Tree

Balanced Binary Tree

Given a binary tree, determine if it is height-balanced.
For this problem, a height-balanced binary tree is defined as a binary tree in which the depth of the two subtrees of every node never differ by more than 1.

Tags: Tree, Depth-first Search

```
/**
 * Definition for binary tree
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public boolean isBalanced(TreeNode root) {
        if (root == null)
            return true;

        if (getHeight(root) == -1)
```

```
            return false;

        return true;
    }

    public int getHeight(TreeNode root) {
        if (root == null)
            return 0;

        int left = getHeight(root.left);
        int right = getHeight(root.right);

        if (left == -1 || right == -1)
            return -1;

        if (Math.abs(left - right) > 1) {
            return -1;
        }

        return Math.max(left, right) + 1;

    }
}
```

Binary Tree Inorder Traversal

Given a binary tree, return the inorder traversal of its nodes' values.

For example:
Given binary tree {1,#,2,3},

```
   1
    \
     2
    /
   3
```

return [1,3,2].

Note: Recursive solution is trivial, could you do it iteratively?

Tags: Tree, Hash Table, Stack

Inorder: left->parent->right

```
/**
 * Definition for binary tree
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public ArrayList<Integer> inorderTraversal(TreeNode root) {
        // IMPORTANT: Please reset any member data you declared, as
```

```
// the same Solution instance will be reused for each test case.
  ArrayList<Integer> lst = new ArrayList<Integer>();

if(root == null)
        return lst;

Stack<TreeNode> stack = new Stack<TreeNode>();
//define a pointer to track nodes
TreeNode p = root;

while(!stack.empty() || p != null){

        // if it is not null, push to stack
        //and go down the tree to left
        if(p != null){
                stack.push(p);
                p = p.left;

        // if no left child
        // pop stack, process the node
        // then let p point to the right
        }else{
```

```
                    TreeNode t = stack.pop();

                    lst.add(t.val);

                    p = t.right;

                }

            }



            return lst;

        }

}
```

Binary Tree Level Order Traversal

Given a binary tree, return the level order traversal of its nodes' values. (ie, from left to right, level by level).

For example:
Given binary tree {3,9,20,#,#,15,7},

```
    3
   / \
  9  20
    /  \
   15   7
```

return its level order traversal as:

```
[
  [3],
  [9,20],
  [15,7]
]
```

confused what "{1,#,2,3}" means? > read more on how binary tree is serialized on OJ.

Tags: Tree, Breadth-first Search

```
/**
```

```
 * Definition for binary tree
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public ArrayList<ArrayList<Integer>> levelOrder(TreeNode root) {
        ArrayList<ArrayList<Integer>> result = new ArrayList<ArrayList<Integer>>();
        //ArrayList<ArrayList<Integer>> res = new ArrayList<ArrayList<Integer>>();
        if(root==null){
            return result;
        }
        Queue<TreeNode> queue = new LinkedList<TreeNode>();
        queue.add(root);

        while(!queue.isEmpty()){
            ArrayList<TreeNode> tempNode = new ArrayList<TreeNode>();
            ArrayList<Integer> tempVal = new ArrayList<Integer>();
```

```
        while(!queue.isEmpty()){
            TreeNode node = queue.remove();
            int val = node.val;
            tempNode.add(node);
            tempVal.add(val);
        }
        result.add(tempVal);

        for(int i=0;i<tempNode.size();i++){
            TreeNode node = tempNode.get(i);
            if(node.left!=null){
                queue.add(node.left);
            }
            if(node.right!=null){
                queue.add(node.right);
            }
        }

    }

    return result;
}
```

}

Binary Tree Level Order Traversal II

Given a binary tree, return the bottom-up level order traversal of its nodes' values. (ie, from left to right, level by level from leaf to root).

For example:
Given binary tree {3,9,20,#,#,15,7},

```
    3
   / \
  9  20
    /  \
   15   7
```

return its bottom-up level order traversal as:

```
[
  [15,7],
  [9,20],
  [3]
]
```

confused what "{1,#,2,3}" means? > read more on how binary tree is serialized on OJ.

Tags: Tree, Breadth-first Search

The same as the former problem, just reverse the result.

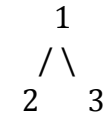
```
ArrayList<ArrayList<Integer>> res = new ArrayList<ArrayList<Integer>>();
……
for(int i=result.size()-1; i>=0; i--){
    res.add(result.get(i));
}

return res;
```

Binary Tree Maximum Path Sum

Given a binary tree, find the maximum path sum.

The path may start and end at any node in the tree.

For example:
Given the below binary tree,



Return 6.

Tags: Tree, Depth-first Search

```
/**
 * Definition for binary tree
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    int max;

    public int maxPathSum(TreeNode root) {
        max = (root == null) ? 0 : root.val;
        findMax(root);
        return max;
    }
```

```
    public int findMax(TreeNode node) {
        if (node == null)
            return 0;

        // recursively get sum of left and right path
        int left = Math.max(findMax(node.left), 0);
        int right = Math.max(findMax(node.right), 0);

        //update maximum here
        max = Math.max(node.val + left + right, max);

        // return sum of largest path of current node
        return node.val + Math.max(left, right);
    }
}
```

Binary Tree Postorder Traversal

Given a binary tree, return the postorder traversal of its nodes' values.

For example:
Given binary tree {1,#,2,3},

```
   1
    \
     2
    /
   3
```

return [3,2,1].

Note: Recursive solution is trivial, could you do it iteratively?

Tags: Tree, Stack

Postorder: left->right->parent

The key to to iterative postorder traversal is the following:

1. The order of "Postorder" is: left child -> right child -> parent node.
2. Find the relation between the previously visited node and the current node
3. Use a stack to track nodes

As we go down the tree, check the previously visited node. If it is the parent of the current node, we should add current node to stack. When there is no children for current node, pop it from stack. Then the previous node become to be under the current node for next loop.

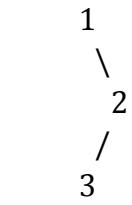
```
//Definition for binary tree
public class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(int x) { val = x; }
}


public class Solution {
    public ArrayList<Integer> postorderTraversal(TreeNode root) {

        ArrayList<Integer> lst = new ArrayList<Integer>();

        if(root == null)
            return lst;

        Stack<TreeNode> stack = new Stack<TreeNode>();
        stack.push(root);

        TreeNode prev = null;
        while(!stack.empty()){
            TreeNode curr = stack.peek();

            // go down the tree.
            //check if current node is leaf, if so, process it and pop stack,
```

```
                //otherwise, keep going down
                if(prev == null || prev.left == curr || prev.right == curr){
                    //prev == null is the situation for the root node
                    if(curr.left != null){
                        stack.push(curr.left);
                    }else if(curr.right != null){
                        stack.push(curr.right);
                    }else{
                        stack.pop();
                        lst.add(curr.val);
                    }

                //go up the tree from left node
                //need to check if there is a right child
                //if yes, push it to stack
                //otherwise, process parent and pop stack
                }else if(curr.left == prev){
                    if(curr.right != null){
                        stack.push(curr.right);
                    }else{
                        stack.pop();
                        lst.add(curr.val);
                    }

                //go up the tree from right node
                //after coming back from right node, process parent node and pop
stack.
                }else if(curr.right == prev){
                    stack.pop();
                    lst.add(curr.val);
                }

                prev = curr;
            }

            return lst;
    }
}
```

Binary Tree Preorder Traversal

Given a binary tree, return the preorder traversal of its nodes' values.

For example:
Given binary tree {1,#,2,3},



return [1,2,3].

Note: Recursive solution is trivial, could you do it iteratively?

Tags: Tree, Stack

Preorder: parent->left->right

Preorder binary tree traversal is a classic interview problem about trees. The key to solve this problem is to understand the following:

1. What is preorder? (parent node is processed before its children)
2. Use Stack from Java Core library

It is not obvious what preorder for some strange cases. However, if you draw a stack and manually execute the program, how each element is pushed and popped is obvious.

The key to solve this problem is using a stack to store left and right children, and push right child first so that it is processed after the left child.

```
public class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;
    TreeNode(int x) { val = x; }
}

public class Solution {
    public ArrayList<Integer> preorderTraversal(TreeNode root) {
        ArrayList<Integer> returnList = new ArrayList<Integer>();

        if(root == null)
            return returnList;

        Stack<TreeNode> stack = new Stack<TreeNode>();
        stack.push(root);
```

```
            while(!stack.empty()){
                TreeNode n = stack.pop();
                returnList.add(n.val);

                if(n.right != null){
                    stack.push(n.right);
                }
                if(n.left != null){
                    stack.push(n.left);
                }

            }
            return returnList;
        }
}
```

Binary Tree Zigzag Level Order Traversal

Given a binary tree, return the zigzag level order traversal of its nodes' values. (ie, from left to right, then right to left for the next level and alternate between).

For example:
Given binary tree {3,9,20,#,#,15,7},

```
    3
   / \
  9  20
    /  \
   15   7
```

return its zigzag level order traversal as:

```
[
  [3],
  [20,9],
  [15,7]
]
```

confused what "{1,#,2,3}" means? > read more on how binary tree is serialized on OJ.

Tags: Tree, Breadth-first Search, Stack

这道题其实还是树的层序遍历 Binary Tree Level Order Traversal，如果不熟悉的朋友可以先看看哈。不过这里稍微做了一点变体，就是在遍历的时候偶数层自左向右，而奇数层自右向左。在 Binary Tree Level Order Traversal 中我们是维护了一个队列来完成遍历，而在这里为了使每次都倒序出来，我们很容易想到用栈

的结构来完成这个操作。有一个区别是这里我们需要一层一层的来处理（原来可以按队列插入就可以，因为后进来的元素不会先处理），所以会同时维护新旧两个栈，一个来读取，一个存储下一层结点。总体来说还是一次遍历完成，所以时间复杂度是 O(n)，空间复杂度最坏是两层的结点，所以数量级还是 O(n)（满二叉树最后一层的结点是 n/2 个）。

```
/**
 * Definition for binary tree
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public ArrayList<ArrayList<Integer>> zigzagLevelOrder(TreeNode root) {
    ArrayList<ArrayList<Integer>> res = new ArrayList<ArrayList<Integer>>();
    if(root==null)
        return res;
    LinkedList<TreeNode> stack = new LinkedList<TreeNode>();
    int level=1;
    ArrayList<Integer> item = new ArrayList<Integer>();
    item.add(root.val);
    res.add(item);
```

```
stack.push(root);
while(!stack.isEmpty())
{
    LinkedList<TreeNode> newStack = new LinkedList<TreeNode>();
    item = new ArrayList<Integer>();
    while(!stack.isEmpty())
    {
        TreeNode node = stack.pop();
        if(level%2==0)
        {
            if(node.left!=null)
            {
                newStack.push(node.left);
                item.add(node.left.val);
            }
            if(node.right!=null)
            {
                newStack.push(node.right);
                item.add(node.right.val);
            }
        }
        else
```

```
                {
                    if(node.right!=null)
                    {
                        newStack.push(node.right);
                        item.add(node.right.val);
                    }
                    if(node.left!=null)
                    {
                        newStack.push(node.left);
                        item.add(node.left.val);
                    }
                }
            }
            level++;
            if(item.size()>0)
                res.add(item);
            stack = newStack;
        }
        return res;
    }
}
```

Construct Binary Tree from Inorder and Postorder Traversal

Given inorder and postorder traversal of a tree, construct the binary tree.

Note:
You may assume that duplicates do not exist in the tree.

Tags: Tree, Array, Depth-first Search

This problem can be illustrated by using a simple example.

Inorder: left->parent->right
Postorder: left->right->parent

in-order: 4 2 5 (1) 6 7 3 8
post-order: 4 5 2 6 7 8 3 (1)

From the post-order array, we know that last element is the root. We can find the root in in-order array. Then we can identify the left and right sub-trees of the root from in-order array.

Using the length of left sub-tree, we can identify left and right sub-trees in post-order array. Recursively, we can build up the tree.

```
//Definition for binary tree
 public class TreeNode {
      int val;
      TreeNode left;
      TreeNode right;
      TreeNode(int x) { val = x; }
 }

public class Solution {
     public TreeNode buildTree(int[] inorder, int[] postorder) {
          int inStart = 0;
          int inEnd = inorder.length-1;
          int postStart =0;
          int postEnd = postorder.length-1;

          return buildTree(inorder, inStart, inEnd, postorder, postStart,
postEnd);
     }
```

```
    public TreeNode buildTree(int[] inorder, int inStart, int inEnd,
                                    int[] postorder, int postStart, int postEnd){
        if(inStart > inEnd || postStart > postEnd)
            return null;

        int rootValue = postorder[postEnd];
        TreeNode root = new TreeNode(rootValue);

        int k=0;
        for(int i=0; i< inorder.length; i++){
            if(inorder[i]==rootValue){
                k = i;
                break;
            }
        }

        root.left = buildTree(inorder, inStart, k-1, postorder, postStart,
postStart+k-(inStart+1));
        // Becuase k is not the length, it it need to -(inStart+1) to get the length
        root.right = buildTree(inorder, k+1, inEnd, postorder,
postStart+k-inStart, postEnd-1);
        // postStart+k-inStart = postStart+k-(inStart+1) +1

        return root;
    }
}
```

Construct Binary Tree from Preorder and Inorder Traversal

Given preorder and inorder traversal of a tree, construct the binary tree.

Note:
You may assume that duplicates do not exist in the tree.

Tags: Tree, Array, Depth-first Search

Inorder: left->parent->right
Preorder: parent->left->right
This problem can be illustrated by using a simple example.

in-order: 4 2 5 (1) 6 7 3 8
post-order: 4 5 2 6 7 8 3 (1)

From the post-order array, we know that last element is the root. We can find the root in in-order array. Then we can identify the left and right sub-trees of the root from in-order array.

Using the length of left sub-tree, we can identify left and right sub-trees in post-order array. Recursively, we can build up the tree.

```
//Definition for binary tree
 public class TreeNode {
      int val;
      TreeNode left;
      TreeNode right;
      TreeNode(int x) { val = x; }
 }

public class Solution {
     public TreeNode buildTree(int[] inorder, int[] postorder) {
          int inStart = 0;
          int inEnd = inorder.length-1;
          int postStart =0;
          int postEnd = postorder.length-1;

          return buildTree(inorder, inStart, inEnd, postorder, postStart,
postEnd);
     }

     public TreeNode buildTree(int[] inorder, int inStart, int inEnd,
                                   int[] postorder, int postStart, int postEnd){
          if(inStart > inEnd || postStart > postEnd)
               return null;

          int rootValue = postorder[postEnd];
          TreeNode root = new TreeNode(rootValue);

          int k=0;
          for(int i=0; i< inorder.length; i++){
               if(inorder[i]==rootValue){
                    k = i;
                    break;
               }
          }

          root.left = buildTree(inorder, inStart, k-1, postorder, postStart,
postStart+k-(inStart+1));
```

```
            // Becuase k is not the length, it it need to -(inStart+1) to get the length
            root.right = buildTree(inorder, k+1, inEnd, postorder,
postStart+k-inStart, postEnd-1);
            // postStart+k-inStart = postStart+k-(inStart+1) +1

            return root;
        }
}
```

Convert Sorted Array to Binary Search Tree

Given an array where elements are sorted in ascending order, convert it to a height balanced BST.

Tags: Tree, Depth-first Search

Straightforward! Recursively do the job.

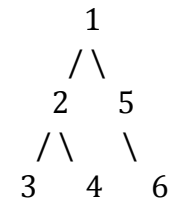
```
// Definition for binary tree
class TreeNode {
    int val;
    TreeNode left;
    TreeNode right;

    TreeNode(int x) {
        val = x;
    }
}

public class Solution {
    public TreeNode sortedArrayToBST(int[] num) {
        if (num.length == 0)
            return null;

        return sortedArrayToBST(num, 0, num.length - 1);
    }

    public TreeNode sortedArrayToBST(int[] num, int start, int end) {
        if (start > end)
            return null;

        int mid = (start + end) / 2;
        TreeNode root = new TreeNode(num[mid]);
```

```
            root.left = sortedArrayToBST(num, start, mid - 1);
            root.right = sortedArrayToBST(num, mid + 1, end);

            return root;
        }
}
```

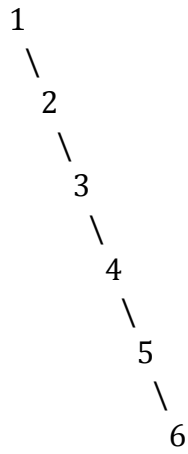
Flatten Binary Tree to Linked List

Given a binary tree, flatten it to a linked list in-place.

For example,
Given



The flattened tree should look like:



click to show hints.

Hints:
If you notice carefully in the flattened tree, each node's right child points to the next node of a pre-order traversal.

Tags: Tree, Depth-first Search
Go down through the left, when right is not null, push right to stack.

```
/**
 * Definition for binary tree
 * public class TreeNode {
```

```
 *          int val;
 *          TreeNode left;
 *          TreeNode right;
 *          TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public void flatten(TreeNode root) {
        Stack<TreeNode> stack = new Stack<TreeNode>();
        TreeNode p = root;

        while(p != null || !stack.empty()){

            if(p.right != null){
                stack.push(p.right);
            }

            if(p.left != null){
                p.right = p.left;
                p.left = null;
            }else if(!stack.empty()){
                TreeNode temp = stack.pop();
                p.right=temp;
            }

            p = p.right;
        }
    }
}
```

Maximum Depth of Binary Tree

Given a binary tree, find its maximum depth.

The maximum depth is the number of nodes along the longest path from the root node down to the farthest leaf node.

Tags: Tree, Depth-first Search

1. Recursive:

```
/**

 * Definition for binary tree
```

```
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public int maxDepth(TreeNode root) {
        if(root == null)    return 0;

        return getDepth(root, 1);
    }

    public int getDepth(TreeNode node, int depth) {
        int left = depth, right = depth;
        if(node.left != null) left = getDepth(node.left, depth + 1);
        if(node.right != null) right = getDepth(node.right, depth + 1);

        return left > right ? left : right;
    }
}
```

2. Non-recursive:

```
/**
 * Definition for binary tree
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public int maxDepth(TreeNode root) {
        if(root == null)    return 0;

        // Non-recursive, use level order triversal
        ArrayList<TreeNode> q = new ArrayList<TreeNode>();
        q.add(root);
        int depth = 0;

        while(!q.isEmpty()) {
            ArrayList<TreeNode> next = new ArrayList<TreeNode>();
            for(TreeNode node : q) {
                if(node.left != null)   next.add(node.left);
```

```
                if(node.right != null)    next.add(node.right);
            }
            q = new ArrayList<TreeNode>(next);
            depth++;
        }

        return depth;
    }
}
```

Minimum Depth of Binary Tree

Given a binary tree, find its minimum depth.

The minimum depth is the number of nodes along the shortest path from the root node down to the nearest leaf node.

Tags: Tree, Depth-first Search

Need to know LinkedList is a queue. add() and remove() are the two methods to manipulate the queue.

```
/**
 * Definition for binary tree
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public int minDepth(TreeNode root) {
        if(root == null){
```

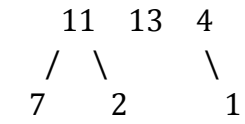
```
            return 0;
        }

        LinkedList<TreeNode> nodes = new LinkedList<TreeNode>();
        LinkedList<Integer> counts = new LinkedList<Integer>();

        nodes.add(root);
        counts.add(1);

        while(!nodes.isEmpty()){
            TreeNode curr = nodes.remove();
            int count = counts.remove();

            if(curr.left != null){
                nodes.add(curr.left);
                counts.add(count+1);
            }

            if(curr.right != null){
                nodes.add(curr.right);
                counts.add(count+1);
            }

            if(curr.left == null && curr.right == null){
                return count;
            }
        }

        return 0;
    }
}
```

Path Sum

Given a binary tree and a sum, determine if the tree has a root-to-leaf path such that adding up all the values along the path equals the given sum.

For example:
Given the below binary tree and sum = 22,

```
              5
             / \
            4   8
           /   / \
```



return true, as there exist a root-to-leaf path 5->4->11->2 which sum is 22.

Tags: Tree, Depth-first Search

Add all node to a queue and store sum value of each node to another queue. When it is a leaf node, check the stored sum value.

For example above, the queue would be: 5 - 4 - 8 - 11 - 13 - 4 - 7 - 2 - 1. It will check node 13, 7, 2 and 1.

This is a typical breadth first search(BFS) problem.

```
/**
 * Definition for binary tree
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public boolean hasPathSum(TreeNode root, int sum) {
        if(root == null){
            return false;
        }
```

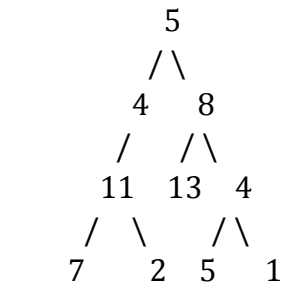
```
        if(root.left==null && root.right==null){

            return root.val==sum;

        }



        return hasPathSum(root.left, sum-root.val) || hasPathSum(root.right, sum-root.val);

    }

}

/**
 * Definition for binary tree
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public boolean hasPathSum(TreeNode root, int sum) {
        if(root == null) return false;

        LinkedList<TreeNode> nodes = new LinkedList<TreeNode>();
        LinkedList<Integer> values = new LinkedList<Integer>();

        nodes.add(root);
        values.add(root.val);

        while(!nodes.isEmpty()){
            TreeNode curr = nodes.poll();
            int sumValue = values.poll();

            if(curr.left == null && curr.right == null && sumValue==sum){
                return true;
            }

            if(curr.left != null){
```

```
                nodes.add(curr.left);
                values.add(sumValue+curr.left.val);
            }

            if(curr.right != null){
                nodes.add(curr.right);
                values.add(sumValue+curr.right.val);
            }
        }

        return false;
    }
}
```

Path Sum II

Given a binary tree and a sum, find all root-to-leaf paths where each path's sum equals the given sum.

For example:
Given the below binary tree and sum = 22,



return
[
    [5,4,11,2],
    [5,8,4,5]
]
Tags: Tree, Depth-first Search

```
/**

 * Definition for binary tree

 * public class TreeNode {

 *     int val;
```

```
 *          TreeNode left;
 *          TreeNode right;
 *          TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public ArrayList<ArrayList<Integer>> pathSum(TreeNode root, int sum) {
        ArrayList<ArrayList<Integer>> res = new ArrayList<ArrayList<Integer>>();
        pathSum(root, sum, res, new ArrayList<Integer>());
        return res;
    }

    private void pathSum(TreeNode root, int sum, ArrayList<ArrayList<Integer>> res, ArrayList<Integer> list){
        if(root == null)
            return;
        list.add(root.val);
        sum -= root.val;
        if(root.left == null && root.right == null && sum == 0)
            res.add(new ArrayList<Integer>(list));
        else{
            pathSum(root.left, sum, res, list);
```

```
                pathSum(root.right, sum, res, list);

            }

            list.remove(list.size() - 1);

        }

}
```

Populating Next Right Pointers in Each Node

Given a binary tree

```
    struct TreeLinkNode {
      TreeLinkNode *left;
      TreeLinkNode *right;
      TreeLinkNode *next;
    }
```

Populate each next pointer to point to its next right node. If there is no next right node, the next pointer should be set to NULL.

Initially, all next pointers are set to NULL.

Note:

You may only use constant extra space.
You may assume that it is a perfect binary tree (ie, all leaves are at the same level, and every parent has two children).
For example,
Given the following perfect binary tree,

```
         1
       /   \
      2     3
     / \   / \
    4   5 6   7
```

After calling your function, the tree should look like:

```
         1 -> NULL
       /   \
      2 -> 3 -> NULL
     / \   / \
    4->5->6->7 -> NULL
```

Tags: Tree, Depth-first Search

```
/**
 * Definition for binary tree with next pointer.
 * public class TreeLinkNode {
 *     int val;
 *     TreeLinkNode left, right, next;
 *     TreeLinkNode(int x) { val = x; }
 * }
 */
public class Solution {
    public void connect(TreeLinkNode root) {
        if(root==null){return;}
        if(root.left!=null){
            root.left.next = root.right;
        }
        if(root.right!=null && root.next!=null){
            root.right.next = root.next.left;
        }
            connect(root.left);
            connect(root.right);
    }
}
```

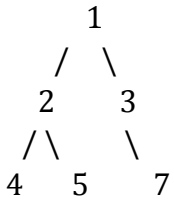
Populating Next Right Pointers in Each Node II

Follow up for problem "Populating Next Right Pointers in Each Node".

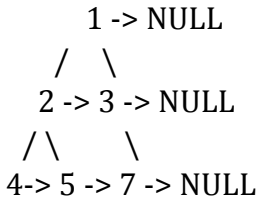
What if the given tree could be any binary tree? Would your previous solution still work?

Note:

You may only use constant extra space.
For example,
Given the following binary tree,



After calling your function, the tree should look like:



Tags: Tree, Depth-first Search

```
/**
 * Definition for binary tree with next pointer.
 * public class TreeLinkNode {
 *     int val;
 *     TreeLinkNode left, right, next;
 *     TreeLinkNode(int x) { val = x; }
 * }
 */
public class Solution {
```

```
public void connect(TreeLinkNode root) {
    if (root == null) return;

    //如果右孩子不为空，左孩子的 next 是右孩子。
    //反之，找 root next 的至少有一个孩子不为空的节点
    if (root.left != null) {
        if (root.right != null) {
            root.left.next = root.right;
        }
        else {
            TreeLinkNode p = root.next;
            while (p != null && p.left == null && p.right == null)
                p = p.next;
            if (p != null)
                root.left.next = p.left == null ? p.right : p.left;
        }
    }

    //右孩子的 next  根节点至少有一个孩子不为空的 next
    if (root.right != null) {
        TreeLinkNode p = root.next;
        while (p != null && p.left == null && p.right == null)
```

```
                p = p.next;
            if (p != null)
                root.right.next = p.left == null ? p.right : p.left;
        }
        connect(root.right);
        connect(root.left);
    }
}
```

Recover Binary Search Tree

Two elements of a binary search tree (BST) are swapped by mistake.

Recover the tree without changing its structure.

Note:
A solution using O(n) space is pretty straight forward. Could you devise a constant space solution?
confused what "{1,#,2,3}" means? > read more on how binary tree is serialized on OJ.

Tags: Tree, Depth-first Search

We can use in-order traverse to find the swapped element. During the traverse, we can find the element that is smaller than the previous node. Using this method we can find the swapped node. Save it and swap them. Done.

```
/**
 * Definition for binary tree
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
```

```
 *         TreeNode right;

 *         TreeNode(int x) { val = x; }

 * }

 */
public class Solution {
   TreeNode node1 = null;
   TreeNode node2 = null;
   public void recoverTree(TreeNode root) {
      inorderTraverse(root);
      int tmp = node1.val;
      node1.val = node2.val;
      node2.val = tmp;
   }

   TreeNode prev = null;
   private void inorderTraverse(TreeNode root) {
      if (root == null)
         return;
      inorderTraverse(root.left);
      if (prev != null) {
         if (root.val <= prev.val) {
            if (node1 == null)
               node1 = prev;
            node2 = root;
         }
      }
      prev = root;
      inorderTraverse(root.right);
   }
}
```

Complexity

The time complexity is O(n). But the space complexity is not constant, since we use recursive function.

Follow-up

After searching, I found there is a way to use O(1) space to do the in-order traverse, which is called Morris traverse.

The Morris traverse is like the following.

Firstly, take the root node as current node.

Then there are two possibilities.

1. If current node doesn't have left child, output the value. And current = current.right.
2. If current node has left child, try to find the precursor node of current node, which is the right-most node of the left child of current. If the right child of it is null (If we don't modify the tree, it should be null), set current as its right child, and current = current.left. Otherwise (It means that we have modify the tree and we have traverse all nodes in the left subtree of current node), set it to null, output current. And current = current.right.

During the traverse, we can find the nodes which are needed to be swapped.

```
public class Solution {
  public void recoverTree(TreeNode root) {
    TreeNode current = root;
    TreeNode prev = null;
    TreeNode node1 = null;
    TreeNode node2 = null;
    while (current != null) {
      if (current.left == null) {
        if (prev != null) {
          if (prev.val >= current.val) {
            if (node1 == null)
              node1 = prev;
            node2 = current;
          }
        }
        prev = current;
        current = current.right;
      } else {
        TreeNode t = current.left;
        while (t.right != null && t.right != current)
          t = t.right;
        if (t.right == null) {
          t.right = current;
          current = current.left;
        } else {
          t.right = null;
```

```
            if (prev != null) {
              if (prev.val >= current.val) {
                if (node1 == null)
                  node1 = prev;
                node2 = current;
              }
            }
            prev = current;
            current = current.right;
          }
        }
      }
      int tmp = node1.val;
      node1.val = node2.val;
      node2.val = tmp;
    }
}
```

The space complexity of this algorithm is O(1).

But what about the time complexity? In fact, we only visit every edge twice. One is for going to that node, and another one is for searching the precursor node. There are only n−1 edges in a tree. So the time complexity is also O(n).

Same Tree

Given two binary trees, write a function to check if they are equal or not.

Two binary trees are considered equal if they are structurally identical and the nodes have the same value.

Tags: Tree, Depth-first Search

```
/**
 * Definition for binary tree
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
```

```
public class Solution {
    public boolean isSameTree(TreeNode p, TreeNode q) {
        if(p == null && q == null){
            return true;
        }else if(p != null && q != null){
            if(p.val == q.val){
                return isSameTree(p.left, q.left) && isSameTree(q.right,
p.right);
            }
        }else{
            return false;
        }

        return false;
    }
}
```

Sum Root to Leaf Numbers

Given a binary tree containing digits from 0-9 only, each root-to-leaf path could represent a number.

An example is the root-to-leaf path 1->2->3 which represents the number 123.

Find the total sum of all root-to-leaf numbers.

For example,

```
    1
   / \
  2   3
```

The root-to-leaf path 1->2 represents the number 12.
The root-to-leaf path 1->3 represents the number 13.

Return the sum = 12 + 13 = 25.

Tags: Tree, Depth-first Search

```
/**
 * Definition for binary tree
 * public class TreeNode {
```

```
 *          int val;
 *          TreeNode left;
 *          TreeNode right;
 *          TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
       public int sumNumbers(TreeNode root) {
            return GenerateSum(root,0);
       }

      public int GenerateSum(TreeNode root, int sum){
            if(root==null) return 0;
            int result = root.val+sum*10;
            if(root.left==null && root.right==null){
                  return result;
            }
            return GenerateSum(root.left,result)+GenerateSum(root.right,result);

      }
}
```

Symmetric Tree

Given a binary tree, check whether it is a mirror of itself (ie, symmetric around its center).

For example, this binary tree is symmetric:

```
    1
   / \
  2   2
 / \ / \
3  4 4  3
```

But the following is not:

```
    1
   / \
  2   2
   \   \
   3    3
```

Note:
Bonus points if you could solve it both recursively and iteratively.

confused what "{1,#,2,3}" means? > read more on how binary tree is serialized on OJ.

Tags: Tree, Depth-first Search

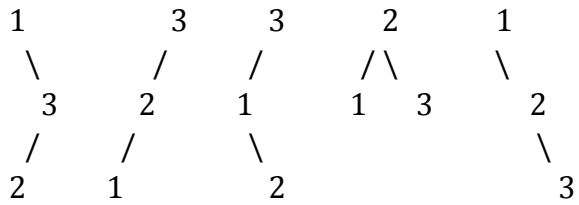
```
/**
 * Definition for binary tree
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
```

```
public class Solution {

    public boolean isSymmetric(TreeNode root) {

        if (root == null) {

            return true;

        }



        return helper(root.left, root.right);

    }



    private boolean helper(TreeNode a, TreeNode b) {

        if (a == null && b == null) return true;

        if (a == null || b == null) return false;    // only one has node in a tree and b tree

        if (a.val != b.val) return false;

        return helper(a.left, b.right) && helper(a.right, b.left); // a goes in in-order traversal, b goes right first then left.

    }

}
```

Unique Binary Search Trees

Given n, how many structurally unique BST's (binary search trees) that store values 1...n?

For example,
Given n = 3, there are a total of 5 unique BST's.



Tags: Tree, Dynamic Programming

In “Unique Binary Search Trees”, we are only required to output the number of the trees. We know that all nodes in the left subtree are smaller than the root. And all nodes in the right subtree are larger than the root. For a integer n, we have n options to be the root. In these options, assuming i is the value that we choose to be the root. The value in left subtree are from 1 to i – 1, and the values in right subtree are from i + 1 to n. If 1 to i – 1 can form p different trees, and i + 1 to n can form q different trees, then we will have p * q trees when i is the root. In fact, the number of different trees depends on how many number to form the tree.

We can use an array to save the number of different trees that n integers can form. We fill the array from bottom to up, starting from 0 to n.

In “Unique Binary Search Trees II”, we need to generate all trees. The algorithm has the same idea. But we don’t just return the numbers. We return the trees that n integers can from. Then we use a nested for-loop to go through every possible combinations of left tree and right tree for a given root. We will do it recursively because it’s the same for the left tree and right tree of root.

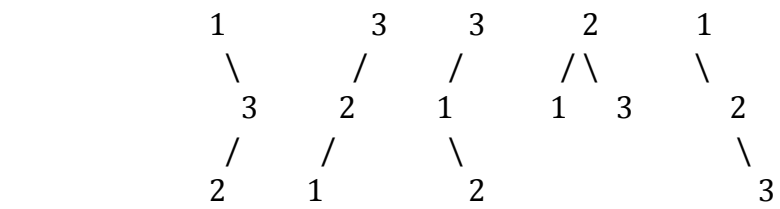
```
public class Solution {
    public int numTrees(int n) {
        int[] N = new int[n + 1];
        N[0] = 1;
        for (int i = 1; i <= n; i++) {
            for (int j = 0; j < i; j++) {
                N[i] += N[j] * N[i - j - 1];
            }
        }
        return N[n];
    }
}
```

The complexity of the first problem is $O(n^2)$.

Unique Binary Search Trees II

Given n, generate all structurally unique BST's (binary search trees) that store values 1...n.

For example,
Given n = 3, your program should return all 5 unique BST's shown below.



confused what "{1,#,2,3}" means? > read more on how binary tree is serialized on OJ.

Tags: Tree, Dynamic Programming

```
public class Solution {
  public List<TreeNode> generateTrees(int n) {
    return generateTrees(1, n);
  }

  public List<TreeNode> generateTrees(int start, int end) {
    List<TreeNode> list = new LinkedList<>();
    if (start > end) {
      list.add(null);
      return list;
    }
    for (int i = start; i <= end; i++) {
      List<TreeNode> lefts = generateTrees(start, i - 1);
      List<TreeNode> rights = generateTrees(i + 1, end);
      for (TreeNode left : lefts) {
        for (TreeNode right : rights) {
          TreeNode node = new TreeNode(i);
          node.left = left;
          node.right = right;
          list.add(node);
        }
      }
    }
    return list;
  }
```

}

Validate Binary Search Tree

Given a binary tree, determine if it is a valid binary search tree (BST).

Assume a BST is defined as follows:

1. The left subtree of a node contains only nodes with keys less than the node's key.
2. The right subtree of a node contains only nodes with keys greater than the node's key.
3. Both the left and right subtrees must also be binary search trees.
confused what "{1,#,2,3}" means? > read more on how binary tree is serialized on OJ.

Tags: Tree, Depth-first Search

```
/**
 * Definition for binary tree
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    public boolean isValidBST(TreeNode root) {
        return isBST(root, Integer.MIN_VALUE, Integer.MAX_VALUE);
    }
```

```
    public boolean isBST(TreeNode root, int min, int max) {
        if (root == null) {
            return true;
        }

        return root.val > min && root.val < max
                && isBST(root.left, min, root.val)
                && isBST(root.right, root.val, max);
    }
}
```

LinkedList

Add Two Numbers

You are given two linked lists representing two non-negative numbers. The digits are stored in reverse order and each of their nodes contain a single digit. Add the two numbers and return it as a linked list.

Input: (2 -> 4 -> 3) + (5 -> 6 -> 4)
Output: 7 -> 0 -> 8

Hide Tags Linked List Math

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
```

```
 *          int val;
 *          ListNode next;
 *          ListNode(int x) {
 *                  val = x;
 *                  next = null;
 *          }
 * }
 */
public class Solution {
        public ListNode addTwoNumbers(ListNode l1, ListNode l2) {
            if(l1==null){
                return l2;
            }

            if(l2==null){
                return l1;
            }

            int len1 = 0;
            int len2 = 0;

            ListNode head = l1;
```

```
while(head!=null){
        ++len1;
        head = head.next;
}

head = l2;
while(head!=null){
        ++len2;
        head = head.next;
}

ListNode longer = len1>=len2 ? l1:l2;
ListNode shorter = len1<len2 ? l1:l2;

ListNode result = null;
ListNode sum = null;
int val = 0;
int carry = 0;

while(shorter!=null){
        val = longer.val+shorter.val+carry;
        carry = val/10;
```

```
        val -= carry*10;


        if(sum == null){
            sum = new ListNode(val);
            result=sum;
        } else {
            sum.next = new ListNode(val);
            sum = sum.next;
        }



        longer = longer.next;
        shorter = shorter.next;
    }


    while(longer!=null){
        val = longer.val+carry;
        carry = val/10;
        val-= carry*10;


        sum.next = new ListNode(val);
        sum = sum.next;
```

```
            longer = longer.next;
        }

        if(carry!=0){
            sum.next = new ListNode(carry);
        }
        return result;
    }
}
```

Convert Sorted List to Binary Search Tree

Given a singly linked list where elements are sorted in ascending order, convert it to a height balanced BST.

Hide Tags Depth-first Search Linked List

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) { val = x; next = null; }
 * }
 */
```

```
/**
 * Definition for binary tree
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
public class Solution {
    ListNode h;
    public TreeNode sortedListToBST(ListNode head) {
        if(head==null){return null;}

        h = head;
        int len = getLen(head);
        return sortedListToBST(0,len-1);
    }

    public int getLen(ListNode head){
        int len = 0;
        ListNode p = head;
```

```
        while(p!=null){
            len++;
            p = p.next;
        }
        return len;
    }

    public TreeNode sortedListToBST(int start, int end){
        while(start>end){return null;}

        int mid = (start+end)/2;
        TreeNode left = sortedListToBST(start, mid-1);
        TreeNode root = new TreeNode(h.val);
        h = h.next;
        TreeNode right = sortedListToBST(mid+1, end);

        root.left = left;
        root.right = right;

        return root;
    }
```

}

Copy List with Random Pointer

A linked list is given such that each node contains an additional random pointer which could point to any node in the list or null.

Return a deep copy of the list.

Hide Tags Hash Table Linked List

copy every node, i.e., duplicate every node, and insert it to the list
copy random pointers for all newly created nodes
break the list to two

```
/**
 * Definition for singly-linked list with a random pointer.
 * class RandomListNode {
 *     int label;
 *     RandomListNode next, random;
 *     RandomListNode(int x) { this.label = x; }
 * };
 */
public RandomListNode copyRandomList(RandomListNode head) {
    if (head == null)
        return null;

    RandomListNode p = head;

    // copy every node and insert to list
    while (p != null) {
        RandomListNode copy = new RandomListNode(p.label);
        copy.next = p.next;
        p.next = copy;
        p = copy.next;
```

```
        }

        // copy random pointer for each new node
        p = head;
        while (p != null) {
            if (p.random != null)
                p.next.random = p.random.next;
            p = p.next.next;
        }

        // break list to two
        p = head;
        RandomListNode newHead = head.next;
        while (p != null) {
            RandomListNode temp = p.next;
            p.next = temp.next;
            if (temp.next != null)
                temp.next = temp.next.next;
            p = p.next;
        }

        return newHead;
    }
```

HashMap

```
public RandomListNode copyRandomList(RandomListNode head) {
        if (head == null)
            return null;
        HashMap<RandomListNode, RandomListNode> map = new
HashMap<RandomListNode, RandomListNode>();
        RandomListNode newHead = new RandomListNode(head.label);

        RandomListNode p = head;
        RandomListNode q = newHead;
        map.put(head, newHead);

        p = p.next;
        while (p != null) {
            RandomListNode temp = new RandomListNode(p.label);
            map.put(p, temp);
            q.next = temp;
            q = temp;
            p = p.next;
        }
```

```
        p = head;
        q = newHead;
        while (p != null) {
            if (p.random != null)
                q.random = map.get(p.random);
            else
                q.random = null;

            p = p.next;
            q = q.next;
        }

        return newHead;
}
```

Insertion Sort List

Sort a linked list using insertion sort.

Hide Tags Linked List Sort

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) {
 *         val = x;
 *         next = null;
 *     }
 * }
 */
```

```
public class Solution {
    public static ListNode insertionSortList(ListNode head) {

        if (head == null || head.next == null)
            return head;

        ListNode newHead = new ListNode(head.val);
        ListNode pointer = head.next;

        // loop through each element in the list
        while (pointer != null) {
            // insert this element to the new list

            ListNode innerPointer = newHead;
            ListNode next = pointer.next;

            if (pointer.val <= newHead.val) {
                ListNode oldHead = newHead;
                newHead = pointer;
                newHead.next = oldHead;
            } else {
                while (innerPointer.next != null) {
```

```
                    if (pointer.val > innerPointer.val && pointer.val <= innerPointer.next.val) {
                        ListNode oldNext = innerPointer.next;
                        innerPointer.next = pointer;
                        pointer.next = oldNext;
                    }

                    innerPointer = innerPointer.next;
                }

                if (innerPointer.next == null && pointer.val > innerPointer.val) {
                    innerPointer.next = pointer;
                    pointer.next = null;
                }
            }

            // finally
            pointer = next;
        }

        return newHead;
```

```
    }
}
```

Linked List Cycle

Given a linked list, determine if it has a cycle in it.

Follow up:
Can you solve it without using extra space?

Hide Tags Linked List Two Pointers

```
/**
 * Definition for singly-linked list.
 * class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) {
 *         val = x;
 *         next = null;
 *     }
 * }
 */
public class Solution {
    public boolean hasCycle(ListNode head) {
        ListNode fast = head;
        ListNode slow = head;
```

```
        if(head == null)
            return false;

        if(head.next == null)
            return false;

        while(fast != null && fast.next != null){
            slow = slow.next;
            fast = fast.next.next;

            if(slow == fast)
                return true;
        }

        return false;
    }
}
```

Linked List Cycle II

Given a linked list, return the node where the cycle begins. If there is no cycle, return null.

Follow up:
Can you solve it without using extra space?

Hide Tags Linked List Two Pointers

```
/**
 * Definition for singly-linked list.
 * class ListNode {
 *         int val;
 *         ListNode next;
 *         ListNode(int x) {
 *                 val = x;
 *                 next = null;
 *         }
 * }
 */
public class Solution {
        public static ListNode detectCycle(ListNode head) {
        ListNode slow = head;
        ListNode fast = head;

        while (true) {
                if (fast == null || fast.next == null) {
                        return null;        //遇到 null 了，说明不存在环
                }
                slow = slow.next;
```

```
            fast = fast.next.next;
            if (fast == slow) {
                break;      //第一次相遇在 Z 点
            }
        }

        slow = head;        //slow 从头开始走，
        while (slow != fast) {      //二者相遇在 Y 点，则退出
            slow = slow.next;
            fast = fast.next;
        }
        return slow;
    }
}
```

Merge k Sorted Lists

Merge k sorted linked lists and return it as one sorted list. Analyze and describe its complexity.

Hide Tags Divide and Conquer Linked List Heap

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
```

```
 *          ListNode next;
 *          ListNode(int x) {
 *                  val = x;
 *                  next = null;
 *          }
 * }
 */
public class Solution {
       public ListNode mergeKLists(ArrayList<ListNode> lists) {
            if(lists==null || lists.size()==0){
                  return null;
            }
            return helper(lists,0,lists.size()-1);
       }

      public ListNode helper(ArrayList<ListNode> lists, int l, int r){
            if(l<r){
                  int m = (l+r)/2;
                  return merge(helper(lists,l,m), helper(lists,m+1,r));
            }
            return lists.get(l);
      }
```

```
private ListNode merge(ListNode l1, ListNode l2)
{
    ListNode dummy = new ListNode(0);
    dummy.next = l1;
    ListNode cur = dummy;
    while(l1!=null && l2!=null)
    {
        if(l1.val<l2.val)
        {
            l1 = l1.next;
        }
        else
        {
            ListNode next = l2.next;
            cur.next = l2;
            l2.next = l1;
            l2 = next;
        }
        cur = cur.next;
    }
    if(l2!=null)
```

```
                cur.next = l2;
            return dummy.next;
        }
}
```

Merge Two Sorted Lists

Merge two sorted linked lists and return it as a new list. The new list should be made by splicing together the nodes of the first two lists.

Hide Tags Linked List

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) {
 *         val = x;
 *         next = null;
 *     }
 * }
 */
public class Solution {
    public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
        if(l1==null){
```

```
        return l2;
    }
    if(l2==null){
        return l1;
    }

    ListNode node = null;
    ListNode head = null;

    while(l1!=null && l2!=null){
    if(l1.val<=l2.val){
        if(node == null){
            node = l1;
            head = node;
        } else {
            node.next = l1;
            node = node.next;
        }

        l1 = l1.next;
    } else {
        if(node == null) {
```

```
                node = l2;
                head = node;
            } else {
                node.next = l2;
                node = node.next;
            }

            l2 = l2.next;
        }
        }

        if(l1!=null){
            node.next = l1;
        } else if(l2!=null) {
            node.next = l2;
        }


        return head;
    }
}
```

Partition List

Given a linked list and a value x, partition it such that all nodes less than x come before nodes greater than or equal to x.

You should preserve the original relative order of the nodes in each of the two partitions.

For example,
Given 1->4->3->2->5->2 and x = 3,
return 1->2->2->4->3->5.

Hide Tags Linked List Two Pointers

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) {
 *         val = x;
 *         next = null;
 *     }
 * }
 */
public class Solution {
    public ListNode partition(ListNode head, int x)
    {
        ListNode leftHead = new ListNode(0);
```

```
        ListNode rightHead = new ListNode(0);
        ListNode cur1 = leftHead;
        ListNode cur2 = rightHead;
        ListNode cur = head;
        while (cur != null) {
            if (cur.val < x) {
                cur1.next = cur;
                cur = cur.next;
                cur1 = cur1.next;
            } else {
                cur2.next = cur;
                cur = cur.next;
                cur2 = cur2.next;
            }
        }
        cur2.next = null;
        cur1.next = rightHead.next;
        return leftHead.next;
    }
}
```

Remove Duplicates from Sorted List

Given a sorted linked list, delete all duplicates such that each element appear only once.

For example,
Given 1->1->2, return 1->2.
Given 1->1->2->3->3, return 1->2->3.

Hide Tags Linked List

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) {
 *         val = x;
 *         next = null;
 *     }
 * }
 */
public class Solution {
    public ListNode deleteDuplicates(ListNode head) {
        if(head==null || head.next==null){return head;}

        ListNode p = head;
```

```
            while( p!= null && p.next != null){
                if(p.val == p.next.val){
                    p.next = p.next.next;
                }else{
                    p = p.next;
                }
            }
            return head;
        }
}
```

Remove Duplicates from Sorted List II

Given a sorted linked list, delete all nodes that have duplicate numbers, leaving only distinct numbers from the original list.

For example,
Given 1->2->3->3->4->4->5, return 1->2->5.
Given 1->1->1->2->3, return 2->3.

Hide Tags Linked List

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
```

```
 *          ListNode(int x) {
 *                  val = x;
 *                  next = null;
 *          }
 * }
 */
public class Solution {
        public ListNode deleteDuplicates(ListNode head) {
                ListNode prev = new ListNode(0);
                prev.next = head;
                head = prev;

                ListNode n1=head;
                while(n1.next!=null){
                        ListNode n2=n1.next;
                        while(n2.next!=null && n2.next.val==n2.val){
                                n2=n2.next;
                        }
                        if(n2!=n1.next){
                                n1.next=n2.next;
                        }else{
                                n1=n1.next;
```

```
                }
            }
            return head.next;
        }
}
```

Remove Nth Node From End of List

Given a linked list, remove the nth node from the end of list and return its head.

For example,

Given linked list: 1->2->3->4->5, and n = 2.

After removing the second node from the end, the linked list becomes 1->2->3->5.
Note:
Given n will always be valid.
Try to do this in one pass.

Hide Tags Linked List Two Pointers

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) {
 *         val = x;
 *         next = null;
 *     }
```

```
 * }
 */
public class Solution {
    public ListNode removeNthFromEnd(ListNode head, int n) {
        ListNode first = head;
        ListNode second = head;
        int i=0;
        while(first!=null && i<n){
            first = first.next;
            ++i;
        }

        if(first==null){
            return second.next;
        }

        while(first.next!=null){
            first=first.next;
            second=second.next;
        }

        second.next = second.next.next;
```

```
            return head;
        }
}
```

Reorder List

Given a singly linked list L: L0→L1→⋯→Ln-1→Ln,
reorder it to: L0→Ln→L1→Ln-1→L2→Ln-2→⋯

You must do this in-place without altering the nodes' values.

For example,
Given {1,2,3,4}, reorder it to {1,4,2,3}.

Hide Tags Linked List

```
/**
 * Definition for singly-linked list.
 * class ListNode {
 *         int val;
 *         ListNode next;
 *         ListNode(int x) {
 *                 val = x;
 *                 next = null;
 *         }
 * }
 */
public class Solution {
```

```
public static void reorderList(ListNode head) {
if (head == null || head.next == null)
        return;

// partition the list into 2 sublists of equal length
ListNode slowNode = head, fastNode = head;
while (fastNode.next != null) {
        fastNode = fastNode.next;
        if (fastNode.next != null) {
                fastNode = fastNode.next;
        } else {
                break;
        }
        slowNode = slowNode.next;
}
// 2 sublist heads
ListNode head1 = head, head2 = slowNode.next;
// detach the two sublists
slowNode.next = null;

// reverse the second sublist
ListNode cur = head2, post = cur.next;
```

```
        cur.next = null;
        while (post != null) {
            ListNode temp = post.next;
            post.next = cur;
            cur = post;
            post = temp;
        }
        head2 = cur; // the new head of the reversed sublist

        // merge the 2 sublists as required
        ListNode p = head1, q = head2;
        while (q != null) {
            ListNode temp1 = p.next;
            ListNode temp2 = q.next;
            p.next = q;
            q.next = temp1;
            p = temp1;
            q = temp2;
        }
    }
}
```

Reverse Linked List II

Reverse a linked list from position m to n. Do it in-place and in one-pass.

For example:
Given 1->2->3->4->5->NULL, m = 2 and n = 4,

return 1->4->3->2->5->NULL.

Note:
Given m, n satisfy the following condition:
$1 \leqslant m \leqslant n \leqslant$ length of list.

Hide Tags Linked List

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) {
 *         val = x;
 *         next = null;
 *     }
 * }
 */
public class Solution {
    public ListNode reverseBetween(ListNode head, int m, int n) {
        if (head==null || head.next==null){
```

```
        return head;
}

ListNode preHead=new ListNode(0);
preHead.next=head;

ListNode pre=preHead;
ListNode current=head;
ListNode end=head;

int countM=1;
int countN=1;
// find M point and N point
while (countM<m || countN<n ){
        if (countM<m){
                pre=pre.next;
                current=current.next;
                countM++;
        }

        if (countN<n){
                end=end.next;
```

```
                countN++;
            }
        }

        // reverse from M point to N Point
        reverse(pre, end.next);
        return preHead.next;

}

private void reverse(ListNode pre, ListNode endNext){
        ListNode cur=pre.next;

        while (cur.next!=endNext){

            ListNode next=cur.next;
            ListNode temp=pre.next;
            pre.next=next;
            cur.next=next.next;

            next.next=temp;
```

```
            }
        }
}
```

Reverse Nodes in k-Group

Given a linked list, reverse the nodes of a linked list k at a time and return its modified list.

If the number of nodes is not a multiple of k then left-out nodes in the end should remain as it is.

You may not alter the values in the nodes, only nodes itself may be changed.

Only constant memory is allowed.

For example,
Given this linked list: 1->2->3->4->5

For k = 2, you should return: 2->1->4->3->5

For k = 3, you should return: 3->2->1->4->5

Hide Tags Linked List

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) {
 *         val = x;
 *         next = null;
```

```
 *         }
 * }
 */
public class Solution {
    public ListNode reverseKGroup(ListNode head, int k) {
        if(head == null || k == 1) return head;
                    ListNode dummy = new ListNode(0);
                    dummy.next = head;
                    ListNode pre = dummy;
                    int i = 0;
                    while(head != null){
                        i++;
                        if(i % k ==0){
                            pre = reverse(pre, head.next);
                            head = pre.next;
                        }else {
                            head = head.next;
                        }
                    }
                    return dummy.next;
    }
```

```
    private static ListNode reverse(ListNode pre, ListNode next){
            ListNode last = pre.next;//where first will be doomed "last"
            ListNode cur = last.next;
            while(cur != next){
                last.next = cur.next;
                cur.next = pre.next;
                pre.next = cur;
                cur = last.next;
            }
            return last;
    }
}
```

Rotate List

Given a list, rotate the list to the right by k places, where k is non-negative.

For example:
Given 1->2->3->4->5->NULL and k = 2,
return 4->5->1->2->3->NULL.

Hide Tags Linked List Two Pointers

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
```

```
 *          ListNode(int x) {
 *                val = x;
 *                next = null;
 *          }
 * }
 */
public class Solution {
      public ListNode rotateRight(ListNode head, int n) {
            if (head==null|| n==0){
                  return head;
            }

            int len=1;
            ListNode last=head;

            // calculate the lenght of given list
            while(last.next!=null){
                  last=last.next;
                  len++;
            }

            last.next=head;
```

```
            // Should considered the situtation that n larger than given list's length
            int k=len-n%len;
            ListNode preHead=last;

            // find the point which are previuse for our target head
            while(k>0){
                preHead=preHead.next;
                k--;
            }

            head=preHead.next;
            preHead.next=null;

            return head;

        }
}
```

Sort List

Sort a linked list in O(n log n) time using constant space complexity.

Hide Tags Linked List Sort

```
/**
 * Definition for singly-linked list.
 * class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) {
 *         val = x;
 *         next = null;
 *     }
 * }
 */
public class Solution {
    public ListNode sortList(ListNode head) {
    return mergeSort(head);
}
private ListNode mergeSort(ListNode head)
{
    if(head == null || head.next == null)
        return head;
    ListNode walker = head;
    ListNode runner = head;
    while(runner.next!=null && runner.next.next!=null)
```

```
        {
            walker = walker.next;
            runner = runner.next.next;
        }
        ListNode head2 = walker.next;
        walker.next = null;
        ListNode head1 = head;
        head1 = mergeSort(head1);
        head2 = mergeSort(head2);
        return merge(head1, head2);
    }
    private ListNode merge(ListNode head1, ListNode head2)
    {
        ListNode helper = new ListNode(0);
        helper.next = head1;
        ListNode pre = helper;
        while(head1!=null && head2!=null)
        {
            if(head1.val<head2.val)
            {
                head1 = head1.next;
            }
```

```
            else
            {
                ListNode next = head2.next;
                head2.next = pre.next;
                pre.next = head2;
                head2 = next;
            }
            pre = pre.next;
        }
        if(head2!=null)
        {
            pre.next = head2;
        }
        return helper.next;
}
}
```

Swap Nodes in Pairs

Given a linked list, swap every two adjacent nodes and return its head.

For example,
Given 1->2->3->4, you should return the list as 2->1->4->3.

Your algorithm should use only constant space. You may not modify the values in the list, only nodes itself can be changed.

Hide Tags Linked List

```
/**
 * Definition for singly-linked list.
 * public class ListNode {
 *     int val;
 *     ListNode next;
 *     ListNode(int x) {
 *         val = x;
 *         next = null;
 *     }
 * }
 */
public class Solution {
    public ListNode swapPairs(ListNode head) {
        ListNode fake = new ListNode(0);
        fake.next = head;
        ListNode pre = fake, cur = head;

        while(cur!=null && cur.next!=null){
            pre.next = cur.next;
            cur.next = cur.next.next;
            pre.next.next = cur;
```

```
                pre = cur;
                cur = pre.next;
            }

            return fake.next;

        }
    }
```