## **[二叉树的Java实现及特点总结](http://www.cnblogs.com/lzq198754/p/5857597.html)**

二叉树是一种非常重要的数据结构，它同时具有数组和链表各自的特点：它可以像数组一样快速查找，也可以像链表一样快速添加。但是他也有自己的缺点：删除操作复杂。

我们先介绍一些关于二叉树的概念名词。

二叉树：是每个结点最多有两个子树的有序树，在使用二叉树的时候，数据并不是随便插入到节点中的，\_x001D\_一个节点的左子节点的关键值必须小于此节点，右子节点的关键值必须大于或者是等于此节点，所以又称二叉查找树、二叉排序树、二叉搜索树。

完全二叉树：若设二叉树的高度为h，除第 h 层外，其它各层 (1～h-1) 的结点数都达到最大个数，第h层有叶子结点，并且叶子结点都是从左到右依次排布，这就是完全二叉树。

满二叉树——除了叶结点外每一个结点都有左右子叶且叶子结点都处在最底层的二叉树。

深度——二叉树的层数，就是深度。

二叉树的特点总结：

大O算法 --> 中型以上

（1）树执行查找、删除、插入的时间复杂度都是O(logN)

（2）遍历二叉树的方法包括前序、中序、后序

（3）非平衡树指的是根的左右两边的子节点的数量不一致

（4） 在非叉树空二中，第i层的结点总数不超过 , i>=1；

（5）深度为h的二叉树最多有个结点(h>=1)，最少有h个结点(平衡)；

（6）对于任意一棵二叉树，如果其叶结点数为N0，而度数为2的结点总数为N2，则N0=N2+1；

二叉树遍历分为三种

先序遍历

首先访问根，再先序遍历左子树，最后先序遍历右子树

中序遍历

首先中序遍历左子树，再访问根，最后中序遍历右子树

后序遍历

首先后序遍历左子树，再后序遍历右子树，最后访问根

二叉树的Java代码实现

首先是树的节点的定义，下面的代码中使用的是最简单的int基本数据类型作为节点的数据，如果要使用节点带有更加复杂的数据类型，换成对应的对象即可。

|  |
| --- |
| /\*\*  \* 树节点  \*  \* **@author** JKMaster  \*  \*/  伪代码...  class TreeNode{  TreeNode leftTreeNode;  .... rightTreeNode;  Int value;  ..... //状态  //get 和 set 方法 用于变量  // 构造  }  **public** **class** TreeNode {  // 左节点  **private** TreeNode lefTreeNode;  // 右节点  **private** TreeNode rightNode;  // 数据  **private** **int** value;  **private** **boolean** isDelete;//节点状态  **public** TreeNode getLefTreeNode() {  **return** lefTreeNode;  }  **public** **void** setLefTreeNode(TreeNode lefTreeNode) {  **this**.lefTreeNode = lefTreeNode;  }  **public** TreeNode getRightNode() {  **return** rightNode;  }  **public** **void** setRightNode(TreeNode rightNode) {  **this**.rightNode = rightNode;  }  **public** **int** getValue() {  **return** value;  }  **public** **void** setValue(**int** value) {  **this**.value = value;  }  **public** **boolean** isDelete() {  **return** isDelete;  }  **public** **void** setDelete(**boolean** isDelete) {  **this**.isDelete = isDelete;  }  **public** TreeNode() {  **super**();  }  **public** TreeNode(**int** value) {  **this**(**null**, **null**, value, **false**);  }  **public** TreeNode(TreeNode lefTreeNode, TreeNode rightNode, **int** value, **boolean** isDelete) {  **super**();  **this**.lefTreeNode = lefTreeNode;  **this**.rightNode = rightNode;  **this**.value = value;  **this**.isDelete = isDelete;  }  @Override  **public** String toString() {  **return** "TreeNode [lefTreeNode=" + lefTreeNode + ", rightNode=" + rightNode + ", value=" + value + ", isDelete="  + isDelete + "]";  }  } |

|  |
| --- |
| /\*\*  \* 二叉树定义  \*  \* **@author** JKMaster  \*  \*/  **public** **class** BinaryTree {  // 根节点  **private** TreeNode root;  **public** TreeNode getRoot() {  **return** root;  }  /\*\*  \* 插入操作  \*  \* **@param** value  \*/  **public** **void** insert(**int** value) {  TreeNode newNode = **new** TreeNode(value);  **if** (root == **null**) {  root = newNode;  root.setLefTreeNode(**null**);  root.setRightNode(**null**);  } **else** {  TreeNode currentNode = root;  TreeNode parentNode;  **while** (**true**) {  parentNode = currentNode;  // 往右放  **if** (newNode.getValue() > currentNode.getValue()) {  currentNode = currentNode.getRightNode();  **if** (currentNode == **null**) {  parentNode.setRightNode(newNode);  **return**;  }  } **else** {  // 往左放  currentNode = currentNode.getLefTreeNode();  **if** (currentNode == **null**) {  parentNode.setLefTreeNode(newNode);  **return**;  }  }  }  }  }  /\*\*  \* 查找  \*  \* **@param** key  \* **@return**  \*/  **public** TreeNode find(**int** key) {  TreeNode currentNode = root;  **if** (currentNode != **null**) {  **while** (currentNode.getValue() != key) {  **if** (currentNode.getValue() > key) {  currentNode = currentNode.getLefTreeNode();  } **else** {  currentNode = currentNode.getRightNode();  }  **if** (currentNode == **null**) {  **return** **null**;  }  }  **if** (currentNode.isDelete()) {  **return** **null**;  } **else** {  **return** currentNode;  }  } **else** {  **return** **null**;  }  }  /\*\*  \* 中序遍历 (主要的)  \*  \* **@param** treeNode  \*/  **public** **void** inOrder(TreeNode treeNode) {  **if** (treeNode != **null** && treeNode.isDelete() == **false**) {  inOrder(treeNode.getLefTreeNode());  System.***out***.println("--" + treeNode.getValue());  inOrder(treeNode.getRightNode());  }  }    /\*\*  \* 先序遍历 (次要)  \*  \* 这三种不同的遍历结构都是一样的，只是先后顺序不一样而已  \*  \* **@param** node  \* 遍历的节点  \*/  **public** **static** **void** preOrderTraverse(TreeNode treeNode) {  **if** (treeNode != **null** ) {  System.***out***.println("--" + treeNode.getValue());  *preOrderTraverse*(treeNode.getLefTreeNode());  *preOrderTraverse*(treeNode.getRightNode());  }  }      /\*\*  \* 后序遍历 (次要)  \*  \* 这三种不同的遍历结构都是一样的，只是先后顺序不一样而已  \*  \* **@param** node  \* 遍历的节点  \*/  **public** **static** **void** postOrderTraverse(TreeNode treeNode) {  **if** (treeNode != **null** ) {  *postOrderTraverse*(treeNode.getRightNode());  System.***out***.println("--" + treeNode.getValue());  *postOrderTraverse*(treeNode.getLefTreeNode());      }  }  } |

|  |
| --- |
| /\*\*  \* 测试  \* **@author** JKMaster  \*  \*/  **public** **class** TestBinaryTree {  **public** **static** **void** main(String[] args) {  BinaryTree tree = **new** BinaryTree();  // 添加数据测试  tree.insert(10);  tree.insert(40);  tree.insert(20);  tree.insert(3);  tree.insert(49);  tree.insert(13);  tree.insert(123);  System.***out***.println("root=" + tree.getRoot().getValue());  // 排序测试  tree.inOrder(tree.getRoot());  // 查找测试  **if** (tree.find(10) != **null**) {  System.***out***.println("找到了");  } **else** {  System.***out***.println("没找到");  }  // 删除测试  tree.find(40).setDelete(**true**);  **if** (tree.find(40) != **null**) {  System.***out***.println("找到了");  } **else** {  System.***out***.println("没找到");  }  }  } |