

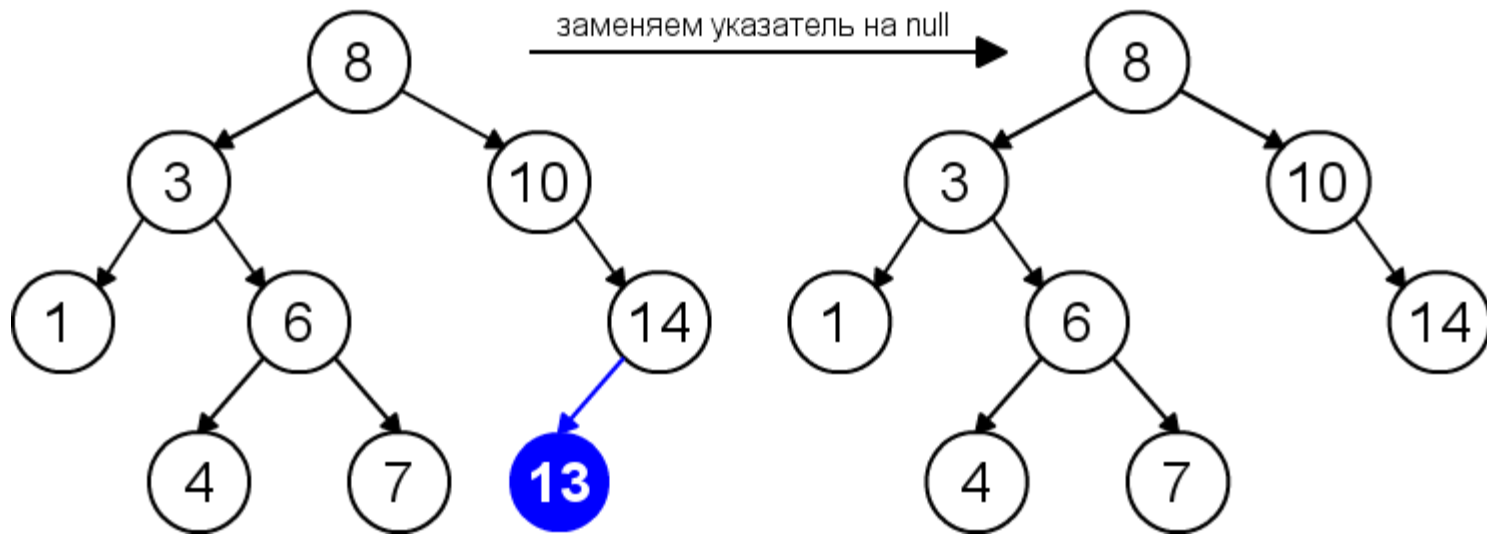
# Двоичное дерево поиска

2013

# Операция удаления

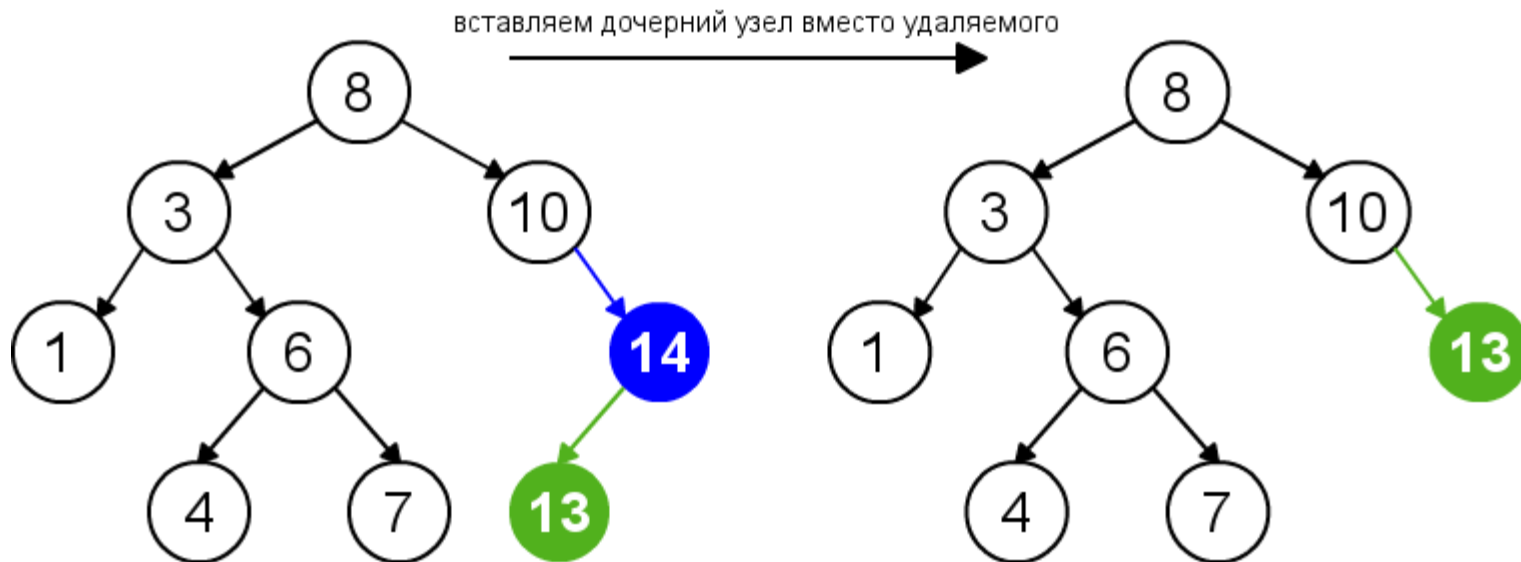
- Рассмотрим три случая
  1. узел является листом
  2. узел имеет только одно поддеревово
  3. у узла два поддерева

# 1 случай



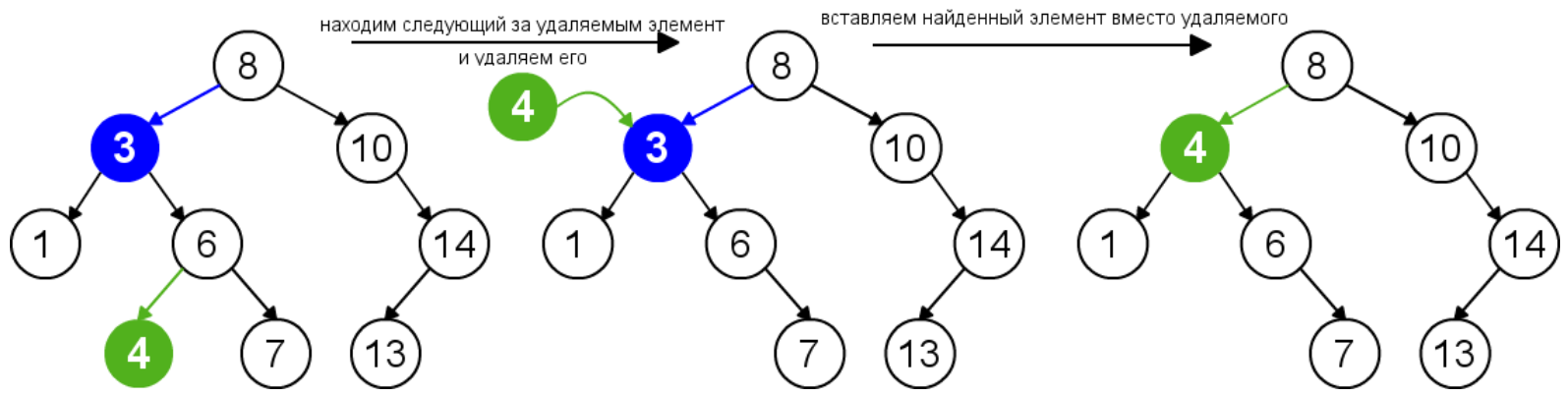
# 2 случай

- одинаково для левого и правого поддеревя



# 3 случай

- Находим самый левый элемент в правом поддереве
- Его значение вставляем вместо удаляемого узла
- А сам этот узел удаляем



# Реализация класса

```
class Node;  
typedef Node* pnode;  
  
class Node {  
    int data;  
    pnode lt, rt;  
    Node () {}  
    Node (int a, pnode t1=0, pnode t2 =0):  
        data(a),lt(t1),rt(t2) {}  
  
friend class BSTree;  
};
```

- Все рекурсивные методы делаем закрытыми
- Для их вызова простые открытые методы



```
class BSTree {
```

```
private:
```

```
    pnode root;
```

```
    void printPref (pnode &t);
```

```
    void copy(pnode t, pnode &newT) const;
```

```
    void destroy (pnode & t);
```

```
    void insertEl (pnode & t, int a);
```

```
    void deleteEl (pnode & t, int a);
```

```
    void deleteNode (pnode &t);
```

```
    void delLeftLeaf(pnode &t,int &repl);
```

public:

```
BSTree() {root=0;}
```

```
BSTree(const BSTree& tree);
```

```
~BSTree();
```

```
void printPref();
```

```
bool isEmpty() const;
```

```
void insertEl(int val);
```

```
void deleteEl (int val);
```

```
};
```

# Открытые методы

```
BSTree :: BSTree(const BSTree& tree)
{
    copy (tree.root, root);
}
BSTree :: ~BSTree()
{
    destroy(root);
}
void BSTree :: printPref()
{
    printPref(root);
}
```

# Открытые методы

```
bool BSTree ::isEmpty() const
{
    return root==0;
}
void BSTree ::insertEl(int val)
{
    insertEl (root,val);
}
void BSTree ::deleteEl(int val)
{
    deleteEl(root,val);
}
```

# Рекурсивные методы

```
void BSTree ::insertEl(pnode &t, int a)
{
    if (t==0)
        t = new Node (a, 0, 0);
    else
        if (a < t->data)
            insertEl(t->lt,a);
        else insertEl(t->rt,a);
}
```

```
void BSTree :: printPref (pnode &t)           //LKR
{
    if (t!=0)
    {
        printPref(t->lt);
        cout<< t->data<<" -";
        printPref(t->rt);
    }
}
```

```
void BSTree :: copy(pnode t, pnode &newT) const //KLR
{
    if ( t!=0)
    {
        newT = new Node(t->data,0,0);
        copy(t->lt,newT->lt);
        copy(t->rt,newT->rt);
    }
    else t=0;
}
```

```
void BSTree :: destroy(pnode &t)           //LRK
{
    if (t!=0)
    {
        destroy(t->lt);
        destroy(t->rt);
        delete t;
        t=0;
    }
}
```



# Три метода, реализующие удаление

- Найти узел по значению, которое нужно удалить и вызвать для этого узла метод исключающий его из дерева

```
void BSTree :: deleteEl(pnode &t, int a)
{
    if (t!=0)
    if (a == t->data)
        deleteNode(t); // узел найден - удаляем
    else
        if (a<t->data)
            deleteEl(t->lt,a); // рекурсия
        else
            deleteEl(t->rt,a); // рекурсия
}
```

- Удаление найденного узла ( работает в корректных условиях, т.к. вызывается, когда узел найден)

```

void BSTree :: deleteNode (pnode &t)
{
    // 1. корень является листом
    // 2. у корня нет левого поддерева
    // 3. у корня нет правого поддерева
    // 4. корень имеет два поддерева

    pnode delNode;
    int repl;
    if ((t->lt ==0) && (t->rt==0)) // 1
    {
        delete t;
        t=0;
    }
}

```

```
else
```

```
    if (t->lt == 0)                // 2
```

```
    {
```

```
        delNode = t;
```

```
        t = t->rt;
```

```
        delNode->rt = 0;
```

```
        delete delNode;
```

```
    }
```

```
else if (t->rt == 0)           // 3
{
    delNode = t;
    t = t->lt;
    delNode->lt = 0;
    delete delNode;
}
```

В 4 случае вызываем вспомогательный метод, который ищет самый левый узел в правом поддереве. запоминаем его значение, а сам узел - удаляем

```
else // 4
{
    delLeftLeaf (t->rt, repl);
    t->data = repl;
}
}
```



```

void BSTree::delLeftLeaf(pnode &t,int &repl)
{
    if (t->lt == 0) // у самого левого нет левых
                    //потомков
    {
        repl = t->data;
        pnode delNode = t;
        t = t->rt;
        delNode->rt=0;
        delete delNode;
    }
    else delLeftLeaf (t->lt,repl);
}

```

# Эффективность операций

- Максимальное количество сравнений, необходимых для выполнения операций поиска, вставки и удаления, равно высоте бинарного дерева поиска
- Для дерева из  $N$  узлов возможные значения высоты  $h$

$$[ \log_2(N+1) ] \leq h \leq N$$

- Бинарное дерево наименьшей высоты называется совершенным или сбалансированным
- Чтобы обеспечить эффективность операций нужно стремиться поддерживать структуру близкую к сбалансированности
- Существуют специальные алгоритмы балансировки