
# Двоичные деревья

Java Developer Level 2

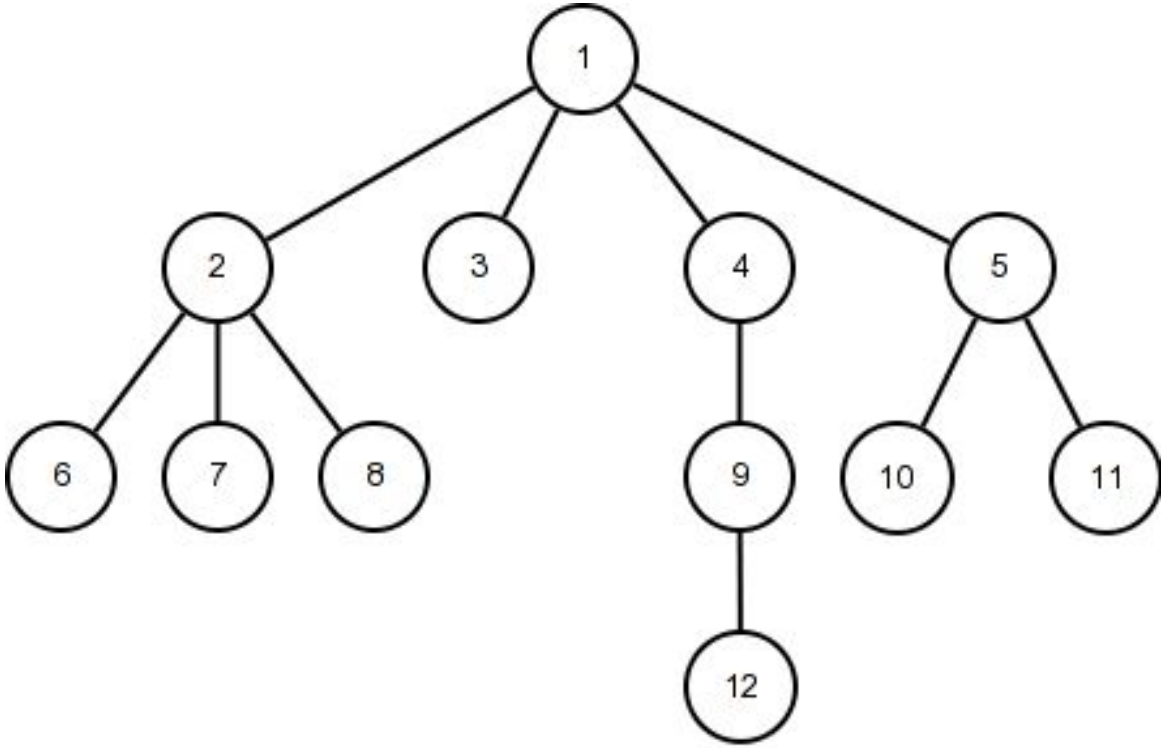
Progwards - Академия компьютерного мастерства

# Содержание лекции


- Дерево
- Двоичное дерево
- Двоичное дерево поиска
- АВЛ-дерево

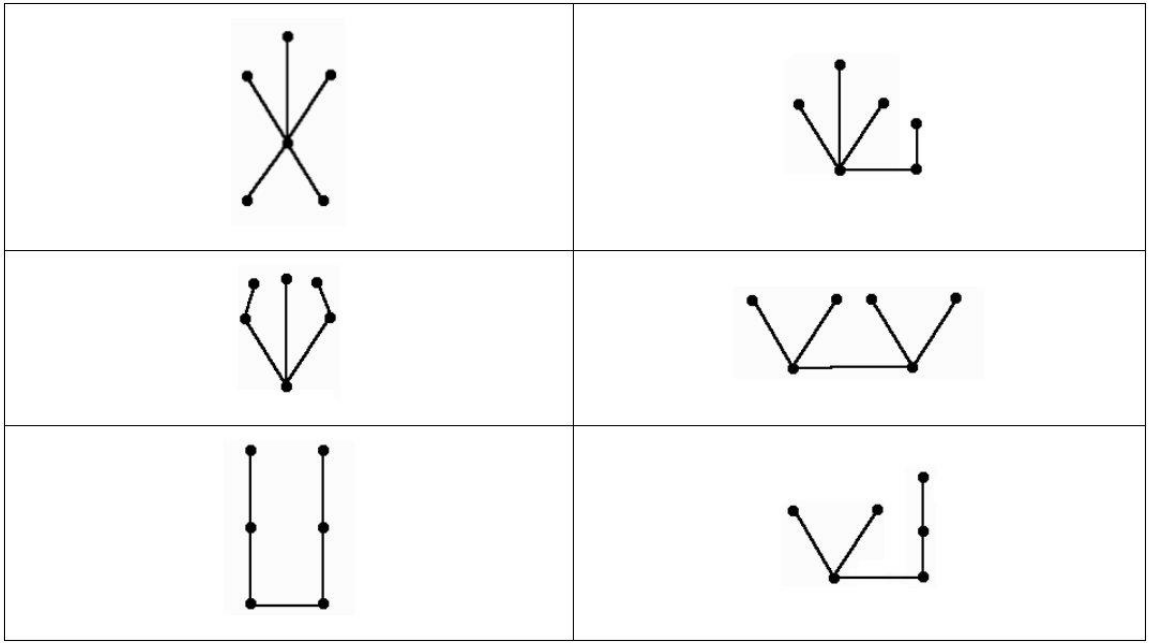
# Что такое дерево

Связанный ацикличный граф.



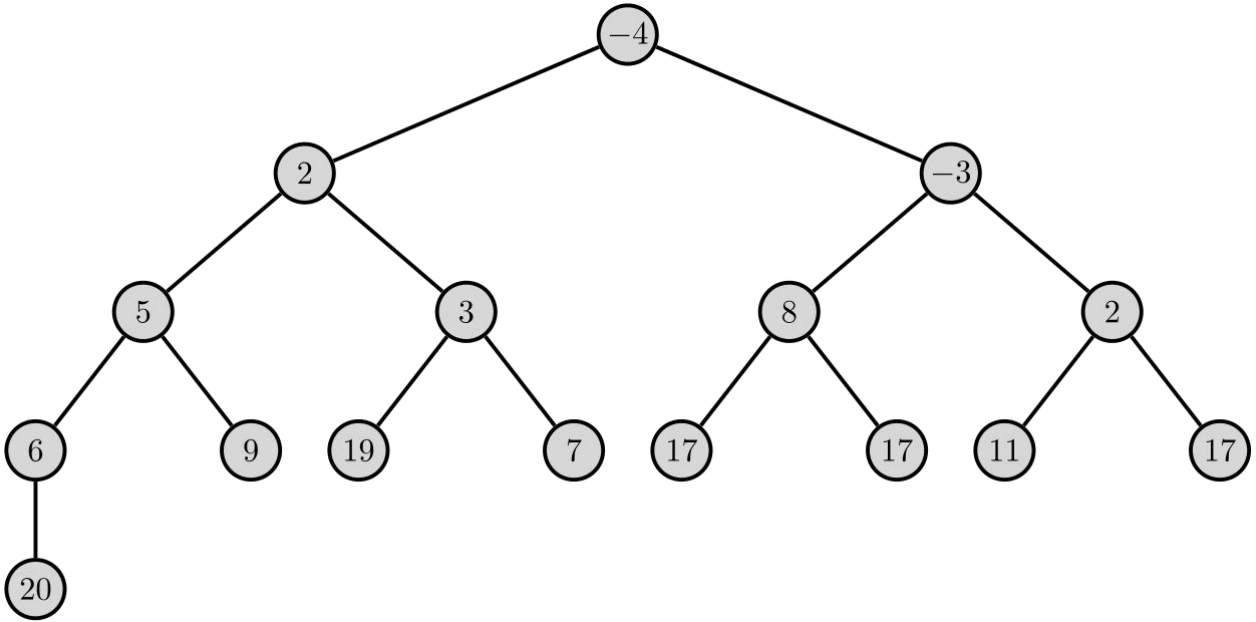
# Свойства дерева

- Отсутствие циклов
- Между парами вершин имеется только  один путь
- Ребра графа не ориентированные



# Термины

- Узел - данные
- Родитель - узел выше по иерархии
- Потомок - узел ниже по иерархии

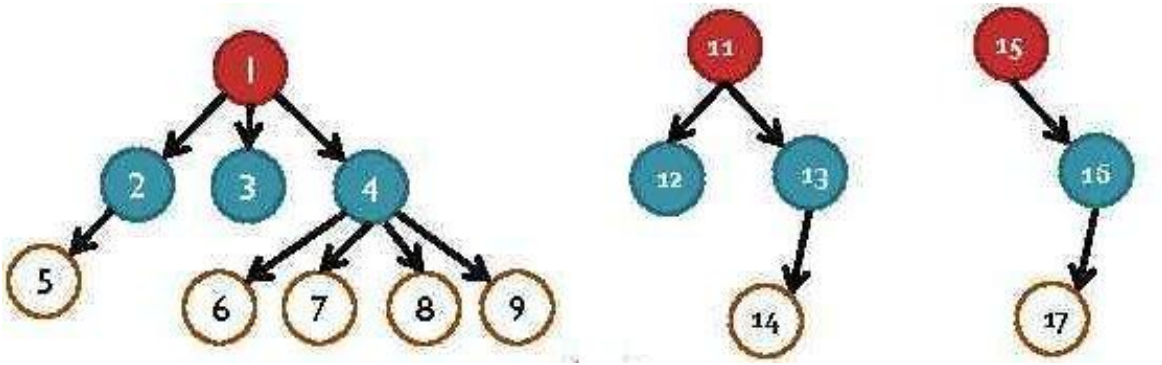


# Термины

- Корневой узел — точка входа в дерево, узел не имеющий предков
- Лист, листовой или терминальный узел — узел, не имеющий дочерних элементов
- Внутренний узел — любой узел дерева, имеющий потомков, и таким образом, не являющийся листовым узлом

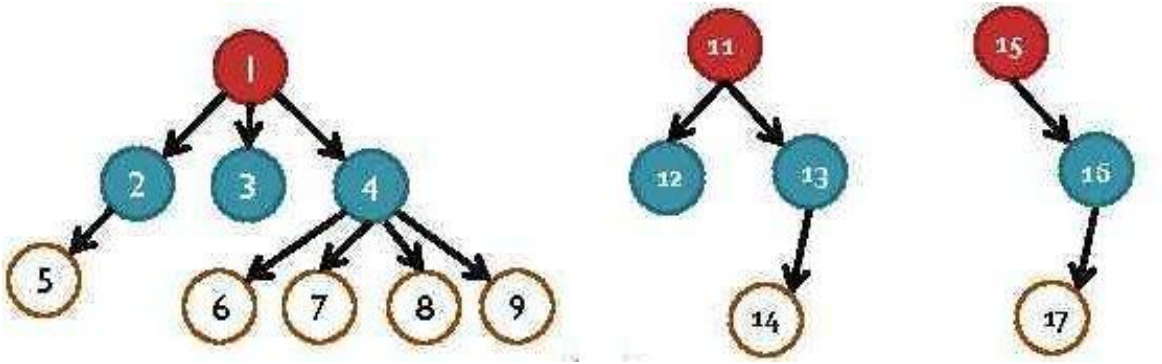
# Термины

- Поддерево - часть дерева с какого либо узла
- Лес - совокупность не связанных деревьев



# Термины

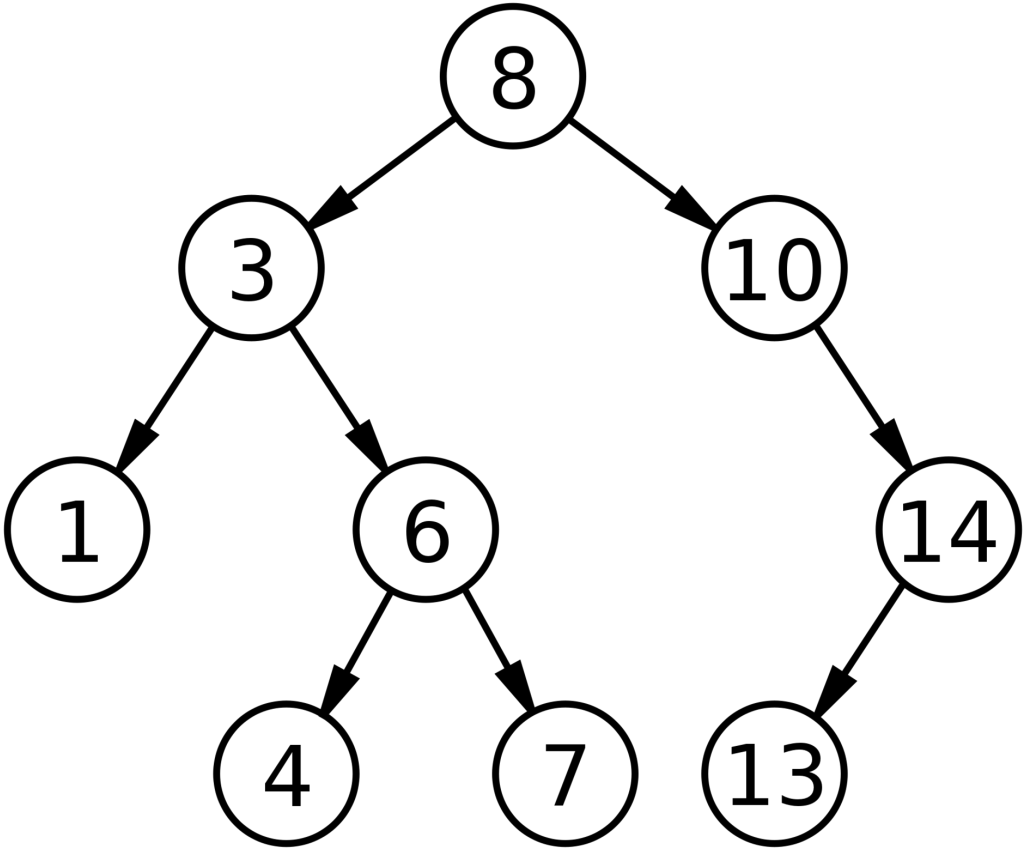
- N-арное дерево, N - макс допустимое количество потомков
- Бинарное (двоичное) дерево
- Список - частный случай дерева



# Двоичные деревья поиска


- N-арность дерева = 2
- Наличие Comparable ключа
- Левое и правое поддерово узла тоже двоичное дерево
- для левого key[left[X]] < key[X]
- для правого key[right[X]] > key[X]

# Двоичное дерево



# Структура данных

```
class TreeLeaf<K, V> {
    K key;
    V value;
    TreeLeaf parent;
    TreeLeaf left;
    TreeLeaf right;
}
```

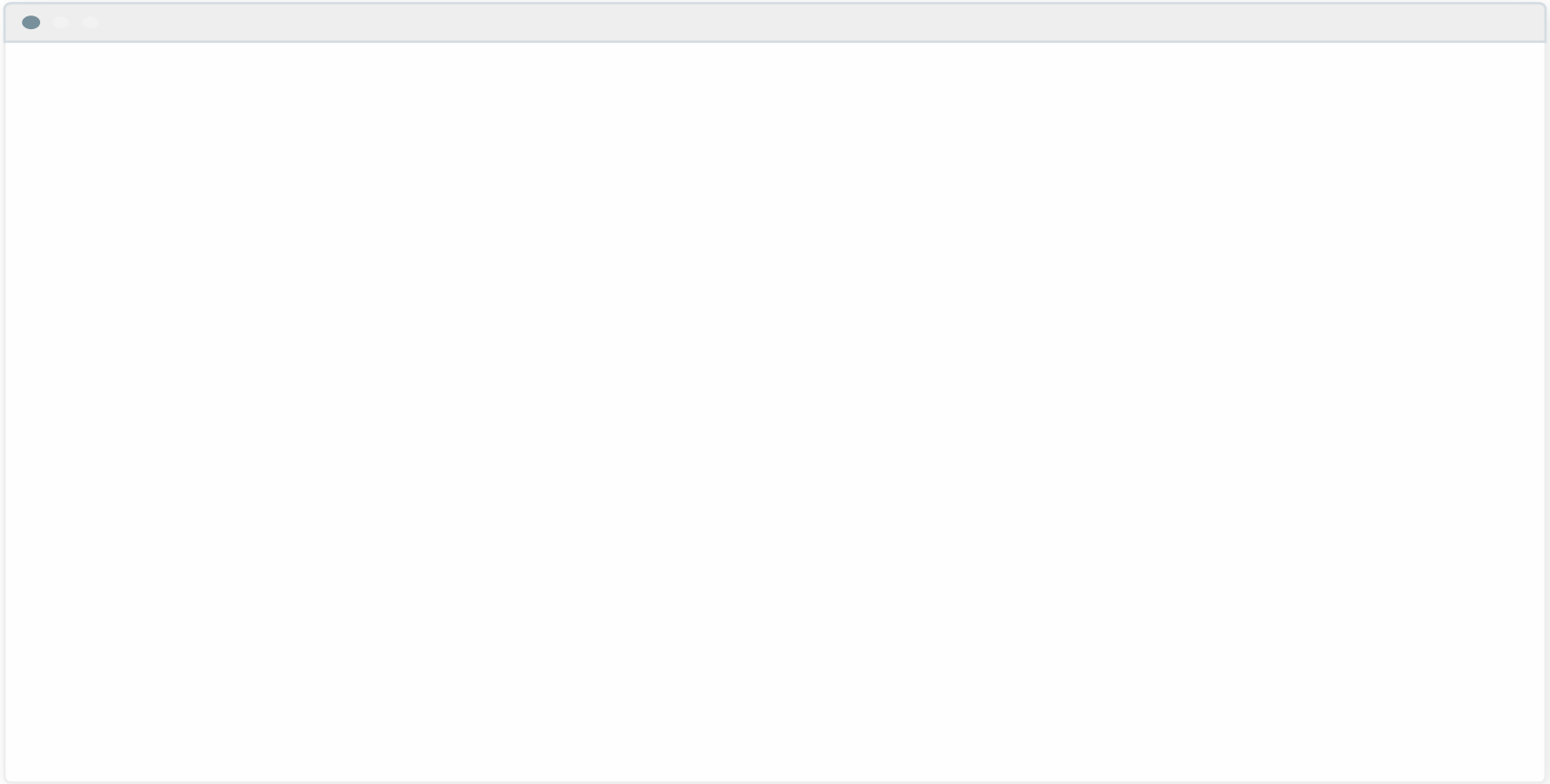
# Структура данных

```
class TreeLeaf<K, V> {
    K key;
    List<V> value;
    TreeLeaf parent;
    TreeLeaf left;
    TreeLeaf right;
}
```

# Алгоритм поиска

```
Найти(ключ):
если поддерево пусто то
  вернуть <ничего не найдено>
если мойКлюч == ключ то
  вернуть моеЗначение
если мойКлюч > ключ то
  вернуть левоеПоддерево.Найти(ключ)
если мойКлюч < ключ то
  вернуть правоеПоддерево.Найти(ключ)
```

# Вставка

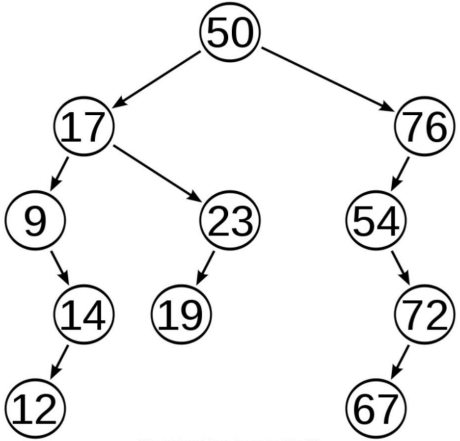


# Удалить текущий узел

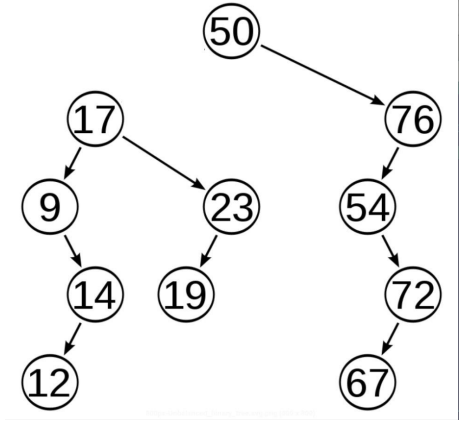
```
УдалитьУзел():
    если родитель не пусто то
        обнулить ссылку на себя у родителя
        если левоеПоддерево не пусто то
            родитель.Вставка(левоеПоддерево)
        если правоеПоддерево не пусто то
            родитель.Вставка(правоеПоддерево)
    иначе
        если левоеПоддерево не пусто то
            корень = левоеПоддерево
        если правоеПоддерево не пусто то
            корень.Вставка(правоеПоддерево)
```

# Удаление узла 17

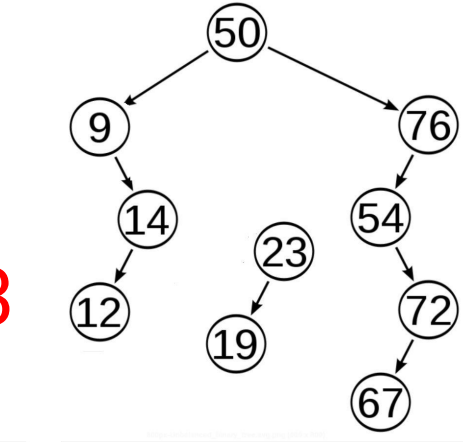
1



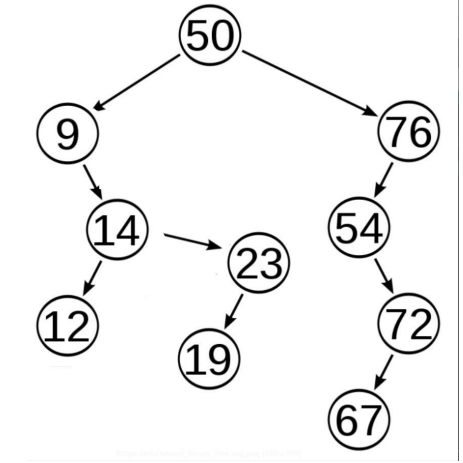

2




3




4



# Алгоритм обхода прямой

```
Обход(действие):
если левоеПоддерево не пустое то
  левоеПоддерево.Обход(действие)
Действие()
если правоеПоддерево не пустое то
  правоеПоддерево.Обход(действие)
```

# Алгоритм обхода обратный

```
Обход(действие):
если правоеПоддерево не пустое то
  правоеПоддерево.Обход(действие)
Действие()
если левоеПоддерево не пустое то
  левоеПоддерево.Обход(действие)
```

# Сложность

| Операция | В среднем | В худшем случае |
|---|---|---|
| Поиск | O(log n) | O(n) |
| Вставка | O(log n) | O(n) |
| Удаление | O(log n) | O(n) |

# АВЛ-деревья

PROGWARDS
Академия компьютерного мастерства

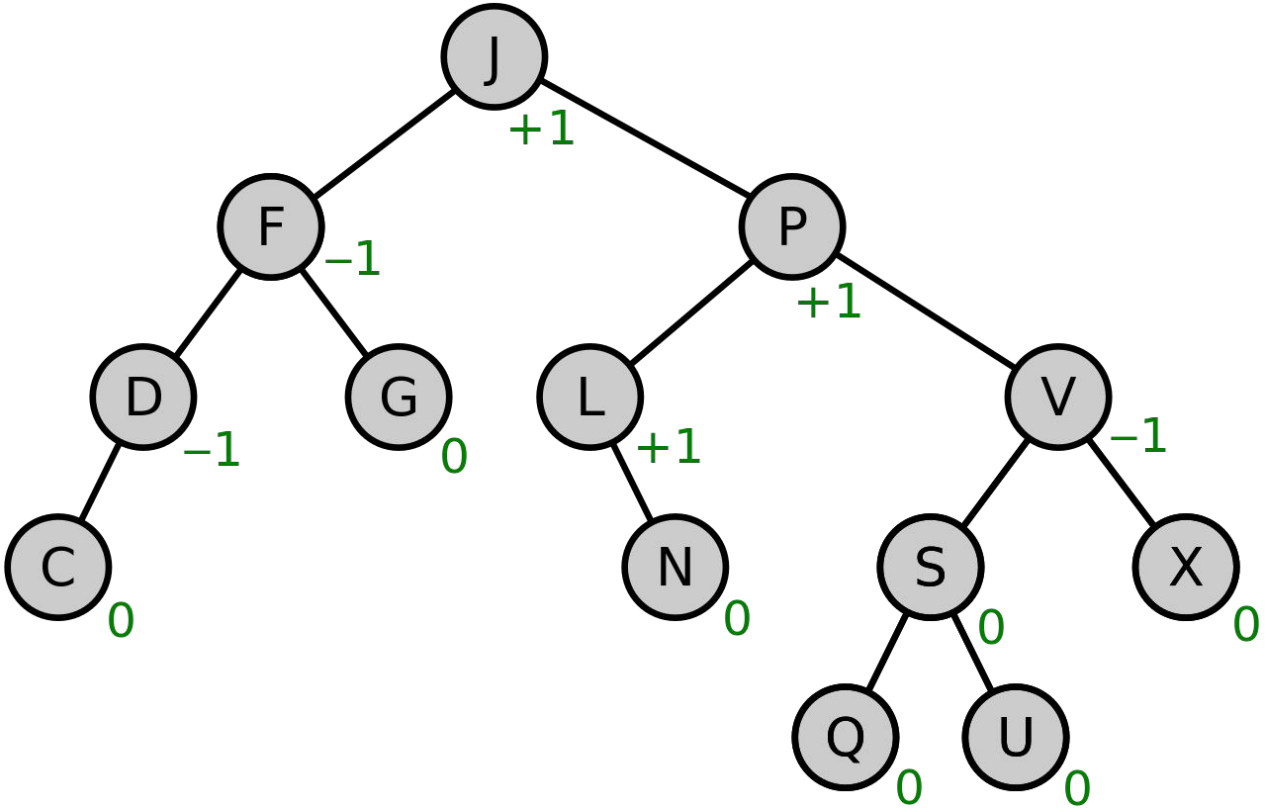
Сбалансированное двоичное дерево поиска

- 1962 год
- Адельсон-Вельский и Ландис

# АВЛ деревья

PROGWARDS
Академия компьютерного мастерства

## В каждой вершине считаем баланс



# Страктура данных, базовые функции

храним **высоту**

```
Высота(узел)
    вернуть узел пусто ? 0 : узел.высота


ПересчитатьВысоту()
    высота = макс(Высота(левое), Высота(правое)) + 1;

Баланс()
    вернуть Высота(левое) - Высота(правое)
```

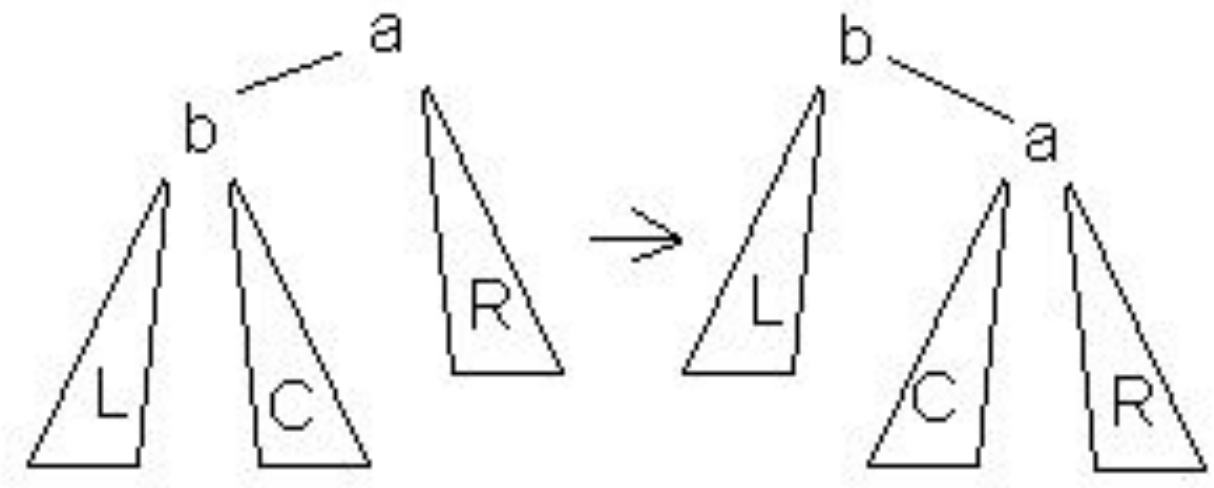
# Алгоритм вставки

PROGWARDS
Академия компьютерного мастерства

```
Вставка(узел)
    ДвоичноеДерево.вставка(узел)
    ПересчитатьВысоту()
    Сбалансировать()
```

# Малое правое вращение, LL

**если** (высота b-поддерева — высота R) == 2 **и**
высота C <= высота L.



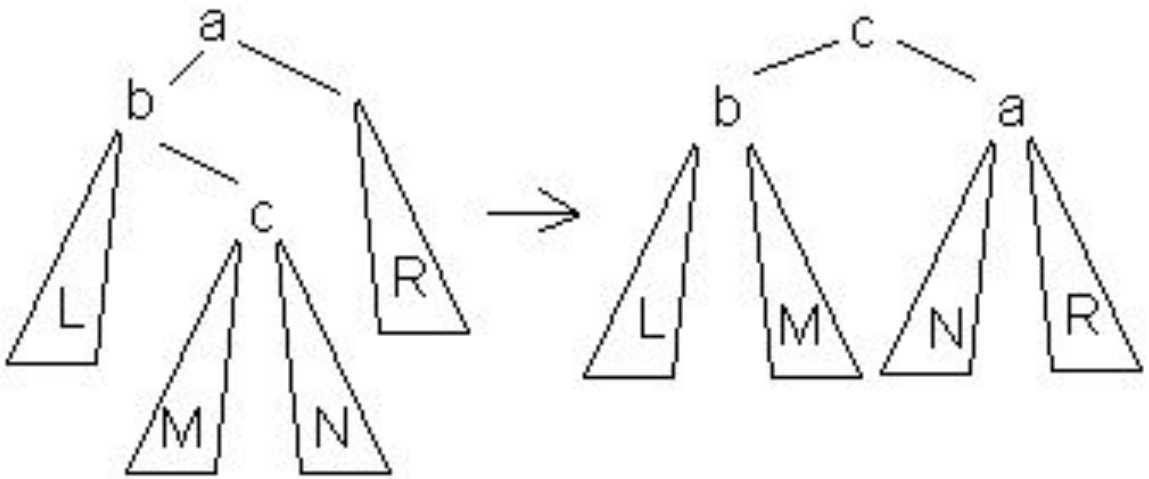
# Малое правое вращение


```
МалоеПравоеВращение()
        b = левое
    c = b.правое
        левое = c
        b.правое = текущий

  b.родитель = родитель
        родитель = b
        c.родитель = текущий
```

# Большое правое вращение, LR

**если** (высота b-поддерева — высота R) == 2 **и** высота c-поддерева > высота L



# Большое правое вращение, LR


БольшоеПравоеВращение()

b = левое

c = b.правое

n = c.правое

m = c.левое

левое = n

b.правое = m

c.правое = **текущий**

c.левое = b

c.родитель = родитель

a.родитель = c

b.родитель = c

n.родитель = **текущий**

m.родитель = b

# Delete

Удалить():

```
если левый не пусто или правый не пусто то
    если баланс > 0 то
        узел = левое.найтиМаксимальный()
    иначе
        узел = правое.найтиМинимальный()


узел.правый = правый
узел.левый = левый
заменить себя у родителя
Сбалансировать() // проверить необходимость
```

# Алгоритм удаления

# DeleteMin


```
УдалитьМинимальный():

    если левый не пусто то
        вернуть левый.НайтиМинимальный()

        родитель.левый = правый
        родитель.Сбалансировать()
        вернуть текущий
```

# Заключение


- Дерево это связный ацикличный граф
- Двоичное дерево, это дерево имеющее не более 2-х потомков
- Двоичное дерево поиска это дерево, упорядоченное по Comparable ключу
- АВЛ-дерево это сбалансированное двоичное дерево поиска
- АВЛ дерево хранит высоту уровней, и считает баланс узла как разницу высот левого и правого поддерева