# 1. 整体学习思路

这里讲的都是普通的数据结构,别纠结与细节问题,对数据结构和算法建立一个框架性的认识.

## 1.1数据结构的存储方式

数据结构存储方式只有两种：数组(顺序存储)和链表(链式存储)

多样化的数据结构都是在链表和数组上的特殊操作。

数组是紧凑型连续存储，有索引，所以可以随机访问。节约存储空间。但扩容就比较麻烦，必须一次性全部复制过去。插入和删除都必须保证后面的数据连续性。时间复杂度O(N)。

链表不连续，靠指针指向下一个元素。不存在扩容问题，删除和插入时间复杂度O（1）。但没有索引所以不能随机访问，消耗更多的存储空间。

## 1.2数据结构的基本操作

各种数据结构存在的目的都是应对不同的场景下，尽可能高效的增删改查。

数据结构的遍历+访问有两种形式：线性和非线性

线性就是for/while迭代为代表，非线性就是递归为代表。

典型的框架有:

（1）数组遍历框架

**void traverse(int[] arr) {**

**for (int i = 0; i < arr.length; i++) {**

**// 迭代访问 arr[i]**

**}**

**}**

（2）链表遍历框架

**/\* 基本的单链表节点 \*/**

**class ListNode {**

**int val;**

**ListNode next;**

**}**

​

**void traverse(ListNode head) {**

**for (ListNode p = head; p != null; p = p.next) {**

**// 迭代访问 p.val**

**}**

**}**

​

**void traverse(ListNode head) {**

**// 递归访问 head.val**

**traverse(head.next);**

**}**

（3）二叉树遍历框架

**/\* 基本的二叉树节点 \*/**

**class TreeNode {**

**int val;**

**TreeNode left, right;**

**}**

**​**

**void traverse(TreeNode root) {**

**traverse(root.left);**

**traverse(root.right);**

**}**

（4）N叉树遍历框架

**/\* 基本的 N 叉树节点 \*/**

**class TreeNode {**

**int val;**

**TreeNode[] children;**

**}**

**​**

**void traverse(TreeNode root) {**

**for (TreeNode child : root.children)**

**traverse(child);**

**}**

N叉树是由二叉树扩展来的，而图的遍历又是N叉树扩展而来。图就是好几棵N叉树的结合体。

## 1.3算法刷题指南

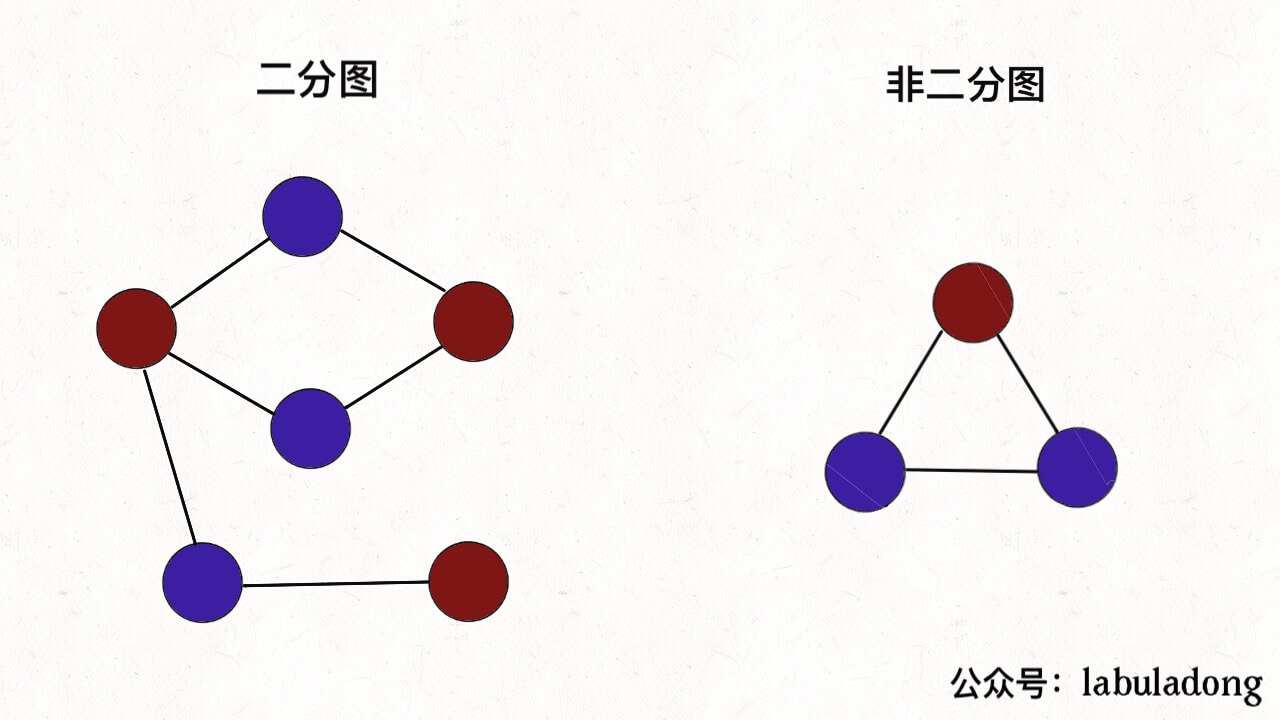
二叉树是最容易培养框架思维的，而且大部分算法技巧，本质上都是树的遍历问题。

学习算法先从刷树的题目开始，从框架上看问题，不要纠结细节。

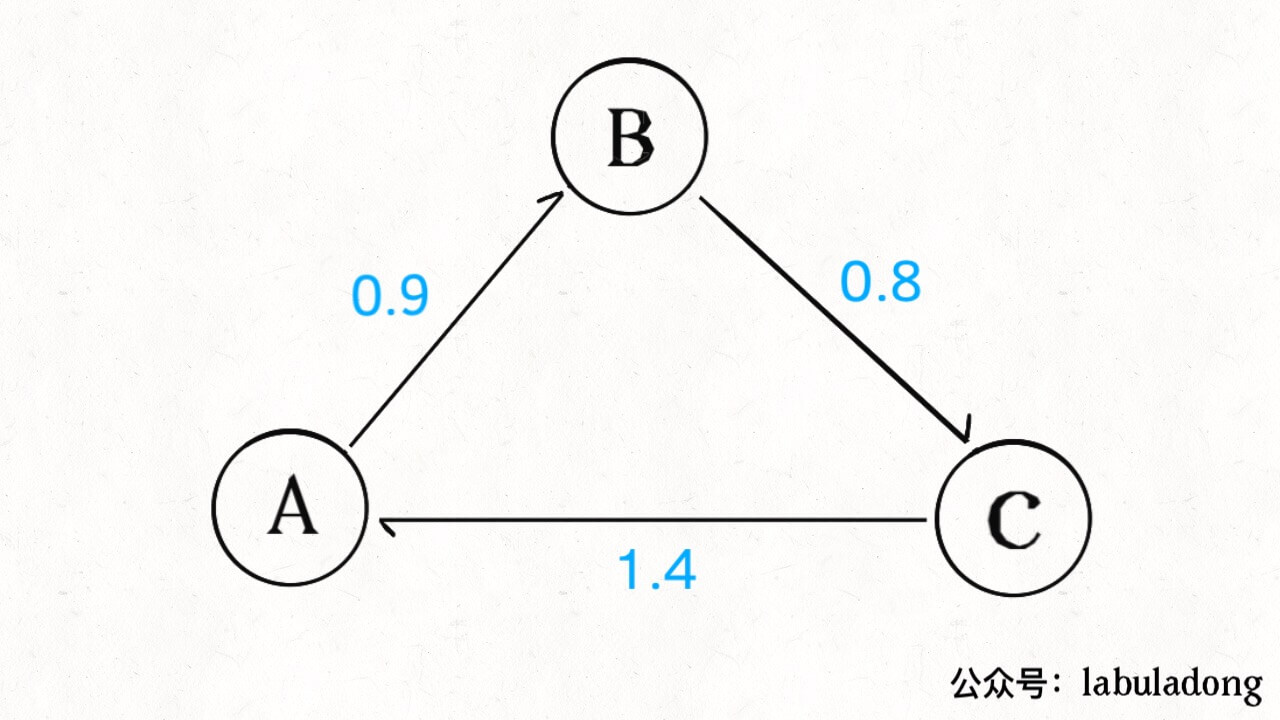
## 1.4数据结构和算法学习读什么书

《算法4》讲解详细，有大量配图，而且把算法联系现实场景。

1.4.1 二分图的应用



1.4.2 套汇的算法



图中加权代表汇率，100单位货币A换成B，再换成C，最后换回A,就可以得到

100\*0.9\*0.8\*1.4=100.8

在图论中经典算法Bellman-Ford算法，可以用于寻找负权重环。寻找权重乘积大于1的环，就转换成了寻找权重和小于0的环，也就是寻找套汇的机会。

# 2. 手刷链表题目

## 2.1递归反转整个链表

先看实现代码:

**ListNode reverse(ListNode head) {**

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

**ListNode last = reverse(head.next);**

**head.next.next = head;**

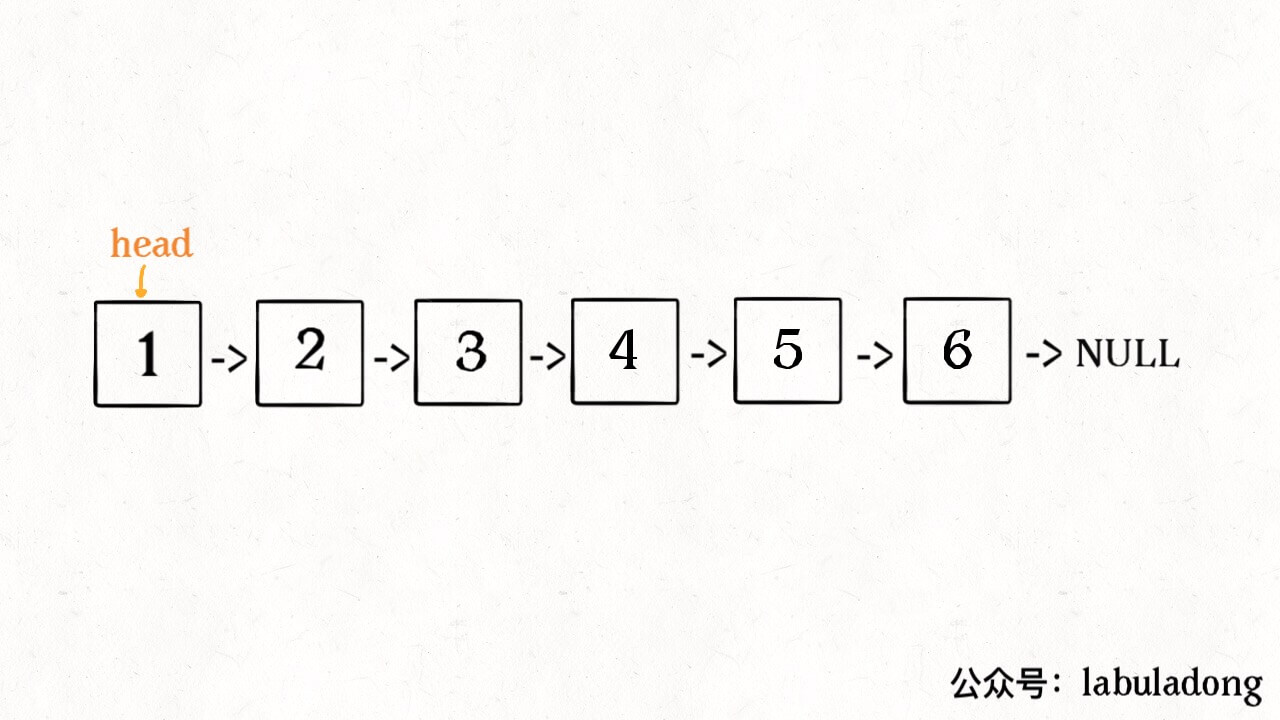
**head.next = null;**

**return last;**

**}**

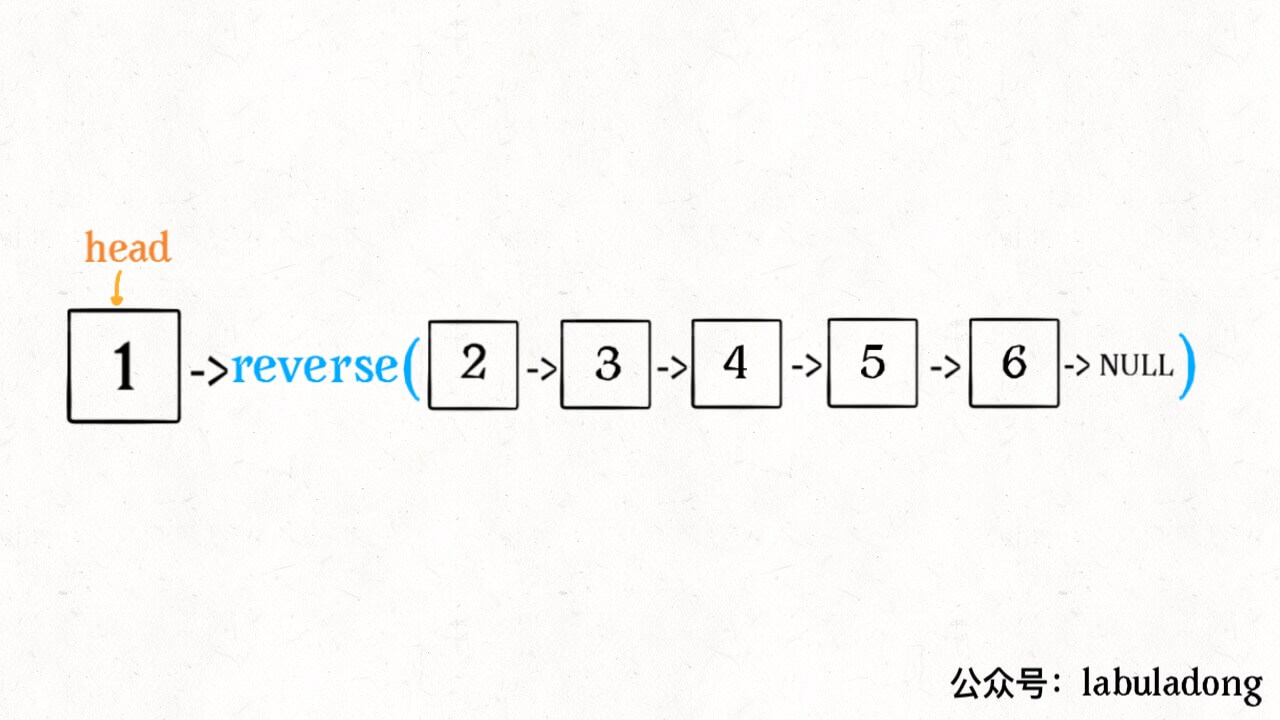
递归算法，最重要的就是明确递归函数的定义。reverse函数定义:

输入一个节点head，将以head为起点的链表反转，并返回反转之后的头结点。

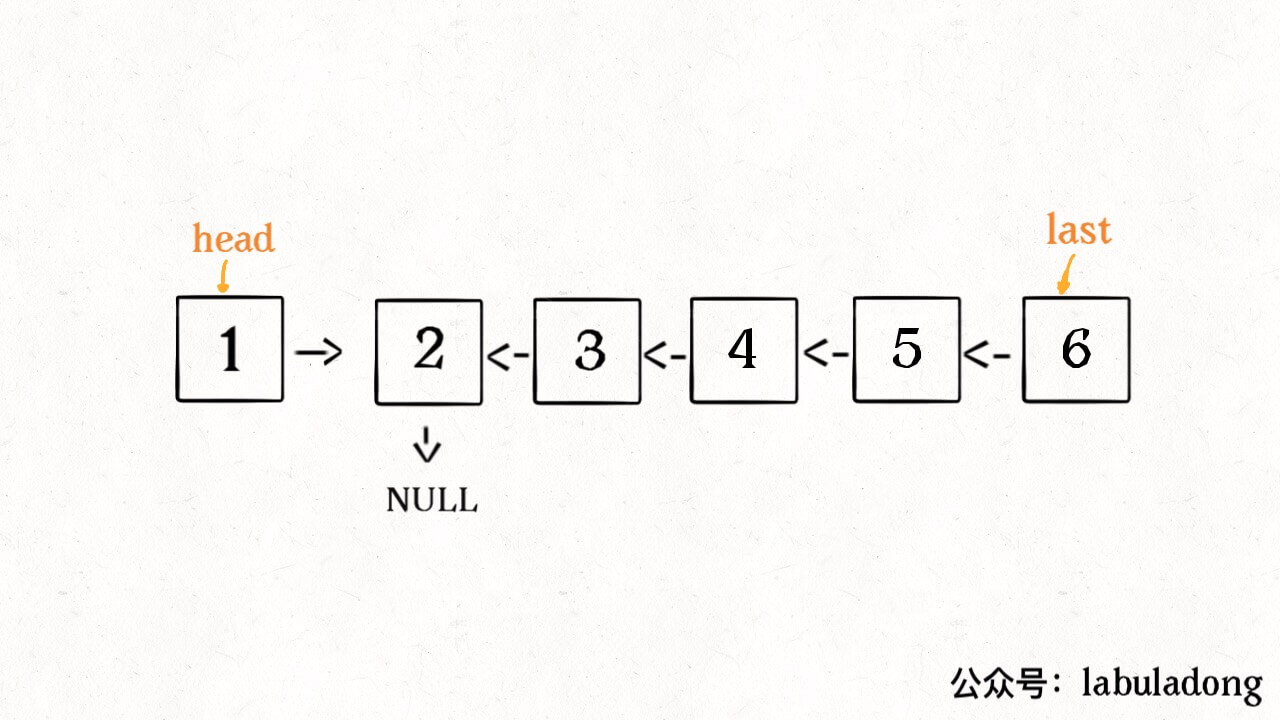


那么输入 reverse(head) 后，会在这里进行递归：

**ListNode last = reverse(head.next);**

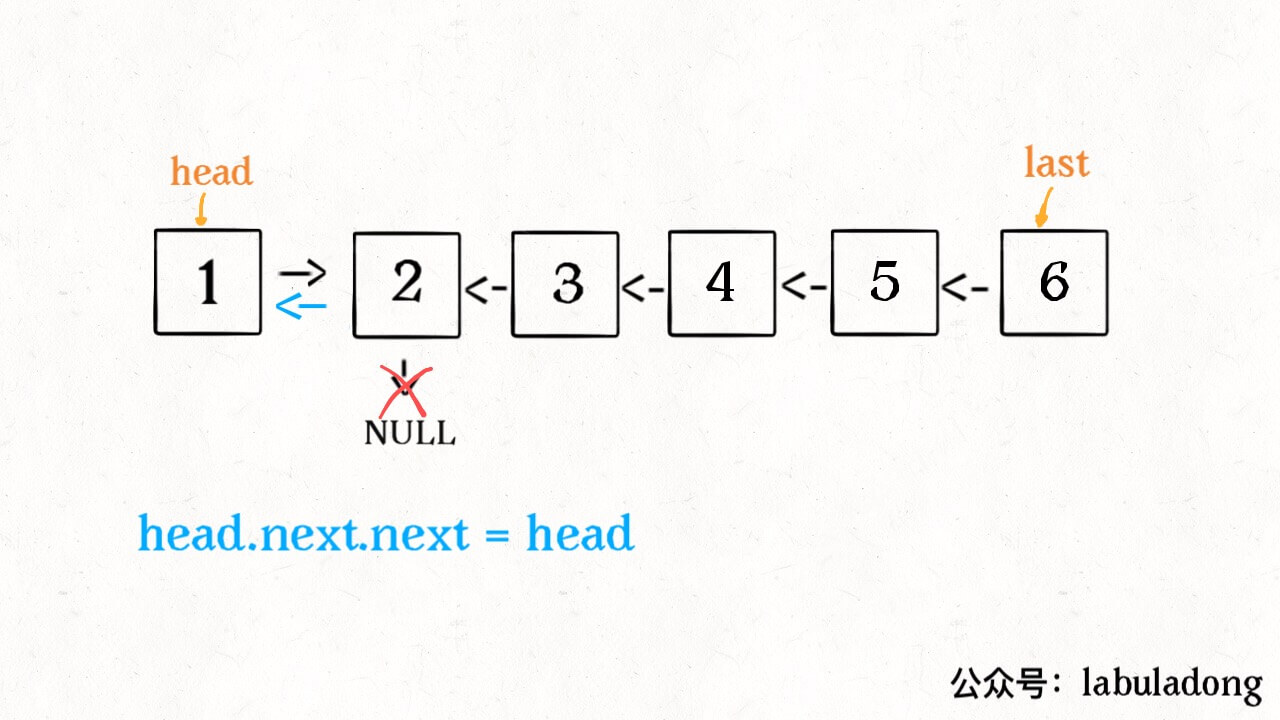


这个 reverse(head.next) 执行完成后，整个链表就成了这样：



反转之后的头结点被last接收，

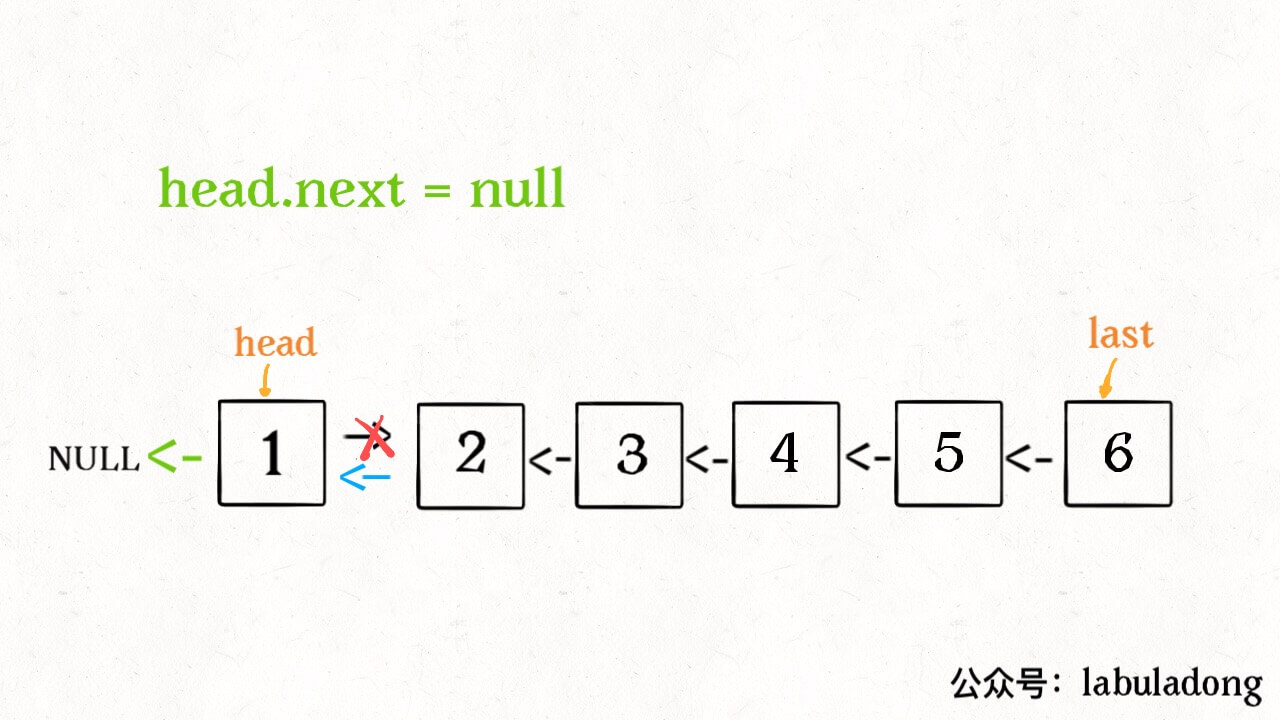
**head.next.next = head;**



接下来

**head.next = null;**

**return last;**



这样链表就反转过来了，但有些地方需要注意

（1）递归函数要有base case ，也就是

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

意思是如果链表只有一个节点的时候反转也是它自己，直接返回即可。

（2）当链表递归反转之后，新的头结点是 last，而之前的 head 变成了最后一个节点，别忘了链表的末尾要指向 null：

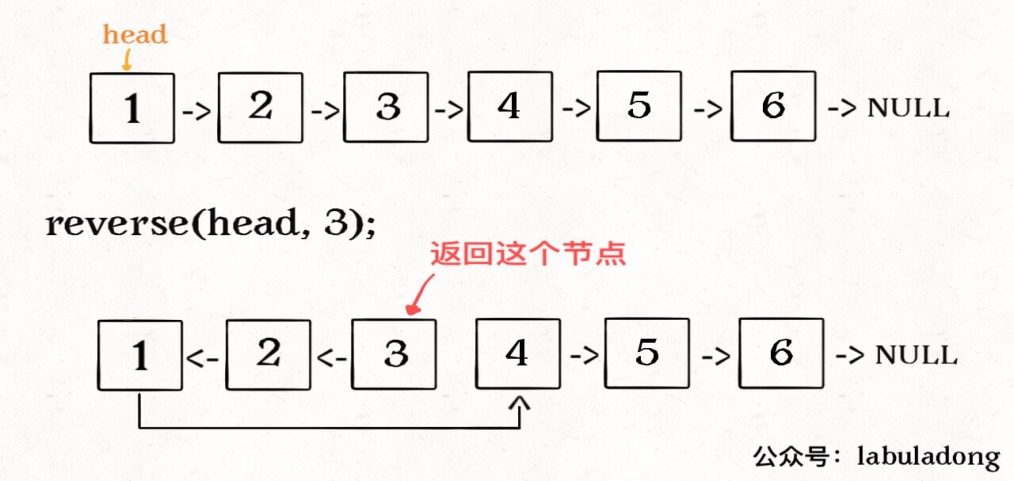
**head.next = null;**

## 2.2反转链表前N个节点

// 将链表的前 n 个节点反转（n <= 链表长度）

ListNode reverseN(ListNode head, int n)

比如说对于下图链表，执行 reverseN(head, 3)



解决思路和前面反转整个链表差不多，

**ListNode successor = null; // 后驱节点**

**​**

**// 反转以 head 为起点的 n 个节点，返回新的头结点**

**ListNode reverseN(ListNode head, int n) {**

**if (n == 1) {**

**// 记录第 n + 1 个节点**

**successor = head.next;**

**return head;**

**}**

**// 以 head.next 为起点，需要反转前 n - 1 个节点**

**ListNode last = reverseN(head.next, n - 1);**

**​**

**head.next.next = head;**

**// 让反转之后的 head 节点和后面的节点连起来**

**head.next = successor;**

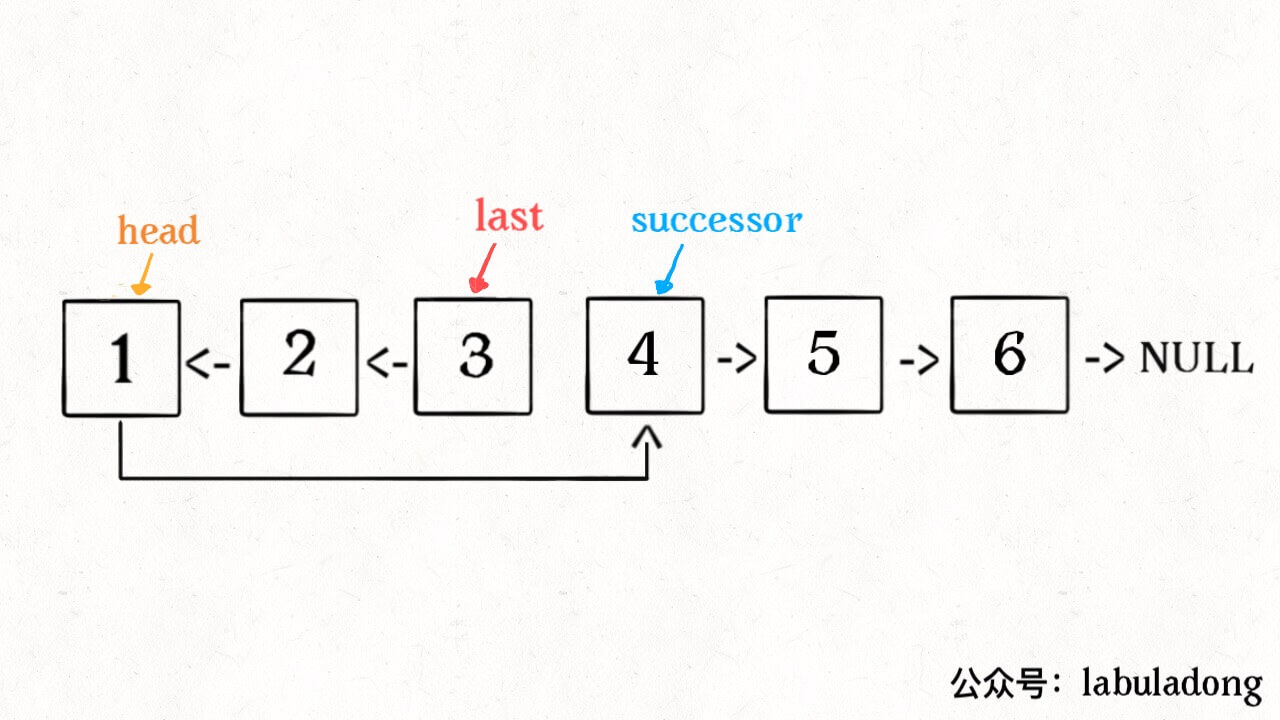
**return last;**

**}**

区别:

（1）base case 变为 n == 1，反转一个元素，就是它本身，同时**要记录后驱节点**。

（2）刚才我们直接把 head.next 设置为 null，因为整个链表反转后原来的 head 变成了整个链表的最后一个节点。但现在 head 节点在递归反转之后不一定是最后一个节点了，所以要记录后驱 successor（第 n + 1 个节点），反转之后将 head 连接上。



## 2.3反转链表的一部分

现在解决我们最开始提出的问题，给一个索引区间 [m,n]（索引从 1 开始），仅仅反转区间中的链表元素。

**ListNode reverseBetween(ListNode head, int m, int n)**

首先，如果 m == 1，就相当于反转链表开头的 n 个元素嘛，也就是我们刚才实现的功能：

**ListNode reverseBetween(ListNode head, int m, int n) {**

**// base case**

**if (m == 1) {**

**// 相当于反转前 n 个元素**

**return reverseN(head, n);**

**}**

**// ...**

**}**

如果 m != 1 怎么办？如果我们把 head 的索引视为 1，那么我们是想从第 m 个元素开始反转对吧；如果把 head.next 的索引视为 1 呢？那么相对于 head.next，反转的区间应该是从第 m - 1 个元素开始的；那么对于 head.next.next 呢……

**ListNode reverseBetween(ListNode head, int m, int n) {**

**// base case**

**if (m == 1) {**

**return reverseN(head, n);**

**}**

**// 前进到反转的起点触发 base case**

**head.next = reverseBetween(head.next, m - 1, n - 1);**

**return head;**

**}**

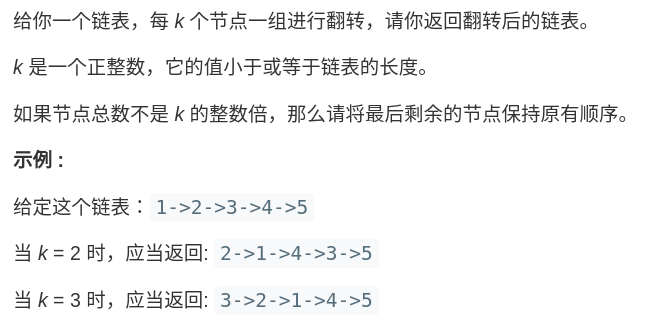
## 2.4反转链表总结

递归不容易理解，原则是不要跳进递归，利用明确的定义来实现算法逻辑。

递归操作链表并不高效，时间复杂度都是O（N）,但空间复杂度是O(N)。考虑效率还是要使用迭代算法更好。

## 2.5解决K个一组反转表

先看题目：

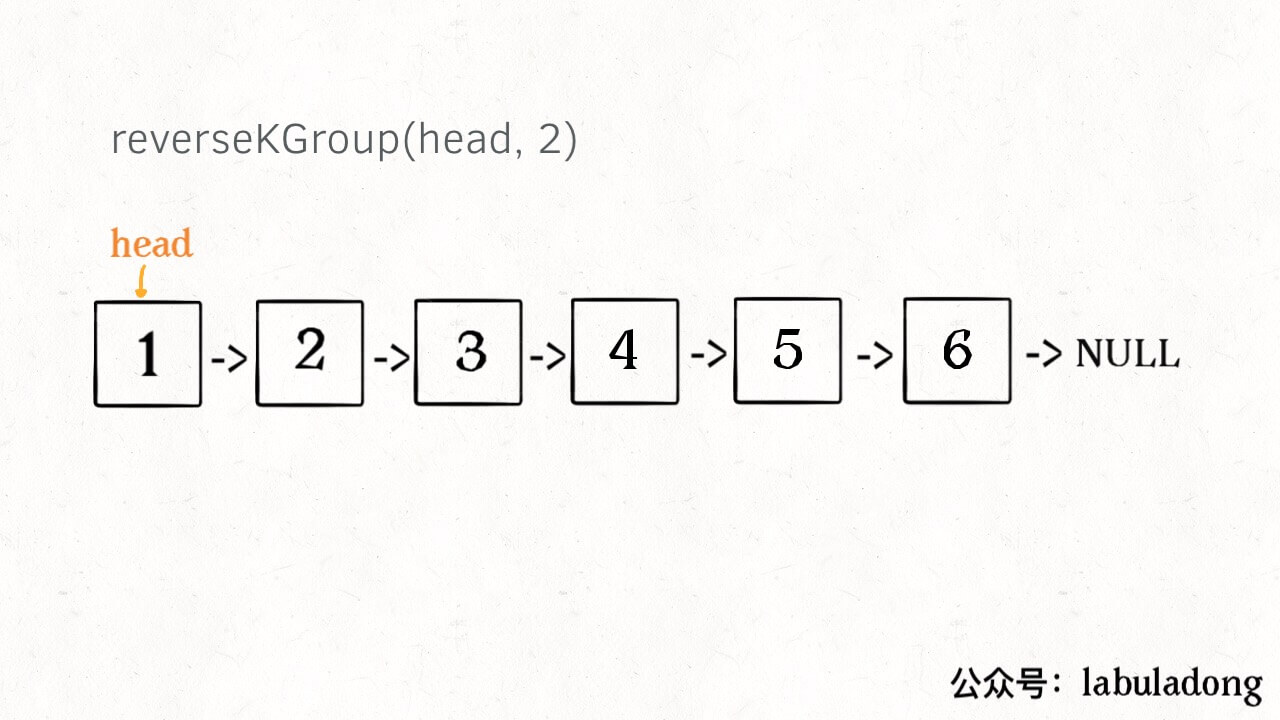


### 2.5.1分析问题

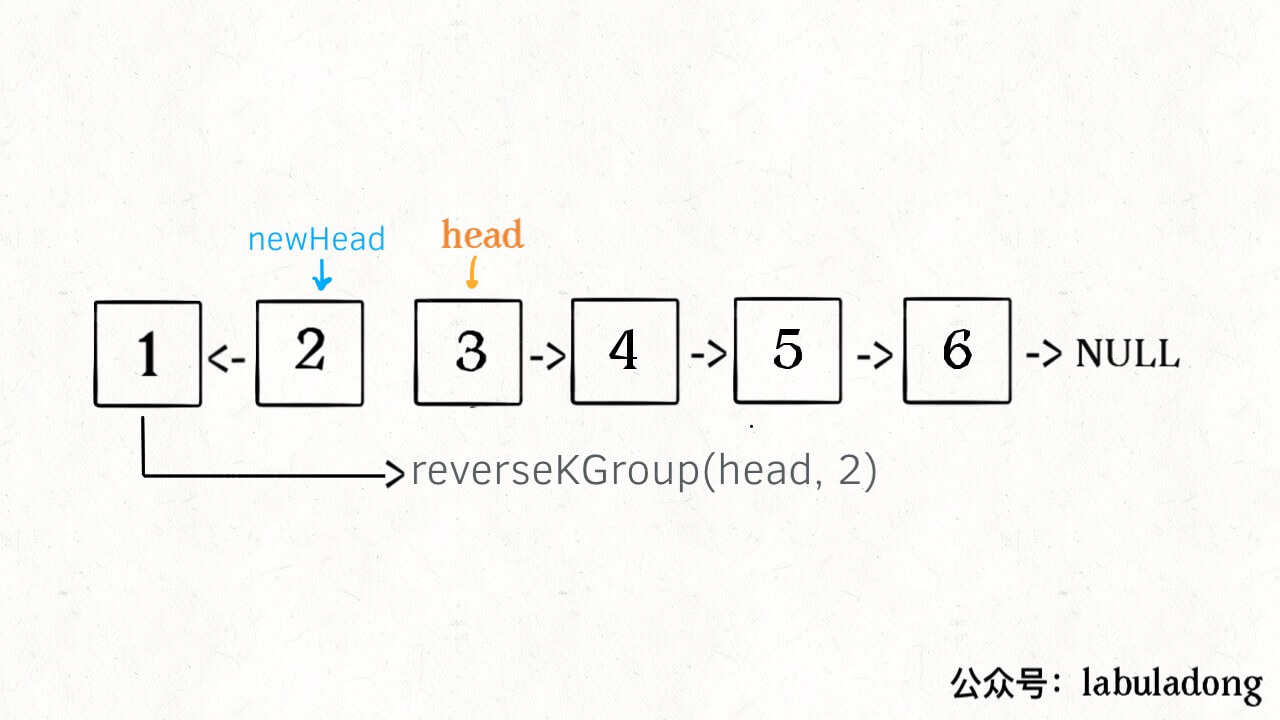
链表是一种兼具递归和迭代性质的数据结构。

那么什么是递归特质？

我们给这个链表调用reverseKGroup(head, 2)，即以 2 个节点为一组反转链表：



如果我设法把前 2 个节点反转，那么后面的那些节点怎么处理？后面的这些节点也是一条链表，而且规模（长度）比原来这条链表小，这就叫**子问题**。



我们可以直接递归调用 reverseKGroup(cur, 2)，因为子问题和原问题的结构完全相同，这就是所谓的递归性质。

那么我们那就可以得到大致的算法流程:

（1）**先反转以 head 开头的 k 个元素**。

（2）**将第 k + 1 个元素作为 head 递归调用 reverseKGroup 函数**。

（3）**将上述两个过程的结果连接起来**。

最后一点值得注意的是，递归函数都有个 base case，对于这个问题是什么呢？

题目说了，如果最后的元素不足 k 个，就保持不变。这就是 base case，待会会在代码里体现。

### 2.5.2代码实现

先实现反转整个链表

**// 反转以 a 为头结点的链表**

**ListNode reverse(ListNode a) {**

**ListNode pre, cur, nxt;**

**pre = null; cur = a; nxt = a;**

**while (cur != null) {**

**nxt = cur.next;**

**// 逐个结点反转**

**cur.next = pre;**

**// 更新指针位置**

**pre = cur;**

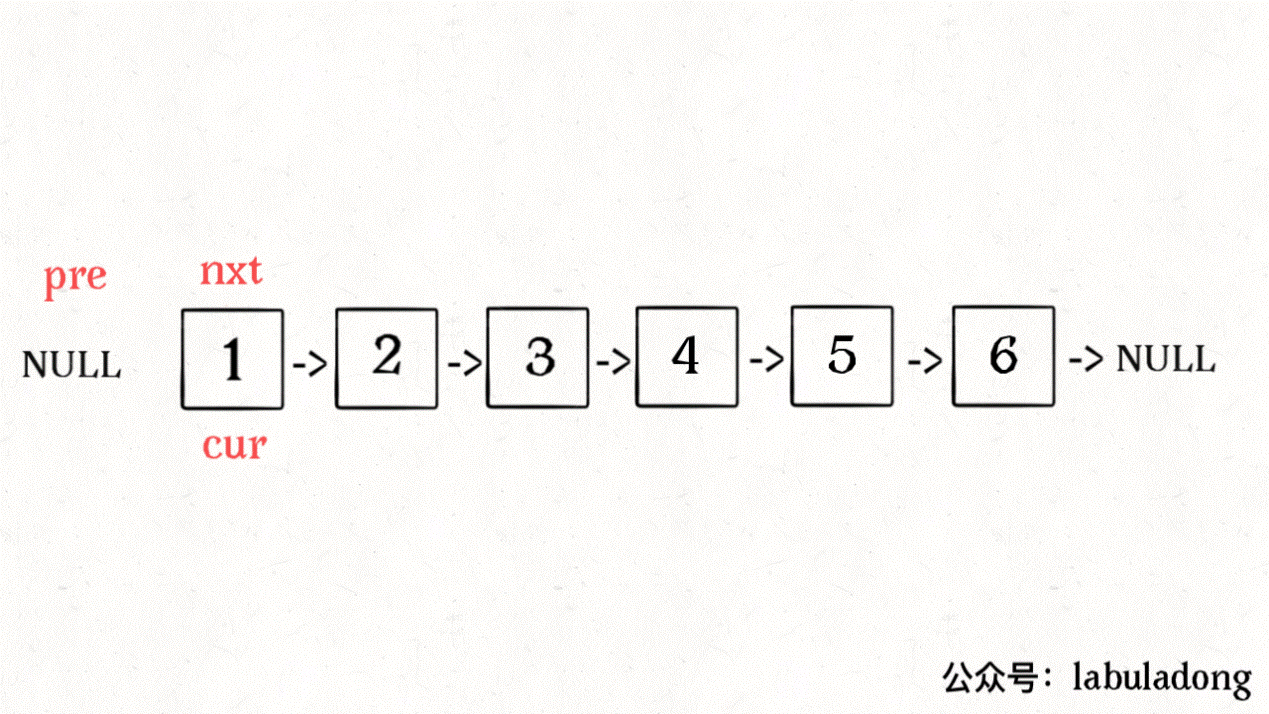
**cur = nxt;**

**}**

**// 返回反转后的头结点**

**return pre;**

**}**



「反转以 a 为头结点的链表」其实就是「反转 a 到 null 之间的结点」，那么如果让你「反转 a 到 b 之间的结点」，你会不会？

只要更改函数签名，并把上面的代码中 null 改成 b 即可：

**/\*\* 反转区间 [a, b) 的元素，注意是左闭右开 \*/**

**ListNode reverse(ListNode a, ListNode b) {**

**ListNode pre, cur, nxt;**

**pre = null; cur = a; nxt = a;**

**while (cur != b) { // while 终止的条件改一下就行了**

**nxt = cur.next;**

**cur.next = pre;**

**pre = cur;**

**cur = nxt;**

**}**

**// 返回反转后的头结点**

**return pre;**

**}**

现在我们迭代实现了反转部分链表的功能，接下来就按照之前的逻辑编写 reverseKGroup 函数即可：

**ListNode reverseKGroup(ListNode head, int k) {**

**if (head == null) return null;**

**// 区间 [a, b) 包含 k 个待反转元素**

**ListNode a, b;**

**a = b = head;**

**for (int i = 0; i < k; i++) {**

**// 不足 k 个，不需要反转，base case**

**if (b == null) return head;**

**b = b.next;**

**}**

**// 反转前 k 个元素**

**ListNode newHead = reverse(a, b);**

**// 递归反转后续链表并连接起来**

**a.next = reverseKGroup(b, k);**

**return newHead;**

**}**

## 2.6判断回文链表

寻找回文串的核心思想是从中心想两端扩展：

**string palindrome(string s, int l, int r) {**

**// 防止索引越界**

**while (l >= 0 && r < s.size()**

**&& s[l] == s[r]) {**

**// 向两边展开**

**l--; r++;**

**}**

**// 返回以 s[l] 和 s[r] 为中心的最长回文串**

**return s.substr(l + 1, r - l - 1);**

**}**

判断字符串是不是回文串，只需要双指针技巧，从两端向中间逼近

**bool isPalindrome(string s) {**

**int left = 0, right = s.length - 1;**

**while (left < right) {**

**if (s[left] != s[right])**

**return false;**

**left++; right--;**

**}**

**return true;**

**}**

### 2.6.1判断回文单链表

**/\*\***

**\* 单链表节点的定义：**

**\* public class ListNode {**

**\* int val;**

**\* ListNode next;**

**\* }**

**\*/**

**boolean isPalindrome(ListNode head);**

**​**

**输入: 1->2->null**

**输出: false**

**输入: 1->2->2->1->null**

**输出: true**

单链表无法倒着遍历，无法使用双指针技巧。那么最简单的办法就是，把原始链表反转存入一条新的链表，然后比较这两条链表是否相同

链表兼具递归结构，树结构不过是链表的衍生。那么，**链表其实也可以有前序遍历和后序遍历**：

模仿双指针实现回文判断的功能：

// 左侧指针

**ListNode left;**

**boolean isPalindrome(ListNode head) {**

**left = head;**

**return traverse(head);**

**}**

**boolean traverse(ListNode right) {**

**if (right == null) return true;**

**boolean res = traverse(right.next);**

**// 后序遍历代码**

**res = res && (right.val == left.val);**

**left = left.next;**

**return res;**

**}**

这样做的核心逻辑就是把链表节点放入一个栈，然后再拿出来，这时候元素顺序就是反的。

当然，无论造一条反转链表还是利用后续遍历，算法的时间和空间复杂度都是 O(N)。下面我们想想，能不能不用额外的空间，解决这个问题呢？

### 2.6.2优化空间复杂度

（1）先通过双指针技巧中的快慢指针找到链表的中点

**ListNode slow, fast;**

**slow = fast = head;**

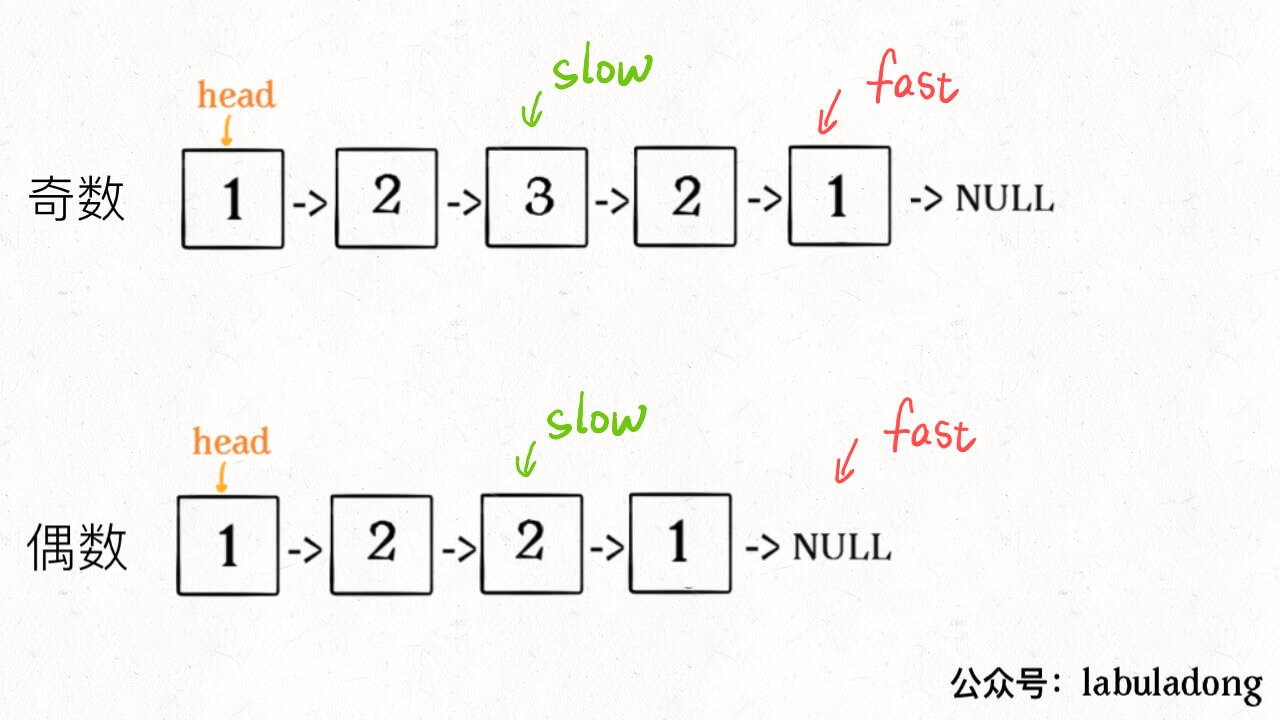
**while (fast != null && fast.next != null) {**

**slow = slow.next;**

**fast = fast.next.next;**

**}**

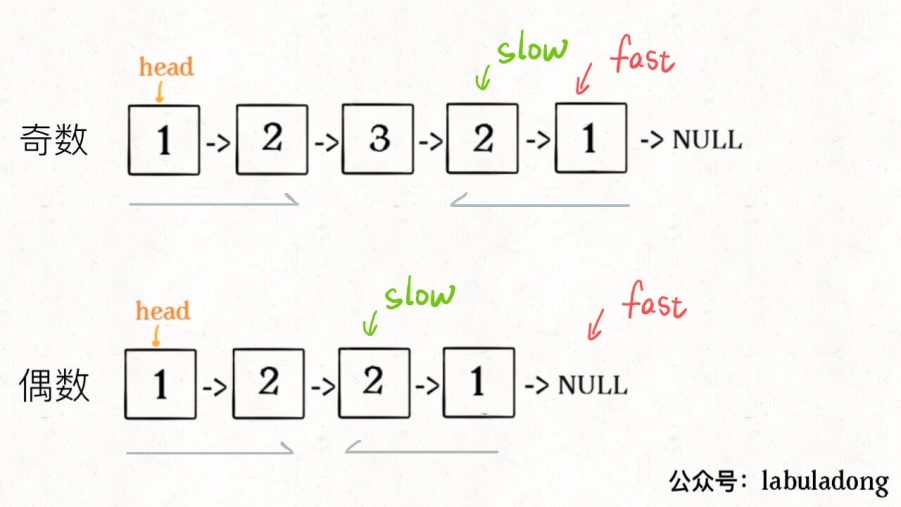
**// slow 指针现在指向链表中点**



（2）**如果fast指针没有指向null，说明链表长度为奇数，slow还要再前进一步**：

**if (fast != null)**

**slow = slow.next;**



（3）**从slow开始反转后面的链表，现在就可以开始比较回文串了**：

**ListNode left = head;**

**ListNode right = reverse(slow);**

**​**

**while (right != null) {**

**if (left.val != right.val)**

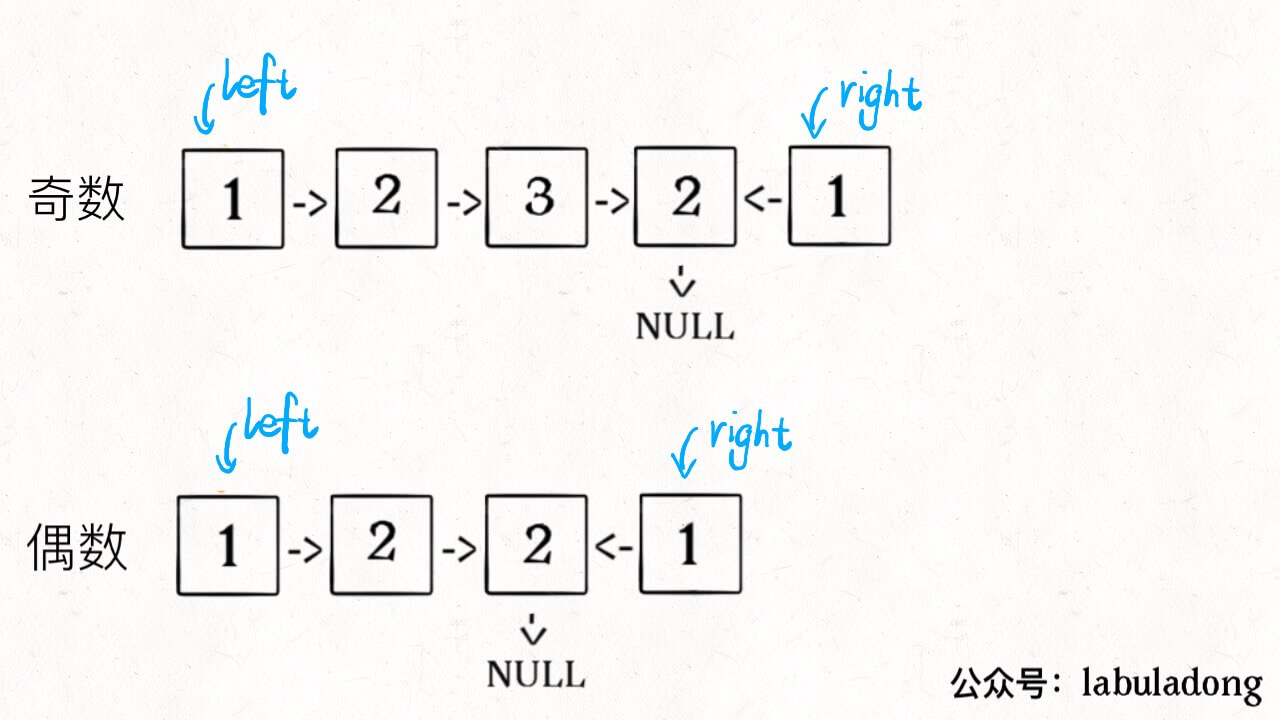
**return false;**

**left = left.next;**

**right = right.next;**

**}**

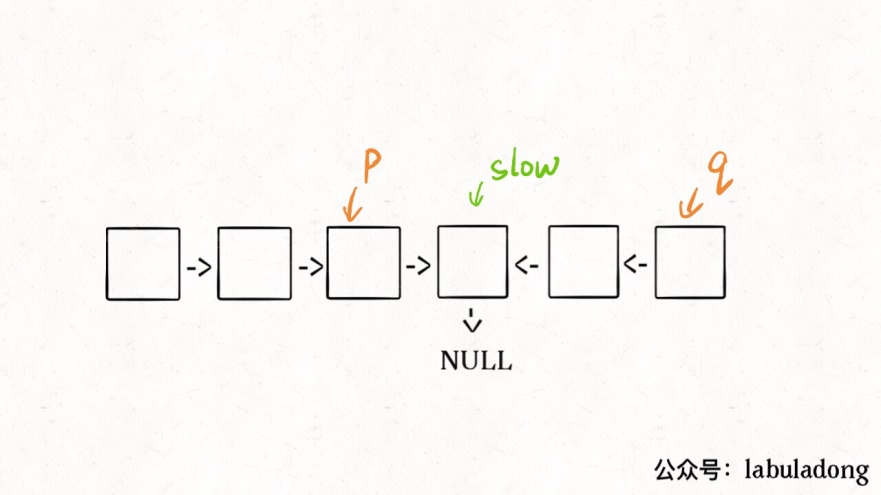
**return true;**



算法总体的时间复杂度 O(N)，空间复杂度 O(1)，已经是最优的了。

我知道肯定有读者会问：这种解法虽然高效，但破坏了输入链表的原始结构，能不能避免这个瑕疵呢？

其实这个问题很好解决，关键在于得到p, q这两个指针位置：



这样，只要在函数 return 之前加一段代码即可恢复原先链表顺序：

**p.next = reverse(q);**

# 3. 手刷二叉树题目

## 3.1二叉树的认知

### 3.1.1二叉树的重要性

快速排序是二叉树的前序遍历，归并排序就是二叉树的后续遍历。

### 3.1.2写递归算法的秘诀

递归算法的关键是要明确函数的定义是什么，然后相信这个定义，利用这个定义推导最终结果，不要跳入递归的细节。

计算一棵二叉树共有多少节点:

**// 定义：count(root) 返回以 root 为根的树有多少节点**

**int count(TreeNode root) {**

**// base case**

**if (root == null) return 0;**

**// 自己加上子树的节点数就是整棵树的节点数**

**return 1 + count(root.left) + count(root.right);**

**}**

写树相关的算法，简单说就是要搞清楚点钱root节点该做什么，然后根据函数定义递归调用子节点，递归调用会让孩子节点做相同的事情。

### 3.1.3写递归算法实践

（1）把二叉树镜像反转

**// 将整棵树反转**

**TreeNode invertTree(TreeNode root) {**

**// base case**

**if (root == null) return null;**

**// 前序遍历位置**

**//root节点需要交换它的左右子节点**

**TreeNode temp =root.left;**

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

**root.right=temp;**

**//让左右子节点继续翻转他们的子节点**

**invertTree(root.left);**

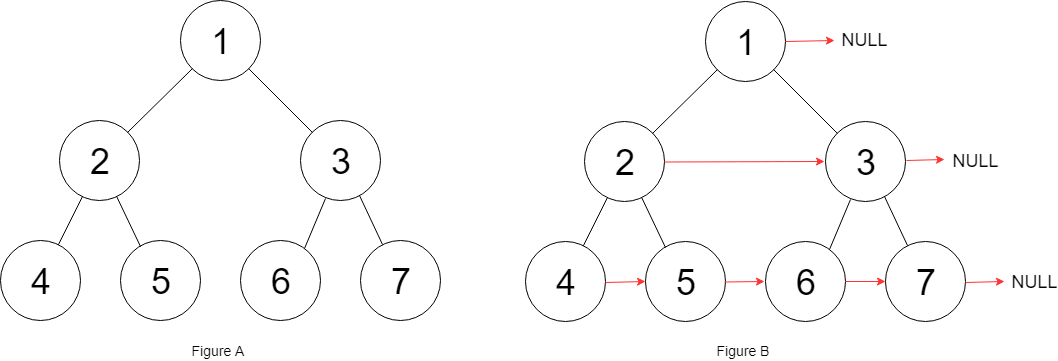
**invertTree(root.right);**

**return root;}**

二叉树的一个难点就是，如何把题目的要求细化成每一个节点需要做的事情

（2）填充二叉树节点的右侧指针

题目意思就是二叉树每一层节点都用next指针连接起来



这个题的难点是如何解决跨父节点依赖。那么一个节点参数肯定是做不到的，我们就用两个节点将其连接起来。

//主函数

**Node connect(){**

**if(root==null) return null;**

**connectTwoNode(root.left,root.right);**

**return root;**

**}**

**//辅助函数**

**void connectTwoNode(Node node1,Node node2){**

**if(node1==null||node2==null){**

**return;**

**}**

**node1.next=node2;**

**// 连接相同父节点的两个子节点**

**connectTwoNode(node1.left, node1.right);**

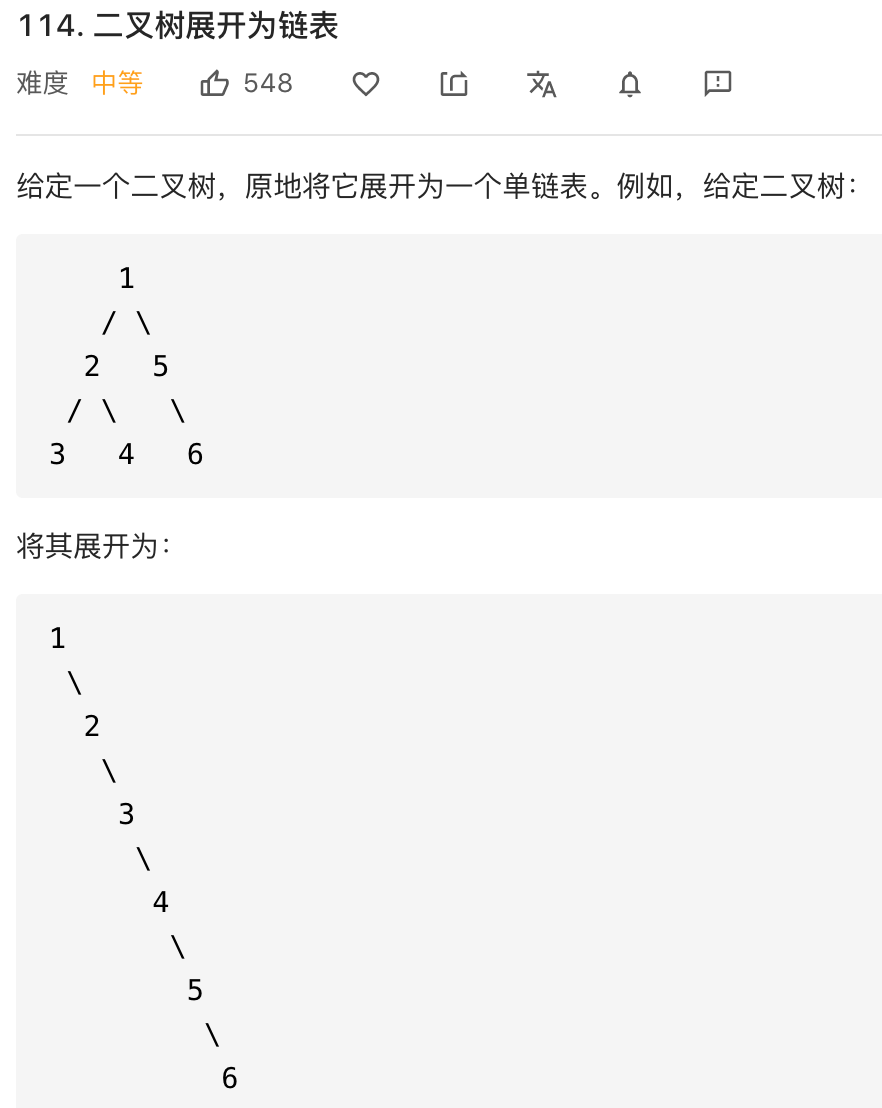
**connectTwoNode(node2.left, node2.right);**

**// 连接跨越父节点的两个子节点**

**connectTwoNode(node1.right, node2.left);**

**}**

（3）把二叉树展开为链表



void flatten(TreeNode root);

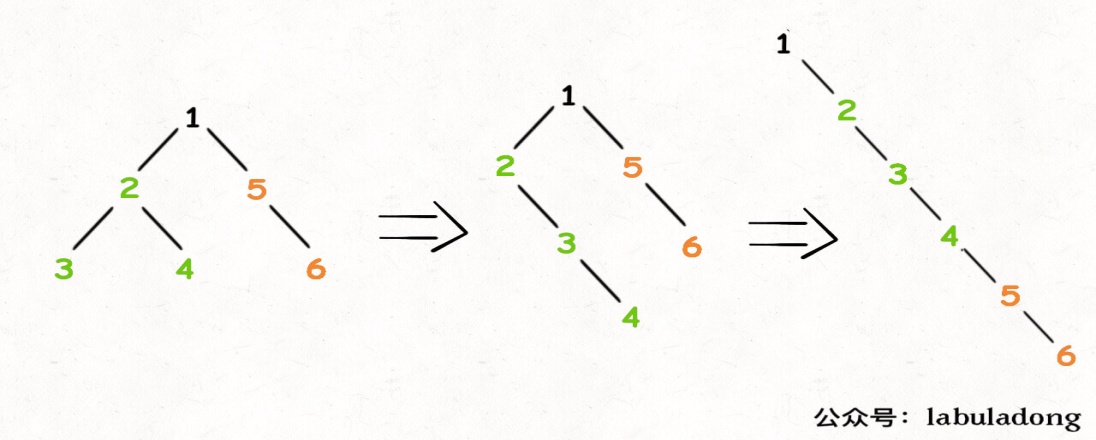
我们尝试给出这个函数的定义：

**给 flatten 函数输入一个节点 root，那么以 root 为根的二叉树就会被拉平为一条链表**。

我们再梳理一下，如何按题目要求把一棵树拉平成一条链表？很简单，以下流程：

1、将 root 的左子树和右子树拉平。

2、将 root 的右子树接到左子树下方，然后将整个左子树作为右子树。

****

**void flatten(TreeNode root){**

**if(root==null) return;**

**flatten(root.left);**

**flatten(root.rght);**

**/\*后续遍历位置\*/**

**//左右字数已经被拉平成一条链表**

**TreeNode left=root.left;**

**TreeNode right=root.right;**

**//左子树作为右子树**

**root.left=null;**

**root.right=left;**

**//将原先的右子树接到当前右子树的末端**

**TreeNode p=root;**

**while(p.right!=null){**

**p=p.right;**

**}**

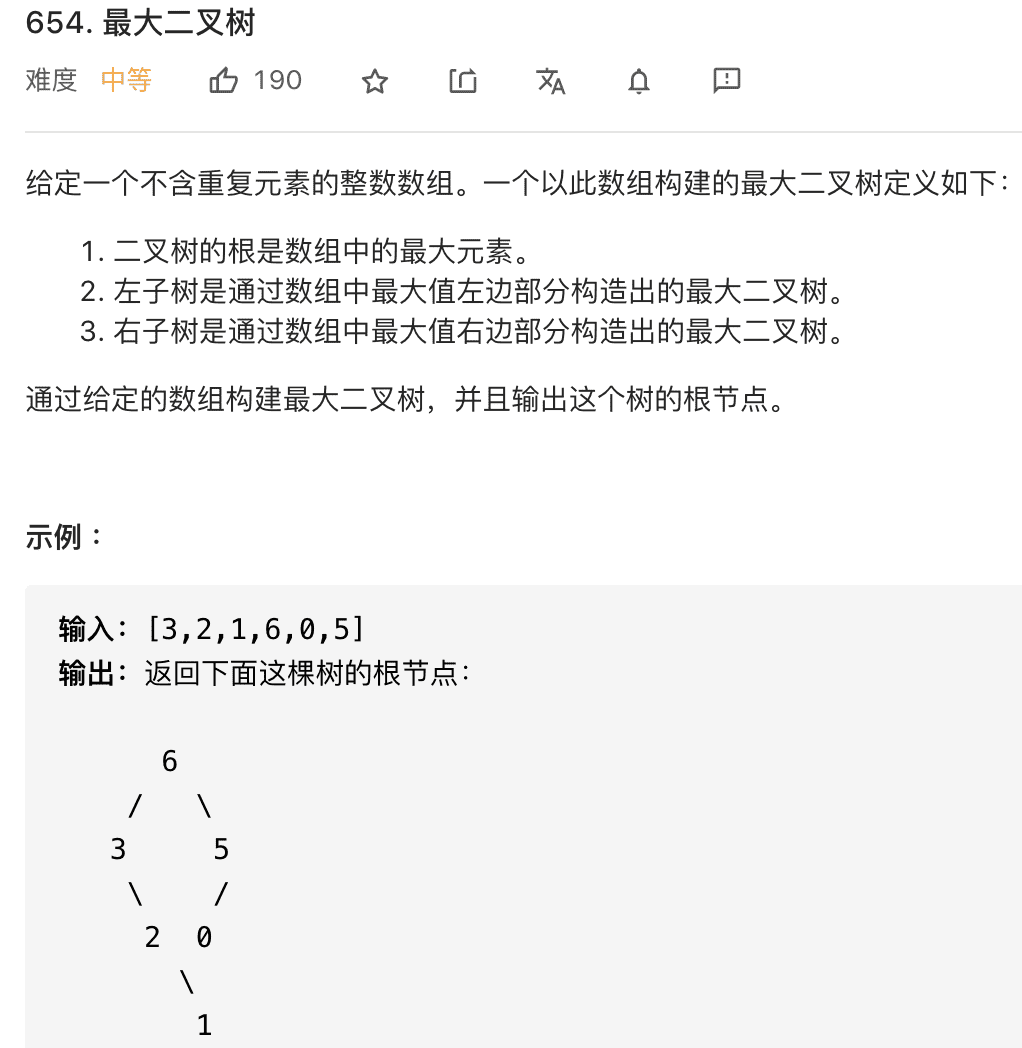
**p.right=right;**

**}**

**递归的魅力在于，你也不知道flatten是如何把左右子树拉平的，但是只要只要它的定义如此，然后相信这个定义，让root做它该做的事情，flatten函数就会按照定义工作。**

## 3.2二叉树算法到底怎么做

### 3.2.1构造最大二叉树

****

**对于每个根节点，只需要找到当前 nums 中的最大值和对应的索引，然后递归调用左右数组构造左右子树即可**。

**/\* 主函数 \*/**

**TreeNode constructMaximumBinaryTree(int[] nums) {**

**return build(nums, 0, nums.length - 1);**

**}**

**​**

**/\* 将 nums[lo..hi] 构造成符合条件的树，返回根节点 \*/**

**TreeNode build(int[] nums, int lo, int hi) {**

**// base case**

**if (lo > hi) {**

**return null;**

**}**

**​**

**// 找到数组中的最大值和对应的索引**

**int index = -1, maxVal = Integer.MIN\_VALUE;**

**for (int i = lo; i <= hi; i++) {**

**if (maxVal < nums[i]) {**

**index = i;**

**maxVal = nums[i];**

**}**

**}**

**​**

**TreeNode root = new TreeNode(maxVal);**

**// 递归调用构造左右子树**

**root.left = build(nums, lo, index - 1);**

**root.right = build(nums, index + 1, hi);**

**return root;**

**}**

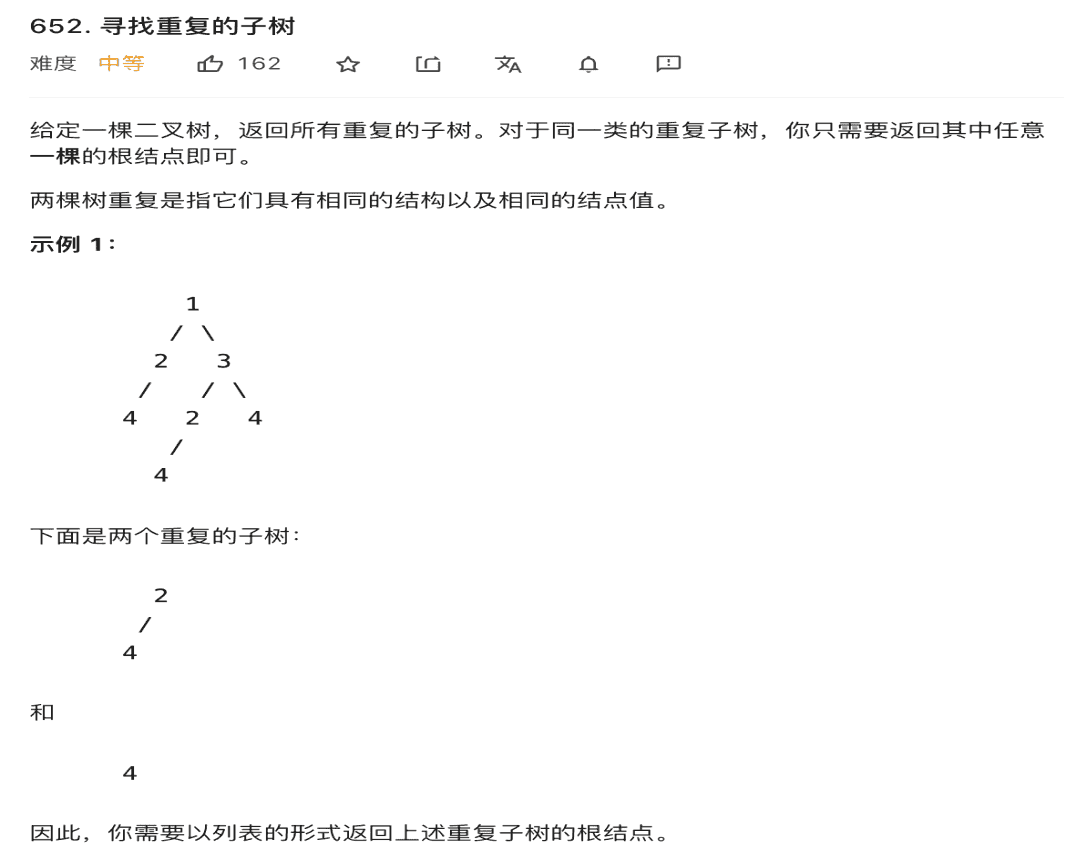
### 3.2.2构造二叉树

**通过前序和中序/后序和中序遍历结果构造二叉树**

## 3.3寻找重复子树

我们学习二叉树框架思维后，还是不太清楚如何用前序还是中序还是后序来遍历。

看题，力扣652，寻找重复子树



输入是一棵二叉树的根节点 root，返回的是一个列表，里面装着若干个二叉树节点，这些节点对应的子树在原二叉树中是存在重复的。

如果想要知道以自己为根的子树是不是重复的，你需要知道什么信息？  
1、以我为根的二叉树长啥样？

2、以其他节点为根的子树长啥样？

要想知道自己长啥样，得先知道自己左右子树长啥样，在加上自己。这就是标准的后续遍历。

二叉树序列化可以通过拼接字符串的方式来进行

**String traverse(TreeNode root) {**

**// 对于空节点，可以用一个特殊字符表示**

**if (root == null) {**

**return "#";**

**}**

**// 将左右子树序列化成字符串**

**String left = traverse(root.left);**

**String right = traverse(root.right);**

**/\* 后序遍历代码位置 \*/**

**// 左右子树加上自己，就是以自己为根的二叉树序列化结果**

**String subTree = left + "," + right + "," + root.val;**

**return subTree;**

**}**

**特殊字符#表示空指针，字符，分隔每个二叉树节点值。**

**知道自己长啥样，怎么知道别人长啥样？**

**HashSet<String> memo = new HashSet<>();**

**LinkedList<TreeNode> res =new LinkedList<>();**