链表(BM1-BM16)

1. 翻转全部 ⭐⭐⭐

- 栈：O(n) ; O(n)
- 原地迁移：O(n); O(1)
  - 新建头节点 null
  - temp 节点指向cur.next
  - cur.next 指向新建头
  - 新建头更新至cur
  - cur 更新至temp
- 递归：O(n); O(n) 递归的空间复杂度需要考虑递归栈的栈空间
  - 返回条件：head==null 或 head.next == null
  - 用next节点记录head.next
  - 递归反转head.next
  - next.next 设为head
  - head.next 设为null

2. 翻转区间

- 先移动m
- 用1的方法翻转n次

3. 每k个翻转

- 递归：
  - 找到下一组的开头tail
  - 在cur 不等于tail之前，一路反转
  - 本轮头节点的next 设为ReverseKGroup(tail, k)

4. 合并两个已排序的链表 ⭐

- 归并

5. 合并k 个已排序的链表 ⭐⭐⭐

- 最小堆归并
- 堆内存n个待合并链表的头节点

6. 判断是否有环 ⭐⭐⭐

- 快慢双指针：
  - slow指针每次前进一个，fast指针每次前进两个
    - 若相遇，说明有环
    - 若没有相遇而是fast 为null，说明没有环

7. 环的入口节点 ⭐

- 6中，快慢指针相遇处一定在环内
- slow 从相遇节点出发

- fast 重新从头出发
- 依然按照快2慢1的步调同时出发，再次相遇时即为环入口处

8. 倒数最后k个节点 ⭐⭐⭐

- 双指针：O(n); O(1)
  - fast 先走k步
  - 随后双指针同步向后

9. 删除链表的倒数第n个节点

- 双指针：O(n), O(1)
  - 按照8 找到倒数第n个节点
  - pre.next = pre.next.next

10. 两个链表的第一个公共节点

- 双指针：令二者走相同的路径
  - 同步推进
  - 当其中一指针为null 时，将其指向另一链表的头

11. 链表相加 ⭐⭐

- 先反转，再从个位开始相加

12. 单链表排序 ⭐⭐⭐

- **辅助数组：T = O(NlogN); S = O(N)**
- **原地分治排序：T = O(NlogN); S = O(1)**
  - 快慢指针，slow，fast将链表分为两半
  - left = sortInList(head); right = sortInList(slow.next);
  - 将left 和 right **归并排序（11）**

13. 判断链表是否回文

- p1 正向遍历
- p2 遍历反转链表

14. 奇偶重排 ⭐⭐

- 双指针
  - odd = head, even = head.next
  - while (even != null && even.next != null)
    - odd.next = even.next
    - odd = odd.next
    - even.next = odd.next
    - even = even.next

15. 删除重复-I & 留一个

- while (cur.next != null)
  - if (cur.val == cur.next.val)
    - cur.next = cur.next.next

    - else
        - cur = cur.next

16. 删除重复-II & 一个不留

- dummy.next = head;
- cur = dummy
- while (cur.next != null && cur.next.next != null)
    - if (cur.next.val == cur.next.next.val)
        - while()
            - del
    - else
        - cur = cur.next

二分查找(BM17-BM22)

1. 升序数组二分查找

- T=O(logN), S=O(logN)

2. 二维数组二分查找 ⭐⭐

- 入口在左下角或右上角，每次i 或j 只变一个，变幅为1
    - 从左下角开始：向右变大，向上变小
    - 从右上角开始：向左变小，向下变大
- T=O(m+n); S=O(1)

3. 寻找峰值(数组爬坡) ⭐⭐

- mid = (left + right) / 2
    - if (nums[mid] < nums[right]) 仍在爬坡，且必有nums[right] > nums[mid]，left = mid + 1;
    - (nums[mid] > nums[right]) mid 已经跨过峰值，right = mid
- T=O(logN); S=O(1)

4. 数组中逆序数对 ⭐⭐⭐

- 暴力检索 T=O(n^2)
- 归并排序 T=O(nlogn), S=O(n)
    - 利用归并排序中两个待合并数组升序的特征，在归并步骤中对比 array[left] 和array[right]
        - 若array[left] > array[right]，则array[right] 小于左侧数组中所有后续元素
        - 反之无事发生

5. 旋转非降序数组(前一半整体挪到后一半之后)的**最小数字** ⭐⭐⭐

- 二分查找 T=O(logN), S=O(1)
    - 对比 array[mid] 和array[right]
        - array[mid] == array[right]：rifght --
        - else-if array[mid] > array[right]：left = mid + 1

    - else：right = mid

6. 对比版本号

- split函数拆分，长度不足时以“0”补齐

二叉树(BM23-BM41)

1. 前序 ⭐⭐
    a. 递归 - T=O(n), S=O(n)
    b. 栈辅助循环 - T=O(n), S=O(n)
2. 中序 ⭐⭐
    a. 递归 - T=O(n), S=O(n)
    b. 栈辅助循环 - T=O(n), S=O(n)
        i. 首先前进到最左端，期间root 若不为null，就直接入栈，并root=root.left
3. 后序 ⭐⭐⭐
    a. 递归 - T=O(n), S=O(n)
    b. 栈辅助循环 - - T=O(n), S=O(n)
        i. 最左端处理作为基础，同上
        ii. 维护pre 指针，标识上次访问处理的节点，若node右节点为空或前序访问节点就是node的右节点，那可以处理当前node，否则node入栈，node=node.right
4. 层序 ⭐⭐⭐
    a. 队列辅助循环 - T=O(n), S=O(n)
    b. 递归 - T=O(n), S=O(n)
        i. 以深度遍历打底，若当前depth表示处于新的一层，就新建一个数组，存入结果的二维数组
        ii. 若当前depth表示处于曾经访问过的层，就从原本的二维数组中取出对应层的数组
5. Z字形层序遍历 - 层序变形 ⭐
6. 最大深度 ⭐⭐⭐
    a. 递归
    b. 层序遍历
7. 路径求和 ⭐⭐
    a. 递归 - 若为null：返回false；若为叶子节点且当前val等于target，返回true；否则返回func(left) || func(right)，递归时target减去当前val
    b. 栈辅助前序遍历，Stack1辅助遍历；Stack2存储“未来取出该节点时，该节点的路径前序和”
8. 二叉搜索树转为双向链表 ⭐
    a. 递归
        i. 链表头节点dummy；辅助节点cur
        ii. 先递归左子树；处理root：cur.right = root; root.left = cur; cur = cur.right；递归右子树
        iii. 返回前断开dummy.right.left
    b. 栈辅助中序遍历
        i. 因为二叉搜索树中序遍历结果就是有序结果，因此按照遍历顺序依次接到链表尾即可
9. 对称二叉树 ⭐
    a. 递归

        i. 条件1：若left 和right 均为null，返回true
        ii. 条件2：若left、right其一不为空，或left.val != right.val 返回false
        iii. recur(left.left, right.right) && recur(left.right, right.left)
10. 合并二叉树 ⭐ ⭐
    a. 递归
        i. t1 为空 return t2；t2为空return t1
        ii. node = new TreeNode(t1.val + t2.val)
        iii. node.left = merge(t1.left, t2.left); node.right = merge(t1.right, t2.right)
    b. 层序遍历
        i. 三队列：q存结果；q1存树1；q2存树2
11. 二叉树镜像 ⭐
    a. 递归
        i. left = Mirror(right); right=Mirror(left);
    b. 栈辅助中序遍历
        i. op：left, right = right, left
12. 判断二叉搜索树⭐ ⭐ ⭐
    a. 中序输出，判断递增 —— T=O(n); S=O(2n)
    b. 存pre表示上次访问，中序递归 —— T=O(n); S=O(n)
    c. 栈辅助中序
13. 判断完全二叉树 ⭐ ⭐
    a. 层序遍历：只能碰到一次null
        i. 若root == null，标记met 为true，直接continue，从此所有节点只能为null
        ii. 若root != null，且met 为true，返回false
14. 判断平衡二叉树 ⭐ ⭐ ⭐
    a. 递归
        i. 左子树depth - 右子树depth 绝对值大于1，不平衡；否则继续向上返回Max(depthL, depthR)+1
        ii. 剪枝：增加变量isBalanced，若已为false，则提前结束
15. BST 最近公共祖先 ⭐ ⭐ ⭐
    a. 遍历
        i. 分别找两节点从根节点开始的路径，当某次访问节点不同时直接跳出，返回上一次的值
    b. 递归：利用BST性质 —— 如果t1 在左，t2在右，当前节点就是共同祖先；都在左，向左遍历，反之亦然
        i. **所有的大小等好都可以带等**
16. 普通二叉树找公共祖先 ⭐ ⭐
    a. 遍历找路径——同上可共用
    b. 层序遍历，HashMap记录每一个值对应的parent值；当两个目标节点都访问之后，开始判断，利用parentMap 逐步上推
    c. 递归
        i. helper函数——如果root为null，返回null
        ii. 如果root.val == t1/t2，返回root
        iii. 如果left为空，说明t1，t2都在右子树，返回右；反之亦然；如果都不为空，说明分别在两侧，返回当前节点
17. 序列化和重构树 ⭐ ⭐ ⭐

a. 层序遍历，为空添加’ # “；构建时为#则null，不加入队列

18. **根据前序和中序队列重建二叉树** ⭐⭐⭐

a. 前序的**第一个节点即为本次构建的root**

b. 找到该节点在中序队列中的位置，左侧为左子树，右侧为右子树

19. 右视图 ⭐

a. 层序遍历，只存最后一个值