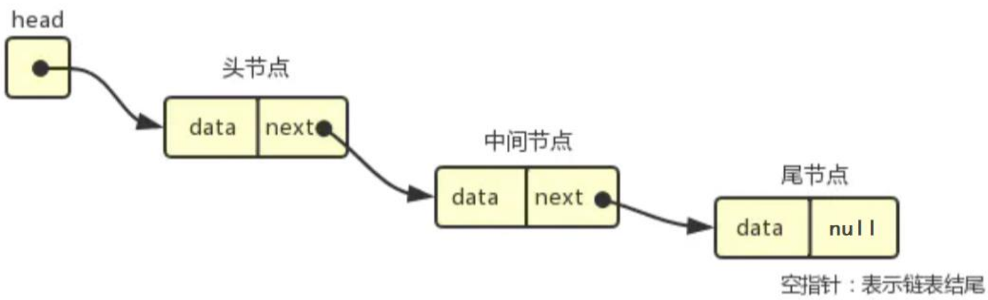
第六章 链表问题讲解

链表（Linked List）是一种常见的基础数据结构，是一种线性表，但是并不会按线性的顺序存储数据，而是在每一个节点里存到下一个节点的指针（Pointer）。



由于不必须按顺序存储，链表在插入的时候可以达到 O(1)的复杂度，比另一种线性表 —— 顺序表快得多，但是查找一个节点或者访问特定编号的节点则需要 O(n) 的时间，而顺序表相应的时间复杂度分别是 O(n) 和 O(1)。

链表允许插入和移除表上任意位置上的节点，但是不允许随机存取。链表有很多种不同的类型：单向链表，双向链表以及循环链表。

6.1 反转链表（#206）

6.1.1 题目说明

反转一个单链表。

示例:

```
输入: 1->2->3->4->5->NULL
输出: 5->4->3->2->1->NULL
```

进阶:

你可以迭代或递归地反转链表。你能否用两种方法解决这道题？

6.1.2 分析

链表的节点结构 ListNode 已经定义好，我们发现，反转链表的过程，其实跟 val 没有关系，只要把每个节点的 next 指向之前的节点就可以了。

从代码实现上看，可以有迭代和递归两种形式。

6.1.3 方法一：迭代

假设存在链表 1→2→3→null，我们想要把它改成 null←1←2←3。

我们只需要依次迭代节点遍历链表，在迭代过程中，将当前节点的 next 指针改为指向前一个元素就可以了。

代码如下：

```
public class ReverseLinkedList {
    public ListNode reverseList(ListNode head) {
        ListNode curr = head;
        ListNode prev = null;
        // 依次迭代遍历链表
        while (curr != null){
            ListNode tempNext = curr.next;
            curr.next = prev;
            prev = curr;
            curr = tempNext;
        }
        return prev;
    }
}
```

复杂度分析

时间复杂度：O(n)，假设 n 是链表的长度，时间复杂度是 O(n)。

空间复杂度：O(1)。

6.1.4 方法二：递归

递归的核心，在于当前只考虑一个节点。剩下部分可以递归调用，直接返回一个反转好的链表，然后只要把当前节点再接上去就可以了。

假设链表为（长度为 m）：

$n_1 \to n_2 \to \cdots \to n_{k-1} \to n_k \to n_{k+1} \to \cdots \to n_m \to null$

若我们遍历到了 n_k，那么认为剩余节点 n_{k+1} 到 n_m 已经被反转。

$n_1 \to n_2 \to \cdots \to n_{k-1} \to n_k \to n_{k+1} \leftarrow \cdots \leftarrow n_m$

我们现在希望 n_{k+1} 的下一个节点指向 n_k，所以，应该有

$n_{k+1}.next = n_k$

代码如下：

```java
public ListNode reverseList(ListNode head) {
    if (head == null || head.next == null){
        return head;
    }
    ListNode restHead = head.next;
    ListNode reversedRest = reverseList(restHead);    // 递归反转
    restHead.next = head;
    head.next = null;
    return reversedRest;
}
```

复杂度分析

时间复杂度：时间复杂度：O(n)，假设 n 是链表的长度，那么时间复杂度为 O(n)。

空间复杂度：O(n)，由于使用递归，将会使用隐式栈空间。递归深度可能会达到 n 层。

6.2 合并两个有序链表（#21）

6.2.1 题目说明

将两个升序链表合并为一个新的**升序**链表并返回。新链表是通过拼接给定的两个链表的所有节点组成的。

示例：

```
输入：1->2->4, 1->3->4
输出：1->1->2->3->4->4
```

6.2.2 分析

链表节点结构已经定义好，而且已经做了升序排列。现在我们需要分别遍历两个链表，然后依次比较，按从小到大的顺序生成新的链表就可以了。这其实就是“归并排序”的思路。

6.2.3 方法一：迭代

最简单的想法，就是逐个遍历两个链表中的节点，依次比对。

我们假设原链表为 list1 和 list2。只要它们都不为空，就取出当前它们各自的头节点就行比较。值较小的那个结点选取出来，加入到结果链表中，并将对应原链表的头（head）指向下一个结点；而值较大的那个结点则保留，接下来继续做比对。

另外，为了让代码更加简洁，我们可以引入一个哨兵节点（sentinel），它的 next 指向结果链表的头结点，它的值设定为-1。

代码如下：

```java
public class MergeTwoSortedLists {

    public ListNode mergeTwoLists(ListNode l1, ListNode l2) {

        //定义一个哨兵节点

        ListNode resultPrev = new ListNode(-1);
```

```
        ListNode prev = resultPrev;
        // 遍历两个链表
        while ( l1 != null && l2 != null ){
            if ( l1.val <= l2.val ){
                prev.next = l1;
                prev = l1;
                l1 = l1.next;
            } else {
                prev.next = l2;
                prev = l2;
                l2 = l2.next;
            }
        }
        prev.next = (l1 == null) ? l2 : l1;
        return resultPrev.next;
    }
}
```

复杂度分析

时间复杂度：O(n + m) ，其中 n 和 m 分别为两个链表的长度。因为每次循环迭代中，l1 和 l2 只有一个元素会被放进合并链表中， 因此 while 循环的次数不会超过两个链表的长度之和。所有其他操作的时间复杂度都是常数级别的，因此总的时间复杂度为 O(n+m)。

空间复杂度：O(1)。我们只需要常数的空间存放若干变量。

6.2.4 方法二：递归

用递归的方式同样可以实现上面的过程。

当两个链表都不为空时，我们需要比对当前两条链的头节点。取出较小的那个节点；而两条链其余的部分，可以递归调用，认为它们已经排好序。所以我们需要做的，就是把前面取出的那个节点，接到剩余排好序的链表头节点前。

代码如下：

```
public ListNode mergeTwoLists(ListNode l1, ListNode l2) {
    if ( l1 == null )
        return l2;
    else if ( l2 == null )
        return l1;
    if ( l1.val <= l2.val ){
        l1.next = mergeTwoLists(l1.next, l2);
        return l1;
    } else {
        l2.next = mergeTwoLists(l1, l2.next);
        return l2;
    }
}
```

复杂度分析

时间复杂度：O(n + m)，其中 nn 和 m 分别为两个链表的长度。因为每次调用递归都会去掉 l1 或者 l2 的头节点（直到至少有一个链表为空），函数 mergeTwoList 至多只会递归调用每个节点一次。因此，时间复杂度取决于合并后的链表长度，即 O(n+m)。

空间复杂度：O(n + m)，其中 n 和 m 分别为两个链表的长度。递归调用 mergeTwoLists 函数时需要消耗栈空间，栈空间的大小取决于递归调用的深度。结束递归调用时 mergeTwoLists 函数最多调用 n+m 次，因此空间复杂度为 O(n+m)。

6.3 删除链表的倒数第 N 个节点（#19）

6.3.1 题目说明

给定一个链表，删除链表的倒数第 n 个节点，并且返回链表的头结点。

示例：

```
给定一个链表: 1->2->3->4->5, 和 n = 2.
当删除了倒数第二个节点后，链表变为 1->2->3->5.
```

说明：

给定的 n 保证是有效的。

进阶：

你能尝试使用一趟扫描实现吗？

6.3.2 分析

在链表中删除某个节点，其实就是将之前一个节点 next，直接指向当前节点的后一个节点，相当于“跳过”了这个节点。

当然，真正意义上的删除，还应该回收节点本身占用的空间，进行内存管理。这一点在 java 中我们可以不考虑，直接由 JVM 的 GC 帮我们实现。

6.3.3 方法一：计算链表长度（二次遍历）

最简单的想法是，我们首先从头节点开始对链表进行一次遍历，得到链表的长度 L。

然后，我们再从头节点开始对链表进行一次遍历，当遍历到第 L-N+1 个节点时，它就是我们需要删除的倒数第 N 个节点。

这样，总共做两次遍历，我们就可以得到结果。

代码如下：

```java
public class RemoveNthNodeFromEnd {
    public ListNode removeNthFromEnd(ListNode head, int n) {
        // 遍历链表，获取长度
        int l = getLength(head);
        // 定义哑节点（哨兵）
        ListNode sentinel = new ListNode(-1);
        sentinel.next = head;
        // 再次遍历，找到倒数第 N 个
        ListNode curr = sentinel;
        for ( int i = 0; i < l - n; i++ ){
            curr = curr.next;
        }
        curr.next = curr.next.next;
        return sentinel.next;
    }
    // 定义一个获取链表长度的方法
    public static int getLength(ListNode head){
        int length = 0;
        while ( head != null ){
            length ++;
            head = head.next;
        }
        return length;
    }
}
```

复杂度分析

时间复杂度：O(L)，其中 L 是链表的长度。只用了两次遍历，是线性时间复杂度。

空间复杂度：O(1)。

6.3.4 方法二：利用栈

另一个思路是利用栈数据结构。因为栈是“先进后出”的，所以我们可以在遍历链表的同时将所有节点依次入栈，然后再依次弹出。

这样，弹出栈的第 n 个节点就是需要删除的节点，并且目前栈顶的节点就是待删除节点的前驱节点。这样一来，删除操作就变得十分方便了。

代码如下：

```
public ListNode removeNthFromEnd(ListNode head, int n) {
    ListNode sentinel = new ListNode(-1);
    sentinel.next = head;
    // 定义栈
    Stack<ListNode> stack = new Stack<>();
    ListNode curr = sentinel;
    // 遍历链表，所有节点入栈
    while ( curr != null ){
        stack.push(curr);
        curr = curr.next;
    }
    // 依次弹栈，弹出 N 个
    for ( int i = 0; i < n; i++ ){
        stack.pop();
    }
    stack.peek().next = stack.peek().next.next;
    return sentinel.next;
}
```

复杂度分析

时间复杂度：O(L)，其中 L 是链表的长度。我们压栈遍历了一次链表，弹栈遍历了 N 个

节点，所以应该耗费 O(L+N)时间。N <= L，所以时间复杂度依然是 O(L)，而且我们可以看出，遍历次数比两次要少，但依然没有达到“一次遍历”的要求。

空间复杂度：O(L)，其中 L 是链表的长度。主要为栈的开销。

6.3.5 方法三：双指针（一次遍历）

我们可以使用两个指针 first 和 second 同时对链表进行遍历，要求 first 比 second 超前 N 个节点。

这样，它们总是保持着 N 的距离，当 first 遍历到链表的末尾（null）时，second 就恰好处于第 L-N+1，也就是倒数第 N 个节点了。

代码如下：

```java
public ListNode removeNthFromEnd(ListNode head, int n) {
    ListNode sentinel = new ListNode(-1);
    sentinel.next = head;
    ListNode first = sentinel, second = sentinel;
    for ( int i = 0; i < n + 1; i++ ){
        first = first.next;
    }
    while ( first != null ){
        first = first.next;
        second = second.next;
    }
    second.next = second.next.next;
    return sentinel.next;
}
```

复杂度分析

时间复杂度：O(L)，其中 L 是链表的长度。这次真正实现了一次遍历。

空间复杂度：O(1)。

第七章 哈希表相关问题讲解

7.1 哈希表数据结构复习

7.1.1 基本概念

哈希表（Hash Table）也叫散列表，是可以根据关键字值(Key value)而直接进行访问的数据结构。也就是说，它通过把关键字值映射到表中一个位置来访问记录，以加快查找的速度。这个映射函数叫做散列函数（哈希函数），存放记录的数组就叫做散列表。

哈希表里保存的数据元素是一组键-值对（key-value pair），它的特性就是可以根据给出的 key 快速访问 value。

哈希表在不考虑冲突的情况下，插入、删除和访问操作时间复杂度均为 O(1)。

7.1.2 核心问题

设计一个哈希表，有两个核心问题需要去解决：

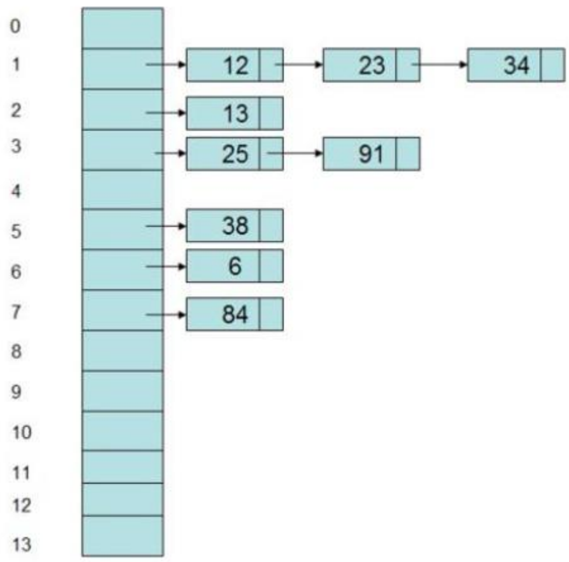
1) 如何设计哈希方法（哈希函数）
2) 如何避免哈希碰撞

哈希方法（hash method，也叫哈希函数）会将键值映射到某块存储空间。

一个好的哈希方法，应该将不同的键值，均匀地分布在存储空间中。理想情况下，每个值都应该有一个对应唯一的散列值。

哈希方法要将大量的键值，映射到一个有限的空间里。这样就有可能会将不同的键值，映射到同一个存储空间，这种情况称为“哈希碰撞”（Hash Collision，也叫“哈希冲突”）。哈希碰撞是不可避免的，但可以用策略来解决哈希碰撞。

为了解决 哈希碰撞，我们利用 **桶** 来存储所有对应的数值。桶可以用 数组 或 链表 来实现（Java 中就是用链表来实现的）。



7.2 只出现一次的数字（#136）

7.2.1 题目说明

给定一个非空整数数组，除了某个元素只出现一次以外，其余每个元素均出现两次。找出那个只出现了一次的元素。

说明：

你的算法应该具有线性时间复杂度。你可以不使用额外空间来实现吗？

示例 1:

```
输入: [2,2,1]
输出: 1
```

示例 2:

```
输入: [4,1,2,1,2]
输出: 4
```

7.2.2 分析

这是基于数组的一道题目。

题目中除了一个元素之外，其它都出现两次。所以我们可以想到，只要把元素是否出现

过记录下来，遍历完数组就可以判断出单独的那个数了。

7.2.3 方法一：暴力法

基本想法是，遍历数组，把当前所有出现的单独元素都另外保存下来。遇到重复的就删除。

代码如下：

```
public int singleNumber(int[] nums) {
    List<Integer> singleList = new ArrayList<>();
    for (Integer num : nums) {
        if (singleList.contains(num))
            singleList.remove(num);
        else
            singleList.add(num);
    }
    return singleList.get(0);
}
```

复杂度分析

时间复杂度：O(n^2)。我们遍历 nums 花费 O(n) 的时间；另外我们还要在列表中遍历，判断是否存在这个数字，再花费 O(n) 的时间，所以总循环时间为 O(n^2)。

空间复杂度：O(n)。我们需要一个大小为 n 的列表保存所有的 nums 中元素。

7.2.4 方法二：保存到 HashMap

由于在列表中查询需要耗费线性时间，所以可以想到，可以把数不保存到列表，而是保存到 HashMap 中，这样查询的时候不就不用再遍历一次了。

代码如下：

```java
public int singleNumber(int[] nums) {
    Map<Integer, Integer> singleMap = new HashMap<>();
    for (Integer num : nums) {
        if (singleMap.get(num) != null)
            singleMap.remove(num);
        else
            singleMap.put(num, 1);
    }
    return singleMap.keySet().iterator().next();
}
```

复杂度分析

时间复杂度：O(n) 。for 循环的时间复杂度是 O(n)。而 HashMap 的 get 操作时间复杂度为 O(1) 。

空间复杂度：O(n) 。HashMap 需要的空间与 nums 中元素个数相等。

7.2.5 方法三：保存到 set

我们可以也利用 set 来进行去重，然后计算 set 中所有元素的总和。得到的总和乘以 2，就是所有元素加了两遍；对比原数组，只多了一个那个落单的数。所以减去原数组的总和，就是要找的那个数。

代码如下：

```java
public int singleNumber3(int[] nums){
    Set<Integer> set = new HashSet<>();
    int arraySum = 0;
    Integer setSum = 0;
    for( int num: nums) {
        set.add(num);
        arraySum += num;
```

```
    }
    for( Integer num: set )
        setSum += num;
    return setSum * 2 - arraySum;
}
```

时间复杂度：O(n) 。计算 sum 和，会将 nums 中的元素遍历一遍，再将 set 中的元素遍历一遍。我们可以认为是遍历了两遍。

空间复杂度：O(n) 。HashSet 需要的空间跟 nums 中元素个数一致。

7.2.6 方法四：位运算

我们回忆一下数学上异或运算的概念：

- 如果对 0 和二进制位做 XOR 运算，得到的仍然是这个二进制位

$a \oplus 0=a$

- 如果对相同的二进制位做 XOR 运算，返回的结果是 0

$a \oplus a=0$

- XOR 满足交换律和结合律

$a \oplus b \oplus a=(a \oplus a) \oplus b=0 \oplus b=b$

所以我们只需要将所有的数进行 XOR 操作，就能得到那个唯一的数字。

代码如下：

```
public int singleNumber(int[] nums) {
    int result = 0;
    for (int num : nums)
        result ^= num;
    return result;
}
```

复杂度分析

时间复杂度：O(n)，其中 n 是数组长度。只需要对数组遍历一次。

空间复杂度：O(1)。

7.3 最长连续序列（#128）

7.3.1 题目说明

给定一个未排序的整数数组 nums ，找出数字连续的最长序列（不要求序列元素在原数组中连续）的长度。

进阶：你可以设计并实现时间复杂度为 O(n) 的解决方案吗？

示例 1：

```
输入：nums = [100,4,200,1,3,2]
输出：4
```

解释：最长数字连续序列是 [1, 2, 3, 4]。它的长度为 4。

示例 2：

```
输入：nums = [0,3,7,2,5,8,4,6,0,1]
输出：9
```

提示：

- 0 <= nums.length <= 104
- -109 <= nums[i] <= 109

7.3.2 分析

要寻找连续序列，关键在于找到当前数的“下一个数”（或者叫“后继”）。

如果有后继，就在数组中继续找，每找到一个后继，当前序列长度就加 1；直到找不到时，就得到了以当前数开始的、最长的连续序列长度。

7.3.3 方法一：暴力法

最简单的实现，就是遍历所有数据，对每一数据都找从它开始的最长连续序列。

寻找连续序列，就是要不停寻找后继。而判断后继是否存在，又要在数组中进行遍历寻找。

代码实现如下：

```
public class LongestConsecutiveSequence {
    public int longestConsecutive(int[] nums) {
        int maxLength = 0;
        for (int i = 0; i < nums.length; i++){
            int currNum = nums[i];
            int currLength = 1;
            // 判断后继是否存在，寻找连续序列
            while ( contains(nums, currNum + 1) ){
                currLength ++;
                currNum ++;
            }
            if ( currLength > maxLength )
                maxLength = currLength;
        }
        return maxLength;
    }
    // 定义一个方法，判断元素 x 是否在数组 nums 中
    public static boolean contains(int[] nums, int x){
        for ( int num: nums ){
            if ( num == x )
                return true;
        }
```

```
        return false;
    }
}
```

复杂度分析

时间复杂度：O(N^3)。我们定义了外层循环遍历数组，内层循环不停寻找后继；另外，在内层循环中每次要判断后继是否存在，还需要遍历数组查找。所以总计是 O(N^3)。

空间复杂度：O(1)。过程中只用到了一些辅助的临时变量。

7.3.4 方法二：哈希表改进

用哈希表（Hash Set）来保存数组中的元素，可以快速判断元素是否存在。这样 contains 可以优化为常数时间复杂度。

代码实现如下：

```
public int longestConsecutive(int[] nums) {
    int maxLength = 0;
    HashSet<Integer> hashSet = new HashSet<>();
    for (int num: nums)
        hashSet.add(num);
    // 遍历数组
    for (int i = 0; i < nums.length; i++){
        int currNum = nums[i];
        int currLength = 1;
        // 寻找连续序列
        while ( hashSet.contains(currNum + 1) ){
            currLength ++;
            currNum ++;
        }
        if ( currLength > maxLength )
```

```java
            maxLength = currLength;
    }
    return maxLength;
}
```

复杂度分析

时间复杂度：O(N^2)。将数组元素保存入 Hash Set 需要。后面由于简化了内层循环中判断后继的过程，只耗费 O(1)时间，所以最终是内外两重循环，最坏情况下时间复杂度为 O(N^2)。

空间复杂度：O(N)。我们用到了一个 Hash Set 来保存数组元素，排除部分重复数据，这仍然需要耗费 O(N)的内存空间。

7.3.5 方法三：哈希表进一步优化

仔细分析上面的算法过程，我们会发现其中执行了很多不必要的枚举。

例如，我们已经寻找过 x 开始的连续序列，已知有一个 x,x+1,x+2,…,x+y 的连续序列。现在要继续寻找 x+1 开始的连续序列，算法会重新寻找它的后继 x+2，而这个过程我们已经做过了。

并且，我们可以确定，这种情况得到的结果（连续序列的长度），肯定不会优于以 x 为起点的答案。因此这部分处理完全没有必要，我们在外层循环的时候碰到这种情况，直接跳过即可。

代码如下：

```java
public int longestConsecutive(int[] nums) {
    int maxLength = 0;
    HashSet<Integer> hashSet = new HashSet<>();
    for (int num : nums)
        hashSet.add(num);
    // 遍历数组
    for (int i = 0; i < nums.length; i++) {
        int currNum = nums[i];
```

```
        int currLength = 1;
        if ( !hashSet.contains(currNum - 1) ) {
            while (hashSet.contains(currNum + 1)) {
                currLength++;
                currNum++;
            }
            if (currLength > maxLength)
                maxLength = currLength;
        }
    }
    return maxLength;
}
```

复杂度分析

时间复杂度：O(N)。外层循环需要 O(n) 的时间复杂度，只有当一个数是连续序列的第一个数的情况下才会进入内层循环，然后在内层循环中匹配连续序列中的数，因此数组中的每个数只会进入内层循环一次。

空间复杂度：O(N)。哈希表保存数组中所有数据需要 O(N)的内存空间。

7.4. LRU 缓存机制（#146）

7.4.1 题目说明

运用你所掌握的数据结构，设计和实现一个 LRU (最近最少使用) 缓存机制。

实现 LRUCache 类：

- LRUCache(int capacity) 以正整数作为容量 capacity 初始化 LRU 缓存
- int get(int key) 如果关键字 key 存在于缓存中，则返回关键字的值，否则返回 -1 。
- void put(int key, int value) 如果关键字已经存在，则变更其数据值；如果关键字不存在，则插入该组「关键字-值」。当缓存容量达到上限时，它应该在写入新数据之前删除最久未使用的数据值，从而为新的数据值留出空间。

进阶：你是否可以在 O(1) 时间复杂度内完成这两种操作？

示例：

```
输入
["LRUCache", "put", "put", "get", "put", "get", "put", "get", "get", "get"]
[[2], [1, 1], [2, 2], [1], [3, 3], [2], [4, 4], [1], [3], [4]]
输出
[null, null, null, 1, null, -1, null, -1, 3, 4]
```

解释

```
LRUCache lRUCache = new LRUCache(2);
lRUCache.put(1, 1); // 缓存是 {1=1}
lRUCache.put(2, 2); // 缓存是 {1=1, 2=2}
lRUCache.get(1);    // 返回 1
lRUCache.put(3, 3); // 该操作会使得关键字 2 作废，缓存是 {1=1, 3=3}
lRUCache.get(2);    // 返回 -1 (未找到)
lRUCache.put(4, 4); // 该操作会使得关键字 1 作废，缓存是 {4=4, 3=3}
lRUCache.get(1);    // 返回 -1 (未找到)
lRUCache.get(3);    // 返回 3
lRUCache.get(4);    // 返回 4
```

提示：

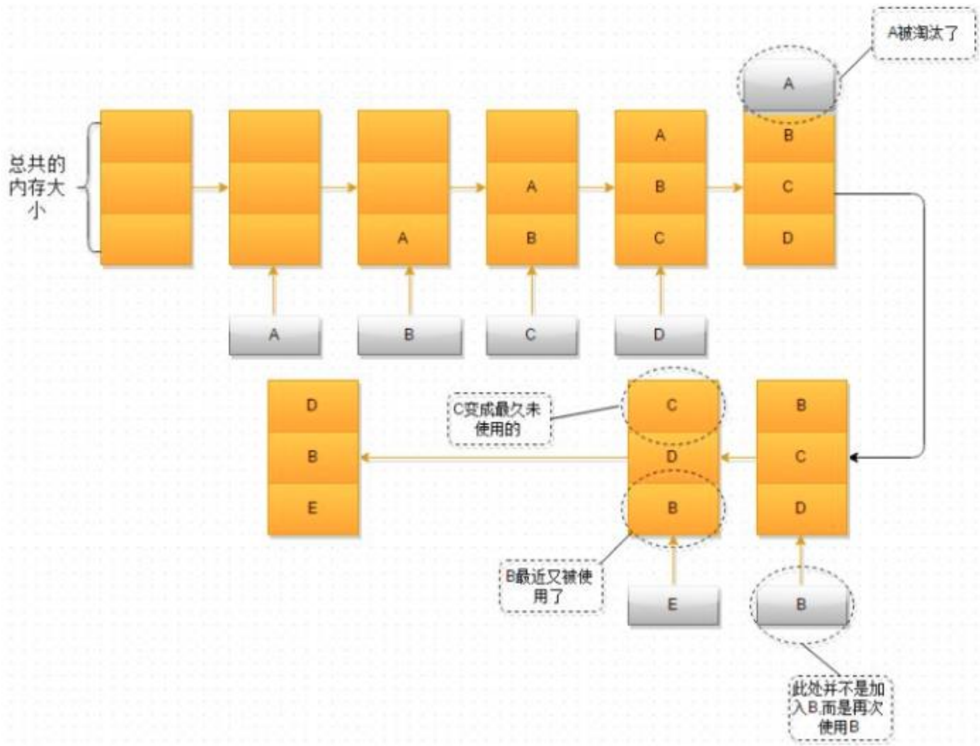
- 1 <= capacity <= 3000
- 0 <= key <= 3000
- 0 <= value <= 104
- 最多调用 3 * 104 次 get 和 put

7.4.2 分析

LRU（Least recently used，最近最少使用）是一种常用的页面置换算法，选择最近最久未使用的页面予以淘汰。

所谓的“最近最久未使用”，就是根据数据的历史访问记录来判断的，其核心思想是“如果数据最近被访问过，那么将来被访问的几率也更高”。

LRU 是最常见的缓存机制，在操作系统的虚拟内存管理中，有非常重要的应用，所以也是面试中的常客。

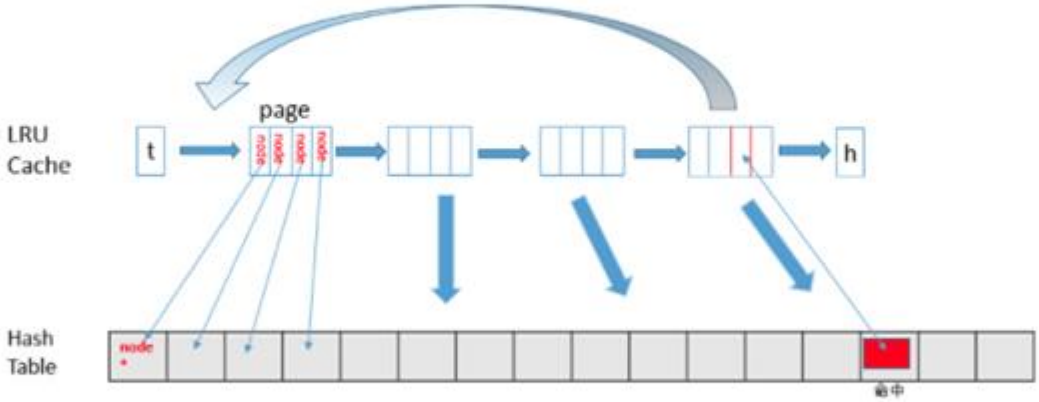


具体实现上，既然保存的是键值对，而且要根据 key 来判断数据是否在缓存中，那么就可以用一个 **HashMap** 来作为缓存的存储数据结构。这样，我们的访问和插入，就都可以以常数时间进行了。

需要额外考虑的是，缓存空间有限，所以这个 HashMap 要有一个容量限制；而且当达到容量上限时，我们会运用 LRU 的策略删除最近最少使用的那个数据。

这就要求我们必须把数据，按照一定的线性结构排列起来，最新访问的数据放在后面，新数据的插入可以“顶掉”最前面的不常访问的数据。这种数据结构其实可以用**链表**来实现。

所以，我们最终可以使用一个哈希表+双向链表的数据结构，来实现 LRU 缓存机制。



7.4.3 方法一：使用 LinkedHashMap

在 java 语言中，其实 java.util 下已经给我们封装好了这样的一个数据结构，就是“链式哈希表”——LinkedHashMap。它本身继承了 HashMap，而它的节点 Entry 除了继承自 HashMap.Node，还定义了 before 和 after 两个指针，从而实现了双向链表。

代码如下：

```
public class LRUCache extends LinkedHashMap<Integer, Integer>{
    private int capacity;
    public LRUCache(int capacity) {
        super(capacity, 0.75f,true);
        this.capacity = capacity;
    }
    public int get(int key) {
        return super.get(key);
    }
    public void put(int key, int value) {
        super.put(key, value);
    }
    @Override
    protected boolean removeEldestEntry(Map.Entry<Integer, Integer>
```

```
eldest) {
        return size() > capacity;
    }
}
```

7.4.4 方法二：自定义哈希表+双向链表

上面的实现虽然简单，但是有取巧的嫌疑，如果在真正的面试中给出这样的代码，很可能面试官是无法满意的。我们需要做的，还是自己实现一个简单的双向链表，而不是直接套用语言自带的封装数据结构。

代码如下：

```
public class LRUCache {
    class Node {
        int key;
        int value;
        Node prev;
        Node next;
        public Node() {}
        public Node(int key, int value) {
            this.key = key;
            this.value = value;
        }
    }
    private HashMap<Integer, Node> hashMap = new HashMap<Integer, Node>();
    private int capacity;
    private int size;
    private Node head, tail;

    public LRUCache(int capacity) {
```

```
        this.capacity = capacity;
        this.size = 0;
        head = new Node();
        tail = new Node();
        head.next = tail;
        tail.prev = head;
    }
    public int get(int key) {
        Node node = hashMap.get(key);
        if (node == null) {
            return -1;
        }
        moveToTail(node);
        return node.value;
    }

    public void put(int key, int value) {
        Node node = hashMap.get(key);
        if (node != null) {
            node.value = value;
            moveToTail(node);
        }
        else {
            Node newNode = new Node(key, value);
            hashMap.put(key, newNode);
            addToTail(newNode);
            size ++;
            if (size > capacity) {
                Node tail = removeHead();
```

```
                hashMap.remove(tail.key);
                size --;
            }
        }
    }
    // 将一个节点移到双向链表末尾
    private void moveToTail(Node node) {
        removeNode(node);
        addToTail(node);
    }
    // 通用方法：删除双向链表中一个节点
    private void removeNode(Node node){
        node.prev.next = node.next;
        node.next.prev = node.prev;
    }
    // 向双向链表末尾，添加一个节点
    private void addToTail(Node node) {
        node.next = tail;    // tail 始终是哑节点，node 插在它前面
        node.prev = tail.prev;
        tail.prev.next = node;    // 原先的末尾节点，next 改为 node
        tail.prev = node;    // tail 的 prev 改为 node
    }
    // 删除双向链表的头节点
    private Node removeHead() {
        Node realHead = head.next;
        removeNode(realHead);
        return realHead;
    }
}
```

复杂度分析

时间复杂度：O(1)。因为使用了 HashMap 和双向链表，对于 put 和 get 操作都可以在 O(1)时间完成。

空间复杂度：O(capacity)，因为哈希表和双向链表最多存储 capacity+1 个元素（超出缓存容量时，大小为 capacity+1）。

第八章 栈和队列相关问题讲解

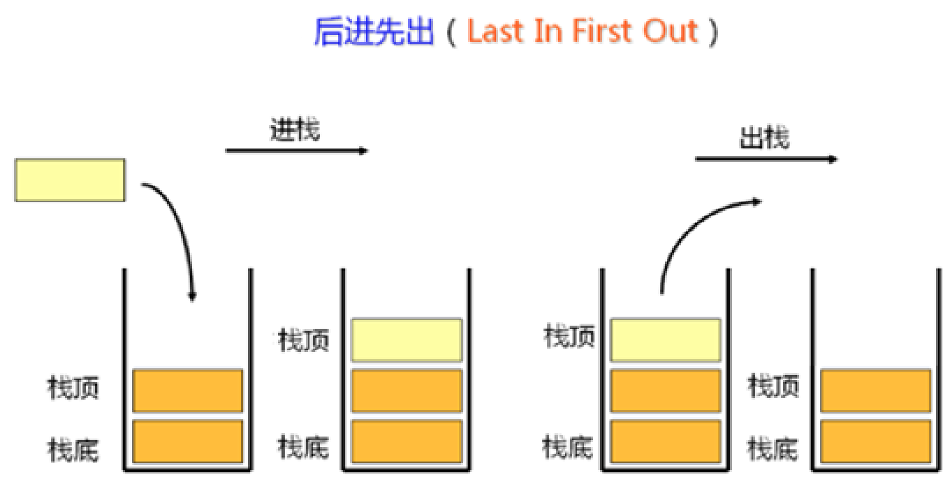
8.1 栈和队列数据结构复习

8.1.1 栈（Stack）

栈（Stack）又名堆栈，它是一种重要的数据结构。从数据结构角度看，栈也是线性表，其特殊性在于栈的基本操作是线性表操作的子集，它是操作受限的线性表，因此，可称为限定性的数据结构。

栈被限定仅在表尾进行插入或删除操作。表尾称为栈顶，相应地，表头称为栈底。所以栈具有“后进先出”（LIFO）的特点。

栈的基本操作除了在栈顶进行插入（入栈，push）和删除（出栈，pop）外，还有栈的初始化，判断是否为空以及取栈顶元素等。

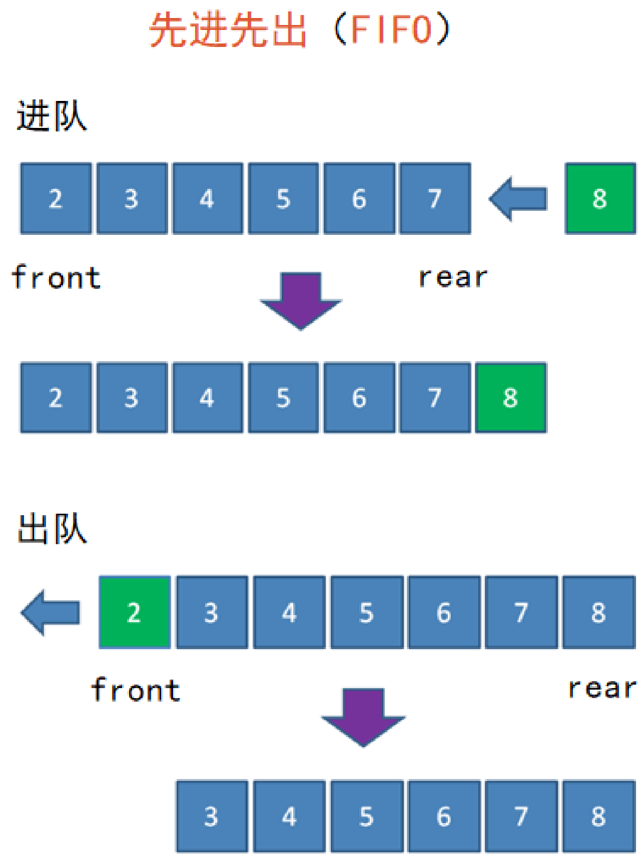


8.1.2 队列

队列（Queue）是一种先进先出（FIFO，First-In-First-Out）的线性表。

在具体应用中通常用链表或者数组来实现。队列只允许在后端（称为 rear）进行插入操作，在前端（称为 front）进行删除操作。

队列的操作方式和堆栈类似，唯一的区别在于队列只允许新数据在后端进行添加。

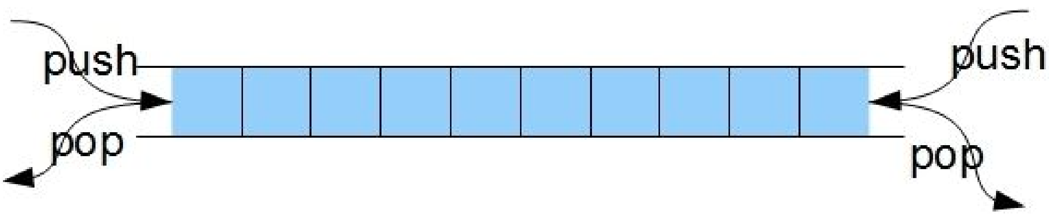


- 双端队列（Deque:double ended queue）

双端队列，是限定插入和删除操作在表的两端进行的线性表。

队列的每一端都能够插入数据项和移除数据项。

相对于普通队列，双端队列的入队和出队操作在两端都可进行。所以，双端队列同时具有队列和栈的性质。



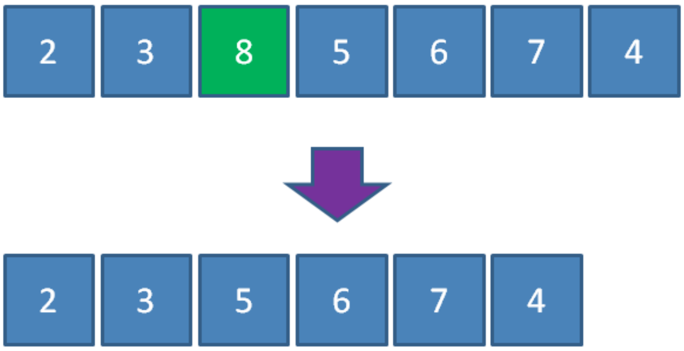
- 优先队列

优先队列不再遵循先入先出的原则，而是分为两种情况：

最大优先队列，无论入队顺序，当前最大的元素优先出队。

最小优先队列，无论入队顺序，当前最小的元素优先出队。

比如有一个最大优先队列，它的最大元素是 8，那么虽然元素 8 并不是队首元素，但出队的时候仍然让元素 8 首先出队：

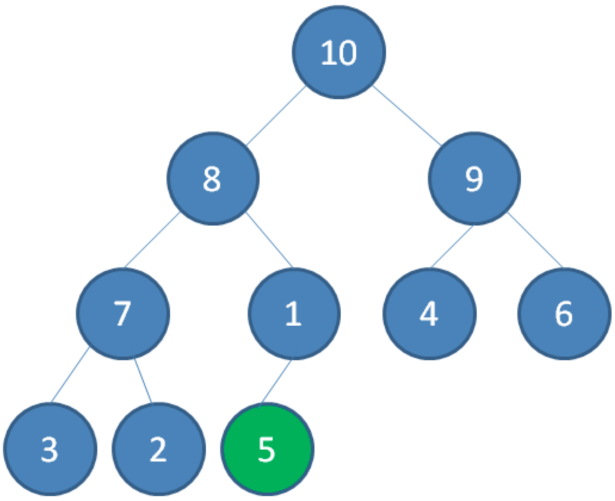


要满足以上需求，利用线性数据结构并非不能实现，但是时间复杂度较高，需要遍历所有元素，最坏时间复杂度 O（n），并不是最理想的方式。

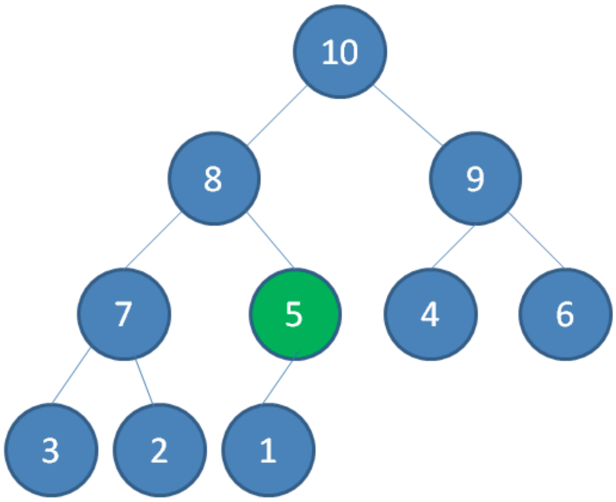
因此，一般是用**大顶堆**（Max Heap，有时也叫最大堆）来实现最大优先队列，每一次入队操作就是堆的插入操作，每一次出队操作就是删除堆顶节点。

入队操作：

1. 插入新节点 5

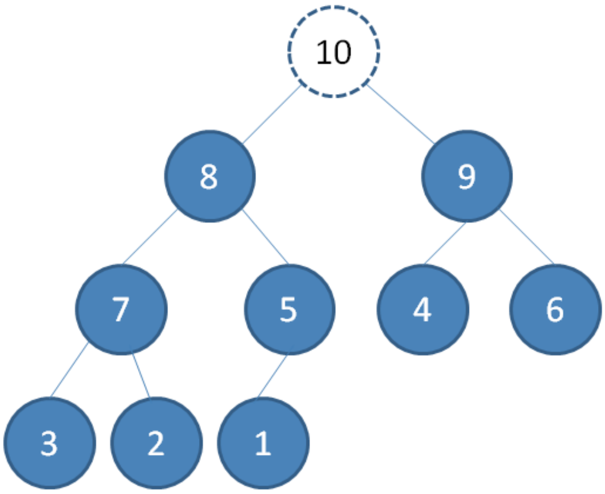


2. 新节点 5 上浮到合适位置。

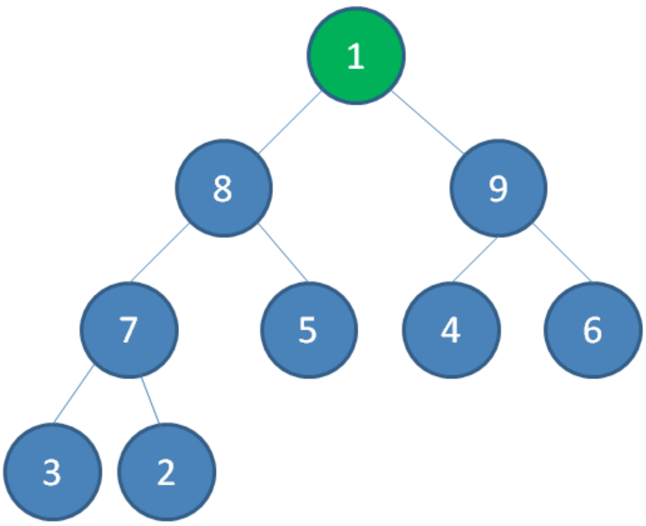


出队操作：

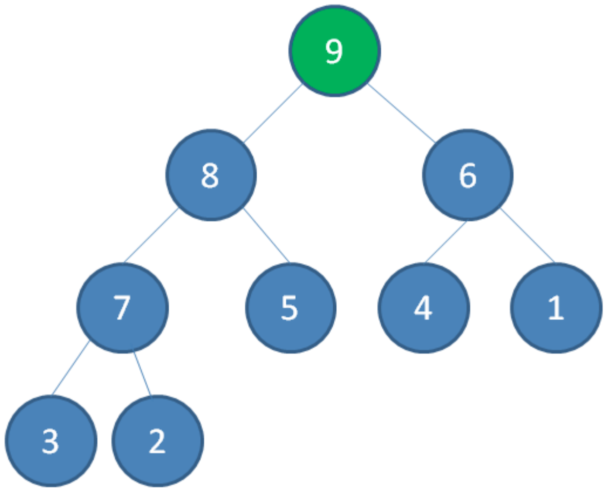
1. 把原堆顶节点 10“出队”



2. 最后一个节点 1 替换到堆顶位置



3.节点 1 下沉，节点 9 成为新堆顶



二叉堆节点上浮和下沉，操作次数不会超过数的深度，所以时间复杂度都是 O(logn)。那么优先队列，入队和出队的时间复杂度，也是 O(logn)。

8.2 使用队列实现栈（#225）

8.2.1 题目说明

使用队列实现栈的下列操作：

- push(x) -- 元素 x 入栈
- pop() -- 移除栈顶元素
- top() -- 获取栈顶元素
- empty() -- 返回栈是否为空

注意:

- 你只能使用队列的基本操作-- 也就是 push to back, peek/pop from front, size, 和 is empty 这些操作是合法的。
- 你所使用的语言也许不支持队列。 你可以使用 list 或者 deque（双端队列）来模拟一个队列，只要是标准的队列操作即可。
- 你可以假设所有操作都是有效的（例如, 对一个空的栈不会调用 pop 或者 top 操

作）。

8.2.2 分析

这道题目涉及到栈和队列两种数据结构。它们的共同特点是，数据元素以线性序列的方式存储；区别在于，元素进出的方式不同。

队列本身对数据元素的保存，是完全符合数据到来次序的，同时也保持这个顺序依次出队。而弹栈操作的实现，是要删除最后进入的数据，相当于反序弹出。

实现的基本思路是，我们可以用一个队列保存当前所有的数据，以它作为栈的物理基础；而为了保证后进先出，我们在数据入队之后，就把它直接移动到队首。

8.2.3 方法一：两个队列实现

可以增加一个队列来做辅助。我们记原始负责存储数据的队列为 queue1，新增的辅助队列为 queue2。

- 当一个数据 x 压栈时，我们不是直接让它进入 queue1，而是先在 queue2 做一个缓存。默认 queue2 中本没有数据，所以当前元素一定在队首。

  queue1：a b

  queue2：x

- 接下来，就让 queue1 执行出队操作，把之前的数据依次输出，同时全部添加到 queue2 中来。这样，queue2 就实现了把新元素添加到队首的目的。

  queue1：

  queue2：x a b

- 最后，我们将 queue2 的内容复制给 queue1 做存储，然后清空 queue2。在代码上，这个实现非常简单，只要交换 queue1 和 queue2 指向的内容即可。

  queue1：x a b

  queue2：

而对于弹栈操作，只要直接让 queue1 执行出队操作，删除队首元素就可以了。

代码如下：

```
public class MyStack {
    Queue<Integer> queue1;
    Queue<Integer> queue2;
    public MyStack() {
        queue1 = new LinkedList<>();
        queue2 = new LinkedList<>();
    }
    public void push(int x) {
        queue2.offer(x);
        while (!queue1.isEmpty()){
            queue2.offer( queue1.poll() );
        }
        Queue<Integer> temp = queue1;
        queue1 = queue2;
        queue2 = temp;
    }
    public int pop() {
        return queue1.poll();
    }
    public int top() {
        return queue1.peek();
    }
    public boolean empty() {
        return queue1.isEmpty();
    }
}
```

复杂度分析

时间复杂度：入栈操作 O(n)，其余操作都是 O(1)。

push：入栈操作，需要将 queue1 中的 n 个元素出队，并入队 n+1 个元素到 queue2，总计 2n+1 次操作。每次出队和入队操作的时间复杂度都是 O(1)，因此入栈操作的时间复杂度是 O(n)。

pop：出栈操作，只是将 queue1 的队首元素出队，时间复杂度是 O(1)。

top：获得栈顶元素，对应获得 queue1 的队首元素，时间复杂度是 O(1)。

isEmpty：判断栈是否为空，只需要判断 queue1 是否为空，时间复杂度是 O(1)。

空间复杂度：O(n)，其中 n 是栈内的元素。需要使用两个队列存储栈内的元素。

8.2.4 方法二：一个队列实现

当一个新的元素 x 压栈时，其实我们可以不借助辅助队列，而是让它直接入队 queue1，它会添加在队尾。然后接下来，只要将之前的所有数据依次出队、再重新入队添加进 queue1，就自然让 x 移动到队首了。

代码如下：

```java
public class MyStack2 {

    Queue<Integer> queue;

    public MyStack2() {

        queue = new LinkedList<>();

    }

    public void push(int x) {

```

```
        int l = queue.size();

        queue.offer(x);

        for (int i = 0; i < l; i++){

            queue.offer( queue.poll() );

        }

    }

    public int pop() {

        return queue.poll();

    }

    public int top() {

        return queue.peek();

    }

    public boolean empty() {

        return queue.isEmpty();
```

```
    }

}
```

复杂度分析

时间复杂度：入栈操作 O(n)，其余操作都是 O(1)。

push：入栈操作，需要将 queue1 中的 n 个元素出队，并入队 n+1 个元素到 queue2，总计 2n+1 次操作。每次出队和入队操作的时间复杂度都是 O(1)，因此入栈操作的时间复杂度是 O(n)。

pop：出栈操作。只是将 queue1 的队首元素出队，时间复杂度是 O(1)。

top：获得栈顶元素，对应获得 queue1 的队首元素，时间复杂度是 O(1)。

isEmpty：判断栈是否为空，只需要判断 queue1 是否为空，时间复杂度是 O(1)。

空间复杂度：O(n)，其中 n 是栈内的元素。需要使用两个队列存储栈内的元素。

8.3 使用栈实现队列（#232）

8.3.1 题目说明

请你仅使用两个栈实现先入先出队列。队列应当支持一般队列的支持的所有操作（push、pop、peek、empty）：

实现 MyQueue 类：

- void push(int x) 将元素 x 推到队列的末尾
- int pop() 从队列的开头移除并返回元素
- int peek() 返回队列开头的元素
- boolean empty() 如果队列为空，返回 true ；否则，返回 false

说明：

- 你只能使用标准的栈操作 —— 也就是只有 push to top, peek/pop from top, size,

和 is empty 操作是合法的。

- 你所使用的语言也许不支持栈。你可以使用 list 或者 deque（双端队列）来模拟一个栈，只要是标准的栈操作即可。

进阶：

你能否实现每个操作均摊时间复杂度为 O(1) 的队列？换句话说，执行 n 个操作的总时间复杂度为 O(n) ，即使其中一个操作可能花费较长时间。

示例：

```
输入：
["MyQueue", "push", "push", "peek", "pop", "empty"]
[[], [1], [2], [], [], []]
输出：
[null, null, null, 1, 1, false]
```

解释：

```
MyQueue myQueue = new MyQueue();
myQueue.push(1); // queue is: [1]
myQueue.push(2); // queue is: [1, 2] (leftmost is front of the queue)
myQueue.peek(); // return 1
myQueue.pop(); // return 1, queue is [2]
myQueue.empty(); // return false
```

提示：

- 1 <= x <= 9
- 最多调用 100 次 push、pop、peek 和 empty
- 假设所有操作都是有效的 （例如，一个空的队列不会调用 pop 或者 peek 操作）

8.3.2 分析

我们要用栈来实现队列。一个队列是 **FIFO** 的，但一个栈是 **LIFO** 的。为了满足队列的

FIFO 的特性，我们需要将入栈的元素次序进行反转，这样在出队时就可以按照入队顺序依次弹出了。

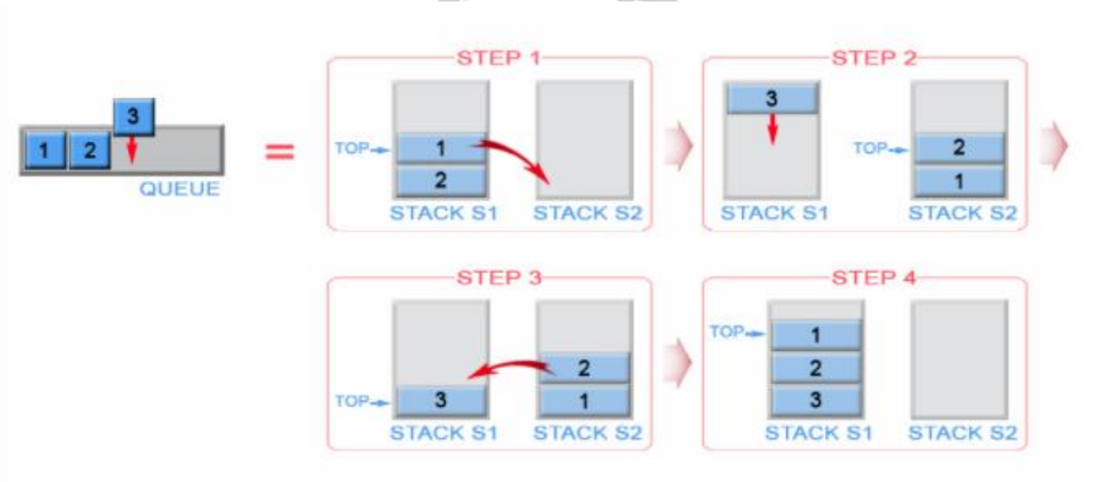
想要反转，简单的想法是只要把所有元素依次弹出，并压入另一个栈，自然就变成本来栈底元素到了栈顶了。所以我们的实现，需要用到两个栈。

8.3.3 方法一：入队时反转

一种直观的思路是，最终的栈里，按照“自顶向下”的顺序保持队列。也就是说，栈顶元素是最先入队的元素，而最新入队的元素要压入栈底。

我们可以用一个栈来存储元素的最终顺序（队列顺序），记作 stack1；用另一个进行辅助反转，记作 stack2。

最简单的实现，就是直接用 stack2，来缓存原始压栈的元素。每次调用 push，就把 stack1 中的元素先全部弹出并压入 stack2，然后把新的元素也压入 stack2；这样 stack2 就是完全按照原始顺序入栈的。最后再把 stack2 中的元素全部弹出并压入 stack1，进行反转。



Pushing element "3" to the queue

代码如下：

```
public class MyQueue {
    Stack<Integer> stack1;
    Stack<Integer> stack2;
    public MyQueue() {
```

```
        stack1 = new Stack<>();
        stack2 = new Stack<>();
    }
    public void push(int x) {
        while (!stack1.isEmpty()){
            stack2.push(stack1.pop());
        }
        stack2.push(x);
        while (!stack2.isEmpty()){
            stack1.push(stack2.pop());
        }
    }
    public int pop() {
        return stack1.pop();
    }
    public int peek() {
        return stack1.peek();
    }
    public boolean empty() {
        return stack1.isEmpty();
    }
}
```

复杂度分析

- 入队

时间复杂度：O(n)

除新元素之外，所有元素都会被压入两次，弹出两次。新元素被压入两次，弹出一次。（当然，我们可以稍作改进，在 stack1 清空之后把新元素直接压入，就只压入一次了）

这个过程产生了 4n+3 次操作，其中 n 是队列的大小。由于入栈操作和弹出操作的时

间复杂度为 O(1)， 所以时间复杂度为 O(n)。

空间复杂度：O(n)

需要额外的内存来存储队列中的元素。

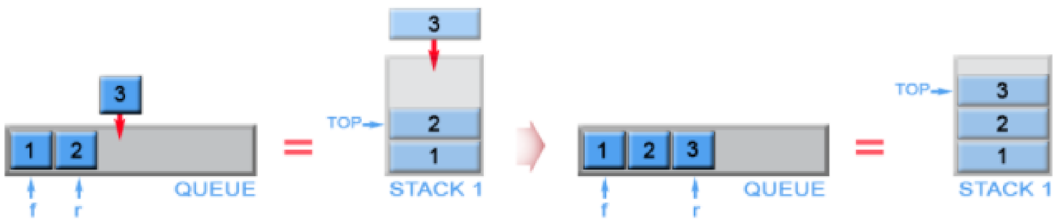
- 其它操作（pop、peek、isEmpty）

时间复杂度：O(1)

空间复杂度：O(1)

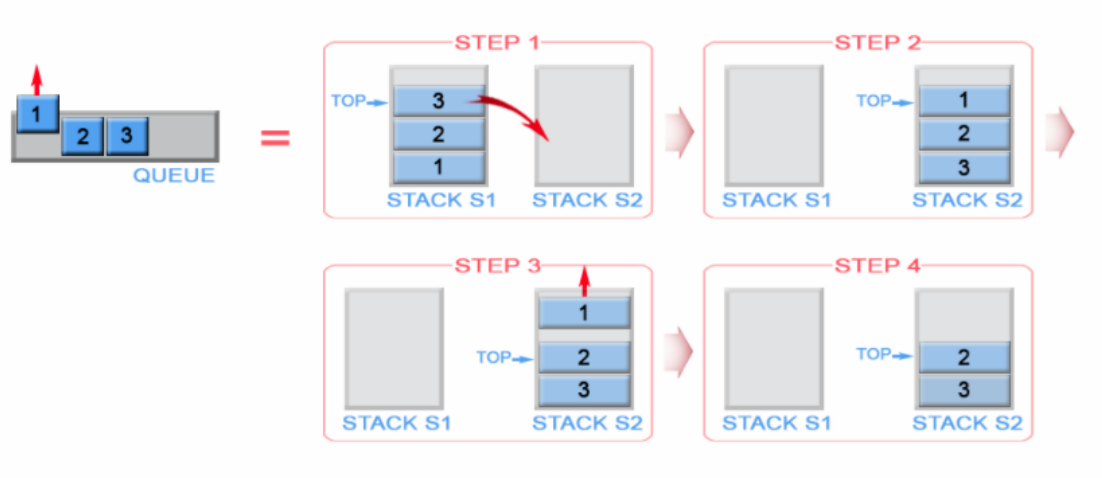
8.3.4 方法二：出队时反转

可以不要在入队时反转，而是在出队时再做处理。



Pushing element "3" to the queue

执行出队操作时，我们想要弹出的是 stack1 的栈底元素。所以需要将 stack1 中所有元素弹出，并压入 stack2，然后弹出 stack2 的栈顶元素。



Popping element "1" from the queue

我们观察可以发现，stack2 中的元素，其实就是保持着队列顺序的，所以完全没必要将它们再压回 stack1，下次出队时，我们只要直接弹出 stack2 中的栈顶元素就可以了。

代码实现如下：

```
public class MyQueue2 {
    Stack<Integer> stack1;
    Stack<Integer> stack2;
    public MyQueue2() {
        stack1 = new Stack<>();
        stack2 = new Stack<>();
    }
    public void push(int x) {
        stack1.push(x);
    }
    public int pop() {
        if (stack2.isEmpty()){
            while (!stack1.isEmpty()){
                stack2.push(stack1.pop());
            }
```

```
        }
        return stack2.pop();
    }
    public int peek() {
        if (stack2.isEmpty()){
            while (!stack1.isEmpty()){
                stack2.push(stack1.pop());
            }
        }
        return stack2.peek();
    }
    public boolean empty() {
        return stack1.isEmpty() && stack2.isEmpty();
    }
}
```

复杂度分析

- 入队（push）

时间复杂度：O(1)。向栈压入元素的时间复杂度为 O(1)

空间复杂度：O(n)。需要额外的内存（stack1 和 stack2 共同存储）来存储队列元素。

- 出队（pop）

时间复杂度： 摊还复杂度 O(1)，最坏情况下的时间复杂度 O(n)

在最坏情况下，stack2 为空，算法需要执行 while 循环进行反转。具体过程是从 stack1 中弹出 n 个元素，然后再把这 n 个元素压入 stack2，在这里 n 代表队列的大小。这个过程产生了 2n 步操作，时间复杂度为 O(n)。

但当 stack2 非空时，只需要直接弹栈，算法就只有 O(1) 的时间复杂度。均摊下来，摊还复杂度为 O(1)。

空间复杂度 ：O(1)

- 取队首元素（peek）和判断是否为空（empty）

时间复杂度：O(1)

空间复杂度：O(1)

8.3.5 摊还复杂度分析

摊还分析（Amortized Analysis，均摊法），用来评价某个数据结构的一系列操作的平均代价。

对于一连串操作而言，可能某种情况下某个操作的代价特别高，但总体上来看，也并非那么糟糕，可以形象的理解为把高代价的操作“分摊”到其他操作上去了，要求的就是均摊后的平均代价。

摊还分析的核心在于，最坏情况下的操作一旦发生了一次，那么在未来很长一段时间都不会再次发生，这样就会均摊每次操作的代价。

摊还分析与平均复杂度分析的区别在于，平均情况分析是平均所有的输入。而摊还分析是平均操作。在摊还分析中，不涉及概率，并且保证在最坏情况下每一个操作的平均性能。

所以摊还分析，往往会用在某一数据结构的操作分析上。

8.4 有效的括号（#20）

8.4.1 题目说明

给定一个只包括 '('，')'，'{'，'}'，'['，']' 的字符串，判断字符串是否有效。

有效字符串需满足：

1. 左括号必须用相同类型的右括号闭合。
2. 左括号必须以正确的顺序闭合。

注意空字符串可被认为是有效字符串。

示例 1:

```
输入: "()"
输出: true
```

示例 2:

```
输入: "()[]{}"
输出: true
```

示例 3:

```
输入: "(]"
输出: false
```

示例 4:

```
输入: "([)]"
输出: false
```

示例 5:

```
输入: "{[]}"
输出: true
```

8.4.2 分析

判断括号的有效性，这是一个非常经典的问题。

由于给定字符串中只包含 '('，')'，'{'，'}'，'['，']'，所以我们不需要额外考虑非法字符的问题。

对于合法的输入字符，关键在于遇到一个“左括号”时，我们会希望在后续的遍历中，遇到一个相同类型的“右括号”将其闭合。

由于规则是：**后遇到的左括号，要先闭合**，因此我们想到，利用一个**栈**可以实现这个功能，将左括号放入栈顶，遇到右括号时弹出就可以了。

8.4.3 具体实现

代码实现非常简单：我们可以创建一个栈，然后遍历字符串。遇到左括号，就压栈；遇到右括号，就判断和当前栈顶的左括号是否匹配，匹配就弹栈，不匹配直接返回 false。

代码如下：

```
public class ValidParentheses {
    public boolean isValid(String s) {
        Deque<Character> stack = new LinkedList<>();
        for (int i = 0; i < s.length(); i++){
            char c = s.charAt(i);
            if ( c == '(' ){
                stack.push(')');
            } else if ( c == '[' ){
                stack.push(']');
            } else if ( c == '{' ){
                stack.push('}');
            } else {
                if (stack.isEmpty()) return false;
                char right = stack.pop();
                if (c != right) return false;
            }
        }
        return stack.isEmpty();
    }
}
```

复杂度分析

时间复杂度：O(n)，其中 n 是字符串 s 的长度。只需要遍历一次字符串。

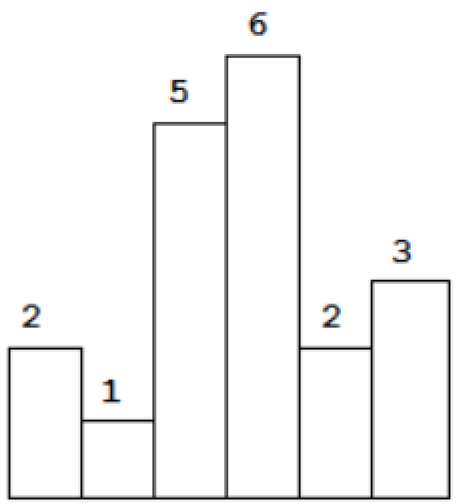
空间复杂度：O(n)。栈中最多会保存字符串中所有的左括号，数量为 O(n)。

8.5 柱状图中最大的矩形（#84）

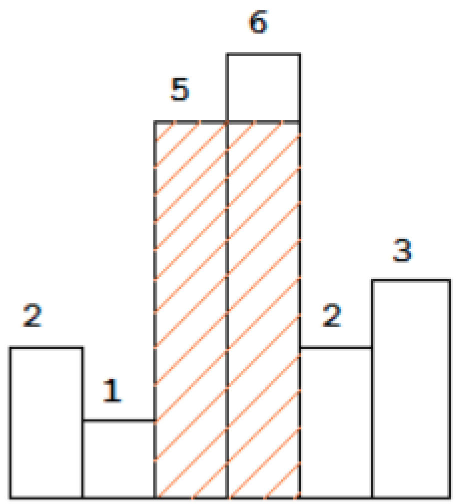
8.5.1 题目说明

给定 n 个非负整数，用来表示柱状图中各个柱子的高度。每个柱子彼此相邻，且宽度为 1 。

求在该柱状图中，能够勾勒出来的矩形的最大面积。



以上是柱状图的示例，其中每个柱子的宽度为 1，给定的高度为 [2,1,5,6,2,3]。



图中阴影部分为所能勾勒出的最大矩形面积，其面积为 10 个单位。

示例:

```
输入: [2,1,5,6,2,3]

输出: 10
```

8.5.2 分析

题目要求计算最大矩形面积，我们可以发现，关键其实就在于确定矩形的“宽”和“高”（即矩形面积计算中的长和宽）。

而宽和高两者间又有制约条件：一定宽度范围内的高，就是最矮那个柱子的高度。

8.5.3 方法一：暴力法

一个简单的思路，就是遍历所有可能的宽度。也就是说，以每个柱子都作为矩形的左右边界进行计算，取出所有面接中最大的那个。

代码如下：

```java
public class LargestRectangleInHistogram {
    public int largestRectangleArea1(int[] heights) {
        int largestArea = 0;
        for ( int left = 0; left < heights.length; left++ ){
            int currHeight = heights[left];
            for ( int right = left; right < heights.length; right++ ){
                currHeight =  (heights[right] < currHeight) ?
heights[right] : currHeight;
                int currArea = (right - left + 1) * currHeight;
                largestArea = (currArea > largestArea) ? currArea :
largestArea;
            }
        }
        return largestArea;
```

```
    }
}
```

复杂度分析

时间复杂度：O(N^2)。很明显，代码中用到了双重循环，需要耗费平方时间复杂度来做遍历计算。这个复杂度显然是比较高的。

空间复杂度：O(1)。只用到了一些辅助变量。

8.5.4 方法二：双指针

我们可以首先遍历数组，以当前柱子的高度，作为考察的矩阵“可行高度”。然后定义左右两个指针，以当前柱子为中心向两侧探寻，找到当前高度的左右边界。

左右边界的判断标准，就是出现了比当前高度矮的柱子，或者到达了数组边界。

代码实现如下：

```
public int largestRectangleArea(int[] heights) {
    int largestArea = 0;
    for ( int i = 0; i < heights.length; i++ ){
        int height = heights[i];
        int left = i, right = i;
        // 寻找左边界
        while ( left >= 0 ){
            if ( heights[left] < height ) break;
            left --;
        }
        // 寻找右边界
        while ( right < heights.length ){
            if ( heights[right] < height ) break;
            right ++;
```

```
        }
        int width = right - left - 1;
        int currArea = height * width;
        largestArea = currArea > largestArea ? currArea : largestArea;
    }
    return largestArea;
}
```

复杂度分析

时间复杂度：O(N^2)。尽管少了一重循环，但在内部依然要去暴力寻找左右边界，这个操作最好情况下时间复杂度为 O(1)，最坏情况下为 O(N)，平均为 O(N)。所以整体的平均时间复杂度仍然是 O(N^2)。

空间复杂度：O(1)。只用到了一些辅助变量。

8.5.5 方法三：双指针优化

在双指针法寻找左右边界的过程中我们发现，如果当前柱子比前一个柱子高，那么它的左边界就是前一个柱子；如果比前一个柱子矮，那么可以跳过之前确定更高的那些柱子，直接从前一个柱子的左边界开始遍历。

这就需要我们记录下每一个柱子对应的左边界，这可以单独用一个数组来保存。

代码如下：

```
public int largestRectangleArea(int[] heights) {
    int n = heights.length;
    int[] lefts = new int[n];
    int[] rights = new int[n];
    int largestArea = 0;
    for ( int i = 0; i < n; i++ ){
        int height = heights[i];
        int left = i - 1;
```

```
        // 向左移动，寻找左边界
        while ( left >= 0 ){
            if ( heights[left] < height ) break;
            left = lefts[left];
        }
        lefts[i] = left;
    }
    for ( int i = n - 1; i >= 0; i-- ){
        int height = heights[i];
        int right = i + 1;
        // 向右移动，寻找右边界
        while ( right < n ){
            if ( heights[right] < height ) break;
            right = rights[right];
        }
        rights[i] = right;
    }
    for ( int i = 0; i < n; i++ ){
        int currArea = ( rights[i] - lefts[i] - 1 ) * heights[i];
        largestArea = currArea > largestArea ? currArea : largestArea;
    }
    return largestArea;
}
```

复杂度分析

时间复杂度：O(N)。我们发现，while 循环内的判断比对总体数量其实是有限的。每次比对，或者是遍历到一个新元素的时候，或者是之前判断发现当前柱子较矮，需要继续和前一个柱子的左边界进行比较。所以总的时间复杂度是 O(N)。

空间复杂度：O(N)。用到了长度为 n 的数组来保存左右边界。

8.5.6 方法四：使用单调栈

从上面的算法中我们可以发现，“找左边界”最重要的，其实就是排除左侧不可能的那些元素，跳过它们不再遍历。

所以我们可以考虑用这样一个数据结构，来保存当前的所有“候选左边界”。

当遍历到一个高度时，就让它和“候选列表”中的高度比较：如果发现它比之前的候选大，可以直接追加在后面；而如果比之前的候选小，就应该删除之前更大的候选。最终，保持一个按照顺序、单调递增的候选序列。

过程中应该按照顺序，先比对最新的候选、再比对较老的候选。显然，我们可以用一个栈来实现这样的功能。

栈中存放的元素具有单调性，这就是经典的数据结构**单调栈**了。

我们用一个具体的例子 [6,7,5,2,4,5,9,3] 来理解单调栈。

我们需要求出每一根柱子的左侧且最近的小于其高度的柱子。初始时的栈为空。

（1）我们枚举 6，因为栈为空，所以 6 左侧的柱子是“哨兵”，位置为 -1。随后我们将 6 入栈。

栈：[6(0)]。（这里括号内的数字表示柱子在原数组中的位置索引）

（2）我们枚举 7，由于 6<7，因此不会移除栈顶元素，所以 7 左侧的柱子是 6，位置为 0。随后我们将 7 入栈。

栈：[6(0), 7(1)]

（3）我们枚举 5，由于 7≥5，因此移除栈顶元素 7。同样地， 6≥5，再移除栈顶元素 6。此时栈为空，所以 5 左侧的柱子是「哨兵」，位置为−1。随后我们将 5 入栈。

栈：[5(2)]

（4）接下来的枚举过程也大同小异。我们枚举 2，移除栈顶元素 5，得到 2 左侧的柱子是「哨兵」，位置为 −1。将 2 入栈。

栈：[2(3)]

（5）我们枚举 4，5 和 9，都不会移除任何栈顶元素，得到它们左侧的柱子分别是 2，4 和 5，位置分别为 3，4 和 5。将它们入栈。

栈：[2(3), 4(4), 5(5), 9(6)]

（6）我们枚举 3，依次移除栈顶元素 9，5 和 4，得到 3 左侧的柱子是 2，位置为 3。将 3 入栈。

栈：[2(3), 3(7)]

这样一来，我们得到它们左侧的柱子编号分别为 [−1,0,−1,−1,3,4,5,3]。

用相同的方法，我们从右向左进行遍历，也可以得到它们右侧的柱子编号分别为 [2,2,3,8,7,7,7,8]，这里我们将位置 8 看作右侧的“哨兵”。

在得到了左右两侧的柱子之后，我们就可以计算出每根柱子对应的左右边界，并求出答案了。

代码如下：

```
public int largestRectangleArea(int[] heights) {
    int n = heights.length;
    int[] lefts = new int[n];
    int[] rights = new int[n];
    int largestArea = 0;
    // 定义一个栈，保存“候选列表”
    Stack<Integer> stack = new Stack<>();
    // 遍历所有柱子，计算左右边界
    for ( int i = 0; i < n; i++ ){
        while ( !stack.isEmpty() && heights[stack.peek()] >= heights[i] ){
            stack.pop();
        }
        lefts[i] = stack.isEmpty() ? -1 : stack.peek();
        stack.push(i);
    }
    stack.clear();
    for ( int i = n - 1; i >= 0; i-- ){
        while ( !stack.isEmpty() && heights[stack.peek()] >= heights[i] ){
```

```java
            stack.pop();
        }
        rights[i] = stack.isEmpty() ? n : stack.peek();
        stack.push(i);
    }
    for ( int i = 0; i < n; i++ ){
        int currArea = ( rights[i] - lefts[i] - 1 ) * heights[i];
        largestArea = currArea > largestArea ? currArea : largestArea;
    }
    return largestArea;
}
```

复杂度分析

时间复杂度：O(N)。每一个位置元素只会入栈一次（在枚举到它时），并且最多出栈一次。因此当我们从左向右/从右向左遍历数组时，对栈的操作的次数就为 O(N)。所以单调栈的总时间复杂度为 O(N)。

空间复杂度：O(N)。用到了单调栈，大小为 O(N)。

8.5.7 方法五：单调栈优化

当一个柱子高度比栈顶元素小时，我们会弹出栈顶元素，这就说明当前柱子就是栈顶元素对应柱子的右边界。所以我们可以只遍历一次，就求出答案。

代码如下：

```java
public int largestRectangleArea(int[] heights) {
    int n = heights.length;
    int[] lefts = new int[n];
    int[] rights = new int[n];
```

```java
    for ( int i = 0; i < n; i++ ) rights[i] = n;


    int largestArea = 0;
    Stack<Integer> stack = new Stack<>();
    for ( int i = 0; i < n; i++ ){
        while ( !stack.isEmpty() && heights[stack.peek()] >= heights[i] ){
            rights[stack.peek()] = i;
            stack.pop();
        }
        lefts[i] = stack.isEmpty() ? -1 : stack.peek();
        stack.push(i);
    }
    for ( int i = 0; i < n; i++ ){
        int currArea = ( rights[i] - lefts[i] - 1 ) * heights[i];
        largestArea = currArea > largestArea ? currArea : largestArea;
    }
    return largestArea;
}
```

复杂度分析

时间复杂度：O(N)。只有一次遍历，同样每个位置入栈一次、最多出栈一次。

空间复杂度：O(N)。用到了单调栈，大小为 O(N)。

第九章 排序相关问题讲解

9.1 排序算法复习

常见的排序算法可以分为两大类：比较类排序，和非比较类排序。

- 比较类排序：通过比较来决定元素间的相对次序，由于其时间复杂度不能突破 O(nlogn)，因此也称为非线性时间比较类排序。
- 非比较类排序：不通过比较来决定元素间的相对次序，它可以突破基于比较排序的时间下界，以线性时间运行，因此也称为线性时间非比较类排序。主要思路是通过将数值以哈希（hash）或分桶（bucket）的形式直接映射到存储空间来实现的。

算法复杂度总览

排序方法	时间复杂度（平均）	时间复杂度（最坏）	时间复杂度（最好）	空间复杂度	稳定性
插入排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定
希尔排序	$O(n^{1.3})$	$O(n^2)$	$O(n)$	$O(1)$	不稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定
堆排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(1)$	不稳定
冒泡排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定
快速排序	$O(nlog_2n)$	$O(n^2)$	$O(nlog_2n)$	$O(nlog_2n)$	不稳定
归并排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	$O(n)$	稳定
计数排序	$O(n+k)$	$O(n+k)$	$O(n+k)$	$O(n+k)$	稳定
桶排序	$O(n+k)$	$O(n^2)$	$O(n)$	$O(n+k)$	稳定
基数排序	$O(n*k)$	$O(n*k)$	$O(n*k)$	$O(n+k)$	稳定

9.1.1 选择排序（Selection Sort）

选择排序是一种简单直观的排序算法。

它的工作原理：首先在未排序序列中找到最小（大）元素，存放到排序序列的起始位置，然后，再从剩余未排序元素中继续寻找最小（大）元素，然后追加到已排序序列的末尾。以

此类推，直到所有元素均排序完毕。

9.1.2 冒泡排序（Bubble Sort）

冒泡排序也是一种简单的排序算法。

它的基本原理是：重复地扫描要排序的数列，一次比较两个元素，如果它们的大小顺序错误，就把它们交换过来。这样，一次扫描结束，我们可以确保最大（小）的值被移动到序列末尾。

这个算法的名字由来，就是因为越小的元素会经由交换，慢慢"浮"到数列的顶端。

9.1.3 插入排序（Insertion Sort）

插入排序的算法，同样描述了一种简单直观的排序。

它的工作原理是：构建一个有序序列。对于未排序数据，在已排序序列中从后向前扫描，找到相应位置并插入。

以上三种简单排序算法，因为需要双重循环，所以时间复杂度均为 O(n^2)。排序过程中，只需要额外的常数空间，所以空间复杂度均为 O(1)。

9.1.4 希尔排序（Shell Sort）

1959 年由 Shell 发明，是第一个突破 $O(n^2)$的排序算法，是简单插入排序的改进版。

它与插入排序的不同之处在于，它会优先比较距离较远的元素。希尔排序又叫**缩小增量排序**。

希尔排序在数组中采用**跳跃式分组**的策略，通过某个增量将数组元素划分为若干组，然后分组进行插入排序，随后逐步缩小增量，继续按组进行插入排序操作，直至增量为 1。

希尔排序中对于增量序列的选择十分重要，直接影响到希尔排序的性能。一些经过优化的增量序列如 Hibbard 经过复杂证明可使得最坏时间复杂度为 O(n^3/2)。

9.1.5 归并排序（Merge Sort）

归并排序是建立在归并操作上的一种有效的排序算法。该算法是采用分治法（Divide and Conquer）的一个非常典型的应用。

将已有序的子序列合并，得到完全有序的序列；即先使每个子序列有序，再使子序列段间有序。若将两个有序表合并成一个有序表，称为 2-路归并。

归并排序的时间复杂度是 O(nlogn)。代价是需要额外的内存空间。

9.1.6 快速排序（Quick Sort）

快速排序的基本思想：通过一趟排序，将待排记录分隔成独立的两部分，其中一部分记录的关键字均比另一部分的关键字小，则可分别对这两部分记录继续进行排序，以达到整个序列有序。

可以看出，快排也应用了分治思想，一般会用递归来实现。

快速排序使用分治法来把一个串（list）分为两个子串（sub-lists）。具体算法描述如下：

- 从数列中挑出一个元素，称为"基准"（pivot，中心，支点）；
- 重新排序数列，所有元素比基准值小的摆放在基准前面，所有元素比基准值大的摆在基准的后面（相同的数可以到任一边）。这个称为分区（partition）操作。在这个分区退出之后，该基准就处于数列的中间位置（它应该在的位置）；
- 递归地（recursive）把小于基准值元素的子数列，和大于基准值元素的子数列排序。

这里需要注意，分区操作在具体实现时，可以设置在序列首尾设置**双指针**，然后分别向中间移动；左指针找到最近的一个大于基准的数，右指针找到最近一个小于基准的数，然后交换这两个数。

代码如下：

```
public class QuickSort {
    public static void qSort( int[] nums, int start, int end ){
        if ( start >= end )
```

```
            return;
        int mid = partition(nums, start, end);
        qSort( nums, start, mid - 1 );
        qSort( nums, mid + 1, end );
    }
    // 定义一个分区方法
    private static int partition( int[] nums, int start, int end ){
        int pivot = nums[start];
        int left = start;
        int right = end;
        while ( left < right ){
            while ( left < right && nums[right] >= pivot )
                right --;
            nums[left] = nums[right];
            while ( left < right && nums[left] <= pivot )
                left ++;
            nums[right] = nums[left];
        }
        nums[left] = pivot;
        return left;
    }
}
```

快速排序的时间复杂度可以做到 O(nlogn)，在很多框架和数据结构设计中都有广泛的应用。

9.1.7 堆排序（Heap Sort）

堆排序是指利用堆这种数据结构所设计的一种排序算法。

堆（Heap）是一个近似完全二叉树的结构，并同时满足堆的性质：即子结点的键值或

索引总是小于（或者大于）它的父节点。

一般情况，将堆顶元素为最大值的叫做“大顶堆”（Max Heap），堆顶为最小值的叫做“小顶堆”。

算法简单来说，就是构建一个大顶堆，取堆顶元素作为当前最大值，然后删掉堆顶元素、将最后一个元素换到堆顶位置，进而不断调整大顶堆、继续寻找下一个最大值。

这个过程有一些类似于选择排序（每次都选取当前最大的元素），而由于用到了二叉树结构进行大顶堆的调整，时间复杂度可以降为 O(nlogn)。

9.1.8 计数排序（Counting Sort）

计数排序不是基于比较的排序算法，其核心在于将输入的数据值转化为键存储在额外开辟的数组空间中。作为一种线性时间复杂度的排序，计数排序要求输入的数据必须是有确定范围的整数。

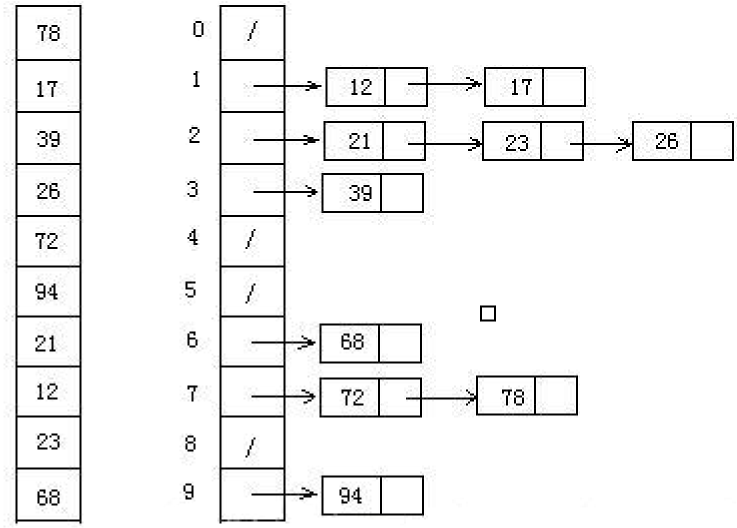
简单来说，就是要找到待排序数组中的最大和最小值，得到所有元素可能的取值范围；然后统计每个值出现的次数。统计完成后，只要按照取值顺序、依次反向填充目标数组就可以了。

计数排序时间复杂度是 O(n+k)，空间复杂度也是 O(n+k)，其排序速度快于任何比较排序算法。当 k 不是很大并且序列比较集中时，计数排序是一个很有效的排序算法。

9.1.9 桶排序（Bucket Sort）

桶排序是计数排序的升级版。它利用了函数的映射关系，高效与否的关键就在于映射函数的确定。

桶排序 (Bucket sort)的工作原理：假设输入数据服从均匀分布，将数据分到有限数量的桶里，每个桶再分别排序。



桶排序最好情况下使用线性时间 O(n)。

桶排序的时间复杂度，取决与对各个桶之间数据进行排序的时间复杂度，因为其它部分的时间复杂度都为 O(n)。很显然，桶划分的越小，各个桶之间的数据越少，排序所用的时间也会越少。但相应的空间消耗就会增大。

9.1.10 基数排序（Radix Sort）

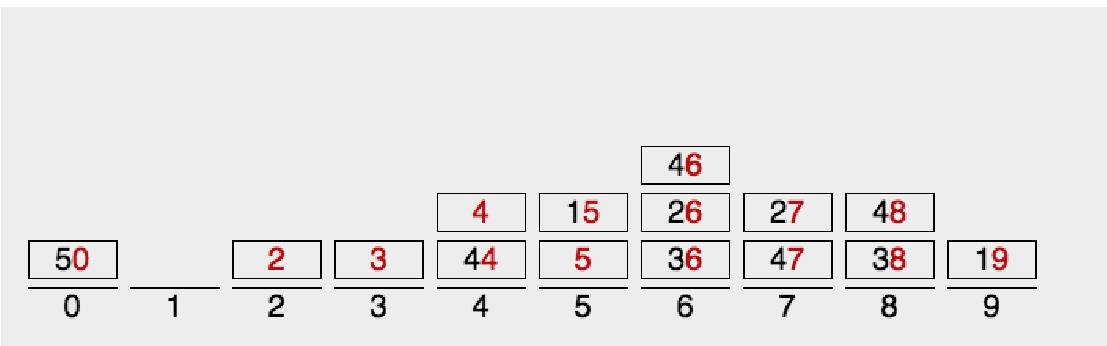
基数排序可以说是桶排序的扩展。

算法原理是按照低位先排序，然后收集；再按照高位排序，然后再收集；依次类推，直到最高位。

最常见的做法，就是取 10 个桶，数值最高有几位，就按照数位排几次。例如：

3	44	38	5	47	15	36	26	27	2	46	4	19	50	48

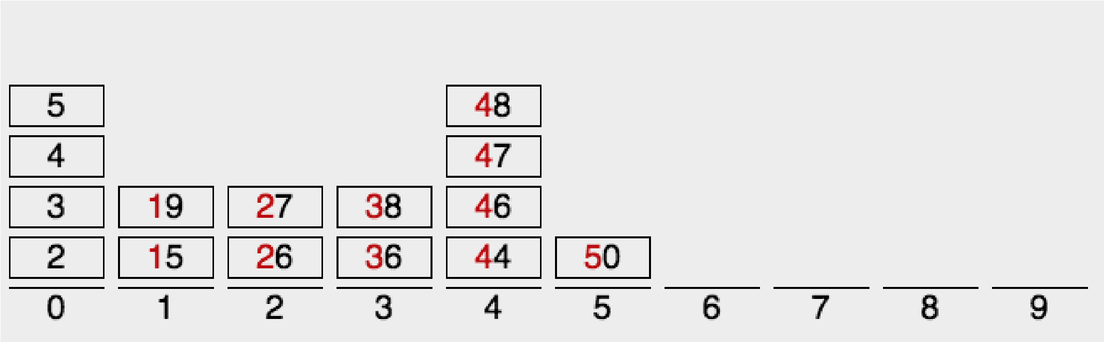
第一次排序：按照个位的值，将每个数保存到对应的桶中：



将桶中的数据依次读出，填充到目标数组中，这时可以保证后面的数据，个位一定比前面的数据大。

50	2	3	44	4	5	15	36	26	46	47	27	38	48	19

第二次排序：按照十位的值，将每个数保存到对应的桶中：



因为每个桶中的数据，都是按照个位从小到大排序的，所以再次顺次读出每个桶中的数据，就得到了完全排序的数组：

2	3	4	5	15	19	26	27	36	38	44	46	47	48	50

基数排序的空间复杂度为 O(n+k)，其中 k 为桶的数量。一般来说 n>>k，因此额外空间需要大概 n 个左右。

9.2 数组中的第 K 个最大元素（#215）

9.2.1 题目说明

在未排序的数组中找到第 k 个最大的元素。请注意，你需要找的是数组排序后的第 k 个最大的元素，而不是第 k 个不同的元素。

示例 1:

```
输入: [3,2,1,5,6,4] 和 k = 2
输出: 5
```

示例 2:

```
输入: [3,2,3,1,2,4,5,5,6] 和 k = 4
输出: 4
```

说明:

- 你可以假设 k 总是有效的，且 1 ≤ k ≤ 数组的长度。

9.2.2 分析

要寻找数组中第 K 大的元素，首先能想到的，当然就是直接排序。只要数组是有序的，那么接下来取出倒数第 K 个元素就可以了。

```java
public class KthLargestElement {
    // 直接调语言内置的排序方法
    public int findKthLargest(int[] nums, int k) {
        Arrays.sort(nums);
        return nums[nums.length - k];
    }
}
```

我们知道，java 的 Arrays.sort()方法底层就是快速排序，所以时间复杂度为 O(nlogn)。

如果实际遇到这个问题，直接调类库方法去排序，显然是不能让面试官满意的。我们应该手动写出排序的算法。

选择、冒泡和插入排序时间复杂度是 O(n^2)，性能较差；二计数排序、桶排序和基数排序尽管时间复杂度低，但需要占用大量的额外空间，而且只有在数据取值范围比较集中、桶数较少时效率比较高。所以实际应用中，排序的实现算法一般采用快速排序，或者归并和堆排序。

对于这道题目而言，其实还可以进一步优化：因为我们只关心第 K 大的元素，其它位置的元素其实可以不排。

基于这样的想法，显然归并这样的算法就无从优化了；但快排和堆排序可以。

9.2.3 方法一：基于快速排序的选择

我们可以改进快速排序算法来解决这个问题：在分区（partition）的过程当中，我们会对子数组进行划分，如果划分得到的位置 q 正好就是我们需要的下标，就直接返回 a[q]；否则，如果 q 比目标下标小，就递归右子区间，否则递归左子区间。这样就可以把原来递归两个区间变成只递归一个区间，提高了时间效率。这就是“快速选择”算法。

另外，我们知道快速排序的性能和“划分”出的子数组的长度密切相关。我们可以引入随机化来加速这个过程，它的时间代价的期望是 O(n)。

代码如下：

```
public int findKthLargest(int[] nums, int k) {
    return quickSelect( nums, 0, nums.length - 1, nums.length - k );
}
// 为了方便递归，定义一个快速选择方法
public int quickSelect( int[] nums, int start, int end, int index ){
    int q = randomPatition( nums, start, end );
    if (q == index){
        return nums[q];
    } else {
        return q > index ? quickSelect(nums, start, q - 1, index) :
quickSelect(nums, q + 1, end, index);
    }
}
// 定义一个随机分区方法
public int randomPatition( int[] nums, int start, int end ){
    Random random = new Random();
    int randIndex = start + random.nextInt(end - start + 1);
    swap(nums, start, randIndex);
    return partition(nums, start, end);
```

```
}
// 定义一个分区方法
public int partition( int[] nums, int start, int end ){
    int pivot = nums[start];
    int left = start;
    int right = end;
    while ( left < right ){
        while ( left < right && nums[right] >= pivot )
            right --;
        nums[left] = nums[right];
        while ( left < right && nums[left] <= pivot )
            left ++;
        nums[right] = nums[left];
    }
    nums[left] = pivot;
    return left;
}
// 定义一个交换元素的方法
public void swap( int[] nums, int i, int j ){
    int temp = nums[i];
    nums[i] = nums[j];
    nums[j] = temp;
}
```

复杂度分析

时间复杂度：O(n)，证明过程可以参考《算法导论》9.2：期望为线性的选择算法。

空间复杂度：O(logn)，递归使用栈空间的空间代价的期望为 O(logn)。

9.2.4 方法二：基于堆排序的选择

我们也可以使用堆排序来解决这个问题。

基本思路是：构建一个大顶堆，做 k-1 次删除操作后堆顶元素就是我们要找的答案。

在很多语言中，都有优先队列或者堆的的容器可以直接使用，但是在面试中，面试官更倾向于让更面试者自己实现一个堆。所以这里我们要手动做一个类似堆排序的实现。

代码如下：

```
public int findKthLargest(int[] nums, int k) {
    int n = nums.length;
    int heapSize = n;
    // 构建大顶堆
    buildMaxHeap( nums, heapSize );
    // 删除 k-1 次堆顶元素
    for ( int i = n - 1; i > n - k; i-- ){
        swap( nums, 0, i );
        heapSize --;
        maxHeapify( nums, 0, heapSize );
    }
    return nums[0];
}

// 构建大顶堆的方法
public void buildMaxHeap( int[] nums, int heapSize ){
    for ( int i = heapSize / 2 - 1; i >= 0; i-- ){
        maxHeapify(nums, i, heapSize);
    }
}
public void maxHeapify( int[] nums, int top, int heapSize ){
```

```
    int left = top * 2 + 1;
    int right = top * 2 + 2;
    int largest = top;
    if ( right < heapSize && nums[right] > nums[largest] ){
        largest = right;
    }
    if ( left < heapSize && nums[left] > nums[largest] ){
        largest = left;
    }
    if ( largest != top ){
        swap( nums, top, largest );
        maxHeapify(nums, largest, heapSize);
    }
}
```

复杂度分析

时间复杂度：O(nlogn)，建堆的时间代价是 O(n)，k-1 次删除的总代价是 O(klogn)，因为 k<n，故渐进时间复杂为 O(n+klogn)=O(nlogn)。

空间复杂度：O(logn)，即递归使用栈空间的空间代价。

9.3 颜色分类（#75）

9.3.1 题目说明

给定一个包含红色、白色和蓝色，一共 n 个元素的数组，原地对它们进行排序，使得相同颜色的元素相邻，并按照红色、白色、蓝色顺序排列。

此题中，我们使用整数 0、 1 和 2 分别表示红色、白色和蓝色。

进阶：

- 你可以不使用代码库中的排序函数来解决这道题吗？

- 你能想出一个仅使用常数空间的一趟扫描算法吗？

示例 1：

```
输入：nums = [2,0,2,1,1,0]
输出：[0,0,1,1,2,2]
```

示例 2：

```
输入：nums = [2,0,1]
输出：[0,1,2]
```

示例 3：

```
输入：nums = [0]
输出：[0]
```

示例 4：

```
输入：nums = [1]
输出：[1]
```

提示：

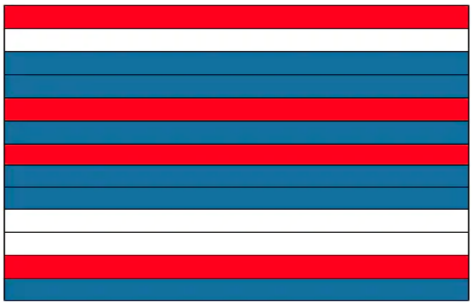
- n == nums.length
- 1 <= n <= 300
- nums[i] 为 0、1 或 2

9.3.2 分析

本题是经典的“荷兰国旗问题”，由计算机科学家 Edsger W. Dijkstra 首先提出。

荷兰国旗是由红白蓝 3 种颜色的条纹拼接而成，如下图所示：

假设这样的条纹有多条，且各种颜色的数量不一，并且随机组成了一个新的图形，新的图形可能如下图所示，但是绝非只有这一种情况：



需求是：把这些条纹按照颜色排好，红色的在上半部分，白色的在中间部分，蓝色的在下半部分，我们把这类问题称作荷兰国旗问题。

本题其实就是荷兰国旗问题的数学描述，它在本质上，其实就是就是一个有重复元素的排序问题。所以可以用排序算法来解决。

当然，最简单的方式，就是直接调 Java 已经内置的排序方法：

```
public void sortColors(int[] nums) {
    Arrays.sort(nums);
}
```

时间复杂度为 O(nlogn)。但显然这不是我们想要的，本题用到的排序算法应该自己实现，而且要根据本题的具体情况进行优化。

9.3.3 方法一：基于选择排序

如果用选择排序的思路，我们可以通过遍历数组，找到当前最小（或最大的数）。

对于本题，因为只有 0，1，2 三个值，我们不需要对每个位置的“选择”都遍历一遍数组，而是最多遍历三次就够了：第一次遍历，把扫描到的 0 全部放到数组头部；第二次遍历，把所有 1 跟在后面；最后一次，把所有 2 跟在最后。

事实上，最后对于 2 的扫描已经没有必要了：因为除了 0 和 1，剩下的位置一定都是 2。所以我们可以用两次扫描，实现这个算法。

代码如下：

```java
public void sortColors(int[] nums) {
    int curr = 0;
    // 第一次遍历，将扫描到的 0 交换到数组头部
    for ( int i = 0; i < nums.length; i++){
        if ( nums[i] == 0 ){
            swap( nums, curr++, i );
        }
    }
    // 第二次遍历，将扫描到的 1 跟在后面
    for ( int i = 0; i < nums.length; i++){
        if ( nums[i] == 1 ){
            swap( nums, curr++, i );
        }
    }
}
public void swap( int[] nums, int i, int j ){
    int temp = nums[i];
    nums[i] = nums[j];
    nums[j] = temp;
}
```

复杂度分析

时间复杂度：O(n)，n 为数组 nums 的长度。需要遍历两次数组。

空间复杂度：O(1)，只用到了常数个辅助变量。

9.3.4 方法二：基于计数排序

根据题目中的提示，要排序的数组中，其实只有 0，1，2 三个值。

所以另一种思路是，我们可以直接统计出数组中 0,1,2 的个数，再根据它们的数量，重

写整个数组。这其实就是计数排序的思路。

代码如下：

```
public class SortColors {
    public void sortColors(int[] nums) {
        int count0 = 0, count1 = 0, count2 = 0;
        // 遍历数组，统计 0，1，2 的个数
        for ( int num: nums ){
            if ( num == 0 )
                count0 ++;
            else if ( num == 1 )
                count1 ++;
            else
                count2 ++;
        }
        // 将 0，1，2 按个数依次填入数组
        for ( int i = 0; i < nums.length; i++ ){
            if ( i < count0 )
                nums[i] = 0;
            else if ( i < count0 + count1 )
                nums[i] = 1;
            else
                nums[i] = 2;
        }
    }
}
```

复杂度分析

时间复杂度：O(n)，n 为数组 nums 的长度。需要遍历两次数组。

空间复杂度：O(1)，只用到了常数个辅助变量。

9.3.5 方法三：基于快速排序

前面的算法，尽管时间复杂度为 O(n)，但都进行了两次遍历。能不能做一些优化，只进行一次遍历就解决问题呢？

一个思路是，使用双指针。所有的 0 移到数组头，所有 2 移到数组尾，1 保持不变就可以了。这其实就是快速排序的思路。

代码如下：

```
public void sortColors(int[] nums) {
    int left = 0, right = nums.length - 1;
    int i = left;
    while ( left < right && i <= right ){
        while ( i <= right && nums[i] == 2 )
            swap( nums, i, right-- );
        if ( nums[i] == 0 )
            swap( nums, i, left++ );
        i++;
    }
}
```

复杂度分析

时间复杂度：O(n)，n 为数组 nums 的长度。双指针法只虚遍历一次数组。

空间复杂度：O(1)，只用到了常数个辅助变量。

9.4 合并区间（#56）

9.4.1 题目说明

给出一个区间的集合，请合并所有重叠的区间。

示例 1:

```
输入: intervals = [[1,3],[2,6],[8,10],[15,18]]
输出: [[1,6],[8,10],[15,18]]
解释: 区间 [1,3] 和 [2,6] 重叠, 将它们合并为 [1,6].
```

示例 2:

```
输入: intervals = [[1,4],[4,5]]
输出: [[1,5]]
解释: 区间 [1,4] 和 [4,5] 可被视为重叠区间。
```

提示：

- intervals[i][0] <= intervals[i][1]

9.4.2 分析

要判断两个区间[a1, b1], [a2, b2]是否可以合并，其实就是判断是否有 a1 <= a2 <= b1，或者 a2 <= a1 <= b2。也就是说，如果某个子区间的左边界在另一子区间内，那么它们可以合并。

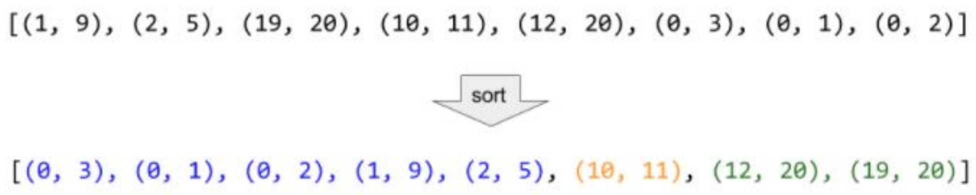
9.4.3 解决方法：排序

一个简单的想法是，我们可以遍历每一个子区间，然后判断它跟其它区间是否可以合并。如果某两个区间可以合并，那么就把它们合并之后，再跟其它区间去做判断。

很明显，这样的暴力算法，时间复杂度不会低于 O(n^2)。有没有更好的方式呢？

这里我们发现，判断区间是否可以合并的关键，在于它们左边界的大小关系。所以我们可以先把所有区间，按照**左边界进行排序**。

那么在排完序的列表中，可以合并的区间一定是连续的。如下图所示，标记为蓝色、黄色和绿色的区间分别可以合并成一个大区间，它们在排完序的列表中是连续的：



具体代码如下：

```
public class MergeIntervals {
    public int[][] merge(int[][] intervals) {
        List<int[]> result = new ArrayList<int[]>();
        // 先对原数组按左边界排序
        Arrays.sort(intervals, new Comparator<int[]>() {
            @Override
            public int compare(int[] o1, int[] o2) {
                return o1[0] - o2[0];
            }
        });
        // 遍历排序后的数组，逐个判断合并
        for ( int[] interval: intervals ){
            int left = interval[0], right = interval[1];
            int length = result.size();
            if ( length == 0 || left > result.get(length - 1)[1] ){
                result.add(interval);
            } else {
                int mergedLeft = result.get(length - 1)[0];
                int mergedRight = Math.max( result.get(length - 1)[1],
```

```
right );
                result.set( length - 1, new int[]{mergedLeft, mergedRight} );
            }
        }
        return result.toArray(new int[result.size()][]);
    }
}
```

复杂度分析

时间复杂度：O(nlogn)，其中 n 为区间的数量。除去排序的开销，我们只需要一次线性扫描，所以主要的时间开销是排序的 O(nlogn)。

空间复杂度：O(logn)，其中 n 为区间的数量。O(logn) 即为快速排序所需要的空间复杂度（递归栈深度）。

第十章 二叉树及递归问题讲解

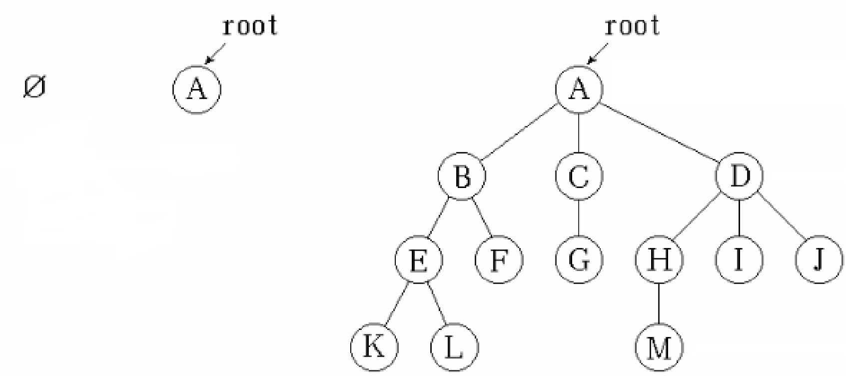
10.1 数和二叉树数据结构复习

10.1.1 树（Tree）

树是一种**非线性**的数据结构，是由 n（n >=0）个结点组成的有限集合。

如果 n==0，树为空树。如果 n>0，树有一个特定的结点，叫做根结点（root）。根结点只有直接后继，没有直接前驱。

除根结点以外的其他结点划分为 m（m>=0）个互不相交的有限集合，T0，T1，T2，...，Tm-1，每个集合都是一棵树，称为根结点的子树（sub tree）。



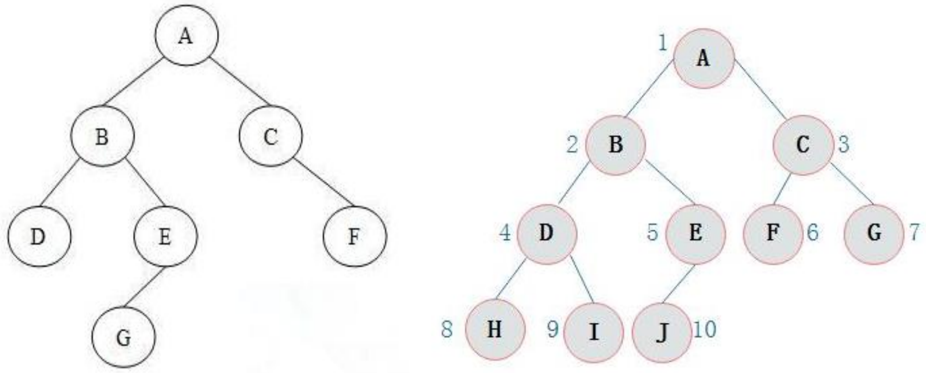
下面是一些其它基本概念：

- 节点的度：节点拥有的子树个数
- 叶子节点（leaf）：度为 0 的节点，也就是没有子树的节点
- 树的高度：树中节点的最大层数，也叫做树的深度

10.1.2 二叉树（Binary Tree）

对于树这种数据结构，使用最频繁的是**二叉树**。

每个节点最多只有 2 个子节点的树，叫做二叉树。二叉树中，每个节点的子节点作为根的两个子树，一般叫做节点的左子树和右子树。



（1）二叉树的性质

二叉树有以下性质：

- 若二叉树的层次从 0 开始，则在二叉树的第 i 层至多有 2^i 个结点(i>=0)
- 高度为 k 的二叉树最多有 2^(k+1) - 1 个结点(k>=-1)(空树的高度为-1)
- 对任何一棵二叉树，如果其叶子结点(度为 0)数为 m，度为 2 的结点数为 n，则 m = n + 1

（2）满二叉树和完全二叉树

- 满二叉树：除了叶子节点外，每个节点都有两个子节点，每一层都被完全填充。
- 完全二叉树：除了最后一层外，每一层都被完全填充，并且最后一层所有节点保持向左对齐。

10.1.3 递归（Recursion）

对于树结构的遍历和处理，最为常用的代码结构就是**递归**（Recursion）。

递归是一种重要的编程技术，该方法用来让一个函数（方法）从其内部调用其自身。一个含直接或间接调用本函数语句的函数，被称之为递归函数。

递归的实现有两个必要条件：

- 必须定义一个“基准条件”，也就是递归终止的条件。在这种情况下，可以直接返回结果，无需继续递归
- 在方法中通过调用自身，向着基准情况前进

一个简单示例就是计算阶乘：0 的阶乘被特别地定义为 1；n 的阶乘可以通过计算 n-1

的阶乘再乘以 n 来求得的。

代码如下：

```
// 递归示例：计算阶乘
public static int factorial(int n){
    if ( n == 0 ) return 1;
    return factorial(n - 1) * n;
}

// 尾递归计算阶乘，需要多一个参数保存"计算状态"
public static int fact(int acc, int n){
    if ( n == 0 ) return acc;
    return fact( acc * n, n - 1 );
}
```

上面的第二种实现，把递归调用置于函数的末尾，即正好在 return 语句之前，这种形式的递归被称为**尾递归** (tail recursion)，其形式相当于循环。一些语言的编译器对于尾递归可以进行优化，节约递归调用的栈资源。

10.1.4 二叉树的遍历

- 中序遍历：即左-根-右遍历，对于给定的二叉树根，寻找其左子树；对于其左子树的根，再去寻找其左子树；递归遍历，直到寻找最左边的节点 i，其必然为叶子，然后遍历 i 的父节点，再遍历 i 的兄弟节点。随着递归的逐渐出栈，最终完成遍历
- 先序遍历：即根-左-右遍历
- 后序遍历：即左-右-根遍历
- 层序遍历：按照从上到下、从左到右的顺序，逐层遍历所有节点。

用递归可以很容易地实现二叉树的先序、中序、后序遍历：

```java
// 遍历二叉树1：先序遍历
public static void printTreePreOrder( TreeNode root ){
    if (root == null) return;
    System.out.print(root.val + "\t");
    printTreePreOrder( root.left );
    printTreePreOrder( root.right );
}
// 遍历二叉树2：中序遍历
public static void printTreeInOrder( TreeNode root ){
    if (root == null) return;
    printTreeInOrder( root.left );
    System.out.print(root.val + "\t");
    printTreeInOrder( root.right );
}
// 遍历二叉树3：后序遍历
public static void printTreePostOrder( TreeNode root ){
    if (root == null) return;
    printTreePostOrder( root.left );
    printTreePostOrder( root.right );
    System.out.print(root.val + "\t");
}
```

层序遍历，则需要借助一个队列：要访问的节点全部放到队列里。当访问一个节点时，就让它的子节点入队，依次访问。

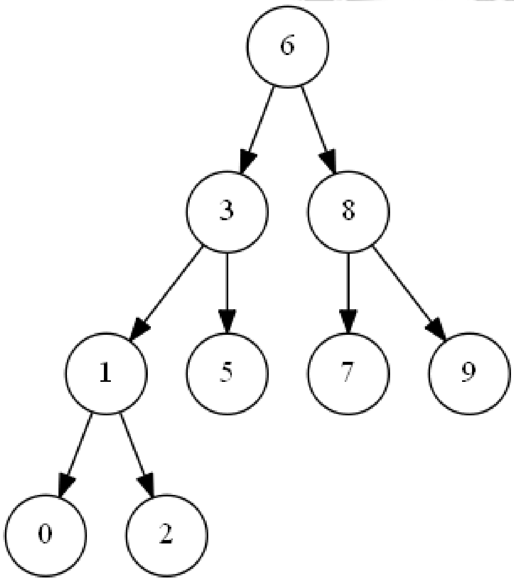
```java
public static void printTreeLevelOrder( TreeNode root ){
    Queue<TreeNode> queue = new LinkedList<>();
    queue.offer(root);
    while ( !queue.isEmpty() ){
        TreeNode curNode = queue.poll();
```

```
        System.out.print(curNode.val + "\t");
        if ( curNode.left != null )
            queue.offer(curNode.left);
        if ( curNode.right != null )
            queue.offer(curNode.right);
    }
}
```

10.1.5 二叉搜索树（Binary Search Tree）

二叉搜索树也称为有序二叉查找树，满足二叉查找树的一般性质，是指一棵空树具有如下性质：

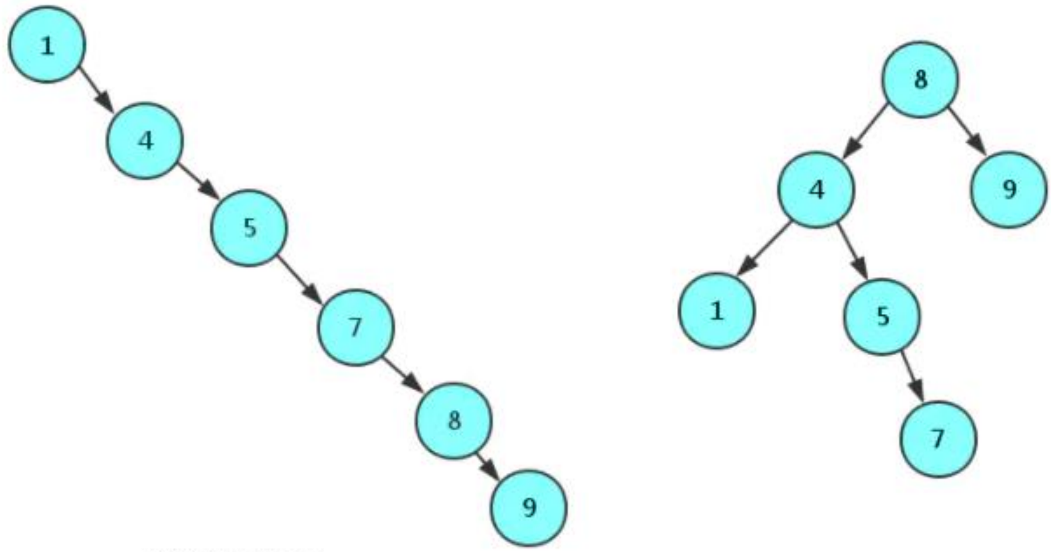
- 任意节点左子树如果不为空，则左子树中节点的值均小于根节点的值
- 任意节点右子树如果不为空，则右子树中节点的值均大于根节点的值
- 任意节点的左右子树，也分别是二叉搜索树
- 没有键值相等的节点



基于二叉搜索树的这种特点，在查找某个节点的时候，可以采取类似于二分查找的思想，快速找到某个节点。n 个节点的二叉查找树，正常的情况下，查找的时间复杂度为 O(logN)。

二叉搜索树的局限性

一个二叉搜索树是由 n 个节点随机构成，所以，对于某些情况，二叉查找树会退化成一个有 n 个节点的线性链表。如下图:



a 退化为线性表　　　　b 普通的二叉查找树

10.1.6 平衡二叉搜索树（AVL 树）

通过二叉搜索树的分析我们发现，二叉搜索树的节点查询、构造和删除性能，与树的高度相关，如果二叉搜索树能够更“平衡”一些，避免了树结构向线性结构的倾斜，则能够显著降低时间复杂度。

平衡二叉搜索树：简称平衡二叉树。由前苏联的数学家 Adelse-Velskil 和 Landis 在 1962 年提出的高度平衡的二叉树，根据科学家的英文名也称为 AVL 树。

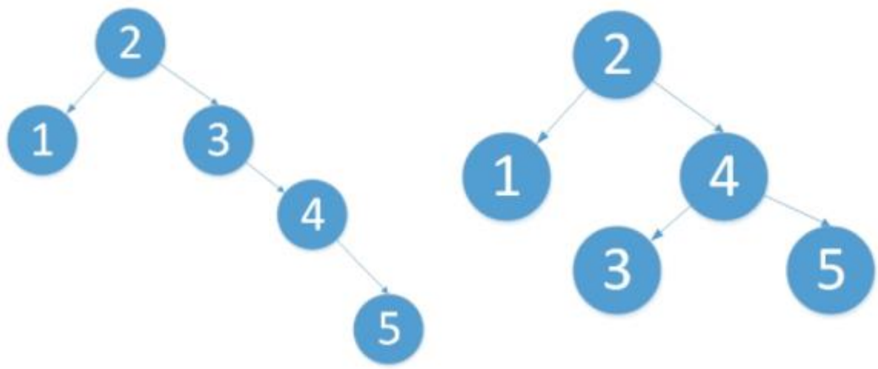
它具有如下几个性质：

- 可以是空树
- 假如不是空树，任何一个结点的左子树与右子树都是平衡二叉树，并且高度之差的绝对值不超过 1

平衡的意思，就是向天平一样保持左右水平，即两边的分量大约相同。如定义，假如一棵树的左右子树的高度之差超过 1，如左子树的树高为 2，右子树的树高为 0，子树树高差的绝对值为 2 就打破了这个平衡。

比如，依次插入 1，2，3 三个结点后，根结点的右子树树高减去左子树树高为 2，树就失去了平衡。我们希望它能够变成更加平衡的样子。

AVL 树是带有平衡条件的二叉搜索树，它是严格的平衡二叉树，平衡条件必须满足(所有节点的左右子树高度差不超过 1)。不管我们是执行插入还是删除操作，只要不满足上面的条件，就要通过旋转来保持平衡，而旋转是非常耗时的。旋转的目的是为了降低树的高度，使其平衡。



使用场景

AVL 树适合用于插入删除次数比较少，但查找多的情况。也在 Windows 进程地址空间管理中得到了使用。

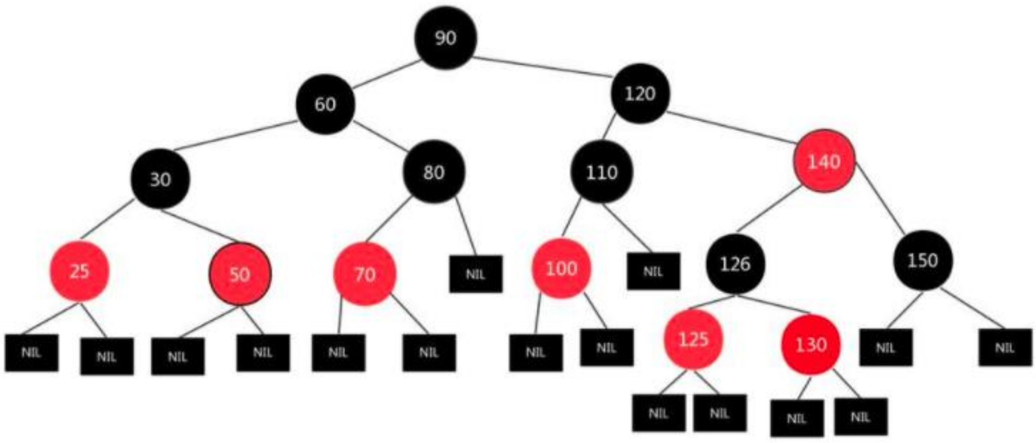
10.1.7 红黑树（Red-Black Tree）

红黑树是一种特殊的二叉查找树。红黑树的每个节点上都有存储位表示节点的颜色，可以是红(Red)或黑(Black)。

性质：

- 节点是红色或黑色
- 根节点是黑色

- 每个叶子节点都是黑色的空节点（NIL 节点）。
- 每个红色节点的两个子节点都是黑色（从每个叶子到根的所有路径上不能有两个连续的红色节点）
- 从任一节点到其每个叶子的所有路径都包含相同数目的黑色节点



在插入一个新节点时，默认将它涂为红色（这样可以不违背最后一条规则），然后进行旋转着色等操作，让新的树符合所有规则。

红黑树也是一种自平衡二叉查找树，可以认为是对 AVL 树的折中优化。

使用场景

红黑树多用于搜索,插入,删除操作多的情况下。红黑树应用比较广泛：

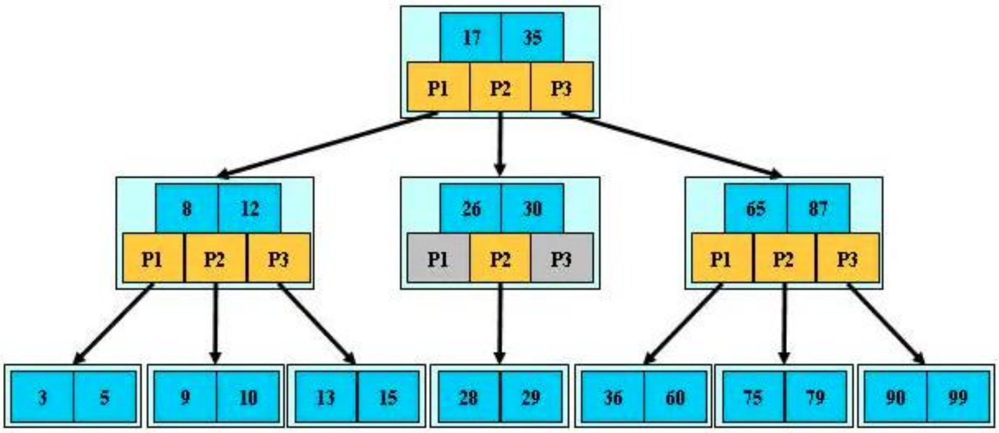
- 广泛用在各种语言的内置数据结构中。比如 C++的 STL 中，map 和 set 都是用红黑树实现的。Java 中的 TreeSet，TreeMap 也都是用红黑树实现的。
- 著名的 linux 进程调度 Completely Fair Scheduler，用红黑树管理进程控制块。
- epoll 在内核中的实现，用红黑树管理事件块
- nginx 中，用红黑树管理 timer 等

10.1.8 B 树（B-Tree）

B 树(B-Tree)是一种自平衡的树，它是一种多路搜索树（并不是二叉的），能够保证数据有序。同时，B 树还保证了在查找、插入、删除等操作时性能都能保持在 O(logn)，为大块数据的读写操作做了优化，同时它也可以用来描述外部存储。

特点：

- 定义任意非叶子结点最多只有 M 个儿子；且 M>2
- 根结点的儿子数为[2, M]
- 除根结点以外的非叶子结点的儿子数为[M/2, M]
- 每个结点存放至少 M/2-1（取上整）和至多 M-1 个关键字；（至少 2 个 key）
- 非叶子结点的关键字个数 = 指向儿子的指针个数 – 1
- 非叶子结点的关键字：K[1], K[2], …, K[M-1]；且 K[i] < K[i+1]
- 非叶子结点的指针：P[1], P[2], …, P[M]，其中 P[1]指向关键字小于 K[1]的子树，P[M]指向关键字大于 K[M-1]的子树，其它 P[i]指向关键字属于(K[i-1], K[i])的子树
- 所有叶子结点位于同一层



M = 3 的 B 树

10.1.9 B+树

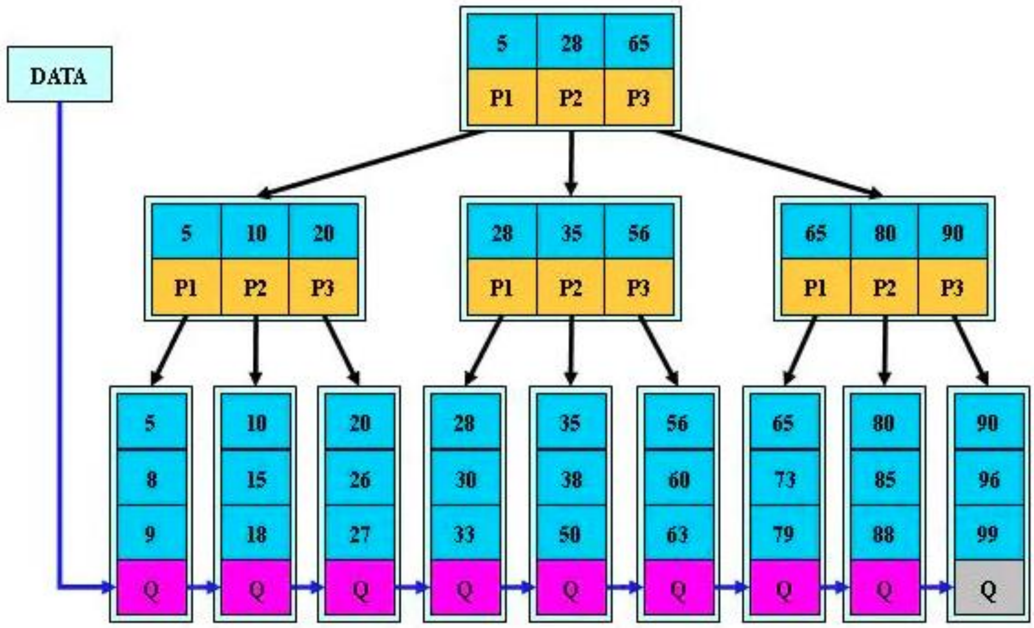
B+树是 B-树的变体，也是一种多路搜索树。

B+的搜索与 B-树也基本相同，区别是 B+树只有达到叶子结点才命中（B-树可以在非叶子结点命中），其性能也等价于在关键字全集做一次二分查找。

B+的特性：

- 所有关键字都出现在叶子结点的链表中（稠密索引），且链表中的关键字恰好是有序的
- 不可能在非叶子结点命中
- 非叶子结点相当于是叶子结点的索引（稀疏索引），叶子结点相当于是存储（关键字）数据的数据层

- 更适合文件索引系统



B+ 树的优点：

- 层级更低，IO 次数更少
- 每次都需要查询到叶子节点，查询性能稳定
- 叶子节点形成有序链表，范围查询方便。这使得 B+树方便进行“扫库”，也是很多文件系统和数据库底层选用 B+树的主要原因。

10.2 翻转二叉树（#226）

10.2.1 题目说明

翻转一棵二叉树。

示例：

输入：

　　4

```
    /   \
   2     7
  / \   / \
 1   3 6   9
输出：
      4
    /   \
   7     2
  / \   / \
 9   6 3   1
```

10.2.2 分析

这是一道很经典的二叉树问题。

显然，我们可以遍历这棵树，分别翻转左右子树，一层层递归调用，就可以翻转整个二叉树了。

10.2.3 方法一：先序遍历

容易想到，我们可以先考察根节点，把左右子树调换，然后再分别遍历左右子树、依次翻转每一部分就可以了。

这对应的遍历方式，就是先序遍历。

代码如下：

```java
public TreeNode invertTree(TreeNode root) {
    if ( root == null ) return null;
    TreeNode temp = root.left;
    root.left = root.right;
    root.right = temp;
    invertTree( root.left );
```

```
    invertTree( root.right );
    return root;
}
```

复杂度分析

时间复杂度：O(N)，其中 N 为二叉树节点的数目。我们会遍历二叉树中的每一个节点，对每个节点而言，我们在常数时间内交换其两棵子树。

空间复杂度：O(logN)。使用的空间由递归栈的深度决定，它等于当前节点在二叉树中的高度。在平均情况下，二叉树的高度与节点个数为对数关系，即 O(logN)。而在最坏情况下，树形成链状，空间复杂度为 O(N)。

10.2.4 方法二：后序遍历

类似地，我们也可以用后序遍历的思路：先递归地处理左右子树，然后再将左右子树调换就可以了。

代码如下：

```
public TreeNode invertTree(TreeNode root) {
    if ( root == null ) return null;
    TreeNode left = invertTree(root.left);
    TreeNode right = invertTree(root.right);
    root.left = right;
    root.right = left;
    return root;
}
```

复杂度分析略，与方法一完全相同。

10.3 平衡二叉树（#110）

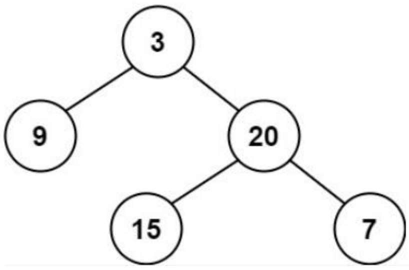
10.3.1 题目说明

给定一个二叉树，判断它是否是高度平衡的二叉树。

本题中，一棵高度平衡二叉树定义为：

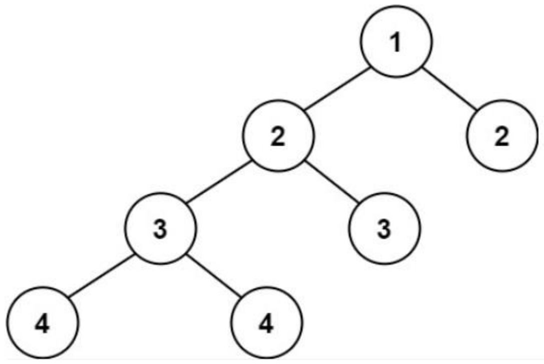
- 一个二叉树每个节点 的左右两个子树的高度差的绝对值不超过 1 。

示例 1：




```
输入：root = [3,9,20,null,null,15,7]
输出：true
```

示例 2：




```
输入：root = [1,2,2,3,3,null,null,4,4]
输出：false
```

示例 3：

```
输入：root = []
输出：true
```

10.3.2 分析

根据定义，当且仅当一棵二叉树的左右子树也都是平衡二叉树时，这棵二叉树是平衡二叉树。

因此可以使用递归的方式，判断二叉树是不是平衡二叉树，递归的顺序可以是自顶向下（类似先序遍历）或者自底向上（类似后序遍历）。

10.3.3 方法一：自顶向下

容易想到的一个方法是，从根节点开始，自顶向下递归地判断左右子树是否平衡。

具体过程是，先分别计算当前节点左右子树的高度，如果高度差不超过 1，那么再递归地分别判断左右子树。这其实就是一个先序遍历的思路。

代码如下：

```
public class BalancedBinaryTree {
    public boolean isBalanced(TreeNode root) {
        if ( root == null ) return true;
        return Math.abs( height(root.left) - height(root.right) ) <= 1
                && isBalanced(root.left)
                && isBalanced(root.right);
    }
    // 定义一个height 方法，用于计算树的高度
    public int height(TreeNode root){
        if ( root == null ) return 0;
        return Math.max( height(root.left), height(root.right) ) + 1;
    }
}
```

复杂度分析

时间复杂度：O(n logn)，其中 n 是二叉树中的节点个数。

最坏情况下，二叉树是满二叉树，需要遍历二叉树中的所有节点，时间复杂度是 O(n)。

对于节点 p，如果它的高度是 d，则计算高度的方法 height(p) 最多会被调用 d 次（即遍历到它的每一个祖先节点时）。

对于平均的情况，一棵树的高度 h 满足 O(h)=O(logn)，因为 d≤h，所以总时间复杂度为 O(nlogn)。对于最坏的情况，二叉树形成链式结构，高度为 O(n)，此时总时间复杂度为 O(n^2)。

空间复杂度：O(logn)，其中 n 是二叉树中的节点个数。空间复杂度主要取决于递归调用的层数，递归调用的层数平均为 O(logn)，最坏情况为 O(n)。

10.3.4 方法二：自底向上

上面的算法通过分析可以看到，每个节点高度的计算，会在它的祖先节点计算时重复调用，这显然是不必要的。

一种优化思路是，可以反过来，自底向上地遍历节点进行判断。计算每个节点的高度时，需要递归地处理左右子树；所以可以先判断左右子树是否平衡，计算出左右子树的高度，再判断当前节点是否平衡。这类似于后序遍历的思路。

这样，计算高度的方法 height，对于每个节点就只调用一次了。

代码如下：

```java
public boolean isBalanced(TreeNode root) {
    if ( root == null ) return true;
    int leftHeight = balancedHeight(root.left);
    int rightHeight = balancedHeight(root.right);
    return leftHeight != -1 && rightHeight != -1 && Math.abs( leftHeight -
rightHeight ) <= 1;
}
// 定义一个height方法
public int balancedHeight(TreeNode root){
    if ( root == null ) return 0;
```

```java
    int leftHeight = balancedHeight(root.left);
    int rightHeight = balancedHeight(root.right);
    // 如果子树不平衡，直接返回-1
    if ( leftHeight == -1 || rightHeight == -1 || Math.abs( leftHeight - rightHeight ) > 1)
        return -1;
    // 如果平衡，高度就是左右子树高度最大值，再加1
    return Math.max( leftHeight, rightHeight ) + 1;
}
```

复杂度分析

时间复杂度： O(n)，其中 n 是二叉树中的节点个数。使用自底向上的递归，每个节点的计算高度和判断是否平衡，都只需要处理一次。最坏情况下需要遍历二叉树中的所有节点，因此时间复杂度是 O(n)。

空间复杂度： O(logn)，其中 n 是二叉树中的节点个数。空间复杂度主要取决于递归调用的层数，递归调用的层数平均为 O(logn)，最坏情况为 O(n)。

10.4 验证二叉搜索树（#98）

10.4.1 题目说明

给定一个二叉树，判断其是否是一个有效的二叉搜索树。

假设一个二叉搜索树具有如下特征：

- 节点的左子树只包含**小于**当前节点的数。
- 节点的右子树只包含**大于**当前节点的数。
- 所有左子树和右子树自身必须也是二叉搜索树。

示例 1:

输入:

```
    2
   / \
  1   3
输出: true
```

示例 2:

```
输入:
    5
   / \
  1   4
     / \
    3   6
输出: false
解释: 输入为: [5,1,4,null,null,3,6]。
     根节点的值为 5 ，但是其右子节点值为 4 。
```

10.4.2 分析

按照二叉搜索树的性质，我们可以想到需要递归地进行判断。

这里需要注意的是，如果二叉搜索树的左右子树不为空，那么左子树中的所有节点，值都应该小于根节点；同样右子树中所有节点，值都大于根节点。

10.4.3 方法一：先序遍历

容易想到的方法是，用先序遍历的思路，自顶向下进行遍历。对于每一个节点，先判断它的左右子节点，和当前节点值是否符合大小关系；然后再递归地判断左子树和右子树。

这里需要注意，仅有当前的节点作为参数，做递归调用是不够的。

当前节点如果是父节点的左子节点，那么以它为根的子树所有节点值必须小于父节点；如果是右子节点，则以它为根的子树所有节点值必须大于父节点。所以我们在递归时，还应该把取值范围的“上下界”信息传入。

代码如下：

```
public class ValidateBST {
    public boolean isValidBST(TreeNode root) {
        if ( root == null ) return true;
        return validator(root.left, null, root.val)
                && validator(root.right, root.val, null);
    }
    // 定义一个辅助校验器
    public boolean validator(TreeNode root, Integer lowerBound, Integer
upperBound){
        if ( root == null ) return true;
        // 1. 如果超出了下界，返回false
        if (lowerBound != null && root.val <= lowerBound) {
            return false;
        }
        // 2. 如果超出了上界，返回false
        if (upperBound != null && root.val >= upperBound) {
            return false;
        }
        // 接下来递归判断左右子树，返回结果
        return validator(root.left, lowerBound, root.val)
                && validator(root.right, root.val, upperBound);
    }
}
```

复杂度分析

时间复杂度 ：O(n)，其中 n 为二叉树的节点个数。在递归调用的时候二叉树的每个节点最多被访问一次，因此时间复杂度为 O(n)。

空间复杂度 ：O(logn)，其中 nn 为二叉树的节点个数。递归函数在递归过程中需要为

每一层递归函数分配栈空间，所以这里需要额外的空间且该空间取决于递归的深度，即二叉树的高度，平均情况为 O(logn)。最坏情况下二叉树为一条链，树的高度为 n ，递归最深达到 n 层，故最坏情况下空间复杂度为 O(n)。

10.4.4 方法二：中序遍历

我们知道，对于二叉搜索树，左子树的节点的值均小于根节点的值，根节点的值均小于右子树的值。因此如果进行中序遍历，得到的序列一定是升序序列。

所以我们的判断其实很简单：进行中序遍历，然后判断是否每个值都大于前一个值就可以了。

代码如下：

```
public boolean isValidBST(TreeNode root) {
    inOrderArray = new ArrayList<>();
    // 中序遍历，得到升序数组
    inOrder(root);
    // 遍历数组，判断是否升序
    for ( int i = 0; i < inOrderArray.size(); i++ ){
        if (i > 0 && inOrderArray.get(i) <= inOrderArray.get(i-1))
            return false;
    }
    return true;
}
private ArrayList<Integer> inOrderArray;
// 中序遍历得到升序数组
public void inOrder(TreeNode root){
    if ( root == null ) return;
    inOrder(root.left);
    inOrderArray.add(root.val);
```

```
    inOrder(root.right);
}
```

复杂度分析

时间复杂度：O(n)，其中 n 为二叉树的节点个数。二叉树的每个节点最多被访问一次，因此时间复杂度为 O(n)。

空间复杂度：O(n)，其中 n 为二叉树的节点个数。用到了额外的数组来保存中序遍历的结果，因此需要额外的 O(n) 的空间。

10.4.5 方法三：用栈实现中序遍历

我们也可以不用递归，而使用栈来实现二叉树的中序遍历。

基本思路是：首先沿着左子树一直搜索，把路径上的所有左子节点压栈；然后依次弹栈，访问的顺序就变成自底向上了。弹栈之后，先处理当前节点，再迭代处理右子节点，就实现了中序遍历的过程。

代码如下：

```
public boolean isValidBST(TreeNode root) {
    Deque<TreeNode> stack = new LinkedList<>();
    double preValue = -Double.MAX_VALUE;
    // 遍历访问所有节点
    while ( root != null || !stack.isEmpty() ){
        // 迭代访问节点的左孩子，并入栈
        while ( root != null ){
            stack.push(root);
            root = root.left;
        }
        // 只要栈不为空，就弹出栈顶元素，依次处理
        if (!stack.isEmpty()){
            root = stack.pop();
```

```
            if ( root.val <= preValue )
                return false;
            preValue = root.val;
            root = root.right;
        }
    }
    return true;
}
```

复杂度分析

时间复杂度：O(n)，其中 n 为二叉树的节点个数。二叉树的每个节点最多被访问一次，因此时间复杂度为 O(n)。

空间复杂度：O(n)，其中 n 为二叉树的节点个数。栈最多存储 n 个节点，因此需要额外的 O(n) 的空间。