# LintCode详解

目录

[LintCode详解 1](#_Toc503825441)

[一、 链表 1](#_Toc503825442)

[2.1 删除链表中的元素 1](#_Toc503825443)

[2.2 删除链表中的元素 1](#_Toc503825444)

[2.3 合并两个排序链表 1](#_Toc503825445)

[2.4 交换链表中的节点 1](#_Toc503825446)

[2.5 带环链表 1](#_Toc503825447)

[2.6 重排链表 2](#_Toc503825448)

[二、 哈希表 2](#_Toc503825449)

[2.1 哈希函数 2](#_Toc503825450)

[三、 栈 3](#_Toc503825451)

[四、 数组 3](#_Toc503825452)

[4.1. 二分查找\*\* 3](#_Toc503825453)

[4.2. 固定和的组合（两个数、三个数、四个数）\*\* 4](#_Toc503825454)

[五、 矩阵 4](#_Toc503825455)

[5.1. 矩阵的之字型遍历（二分查找） 4](#_Toc503825456)

[5.2. 搜索二维矩阵 4](#_Toc503825457)

[5.3. 判断数独是否合法 4](#_Toc503825458)

[六、 二叉树 4](#_Toc503825459)

[七、 字符串 5](#_Toc503825460)

[八、 动态规划（CSDN博客总结） 5](#_Toc503825461)

[Java知识点补充（LintCode） 6](#_Toc503825462)

[一、 if、if…else if、if…if 6](#_Toc503825463)

[二、 对象声明 6](#_Toc503825464)

[三、 int转String，再每位平方 6](#_Toc503825465)

[四、 String和char[]的长度 6](#_Toc503825466)

## 链表

### 删除链表中的元素

题目大意：删除链表中等于给定值val的所有节点。

解题思路：声明dummy作为单链表的头节点，只需要一个指针head指向待删除节点的前节点，从dummy开始，判断指针后节点的数值是否等于val，循环终止的条件为head.next != null。

**注意：**

1. 声明dummy节点：以便对“1🡪null”的处理。
2. 一个指针：删除节点必须记录删除前的位置，可以不用记录待删除节点的位置；若指针记录删除节点的位置，则尾节点无法删除。
3. 循环终止的条件。

代码：170927\_452\_removeElements.java

### 删除链表中的元素

题目大意：2🡪1🡪5🡪null和5🡪9🡪2🡪null，输出和7🡪0🡪8🡪null。

解题思路：循环中用sum记录上一次进位的情况，如果list1不为空，加上值，**接着**，如果list2不为空，加上值，一次循环完，修改sum的值为其十位数。

**注意：**

1. 情况分析，易忽略进位的情况“空，空，进位”。
2. 顺序if，见Java记录一。

### 合并两个排序链表

题目大意：将两个排序链表合并为一个新的排序链表。1🡪3🡪8🡪11🡪15🡪null和2🡪null合并为1🡪2🡪3🡪8🡪11🡪15🡪null。

解题思路：“合并”想到“**递归**”。

*Ps:对于连续的相同的操作，除了迭代循环，可以考虑是否有递归的方式。*

**注意：**递归时指针移动的处理。

### 交换链表中的节点

注意：可以直接交换值，以避免指针操作的麻烦。

交换时的指针操作：

代码：171217\_511\_swapNodes.java

### 带环链表

1. 如何判断是否有环？

解题思路：如果有两个头结点指针，一个走的快，一个走的慢，那么若干步以后，快的指针总会超过慢的指针一圈。

1. 如何计算环的长度？

解题思路：第一次相遇（超一圈）时开始计数，第二次相遇时停止计数。

1. 如何判断环的入口点：

解题思路：碰撞点p到连接点的距离=头指针到连接点的距离，因此，分别从碰撞点、头指针开始走，相遇的那个点就是连接点。

1. 如何判断两个链表（不带环）是否相交？

解题思路：将其中的一个链表首尾相连，然后判断另一个链表是否带环即可。

**注意：环链表首先想到是否快慢指针。**

代码：171222\_99\_ reorderList.java

### 重排链表

题目大意：给定一个单链表L: L0→L1→…→Ln-1→Ln，重新排列后为：L0→Ln→L1→Ln-1→L2→Ln-2→…

必须在不改变节点值的情况下进行原地操作。

例如：给出链表 1->2->3->4->null，重新排列后为1->4->2->3->null。

解题思路：1. 快慢指针得到链表中间位置；2. 将后半段倒序排列；3. 后半段插入前半段合并

**注意：得到链表中间位置的方法：快慢指针。**

## 哈希表

哈希（Hash）是一种数据编码方式，将大尺寸的数据（如一句话，一张图片，一段音乐、一个视频等）浓缩到一个数字中，从而方便地实现数据匹配·查找的功能。

哈希算法并不是一个特定的算法而是一类算法的统称。哈希算法也叫散列算法，一般来说满足这样的关系：f(data)=key，输入任意长度的data数据，经过哈希算法处理后输出一个定长的数据key。同时这个过程是**不可逆**的，无法由key逆推出data。

Eg: 比如这里有一万首歌，要求按照某种方式保存好。到时候给你一首新的歌（命名为X），要求你确认新的这首歌是否在那一万首歌之内。无疑，将一万首歌一个一个比对非常慢。但如果存在一种方式，能将一万首歌的每一首的数据浓缩到一个数字（称为哈希码）中，于是得到一万个数字，那么用同样的算法计算新的歌X的编码，看看歌X的编码是否在之前那一万个数字中，就能知道歌X是否在那一万首歌中。将一首歌的5M字节数据浓缩到一个数字中的算法就是哈希算法。那一万首歌按照各自的编码数字从小到大排序后得到的一个表就是哈希表。显然，由于信息量的丢失，有可能多首歌的哈希码是同一个。好的哈希算法会尽量减少这种冲突，让不同的歌有不同的哈希码。最差的哈希算法自然就是所有的歌用那个算法算出来的都是同一个哈希码。

### 哈希函数

题目大意：实现哈希算法。

代码：170929\_128\_hashCode.java

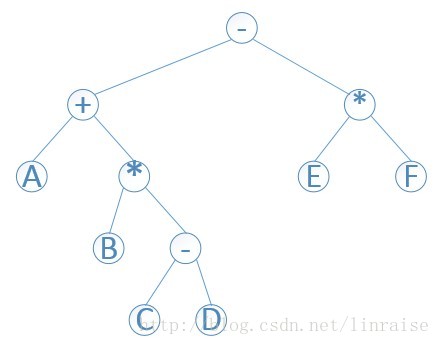
|  |  |
| --- | --- |
| 1  2  3  4 | Map**<**Character**,** Integer**>** map **=** **new** HashMap**<**Character**,** Integer**>();**  map**.**get**(**key**);**  map**.**put**(**c**,** map**.**get**(**key**)+**1**);**  map**.**containsValue**(**value**);** |

拓展：遍历的简写：for (Integer iter : map.values())

## 栈

|  |  |
| --- | --- |
| 1  2  3  4  5 | Stack**<**Character**>** symbol **=** **new** Stack**<**Character**>();**  sta**.**peek**();** //查看栈顶而不移除  sta**.**push**();** //压栈  sta**.**pop**();** //出栈  sta**.**empty**();** //是否为空 |

1. **\*波兰表达式，逆波兰表达式**



波兰式即**前缀表达式**，表达式的前缀表达式是由相应的语法树的前序遍历的结果得到的。计算方式：

用一个栈S2来实现计算，扫描从右往左进行，如果扫描到操作数，则压进S2，如果扫描到操作符，则从S2弹出两个操作数进行相应的操作，并将结果压进S2(S2的个数出2个进1个),当扫描结束后，S2的栈顶就是表达式结果。

eg: + A \* B - C D \* E F

逆波兰式即后缀表达式。计算方式：可以用一个栈S2来实现计算，扫描从左往右进行，如果扫描到操作数，则压进S2，如果扫描到操作符，则从S2弹出两个操作数进行相应的操作，并将结果压进S2(S2的个数出2个进1个),当扫描结束后，S2的栈顶就是表达式结果。

eg: A B C D - \* + E F \* -

## 数组

### 二分查找\*\*

题目包括：返回位置、第一个的位置、最后一个的位置，递归、非递归。

解题思路：共设置三个指针，分别记录start、middle、end。

1. 非递归：最外层循环设置判断(start + 1 < end)，即区间的长度小于2。

返回不同位置的区别在于：以第一个位置为例，在找到int[]中的值 >= target时，需要将其middle作为end继续循环，直到区间长度限制。返回最后位置时，在int[] <= target时，将middle作为start继续循环。普通位置，找到 == 就可以直接输出。

1. 递归：最外层判定(start <= end)，内部判断(target ? nums[mid])，大于则右边递归，直接将end设置为mid进行return；小于则左边递归。

*详见Git中Practice项目中的BinarySearch()：*

*https://github.com/twostarxx/Practice/blob/master/src/practice/BinarySearch.java*

### 固定和的组合（两个数、三个数、四个数）\*\*

## 矩阵

### 矩阵的之字型遍历（二分查找）

题目大意：[[1, 2, 3, 4], [5, 6, 7, 8], [9, 10, 11, 12]]，返回 [1, 2, 5, 9, 6, 3, 4, 7, 10, 11, 8, 12]。

解题思路：判断遍历的顺序，进而列举并分类转折的情况。以上题为例进行讲解。

顺序：斜上 🡪 向右 🡪 斜下 🡪 向下 🡪 斜上 🡪 向右 🡪 斜下 🡪 向右 🡪 斜上 🡪 向下

合并后分类：斜上 🡪 向右 or 向下；斜下 🡪 向下 or 向右。（此处的向右或向下有先后顺序影响）

斜上用while，然后向右or向下用if；斜下用while，然后向下or向右用if，最外层用while循环完所有的点。

代码题号：185

### 搜索二维矩阵

题目大意：用O(logn)搜索m\*n矩阵中的值。这个矩阵具有一下特性：每行中的整数从左到右是排序的；每行 第一个数大于上一行的最后一个整数。

解题思路：看到搜索，最先想到用二分法。此题可简单用二分法搜索，将二维看成一维，将index计算成行纵坐标即可。第二种方法：先二分搜索所在行，再二分搜索所在列。

代码题号：28

### 判断数独是否合法

题目大意：

解题思路

知识点：Set类的用法。

代码题号：389

## 二叉树

1. 二叉树的遍历
2. 二叉搜索树

定义：或者是一棵空树，或者是具有一下性质的二叉树：若它的左子树不空，则左子树上的所有节点的值均小于它的根节点的值；若它的右子树不空，则右子树上的节点的值均大于它的根节点的值；它的左右子树也分别为二叉搜索树。

1. 判断是否为子树

解题思路：对T1的左右子树进行递归，判断其左右子树是否与T2相同。

代码题号：245

1. **递归函数的框架**

part1：端节点的处理。

if (root == null)

if (root.left == null)

if (root.right == null)

part：每层的处理，或者左右子树的处理

每层：return fun(root.left) > fun(root.right ? fun(root.left) : fun(root.right)) + 1;

左右的处理：

if (root.left != null) fun(root.left);

注：当root.left == null时，仍然可以将null 传参入新的递归。在part1的端节点可以直接处理。

## 字符串

常用转换：

1. int 🡪 String: String.valueOf(num)
2. char 🡪 int（获取数值）: int n = ‘1’ – ‘0’ // 必须要减去0才是获取数值

## 动态规划（CSDN博客总结）

1. 最优动态规划

eg: 110.最小路径和，109.数字三角形

解题思路：逆推，全局最优则局部最优。

1. 遍历动态规划

eg: 114.不同的路径

解题思路：顺推，局部遍历进而全局遍历。

## 贪心算法

1. 最大子数组

# Java知识点补充（LintCode）

## if、if…else if、if…if

if…else if：多重if，在条件1不满足的情况下，才会进行条件2的判断。

if…if：多个判断，顺序执行。

## 对象声明

p.next = new ListNode(sum % 10);

## int转String，再每位平方

|  |  |
| --- | --- |
| 1  2  3  4  5 | int n **=** 324**;**  String cn **=** n **+** ""**;** //int转String的常用方法  **for** **(**int i**=**0**;** i **<** cn**.**length**();** i**++){**  m **+=** **(**Integer**.**parseInt**(**cn**.**substring**(**i**,**i**+**1**)))\*(**Integer**.**parseInt**(**cn**.**substring**(**i**,**i**+**1**)));** //Java中^2表示亦或，而不是平方  **}** |

次方：

import java.lang.Math;

Math.pow(double a,double b) 返回第一个参数的第二个参数次幂的值。

## String和char[]的长度

String .length**()**;

char[] .length;

## 负数减负数

负数减负数会存在溢出的情况，比如(-5)-(-4)。因此用Integer.nax()-Integr.min()更为保险。