Лекция 7
Бинарные деревья поиска

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)
Весенний семестр, 2016

АТД «Словарь» (dictionary)

- **Словарь** (dictionary) – структура данных для хранения пар вида «ключ» – «значение» (key – value)
- **Альтернативные название** – ассоциативный массив (associative array, map)
- В словаре может быть только одна пара с заданным ключом

Ключ (key)	Значение (value)
890	Слон
1200	Кит
260	Лев
530	Жираф

АТД «Словарь» (dictionary)

Операция	Описание
Add(*map*, *key, value*)	Добавляет в словарь *map* пару (*key*, *value*)
Lookup(*map, key*)	Возвращает из словаря *map* значение ассоциированное с ключом *key*
Remove(*map, key*)	Удаляет из словаря *map* пару с ключом *key*
Min(*map*)	Возвращает из словаря *map* минимальное значение
Max(*map*)	Возвращает из словаря *map* максимальное значение

Реализация АТД «Словарь»

- Реализации словарей отличаются вычислительной сложностью операций и объемом требуемой памяти для хранения пар «ключ-значение»

- Распространение получили следующие реализации:

1. **Деревья поиска (Search trees)**
2. **Хэш-таблицы (Hash tables)**
3. **Списки с пропусками (Skip lists)**
4. Связные списки, массивы

Реализация словаря на основе массива

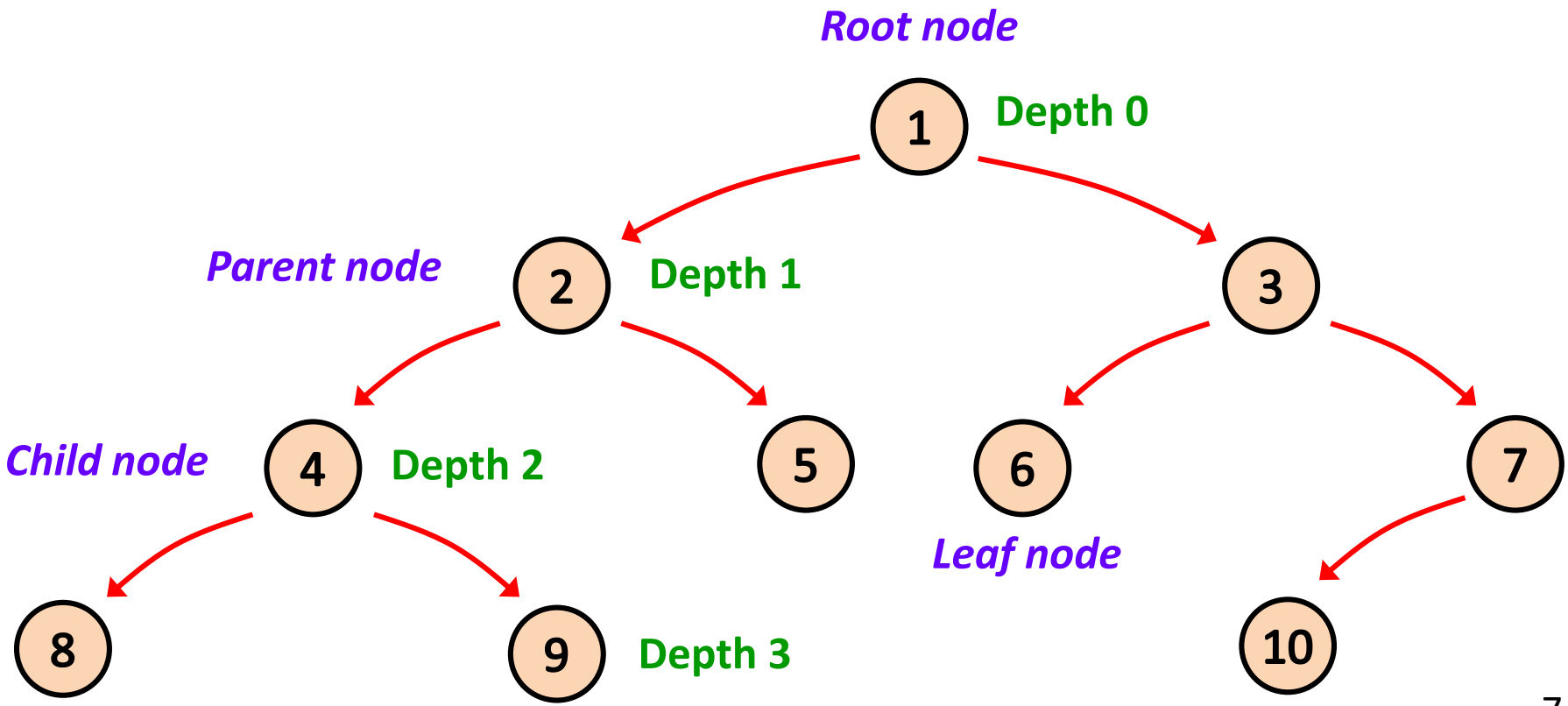
Операция	Неотсортированный массив	Отсортированный массив
Add (*map*, *key, value*)	*O*(1) (добавление в конец)	*O*(*n*) (поиск позиции)
Lookup (*map, key*)	*O*(*n*)	*O*(log*n*) (бинарный поиск)
Remove (*map, key*)	*O*(*n*) (поиск элемента и перенос последнего на место удаляемого)	*O*(*n*) (перемещение элементов)
Min(*map*)	*O*(*n*)	*O*(1) (элемент *v*[1])
Max(*map*)	*O*(*n*)	*O*(1) (элемент *v*[*n*])

Реализация словаря на основе связного списка

Операция	Неотсортированный связный список	Отсортированный связный список
Add (*map*, *key, value*)	*O*(1) (добавление в начало)	*O*(*n*) (поиск позиции)
Lookup (*map, key*)	*O*(*n*)	*O*(*n*)
Remove (*map, key*)	*O*(*n*) (поиск элемента)	*O*(*n*) (поиск элемента)
Min(*map*)	*O*(*n*)	*O*(1)
Max(*map*)	*O*(*n*)	*O*(*n*) или *O*(1), если поддерживать указатель на последний элемент

Бинарные деревья (binary trees)

- ***Бинарное дерево* (binary tree)** – это дерево (структура данных), в котором каждый узел (node) имеет не более двух дочерних узлов (child nodes)



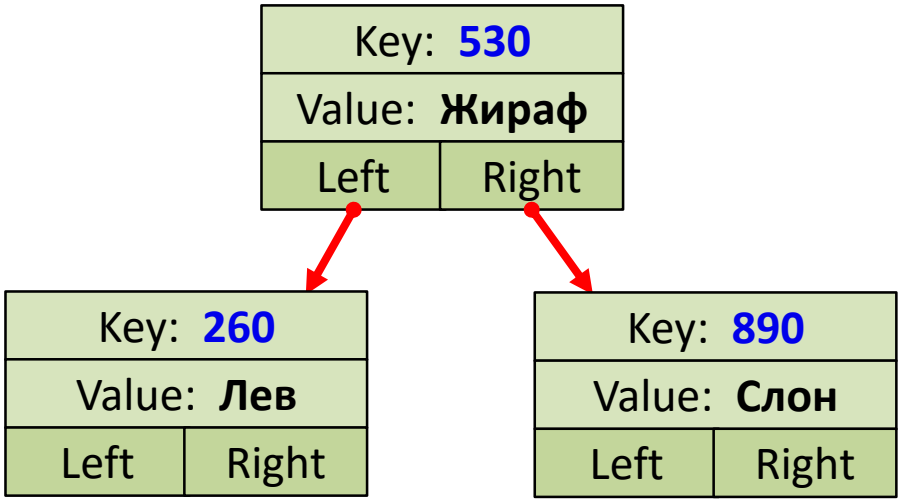
Бинарные деревья поиска (binary search trees)

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

  1) каждый узел ***x*** (node) имеет не более двух дочерних узлов (child nodes) и содержит ключ (key) и значение (value)

  2) ключи всех узлов левого поддерева узла *x* меньше значения его ключа

  3) ключи всех узлов правого поддерева узла *x* больше значения его ключа

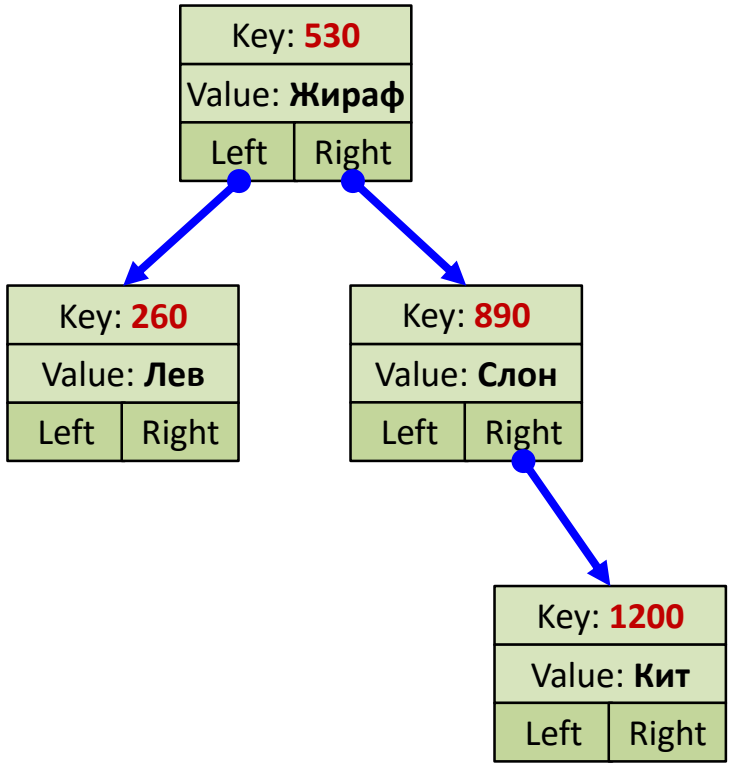


Двоичные деревья поиска (binary search trees)

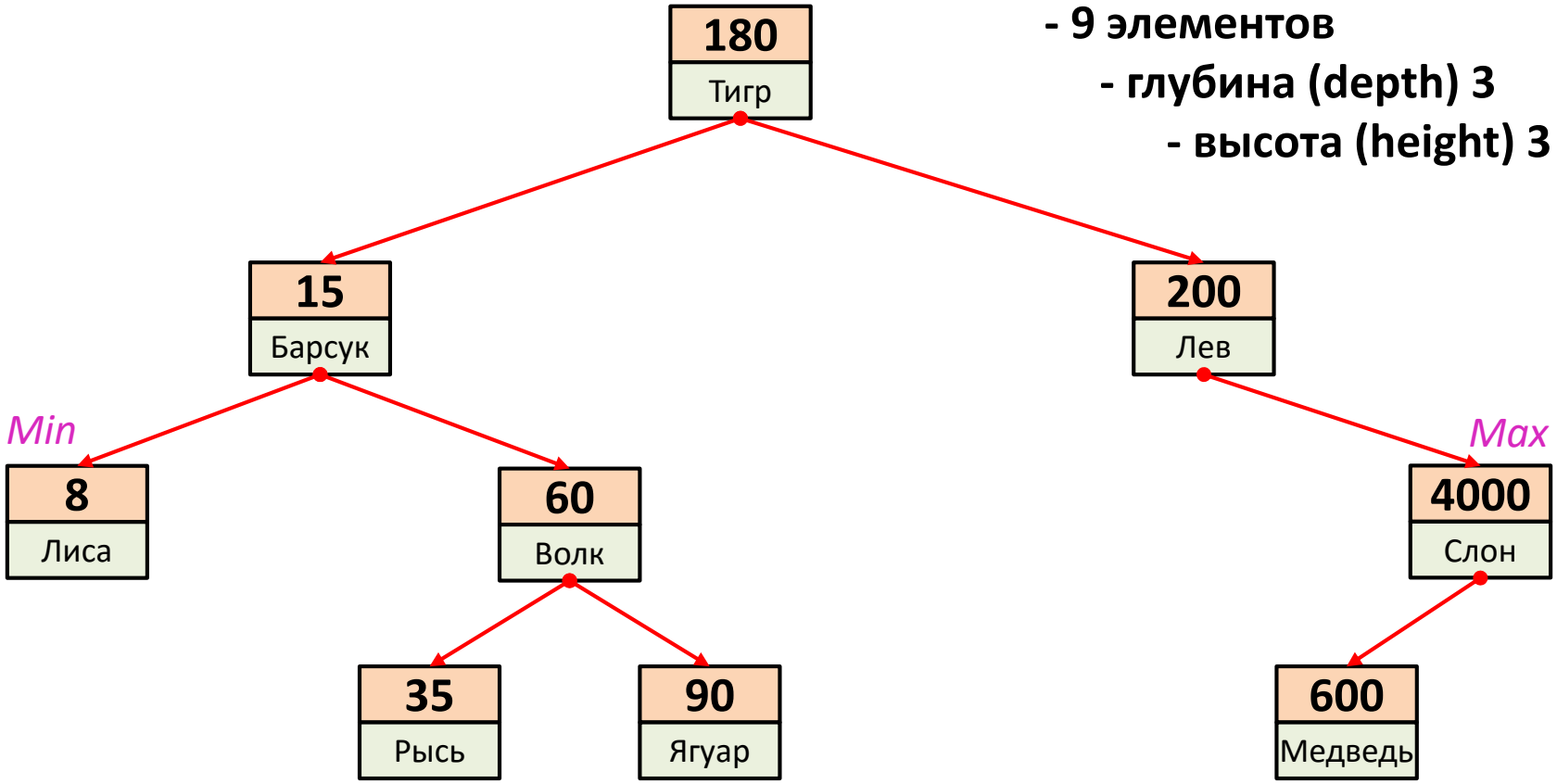
Словарь

Ключ (Key)	Значение (Value)
530	Жираф
260	Лев
890	Слон
1200	Кит

Двоичное дерево поиска



Двоичные деревья поиска (Binary search trees)



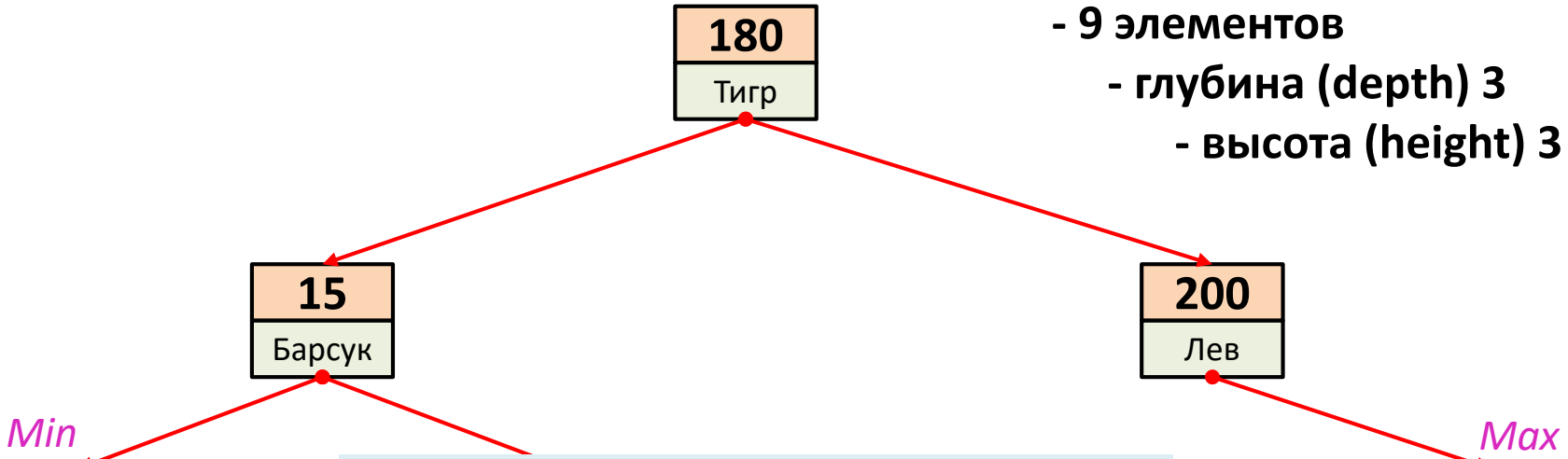
- 9 элементов

- глубина (depth) 3

- высота (height) 3

- **Упорядоченный словарь** (ordered map) – словарь, обеспечивающий перебор элементов в упорядоченной последовательности
- Операции Prev(key), Next(key)

Двоичные деревья поиска (Binary search trees)



- 9 элементов
- глубина (depth) 3
- высота (height) 3

```
// Проход по возрастанию ключей
node = Min(Map)
max = Max(Map)
while node != max do
    // Обработка узла node ...
    node = Next(node)
end while
```

- **Упорядоченный словарь** (ordered map) – словарь, обеспечивающий перебор элементов в упорядоченной последовательности
- Операции Prev(key), Next(key)

Двоичные деревья поиска (Binary search trees)

```
#include <stdio.h>
#include <stdlib.h>

struct bstree {
    int key;          /* Ключ */
    char *value;      /* Данные */

    struct bstree *left;
    struct bstree *right;
};
```

Создание элемента BST

```
struct bstree *bstree_create(int key,
                             char *value)
{
    struct bstree *node;

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

$T_{Create} = O(1)$

Создание элемента BST

```
int main()
{
    struct bstree *tree;

    tree = bstree_create(180, "Tigr");

    return 0;
}
```

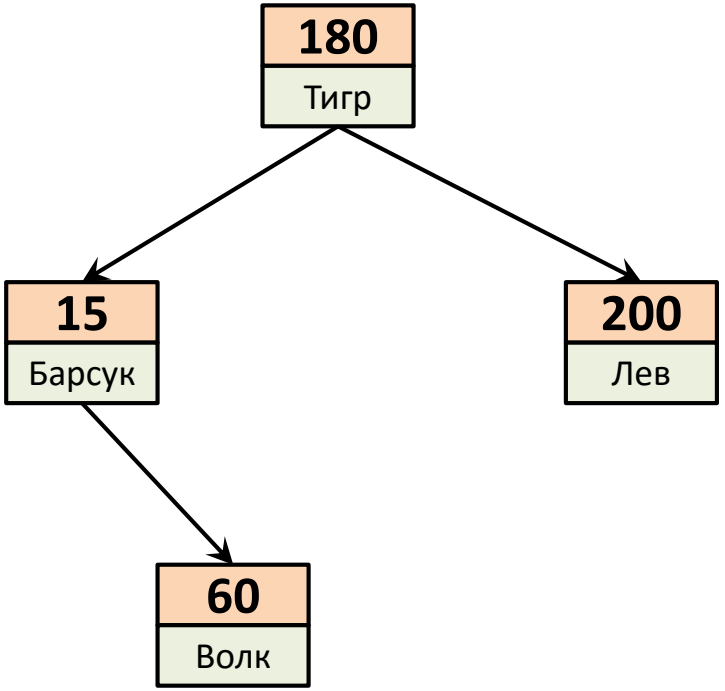
Добавление элемента в BST

1. Добавление элемента (**180**, Тигр)

2. Добавление элемента (**200**, Лев)

3. Добавление элемента (**15**, Барсук)

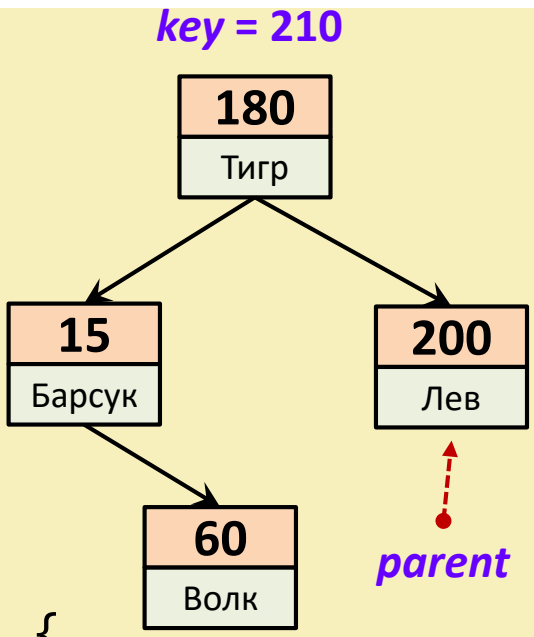
4. Добавление элемента (**60**, Волк)



Ищем листовой узел (leaf node) для вставки нового элемента

Добавление элемента в BST

```
void bstree_add(struct bstree *tree,
                int key, char *value)
{
    struct bstree *parent, *node;

    if (tree == NULL)
        return;

    /* Отыскиваем листовой узел */
    for (parent = tree; tree != NULL; ) {
        parent = tree;
        if (key < tree->key)
            tree = tree->left;
        else if (key > tree->key)
            tree = tree->right;
        else
            return;
    }
```



Добавление элемента в BST (продолжение)

```
    /* Создаем элемент и связываем с узлом */
    node = bstree_create(key, value);
    if (key < parent->key)
        parent->left = node;
    else
        parent->right = node;
}
```

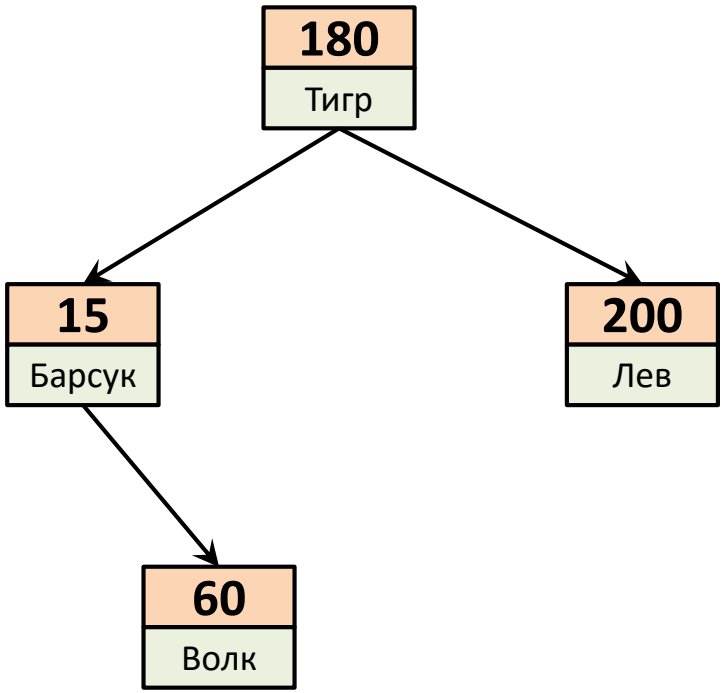
$T_{Add} = O(h)$

- При добавлении элемента необходимо спуститься от корня дерева до листа – это требует количества операций порядка высоты *h* дерева

- Поиск листа – $O(h)$, создание элемента и корректировка указателей – $O(1)$

Поиск элемента в BST

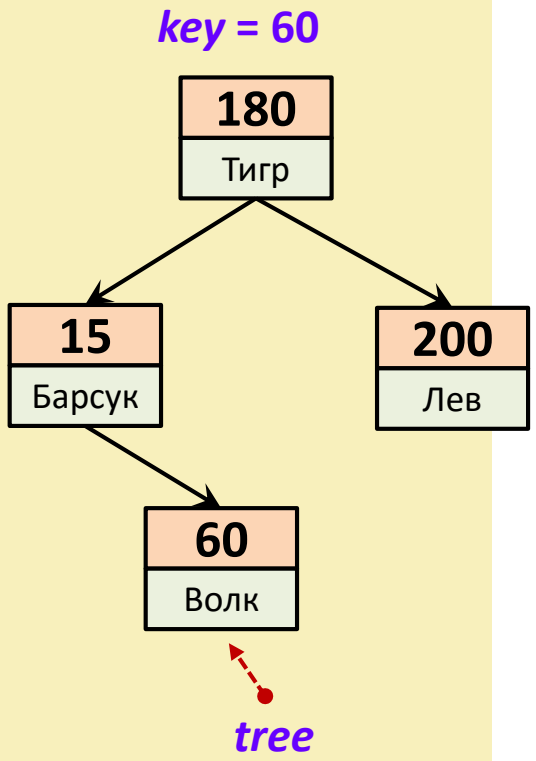
1. Сравниваем ключ корневого узла с искомым. Если совпали, то элемент найден
2. Переходим к левому или правому дочернему узлу и повторяем шаг 1



Возможны рекурсивная и не рекурсивная реализации

Поиск элемента в BST

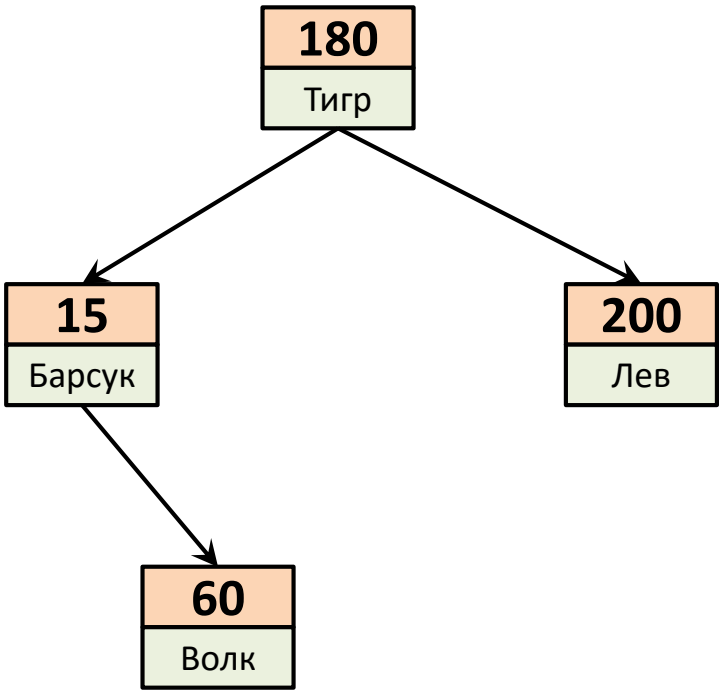
```
struct bstree *bstree_lookup(struct bstree *tree,
                             int key)
{
    while (tree != NULL) {
        if (key == tree->key) {
            return tree;
        } else if (key < tree->key) {
            tree = tree->left;
        } else {
            tree = tree->right;
    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



$T_{Lookup} = O(h)$

Поиск минимального элемента в BST

- Минимальный элемент всегда расположен в левом поддереве корневого узла
- Требуется найти самого левого потомка корневого узла



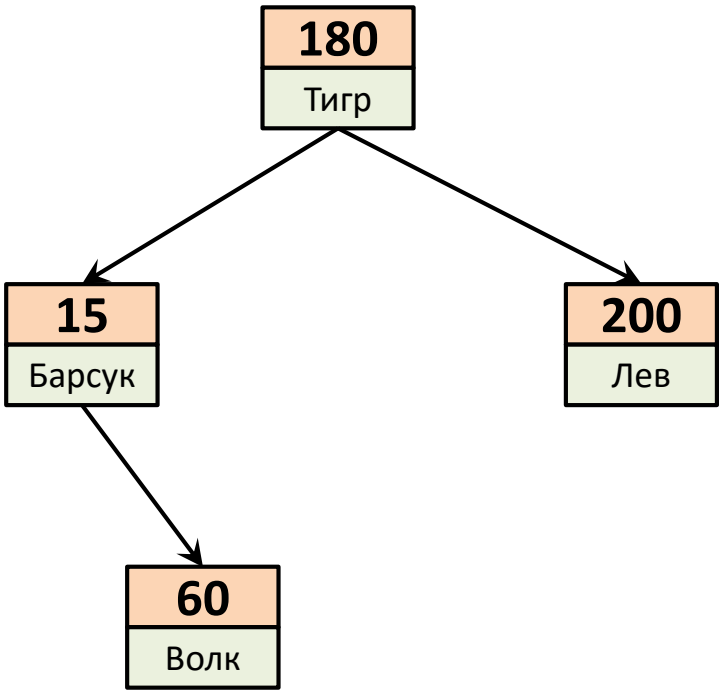
Поиск минимального элемента в BST

```
struct bstree *bstree_min(struct bstree *tree)
{
    if (tree == NULL)
        return NULL;

    while (tree->left != NULL)
        tree = tree->left;
    return tree;
}
```

$T_{Min} = O(h)$

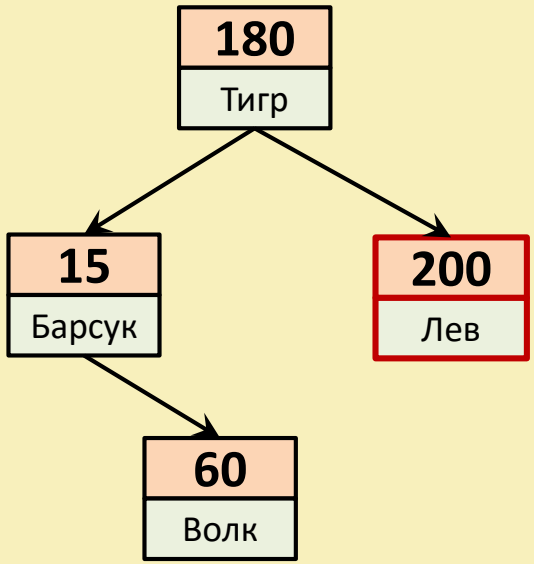
Поиск максимального элемента в BST

- Максимальный элемент всегда расположен в правом поддереве корневого узла
- Требуется найти самого правого потомка корневого узла



Поиск максимального элемента в BST

```
struct bstree *bstree_max(struct bstree *tree)
{
    if (tree == NULL)
        return NULL;

    while (tree->right != NULL)
        tree = tree->right;
    return tree;
}
```



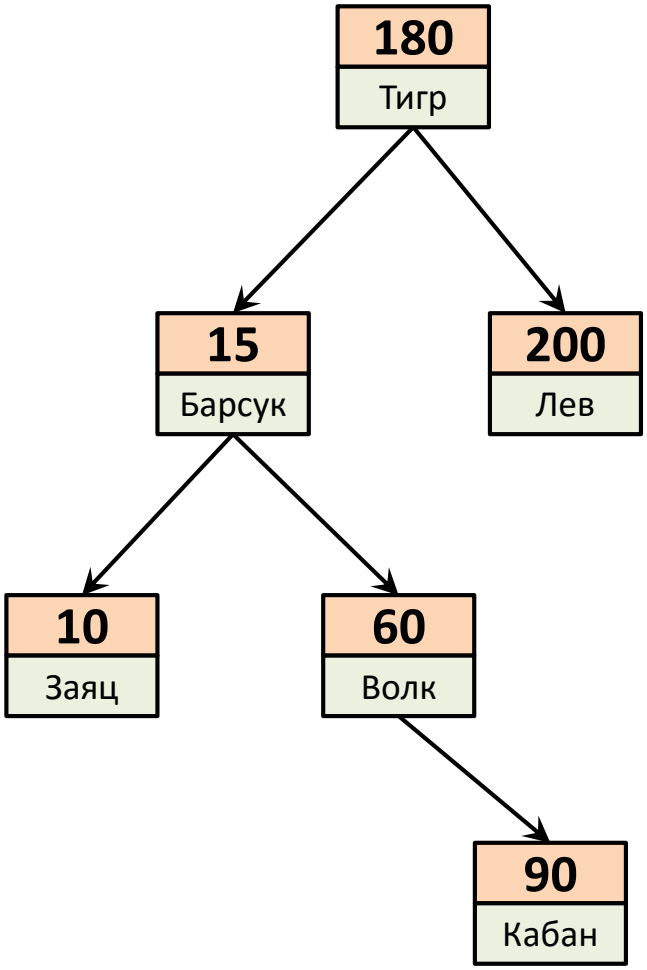
$T_{Max} = O(h)$

Пример

```
int main()
{
    struct bstree *tree, *node;

    tree = bstree_create(180, "Tigr");
    bstree_add(tree, 200, "Lev");
    bstree_add(tree, 60, "Volk");

    node = bstree_lookup(tree, 200);
    printf(“Value = %s\n", node->value);

    node = bstree_min(tree);
    printf("Min: value = %s\n", node->value);

    return 0;
}
```

Удаление элемента из BST

1. Находим узел *z* с заданным ключом – $O(n)$
2. Возможны 3 ситуации:
   - узел *z* не имеет дочерних узлов
   - узел *z* имеет 1 дочерний узел
   - узел *z* имеет 2 дочерних узла



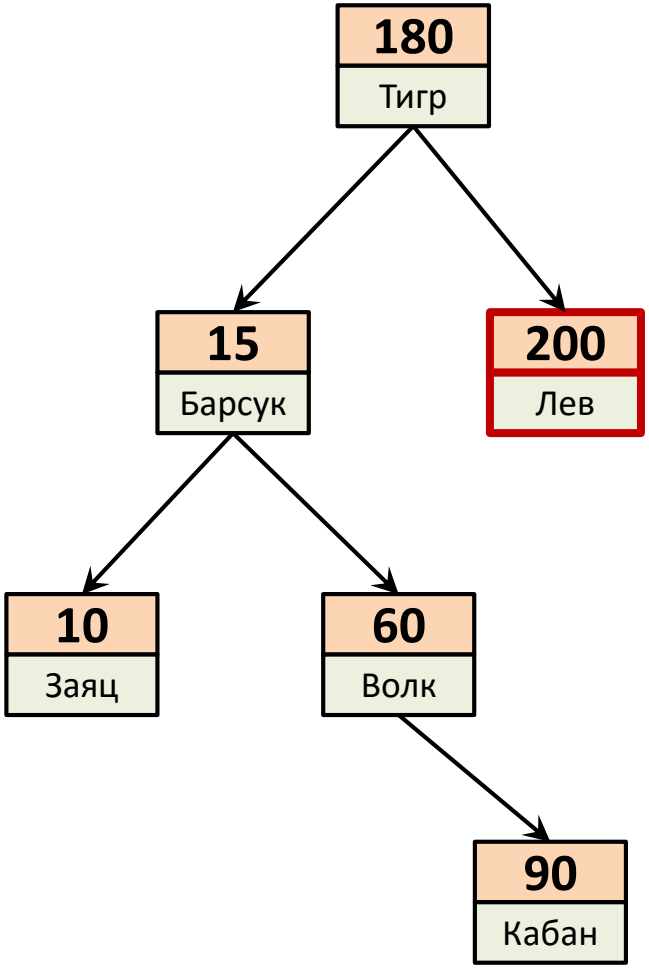
Удаление элемента из BST

Удаление узла **“Лев”** (случай 1)

1. Находим и удаляем узел “Лев” из памяти (free)

2. Родительский указатель (left или right) устанавливаем в значение NULL

   “Тигр”->right = NULL

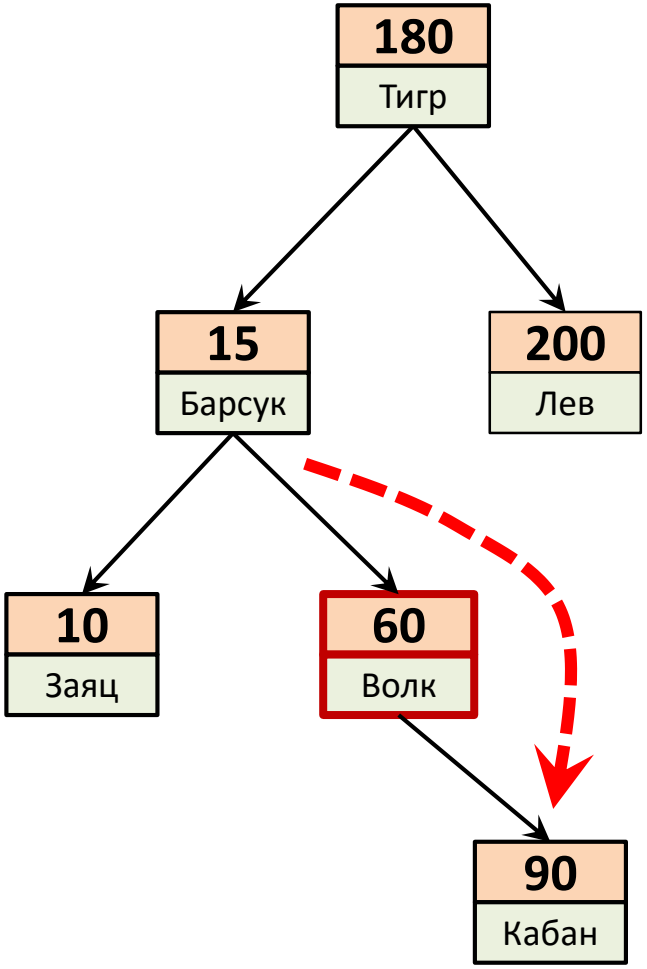


Удаление элемента из BST

Удаление узла **“Волк”** (случай 2)

1. Находим узел “Волк”
2. Родительский указатель узла “Волк” (left или right) устанавливаем на его дочерний элемент
3. Удаляем узел “Волк” из памяти

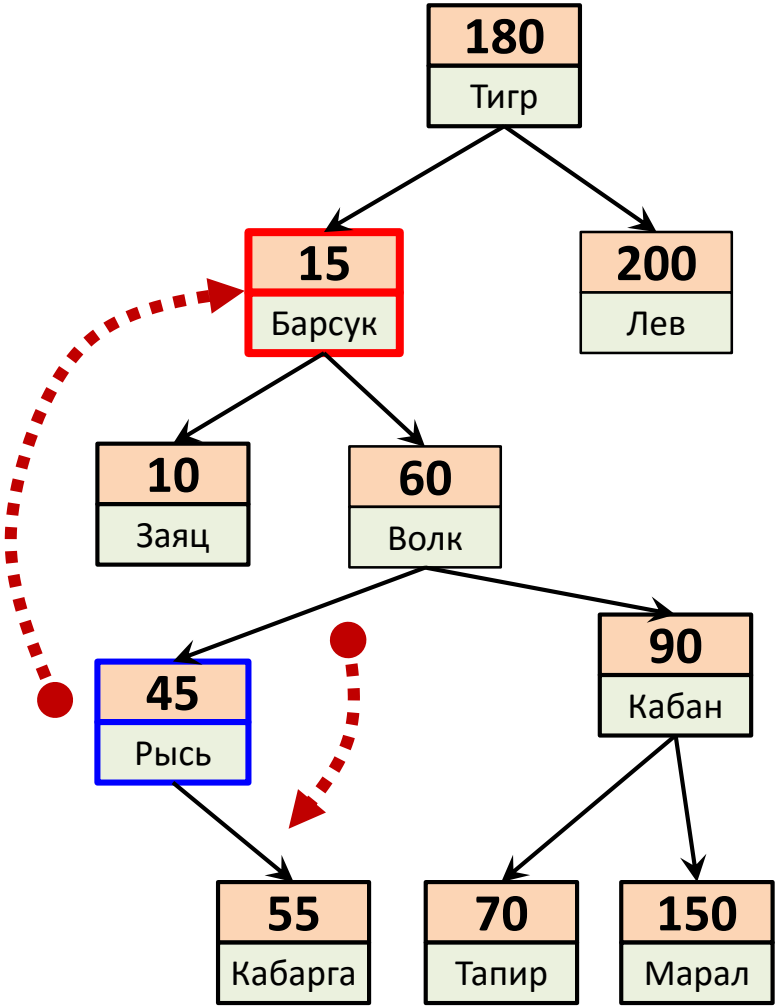
“Барсук”->right = “Волк”->right



Удаление элемента из BST

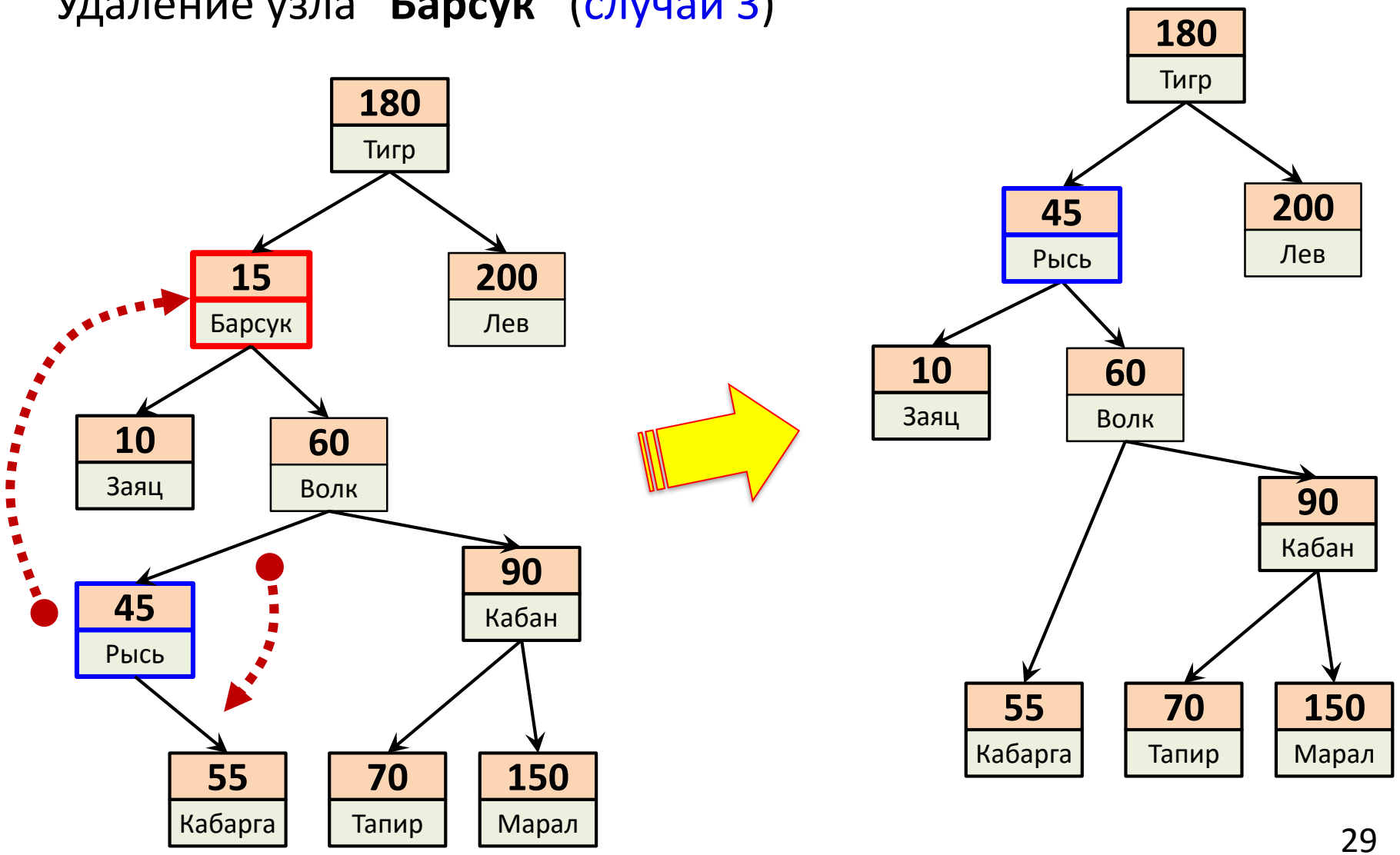
Удаление узла **“Барсук”** (случай 3)

1. Находим узел “Барсук”
2. Находим узел с минимальным ключом в правом поддереве узла “Барсук” – **самый левый лист в поддереве** (узел “Рысь”)
3. Заменяем узел “Барсук” узлом “Рысь”



Удаление элемента из BST

Удаление узла **“Барсук”** (случай 3)

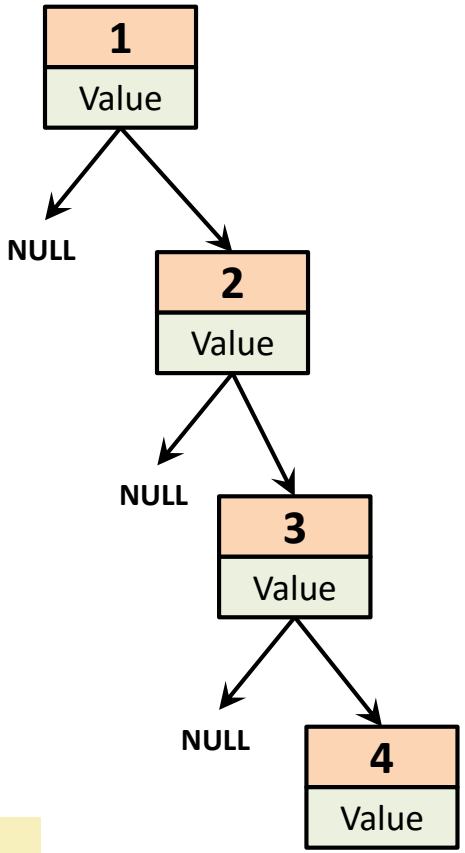


Анализ эффективность BST

1. Операции имеют трудоемкость пропорциональную высоте *h* дерева

2. В **худшем** случае высота дерева ***O*(*n*)** (вставка элементов в отсортированной последовательности)

3. В **среднем** случае высота дерева ***O*(log*n*)**




```
bstree_add(tree, “Item”, 1);
bstree_add(tree, “Item”, 2);
bstree_add(tree, “Item”, 3);
bstree_add(tree, “Item”, 4);
```

Дерево вырождается в связный список

Реализация словаря на основе BST

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

Сбалансированные деревья поиска

- **Сбалансированное по высоте дерево поиска (self-balancing binary search tree)** – дерево поиска, в котором высоты поддеревьев узла различаются не более чем на заданную константу *k*

- Баланс высоты поддерживается при выполнении операций добавления и удаления элементов

- Типы сбалансированных деревьев поиска:
  - Красно-черные деревья (Red-black tree): $h \leq 2\log_2(n+1)$
  - АВЛ-деревья (AVL-tree): $h < 1.4405 \cdot \log_2(n+2) - 0.3277$
  - B-деревья
  - Деревья Ван Эмде Боаса
  - …

Все операции на красно-черном дереве и АВЛ-дереве в худшем случае выполняются за время *O*(log*n*)

Домашнее чтение

- [DSABook, Глава 9]
- Прочитать в «практике программирования»
  [KR, С. 67] «2.8 Деревья»
- Прочитать в [CLRS, С. 328] раздел об удалении узла из бинарного дерева поиска (функции Tree-Delete, Transplant)
- Прочитать про обходы дерева в глубину
  (pre-order, in-order и post-order)
- Как освободить память из под всего дерева зная указатель на корень?