

Лекция 3

АВЛ-деревья (AVL trees)

Курносов Михаил Георгиевич

E-mail: mkurnosov@gmail.com

WWW: www.mkurnosov.net

Курс «Структуры и алгоритмы обработки данных»

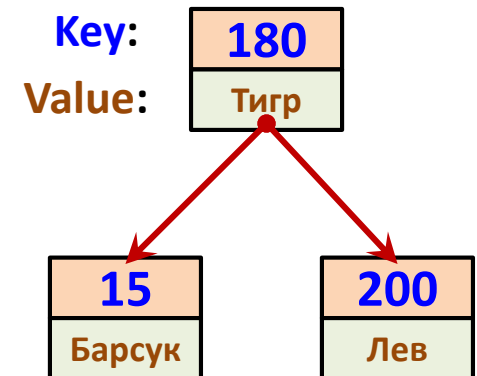
Сибирский государственный университет телекоммуникаций и информатики (Новосибирск)

Осенний семестр, 2015

Двоичные деревья поиска

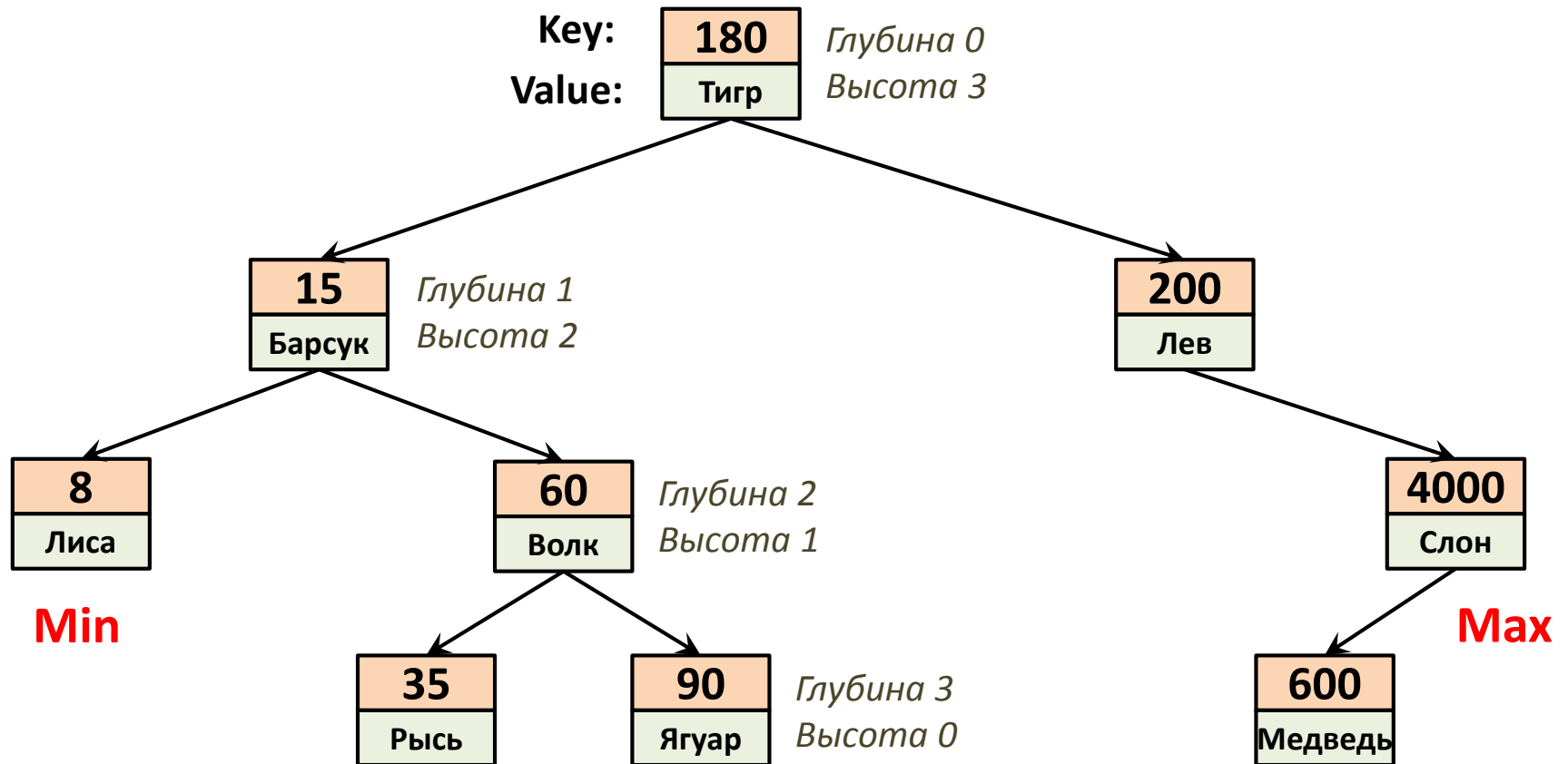
- **Двоичное дерево поиска (binary search tree, BST)** – это двоичное дерево, в котором:

- 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

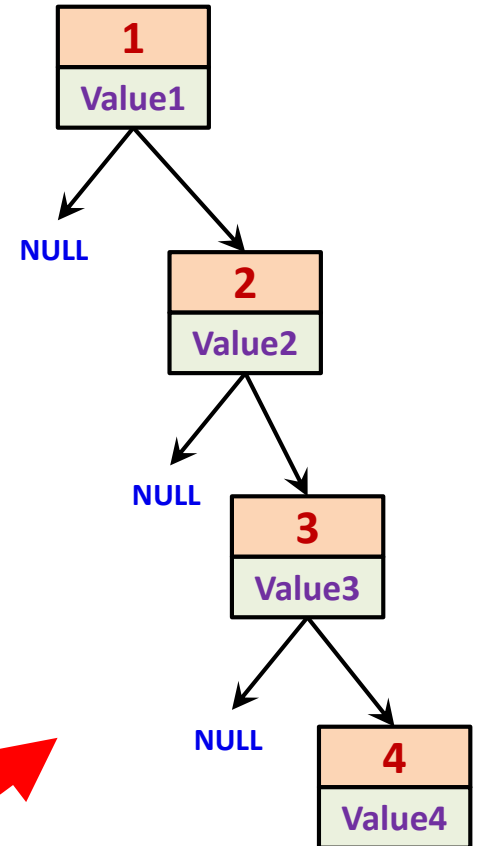
1. Операции над BST имеют трудоемкость пропорциональную высоте h дерева
2. В среднем случае высота дерева $O(\log n)$
3. В худшем случае элементы добавляются по возрастанию (убыванию) ключей – дерево вырождается в список длины $O(n)$

`bstree_add(1, value1)`

`bstree_add(2, value2)`

`bstree_add(3, value3)`

`bstree_add(4, value4)`



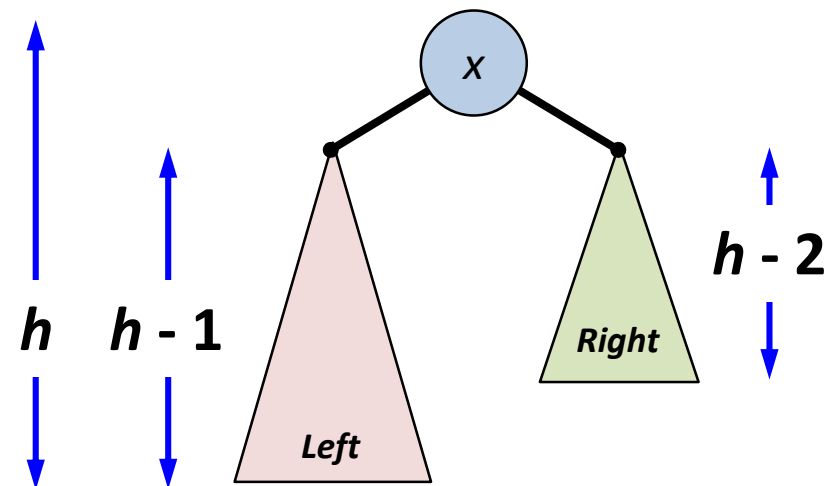
Сбалансированные деревья поиска

- **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
 - ☐ ...

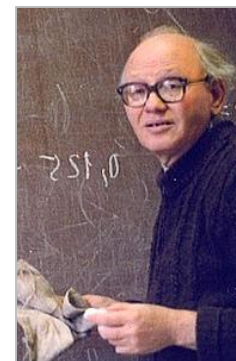
AVL-деревья

- **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

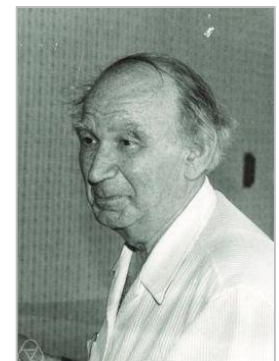
- GNU libavl
- libdict
- Python avllib
- avlmap



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



Адельсон-Вельский Г.М.



Ландис Е.М.

AVL-деревья

- Georgy Adelson-Velsky, G., Evgenii Landis. **An algorithm for the organization of information //** Proc. of the USSR Academy of Sciences 146, P. 263–266. English transl. by Myron J. Ricci

AN ALGORITHM FOR THE ORGANIZATION OF INFORMATION

G. M. ADEL'SON-VEL'SKIĬ AND E. M. LANDIS

In the present article we discuss the organization of information contained in the cells of an automatic calculating machine. A three-address machine will be used for this study.

Statement of the problem. The information enters a machine in sequence from a certain reserve. The information element is contained in a group of cells which are arranged one after the other. A certain number (the information estimate), which is different for different elements, is contained in the information element. The information must be organized in the memory of the machine in such a way that at any moment a very large number of operations is not required to scan the information with the given evaluation and to record the new information element.

An algorithm is proposed in which both the search and the recording are carried out in $O \lg N$ operations, where N is the number of information elements which have entered at a given moment.

A part of the memory of the machine is set aside to store the incoming information. The information elements are arranged there in their order of entry. Moreover, in another part of the memory a "reference board" [1] is formed, each cell of which corresponds to one of the information elements.

The reference board is a dyadic tree (Figure 1a): each of its cells has no more than one left cell, and no more than one right cell subordinated to it. Direct subordination induces subordination (partial ordering). In addition, for each cell of the tree, all the cells which are subordinate to a left (right) directly subordinate cell, will be arranged further to the left (right) than the given cell. Moreover, we assume that there is a cell (the head) to which all the others are subordinate. By transitivity, the conception "further to the left" and "further to the right" extends to the aggregate of all the cell pairs, and this aggregate becomes ordered. Thus, a given order of cells in a reference board should coincide with the order of arrangement of the estimates of the corresponding information elements (to be specific, we shall consider the estimates as increasing from left to right).

In the first address of each cell of the reference board, a place is indicated where the corresponding information element is located. The addresses of the cells of the reference board, which are directly subordinate on the left and right respectively to the given cell, are located in the second and third addresses. If a cell has no directly subordinate cells on either side, then there is zero in the corresponding address. The head address is stored in a certain fixed cell h .

Let us call the sequence of the cells of the tree a *chain* in which each previous cell is directly

subordinate to the following. For each cell of the tree, we shall designate as the length of the left (right) branch the maximum length of the chain which consists of cells subordinate to the given one and located more to the left (right) than the given cell. Any chain whose length is equal to the length of the branch is called the branch pivot.

The *admissible tree* will be such that for each of its cells, the length of the left branch differs from the length of the right branch by no more than unity (Figure 1b).

Two orders for the information on the branch lengths are distinguished in each cell of the address board. If the left branch is longer than the right, then 1.0 stands; if the right branch is longer than the left, then 0.1 stands; and if they are equal, then 0.0. Moreover, it is considered that if there are no subordinate cells on either side, then the length of the branch on this side is equal to zero.



Figure 1

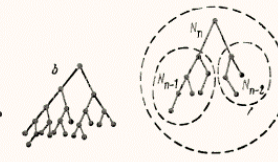


Figure 2

The recording algorithm is such that at each moment, the reference board is an admissible tree.

Lemma 1. Let the number of cells of the admissible tree be equal to N . Then the maximum length of the branch is not greater than $(3/2) \log_2 (N + 1)$.

Proof. Let us denote by N_n the minimum number of cells in the admissible tree when the given maximum length of the branch is n . Then it can be easily proven (see Figure 2) that $N_n = N_{n-1} + N_{n-2} + 1$.

When we solve this equation in finite remainders, we get

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

Whence

$$n < \log_{\frac{1+\sqrt{5}}{2}} (N + 1) < \frac{3}{2} \log_2 (N + 1),$$

q.e.d.

The search algorithm of the information element with the given estimate consists in the following. We compare the given estimate with the estimate of the information element which corresponds to the head. Depending on the result of the comparison, we now compare the given estimate with the estimate of the information element which corresponds to the left or right directly subordinate cell head. Let k comparison steps be made of the given estimate m with the estimate m_u of the information element which corresponds to a certain cell u of the reference board. If $m < m_u$ ($m > m_u$), then at step number $(k + 1)$, a comparison is made of the estimate m with the estimate of the information element corresponding to the cell directly subordinate to u on the left (right). If $m = m_u$, the search is complete.

AVL tree

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): $O(n)$

AVL-деревья

- **Основная идея**

Если вставка или удаление элемента приводит к нарушению сбалансированности дерева, то необходимо выполнить его балансировку

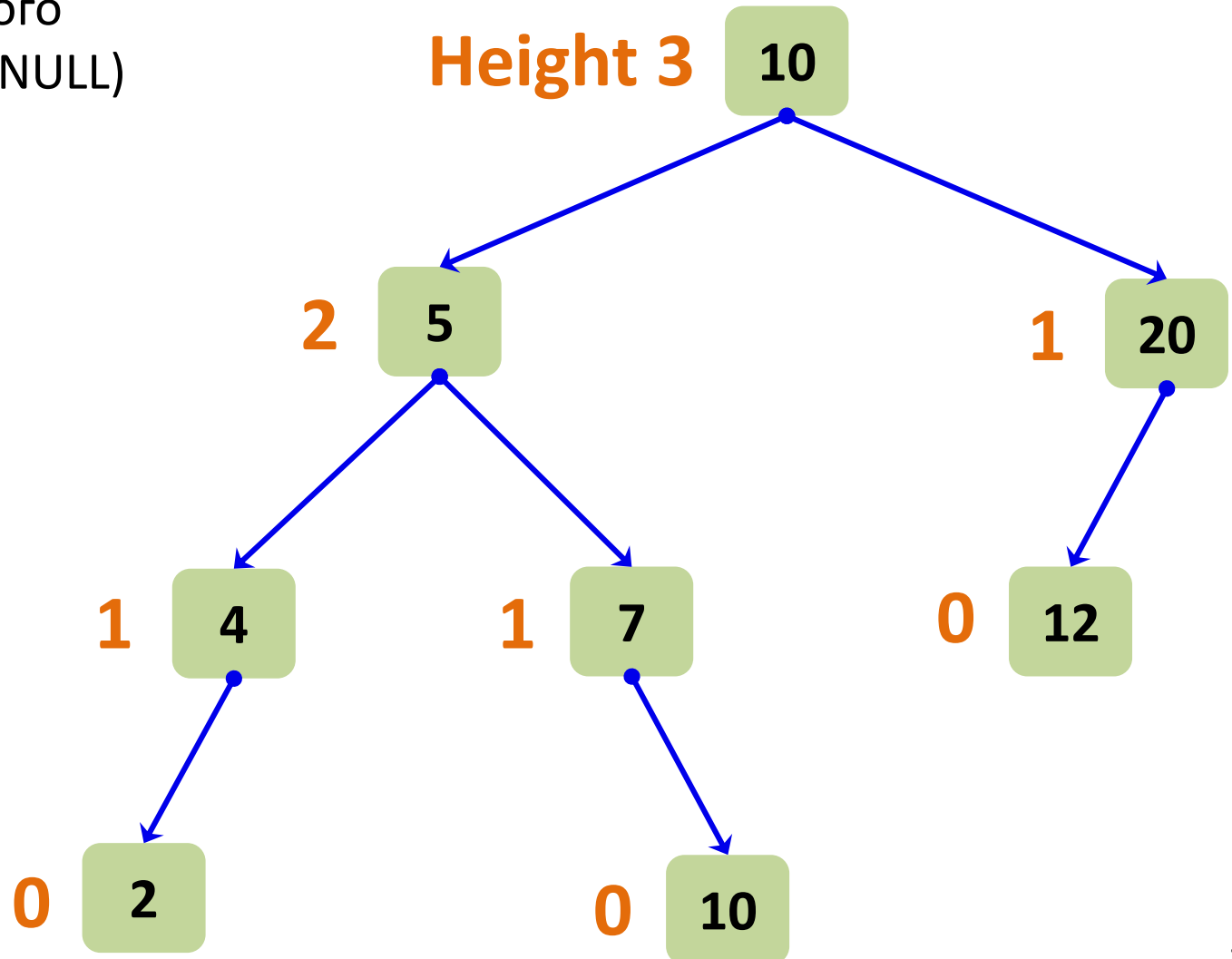
- **Коэффициент сбалансированности узла (balance factor)** – это разность высот его левого и правого поддеревьев

- В AVL-дереве коэффициент сбалансированности любого узла принимает значения из множества $\{-1, 0, 1\}$

- **Высота узла (height)** – это длина наибольшего пути от него до дочернего узла, являющегося листом

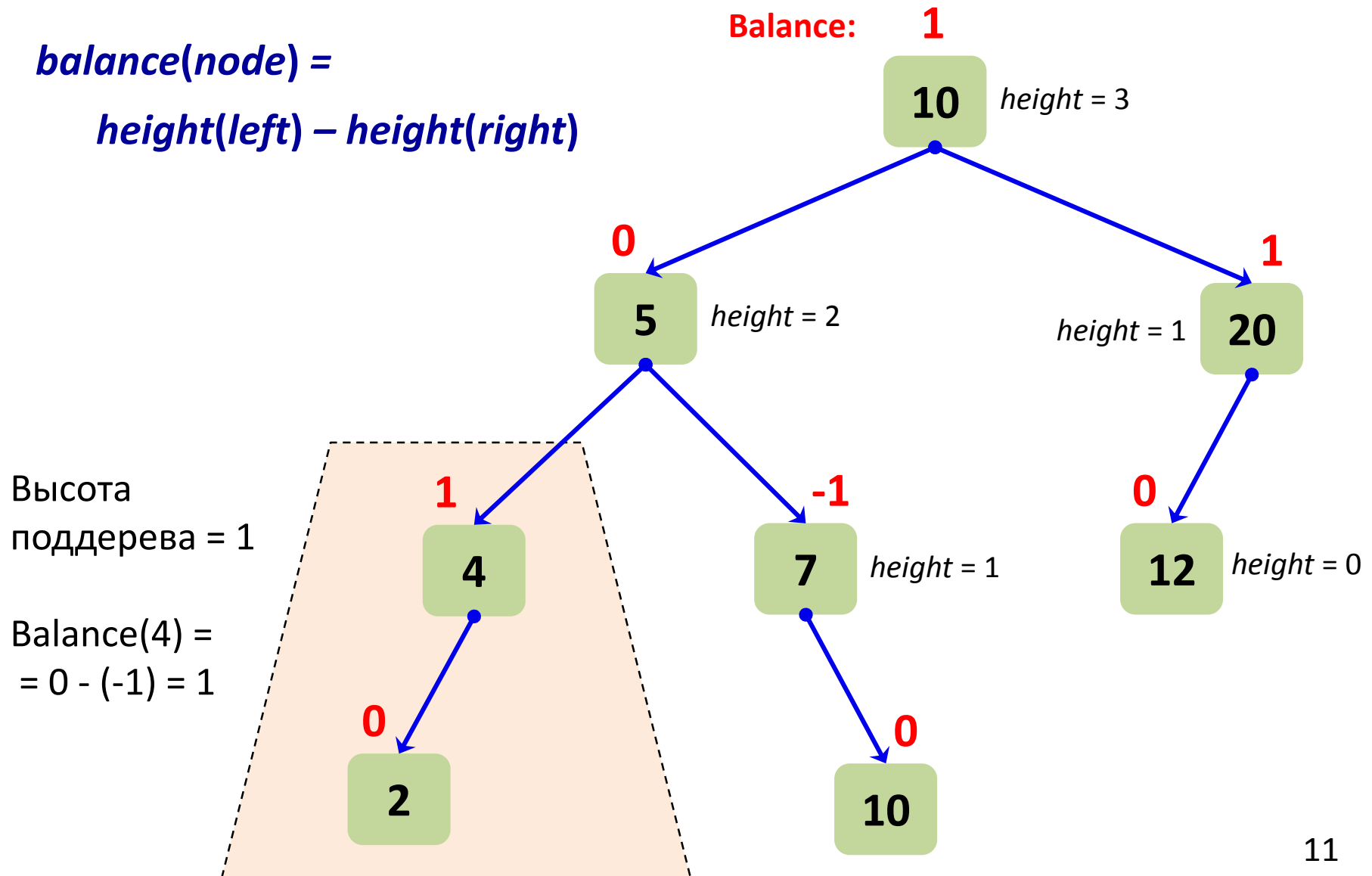
Высота узла (node height)

Высота пустого
поддерева (NULL)
равна -1



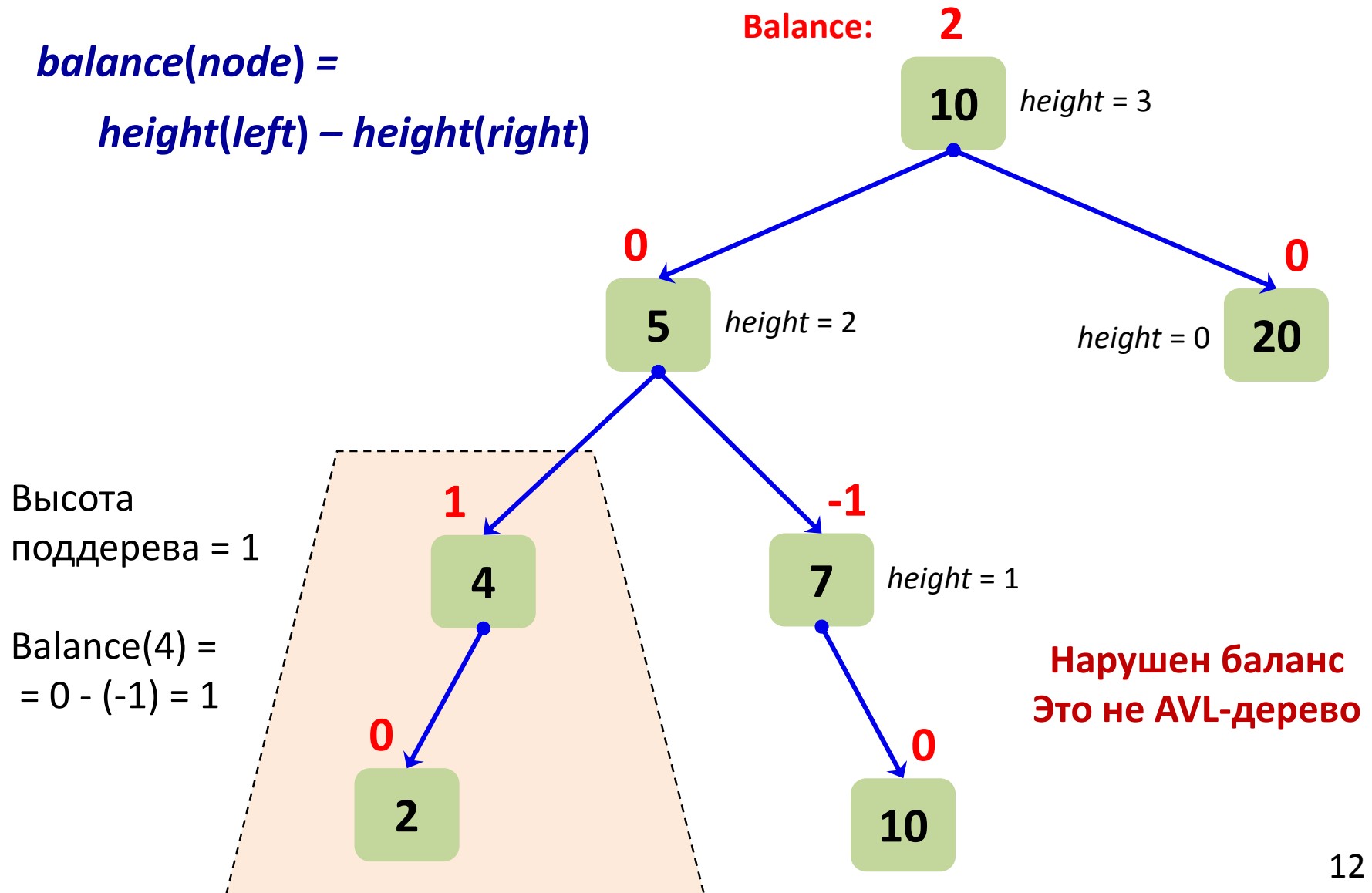
Коэффициент сбалансированности

$$\text{balance}(\text{node}) = \text{height}(\text{left}) - \text{height}(\text{right})$$



Коэффициент сбалансированности

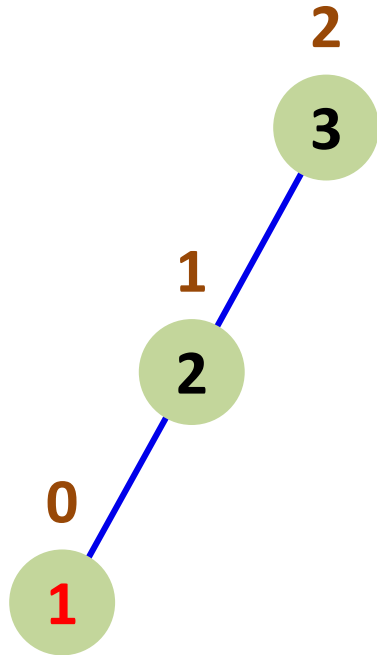
$$\text{balance}(\text{node}) = \text{height}(\text{left}) - \text{height}(\text{right})$$



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)
- Повороты:
 - 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)

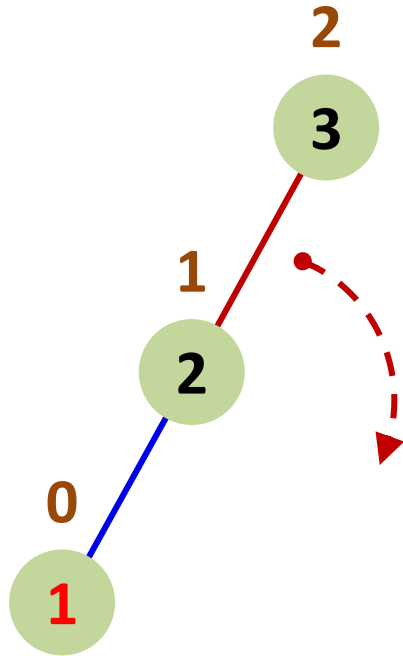
Правый поворот (R-rotation)



Left Left Case

- В левое поддереву узла 3 добавили элемент **1**
- Дерево с корнем в узле 3 не сбалансированно
 $H(Left) = 1 > H(Right) = -1$
- Необходимо увеличить высоту правого поддерева

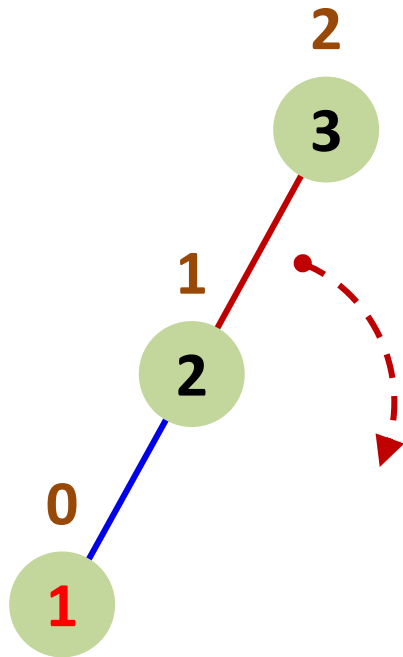
Правый поворот (R-rotation)



Left Left Case

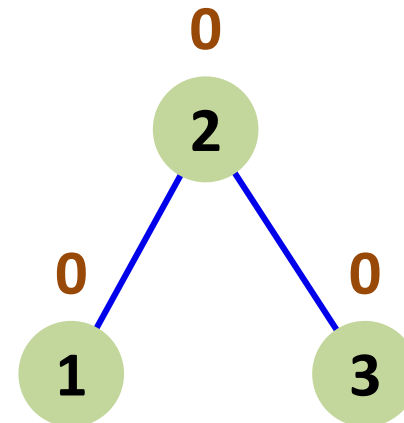
- Поворачиваем ребро, связывающее *корень* и его *левый* дочерний узел, *вправо*

Правый поворот (R-rotation)



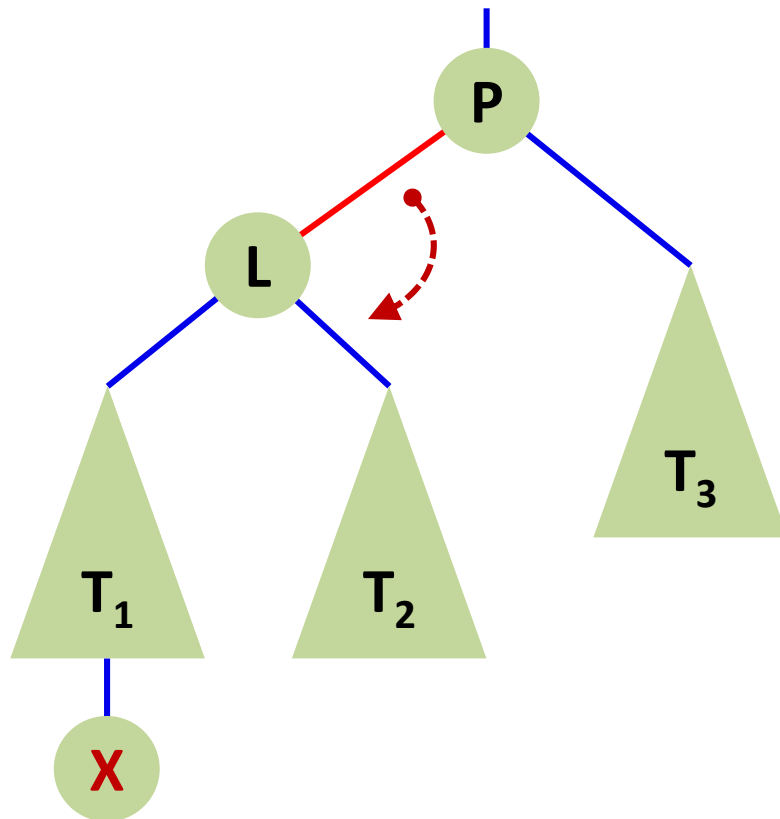
Left Left Case

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

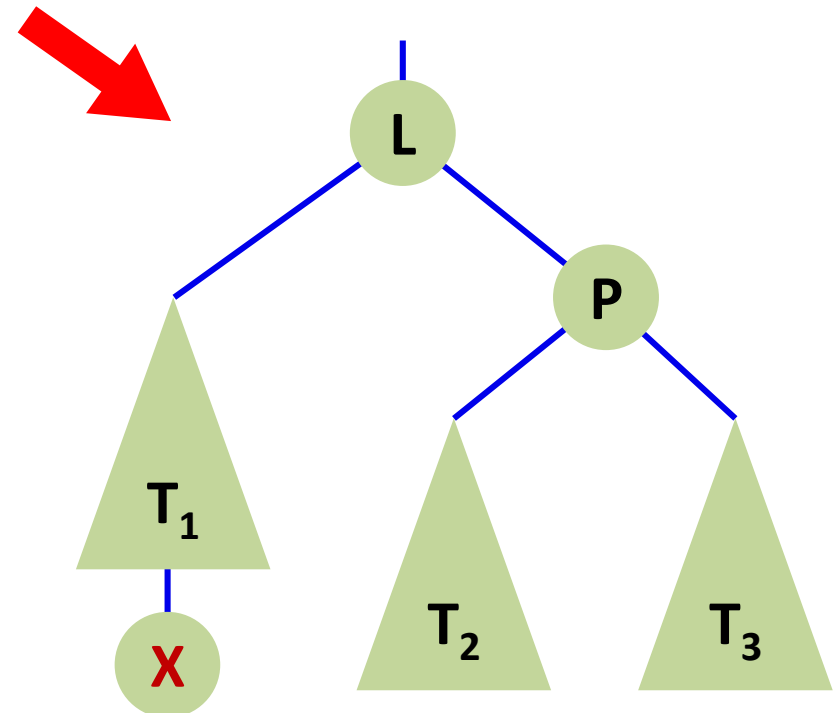


Дерево
сбалансированно

Правый поворот (R-rotation)

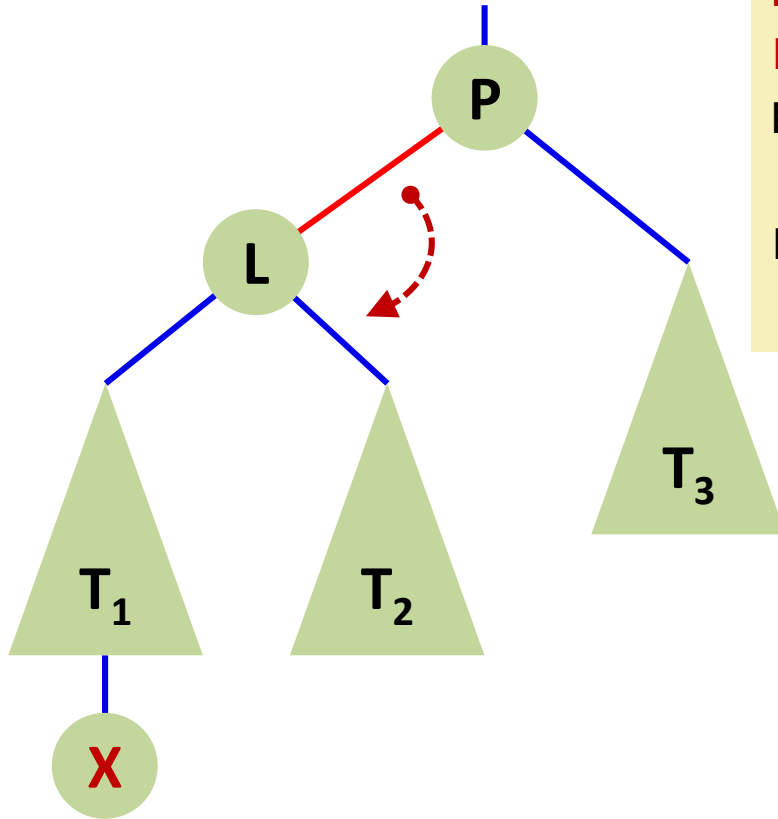


Правый поворот
в общем случае



- В левое поддерево вставлен элемент X
- Дерево не сбалансированно $H(\text{Left}) > H(\text{Right})$

Правый поворот (R-rotation)

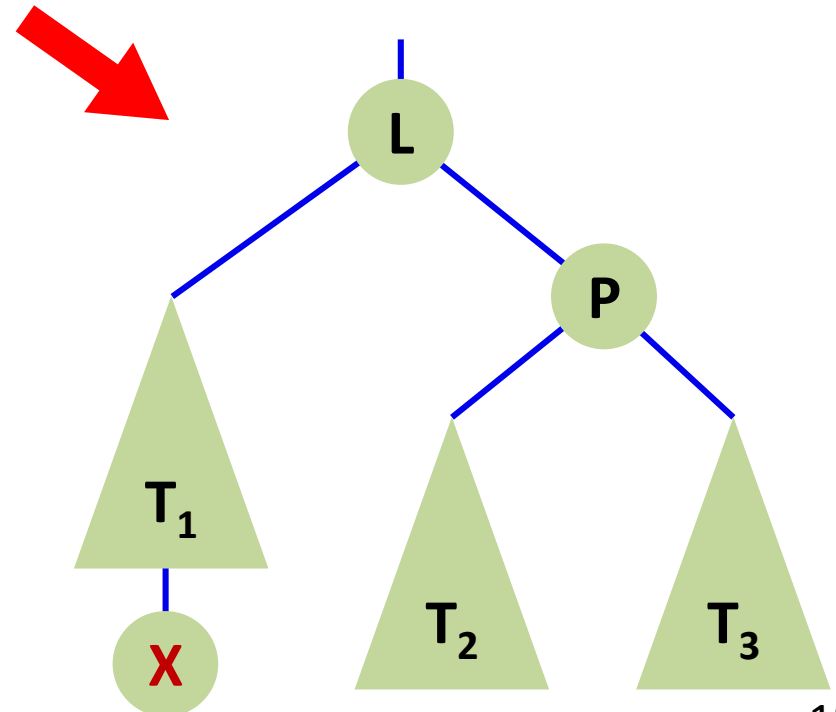


$P.left = L.right$

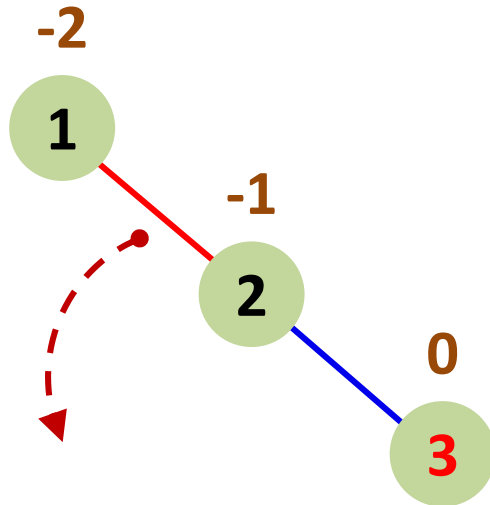
$L.right = P$

$P.height = 1 + \max(P.left.height, P.right.height)$

$L.height = 1 + \max(L.left.height, P.height)$



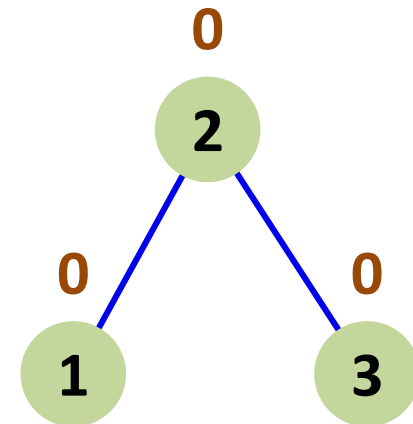
Левый поворот (L-rotation)



Right Right Case

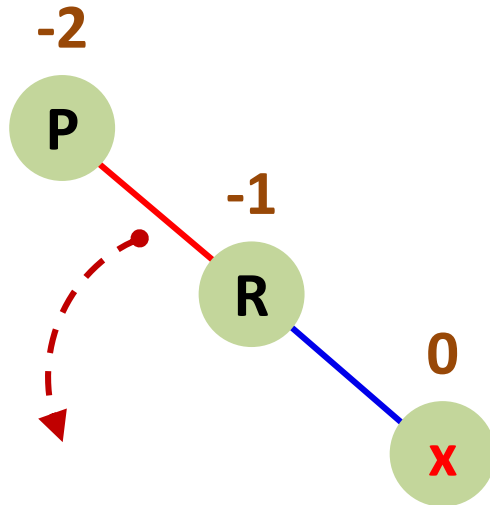


- В правое поддереву узла 1 вставлен элемент **3**
- Поворачиваем ребро, связывающее *корень* и его *правый* дочерний узел, влево



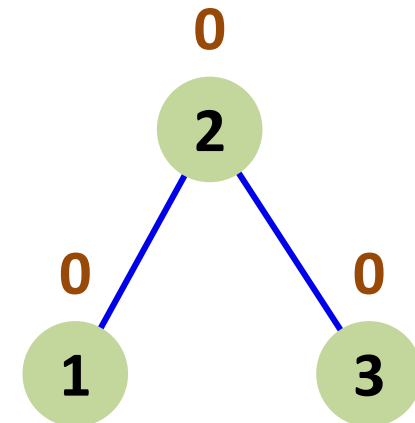
Дерево
сбалансированно

Левый поворот (L-rotation)



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

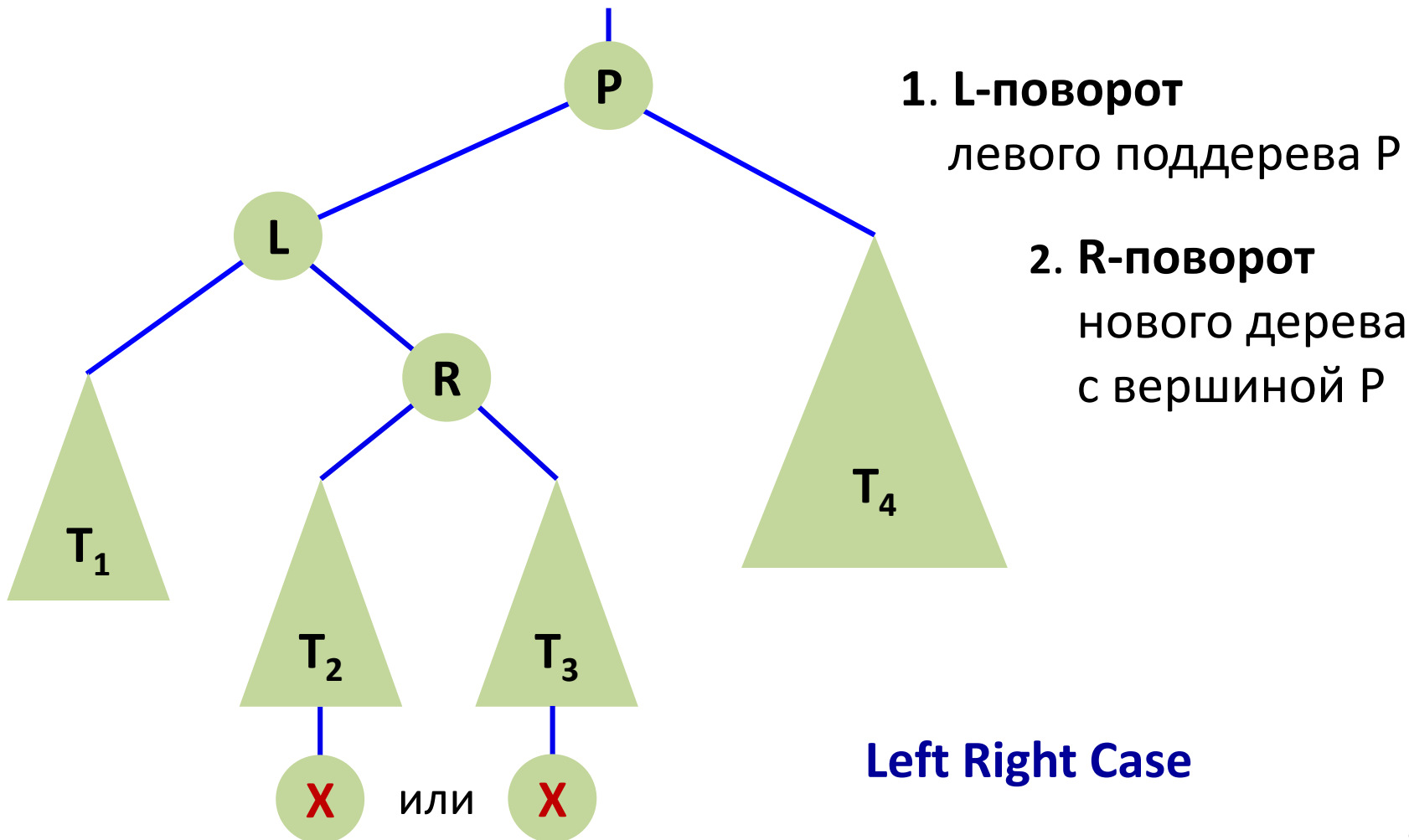
Right Right Case



Дерево
сбалансированно

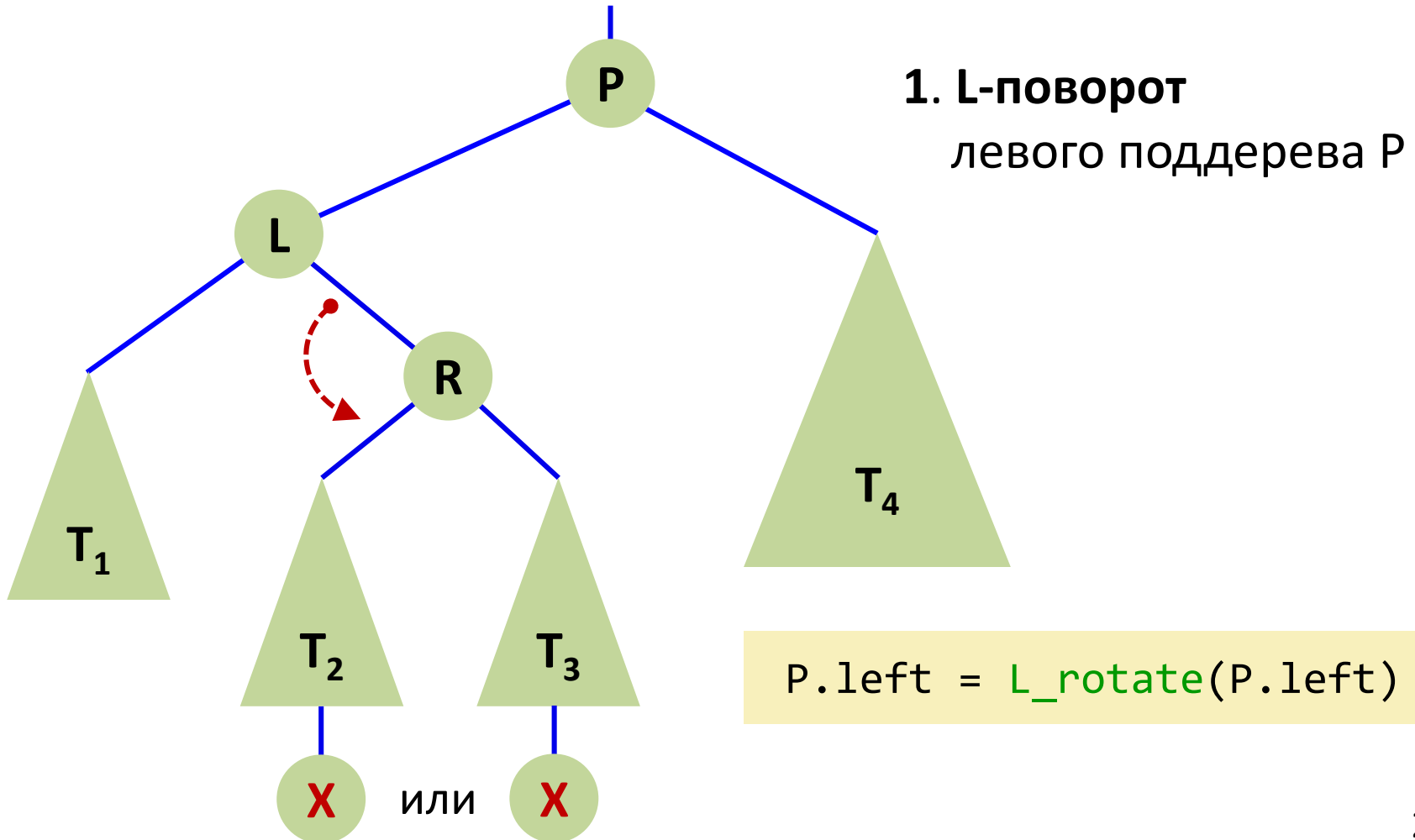
Двойной лево-правый поворот (LR-rotation)

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



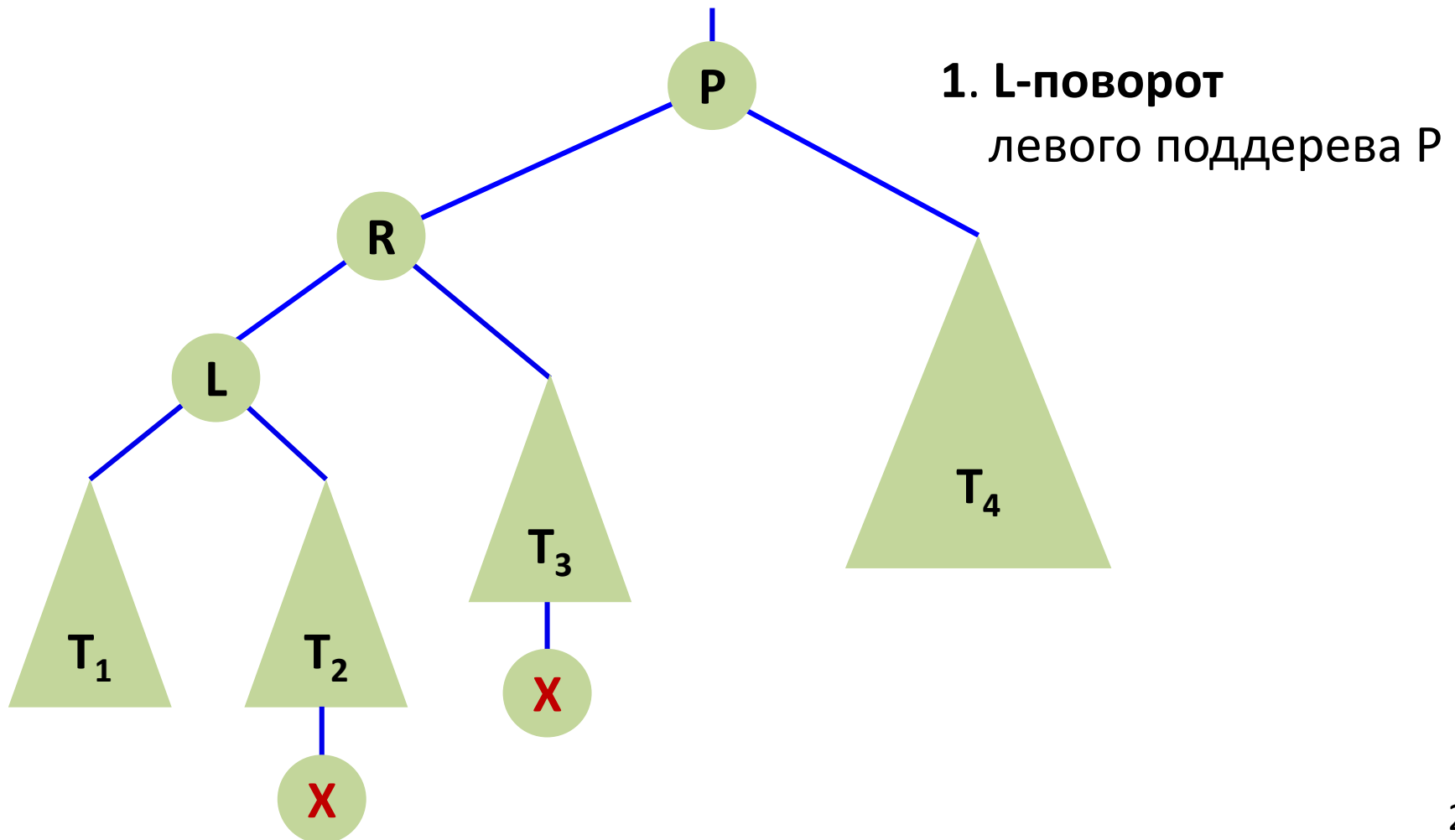
Двойной лево-правый поворот (LR-rotation)

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



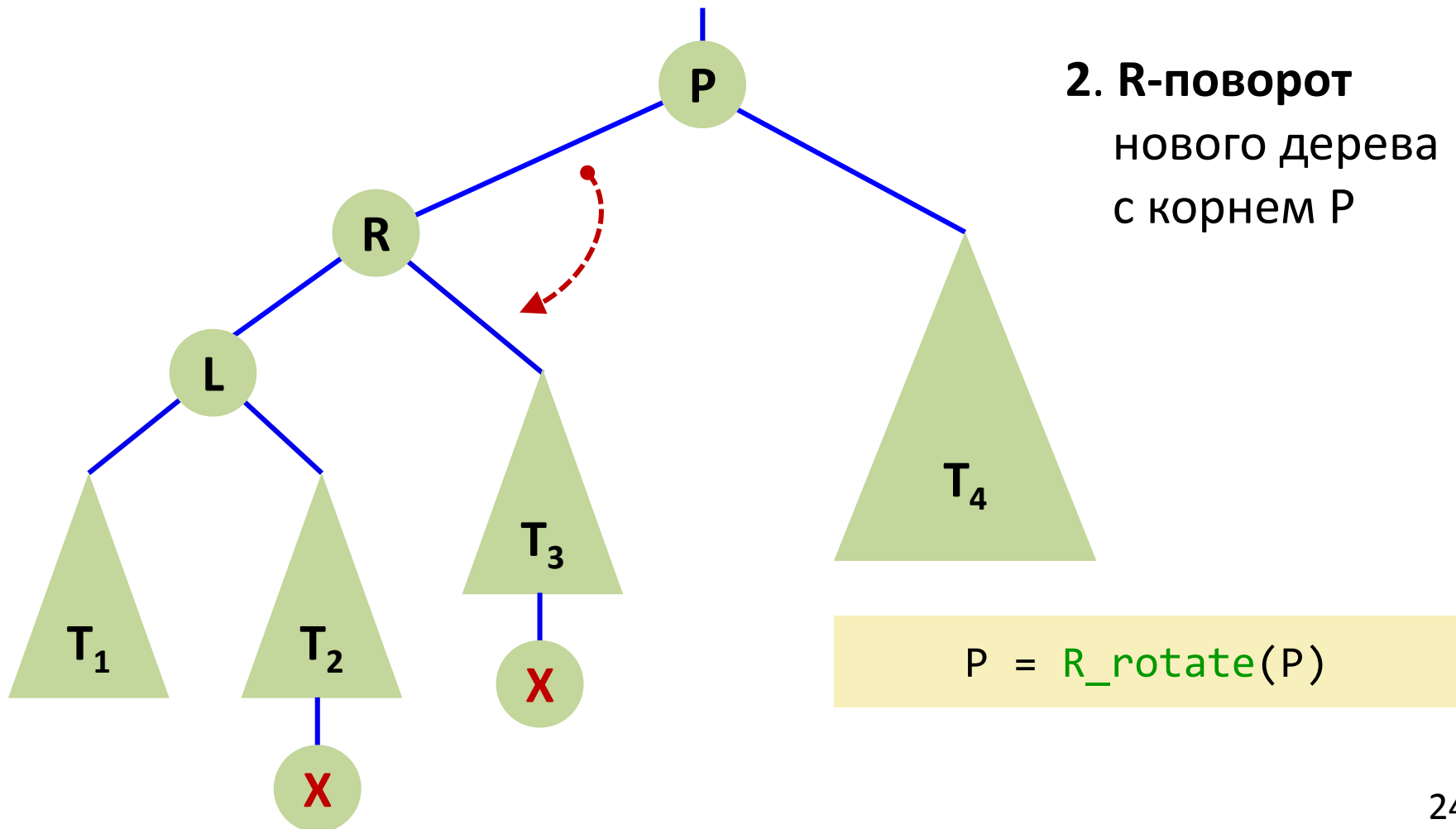
Двойной лево-правый поворот (LR-rotation)

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



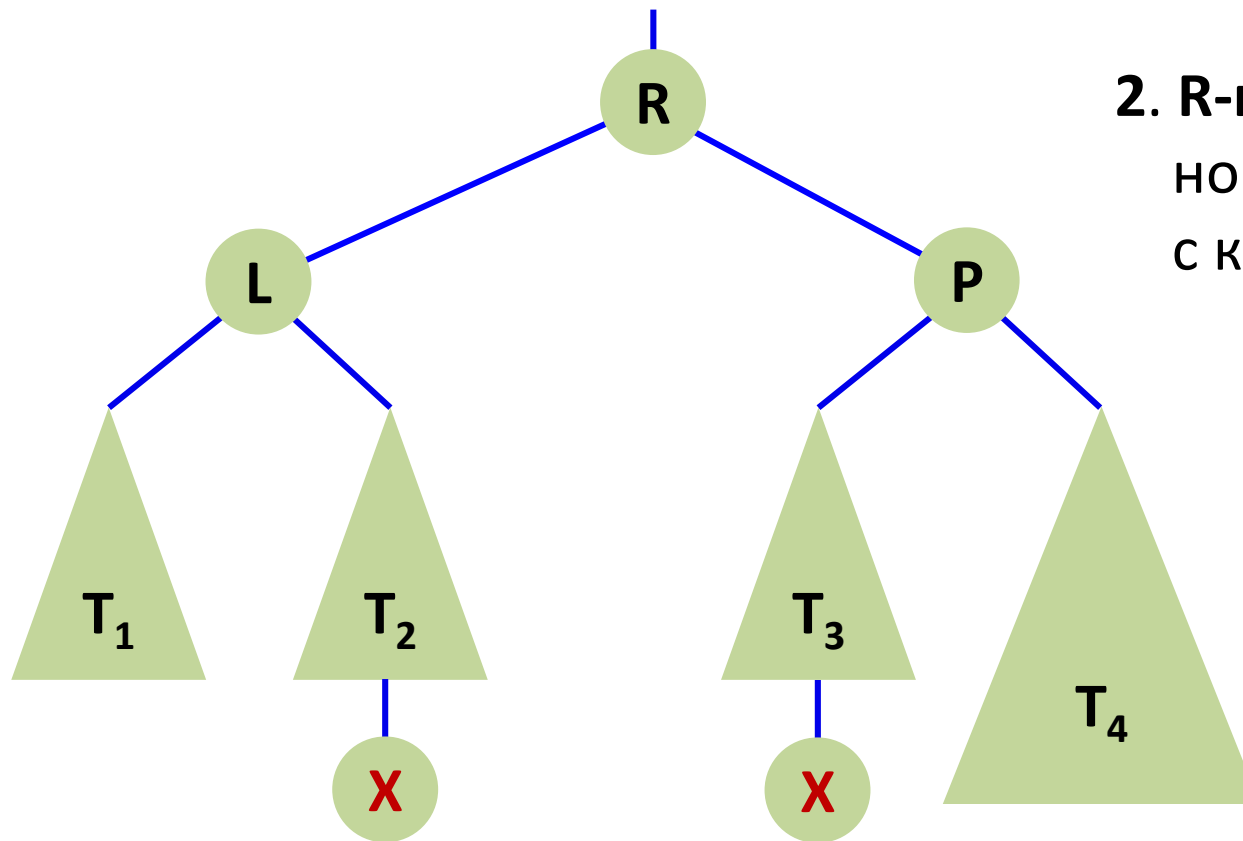
Двойной лево-правый поворот (LR-rotation)

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



Двойной лево-правый поворот (LR-rotation)

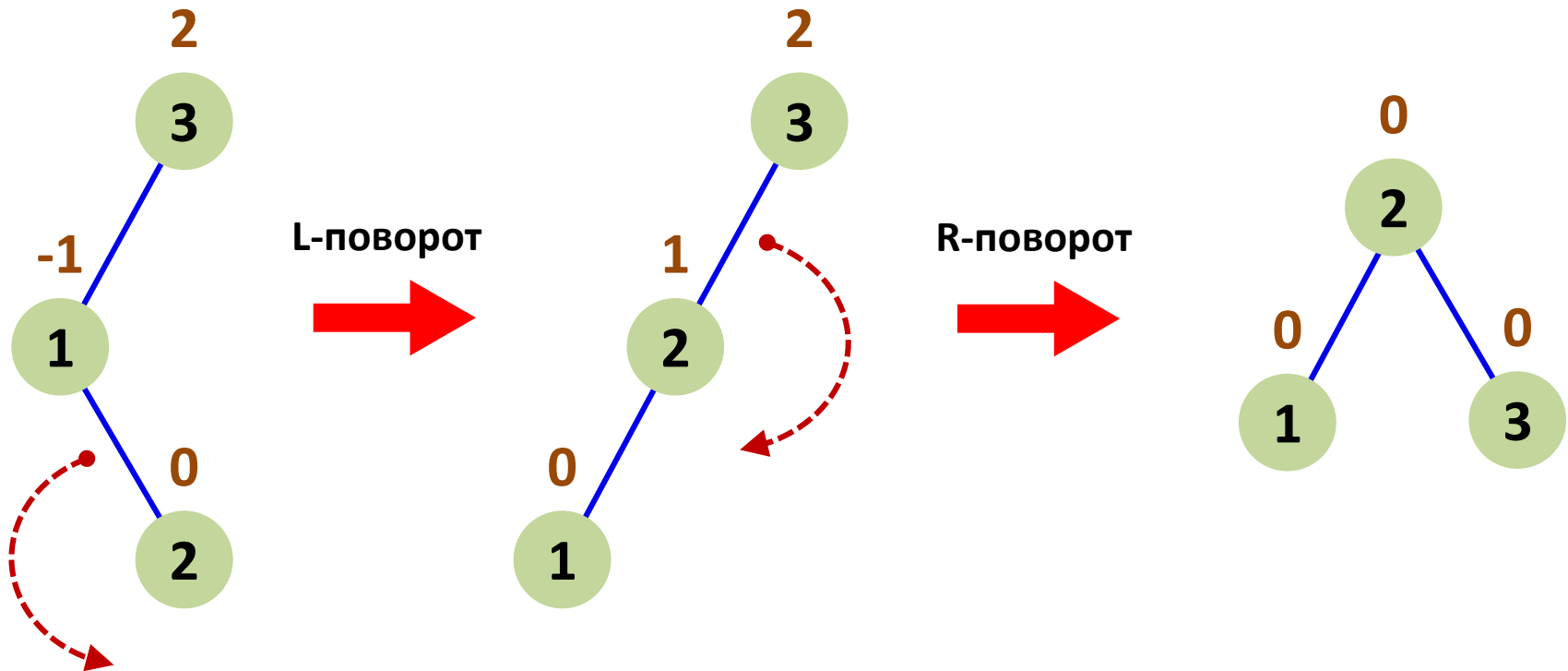
- **LR-поворот** выполняется после добавления элемента в правое поддереву левого дочернего узла дерева



2. R-поворот
нового дерева
с корнем Р

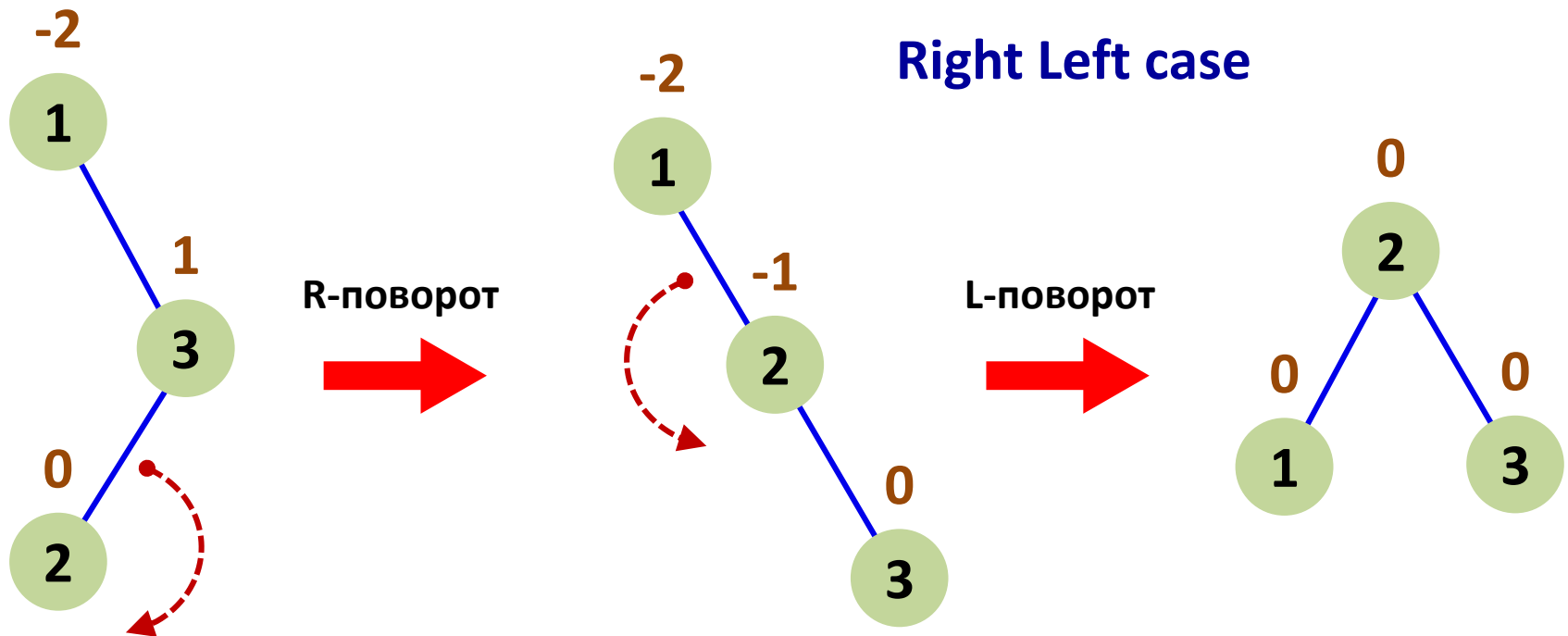
Двойной лево-правый поворот (LR-rotation)

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



Двойной право-левый поворот (RL-rotation)

- **RL-поворот** выполняется после добавления элемента в левое поддереву правого дочерного узла дерева



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

Повороты в AVL-дереве

- Вычислительная сложность любого поворота $O(1)$
- Любой поворот сохраняет свойства бинарного дерева поиска (распределение ключей по левыми и правым поддеревьям)

Анализ эффективности AVL-деревьев

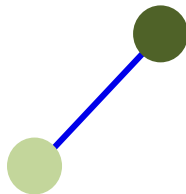
- Оценим сверху высоту h AVL-дерева, содержащего N внутренних узлов (узлов, имеющих дочерние вершины)
- Обозначим через $N(h)$ минимальное количество внутренних узлов необходимых для формирования AVL-дерева высоты h

$$N(0) = 0$$



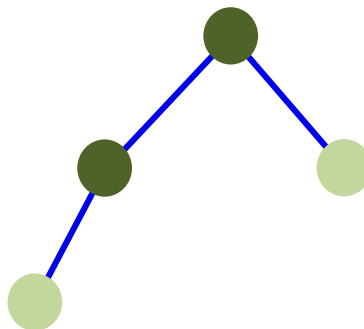
$$h = 0$$

$$N(1) = 1$$



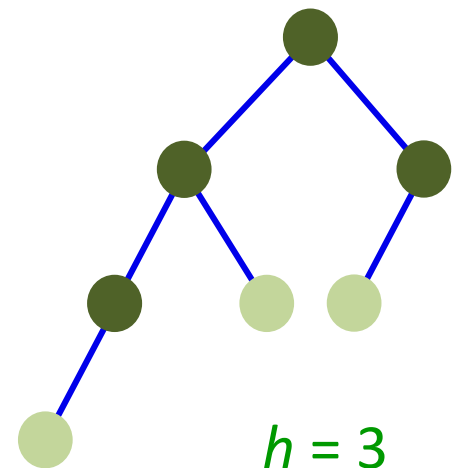
$$h = 1$$

$$N(2) = 2$$



$$h = 2$$

$$N(3) = 4$$



$$h = 3$$

AVL-деревья различной высоты

Анализ эффективности AVL-деревьев

- Значения $N(h)$: 0, 1, 2, 4, 7, 12, 20, 33, 54, ...

- Видно, что

$$N(h) = N(h - 1) + N(h - 2) + 1, \text{ для } h = 2, 3, \dots$$

- Значения последовательности $N(h)$ можно выразить через значения последовательности Фибоначчи

$$F_n = F_{n-1} + F_{n-2}, \quad F_0 = 0, F_1 = 1$$

- Значения F_n : 0, 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ...
- Значения $N(h)$: 0, 1, 2, 4, 7, 12, 20, 33, 54, ...

$$N(h) = F(h + 2) - 1, \text{ для } h \geq 0$$

Анализ эффективности AVL-деревьев

- Выразим из $N(h) = F(h + 2) - 1$ значение высоты h AVL-дерева, состоящего из $N(h)$ внутренних узлов
- По формуле Бине можно найти приближенное значение n -го члена последовательности Фибоначчи

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

$$\varphi = \frac{1 + \sqrt{5}}{2} \text{ — отношение золотого сечения (gold ratio)}$$

$$F_n = \left\lceil \frac{\varphi^n}{\sqrt{5}} \right\rceil, \quad n \geq 0.$$

Анализ эффективности AVL-деревьев

$$N(h) = \left\lceil \frac{\varphi^{h+2}}{\sqrt{5}} \right\rceil - 1 > \frac{\varphi^{h+2}}{\sqrt{5}} - 2$$

$$\sqrt{5}(N(h) + 2) > \varphi^{h+2}$$

$$\log_{\varphi}(\sqrt{5}(N(h) + 2)) > h + 2$$

$$h < \log_{\varphi} \sqrt{5}(N(h) + 2) - 2$$

$$h < \log_{\varphi} \sqrt{5} + \log_{\varphi}(N(h) + 2) - 2$$

$$h < \frac{\lg \sqrt{5}}{\lg \varphi} + \frac{1}{\log_2 \varphi} \log_2(N(h) + 2) - 2$$

$$h < \frac{\lg \sqrt{5}}{\lg \varphi} + \frac{\lg 2}{\lg \varphi} \log_2(N(h) + 2) - 2$$

Анализ эффективности AVL-деревьев

$$h < \frac{\lg \sqrt{5}}{\lg \varphi} + \frac{\lg 2}{\lg \varphi} \log_2(N(h) + 2) - 2$$

$$h < 1.6723 + 1.44 \log_2(N(h) + 2) - 2$$

$$***h < 1.4405 \log_2(N(h) + 2) - 0.3277***$$

$$***h = O(\log(n + 2))***$$

Анализ эффективности AVL-деревьев

- Оценка сверху высоты $h(n)$ AVL-дерева [Knuth, Vol. 3], [Wirth89]:

$$\log_2(n + 1) \leq h(n) < 1.4405 \log_2(n + 2) - 0.3277$$

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

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

Реализация

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

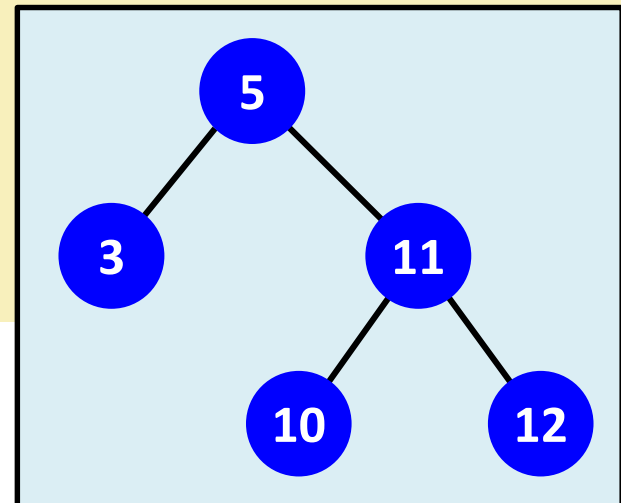
Построение AVL-дерева

```
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);

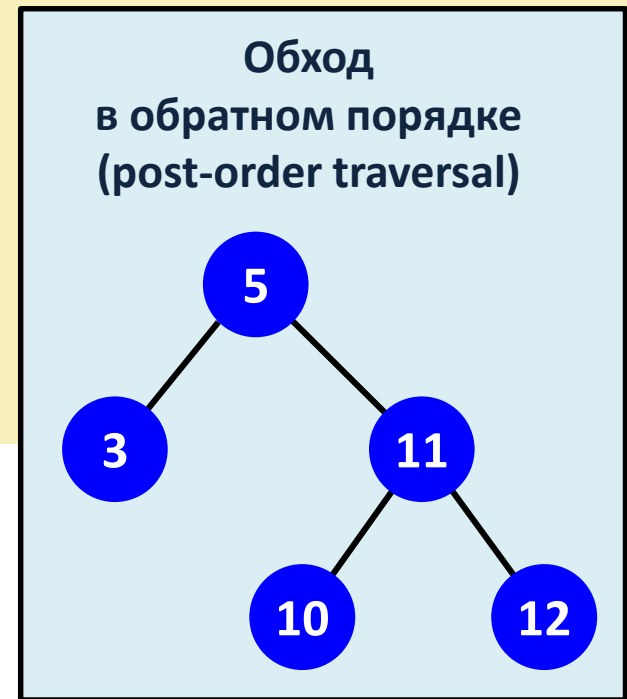
    avltree_free(tree);
    return 0;
}
```

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);
}
```

$$T_{Free} = O(n)$$



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;  
}
```

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

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;  
}
```

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

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);
}
```

- $T_{Height} = O(1)$
- $T_{Balance} = O(1)$

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_rightright_rotate(tree);
        }
    }
}
```

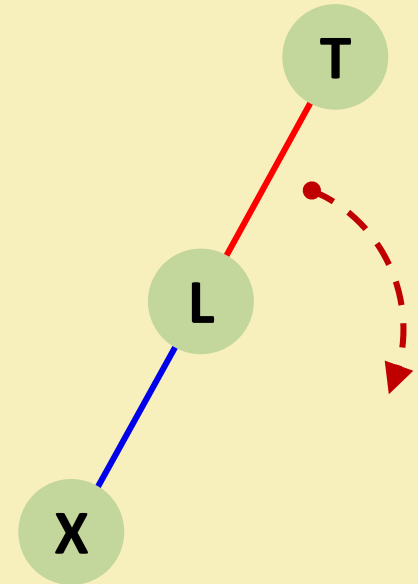
Добавление узла (окончание)

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

- $T_{Add} = O(\log n)$
- Поиск листа для вставки нового элемента выполняется за время $O(\log n)$
- Повороты выполняются за время $O(1)$, их количество не может превышать $O(\log n)$ при подъеме от нового элемента к корню (высота AVL-дерева $O(\log n)$)

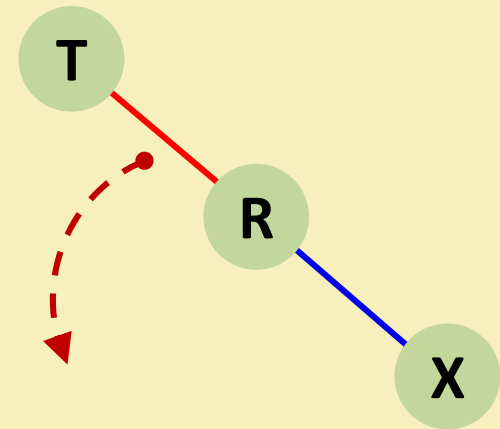
R-поворот (left left case)

```
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_rightright_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;

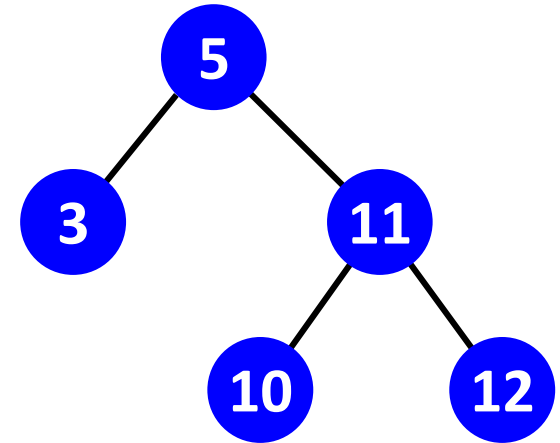
    if (tree == NULL)
        return;

    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

Вирт Н. **Алгоритмы и структуры данных.** – М.: Мир, 1989.
[Глава 4.5, С. 272—286]

Ленивое удаление элементов (lazy deletion)

- С каждым узлом AVL-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 AVL-дерево содержащее все не удалённые узлы (*deleted* = 0)
- Поиск не удалённых элементов и их вставка в новое AVL-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. Кнут Д. **Искусство программирования, Том 3. Сортировка и поиск.** – М.: Вильямс, 2007. – 824 с.
4. AVL-деревья // Сайт RSDN.ru. –
URL: <http://www.rsdn.ru/article/alg/bintree/avl.xml>
5. To Google:
**“avl tree” || “avl tree ext:pdf” ||
“avl tree ext:ppt”**