1. CodeExercise：
    1. 简介：
    2. 日常刷题：
        1. 2022.6.21——数组
            1. 26. 删除有序数组中的重复项
            2. 83. 删除排序链表中的重复元素
            3. 27. 移除元素
            4. 283. 移动零
            5. 167. 两数之和 II - 输入有序数组
            6. 344. 反转字符串
            7. 5. 最长回文子串
        2. 2022.6.22——二叉树
            1. 104. 二叉树的最大深度
            2. 144. 二叉树的前序遍历
            3. 543. 二叉树的直径
        3. 2022.6.23——二叉树
            1. 100. 相同的树
            2. 111. 二叉树的最小深度
            3. 637. 二叉树的层平均值
            4. 102. 二叉树的层序遍历
            5. 107. 二叉树的层序遍历 II
            6. 103. 二叉树的锯齿形层序遍历
        4. 2022.6.24——链表
            1. 21. 合并两个有序链表
            2. 86. 分隔链表
            3. 19. 删除链表的倒数第 N 个结点
            4. 876. 链表的中间结点
        5. 2022.6.25——链表
            1. 206. 反转链表
            2. 92. 反转链表 II
    3. 学习笔记
        1. 2022.6.23——二叉树笔记
            1. 1.二叉树遍历框架
            2. 2.数组&链表的二叉树遍历法
            3. 3.层序遍历
        2. 2022.6.24——二叉堆学习
            1. 1.二叉堆概览

CodeExercise：

简介：

记录从大一暑假开始的刷题之旅。

PS：这个代码库保存了我从大一入学所刷的所有算法题，只是从2022年6月21日开始整理。

日常刷题：

2022.6.21——数组

周二——雷阵雨转阴——35℃/22℃

今天开始刷LeetCode中的《初涉算法》篇，先从数组开始：

26. 删除有序数组中的重复项

类型： 数组 双指针

```
class Solution {
public:
    int removeDuplicates(vector<int>& nums) {
        int fast = 0,slow = 0;
        while(fast<nums.size()) {
            if(nums[fast++]!=nums[slow]) {
                nums[++slow] = nums[fast-1];
            }
        }
        return slow+1;
```

```
    }
};
```

时间复杂度： O(N) 空间复杂度： O(1)

使用了**fast**和**slow**两个快慢指针进行判断，每次循环快指针都会向前走一位，通过快指针指向的位置和慢指针比较，若和慢指针所指向的数不同则慢指针走一位并将快指针所对应的值赋给慢指针指向的位置。

最后慢指针所指向的位置则为数组长度-1，所以返回**slow+1**即可。

详情参考：双指针技巧秒杀七道数组题目 :: labuladong的算法小抄

由上一题可延伸：

83. 删除排序链表中的重复元素

类型： 链表

```
class Solution {
public:
    ListNode* deleteDuplicates(ListNode* head) {
        if(head==nullptr) return nullptr;
        ListNode *slow = head,*fast = head;
        while(fast->next!=nullptr) {
            fast = fast->next;
            if(fast->val!=slow->val) {
                slow = slow->next;
                slow->val = fast->val;
            }
        }
        slow->next = nullptr;
        return head;
    }
};
```

时间复杂度： O(N) 空间复杂度： O(1)

同样使用快慢指针，详情见上文。

27. 移除元素

类型： 数组 双指针

```
class Solution {
public:
```

```
    int removeElement(vector<int>& nums, int val) {
        int slow = 0,fast = 0;
        while(fast!=nums.size()) {
            if(nums[fast++]!=val) {
                nums[slow++] = nums[fast-1];
            }
        }
        return slow;
    }
};
```

时间复杂度： O(N) 空间复杂度： O(1)

同样利用双指针，将重复的元素覆盖，最后返回数组长度为slow的值。

283. 移动零

类型： 数组 双指针

```
class Solution {
public:
    void moveZeroes(vector<int>& nums) {
        int slow = 0,fast = 0;
        while(fast!=nums.size()) {
            if(nums[fast++]!=0) {
                nums[slow++] = nums[fast-1];
            }
        }
        while(slow!=nums.size()) {
            nums[slow++] = 0;
        }
    }
};
```

时间复杂度： O(N) 空间复杂度： O(1)

利用上一题的做法最后在将**slow**指针往后的多余数替换为0。

167. 两数之和 II - 输入有序数组

类型： 数组 双指针 二分查找

```
class Solution {
public:
    vector<int> twoSum(vector<int>& numbers, int target) {
        int left = 0,right = numbers.size()-1;
        while(left<right) {
            int sum = numbers[left] + numbers[right];
```

```
            if(sum == target) {
                return {left+1,right+1};
            } else if(sum < target) {
                left++;
            } else {
                right--;
            }
        }
        return {0,0};
    }
};
```

时间复杂度： O(n log n) 空间复杂度： O(1)

通过二分查找的方法先从数组的两端的和开始比较，若**sum**比**target**小，则**left+1**；若**sum**比**target**大，则**right-1**。

344. 反转字符串

类型： 递归 双指针 字符串

```
class Solution {
public:
    void reverseString(vector<char>& s) {
        int left = 0,right = s.size() - 1;
        while(left<right) {
            char temp = s[left];
            s[left] = s[right];
            s[right] = temp;
            left++;right--;
        }
    }
};
```

时间复杂度： O(N) 空间复杂度： O(1)

通过双指针从头和尾同时进行。

5. 最长回文子串

类型： 字符串 动态规划

```
class Solution {
public:
    string longestPalindrome(string s) {
        string ans = "";
        for(int i=0;i<s.length();i++) {
            string s1 = charge(s,i,i);
```

```
            string s2 = charge(s,i,i+1);
            ans = ans.length() > s1.length() ? ans : s1;
            ans = ans.length() > s2.length() ? ans : s2;
        }
        return ans;
    }
    string charge(string s,int left,int right) {
        while(left>=0 && right < s.length() && s[left]==s[right]) {
            left--;
            right++;
        }
        return s.substr(left+1,right-left-1);
    }
};
```

时间复杂度： O(n²) 空间复杂度： O(1)

先确定中心点的位置，计算回文字符串长度分别为奇数和偶数的情况，利用双指针求出每一个回文字符串，并通过比较实时更新最长回文字符串。

2022.6.22——二叉树

周三——雷阵雨转阴——32℃/26℃

今天决定通过labuladong的算法教程来学习数据结构和算法并刷题，详情见：东哥带你刷二叉树（纲领篇） :: labuladong的算法小抄

104. 二叉树的最大深度

类型： 树 深度优先搜索 广度优先搜索 二叉树

1.遍历法

```
class Solution {
    int depth = 0,ans = 0;
    public int maxDepth(TreeNode root) {
        traverse(root);
        return ans;
    }
    public void traverse(TreeNode root) {
        if(root==null) return;
        depth++;
        if(root.left==null&&root.right==null) {
            ans = Math.max(ans,depth);
        }
        traverse(root.left);
        traverse(root.right);
        depth--;
```

```
    }
}
```

时间复杂度： O(N) 空间复杂度： O(N)

遍历二叉树计算法通过前序计算最大深度，后序深度-1来模拟深度。

2.分解问题法

```
class Solution {
    public int maxDepth(TreeNode root) {
        if(root==null) return 0;
        int leftNode = maxDepth(root.left);
        int rightNode = maxDepth(root.right);
        int ans = Math.max(leftNode,rightNode) + 1;
        return ans;
    }
}
```

时间复杂度： O(N) 空间复杂度： O(N)

通过直接从子树的最大深度来求。

144. 二叉树的前序遍历

类型： 栈 树 深度优先搜索 二叉树

1.遍历法

```
class Solution {
    List<Integer> ans = new LinkedList<>();
    public List<Integer> preorderTraversal(TreeNode root) {
        traverse(root);
        return ans;
    }
    public void traverse(TreeNode root) {
        if(root==null) return;
        ans.add(root.val);
        traverse(root.left);
        traverse(root.right);
    }
}
```

时间复杂度： O(N) 空间复杂度： O(N)

2.分解问题法

```
class Solution {
    public List<Integer> preorderTraversal(TreeNode root) {
        List<Integer> ans = new LinkedList<>();
        if(root==null) return ans;
        ans.add(root.val);
        ans.addAll(preorderTraversal(root.left));
        ans.addAll(preorderTraversal(root.right));
        return ans;
    }
}
```

时间复杂度： O(n²) 空间复杂度： O(N)

并不常用，仅仅针对java。

543. 二叉树的直径

类型： 树 深度优先搜索 二叉树

1.前序遍历法

```
class Solution {
    int maxDiameter = 0;
    public int diameterOfBinaryTree(TreeNode root) {
        traverse(root);
        return maxDiameter;
    }
    public void traverse(TreeNode root) {
        if(root==null) return;

        int leftDiameter = maxDepth(root.left);
        int rightDiameter = maxDepth(root.right);

        maxDiameter = Math.max(maxDiameter,leftDiameter+rightDiameter);

        traverse(root.left);
        traverse(root.right);
    }
    public int maxDepth(TreeNode root) {
        if(root==null) return 0;

        int leftMax = maxDepth(root.left);
        int rightMax = maxDepth(root.right);

        return Math.max(leftMax,rightMax) + 1;
    }
}
```

时间复杂度： O(n²) 空间复杂度： O(N)

先用后序遍历求出每个节点左右的最大深度，再求出每个节点的最大直径。

2.后序遍历法

```
class Solution {
    int maxDiameter = 0;
    public int diameterOfBinaryTree(TreeNode root) {
        maxDepth(root);
        return maxDiameter;
    }
    public int maxDepth(TreeNode root) {
        if(root==null) return 0;

        int leftMax = maxDepth(root.left);
        int rightMax = maxDepth(root.right);

        maxDiameter = Math.max(maxDiameter,leftMax+rightMax);

        return Math.max(leftMax,rightMax) + 1;
    }
}
```

时间复杂度： O(N) 空间复杂度： O(N)

将两种计算全部用后序遍历的方法计算，可以将时间复杂度降到O(N)。

2022.6.23——二叉树

周四——小雨转大雨——33℃/26℃

今天继续练习二叉树的题目。

100. 相同的树

类型： 树 深度优先搜索 广度优先搜索 二叉树

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

```
    }
}
```

时间复杂度： O(min(m,n)) 空间复杂度： O(min(m,n))

111. 二叉树的最小深度

类型： 树 深度优先搜索 广度优先搜索 二叉树

```
class Solution {
    public int minDepth(TreeNode root) {
        //  层序遍历
        if(root==null) return 0;
        Queue<TreeNode> q = new LinkedList<>();
        q.offer(root);
        int depth = 1;
        while(!q.isEmpty()){
            int len = q.size();
            for(int i=0;i<len;i++) {
                TreeNode node = q.poll();
                if(node.left==null&&node.right==null) {
                    return depth;
                }
                if(node.left!=null) {
                    q.offer(node.left);
                }
                if(node.right!=null) {
                    q.offer(node.right);
                }
            }
            depth++;
        }
        return depth;
    }
}
```

时间复杂度： O(N) 空间复杂度： O(N)

637. 二叉树的层平均值

类型： 树 深度优先搜索 广度优先搜索 二叉树

```
class Solution {
    public List<Double> averageOfLevels(TreeNode root) {
        if(root==null) return null;
        Queue<TreeNode> q = new LinkedList<>();
        List<Double> l = new LinkedList<>();
        q.offer(root);
        while(!q.isEmpty()) {
```

```
            int len = q.size();
            double ans = 0;
            for(int i=0;i<len;i++) {
                TreeNode node = q.poll();
                double val = node.val;
                ans += val;
                if(node.left!=null) {
                    q.offer(node.left);
                }
                if(node.right!=null) {
                    q.offer(node.right);
                }
            }
            ans /= len*1.0;
            l.add(ans);
        }
        return l;
    }
}
```

时间复杂度： O(N) 空间复杂度： O(N)

102. 二叉树的层序遍历

类型： 树 广度优先搜索 二叉树

```
class Solution {
    public List<List<Integer>> levelOrder(TreeNode root) {
        List<List<Integer>> ans = new LinkedList<>();
        if(root==null) return ans;
        Queue<TreeNode> q = new LinkedList<>();
        q.offer(root);
        while(!q.isEmpty()) {
            int len = q.size();
            List<Integer> l = new LinkedList<>();
            for(int i=0;i<len;i++) {
                TreeNode node = q.poll();
                l.add(node.val);
                if(node.left!=null) {
                    q.offer(node.left);
                }
                if(node.right!=null) {
                    q.offer(node.right);
                }
            }
            ans.add(l);
        }
        return ans;
    }
}
```

时间复杂度： O(N) 空间复杂度： O(N)

107. 二叉树的层序遍历 II

类型： 树 广度优先搜索 二叉树

```
class Solution {
    public List<List<Integer>> levelOrderBottom(TreeNode root) {
        LinkedList<List<Integer>> ans = new LinkedList<>();
        if(root==null) return ans;
        Queue<TreeNode> q = new LinkedList<>();
        q.offer(root);
        while(!q.isEmpty()) {
            int len = q.size();
            List<Integer> l = new LinkedList<>();
            for(int i=0;i<len;i++) {
                TreeNode node = q.poll();
                l.add(node.val);
                if(node.left!=null) {
                    q.offer(node.left);
                }
                if(node.right!=null) {
                    q.offer(node.right);
                }
            }
            ans.addFirst(l);
        }
        return ans;
    }
}
```

时间复杂度： O(N) 空间复杂度： O(N)

103. 二叉树的锯齿形层序遍历

类型： 树 广度优先搜索 二叉树

```
class Solution {
    public List<List<Integer>> zigzagLevelOrder(TreeNode root) {
        List<List<Integer>> ans = new LinkedList<>();
        if(root==null) return ans;
        Queue<TreeNode> q = new LinkedList<>();
        q.offer(root);
        boolean flag = true;
        while(!q.isEmpty()) {
            int len = q.size();
            LinkedList<Integer> l = new LinkedList<>();
            for(int i=0;i<len;i++) {
                TreeNode node = q.poll();
                if(flag) {
                    l.add(node.val);
                } else {
                    l.addFirst(node.val);
```

```
                }
                if(node.left!=null) {
                    q.offer(node.left);
                }
                if(node.right!=null) {
                    q.offer(node.right);
                }
            }
            flag = !flag;
            ans.add(l);
        }
        return ans;
    }
}
```

时间复杂度： O(N) 空间复杂度： O(N)

2022.6.24——链表

周五——小雨——30℃/26℃

今天开始从数据结构部分刷题。

21. 合并两个有序链表

类型： 递归 链表

```
class Solution {
    public ListNode mergeTwoLists(ListNode list1, ListNode list2) {
        ListNode head = new ListNode(-1),p = head;
        ListNode p1 = list1,p2 = list2;
        while(p1!=null && p2!=null) {
            if(p1.val<p2.val) {
                p.next = p1;
                p1 = p1.next;
            } else{
                p.next = p2;
                p2 = p2.next;
            }
            p = p.next;
        }
        if(p1!=null) {
            p.next = p1;
        }
        if(p2!=null) {
            p.next = p2;
        }
        return head.next;
    }
}
```

时间复杂度： O(m+n) 空间复杂度： O(m+n)

86. 分隔链表

类型： 链表 双指针

```
class Solution {
    public ListNode partition(ListNode head, int x) {
        ListNode dummy1 = new ListNode(-1);
        ListNode dummy2 = new ListNode(-1);
        ListNode p1 = dummy1,p2 = dummy2;
        ListNode p = head;
        while(p!=null) {
            if(p.val<x) {
                p1.next = p;
                p1 = p1.next;
            } else {
                p2.next = p;
                p2 = p2.next;
            }
            ListNode temp = p.next;
            p.next = null;
            p = temp;
        }
        p1.next = dummy2.next;
        return dummy1.next;
    }
}
```

时间复杂度： O(N) 空间复杂度： O(N)

利用两个子链表，将大于等于x和小于x的节点分别存入不同的子链表，最后再将两个子链表连接。注意在每次存入子链表后要断开next的节点。

19. 删除链表的倒数第 N 个结点

类型： 链表 双指针

```
class Solution {
    public ListNode removeNthFromEnd(ListNode head, int n) {
        ListNode dummy = new ListNode(-1);
        dummy.next = head;
        ListNode p1 = dummy,p2 = dummy;
        while(n-->=0) {
            p1 = p1.next;
        }
        while(p1!=null) {
            p1 = p1.next;
```

```
            p2 = p2.next;
        }
        p2.next = p2.next.next;
        return dummy.next;
    }
}
```

时间复杂度： O(N) 空间复杂度： O(N)

利用两个指针，第一个先向前走n+1次，然后两个节点一起走，直到第一个节点走到尾部时第二个节点就到了倒数第n+1个节点处，然后将第二个节点的下一个删除。

876. 链表的中间结点

类型： 链表 双指针

```
class Solution {
    public ListNode middleNode(ListNode head) {
        ListNode slow = head,fast = head;
        while(fast!=null&&fast.next!=null) {
            fast = fast.next.next;
            slow = slow.next;
        }
        return slow;
    }
}
```

时间复杂度： O(N) 空间复杂度： O(N)

同上。

2022.6.25——链表

周六——雷阵雨转阴——33℃/28℃

今天继续刷链表的题

206. 反转链表

类型： 递归 链表

```
class Solution {
    public ListNode reverseList(ListNode head) {
        if(head==null || head.next==null) {
```

```
            return head;
        }
        ListNode last = reverseList(head.next);
        head.next.next = head;
        head.next = null;
        return last;
    }
}
```

时间复杂度： O(N) 空间复杂度： O(N)

利用递归的方法实现，代码简洁且不容易出错，空间使用会过多。

92. 反转链表 II

类型： 递归 链表

```
class Solution {
    ListNode tail = null;
    public ListNode reverseBetween(ListNode head, int left, int right) {
        if(left == 1) {
            return reverse(head,right);
        }
        head.next = reverseBetween(head.next,left-1,right-1);
        return head;
    }
    public ListNode reverse(ListNode head,int n) {
        if(n==1) {
            tail = head.next;
            return head;
        }
        ListNode last = reverse(head.next,n-1);
        head.next.next = head;
        head.next = tail;
        return last;
    }
}
```

时间复杂度： O(N) 空间复杂度： O(N)

同上，只是加上了边界管理，详情见递归魔法：反转单链表 :: labuladong的算法小抄

学习笔记

2022.6.23——二叉树笔记

1.二叉树遍历框架

```
void traverse(TreeNode root) {
    if (root == null) {
        return;
    }
    // 前序位置
    traverse(root.left);
    // 中序位置
    traverse(root.right);
    // 后序位置
}
```

2.数组&链表的二叉树遍历法

```
/* 迭代遍历数组 */
void traverse(int[] arr) {
    for (int i = 0; i < arr.length; i++) {

    }
}

/* 递归遍历数组 */
void traverse(int[] arr, int i) {
    if (i == arr.length) {
        return;
    }
    // 前序位置
    traverse(arr, i + 1);
    // 后序位置
}

/* 迭代遍历单链表 */
void traverse(ListNode head) {
    for (ListNode p = head; p != null; p = p.next) {

    }
}

/* 递归遍历单链表 */
void traverse(ListNode head) {
    if (head == null) {
        return;
    }
    // 前序位置
    traverse(head.next);
    // 后序位置
}
```

3.层序遍历

```
// 输入一棵二叉树的根节点，层序遍历这棵二叉树
void levelTraverse(TreeNode root) {
    if (root == null) return;
    Queue<TreeNode> q = new LinkedList<>();
    q.offer(root);

    // 从上到下遍历二叉树的每一层
    while (!q.isEmpty()) {
        int sz = q.size();
        // 从左到右遍历每一层的每个节点
        for (int i = 0; i < sz; i++) {
            TreeNode cur = q.poll();
            // 将下一层节点放入队列
            if (cur.left != null) {
                q.offer(cur.left);
            }
            if (cur.right != null) {
                q.offer(cur.right);
            }
        }
    }
}
```

```
List<List<Integer>> res = new ArrayList<>();

List<List<Integer>> levelTraverse(TreeNode root) {
    if (root == null) {
        return res;
    }
    // root 视为第 0 层
    traverse(root, 0);
    return res;
}

void traverse(TreeNode root, int depth) {
    if (root == null) {
        return;
    }
    // 前序位置，看看是否已经存储 depth 层的节点了
    if (res.size() <= depth) {
        // 第一次进入 depth 层
        res.add(new LinkedList<>());
    }
    // 前序位置，在 depth 层添加 root 节点的值
    res.get(depth).add(root.val);
    traverse(root.left, depth + 1);
    traverse(root.right, depth + 1);
}
```

```
List<List<Integer>> res = new LinkedList<>();

List<List<Integer>> levelTraverse(TreeNode root) {
```

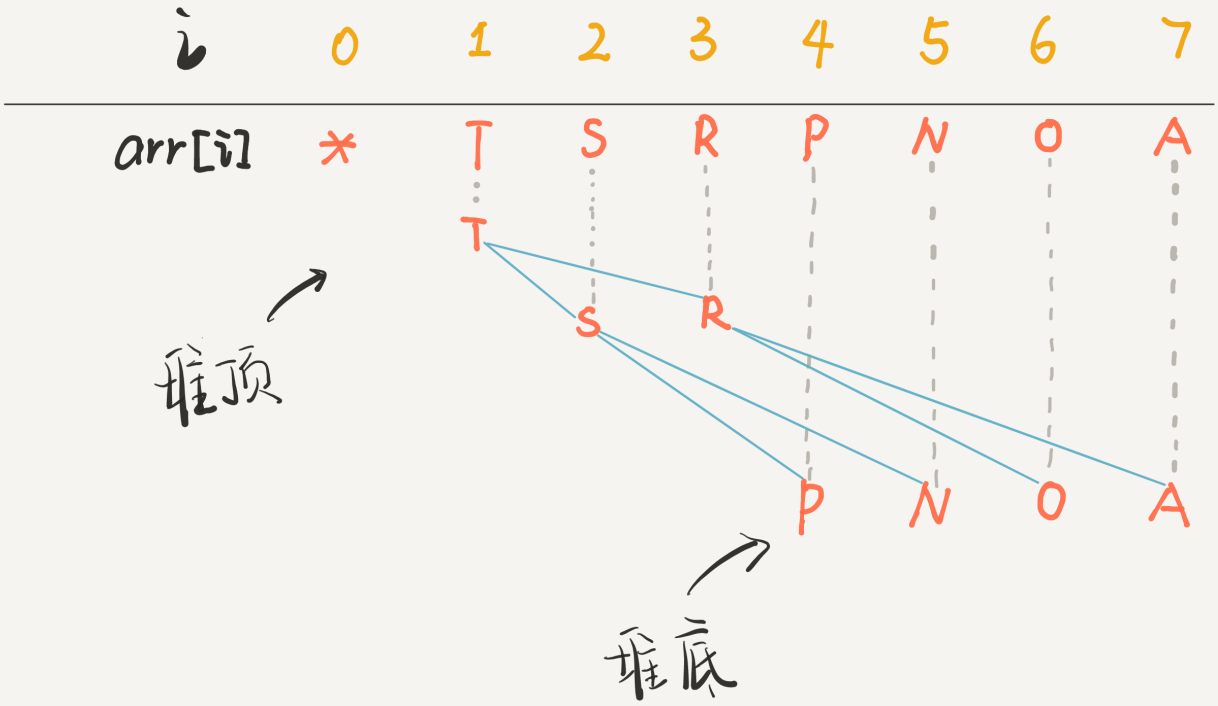
```
    if (root == null) {
        return res;
    }
    List<TreeNode> nodes = new LinkedList<>();
    nodes.add(root);
    traverse(nodes);
    return res;
}

void traverse(List<TreeNode> curLevelNodes) {
    // base case
    if (curLevelNodes.isEmpty()) {
        return;
    }
    // 前序位置，计算当前层的值和下一层的节点列表
    List<Integer> nodeValues = new LinkedList<>();
    List<TreeNode> nextLevelNodes = new LinkedList<>();
    for (TreeNode node : curLevelNodes) {
        nodeValues.add(node.val);
        if (node.left != null) {
            nextLevelNodes.add(node.left);
        }
        if (node.right != null) {
            nextLevelNodes.add(node.right);
        }
    }
    // 前序位置添加结果，可以得到自顶向下的层序遍历
    res.add(nodeValues);
    traverse(nextLevelNodes);
    // 后序位置添加结果，可以得到自底向上的层序遍历结果
    // res.add(nodeValues);
}
```

2022.6.24——二叉堆学习

1.二叉堆概览

二叉堆就是用数组来维护的二叉树

```
// 父节点的索引
int parent(int root) {
    return root / 2;
}
// 左孩子的索引
int left(int root) {
    return root * 2;
}
// 右孩子的索引
int right(int root) {
    return root * 2 + 1;
}
```



2.优先级队列概览

```
public class MaxPQ
    <Key extends Comparable<Key>> {
    // 存储元素的数组
    private Key[] pq;
    // 当前 Priority Queue 中的元素个数
    private int size = 0;

    public MaxPQ(int cap) {
        // 索引 0 不用，所以多分配一个空间
        pq = (Key[]) new Comparable[cap + 1];
    }

    /* 返回当前队列中最大元素 */
    public Key max() {
        return pq[1];
    }

    /* 插入元素 e */
    public void insert(Key e) {...}

    /* 删除并返回当前队列中最大元素 */
    public Key delMax() {...}

    /* 上浮第 x 个元素，以维护最大堆性质 */
    private void swim(int x) {...}
```

```
    /* 下沉第 x 个元素，以维护最大堆性质 */
    private void sink(int x) {...}

    /* 交换数组的两个元素 */
    private void swap(int i, int j) {
        Key temp = pq[i];
        pq[i] = pq[j];
        pq[j] = temp;
    }

    /* pq[i] 是否比 pq[j] 小？ */
    private boolean less(int i, int j) {
        return pq[i].compareTo(pq[j]) < 0;
    }

    /* 还有 left, right, parent 三个方法 */
}
```

3.实现swim和sink

上浮：

```
private void swim(int x) {
    // 如果浮到堆顶，就不能再上浮了
    while (x > 1 && less(parent(x), x)) {
        // 如果第 x 个元素比上层大
        // 将 x 换上去
        swap(parent(x), x);
        x = parent(x);
    }
}
```

下沉：

```
private void sink(int x) {
    // 如果沉到堆底，就沉不下去了
    while (left(x) <= size) {
        // 先假设左边节点较大
        int max = left(x);
        // 如果右边节点存在，比一下大小
        if (right(x) <= size && less(max, right(x)))
            max = right(x);
        // 结点 x 比俩孩子都大，就不必下沉了
        if (less(max, x)) break;
        // 否则，不符合最大堆的结构，下沉 x 结点
        swap(x, max);
        x = max;
    }
}
```

4.实现delMax和insert

insert：

```
public void insert(Key e) {
    size++;
    // 先把新元素加到最后
    pq[size] = e;
    // 然后让它上浮到正确的位置
    swim(size);
}
```

delMax：

```
public Key delMax() {
    // 最大堆的堆顶就是最大元素
    Key max = pq[1];
    // 把这个最大元素换到最后，删除之
    swap(1, size);
    pq[size] = null;
    size--;
    // 让 pq[1] 下沉到正确位置
    sink(1);
    return max;
}
```

2022.6.25——链表

1.反转指定区间链表元素

主要分两部分，一部分是先递归到起点位置，一部分是执行反转right-left长度的链表元素。

```
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;
}
```

先前进到m=1时，就等同于反转前n-m个链表元素，注意最终的答案是在if(m==1)中返回的，下面的return只是将前进的m-1步又退回去了。

```
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;
}
```

反转前n个链表元素部分和反转整个链表相似，只是将head.next = null改成了 = successor，在到了n = 1处就将head.next的节点信息传给successor作为反转部分反转后尾结点的下一个，也就能连接上非反转部分。

补充内容：