Абстрактный тип данных «словарь». Основные операции словаря. Бинарные деревья поиска. Основные операции, их вычислительная сложность. Анализ эффективности бинарного дерева поиска в среднем и худшем случае.

АТД «Словарь» *(dictionary)*

- **Словарь** *(dictionary)* – это 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)
373	Кот
874	Волк
265	Рысь
123	Койот

Основные операции:

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

- 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

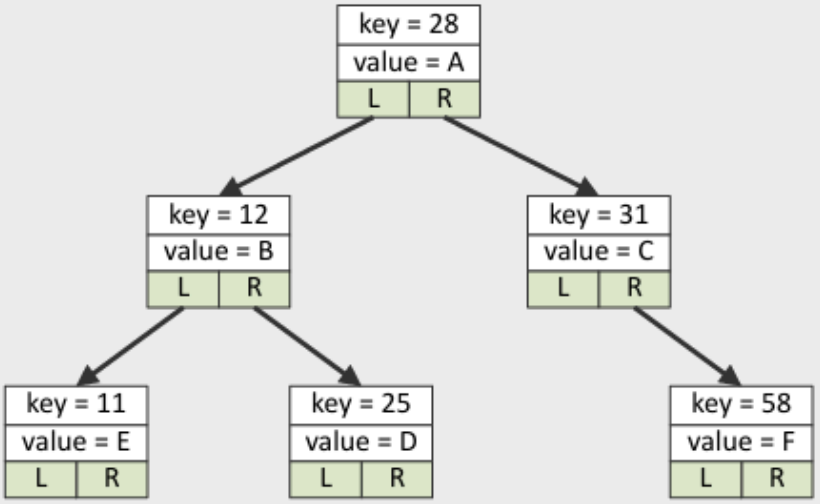
Бинарное дерево поиска

- **Бинарное дерево** *(binary tree)* – это дерево *(структура данных)*, в котором каждый узел *(node)* имеет не более двух дочерних узлов *(child nodes)*
- **Бинарное (двоичное) дерево поиска** *(binary search tree, BST)* – это двоичное дерево, в котором:
  - каждый узел x *(node)* имеет не более двух дочерних узлов *(child nodes)* и содержит ключ *(key)* и значение *(value)*
  - ключи всех узлов левого поддерева узла x меньше значения его ключа
  - ключи всех узлов правого поддерева узла x

Словарь

Ключ (key)	Значение (value)
28	A
12	B
31	C
25	D
58	E
11	F

Бинарное дерево поиска



9

```c
#include <stdio.h>
#include <stdlib.h>
struct bstree {
    int key; // Ключ
    char *value; // Данные
    struct bstree *left;
    struct bstree *right;
};
```

Создание элемента BST

```c
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;
}
```

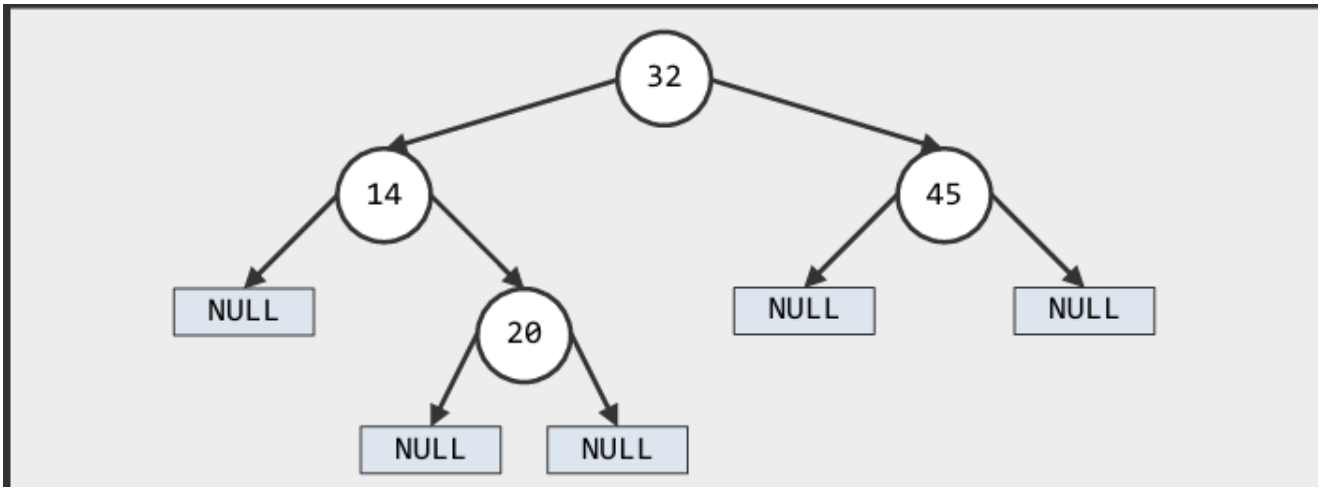
``

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

- 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

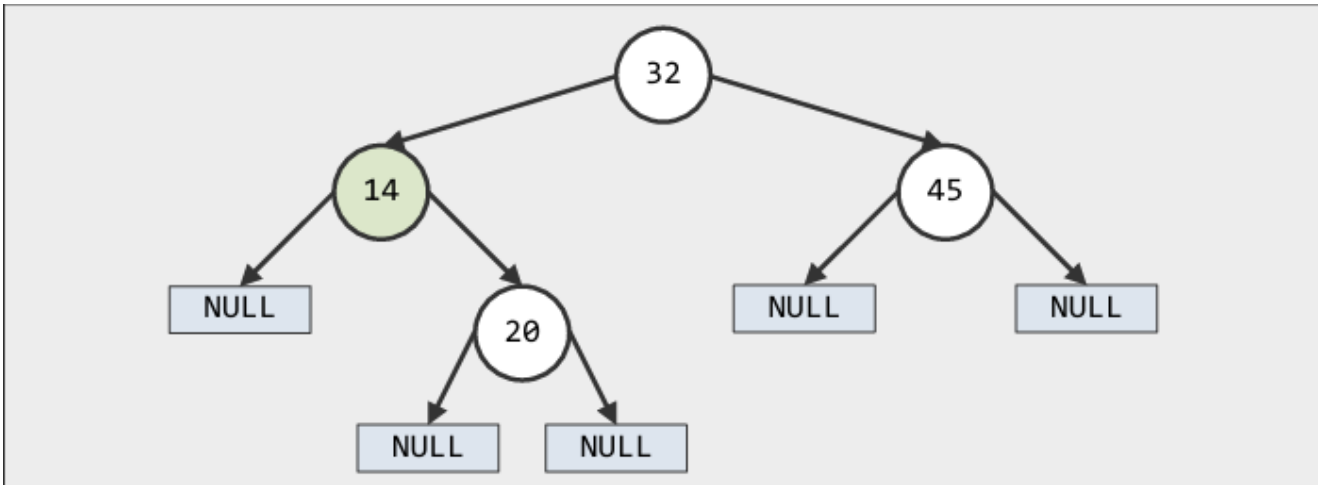


1. Сравниваем ключ корневого узла с искомым. Если совпали, то элемент найден

2. Переходим к левому или правому дочернему узлу и повторяем шаг 1
   (Возможны рекурсивная и не рекурсивная реализации)

```
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;
}
```

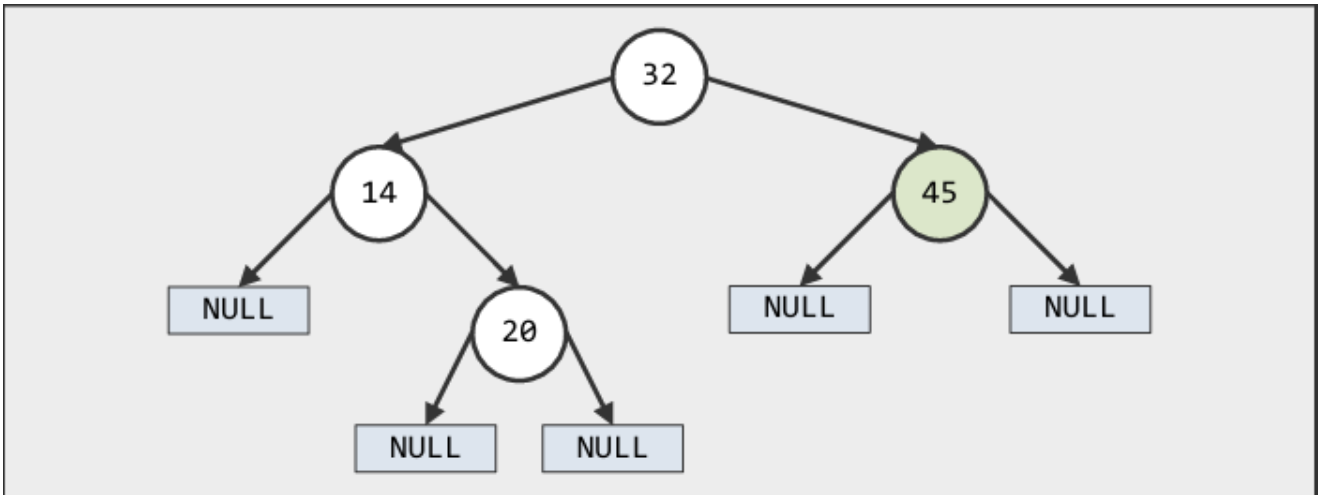
Поиск минимального элемента в BST



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

```
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



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

```
struct bstree *bstree_max(struct bstree *tree)
{
    if (tree == NULL)
        return NULL;
    while (tree->right != NULL)
        tree = tree->right;
    return tree;
}
```

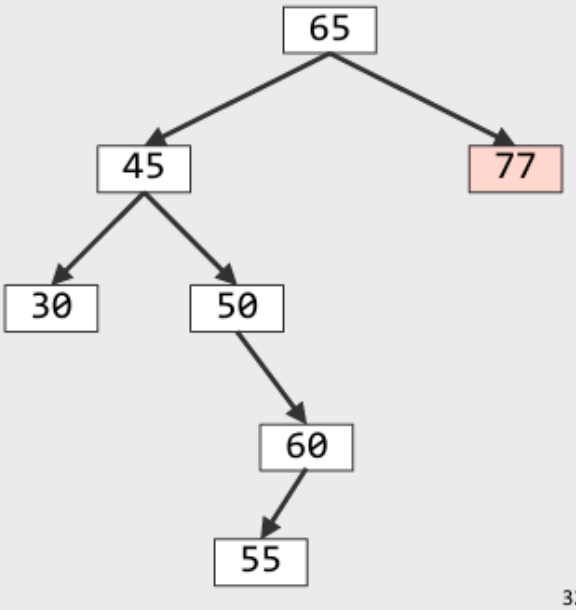
Удаление элемента из BST

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

- Удаление узла «77» (случай 1)

1. Находим и удаляем узел «77» из памяти (*free*)
2. Родительский указатель (*left* или *right*) устанавливаем в значение NULL

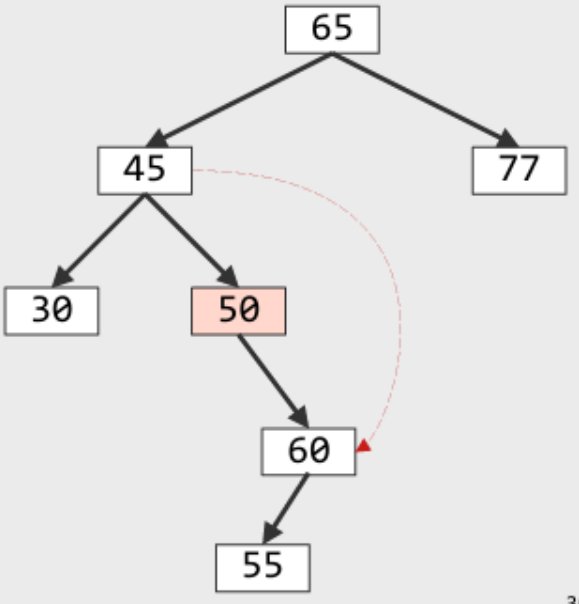
```
«65»->right = NULL
```

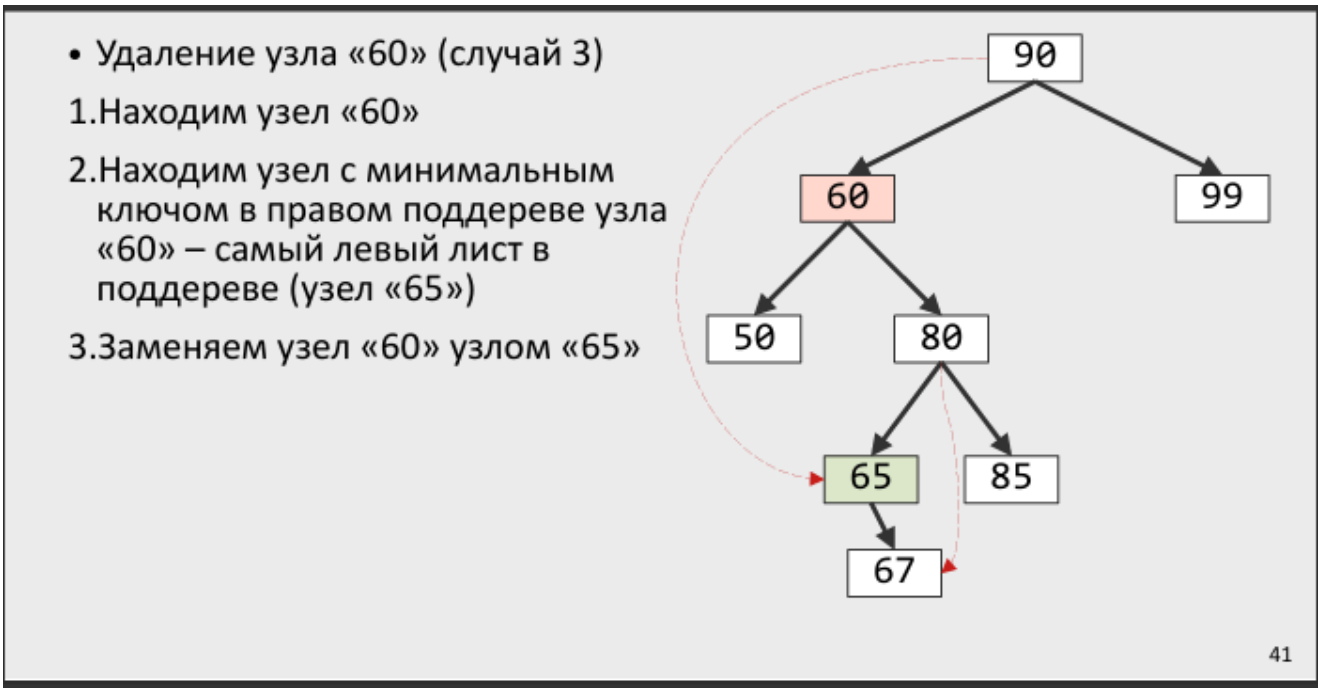


- Удаление узла «50» (случай 2)

1. Находим узел «50»
2. Родительский указатель узла «50» (*left* или *right*) устанавливаем на его дочерний элемент
3. Удаляем узел «50» из памяти

```
«45»->right = «50»->right
```





Анализ эффективности BST

1. Операции имеют трудоемкость пропорциональную высоте *h* дерева
2. В **худшем** случае высота дерева *O(n) (вставка элементов в отсортированной последовательности, дерево вырождается в связный список)*
3. В **среднем** случае высота дерева *O(logn)*

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**

Вычислительная сложность основных операций 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*)