Двоичное де́рево — структура данных, в которой **каждый** узел (родительский) имеет не более двух потомков (правый и левый наследник).

На практике обычно используются два вида двоичных деревьев — **двоичное дерево поиска** и **пирамида** (куча). Сегодня мы рассмотрим двоичное дерево поиска.

Преимущества двоичного дерева
В чем состоит преимущество хранения данных в виде двоичного дерева поиска? Представьте, что у нас есть справочник на 100 страниц, и нам нужно найти определенную страницу, на которой будет записаны необходимые нам данные. Также мы знаем по содержанию, какая конкретно страница нам нужна. Если мы будет идти обычным путем, придется перелистывать подряд по одной странице, пока не доберемся до необходимой нам. То есть, мы переберем от 1 до 100 страниц, покуда не окажемся на нужном месте. К примеру, если нам нужна 77 страница, то у нас будет 77 переборов. Если говорить о временной сложности, то это будет **O(N)**. Но ведь это же можно сделать более эффективно? Давайте попробуем сделать то же самое, но уже с помощью **двоичного поиска**:

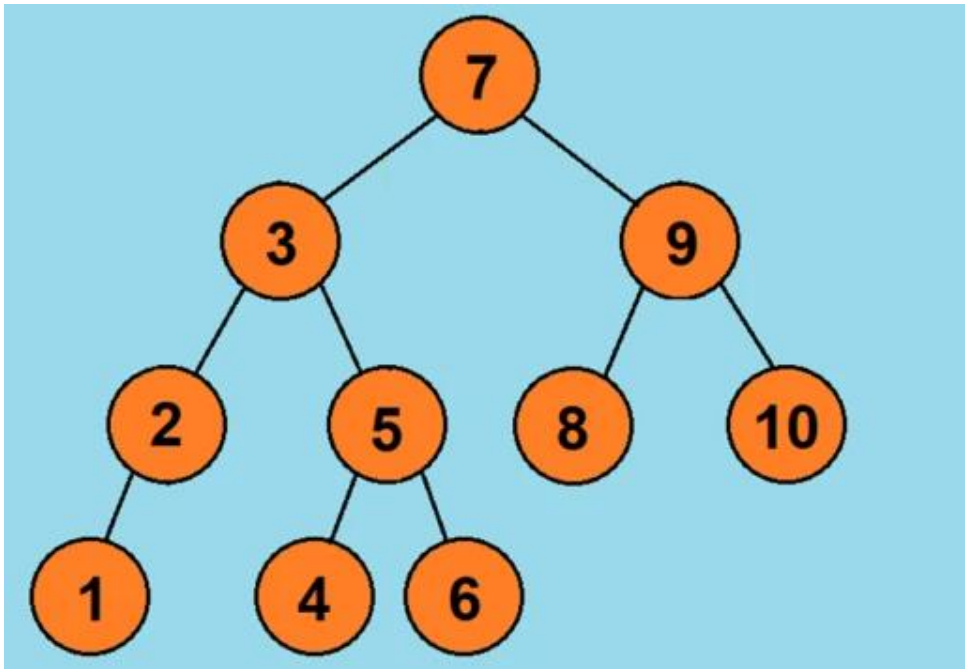
1. Делим справочник на две части, первая — от 1 до 50, вторая 51-100. Мы точно знаем, в которой из этих частей наша страница: если опять же брать 77 страницу — во второй части книги.
2. Далее рассматриваем вторую часть и делим её на две — от 51 до 75, от 76 до 100. В этом случае наша страница будет опять во второй половине, в промежутке 76-100.
3. Далее делим и этот промежуток на 2 и получаем 76-88 и 89-100. Страница входит в первый промежуток, поэтому второй отметаем.
4. Далее промежутки: 76-81 и 82-88, берем первый.
5. 76-78 и 79-81, берем первый.
6. 76 и 77-78, берем второй.
7. 77 и 78, берем первый и получаем нашу страницу — 77.

Вместо 77 шагов нам понадобилось всего 7! Временная сложность данного поиска будет **O(log(N))**.

Правила построения дерева двоичного поиска
Двоичное дерево поиска строится по определенным правилам:

- каждый узел имеет не более двух детей;
- каждое значение, меньшее, чем значение узла, становится левым ребенком или ребенком левого ребенка;
- каждое значение, большее или равное значению узла, становится правым ребенком или ребенком правого ребенка.

К примеру, у нас есть ряд из чисел от 1 до 10. Давайте посмотрим, как будет выглядеть дерево двоичного поиска для этого ряда:



Давайте подумаем, соблюдаются ли тут все условия бинарного дерева:

- все узлы имеют не более двух наследников, первое условие соблюдается;
- если мы рассмотрим к примеру узел со значением 7(или любой другой), то увидим что все значения элементов в левом поддереве будут меньше, в правом — больше. А это значит, что условия 2 и 3 соблюдены.

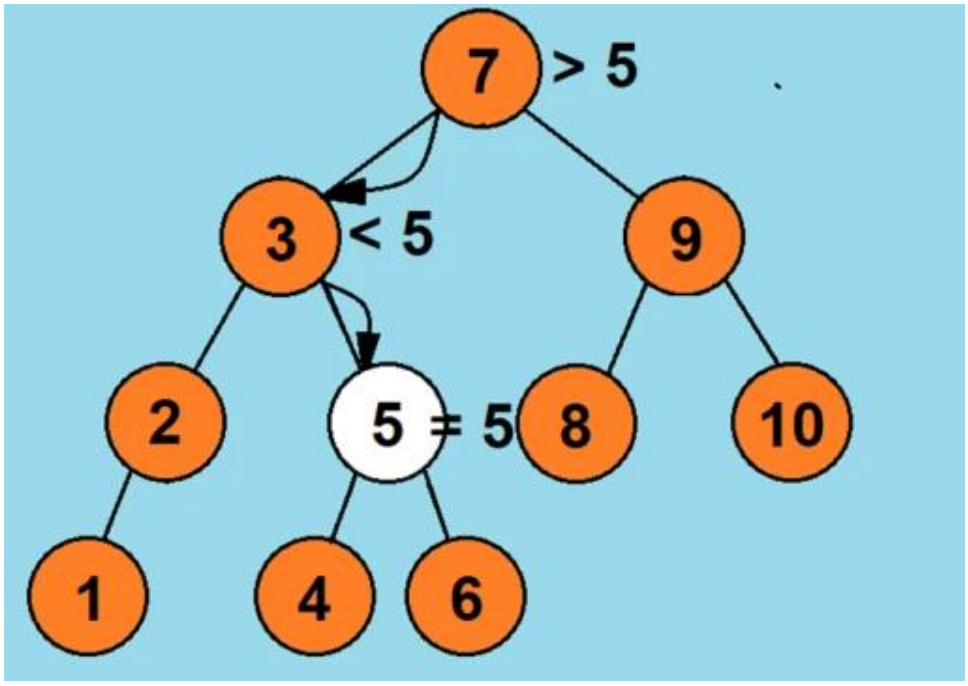
Разберем, каким образом происходят основные операции — вставка, удаление, поиск.

Поиск элемента
При поиске элемента с заданным значением мы начинаем с корневого элемента:

8. Если он равен искомому значению, корневой элемент и есть искомый, если нет — сравниваем значения корневого и искомого.
9. Если искомый элемент больше, мы переходим к правому потомку, если нет — к левому.
10. Если элемент не найден, применяем шаги 1 и 2, но уже к потомку (правому или левому) до тех пор, пока элемент не будет найден.

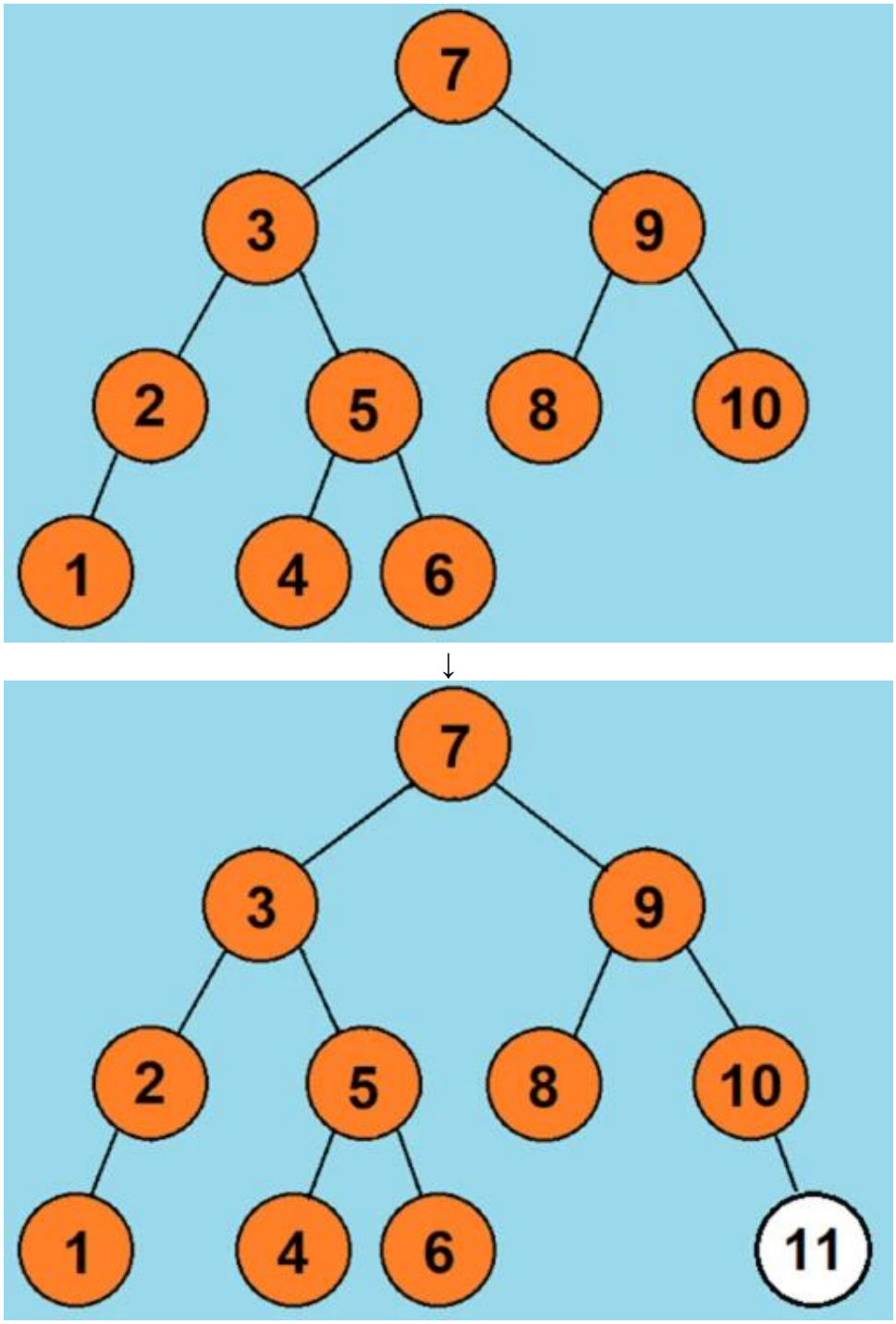
К примеру, в продемонстрированном выше дереве двоичного поиска мы хотим найти элемент со значением 5:

- сравниваем его с корневым элементом, видим, что корневой больше, поэтому мы переходим к левому потомку, который имеет значение 3;
- сравниваем искомый и элемент со значением 3, видим, что искомый больше, поэтому нам нужен правый потомок рассматриваемого элемента, а именно — элемент со значением 5;
- сравниваем этого потомка с искомым и видим, что значения равны — элемент найден.



Вставка элемента
Алгоритм вставки тоже весьма и весьма прост:

11. Сравниваем новый с корневым (если его нет — новый элемент и есть корневой).
12. Если новый элемент:
    - меньше, то переходим к левому наследнику, если его нет, новый элемент и занимает место левого наследника, и алгоритм закончен;
    - больше или равен корневому, то мы переходим к правому наследнику. И аналогично, если данного элемента нет, то новый элемент займет место правого элемента и алгоритм закончен.
13. Для нового рассматриваемого элемента, который был правым или левым из предыдущего шага, повторяем шаги 1 и 2 до тех пор, пока вставляемый элемент не станет на свое место.
    В качестве примера мы захотим вставить в рассматриваемое выше дерево, элемент со значением 11:
    - сравниваем с корневым элементом 7 и видим, что корневой меньше, поэтому переходим к правому наследнику;
    - следующий рассматриваемый элемент имеет значение 9, что меньше чем новый 11, поэтому переходим к правому наследнику;
    - правый наследник имеет значение 10, что опять же меньше, поэтому мы переходим к первому элементу, а так как его нет, то новый элемент со значением 11 и становится на это место.

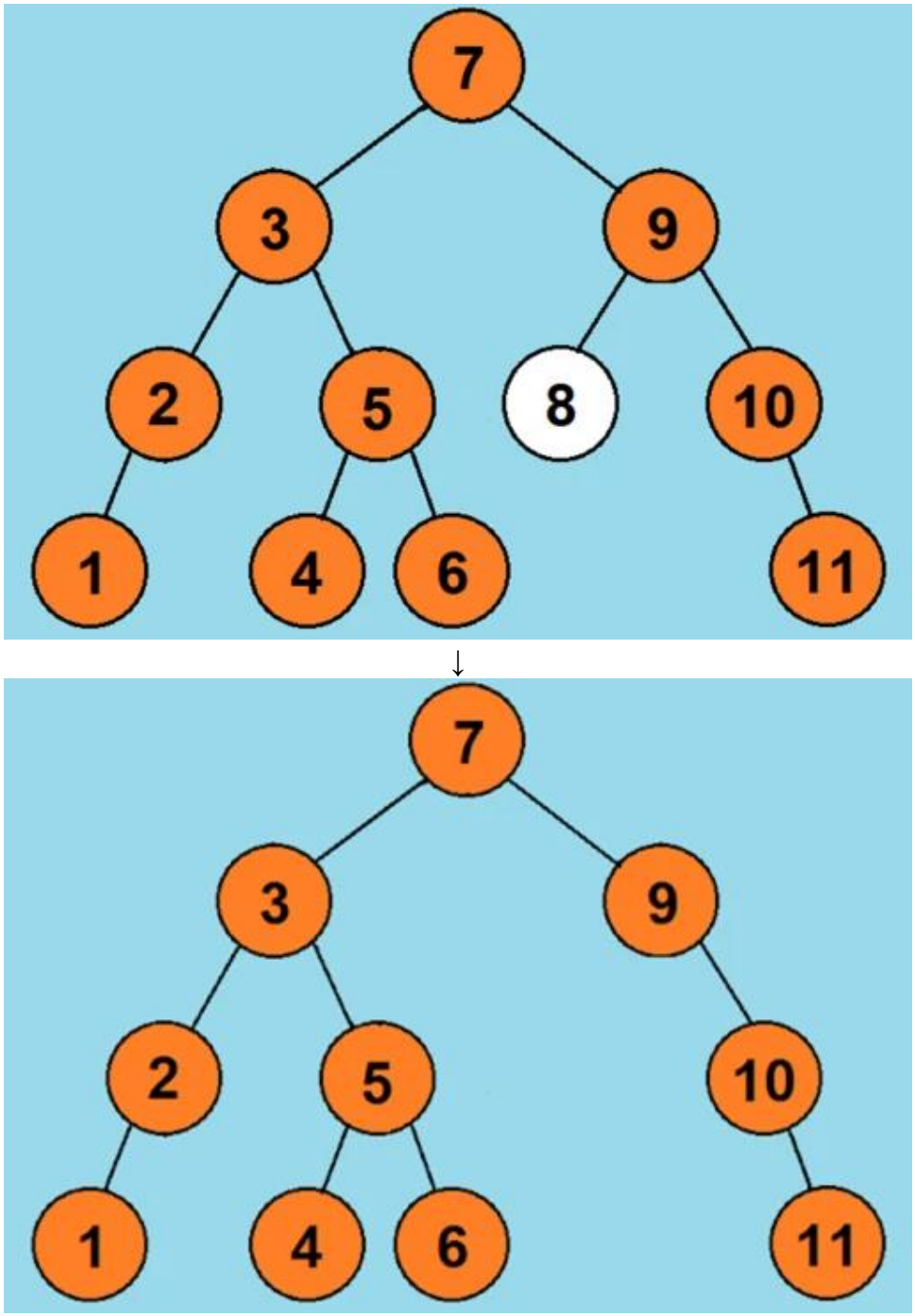


Удаление элемента

Пожалуй, из всех операций с деревьями двоичного поиска, удаление — наиболее сложная. В первую очередь происходит поиск удаляемого элемента, после нахождения которого следует этап, у которого могут быть три вариации:

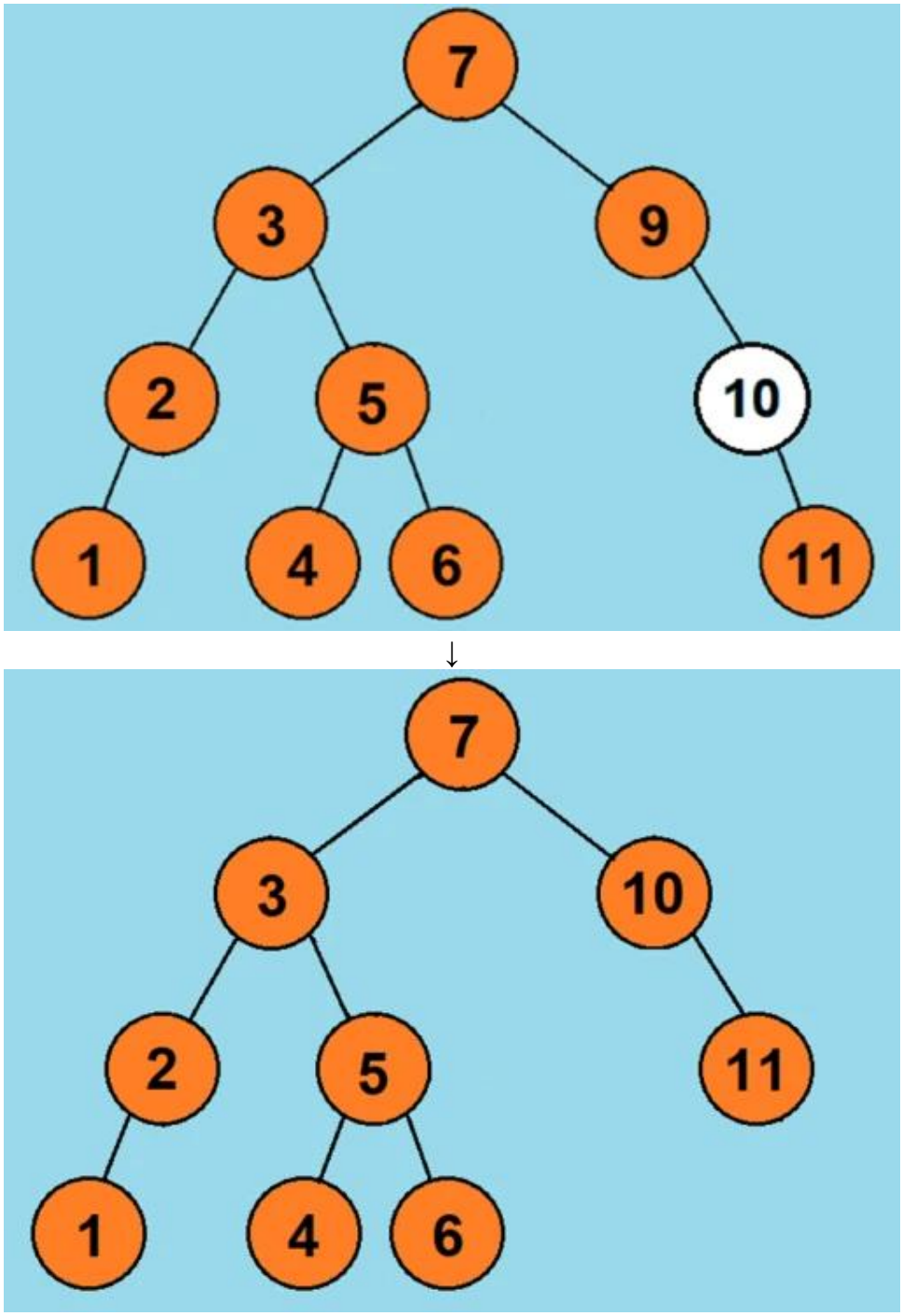
14. Удаляемый узел является листовым (не имеет потомков).

Пожалуй, самый простой. Всё сводится к тому, что мы его просто отсекаем от дерева, без лишних манипуляций. К примеру, из нашего дерева мы удаляем узел со значением 8:



15. Удаляемый узел имеет одного потомка.

В таком случае мы удаляем выбранный узел, и на его место ставим его потомка (по сути просто вырежем выбранный узел из цепочки). В качестве примера удалим из дерева узел со значением 9:



16. Удаляемый узел имеет двух потомков.

Самая интересная часть. Ведь если удаляемый узел имеет сразу двух потомков, нельзя просто так заменить его одним из этих потомков (особенно если у потомка есть собственные потомки).

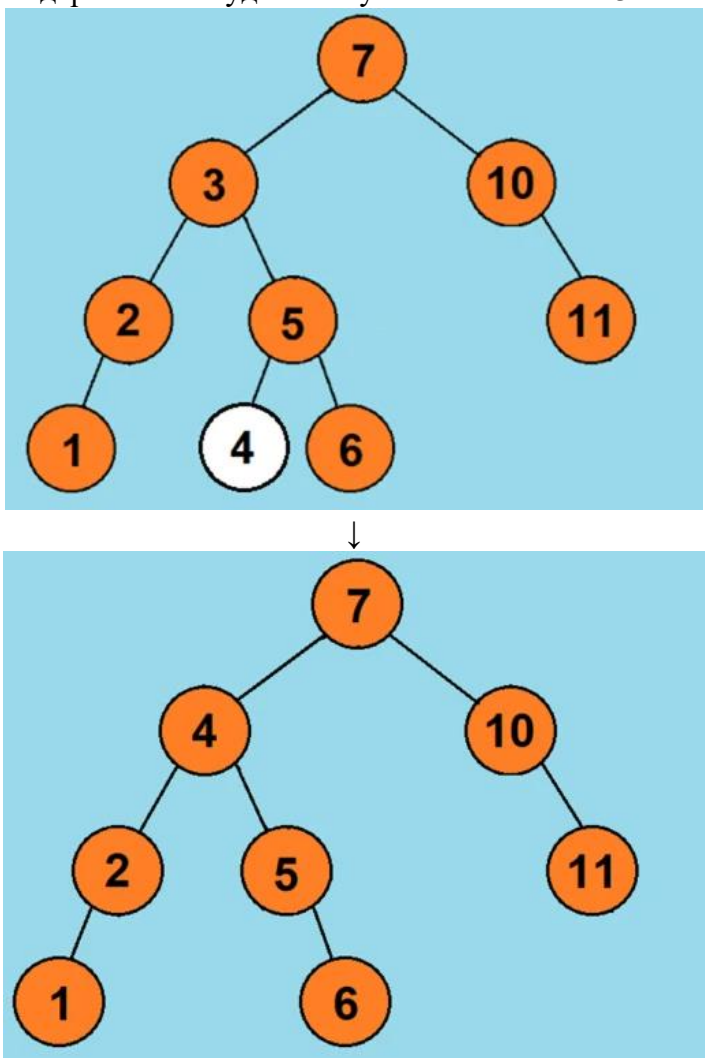
Пример: в рассматриваемом выше дереве, какой узел должен быть левым потомком узла 3?

Если немного задуматься, то станет очевидно, что это должен быть узел со значением 4. Ну а если дерево не будет таким простым? Если оно будет вмещать сотни значений, будет ли так просто понять, кто будет преемником?

Понятное дело, что нет. Поэтому тут нужен свой небольшой алгоритм поиска приемника:

a. Сперва мы должны перейти к правому потомку выбранного узла, значение которого должно быть больше значения узла.
b. После мы переходим к левому потомку правого потомка (если такой существует), дальше — к левому потомку левого потомка и т. д., следуя вниз по цепи левых потомков.
c. Соответственно, последний левый потомок на этом пути и будет являться преемником исходного узла.

Давайте немного обобщим этот небольшой алгоритм: в поддереве правого потомка исходного узла все узлы больше удаляемого, что можно понять из основных правил дерева двоичного поиска. В этом поддереве мы и ищем наименьшее значение. Иными словами, мы ищем наименьший узел в наборе узлов, больших исходного узла. Этот наименьший узел среди больших и будет наиболее подходящим преемником. Как будет выглядеть дерево после удаления узла со значением 3:



А теперь пришло время перейти от практики к теории. Давайте взглянем, каким же образом можно отобразить данную структуру данных в Java. Класс одного узла: **class** Node {

private int value; // ключ узла

```
  private Node leftChild; // Левый узел потомок
  private Node rightChild; // Правый узел потомок

  public void printNode() { // Вывод значения узла в консоль
    System.out.println(" Выбранный узел имеет значение :" + value);
  }

  public int getValue() {
    return this.value;
  }

  public void setValue(final int value) {
    this.value = value;
  }

  public Node getLeftChild() {
    return this.leftChild;
  }

  public void setLeftChild(final Node leftChild) {
    this.leftChild = leftChild;
  }

  public Node getRightChild() {
    return this.rightChild;
  }

  public void setRightChild(final Node rightChild) {
    this.rightChild = rightChild;
  }

  @Override
  public String toString() {
    return "Node{" +
        "value=" + value +
        ", leftChild=" + leftChild +
        ", rightChild=" + rightChild +
        '}';
  }
} Ничего особо сложного: каждый элемент имеет ссылку на левого и правого потомка.
А теперь, пожалуй, самый важный класс — класс дерева: class Tree {
  private Node rootNode; // корневой узел

  public Tree() { // Пустое дерево
    rootNode = null;
  }
```

```
  public Node findNodeByValue(int value) { // поиск узла по значению
    Node currentNode = rootNode; // начинаем поиск с корневого узла
    while (currentNode.getValue() != value) { // поиск покуда не будет найден элемент
или не будут перебраны все
      if (value < currentNode.getValue()) { // движение влево?
        currentNode = currentNode.getLeftChild();
      } else { //движение вправо
        currentNode = currentNode.getRightChild();
      }
      if (currentNode == null) { // если потомка нет,
        return null; // возвращаем null
      }
    }
    return currentNode; // возвращаем найденный элемент
  }

  public void insertNode(int value) { // метод вставки нового элемента
    Node newNode = new Node(); // создание нового узла
    newNode.setValue(value); // вставка данных
    if (rootNode == null) { // если корневой узел не существует
      rootNode = newNode;// то новый элемент и есть корневой узел
    }
    else { // корневой узел занят
      Node currentNode = rootNode; // начинаем с корневого узла
      Node parentNode;
      while (true) // мы имеем внутренний выход из цикла
      {
        parentNode = currentNode;
        if(value == currentNode.getValue()) {   // если такой элемент в дереве уже есть,
не сохраняем его
          return;    // просто выходим из метода
        }
        else  if (value < currentNode.getValue()) {    // движение влево?
          currentNode = currentNode.getLeftChild();
          if (currentNode == null){ // если был достигнут конец цепочки,
            parentNode.setLeftChild(newNode); //  то вставить слева и выйти из методы
            return;
          }
        }
        else { // Или направо?
          currentNode = currentNode.getRightChild();
          if (currentNode == null) { // если был достигнут конец цепочки,
            parentNode.setRightChild(newNode);  //то вставить справа
            return; // и выйти
          }
        }
```

```
        }
    }
}

public boolean deleteNode(int value) // Удаление узла с заданным ключом
{
    Node currentNode = rootNode;
    Node parentNode = rootNode;
    boolean isLeftChild = true;
    while (currentNode.getValue() != value) { // начинаем поиск узла
        parentNode = currentNode;
        if (value < currentNode.getValue()) { // Определяем, нужно ли движение влево?
            isLeftChild = true;
            currentNode = currentNode.getLeftChild();
        }
        else { // или движение вправо?
            isLeftChild = false;
            currentNode = currentNode.getRightChild();
        }
        if (currentNode == null)
            return false; // узел не найден
    }

    if (currentNode.getLeftChild() == null && currentNode.getRightChild() == null) { //
узел просто удаляется, если не имеет потомков
        if (currentNode == rootNode) // если узел - корень, то дерево очищается
            rootNode = null;
        else if (isLeftChild)
            parentNode.setLeftChild(null); // если нет - узел отсоединяется, от родителя
        else
            parentNode.setRightChild(null);
    }
    else if (currentNode.getRightChild() == null) { // узел заменяется левым поддеревом,
если правого потомка нет
        if (currentNode == rootNode)
            rootNode = currentNode.getLeftChild();
        else if (isLeftChild)
            parentNode.setLeftChild(currentNode.getLeftChild());
        else
            parentNode.setRightChild(currentNode.getLeftChild());
    }
    else if (currentNode.getLeftChild() == null) { // узел заменяется правым поддеревом,
если левого потомка нет
        if (currentNode == rootNode)
            rootNode = currentNode.getRightChild();
        else if (isLeftChild)
```

```
            parentNode.setLeftChild(currentNode.getRightChild());
        else
            parentNode.setRightChild(currentNode.getRightChild());
    }
    else { // если есть два потомка, узел заменяется преемником
        Node heir = receiveHeir(currentNode);// поиск преемника для удаляемого узла
        if (currentNode == rootNode)
            rootNode = heir;
        else if (isLeftChild)
            parentNode.setLeftChild(heir);
        else
            parentNode.setRightChild(heir);
    }
    return true; // элемент успешно удалён
}

// метод возвращает узел со следующим значением после передаваемого аргументом.
// для этого он сначала переходим к правому потомку, а затем
// отслеживаем цепочку левых потомков этого узла.
private Node receiveHeir(Node node) {
    Node parentNode = node;
    Node heirNode = node;
    Node currentNode = node.getRightChild(); // Переход к правому потомку
    while (currentNode != null) // Пока остаются левые потомки
    {
        parentNode = heirNode;// потомка задаём как текущий узел
        heirNode = currentNode;
        currentNode = currentNode.getLeftChild(); // переход к левому потомку
    }
    // Если преемник не является
    if (heirNode != node.getRightChild()) // правым потомком,
    { // создать связи между узлами
        parentNode.setLeftChild(heirNode.getRightChild());
        heirNode.setRightChild(node.getRightChild());
    }
    return heirNode;// возвращаем приемника
}

public void printTree() { // метод для вывода дерева в консоль
    Stack globalStack = new Stack(); // общий стек для значений дерева
    globalStack.push(rootNode);
    int gaps = 32; // начальное значение расстояния между элементами
    boolean isRowEmpty = false;
    String separator = "-----------------------------------------------------------------";
    System.out.println(separator);// черта для указания начала нового дерева
    while (isRowEmpty == false) {
```

```
        Stack localStack = new Stack(); // локальный стек для задания потомков элемента
        isRowEmpty = true;

        for (int j = 0; j < gaps; j++)
           System.out.print(' ');
        while (globalStack.isEmpty() == false) { // покуда в общем стеке есть элементы
           Node temp = (Node) globalStack.pop(); // берем следующий, при этом удаляя его
из стека
           if (temp != null) {
              System.out.print(temp.getValue()); // выводим его значение в консоли
              localStack.push(temp.getLeftChild()); // соохраняем в локальный стек,
наследники текущего элемента
              localStack.push(temp.getRightChild());
              if (temp.getLeftChild() != null ||
                    temp.getRightChild() != null)
                 isRowEmpty = false;
           }
           else {
              System.out.print("__");// - если элемент пустой
              localStack.push(null);
              localStack.push(null);
           }
           for (int j = 0; j < gaps * 2 - 2; j++)
              System.out.print(' ');
        }
        System.out.println();
        gaps /= 2;// при переходе на следующий уровень расстояние между элементами
каждый раз уменьшается
        while (localStack.isEmpty() == false)
           globalStack.push(localStack.pop()); // перемещаем все элементы из локального
стека в глобальный
     }
     System.out.println(separator);// подводим черту
  }
}
```

Опять же, ничего особо сложного. Присутствуют операции для дерева двоичного поиска, описанные ранее, плюс метод для отображения дерева в консоли. Ну а теперь взглянем на дерево в действии:

```
public class Application {
  public static void main(String[] args) {
     Tree tree = new Tree();
     // вставляем узлы в дерево:
     tree.insertNode(6);
     tree.insertNode(8);
     tree.insertNode(5);
     tree.insertNode(8);
     tree.insertNode(2);
```

```
        tree.insertNode(9);
        tree.insertNode(7);
        tree.insertNode(4);
        tree.insertNode(10);
        tree.insertNode(3);
        tree.insertNode(1);
        // отображение дерева:
        tree.printTree();

        // удаляем один узел и выводим оставшееся дерево в консоли
        tree.deleteNode(5);
        tree.printTree();

        // находим узел по значению и выводим его в консоли
        Node foundNode = tree.findNodeByValue(7);
        foundNode.printNode();
    }
```
} Операции поиска/вставки/удаления в дереве двоичного поиска имеют временную сложность **O(log(N))**. Но это в лучшем случае. Вообще, временная сложность операций варьируется от **O(log(N))** до **O(N)**. Это зависит от степени вырожденности дерева.