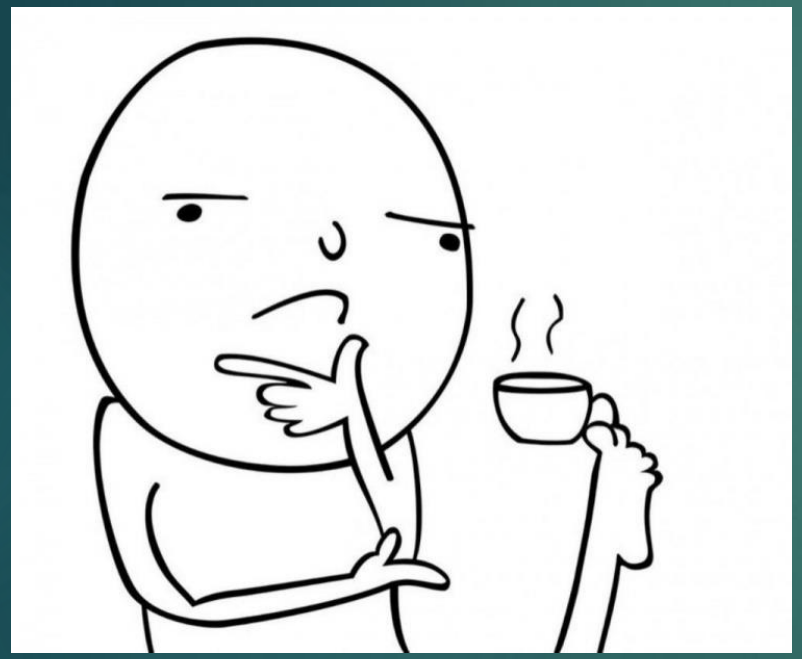
Двоичные деревья

ЛЕКЦИЯ 9

Для чего нужны двоичные деревья?



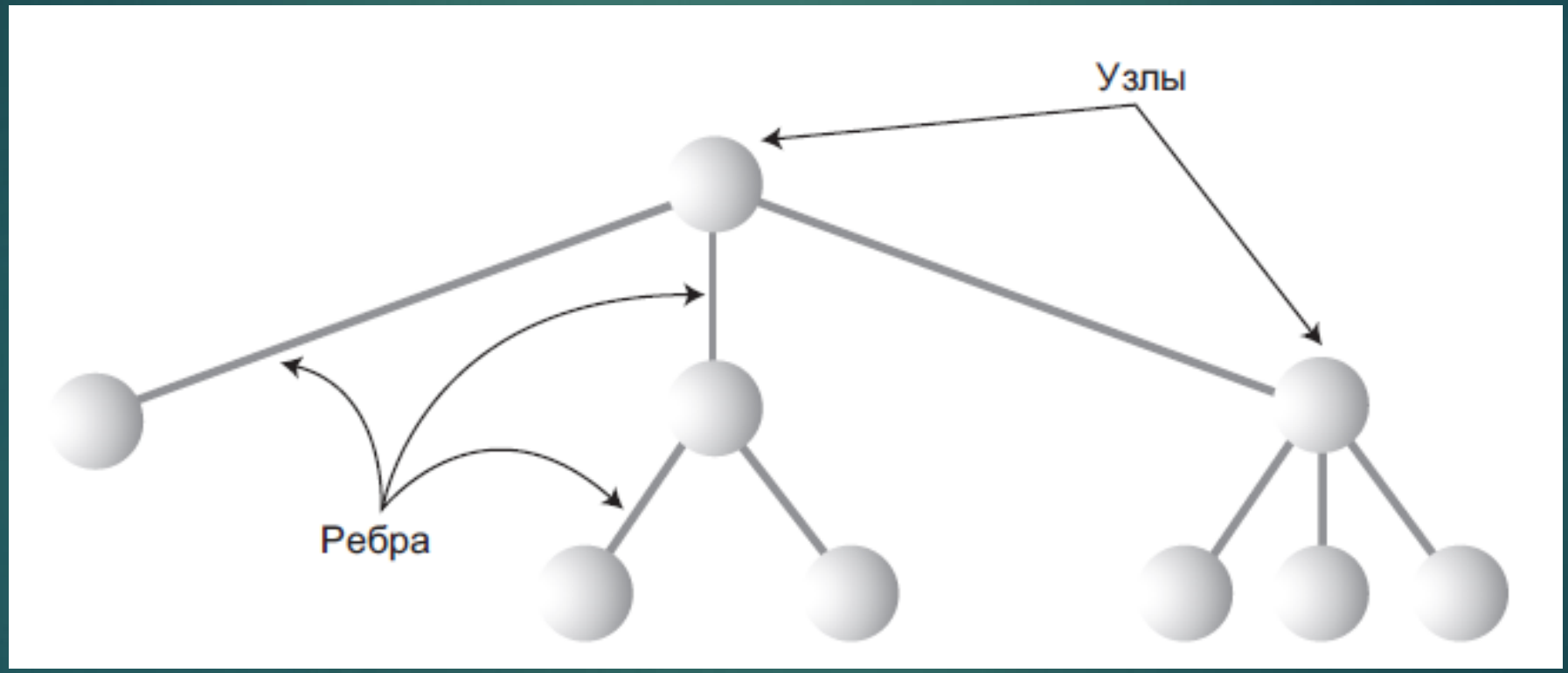
Двоичное дерево сочетает в себе преимущества двух других структур: упорядоченного массива и связанного списка. Поиск в дереве выполняется так же быстро, как в упорядоченном массиве, а операции вставки и удаления элементов так же быстро, как в связанном списке.

Что называется деревом?

Двоичное дерево — иерархическая структура данных, в которой каждый узел имеет не более двух потомков (детей). Как правило, первый называется родительским узлом, а дети называются левым и правым наследниками.

В программах узлы часто представляют сущности: людей, детали машин, забронированные авиабилеты и т. д., то есть типичные элементы, сохраняемые в любых структурах данных. В ООП-языках, к числу которых относится Java, сущности реального мира представляются в виде объектов.

Обобщенное (не двоичное) дерево



Что называется деревом?

Ребра (соединительные линии между узлами) представляют отношения между узлами. Упрощенно говоря, программа может легко (и быстро) перейти от узла к узлу, если между ними имеется соединительная линия. Более того, переходы между узлами возможны *только* по соединительным линиям. В общем случае перемещение происходит только в одном направлении: от корневого узла вниз.

Терминология

Представьте, как кто-то переходит от узла к узлу по соединяющим их ребрам. Полученная последовательность узлов называется ***путем*** (path).

Узел на верхнем уровне дерева называется ***корневым узлом*** (***корнем***).

Любой узел (кроме корневого) имеет ровно одно ребро, уходящее вверх к другому узлу. Узел, расположенный выше него, называется ***родительским узлом*** (или просто ***родителем***) по отношению к данному узлу.

Любой узел может иметь одно или несколько ребер, соединяющих его с узлами более низкого уровня. Такие узлы, находящиеся ниже заданного узла, называются его ***потомками***.

Терминология

Узел, не имеющий потомков, называется ***листовым узлом*** (или просто ***листом***).

Любой узел может рассматриваться как корень ***поддерева***, состоящего из его потомков, потомков его потомков и т. д.

Переход программы к узлу (обычно с целью выполнения некоторой операции, например проверки значения одного из полей данных или вывода) называется ***посещением***. Простое прохождение мимо узла на пути от одного узла к другому посещением не считается.

Обходом дерева называется посещение всех его узлов в некотором заданном порядке. Например, все узлы дерева могут перебираться в порядке возрастания ключей. Как будет вскоре показано, существуют и другие способы обхода деревьев.

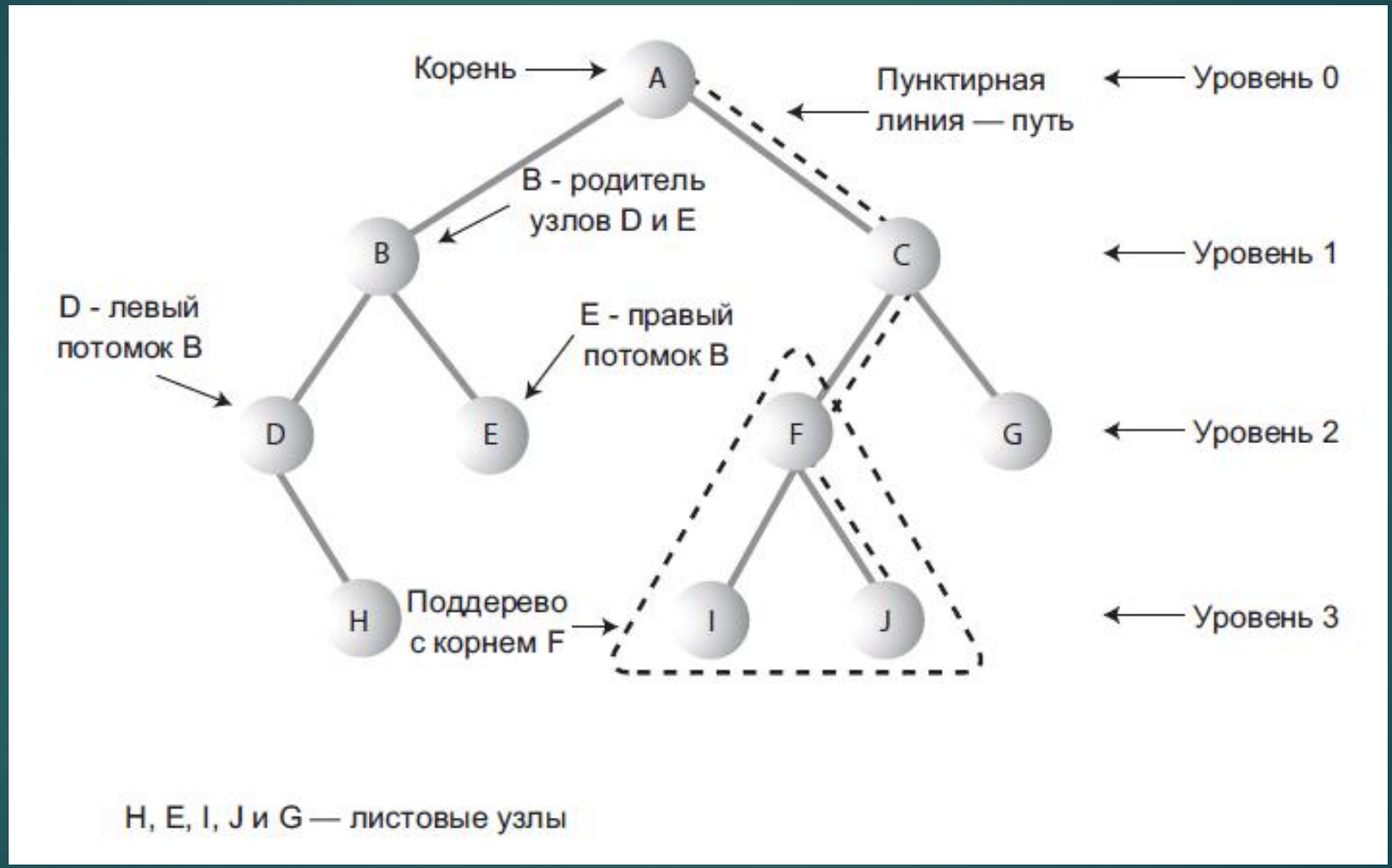
Терминология

Уровнем узла называется количество поколений, отделяющих его от корня. Если считать, что корень находится на уровне 0, то его потомки находятся на уровне 1, потомки потомков — на уровне 2 и т. д.

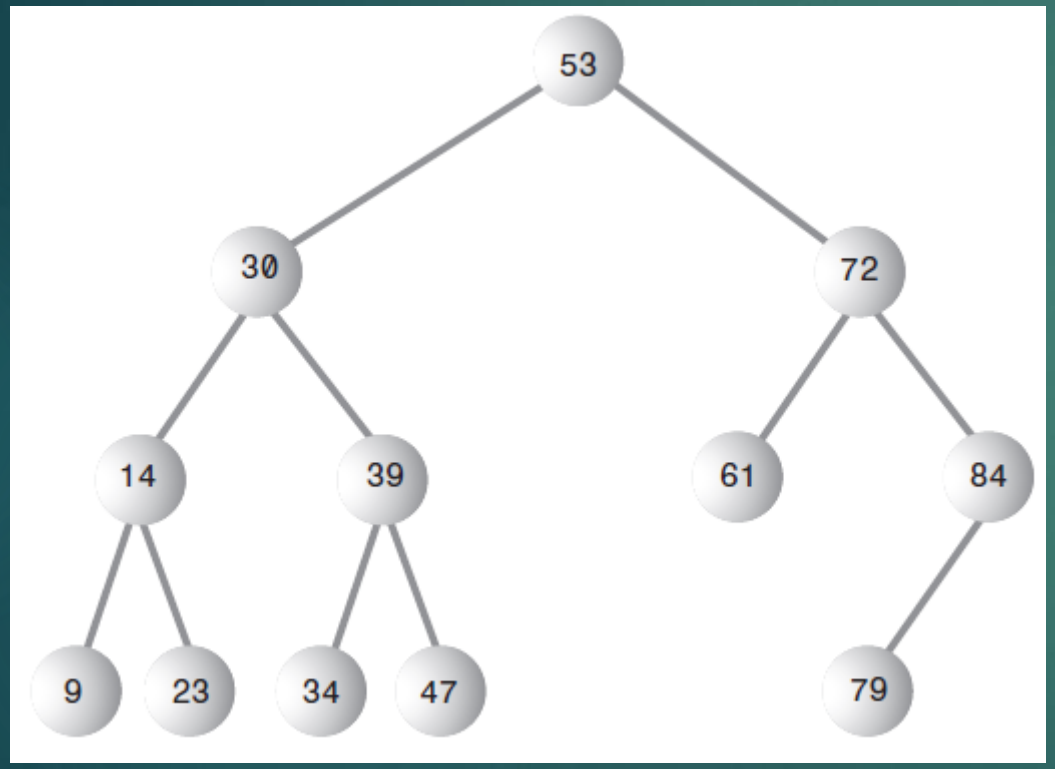
Как вы уже знаете, одно из полей данных объекта часто назначается ***ключевым***. Ключ используется при поиске элемента или выполнения с ним других операций.

Если каждый узел дерева имеет не более двух потомков, такое дерево называется ***двоичным***.

Терминология



Дерево двоичного поиска

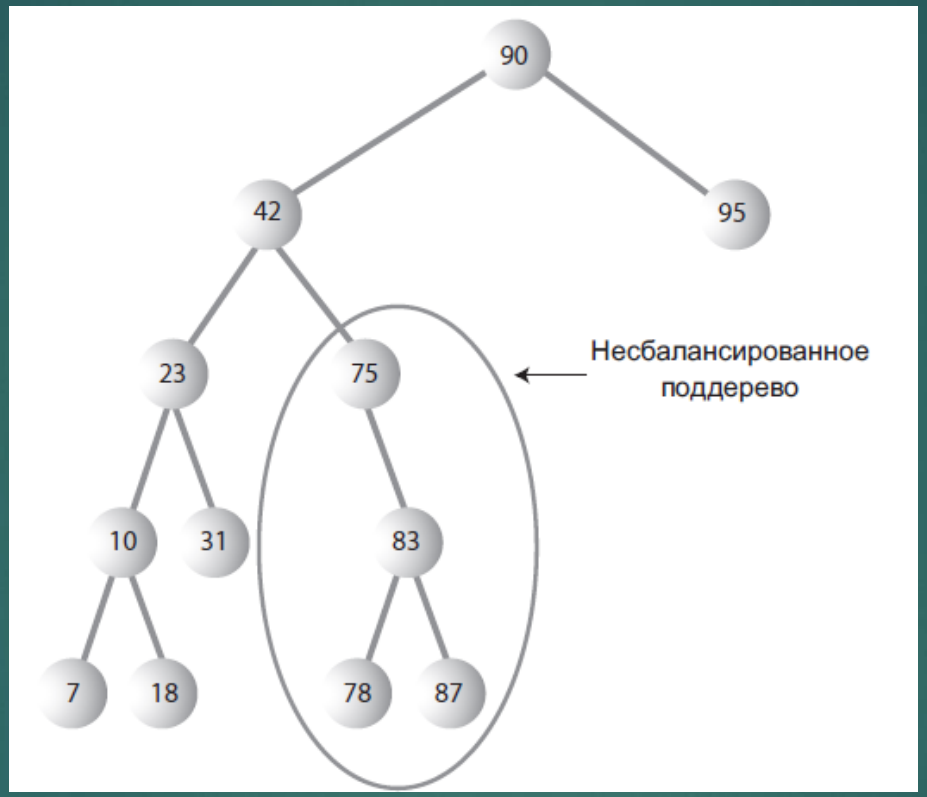


Определяющая характеристика дерева двоичного поиска: ключ левого потомка узла должен быть меньше, чем у родителя, а ключ правого потомка — больше либо равен ключу родителя.

Аналогия

Распространенный пример дерева — иерархическая файловая структура в компьютерной системе. Корневой каталог устройства (во многих системах обозначаемый символом \ — как в C:\) является корнем дерева. Каталоги, непосредственно вложенные в корневой каталог, являются его потомками. Количество уровней вложения подкаталогов может быть сколь угодно большим. Файлы соответствуют листьям, так как они не имеют потомков.

Несбалансированные деревья



Некоторые деревья являются несбалансированными, то есть большинство узлов сосредоточено с одной или с другой стороны корня.

Представление деревьев в коде Java

Для начала нам понадобится класс для представления объектов узлов. Класс содержит данные, представляющие хранимые объекты (например, описания работников для базы данных отдела кадров), а также ссылки на каждого из двух потомков текущего узла.

```
class TreeNode
{
        int iData; // Данные, используемые в качестве ключа
        double fData; // Другие данные
        TreeNode leftChild; // Левый потомок узла
        TreeNode rightChild; // Правый потомок узла

        public void displayNode()
        {
                System.out.print('{');
                System.out.print(iData);
                System.out.print(", ");
                System.out.print(dData);
                System.out.print("} ");

        }
}
```

Представление деревьев в коде Java

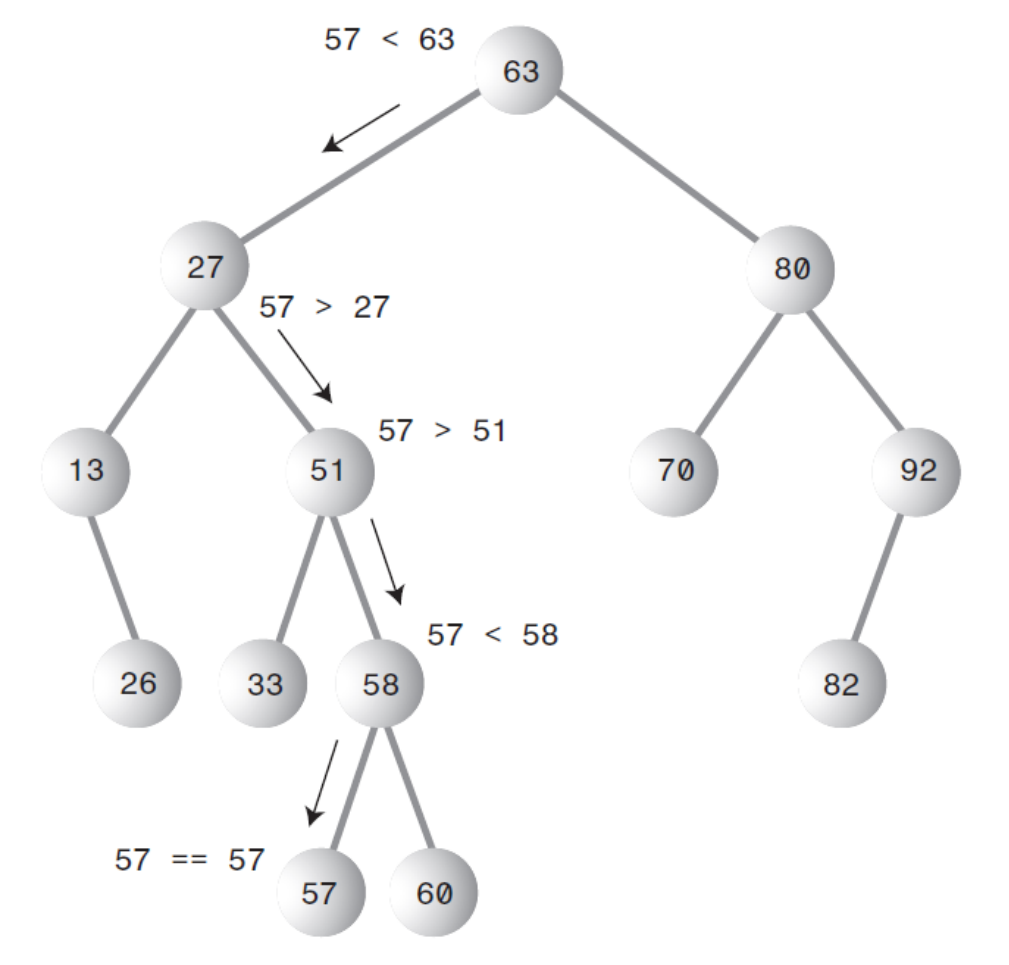
Нам также понадобится класс для представления не отдельных узлов, а всего дерева. Этот класс будет называться Tree. Он содержит только одно поле: переменную Node, в которой хранится корень дерева. Поля для других узлов не нужны, поскольку доступ к ним осуществляется через корневой узел.

```
class Tree
{
        private TreeNode root; // Единственное поле данных
        public void find(int key)
        { }

        public void insert(int id, double dd)
        {}

        public void delete(int id)
        {}
// Другие методы
} // Конец класса Tree
```

Поиск узла

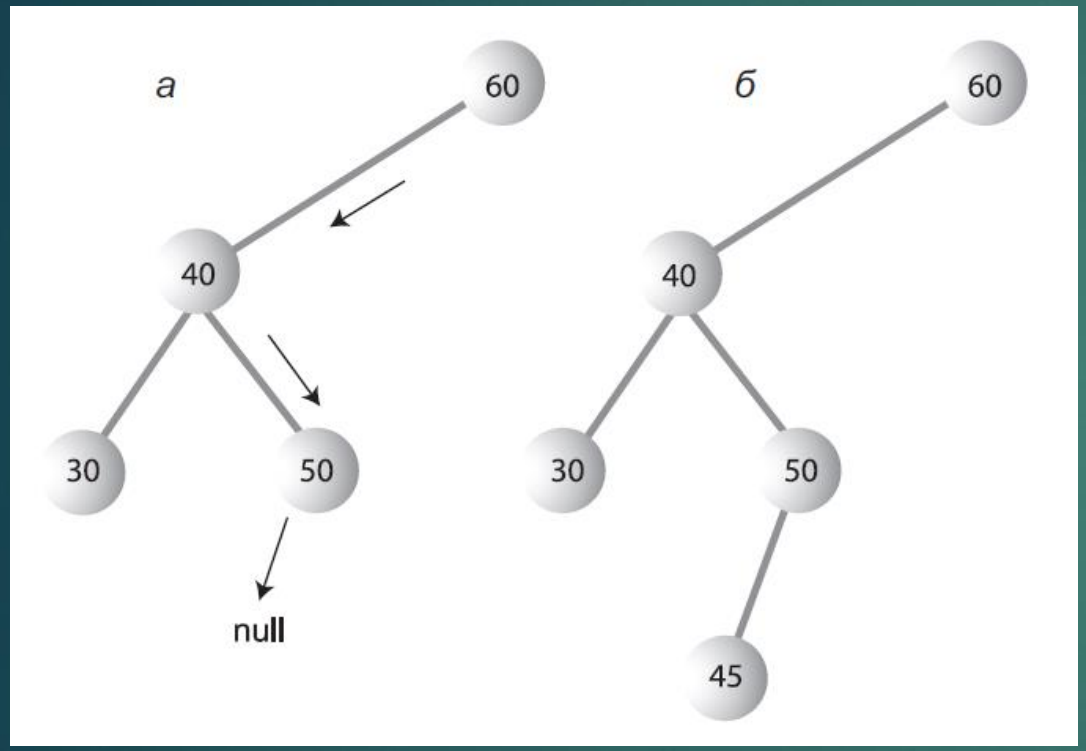



```
public TreeNode find(int key) // Поиск узла с заданным ключом
{ // (предполагается, что дерево не пустое)
      TreeNode current = root; // Начать с корневого узла

      while(current.iData != key) // Пока не найдено совпадение
      {
            if(key < current.iData) // Двигаться налево?
                  current = current.leftChild;
            else
                  current = current.rightChild; // Или направо?
            if(current == null) // Если потомка нет,
                  return null; // поиск завершился неудачей
      }

      return current; // Элемент найден
}
```

Вставка узла



Вставка узла: а — до вставки; б — после вставки

```
public void insert(int id, double dd) {
        TreeNode newNode = new TreeNode (); // Создание нового узла
        newNode.iData = id; // Вставка данных
        newNode.dData = dd;
        if(root==null) // Корневой узел не существует
                root = newNode;
        else { // Корневой узел занят
                TreeNode current = root; // Начать с корневого узла
                TreeNode parent;

                while(true){ // (Внутренний выход из цикла)
                        parent = current;
                        if(id < current.iData){ // Двигаться налево?
                                current = current.leftChild;
                                if(current == null) { // Если достигнут конец цепочки
                                                      // вставить слева
                                        parent.leftChild = newNode;
                                        return;
                                }
                        }
                        else { // Или направо?
                                current = current.rightChild;
                                if(current == null) { // Если достигнут конец цепочки,
                                                      // вставить справа
                                        parent.rightChild = newNode;
                                        return;
                                }
                        }
                }
        }
}
```

Обход дерева

Для деревьев двоичного поиска чаще всего применяется алгоритм симметричного обхода

Симметричный обход

При симметричном обходе двоичного дерева все узлы перебираются в порядке возрастания ключей. Если вам потребуется создать отсортированный список данных двоичного дерева — это одно из возможных решений.

Простейший способ обхода основан на использовании рекурсии. При вызове рекурсивного метода для обхода всего дерева в аргументе передается узел. В исходном состоянии этим узлом является корень дерева. Метод должен выполнить только три операции:

1. Вызов самого себя для обхода левого поддерева узла.

2. Посещение узла.

3. Вызов самого себя для обхода правого поддерева узла.

Не забудьте, что посещение узла подразумевает выполнение некоторой операции: вывод данных, запись в файл и т. д.

Симметричный обход

```
private void inOrder(TreeNode localRoot) {
	if(localRoot != null) {
		inOrder(localRoot.leftChild);
		System.out.print(localRoot.iData + " ");
		inOrder(localRoot.rightChild);
	}
}
```

Прямой и обратный обход

Прямой обход preorder() последовательность выглядит так:

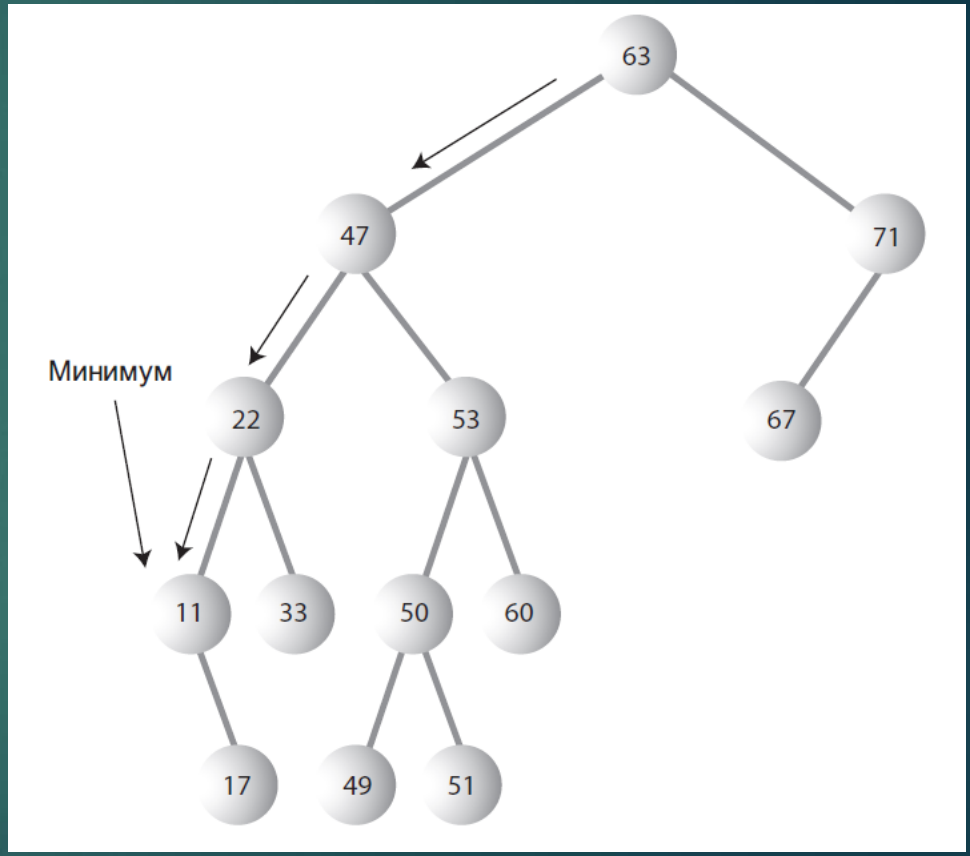
1. Посещение узла.
2. Вызов самого себя для обхода левого поддерева узла.
3. Вызов самого себя для обхода правого поддерева узла.

В третьем алгоритме обхода — обратном — метод выполняет те же шаги в иной последовательности:

1. Вызов самого себя для обхода левого поддерева узла.
2. Вызов самого себя для обхода правого поддерева узла.
3. Посещение узла.

Поиск минимума и максимума

```
public TreeNode minimum() // Возвращает узел с
минимальным ключом
{
	TreeNode current, last;
	current = root; // Обход начинается с корневого узла
	while(current != null) // и продолжается до низа
	{
		last = current; // Сохранение узла
		current = current.leftChild; // Переход к левому
потомку
	}
	return last;
}
```

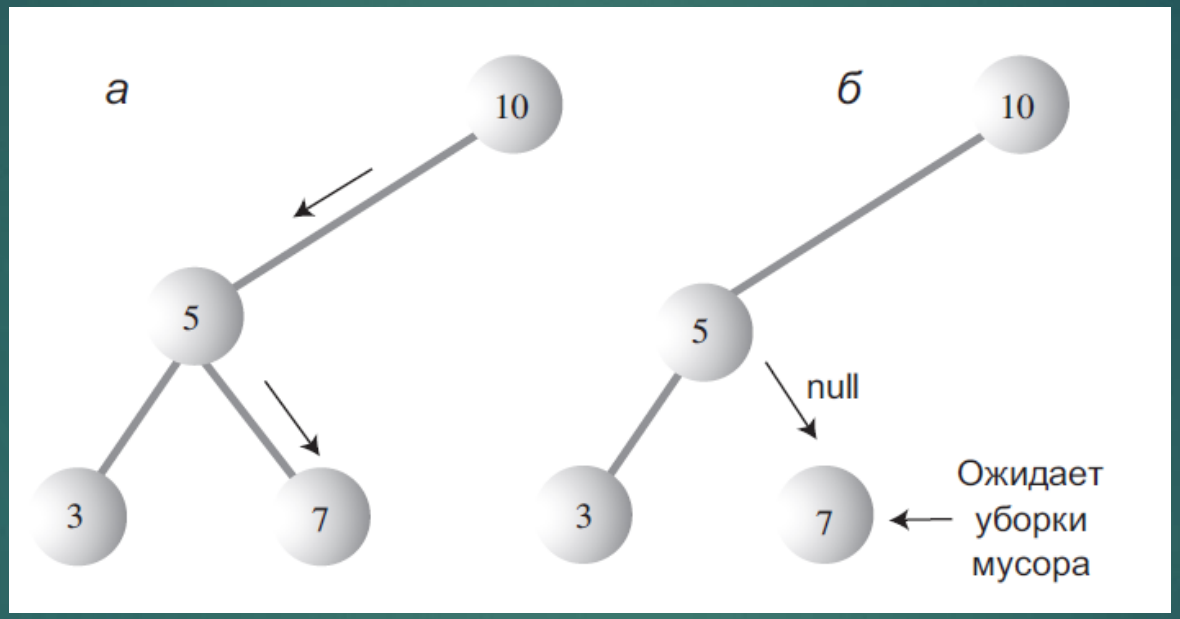


Удаление узла

Удаление узлов является самой сложной из стандартных операций с деревьями двоичного поиска. Удаление начинается с поиска удаляемого узла. Когда узел будет найден, необходимо рассмотреть три возможных случая:

1. Удаляемый узел является листовым (не имеет потомков).
2. Удаляемый узел имеет одного потомка.
3. Удаляемый узел имеет двух потомков.

Случай 1. Удаляемый узел не имеет потомков



Чтобы удалить листовой узел, достаточно изменить поле соответствующего потомка в родительском узле, сохранив в нем null вместо ссылки на узел. Узел продолжает существовать, но перестает быть частью дерева

Случай 1. Удаляемый узел не имеет потомков

```
public boolean delete(int key) { // Удаление узла с заданным ключом (предполагается, что дерево не пусто)
        TreeNode current = root;
        TreeNode parent = root;
        boolean isLeftChild = true;

        while(current.iData != key) { // Поиск узла
                parent = current;
                if(key < current.iData) {  // Двигаться налево?
                        isLeftChild = true;
                        current = current.leftChild;
                }
                else {  // Или направо?
                        isLeftChild = false;
                        current = current.rightChild;
                }

                if(current == null) // Конец цепочки
                        return false; // Узел не найден
}
// Удаляемый узел найден
// Продолжение...}
```

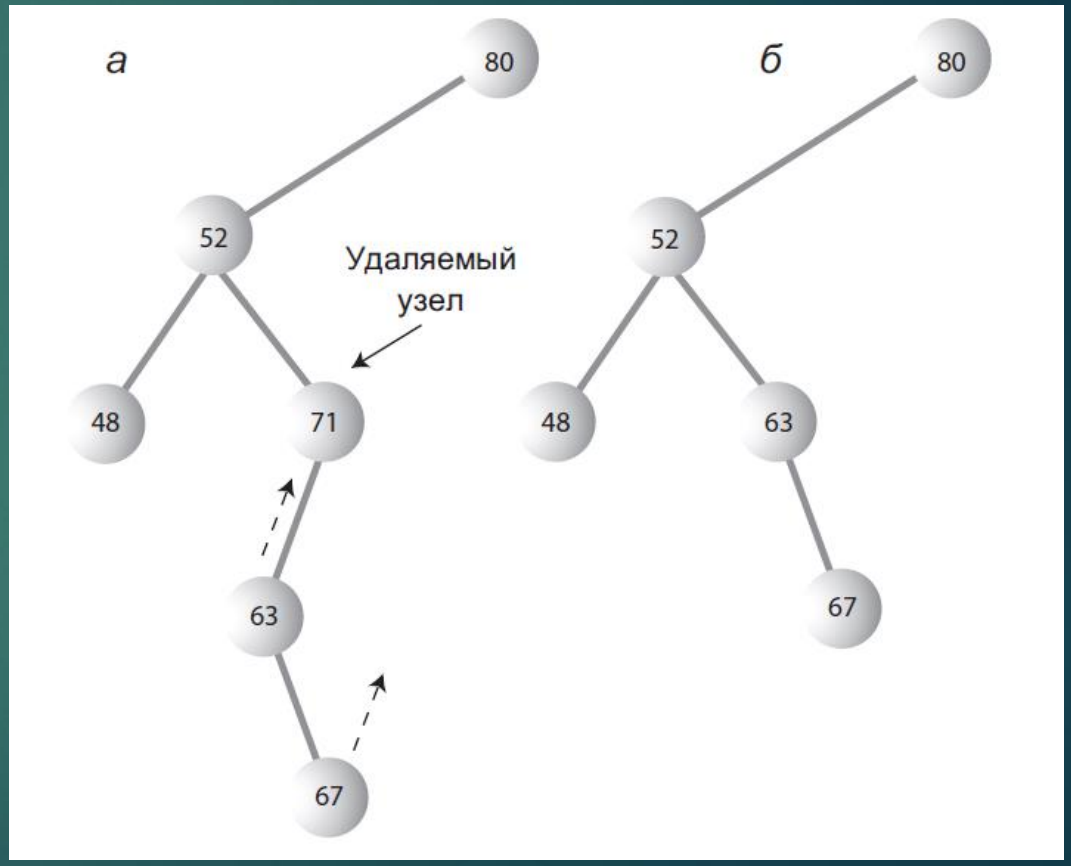
Случай 1. Удаляемый узел не имеет потомков

```
// Продолжение delete()...
// Если узел не имеет потомков, он просто удаляется.
if(current.leftChild==null && current.rightChild==null) {

        if(current == root) // Если узел является корневым,
                root = null; // дерево очищается
        else if(isLeftChild)
                parent.leftChild = null; // Узел отсоединяется
        else // от родителя
                parent.rightChild = null;
}
// Продолжение...
```

Случай 2. Удаляемый узел имеет одного потомка

Второй случай тоже обходится без особых сложностей. Узел имеет только две связи: с родителем и со своим единственным потомком. Требуется «вырезать» узел из этой цепочки, соединив родителя с потомком напрямую. Для этого необходимо изменить соответствующую ссылку в родителе (leftChild или rightChild), чтобы она указывала на потомка удаляемого узла.

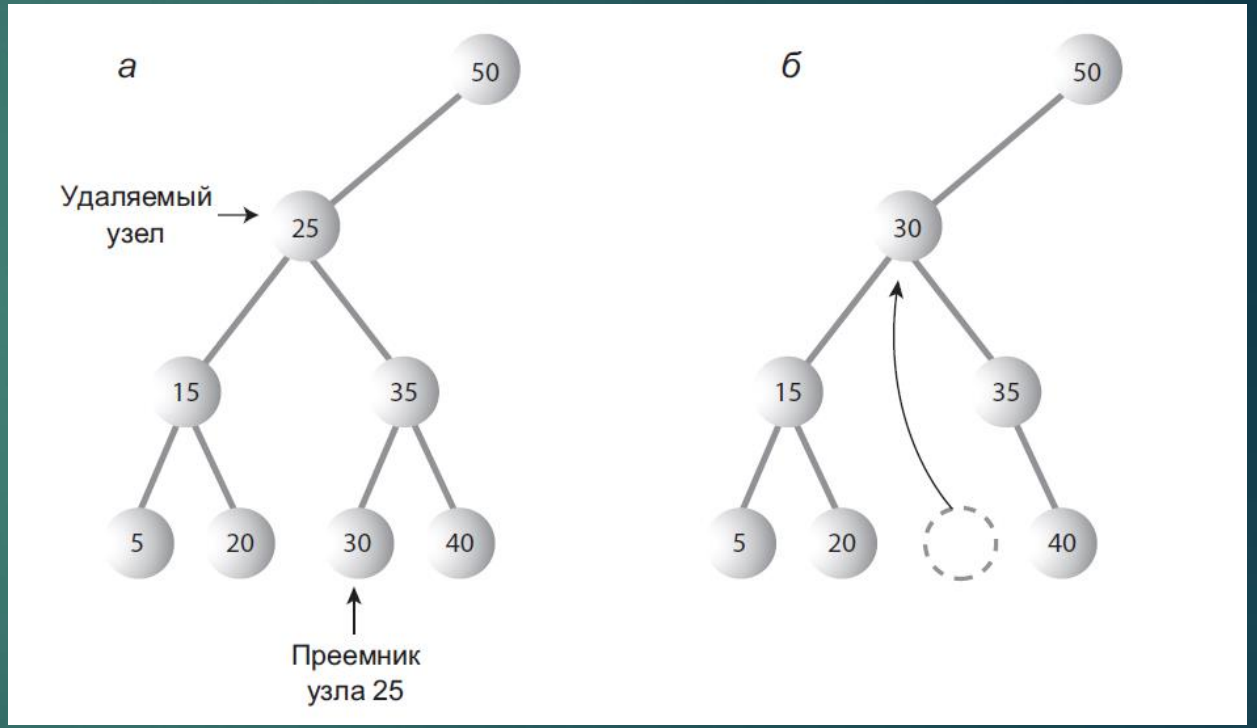


Случай 2. Удаляемый узел имеет одного потомка

```
// Продолжение delete()...
// Если нет правого потомка, узел заменяется левым поддеревом
else if(current.rightChild==null)
        if(current == root)
                root = current.leftChild;
        else if(isLeftChild) // Левый потомок родителя
                parent.leftChild = current.leftChild;
        else // Правый потомок родителя
                parent.rightChild = current.leftChild;
// Если нет левого потомка, узел заменяется правым поддеревом
else if(current.leftChild==null)
        if(current == root)
                root = current.rightChild;
        else if(isLeftChild)  // Левый потомок родителя
                parent.leftChild = current.rightChild;
        else // Правый потомок родителя
                parent.rightChild = current.rightChild;
// Продолжение...
```

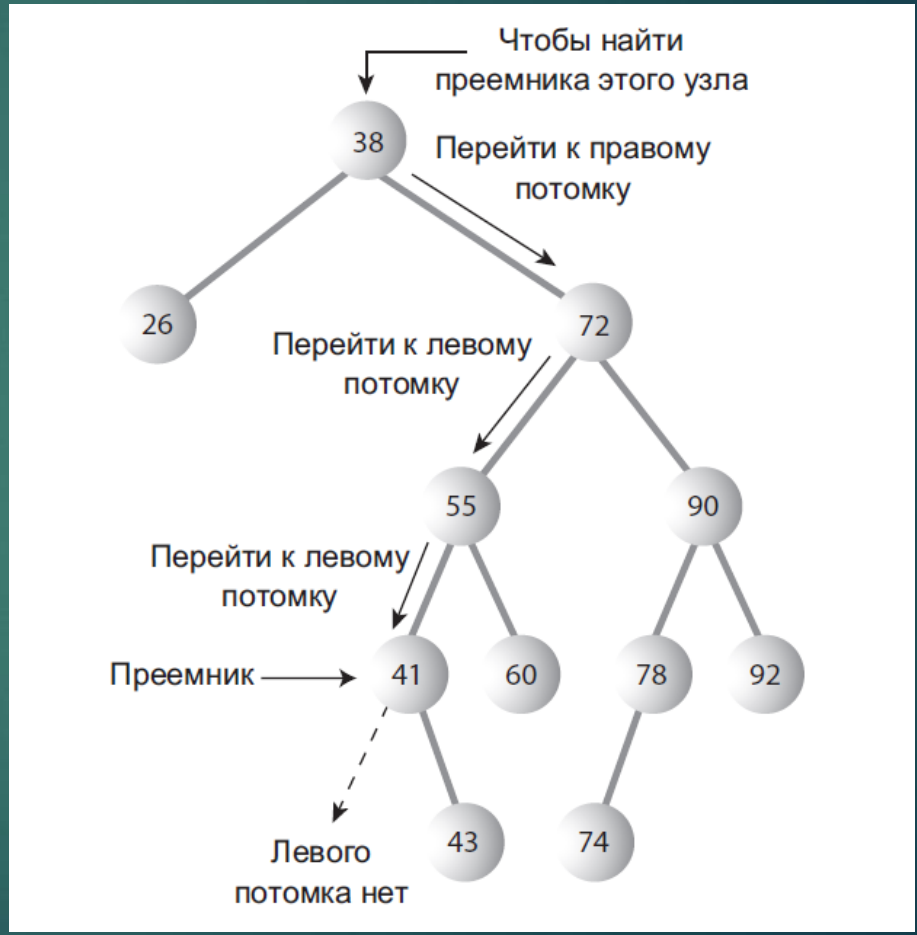
Удаляемый узел имеет двух потомков

Мы работаем с деревом двоичного поиска, в котором узлы располагаются в порядке возрастания ключей. Для каждого узла узел со следующим по величине ключом называется его преемником. Чтобы удалить узел с двумя потомками, замените его преемником.

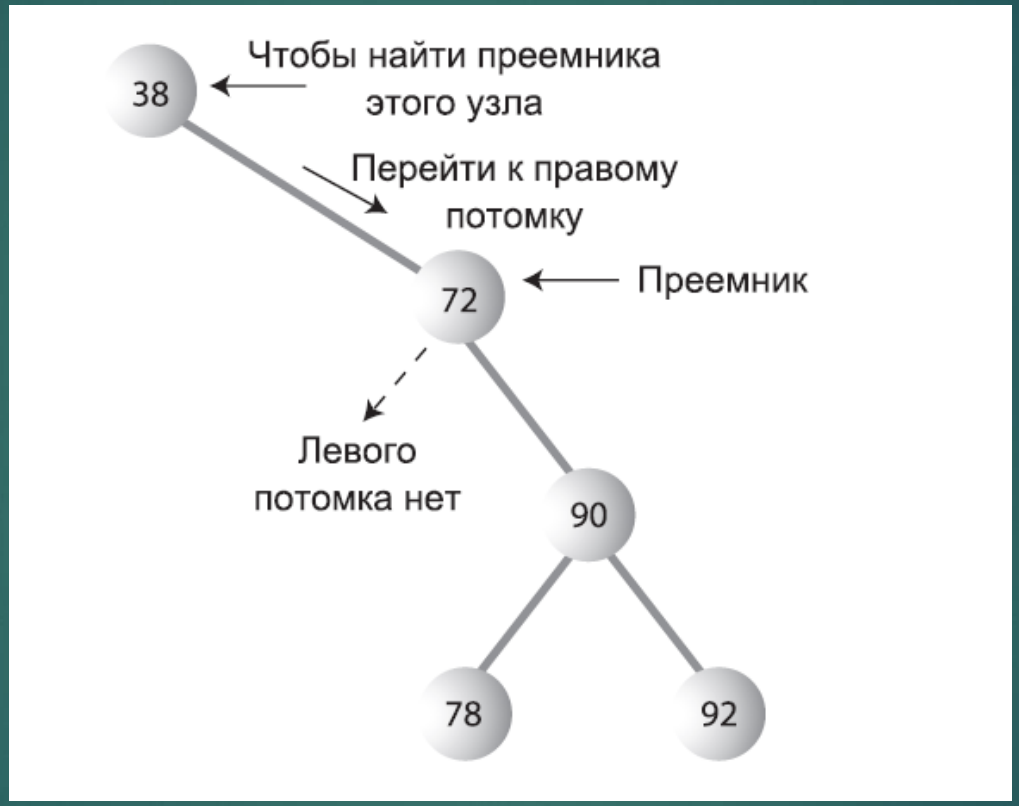


Поиск преемника

Сначала программа переходит к правому потомку исходного узла, ключ которого должен быть больше ключа узла. Затем она переходит к левому потомку правого потомка (если он существует), к левому потомку левого потомка и т. д., следуя вниз по цепочке левых потомков.



Поиск преемника



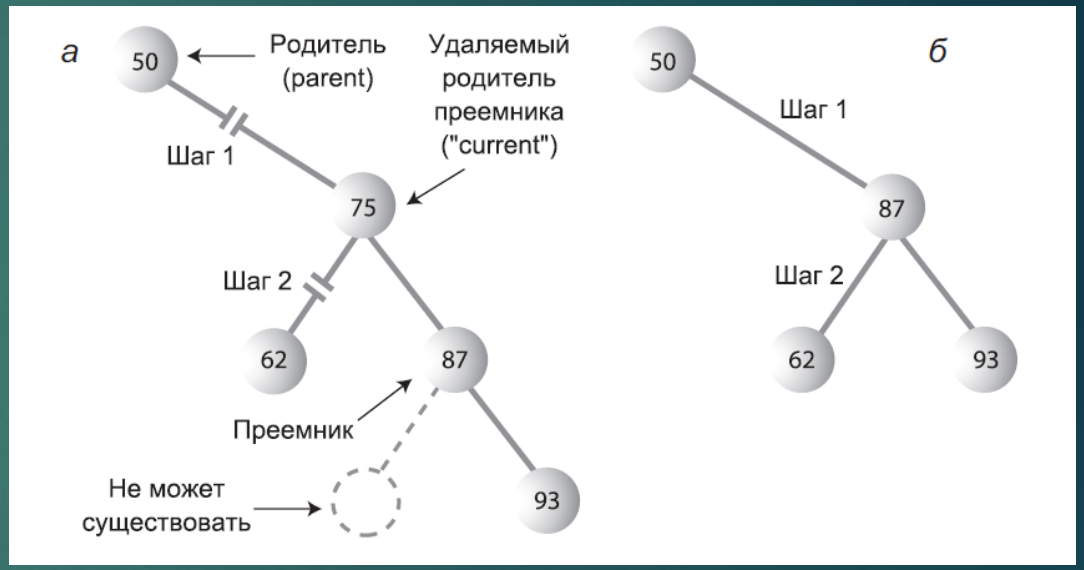
Если у правого потомка исходного узла нет левых потомков, то сам правый потомок становится преемником

Преемник является правым потомком delNode

Если successor является правым потомком current, ситуация немного упрощается, потому что мы можем просто переместить все поддерево, корнем которого является преемник, и вставить его на место удаленного узла. Эта операция выполняется всего за два шага:

1. Отсоединить current от поля rightChild (или leftChild) его родителя. Сохранить в поле ссылку на преемника.

2. Отсоединить левого потомка current от current и сохранить ссылку на него в поле leftChild объекта successor.



Преемник входит в число левых потомков правого потомка delNode

Если преемник входит в число левых потомков правого потомка удаляемого узла, то удаление выполняется за четыре шага:

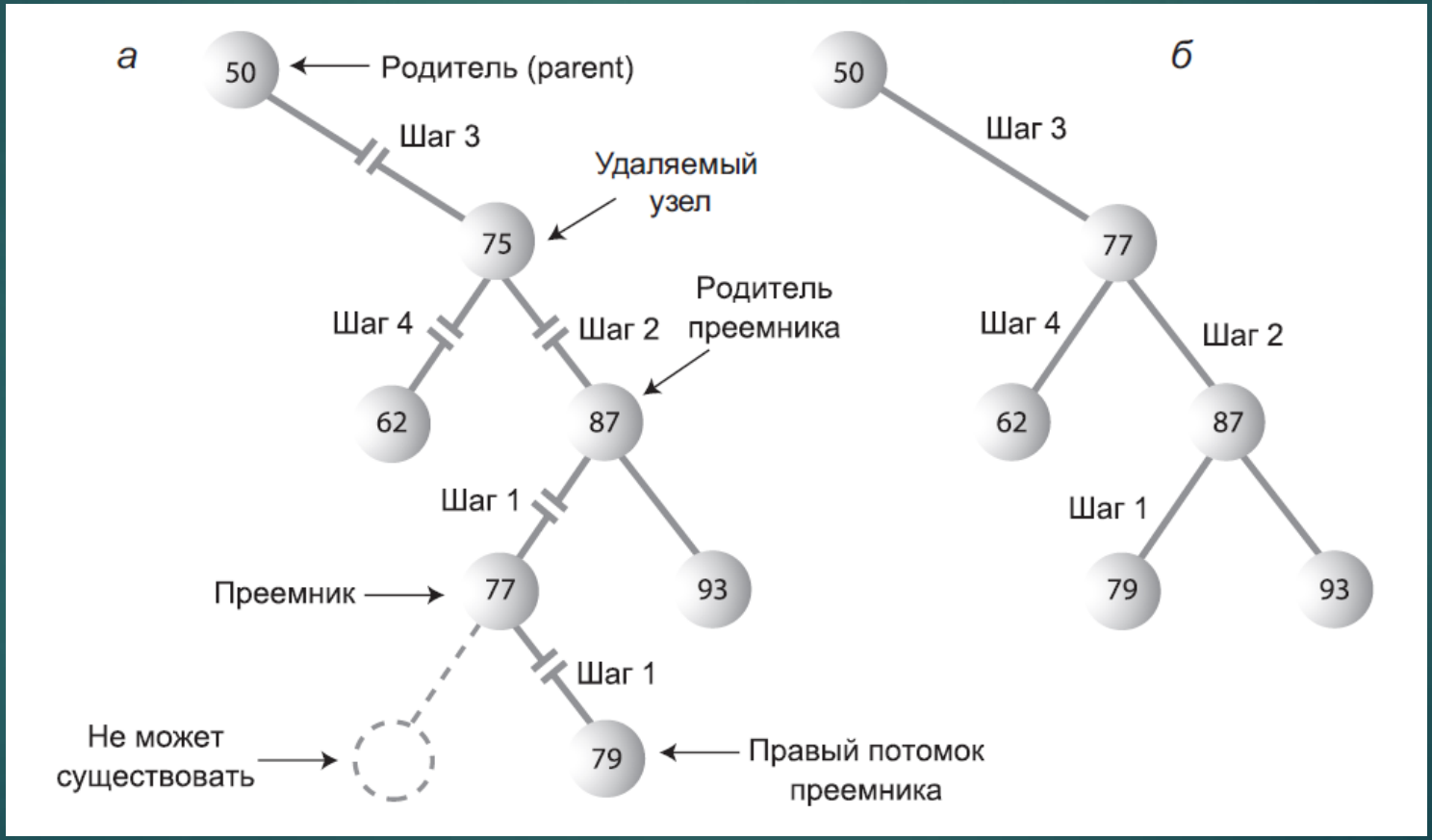
1. Сохранить ссылку на правого потомка преемника в поле leftChild родителя преемника.

2. Сохранить ссылку на правого потомка удаляемого узла в поле rightChild преемника.

3. Убрать current из поля rightChild его родителя и сохранить в этом поле ссылку на преемника successor.

4. Убрать ссылку на левого потомка current из объекта current и сохранить ее в поле leftChild объекта successor.

Преемник входит в число левых потомков правого потомка delNode



```java
public void displayTree()
{
    Stack globalStack = new Stack();
    globalStack.push(root);
    int nBlanks = 32;
    boolean isRowEmpty = false;
    System.out.println("......................................................");
    while(isRowEmpty==false)
    {
        Stack localStack = new Stack();
        isRowEmpty = true;
        for(int j=0; j<nBlanks; j++)
            System.out.print(' ');
        while(globalStack.isEmpty()==false)
        {
            TreeNode temp = (TreeNode)globalStack.pop();
            if(temp != null)
            {
                System.out.print(temp.iData);
                localStack.push(temp.leftChild);
                localStack.push(temp.rightChild);
                if(temp.leftChild != null ||
                        temp.rightChild != null)
                    isRowEmpty = false;
            }
            else
            {
                System.out.print("--");
                localStack.push(null);
                localStack.push(null);
            }
            for(int j=0; j<nBlanks*2-2; j++)
                System.out.print(' ');
        }
        System.out.println();
        nBlanks /= 2;
        while(localStack.isEmpty()==false)
            globalStack.push( localStack.pop() );
    }
    System.out.println("......................................................");
}
```

Эффективность двоичных деревьев

Время выполнения стандартных операций с деревом пропорционально логарифму N по основанию 2. В O-синтаксисе время выполнения таких операций обозначается O(log2N).

Сравните дерево с другими структурами данных, рассматривавшимися до настоящего момента. В неупорядоченном массиве или связанном списке, содержащем 1 000 000 элементов, поиск нужного элемента требует в среднем 500 000 сравнений, но для дерева с 1 000 000 узлов хватает 20 (и менее) сравнений.

Итоги

- Деревья состоят из узлов, соединенных ребрами.
- В двоичном дереве узел имеет не более двух потомков.
- Поиск, вставка и удаление в деревьях выполняются за время O(log N).
- Простейшие алгоритмы обхода — прямой, симметричный и обратный.
- Несбалансированным называется дерево, у которого корень имеет больше левых потомков (во всех поколениях), чем правых (или наоборот).
- Деревья могут представляться в памяти компьютера в виде массива, хотя представление со ссылками является более распространенным.