Лекция 2.
АВЛ-деревья

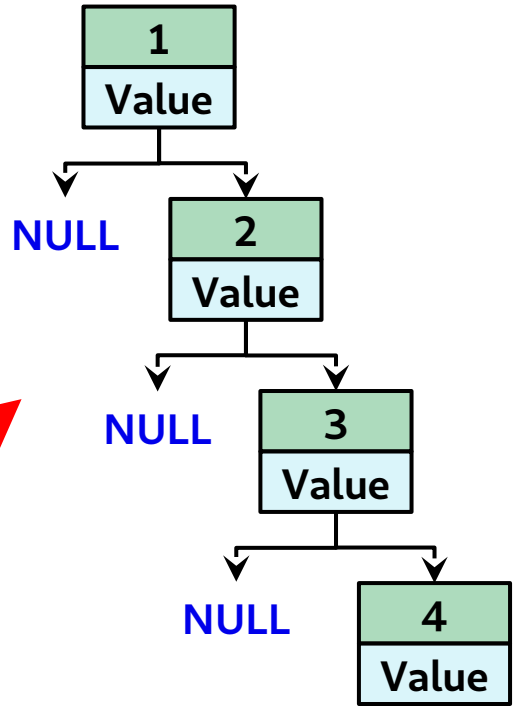
Даниил Михайлович Берлизов
Старший преподаватель Кафедры вычислительных систем СибГУТИ
E-mail: sillyhat34@gmail.com

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

Двоичные деревья поиска

- Операции над двоичным деревом имеют трудоёмкость, пропорциональную высоте ***h*** дерева
- В среднем случае высота дерева ***O*(log*n*)**
- 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

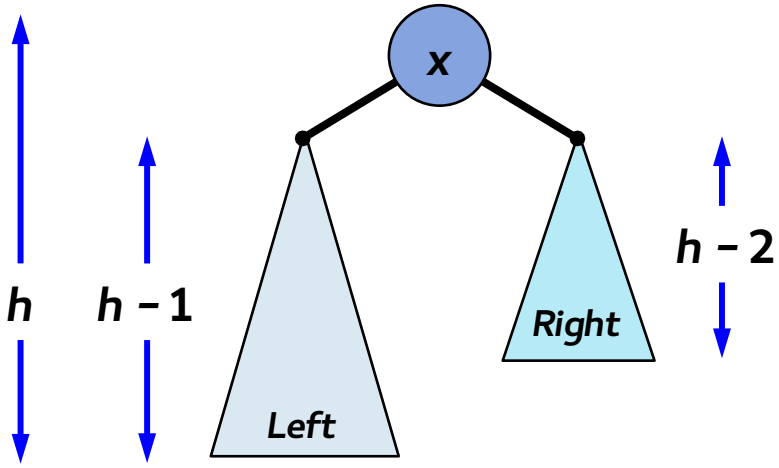
Сбалансированное дерево поиска (*self-balancing binary search tree*) — дерево поиска,
в котором высота поддеревьев любого узла различается не более, чем на заданную константу *k*

Сбалансированные деревья поиска:

- Красно-чёрные деревья (*red-black trees*)
- **АВЛ-деревья (*AVL trees*)**
- 2-3-деревья (*2-3 trees*)
- B-деревья (*B-trees*)
- AA trees
- ...

АВЛ-деревья

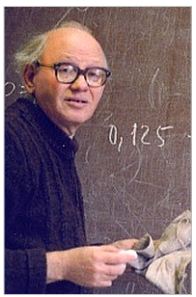
АВЛ-дерево (*AVL tree*) — сбалансированное по высоте двоичное дерево поиска, в котором у любой вершины высота левого и правого поддеревьев различаются не более, чем на 1



Авторы: Г. М. Адельсон-Вельский, Е. М. Ландис

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

Применение: GNU libavl, libdict, Python avlib



Г. М. Адельсон-Вельский

Е. М. Ландис

АВЛ-деревья

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*)

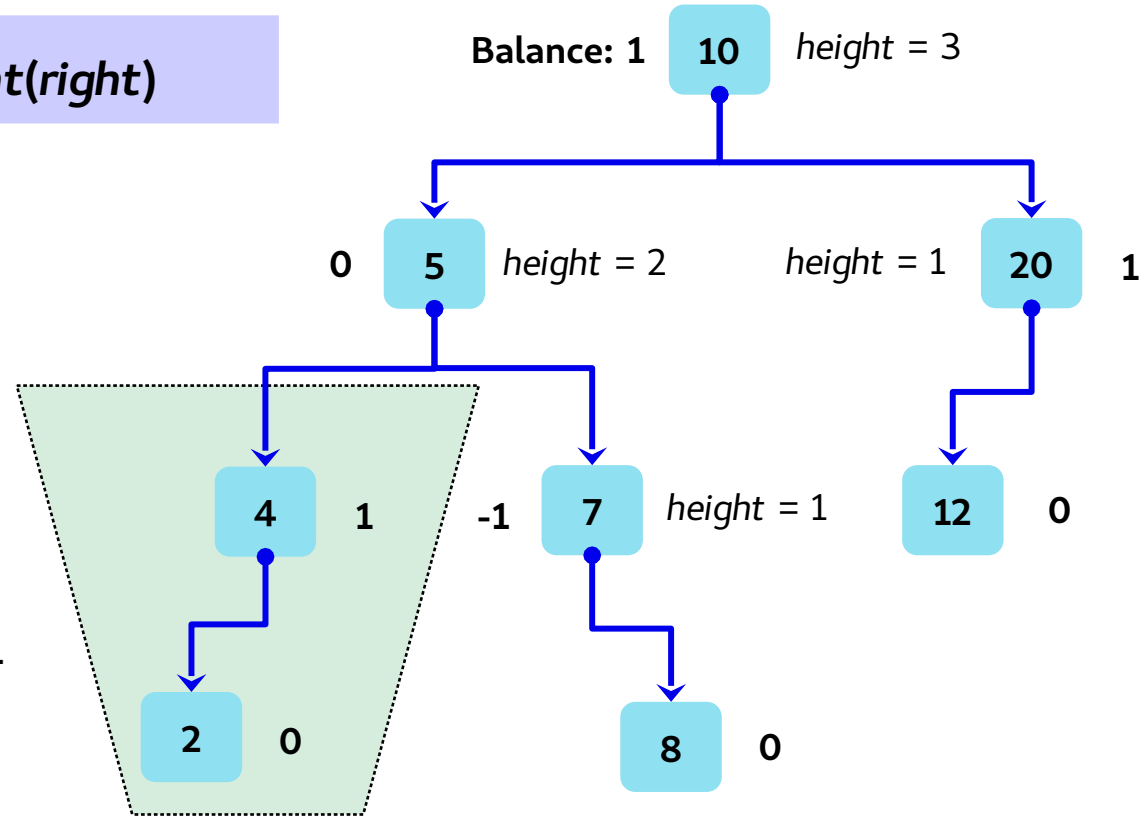
Сложность по памяти (space complexity): *O*(*n*)

АВЛ-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 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, являющегося листом

АВЛ-деревья

balance(node) = height(left) – height(right)



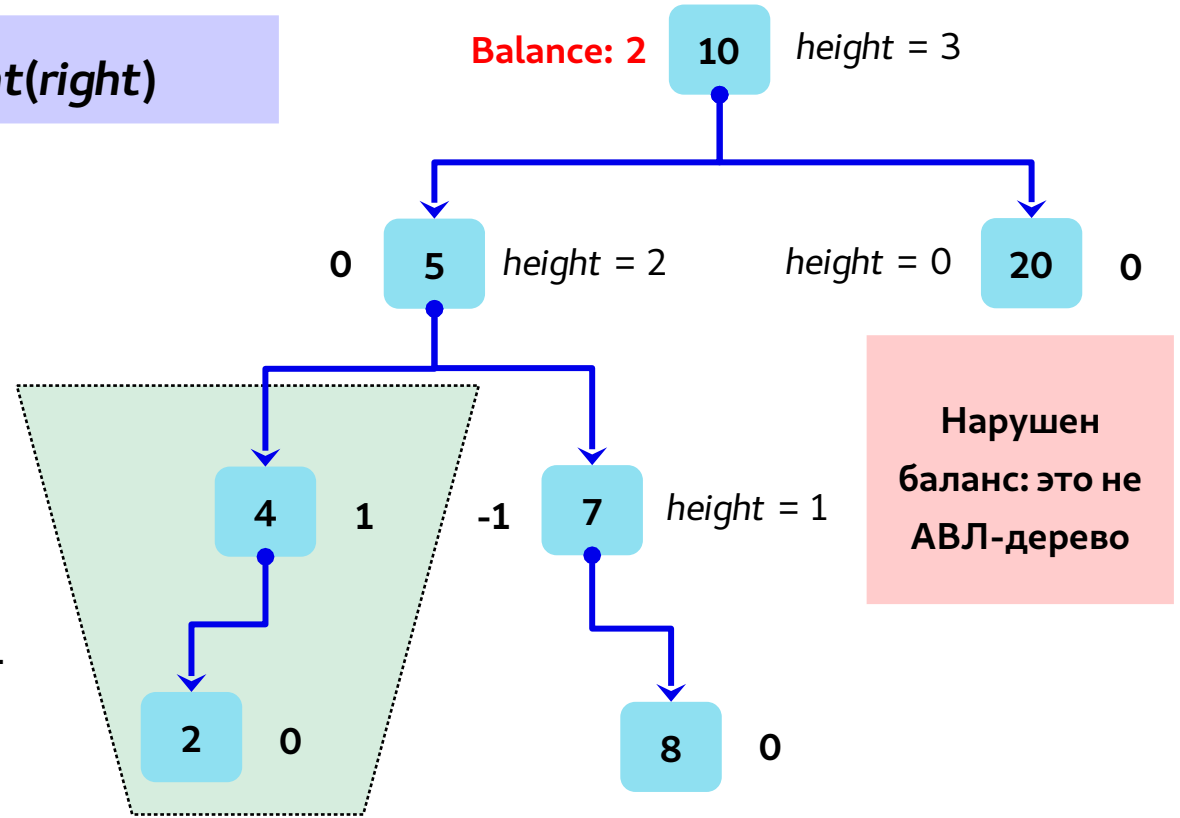
Высота поддерева = 1

***Balance*(4) = 0 – (–1) = 1**

АВЛ-деревья

balance(node) = height(left) – height(right)

AVLTree_Remove(12)



Высота поддерева = 1

***Balance*(4) = 0 – (–1) = 1**

Нарушен баланс: это не АВЛ-дерево

Балансировка АВЛ-дерева

- 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*)

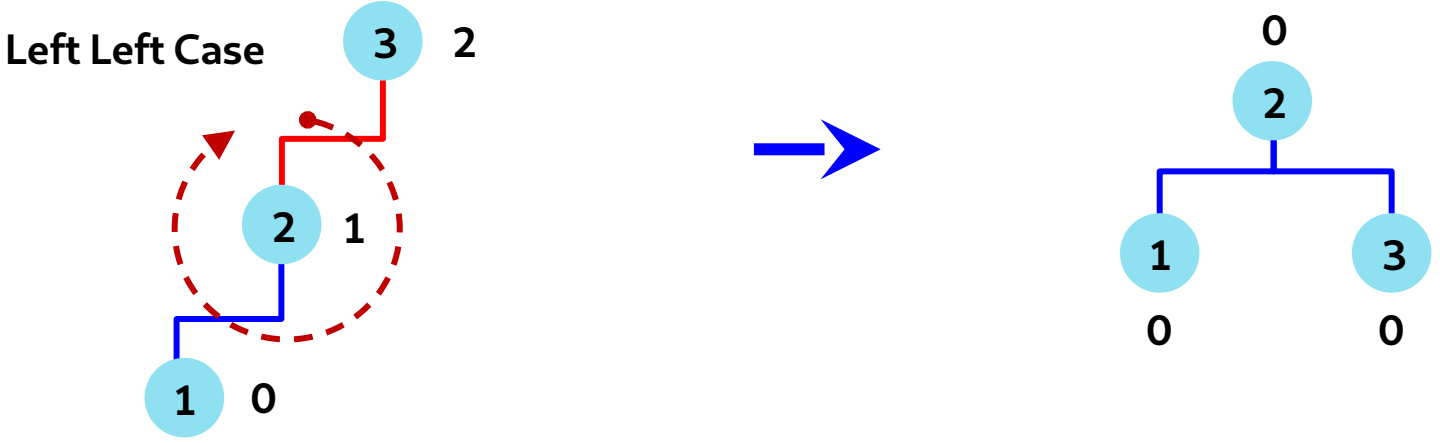
Повороты:

- 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 АВЛ-дерева (*R-rotation*)

- В левое поддерево узла 3 добавили элемент 1
- Дерево с корнем в узле 3 не сбалансировано
- *h*(*left*) = 1 > *h*(*right*) = -1
- Необходимо увеличить высоту правого поддерева

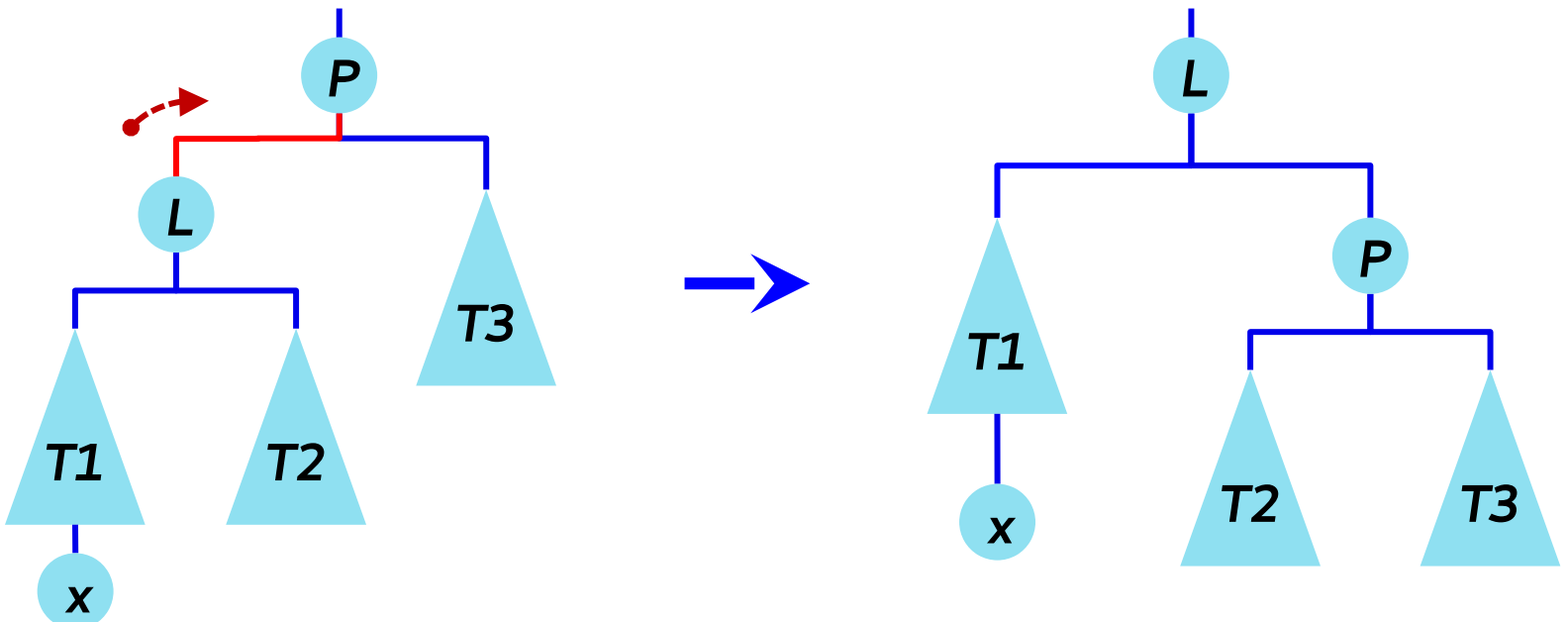
Поворачиваем ребро, связывающее *корень* и его *левый* дочерний узел, *вправо*



Правый поворот АВЛ-дерева (*R-rotation*)

Правый поворот в общем случае:

В левое поддерево вставлен элемент *x*, дерево не сбалансировано: *h*(*left*) > *h*(*right*)



Правый поворот АВЛ-дерева (*R-rotation*)

Правый поворот в общем случае:

В левое поддерево вставлен элемент *x*, дерево не сбалансировано: *h*(*left*) > *h*(*right*)

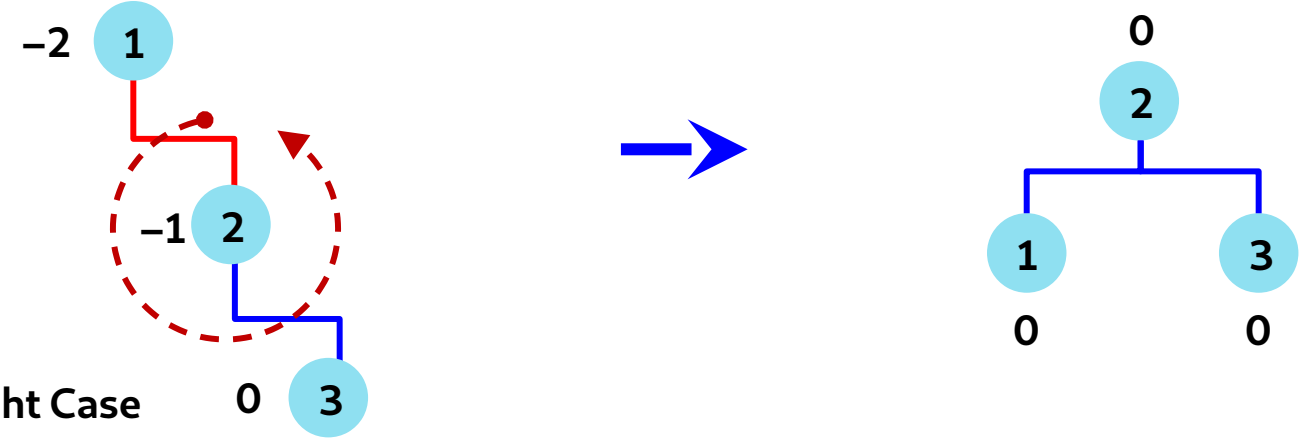
```
function AVLTree_RRotate(P)
    P.left = L.right
    L.right = P
    P.height = 1 + max(P.left.height, P.right.height)
    L.height = 1 + max(L.left.height, P.height)
end function
```

$T_{RightRotate} = O(1)$

Левый поворот АВЛ-дерева (*L-rotation*)

- В правое поддерево узла 1 добавили элемент 3
- Дерево с корнем в узле 1 не сбалансировано
- *h*(*left*) = -1 < *h*(*right*) = 1
- Необходимо увеличить высоту левого поддерева

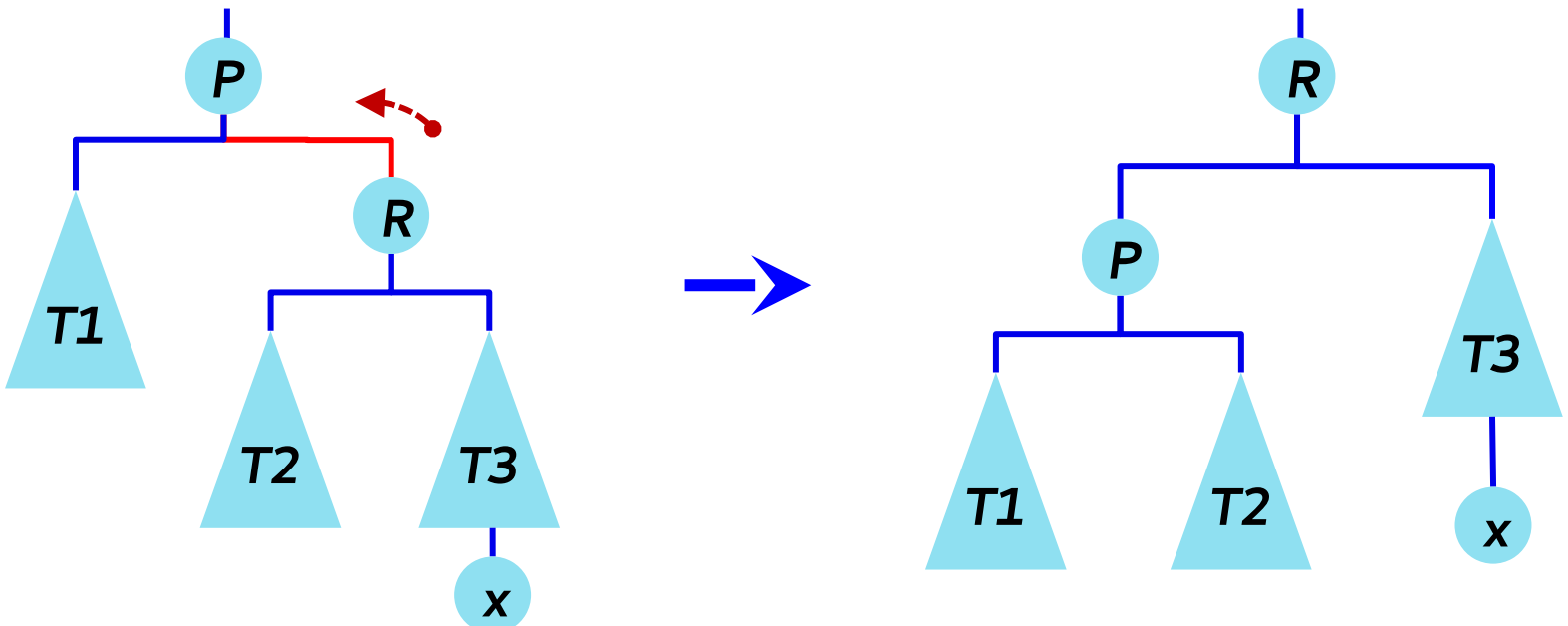
Поворачиваем ребро, связывающее *корень* и его *правый* дочерний узел, *влево*



Левый поворот АВЛ-дерева (*L-rotation*)

Левый поворот в общем случае:

В правое поддерево вставлен элемент *x*, дерево не сбалансировано: *h*(*left*) < *h*(*right*)



Левый поворот АВЛ-дерева (*L-rotation*)

Левый поворот в общем случае:

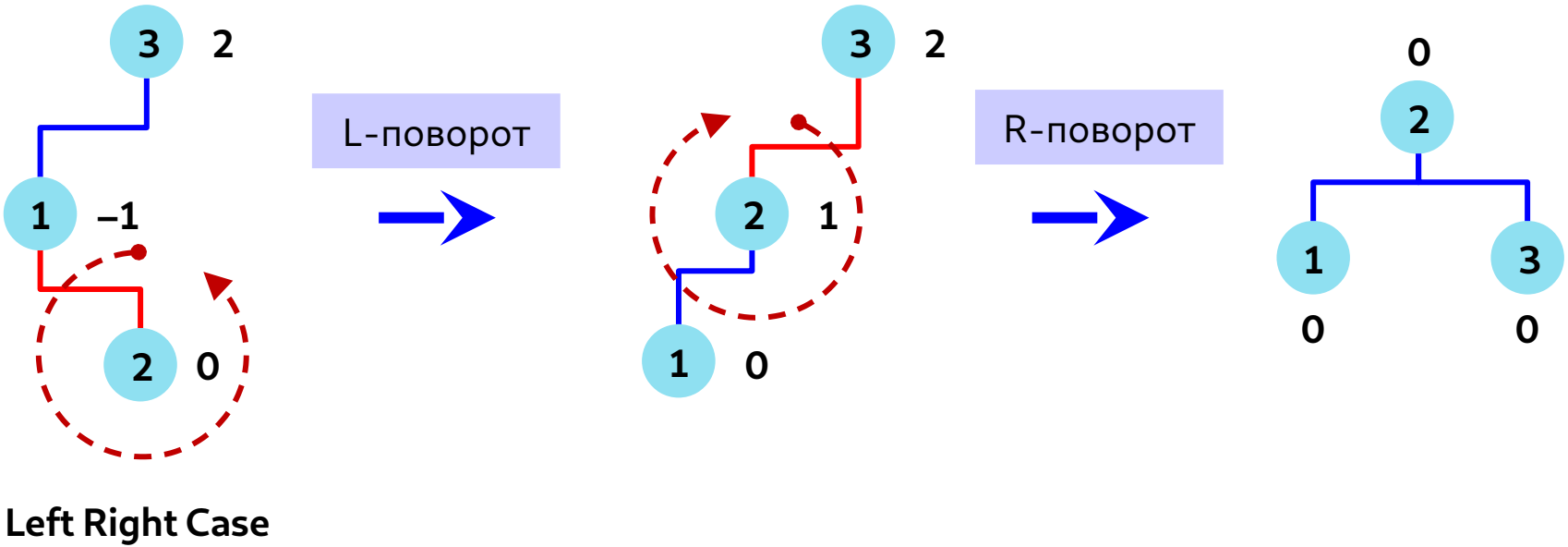
В правое поддерево вставлен элемент *x*, дерево не сбалансировано: *h*(*left*) < *h*(*right*)

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

$T_{LeftRotate} = O(1)$

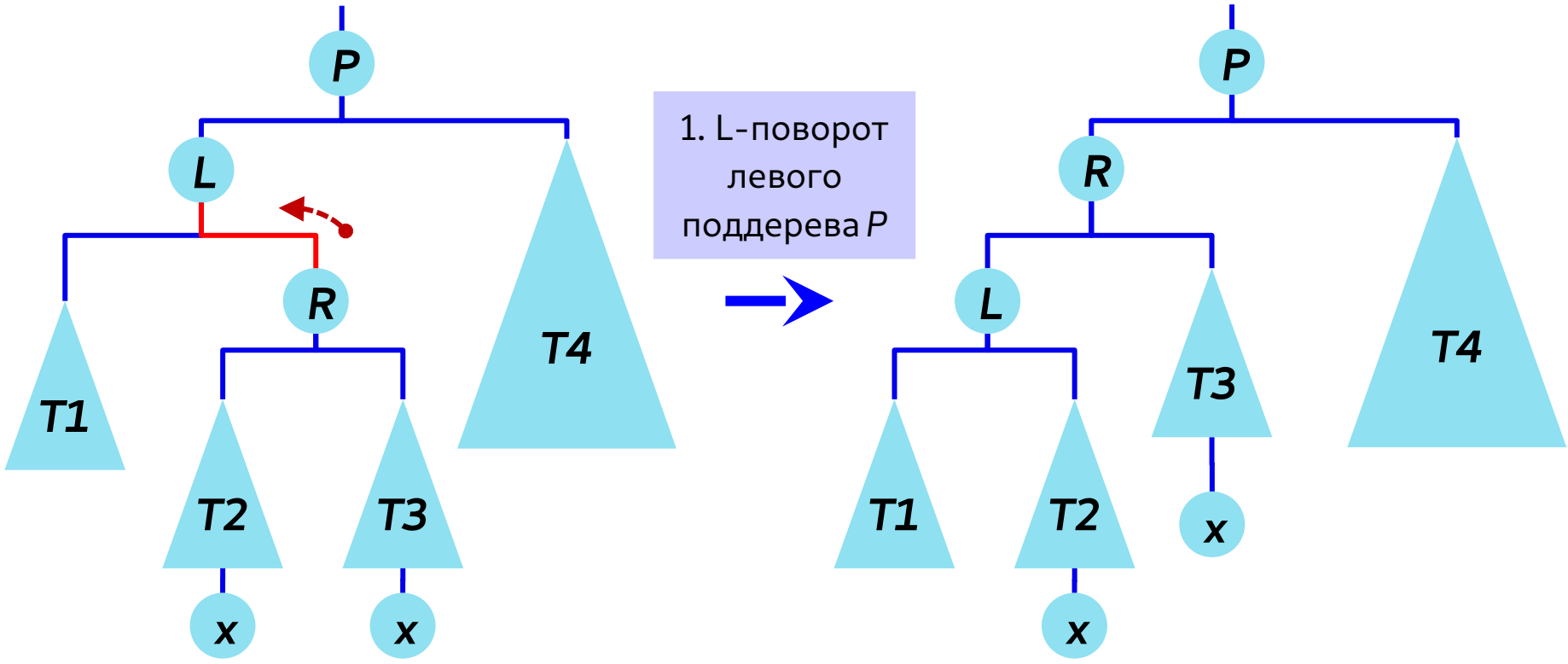
Двойной лево-правый поворот АВЛ-дерева (*LR-rotation*)

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

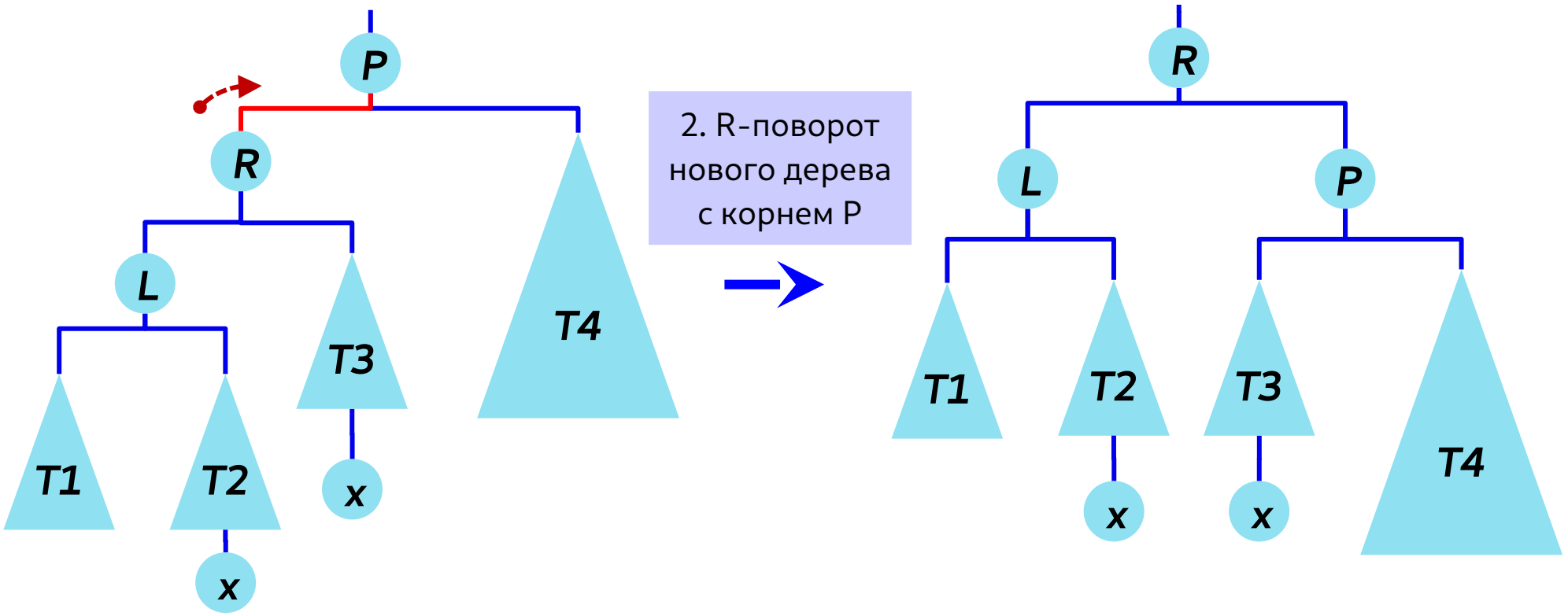


Left Right Case

Двойной лево-правый поворот АВЛ-дерева (*LR-rotation*)



Двойной лево-правый поворот АВЛ-дерева (*LR-rotation*)



Двойной лево-правый поворот АВЛ-дерева (*LR-rotation*)

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

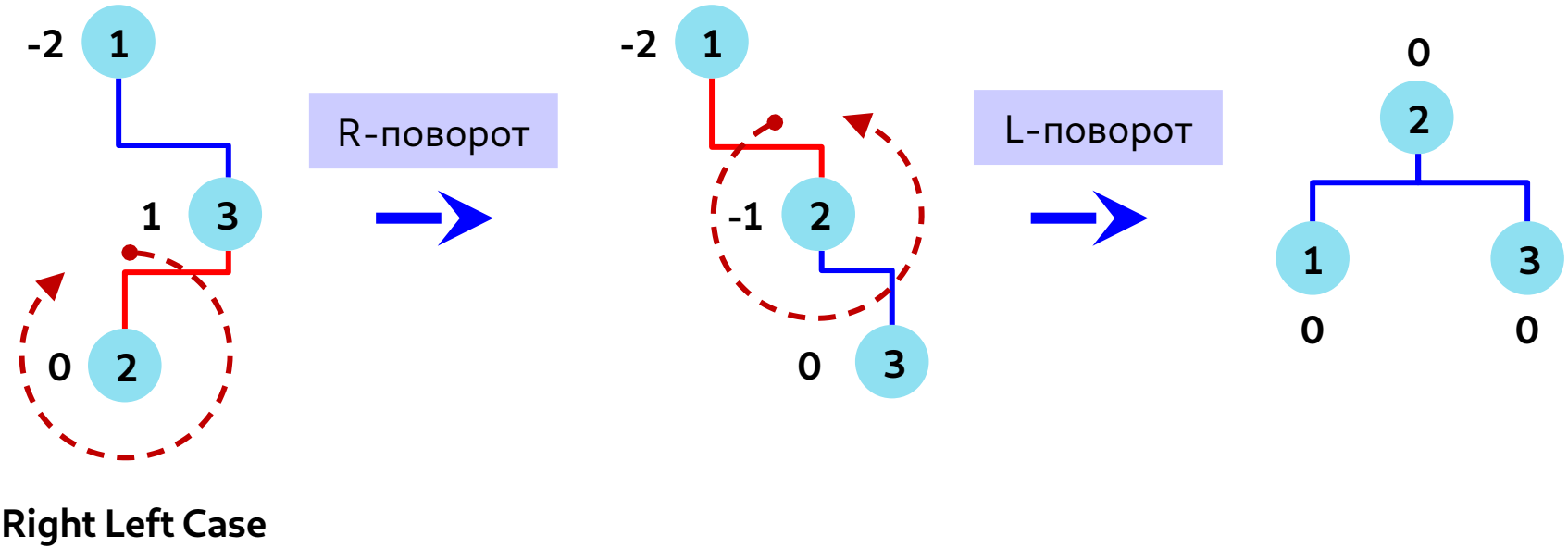
1. L-поворот левого поддерева P
2. R-поворот нового дерева с корнем P

```
function AVLTree_LRRotate(P)
    P.left = AVLTree_LRotate(P.left)
    P = AVLTree_RRotate(P)
end function
```

$T_{LeftRightRotate} = O(1)$

Двойной право-левый поворот АВЛ-дерева (*RL-rotation*)

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



Right Left Case

Двойной право-левый поворот АВЛ-дерева (*RL-rotation*)

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

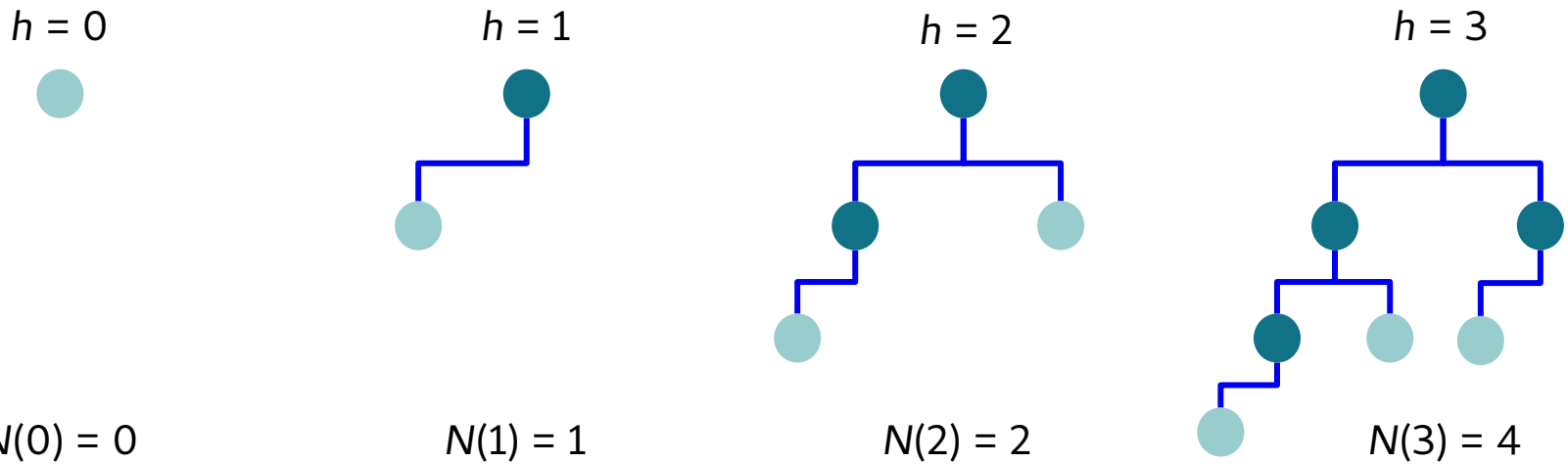
1. R-поворот правого поддерева P
2. L-поворот нового дерева с корнем P

```
function AVLTree_RLRotate(P)
    P.right = AVLTree_RRotate(P.right)
    P = AVLTree_LRotate(P)
end function
```

$T_{RightLeftRotate} = O(1)$

Анализ эффективности АВЛ-деревьев

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



Анализ эффективности АВЛ-деревьев

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

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

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

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

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

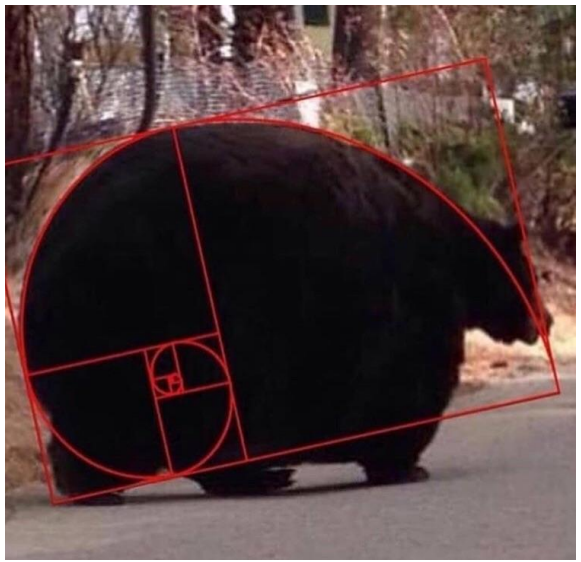
Анализ эффективности АВЛ-деревьев

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

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

$\phi = \frac{1+\sqrt{5}}{2}$ — отношение золотого сечения

$$\boxed{F_n = \left[\frac{\phi^n}{\sqrt{5}}\right], n \geq 0.}$$



Анализ эффективности АВЛ-деревьев

$$N(h)=[\frac{\phi^{h+2}}{\sqrt{5}}]-1>\frac{\phi^{h+2}}{\sqrt{5}}-2;$$

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

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

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

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

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

Анализ эффективности АВЛ-деревьев

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

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

$$\boldsymbol{h < 1.4405\log_2(N(h)+2) - 0.3277}$$

$$\boldsymbol{h = O(\log(n+2))}$$

Анализ эффективности АВЛ-деревьев

- Представлены следующие оценки эффективности АВЛ-деревьев:

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

Кнут Д. Искусство программирования, Том 3. Сортировка и поиск. — М.: Вильямс, 2007. — 824 с.

Вирт Н. Алгоритмы и структуры данных. — М.: Мир, 1989. — 360 с.

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

Левитин А.В. Алгоритмы: введение в разработку и анализ. — М.: Вильямс, 2006. — 576 с.

Программная реализация АВЛ-дерева, структура узла

```
struct avltree {
    int key;
    char *value;

    int height;
    struct avltree *left;
    struct avltree *right;
};
```

Удаление всех узлов дерева, обход в обратном порядке

```
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 в АВЛ-дереве 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 АВЛ-дерева

```
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);
}
```

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

Добавление узла в АВЛ-дерево

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

Добавление узла в АВЛ-дерево (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 в АВЛ-дерево (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 в АВЛ-дерево (окончание)

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

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

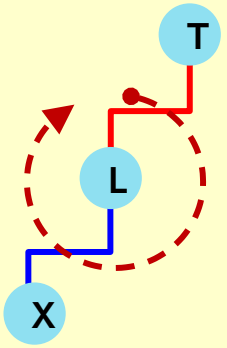
- Поиск листа для вставки нового элемента выполняется за время $O(\log n)$
- Повороты выполняются за время $O(1)$, их количество не может превышать $O(\log n)$ при подъёме от нового элемента к корню (высота AVL-дерева — $O(\log n)$)

Правый 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;
}
```



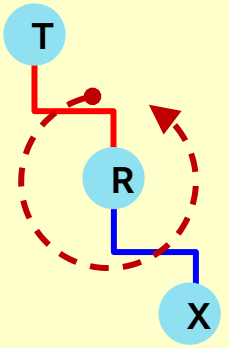
$T_{RightRotate} = O(1)$

Левый 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;
}
```



$T_{LeftRotate} = O(1)$

LR- и RL-повороты

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

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

$$T_{LeftRightRotate} = T_{RightLeftRotate} = O(1)$$

Удаление узлов из АВЛ-дерева

- 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]**

Алгоритм ленивого удаления узлов (*lazy deletion*)

- С каждым узлом АВЛ-дерева ассоциирован флаг **deleted**
- При удалении узла находим его в дереве и устанавливаем флаг **deleted = 1** (реализация 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** создаём новое АВЛ-дерево, содержащее все неудалённые узлы (**deleted = 0**)
- Поиск неудалённых элементов и их вставка в новое АВЛ-дерево реализуется за время $O(n \log n)$

Дальнейшее чтение

1. Более подробно изучить устройство АВЛ-деревьев:
   - Левитин А. В. Алгоритмы: введение в разработку и анализ. — М.: Вильямс, 2006. — 576 с.
   - Вирт Н. Алгоритмы и структуры данных. — М.: Мир, 1989. — 360 с.
   - Кнут Д. Искусство программирования, Том 3. Сортировка и поиск. — М.: Вильямс, 2007. — 824 с.

2. Познакомиться со следующими древовидными структурами данных:
   - Косые деревья (*splay trees*)
   - Списки с пропусками (*skip lists*)

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.