Лекция 7.
Бинарные деревья поиска

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

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

АТД «Словарь» (dictionary)

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

Ключ (key)	Значение (value)
4	Flamingo
14	Fox
5400	Elephant
12	Koala

АТД «Словарь» (dictionary)

Операция	Описание
Add(*map*, *key*, *value*)	Добавляет в словарь *map* пару (*key*, *value*)
Lookup(*map*, *key*)	Возвращает из словаря *map* значение *value*, ассоциированное с ключом *key*
Delete(*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

Реализации АТД «Словарь»

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

- Распространение получили следующие реализации:
  - **Деревья поиска** (search trees)
  - **Хеш-таблицы** (hash tables)
  - **Списки с пропусками** (skip lists)
  - **Связные списки, массивы**

Реализация словаря на основе массива

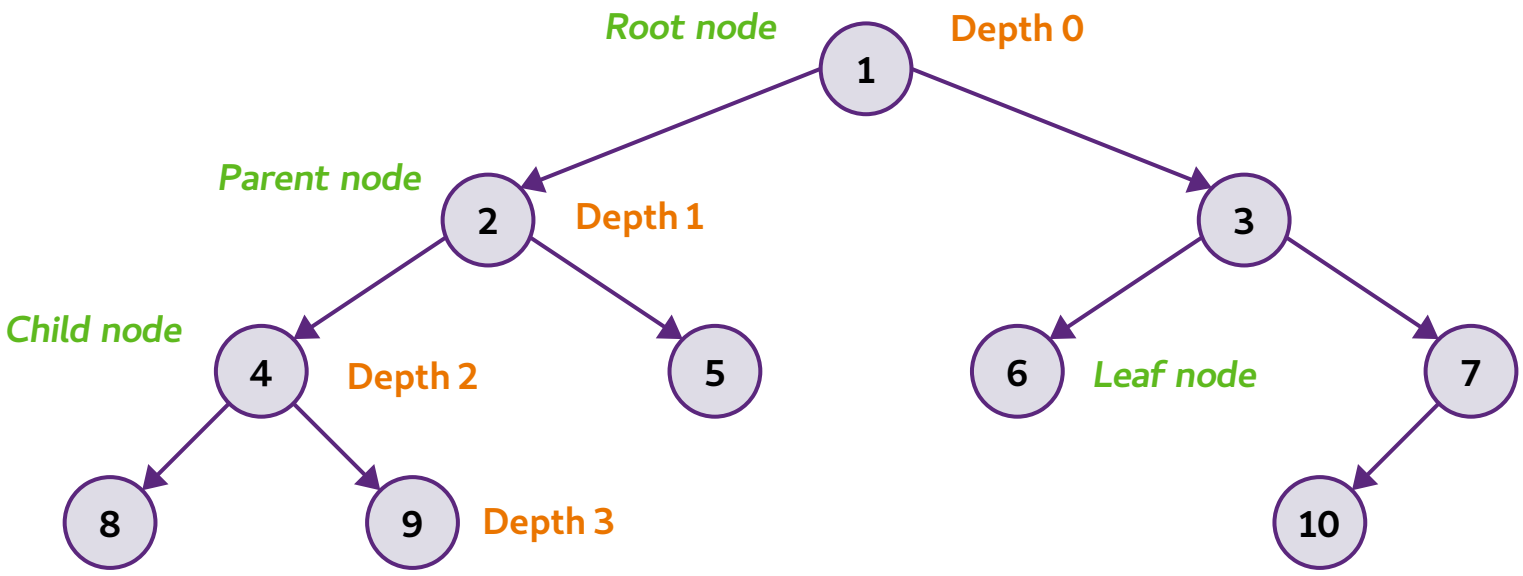
Операция	Неупорядоченный массив	Упорядоченный массив
Add(*map*, *key*, *value*)	$O(1)$ (добавление в конец)	$O(n)$ (поиск позиции)
Lookup(*map*, *key*)	$O(n)$	$O(\log n)$ (бинарный поиск)
Delete(*map*, *key*)	$O(n)$ (поиск элемента и перенос последнего на место удаляемого)	$O(n)$ (перемещение элементов)
Min(*map*)	$O(n)$	$O(1)$ (элемент $a[1]$)
Max(*map*)	$O(n)$	$O(1)$ (элемент $a[n]$)

Реализация словаря на основе связного списка

Операция	Неупорядоченный список	Упорядоченный список
Add(*map*, *key*, *value*)	$O(1)$ (добавление в начало)	$O(n)$ (поиск позиции)
Lookup(*map*, *key*)	$O(n)$	$O(n)$
Delete(*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

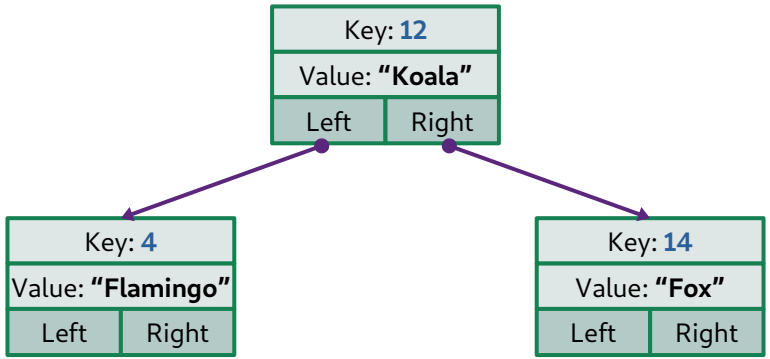
Бинарные деревья (binary trees)

- **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)

- **Бинарное дерево поиска** (двоичное дерево поиска, binary search tree, BST) — это бинарное дерево, в котором:
  - Каждый узел *x* имеет не более двух дочерних узлов и содержит пару «ключ» — «значение»
  - Ключи всех узлов левого поддерева *x* меньше значения его ключа
  - Ключи всех узлов правого поддерева *x* больше значения его ключа

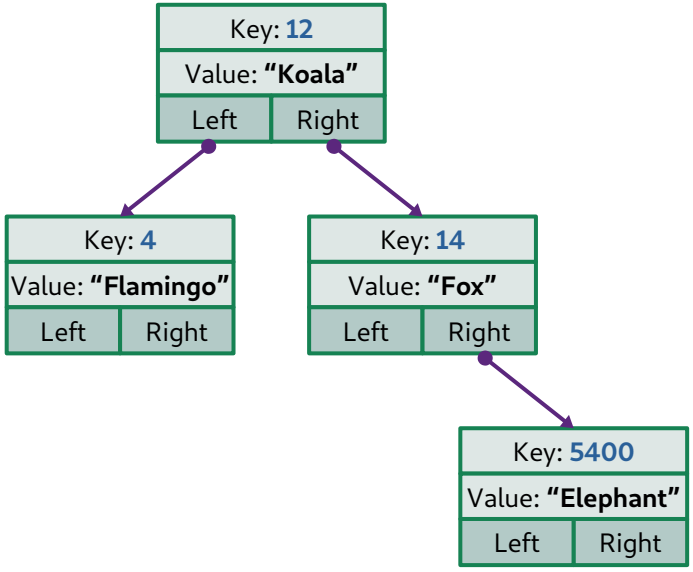


Бинарные деревья поиска (binary search trees)

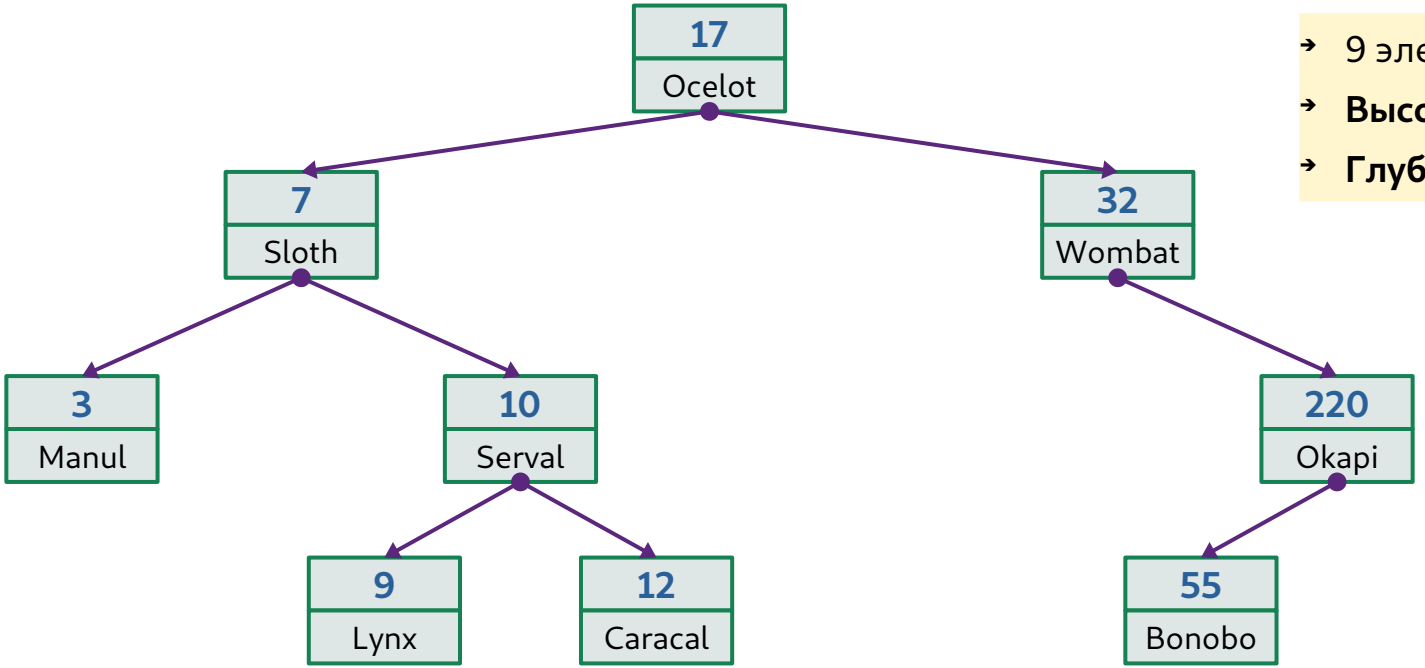
Словарь

Ключ (key)	Значение (value)
4	Flamingo
14	Fox
5400	Elephant
12	Koala

Бинарное дерево поиска



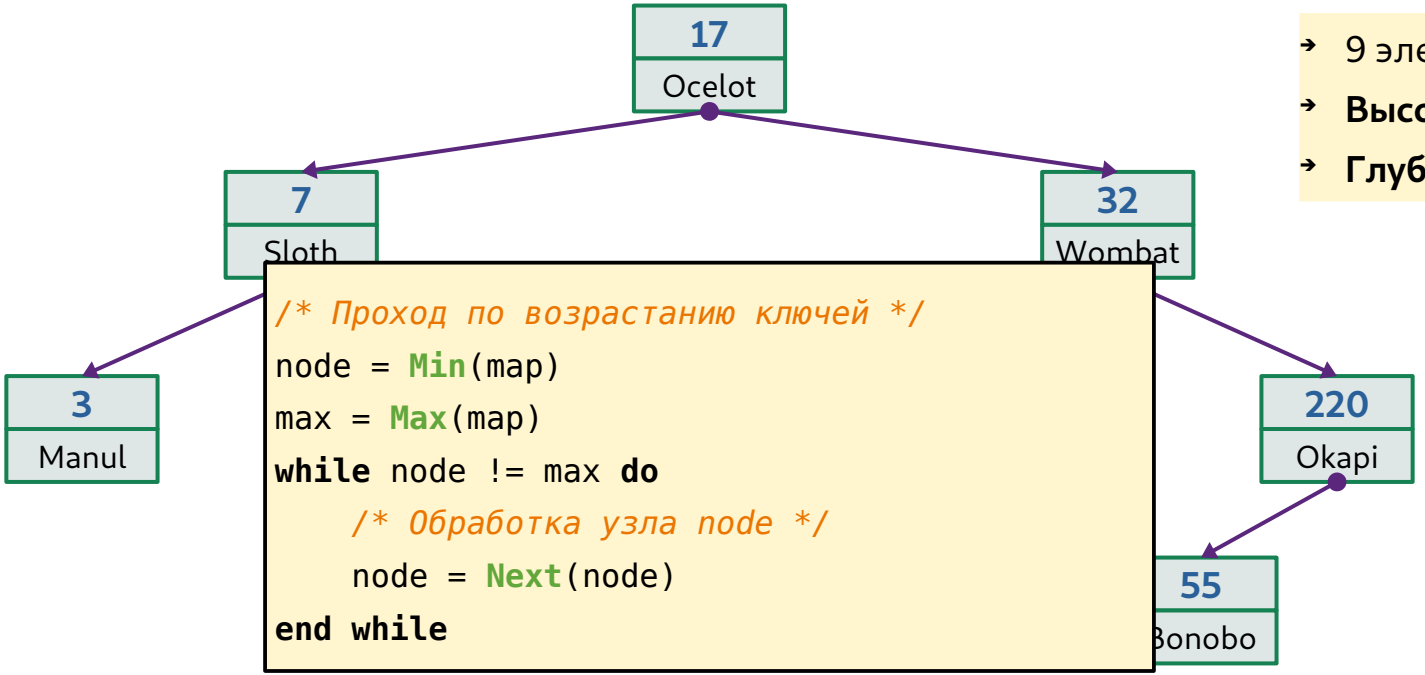
Бинарные деревья поиска (binary search trees)



- 9 элементов
- **Высота** (height): 3
- **Глубина** (depth): 3

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

Бинарные деревья поиска (binary search trees)




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

- 9 элементов
- **Высота** (height): 3
- **Глубина** (depth): 3

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

Бинарные деревья поиска (binary search trees)

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

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

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

Создание узла бинарного дерева поиска

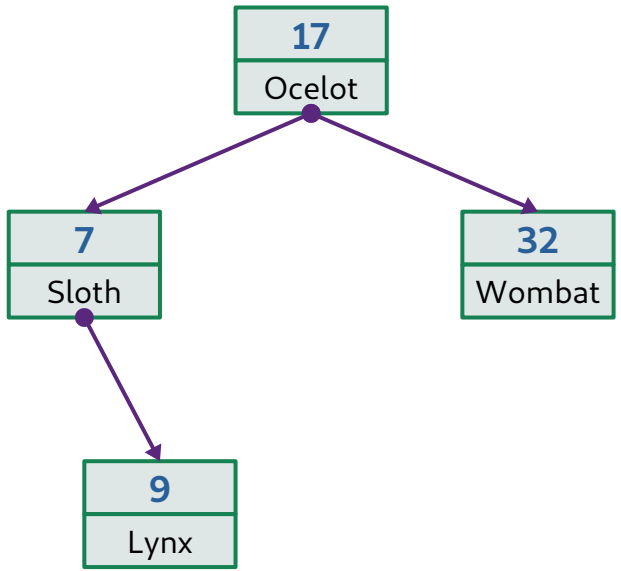
```
struct bstree *bstree_create(int key, char *value)
{
    struct bstree *node;

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

$T_{Create} = O(1)$

Добавление элемента в BST

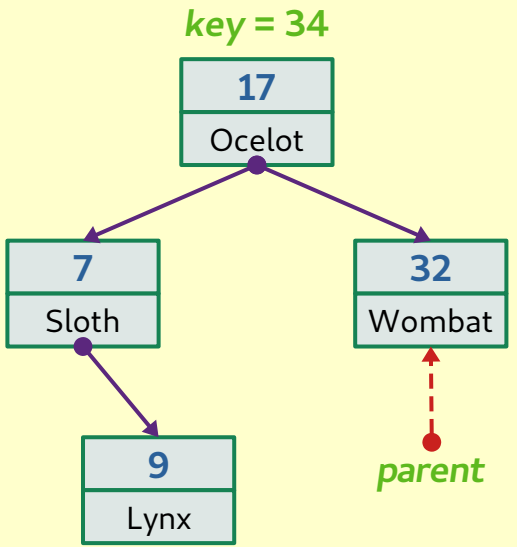
1. Добавление элемента (**17**, Ocelot)
2. Добавление элемента (**32**, Wombat)
3. Добавление элемента (**7**, Sloth)
4. Добавление элемента (**9**, Lynx)

Ищем листовой узел (leaf node) для вставки нового элемента



Добавление элемента в BST

```
void *bstree_add(struct bstree *tree, int key, char *value)
{
    if (tree == NULL)
        return;
    struct bstree *parent, *node;
    while (tree != NULL) {
        parent = tree;
        if (key < tree->key)
            tree = tree->left;
        else if (key > tree->key)
            tree = tree->right;
        else
            return;
    }
```



$T_{Add} = O(h)$

Добавление элемента в BST (продолжение)

```
    node = bstree_create(key, value);
    if (key < parent->key)
        parent->left = node;
    else
        parent->right = node;
}
```

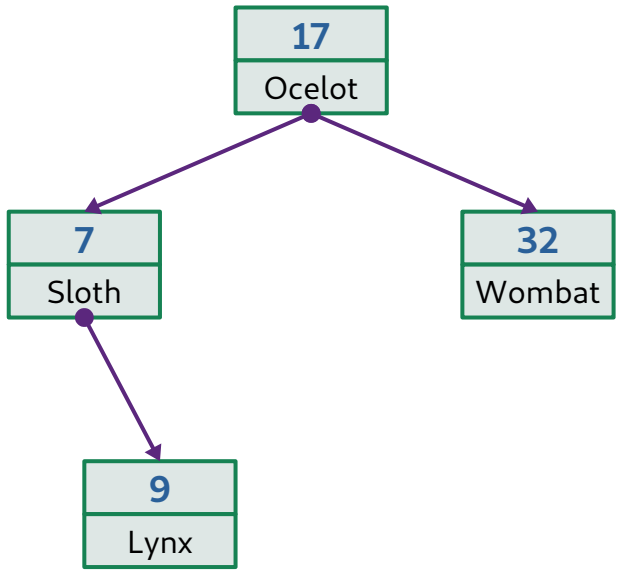
$T_{Add} = O(h)$

- 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

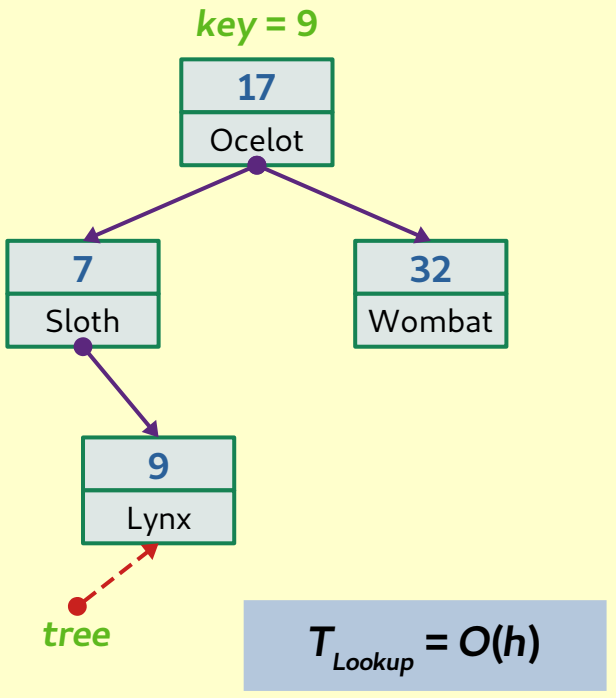
1. Сравниваем ключ корневого элемента с искомым. Если совпали, то элемент найден
2. Переходим к левому или правому дочернему узлу и повторяем шаг 1

Возможны рекурсивная и итеративная реализации



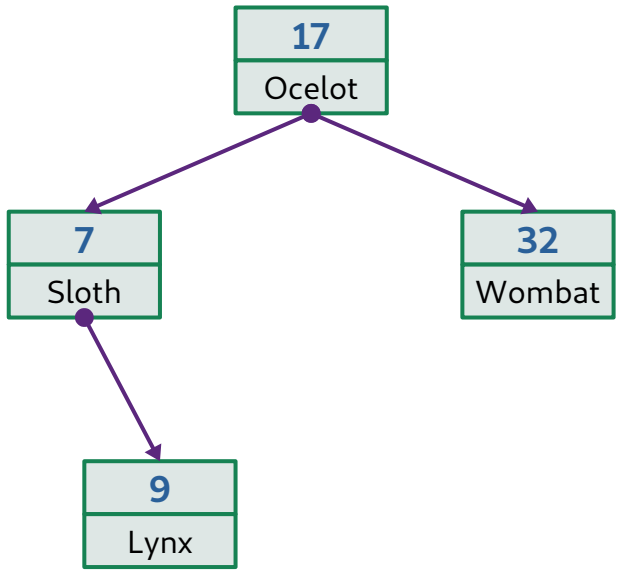
Поиск элемента в BST

```
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;
}
```



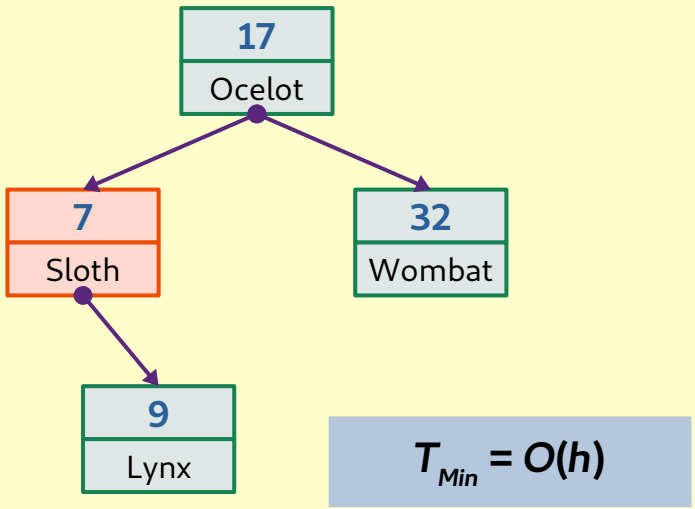
Поиск минимального элемента в BST

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



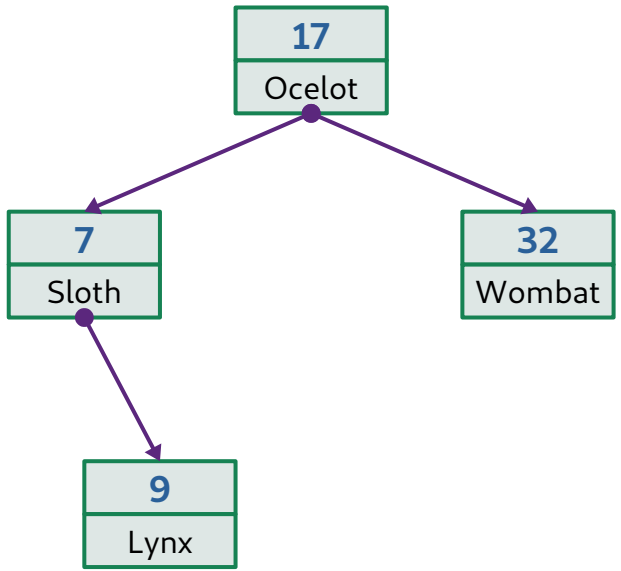
Поиск минимального элемента в BST

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

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



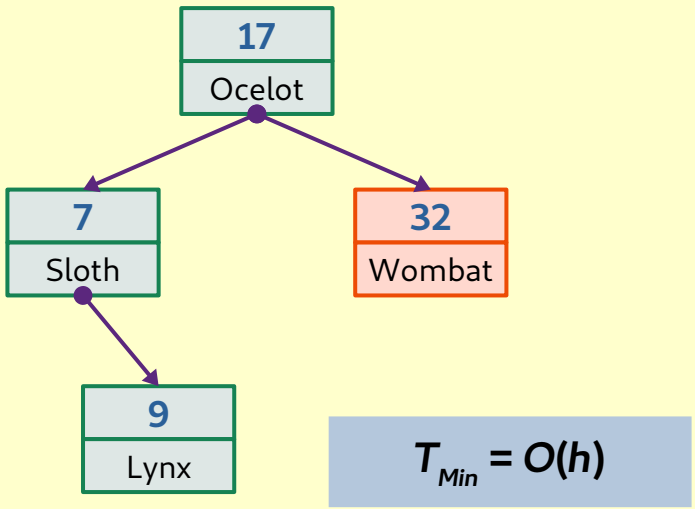
Поиск максимального элемента в BST

- 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;
}
```

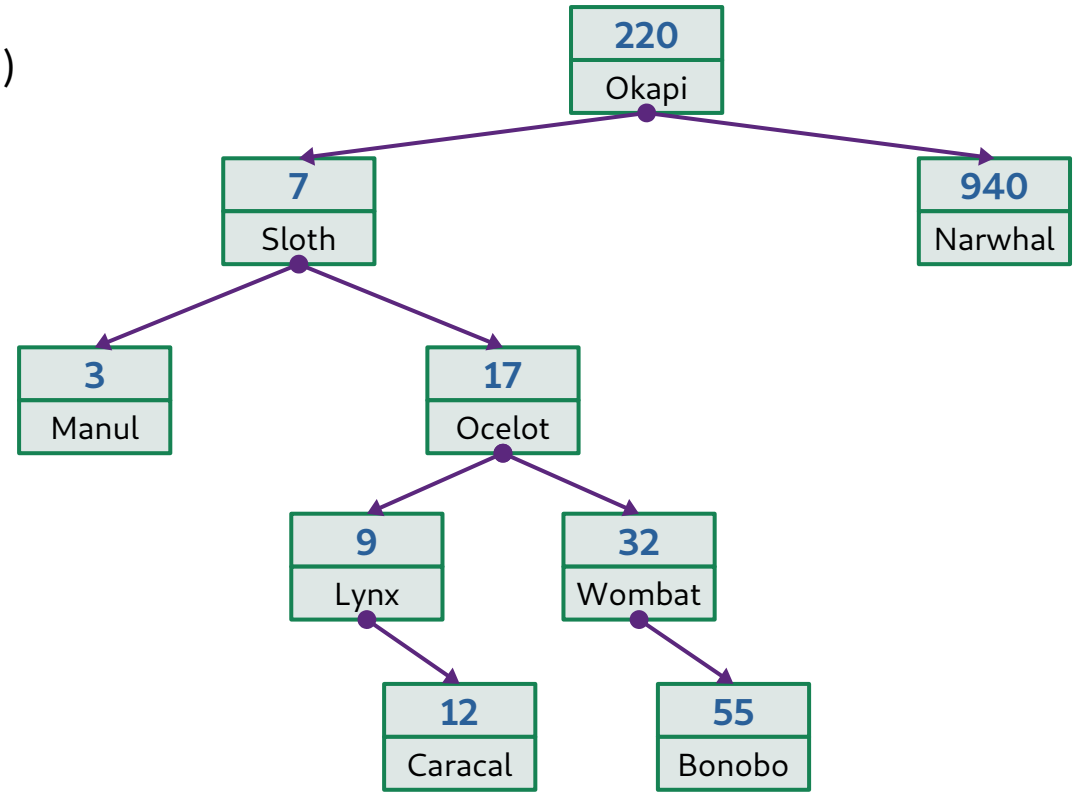


Пример использования BST

```
int main()
{
    struct bstree *tree, *node;
    tree = bstree_create(12, “Koala”);
    tree = bstree_add(tree, 4, “Flamingo”);
    tree = bstree_add(tree, 14, “Fox”);

    node = bstree_lookup(tree, 4);
    printf(“Found value for key %d: %s\n”, node->key, node->value);

    node = bstree_min(tree);
    printf(“Minimal key: %d, value: %s\n”, node->key, node->value);
    return 0;
}
```

Удаление элемента из BST

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

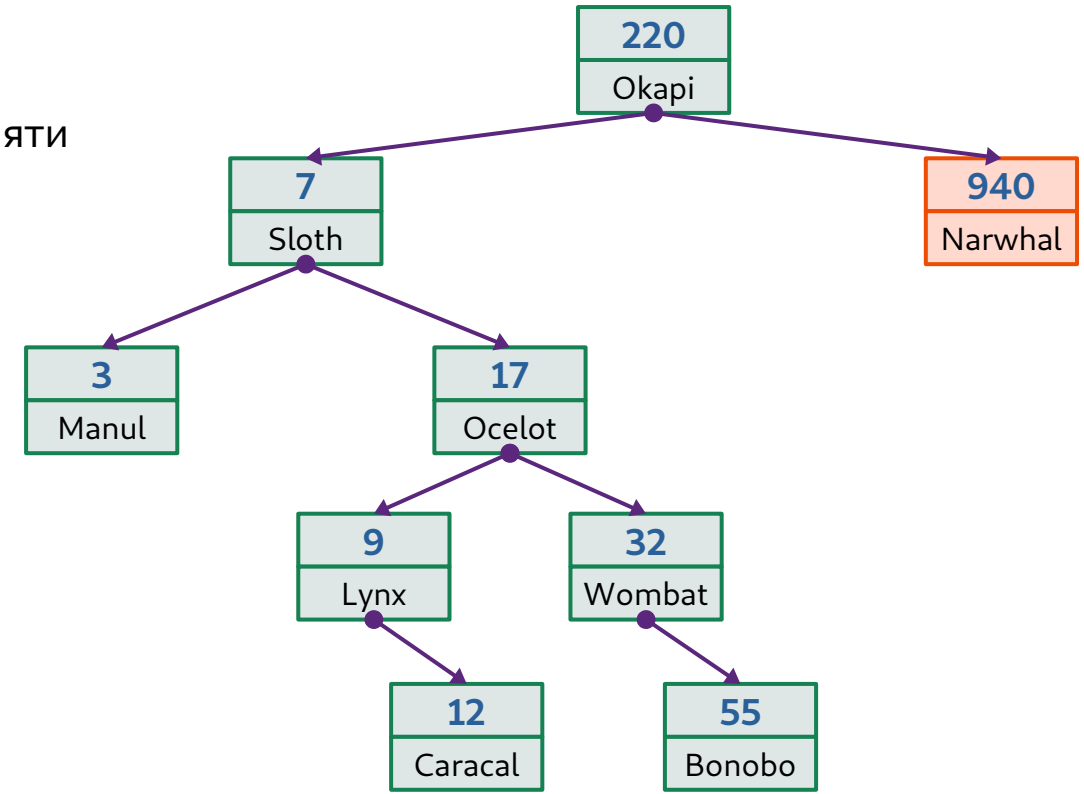


Удаление элемента из BST

Удаление узла “**Narwhal**” (случай 1)

1. Находим и удаляем узел “Narwhal” из памяти (free)
2. Родительский указатель (left или right) устанавливаем в значение NULL

“Okapi”->right = NULL

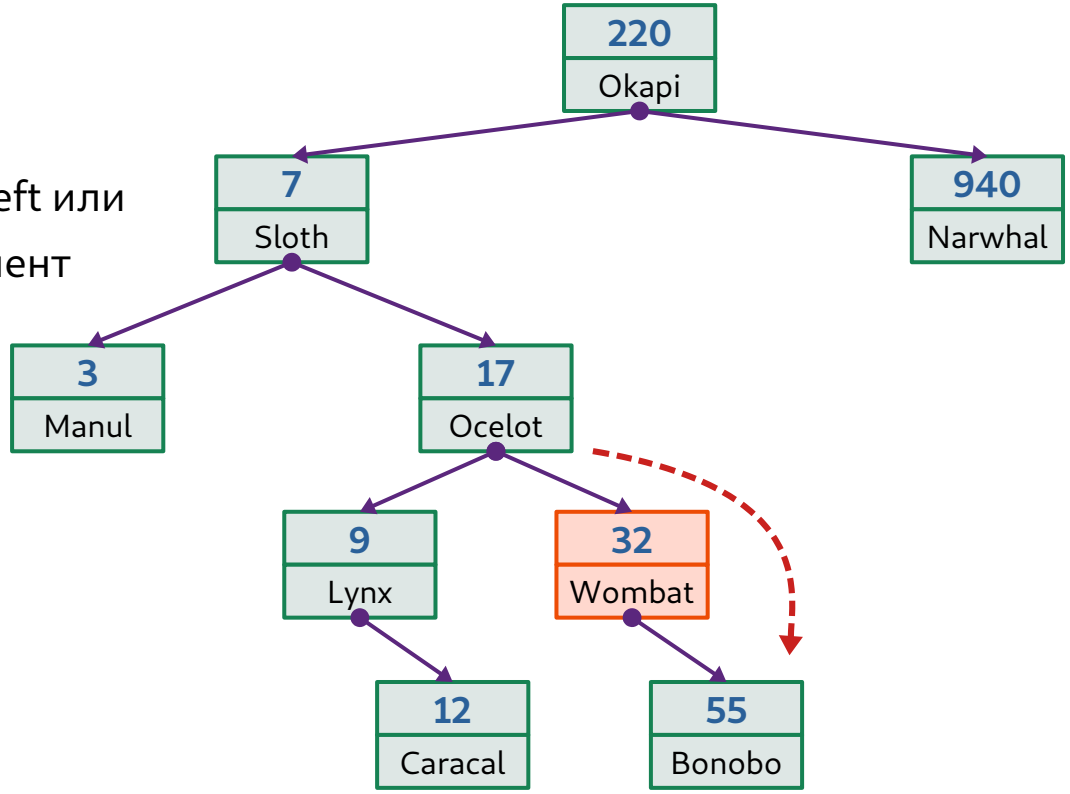


Удаление элемента из BST

Удаление узла “**Wombat**” (случай 2)

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

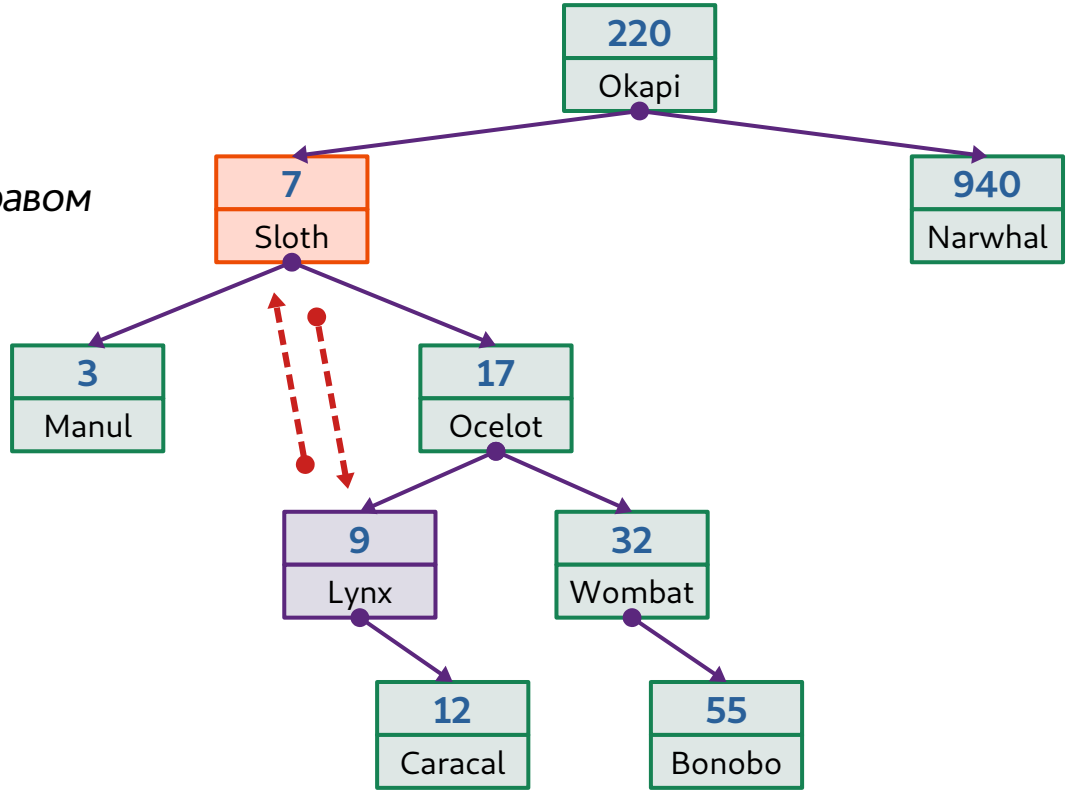
“Ocelot”->right = “Wombat”->right



Удаление элемента из BST

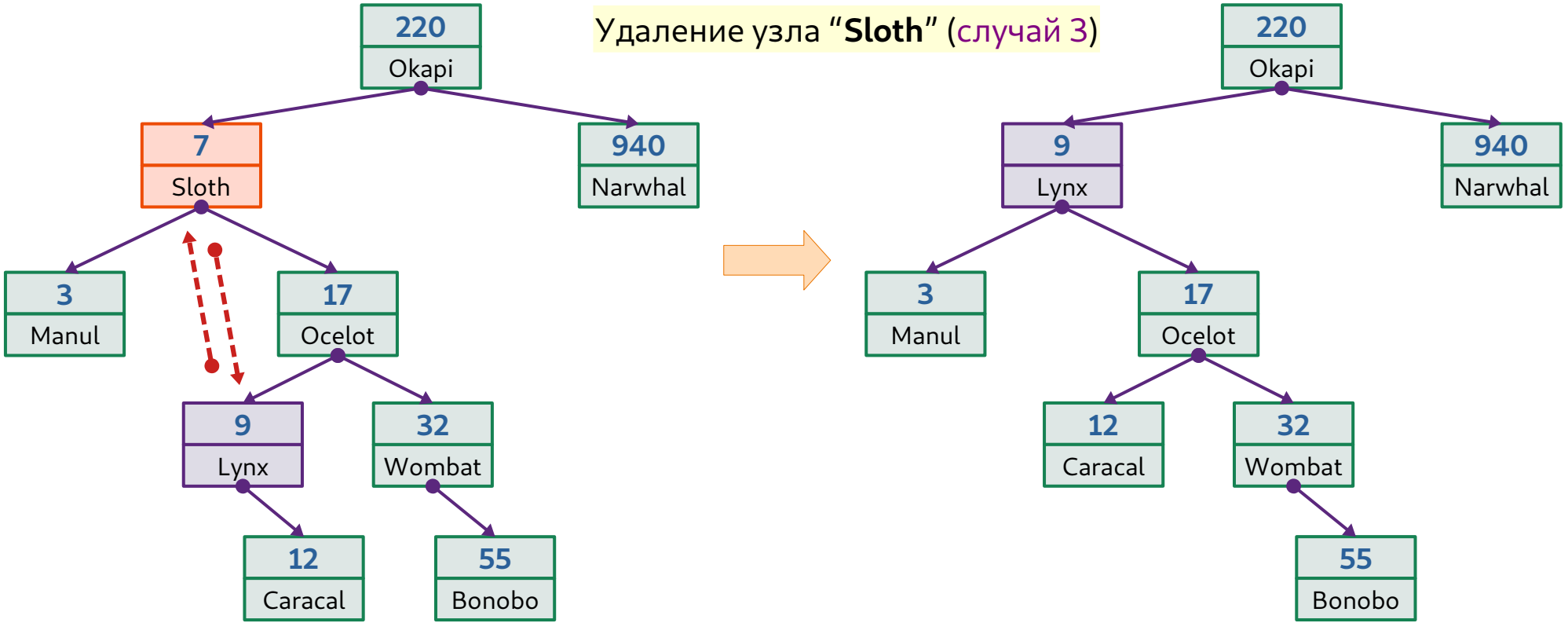
Удаление узла “**Sloth**” (случай 3)

1. Находим узел “Sloth”
2. Находим узел с минимальным ключом *в правом поддереве* “Sloth” — узел “Lynx”
3. Заменяем узел “Sloth” узлом “Lynx”



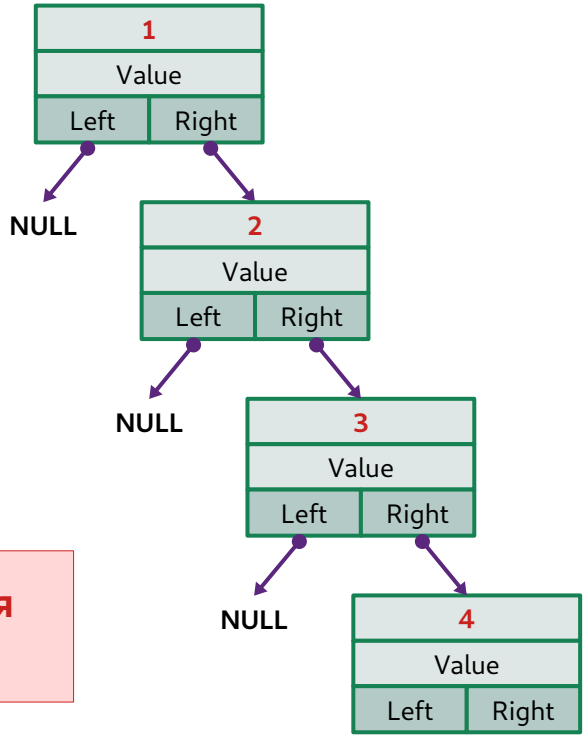
Удаление элемента из BST

Удаление узла “**Sloth**” (случай 3)



Анализ эффективности BST

- Операции бинарного дерева поиска имеют трудоёмкость, пропорциональную высоте *h* дерева
- В **худшем** случае высота дерева составляет ***O*(*n*)** — элементы добавляются в упорядоченной последовательности
- В **среднем** случае высота дерева — ***O*(log*n*)**




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

Дерево вырождается в связный список

Реализация словаря на основе бинарного дерева поиска

Операция	Средний случай (average case)	Худший случай (worst case)
Add(*map*, *key*, *value*)	$O(\log n)$	$O(n)$
Lookup(*map*, *key*)	$O(\log n)$	$O(n)$
Delete(*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 дерево поиска** (self-balancing search tree) — это дерево поиска, которое динамически корректирует свою структуру для обеспечения высоты не более $O(\log n)$
- Баланс высоты поддерживается при выполнении операций, изменяющих дерево (Insert, Delete)

- Типы сбалансированных деревьев поиска:
  - Красно-чёрное дерево (red-black tree): $h \leq 2\log_2(n + 1)$
  - АВЛ-дерево (AVL tree): $h < 1.4405 \cdot \log_2(n + 2) - 0.3277$
  - B-дерево (B-tree)
  - AA-дерево (AA tree)
  - ...

Все операции красно-чёрного дерева и АВЛ-дерева выполняются за время $O(\log n)$ в худшем случае

Домашнее чтение

- **[DSABook] Глава 9. «Бинарные деревья». Глава 10. «Бинарные деревья поиска»**
- Прочитать в «Практике программирования» [Kernighan, С. 67] раздел 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, post-order)
- Как освободить память из-под всего дерева, зная указатель на его корень?

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»
Весенний семестр, 2021 г.