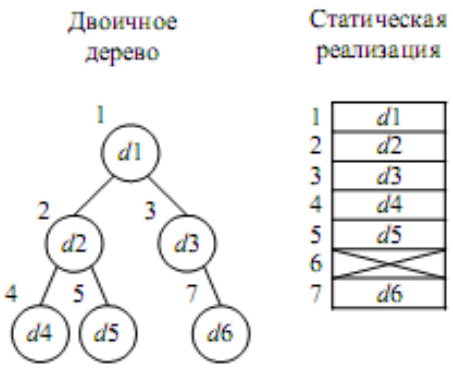
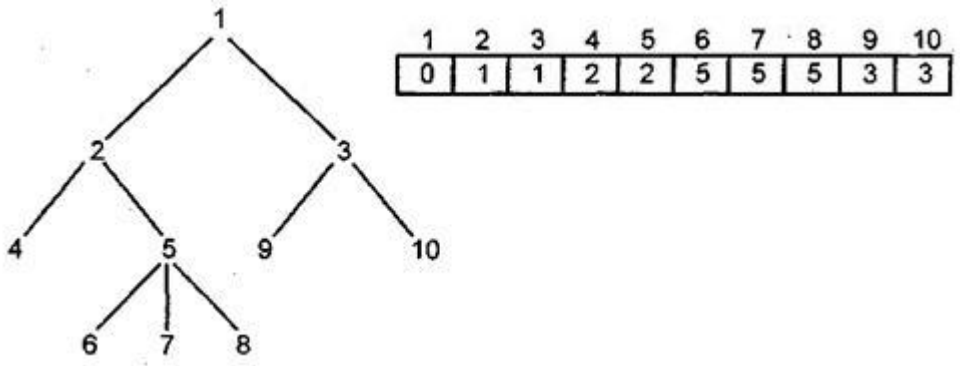
Деревья как структуры данных. Способы реализации

Общие принципы реализации деревьев



Реализация на массивах или списках

Структуру дерева задает массив, хранящий индексы родителей, данные (метки узлов) могут храниться в другом отдельном массиве



Пары parent-child – ребра дерева

1	2
1	3
2	4
2	5
3	9
3	10
5	6
5	7
5	8

Списки parent-children – списки дочерних

1	2	3	
2	4	5	
3	9	10	
4			
5	6	7	8
6			
7			
8			
9			
10			

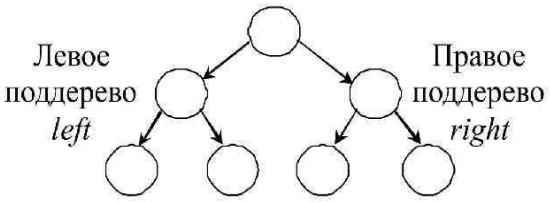
Преимущества – экономия памяти.
Недостатки – потеря производительности.

Динамическая реализация

Класс для отдельного узла дерева

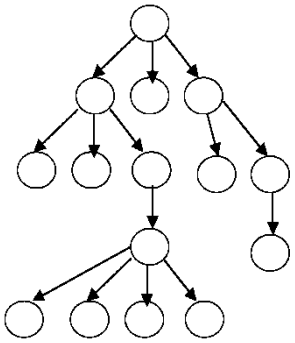
а) для двоичного дерева

```
class BinaryTreeNode<T> {
    private T data;              // данные узла
    private BinaryTreeNode<T> parent;  // родительский узел, необязательно
    private BinaryTreeNode<T> left;    // левый потомок
    private BinaryTreeNode<T> right;   // правый потомок

    public BinaryTreeNode(T data) {
        this.data = data;
        this.parent = null;
        this.left = null;
        this.right = null;
    }
// конструкторы, геттеры, сеттеры, ...
}
```



б) для дерева с произвольным количеством дочерних узлов

```
public class TreeNode<T> {

    private T data;                      // данные узла

    private TreeNode<T> parent;          // родительский узел, во многоих
                                         // задачах необязательный элемент

    private List<TreeNode<T>> children;  // список дочерних узлов

    public TreeNode(T data) {
        this.data = data;
        this.parent = null;
        this.children = new ArrayList<>();
        // или   this.children = new LinkedList<>();    ...
    }
// ...
    public TreeNode<T> add(T value) {
        TreeNode<T> newNode = new TreeNode<>(value);
        newNode.parent = this;
        this.children.add(newNode);
        return newNode;
    }
// конструкторы, геттеры, сеттеры, ...
}
```



Класс дерева

Его формировать необязательно, т.к. каждый узел можно рассматривать как корень поддерева и строить всю обработку, исходя из узла, но в ряде задач может быть удобно иметь и отдельный класс для дерева

```
class Tree<T> {
    private TreeNode<T> root;   // корень может быть типа TreeNode<T> или BinaryTreeNode<T> ,…
    private int count;  // количество узлов необязательно, но удобно

    public int getCount() {
        return count;
    }
     // специфические методы:
     // добавления, вставка, удаление узлов
     // обход всего дерева
     // проверки на выполнения нек.условий, пустота, сбалансированность, ...
     // чтение, запись, преобразование в др.структуру данных
     // ...
}
```

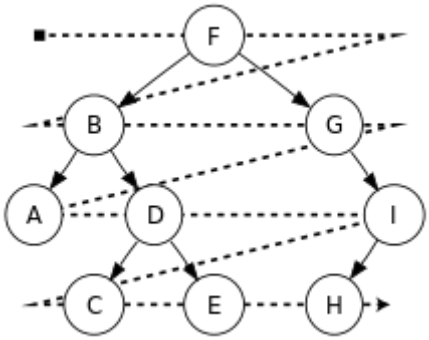
Обход деревьев

Обход деревьев (tree traversals) – последовательная обработка (просмотр, изменение и т.п.) всех узлов дерева, при котором каждый узел обрабатывается строго один раз. При этом получается линейная расстановка узлов дерева.

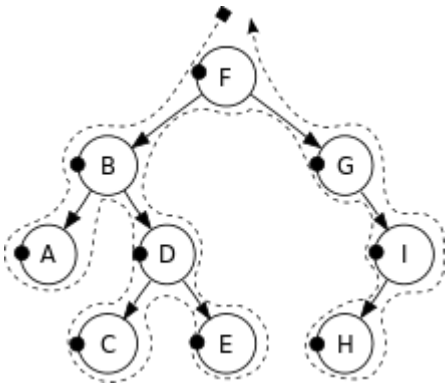
В отличие от линейных структур типа списков и массивов, у которых есть каноничный, прямой способ обхода, деревья можно обходить несколькими способами, в зависимости от поставленной задачи. Начиная с корня, можно применять необходимое действия (именуемое в дальнейшем "визит") как к самому узлу, так и к его левой или правой ветви. Порядок, в котором операции применяются, и будет определять способ обхода.

В зависимости от траекторий выделяют два типа обхода:
— горизонтальный (в ширину) и
— вертикальный (в глубину).

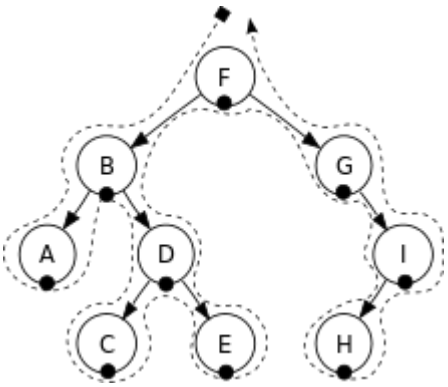
Горизонтальный обход подразумевает обход дерева по уровням (level-ordered) – вначале обрабатываются все узлы текущего уровня, после чего осуществляется переход на нижний уровень.



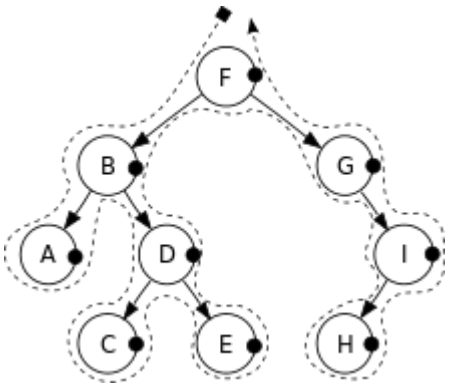
При вертикальном обходе порядок обработки текущего узла и узлов его правого и левого поддеревьев варьирует и по этому признаку выделяют три варианта вертикального обхода:
— прямой (префиксный, pre-ordered): вершина – левое поддерево – правое поддерево;
— обратный (инфиксный, in-ordered): левое поддерево – вершина – правое поддерево; и
— концевой (постфиксный, post-ordered): левое поддерево – правое поддерево – вершина.



Pre-order
F B A D C E G I H



In-order
A B C D E F G H I



Post-order
A C E D B H I G F

Сам обход во всех случаях в принципе один и тот же, различается порядок обработки. Для представления в каком порядке будет проходить обработка узлов дерева удобно следовать по «контуру обхода». При прямом обходе узел будет обработан в точке слева от узла, при обратном снизу от узла и при концевом, соответственно, справа от узла.
Другими словами «находясь» в некотором узле, нам нужно знать, нужно ли его обрабатывать и куда двигаться дальше.

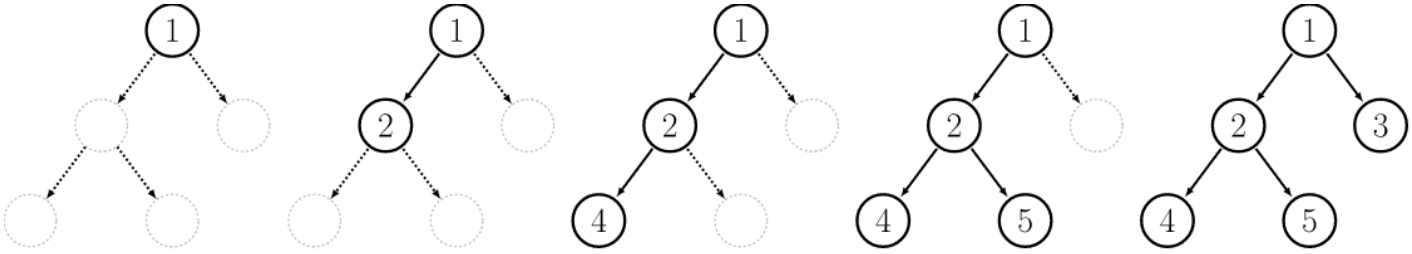
Обход дерева в глубину

Все три варианта вертикального обхода элементарно реализуются рекурсивными функциями.
Допустим, что при обходе нам надо выводить данные узла, это пока и будет целевой операцией.

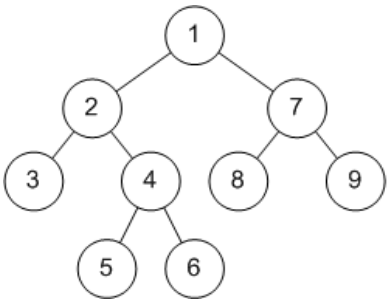
Прямой порядок (pre-order)

- Посетить корень (родителя)
- Обойти левое поддерево
- Обойти правое поддерево

Пример 1. Прямой обход двоичного дерева



Пример 2.



Во вторых примерах этого раздела
числа изображают не ключи узлов, а порядок их обхода

```
public void preOrderPrint (BinaryTreeNode<T> node) {
   // база рекурсии - прерываем процесс, если ссылка пустая
   if (node == null) return;

    // в первую очередь обработать текущий узел
   System.out.print(node.getData() + " ");    // visit(node) – целевая операция вывод или
                                              // другие нужные действия с данными этого узла
   preOrderPrint(node.getLeft()); // затем уйти в левое поддерево

   preOrderPrint(node.getRight()); // затем уйти в правое поддерево
}
```

Вертикальный прямой обход pre-order, итеративно:

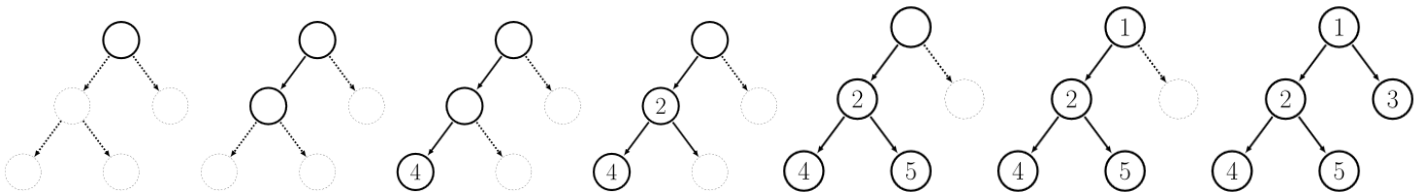
обрабатываем текущий узел, при наличии правого поддерева добавляем его в стек для последующей обработки. Переходим к узлу левого поддерева. Если левого узла нет, переходим к верхнему узлу из стека.

```
public void preOrderTraversalPrint_Iter (BinaryTreeNode<T> node) {
  if (node == null) return;

  Stack<BinaryTreeNode<T>> stack = new Stack<>();
  stack.push(node);

  while (!stack.isEmpty()) {
      // в первую очередь обработать текущий узел
     node = stack.pop();                        // достать его из стека
     System.out.print(node.getData() + " ");    //- visit(node);
     if (node.getRight()!=null)                 // запланировать обработку правого поддерева (позже)
       stack.push(node.getRight());             // поместить в стек его корень

     if (node.getLeft()!=null)                  // запланировать обработку левого поддерева (раньше)
       stack.push(node.getLeft());              // поместить в стек его корень
   }
}
```

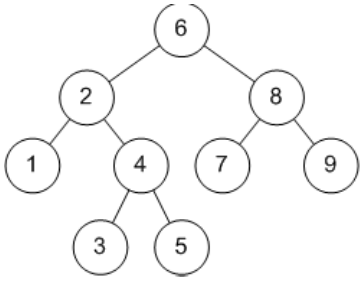
Симметричный или поперечный (in-order)

- Обойти левое поддерево
- Посетить корень (родителя)
- Обойти правое поддерево

Пример 1. Поперечный обход двоичного дерева



Пример 2. Поперечный обход двоичного дерева




```
public void inOrderPrint (BinaryTreeNode<T> node) {

    // база рекурсии - прерываем процесс, если ссылка пустая
    if (node == null) return;

     // в первую очередь попытаться обработать левое поддерево
    inOrderPrint(node.getLeft());

    System.out.print(node.getData() + " "); // затем  текущий узел - visit(node);

    inOrderPrint(node.getRight()); // затем уйти в правое поддерево
}
```

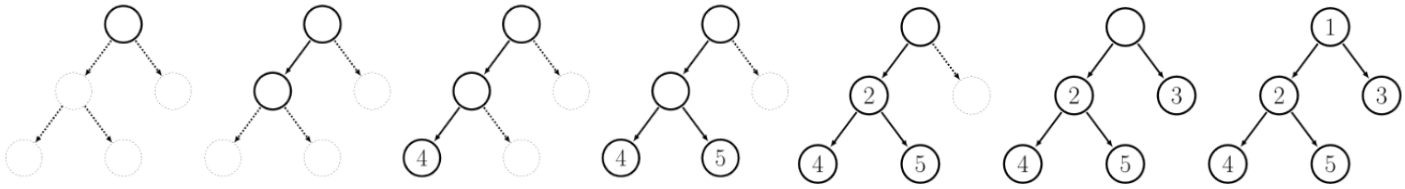
Вертикальный поперечный (симметричный, центральный) обход in-order, итеративно:
из текущего узла «спускаемся» до самого нижнего левого узла, добавляя в стек все посещенные узлы. Обрабатываем верхний узел из стека. Если в текущем узле имеется правое поддерево, начинаем следующую итерацию с правого узла. Если правого узла нет, пропускаем шаг со спуском и переходим к обработке следующего узла из стека.

```
public void inOrderTraversalPrint_Iter (BinaryTreeNode<T> node) {
    if (node == null) return;

    Stack<BinaryTreeNode<T>> stack = new Stack<>();
    //   stack.push(node);

   while (!stack.isEmpty() || node != null) {
        //  текущий узел обрабатывается не сразу, сначала он помещатся в стек
      if (node != null) {
            stack.push(node);
            node = node.getLeft(); // и рассматривается его левое поддерево
      }
        else {                    // в поддереве левая часть просмотрена
          node = stack.pop();                    // достать из стека "центральный" элемент
          System.out.print(node.getData() + " ");    // обработать - visit(node);
          node = node.getRight();                    // перейти к правому поддереву
      }
    }
}
```

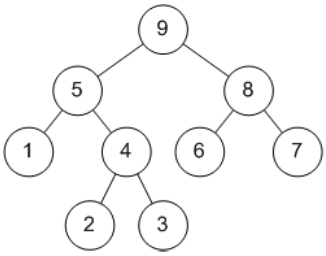
Обратный порядок (post-order)

- Обойти левое поддерево
- Обойти правое поддерево
- Посетить корень (родителя)

Пример 1. Обратный обход двоичного дерева



Пример 2. Обратный обход двоичного дерева




```
public void postOrderPrint (BinaryTreeNode<T> node) {

    // база рекурсии - прерываем процесс, если ссылка пустая
   if (node == null) return;
    // в первую очередь попытаться обработать левое поддерево
   postOrderPrint(node.getLeft());
   postOrderPrint(node.getRight()); // затем уйти в правое поддерево
   System.out.print(node.getData() + " "); // затем  текущий узел - visit(node);
}
```

Вертикальный обратный (концевой) обход post-order, итеративно:

Помимо порядка спуска здесь нужно знать обработано ли уже правое поддерево, только тогда можно обрабатывать центральный родительский узел.

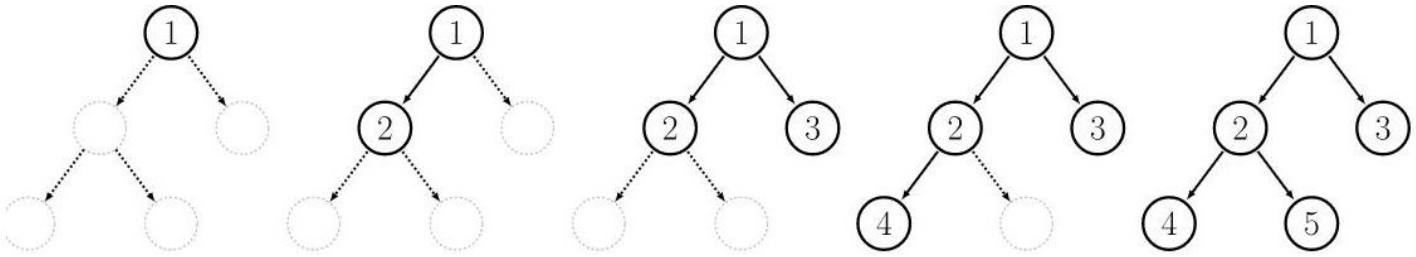
```
public void postOrderTraversalPrint_Iter (BinaryTreeNode<T> node) {
   if (node == null) return;
   Stack<BinaryTreeNode<T>> stack = new Stack<>();
   BinaryTreeNode<T> lastVisit = null;
   while (!stack.isEmpty() || node != null) {
       //  текущий узел обрабатывается не сразу, сначала он помещатся в стек
      if (node != null) {
           stack.push(node);
           node = node.getLeft();      // и рассматривается его левое поддерево
      }
        else { // в поддереве левая часть просмотрена
           // теперь надо достать правый элемент, потом центральный
           BinaryTreeNode<T> pn = stack.peek();
           //если пришли из левого потомка и если правый потомок существует
           if (pn.getRight()!=null && lastVisit != pn.getRight()) {
           node = pn.getRight();                    // перейти к рассмотрению правого поддерева
        }
           else {  // нет ничего правее или они уже обработаны
          System.out.print(pn.getData() + " "); //обработать узел с верхушки стека visit(node)
          lastVisit = stack.pop();                // сохранить его как последний обработанный
        }
      }
    }
  }
}
```

Замечание. Итеративные реализации обходов с использованием контейнеров
Наиболее простыми и понятными считаются рекурсивные алгоритмы. При сведении к итеративному алгоритму часть узлов придётся "откладывать" для дальнейшей обработки.
В случае использования итераций необходимо хранить сведенья о посещенных, но не обработанных узлах. Используются контейнеры типа стек (для вертикального обхода) и очередь (для горизонтального обхода).
Итеративная реализация обхода в глубину требуют использования стека, он нужен для того, чтобы "откладывать" на потом обработку некоторых узлов.

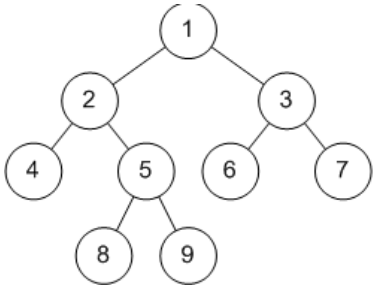
Горизонтальный обход в ширину - Level-order, итеративная реализация на очереди

Обход в ширину подразумевает, что
- сначала обрабатывается корень,
- затем, слева направо, все ветви первого уровня,
- затем все ветви второго уровня
и т.д.

Пример 1. Обход двоичного дерева в ширину



Пример 2. Обход двоичного дерева в ширину




```
public void inLevelTraversalPrint(BinaryTreeNode<T> node) {

    Queue<BinaryTreeNode<T>> queue = new ArrayDeque<>();
    queue.add(node);

    while (!queue.isEmpty()){
        // в первую очередь обработать текущий узел
       node = queue.poll();              // достать его из очереди
       System.out.print(node.getData() + " ");    //  - visit(node);

       if (node.getLeft()!=null)            // запланировать обработку левого поддерева (раньше)
          queue.add(node.getLeft());

         if (node.getRight()!=null)           // запланировать обработку правого поддерева (позже)
           queue.add(node.getRight());
    }
}
```

Пусть мы находимся в корне дерева. Далее необходимо посетить всех наследников корня. Значит, нужно положить в контейнер сначала узел, затем его наследников, при этом узел далее должен быть обработан первым. То есть, элемент, который вошёл первым должен быть обработан первым. Это очередь, и в этом примере мы будем использовать готовую реализацию очереди.

Обрабатываем первый в очереди узел, при наличии дочерних узлов заносим их в конец очереди. Переходим к следующей итерации.

Замечание 2. В приведенных выше реализациях при обходе узлов для каждого из них просто выводились данные на экран.

Чтобы получить более гибкую реализацию, можно в метод обхода дерева передавать функцию, которая могла бы работать с узлом. Используем для этого функциональные интерфейсы, ссылки на методы или лямбда-выражения.

Пример 1. Консьюмер-интерфейс над типом T позволит выполнить какую-либо процедуру (в виде цепочки действий) над данными каждого узла

```
public void preOrder_Func (BinaryTreeNode<T> node, Consumer<T> func) {
    if (node == null) return;

    func.accept(node.getData());    //обработка узла

    preOrder_Func(node.getLeft(), func); //
    preOrder_Func(node.getRight(), func); //
}
```

Вызов для вывода данных узла (то же делали предыдущие реализации)

```
root.preOrder_Func(root, p -> System.out.println(p));
```

Пример 2. Консьюмер-интерфейс над типом узла BinaryTreeNode<T> позволит в том числе и изменять данные каждого узла

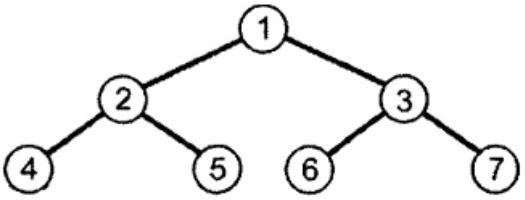
```
public void preOrder_FuncNode (BinaryTreeNode<T> node, Consumer<BinaryTreeNode<T>> func) {
    if (node == null) return;

    func.accept(node);    // обработка узла

    preOrder_FuncNode(node.getLeft(), func); //
    preOrder_FuncNode(node.getRight(), func); //
}
```

Вызов для удвоения числа в данных узла

```
root.preOrder_FuncNode(root, p ->  p.setData(2 * p.getData()) );
```

Примеры вызова (тестирующий метод)

```
  public static void main(String[] args) {

// сформирвоать дерево
        BinaryTreeNode<Integer> root = new BinaryTreeNode<>(1);
        root.setLeft(new BinaryTreeNode<>(2));
        root.setRight(new BinaryTreeNode<>(3));

        root.getLeft().setLeft(new BinaryTreeNode<>(4));
        root.getLeft().setRight(new BinaryTreeNode<>(5));
        root.getRight().setLeft(new BinaryTreeNode<>(6));
        root.getRight().setRight(new BinaryTreeNode<>(7));
```




```
// обойти разными способами и распечать данные
        root.preOrderPrint(root);      System.out.println(); // pre-order 1 2 4 5 3 6 7
        root.inOrderPrint(root);       System.out.println();   // in-order 4 2 5 1 6 3 7
        root.postOrderPrint(root);     System.out.println();   // post-order 4 5 2 6 7 3 1
        root.inLevelPrint(root);       System.out.println();   // level-order 1 2 3 4 5 6 7

// управление целевой операцией обхода
        root.preOrder_Func(root, p -> System.out.println(p));  System.out.println(); // вывод на экран
        root.preOrder_FuncNode(root, p ->  p.setData(2 * p.getData()) ); // удвоение числового поля данных

// итеративные обходы в глубину
        root.preOrderTraversalPrint_Iter(root);         System.out.println();
        root.inOrderTraversalPrint_Iter(root);          System.out.println();
        root.postOrderTraversalPrint_Iter(root);        System.out.println();
   }
```

Обход бесконечных деревьев

Бывают ситуации, когда необходимо обработать бесконечное дерево. Дерево может генерироваться, когда мы обращаемся к нему (например, мы обходим сайт, страницы которого генерируются сервером во время обращения), либо его размер просто не известен (и возможно велик).

Если дерево растёт бесконечно в глубину, то его можно обрабатывать, используя проход в ширину. То есть, известно, что если спускаться вниз по ветви, то до конца мы не дойдём, но на данном уровне дерево имеет конечный размер.

Если дерево растёт бесконечно в ширину, но при этом имеет конечную глубину (то есть, у узла не два наследника, а из бесконечно много), то можно использовать поиск в глубину.

Обработку бесконечного дерева можно заканчивать например, когда обработано достаточно большое количество узлов или их значения достигли какой-то величины.

Дополнительные материалы

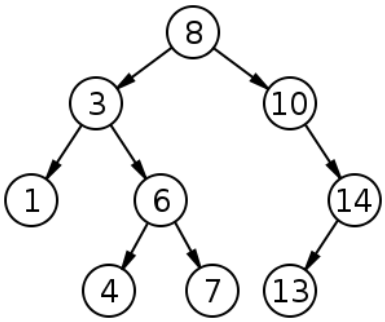
1. Обход дерева https://ru.wikipedia.org/wiki/%D0%9E%D0%B1%D1%85%D0%BE%D0%B4_%D0%B4%D0%B5%D1%80%D0%B5%D0%B2%D0%B0
2. Обход бинарных деревьев: рекурсия, итерации и указатель на родителя https://habr.com/ru/post/144850/

Деревья поиска

Определение

Двоичное дерево поиска (binary search tree, BST) — структура данных для работы с упорядоченными множествами.

Бинарное дерево поиска обладает следующим свойством:
если x — узел бинарного дерева с ключом k, то все узлы в левом поддереве должны иметь ключи, меньшие k, а в правом поддереве большие k.



Отношение БОЛЬШЕ и МЕНЬШЕ – это не обязательно естественная сортировка по величине, это некоторая бинарная операция, которая позволяет разбить элементы на две группы, т.е. можно применять любой подходящий компаратор.

Для наборов данных с повторяющимися ключами обычно добавляют условия что узлы правого поддерева могут совпадать с родительскими ключами.

Общая структура классов (на примере целочисленных элементов данных)

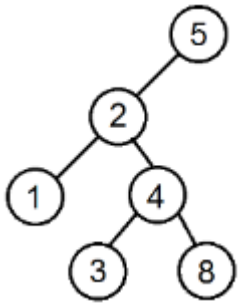
```
class Node {
    private int key;
    private Node left;
    private Node right;
    private Node parent;

    public Node() {
    }

    public Node(int key, Node left, Node right, Node parent) {
        this.key = key;
        this.left = left;
        this.right = right;
        this.parent = parent;
    }
// конструкторы, геттеры, сеттеры, ...
}
```

```
class BinarySearchTree {
    private Node root;
   // private int count; // вершин в дереве; необязательное поле;

// конструкторы, геттеры, сеттеры, ...
// специальные методы для операций дерева, ...
}
```

Основные операции

Обход дерева поиска – совершается как для любых двоичных деревьев

Проверка того, что заданное дерево является деревом поиска



Пример дерева, для которого недостаточно проверки лишь его соседних вершин
Например, в этом дереве вершина с номером 8 находится левее вершины, в которой лежит 5, чего не должно быть в дереве поиска, однако после проверки функция бы вернула true

Для того чтобы решить эту задачу, применим обход в глубину. Запустим от корня рекурсивную логическую функцию, которая выведет true, если дерево является BST и false в противном случае. Чтобы дерево не являлось BST, в нём должна быть хотя бы одна вершина, которая не попадает под определение дерева поиска. То есть достаточно найти всего одну такую вершину, чтобы выйти из рекурсии и вернуть значение false. Если же, дойдя до листьев, функция не встретит на своём пути такие вершины, она вернёт значение true

```
private boolean isBST(Node node, int lo, int hi)
{
    if (node == null) return true;

    if (node.getKey() <= lo || node.getKey() >= hi) return false;

    return isBST(node.getLeft(), lo, node.getKey()) && isBST(node.getRight(), node.getKey(), hi);
}

// метод-обертка для удобства вызова рекурсивного isBST вне класса
public boolean isBST()
{
    return isBST(root, Integer.MIN_VALUE, Integer.MAX_VALUE);
}
```

Функция принимает на вход исследуемую вершину, а также два значения: lo и hi, которые до вызова равнялись соответственно, `Integer.MIN_VALUE, Integer.MAX_VALUE`, т.е. ни один ключ дерева не превосходит их по модулю.

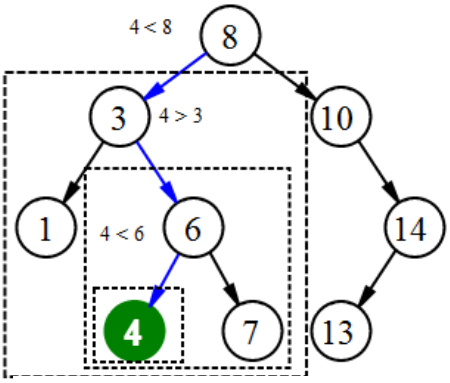
Время работы алгоритма — $O(n)$, где n — количество вершин в дереве.

Поиск элементов (полный обход дерева не нужен)

Поиск нужного узла по значению похож на алгоритм бинарного поиска в отсортированном массиве. Если значения больше узла, то продолжаем поиск в правом поддереве, если меньше, то продолжаем в левом. Если узлов уже нет, то элемент не содержится в дереве.

Для поиска элемента в бинарном дереве поиска можно воспользоваться функцией, которая принимает в качестве параметров корень дерева и искомый ключ.

```
private Node getNodeByKey(Node node, int key) {

    if (node==null || node.getKey()==key) return node;

    if (key < node.getKey())
        return getNodeByKey(node.getLeft(), key );
    else
        return getNodeByKey(node.getRight(), key );
}
// метод-обертка для вызова
public Node getNodeByKey(int key) {
    return getNodeByKey(root, key);
}
```

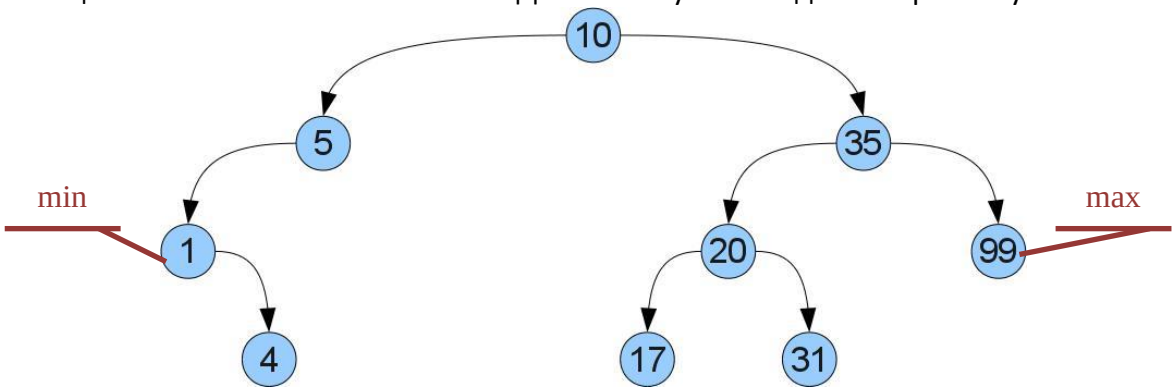


Поиск элемента 4

Для каждого узла функция сравнивает значение его ключа с искомым ключом. Если ключи одинаковы, то функция возвращает текущий узел, в противном случае функция вызывается рекурсивно для левого или правого поддерева. Узлы, которые посещает функция образуют нисходящий путь от корня, так что время ее работы *O*(*h*), где *h* — высота дерева, т.е. O (log n) в среднем и O (n) в худшем случае

Поиск минимума и максимума

Известно, что слева от узла располагается элемент, который меньше чем текущий узел. Из чего следует, что если у узла нет левого наследника, то он является минимумом в дереве. Таким образом, можно найти минимальный элемент дерева, необходимо просто следовать указателям *left* от корня дерева, пока не встретится значение *null*.
Аналогично ищется и максимальный элемент. Для этого нужно следовать правым указателям.




```
public Node getMin(Node node) {
    while(node.getLeft() != null)
        node = node.getLeft();
    return node;
}

public Node getMax(Node node) {
    while(node.getRight() != null)
        node = node.getRight();
    return node;
}
```

Данные функции принимают корень поддерева, и возвращают минимальный (максимальный) элемент в поддереве. Обе процедуры выполняются за время *O*(*h*).

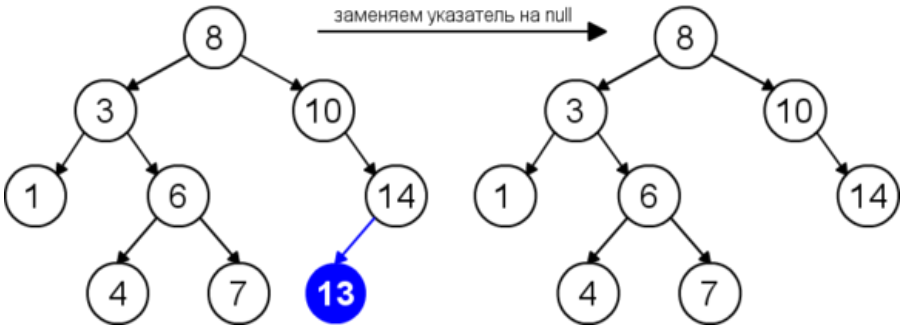
Удаление

Для удаления узла из бинарного дерева поиска нужно рассмотреть три возможные ситуации.

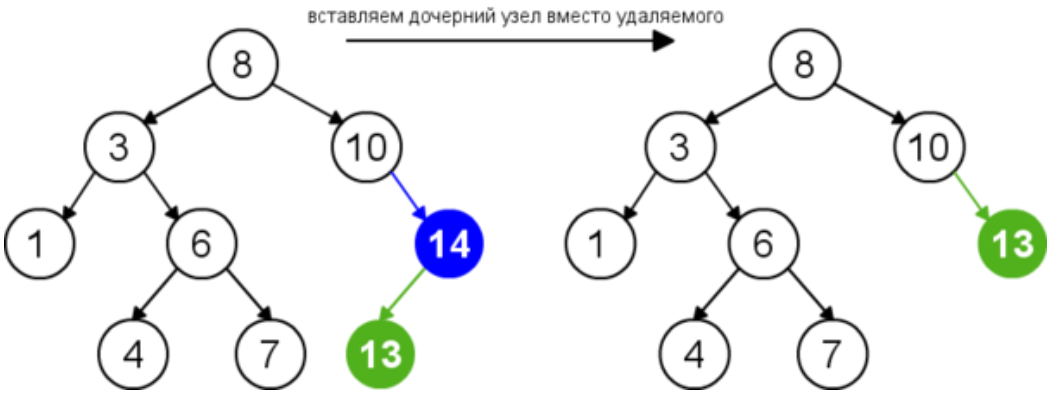
1. Если у узла нет дочерних узлов, то у его родителя нужно просто заменить указатель на *null*.
2. Если у узла есть только один дочерний узел, то нужно создать новую связь между родителем удаляемого узла и его дочерним узлом.
3. Если у узла два дочерних узла, то надо найти следующий за ним элемент (у этого элемента не будет левого потомка), его правого потомка подвесить на место найденного элемента, а удаляемый узел заменить найденным узлом.

Так свойство бинарного дерева поиска не будет нарушено.
Данная реализация удаления не увеличивает высоту дерева. Время работы алгоритма — $O(h)$

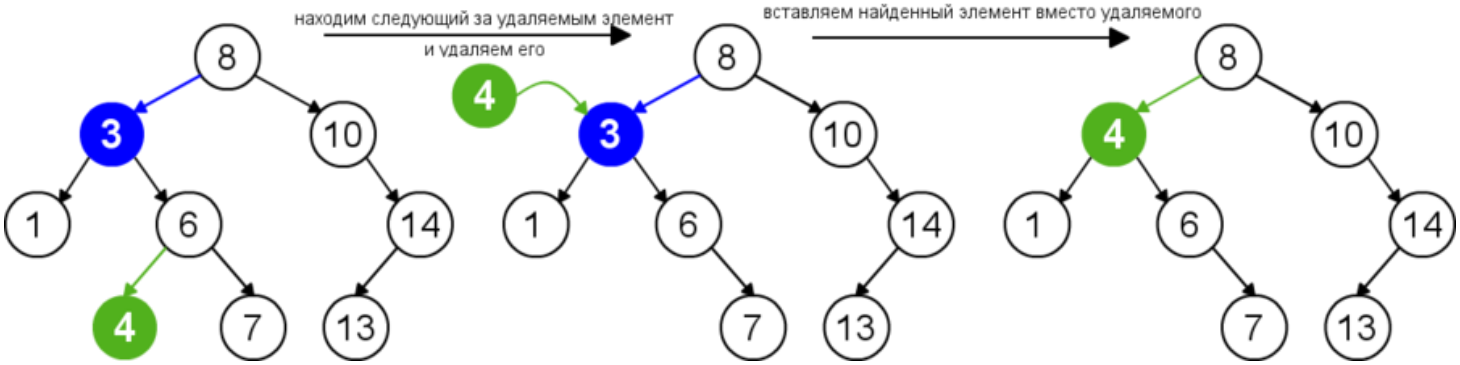
Удаление листа



Удаление узла с одним дочерним узлом



Удаление узла с двумя дочерними узлами



```
// вспомогательный метод для замещения узла a узлом b

    private void replace(Node a, Node b) {
        if (a.getParent() == null)   // a - корень
            root = b;
        else if (a == a.getParent().getLeft()) //если a - левый сын
            a.getParent().setLeft( b );   // b уходит влево от родителя a
        else
            a.getParent().setRight( b );// b уходит вправо
        if (b != null)
            b.setParent(a.getParent()); // сменить родителя у b
    }

// основной метод удаления узла с ключом key

private void remove(Node node, int key) {
     if (node == null) return;
     // поиск узла с ключом key
     if (key < node.getKey())
         remove(node.getLeft(), key);
     else if (key > node.getKey())
         remove(node.getRight(), key);
     else // узел найден

        //третий случай: у узла два потомка
         if (node.getLeft() != null && node.getRight() != null) {
             // найти в правом поддереве минимум и заменить им узел
         Node m = node.getRight();
         while (m.getLeft() != null)
             m = m.getLeft();
         node.setKey(m.getKey());;
         replace(m, m.getRight());
     } else
             // второй случай: у узла один потомок
       if (node.getLeft() != null) {
         replace(node, node.getLeft());    // замещаем левым поддеревом
     } else if (node.getRight() != null) { // или
         replace(node, node.getRight()); // замещаем правым поддеревом
     } else {
       // первый случай: узел - лист
         replace(node, null);
     }
 }

// метод обертка для вызова

    public void remove(int key) {
        remove(root, key);
    }
```

Вставка нового узла с данными

Операция вставки работает аналогично поиску элемента, только при обнаружении у элемента отсутствия ребенка нужно подвесить на него вставляемый элемент.

Т.е. двигаясь от корня сравниваем вставляемые данные (ключ) со значением текущего узла. Если ключ меньше, то перейти к его левому поддереву, в противном случае – к правому. Повторяем это действие, пока можем идти дальше. Если нет поддерева, к которому мы можем спуститься, то вместо него мы вставляем узел со значением ключа.

```
public Node insert(Node t, Node p, int key) {
    if (t == null)
        return  new Node(key, null, null,  p);
    if (key < t.key)
        t.left = insert(t.left, t, key);
    else
        t.right = insert(t.right, t, key);
    return t;
}
```

Время работы алгоритма — $O(h)$, т.е. O (log n) в среднем и O (n) в худшем случае

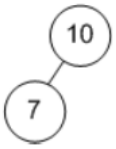
Замечание. Проблема балансировки

Структура дерева зависит от порядка элементов на входе, т.е. окончательная форма дерева зависит от порядка вставки элементов.

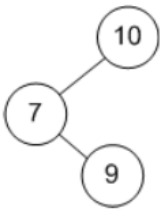
Если элементы не упорядочены и их значения распределены равномерно, то дерево будет достаточно сбалансированным, и путь от вершины до всех листьев будет одинаковый. В таком случае максимальное время доступа до листа равно log(n), где n – это число узлов, то есть равно высоте дерева.

Но это только в самом благоприятном случае. Если же элементы упорядочены, то дерево не будет сбалансировано и растянется в одну сторону, как список; тогда время доступа до последнего узла будет порядка n. Это слабая сторона ДДП, из-за чего применение этой структуры ограничено.

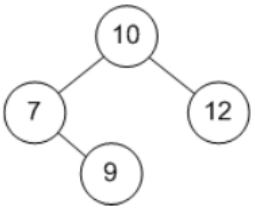
Дерево с одним узлом. Равных NULL потомков не рисуем



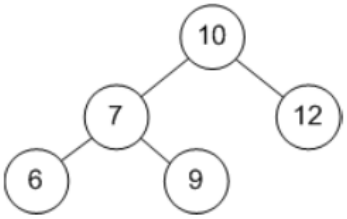
Если значение меньше, то помещаем его слева



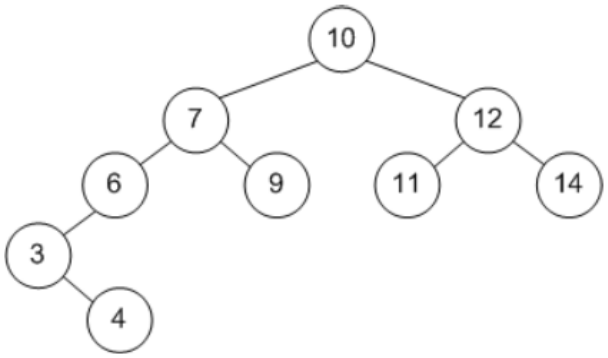
Двоичное дерево поиска после добавления узлов 10, 7, 9



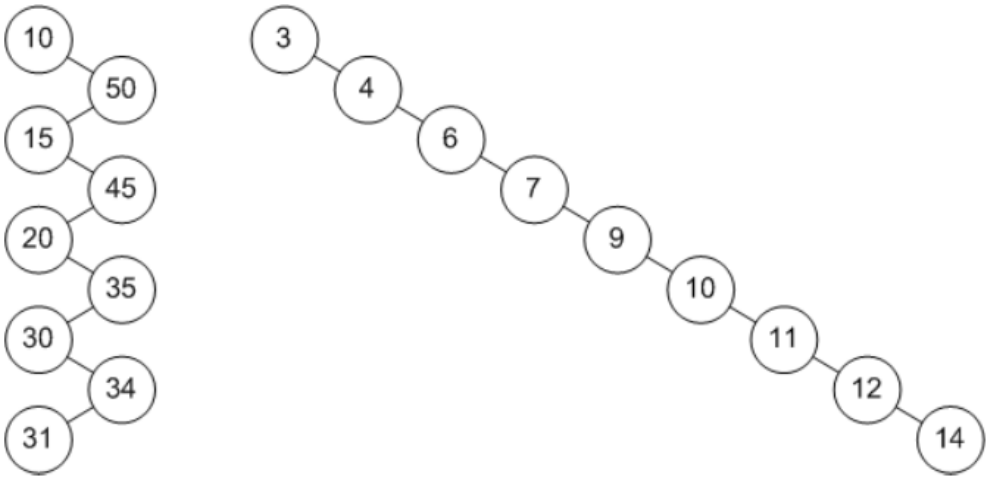
10, 7, 9, 12



10, 7, 9, 12, 6



Добавили последовательно 10, 7, 9, 12, 6, 14, 3, 11, 4



Дерево, которое получили вставкой чередующихся возрастающей и убывающей последовательностей (слева) и полученное при вставке упорядоченной последовательности (справа)

Тестирующий метод

```
public static void main(String[] args) {
    BinarySearchTree tree = new BinarySearchTree();

    tree.setRoot(tree.insert(10));
    tree.insert( 7);
    tree.insert( 9);
    tree.insert( 12);
    tree.insert(6);
    tree.insert( 14);
    tree.insert(3);
    tree.insert( 11);
    tree.insert(4);

    tree.inLevelTraversalPrint(tree.getRoot());
    System.out.println();

    System.out.println(tree.isBST());

    System.out.println(tree.getNodeByKey(12).getKey());

    tree.remove(10);
    tree.inLevelTraversalPrint(tree.getRoot());
}
```