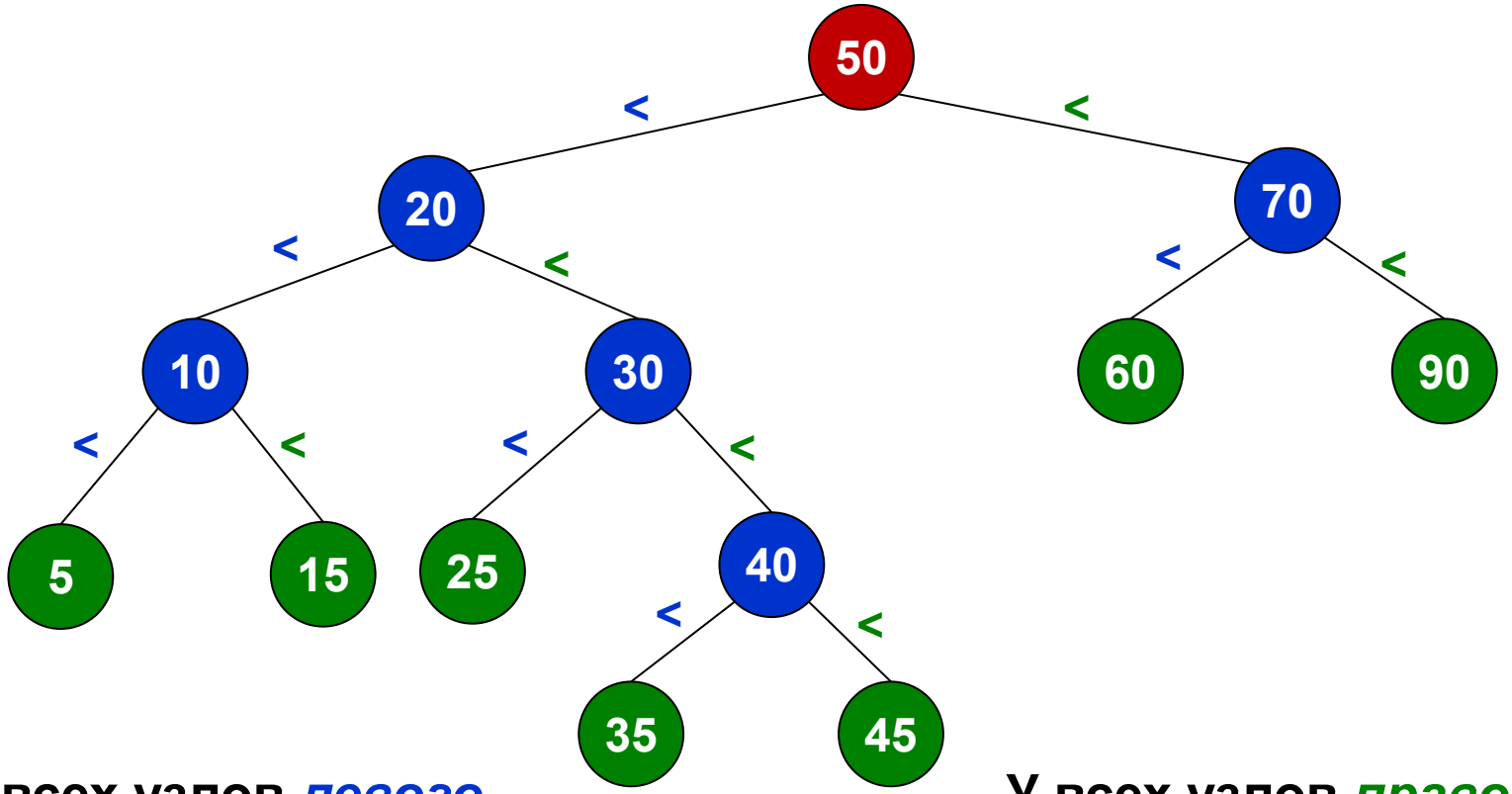
Глава 2. Деревья
п2. Двоичное дерево поиска

Определение двоичного дерева поиска

Двоичное дерево называется **деревом поиска**, если для каждого узла дерева выполняется условие:

значение ключа в узле
больше значений ключей в левом поддереве и
меньше значений ключей в правом поддереве

Пример дерева поиска



У всех узлов *левого* поддерева произвольного узла X значения ключей данных *меньше*, нежели значение ключа данных самого узла X.

У всех узлов *правого* поддерева произвольного узла X значения ключей данных *больше*, нежели значение ключа данных самого узла X

Операции над деревьями поиска

Основные операции при работе с деревьями поиска не отличаются от обычного двоичного дерева:

- Добавление узла
- Удаление узла
- Поиск узла
- Обход узлов

Главное требование при выполнении этих операций – **сохранение** указанного выше **свойства двоичного дерева поиска**:

значение ключа в узле

больше значений ключей в левом поддереве и

меньше значений ключей в правом поддереве

Добавление узла в дерево поиска

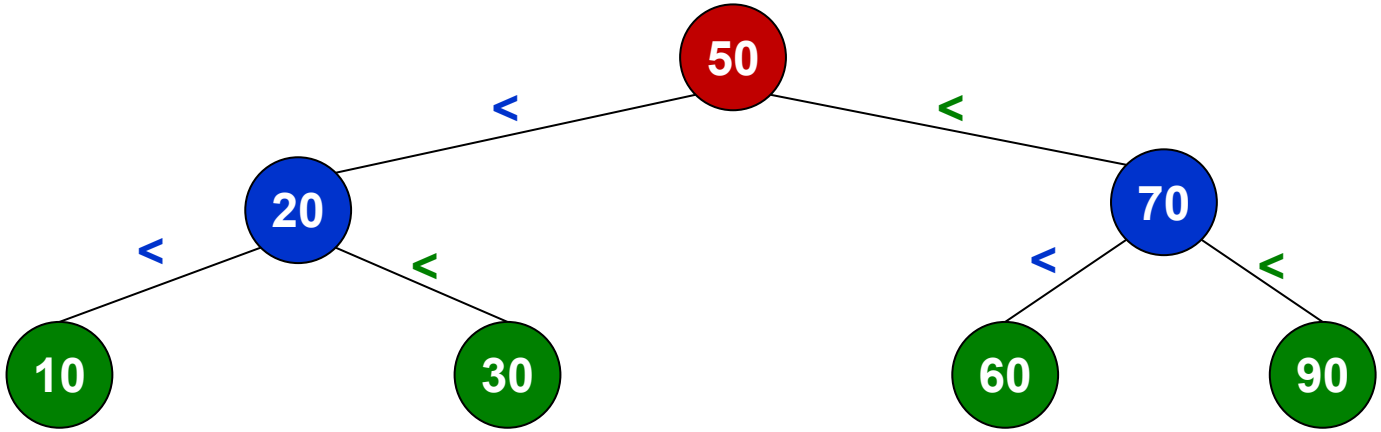
1. Если корень поддерева пуст, поместить в него заданный ключ; закончить алгоритм.
2. Если новый ключ **меньше** ключа корня: перейти в левое поддерево и повторить с п.1.
3. Иначе, перейти в правое поддерево и повторить с п.1.

Замечание: при равенстве ключей в зависимости от конкретной задачи можно либо выйти и не помещать такое же значение, либо поместить, переходя, например, вправо (как будет далее).

Добавление узла в дерево поиска

```
//ToDo: сделать метод родителя виртуальным
Node *addNode(Node *root, int key) override
{
    if (!root) {
        root = new Node(key);
    } else if (key < root->key()) {
        root->setLeftChild(addNode(root->leftChild(), key));
    } else {
        root->setRightChild(addNode(root->rightChild(), key));
    }

    return root;
}
```

Добавление узла в дерево поиска

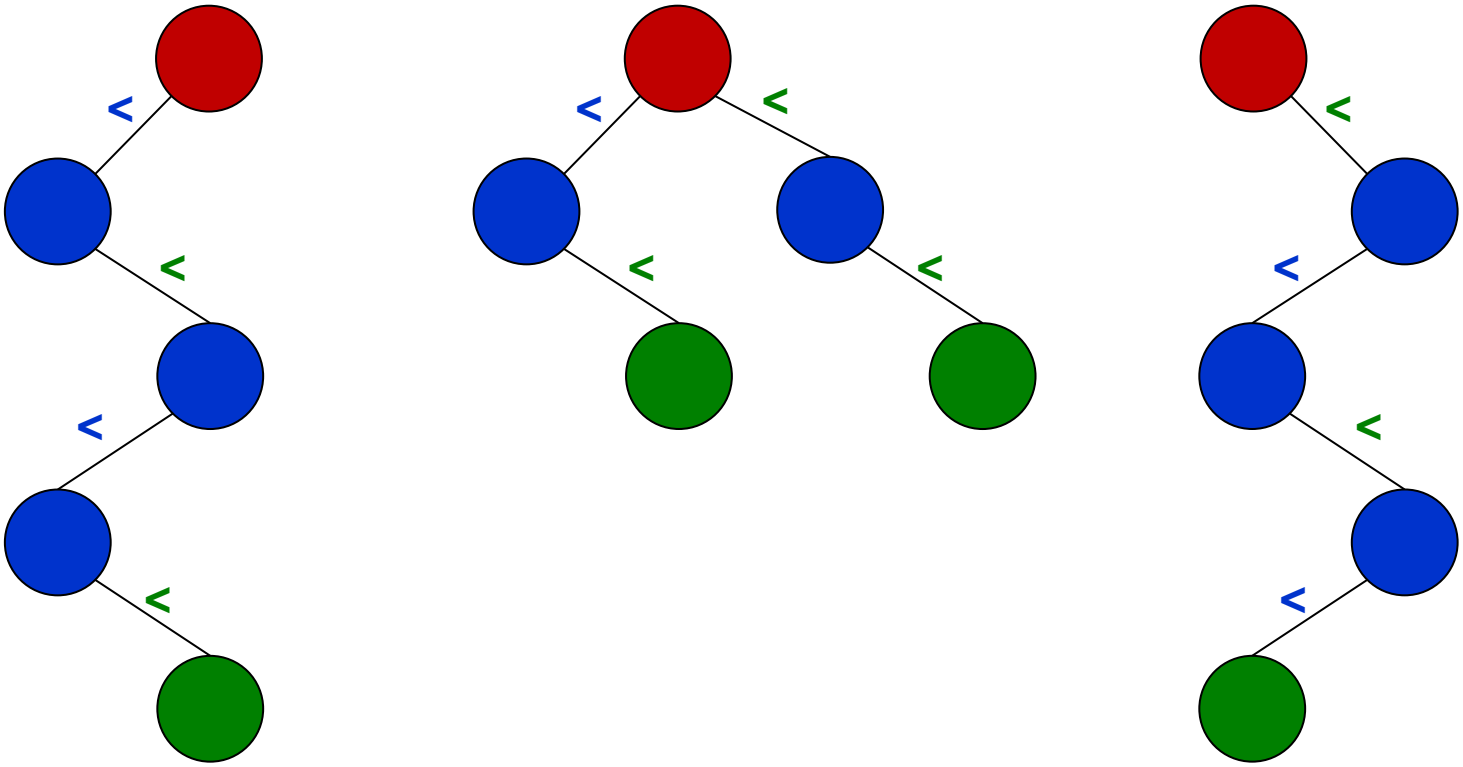


50, 70, 60, 20, 30, 90, 10

```
if (!root) {
    root = new Node(key);
} else if (root->key() < key) {
    root->setLeftChild(addNode(root->leftChild(), key));
} else {
    root->setRightChild(addNode(root->rightChild(), key));
}
```

Добавление узла в дерево поиска

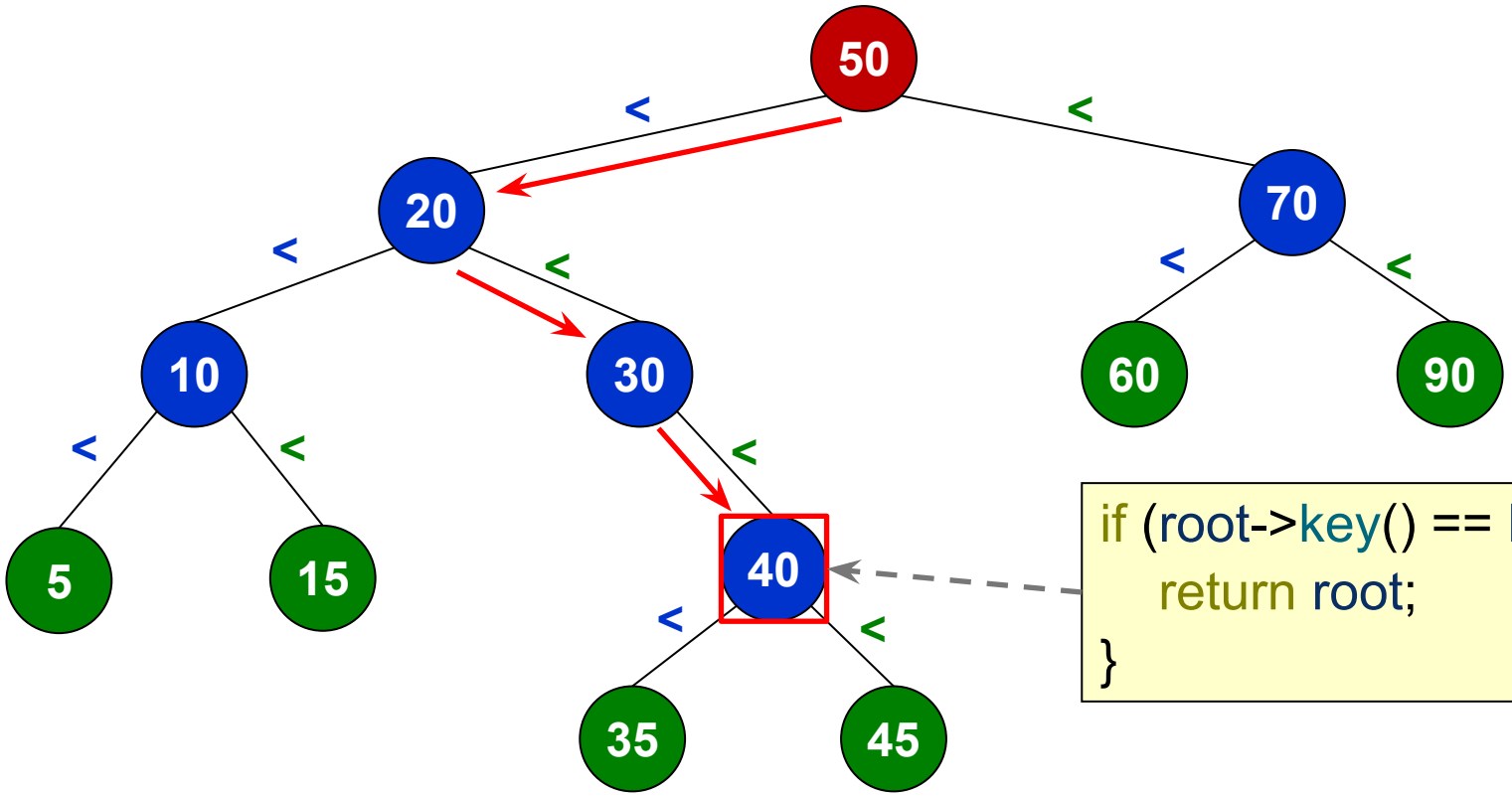
Задание: расставьте числа **10 20 30 40 50** в следующие деревья поиска:



Поиск узла в дереве поиска

```
Node *findNode(Node *root, int key)
```

key = 40

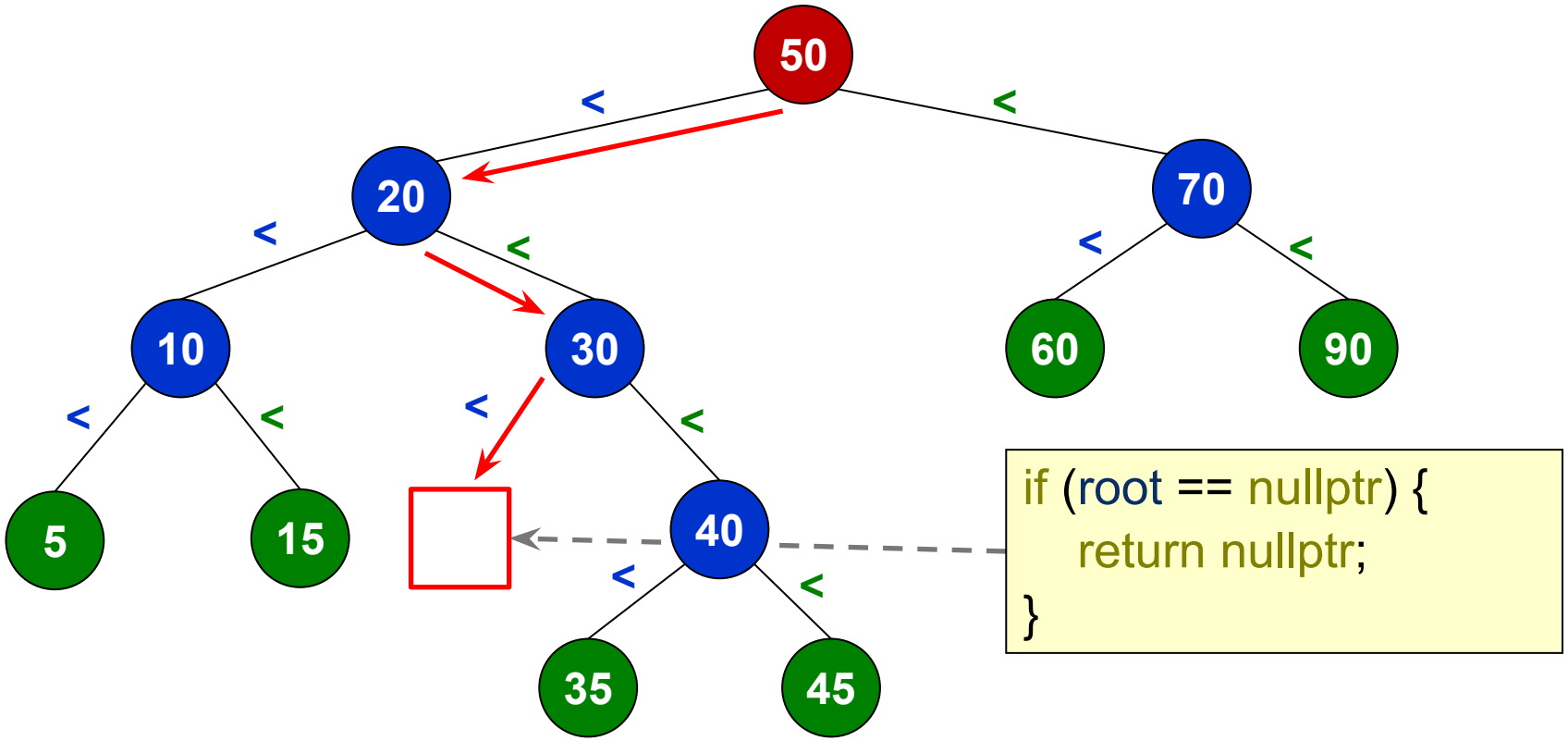



```
if (root->key() == key) {
    return root;
}
```

Поиск узла в дереве поиска

```
Node *findNode(Node *root, int key)
```

key = 25



Удаление узла из дерева поиска

Аналогично двоичному дереву, при удалении узла из дерева поиска возможны три случая:

1. Удаляемый узел не имеет поддеревьев (лист);
2. Удаляемый узел имеет одно поддерево;
3. Удаляемый узел имеет оба поддерева.

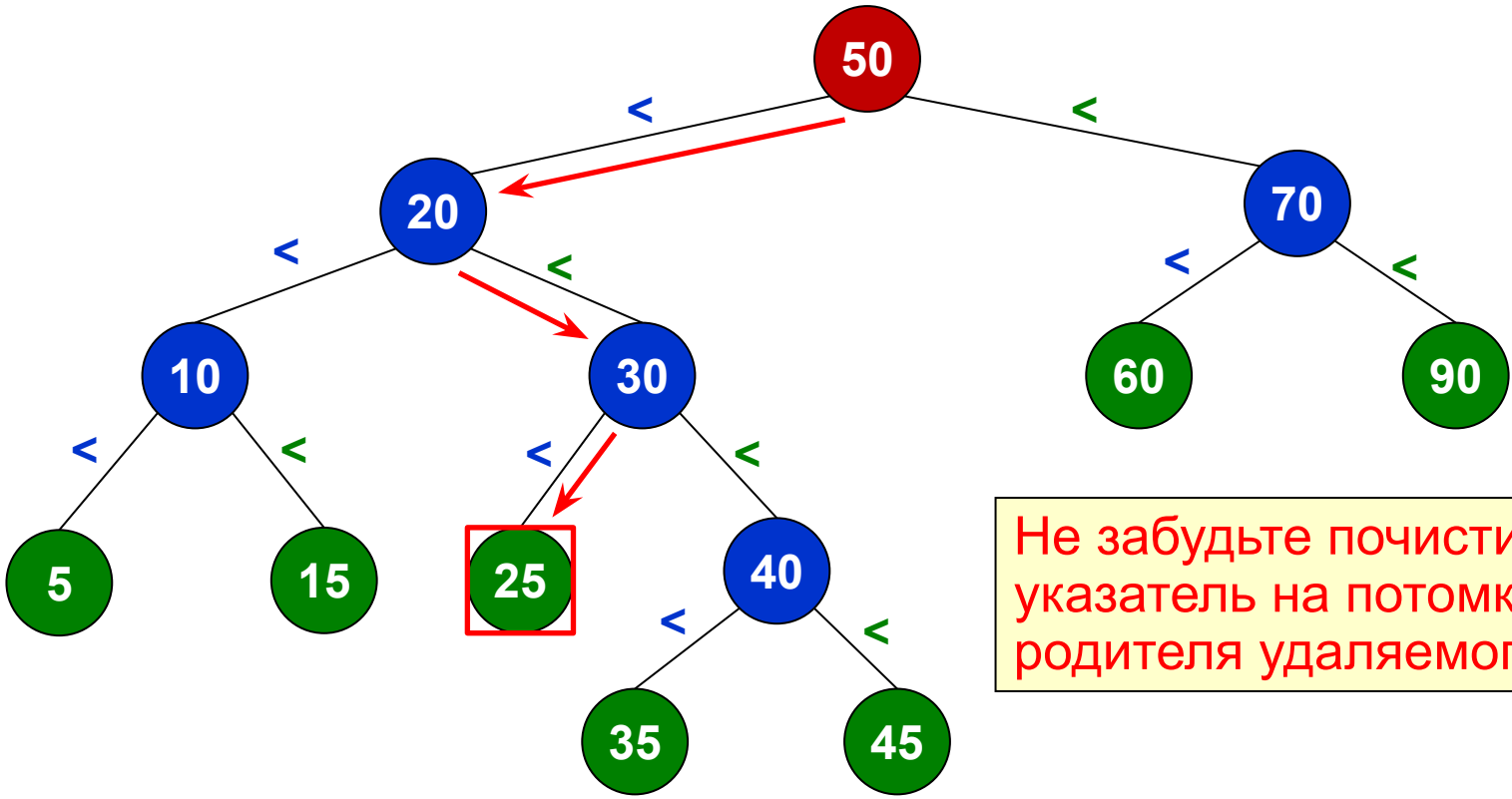
Первые два случая по реализации ничем не отличаются от рассмотренных в прошлый раз для обычного двоичного дерева, т.к. правило дерева поиска не нарушается.

Если удаляемый узел имеет оба поддерева, рекомендуется заменить его узлом, ключ в котором наиболее близок по значению (т.е. либо **больше** любого ключа в левом поддереве, либо **меньше** любого в правом)

Удаление узла из дерева поиска

```
bool removeNode(Node *root, int key)
```

key = 25

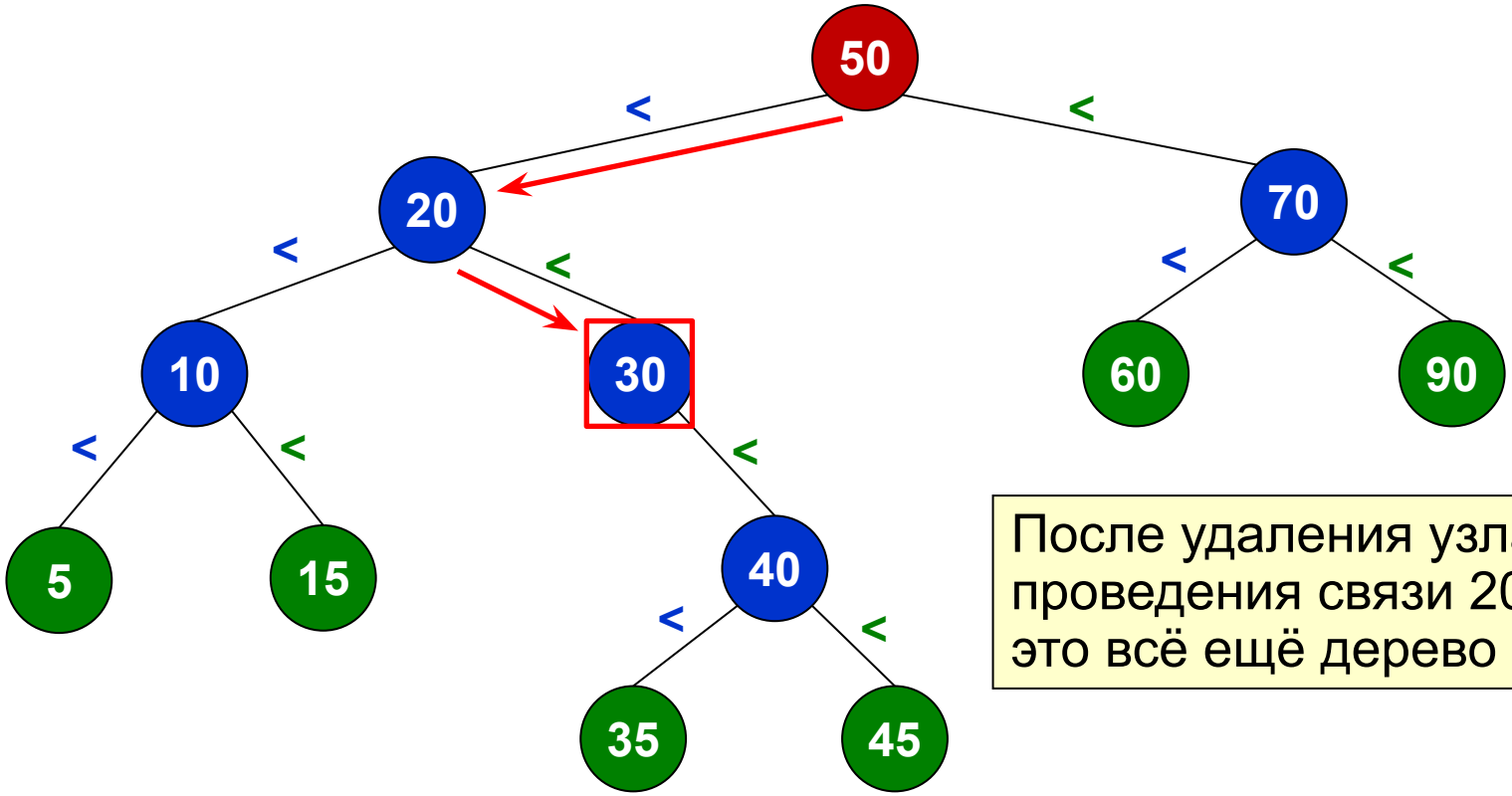


Не забудьте почистить указатель на потомка у родителя удаляемого узла!

Удаление узла из дерева поиска

```
bool removeNode(Node *root, int key)
```

key = 30

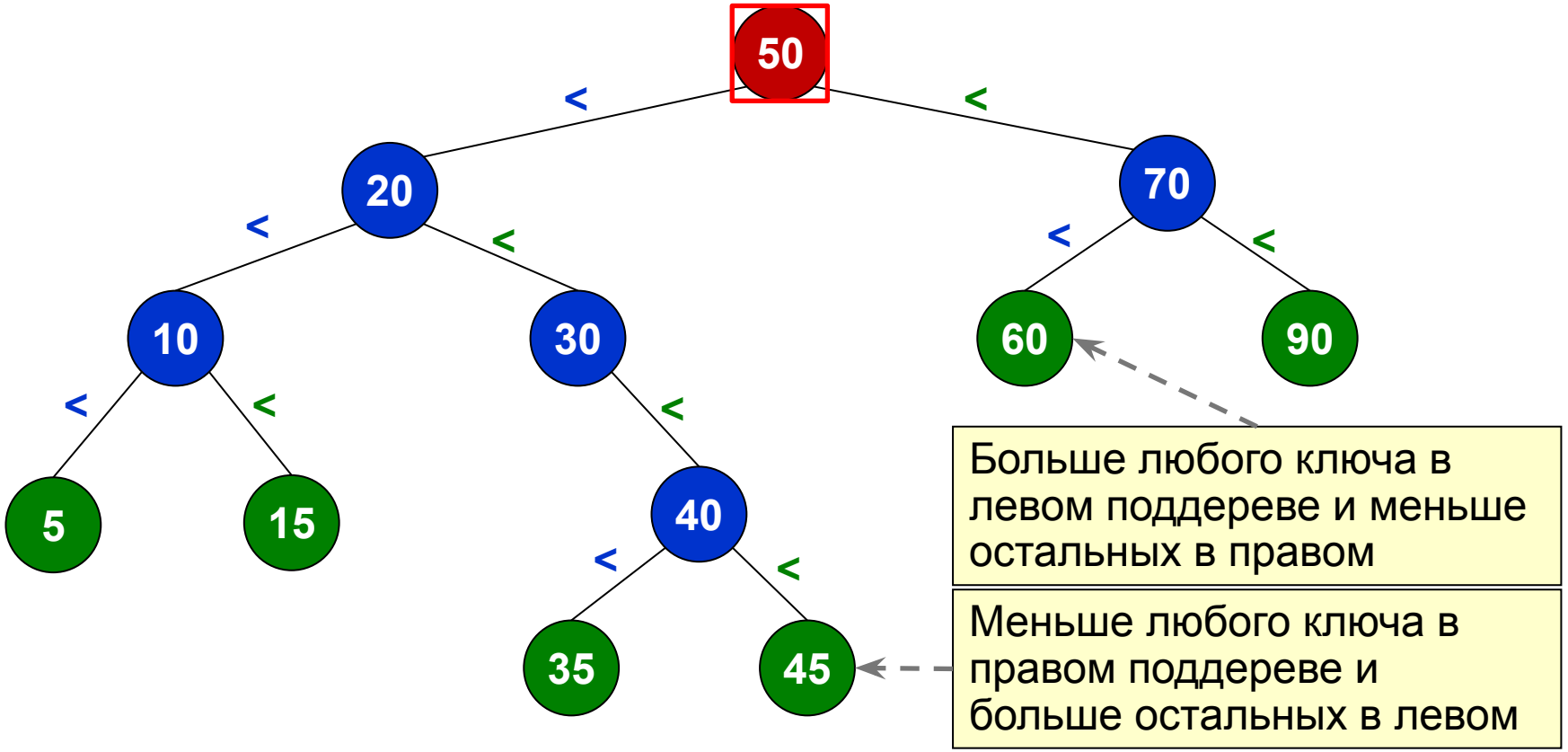


После удаления узла 30 и проведения связи 20 - 40 это всё ещё дерево поиска!

Удаление узла из дерева поиска

```
bool removeNode(Node *root, int key)
```

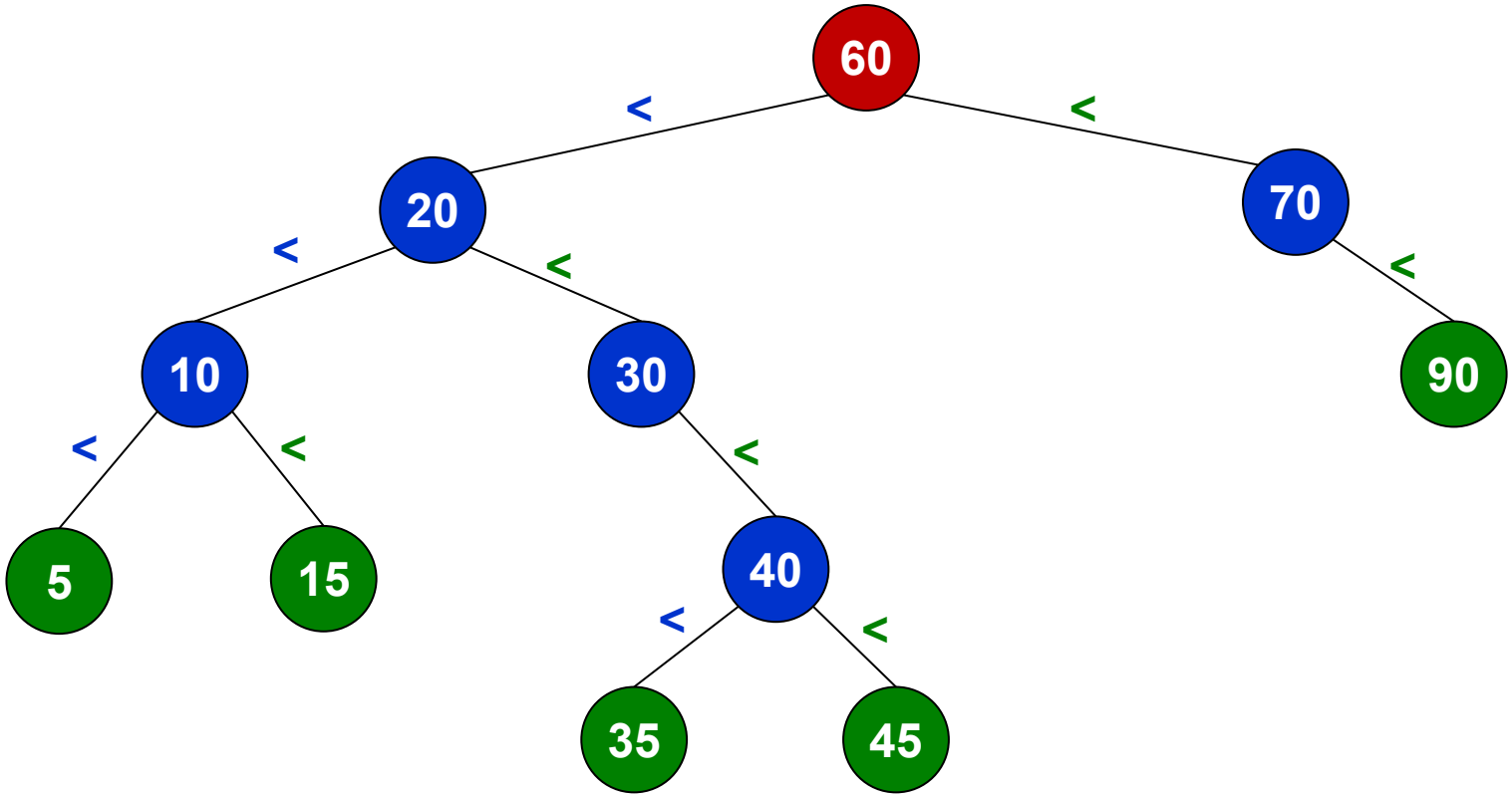
key = 50, до удаления



Удаление узла из дерева поиска

```
bool removeNode(Node *root, int key)
```

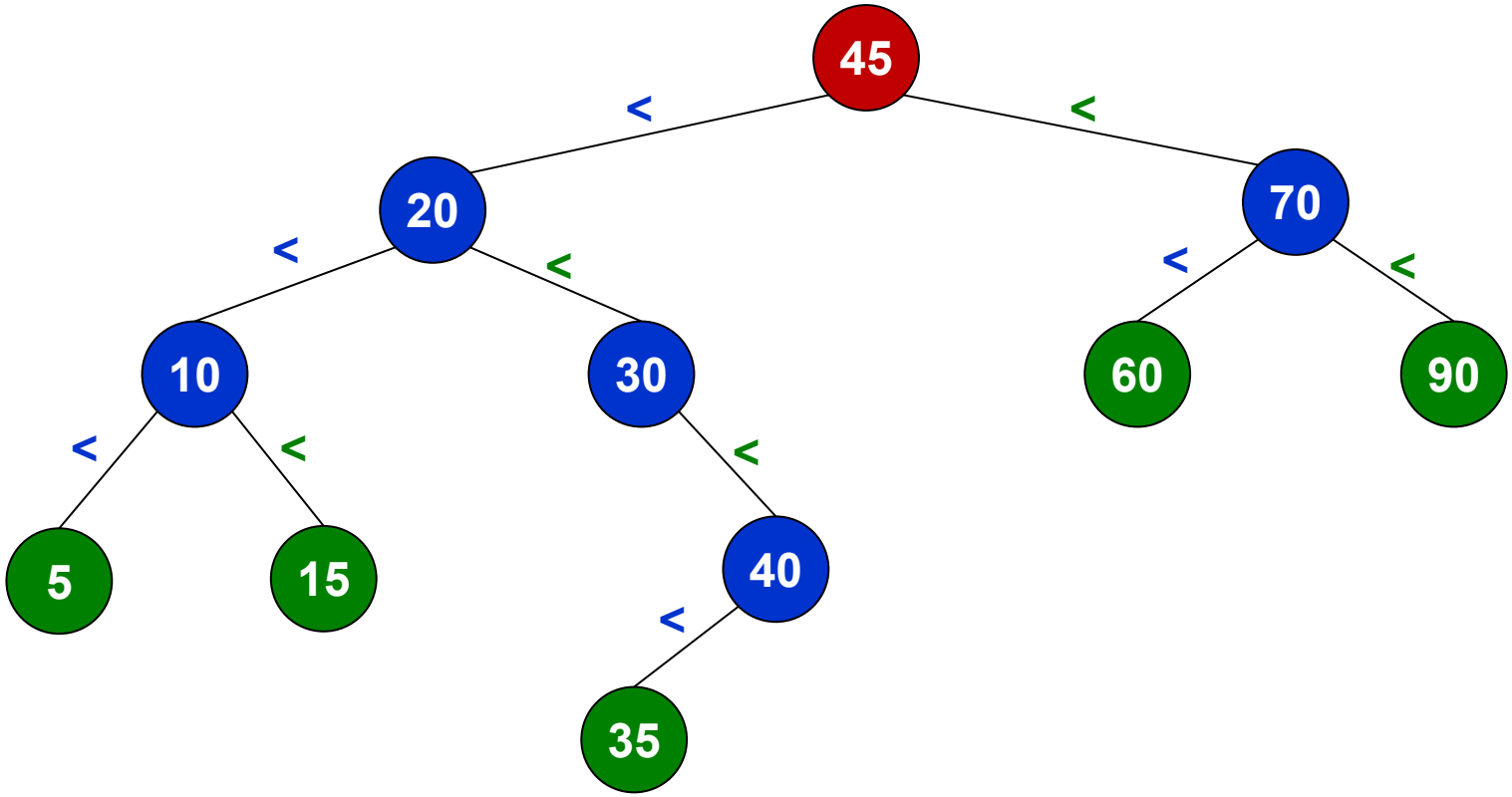
key = 50, замена на минимум из правого поддерева



Удаление узла из дерева поиска

```
bool removeNode(Node *root, int key)
```

key = 50, замена на максимум левого поддерева

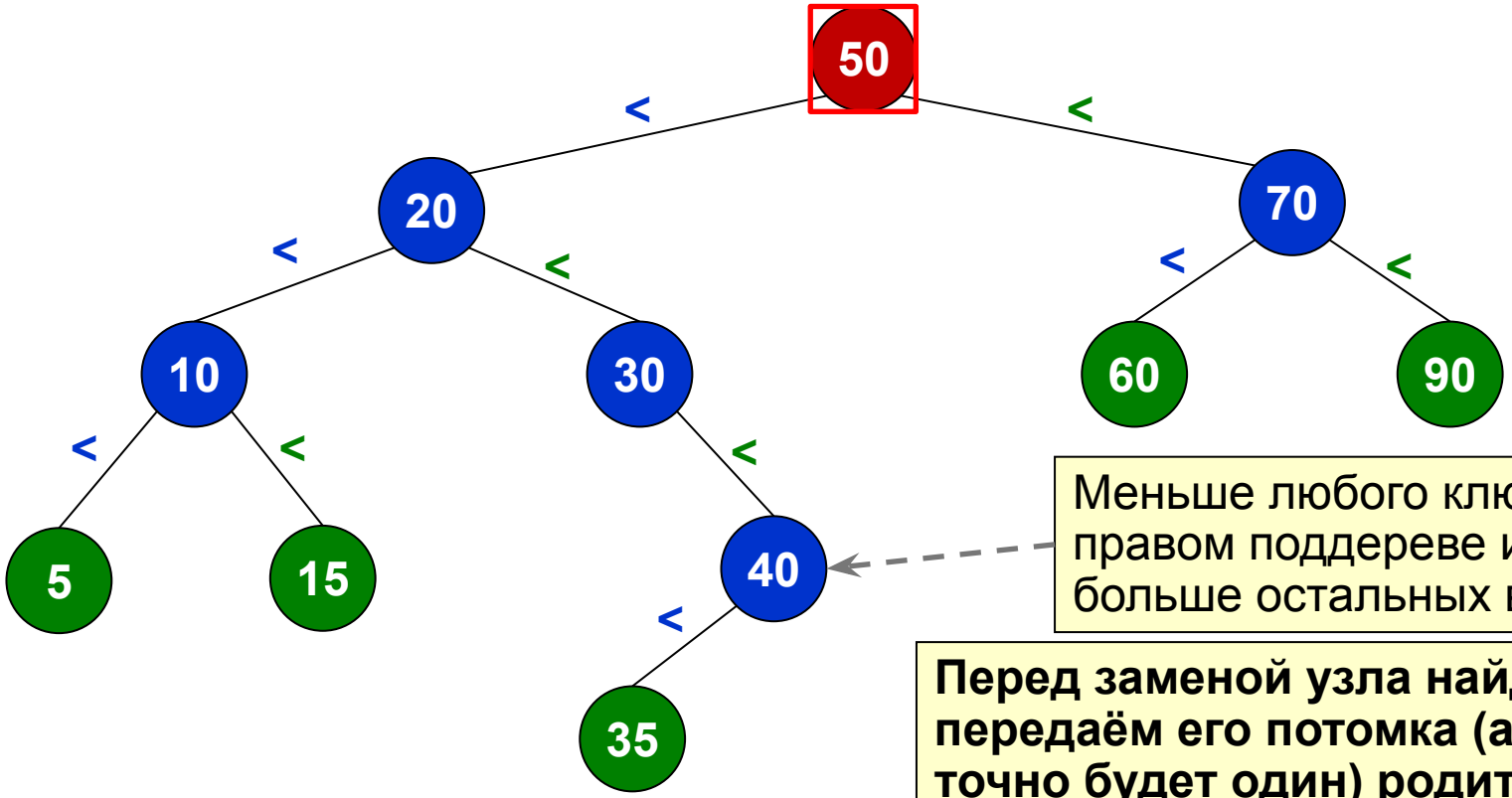


Удаление узла из дерева поиска

```
bool removeNode(Node *root, int key)
```

key = 50, до удаления

Пример с заменой не на лист



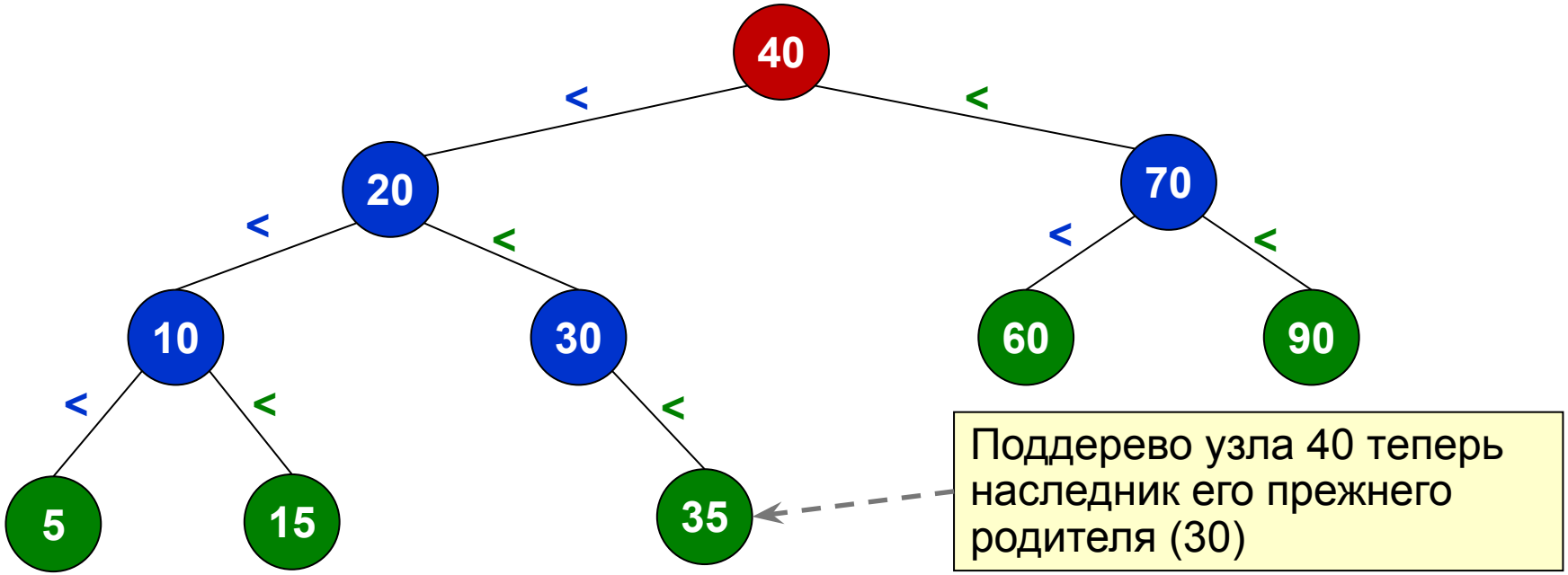
Меньше любого ключа в правом поддереве и больше остальных в левом

Перед заменой узла найденным передаём его потомка (а он точно будет один) родителю

Удаление узла из дерева поиска

```
bool removeNode(Node *root, int key)
```

key = 50, замена на максимум левого поддерева



Удаление узла из дерева поиска

Алгоритм определения узла **r**, который заменит удаляемый узел **n**:

1. Тривиальные случаи: у **n** есть только один потомок. Тогда **r** равен этому потомку. Алгоритм окончен, дополнительных действий не требуется.
2. Спуск по правой ветви: **r** = **n->rightChild()**.
3. Если **r->leftChild() == nullptr**, замена найдена. Установить левого потомка **n** в качестве левого потомка **r**. Алгоритм окончен.
4. Поиск в поддереве с корнем **r->leftChild()** узла, у которого нет левого потомка (далее **r** указывает на найденный узел, **rp** указывает на его родителя).
5. Передаём правое поддерево узла **r** его родителю: **rp->setLeftChild(r->rightChild())**;
6. Передаём потомков узла **n** узлу **r**. Алгоритм окончен.

*Всё, что останется для завершения удаления – заменить **n** на **r** в родителе **n** и очистить память **n**.*

Удаление узла из дерева поиска

Замечание: аналогично можно выполнить удаление ключа, спускаясь по левой ветви от **n** и найдя самый правый узел (с пустым правым потомком), т.е. поиск наиболее близкого ключа с меньшим значением, чем **n->key()**. Далее выполнить симметричные действия.

Замечание 2: т.к. при поиске потомка спуск всегда происходит только по одной ветви (левой/правой), он легко реализуется обычным циклом.

Использование функции удаления

Пусть функция удаления узла имеет сигнатуру:

```
Node *removeNode(Node *node)
```

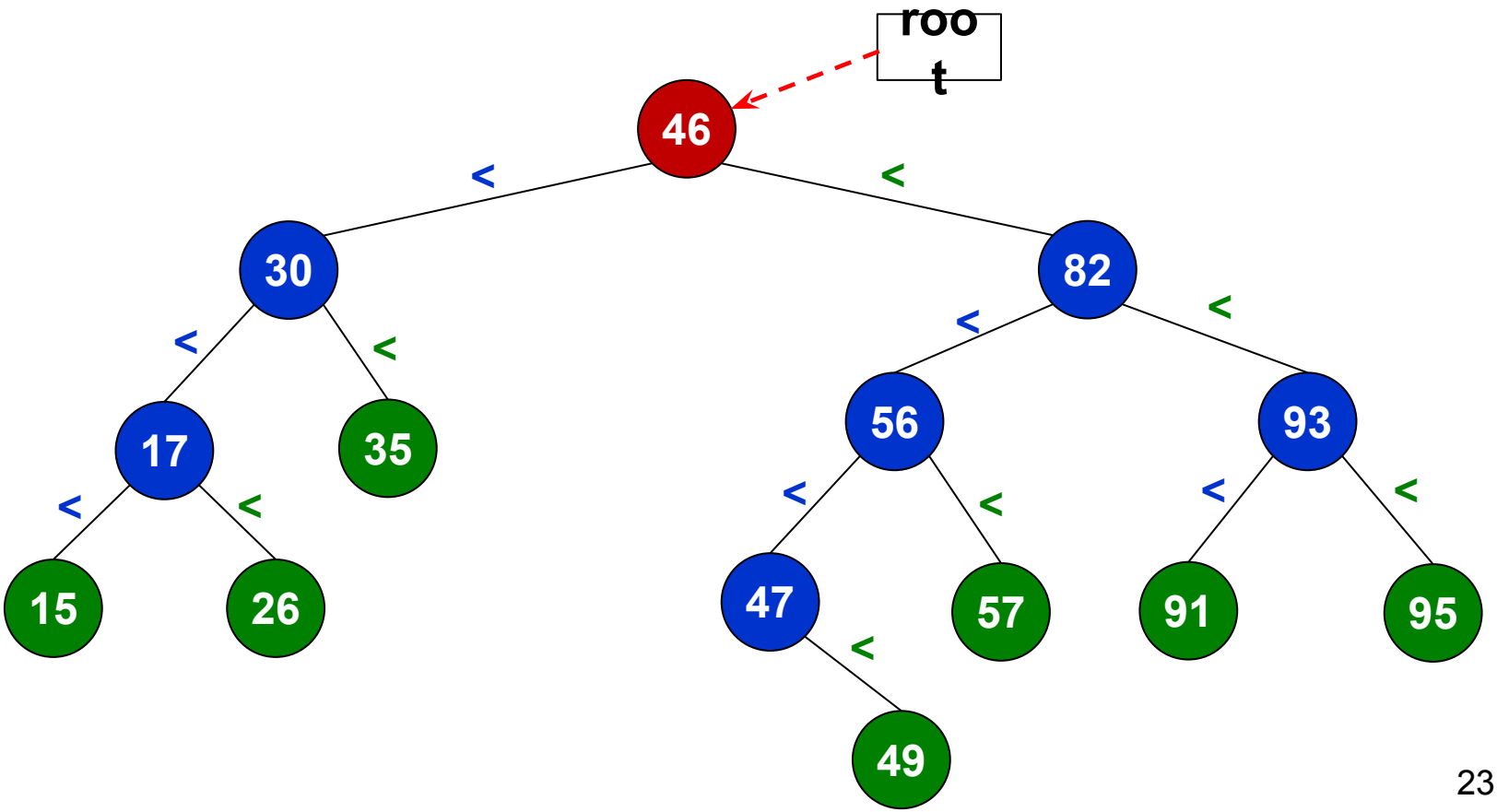
И удаляет переданный узел, возвращая указатель на узел, который его заменил.

Как применить эту функцию для удаления всех узлов с чётными ключами из поддерева?

Использование функции удаления

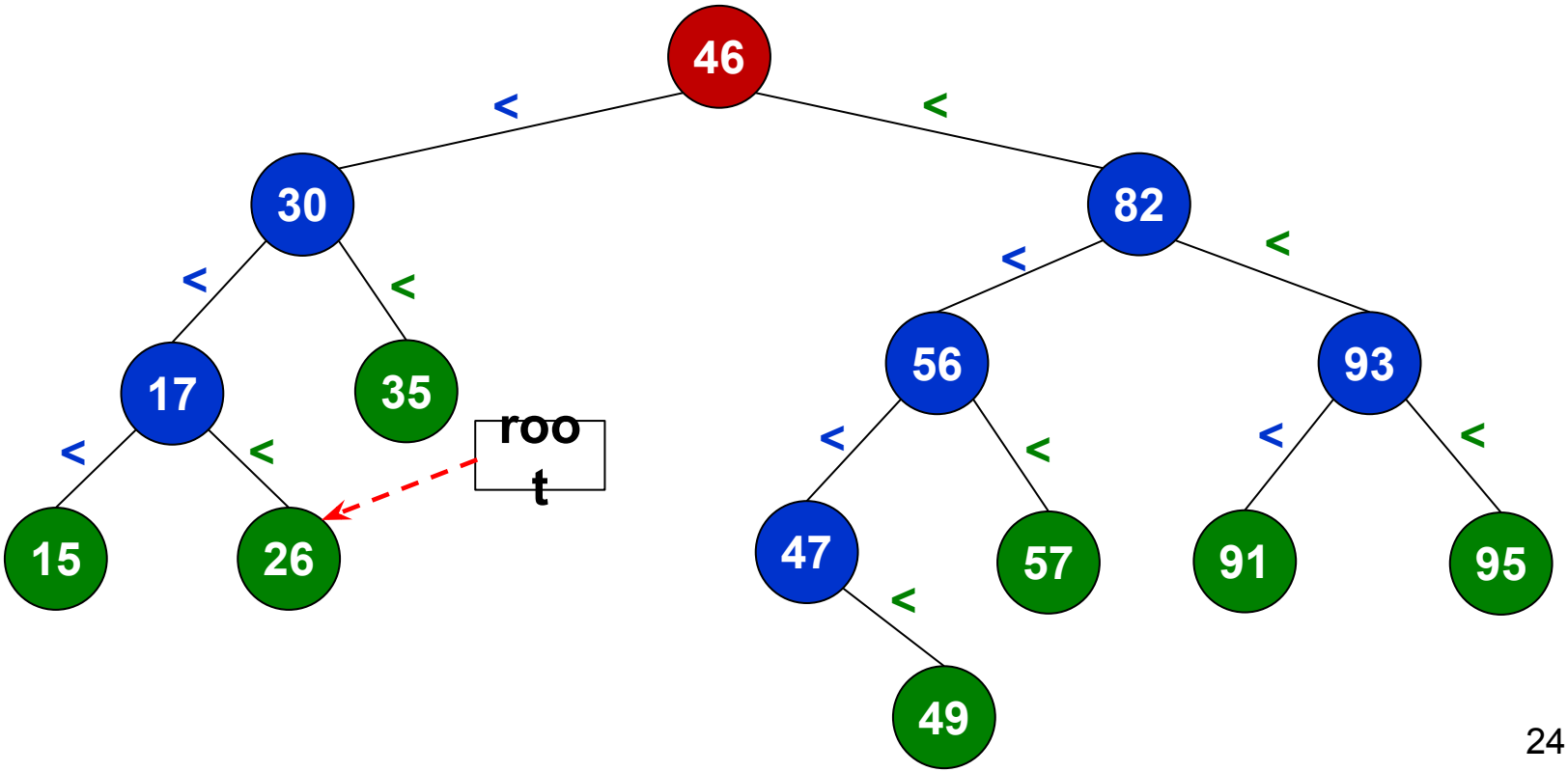
```
Node *removeEvenNodes(Node *root)
{
    if (root == nullptr) {
        return nullptr;
    }

    root->setLeftChild(removeEvenNodes(root->leftChild()));
    root->setRightChild(removeEvenNodes(root->rightChild()));
    if (root->key() % 2 == 0) {
        return removeNode(root);
    }
    return root;
}
```

Использование функции удаления

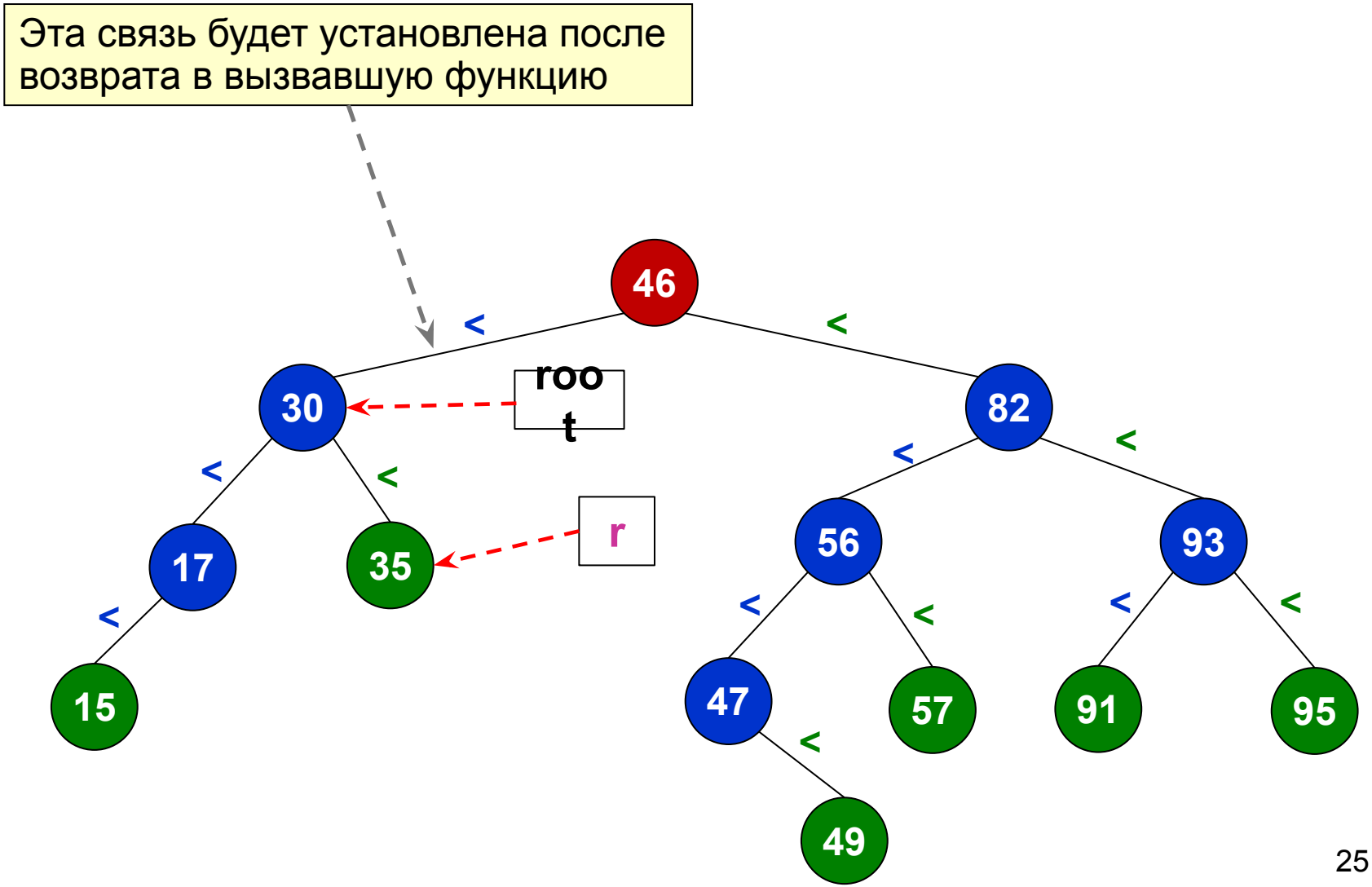


Использование функции удаления

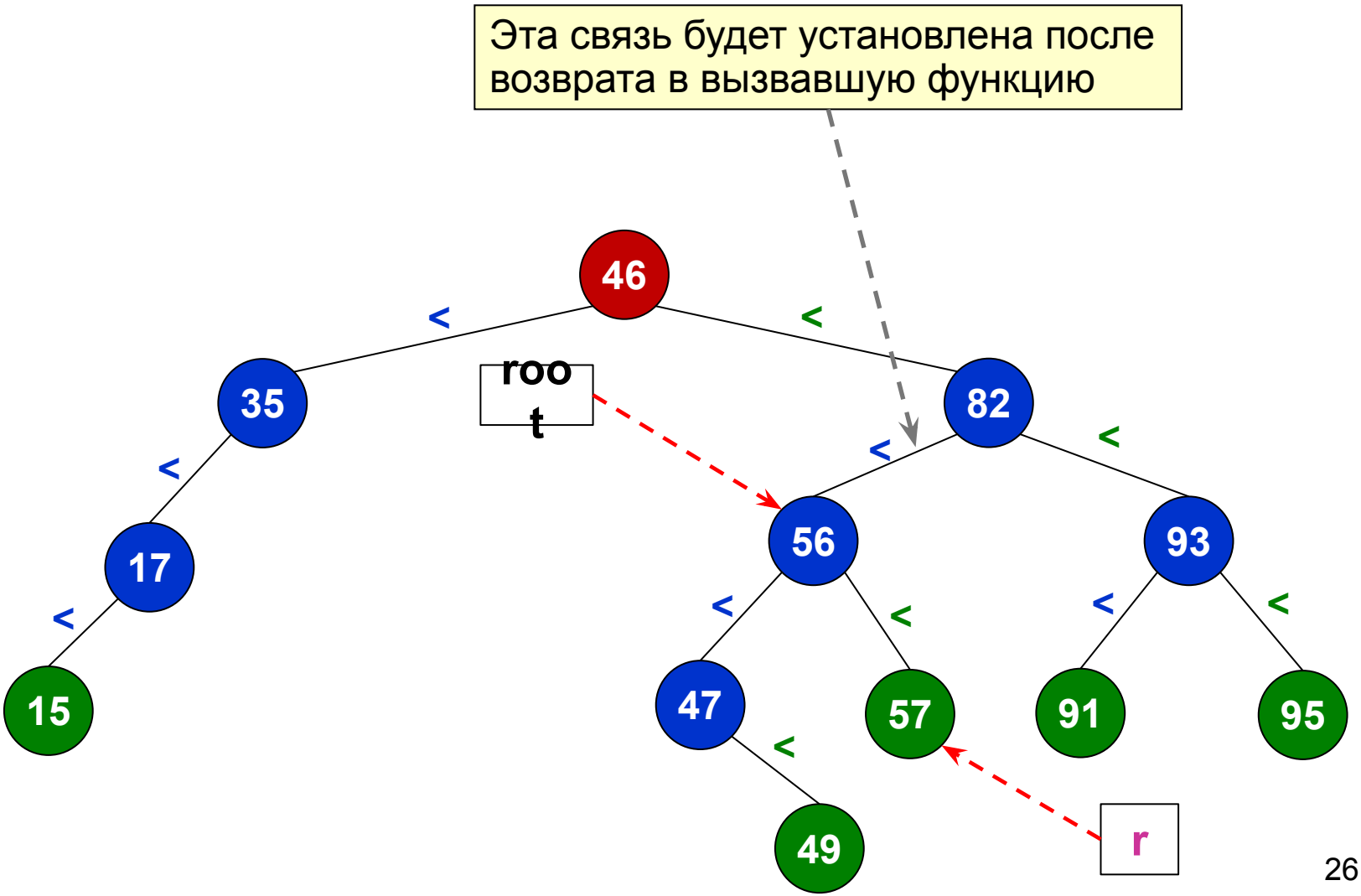
Спустя три рекурсивных вызова, один возврат и ещё один рекурсивный вызов будет удалена первая из вершин



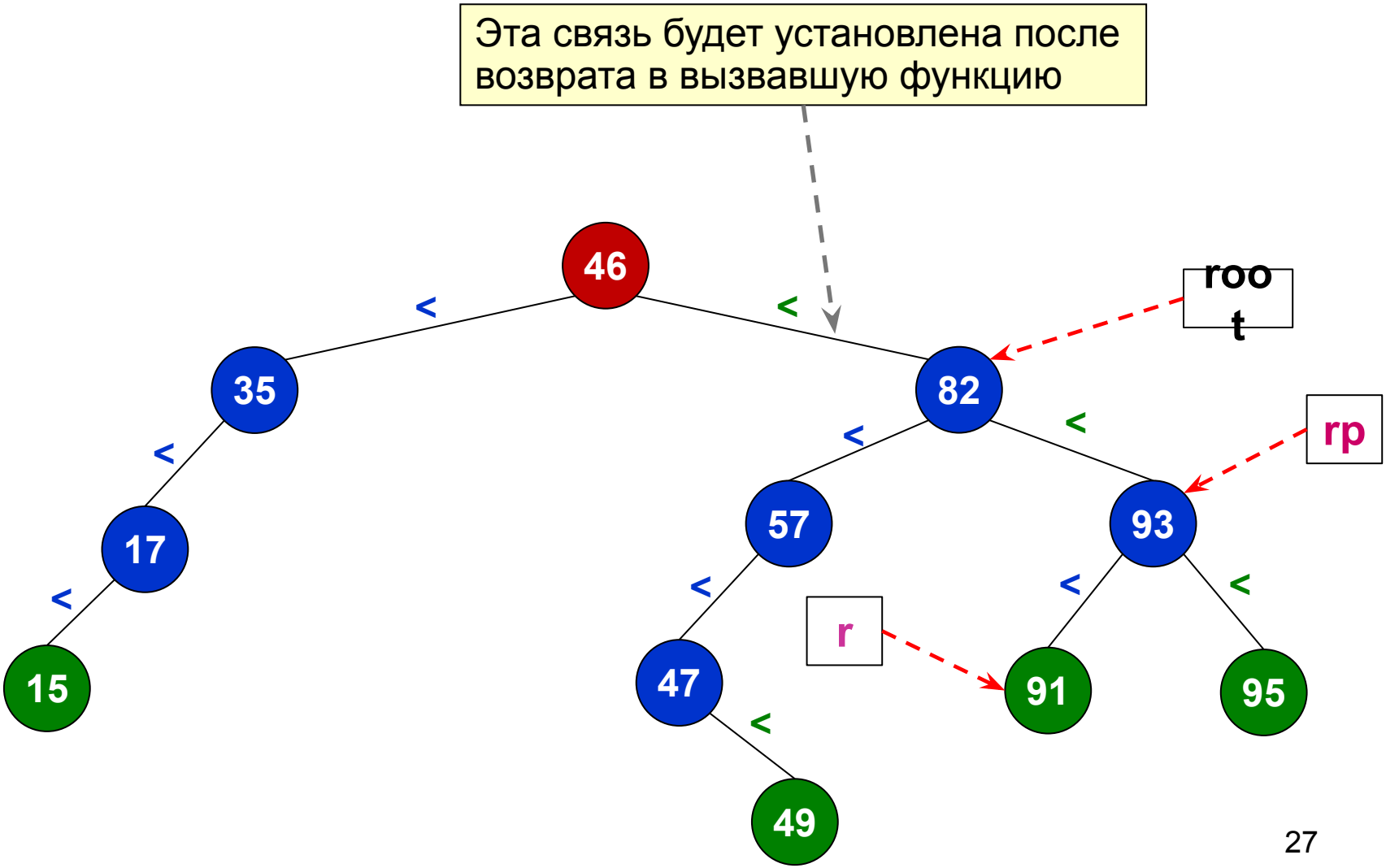
Использование функции удаления



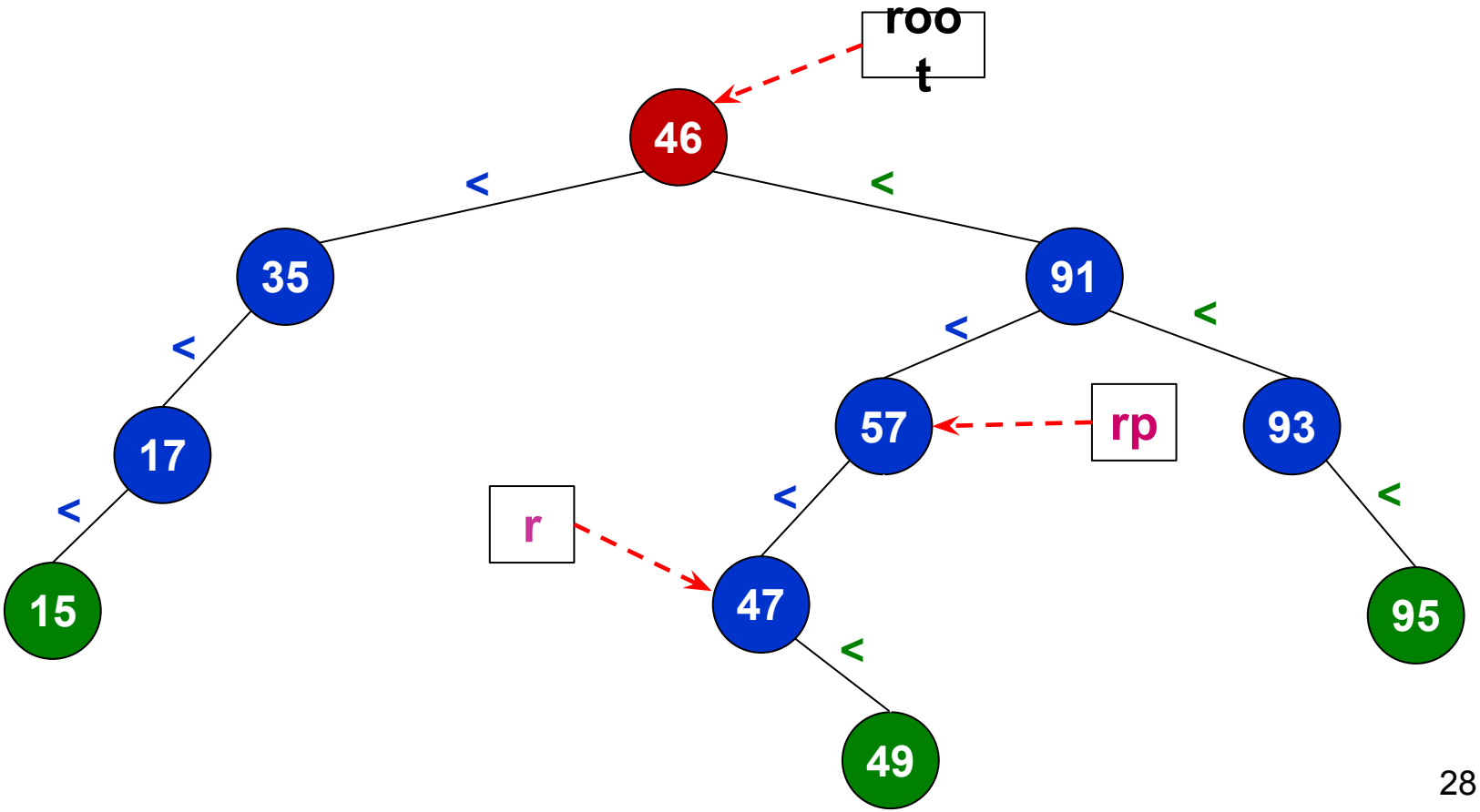
Использование функции удаления



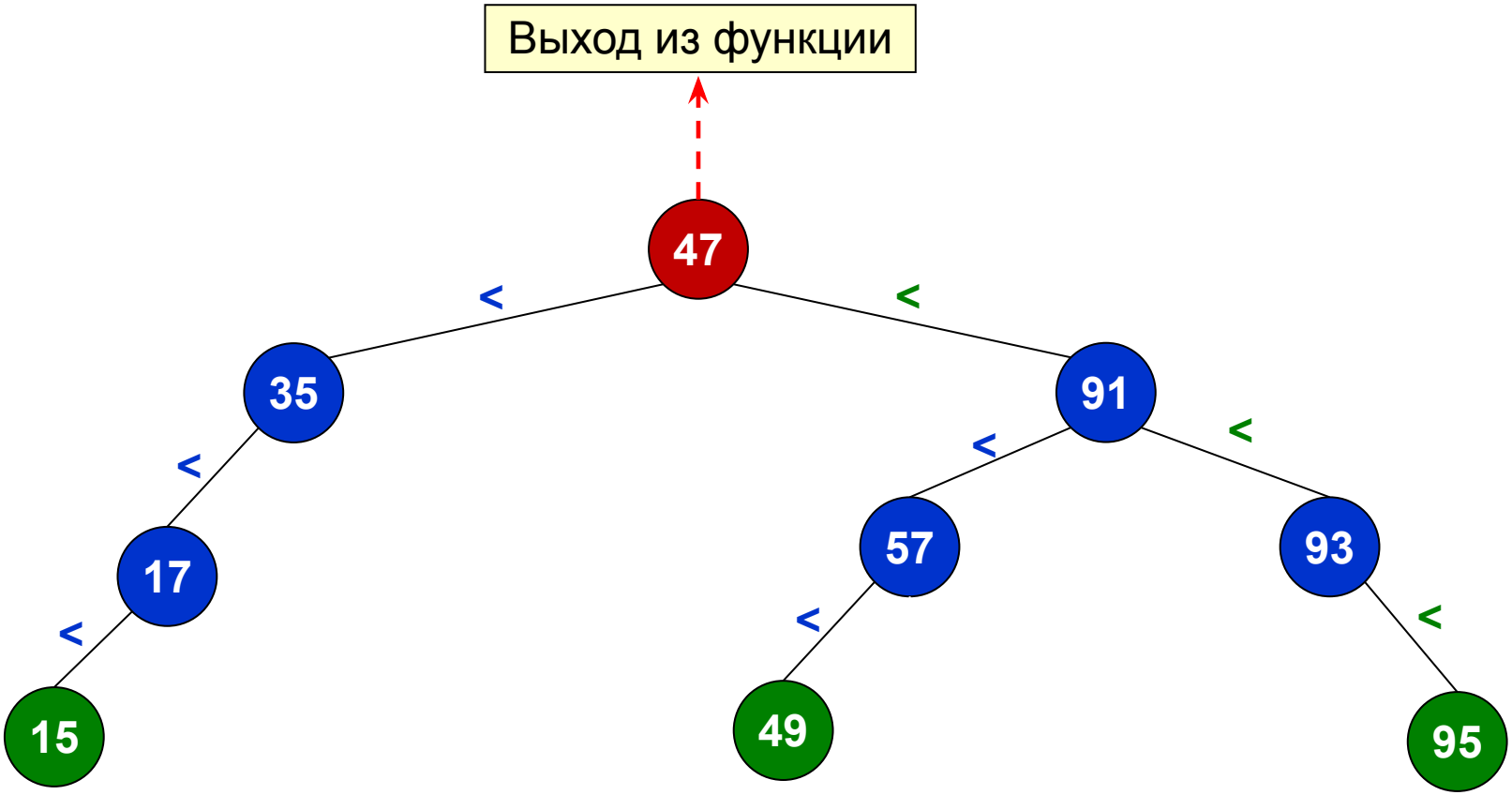
Использование функции удаления



Использование функции удаления



Использование функции удаления



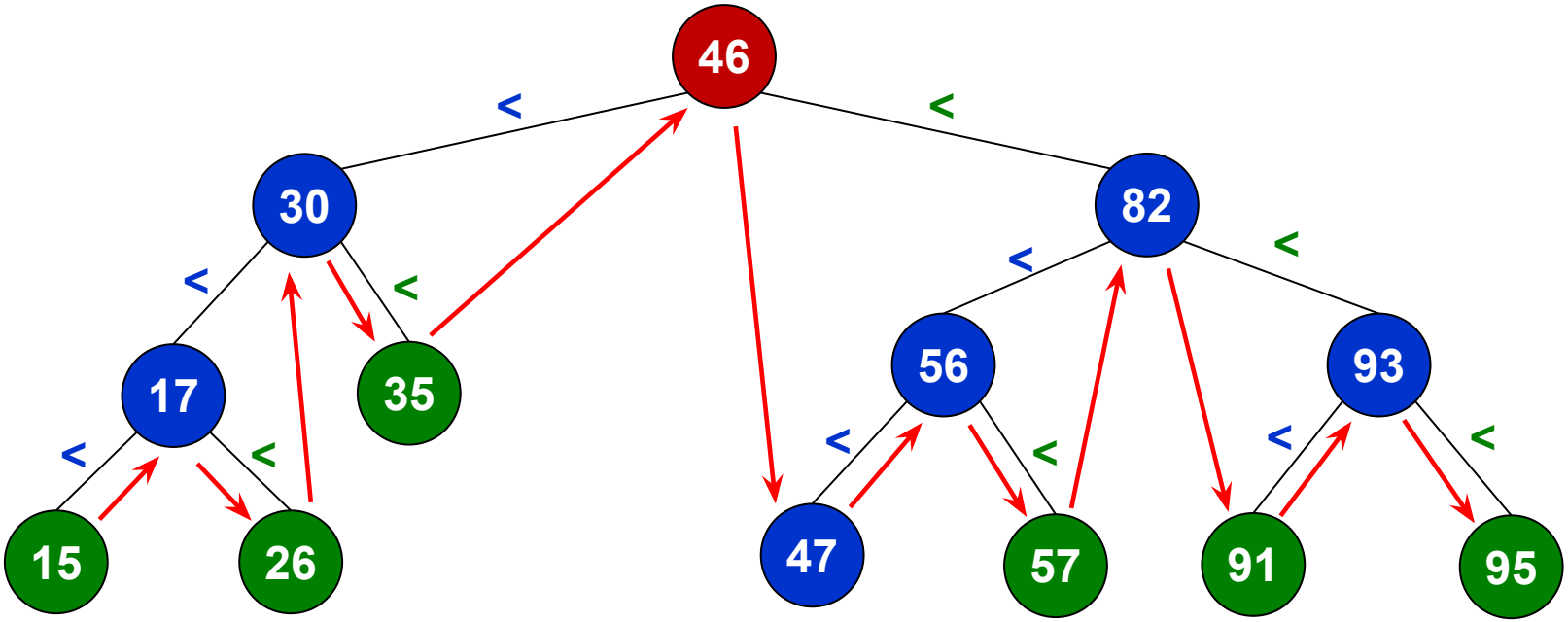
Обход дерева поиска

Обход дерева поиска может выполняться по тем же правилам, что и обход любого двоичного дерева:

К-Л-П, К-П-Л, Л-К-П, П-К-Л, Л-П-К, по уровням.

Рассмотрим обход Л-К-П для дерева поиска.

Л-К-П обход дерева поиска



При Л-К-П обходе дерева поиска узлы будут перебираться по возрастанию ключей:

15, 17, 26, 30, 35, 46, 47, 56, 57, 82, 91, 93, 95

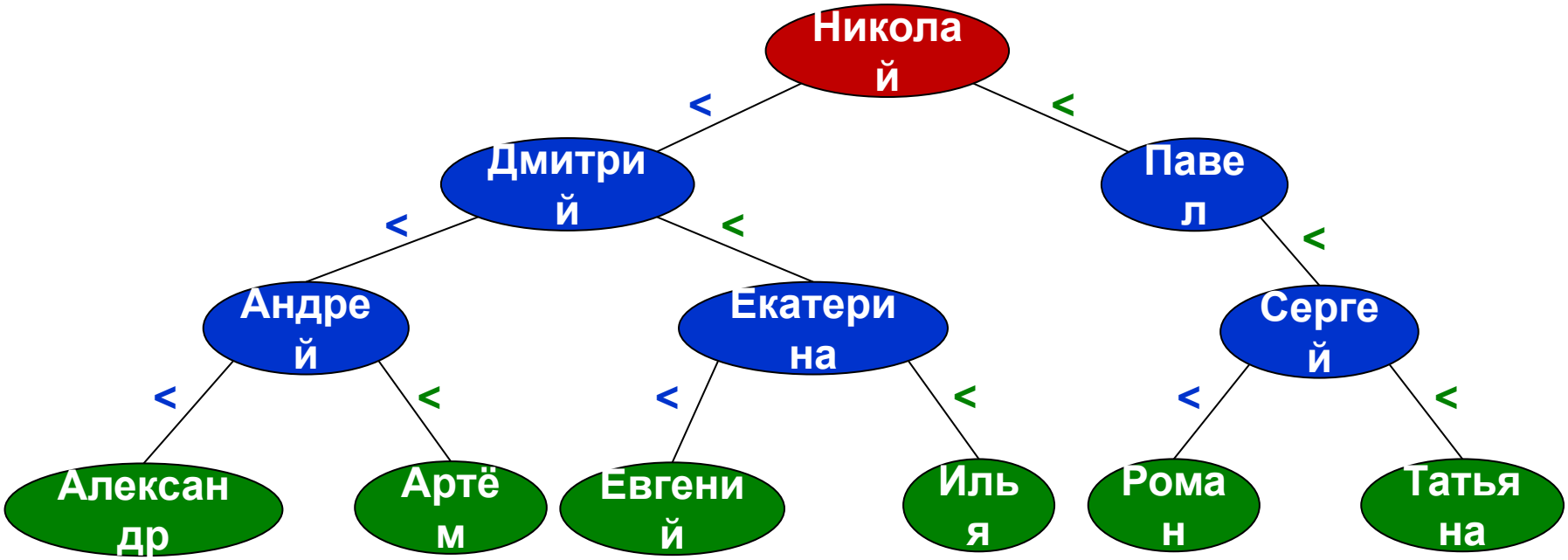
Такой обход называется **симметричным**.

Л-К-П обход дерева поиска

```
//Пример реализации обхода
//Вместо вывода в этот раз складываем ключи узлов в вектор
void SearchTree::lnrKeys(Node *root, std::vector<int> &keys) const
{
    if (!root) {
        return;
    }

    lnrPrint(root->leftChild());
    keys->push_back(root->key());
    lnrPrint(root->rightChild());
}
```

Дерево поиска для символьных данных

Дерево поиска может быть построено и для символьных / текстовых данных:



Заключение

Двоичное дерево поиска может быть использовано как способ такой организации данных, при котором поиск выполняется по достаточно простому алгоритму.

Причём число сравнений при поиске равно номеру уровня, на котором находится искомый элемент. Чем меньше высота дерева, тем меньше уровней. В идеальном дереве уровней **$O(\log_2 n)$** и, следовательно, поиск в нём сравним с бинарным поиском. В следующих темах будет рассмотрен метод построения сбалансированного дерева, которое близко к идеальному.

При необходимости обход дерева может быть использован для получения упорядоченного массива данных, как было показано выше.