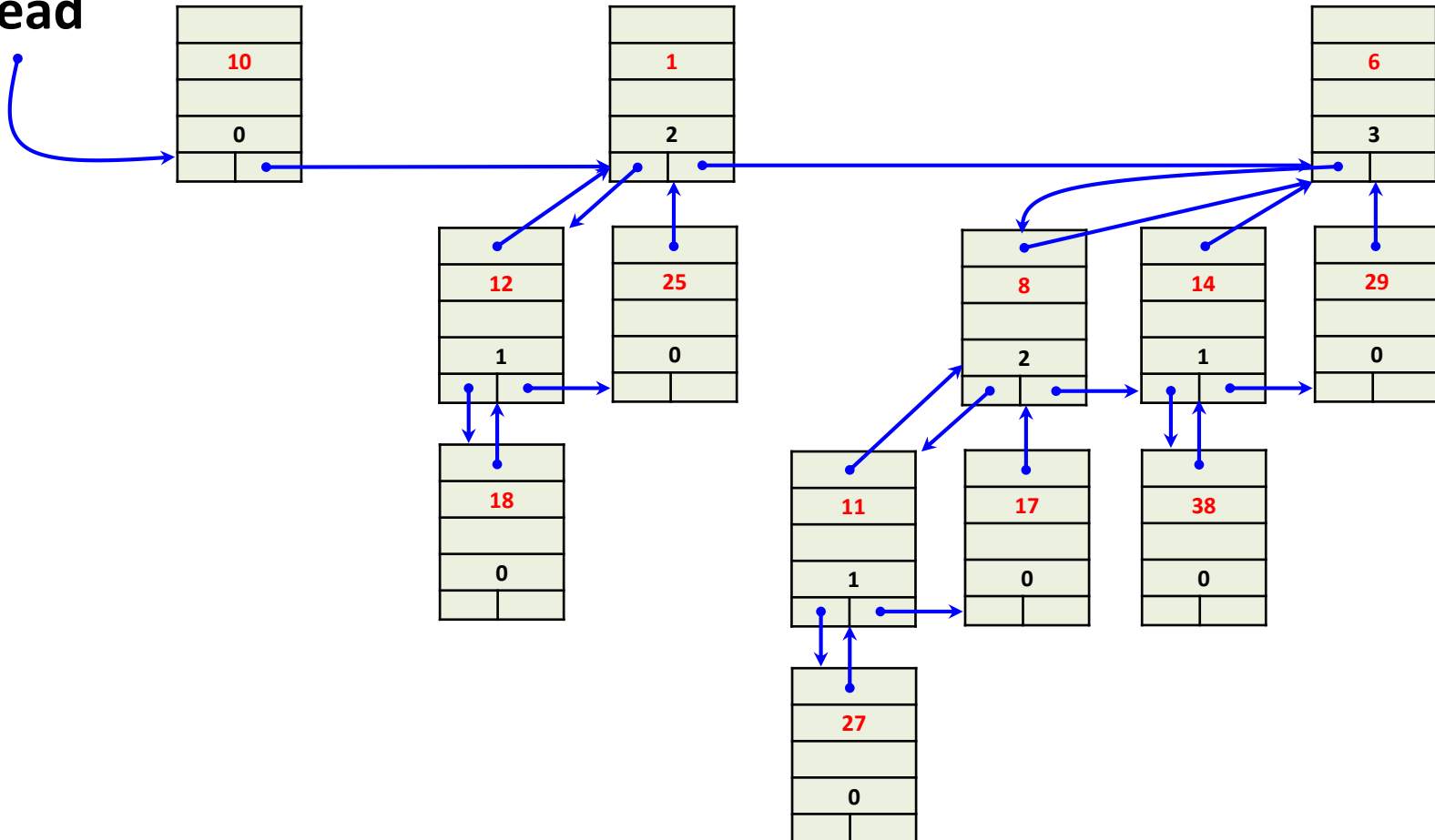
Узел биномиальной кучи

- Каждый узел биномиальной кучи (биномиального дерева) содержит следующие поля:
 - **key** – приоритет узла (вес, ключ)
 - **value** – данные
 - **degree** – количество дочерних узлов
 - **parent** – указатель на родительский узел
 - **child** – указатель на крайний левый дочерний узел
 - **sibling** – указатель на правый сестринский узел



Представление биномиальных куч

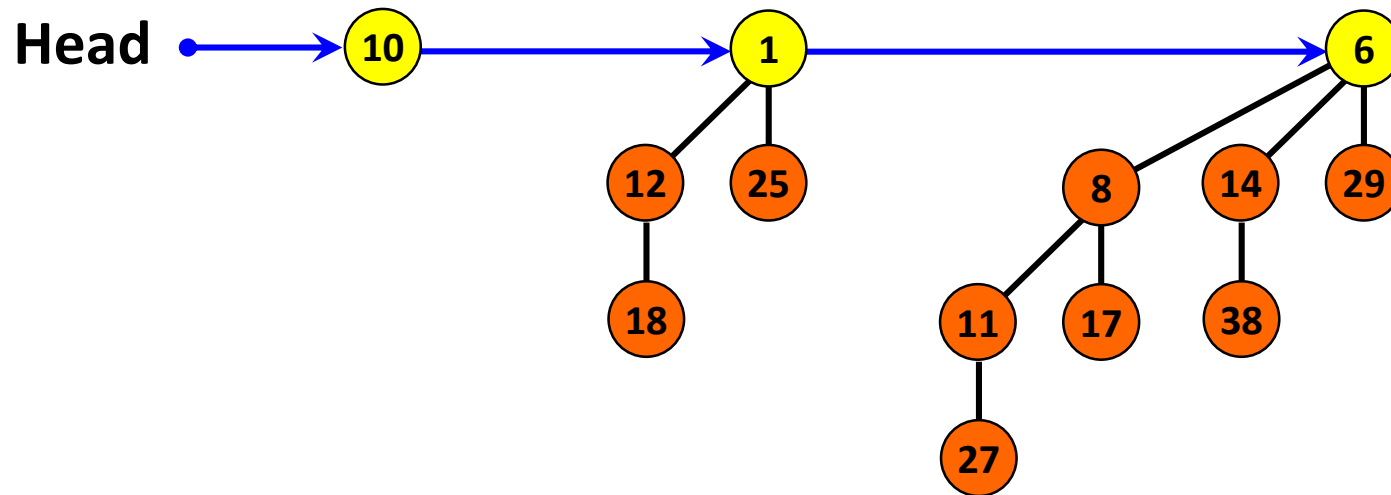
Head



Биномиальная куча из 13 узлов (деревья B_0 , B_2 и B_3)

Поиск минимального узла (FindMin)

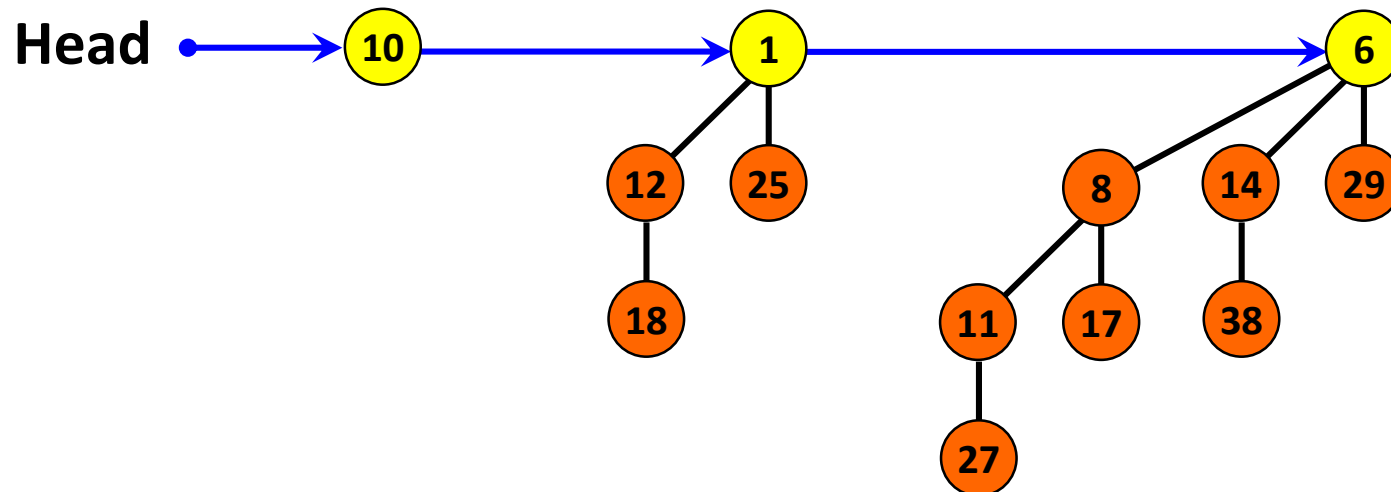
- В корне каждого биномиального дерева хранится его минимальный/максимальный ключ
- Для поиска минимального/максимального элемента в биномиальной куче требуется пройти по списку из $\lfloor \log(n) \rfloor + 1$ корней



Поиск минимального узла (FindMin)

```
function BinomialHeapMin(heap)
  x = heap
  min.key = Infinity
  while x != NULL do
    if x.key < min.key then
      min = x
    x = x.sibling
  end while
  return min
end function
```

$$T_{Min} = O(\log n)$$



Поиск минимального узла (FindMin)

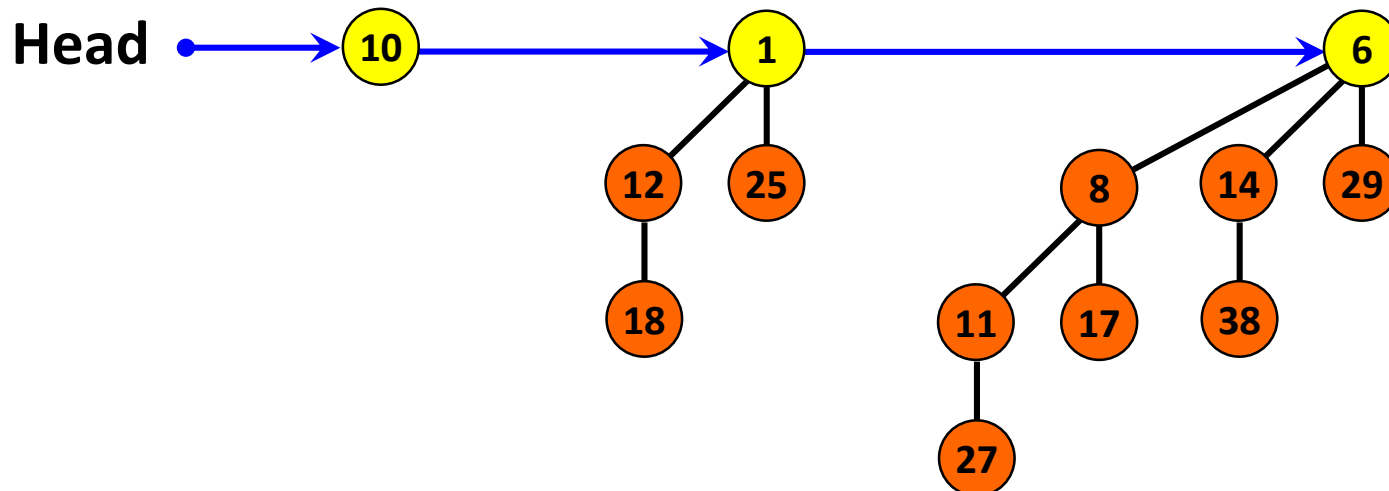
```
function BinomialHeapMin(heap)
  x = heap
  min.key = Infinity
```

$$T_{Min} = O(\log n)$$

Как реализовать поиск минимального/максимального ключа за время $O(1)$?

Поддерживать указатель на корень дерева (узел), в котором находится экстремальный ключ?

end function

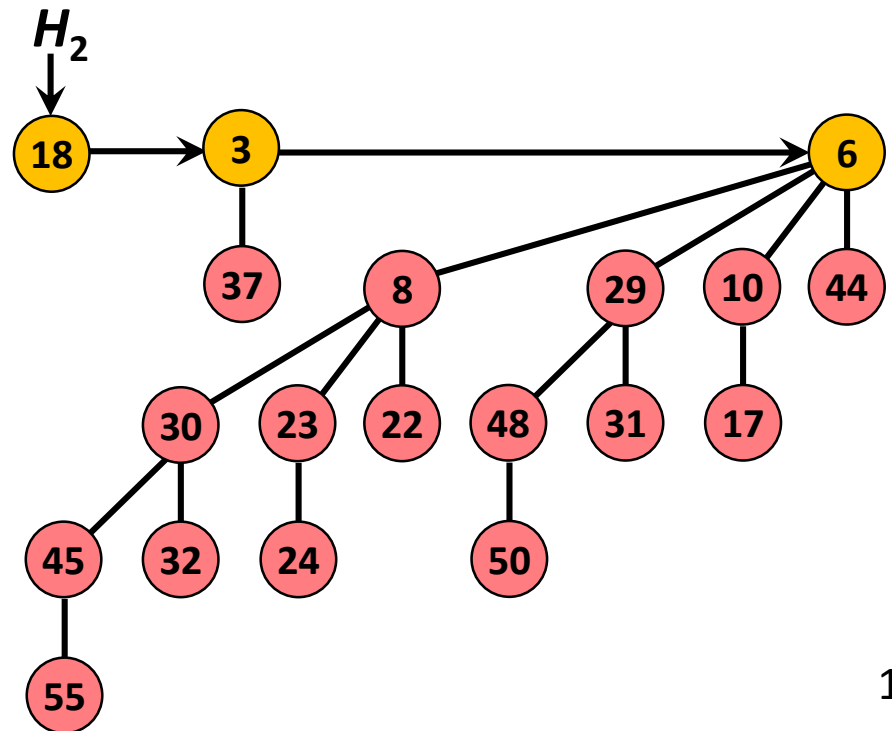
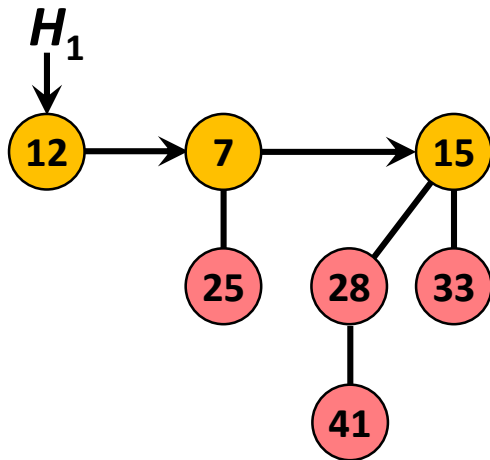


Слияние куч (Union)

- Для слияния (union, merge) двух куч H_1 и H_2 в новую кучу H необходимо:
 1. Слить списки корней H_1 и H_2 в один упорядоченный список
 2. Восстановить свойства биномиальной кучи H
- После слияния списков корней известно, что в куче H имеется не более двух корней с одинаковой степенью и они соседствуют

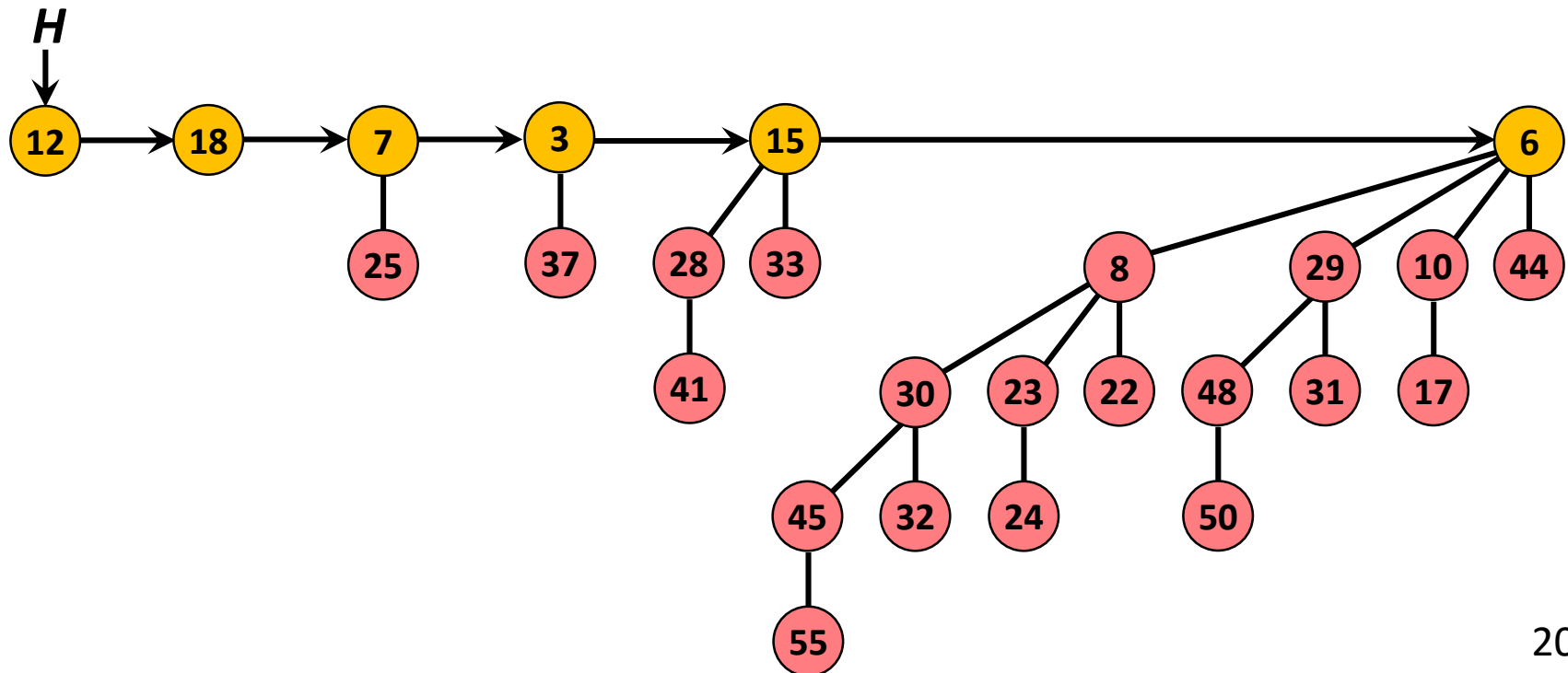
Слияние списков корней

- Списки корней H_1 и H_2 упорядочены по возрастанию степеней узлов
- Слияние выполняется аналогично слиянию упорядоченных подмассивов в MergeSort (сортировке слиянием)



Слияние списков корней

- Списки корней H_1 и H_2 упорядочены по возрастанию степеней узлов
- Слияние выполняется аналогично слиянию упорядоченных подмассивов в MergeSort (сортировке слиянием)

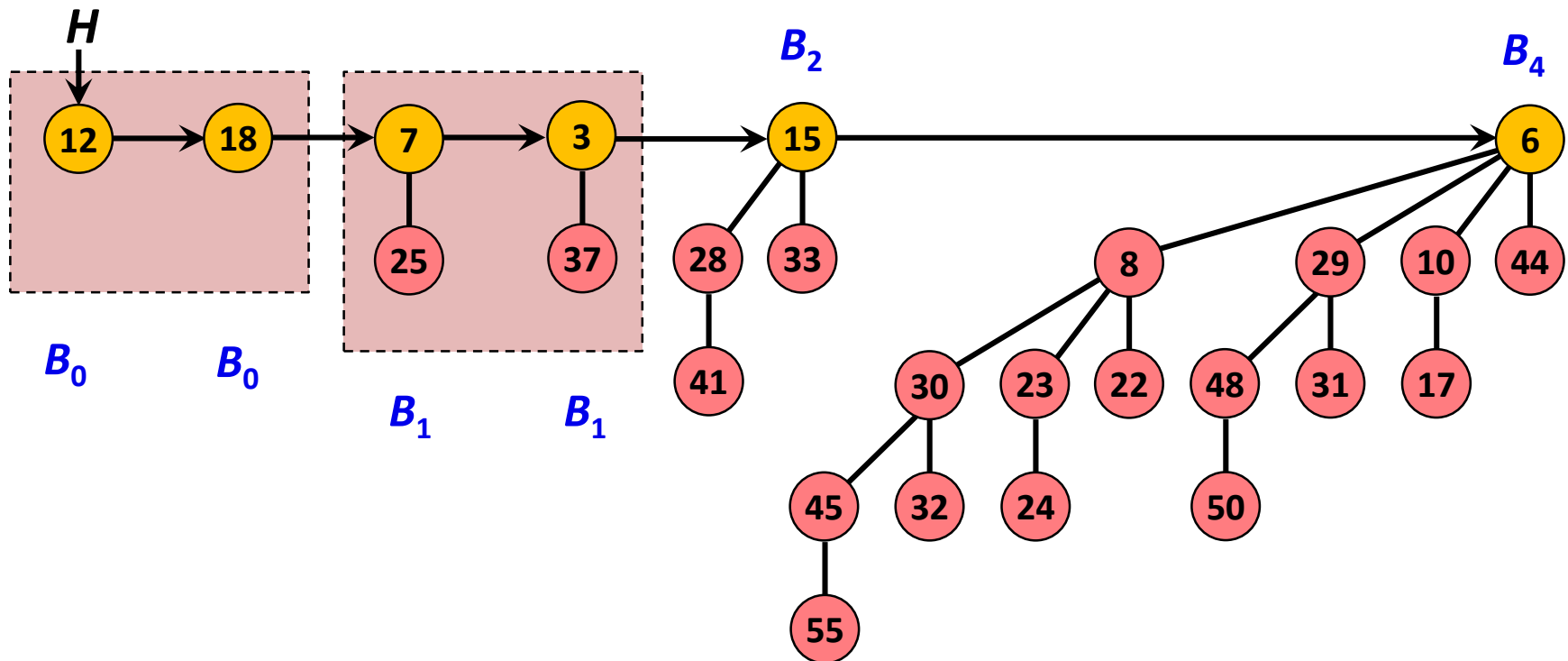


Слияние списков корней

```
function BinomialHeapListMerge(h1, h2)
  h = NULL
  while h1 != NULL && h2 != NULL do
    if h1.degree <= h2.degree then
      LinkedList_AddEnd(h, h1)
      h1 = h1.next
    else
      LinkedList_AddEnd(h, h2)
      h2 = h2.next
    end if
  end while
  while h1 != NULL do
    LinkedList_AddEnd(h, h1)
    h1 = h1.next
  end while
  while h2 != NULL do
    LinkedList_AddEnd(h, h2)
    h2 = h2.next
  end while
  return h
end function
```

$$T_{ListMerge} = O(\log(\max(n_1, n_2)))$$

Слияние куч (Union)



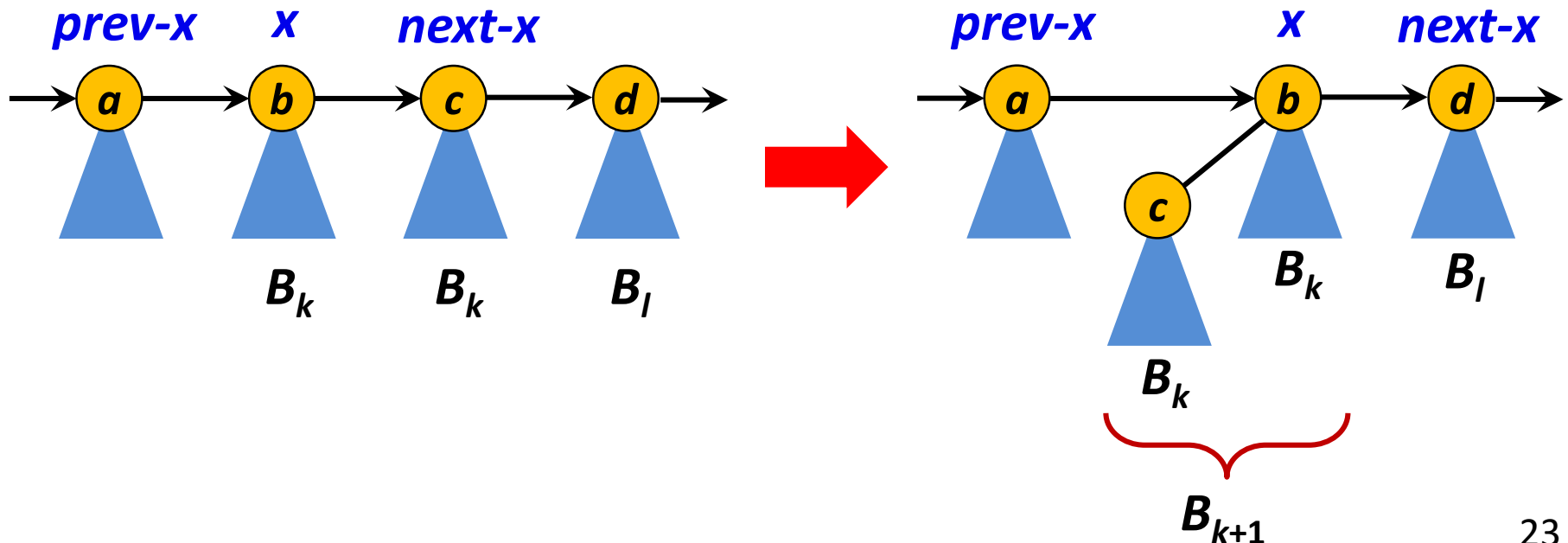
Ситуация после слияния списков корней

- Свойство 1 биномиальной кучи выполняется
- Свойство 2 **не выполняется**: два дерева B_0 и два дерева B_1

Случай 3

$x.degree = next-x.degree \neq next-x.sibling.degree$
 $x.key \leq next-x.key$

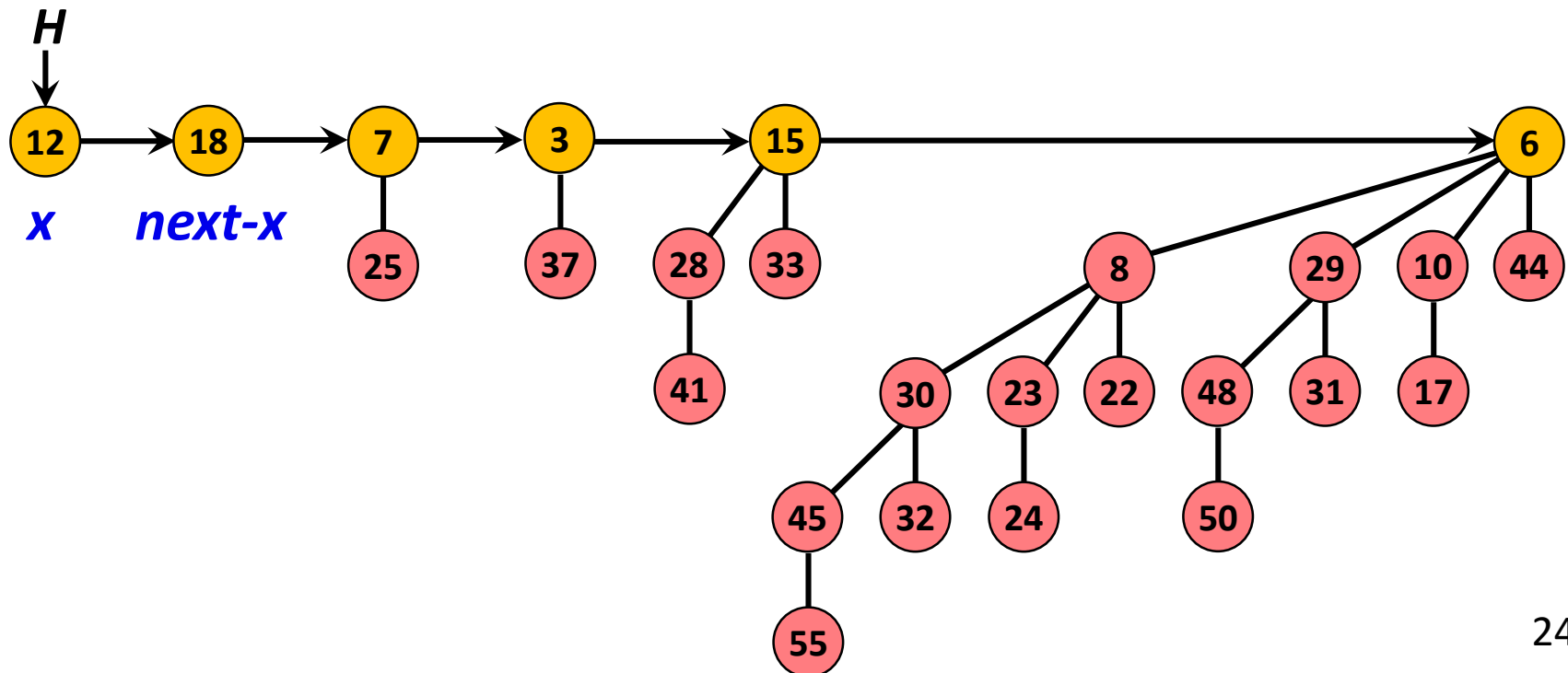
```
x.sibling = next-x.sibling  
BinomialTreeLink(next-x, x)  
next-x = x.sibling
```



Случай 3

$$x.degree = next-x.degree \neq next-x.sibling.degree$$
$$x.key \leq next-x.key$$

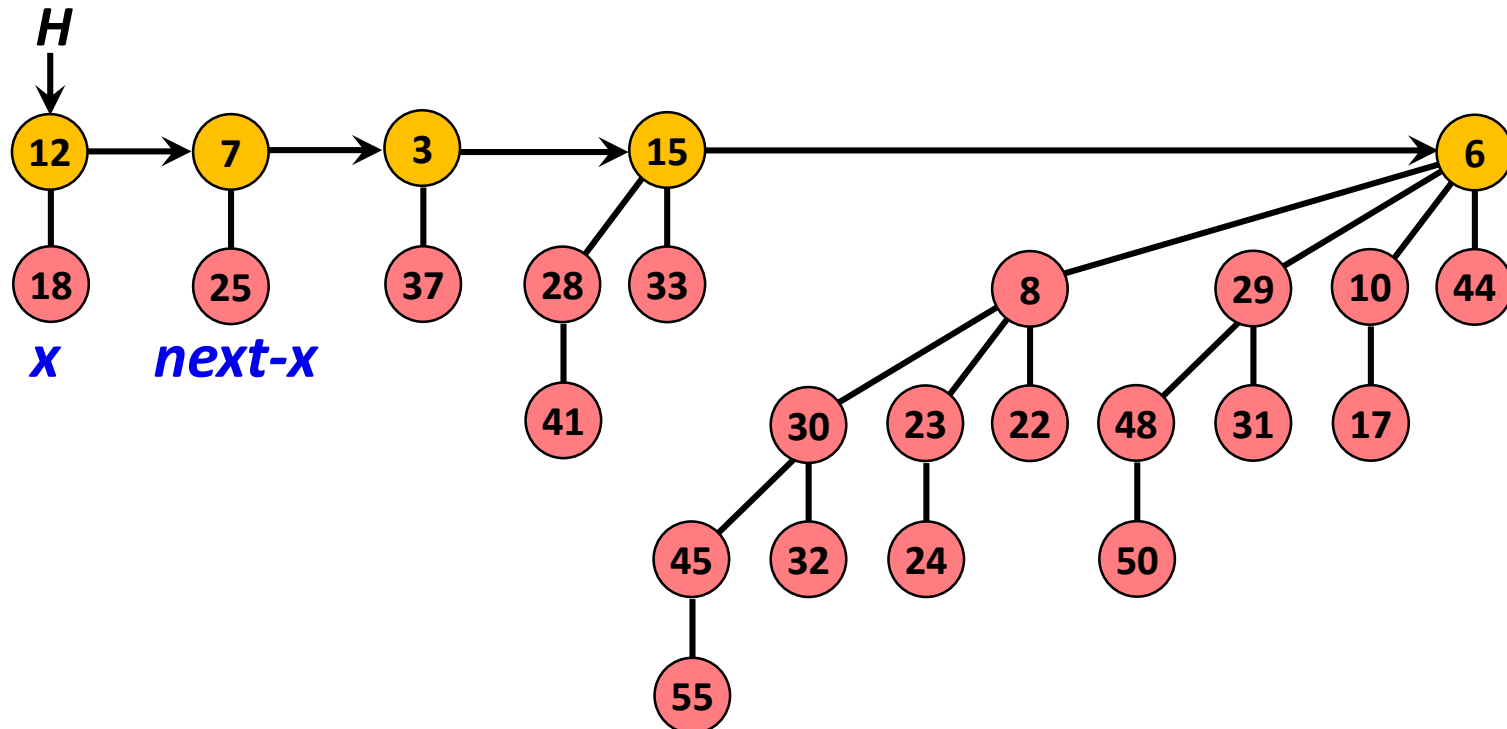
- Деревья x и $next-x$ связываются
- Узел $next-x$ становится левым дочерним узлом x



Случай 3

$$x.degree = next-x.degree \neq next-x.sibling.degree$$
$$x.key \leq next-x.key$$

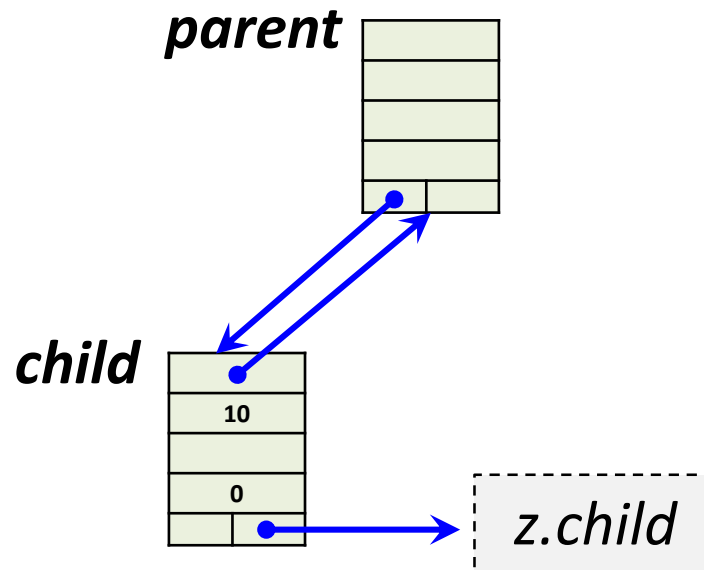
```
x.sibling = next-x.sibling  
BinomialTreeLink(next-x, x)  
next-x = x.sibling
```



Связывание биномиальных деревьев

```
function BinomialTreeLink(child, parent)
  child.parent = parent
  child.sibling = parent.child
  parent.child = child
  parent.degree = parent.degree + 1
end function
```

$$T_{TreeLink} = O(1)$$



Случай 2

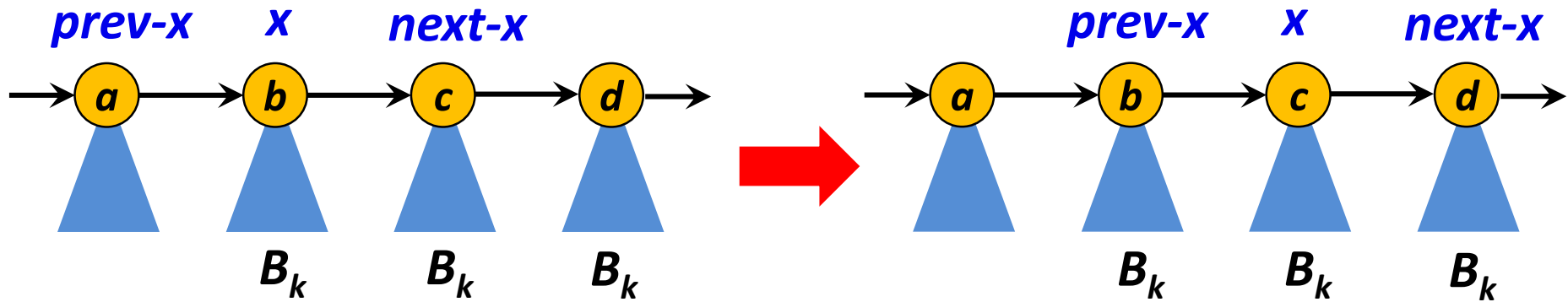
$$x.degree = next-x.degree = next-x.sibling.degree$$

/ Перемещаем указатели по списку корней */*

`prev-x = x`

`x = next-x`

`next-x = x.sibling`



Случай 2

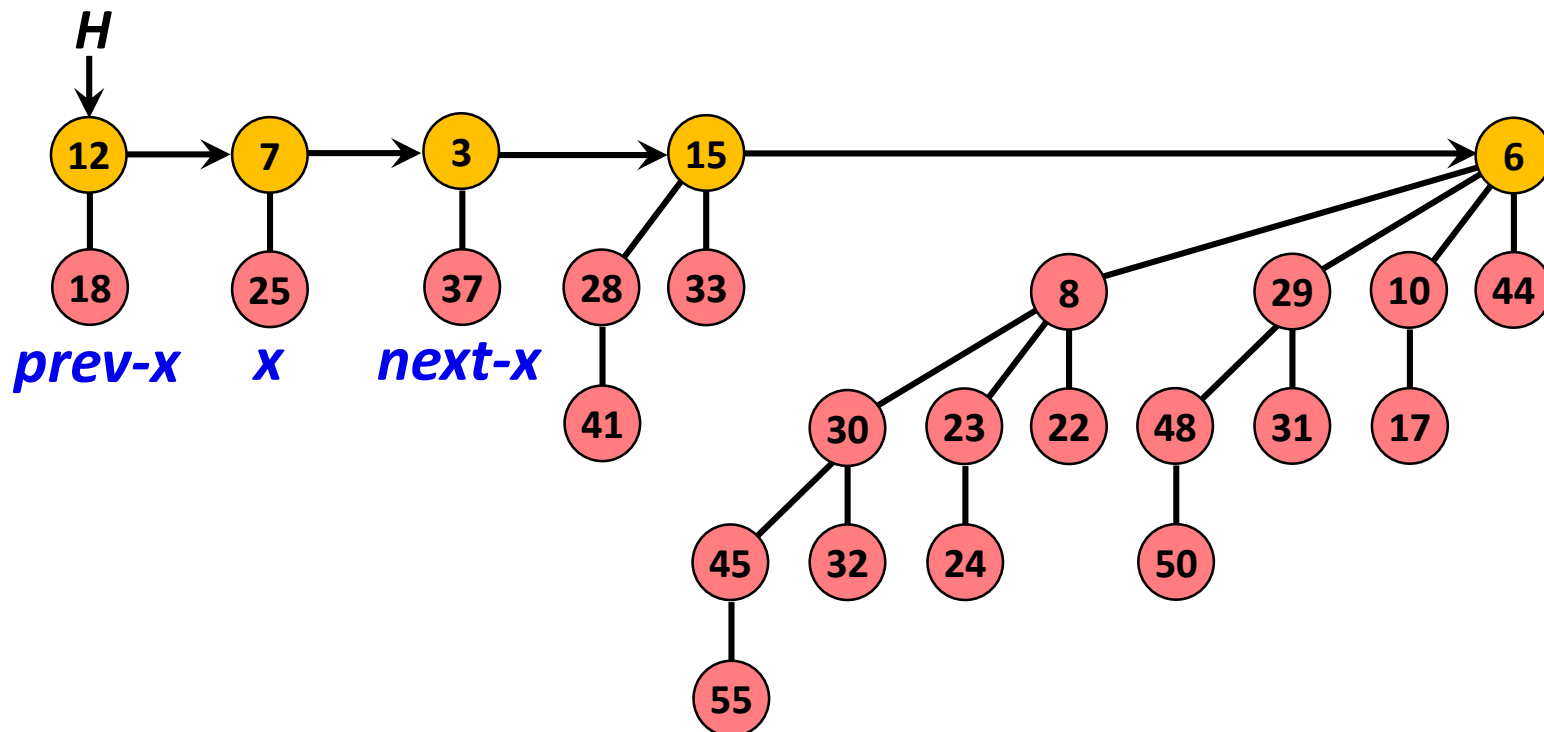
$$x.degree = next-x.degree = next-x.sibling.degree$$

/ Перемещаем указатели по списку корней */*

`prev-x = x`

`x = next-x`

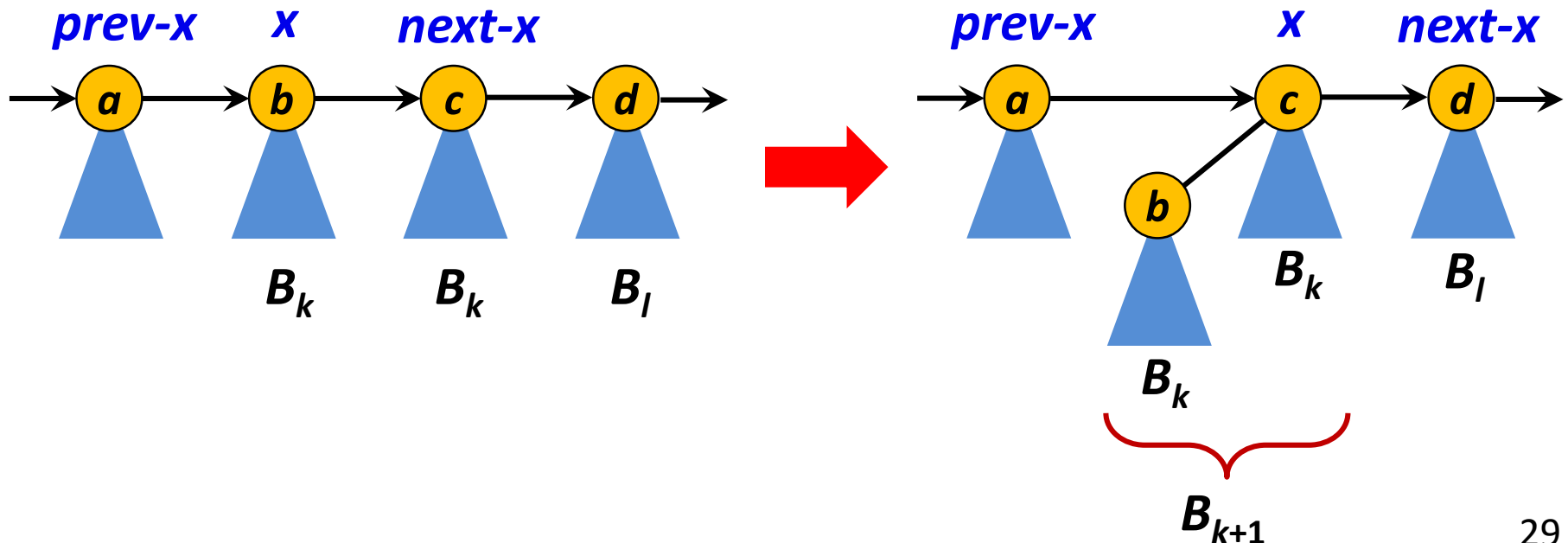
`next-x = x.sibling`



Случай 4

$x.degree = next-x.degree \neq next-x.sibling.degree$
 $x.key > next-x.key$

```
prev-x.sibling = next-x  
BinomialTreeLink(x, next-x)  
x = next-x  
next-x = x.sibling
```



Случай 1

$$x.degree \neq next-x.degree$$

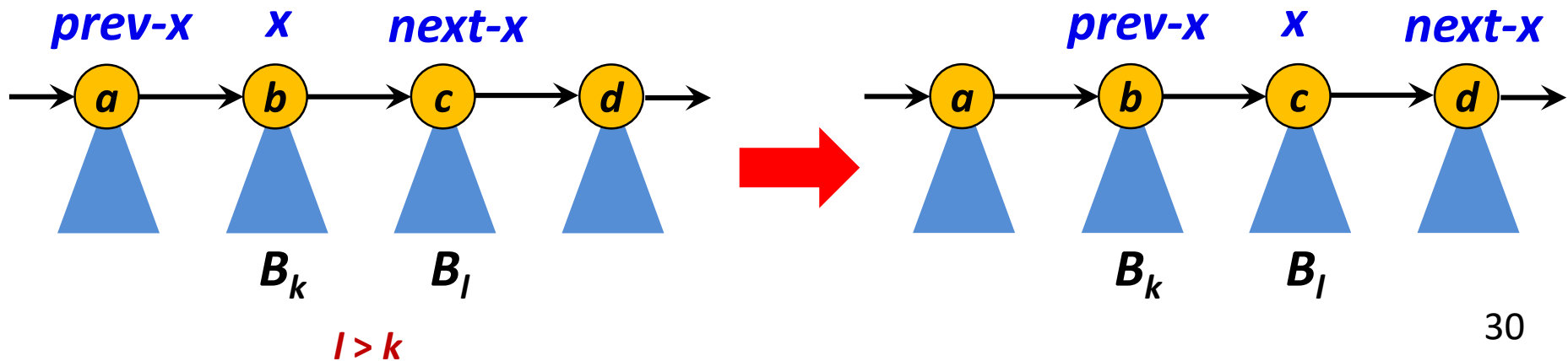
- Узел x – корень дерева B_k
- Узел $next-x$ – корень дерева B_l , $l > k$

/* Перемещаем указатели по списку корней */

prev-x = x

x = next-x

next-x = x.sibling



Слияние биномиальных куч (Union)

```
function BinomialHeapUnion(h1, h2)
    h = BinomialHeapListMerge(h1, h2)
    prev-x = NULL
    x = h
    next-x = x.sibling
    while next-x != NULL do
        if (x.degree != next-x.degree) OR
            (next-x.sibling != NULL AND
             next-x.sibling.degree = x.degree)
        then
            prev-x = x                                /* Случаи 1 и 2 */
            x = next-x
        else if x.key <= next-x.key then
            x.sibling = next-x.sibling                /* Случай 3 */
            BinomialTreeLink(next-x, x)
```

Слияние биномиальных куч (Union)

```
else
    /* Случай 4 */
    if prev-x = NULL then
        h = next-x
    else
        prev-x.sibling = next-x
    end if
    BinomialTreeLink(x, next-x)
    x = next-x
end if
next-x = x.sibling
end while
return h
end function
```

$$T_{Union} = O(\log n)$$

Слияние биномиальных куч (Union)

- Вычислительная сложность слияния двух биномиальных куч в худшем случае равна $O(\log(n))$

- Длина списка корней не превышает

$$\log(n_1) + \log(n_2) + 2 = O(\log(n))$$

- Цикл `while` в функции `BinomialHeapUnion` выполняется не более $O(\log(n))$ раз
- На каждой итерации цикла указатель перемещается по списку корней вправо на одну позицию или удаляется один узел – это требует времени $O(1)$

Вставка узла (Insert)

- Создаем биномиальную кучу из одного узла x – биномиального дерева B_0
- Сливаем исходную кучу H и кучу из узла x

```
function BinomialHeapInsert(h, key, value)
    x.key = key
    x.value = value
    x.degree = 0
    x.parent = NULL
    x.child = NULL
    x.sibling = NULL
    return BinomialHeapUnion(h, x)
end function
```

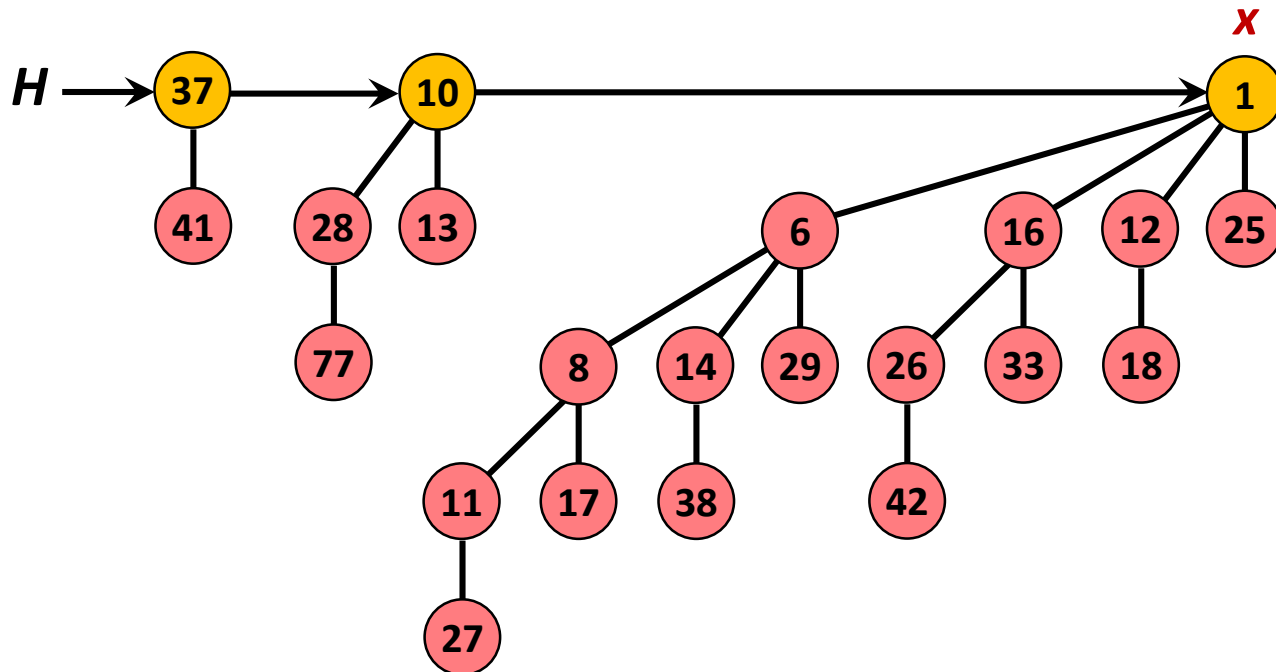
$$T_{Insert} = O(\log n)$$

Удаление минимального узла (DeleteMin)

1. В списке корней кучи H отыскиваем корень x с минимальным ключом и удаляем x из списка корней (разрываю связь)
2. Инициализируем пустую кучу Z
3. Меняем порядок следования дочерних узлов корня x на обратный, у каждого дочернего узла устанавливаем поле `parent` в `NULL`
4. Устанавливаем заголовок кучи Z на первый элемент нового списка корней
5. Сливаем кучи H и Z
6. Возвращаем x

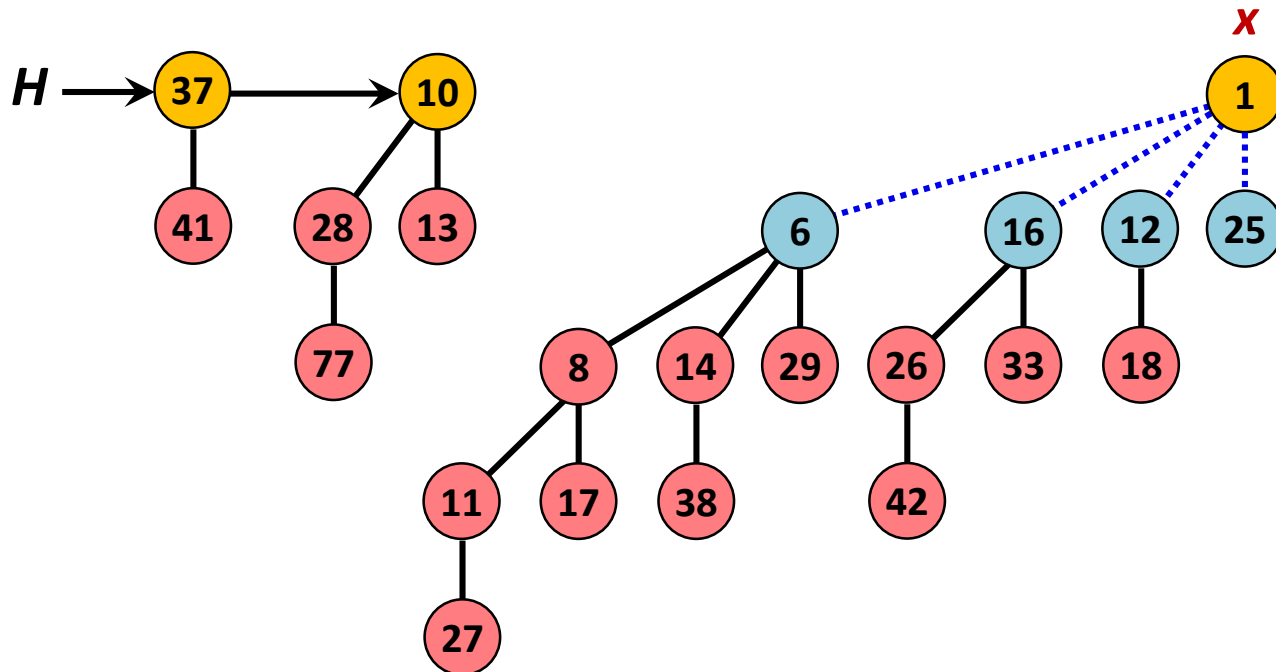
Удаление минимального узла (DeleteMin)

- В списке корней кучи H отыскиваем корень x с минимальным ключом и удаляем x из списка корней (разрываем связь)



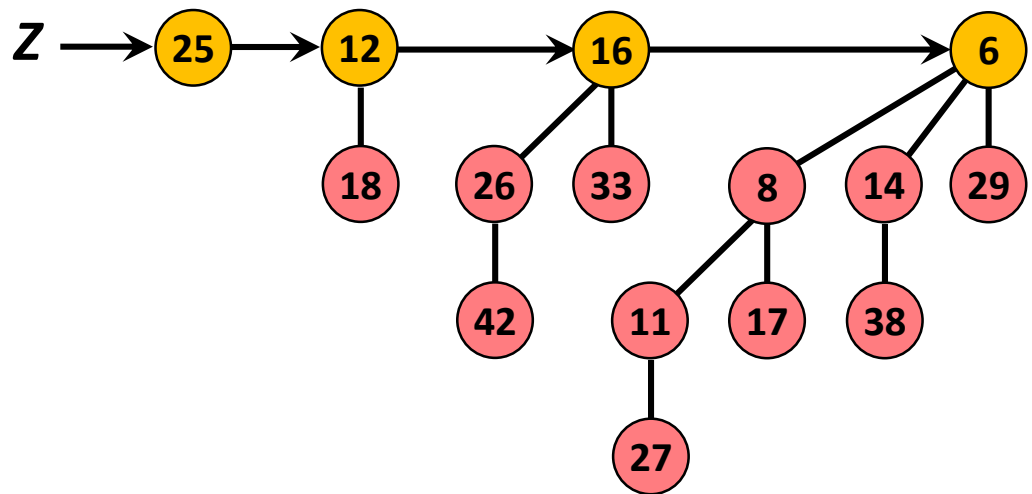
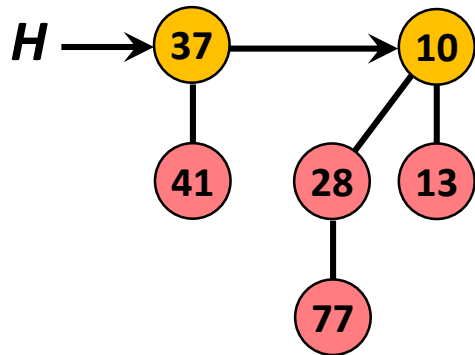
Удаление минимального узла (DeleteMin)

- Меняем порядок следования дочерних узлов корня x на обратный, у каждого дочернего узла устанавливаем поле `parent` в `NULL`
- Дочерние узлы корня x образуют биномиальную кучу Z



Удаление минимального узла (DeleteMin)

- Сливаем кучи H и Z



Удаление минимального узла (DeleteMin)

```
function BinomialHeapDeleteMin(h)
  /* Lookup and unlink min node */
  x = h
  xmin.key = Infinity
  prev = NULL
  while x != NULL do
    if x.key < xmin.key then
      xmin = x
      prevmin = prev
    end if
    prev = x
    x = x.sibling
  end while
  if prevmin != NULL then
    prevmin.sibling = xmin.sibling
  else
    h = xmin.sibling
```

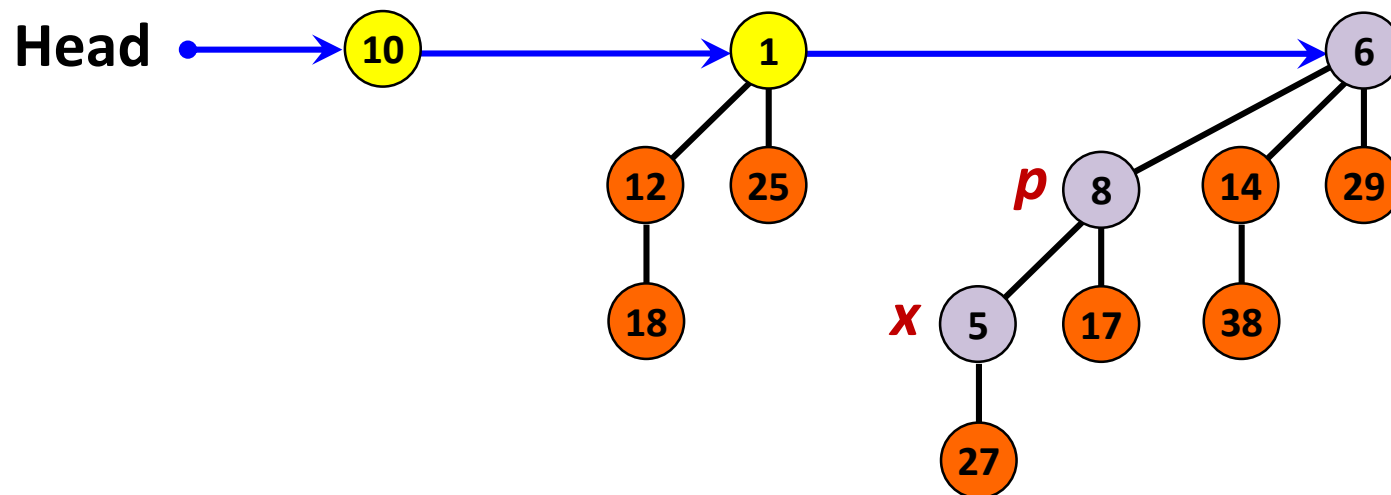
Удаление минимального узла (DeleteMin)

```
/* Reverse linked list */  
child = xmin.child  
prev = NULL  
while child != NULL do  
    sibling = child.sibling  
    child.sibling = prev  
    prev = child  
    child = sibling  
end while  
return BinomialHeapUnion(h, prev)  
end function
```

$$T_{DeleteMin} = O(\log n)$$

Уменьшение ключа (DecreaseKey)

1. Получаем указатель на узел x и изменяем у него ключ ($newkey \leq x.key$)
2. Проверяем значение ключа родительского узла, если он меньше ключа x , то выполняем обмен ключей (и данных), повторяем обмены пока не поднимемся до корня текущего биномиального дерева



Уменьшение ключа (DecreaseKey)

```
function BinomialHeapDecreaseKey(h, x, key)
    if x.key < k then
        return Error
    x.key = key
    y = x
    z = y.parent
    while z != NULL AND y.key < z.key do
        temp = y.key
        y.key = z.key
        z.key = temp
        y = z
        z = y.parent
    end while
end function
```

$$T_{DecreaseKey} = O(\log n)$$

Удаление узла (Delete)

```
function BinomialHeapDelete(h, x)
    BinomialHeapDecreaseKey(h, x, -Infinity)
    BinomialHeapDeleteMin(h)
end function
```

$$T_{Delete} = O(\log n)$$

Узел биномиального дерева

```
struct bmheap {  
    int key;  
    char *value;  
    int degree;  
    struct bmheap *parent;  
    struct bmheap *child;  
    struct bmheap *sibling;  
};
```

Пример использования bmheap

```
int main()
{
    struct bmheap *h = NULL, *node;

    h = bmheap_insert(h, 10, "10");
    h = bmheap_insert(h, 12, "12");
    h = bmheap_insert(h, 3, "3");
    h = bmheap_insert(h, 56, "56");

    node = bmheap_min(h);
    printf("Min = %d\n", node->key);
    h = bmheap_deletemin(h);

    bmheap_free(h);
    return 0;
}
```

Создание узла биномиального дерева

```
struct bmheap *bmheap_create(int key, char *value)
{
    struct bmheap *h;

    h = (struct bmheap *)malloc(sizeof(*h));
    if (h != NULL) {
        h->key = key;
        h->value = value;
        h->degree = 0;
        h->parent = NULL;
        h->child = NULL;
        h->sibling = NULL;
    }
    return h;
}
```

Поиск минимального элемента

```
struct bmheap *bmheap_min(struct bmheap *h)
{
    struct bmheap *minnode, *node;
    int minkey = ~0U >> 1;          /* INT_MAX */

    for (node = h; node != NULL;
         node = node->sibling)
    {
        if (node->key < minkey) {
            minkey = node->key;
            minnode = node;
        }
    }
    return minnode;
}
```

Слияние биномиальных куч

```
struct bmheap *bmheap_union(struct bmheap *a, struct bmheap *b)
{
    struct bmheap *h, *prevx, *x, *nextx;

    h = bmheap_mergelists(a, b);
    prevx = NULL;
    x = h;
    nextx = h->sibling;
    while (nextx != NULL) {
        if ((x->degree != nextx->degree) ||
            (nextx->sibling != NULL &&
             nextx->sibling->degree == x->degree))
        {
            /* Cases 1 & 2 */
            prevx = x;
            x = nextx;
        }
    }
```


Слияние биномиальных куч

```
    else if (x->key <= nextx->key) {
        /* Case 3 */
        x->sibling = nextx->sibling;
        bmheap_linktrees(nextx, x);
    } else {
        /* Case 4 */
        if (prevx == NULL) {
            h = nextx;
        } else {
            prevx->sibling = nextx;
        }
        bmheap_linktrees(x, nextx);
        x = nextx;
    }
    nextx = x->sibling;
}
return h;
}
```

Слияние списков корней

```
struct bmheap *bmheap_mergelists(struct bmheap *a,
                                struct bmheap *b)
{
    struct bmheap *head, *sibling, *end;

    end = head = NULL;
    while (a != NULL && b != NULL) {
        if (a->degree < b->degree) {
            sibling = a->sibling;
            if (end == NULL) {
                end = a;
                head = a;
            } else {
                end->sibling = a;    /* Add to the end */
                end = a;
                a->sibling = NULL;
            }
            a = sibling;
        }
    }
}
```

Слияние списков корней

```
    else {
        sibling = b->sibling;
        if (end == NULL) {
            end = b;
            head = b;
        } else {
            end->sibling = b;    /* Add to the end */
            end = b;
            b->sibling = NULL;
        }
        b = sibling;
    }
}
```

Слияние списков корней

```
while (a != NULL) {
    sibling = a->sibling;
    if (end == NULL) {
        end = a;
    } else {
        end->sibling = a;
        end = a;
        a->sibling = NULL;
    }
    a = sibling;
}
```

Слияние списков корней

```
while (b != NULL) {
    sibling = b->sibling;
    if (end == NULL) {
        end = b;
    } else {
        end->sibling = b;
        end = b;
        b->sibling = NULL;
    }
    b = sibling;
}
return head;
}
```

Связывание деревьев

```
void bmheap_linktrees(struct bmheap *y,  
                      struct bmheap *z)  
{  
    y->parent = z;  
    y->sibling = z->child;  
    z->child = y;  
    z->degree++;  
}
```

Вставка элемента в биномиальную кучу

```
struct bmheap *bmheap_insert(struct bmheap *h,  
                             int key, char *value)  
{  
    struct bmheap *node;  
  
    if ((node = bmheap_create(key, value)) == NULL)  
        return NULL;  
    if (h == NULL)  
        return node;  
  
    return bmheap_union(h, node);  
}
```

Удаление минимального элемента

```
struct bmheap *bmheap_deletemin(struct bmheap *h)
{
    struct bmheap *x, *prev, *xmin, *prevmin,
                  *child, *sibling;
    int minkey = ~0U >> 1;          /* INT_MAX */

    /* Lookup and unlink min node */
    x = h;
    prev = NULL;
    while (x != NULL) {
        if (x->key < minkey) {
            minkey = x->key;
            xmin = x;
            prevmin = prev;
        }
        prev = x;
        x = x->sibling;
    }
}
```


Удаление минимального элемента

```
if (prevmin != NULL)
    prevmin->sibling = xmin->sibling;
else
    h = xmin->sibling;
```

```
/* Reverse linked list */
```

```
child = xmin->child;
prev = NULL;
while (child != NULL) {
    sibling = child->sibling;
    child->sibling = prev;
    prev = child;
    child = sibling;
}
free(xmin);
return bmheap_union(h, prev);
}
```

Задания

- Прочитать о левосторонней очереди (Leftist heap)
- Прочитать о скошенной очереди (Skew heap)
- Прочитать о куче Бродала (Brodal heap)