Лекция 10.
Биномиальные кучи

Даниил Михайлович Берлизов
Старший преподаватель Кафедры вычислительных систем СибГУТИ
E-mail: sillyhat34@gmail.com

Курс «Структуры и алгоритмы обработки данных»
Осенний семестр, 2021 г.

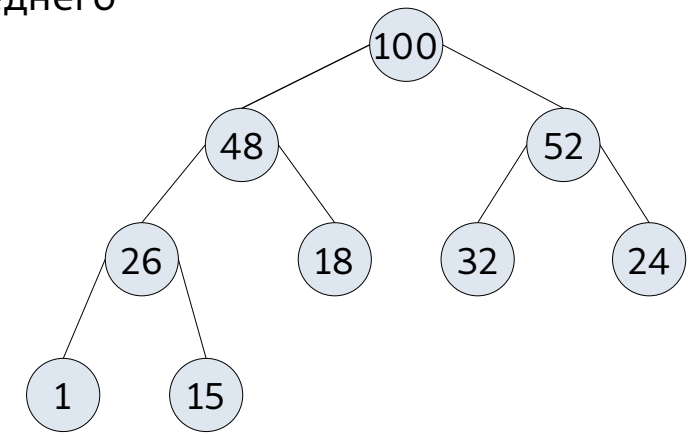
Очередь с приоритетом (priority queue)

- **Очередь с приоритетом** (*priority queue*) — это очередь, в которой элементы имеют приоритет (вес)
- **Поддерживаемые операции:**
- **Insert(*key*, *value*)** — добавление в очередь значения *value* с приоритетом (весом, ключом) *key*
- **DeleteMin / DeleteMax** — удаление элемента с минимальным / максимальным приоритетом
- **Min / Max** — возврат элемента с минимальным / максимальным ключом
- **DecreaseKey** — изменение приоритета (значения ключа) заданного элемента
- **Merge(q_1, q_2)** — слияние двух очередей в одну

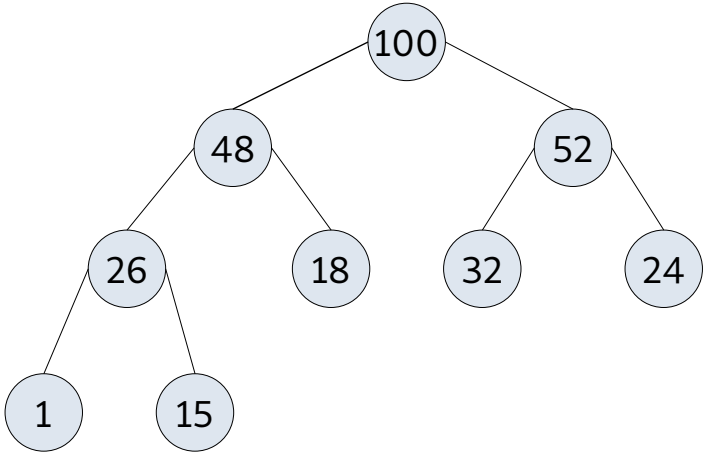
Значение (*value*)	**Приоритет (*key*)**
Слон	4
Кит	1
Борис	12

Бинарная куча (binary heap)

- **Бинарная куча** (*пирамида, сортирующее дерево, binary heap*) — бинарное дерево, удовлетворяющее следующим условиям:
  - Приоритет (ключ) любой вершины **не меньше** (для *max-heap*) или **не больше** (для *min-heap*) приоритета её потомков
  - Дерево является завершённым бинарным деревом (*complete binary tree*) — все уровни заполнены слева направо, возможно за исключением последнего

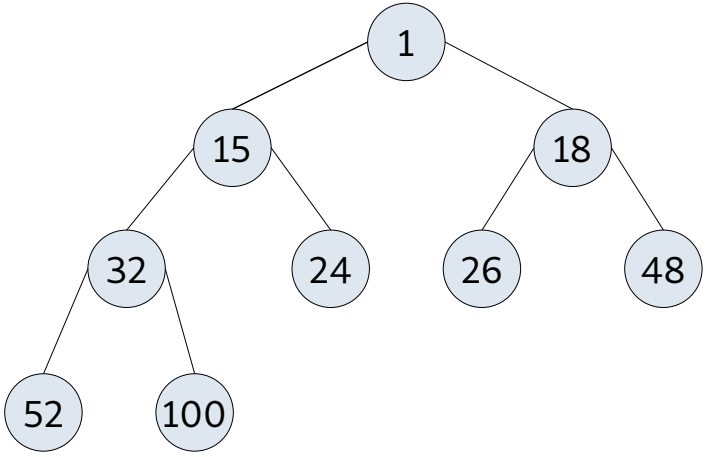


Бинарная куча (binary heap)



max-heap

Приоритет любой вершины **не меньше (≥)** приоритета потомков



min-heap

Приоритет любой вершины **не больше (≤)** приоритета потомков

Очередь с приоритетом (priority queue)

- В таблице приведены трудоёмкости операций различных очередей 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	$\Theta(n)$	$\Omega(\log n)$	$\Theta(1)$	$\Theta(1)$*	$\Theta(1)$

Биномиальная куча (binomial heap)

- **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** (*binomial tree*) — это рекурсивно определяемое дерево высоты k, в котором:
- Количество узлов равно 2^k
- Количество узлов на уровне $i = 0, 1, \ldots, k$ равно

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

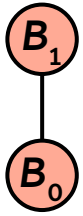
(количество сочетаний из k по i)

- 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

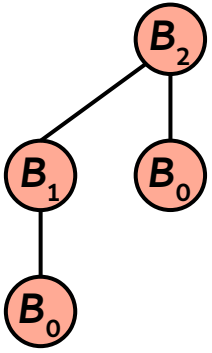
Биномиальное дерево (binomial tree)

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

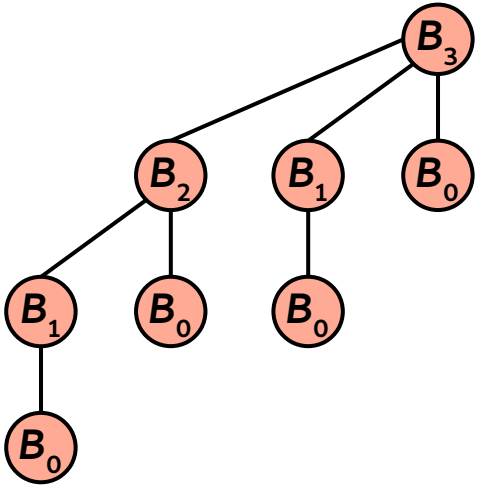
Биномиальное дерево B_1 ($n = 2^1 = 2$)



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)

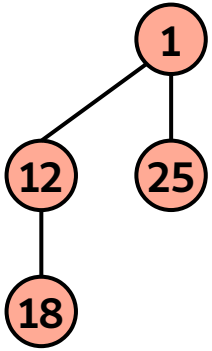
- **Биномиальная куча** (*binomial heap*, *пирамида*) — это множество биномиальных деревьев, которые удовлетворяют свойствам биномиальных куч:
  1. каждое **биномиальное дерево упорядочено** (*ordered*) в соответствии со свойствами неубывающей или невозрастающей кучи (*min-heap*/*max-heap*): ключ узла не меньше/не больше ключа его родителя
  2. для любого целого $k \geq 0$ имеется **не более одного дерева**, чей корень имеет степень k

- Биномиальная куча, содержащая n узлов, состоит не более, чем из $\lfloor \log n + 1 \rfloor$ биномиальных деревьев

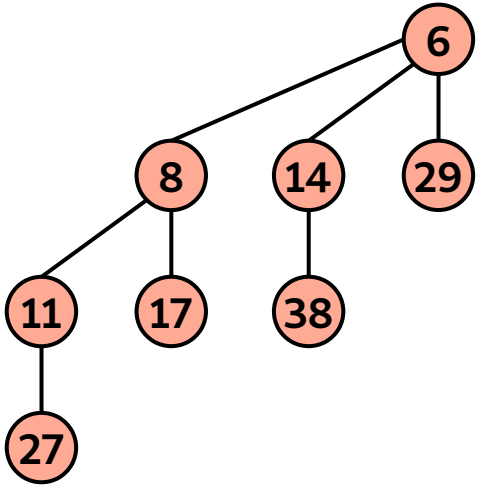
Биномиальная куча (binomial heap)

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

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



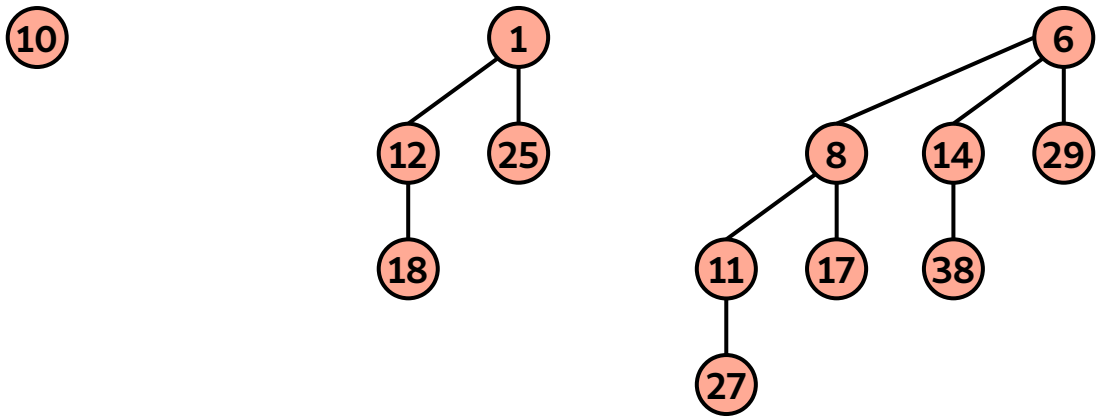
Биномиальное дерево B_3 ($n = 2^3 = 8$)



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

Биномиальная куча (binomial heap)

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

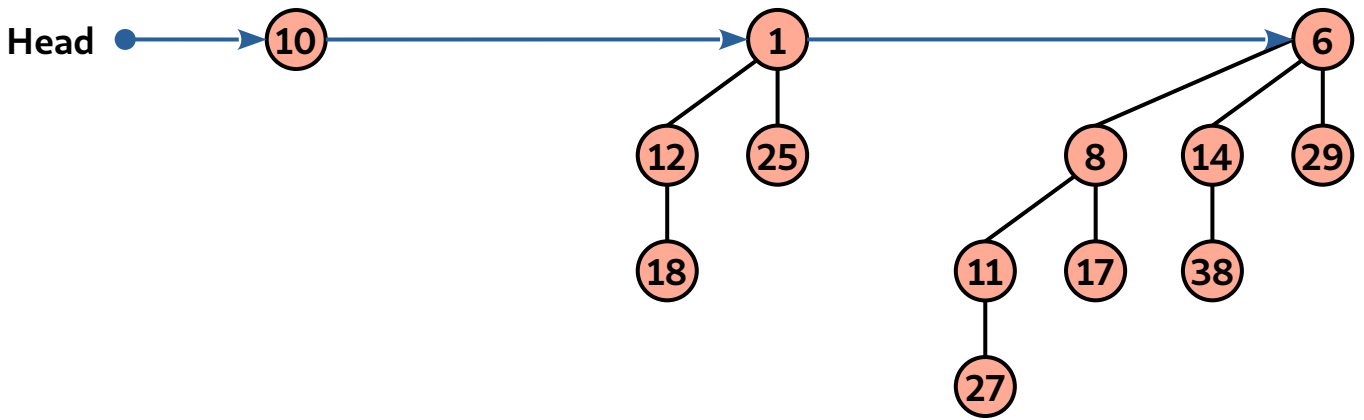


Биномиальная куча из 13 узлов (упорядоченные биномиальные деревья B_0, B_2 и B_3)

$13_{10} = 1101_2$

Биномиальная куча (binomial heap)

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

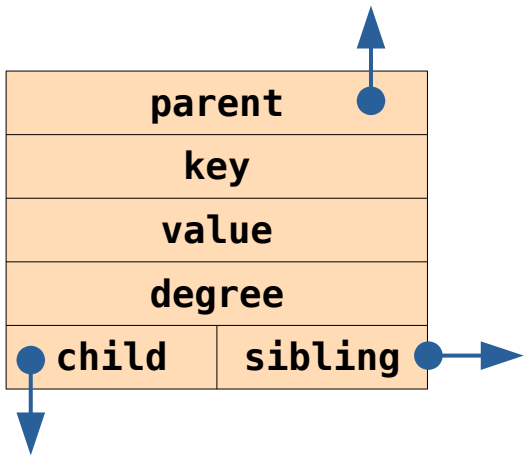


Биномиальная куча из 13 узлов (упорядоченные биномиальные деревья B_0, B_2 и B_3)

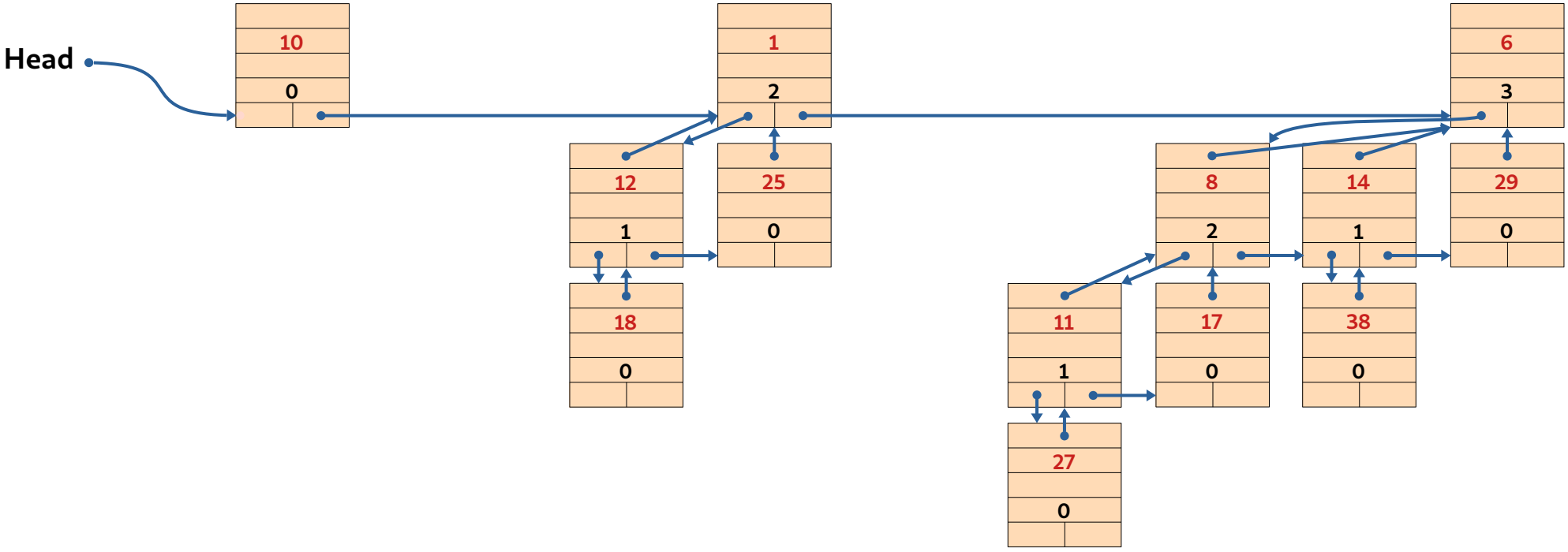
$13_{10} = 1101_2$

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



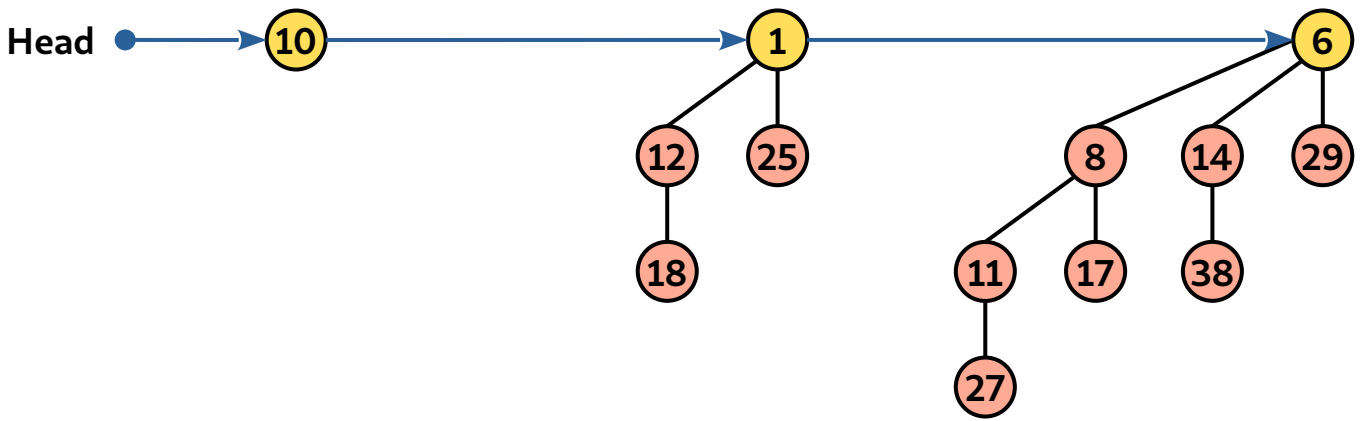
Представление биномиальных куч



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

Поиск минимального узла

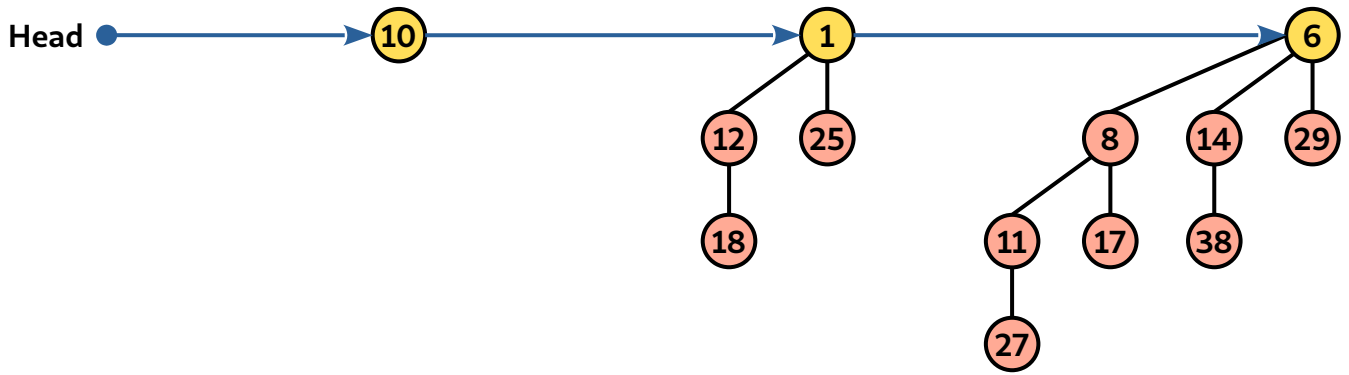
- В корне каждого биномиального дерева хранится его минимальный/максимальный ключ
- Для поиска минимального/максимального элемента в биномиальной куче требуется пройти по списку из $\lfloor \log n + 1 \rfloor$ корней



Поиск минимального узла

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

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



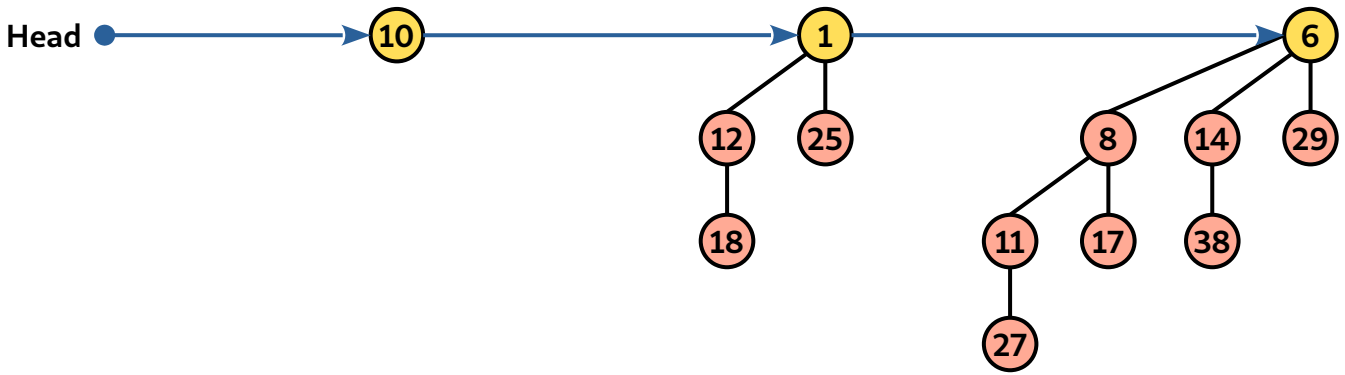
Поиск минимального узла

```
function BinomialHeapFindMin(heap)
    x = heap
    minkey = Infinity
    while x != NULL do
        if x.key < minkey then
```

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

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

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

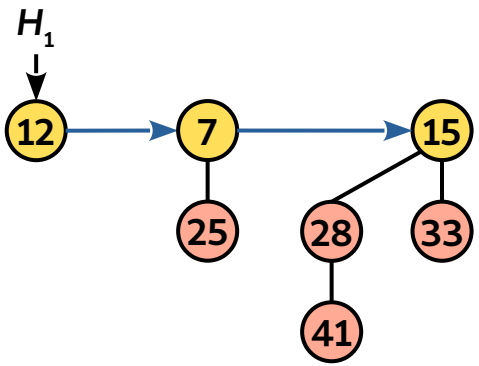


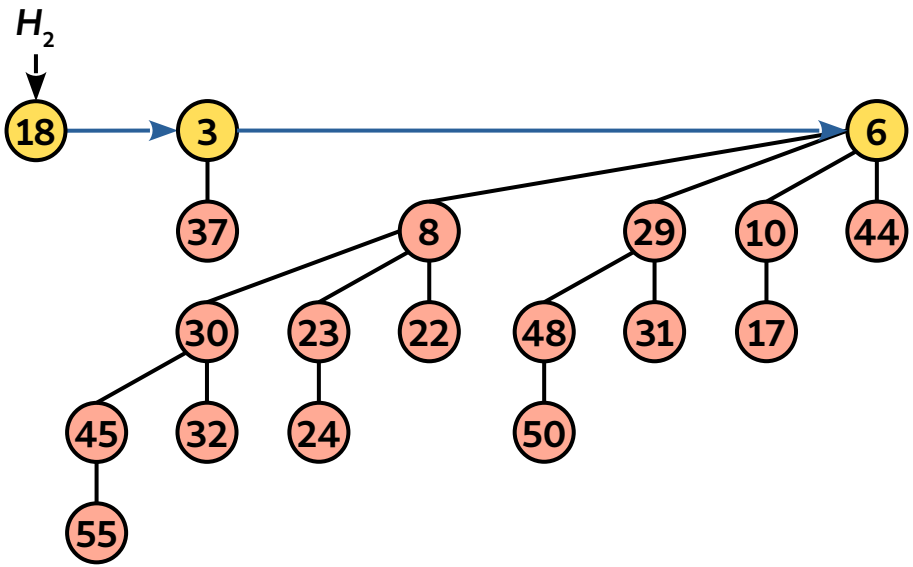
Слияние биномиальных куч (Union)

- 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

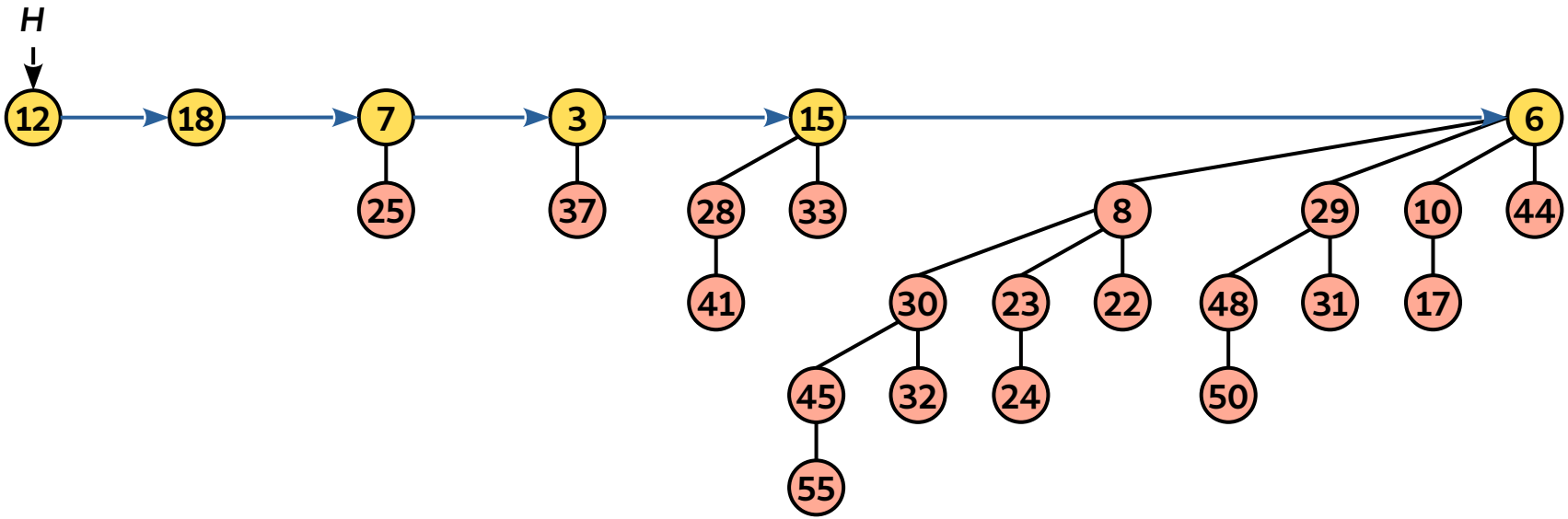
- Списки корней H_1 и H_2 упорядочены по возрастанию степеней узлов
- Слияние выполняется аналогично слиянию упорядоченных подмассивов в сортировке слиянием (MergeSort)





Слияние списков корней

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

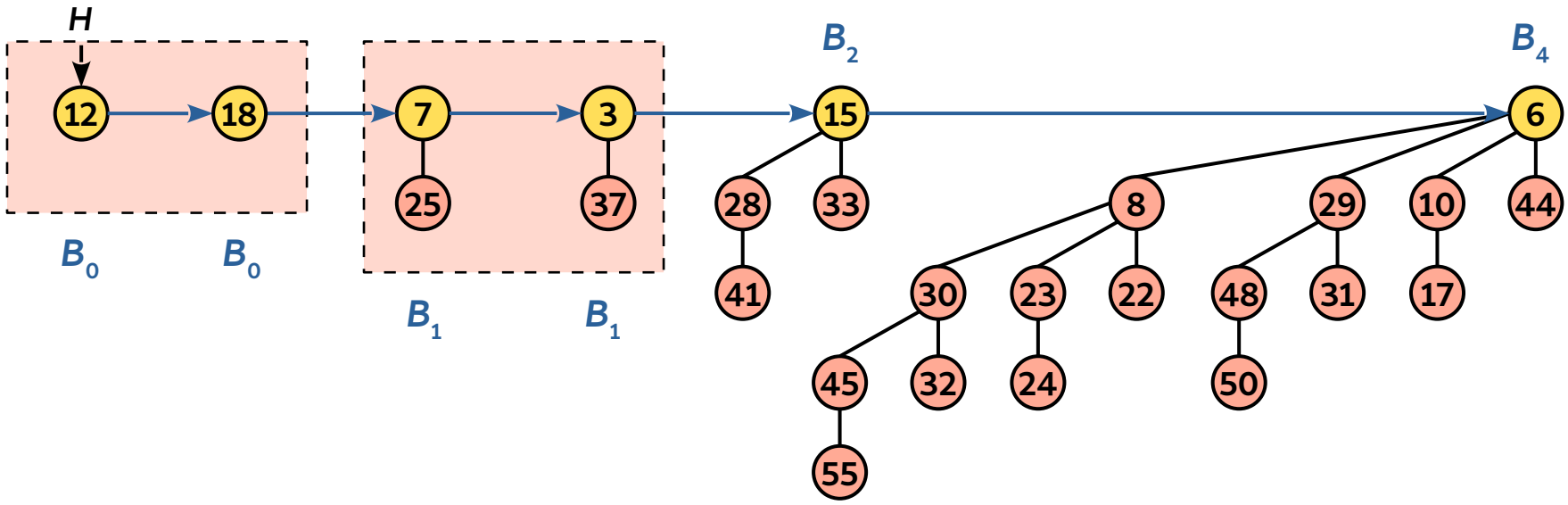


Слияние списков корней

```
function BinomialHeapListMerge(h1, h2)
    h = NULL
    while h1 != NULL AND 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 биномиальных куч (Union)



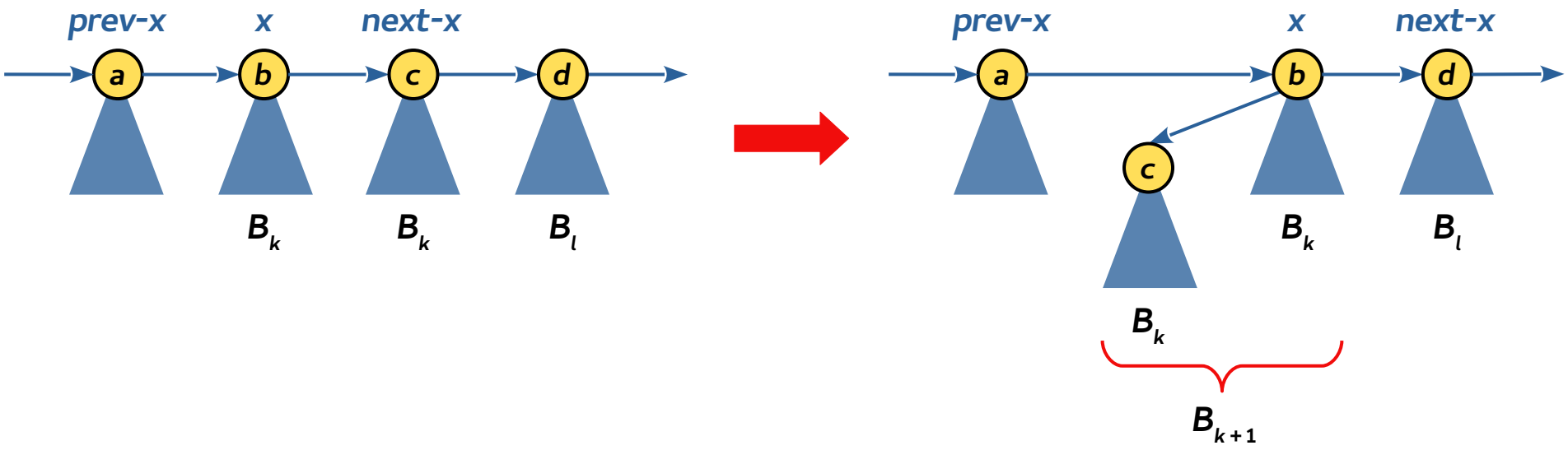
Ситуация после слияния списков корней:

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

Слияние биномиальных куч: случай 3

x.degree = next-x.degree ≠ next-x.sibling.degree, *x.key ≤ next-x.key*

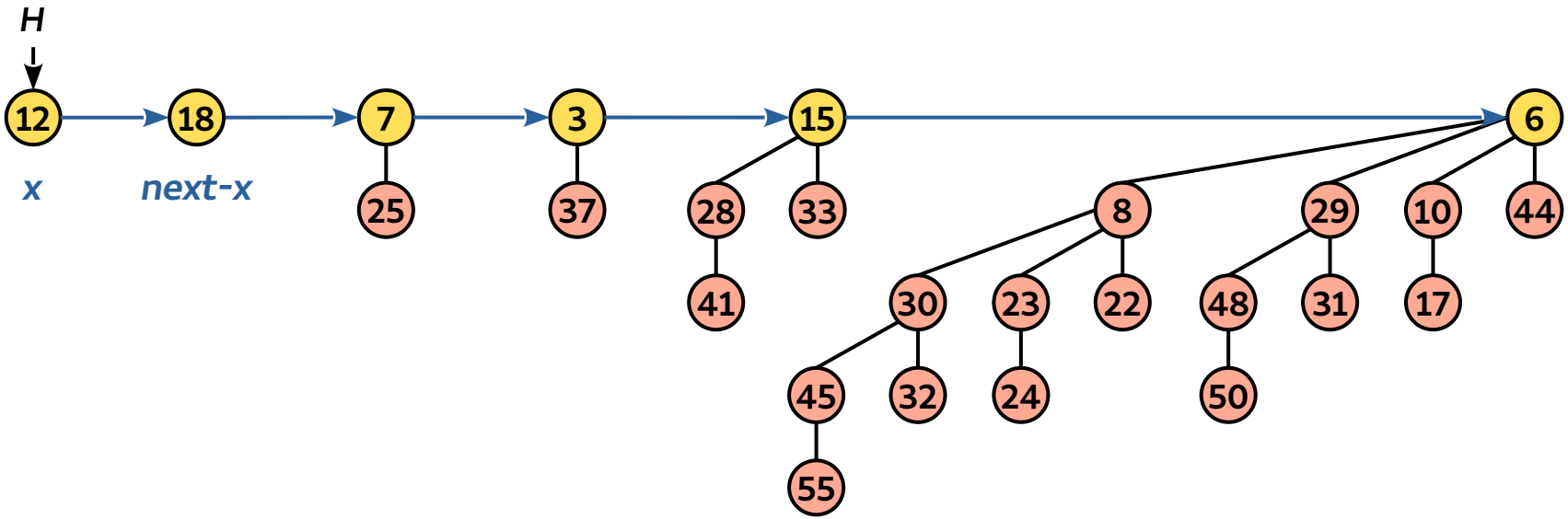
```
x.sibling = next-x.sibling
BinomialTreeLink(next-x, x)
next-x = x.sibling
```



Слияние биномиальных куч: случай 3

$x.degree = next\text{-}x.degree \neq next\text{-}x.sibling.degree$, $x.key \leq next\text{-}x.key$

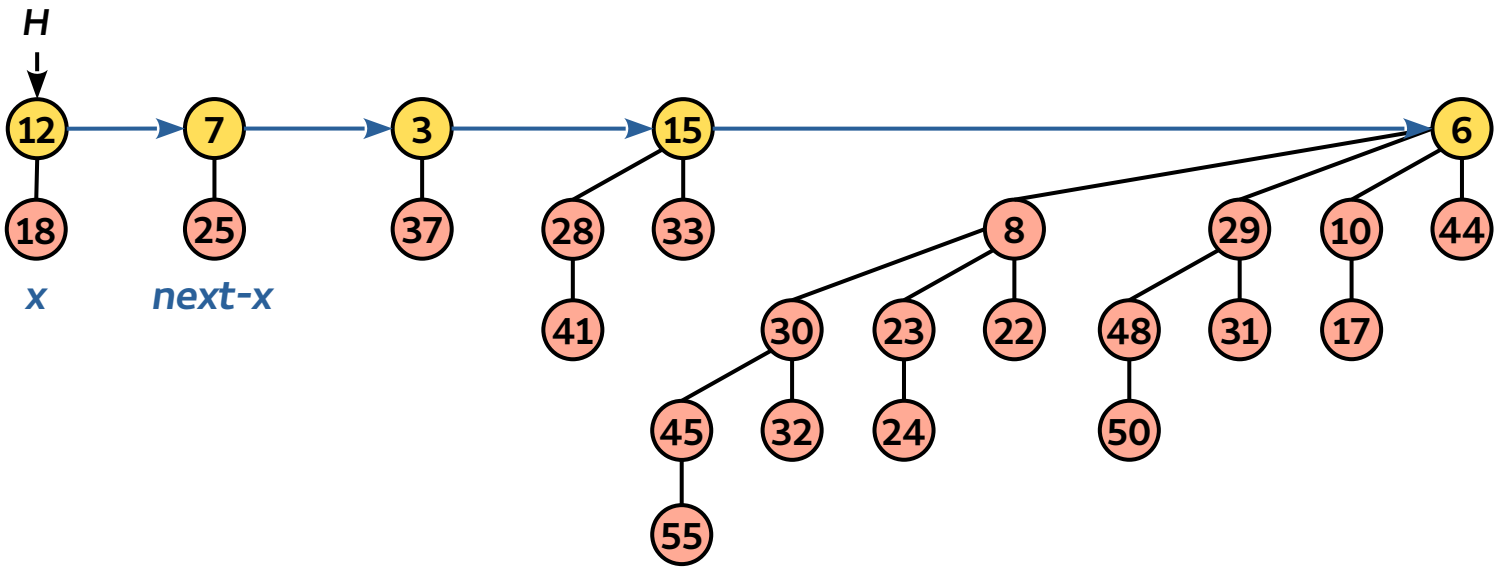
- Деревья *x* и *next-x* связываются
- Узел *next-x* становится левым дочерним узлом *x*



Слияние биномиальных куч: случай 3

x.degree = next-x.degree ≠ next-x.sibling.degree, *x.key ≤ next-x.key*

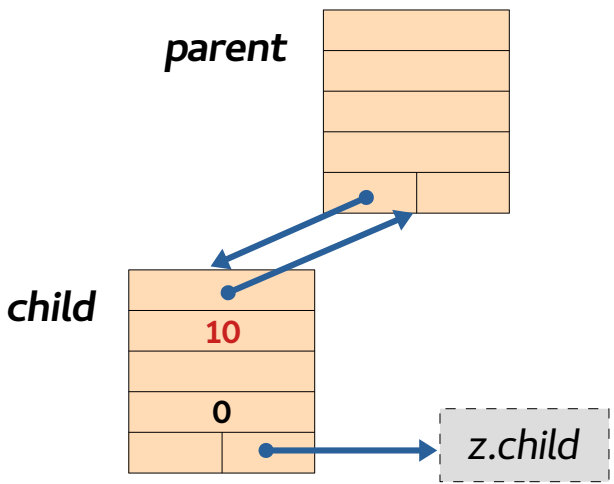
```
x.sibling = next-x.sibling
BinomialTreeLink(next-x, x)
next-x = x.sibling
```



Связывание биномиальных деревьев

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

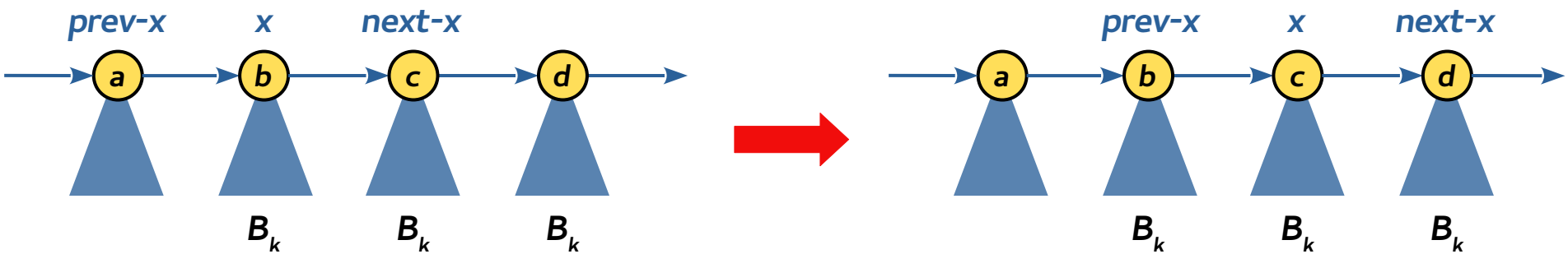
$T_{TreeLink} = O(1)$



Слияние биномиальных куч: случай 2

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

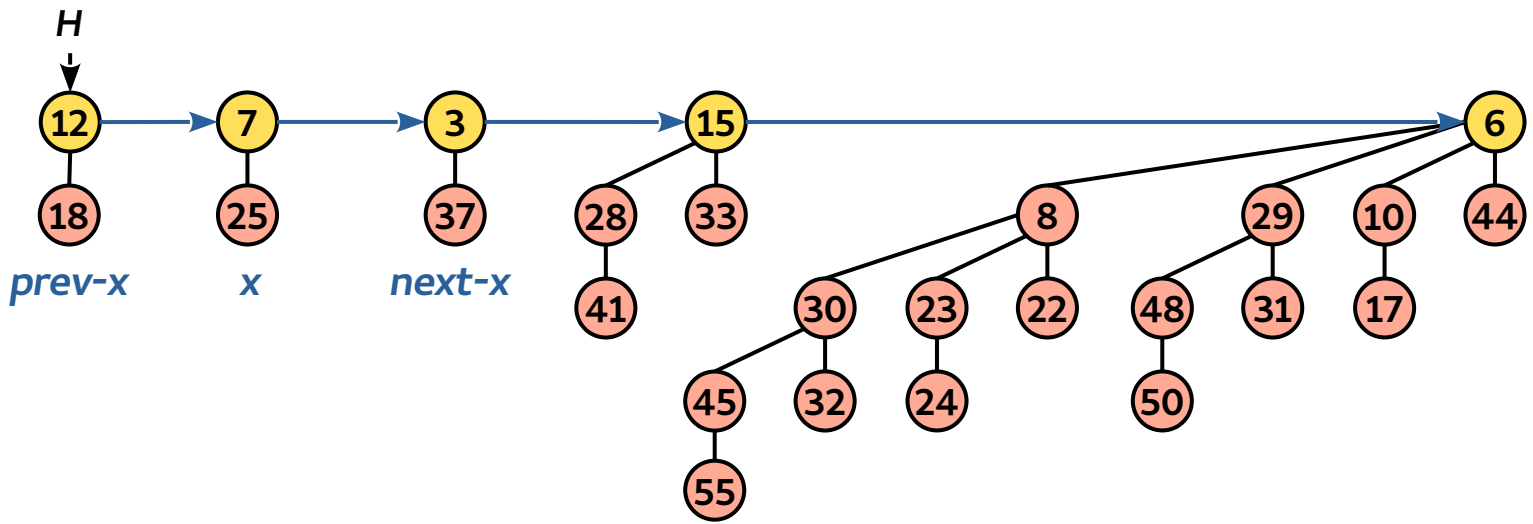
```
/* Перемещаем указатели по списку корней */
prev-x = x
x = next-x
next-x = x.sibling
```



Слияние биномиальных куч: случай 2

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

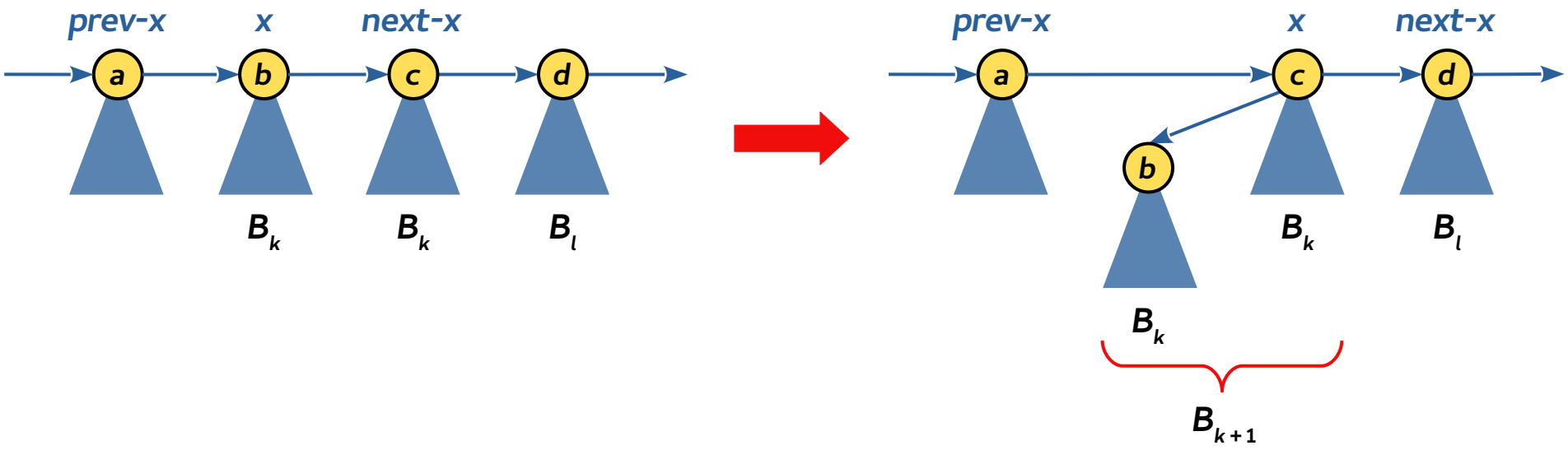
```
/* Перемещаем указатели по списку корней */
prev-x = x
x = next-x
next-x = x.sibling
```



Слияние биномиальных куч: случай 4

x.degree = next-x.degree ≠ next-x.sibling.degree, *x.key > next-x.key*

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

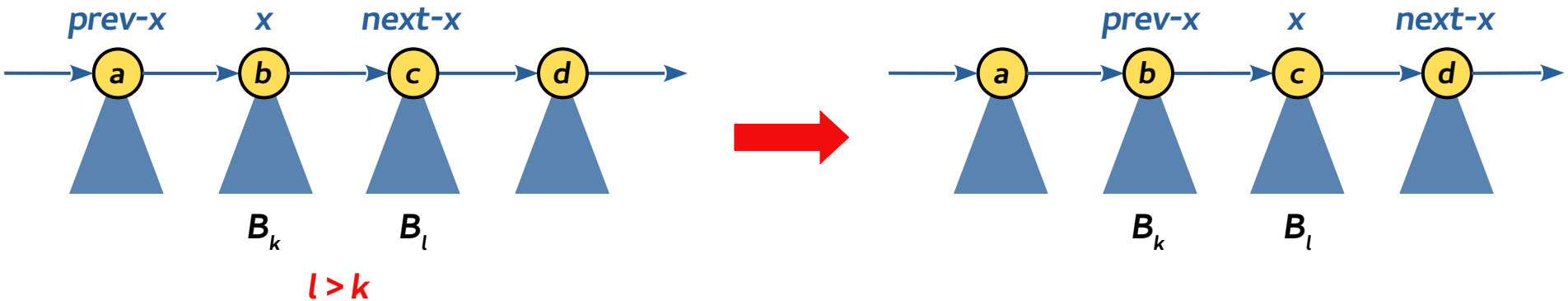


Слияние биномиальных куч: случай 1

x.degree ≠ *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)

- Создаём биномиальную кучу из одного узла *x* — биномиального дерева B_0
- Сливаем исходную кучу *H* и кучу из узла *x*

```
function BinomialTreeInsert(x, 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
```

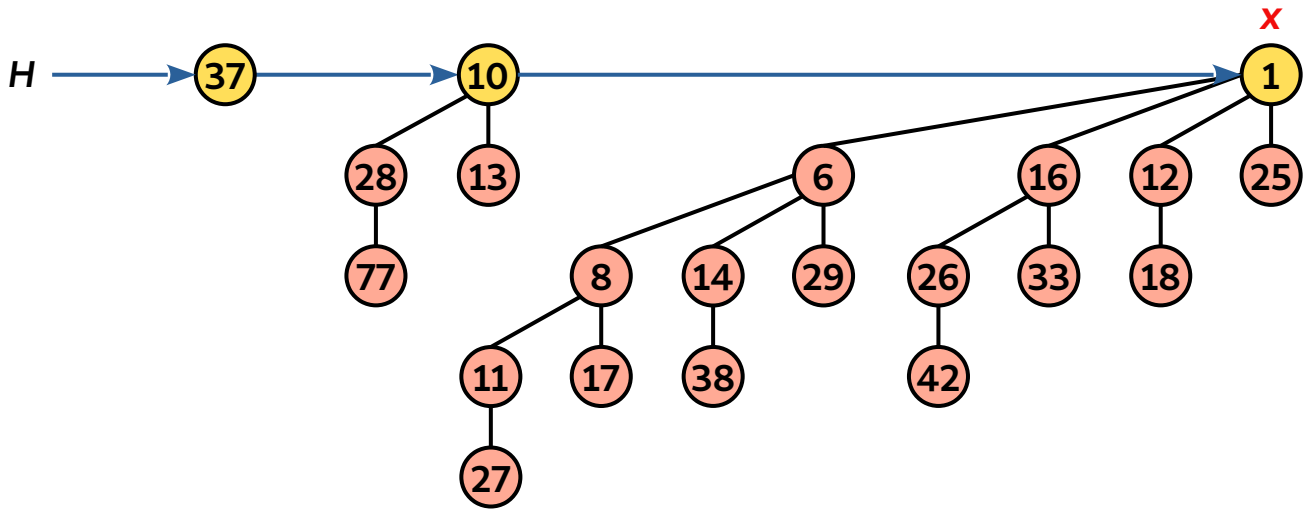
$T_{Insert} = O(\log n)$

Удаление минимального узла (DeleteMin)

1. В списке корней кучи *H* отыскиваем корень *x* с минимальным ключом и удаляем *x* из списка корней (разрываем 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*

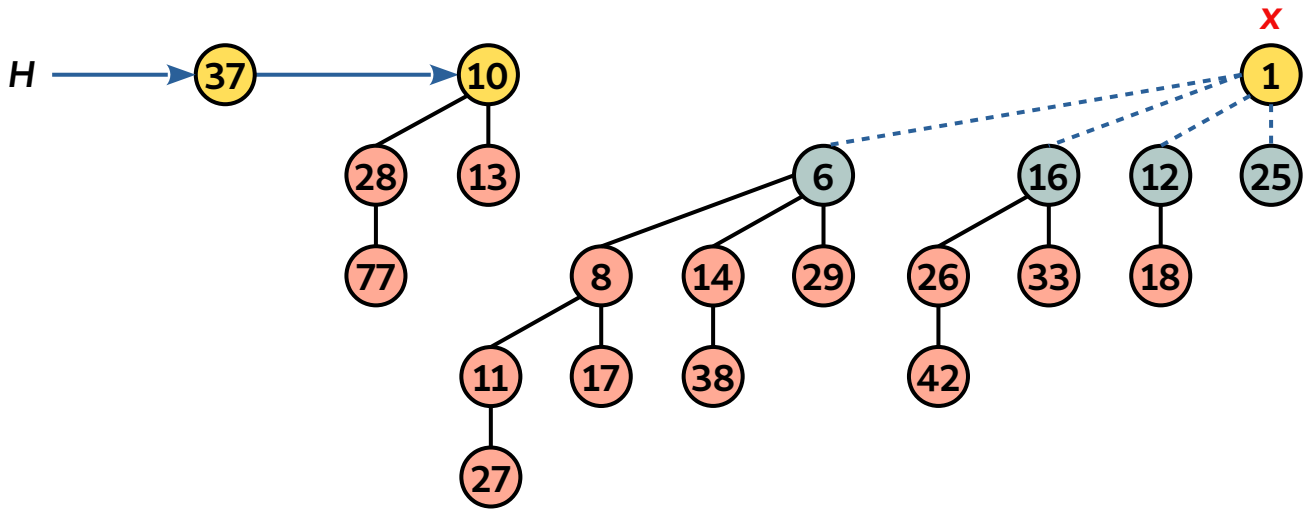
Удаление минимального узла (DeleteMin)

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



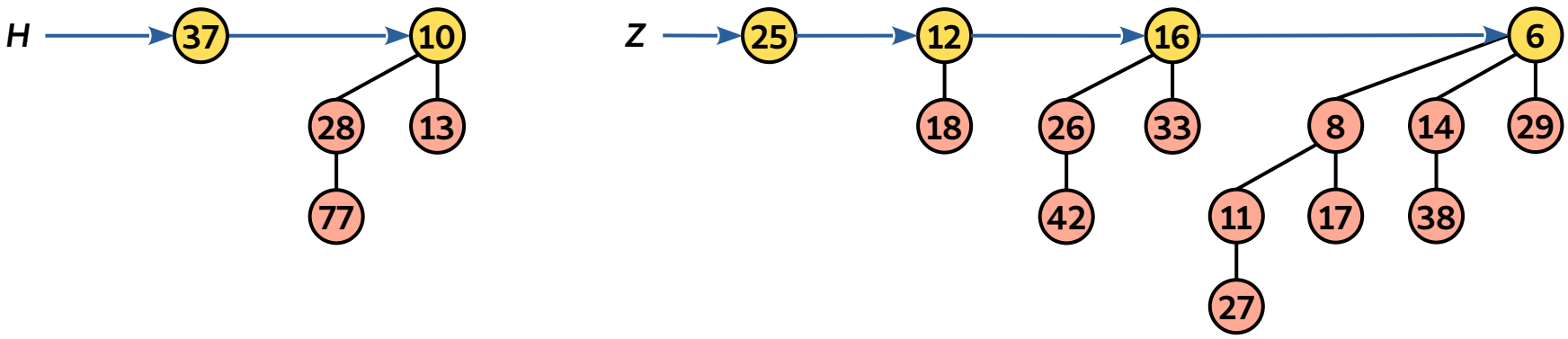
Удаление минимального узла (DeleteMin)

- 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)
    /* Поиск и отцепление минимального элемента */
    x = h;
    xminkey = Infinity
    prev = NULL
    while x != NULL do
        if x.key < xminkey 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
```

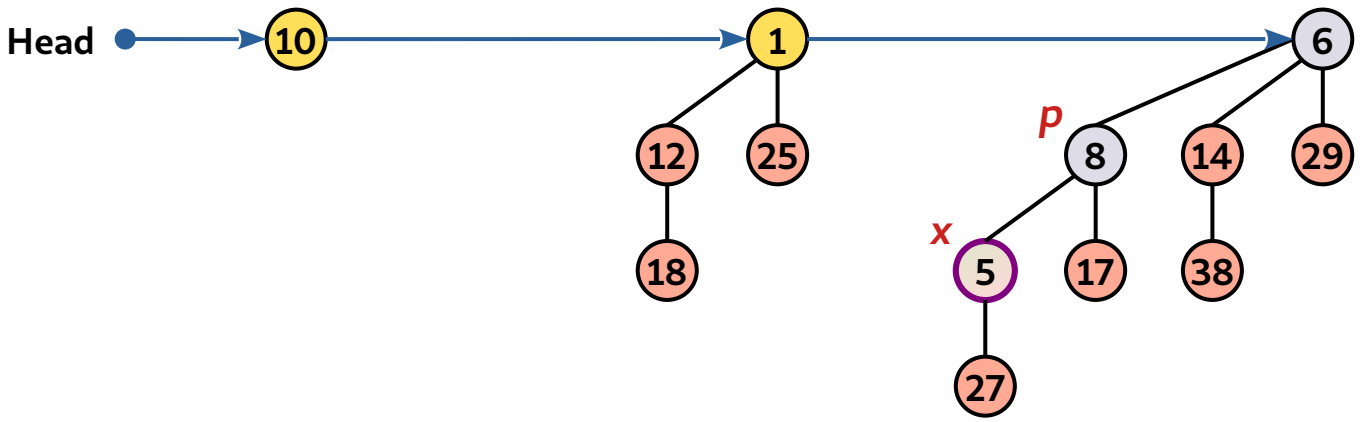
Удаление минимального узла (DeleteMin)

```
    /* Разворот связного списка */
    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)

- Получаем указатель на узел *x* и изменяем у него ключ (*newkey* ≤ *x.key*)
- Проверяем значение ключа родительского узла: если он меньше ключа *x*, 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 биномиальной кучи

```
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;
};
```

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))
        {
            /* Случаи 1 и 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;    /* Добавление в конец */
                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;    /* Добавление в конец */
                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 */

    /* Поиск и отцепление минимального элемента */
    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;

    /* Разворот связного списка */
    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);
}
```

Дальнейшее чтение

- Ознакомиться со следующими реализациями очередей с приоритетом:
  - Левосторонняя очередь (*leftist heap*)
  - Скошенная очередь (*skew heap*)
  - Очередь Бродала (*Brodal heap*)

ご清聴ありがとうございました!

Даниил Михайлович Берлизов
Старший преподаватель Кафедры вычислительных систем СибГУТИ
E-mail: sillyhat34@gmail.com

Курс «Структуры и алгоритмы обработки данных»
Осенний семестр, 2021 г.