Лекция 3: АВЛ-деревья (AVL tree)

Курносов Михаил Георгиевич

к.т.н. доцент Кафедры вычислительных систем

Сибирский государственный университет телекоммуникаций и информатики

http://www.mkurnosov.net

Двоичные деревья поиска

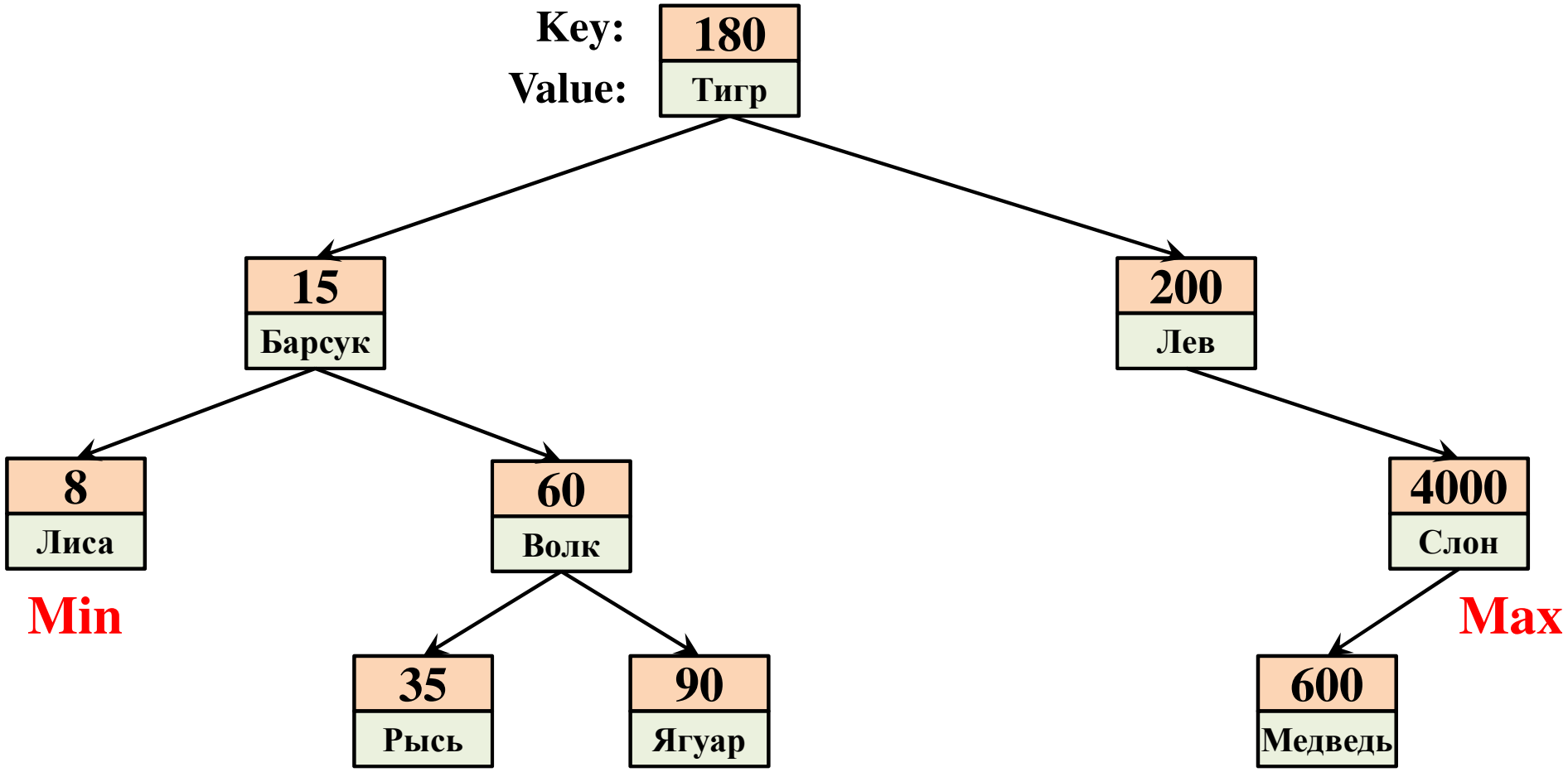
Двоичное дерево поиска (Binary Search Tree, BST) – это двоичное дерево, в котором:

1) каждый узел (node) имеет не более двух дочерних узлов (child nodes)

2) каждый узел содержит ключ (key) и значение (value) и для него выполняются следующие условия:

- ключи всех узлов левого поддерева меньше значения ключа родительского узла
- ключи всех узлов правого поддерева больше значения ключа родительского узла

Двоичные деревья поиска (Binary Search Trees)



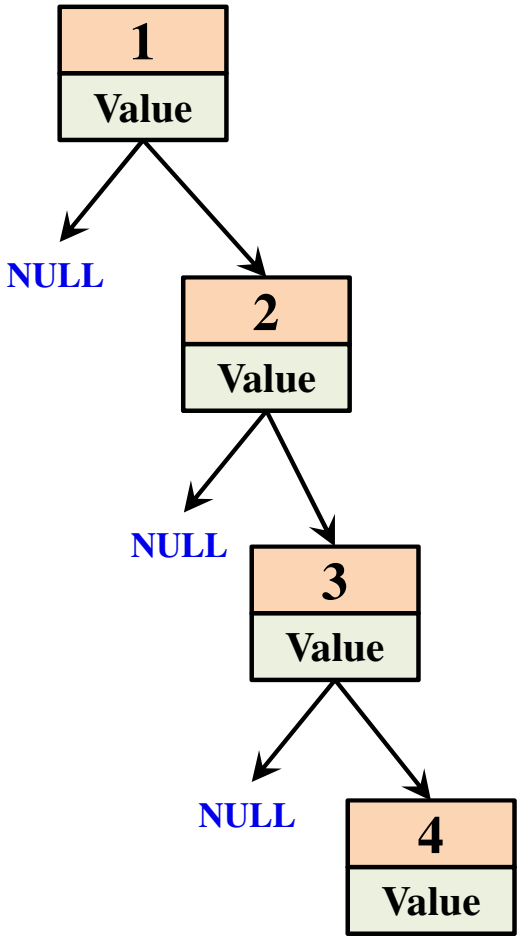
9 узлов, глубина (depth) = 3

Двоичные деревья поиска (Binary Search Trees)

1. Операции имеют трудоемкость пропорциональную высоте 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 n




```
bstree_add(1, value)
bstree_add(2, value)
bstree_add(3, value)
bstree_add(4, value)
```

Двоичные деревья поиска (Binary Search Trees)

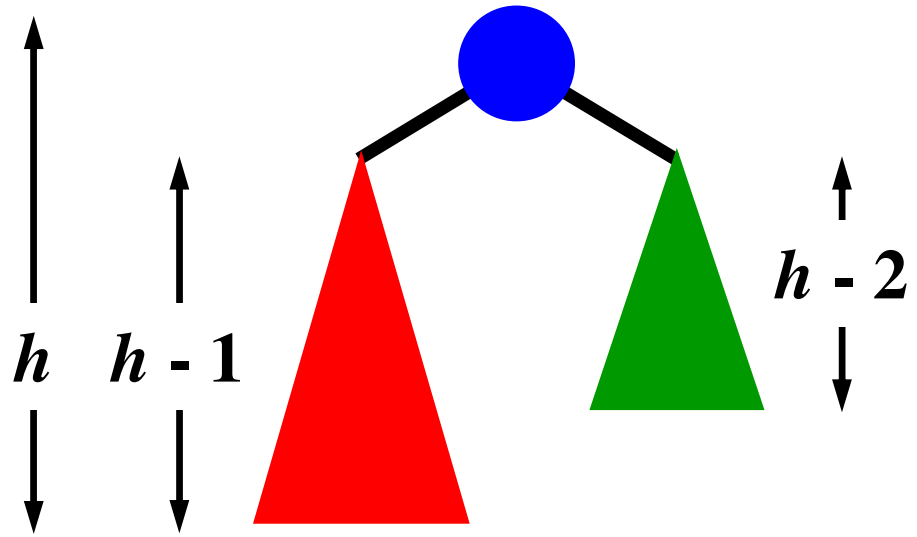
Операция	Средний случай (average case)	Худший случай (worst case)
Add(*key, value*)	$O(\log n)$	$O(n)$
Lookup(*key*)	$O(\log n)$	$O(n)$
Remove(*key*)	$O(\log n)$	$O(n)$
Min	$O(\log n)$	$O(n)$
Max	$O(\log n)$	$O(n)$

Сбалансированные деревья поиска

- **Сбалансированное дерево поиска (self-balancing binary search tree)** – дерево поиска, в котором высота поддеревьев любого узла различаются не более чем на заданную константу k
- Виды сбалансированных деревьев поиска:
  - АВЛ-деревья (AVL tree)
  - Красно-черные деревья (Red-black tree)
  - B-деревья (B-tree)
  - Splay tree
  - AA tree
  - Treap
  - Scapegoat tree
  - …

AVL-деревья

- **АВЛ-дерево (AVL tree)** – сбалансированное по высоте двоичное дерево поиска, в котором у любой вершины высота левого и правого поддеревьев различаются не более чем на 1



- GNU libavl
- libdict
- Python avllib
- avlmap

- Авторы:

  Адельсон-Вельский Г.М., Ландис Е.М. **Один алгоритм организации информации** // Доклады АН СССР. – **1962**. Т. 146, № 2. – С. 263–266.

AVL tree

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

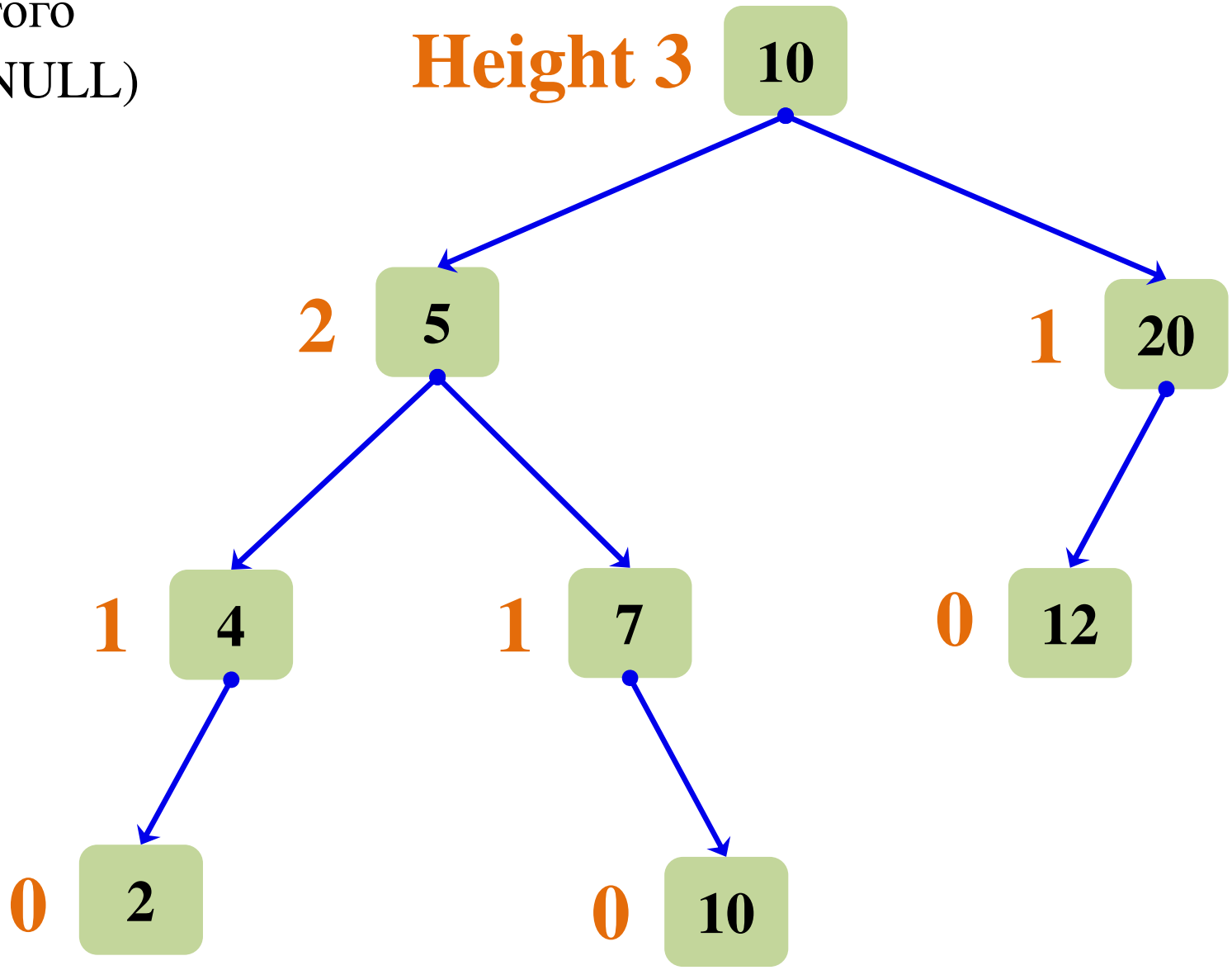
Сложность по памяти: $O(n)$

AVL-деревья

- **Основная идея**
  Если вставка или удаление элемента приводит к нарушению сбалансированности дерева, то выполняется его балансировка
- **Коэффициент сбалансированности узла (balance factor)** – это разность высот его левого и правого поддеревьев
- В АВЛ-дереве коэффициент сбалансированности любого узла принимает значения из множества {-1, 0, 1}
- **Высота узла (height)** – это длина наибольшего пути от него до дочернего узла, являющего листом
- Высота листа равна 0
- Высота пустого поддерева (NULL) равна -1

Высота узла (Node height)

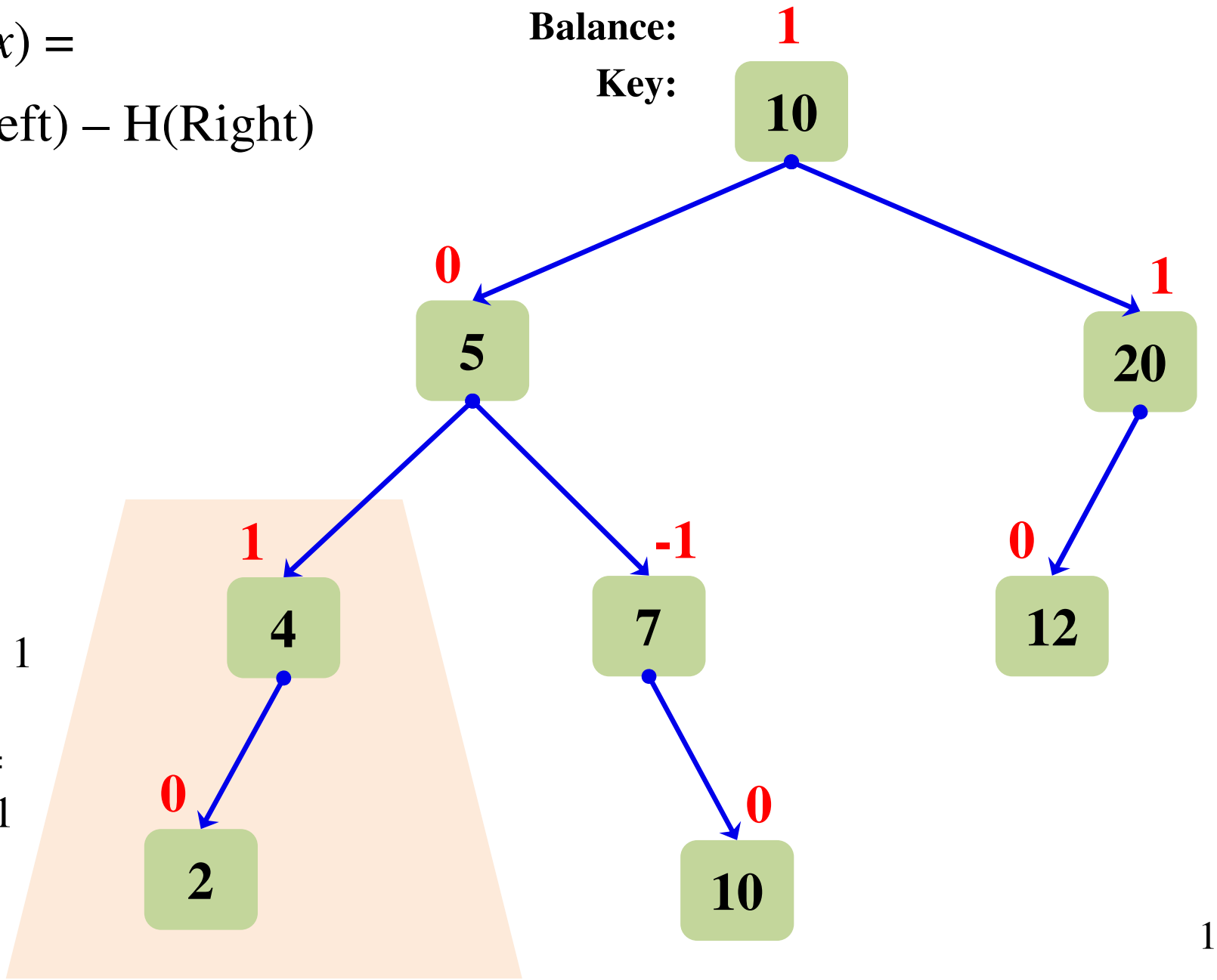
Высота пустого поддерева (NULL) равна -1



Коэффициент сбалансированности

Balance(x) =

H(Left) – H(Right)



Высота поддерева = 1

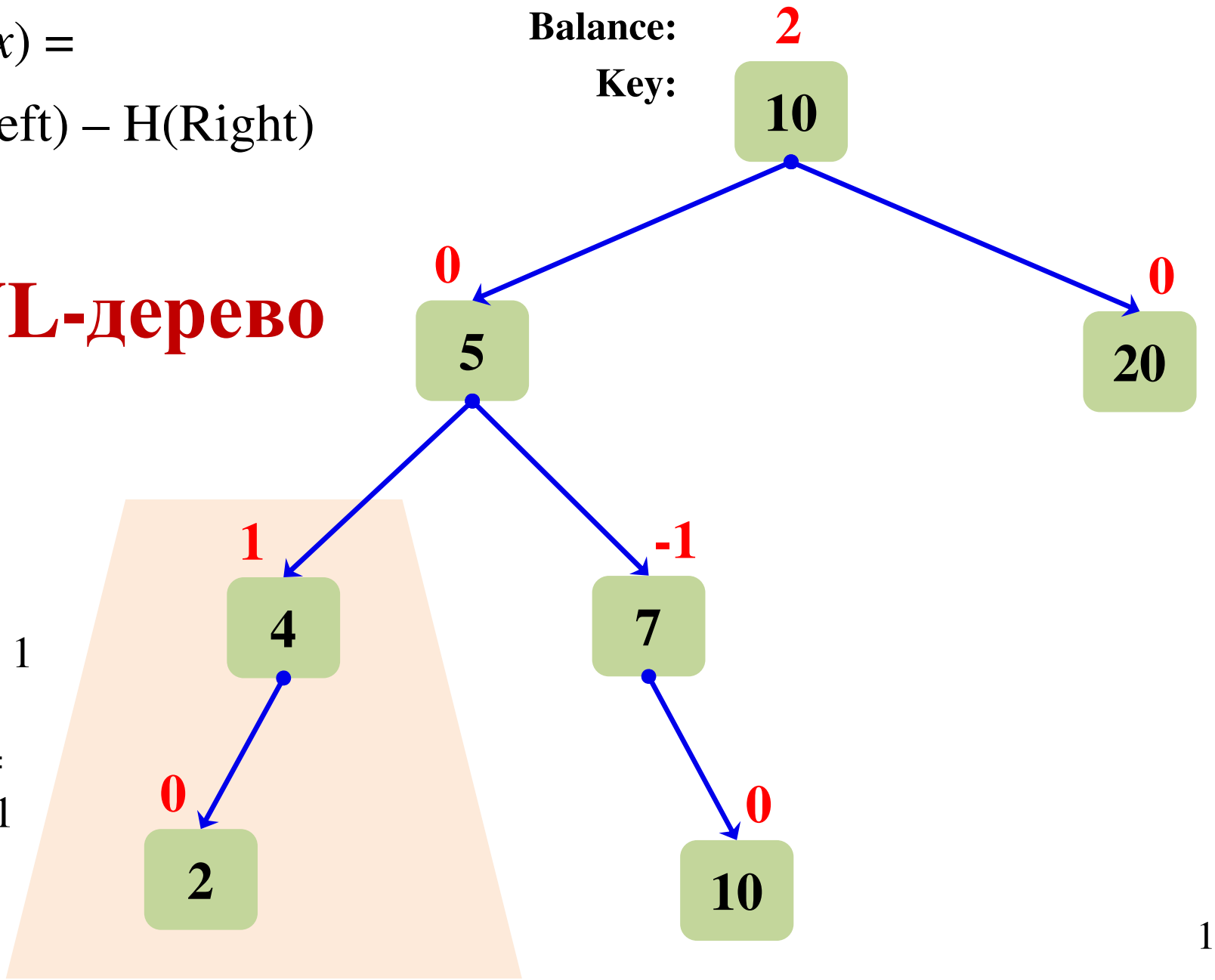
Balance(4) =
= 0 - (-1) = 1

Коэффициент сбалансированности

Balance(x) =

H(Left) – H(Right)

Не AVL-дерево



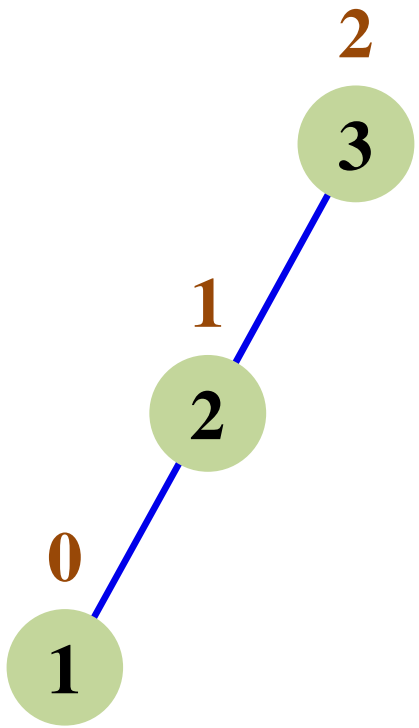
Высота поддерева = 1

Balance(4) =
= 0 - (-1) = 1

Балансировка дерева (Rebalancing)

- **После добавления** нового элемента необходимо обновить коэффициенты сбалансированности родительских узлов
- Если любой родительский узел принял значение -2 или 2, то необходимо выполнить балансировку поддерева путем **поворота (rotation)**
- Типы поворотов:
  - Одиночный правый поворот (R-rotation, single right rotation)
  - Одиночный левый поворот (L-rotation, single left rotation)
  - Двойной лево-правый поворот (LR-rotation, double left-right rotation)
  - Двойной право-левый поворот (RL-rotation, double right-left rotation)

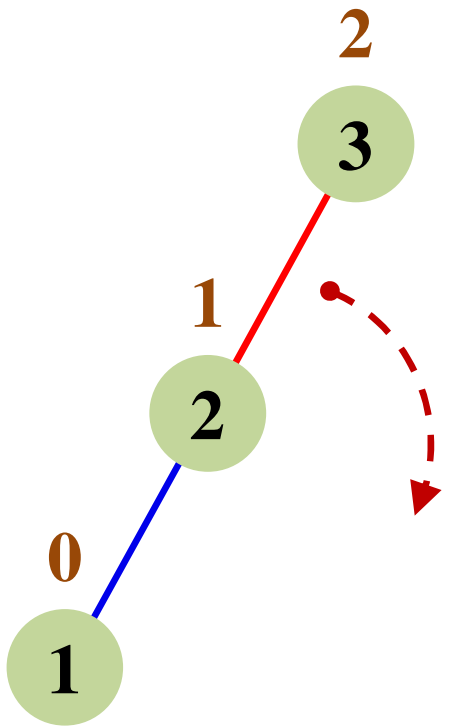
Правый поворот (R-rotation)



- В левое поддерево добавили элемент 1
- Дерево не сбалансированно H(Left)=1 > H(Right)=-1
- Необходимо увеличить высоту правого поддерева

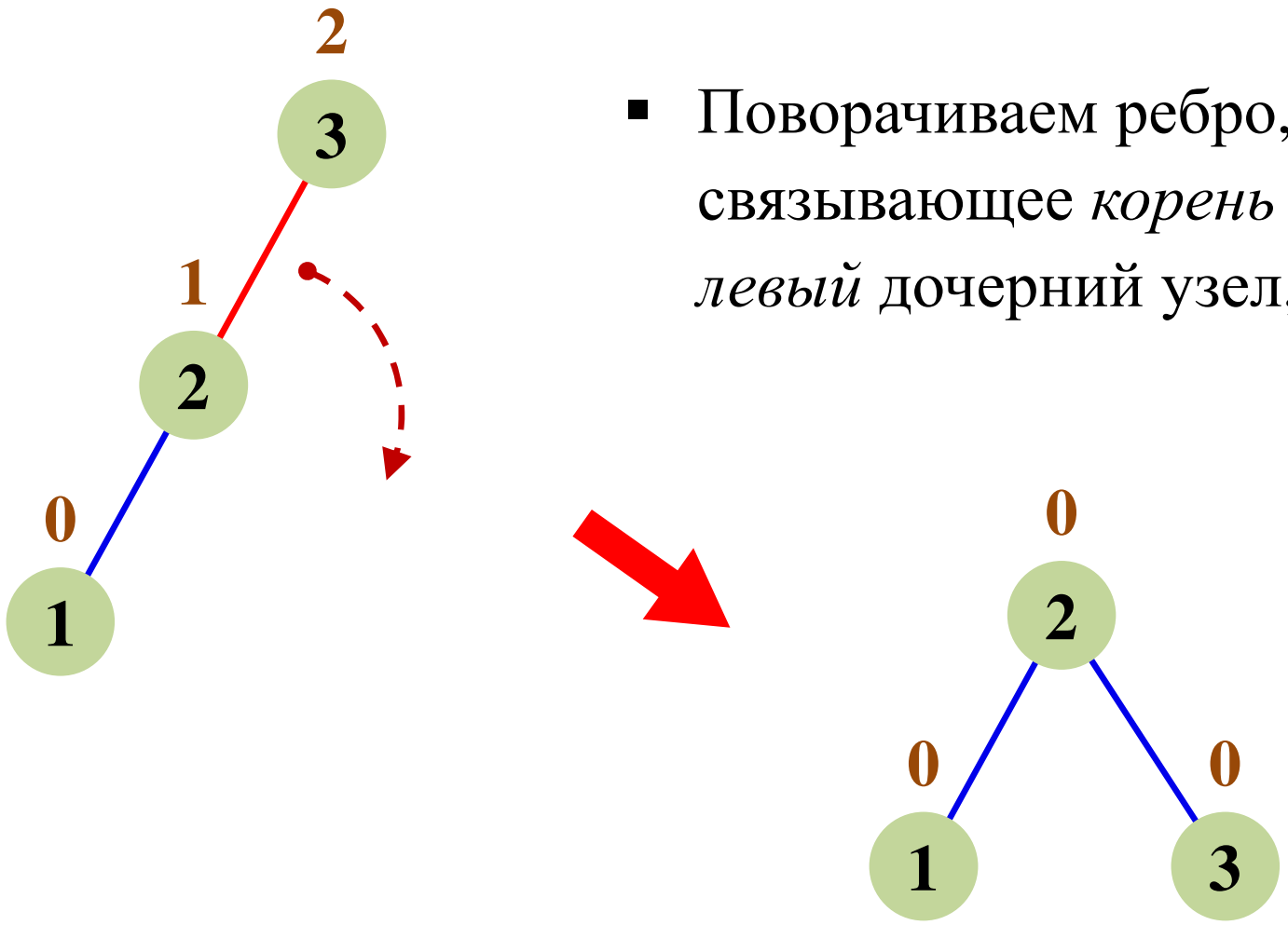
Left Left case

Правый поворот (R-rotation)



- Поворачиваем ребро, связывающее *корень* и его *левый* дочерний узел, *вправо*

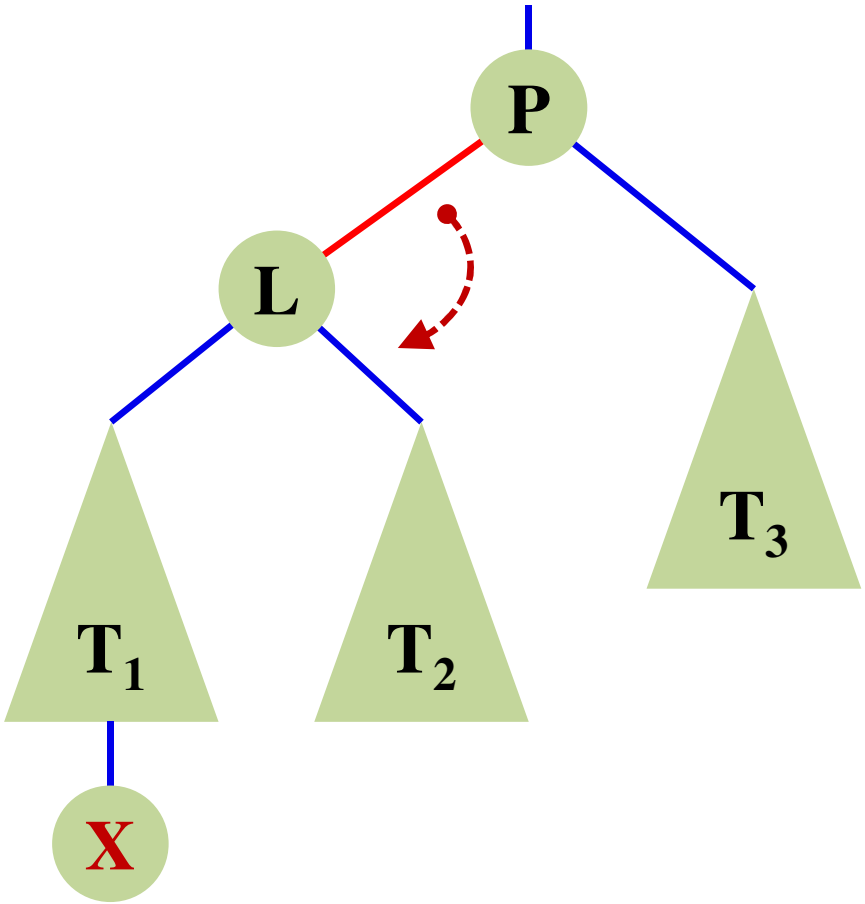
Правый поворот (R-rotation)



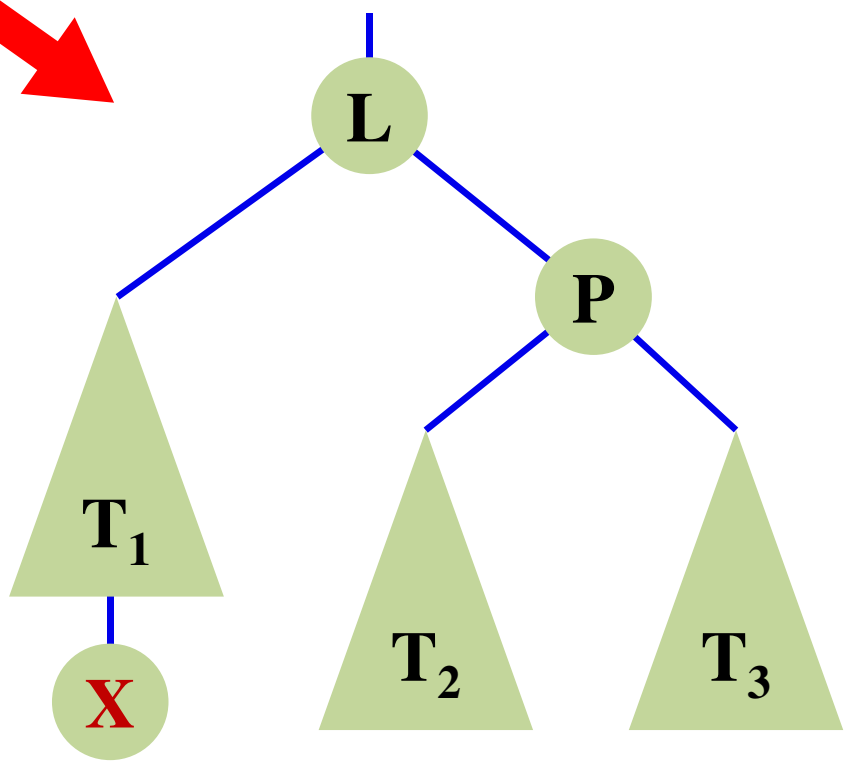
- Поворачиваем ребро, связывающее *корень* и его *левый* дочерний узел, *вправо*

Дерево сбалансированно

Правый поворот (R-rotation)

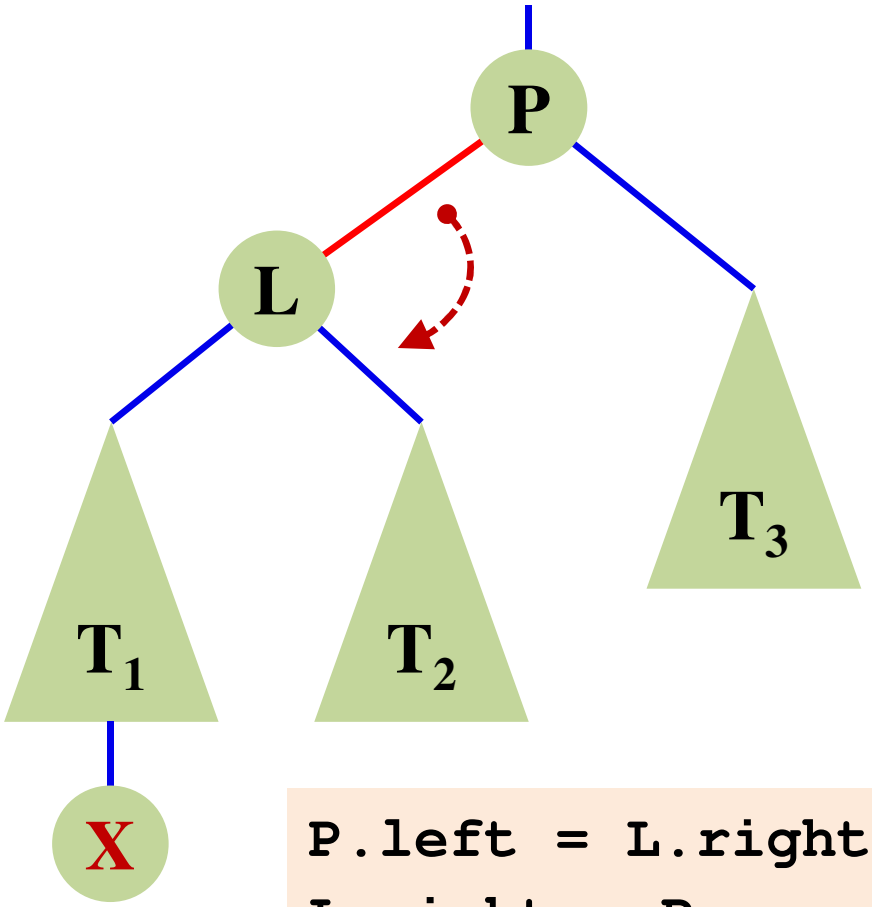


Правый поворот
в общем случае



- В левое поддерево вставлен элемент X
- Дерево не сбалансированно H(Left) > H(Right)

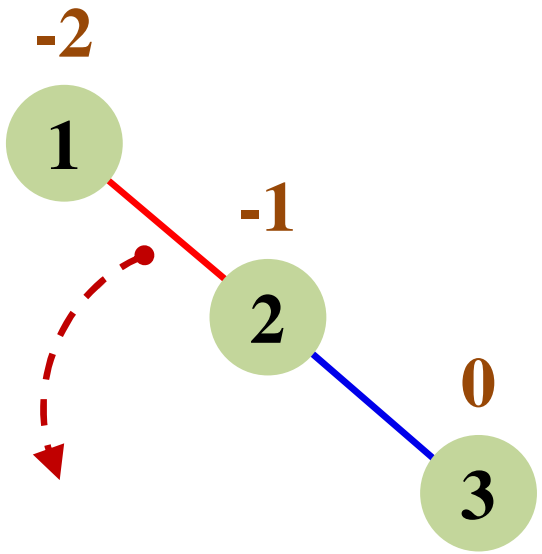
Правый поворот (R-rotation)



Правый поворот в общем случае

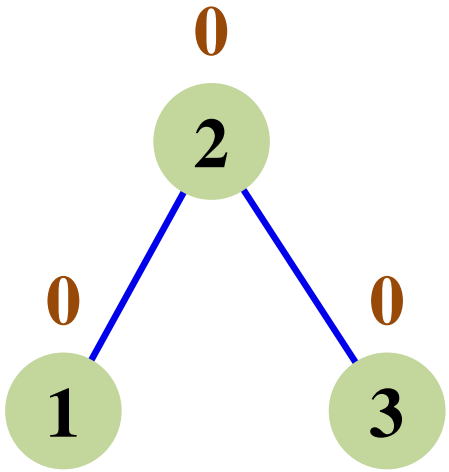
```
P.left = L.right
L.right = P
P.height = max(P.left.height,
               P.right.height) + 1
L.height = max(L.left.height,
               P.height) + 1
```

Левый поворот (L-rotation)



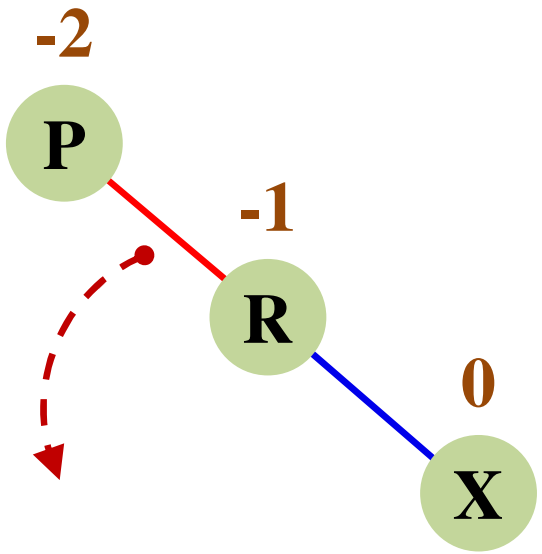
- В правое поддерево вставлен элемент 3
- Поворачиваем ребро, связывающее *корень* и его *правый* дочерний узел, *влево*

Right Right case



Дерево сбалансированно

Левый поворот (L-rotation)

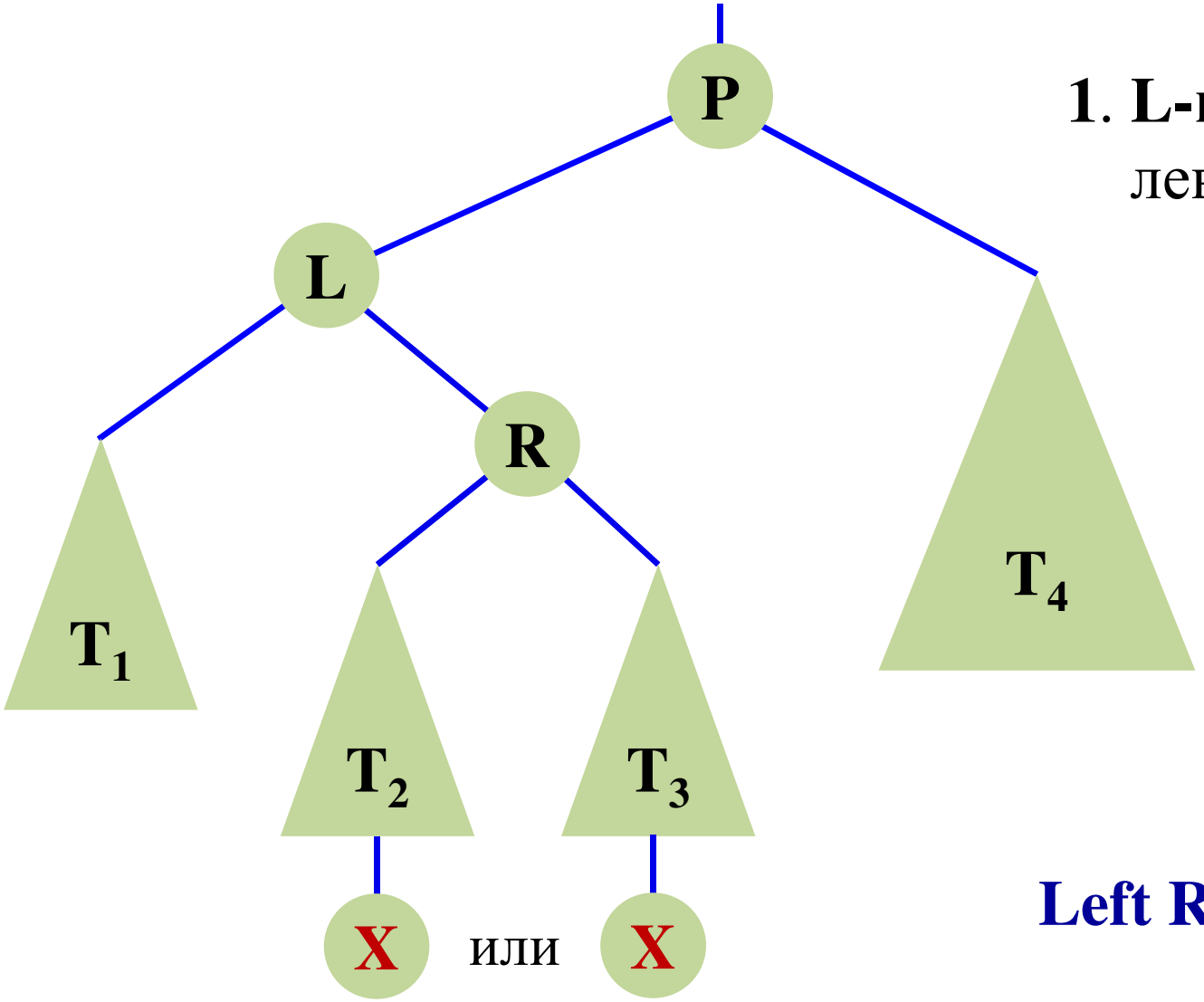


- В правое поддерево вставлен элемент 3
- Поворачиваем ребро, связывающее *корень* и его *правый* дочерний узел, *влево*

```
P.right = R.left
R.left = P
P.height = max(P.left.height, P.right.height) + 1
R.height = max(R.right.height, P.height) + 1
```

Двойной лево-правый поворот (LR-rotation)

- LR-поворот выполняется после добавления элемента в правое поддерево левого дочернего узла дерева

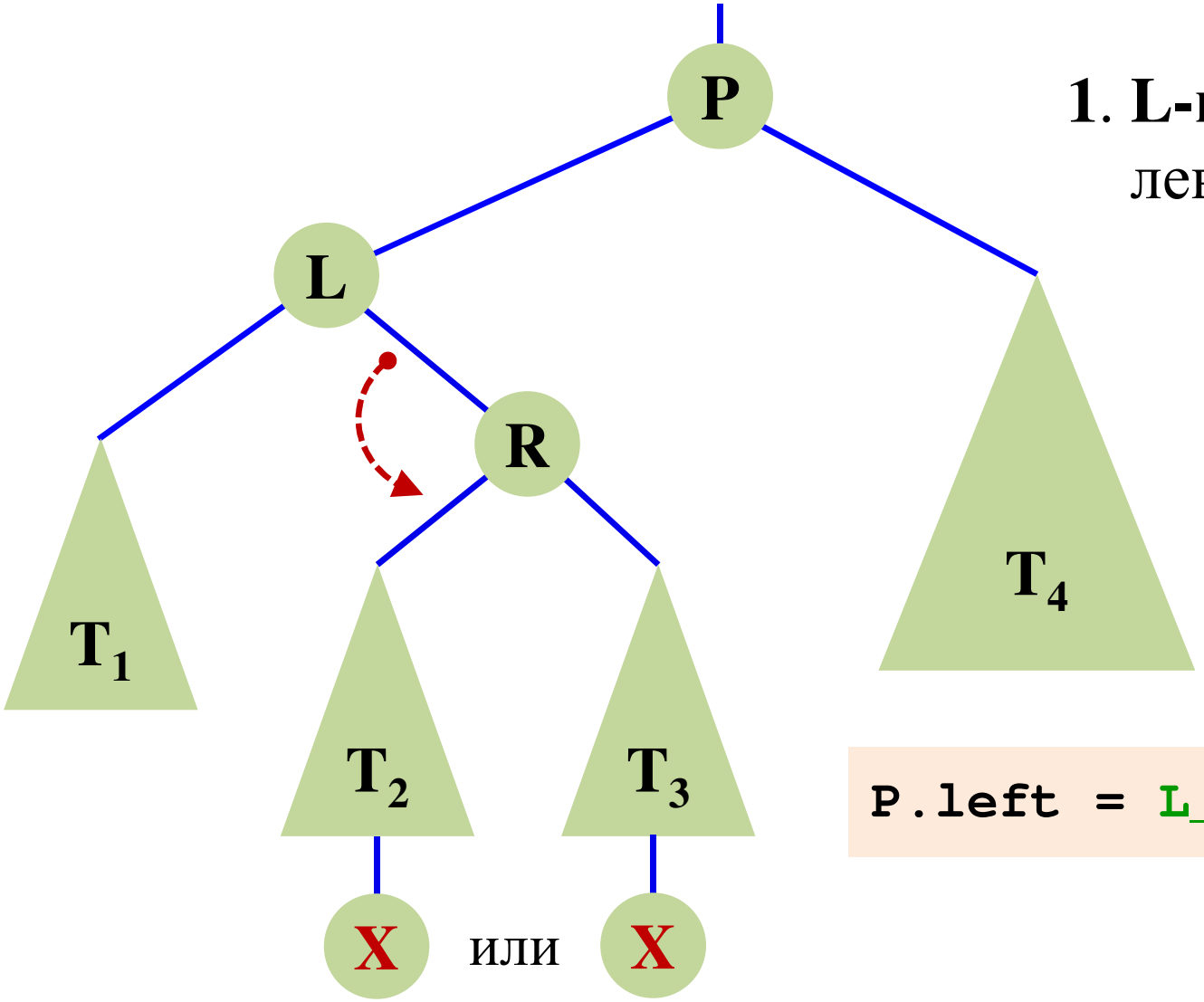


1. **L-поворот** левого поддерева P
2. **R-поворот** нового дерева с вершиной P

Left Right case

Двойной лево-правый поворот (LR-rotation)

- LR-поворот выполняется после добавления элемента в правое поддерево левого дочернего узла дерева

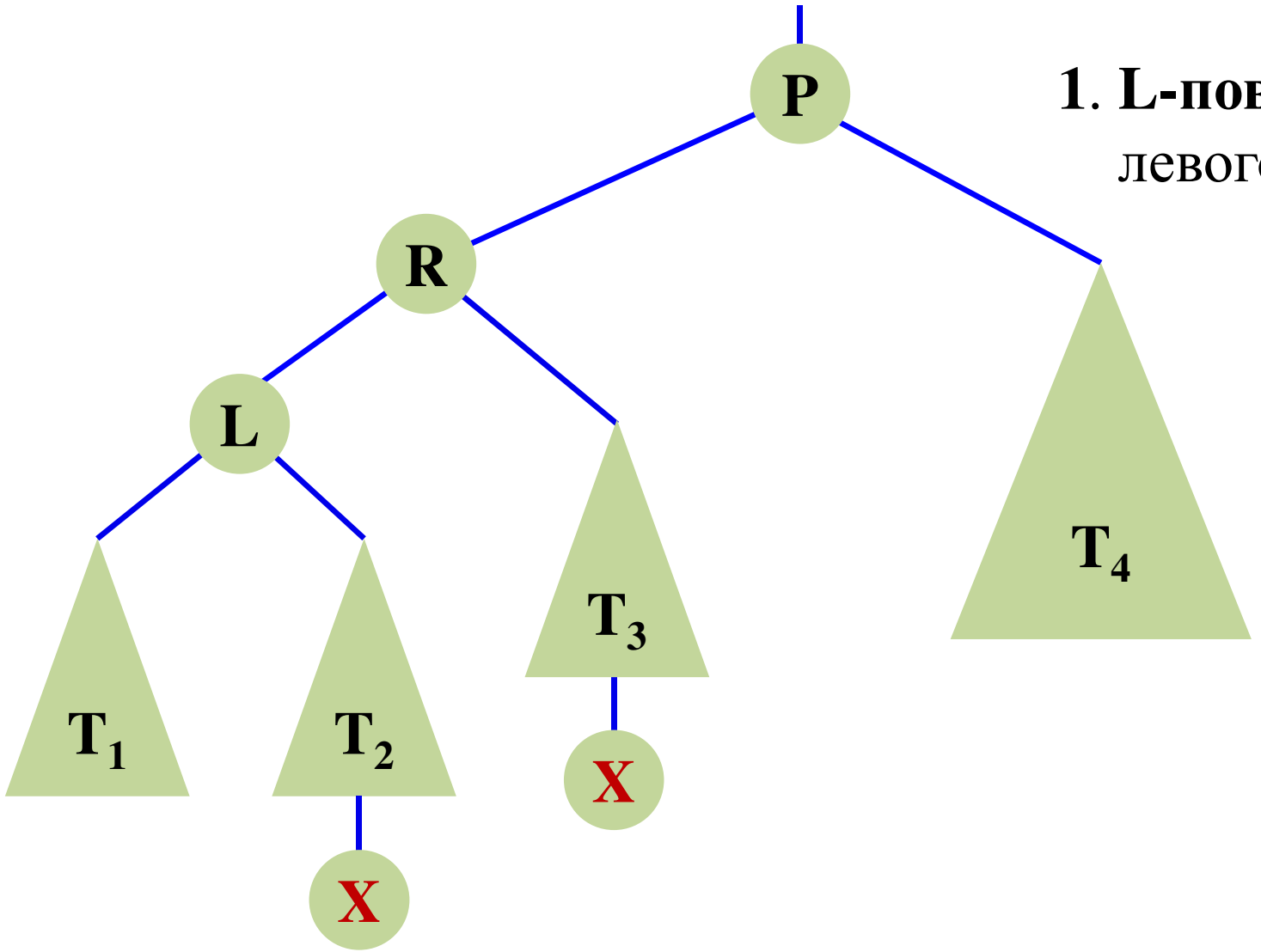


1. L-поворот левого поддерева P

```
P.left = L_rotate(P.left)
```

Двойной лево-правый поворот (LR-rotation)

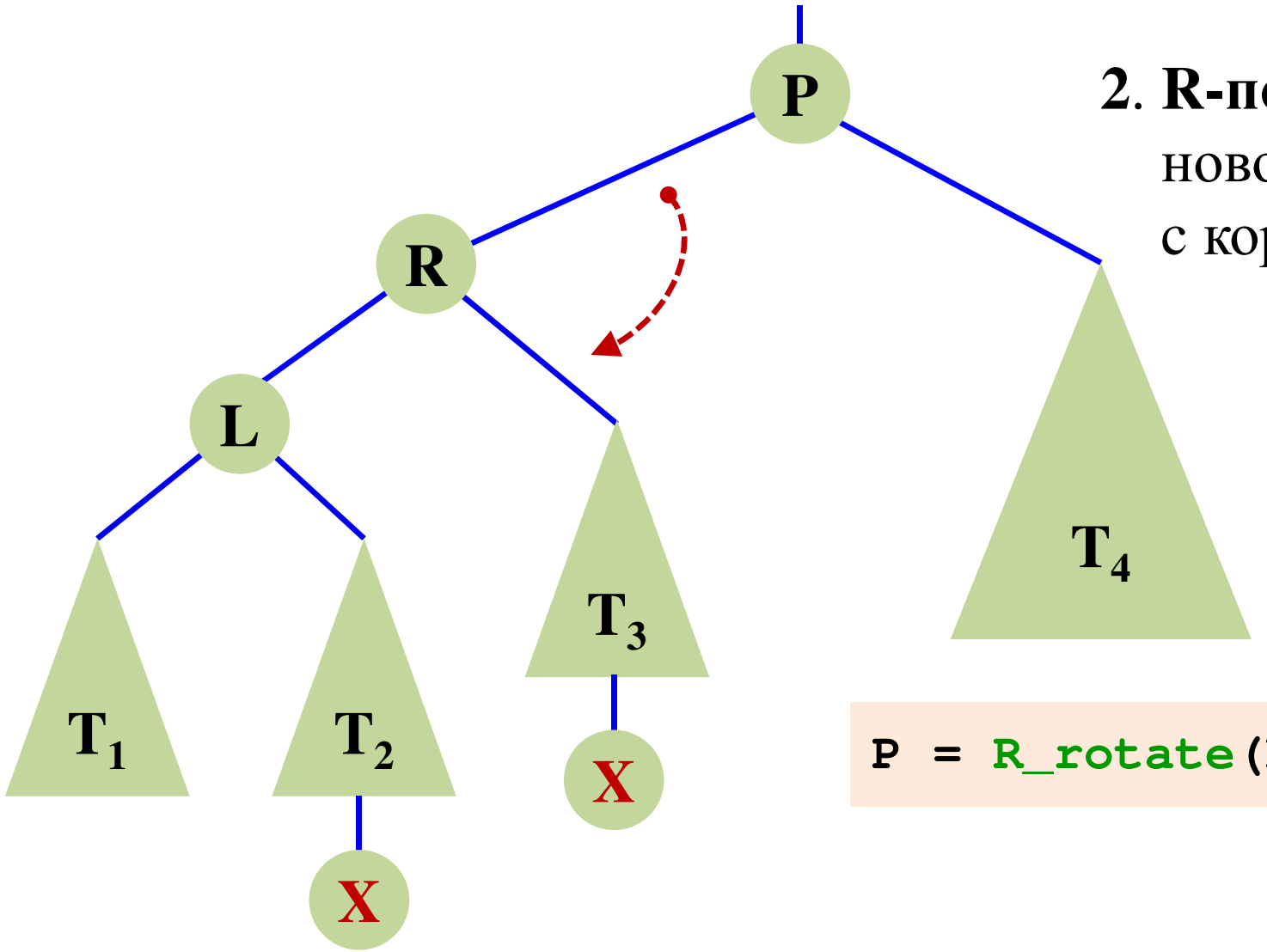
- LR-поворот выполняется после добавления элемента в правое поддерево левого дочернего узла дерева



1. L-поворот левого поддерева P

Двойной лево-правый поворот (LR-rotation)

- LR-поворот выполняется после добавления элемента в правое поддерево левого дочернего узла дерева

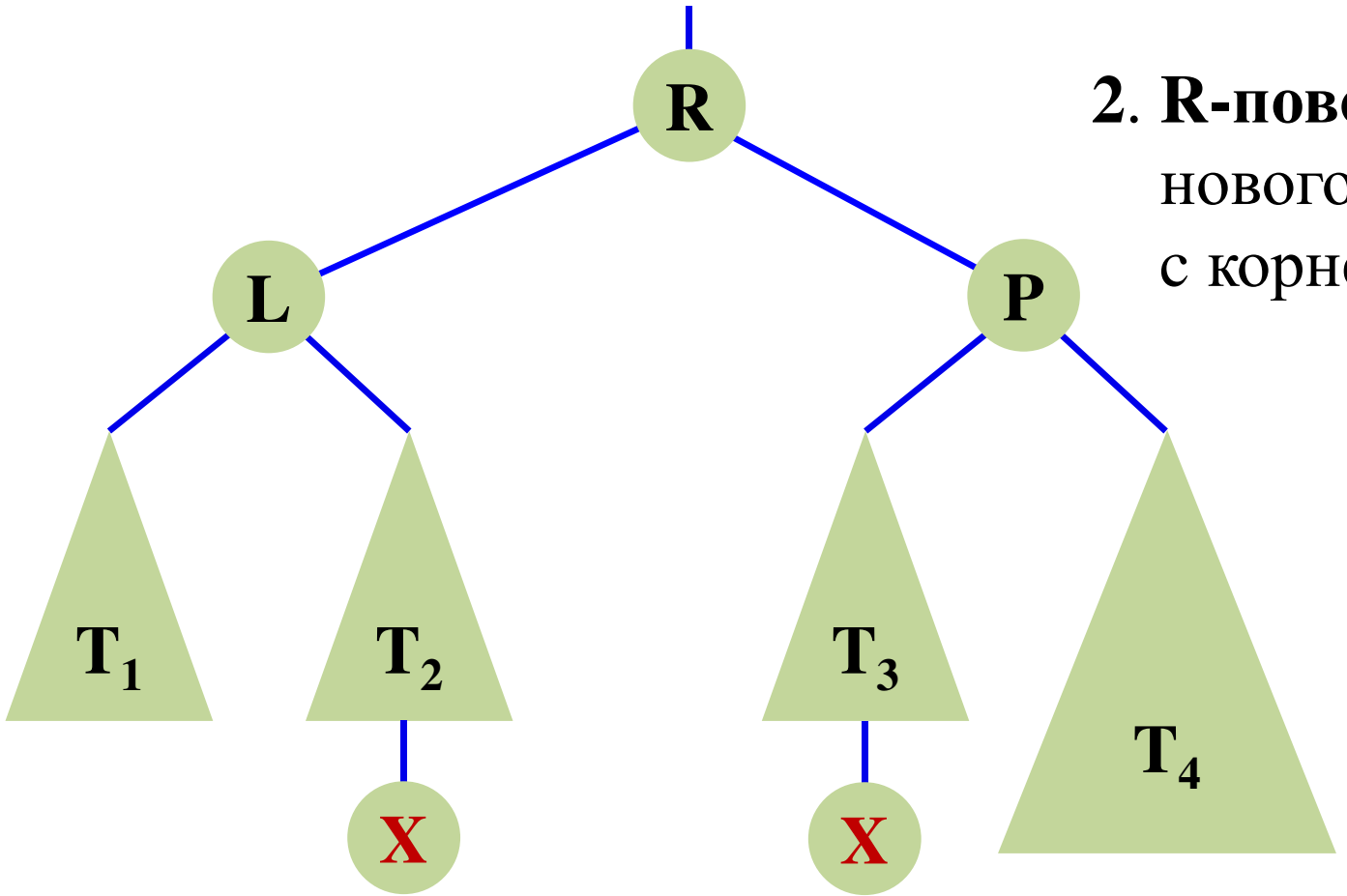


2. **R-поворот** нового дерева с корнем P

```
P = R_rotate(P)
```

Двойной лево-правый поворот (LR-rotation)

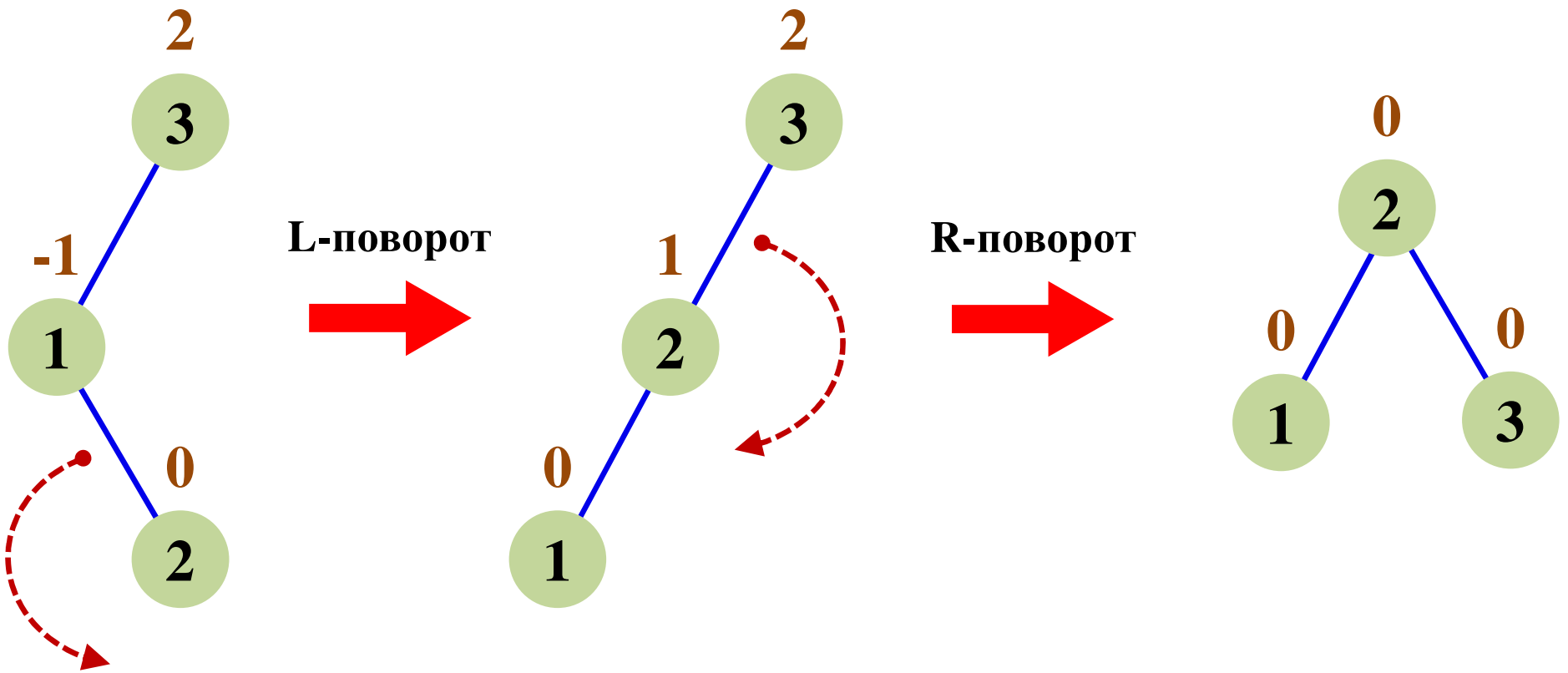
- LR-поворот выполняется после добавления элемента в правое поддерево левого дочернего узла дерева



2. **R-поворот** нового дерева с корнем P

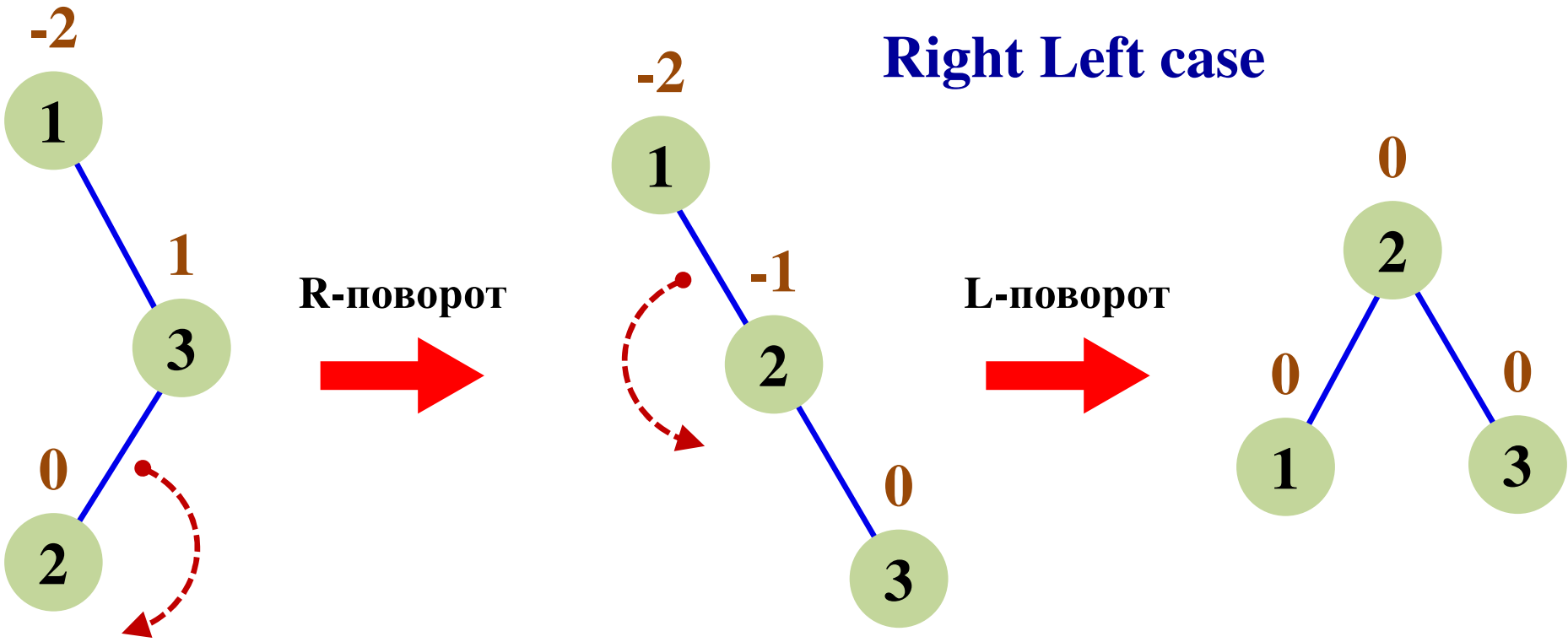
Двойной лево-правый поворот (LR-rotation)

- LR-поворот выполняется после добавления элемента в правое поддерево левого дочернего узла дерева



Двойной право-левый поворот (RL-rotation)

- RL-поворот выполняется после добавления элемента в левое поддерево правого дочернего узла дерева




```
P.right = R_rotate(P.right)
P = L_rotate(P)
```

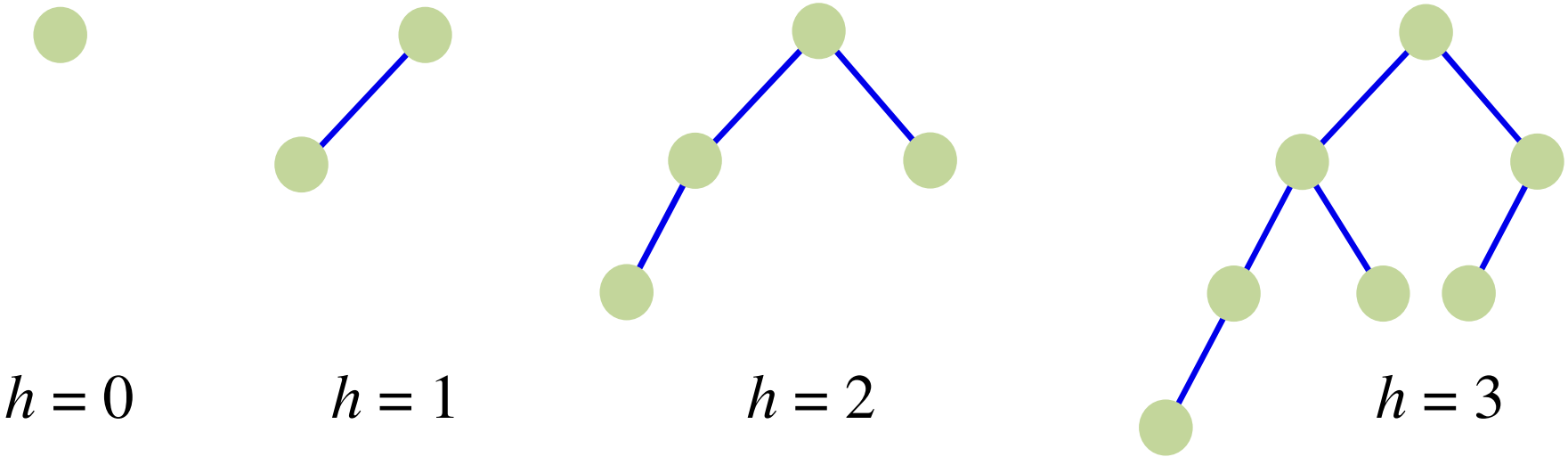
Повороты в АВЛ-дереве

- Любой поворот выполняется за константное время – вычислительная сложность $O(1)$
- Любой поворот сохраняет свойства бинарного дерева поиска (распределение ключей по левыми правым поддеревьям)

Анализ эффективности АВЛ-деревьев

- Оценим сверху высоту АВЛ-дерева, содержащего n элементов
- Обозначим через $N(h)$ минимальное количество узлов необходимых для формирования АВЛ-дерева высоты h

$$N(-1) = 0,\ N(0) = 1,\ N(1) = 2,\ N(2) = 4,\ N(3) = 7,\ \ldots$$
$$0, 1, 2, 4, 7, 12, 20, 33, 54, \ldots$$



$h = 0$ $h = 1$ $h = 2$ $h = 3$

Анализ эффективности АВЛ-деревьев

$$N(h) = N(h-1) + N(h-2) + 1$$

- $N(h)$: 1, 2, 4, 7, 12, 20, 33, 54, …
- Fibonacci(h): 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, …

$$\boldsymbol{N(h) = F(h+3) - 1}, \text{ для } h \geq 0$$

Из формулы Бине для h-го члена последовательности Фибоначчи

$$F_n = \frac{\left(\frac{1+\sqrt{5}}{2}\right)^n - \left(\frac{1-\sqrt{5}}{2}\right)^n}{\sqrt{5}} = \frac{\varphi^n - (-\varphi)^{-n}}{\varphi - (-\varphi)^{-1}} = \frac{\varphi^n - (-\varphi)^{-n}}{2\varphi - 1}$$

$$\varphi = \frac{1+\sqrt{5}}{2}$$ – золотое сечение

$$F_n \sim \frac{\varphi^n}{\sqrt{5}}$$

Анализ эффективности АВЛ-деревьев

- Оценка сверху высоты $h(n)$ АВЛ-дерева [Wirth89]:

$$\log(n+1) \le h(n) \le 1.4404 \cdot \log(n+2) - 0.328$$

- Оценка сверху высоты $h(n)$ АВЛ-дерева[Levitin2006]:

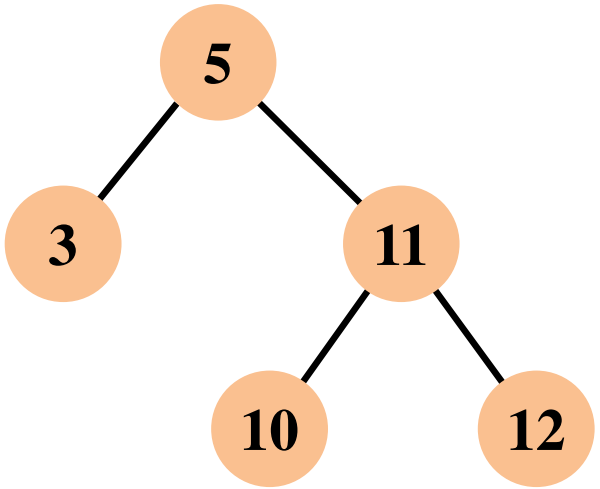
$$\lfloor \log_2 n \rfloor \le h(n) \le 1.4405 \log_2(n+2) - 1.3277$$

AVL Tree

```
struct avltree {
    int key;
    char *value;
    int height;
    struct avltree *left;
    struct avltree *right;
};
```

Построение AVL-дерева

```
int main()
{
    struct avltree *tree = NULL;

    tree = avltree_add(tree, 10, "10");
    tree = avltree_add(tree, 5, "5");
    tree = avltree_add(tree, 3, "3");

    tree = avltree_add(tree, 11, "11");
    tree = avltree_add(tree, 12, "12");
    avltree_print(tree);

    avltree_free(tree);
    return 0;
}
```

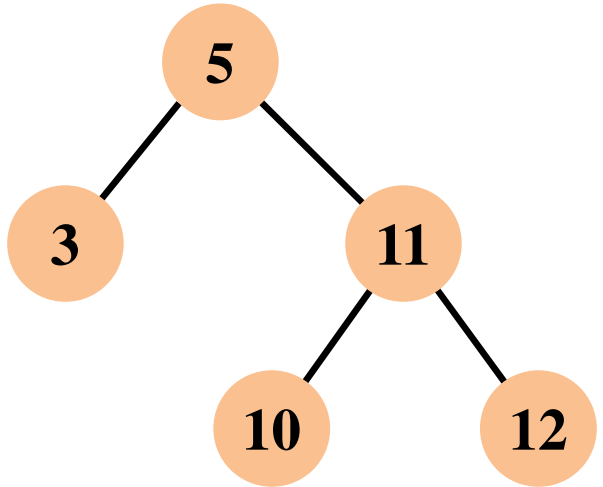


Удаление узлов из AVL-дерева

```
int main()
{
    struct avltree *tree = NULL;

    tree = avltree_add(tree, 5, "5");
    tree = avltree_add(tree, 3, "3");
    /* Code */

    tree = avltree_delete(tree, 5);

    avltree_free(tree);
    return 0;
}
```

Удаление всех узлов из AVL-дерева

```
void avltree_free(struct avltree *tree)
{
    if (tree == NULL)
        return;

    avltree_free(tree->left);
    avltree_free(tree->right);
    free(tree);
}
```



Поиск узла по ключу

```
struct avltree *avltree_lookup(
    struct avltree *tree, int key)
{
    while (tree != NULL) {
        if (key == tree->key) {
            return tree;
        } else if (key < tree->key) {
            tree = tree->left;
        } else {
            tree = tree->right;
        }
    }
    return tree;
}
```

Создание узла

```
struct avltree *avltree_create(int key,
                               char *value)
{
    struct avltree *node;

    node = malloc(sizeof(*node));
    if (node != NULL) {
        node->key = key;
        node->value = value;
        node->left = NULL;
        node->right = NULL;
        node->height = 0;
    }
    return node;
}
```

Высота и баланс узла (поддерева)

```
int avltree_height(struct avltree *tree)
{
    return (tree != NULL) ? tree->height : -1;
}

int avltree_balance(struct avltree *tree)
{
    return avltree_height(tree->left) -
           avltree_height(tree->right);
}
```

Добавление узла

```
struct avltree *avltree_add(
    struct avltree *tree, int key, char *value)
{
    if (tree == NULL) {
        /* Insert new item */
        return avltree_create(key, value);
    }
```

Добавление узла (продолжение)

```
if (key < tree->key) {
    /* Insert into left subtree */
    tree->left = avltree_add(tree->left,
                             key, value);
    if (avltree_height(tree->left) -
        avltree_height(tree->right) == 2)
    {
        /* Subtree is unbalanced */
        if (key < tree->left->key) {
            /* Left left case */
            tree = avltree_right_rotate(tree);
        } else {
            /* Left right case */
            tree = avltree_leftright_rotate(tree);
        }
    }
}
```

Добавление узла (продолжение)

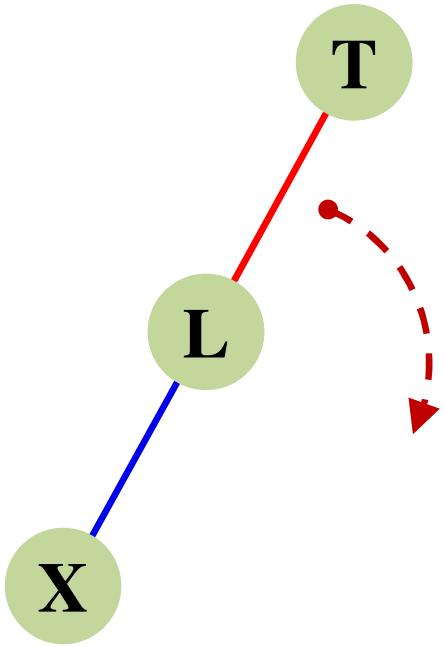
```
else if (key > tree->key) {
    /* Insert into right subtree */
    tree->right = avltree_add(tree->right,
                              key, value);
    if (avltree_height(tree->right) -
        avltree_height(tree->left) == 2)
    {
        /* Subtree is unbalanced */
        if (key > tree->right->key) {
            /* Right right case */
            tree = avltree_left_rotate(tree);
        } else {
            /* Right left case */
            tree = avltree_rightleft_rotate(tree);
        }
    }
}
```

Добавление узла (конец)

```
    tree->height = imax2(
                       avltree_height(tree->left),
                       avltree_height(tree->right)
                   ) + 1;
    return tree;
}
```

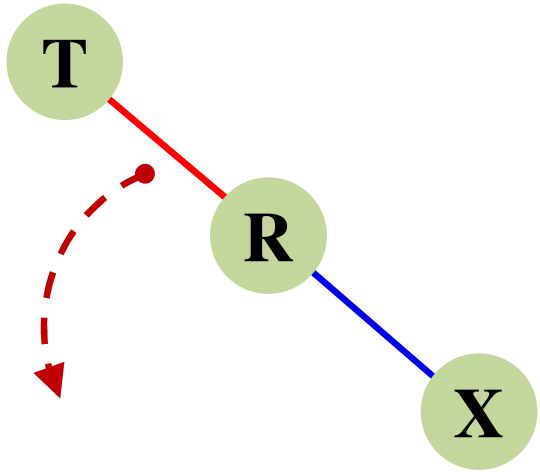
R-поворот (left left case)

```
struct avltree *avltree_right_rotate(
    struct avltree *tree)
{
    struct avltree *left;

    left = tree->left;
    tree->left = left->right;
    left->right = tree;

    tree->height = imax2(
                   avltree_height(tree->left),
                   avltree_height(tree->right)) + 1;
    left->height = imax2(
                   avltree_height(left->left),
                   tree->height) + 1;
    return left;
}
```



L-поворот (right right case)

```
struct avltree *avltree_left_rotate(
    struct avltree *tree)
{
    struct avltree *right;

    right = tree->right;
    tree->right = right->left;
    right->left = tree;

    tree->height = imax2(
                   avltree_height(tree->left),
                   avltree_height(tree->right)) + 1;
    right->height = imax2(
                    avltree_height(right->right),
                    tree->height) + 1;
    return right;
}
```



LR-поворот (left right case)

```
struct avltree *avltree_leftright_rotate(
    struct avltree *tree)
{
    tree->left = avltree_left_rotate(tree->left);
    return avltree_right_rotate(tree);
}
```

RL-поворот (right left case)

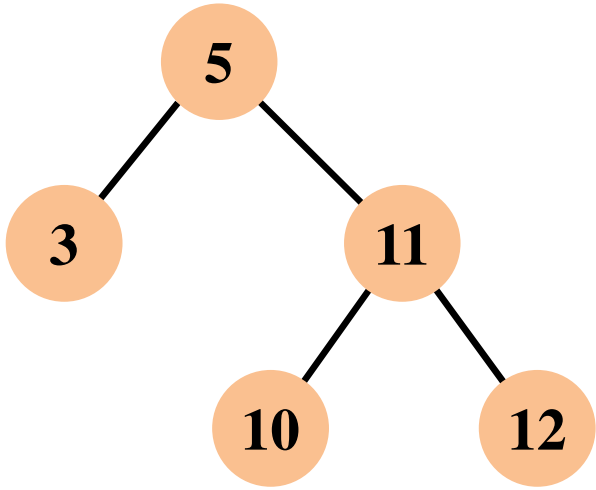
```
struct avltree *avltree_rightleft_rotate(
    struct avltree *tree)
{
    tree->right = avltree_right_rotate(tree->right);
    return avltree_left_rotate(tree);
}
```

Вывод дерева на экран

```
void avltree_print_dfs(struct avltree *tree, int level)
{
    int i;

    if (tree == NULL)
        return;

    for (i = 0; i < level; i++)
        printf("    ");
    printf("%d\n", tree->key);

    avltree_print_dfs(tree->left, level + 1);
    avltree_print_dfs(tree->right, level + 1);
}
```

Вывод дерева на экран

```
tree = avltree_add(tree, 10, "10");
tree = avltree_add(tree, 5, "5");
tree = avltree_add(tree, 3, "3");
tree = avltree_add(tree, 11, "11");
tree = avltree_add(tree, 12, "12");
avltree_print_dfs(tree, 0);
```




```
5
    3
    11
        10
        12
```

Удаление элемента

- Удаление элемента выполняется аналогично добавлению
- После удаления может нарушиться баланс нескольких родительских вершин
- После удаления вершины может потребоваться порядка $O(\log n)$ поворотов поддеревьев

Ленивое удаление элементов (Lazy Deletion)

- С каждым узлом АВЛ-дерева ассоциирован флаг *deleted*
- При удалении узла находим его в дереве и устанавливаем флаг *deleted* = 1 (реализуется за время $O(\log n)$)
- При вставке нового узла с таким же ключом как и у удалённого элемента, устанавливаем у последнего флаг *deleted* = 0 (в поле данных копируем новое значение)
- При достижении порогового значения количества узлов с флагом *deleted* = 1 создаем новое АВЛ-дерево содержащее все не удалённые узлы (*deleted* = 0)
- Поиск не удалённых элементов и их вставка в новое АВЛ-дерево реализуется за время $O(n\log n)$

Литература

1. Левитин А.В. **Алгоритмы: введение в разработку и анализ**. – М.: Вильямс, 2006. – 576 с. **(С. 267-271)**

2. Вирт Н. **Алгоритмы и структуры данных**. – М.: Мир, 1989. – 360 с. **(С. 272-286)**

3. AVL-деревья // Сайт RSDN.ru. – URL: `http://www.rsdn.ru/article/alg/bintree/avl.xml`

4. To Google:

“avl tree” || “avl tree ext:pdf” || “avl tree ext:ppt”