Алгоритмы и Алгоритмические Языки

Семинар #15:

1. Структура данных “Бинарное дерево поиска”.
2. Поиск в бинарном дереве, обход дерева.
3. Добавление и удаление элементов.
4. Стратегия аллокации узлов в структуре данных.

03.12.2025

Структура данных «Бинарное дерево поиска»

Структура данных «Бинарное дерево поиска»

```
❯ python3
>>> d = dict()
>>> d[12] = "aaa"
>>> d[155] = "bbb"
>>> d
{12: 'aaa', 155: 'bbb'}
```

Бинарное дерево поиска – одна из реализаций API ассоциативного хранилища данных (key-value storage, см. семинар №13).

```
RetCode tree_alloc (Tree* tree);
RetCode tree_free  (Tree* tree);
RetCode tree_search(Tree* tree, Key_t key, Value_t* res, bool* found);
RetCode tree_set   (Tree* tree, Key_t key, Value_t value);
RetCode tree_remove(Tree* tree, Key_t key, Value_t* ret, bool* found);
void    tree_print (Tree* tree);
```

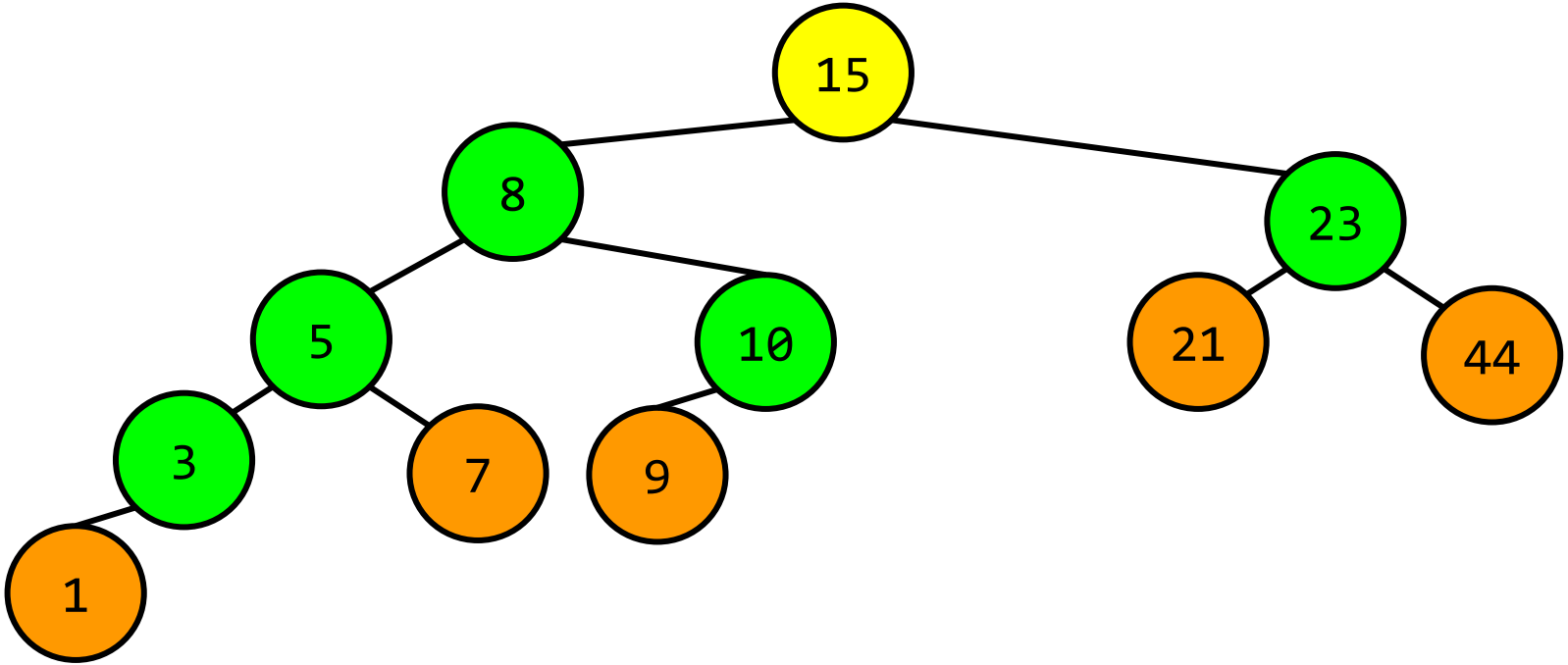
Структура данных «Бинарное дерево поиска»

```
struct TreeNode
{
    // Указатель на родительский узел.
    struct TreeNode* parent;
    // Указатель на левый дочерний узел.
    struct TreeNode* left;
    // Указатель на правый дочерний узел.
    struct TreeNode* right;

    // Ключ узла дерева.
    Key_t key;
    // Значение узла дерева.
    Value_t value;
};
```

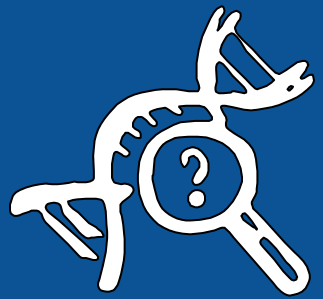
Структура данных «Бинарное дерево поиска»

Типы узлов: **корневой узел**, **внутренние узлы**, **листовые узлы**.

Инвариант: `MAXKEY(node.left) < node.key < MINKEY(node.right)`



Поиск в бинарном дереве, обходы дерева



Поиск в бинарном дереве

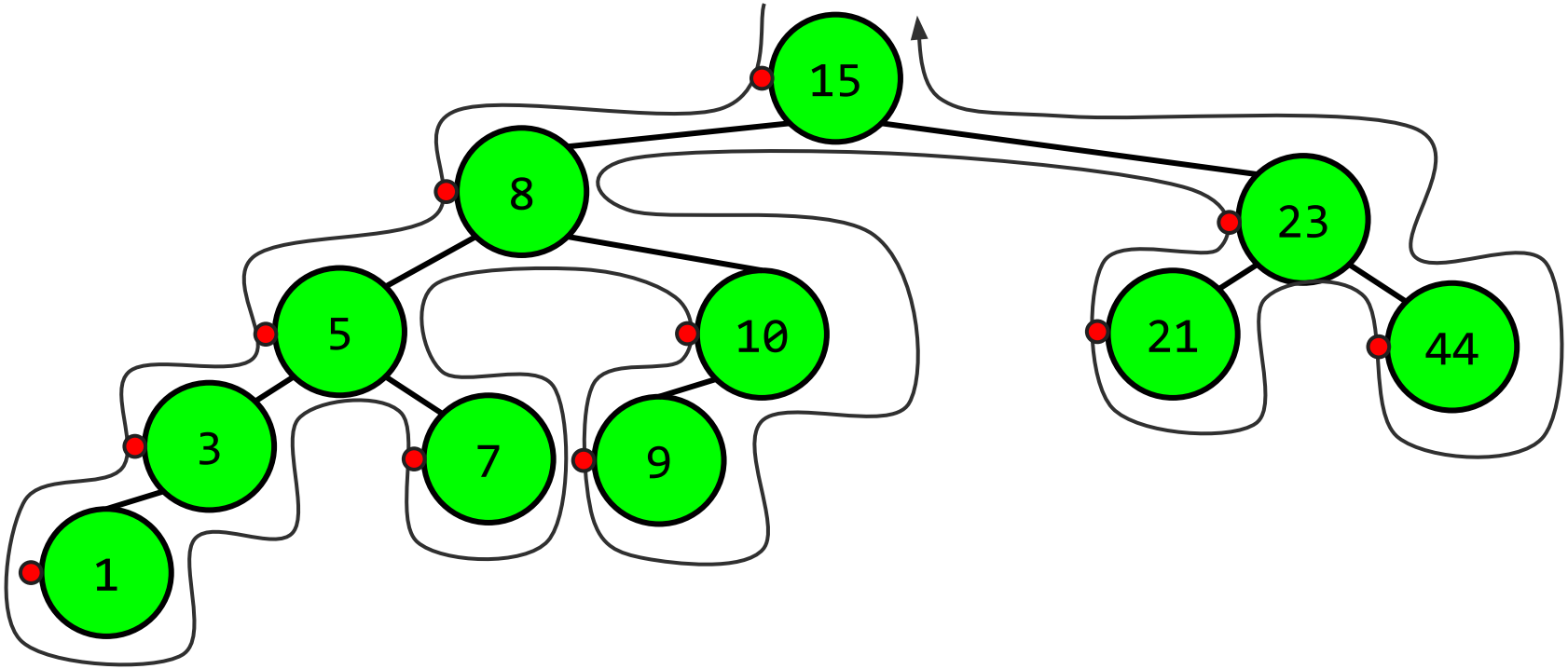
```
TreeNode* tree_search(TreeNode* node, Key_t key) {
    if (node == NULL || key == node->key) {
        // Узел с заданным ключом найден или отсутствует в дереве.
        return node;
    }

    if (key < node->key) {
        // Ищем ключ в левом поддереве.
        return tree_search(node->left, key);
    }

    // Ищем ключ в правом поддереве.
    return tree_search(node->right, key);
}
```

Нюанс: цикл быстрее и надёжнее рекурсии (см. пример №15).

Прямой обход бинарного дерева

Вершины в порядке прямого обхода (**pre-order**, **NLR** (node-left-right)):

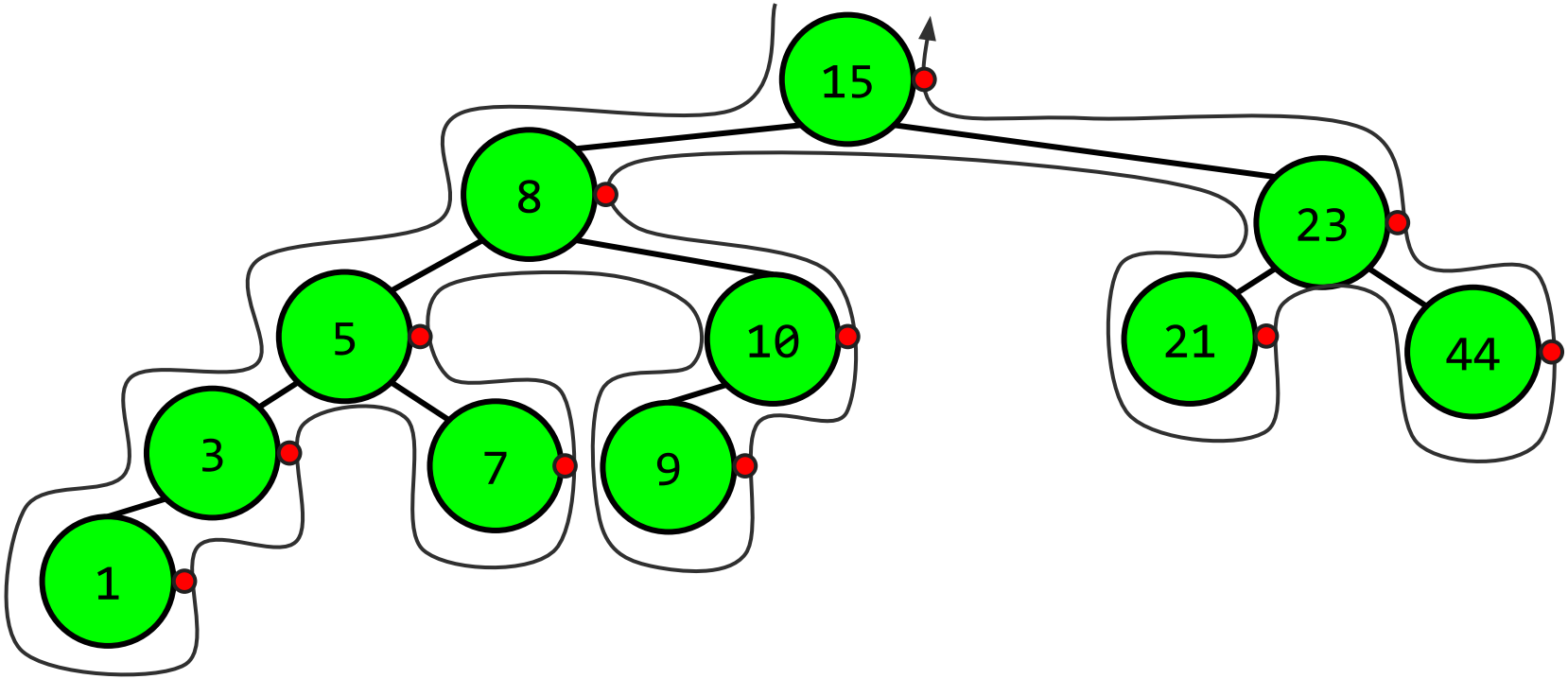
`15 8 5 3 1 7 10 9 23 21 44`



Обратный обход бинарного дерева

Вершины при обратном обходе (**post-order**, **LRN** (left-right-node)):

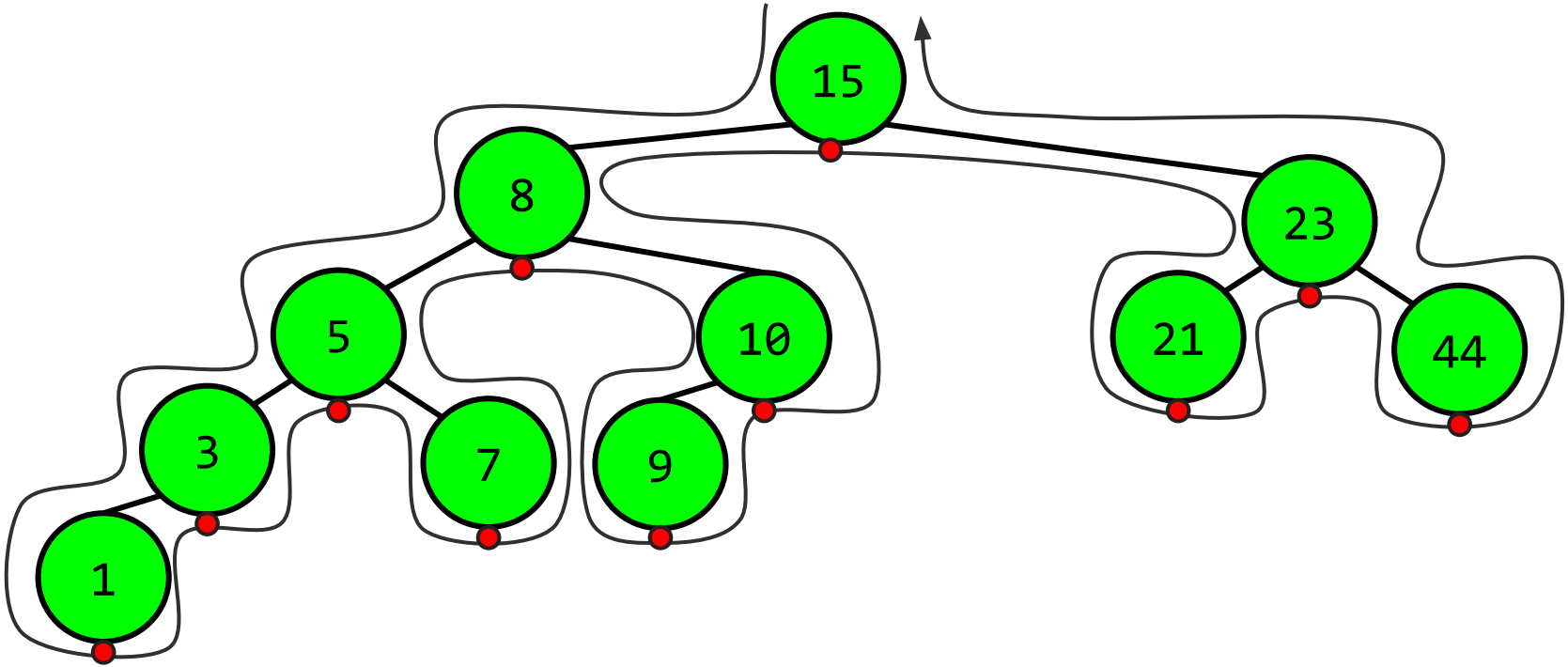
`1 3 7 5 9 10 8 21 44 23 15`



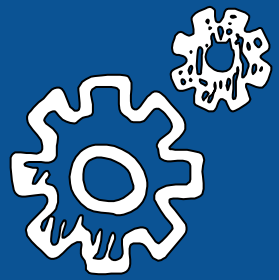
Центрированный обход бинарного дерева

Вершины при центрированном обходе (**in-order**, **LNR**):

`1 3 5 7 8 9 10 15 21 23 44`

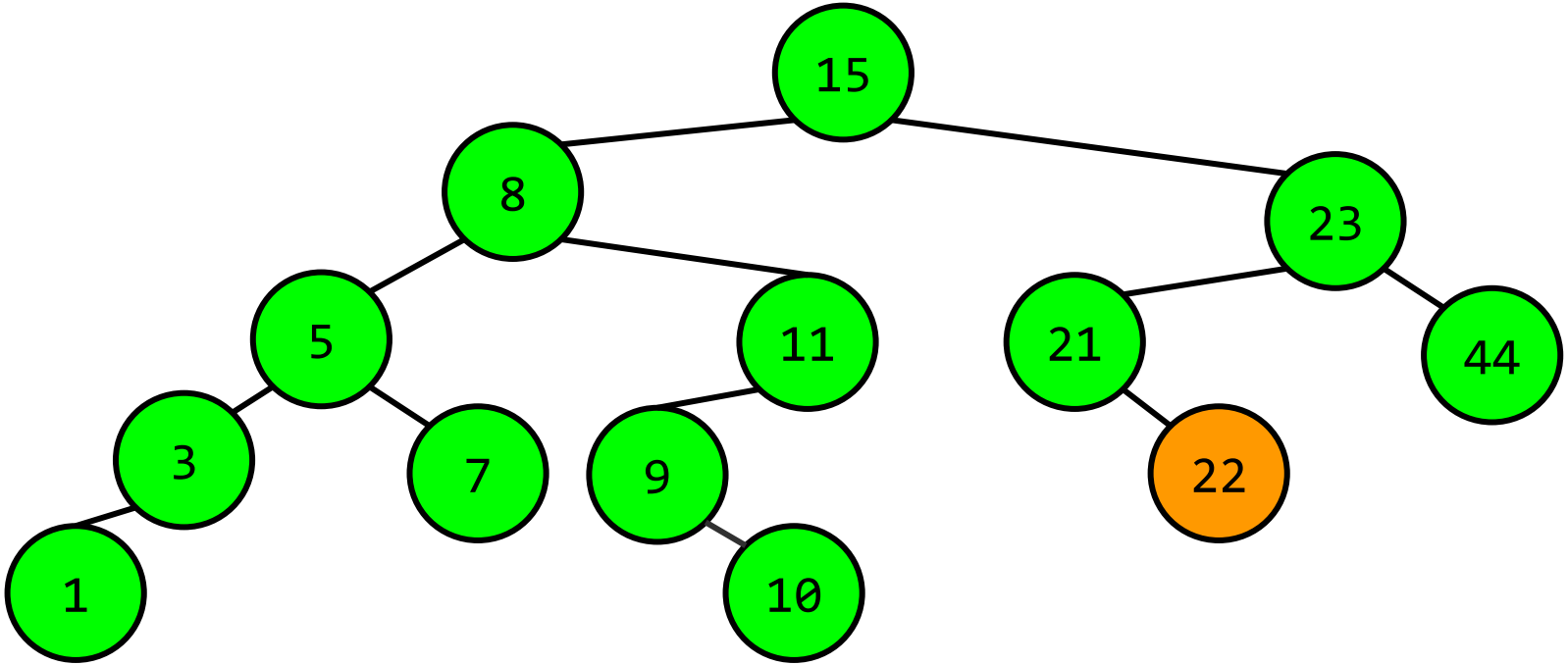


Добавление и удаление элементов



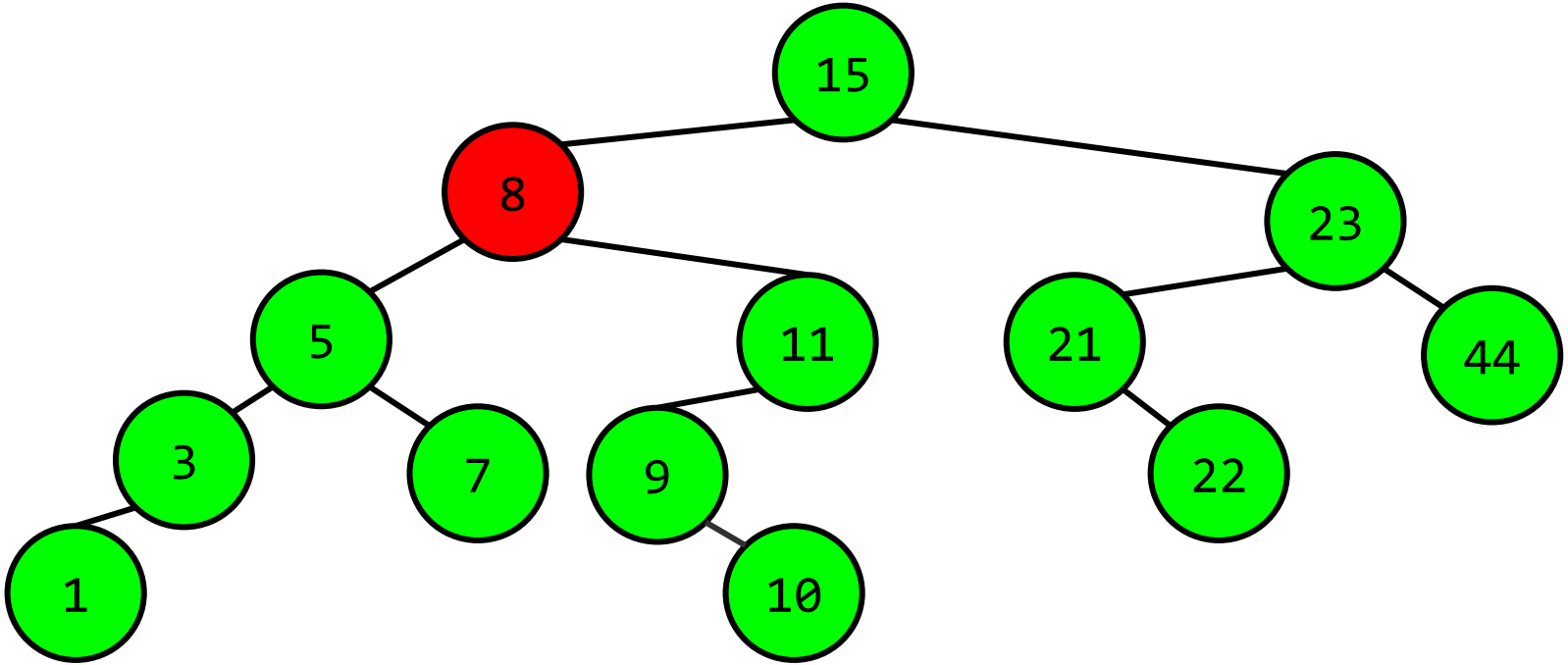
Добавление элемента

```
tree_insert(tree, 22)
```



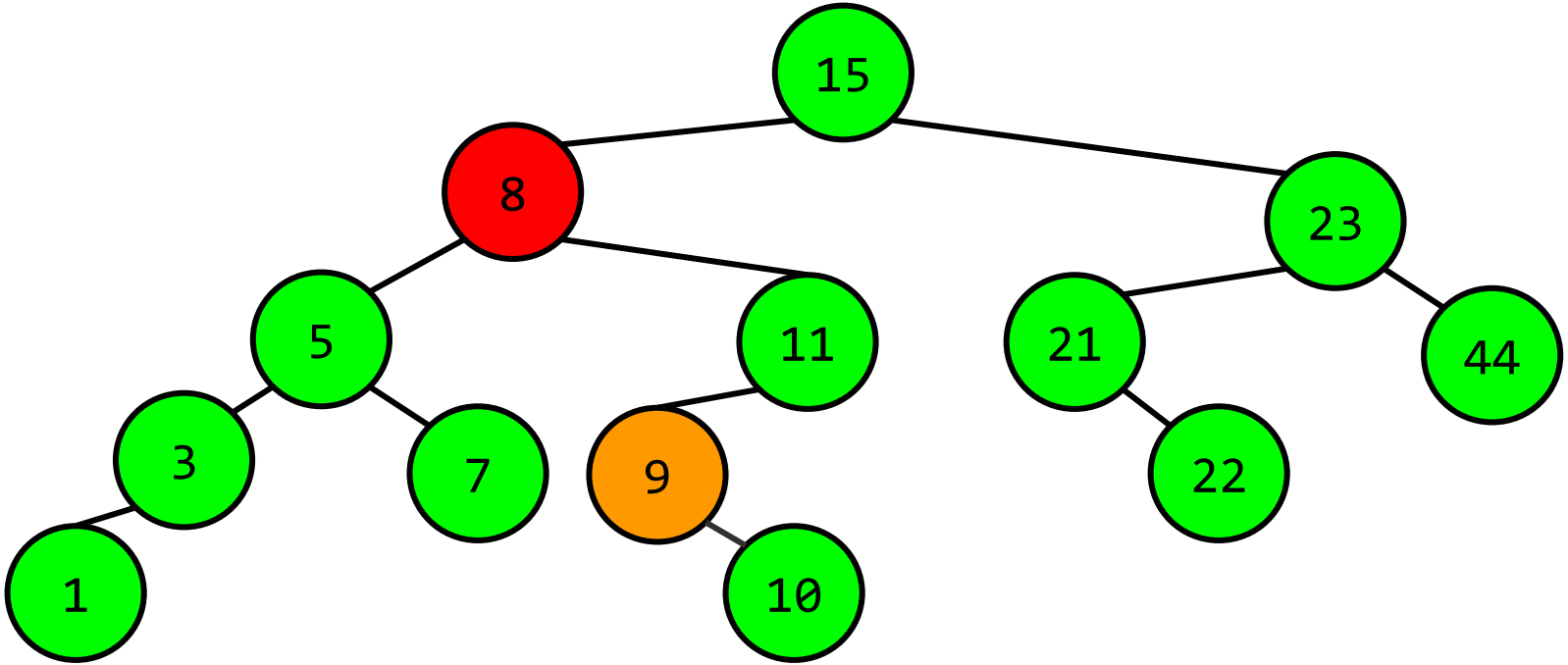
Удаление элемента: поиск места удаления

```
tree_remove(tree, 8)
```



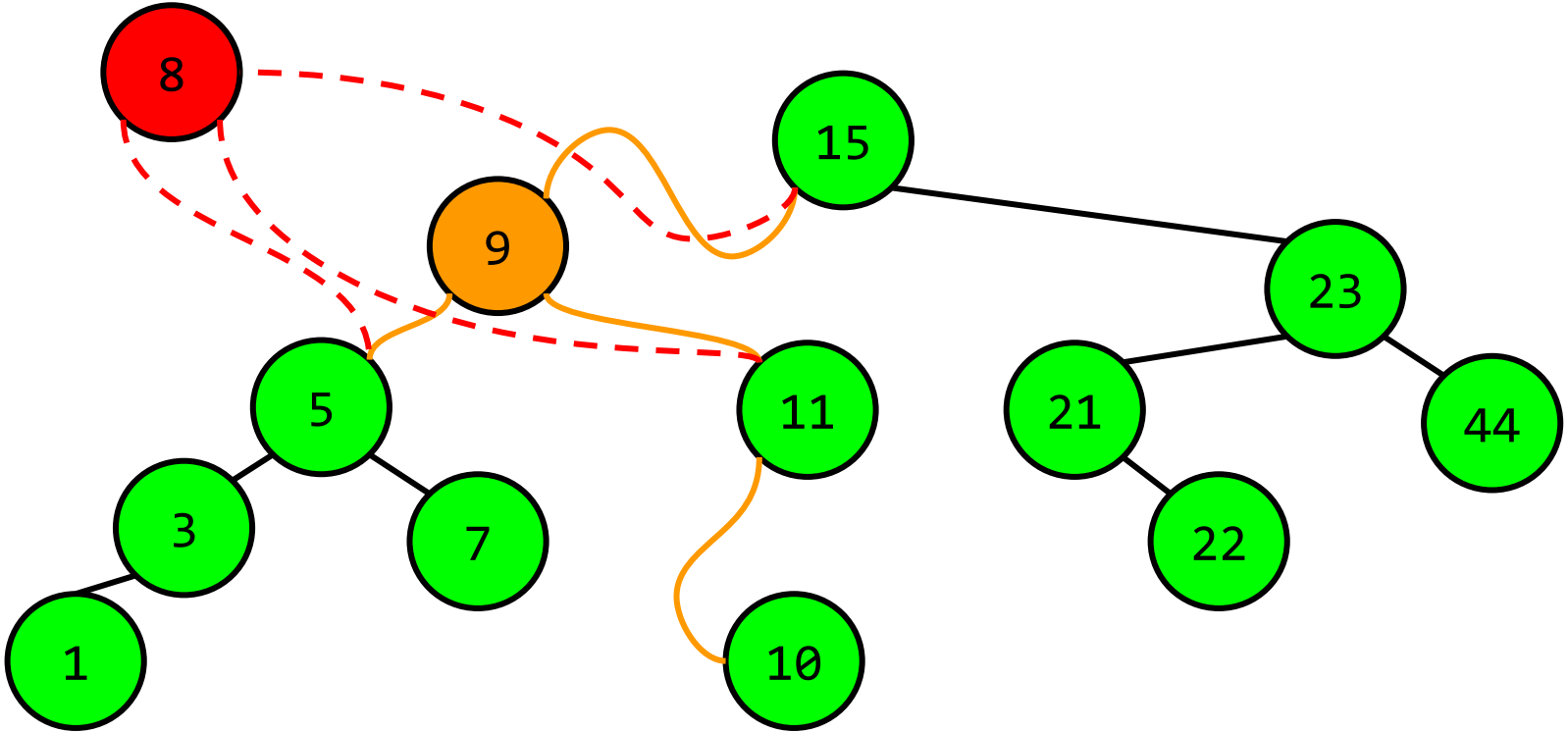
Удаление элемента: поиск следующего элемента

```
tree_remove(tree, 8)
```



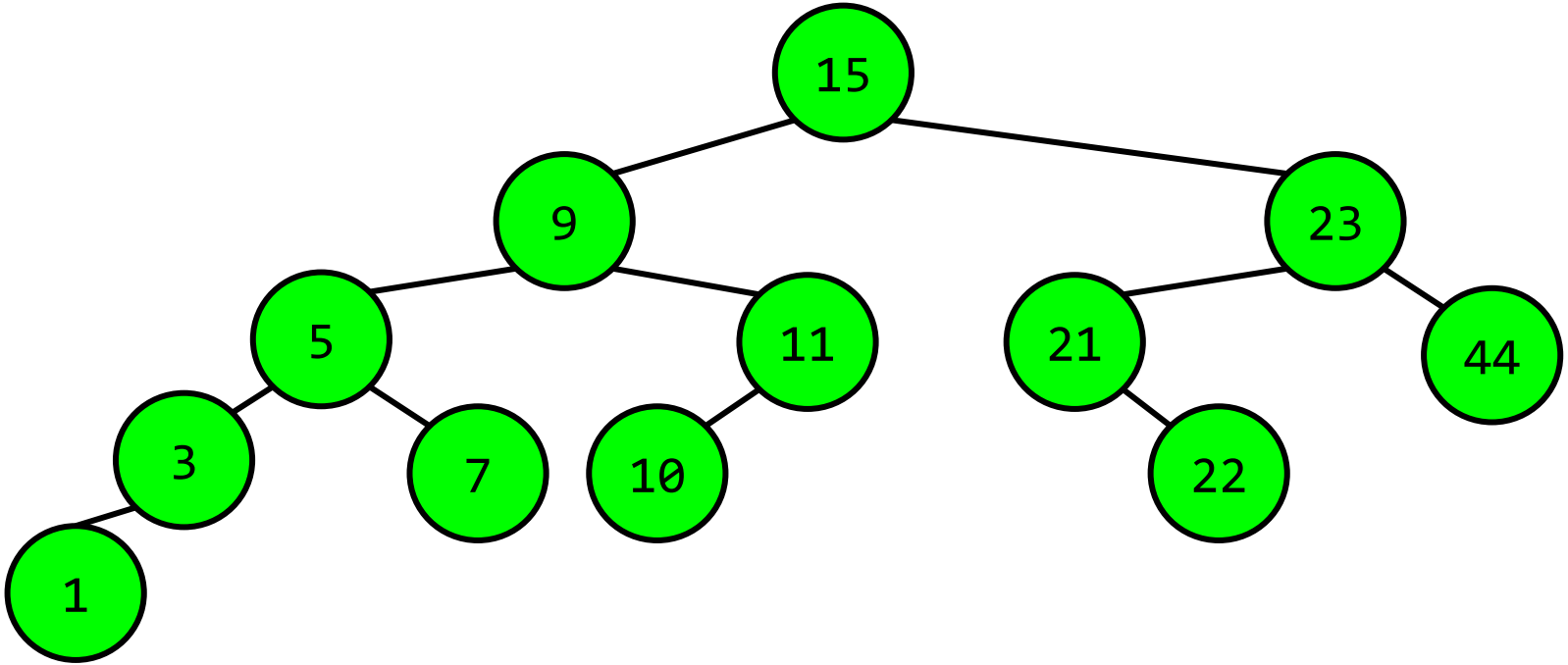
Удаление элемента: пересадка деревьев

```
tree_remove(tree, 8)
```

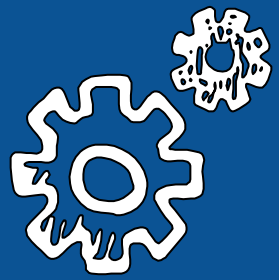


Удаление элемента: результат

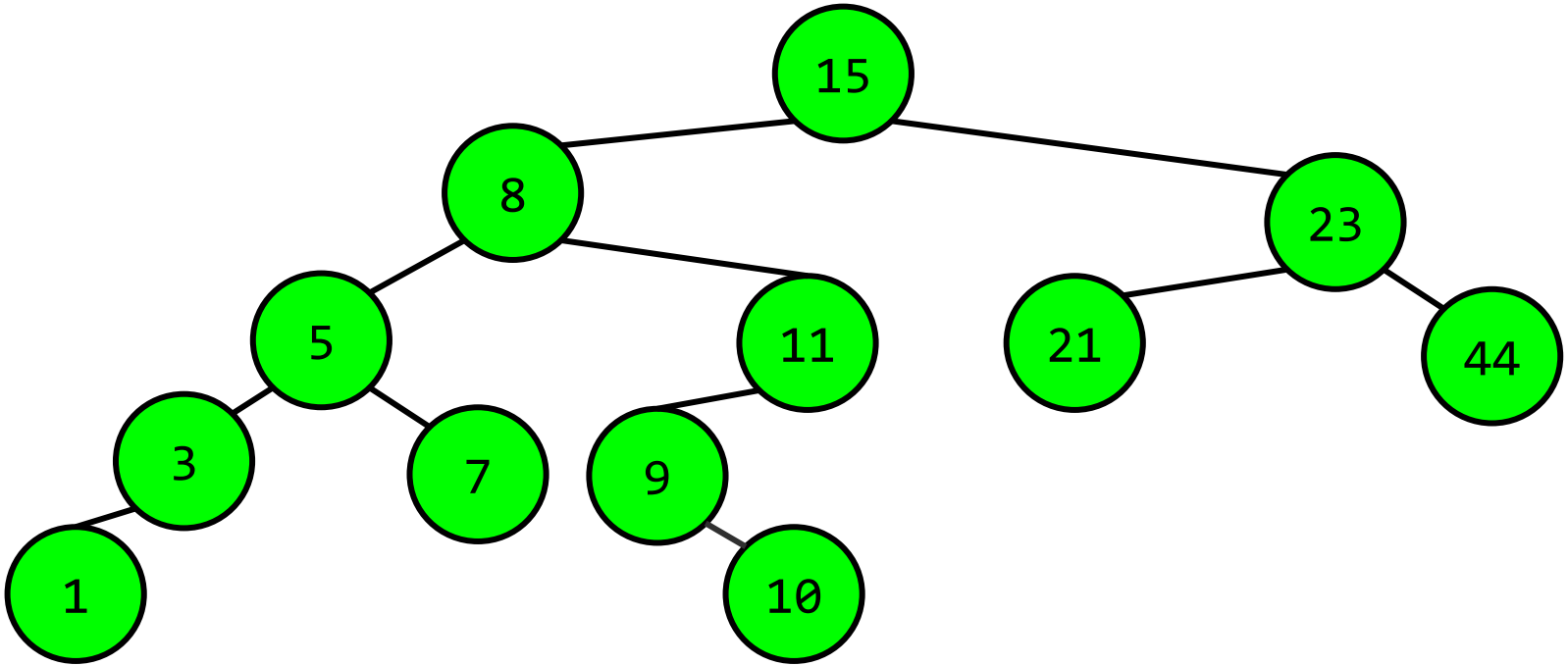
```
tree_remove(tree, 8)
```



Стратегия аллокации узлов в структуре данных

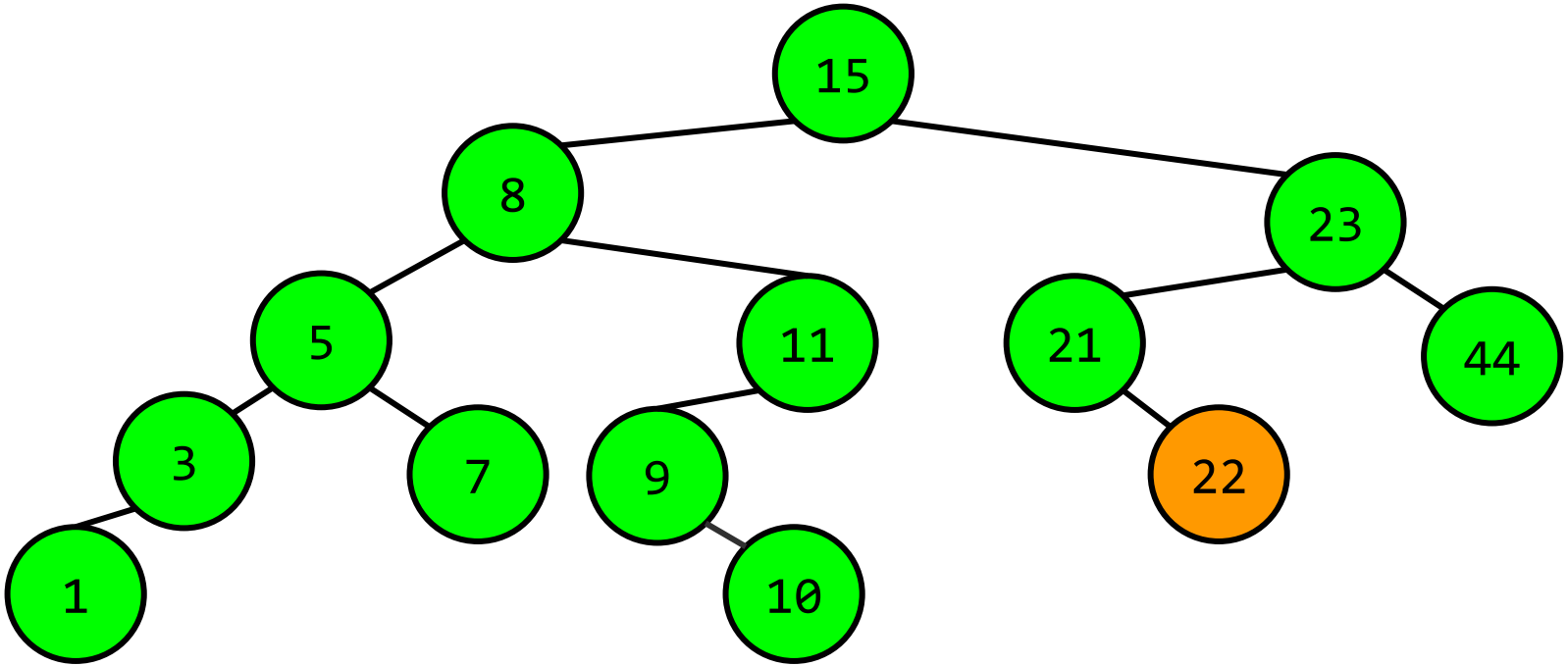


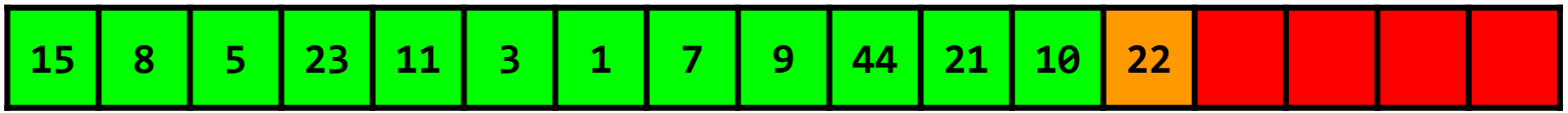
Размещение узлов дерева в динамическом массиве



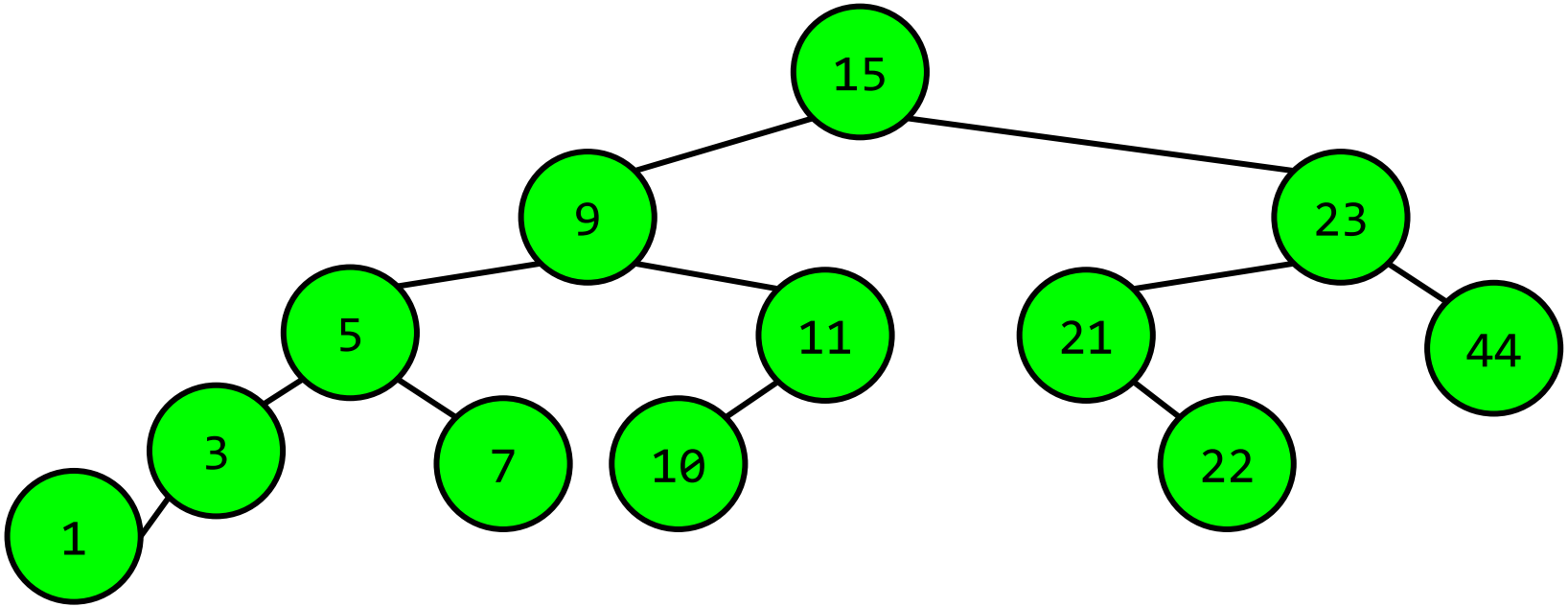
15	8	5	23	11	3	1	7	9	44	21	10					

Добавление элемента





Удаление элемента



15	8	5	23	11	3	1	7	9	44	21	10	22				
15	22	5	23	11	3	1	7	9	44	21	10					

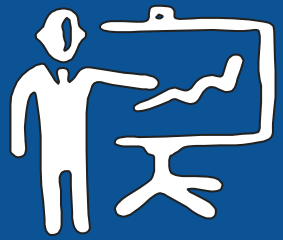
Собственная стратегия аллокации: зачем?

По сравнению с менеджментом памяти через `malloc+free.`

Плюсы:
1. Быстрее за счёт более редких вызовов `malloc+free` (я надеюсь).
2. Не способствует фрагментация памяти.
3. Меньше метаданных (указатели -> индексы, метаданные malloc).

Минусы:
1. Код сложнее.
2. Есть зависимость от фрагментации памяти.
3. Есть копирование `Value_t` (потенциально большой объект).
4. Не поможет на экзамене `:(`

Вопросы?



Красивые иконки взяты с сайта handdrawngoods.com