ТЕМА 8.
ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ.
Нелинейные списки

8.1. Древовидные структуры данных

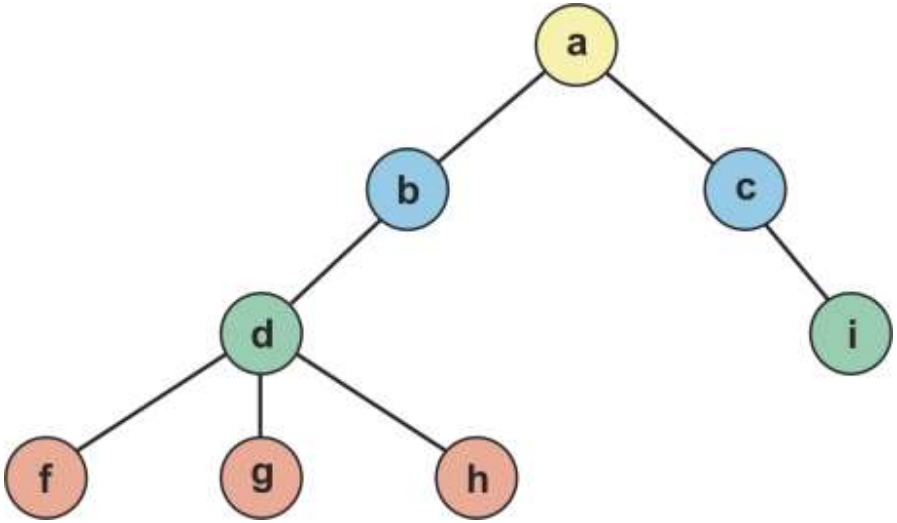
Все данные называются **узлами**.

Связи между узлами называется **ветвями**.

Самый верхний узел – **корень дерева** (*a*).

Узел, не имеющие связей – **лист** дерева (*f, g, h, i*).

Узел, не являющийся листом – **внутренний узел** *(b, d, c или a).*



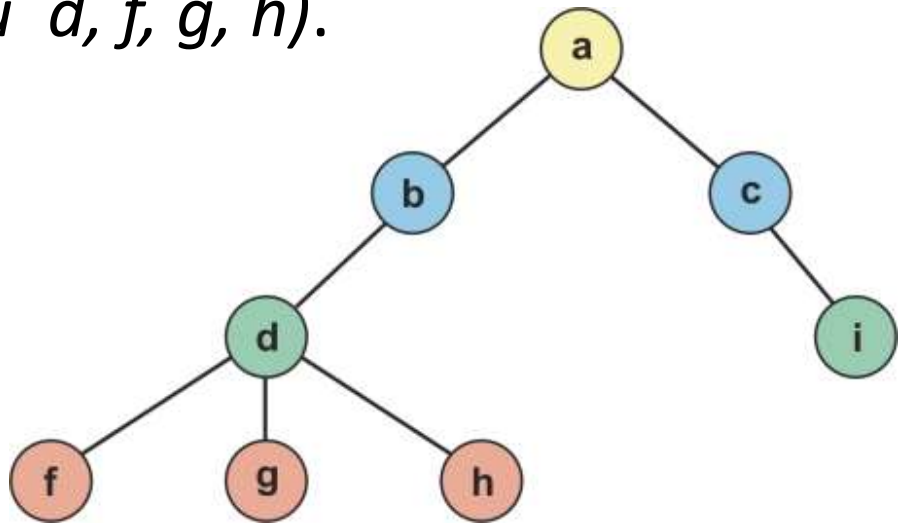
Узел, находящийся непосредственно над другим – **родительский узел** *(например, c для i)*

Узел, находящийся непосредственно под другим – **дочерний узел** *(например, I для c)*.

Все узлы, находящиеся непосредственно выше узла – **предки** (*для узла d предки b и a*)

Все узлы, находящиеся непосредственно ниже узла – **потомки** *(для узла b потомки d, f, g, h)*.

Узлы, имеющие одного и того-же родителя – **сестринские узлы** *(f, g, h)*.



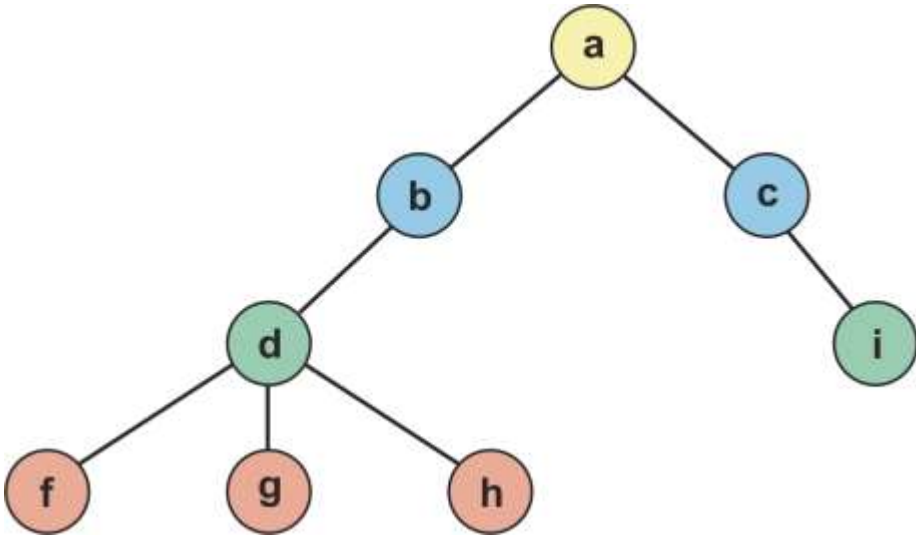
Степень (порядок) узла – количество дочерних узлов (*для узла b порядок 1, для узла d порядок 3*).

Степеть дерева – максимальный порядок его узлов (*рассматриваемое дерево имеет третий порядок*)

Дерево второй степени называется **бинарным (двоичным)**.

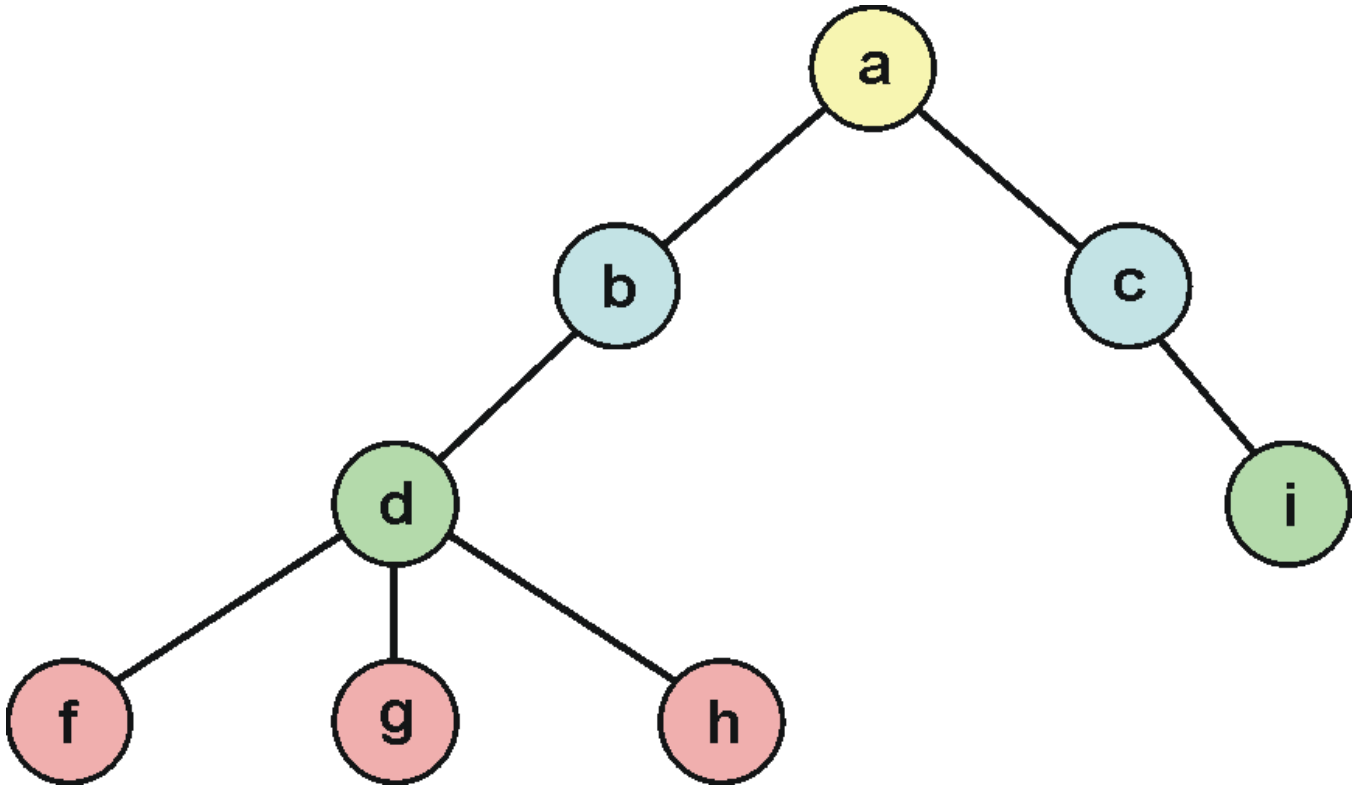
Глубина узла – число предков плюс один (*например, для узла d глубина равна 3*).

Глубина дерева – наибольшая глубина всех узлов (*для данного дерева – 4*).

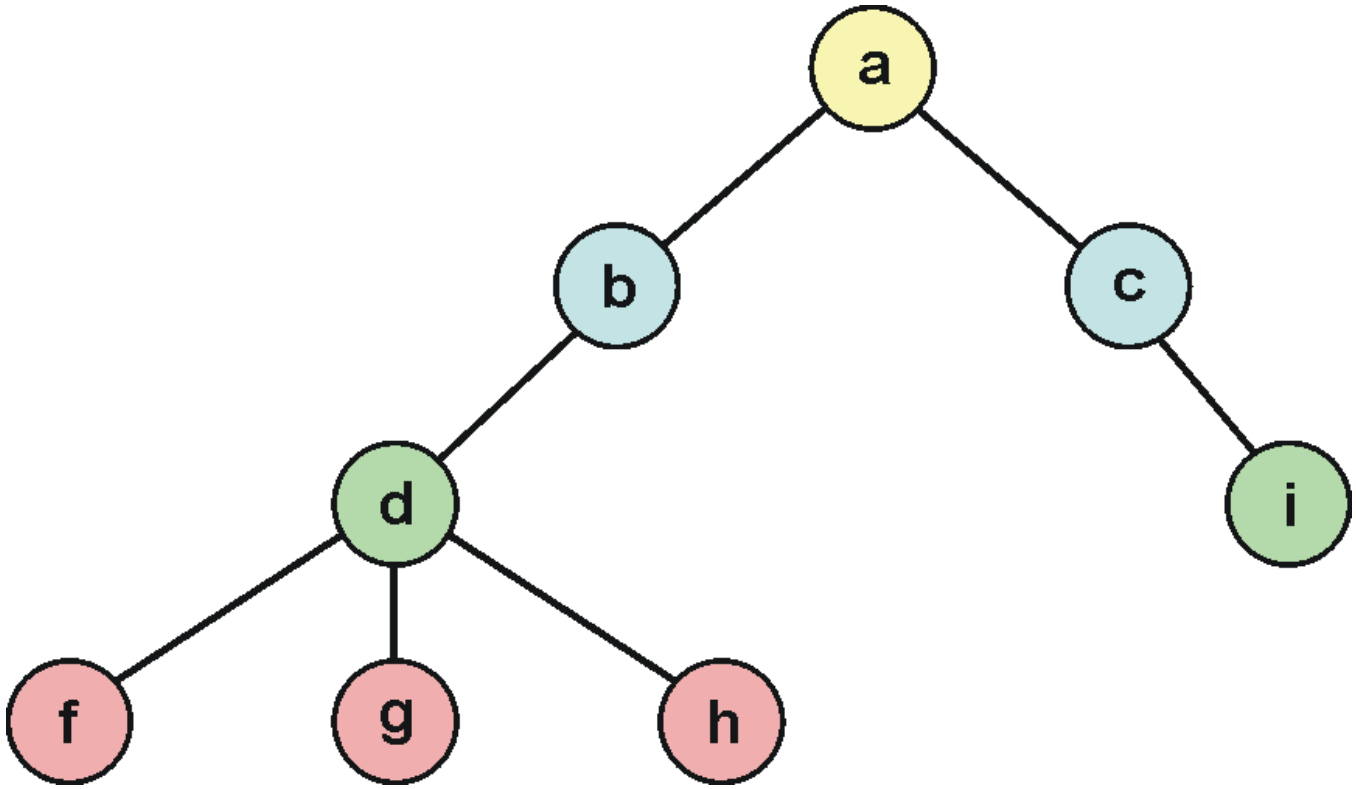


Обход дерева – последовательное обращение ко всем его узлам.

```
void obh(ttree* p)
{
    if (p == NULL) return;
                    // вывод при прямом обходе
    obh(p->a1);
    obh(p->a2);
    …
        obh tree(p->an);
                    // вывод при обратном обходе
}
```



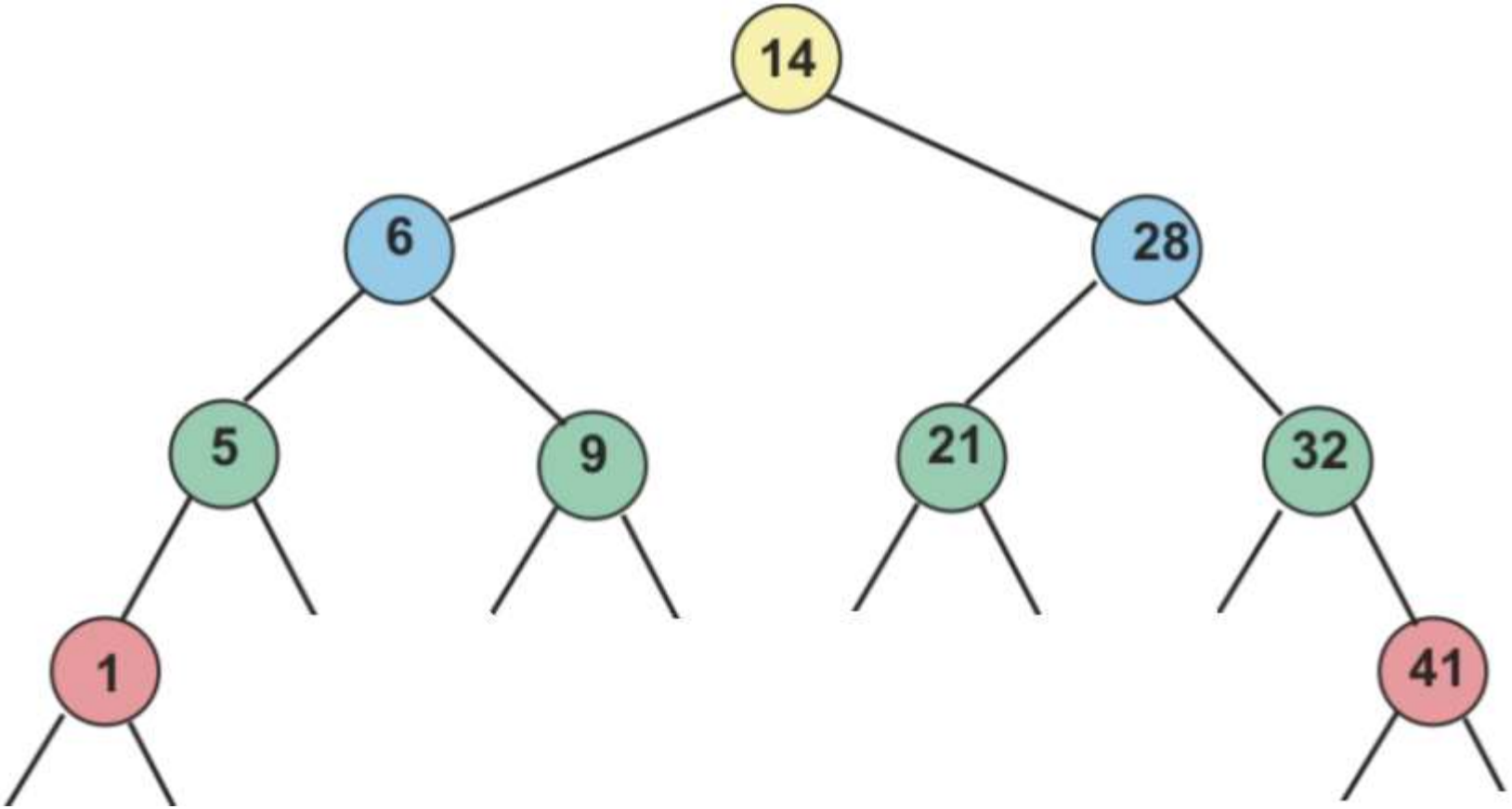
Прямой обход: a b d f g h c i



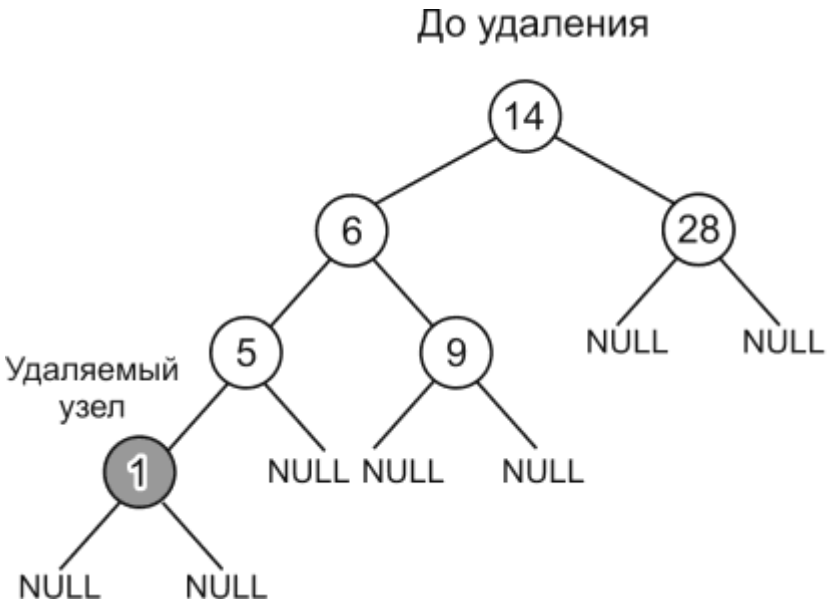
Обратный обход: f g h d b i c a

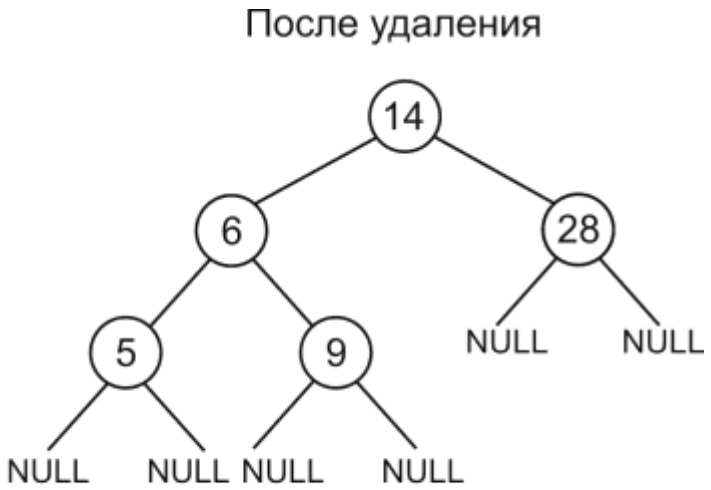
8.2. Двоичное дерево поиска

key: **1, 5, 6, 9, 14, 21, 28, 32, 41.**

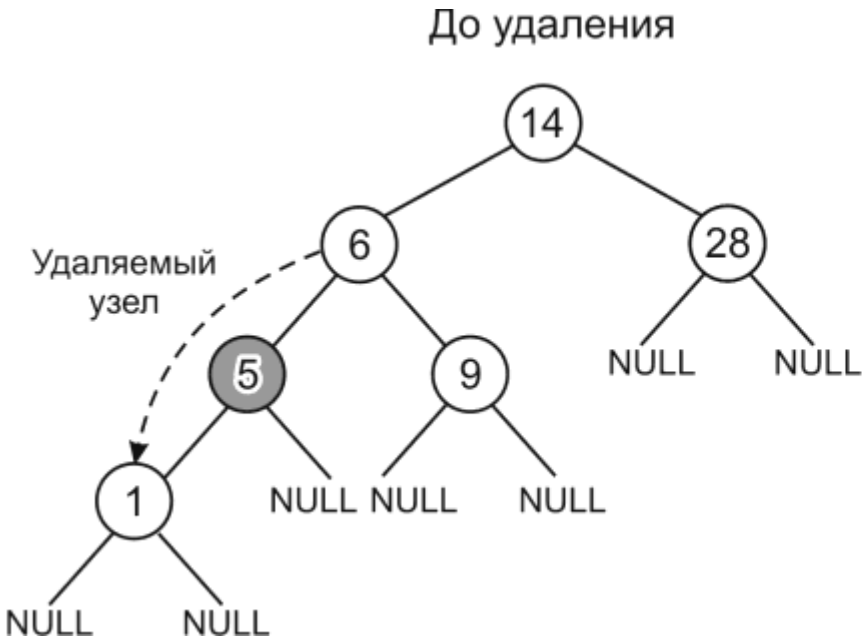


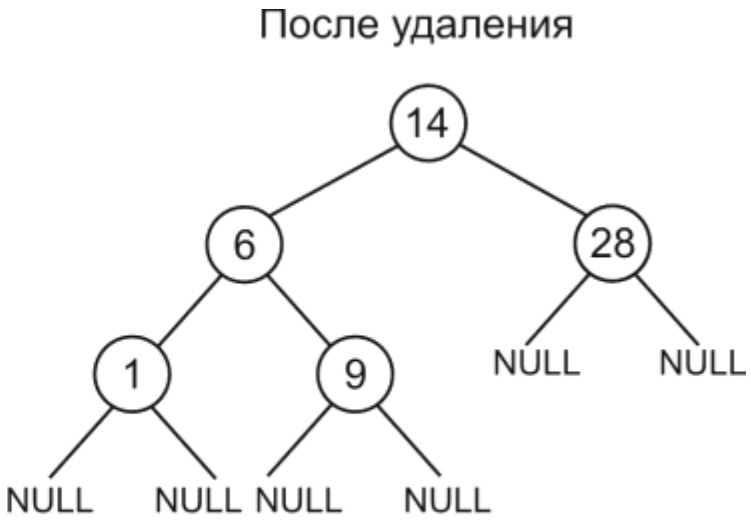
1. Если удаляется узел, не имеющий потомков



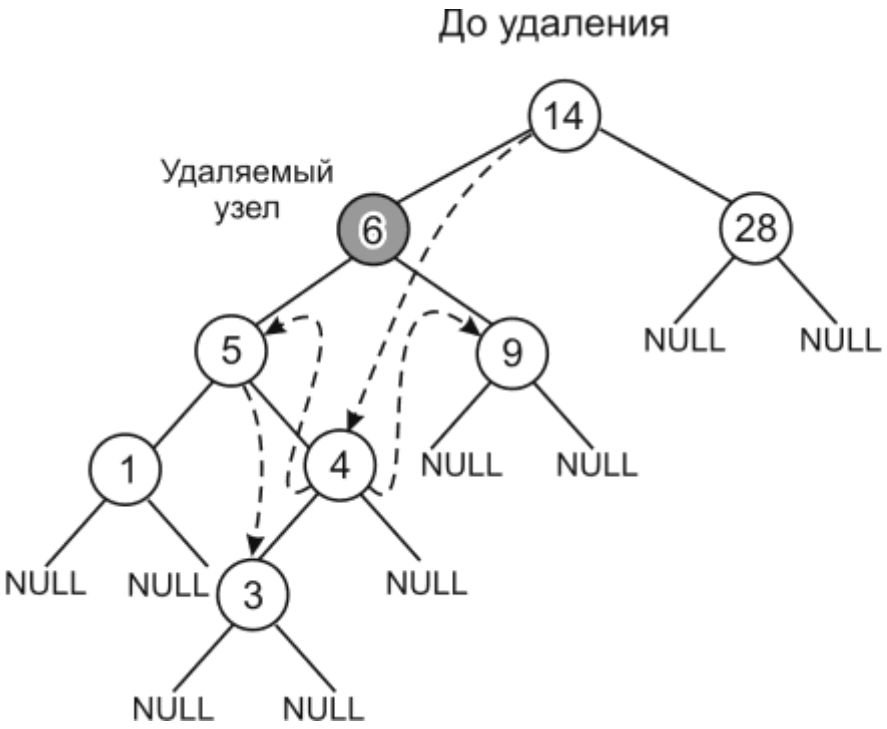


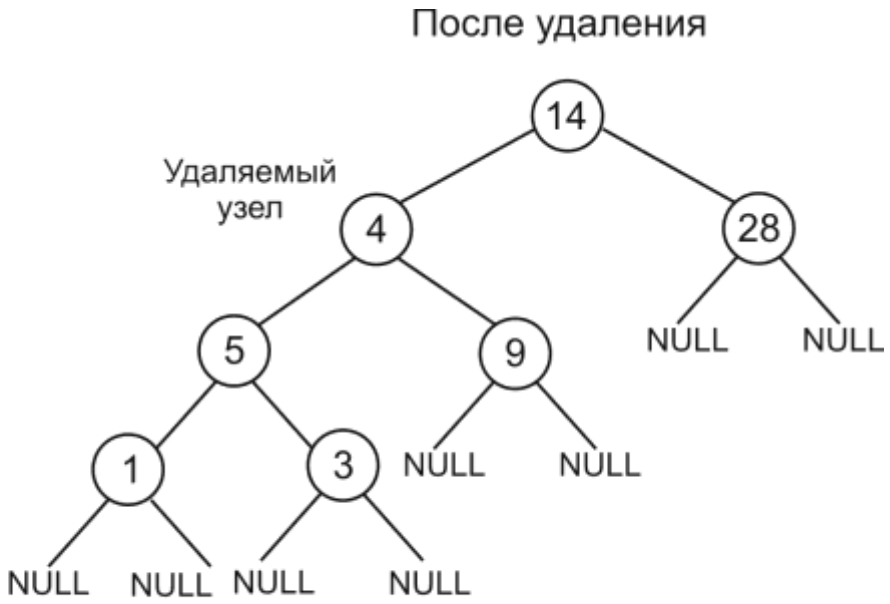
2. Если удаляется узел, имеющий одну дочь.





3. Если удаляется узел, имеющий двух дочерей, то удаляемый узел заменяется узлом, имеющим наибольший ключ в левом поддереве либо наименьший ключ в правом поддереве.





Пример 8.1. Работа с двоичным деревом поиска.

```
#include <iostream>
using namespace std;

struct TNode {
  int inf;    // Информационная часть узла
  TNode* left;   // Адресная часть
  TNode* right;  // Адресная часть
};
```

```
struct TTree {

TNode* root = nullptr;  // Указатель на корень дерева

// Проверка наличия элементов в дереве
    bool empty() {
if (root) return false;
        else return true;
    }
```

```
void push(int inf) {  // Добавление элемента в дерево
    TNode* nu = new TNode;
    nu->inf = inf;
    nu->left = nullptr;
    nu->right = nullptr;
```

```
if (!root) { root = nu; return; }
        bool b;
        TNode* pdel = root;  TNode* del = root;
        while (del) {
            pdel = del;
            b = inf < del->inf;
            if (b) del = del->left;
            else del = del->right;
        }
        if (b) pdel->left = nu;
        else pdel->right = nu;
    }
```

```
void print(TNode* p) // Симметричный обход дерева
    {
      if (!p) return;
      print(p->left);
        cout << p->inf << " ";
      print(p->right);
    }
```

```
//  Удаление элемента из дерева
    void pop(int x) {
TNode* del = root, * pdel = root, * rep, * prep;
            // Поиск удаляемого узла
            while (del && del->inf != x) {
                pdel = del;
                if (x < del->inf )  del = del->left;
                                    else del = del->right;
            }

            if (!del) return;
```

```
// Поиск замещающего узла
        prep = del;
    if (!del->left) rep = del->right;
    else {
                rep = del->left;
                while (rep->right != NULL) {
                    prep = rep;
                    rep = rep->right;
                }
     }
```

```
// Замена удаляемого узла
            if (rep){   // Если не лист
if (prep->left == rep) prep->left = rep->left;
                    else   prep->right = rep->left;
                  rep->right = del->right;
                  rep->left = del->left;
                   }
```

```
// Подключение замещающего узла
    if (del == root) root = rep; // Если узел - корень
            else
                if (pdel->left == del) pdel->left = rep;
                else  pdel->right = rep;
            // Удаление узла
            delete del;
        }
```

```
int search_max() { // Поиск максимального
   TNode* p = root;
      while (p->right != nullptr) p = p->right;
      return p->inf;
   }

   int search_min() // Поиск минимального
      TNode* p = root;
         while (p->left != nullptr) p = p->left;
      return p->inf;
   }
```

```
TNode* search(int key) // Поиск по ключу
    {
        TNode* p = root;
        while (p)
        {
            if (p->inf == key) return p;
            if (key < p->inf) p = p->left;
                        else p = p->right;
        }
     }
```

```
TNode* pop_all(TNode* p) // Удаление дерева
    {
        if (!p) return nullptr;
        pop_all(p->left);
        pop_all(p->right);
        delete(p);
    }
};
```

```
void main() {
    TTree s;
s.push(4); s.push(1); s.push(2); s.push(9); s.push(6);
          s.print(s.root); // Выводит: 4 2 1 6 9
   s.pop(4);
   s.print(s.root); // Выводит: 4 2 1 6 9
          cout << "min = " << s.search_min() << endl;
          cout << "max = " << s.search_max() << endl;
   TNode* m = s.search(6);
     if (m) cout << m->inf << endl;
    s.root = s.pop_all(s.root);
    if (s.empty()) cout << "Tree removed";
 }
```