

Лекция 2

Красно-черные деревья (red-black trees)

Курносов Михаил Георгиевич

E-mail: mkurnosov@gmail.com

WWW: www.mkurnosov.net

Курс «Структуры и алгоритмы обработки данных»

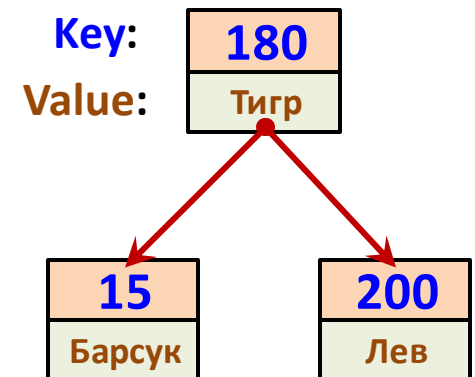
Сибирский государственный университет телекоммуникаций и информатики (Новосибирск)

Осенний семестр, 2015

Двоичные деревья поиска

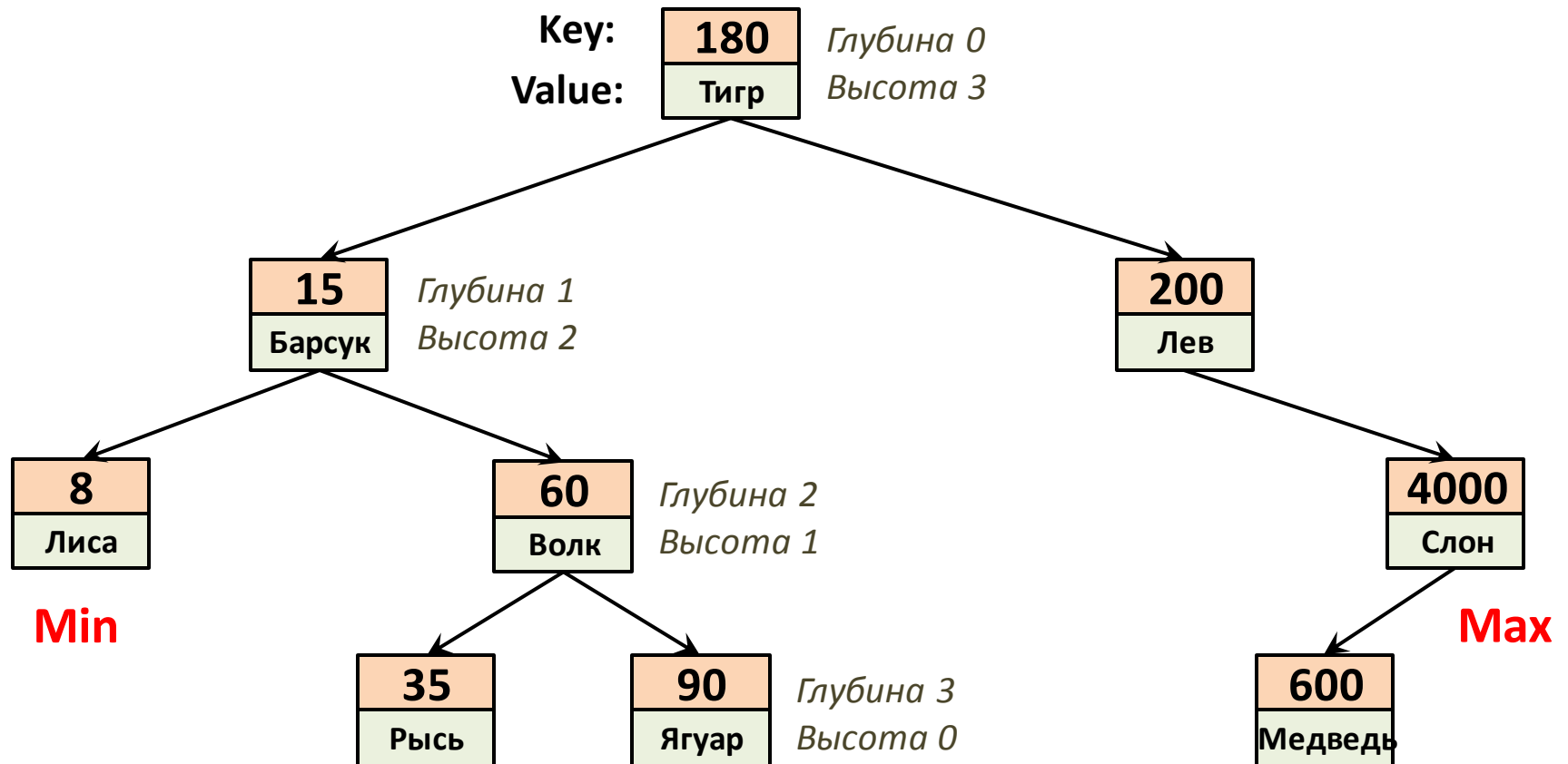
- **Двоичное дерево поиска (binary search tree, BST)** – это двоичное дерево, в котором:

- 1) каждый узел (node) имеет не более двух дочерних узлов (child nodes)
- 2) каждый узел содержит *ключ (key)* и *значение (value)*
- 3) ключи всех узлов левого поддерева меньше значения ключа родительского узла
- 4) ключи всех узлов правого поддерева больше значения ключа родительского узла



Двоичные деревья поиска используются для реализации словарей (map, associative array) и множеств (set)

Двоичные деревья поиска



9 узлов, высота (height) = 3, глубина (depth) = 3

Двоичные деревья поиска

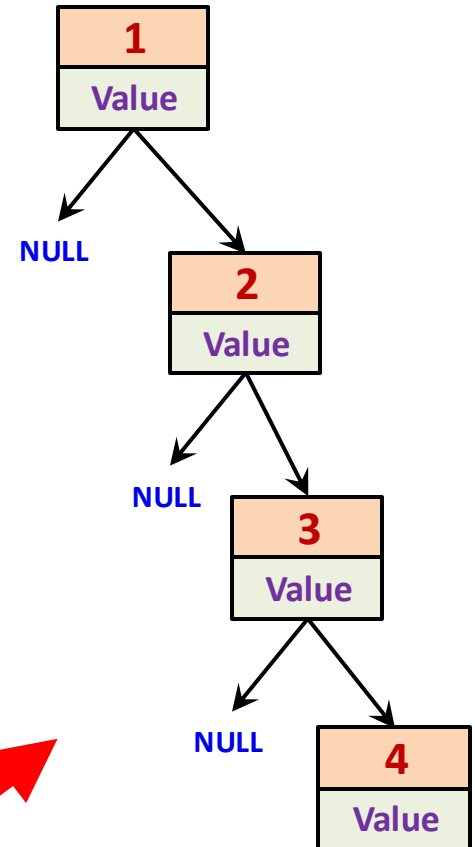
1. Операции над BST имеют трудоемкость пропорциональную высоте h дерева
2. В среднем случае высота дерева $O(\log n)$
3. В худшем случае элементы добавляются по возрастанию (убыванию) ключей – дерево вырождается в список длины $O(n)$

`bstree_add(1, value)`

`bstree_add(2, value)`

`bstree_add(3, value)`

`bstree_add(4, value)`

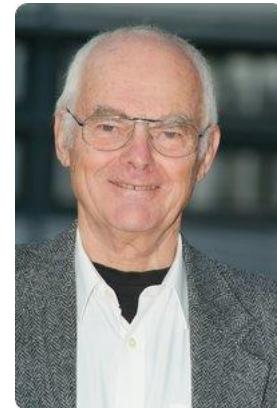


Сбалансированные деревья поиска

- **Сбалансированное дерево поиска (self-balancing binary search tree)** – дерево поиска, в котором высота поддеревьев любого узла различается не более чем на заданную константу k
- Сбалансированные деревья поиска:
 - ☐ **Красно-черные деревья (red-black trees)**
 - ☐ АВЛ-деревья (AVL trees)
 - ☐ 2-3-деревья (2-3 trees)
 - ☐ В-деревья (B-trees)
 - ☐ AA trees
 - ☐ ...

Красно-черные деревья (red-black trees)

- Автор **Rudolf Bayer**
Technical University of Munich, Germany, 1972
- В работе автора дерево названо
«symmetric binary B-tree»
- В работе Р. Седжвика (1978) дано современное
название «red-black tree»



- [1] Rudolf Bayer. **Symmetric binary B-Trees: Data structure and maintenance algorithms** // Acta Informatica. – 1972. – Vol. 1, No. 4 – pp. 290-306.
- [2] Guibas L., Sedgwick R. **A Dichromatic Framework for Balanced Trees** // Proc. of the 19th Annual Symposium on Foundations of Computer Science, 1978. – pp. 8-21.

Красно-черные деревья (red-black trees)

Операция	Средний случай (average case)	Худший случай (worst case)
Add (<i>key, value</i>)	$O(\log n)$	$O(\log n)$
Lookup (<i>key</i>)	$O(\log n)$	$O(\log n)$
Remove (<i>key</i>)	$O(\log n)$	$O(\log n)$
Min	$O(\log n)$	$O(\log n)$
Max	$O(\log n)$	$O(\log n)$

Сложность по памяти (space complexity): $O(n)$

Применение красно-черных деревьев

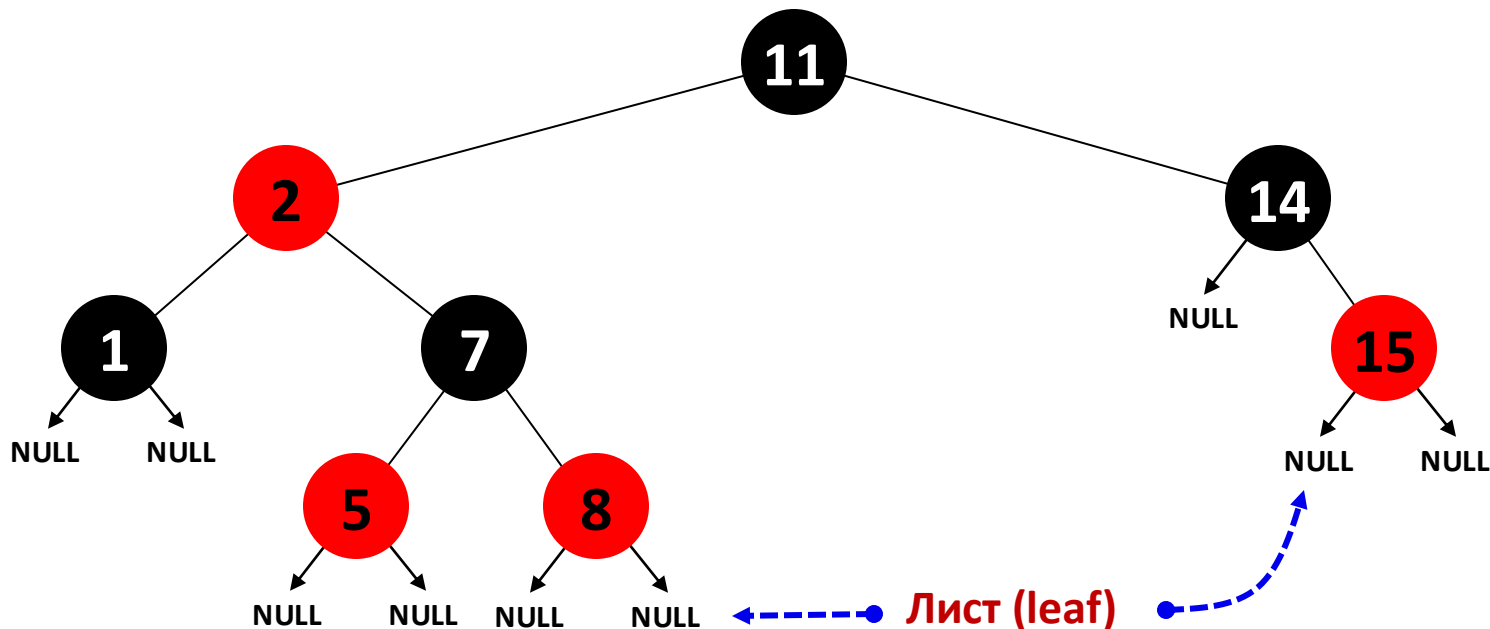
- **GNU libstdc++** (/usr/include/c++/bits)
std::map, std::multimap, std::set, std::multiset
- **LLVM libc++**
std::map, std::set
- **Java**
java.util.TreeMap, java.util.TreeSet
- **Microsoft .NET 4.5 Framework Class Library**
SortedDictionary, SortedSet
- **Ядро Linux**
<https://www.kernel.org/doc/Documentation/rbtree.txt>
<https://github.com/torvalds/linux/blob/master/tools/lib/rbtree.c>
 - ☐ CFS scheduler: процессы хранятся в rbtree
 - ☐ Virtual memory areas (VMAs)
 - ☐ High-resolution timer (organize outstanding timer requests)
 - ☐ Ext3 filesystem tracks directory entries
 - ☐ epoll file descriptors, cryptographic keys, ...

Красно-черные деревья (red-black trees)

- **Красно-черное дерево** (red-black tree, RB-tree) – это бинарное дерево поиска, для которого выполняются *красно-черные свойства* (red-black properties):
 - 1) Каждый узел дерева является либо **красным (red)**, либо **черным (black)**
 - 2) Корень дерева является черным узлом
 - 3) Каждый лист дерева (NULL) является черным узлом
 - 4) У красного узла оба дочерних узла – черные
 - 5) У любого узла все пути от него до листьев, являющихся его потомками, содержат одинаковое количество черных узлов

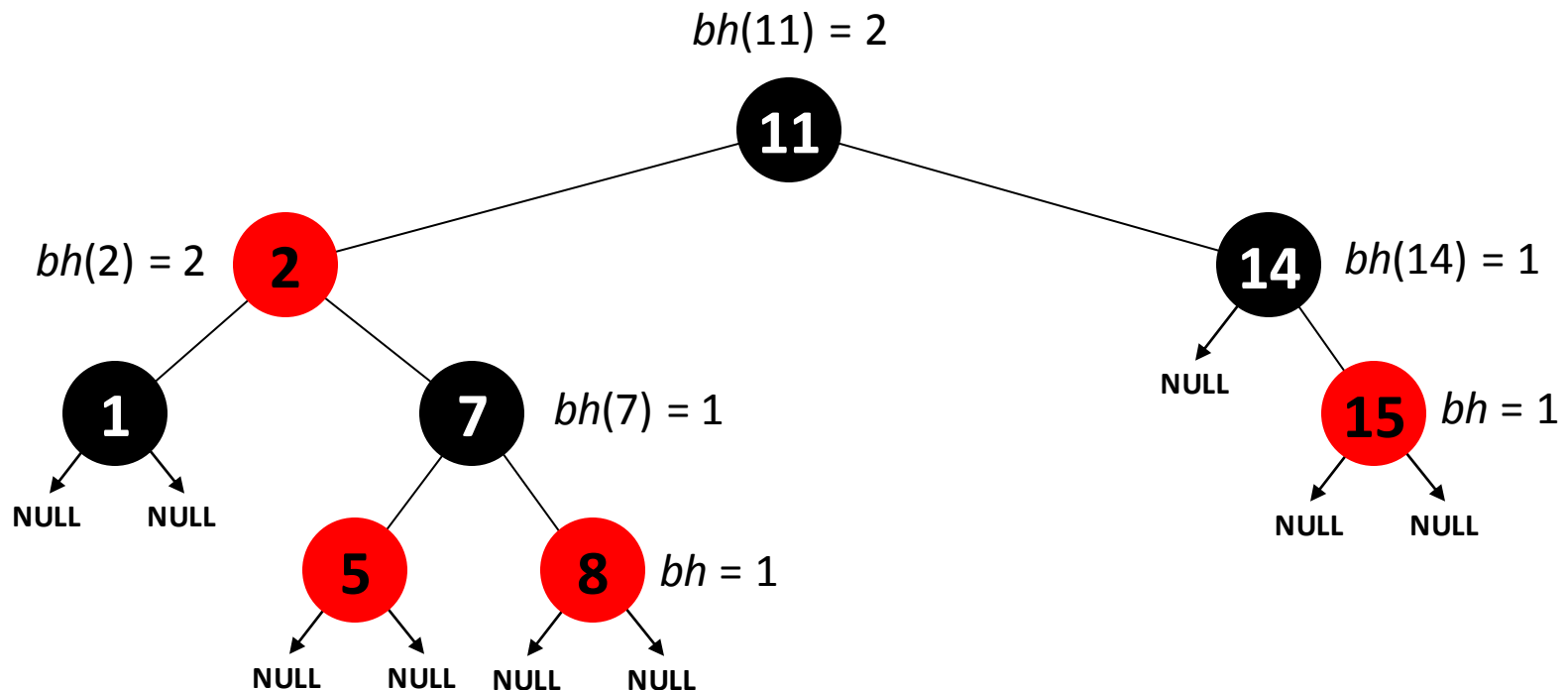
Пример красно-черного дерева

- 1) Каждый узел является красным, либо черным
- 2) Корень дерева является черным
- 3) Каждый лист дерева (NULL) является черным
- 4) У красного узла оба дочерних узла – черные
- 5) У любого узла все пути от него до листьев, являющихся его потомками, содержат одинаковое число черных узлов



Черная высота узла (black height)

- Черная высота $bh(x)$ узла (black height) – это количество черных узлов на пути от узла x (не считая его) до листа
- Черная высота дерева – это черная высота его корня



Высота красно-черного дерева

Лемма. *Красно-черное дерево с n внутренними узлами имеет высоту, не превышающую $2\log_2(n + 1)$*

Доказательство [CLRS, С. 342]

- Покажем по индукции, что любое поддереву с вершиной в узле x содержит не менее $2^{bh(x)} - 1$ внутренних узлов
- **Базис индукции**
Если высота h узла x равна 0, то узел x – это лист (NULL), а его поддереву содержит не менее $2^{bh(x)} - 1 = 2^0 - 1 = 0$ внутренних узлов

Высота красно-черного дерева

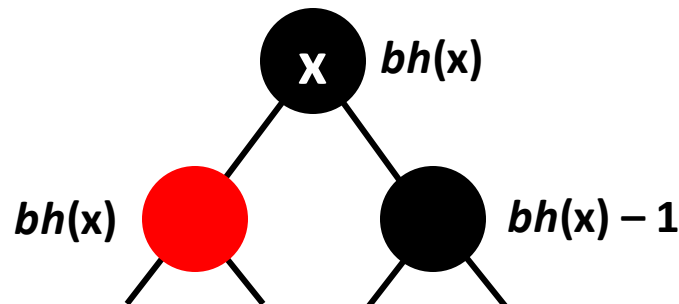
Лемма. Красно-черное дерево с n внутренними узлами имеет высоту, не превышающую $2\log_2(n + 1)$

Доказательство (продолжение)

- Шаг индукции

Рассмотрим узел x , который имеет положительную высоту $h(x)$ – внутренний узел с двумя потомками

- Каждый дочерний узел имеет черную высоту либо $bh(x)$, либо $bh(x) - 1$, в зависимости от его цвета



Высота красно-черного дерева

Лемма. *Красно-черное дерево с n внутренними узлами имеет высоту, не превышающую $2\log_2(n + 1)$*

Доказательство (продолжение)

- Поскольку высота потомка x меньше высоты узла x , мы можем использовать предположение индукции и сделать вывод о том, что каждый потомок x имеет как минимум $2^{bh(x) - 1} - 1$ внутренних узлов
- Тогда все дерево с корнем в узле x содержит не менее

$$(2^{bh(x) - 1} - 1) + (2^{bh(x) - 1} - 1) + 1 = 2^{bh(x)} - 1$$

внутренних узлов.

Высота красно-черного дерева

Лемма. *Красно-черное дерево с n внутренними узлами имеет высоту, не превышающую $2\log_2(n + 1)$*

Доказательство (продолжение)

- Получили, что в дереве x число n внутренних узлов

$$n \geq 2^{bh(x)} - 1$$

- По свойству 4 – как минимум половина узлов на пути от корня к листу черные, тогда

$$bh(x) \geq h(x) / 2$$

- Следовательно

$$n \geq 2^{h(x) / 2} - 1$$

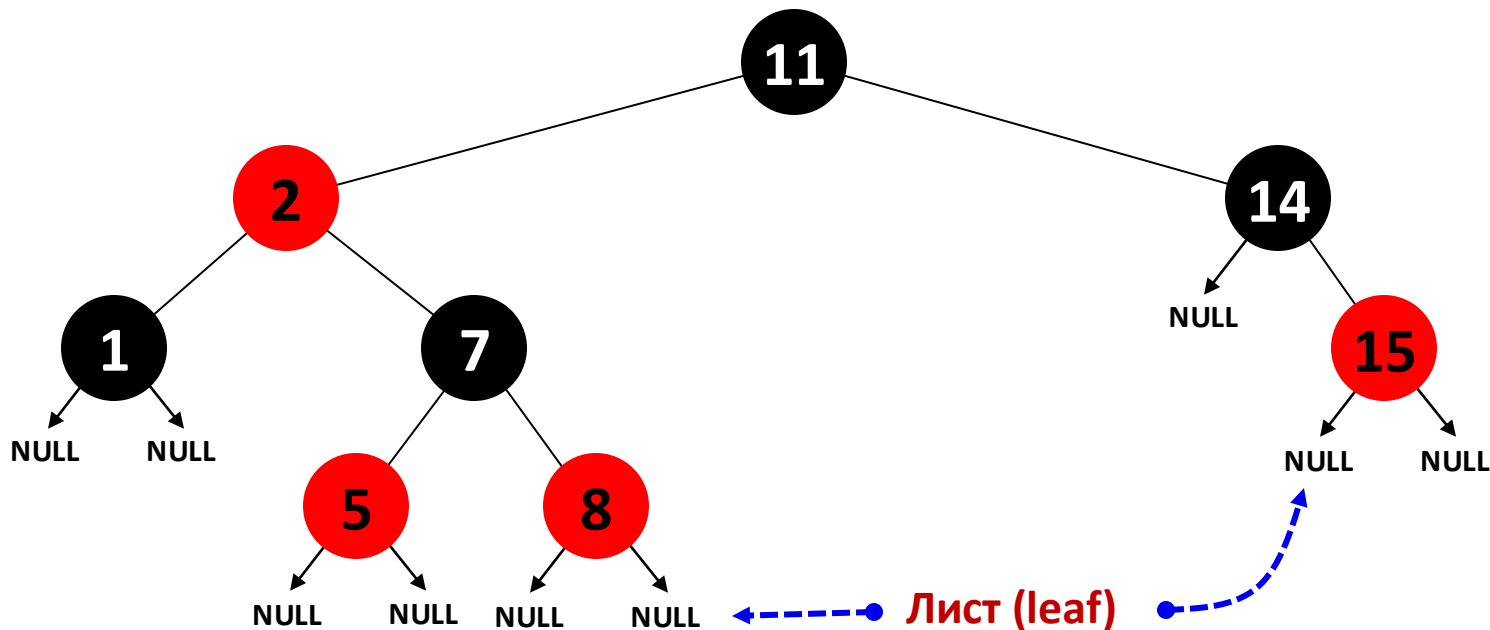
- Логарифмируем

$$\log_2(n + 1) \geq h(x) / 2$$

$$h(x) \leq 2\log_2(n + 1)$$

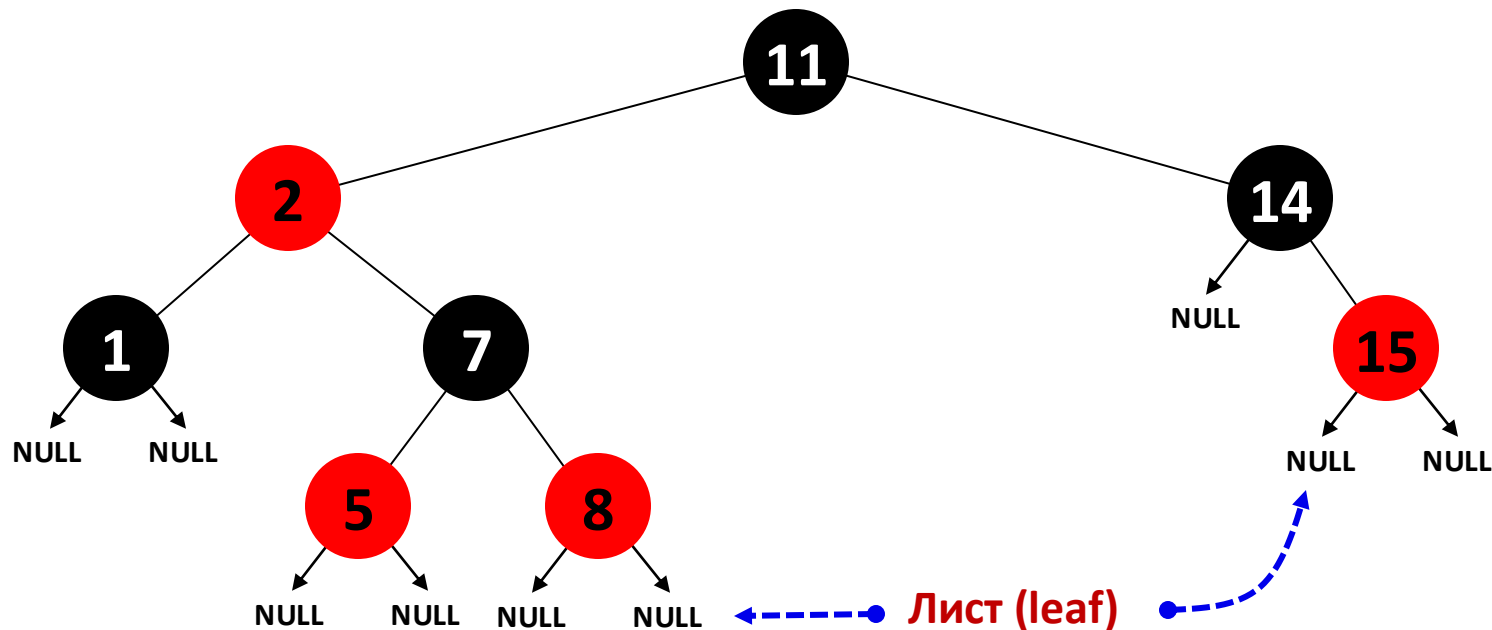
Структура узла дерева

- **node.parent** – указатель на родительский узел
- **node.left** – указатель на левый дочерний узел
- **node.right** – указатель на правый дочерний узел
- **node.color** – цвет узла (RED, BLACK)
- **T.root** – указатель на корень дерева



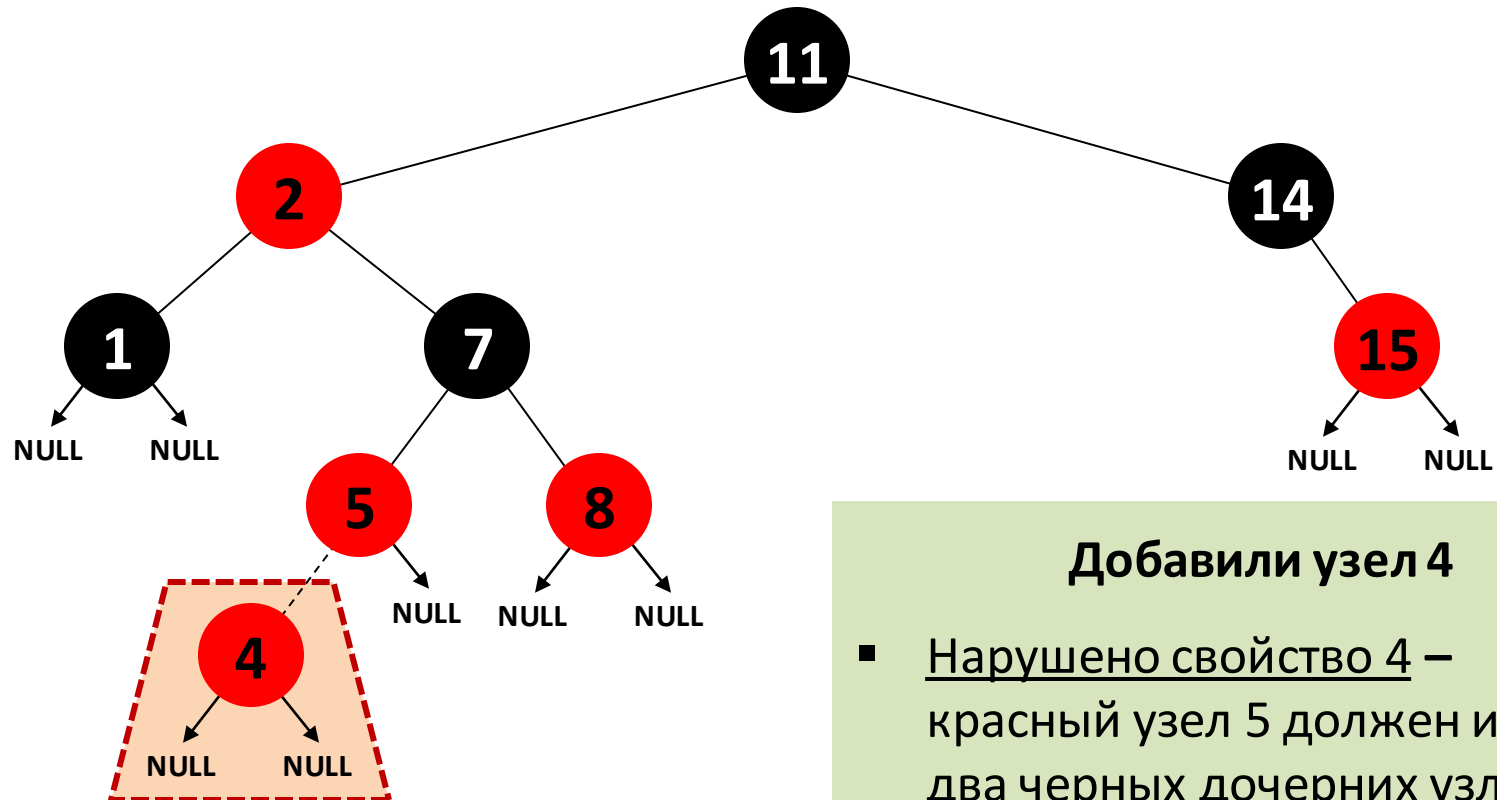
Структура узла дерева

Для удобства будем считать, что все листья (узлы NULL) – это указатели на один и тот же ограничивающий узел черного цвета *T.null*



Добавление элемента (add, insert)

1. Находим лист для вставки нового элемента
2. Создаем элемент и окрашиваем его в **красный** цвет
3. Перекрашиваем узлы и выполняем повороты



Добавление элемента (add, insert)

```
function RBTree_Add(T, key, value)
  tree = T.root
  while tree != null do
    if key < tree.key then
      tree = tree.left
    else if key > tree.key then
      tree = tree.right
    else
      return  /* Ключ уже присутствует в дереве */
    end if
  end while
```

Добавление элемента (add, insert)

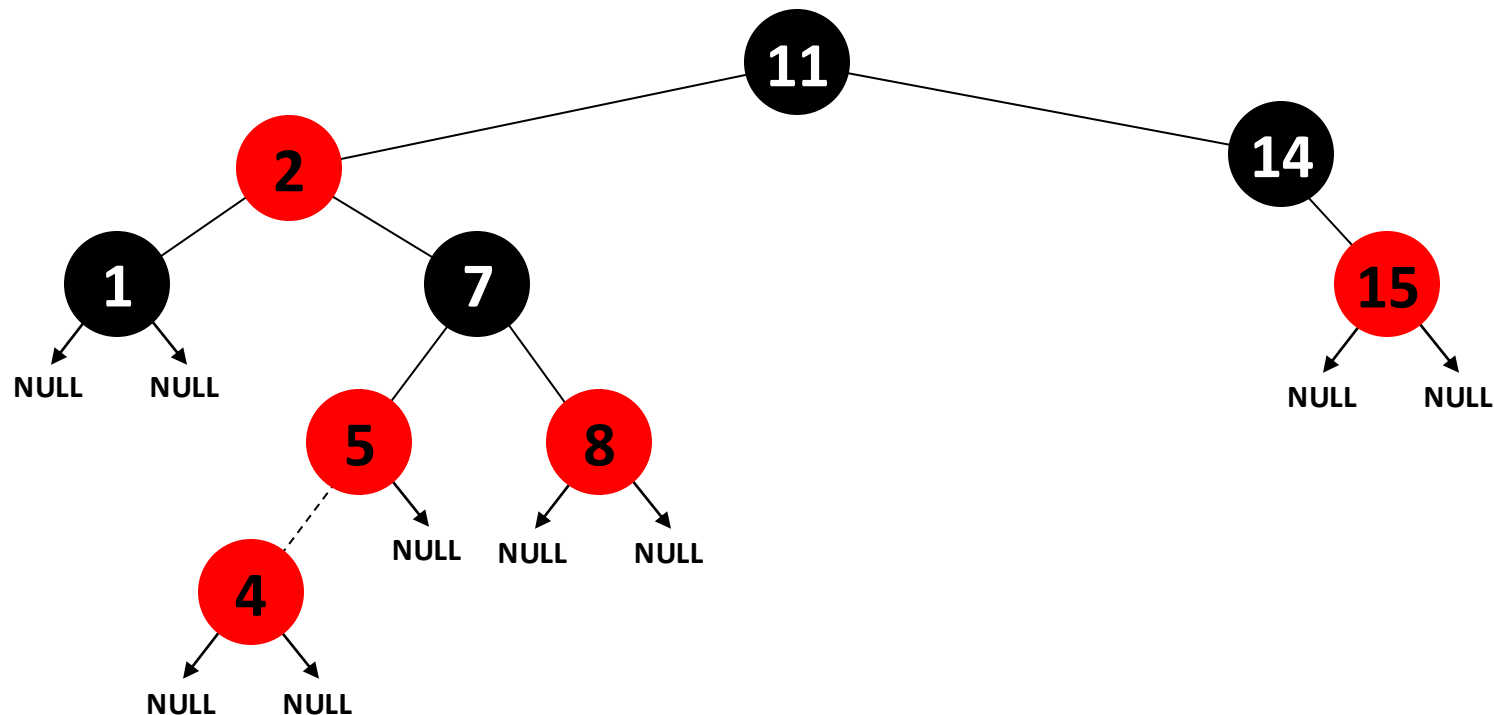
```
node = CreateNode(key, value)
if T.root = NULL then           /* Пустое дерево */
    T.root = node
else
    if key < tree.parent.key then
        tree.parent.left = node
    else
        tree.parent.right = node
    end if
end if

node.color = RED
RBTree_Fixup(T, node)
end function
```

Нарушение свойств красно-черного дерева

Какие свойства красно-черного дерева могут быть нарушены после вставки нового узла (красного цвета)?

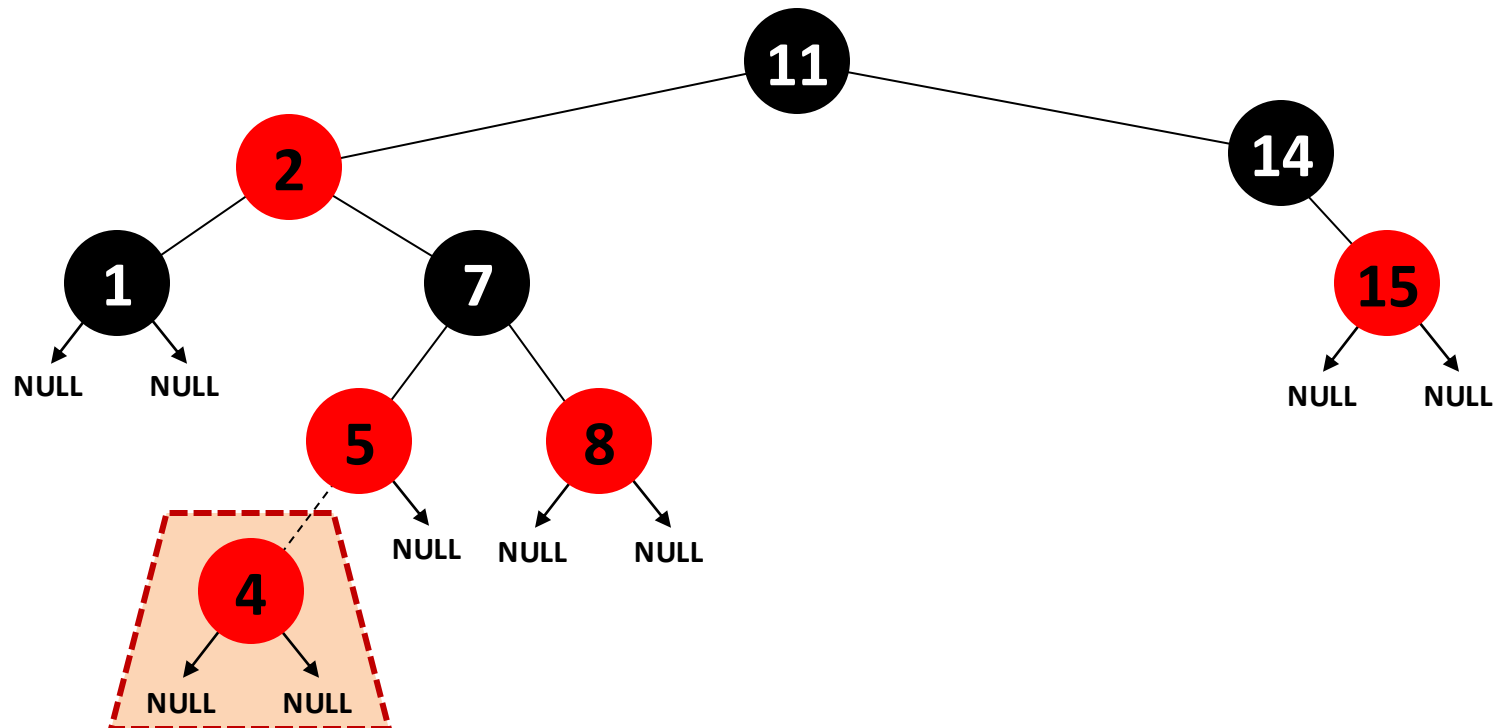
- 1) каждый узел является красным, либо черным – **выполняется**
- 2) корень дерева является черным – **может быть нарушено** (например, при добавление первого элемента)



Нарушение свойств красно-черного дерева

Какие свойства красно-черного дерева могут быть нарушены после вставки нового узла (красного цвета)?

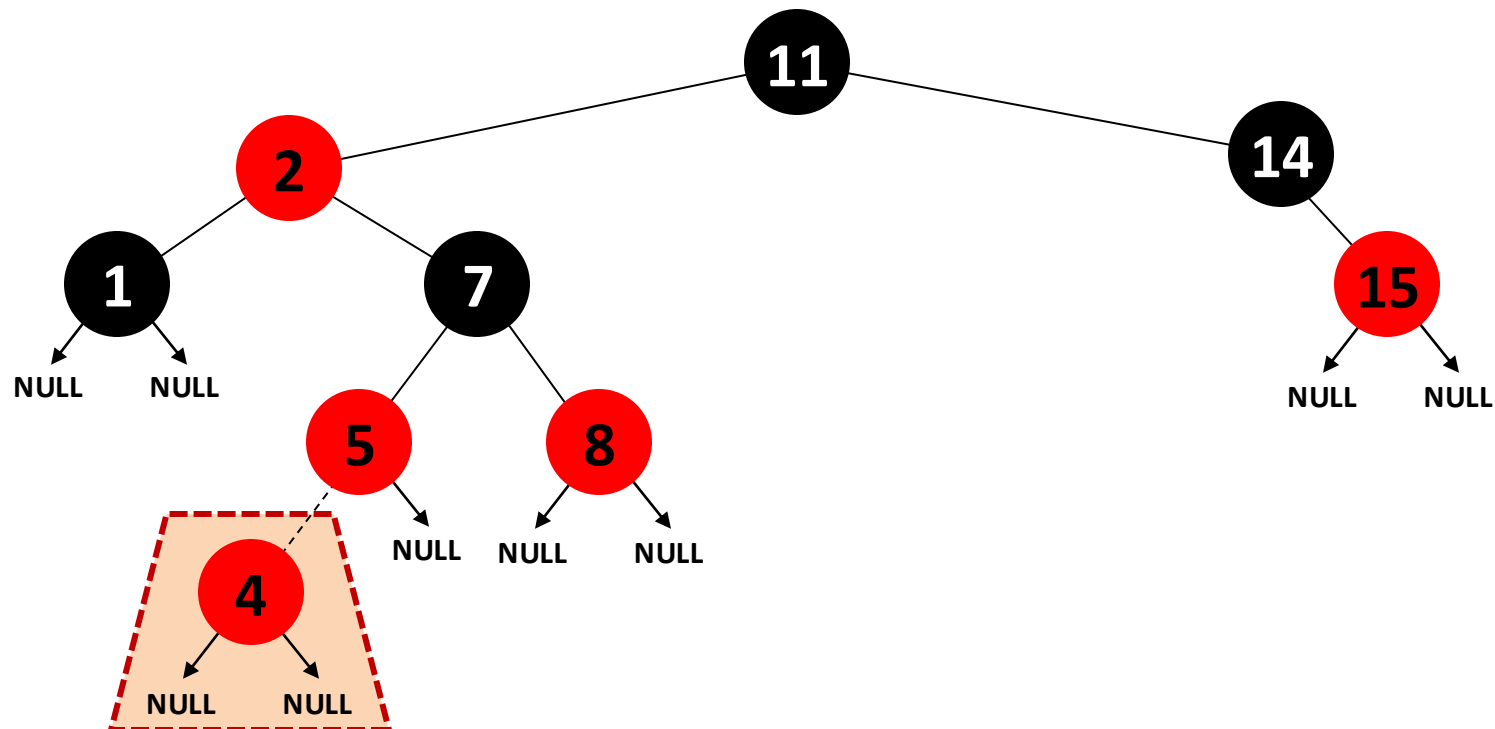
- 3) каждый лист дерева (NULL) является черным – **выполняется**
- 4) у красного узла оба дочерних узла являются черными – **может быть нарушено**



Нарушение свойств красно-черного дерева

Какие свойства красно-черного дерева могут быть нарушены после вставки нового узла (красного цвета)?

- 5) у любого узла все пути от него до листьев (его потомков), содержат одинаковое число черных узлов – **выполняется** (новый узел замещает черный NULL, но сам имеет два черных дочерних NULL)



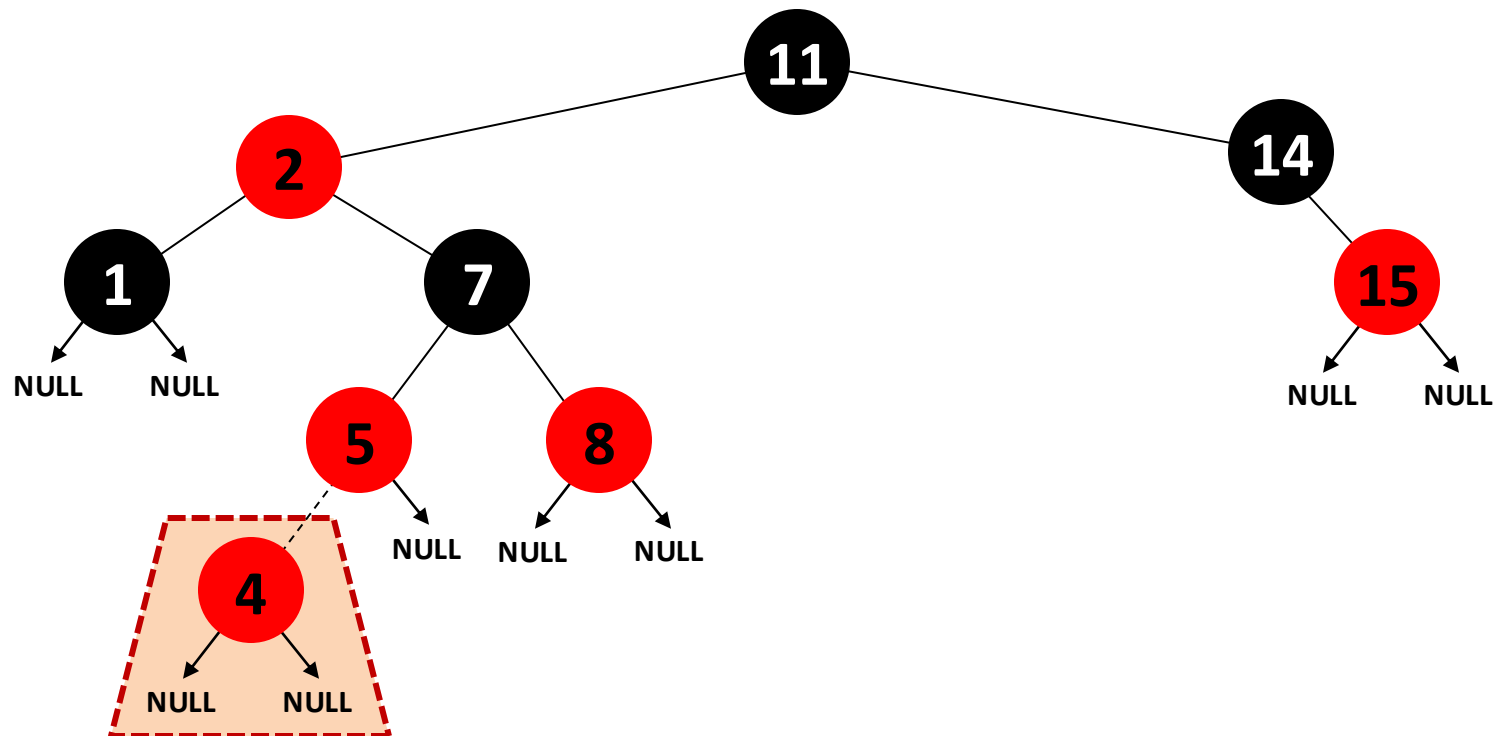
Нарушение свойств красно-черного дерева

После добавления нового элемента свойства 2 и 4 могут быть нарушены

- 1) Каждый узел является красным, либо черным – **выполняется**
- 2) Корень дерева является черным – **может быть нарушено** (например, при добавление первого элемента)
- 3) Каждый лист дерева (NULL) является черным – **выполняется**
- 4) У красного узла оба дочерних узла являются черными – **может быть нарушено**
- 5) У любого узла все пути от него до листьев, являющихся его потомками, содержат одинаковое число черных узлов – **выполняется** (новый узел замещает черный NULL, но сам имеет два черных дочерних NULL)

Восстановление красно-черного дерева

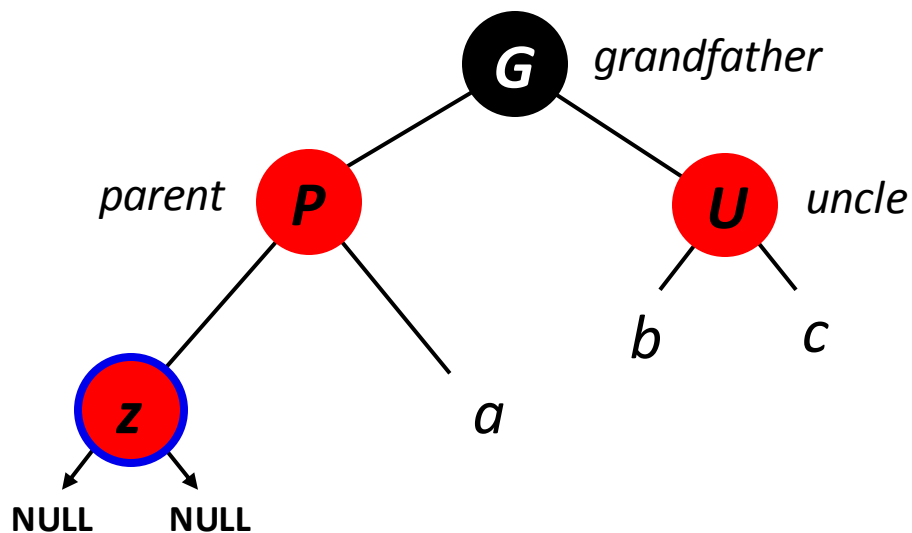
- Возможно 6 случаев, нарушающих свойства красно-черного дерева (3 случая симметричны другим трем)
- Восстановление свойств начинаем с нового элемента и продвигаемся вверх к корню дерева



Восстановление красно-черного дерева

Случай 1: «дядя» U (uncle) узла z – красный
(добавили узел z)

- Узел z красный
- Дядя U узла z – красный
- Узел P – это корень левого поддерева своего родителя G
- Родительский узел P узла z красный



У узла P
нарушено
свойство 4

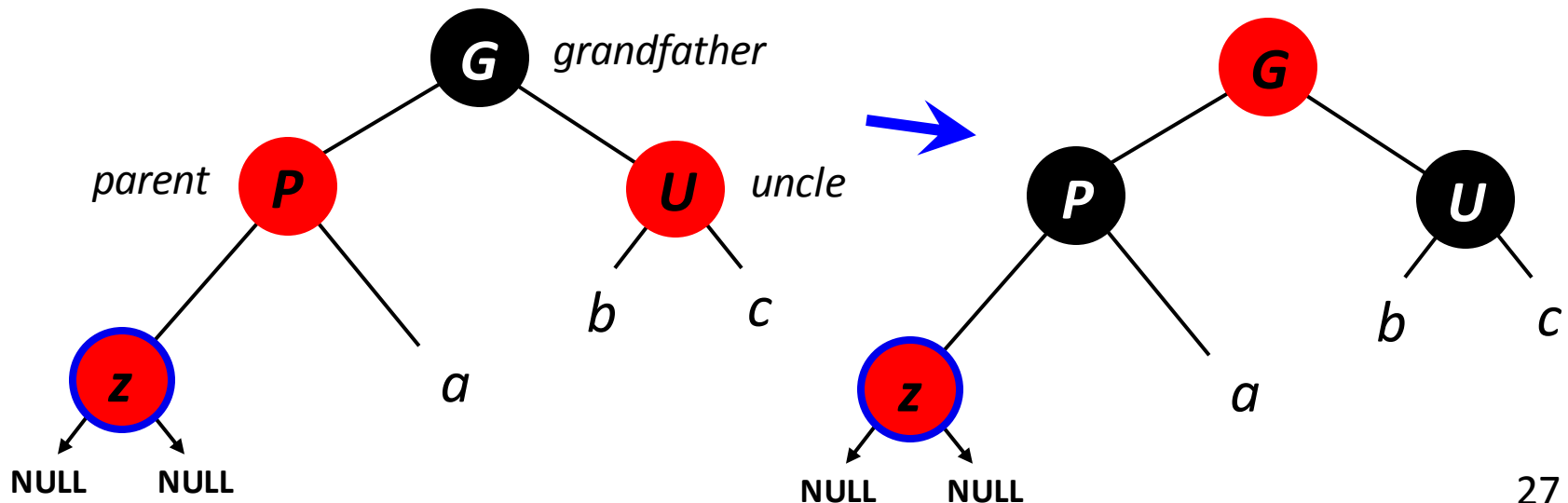
Восстановление красно-черного дерева

Случай 1: «дядя» U (uncle) узла z – красный
(добавили узел z)

- Узел z красный
- Дядя U узла z – красный
- Узел P – это корень левого
- Родительский узел P узла z

Перекрашиваем узлы

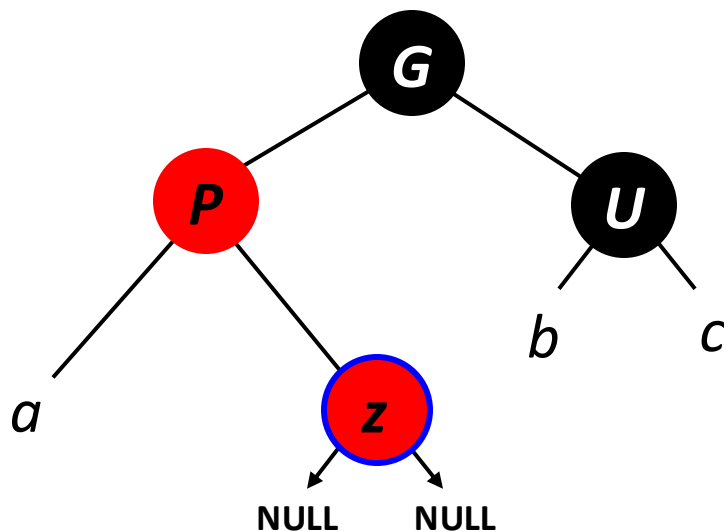
- P – черный ($z.p$)
- U – черный ($z.p.p.right$)
- G – **красный** ($z.p.p$)



Восстановление красно-черного дерева

Случай 2

- Дядя U узла z – черный
- Узел z – правый потомок P
- Узел z красный
- Узел P – корень левого поддерева своего родителя G
- Родительский узел P узла z красный



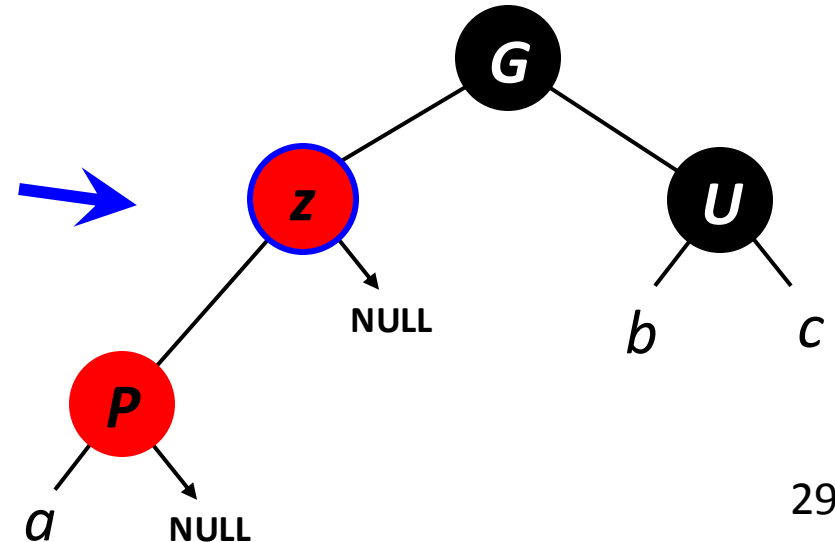
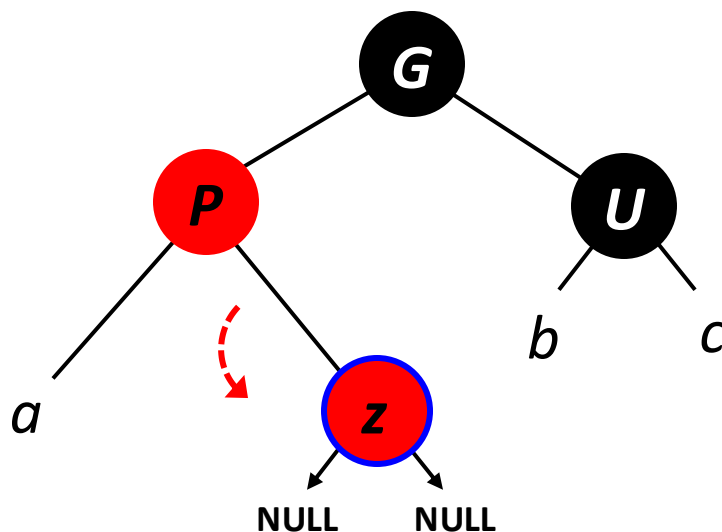
P нарушает
свойство 4

Восстановление красно-черного дерева

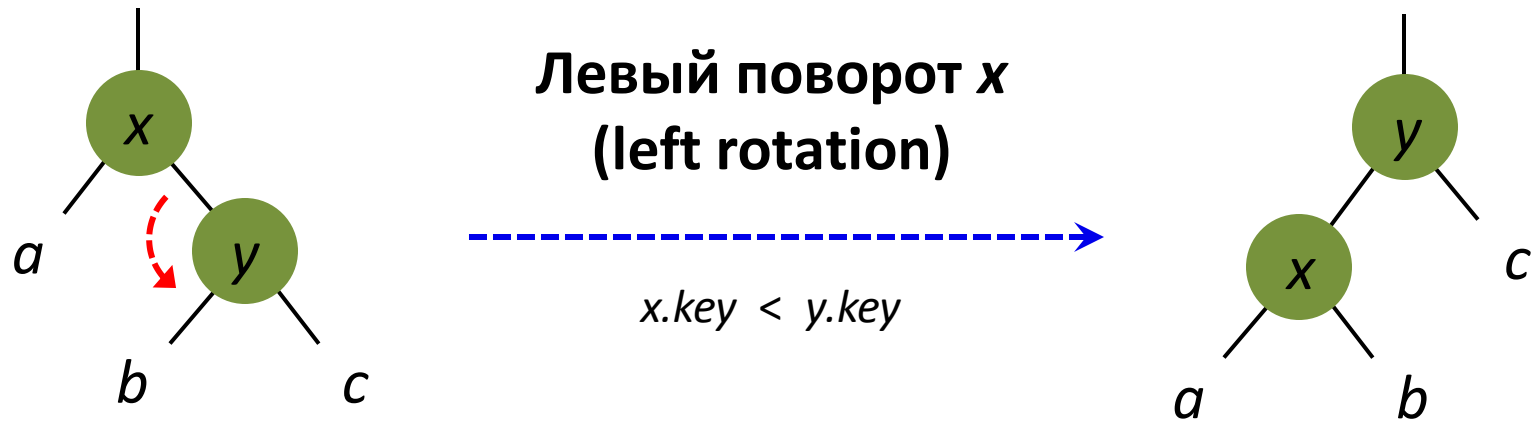
Случай 2

- Дядя U узла z – черный
- Узел z – правый потомок P
- Узел z красный
- Узел P – корень левого поддеревья
- Родительский узел P узла z крас

Переходим к случаю 3
путем поворота
дерева P влево



Левый поворот дерева (left rotation)



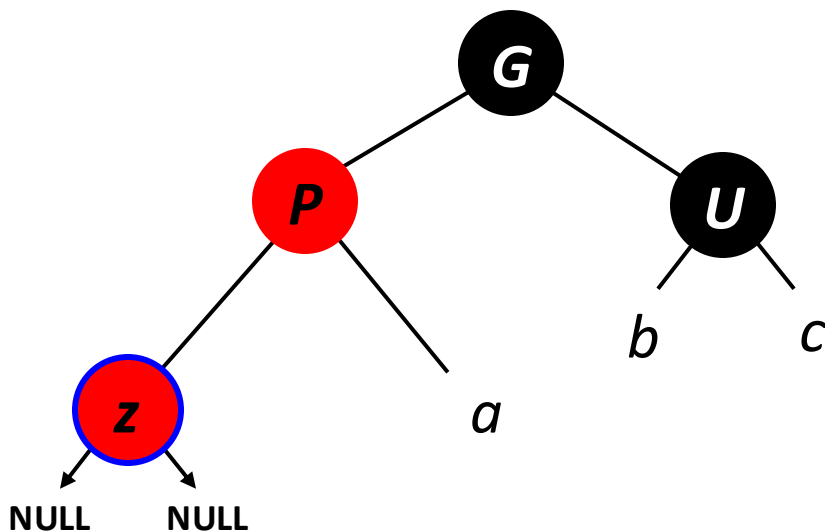
```
function LeftRotate(x)
    y = x.right
    x.right = y.left           /* Subtree b */
    if y.left != NULL then
        y.left.parent = x    /* Setup parent of b */
    y.parent = x.parent
    if x = x.parent.left then /* x is left subtree */
        x.parent.left = y
    else
        x.parent.right = y
    y.left = x
    x.parent = y
end function
```

$$T_{\text{LeftRotate}} = O(1)$$

Восстановление красно-черного дерева

Случай 3

- Дядя U узла z – черный
- Узел z – левый потомок P
- Узел z красный
- Узел P – это корень левого поддерева своего родителя G
- Родительский узел P узла z красный



P нарушает
свойство 4

Восстановление красно-черного дерева

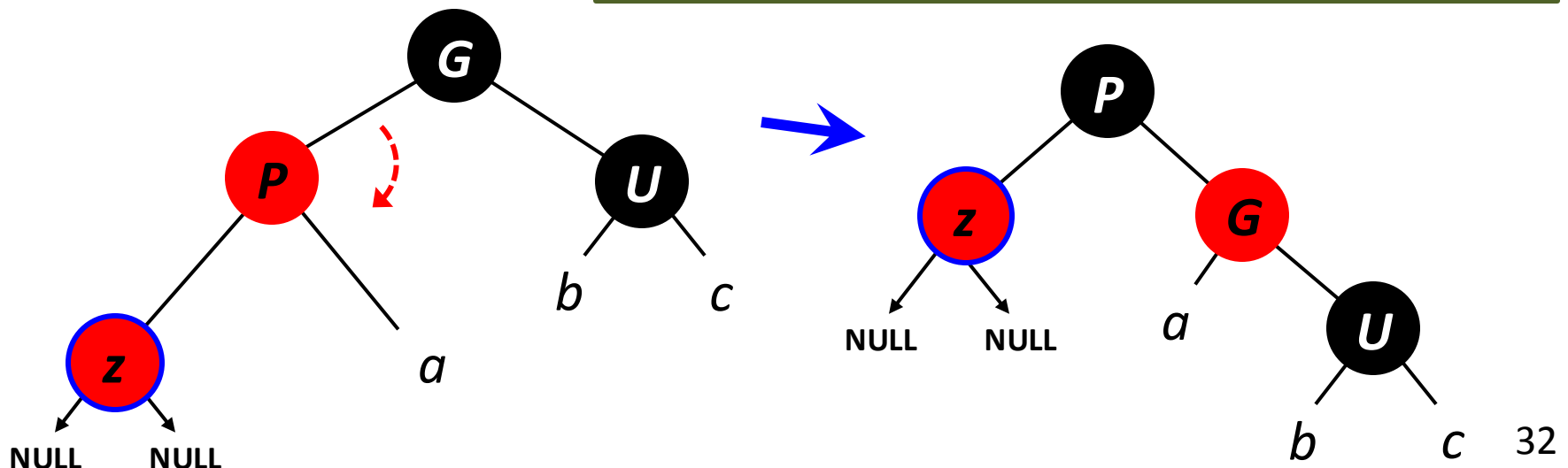
Случай 3

- Дядя U узла z – черный
- Узел z – левый потомок P
- Узел z красный
- Узел P – это корень
- Родительский узел U

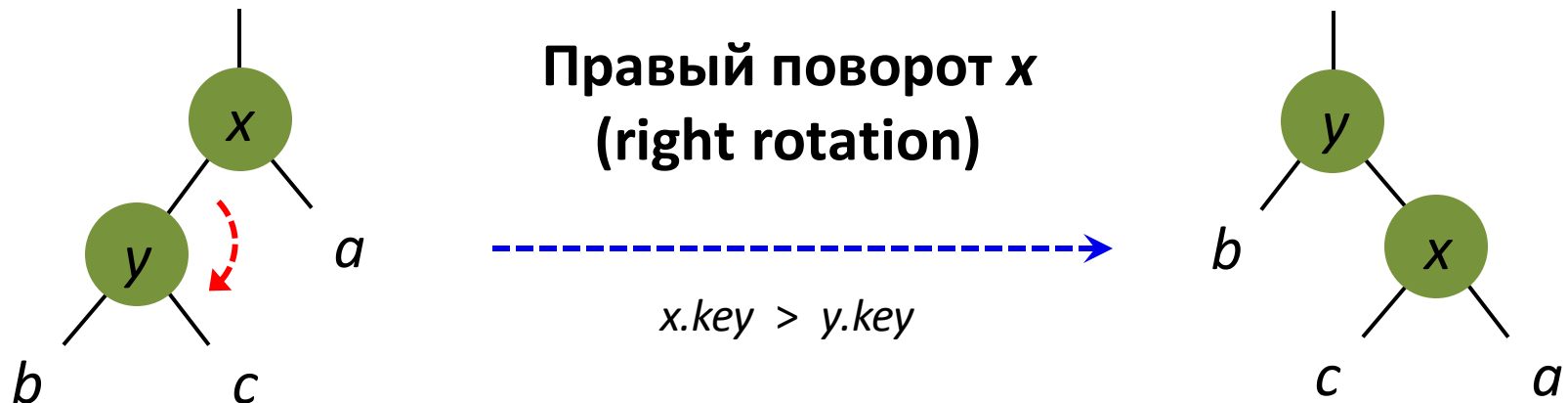
1. Перекрашиваем вершины

- P – черный
- G – **красный**

2. Поворачиваем дерево G вправо



Правый поворот дерева (right rotation)



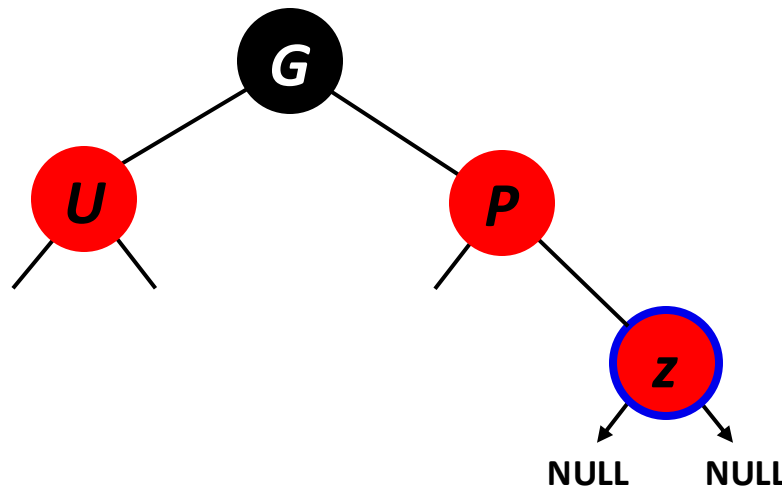
```
function RightRotate(x)
  y = x.left
  x.left = y.right          /* Subtree c */
  if y.right != NULL then
    y.right.parent = x     /* Setup parent of c */
  y.parent = x.parent
  if x = x.parent.left then /* x is left subtree */
    x.parent.left = y
  else
    x.parent.right = y
  y.right = x
  x.parent = y
end function
```

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

Восстановление красно-черного дерева

Случаи 4, 5 и 6 симметричны случаям 1, 2 и 3

- Узел P – это корень **правого** поддерева своего родителя G
- Узел z красный
- Родительский узел P узла z красный
- Узел U *черный* или *красный*
- Узел z – *левый* или *правый* дочерний элемент P



Восстановление красно-черного дерева

```
function RBTree_Add_Fixup(T, z)
  while z.parent.color = RED do
    if z.parent = z.parent.parent.left then
      /* z in left subtree of G */
      y = z.parent.parent.right;    /* Uncle */
      if y.color = RED then
        /* Case 1 */
        z.parent.color = BLACK
        y.color = BLACK
        z.parent.parent.color = RED
        z = z.parent.parent
      else
        if z = z.parent.right then
          /* Case 2 --> Case 3 */
          z = z.parent
          RBTree_RotateLeft(T, z)
        end if
      end if
    end if
  end while
end function
```

Восстановление красно-черного дерева

```
        /* Case 3 */
        z.parent.color = BLACK
        z.parent.parent.color = RED
        RBTREE_RotateRight(T, z.parent.parent)
    end if
else
    /* z in right subtree of G */
    /* ... */
end if
end while
T.root.color = BLACK
end function
```

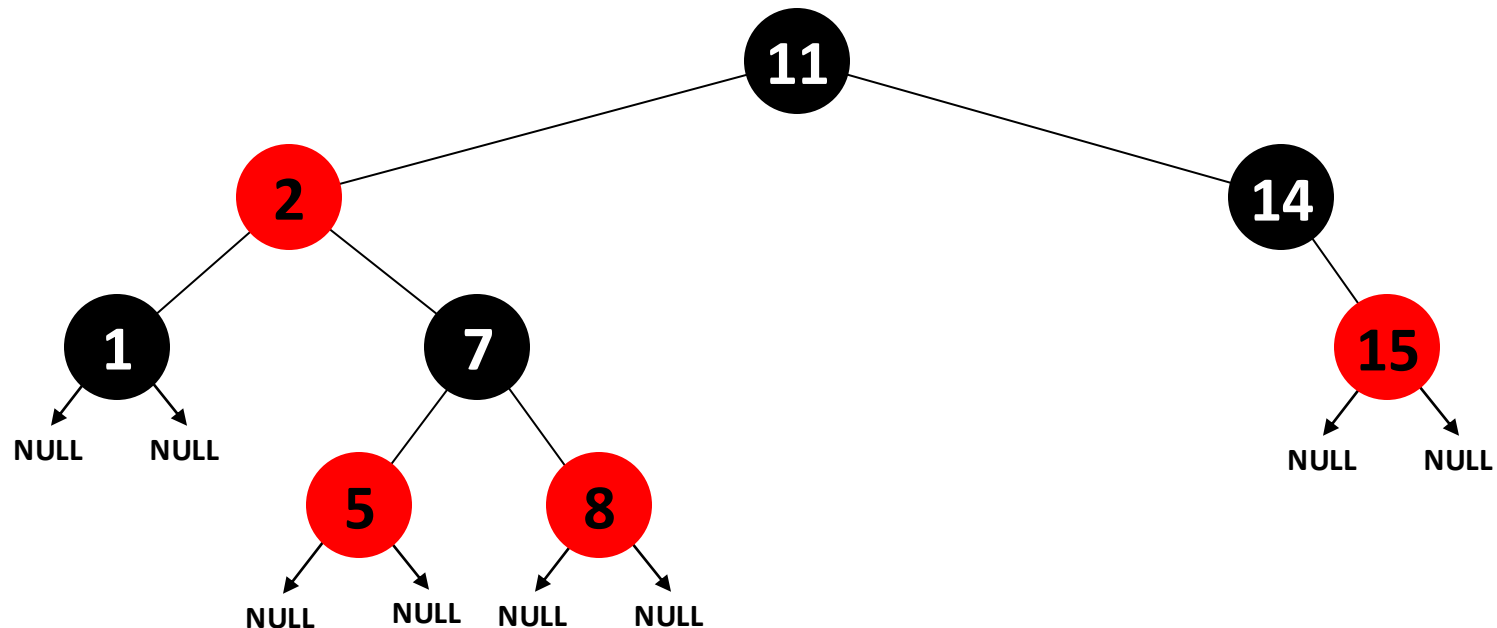
$$T_{Fixup} = O(\log n)$$

- Цикл **while** повторно выполняется только в случае 1, указатель *z* перемещается вверх по дереву на 2 уровня (количество итераций цикла в худшем случае – $O(\log n)$)
- Никогда не выполняется больше двух поворотов (в случаях 2 и 3 цикл **while** завершает работу)

Удаление элемента

[CLRS 3ed, C. 356]

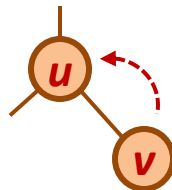
1. По заданному ключу находим элемент для удаления
2. Удаляем элемент (как в случае обычного дерева поиска)
3. Перекрашивая узлы и выполняя повороты восстанавливаем структуру красно-черного дерева



Перемещение узлов (transplant)

```
function RBTree_Transplant(T, u, v)
  if u.parent = NULL then
    T.root = v
  else if u = u.parent.left then
    /* Узел u в левом поддереве родителя */
    u.parent.left = v
  else
    /* Узел u в правом поддереве родителя */
    u.parent.right = v
  end if
  v.parent = u.parent
end function
```

Заменяет узел u вершиной v



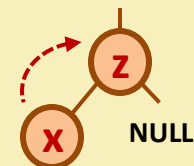
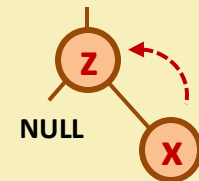
Удаление элемента

```
function RBTREE_Delete(T, key)
    z = RBTREE_Lookup(T, key)
    y = z
    y_color = y.color

    if z.left = NULL then
        /* Нет левого поддерева */
        x = z.right
        RBTREE_Transplant(T, z, z.right)

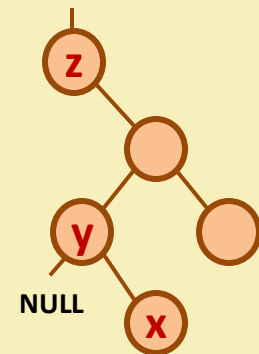
    else if z.right = NULL then
        /* Нет правого поддерева */
        x = z.left
        RBTREE_Transplant(T, z, z.left)

    else
```



Удаление элемента

```
/* Узел z имеет оба поддерева */  
y = RBTMin(z.right)  
y_color = y.color  
x = y.right  
if y.parent = z then  
    x.parent = y  
else  
    RBTTransplant(T, y, y.right)  
    y.right = z.right  
    y.right.parent = y  
end if  
RBTTransplant(T, z, y)  
y.left = z.left  
y.left.parent = y  
y.color = z.color  
end if /* имеет два поддерева */
```



Удаление элемента

```
    if y_color = BLACK then  
        RBTREE_Delete_Fixup(T, x)  
    end if  
end function
```

- Если удаляется красный узел, то свойства красно-черного дерева сохраняются

Удаление и восстановление свойств

- **Свойство 2:** удалили черный корень дерева, его заменил красный потомок
- **Свойство 4:** если x и $x.parent$ красные
- **Свойство 5:** перемещение u в дереве приводит к тому, что путь содержащий u , теперь имеет на один черный узел меньше

Удаление элемента

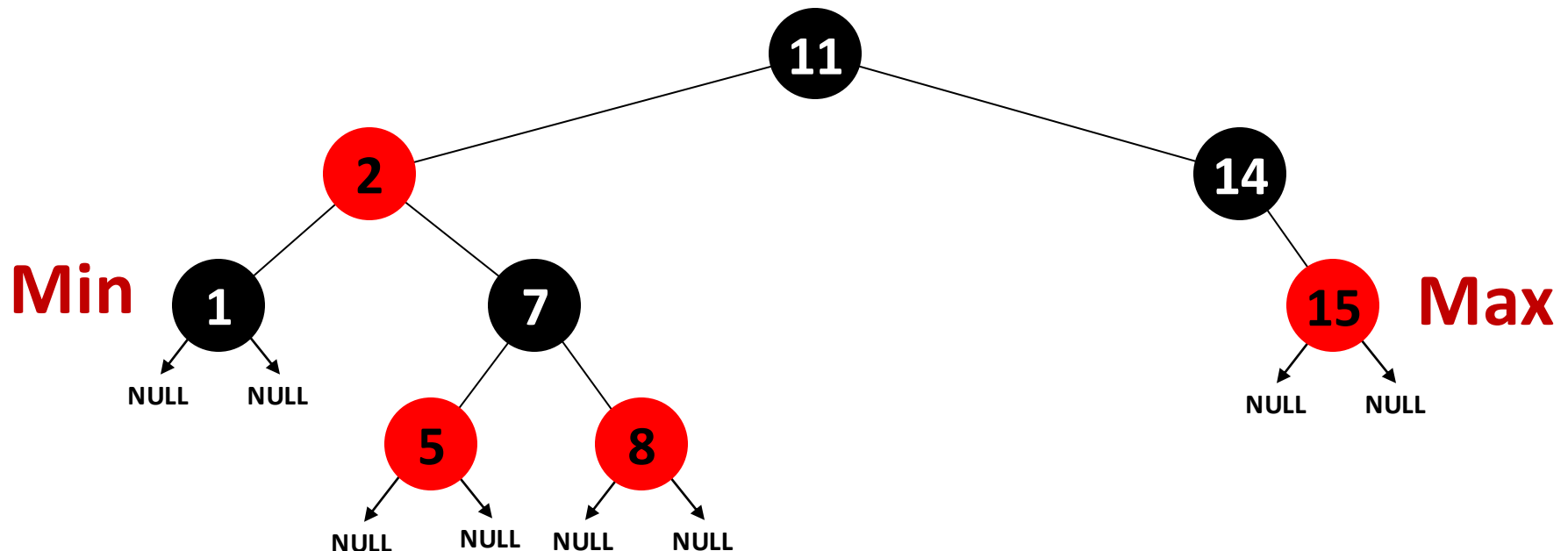
```
function RBTree_Delete_Fixup(T, x)
  while x != T.root and x.color = BLACK do
    if x = x.parent.left then
      w = x.parent.right
      if w.color = RED then           /* Case 1 */
        w.color = BLACK
        x.parent.color = RED
        RBTree_RotateLeft(T, x.parent)
        w = x.parent.right
      end if
      if w.left.color = BLACK and    /* Case 2 */
        w.right.color = BLACK then
        w.color = RED
        x = x.parent
      else
```

Удаление элемента

```
        if w.right.color = BLACK then
            w.left.color = BLACK      /* Case 3 */
            w.color = RED
            RBTREE_RotateRight(T, w)
            w = x.parent.right
        end if
        w.color = x.parent.color      /* Case 4 */
        x.parent.color = BLACK
        w.right.color = BLACK
        RBTREE_RotateLeft(T, x.parent)
        x = T.root
    end if
else
    /* Case 5-8 */
end if
end while
x.color = BLACK
end function
```

Поиск элемента, min, max

- Поиск элемента по ключу (lookup), поиск min/max элемента выполняются точно также как и в бинарных деревьях поиска (см. лекцию 5, весенний семестр)



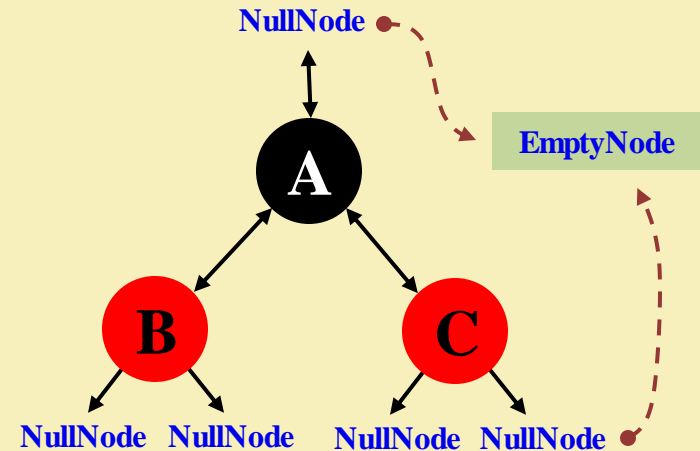
Реализация red-black tree (rbtree)

```
#define COLOR_RED    0
#define COLOR_BLACK  1
```

```
struct rbtree {
    int key;
    char *value;
    int color;

    struct rbtree *parent;
    struct rbtree *left;
    struct rbtree *right;
};
```

```
struct rbtree EmptyNode = {0, 0, COLOR_BLACK,
                           NULL, NULL, NULL};
struct rbtree *NullNode = &EmptyNode;
```



Пример формирования дерева

```
int main(int argc, char **argv)
{
    struct rbtree *tree = NULL;

    tree = rbtree_add(tree, 10, "10");
    tree = rbtree_add(tree, 5, "5");
    tree = rbtree_add(tree, 3, "3");
    tree = rbtree_add(tree, 11, "11");
    tree = rbtree_add(tree, 12, "12");
    tree = rbtree_add(tree, 6, "6");
    tree = rbtree_add(tree, 8, "8");
    tree = rbtree_add(tree, 9, "9");

    rbtree_print(tree);
    rbtree_free(tree);
    return 0;
}
```

Добавление узла

```
struct rbtree *rbtree_add(struct rbtree *root,
                          int key, char *value)
{
    struct rbtree *node, *parent = NullNode;

    /* Search leaf for new element */
    for (node = root; node != NullNode && node != NULL; )
    {
        parent = node;
        if (key < node->key)
            node = node->left;
        else if (key > node->key)
            node = node->right;
        else
            return root;
    }
}
```


Добавление узла (продолжение)

```
node = malloc(sizeof(*node));  
if (node == NULL)  
    return NULL;
```

```
node->key = key;  
node->value = value;
```

```
node->color = COLOR_RED;  
node->parent = parent;  
node->left = NullNode;  
node->right = NullNode;
```

Добавление узла (продолжение)

```
if (parent != NullNode) {
    if (key < parent->key)
        parent->left = node;
    else
        parent->right = node;
} else {
    root = node;
}
return rbtree_fixup_add(root, node);
}
```

Восстановление свойств после добавления

```
struct rbtree *rbtree_fixup_add(struct rbtree *root,  
                                struct rbtree *node)  
{  
    struct rbtree *uncle;  
  
    /* Current node is RED */  
    while (node != root &&  
           node->parent->color == COLOR_RED)  
    {  
        if (node->parent ==  
            node->parent->parent->left)  
        {  
            /* Node in left tree of grandfather */  
            uncle = node->parent->parent->right;
```

Восстановление свойств после добавления

```
if (uncle->color == COLOR_RED) {
    /* Case 1 - uncle is RED */
    node->parent->color = COLOR_BLACK;
    uncle->color = COLOR_BLACK;
    node->parent->parent->color = COLOR_RED;
    node = node->parent->parent;
} else {
    /* Cases 2 & 3 - uncle is BLACK */
    if (node == node->parent->right) {
        /* Reduce case 2 to case 3 */
        node = node->parent;
        root = rbtree_left_rotate(root, node);
    }
    /* Case 3 */
    node->parent->color = COLOR_BLACK;
    node->parent->parent->color = COLOR_RED;
    root = rbtree_right_rotate(root,
                                node->parent->parent);
}
```

Восстановление свойств после добавления

```
    } else {  
        /*  
        * Node in right tree of grandfather  
        *  
        * Cases 4, 5, 6 - node in right tree  
        * of grandfather  
        */  
        uncle = node->parent->parent->left;  
        if (uncle->color == COLOR_RED) {  
            /* Uncle is RED - case 4 */  
            node->parent->color = COLOR_BLACK;  
            uncle->color = COLOR_BLACK;  
            node->parent->parent->color = COLOR_RED;  
            node = node->parent->parent;  
        } else {  
  
            /* Uncle is BLACK */
```

Восстановление свойств после добавления

```
        /* Uncle is BLACK */
        if (node == node->parent->left) {
            node = node->parent;
            root = rbtree_right_rotate(root, node);
        }
        node->parent->color = COLOR_BLACK;
        node->parent->parent->color = COLOR_RED;
        root = rbtree_left_rotate(root,
                                   node->parent->parent);
    }
}
root->color = COLOR_BLACK;
return root;
}
```

Левый поворот (left rotate)

```
struct rbtree *rbtree_left_rotate(  
    struct rbtree *root, struct rbtree *node)  
{  
    struct rbtree *right = node->right;  
  
    /* Create node->right link */  
    node->right = right->left;  
    if (right->left != NullNode)  
        right->left->parent = node;  
  
    /* Create right->parent link */  
    if (right != NullNode)  
        right->parent = node->parent;
```

Левый поворот (left rotate)

```
if (node->parent != NullNode) {
    if (node == node->parent->left)
        node->parent->left = right;
    else
        node->parent->right = right;
} else {
    root = right;
}

right->left = node;
if (node != NullNode)
    node->parent = right;
return root;
}
```


Правый поворот (right rotate)

```
struct rbtree *rbtree_right_rotate(  
    struct rbtree *root, struct rbtree *node)  
{  
    struct rbtree *left = node->left;  
  
    /* Create node->left link */  
    node->left = left->right;  
    if (left->right != NullNode)  
        left->right->parent = node;  
  
    /* Create left->parent link */  
    if (left != NullNode)  
        left->parent = node->parent;
```

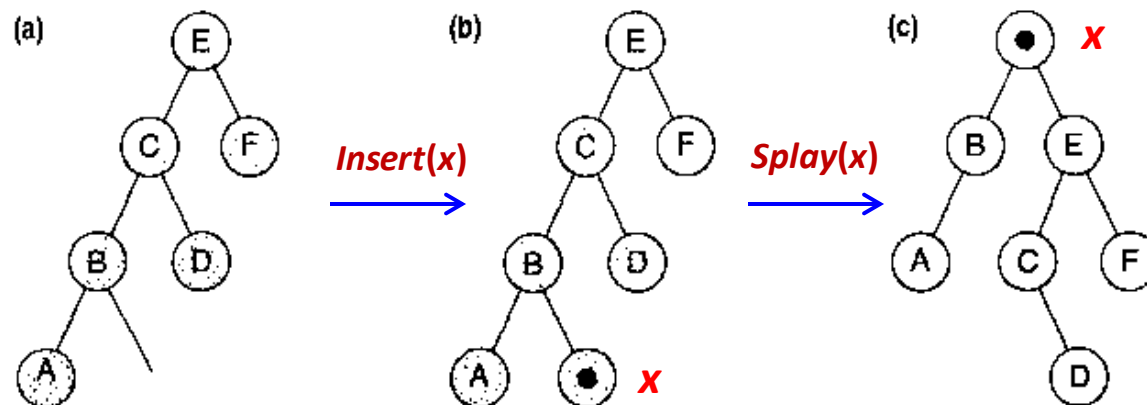
Правый поворот (right rotate)

```
if (node->parent != NullNode) {
    if (node == node->parent->right)
        node->parent->right = left;
    else
        node->parent->left = left;
} else {
    root = left;
}

left->right = node;
if (node != NullNode)
    node->parent = left;
return root;
}
```

Косые деревья (Splay tree)

- **Косое дерево, расширяющееся дерево, скошенное дерево (Splay tree)** – это дерево поиска, обеспечивающее быстрый доступ к часто используемым узлам
- Sleator D., Tarjan R. Self-Adjusting Binary Search Trees
// Journal of the ACM. 1985. Vol. 32 (3). P. 652–686
- **Добавление элемента x в дерево**
 - 1) Спускаемся по дереву, отыскиваем лист для вставки элемента x (как в традиционных BST) и создаем его
 - 2) Применяем к узлу x процедуру Splay, которая поднимает его в корень дерева



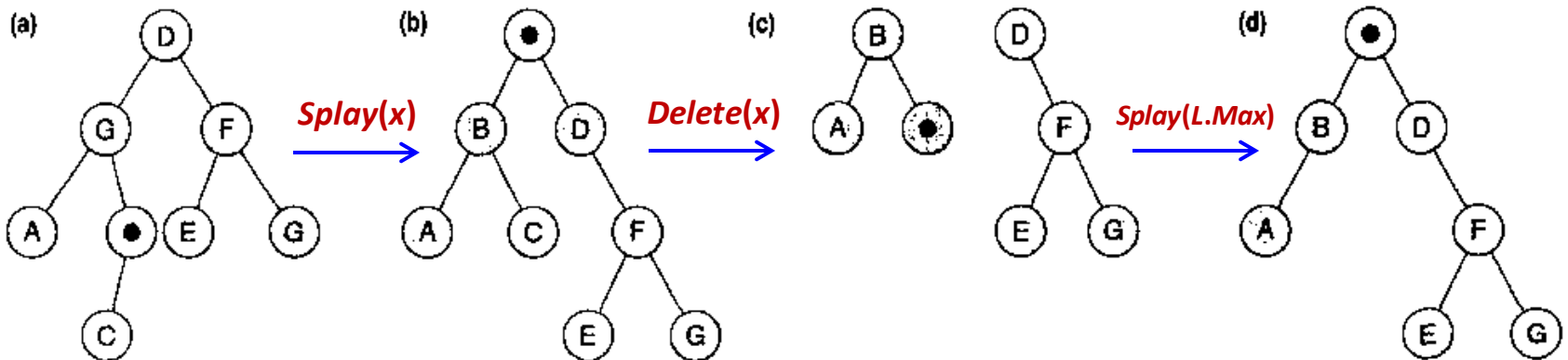
Косые деревья (Splay tree)

■ Удаление элемента x из дерева

- 1) Отыскиваем узел x и удаляем его (как в традиционных BST)
- 2) Применяем к родителю узла x процедуру Splay

или

- 1) Применяем к узлу x процедуру Splay и удаляем его – образуется 2 поддерева L и R
- 2) Реализуется один из методов:
метод 1 – к максимальному элементу поддерева L применяется процедура Splay
метод 2 – к минимальному элементу поддерева R применяется процедура Splay



Косые деревья (Splay tree)

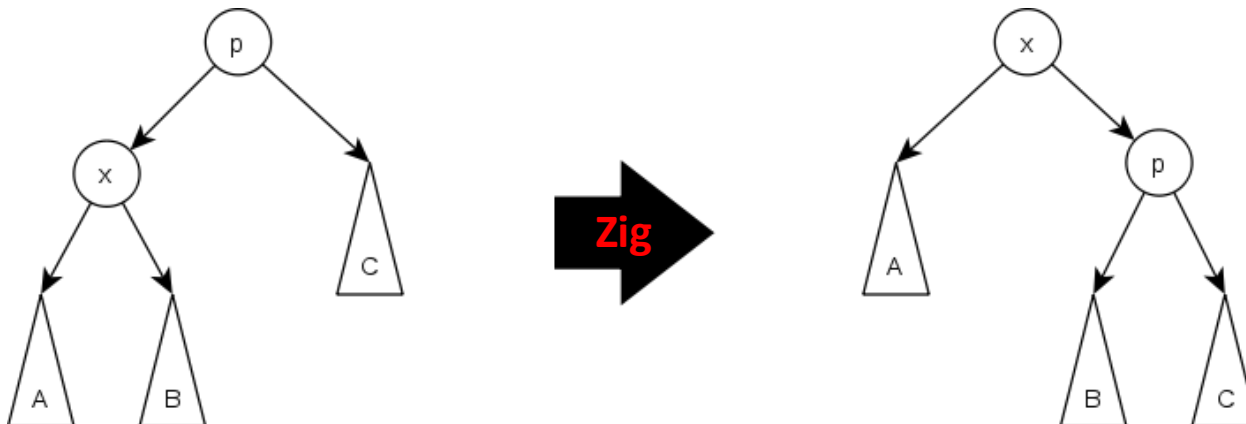
- **Поиск элемента x**

- 1) Отыскиваем узел x (как в традиционных BST)
- 2) При нахождении элемента запускаем Splay для него

- **Процедура Splay(x)**

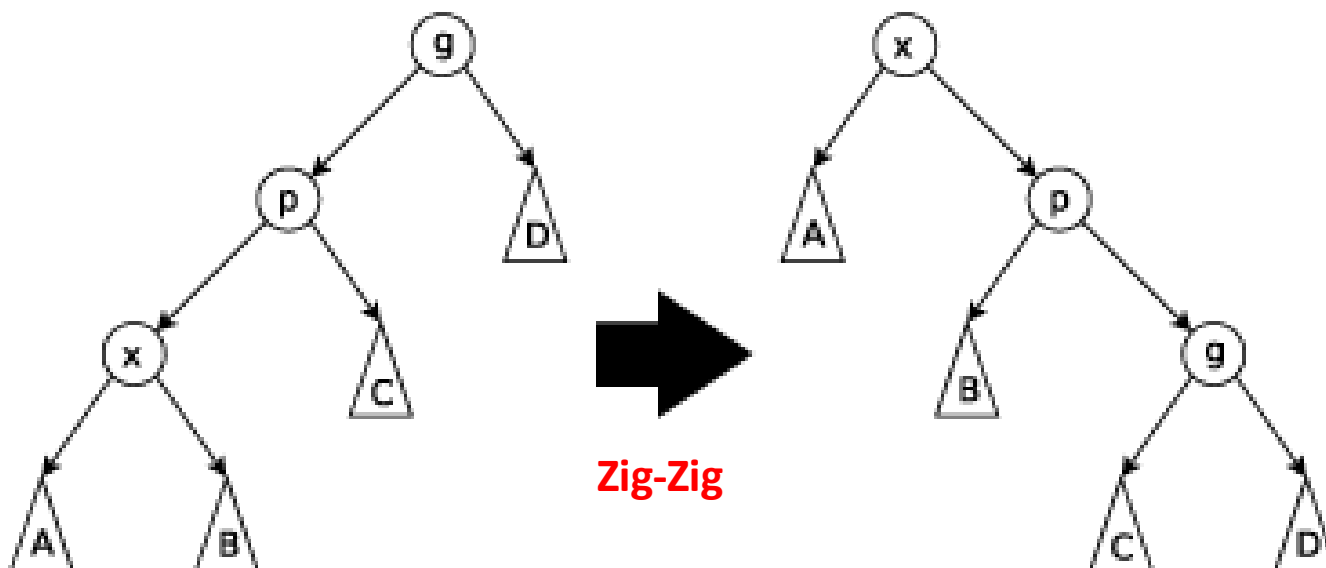
- Splay перемещает узел x в корень при помощи трех операций: Zig, Zig-Zig и Zig-Zag (пока x не станет корнем)
- Пусть p – родитель узла x , g – родитель p (дед узла x)

- **Zig** – выполняется если p корень дерева. Дерево поворачивается по ребру между x и p



Косые деревья (Splay tree)

- Процедура **Splay(x)**
 - Splay перемещает узел x в корень при помощи трех операций: Zig, Zig-Zig и Zig-Zag (пока x не станет корнем)
 - Пусть p – родитель узла x , g – родитель p (дед узла x)
- **Zig-Zig** – выполняется, когда и x , и p являются левыми (или правыми) дочерними элементами. Дерево поворачивается по ребру между g и p , а затем – по ребру между p и x

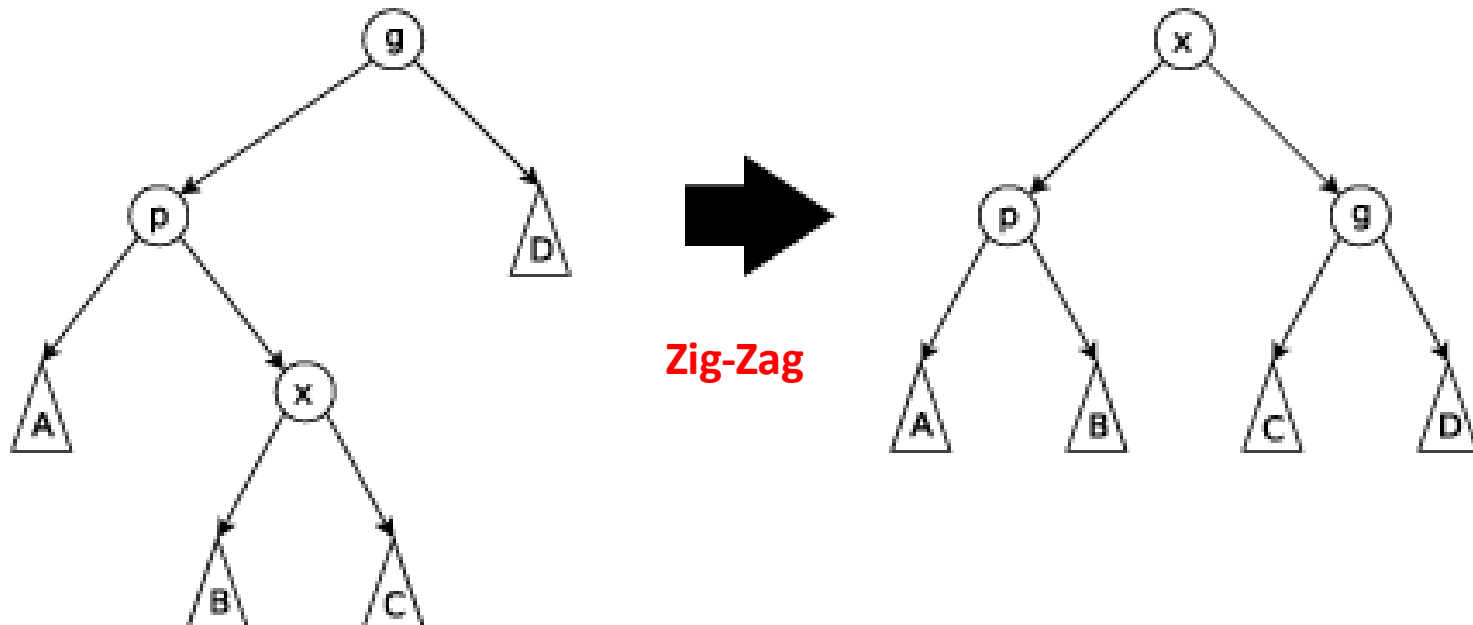


Косые деревья (Splay tree)

■ Процедура Splay(x)

- Splay перемещает узел x в корень при помощи трех операций: Zig, Zig-Zig и Zig-Zag (пока x не станет корнем)
- Пусть p – родитель узла x , g – родитель p (дед узла x)

- **Zig-Zag** – выполняется, когда x является правым сыном, а p – левым (или наоборот). Дерево поворачивается по ребру между p и x , а затем – по ребру между x и g



Косые деревья (Splay tree)

```
void splaytree_splay(struct splaytree_node *x)
{
    while (x->parent) {
        if (!x->parent->parent) {
            if (x->parent->left == x)
                splaytree_right_rotate(x->parent);
            else
                splaytree_left_rotate(x->parent );
        } else if (x->parent->left == x && x->parent->parent->left == x->parent) {
            splaytree_right_rotate(x->parent->parent);
            splaytree_right_rotate(x->parent);
        } else if (x->parent->right == x && x->parent->parent->right == x->parent) {
            splaytree_left_rotate(x->parent->parent);
            splaytree_left_rotate(x->parent);
        } else if (x->parent->left == x && x->parent->parent->right == x->parent) {
            splaytree_right_rotate(x->parent);
            splaytree_left_rotate(x->parent );
        } else {
            splaytree_left_rotate(x->parent);
            splaytree_right_rotate(x->parent);
        }
    }
}
```

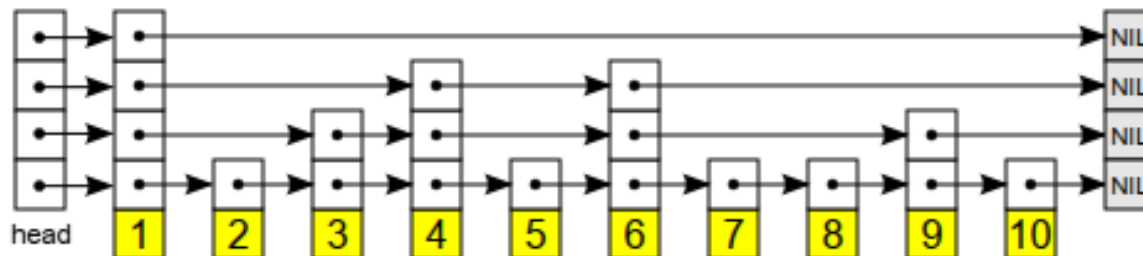

Косые деревья (Splay tree)

Операция	Средний случай (average case)	Худший случай (worst case)
Add (<i>key, value</i>)	$O(\log n)$	<u>Amortized</u> $O(\log n)$
Lookup (<i>key</i>)	$O(\log n)$	
Remove (<i>key</i>)	$O(\log n)$	

- Сложность по памяти: $O(n)$
- В худшем случае дерево может иметь высоту $O(n)$

Списки с пропусками (Skip list)

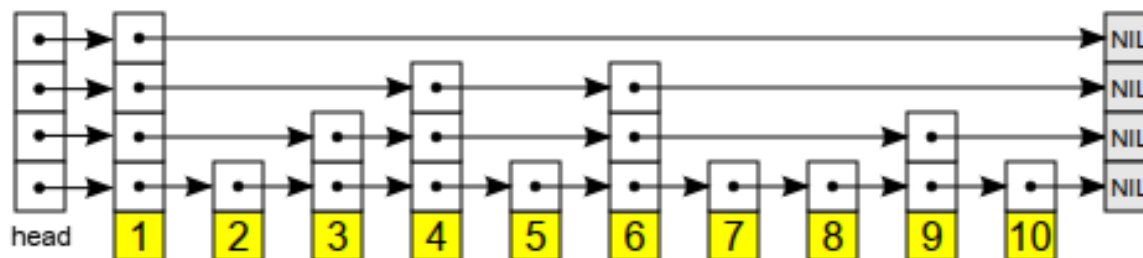
- **Список с пропусками (Skip list)** – это структура данных для реализации словаря, основанная на нескольких параллельных отсортированных связных списках (пропускающих узлы)
- Pugh W. **Skip lists: a probabilistic alternative to balanced trees**
// Communications of the ACM. 1990. Vol. 33 (6), P. 668–676
- **Применение на практике**
 - Cyrus IMAP server (backend DB implementation)
 - QMap (Qt 4.7)
 - Redis persistent key-value store
 - ...



Списки с пропусками (Skip list)

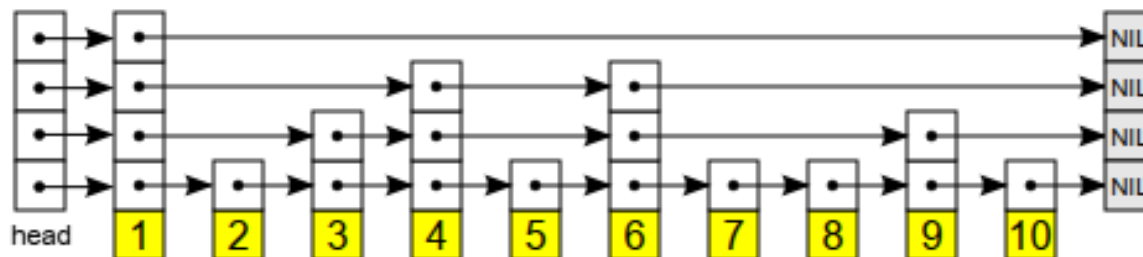
- **Список с пропусками (Skip list)** – это структура данных для реализации словаря, основанная на нескольких параллельных отсортированных связанных списках (пропускающих узлы)
- Pugh W. **Skip lists: a probabilistic alternative to balanced trees**
// Communications of the ACM. 1990. Vol. 33 (6), P. 668–676

Основная идея Skip list – это реализация бинарного поиска для связанных списков (выполнять поиск быстрее чем, тривиальный проход по списку за $O(n)$)



Списки с пропусками (Skip list)

- Нижний уровень списка с пропусками – это упорядоченный связный список, включающий все узлы (все узлы входят в него с вероятностью 1)
- Узел с уровня i присутствует на уровне $i + 1$ с наперед заданной (фиксированной) вероятностью p , например, $1/2$ или $1/4$
- Связи i узлов образует односвязные списки, пропускающие узлы, содержащий менее i связей
- Каждый из t^j узлов должен содержать по меньшей мере $j + 1$ связей
- При вставке новых узлов используется рандомизация



Списки с пропусками (Skip list)

Операция	Средний случай (average case)	Худший случай (worst case)
Add (<i>key, value</i>)	$O(\log n)$	$O(n)$
Lookup (<i>key</i>)	$O(\log n)$	$O(n)$
Remove (<i>key</i>)	$O(\log n)$	$O(n)$

Сложность по памяти: $O(n \log n)$

Литература

1. Кормен Т.Х., Лейзерсон Ч.И., Ривест Р.Л., Штайн К.
Алгоритмы: построение и анализ. – 3-е изд. – М.: Вильямс, 2013 [[CLRS 3ed](#), **С. 341**]
2. Седжвик Р. **Фундаментальные алгоритмы на С++.**
Анализ/Структуры данных/Сортировка/Поиск. – К.:
ДиаСофт, 2001. – 688 с. (**С. 545**)
3. To Google: **red-black tree**

Задание

- Изучить алгоритм удаления элемента из красно-черного дерева [CLRS 3ed, С. 356]
- Прочитать о **списках с пропусками** (skip lists)
 - [Sedgewik, С. 555]
 - Pugh W. **A Skip List Cookbook** //
<http://cg.scs.carleton.ca/~morin/teaching/5408/refs/p90b.pdf>