二叉树的遍历根据输出根节点的顺序可以分别为先序遍历、中序遍历和后序遍历。每种遍历顺序都可分为递归实现和非递归实现。

总结：三种遍历方式的递归实现类似，三种遍历方式的非递归实现类似，只不过是输出节点的次序不同。先序遍历是遇到节点在其加入到stack中就加入到打印列表中，中序遍历是每次先把左子树的节点都加入到stack中，然后在弹出栈顶元素的时候把其加入到打印列表中，后序遍历则是先把左子树的所有节点加入到stack中，针对读到(只读不弹出)栈顶元素的操作则看其右子树是否为空或者右孩子已经加入到打印列表中则把栈顶元素弹出stack，并把其加入到打印列表中，如果不满足有孩子为空或者右孩子已经加入到打印列表中的条件，则把其右节点加入到stack中，继续循环实现。

遍历二叉树的算法中的基本操作是访问结点，则不论按哪一种方式进行遍历，对含n个结点的二叉树，其时间复杂度均为O(n)。所需辅助空间为遍历过程中栈的最大容量，即树的深度，最坏情况下为n，则空间复杂度为O(n)。(具体见数据结构课本P131)

相关题目在《剑指offer》中面试题6,18,23,4,25,27,50等。

1. 二叉树的先序遍历的递归实现

https://leetcode.com/problems/binary-tree-preorder-traversal/

先序递归实现则是遇到根节点则输出，然后分别遍历其左子树和右子树。

|  |
| --- |
| public List<Integer> preorderTraversal(TreeNode root) {  List<Integer> result = new ArrayList<Integer>();  if(root == null){  return result;  }else{  /\*遇到根节点则加入到结果链表中\*/  result.add(root.val);  /\*遍历左子树\*/  List<Integer> left = preorderTraversal(root.left);  result.addAll(left);  /\*遍历左子树\*/  List<Integer> right = preorderTraversal(root.right);  result.addAll(right);  return result;  }  } |

1. 二叉树的先序遍历的非递归实现

利用栈来存放二叉树中的元素。

Step1:首先把根节点放到stack和list中。然后进行如下循环操作，直到stack为空。

Step2:对放到栈中的每一个元素都必须先把其最左边所有分之的节点放到栈中，同时把放到栈中每个元素都加入到list中，作为结果返回；

Step3:然后取出栈顶元素，判断其右节点是否为空，不为空的情况下把它加入到stack中和list中。

重点就是要用node记录表示最后一次放入到stack中的节点，要一直把node节点的最左分支的所有节点都放到stack中，然后弹出栈顶元素用popnode来表示，再判断popnode的右节点是否为空，不为空则把其放入到stack中，要更新node节点为popnode的右节点，如果popnode的右节点为空，则不把其放入到stack中，并且node节点维持之前不变。

|  |
| --- |
| /\* 采用迭代的方式先序遍历  \* 利用栈来存放二叉树中的元素。  \* Step1:首先把根节点放到stack和list中。然后进行如下循环操作，知道stack为空。  \* Step2:对放到栈中的每一个元素都必须先把其最左边所有分之的节点放到栈中，同时把放到栈中每个元素都加入到list中，作为结果返回；  \* Step3:然后取出栈顶元素，判断其右节点是否为空，不为空的情况下把它加入到stack中和list中。  \*/  public List<Integer> preorderTraversal(TreeNode root) {  List<Integer> list = new ArrayList<Integer>();  if(root == null){  return list;  }else{  TreeNode node = root;  Stack stack = new Stack();    stack.push(node);//node表示最后一次放到stack中的元素  list.add(root.val);  while(!stack.isEmpty()){  while(node.left != null){//一直把node节点的所有最左分支都放入的stack中  node = node.left;  list.add(node.val);//把遇到的根节点都加入到list中  stack.push(node);  }    TreeNode popnode= (TreeNode) stack.pop();//弹出栈顶元素    if(popnode.right != null){  node = popnode.right;  stack.push(node);  list.add(node.val);//把遇到的根节点都加入到list中  }  }    }  return list;  } |

|  |
| --- |
| /\* 迭代的思想，参考网上的  \* 与方法二中自己想的一样，保留一个stack和list.  \* 充分利用栈的性质，先进后出。往stack中弹出一个元素之后依次压入其右节点和左节点。这样在弹栈的时候必然是先拿到其左节点的值，将其加入到list中，  \* 然后再依次压入其右节点和左节点。  \*/  public List<Integer> preorderTraversal(TreeNode root) {  List<Integer> list = new ArrayList<Integer>();  if(root == null){  return list;  }else{  TreeNode node = root;  Stack stack = new Stack();  stack.push(node);    while(!stack.isEmpty()){  TreeNode popnode= (TreeNode) stack.pop();//弹出栈顶元素  list.add(popnode.val);  if(popnode.right != null){  stack.push(popnode.right);  }  if(popnode.left != null){  stack.push(popnode.left);  }  }    }  return list;  } |

1. 二叉树的中序遍历的递归实现

https://leetcode.com/problems/binary-tree-inorder-traversal/

|  |
| --- |
| public List<Integer> inorderTraversal(TreeNode root) {  List<Integer> list = new ArrayList<Integer>();  if(root == null){  return list;  }else{  list.addAll(inorderTraversal(root.left));  list.add(root.val);  list.addAll(inorderTraversal(root.right));  }  return list;  } |

1. 二叉树的中序遍历的非递归实现

与二叉树的中序遍历的非递归实现类似，只不过改成每次在弹栈的时候拿元素，这样就保证了先拿到左孩子节点的值，然后再拿根节点，最后拿右子树的值。

|  |
| --- |
| /\*\*  \* 用迭代的思想做二叉树的中序遍历。  \* @param root  \* @return  \*/  public List<Integer> inorderTraversal(TreeNode root) {  List<Integer> list = new ArrayList<Integer>();  if(root == null){  return list;  }else{  TreeNode node = root;  Stack stack = new Stack();  stack.push(node);    while(!stack.isEmpty()){  while(node.left != null){//向左走到尽头，把最左分支的结点都入栈。  node = node.left;  stack.push(node);  }  TreeNode popnode= (TreeNode) stack.pop();//弹出栈顶元素  list.add(popnode.val);  /\*如果弹栈的元素有右孩子，则让其右孩子入栈，进行下一次循环\*/  if(popnode.right != null){  node = popnode.right;  stack.push(node);  }  }  }  return list;  } |

1. 二叉树的后序遍历的递归实现

<https://leetcode.com/problems/binary-tree-postorder-traversal/>

|  |
| --- |
| /\*\*  \* 用递归的方式后续遍历二叉树。  \* @date 20160515  \* @param root  \* @return  \*/  public List<Integer> postorderTraversal(TreeNode root) {  List<Integer> list = new ArrayList<Integer>();  if(root == null){  return list;  }else{  list.addAll(postorderTraversal(root.left));  list.addAll(postorderTraversal(root.right));  list.add(root.val);  }  return list;    } |

1. 二叉树的后序遍历的非递归实现

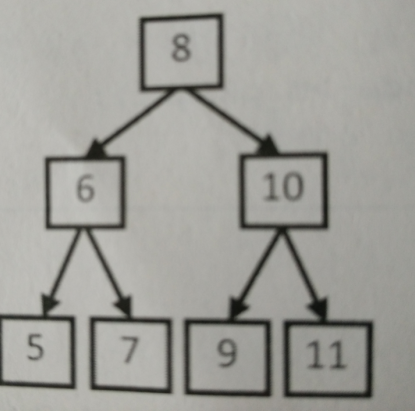
|  |
| --- |
| /\*\*  \* 用迭代的方式后续遍历二叉树。  \* @date 20160515  \* @param root  \* @return  \*/  public List<Integer> postorderTraversal(TreeNode root) {  List<Integer> result = new ArrayList<Integer>();//存目前找到的结果  Set<TreeNode> resultNode = new HashSet<TreeNode>();//存与目前找到的结果对应的结点  if(root == null){  return result;  }  Stack<TreeNode> stack = new Stack<TreeNode>();  /\*先把根结放进去\*/  TreeNode node = root;//node表示要放到stack中的结点。  stack.add(node);  /\*进入循环，循环结束的条件是栈为空\*/  while(!stack.isEmpty()){  while(node.left != null){  node = node.left;  stack.add(node);  }  TreeNode top = stack.peek();//读到栈顶元素  /\*如果栈顶元素没有右孩子或者右孩子已经找到，则可以把栈顶元素取出并放到结果集中\*/  if(top.right == null || resultNode.contains(top.right)){  TreeNode popnode = stack.pop();  result.add(popnode.val);  resultNode.add(popnode);  }else {//如果栈顶有右孩子且该右孩子尚未找到  node = top.right;  stack.add(node);  }    }  return result;    } |

1. 从上往下打印二叉树(按照层次遍历二叉树，打印结果为一行)

题目：从上往下打印出二叉树的每个节点，同层节点从左至右打印。

如下所示的二叉树，按照层次打印之后得到的结果是一个链表，

链表中的元素是8,6,10,5,7,9,11

扩展题目：图的广度优先遍历

分析：按照层次打印二叉树，重点是找到合适的容器队列来存储遍历到的元素。

当队列不为空时，依次把读到的元素的子节点加到队列中，然后在队列中删除该节点。一直重复直到队列为空。初始化时把队头结点（不为null时）放到队列中。

|  |
| --- |
| /\*\*  \* 从上往下打印出二叉树的每个节点，同层节点从左至右打印。  \*/  **public** ArrayList<Integer> PrintFromTopToBottom(TreeNode root) {  ArrayList<Integer> list = **new** ArrayList<Integer>();  Queue<TreeNode> queue = **new** ArrayBlockingQueue<TreeNode>(10);  **if**(root == **null**){  **return** list;  }  queue.add(root);  /\*当队列不为空时，依次把读到的元素的子节点加到队列中，然后在队列中删除该节点。一直重复直到队列为空\*/  **while**(!queue.isEmpty()){  TreeNode node = queue.poll();//获取并删除队列的头结点  list.add(node.val);  **if**(node.left != **null**){  queue.add(node.left);  }  **if**(node.right != **null**){  queue.add(node.right);  }  }  **return** list;  } |

1. 把二叉树打印成多行

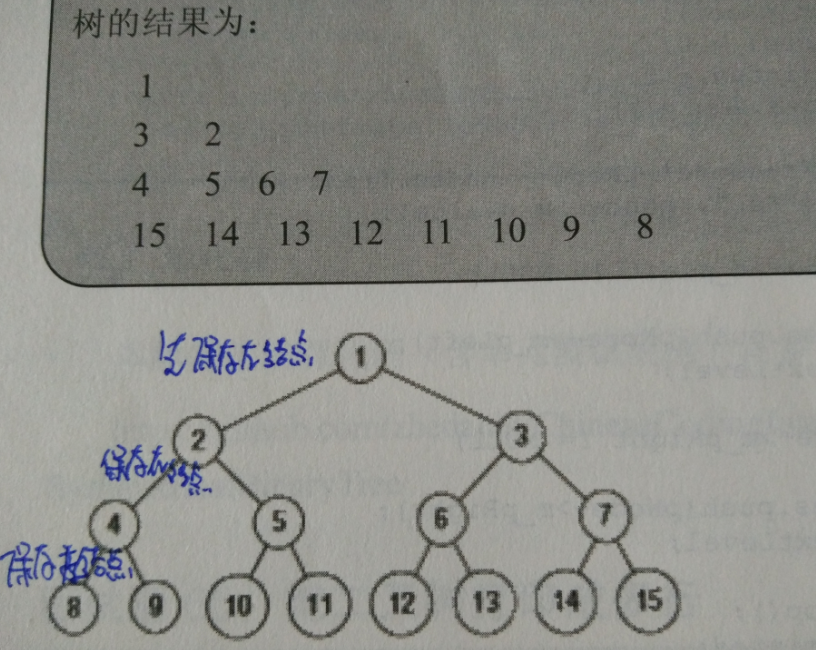
题目：从上到下按层打印二叉树，同一层结点从左至右输出。每一层输出一行。

分析:

|  |
| --- |
| /\*\*  \* 从上到下按层打印二叉树，同一层结点从左至右输出。每一层输出一行。  \* 首先找到当前层的所有结点加入到最后的结果中，然后找到逐个把当前层的元素拿出来并从链表中删除，然后找到这个元素的孩子节点，  \* 将其加入到链表的尾端，这样把当前层结点的所有孩子节点找到之后，链表中就只剩下一层的节点，然后把下一层看做当前层继续循环。  \* 循环结束的条件是链表为空。  \*/    ArrayList<ArrayList<Integer> > Print(TreeNode pRoot) {  /\*新建一个链表作为最后的结果返回\*/  ArrayList<ArrayList<Integer> > result = **new** ArrayList<ArrayList<Integer> >();  **if**(pRoot == **null**){  **return** result;  }  ArrayList<TreeNode> tempList = **new** ArrayList<TreeNode>();//临时存放节点  ArrayList<Integer> valList ;  valList = **new** ArrayList<Integer>();//存放正在寻找的那一层的结点值  tempList.add(pRoot);  valList.add(pRoot.val);  **int** current = 1;//记录当前层还有剩需要找其孩子节点的节点个数。  **int** temp = 0;//记录目前找到的下一层的节点数    **while**(!tempList.isEmpty()){  /\*把目前找到的那一层的结果放到最后的结果中\*/  result.add(valList);  valList = **new** ArrayList<Integer>();  /\*继续通过当前层找下一层\*/  **while**(current!=0){  /\*找到当前层中目前剩余的第一个结点，拿到之后从链表中删除\*/  TreeNode node = tempList.get(0);  tempList.remove(0);  current--;    /\*若其左右节点存在则加入到tempList中\*/  **if**(node.left != **null**){  tempList.add(node.left);  valList.add(node.left.val);  temp++;  }  **if**(node.right != **null**){  tempList.add(node.right);  valList.add(node.right.val);  temp++;  }  }  /\*为下一次寻找做初始化工作\*/  current = temp;  temp = 0;  }    **return** result;  } |

1. 按之字形顺序打印二叉树

题目：请实现一个函数按照之字形打印二叉树，即第一行按照从左到右的顺序打印，第二层按照从右至左的顺序打印，第三行按照从左到右的顺序打印，其他行以此类推。



分析：借助两个栈来实现。stack1保存当前正在打印层的结点，Stack2保存stack1正在打印出的结点的孩子结点。当stack1为奇数层时则先放其左孩子，为偶数层时则先放其右孩子。整个循环结束的条件是stack1为空。

|  |
| --- |
| /\*\*  \* 请实现一个函数按照之字形打印二叉树，即第一行按照从左到右的顺序打印，  \* 第二层按照从右至左的顺序打印，第三行按照从左到右的顺序打印，其他行以此类推。  \* 借助两个栈来实现。stack1保存当前正在打印层的结点，Stack2保存stack2正在打印出的结点的孩子结点。  \* 当stack1为奇数层时则先放其左孩子，为偶数层时则先放其右孩子。  \* 整个循环结束的条件是stack1为空。  \*/  **public** ArrayList<ArrayList<Integer> > Print(TreeNode pRoot) {  ArrayList<ArrayList<Integer> > result = **new** ArrayList<ArrayList<Integer> >();  **if**(pRoot == **null**){  **return** result;  }  /\*stack1保存当前正在打印层的结点，Stack2保存stack2正在打印出的结点的孩子结点。\*/  Stack<TreeNode> stack1 = **new** Stack<TreeNode>();  Stack<TreeNode> stack2 = **new** Stack<TreeNode>();  ArrayList<Integer> list;  TreeNode node ;  **boolean** isEnven = **false**;//表示当前层是否为偶数层  /\*初始化stack1为根结点\*/  stack1.push(pRoot);  /\*循环直到某一层需要打印的结点个数为0\*/  **while**(!stack1.isEmpty()){  list = **new** ArrayList<Integer>();  /\*循环打印出当前层的所有结点\*/  **while**(!stack1.isEmpty()){  node = stack1.pop();  list.add(node.val);  **if**(isEnven){//偶数层则先放它的右孩子  **if**(node.right!=**null**){  stack2.add(node.right);  }  **if**(node.left!=**null**){  stack2.add(node.left);  }  }**else**{//奇数层先放它的左孩子  **if**(node.left!=**null**){  stack2.add(node.left);  }  **if**(node.right!=**null**){  stack2.add(node.right);  }  }  }//结束里层while循环    /\*把当前层的打印结果加到之前的结果中\*/  result.add(list);    stack1.addAll(stack2);  stack2.clear();  /\*偶数层变奇数层，奇数层变偶数层\*/  **if**(isEnven){  isEnven = **false**;  }**else**{  isEnven = **true**;  }  }  **return** result;  } |