## 单链表反转

方法一：逆指针法;使用3个指针遍历单链表，逐个链接点进行反转

p=head;

q = head.next;//初始化

While(q.next){

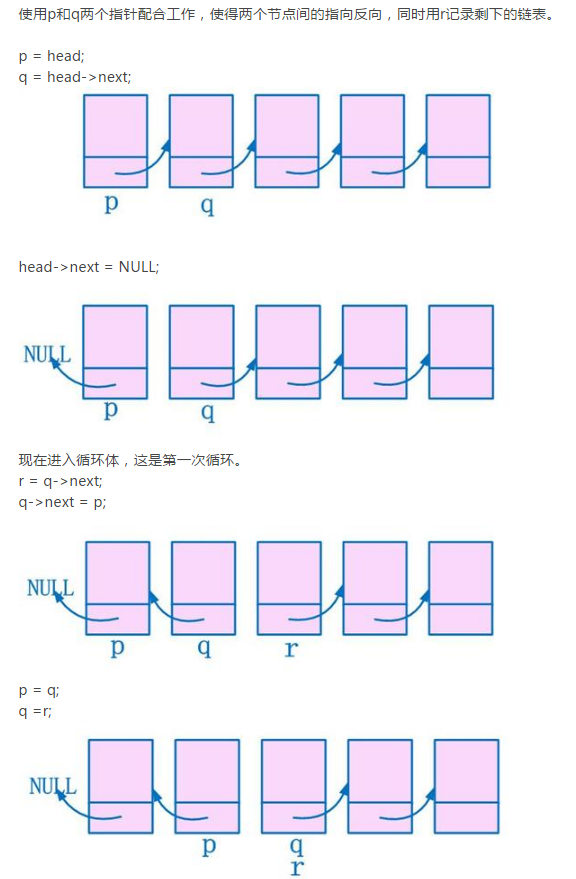
r = q.next;//临时指针r存储着下一个要反转的节点

q.next = p;//完成指向的反转

p = q;//p和q赋值为下次循环要反转指向的两个节点

q = r;

}



方法二：头插法 每次将下一个节点放到第一个位置

p = head;

pre.next = head;

while(p.next){

q = p.next;//每次将q移到第一个位置

p.next = q.next;

q.next = pre.next;//pre.next一直存储着当前链表的第一个节点

pre.next = q;

}

## 二叉树

由数组构建二叉树:如果二叉树的某个节点在数组中的下标为n，那么其左右子节点下表分为为2n+1,2n+2. 所以构建思路为取数组中所有的父节点，完善其子节点信息。而叶子节点的左右子节点都为null。

### 建立二叉树

public class BinTreeTraverse2 {

private int[] array = { 1, 2, 3, 4, 5, 6, 7, 8, 9 };

private static List<Node> nodeList = null;

private static class Node {//内部类：节点

Node left;

Node right;

int data;

Node(int newData) {

left = null;

right = null;

data = newData;

}

}

public void createBinTree() {

nodeList = new LinkedList<Node>();//树的所有节点存在list中

// 将一个数组的值依次转换为Node节点

for (int Index = 0; Index < array.length; Index++) {

nodeList.add(new Node(array[Index]));

}

// 对前lastIndex-1个父节点按照父节点与孩子节点的数字关系建立二叉树

for (int index = 0; index < array.length / 2 - 1; index++) {

// 左孩子

nodeList.get(index).left = nodeList.get(index \* 2 + 1);

// 右孩子

nodeList.get(index).right = nodeList.get(index \* 2 + 2);

}

// 最后一个父节点:因为最后一个父节点可能没有右孩子，所以单独拿出来处理

int lastIndex = array.length / 2 - 1;

// 左孩子

nodeList.get(lastIndex).left = nodeList.get(lastIndex \* 2 + 1);

// 右孩子,如果数组的长度为奇数才建立右孩子

if (array.length % 2 == 1) {

nodeList.get(lastIndex).right = nodeList.get(lastIndex\*2+2);

}

}

### 遍历二叉树

深度优先遍历：先序遍历

先序：先访问父节点然后左右子节点。使用递归方式,中序后序同理。

public static void preOrderTraverse(Node node) {

if (node == null)

return;

System.out.print(node.data + " ");

preOrderTraverse(node.left);

preOrderTraverse(node.right);

}

迭代法深度优先遍历(先序遍历)二叉树:使用栈

(LinkedList为双端队列可以同时实现栈和队列)

public void preOrderTraverse2(Node root) {

Stack<Node> stack = new Stack<>();

stack.push(root);

while(!stack.isEmpty()){

Node node=stack.pop();

System.out.print(node.value+" ");

if(node.right!=null){

stack.push(node.right);

}

if(node.left!=null){

stack.push(node.left);

}

}

}

分层遍历二叉树(广度优先遍历)：使用队列

    public static void levelTraversal(Node root) {

        if (root == null) {

            return;

        }

        LinkedList<Node> queue = new LinkedList<Node>();

        queue.push(root);

    //将根节点压入队列

        while (!queue.isEmpty()) {

            Node cur = queue.removeFirst();

            System.out.print(cur.val + " ");

//左子节点或右子节点不为空，将其压入队列

            if (cur.left != null) {

                queue.add(cur.left);

            }

            if (cur.right != null) {

                queue.add(cur.right);

            }

        }

    }

### 求二叉树节点个数

递归方式，如果二叉树为空，则节点数为0;如果不为空，二叉树节点个数为左节点个数+右节点个数+1。

public int nodeNum(Node root){

if(root == null){

return 0;

}else{

return nodeNum(root.left) + nodeNum(root.right) +1;

}

}

迭代方式：

    public static int getNodeNum(Node root) {

        if(root == null){

            return 0;

        }

        int count = 1;

//LinkedList实现了Queue接口

        Queue<Node> queue = new LinkedList<Node>();

        queue.add(root);

        while(!queue.isEmpty()){

            Node cur = queue.remove();      // 从队头位置移除

            if(cur.left != null){           // 如果有左孩子，加到队尾

                queue.add(cur.left);

                count++;

            }

            if(cur.right != null){      // 如果有右孩子，加到队尾

                queue.add(cur.right);

                count++;

            }

        }

        return count;

    }

### 求二叉树高度

递归：如果二叉树为空，二叉树高度为0；如果不为空二叉树深度=max(左子树深度，右子树深度)+1.

迭代法：利用队列queue

    public static int getDepth(Node root) {

        if(root == null){

            return 0;

        }

        int depth = 0;                          // 深度

        int currentLevelNodes = 1;      // 当前Level node的数量

        int nextLevelNodes = 0;         // 下一层Level node的数量

        LinkedList<Node> queue = new LinkedList<Node>();

        queue.add(root);

/\*\*

\*每遍循环都是在遍历同一层的节点并把其子节点个数增加到下一层节点个数中;当\*currentlevelNodes为0时表明本层节点遍历完毕,将当前节点个数初始化为下一 \*层节点个数开始新循环

\*/

        while( !queue.isEmpty() ){

            Node cur = queue.remove();      // 从队头位置移除

            currentLevelNodes--;            // 减少当前Level node的数量

            if(cur.left != null){               // 如果有左孩子，加到队尾

                queue.add(cur.left);

                nextLevelNodes++;           // 并增加下一层Level node的数量

            }

            if(cur.right != null){          // 如果有右孩子，加到队尾

                queue.add(cur.right);

                nextLevelNodes++;

            }

            if(currentLevelNodes == 0){ // 说明已经遍历完当前层的所有节点

                depth++;                       // 增加高度

                currentLevelNodes = nextLevelNodes;     // 初始化下一层的遍历

                nextLevelNodes = 0;

            }

        }

        return depth;

}

### 求二叉树第K层的节点个数

递归解法： 如果二叉树为空或者k<1返回0；如果二叉树不为空并且k==1，返回1(根节点)

如果二叉树不为空且k>1，返回root左子树中k-1层的节点个数与root右子树k-1层节点个数之和。求以root为根的k层节点数目等价于求以root左孩子为根的k-1层节点数目加上以root右孩子为根的k-1层节点数目。

    public static int getNodeNumKthLevelRec(Node root, int k) {

        if (root == null || k < 1) {

            return 0;

        }

        if (k == 1) {

            return 1;

        }

        int numLeft = getNodeNumKthLevelRec(root.left, k - 1);

// 求root左子树的k-1层节点数

        int numRight = getNodeNumKthLevelRec(root.right, k - 1);

// 求root右子树的k-1层节点数

        return numLeft + numRight;

    }

迭代解法：

    public static int getNodeNumKthLevel(Node root, int k){

        if(root == null){

            return 0;

        }

        Queue<Node> queue = new LinkedList<Node>();

        queue.add(root);

        int i = 1;

        int currentLevelNodes = 1;      // 当前Level，node的数量

        int nextLevelNodes = 0;         // 下一层Level，node的数量

        while( !queue.isEmpty() && i<k){

            Node cur = queue.remove();   // 从队头位置移除

            currentLevelNodes--;         // 减少当前Level node的数量

            if(cur.left != null){        // 如果有左孩子，加到队尾

                queue.add(cur.left);

                nextLevelNodes++;        // 并增加下一层Level node的数量

            }

            if(cur.right != null){      // 如果有右孩子，加到队尾

                queue.add(cur.right);

                nextLevelNodes++;

            }

            if(currentLevelNodes == 0){ // 说明已经遍历完当前层的所有节点

                currentLevelNodes = nextLevelNodes; // 初始化下一层的遍历

                nextLevelNodes = 0;

                i++;

            }

        }

        return currentLevelNodes;

    }

### 判断二叉树是不是完全二叉树

迭代:若设二叉树的深度为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 层所有的结点都连续集中在最左边，这就是完全二叉树。有如下算法，按层次（从上到下，从左到右）遍历二叉树，当遇到一个节点的左子树为空时，则该节点右子树必须为空，且后面遍历的节点左右子树都必须为空，否则不是完全二叉树。

public static boolean isCompleteBinaryTree(Node root){

if(root == null){

return false;

}

Queue<Node> queue = new LinkedList<Node>();

queue.add(root);

boolean mustHaveNoChild = false;

boolean result = true;

while( !queue.isEmpty() ){

Node cur = queue.remove();

if(mustHaveNoChild){//已经出现有空子树的节点则后面出现的必须为叶节点

if(cur.left!=null || cur.right!=null){

result = false;

break;

}

} else {

// 如果左子树和右子树都非空，则继续遍历

if(cur.left!=null && cur.right!=null){

queue.add(cur.left);

queue.add(cur.right);

}

//如果左子树非空但右子树为空,说明已经出现空节点,之后必须都为空子树

else if(cur.left!=null && cur.right==null){

mustHaveNoChild = true;

queue.add(cur.left);

}

//如果左子树为空但右子树非空，说明这棵树不是完全二叉完全树

else if(cur.left==null && cur.right!=null){

result = false;

break;

}else{// 如果左右子树都为空，则后面的必须也都为空子树

mustHaveNoChild = true;

}

}

}

return result;

}

### 求二叉树叶子节点的个数

递归法: 当为叶子节点时返回1,否则为左子树和右子树叶节点之和

迭代法:将树放到队列 左右节点都为空的节点为叶节点

public static int getNodeNumLeaf(Node root) {

if(root == null){

return 0;

}

Queue<Node> queue = new LinkedList<Node>();

queue.add(root);

int leafNodes = 0;

while( !queue.isEmpty() ){

Node cur = queue.remove(); // 从队头位置移除

if(cur.left != null){ // 如果有左孩子，加到队尾

queue.add(cur.left);

}

if(cur.right != null){ // 如果有右孩子，加到队尾

queue.add(cur.right);

}

if(cur.left==null && cur.right==null){ // 叶子节点

leafNodes++;

}

}

return leafNodes;

}

### 判断二叉树是不是平衡二叉树

递归解法

public static boolean isAVLRec(Node root) {

if(root == null){ // 如果二叉树为空，返回真

return true;

}

// 如果左子树和右子树高度相差大于1，则非平衡二叉树

if(Math.abs(getDepthRec(root.left) - getDepthRec(root.right)) > 1){

return false;

}

// 递归判断左子树和右子树是否为平衡二叉树

return isAVLRec(root.left) && isAVLRec(root.right);

}

### 反转二叉树

递归解法：如果二叉树为空，返回空；如果二叉树不为空，反转左右子树，然后交换左子树和右子树

public static Node mirrorRec(Node root) {

if (root == null) {

return null;

}

Node left = mirrorRec(root.left);

Node right = mirrorRec(root.right);

root.left = right;

root.right = left;

return root;

}

### 由前序遍历序列和中序遍历序列重建二叉树

递归解法：前序遍历的第一个节点是根，在中序遍历中找到这个根所在的位置，那么中序遍历中根位置左边的便是整个树的左子树，根右边的是整个树的右子树，然后找到左子树的长度length，前序遍历中根节点之后的length个长度都是左子树的节点，然后对左子树找到根，再在其中序遍历中找到对应根的位置，依次递归即可。

public static TreeNode rebuildBinaryTreeRec(List<Integer> preOrder, List<Integer> inOrder){

TreeNode root = null;

List<Integer> leftPreOrder;

List<Integer> rightPreOrder;

List<Integer> leftInorder;

List<Integer> rightInorder;

int inorderPos;

int preorderPos;

if ((preOrder.size() != 0) && (inOrder.size() != 0)) {

// 把preorder的第一个元素作为root

root = new TreeNode(preOrder.get(0));

//根据root节点位置，把preorder，inorder分别划分为 root左侧和右侧 的两个子区间

inorderPos = inOrder.indexOf(preOrder.get(0));

//inorder序列的分割点 即为前序遍历根节点在终序遍历中的位置

leftInorder = inOrder.subList(0, inorderPos);

rightInorder = inOrder.subList(inorderPos + 1, inOrder.size());

// preorder序列的分割点 左子树长度

preorderPos = leftInorder.size();

leftPreOrder = preOrder.subList(1, preorderPos + 1);

rightPreOrder = preOrder.subList(preorderPos + 1, preOrder.size());

root.left = rebuildBinaryTreeRec(leftPreOrder, leftInorder);

// root的左子树就是preorder和inorder的左侧区间而形成的树

root.right = rebuildBinaryTreeRec(rightPreOrder, rightInorder); // root的右子树就是preorder和inorder的右侧区间而形成的树

}

return root;

}

## 堆栈

### 按升序对栈排序

首先申请一个栈sta来存放数据栈，再申请一个辅助栈help来存放临时数据，然后比较sta弹出的栈顶的值res与help栈顶元素的大小。

当sta栈不为空时：

1、如果help.empty()或者res<=help.peek ()，那么就把res的值压入help栈中；

2、如果help不为空并且res>help.peek ()，那么就把help中栈顶的值弹出并压入sta栈，再继续比较res和help.peek()的值，直到res<=help.peek()时把res的值压入help栈中。

void StackSort(Stack<Integer> sta) {

stack<Integer> help;

while(!sta.empty()){

int res=sta.peek();

sta.pop();

if(help.empty()||res<=help.peek())

help.push(res);

else{

while(!help.empty()&&res>help.peek()){

sta.push(help.peek());

help.pop();

}

help.push(res);

}

}

while(!help.empty()){

sta.push(help.peek());

help.pop();

}

}