



尚硅谷 Java 数据结构和算法

尚硅谷-韩顺平



第 1 章 内容介绍和授课方式.....	1
1.1 数据结构和算法内容介绍.....	1
1.1.1 先看几个经典的算法面试题.....	1
1.1.2 数据结构和算法的重要性.....	3
1.1.3 本套数据结构和算法内容介绍.....	4
1.1.4 课程亮点和授课方式.....	4
第 2 章 数据结构和算法概述.....	5
2.1.1 数据结构和算法的关系.....	5
2.2 看几个实际编程中遇到的问题.....	5
2.2.1 问题一-字符串替换问题.....	5
2.2.2 一个五子棋程序.....	5
2.2.3 约瑟夫(Josephu)问题(丢手帕问题).....	6
2.2.4 其它常见算法问题:.....	6
2.3 线性结构和非线性结构.....	7
2.3.1 线性结构.....	7
2.3.2 非线性结构.....	7
第 3 章 稀疏数组和队列.....	8
3.1 稀疏 SPARSEARRAY 数组.....	8
3.1.1 先看一个实际的需求.....	8
3.1.2 基本介绍.....	8
3.1.3 应用实例.....	9
3.1.4 课后练习.....	14
3.2 队列.....	14
3.2.1 队列的一个使用场景.....	14
3.2.2 队列介绍.....	14
3.2.3 数组模拟队列思路.....	15
3.2.4 数组模拟环形队列.....	21
第 4 章 链表.....	28
4.1 链表(LINKED LIST)介绍.....	28
4.2 单链表的应用实例.....	29
4.3 单链表面试题(新浪、百度、腾讯).....	38
4.4 双向链表应用实例.....	44
4.4.1 双向链表的操作分析和实现.....	44
4.4.2 课堂作业和思路提示.....	53
4.5 单向环形链表应用场景.....	53
4.6 单向环形链表介绍.....	53



4.7 JOSEPHU 问题.....	54
4.8 JOSEPHU 问题的代码实现.....	55
第 5 章 栈.....	62
5.1 栈的一个实际需求.....	62
5.2 栈的介绍.....	62
5.3 栈的应用场景.....	63
5.4 栈的快速入门.....	63
5.5 栈实现综合计算器(中缀表达式).....	68
5.6 逆波兰计算器.....	77
5.7 中缀表达式转换为后缀表达式.....	81
5.7.1 具体步骤如下:.....	81
5.7.2 举例说明:.....	81
5.7.3 代码实现中缀表达式转为后缀表达式.....	82
5.8 逆波兰计算器完整版.....	91
5.8.1 完整版的逆波兰计算器, 功能包括.....	91
第 6 章 递归.....	100
6.1 递归应用场景.....	100
6.2 递归的概念.....	100
6.3 递归调用机制.....	100
6.4 递归能解决什么样的问题.....	102
6.5 递归需要遵守的重要规则.....	103
6.6 递归-迷宫问题.....	103
6.6.1 迷宫问题.....	103
6.6.2 代码实现:.....	103
6.6.3 对迷宫问题的讨论.....	108
6.7 递归-八皇后问题(回溯算法).....	108
6.7.1 八皇后问题介绍.....	108
6.7.2 八皇后问题算法思路分析.....	109
6.7.3 八皇后问题算法代码实现.....	110
第 7 章 排序算法.....	114
7.1 排序算法的介绍.....	114
7.2 排序的分类:	114
7.3 算法的时间复杂度.....	114
7.3.1 度量一个程序(算法)执行时间的两种方法.....	114
7.3.2 时间频度.....	115
7.3.3 时间复杂度.....	117
7.3.4 常见的时间复杂度.....	117



7.3.5 平均时间复杂度和最坏时间复杂度.....	120
7.4 算法的空间复杂度简介.....	121
7.4.1 基本介绍.....	121
7.5 冒泡排序.....	121
7.5.1 基本介绍.....	121
7.5.2 演示冒泡过程的例子(图解).....	122
7.5.3 冒泡排序应用实例.....	122
7.6 选择排序.....	127
7.6.1 基本介绍.....	127
7.6.2 选择排序思想.....	127
7.6.3 选择排序思路分析图.....	128
7.6.4 选择排序应用实例.....	129
7.7 插入排序.....	134
7.7.1 插入排序法介绍.....	134
7.7.2 插入排序法思想.....	134
7.7.3 插入排序思路图.....	134
7.7.4 插入排序法应用实例.....	135
7.8 希尔排序.....	140
7.8.1 简单插入排序存在的问题.....	140
7.8.2 希尔排序法介绍.....	140
7.8.3 希尔排序法基本思想.....	141
7.8.4 希尔排序法的示意图.....	141
7.8.5 希尔排序法应用实例.....	142
7.9 快速排序.....	147
7.9.1 快速排序法介绍.....	147
7.9.2 快速排序法示意图.....	147
7.9.3 快速排序法应用实例.....	148
7.10 归并排序.....	152
7.10.1 归并排序介绍.....	152
7.10.2 归并排序思想示意图 1-基本思想.....	152
7.10.3 归并排序思想示意图 2-合并相邻有序子序列.....	153
7.10.4 归并排序的应用实例.....	153
7.11 基数排序.....	158
7.11.1 基数排序(桶排序)介绍.....	158
7.11.2 基数排序基本思想.....	158
7.11.3 基数排序图文说明.....	159
7.11.4 基数排序代码实现.....	160
7.11.5 基数排序的说明.....	167
7.12 常用排序算法总结和对比.....	167



7.12.1 一张排序算法的比较图.....	167
7.12.2 相关术语解释:	168
第 8 章 查找算法.....	169
8.1 查找算法介绍.....	169
8.2 线性查找算法.....	169
8.3 二分查找算法.....	170
8.3.1 二分查找:	170
8.3.2 二分查找算法的思路.....	170
8.3.3 二分查找的代码.....	171
8.4 插值查找算法.....	176
8.4.1 插值查找应用案例:	176
8.4.2 插值查找注意事项:	180
8.5 斐波那契(黄金分割法)查找算法.....	180
8.5.1 斐波那契(黄金分割法)查找基本介绍:.....	180
8.5.2 斐波那契(黄金分割法)原理:.....	180
8.5.3 斐波那契查找应用案例:	181
第 9 章 哈希表.....	185
9.1 哈希表(散列)-GOOGLE 上机题.....	185
9.2 哈希表的基本介绍.....	185
9.3 GOOGLE 公司的一个上机题:.....	186
第 10 章 树结构的基础部分.....	195
10.1 二叉树.....	195
10.1.1 为什么需要树这种数据结构.....	195
10.1.2 树示意图.....	196
10.1.3 二叉树的概念.....	197
10.1.4 二叉树遍历的说明.....	198
10.1.5 二叉树遍历应用实例(前序,中序,后序).....	199
10.1.6 二叉树-查找指定节点.....	205
10.1.7 二叉树-删除节点.....	216
10.1.8 二叉树-删除节点.....	220
10.2 顺序存储二叉树.....	220
10.2.1 顺序存储二叉树的概念.....	220
10.2.2 顺序存储二叉树遍历.....	221
10.2.3 顺序存储二叉树应用实例.....	224
10.3 线索化二叉树.....	224
10.3.1 先看一个问题.....	224
10.3.2 线索二叉树基本介绍.....	224



10.3.3 线索二叉树应用案例.....	225
10.3.4 遍历线索化二叉树.....	242
10.3.5 线索化二叉树的课后作业:.....	243
第 11 章 树结构实际应用.....	244
11.1 堆排序.....	244
11.1.1 堆排序基本介绍.....	244
11.1.2 堆排序基本思想.....	245
11.1.3 堆排序步骤图解说明.....	245
11.1.4 堆排序代码实现.....	250
11.2 赫夫曼树.....	254
11.2.1 基本介绍.....	254
11.2.2 赫夫曼树几个重要概念和举例说明.....	254
11.2.3 赫夫曼树创建思路图解.....	255
11.2.4 赫夫曼树的代码实现.....	256
11.3 赫夫曼编码.....	261
11.3.1 基本介绍.....	261
11.3.2 原理剖析.....	261
11.3.3 最佳实践-数据压缩(创建赫夫曼树).....	264
11.3.4 最佳实践-数据压缩(生成赫夫曼编码和赫夫曼编码后的数据).....	266
11.3.5 最佳实践-数据解压(使用赫夫曼编码解码).....	268
11.3.6 最佳实践-文件压缩.....	271
11.3.7 最佳实践-文件解压(文件恢复).....	273
11.3.8 代码汇总, 把前面所有的方法放在一起.....	275
11.3.9 赫夫曼编码压缩文件注意事项.....	294
11.4 二叉排序树.....	294
11.4.1 先看一个需求.....	294
11.4.2 解决方案分析.....	294
11.4.3 二叉排序树介绍.....	295
11.4.4 二叉排序树创建和遍历.....	296
11.4.5 二叉排序树的删除.....	296
11.4.6 二叉排序树删除结点的代码实现:.....	298
11.4.7 课后练习: 完成老师代码, 并使用第二种方式来解决.....	309
11.5 平衡二叉树(AVL 树).....	309
11.5.1 看一个案例(说明二叉排序树可能的问题).....	309
11.5.2 基本介绍.....	310
11.5.3 应用案例-单旋转(左旋转).....	310
11.5.4 应用案例-单旋转(右旋转).....	312
11.5.5 应用案例-双旋转.....	313



第 12 章 多路查找树.....	328
12.1 二叉树与 B 树.....	328
12.1.1 二叉树的问题分析.....	328
12.1.2 多叉树.....	328
12.1.3 B 树的基本介绍.....	329
12.2 2-3 树.....	329
12.2.1 2-3 树是最简单的 B 树结构，具有如下特点:.....	329
12.2.2 2-3 树应用案例.....	330
12.2.3 其它说明.....	330
12.3 B 树、B+树和 B*树.....	331
12.3.1 B 树的介绍.....	331
12.3.2 B 树的介绍.....	331
12.3.3 B+树的介绍.....	332
12.3.4 B*树的介绍.....	333
第 13 章 图.....	334
13.1 图基本介绍.....	334
13.1.1 为什么要有图.....	334
13.1.2 图的举例说明.....	334
13.1.3 图的常用概念.....	334
13.2 图的表示方式.....	336
13.2.1 邻接矩阵.....	336
13.2.2 邻接表.....	336
13.3 图的快速入门案例.....	337
13.4 图的深度优先遍历介绍.....	338
13.4.1 图遍历介绍.....	338
13.4.2 深度优先遍历基本思想.....	338
13.4.3 深度优先遍历算法步骤.....	338
13.4.4 深度优先算法的代码实现.....	339
13.5 图的广度优先遍历.....	340
13.5.1 广度优先遍历基本思想.....	340
13.5.2 广度优先遍历算法步骤.....	340
13.5.3 广度优先算法的图示.....	341
13.6 广度优先算法的代码实现.....	341
13.7 图的代码汇总.....	343
13.8 图的深度优先 VS 广度优先.....	351
第 14 章 程序员常用 10 种算法.....	352
14.1 二分查找算法(非递归).....	352



14.1.1 二分查找算法(非递归)介绍.....	352
14.1.2 二分查找算法(非递归)代码实现.....	352
14.2 分治算法.....	354
14.2.1 分治算法介绍.....	354
14.2.2 分治算法的基本步骤.....	354
14.2.3 分治(Divide-and-Conquer(P))算法设计模式如下：	355
14.2.4 分治算法最佳实践-汉诺塔.....	355
14.3 动态规划算法.....	357
14.3.1 应用场景-背包问题.....	357
14.3.2 动态规划算法介绍.....	358
14.3.3 动态规划算法最佳实践-背包问题.....	358
14.3.4 动态规划-背包问题的代码实现.....	360
14.4 KMP 算法.....	363
14.4.1 应用场景-字符串匹配问题.....	364
14.4.2 暴力匹配算法.....	364
14.4.3 KMP 算法介绍.....	366
14.4.4 KMP 算法最佳应用-字符串匹配问题.....	366
14.5 贪心算法.....	374
14.5.1 应用场景-集合覆盖问题.....	374
14.5.2 贪心算法介绍.....	374
14.5.3 贪心算法最佳应用-集合覆盖.....	375
14.5.4 贪心算法注意事项和细节.....	380
14.6 普里姆算法.....	381
14.6.1 应用场景-修路问题.....	381
14.6.2 最小生成树.....	381
14.6.3 普里姆算法介绍.....	382
14.6.4 普里姆算法最佳实践(修路问题).....	383
14.7 克鲁斯卡尔算法.....	388
14.7.1 应用场景-公交站问题.....	388
14.7.2 克鲁斯卡尔算法介绍.....	388
14.7.3 克鲁斯卡尔算法图解说明.....	389
3.1.1 克鲁斯卡尔算法图解.....	390
3.1.2 克鲁斯卡尔算法分析.....	392
3.1.3 如何判断是否构成回路-举例说明(如图).....	392
3.1.4 克鲁斯卡尔算法的代码说明.....	393
14.7.4 克鲁斯卡尔最佳实践-公交站问题.....	393
14.8 迪杰斯特拉算法.....	401
14.8.1 应用场景-最短路径问题.....	401
14.8.2 迪杰斯特拉(Dijkstra)算法介绍.....	402



14.8.3 迪杰斯特拉(Dijkstra)算法过程.....	402
14.8.4 迪杰斯特拉(Dijkstra)算法最佳应用-最短路径.....	403
14.9 弗洛伊德算法.....	412
14.9.1 弗洛伊德(Floyd)算法介绍.....	412
14.9.2 弗洛伊德(Floyd)算法图解分析.....	412
14.9.3 弗洛伊德(Floyd)算法最佳应用-最短路径.....	415
14.10 马踏棋盘算法.....	419
14.10.1 马踏棋盘算法介绍和游戏演示.....	419
14.10.2 马踏棋盘游戏代码实现.....	420



第 1 章 内容介绍和授课方式

1.1 数据结构和算法内容介绍

1.1.1 先看几个经典的算法面试题

➤ 字符串匹配问题：：

1) 有一个字符串 str1= ""硅谷 尚硅谷你尚硅 尚硅谷你尚硅谷你尚硅你好""，和一个子串 str2="尚硅谷你尚硅谷你"

2) 现在要判断 str1 是否含有 str2，如果存在，就返回第一次出现的位置，如果没有，则返回-1

3) 要求用最快的速度来完成匹配

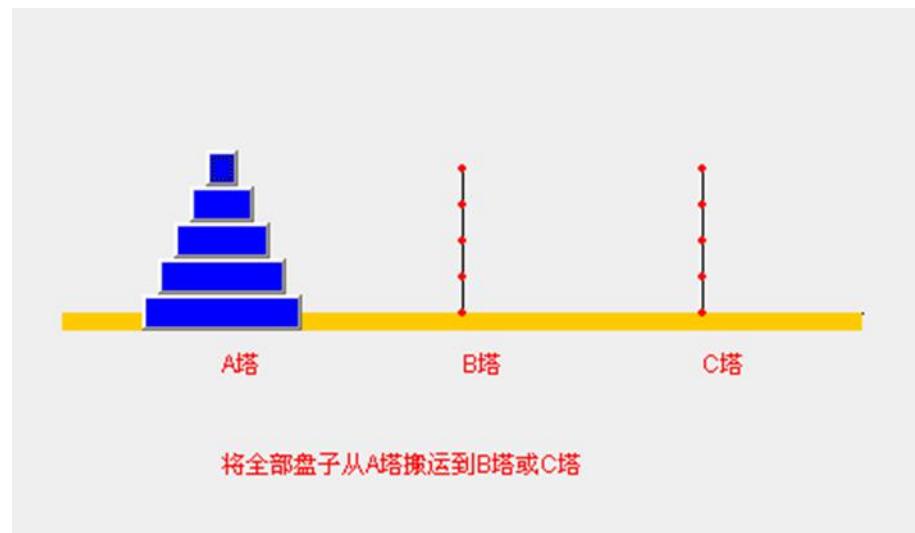
4) 你的思路是什么？

● 暴力匹配 [简单，但是效率低]

● KMP 算法《部分匹配表》

➤ 汉诺塔游戏，

请完成汉诺塔游戏的代码：要求：1) 将 A 塔的所有圆盘移动到 C 塔。并且规定，在 2) 小圆盘上不能放大圆盘，3) 在三根柱子之间一次只能移动一个圆盘



➤ 八皇后问题

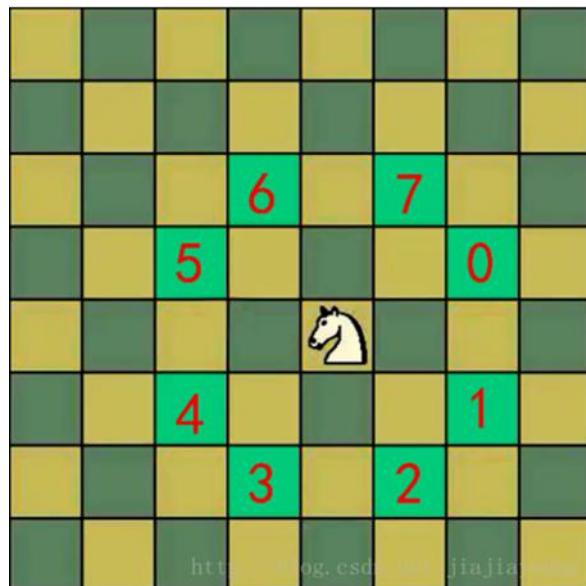
八皇后问题，是一个古老而著名的问题，是回溯算法的典型案例。该问题是国际西洋棋棋手马克斯·贝瑟尔于1848年提出：在 8×8 格的国际象棋上摆放八个皇后，使其不能互相攻击，即：任意两个皇后都不能处于同一行、同一列或同一斜线上，问有多少种摆法。【92】=> 分治算法



➤ 马踏棋盘算法介绍和游戏演示

- 1) 马踏棋盘算法也被称为骑士周游问题
- 2) 将马随机放在国际象棋的 8×8 棋盘 $\text{Board}[0 \sim 7][0 \sim 7]$ 的某个方格中，马按走棋规则(马走日字)进行移动。要求每个方格只进入一次，走遍棋盘上全部64个方格
- 3) 游戏演示: http://www.4399.com/flash/146267_2.htm

- 4) 会使用到图的深度优化遍历算法(DFS) + 贪心算法优化



1.1.2 数据结构和算法的重要性

- 1) 算法是程序的灵魂，优秀的程序可以在海量数据计算时，依然保持高速计算
- 2) 一般来讲 程序会使用了内存计算框架(比如 Spark)和缓存技术(比如 Redis 等)来优化程序,再深入的思考一下，这些计算框架和缓存技术， 它的核心功能是哪个部分呢？
- 3) 拿实际工作经历来说, 在 Unix 下开发服务器程序，功能是要支持上千万人同时在线，在上线前，做内测，一切 OK,可上线后，服务器就支撑不住了, 公司的 CTO 对代码进行优化，再次上线，坚如磐石。你就能感受到程序是有灵魂的，就是算法。
- 4) 目前程序员面试的门槛越来越高，很多一线 IT 公司(大厂)，都会有数据结构和算法面试题(负责的告诉你，肯定有的)
- 5) 如果你不永远都是代码工人,那就花时间来研究下数据结构和算法

1.1.3 本套数据结构和算法内容介绍



尚硅谷 韩顺平 数据结构和算法2019 内容介绍.zip

1.1.4 课程亮点和授课方式

- 1) 课程深入,非蜻蜓点水
- 2) 课程成体系, 非星星点灯
- 3) 高效而愉快的学习 , 数据结构和算法很有用, 很好玩
- 4) 数据结构和算法很重要, 但是相对困难, 我们努力做到通俗易懂
- 5) 采用 应用场景->数据结构或算法->剖析原理->分析实现步骤(图解)->代码实现 的步骤讲解 [比如: 八皇后问题和动态规划算法]
- 6) 课程目标: 让大家掌握本质 , 到达能在工作中灵活运用解决实际问题和优化程序的目的.

第 2 章 数据结构和算法概述

2.1.1 数据结构和算法的关系

- 1) 数据 data 结构(structure)是一门研究组织数据方式的学科，有了编程语言也就有了数据结构.学好数据结构可以编写出更加漂亮,更加有效率的代码。
- 2) 要学习好数据结构就要多多考虑如何将生活中遇到的问题,用程序去实现解决.
- 3) 程序 = 数据结构 + 算法
- 4) 数据结构是算法的基础, 换言之, 想要学好算法, 需要把数据结构学到位。

2.2 看几个实际编程中遇到的问题

2.2.1 问题一-字符串替换问题

Java 代码:

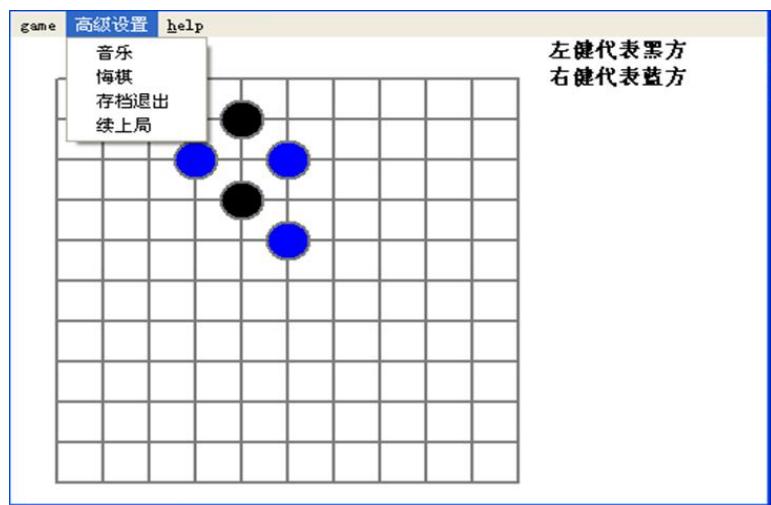
```
public static void main(String[] args) {  
    String str = "Java,Java, hello,world!";  
    String newStr = str.replaceAll("Java", "尚硅谷~"); //算法  
    System.out.println("newStr=" + newStr);  
}
```



问：试写出用单链表表示的字符串类及字符串结点类的定义，并依次实现它的构造函数、以及计算串长度、串赋值、判断两串相等、求子串、两串连接、求子串在串中位置等7个成员函数。

小结：需要使用到单链表数据结构

2.2.2 一个五子棋程序



如何判断游戏的输赢，并可以完成存盘退出和继续上局的功能

- 1) 棋盘 二维数组=>(稀疏数组)-> 写入文件 【存档功能】
- 2) 读取文件-> 稀疏数组-> 二维数组 -> 棋盘 【接上局】

2.2.3 约瑟夫(Josephu)问题(丢手帕问题)

- 1) Josephu 问题为：设编号为 1, 2, … n 的 n 个人围坐一圈，约定编号为 k ($1 \leq k \leq n$) 的人从 1 开始报数，数到 m 的那个人出列，它的下一位又从 1 开始报数，数到 m 的那个人又出列，依次类推，直到所有人出列为止，由此产生一个出队编号的序列。
- 2) 提示：用一个不带头结点的循环链表来处理 Josephu 问题：先构成一个有 n 个结点的单循环链表（单向环形链表），然后由 k 结点起从 1 开始计数，计到 m 时，对应结点从链表中删除，然后再从被删除结点的下一个结点又从 1 开始计数，直到最后一个结点从链表中删除算法结束。
- 3) 小结：完成约瑟夫问题，需要使用到单向环形链表 这个数据结构

2.2.4 其它常见算法问题：



- 1) 修路问题 => 最小生成树(加权值)【数据结构】 + 普利姆算法
- 2) 最短路径问题 => 图+弗洛伊德算法
- 3) 汉诺塔 => 分支算法
- 4) 八皇后问题 => 回溯法

2.3 线性结构和非线性结构

数据结构包括：线性结构和非线性结构。

2.3.1 线性结构

- 1) 线性结构作为最常用的数据结构，其特点是**数据元素之间存在一对一的线性关系**
- 2) 线性结构有两种不同的存储结构，即**顺序存储结构(数组)**和**链式存储结构(链表)**。顺序存储的线性表称为**顺序表**，顺序表中的**存储元素是连续的**
- 3) 链式存储的线性表称为**链表**，链表中的**存储元素不一定是连续的**，元素节点中存放数据元素以及相邻元素的地址信息
- 4) 线性结构常见的有：**数组、队列、链表和栈**，后面我们会详细讲解.

2.3.2 非线性结构

非线性结构包括：二维数组，多维数组，广义表，树结构，图结构

第 3 章 稀疏数组和队列

3.1 稀疏 sparsearray 数组

3.1.1 先看一个实际的需求

- 编写的五子棋程序中，有存盘退出和续上盘的功能。



- 分析问题：

因为该二维数组的很多值是默认值 0，因此记录了很多没有意义的数据。->稀疏数组。

3.1.2 基本介绍

当一个数组中大部分元素为 0，或者为同一个值的数组时，可以使用稀疏数组来保存该数组。

稀疏数组的处理方法是：

- 1) 记录数组一共有几行几列，有多少个不同的值
- 2) 把具有不同值的元素的行列及值记录在一个小规模的数组中，从而缩小程序的规模

- 稀疏数组举例说明

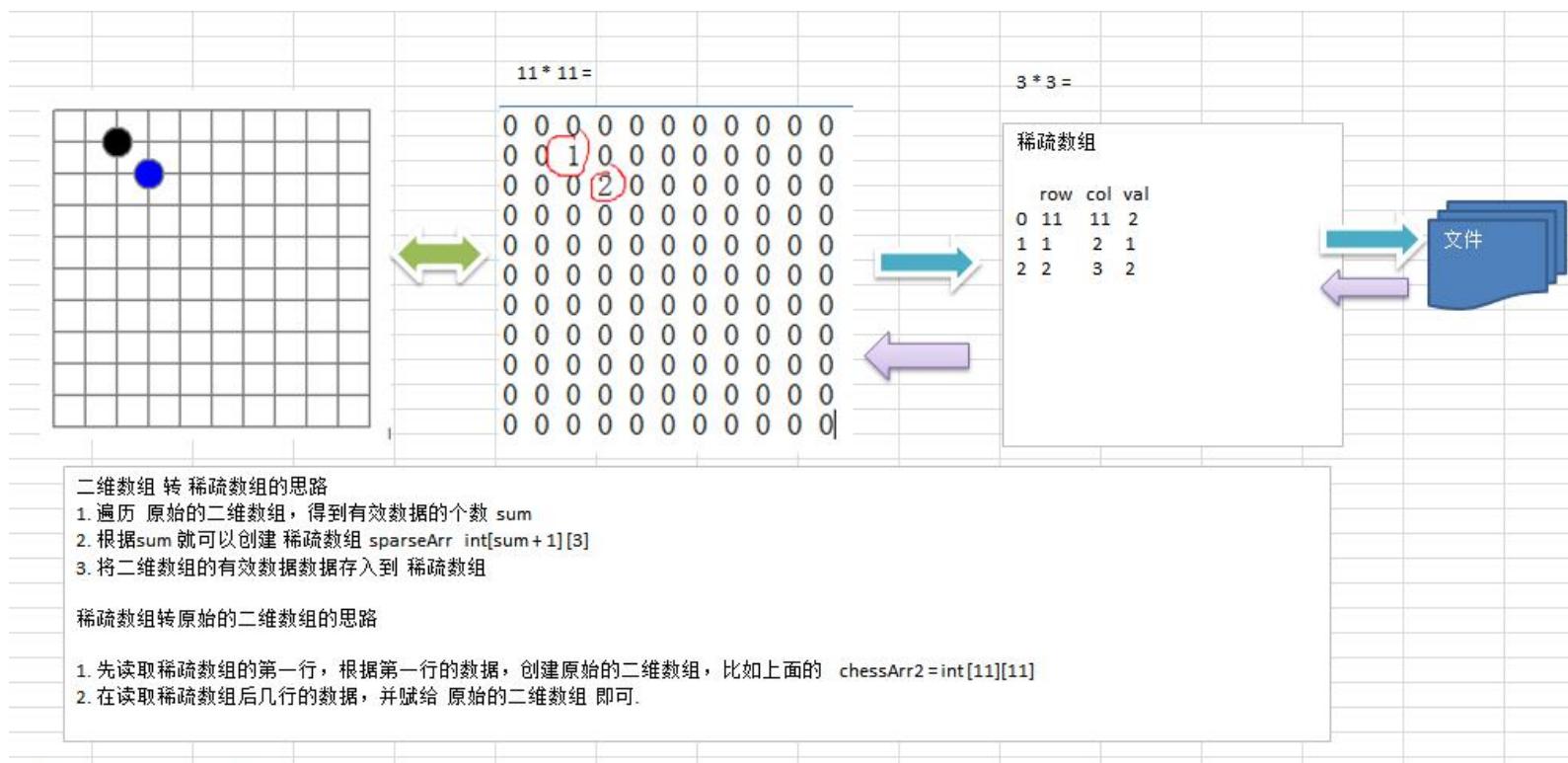


The diagram illustrates the conversion of a sparse matrix into a compressed row representation. On the left, a 6x6 matrix is shown with non-zero elements highlighted in red. A red arrow points from the matrix to a table on the right. The table has columns labeled '行 (row)' and '列 (col)'. The first row of the table corresponds to the first non-zero element in the matrix, with values [0], 6, 7, and 8 respectively. Subsequent rows correspond to the other non-zero elements, with the fifth row highlighted in red. A red arrow points from the fifth row of the table to the matrix, indicating the mapping between the compressed representation and the original sparse matrix.

	行 (row)	列 (col)	值 (value)	稀疏数组
[0]	6	7	8	
[1]	0	3	22	
[2]	0	6	15	
[3]	1	1	11	
[4]	1	5	17	
[5]	2	3	-6	
[6]	3	5	39	
[7]	4	0	91	
[8]	5	2	28	

3.1.3 应用实例

- 1) 使用稀疏数组，来保留类似前面的二维数组(棋盘、地图等等)
- 2) 把稀疏数组存盘，并且可以从新恢复原来的二维数组数
- 3) 整体思路分析



二维数组 转 稀疏数组的思路

1. 遍历 原始的二维数组，得到有效数据的个数 sum
2. 根据sum 就可以创建 稀疏数组 sparseArr int[sum + 1][3]
3. 将二维数组的有效数据存入到 稀疏数组

稀疏数组转原始的二维数组的思路

1. 先读取稀疏数组的第一行，根据第一行的数据，创建原始的二维数组，比如上面的 chessArr2 = int[11][11]
2. 在读取稀疏数组后几行的数据，并赋给 原始的二维数组 即可。

4) 代码实现

```
package com.atguigu.sparsearray;

public class SparseArray {

    public static void main(String[] args) {
        // 创建一个原始的二维数组 11 * 11
        // 0: 表示没有棋子， 1 表示 黑子 2 表蓝子
        int chessArr1[][] = new int[11][11];
        chessArr1[1][2] = 1;
        chessArr1[2][3] = 2;
        chessArr1[4][5] = 2;
    }
}
```



```
// 输出原始的二维数组
System.out.println("原始的二维数组~~");
for (int[] row : chessArr1) {
    for (int data : row) {
        System.out.printf("%d\t", data);
    }
    System.out.println();
}

// 将二维数组 转 稀疏数组的思
// 1. 先遍历二维数组 得到非 0 数据的个数
int sum = 0;
for (int i = 0; i < 11; i++) {
    for (int j = 0; j < 11; j++) {
        if (chessArr1[i][j] != 0) {
            sum++;
        }
    }
}

// 2. 创建对应的稀疏数组
int sparseArr[][] = new int[sum + 1][3];
// 给稀疏数组赋值
sparseArr[0][0] = 11;
sparseArr[0][1] = 11;
sparseArr[0][2] = sum;
```



```
// 遍历二维数组，将非 0 的值存放到 sparseArr 中
int count = 0; //count 用于记录是第几个非 0 数据
for (int i = 0; i < 11; i++) {
    for (int j = 0; j < 11; j++) {
        if (chessArr1[i][j] != 0) {
            count++;
            sparseArr[count][0] = i;
            sparseArr[count][1] = j;
            sparseArr[count][2] = chessArr1[i][j];
        }
    }
}

// 输出稀疏数组的形式
System.out.println();
System.out.println("得到稀疏数组为~~~~");
for (int i = 0; i < sparseArr.length; i++) {
    System.out.printf("%d\t%d\t%d\n", sparseArr[i][0], sparseArr[i][1], sparseArr[i][2]);
}
System.out.println();

// 将稀疏数组 --> 恢复成 原始的二维数组
/*
 * 1. 先读取稀疏数组的第一行，根据第一行的数据，创建原始的二维数组，比如上面的 chessArr2 = int
 [11][11]
```



2. 在读取稀疏数组后几行的数据，并赋给 原始的二维数组 即可。

*/

//1. 先读取稀疏数组的第一行，根据第一行的数据，创建原始的二维数组

```
int chessArr2[][] = new int[sparseArr[0][0]][sparseArr[0][1]];
```

//2. 在读取稀疏数组后几行的数据(从第二行开始)，并赋给 原始的二维数组 即可

```
for(int i = 1; i < sparseArr.length; i++) {  
    chessArr2[sparseArr[i][0]][sparseArr[i][1]] = sparseArr[i][2];  
}
```

// 输出恢复后的二维数组

```
System.out.println();  
System.out.println("恢复后的二维数组");
```

```
for (int[] row : chessArr2) {  
    for (int data : row) {  
        System.out.printf("%d\t", data);  
    }  
    System.out.println();  
}  
}
```

3.1.4 课后练习

要求:

- 1) 在前面的基础上, 将稀疏数组保存到磁盘上, 比如 map.data
- 2) 恢复原来的数组时, 读取 map.data 进行恢复

3.2 队列

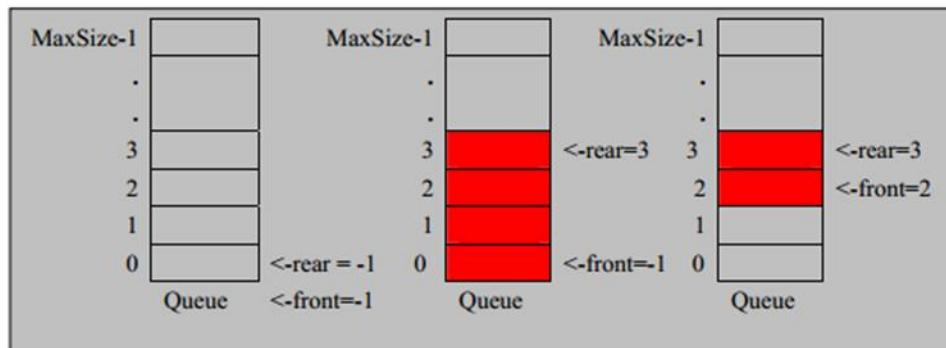
3.2.1 队列的一个使用场景

银行排队的案例:



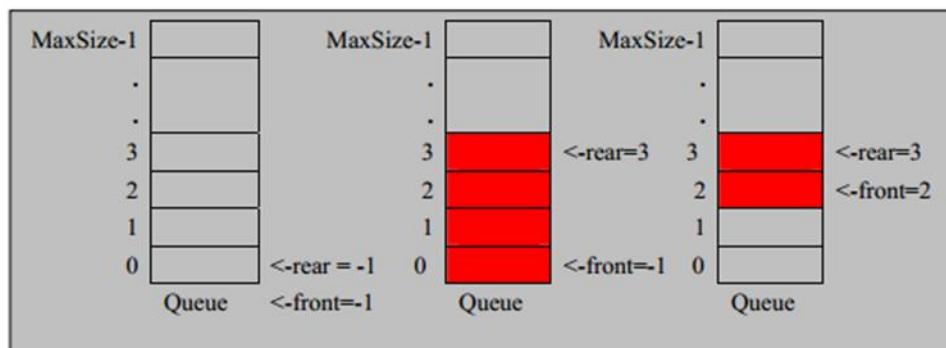
3.2.2 队列介绍

- 1) 队列是一个有序列表, 可以用数组或是链表来实现。
- 2) 遵循先入先出的原则。即: 先存入队列的数据, 要先取出。后存入的要后取出
- 3) 示意图: (使用数组模拟队列示意图)



3.2.3 数组模拟队列思路

- 队列本身是有序列表，若使用数组的结构来存储队列的数据，则队列数组的声明如下图，其中 `maxSize` 是该队列的最大容量。
- 因为队列的输出、输入是分别从前后端来处理，因此需要两个变量 `front` 及 `rear` 分别记录队列前后端的下标，`front` 会随着数据输出而改变，而 `rear` 则是随着数据输入而改变，如图所示：



- 当我们将数据存入队列时称为”`addQueue`”，`addQueue` 的处理需要有两个步骤：思路分析
 - 1) 将尾指针往后移：`rear+1`，当 `front == rear` 【空】
 - 2) 若尾指针 `rear` 小于队列的最大下标 `maxSize-1`，则将数据存入 `rear` 所指的数组元素中，否则无法存入数据。

`rear == maxSize - 1` [队列满]

- 代码实现

```
package com.atguigu.queue;
```



```
import java.util.Scanner;

public class ArrayQueueDemo {

    public static void main(String[] args) {
        //测试一把
        //创建一个队列
        ArrayQueue queue = new ArrayQueue(3);
        char key = ' '; //接收用户输入
        Scanner scanner = new Scanner(System.in);//
        boolean loop = true;
        //输出一个菜单
        while(loop) {
            System.out.println("s(show): 显示队列");
            System.out.println("e(exit): 退出程序");
            System.out.println("a(add): 添加数据到队列");
            System.out.println("g(get): 从队列取出数据");
            System.out.println("h(head): 查看队列头的数据");
            key = scanner.next().charAt(0);//接收一个字符
            switch (key) {
                case 's':
                    queue.showQueue();
                    break;
                case 'a':
                    System.out.println("输出一个数");
                    int value = scanner.nextInt();
                    queue.add(value);
                    break;
                case 'g':
                    if(queue.isEmpty()) {
                        System.out.println("队列为空");
                    } else {
                        System.out.println("队列头的数据为: " + queue.get());
                    }
                    break;
                case 'h':
                    if(queue.isEmpty()) {
                        System.out.println("队列为空");
                    } else {
                        System.out.println("队列头的数据为: " + queue.getHead());
                    }
                    break;
                case 'e':
                    System.out.println("感谢使用");
                    loop = false;
                    break;
                default:
                    System.out.println("输入错误");
            }
        }
    }
}
```



```
queue.addQueue(value);

break;

case 'g': //取出数据

try {

    int res = queue.getQueue();

    System.out.printf("取出的数据是%d\n", res);

} catch (Exception e) {

    // TODO: handle exception

    System.out.println(e.getMessage());

}

break;

case 'h': //查看队列头的数据

try {

    int res = queue.headQueue();

    System.out.printf("队列头的数据是%d\n", res);

} catch (Exception e) {

    // TODO: handle exception

    System.out.println(e.getMessage());

}

break;

case 'e': //退出

scanner.close();

loop = false;

break;

default:

break;
```



```
    }

}

System.out.println("程序退出~~");

}

}

// 使用数组模拟队列-编写一个 ArrayQueue 类
class ArrayQueue {

    private int maxSize; // 表示数组的最大容量
    private int front; // 队列头
    private int rear; // 队列尾
    private int[] arr; // 该数据用于存放数据, 模拟队列

    // 创建队列的构造器
    public ArrayQueue(int arrMaxSize) {
        maxSize = arrMaxSize;
        arr = new int[maxSize];
        front = -1; // 指向队列头部, 分析出 front 是指向队列头的前一个位置.
        rear = -1; // 指向队列尾, 指向队列尾的数据(即就是队列最后一个数据)
    }

    // 判断队列是否满
    public boolean isFull() {
        return rear == maxSize - 1;
    }
}
```



```
}
```

```
// 判断队列是否为空
```

```
public boolean isEmpty() {
```

```
    return rear == front;
```

```
}
```

```
// 添加数据到队列
```

```
public void addQueue(int n) {
```

```
    // 判断队列是否满
```

```
    if (isFull()) {
```

```
        System.out.println("队列满， 不能加入数据~");
```

```
        return;
```

```
}
```

```
    rear++; // 让 rear 后移
```

```
    arr[rear] = n;
```

```
}
```

```
// 获取队列的数据，出队列
```

```
public int getQueue() {
```

```
    // 判断队列是否空
```

```
    if (isEmpty()) {
```

```
        // 通过抛出异常
```

```
        throw new RuntimeException("队列空， 不能取数据");
```

```
}
```

```
    front++; // front 后移
```



```
return arr[front];\n\n}\n\n// 显示队列的所有数据\npublic void showQueue() {\n    // 遍历\n    if (isEmpty()) {\n        System.out.println("队列空的， 没有数据~~");\n        return;\n    }\n    for (int i = 0; i < arr.length; i++) {\n        System.out.printf("arr[%d]=%d\n", i, arr[i]);\n    }\n}\n\n// 显示队列的头数据， 注意不是取出数据\npublic int headQueue() {\n    // 判断\n    if (isEmpty()) {\n        throw new RuntimeException("队列空的， 没有数据~~");\n    }\n    return arr[front + 1];\n}
```

➤ 问题分析并优化

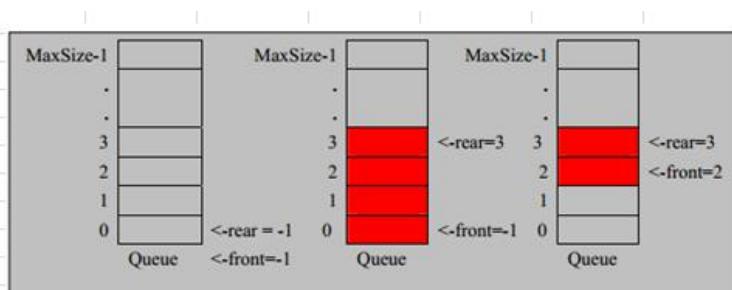
- 1) 目前数组使用一次就不能用， 没有达到复用的效果
- 2) 将这个数组使用算法， 改进成一个环形的队列 取模： %

3.2.4 数组模拟环形队列

对前面的数组模拟队列的优化，充分利用数组。因此将数组看做是一个环形的。(通过取模的方式来实现即可)

➤ 分析说明：

- 1) 尾索引的下一个为头索引时表示队列满，即将队列容量空出一个作为约定,这个在做判断队列满的时候需要注意 $(rear + 1) \% maxSize == front$ 【满】
- 2) $rear == front$ 【空】
- 3) 分析示意图：



思路如下：

1. `front` 变量的含义做一个调整： `front` 就指向队列的第一个元素, 也就是说 `arr[front]` 就是队列的第一个元素
`front` 的初始值 = 0
2. `rear` 变量的含义做一个调整： `rear` 指向队列的最后一个元素的后一个位置. 因为希望空出一个空间做为约定.
`rear` 的初始值 = 0
3. 当队列满时，条件是 $(rear + 1) \% maxSize = front$ 【满】
4. 对队列为空的条件， $rear == front$ 空
5. 当我们这样分析，队列中有效的数据的个数 $(rear + maxSize - front) \% maxSize // rear = 1 front = 0$
6. 我们就可以在原来的队列上修改得到，一个环形队列

➤ 代码实现

```
package com.atguigu.queue;
```



```
import java.util.Scanner;

public class CircleArrayQueueDemo {

    public static void main(String[] args) {

        //测试一把
        System.out.println("测试数组模拟环形队列的案例~~~");

        // 创建一个环形队列
        CircleArray queue = new CircleArray(4); //说明设置 4, 其队列的有效数据最大是 3
        char key = ' '; // 接收用户输入
        Scanner scanner = new Scanner(System.in);//
        boolean loop = true;
        // 输出一个菜单
        while (loop) {
            System.out.println("s(show): 显示队列");
            System.out.println("e(exit): 退出程序");
            System.out.println("a(add): 添加数据到队列");
            System.out.println("g(get): 从队列取出数据");
            System.out.println("h(head): 查看队列头的数据");
            key = scanner.next().charAt(0);// 接收一个字符
            switch (key) {
                case 's':
                    queue.showQueue();
                    break;
            }
        }
    }
}
```



```
break;

case 'a':
    System.out.println("输出一个数");
    int value = scanner.nextInt();
    queue.addQueue(value);
    break;

case 'g': // 取出数据
try {
    int res = queue.getQueue();
    System.out.printf("取出的数据是%d\n", res);
} catch (Exception e) {
    // TODO: handle exception
    System.out.println(e.getMessage());
}
break;

case 'h': // 查看队列头的数据
try {
    int res = queue.headQueue();
    System.out.printf("队列头的数据是%d\n", res);
} catch (Exception e) {
    // TODO: handle exception
    System.out.println(e.getMessage());
}
break;

case 'e': // 退出
scanner.close();
```



```
loop = false;
break;
default:
break;
}
}

System.out.println("程序退出~~");
}

}

class CircleArray {
private int maxSize; // 表示数组的最大容量
//front 变量的含义做一个调整： front 就指向队列的第一个元素，也就是说 arr[front] 就是队列的第一个元素
//front 的初始值 = 0
private int front;
//rear 变量的含义做一个调整： rear 指向队列的最后一个元素的后一个位置. 因为希望空出一个空间做为约定.
//rear 的初始值 = 0
private int rear; // 队列尾
private int[] arr; // 该数据用于存放数据，模拟队列

public CircleArray(int arrMaxSize) {
maxSize = arrMaxSize;
arr = new int[maxSize];
}
```



```
// 判断队列是否满
public boolean isFull() {
    return (rear + 1) % maxSize == front;
}

// 判断队列是否为空
public boolean isEmpty() {
    return rear == front;
}

// 添加数据到队列
public void addQueue(int n) {
    // 判断队列是否满
    if (isFull()) {
        System.out.println("队列满，不能加入数据~");
        return;
    }
    // 直接将数据加入
    arr[rear] = n;
    // 将 rear 后移，这里必须考虑取模
    rear = (rear + 1) % maxSize;
}

// 获取队列的数据，出队列
public int getQueue() {
```



```
// 判断队列是否空
if (isEmpty()) {
    // 通过抛出异常
    throw new RuntimeException("队列空，不能取数据");
}

// 这里需要分析出 front 是指向队列的第一个元素
// 1. 先把 front 对应的值保留到一个临时变量
// 2. 将 front 后移，考虑取模
// 3. 将临时保存的变量返回
int value = arr[front];
front = (front + 1) % maxSize;
return value;

}

// 显示队列的所有数据
public void showQueue() {
    // 遍历
    if (isEmpty()) {
        System.out.println("队列空的，没有数据~~");
        return;
    }

    // 思路：从 front 开始遍历，遍历多少个元素
    // 动脑筋
    for (int i = front; i < front + size() ; i++) {
        System.out.printf("arr[%d]=%d\n", i % maxSize, arr[i % maxSize]);
    }
}
```



```
}

}

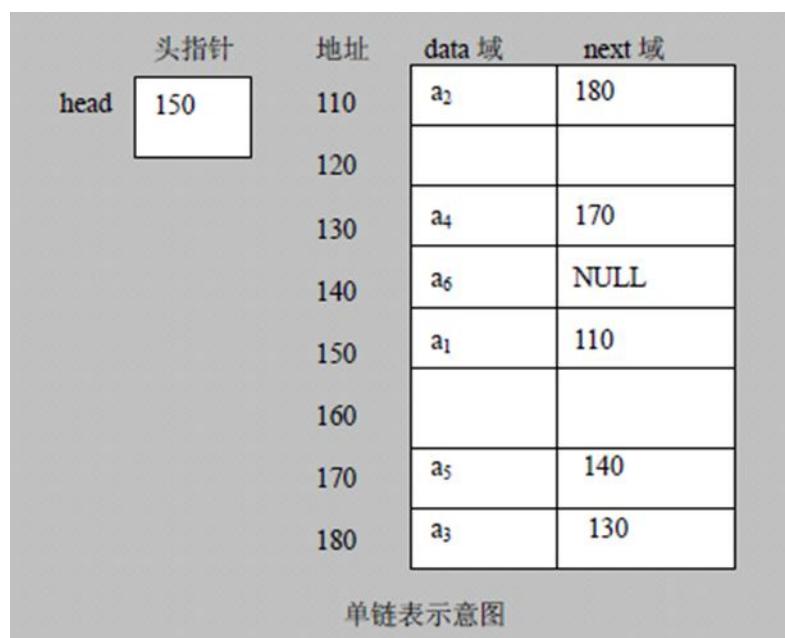
// 求出当前队列有效数据的个数
public int size() {
    // rear = 2
    // front = 1
    // maxSize = 3
    return (rear + maxSize - front) % maxSize;
}

// 显示队列的头数据， 注意不是取出数据
public int headQueue() {
    // 判断
    if (isEmpty()) {
        throw new RuntimeException("队列空的， 没有数据~~");
    }
    return arr[front];
}
}
```

第 4 章 链表

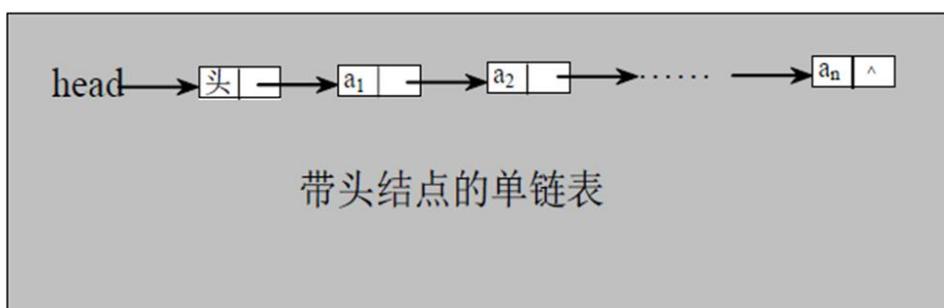
4.1 链表(Linked List)介绍

链表是有序的列表，但是它在内存中是存储如下



小结上图：

- 1) 链表是以节点的方式来存储,是链式存储
 - 2) 每个节点包含 data 域, next 域: 指向下一个节点.
 - 3) 如图: 发现链表的各个节点不一定是连续存储.
 - 4) 链表分带头节点的链表和没有头节点的链表, 根据实际的需求来确定
- 单链表(带头结点) 逻辑结构示意图如下



4.2 单链表的应用实例

使用带 head 头的单向链表实现 - 水浒英雄排行榜管理完成对英雄人物的增删改查操作，注：删除和修改，查找可以考虑学员独立完成，也可带学员完成

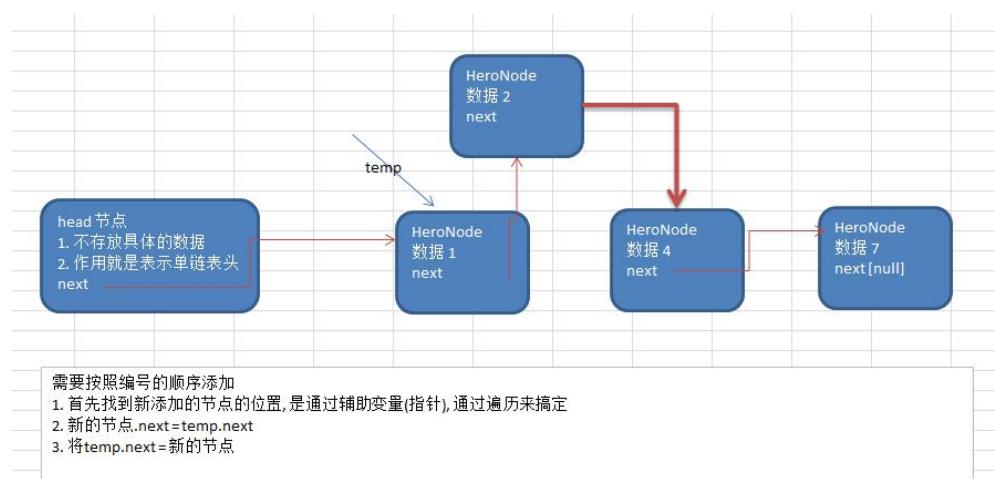
- 1) 第一种方法在添加英雄时，直接添加到链表的尾部

思路分析示意图：



- 2) 第二种方式在添加英雄时，根据排名将英雄插入到指定位置(如果有这个排名，则添加失败，并给出提示)

思路的分析示意图：

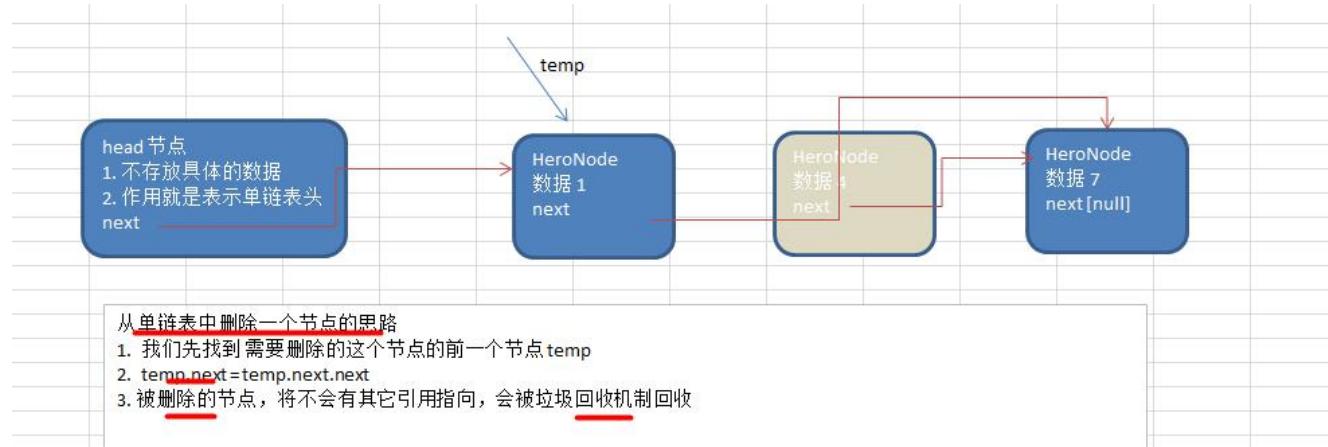


3) 修改节点功能

思路(1) 先找到该节点, 通过遍历, (2) temp.name = newHeroNode.name ; temp.nickname= newHeroNode.nickname

4) 删除节点

思路分析的示意图:



5) 完成的代码演示:

```
package com.atguigu.linkedlist;

public class SingleLinkedListDemo {

    public static void main(String[] args) {
        //进行测试
        //先创建节点
        HeroNode hero1 = new HeroNode(1, "宋江", "及时雨");
        HeroNode hero2 = new HeroNode(2, "卢俊义", "玉麒麟");
        HeroNode hero3 = new HeroNode(3, "吴用", "智多星");
        HeroNode hero4 = new HeroNode(4, "林冲", "豹子头");

        //创建要给链表
    }
}
```



```
SingleLinkedList singleLinkedList = new SingleLinkedList();

//加入
//    singleLinkedList.add(hero1);
//    singleLinkedList.add(hero4);
//    singleLinkedList.add(hero2);
//    singleLinkedList.add(hero3);

//加入按照编号的顺序
singleLinkedList.addByOrder(hero1);
singleLinkedList.addByOrder(hero4);
singleLinkedList.addByOrder(hero2);
singleLinkedList.addByOrder(hero3);

//显示一把
singleLinkedList.list();

//测试修改节点的代码
HeroNode newHeroNode = new HeroNode(2, "小卢", "玉麒麟~~");
singleLinkedList.update(newHeroNode);

System.out.println("修改后的链表情况~~");
singleLinkedList.list();

//删除一个节点
singleLinkedList.del(1);
singleLinkedList.del(4);
```



```
System.out.println("删除后的链表情况~~");
singleLinkedList.list();

}

//定义 SingleLinkedList 管理我们的英雄
class SingleLinkedList {
    //先初始化一个头节点，头节点不要动，不存放具体的数据
    private HeroNode head = new HeroNode(0, "", "");

    //添加节点到单向链表
    //思路，当不考虑编号顺序时
    //1. 找到当前链表的最后节点
    //2. 将最后这个节点的 next 指向 新的节点
    public void add(HeroNode heroNode) {

        //因为 head 节点不能动，因此我们需要一个辅助遍历 temp
        HeroNode temp = head;
        //遍历链表，找到最后
        while(true) {
            //找到链表的最后
            if(temp.next == null) {//
                break;
            }
            temp = temp.next;
        }
        temp.next = heroNode;
    }
}
```



```
        break;  
    }  
    //如果没有找到最后，将将 temp 后移  
    temp = temp.next;  
}  
  
//当退出 while 循环时，temp 就指向了链表的最后  
//将最后这个节点的 next 指向 新的节点  
temp.next = heroNode;  
}  
  
//第二种方式在添加英雄时，根据排名将英雄插入到指定位置  
//(如果有这个排名，则添加失败，并给出提示)  
public void addByOrder(HeroNode heroNode) {  
    //因为头节点不能动，因此我们仍然通过一个辅助指针(变量)来帮助找到添加的位置  
    //因为单链表，因为我们找的 temp 是位于 添加位置的前一个节点，否则插入不了  
    HeroNode temp = head;  
    boolean flag = false; // flag 标志添加的编号是否存在，默认为 false  
    while(true) {  
        if(temp.next == null) {//说明 temp 已经在链表的最后  
            break; //  
        }  
        if(temp.next.no > heroNode.no) { //位置找到，就在 temp 的后面插入  
            break;  
        } else if (temp.next.no == heroNode.no) {//说明希望添加的 heroNode 的编号已然存在  
            flag = true; //说明编号存在
```



```
        break;
    }
    temp = temp.next; //后移， 遍历当前链表
}
//判断 flag 的值
if(flag) { //不能添加， 说明编号存在
    System.out.printf("准备插入的英雄的编号 %d 已经存在了， 不能加入\n", heroNode.no);
} else {
    //插入到链表中, temp 的后面
    heroNode.next = temp.next;
    temp.next = heroNode;
}
}

//修改节点的信息， 根据 no 编号来修改， 即 no 编号不能改。
//说明
//1. 根据 newHeroNode 的 no 来修改即可
public void update(HeroNode newHeroNode) {
    //判断是否空
    if(head.next == null) {
        System.out.println("链表为空~");
        return;
    }
    //找到需要修改的节点， 根据 no 编号
    //定义一个辅助变量
    HeroNode temp = head.next;
```



```
boolean flag = false; //表示是否找到该节点
while(true) {
    if (temp == null) {
        break; //已经遍历完链表
    }
    if(temp.no == newHeroNode.no) {
        //找到
        flag = true;
        break;
    }
    temp = temp.next;
}
//根据 flag 判断是否找到要修改的节点
if(flag) {
    temp.name = newHeroNode.name;
    temp.nickname = newHeroNode.nickname;
} else { //没有找到
    System.out.printf("没有找到 编号 %d 的节点, 不能修改\n", newHeroNode.no);
}
}

//删除节点
//思路
//1. head 不能动, 因此我们需要一个 temp 辅助节点找到待删除节点的前一个节点
//2. 说明我们在比较时, 是 temp.next.no 和 需要删除的节点的 no 比较
public void del(int no) {
```



```
HeroNode temp = head;
boolean flag = false; // 标志是否找到待删除节点的
while(true) {
    if(temp.next == null) { //已经到链表的最后
        break;
    }
    if(temp.next.no == no) {
        //找到的待删除节点的前一个节点 temp
        flag = true;
        break;
    }
    temp = temp.next; //temp 后移， 遍历
}
//判断 flag
if(flag) { //找到
    //可以删除
    temp.next = temp.next.next;
} else {
    System.out.printf("要删除的 %d 节点不存在\n", no);
}
}

//显示链表[遍历]
public void list() {
    //判断链表是否为空
    if(head.next == null) {
```



```
System.out.println("链表为空");

return;
}

//因为头节点，不能动，因此我们需要一个辅助变量来遍历

HeroNode temp = head.next;

while(true) {

    //判断是否到链表最后

    if(temp == null) {

        break;

    }

    //输出节点的信息

    System.out.println(temp);

    //将 temp 后移，一定小心

    temp = temp.next;

}

}

}

//定义 HeroNode，每个 HeroNode 对象就是一个节点

class HeroNode {

    public int no;

    public String name;

    public String nickname;

    public HeroNode next; //指向下一个节点

    //构造器

    public HeroNode(int no, String name, String nickname) {
```



```
this.no = no;  
this.name = name;  
this.nickname = nickname;  
}  
  
//为了显示方法，我们重新 toString  
  
@Override  
public String toString() {  
    return "HeroNode [no=" + no + ", name=" + name + ", nickname=" + nickname + "]";  
}  
  
}
```

4.3 单链表面试题(新浪、百度、腾讯)

单链表的常见面试题有如下：

1) 求单链表中有效节点的个数

代码如下：

```
//方法：获取到单链表的节点的个数(如果是带头结点的链表，需求不统计头节点)
```

```
/**  
 *  
 * @param head 链表的头节点  
 * @return 返回的就是有效节点的个数  
 */  
  
public static int getLength(HeroNode head) {  
    if(head.next == null) { //空链表
```



```
return 0;

}

int length = 0;
//定义一个辅助的变量，这里我们没有统计头节点
HeroNode cur = head.next;
while(cur != null) {
    length++;
    cur = cur.next; //遍历
}
return length;
}
```

2) 查找单链表中的倒数第 k 个结点 【新浪面试题】

代码演示：

```
//查找单链表中的倒数第 k 个结点 【新浪面试题】

//思路
//1. 编写一个方法，接收 head 节点，同时接收一个 index
//2. index 表示是倒数第 index 个节点
//3. 先把链表从头到尾遍历，得到链表的总的长度 getLength
//4. 得到 size 后，我们从链表的第一个开始遍历 (size-index)个，就可以得到
//5. 如果找到了，则返回该节点，否则返回 null

public static HeroNode findLastIndexNode(HeroNode head, int index) {
    //判断如果链表为空，返回 null
    if(head.next == null) {
        return null;//没有找到
    }
}
```

```

//第一个遍历得到链表的长度(节点个数)

int size = getLength(head);

//第二次遍历 size-index 位置，就是我们倒数的第 K 个节点

//先做一个 index 的校验

if(index <=0 || index > size) {

    return null;

}

//定义给辅助变量， for 循环定位到倒数的 index

HeroNode cur = head.next; //3 // 3 - 1 = 2

for(int i =0; i< size - index; i++) {

    cur = cur.next;

}

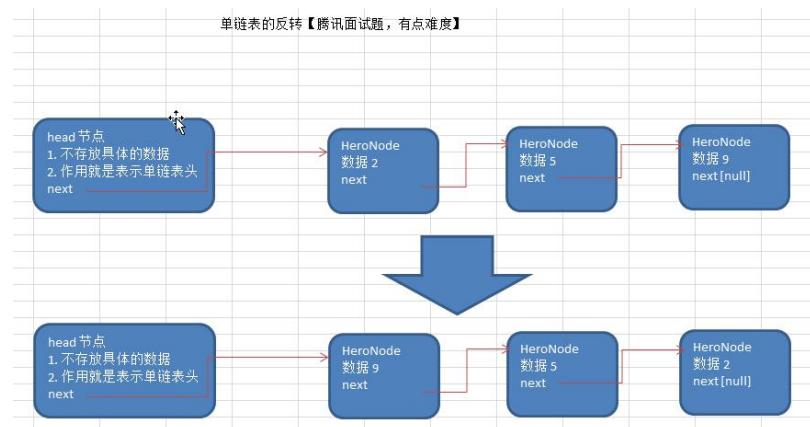
return cur;

}

```

3) 单链表的反转【腾讯面试题，有点难度】

➤ 思路分析图解





➤ 代码实现

```
//将单链表反转

public static void reverselist(HeroNode head) {
    //如果当前链表为空，或者只有一个节点，无需反转，直接返回
    if(head.next == null || head.next.next == null) {
        return ;
    }

    //定义一个辅助的指针(变量)，帮助我们遍历原来的链表
    HeroNode cur = head.next;
    HeroNode next = null; // 指向当前节点[cur]的下一个节点
    HeroNode reverseHead = new HeroNode(0, "", "");

    //遍历原来的链表，每遍历一个节点，就将其取出，并放在新的链表 reverseHead 的最前端
    //动脑筋
    while(cur != null) {
        next = cur.next; //先暂时保存当前节点的下一个节点，因为后面需要使用
        cur.next = reverseHead.next;
        reverseHead.next = cur;
        cur = next;
    }
}
```

```
cur.next = reverseHead.next; //将 cur 的下一个节点指向新的链表的最前端  
reverseHead.next = cur; //将 cur 连接到新的链表上  
cur = next; //让 cur 后移  
}  
  
//将 head.next 指向 reverseHead.next，实现单链表的反转  
head.next = reverseHead.next;  
}
```

4) 从尾到头打印单链表 【百度，要求方式 1：反向遍历。 方式 2：Stack 栈】

➤ 思路分析图解



➤ 代码实现

写了一个小程序，测试 Stack 的使用

```
package com.atguigu.linkedlist;  
  
import java.util.Stack;  
  
//演示栈 Stack 的基本使用  
public class TestStack {
```



```
public static void main(String[] args) {  
    Stack<String> stack = new Stack();  
    // 入栈  
    stack.add("jack");  
    stack.add("tom");  
    stack.add("smith");  
  
    // 出栈  
    // smith, tom , jack  
    while (stack.size() > 0) {  
        System.out.println(stack.pop());//pop 就是将栈顶的数据取出  
    }  
}  
}
```

单链表的逆序打印代码:

```
//方式 2:  
//可以利用栈这个数据结构，将各个节点压入到栈中，然后利用栈的先进后出的特点，就实现了逆序打印的效果  
public static void reversePrint(HeroNode head) {  
    if(head.next == null) {  
        return;//空链表，不能打印  
    }
```



```
//创建要给一个栈，将各个节点压入栈
Stack<HeroNode> stack = new Stack<HeroNode>();
HeroNode cur = head.next;
//将链表的所有节点压入栈
while(cur != null) {
    stack.push(cur);
    cur = cur.next; //cur 后移，这样就可以压入下一个节点
}
//将栈中的节点进行打印, pop 出栈
while (stack.size() > 0) {
    System.out.println(stack.pop()); //stack 的特点是先进后出
}
```

5) 合并两个有序的单链表，合并之后的链表依然有序【课后练习.】

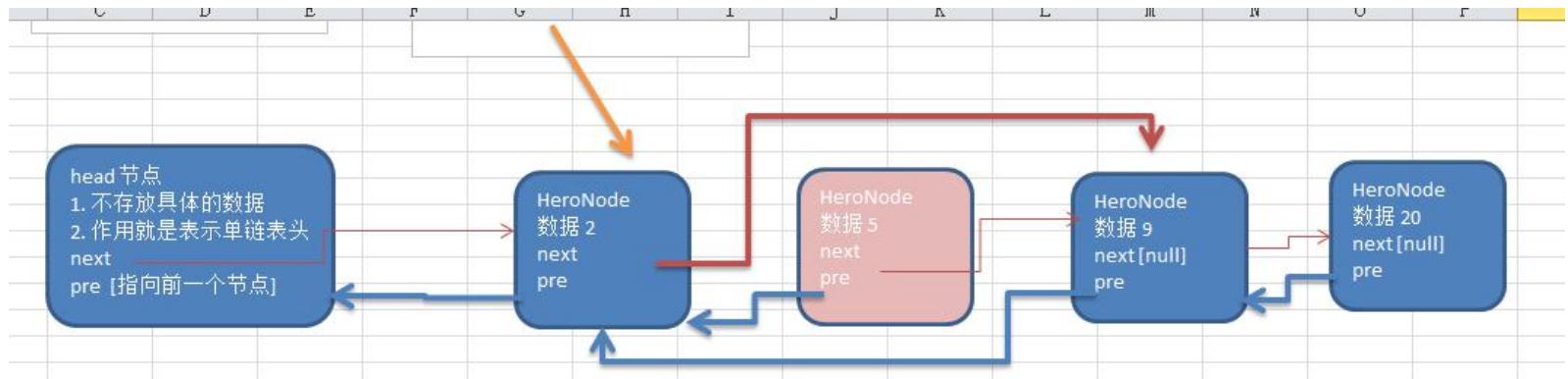
4.4 双向链表应用实例

4.4.1 双向链表的操作分析和实现

使用带 head 头的双向链表实现 - 水浒英雄排行榜

➤ 管理单向链表的缺点分析：

- 1) 单向链表，查找的方向只能是一个方向，而双向链表可以向前或者向后查找。
- 2) 单向链表不能自我删除，需要靠辅助节点，而双向链表，则可以自我删除，所以前面我们单链表删除时节点，总是找到 temp,temp 是待删除节点的前一个节点(认真体会).
- 3) 分析了双向链表如何完成遍历，添加，修改和删除的思路



对上图的说明：

分析 双向链表的遍历，添加，修改，删除的操作思路==> 代码实现

1) 遍历 方和 单链表一样，只是可以向前，也可以向后查找

2) 添加 (默认添加到双向链表的最后)

(1) 先找到双向链表的最后这个节点

(2) temp.next = newHeroNode

(3) newHeroNode.pre = temp;

3) 修改 思路和 原来的单向链表一样.

4) 删除

(1) 因为是双向链表，因此，我们可以实现自我删除某个节点

(2) 直接找到要删除的这个节点，比如 temp

(3) temp.pre.next = temp.next

(4) temp.next.pre = temp.pre;

➤ 双向链表的代码实现

```

package com.atguigu.linkedlist;

public class DoubleLinkedListDemo {

    public static void main(String[] args) {
  
```



```
// 测试
System.out.println("双向链表的测试");

// 先创建节点
HeroNode2 hero1 = new HeroNode2(1, "宋江", "及时雨");
HeroNode2 hero2 = new HeroNode2(2, "卢俊义", "玉麒麟");
HeroNode2 hero3 = new HeroNode2(3, "吴用", "智多星");
HeroNode2 hero4 = new HeroNode2(4, "林冲", "豹子头");

// 创建一个双向链表
DoubleLinkedList doubleLinkedList = new DoubleLinkedList();
doubleLinkedList.add(hero1);
doubleLinkedList.add(hero2);
doubleLinkedList.add(hero3);
doubleLinkedList.add(hero4);

doubleLinkedList.list();

// 修改
HeroNode2 newHeroNode = new HeroNode2(4, "公孙胜", "入云龙");
doubleLinkedList.update(newHeroNode);
System.out.println("修改后的链表情况");
doubleLinkedList.list();

// 删除
doubleLinkedList.del(3);
System.out.println("删除后的链表情况~~");
doubleLinkedList.list();
```



```
}

}

// 创建一个双向链表的类
class DoubleLinkedList {

    // 先初始化一个头节点, 头节点不要动, 不存放具体的数据
    private HeroNode2 head = new HeroNode2(0, "", "");

    // 返回头节点
    public HeroNode2 getHead() {
        return head;
    }

    // 遍历双向链表的方法
    // 显示链表[遍历]
    public void list() {
        // 判断链表是否为空
        if (head.next == null) {
            System.out.println("链表为空");
            return;
        }
    }
}
```



```
// 因为头节点，不能动，因此我们需要一个辅助变量来遍历

HeroNode2 temp = head.next;

while (true) {

    // 判断是否到链表最后

    if (temp == null) {

        break;

    }

    // 输出节点的信息

    System.out.println(temp);

    // 将 temp 后移，一定小心

    temp = temp.next;

}

}

// 添加一个节点到双向链表的最后.

public void add(HeroNode2 heroNode) {

    // 因为 head 节点不能动，因此我们需要一个辅助遍历 temp

    HeroNode2 temp = head;

    // 遍历链表，找到最后

    while (true) {

        // 找到链表的最后

        if (temp.next == null) {//

            break;

        }

        // 如果没有找到最后，将将 temp 后移
```



```
temp = temp.next;
}

// 当退出 while 循环时, temp 就指向了链表的最后
// 形成一个双向链表
temp.next = heroNode;
heroNode.pre = temp;
}

// 修改一个节点的内容, 可以看到双向链表的节点内容修改和单向链表一样
// 只是 节点类型改成 HeroNode2
public void update(HeroNode2 newHeroNode) {
    // 判断是否空
    if (head.next == null) {
        System.out.println("链表为空~");
        return;
    }
    // 找到需要修改的节点, 根据 no 编号
    // 定义一个辅助变量
    HeroNode2 temp = head.next;
    boolean flag = false; // 表示是否找到该节点
    while (true) {
        if (temp == null) {
            break; // 已经遍历完链表
        }
        if (temp.no == newHeroNode.no) {
            // 找到

```



```
flag = true;
break;
}
temp = temp.next;
}

// 根据 flag 判断是否找到要修改的节点
if (flag) {
    temp.name = newHeroNode.name;
    temp.nickname = newHeroNode.nickname;
} else { // 没有找到
    System.out.printf("没有找到 编号 %d 的节点, 不能修改\n", newHeroNode.no);
}
}

// 从双向链表中删除一个节点,
// 说明
// 1 对于双向链表, 我们可以直接找到要删除的这个节点
// 2 找到后, 自我删除即可
public void del(int no) {

    // 判断当前链表是否为空
    if (head.next == null) { // 空链表
        System.out.println("链表为空, 无法删除");
        return;
    }
}
```



```
HeroNode2 temp = head.next; // 辅助变量(指针)
boolean flag = false; // 标志是否找到待删除节点的
while (true) {
    if (temp == null) { // 已经到链表的最后
        break;
    }
    if (temp.no == no) {
        // 找到的待删除节点的前一个节点 temp
        flag = true;
        break;
    }
    temp = temp.next; // temp 后移, 遍历
}
// 判断 flag
if (flag) { // 找到
    // 可以删除
    // temp.next = temp.next.next;[单向链表]
    temp.pre.next = temp.next;
    // 这里我们的代码有问题?
    // 如果是最后一个节点, 就不需要执行下面这句话, 否则出现空指针
    if (temp.next != null) {
        temp.next.pre = temp.pre;
    }
} else {
    System.out.printf("要删除的 %d 节点不存在\n", no);
}
```



```
}

}

// 定义 HeroNode2 , 每个 HeroNode 对象就是一个节点
class HeroNode2 {

    public int no;
    public String name;
    public String nickname;
    public HeroNode2 next; // 指向下一个节点, 默认为 null
    public HeroNode2 pre; // 指向前一个节点, 默认为 null
    // 构造器

    public HeroNode2(int no, String name, String nickname) {
        this.no = no;
        this.name = name;
        this.nickname = nickname;
    }

    // 为了显示方法, 我们重新 toString
    @Override
    public String toString() {
        return "HeroNode [no=" + no + ", name=" + name + ", nickname=" + nickname + "]";
    }
}
```

4.4.2课堂作业和思路提示

双向链表的第二种添加方式,按照编号顺序 [示意图]按照单链表的顺序添加, 稍作修改即可.

4.5 单向环形链表应用场景

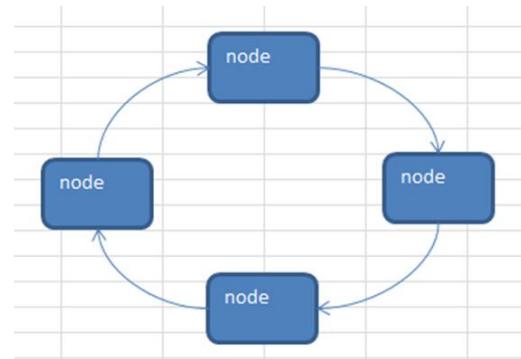
Josephu(约瑟夫、约瑟夫环) 问题

Josephu 问题为: 设编号为 1, 2, … n 的 n 个人围坐一圈, 约定编号为 k ($1 \leq k \leq n$) 的人从 1 开始报数, 数到 m 的那个人出列, 它的下一位又从 1 开始报数, 数到 m 的那个人又出列, 依次类推, 直到所有人出列为止, 由此产生一个出队编号的序列。

提示: 用一个不带头结点的循环链表来处理 Josephu 问题: 先构成一个有 n 个结点的单循环链表, 然后由 k 结点起从 1 开始计数, 计到 m 时, 对应结点从链表中删除, 然后再从被删除结点的下一个结点又从 1 开始计数, 直到最后一个结点从链表中删除算法结束。



4.6 单向环形链表介绍



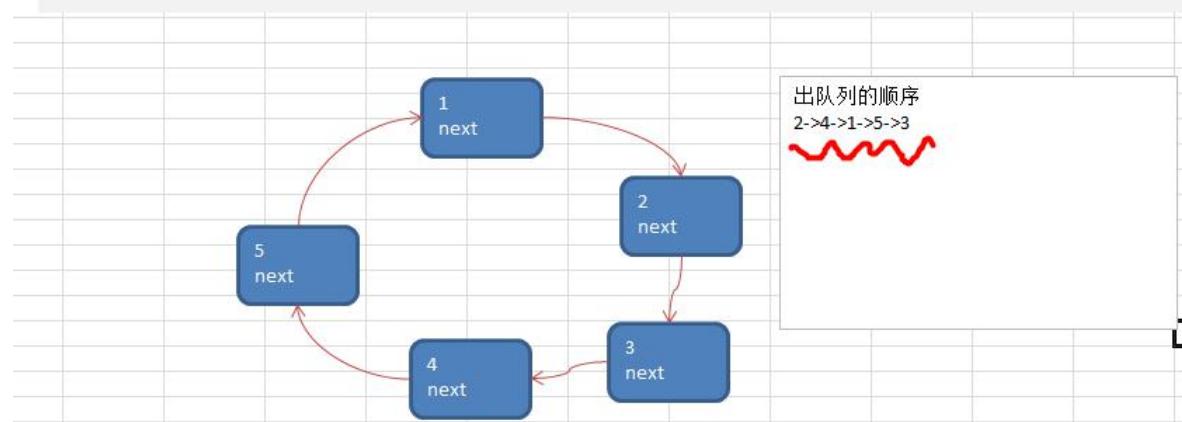
4.7 Josephu 问题

➤ 约瑟夫问题的示意图

Josephu 问题为：设编号为 $1, 2, \dots, n$ 的 n 个人围坐一圈，约定编号为 k ($1 \leq k \leq n$) 的人从 1 开始报数，数到 m 的那个人出列，它的下一位又从 1 开始报数，数到 m 的那个人又出列，依次类推，直到所有人出列为止，由此产生一个出队编号的序列。

$n = 5$, 即有 5 个人
 $k = 1$, 从第一个人开始报数
 $m = 2$, 数 2 下

单向环形链表完成，约瑟夫问题.



➤ Josephu 问题

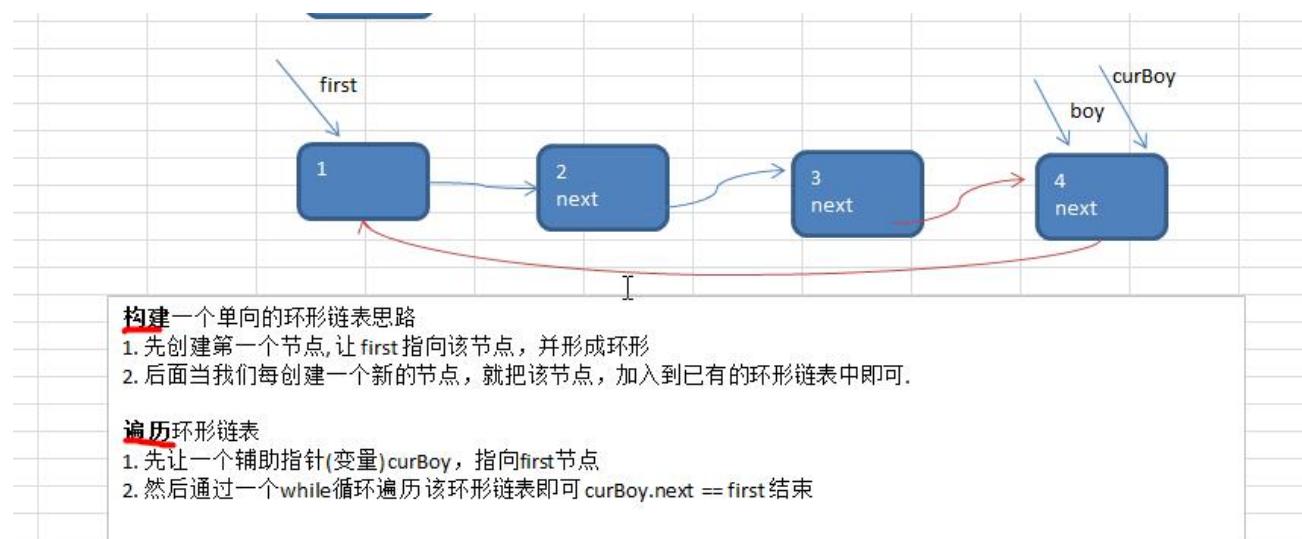
Josephu 问题为：设编号为 $1, 2, \dots, n$ 的 n 个人围坐一圈，约定编号为 k ($1 \leq k \leq n$) 的人从 1 开始报数，数到 m 的那个人出列，它的下一位又从 1 开始报数，数到 m 的那个人又出列，依次类推，直到所有人出列为止，由此产生一个出队编号的序列。

➤ 提示

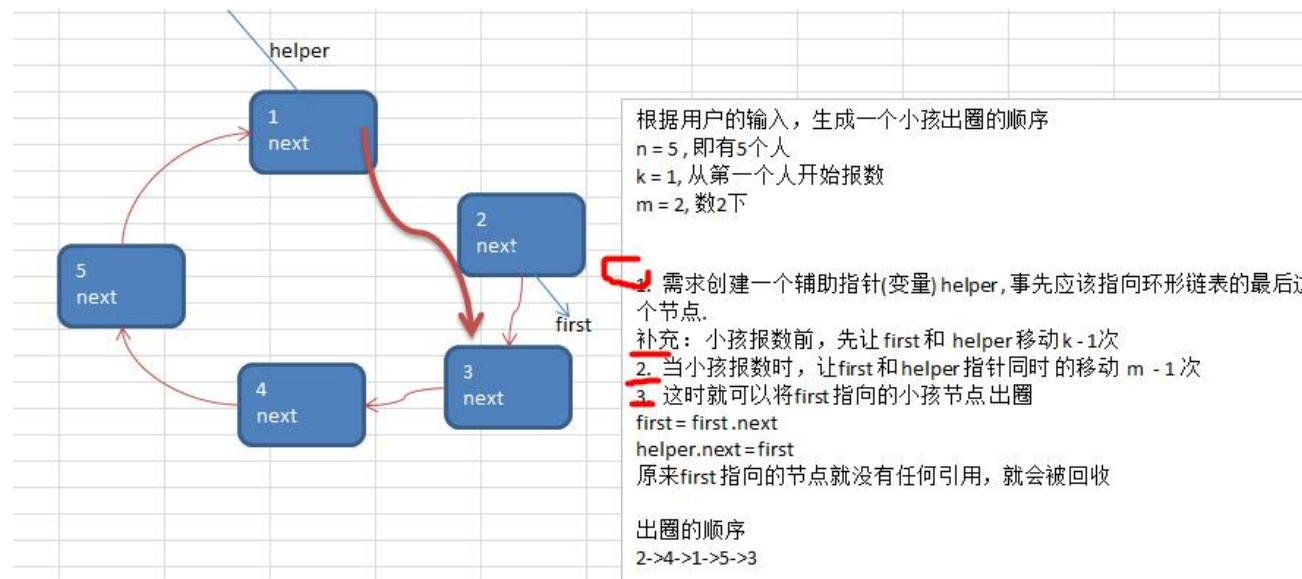
用一个不带头结点的循环链表来处理 Josephu 问题：先构成一个有 n 个结点的单循环链表，然后由 k 结点起从 1 开始计数，计到 m 时，对应结点从链表中删除，然后再从被删除结点的下一个结点又从 1 开始计数，直到最后一个

结点从链表中删除算法结束。

➤ 约瑟夫问题-创建环形链表的思路图解



➤ 约瑟夫问题-小孩出圈的思路分析图



4.8 Josephu 问题的代码实现

```
package com.atguigu.linkedlist;
```



```
public class Josepfu {  
  
    public static void main(String[] args) {  
        // 测试一把看看构建环形链表，和遍历是否 ok  
        CircleSingleLinkedList circleSingleLinkedList = new CircleSingleLinkedList();  
        circleSingleLinkedList.addBoy(125); // 加入 5 个小孩节点  
        circleSingleLinkedList.showBoy();  
  
        // 测试一把小孩出圈是否正确  
        circleSingleLinkedList.countBoy(10, 20, 125); // 2->4->1->5->3  
    }  
  
}  
  
// 创建一个环形的单向链表  
class CircleSingleLinkedList {  
    // 创建一个 first 节点, 当前没有编号  
    private Boy first = null;  
  
    // 添加小孩节点, 构建成一个环形的链表  
    public void addBoy(int nums) {  
        // nums 做一个数据校验  
        if (nums < 1) {  
            System.out.println("nums 的值不正确");  
            return;  
        }  
    }  
}
```



```
Boy curBoy = null; // 辅助指针，帮助构建环形链表
// 使用 for 来创建我们的环形链表
for (int i = 1; i <= nums; i++) {
    // 根据编号，创建小孩节点
    Boy boy = new Boy(i);
    // 如果是第一个小孩
    if (i == 1) {
        first = boy;
        first.setNext(first); // 构成环
        curBoy = first; // 让 curBoy 指向第一个小孩
    } else {
        curBoy.setNext(boy);//
        boy.setNext(first);//
        curBoy = boy;
    }
}

// 遍历当前的环形链表
public void showBoy() {
    // 判断链表是否为空
    if (first == null) {
        System.out.println("没有任何小孩~~");
        return;
    }
    // 因为 first 不能动，因此我们仍然使用一个辅助指针完成遍历
```



```
Boy curBoy = first;
while (true) {
    System.out.printf("小孩的编号 %d \n", curBoy.getNo());
    if (curBoy.getNext() == first) {// 说明已经遍历完毕
        break;
    }
    curBoy = curBoy.getNext(); // curBoy 后移
}
}

// 根据用户的输入，计算出小孩出圈的顺序
/**
 *
 * @param startNo
 *          表示从第几个小孩开始数数
 * @param countNum
 *          表示数几下
 * @param nums
 *          表示最初有多少小孩在圈中
 */
public void countBoy(int startNo, int countNum, int nums) {
    // 先对数据进行校验
    if (first == null || startNo < 1 || startNo > nums) {
        System.out.println("参数输入有误， 请重新输入");
        return;
    }
}
```



```
// 创建要给辅助指针,帮助完成小孩出圈

Boy helper = first;

// 需求创建一个辅助指针(变量) helper , 事先应该指向环形链表的最后这个节点

while (true) {

    if (helper.getNext() == first) { // 说明 helper 指向最后小孩节点

        break;

    }

    helper = helper.getNext();

}

//小孩报数前, 先让 first 和 helper 移动 k - 1 次

for(int j = 0; j < startNo - 1; j++) {

    first = first.getNext();

    helper = helper.getNext();

}

//当小孩报数时, 让 first 和 helper 指针同时 的移动 m - 1 次, 然后出圈

//这里是一个循环操作, 知道圈中只有一个节点

while(true) {

    if(helper == first) { //说明圈中只有一个节点

        break;

    }

    //让 first 和 helper 指针同时 的移动 countNum - 1

    for(int j = 0; j < countNum - 1; j++) {

        first = first.getNext();

        helper = helper.getNext();

    }

    //这时 first 指向的节点, 就是要出圈的小孩节点
```



```
System.out.printf("小孩%d 出圈\n", first.getNo());  
    //这时将 first 指向的小孩节点出圈  
    first = first.getNext();  
    helper.setNext(first); //  
  
}  
System.out.printf("最后留在圈中的小孩编号%d \n", first.getNo());  
  
}  
  
}  
  
// 创建一个 Boy 类，表示一个节点  
class Boy {  
    private int no;// 编号  
    private Boy next; // 指向下一个节点,默认 null  
  
    public Boy(int no) {  
        this.no = no;  
    }  
  
    public int getNo() {  
        return no;  
    }  
  
    public void setNo(int no) {  
        this.no = no;  
    }  
}
```



```
}
```

```
public Boy getNext() {  
    return next;  
}  
  
public void setNext(Boy next) {  
    this.next = next;  
}  
  
}
```

第 5 章 栈

5.1 栈的一个实际需求

请输入一个表达式

计算式:[$7*2*2-5+1-5+3-3$] 点击计算【如下图】

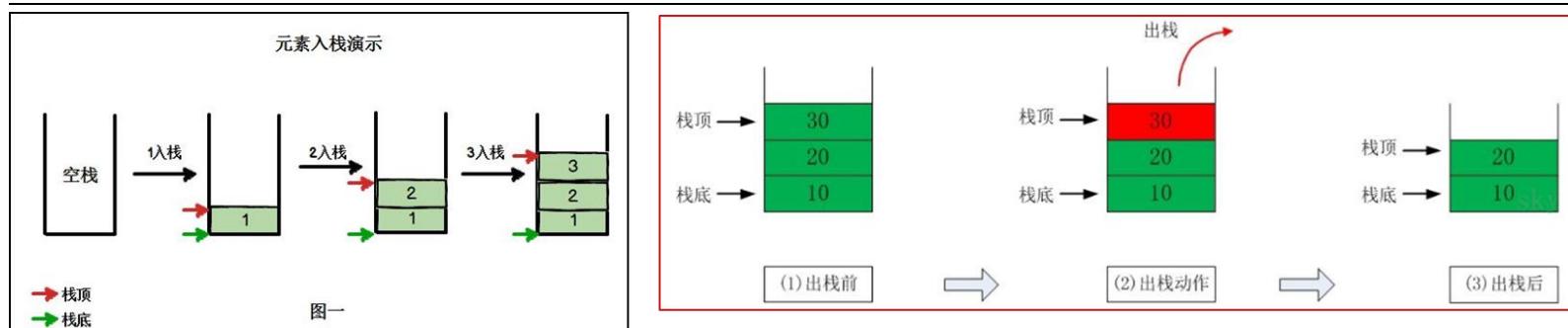
请输入一个表达式

计算式:

请问：计算机底层是如何运算得到结果的？ 注意不是简单的把算式列出运算,因为我们看这个算式 $7 * 2 * 2 - 5$, 但是计算机怎么理解这个算式的(对计算机而言, 它接收到的就是一个字符串), 我们讨论的是这个问题。-> 栈

5.2 栈的介绍

- 1) 栈的英文为(stack)
- 2) 栈是一个**先入后出(FILO-First In Last Out)**的有序列表。
- 3) 栈(stack)是限制线性表中元素的插入和删除只能在线性表的同一端进行的一种特殊线性表。允许插入和删除的一端, 为**变化的一端, 称为栈顶(Top)**, 另一端为**固定的一端, 称为栈底(Bottom)**。
- 4) 根据栈的定义可知, 最先放入栈中元素在栈底, 最后放入的元素在栈顶, 而删除元素刚好相反, 最后放入的元素最先删除, 最先放入的元素最后删除
- 5) 图解方式说明出栈(pop)和入栈(push)的概念

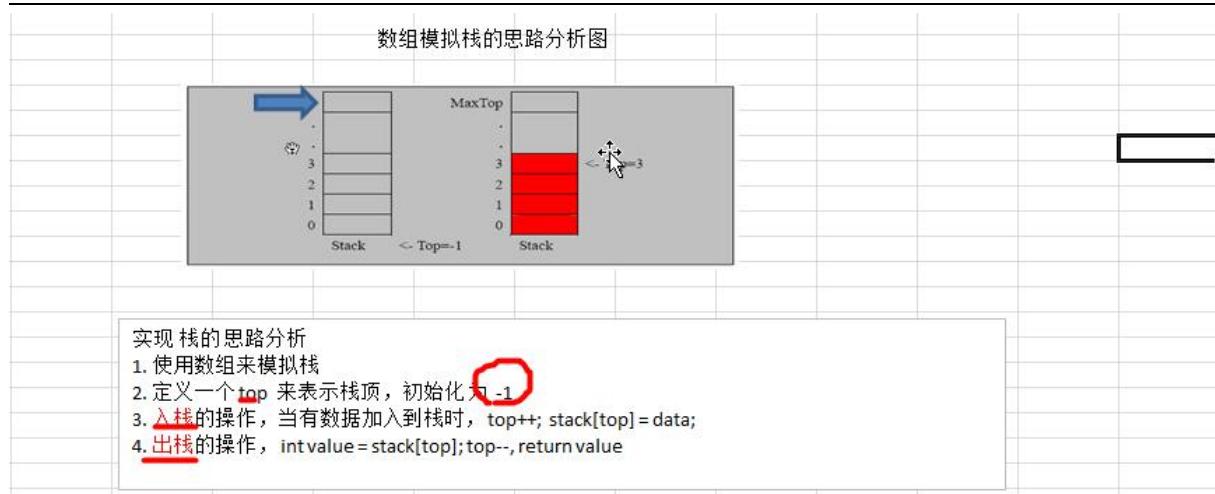


5.3 栈的应用场景

- 1) 子程序的调用：在跳往子程序前，会先将下个指令的地址存到堆栈中，直到子程序执行完后再将地址取出，以回到原来的程序中。
- 2) 处理递归调用：和子程序的调用类似，只是除了储存下一个指令的地址外，也将参数、区域变量等数据存入堆栈中。
- 3) 表达式的转换[中缀表达式转后缀表达式]与求值(实际解决)。
- 4) 二叉树的遍历。
- 5) 图形的深度优先(depth — first)搜索法。

5.4 栈的快速入门

- 1) 用数组模拟栈的使用，由于栈是一种有序列表，当然可以使用数组的结构来储存栈的数据内容，下面我们就用数组模拟栈的出栈，入栈等操作。
- 2) 实现思路分析，并画出示意图



3) 代码实现

```

package com.atguigu.stack;

import java.util.Scanner;

public class ArrayStackDemo {

    public static void main(String[] args) {
        //测试一下 ArrayStack 是否正确
        //先创建一个 ArrayStack 对象->表示栈
        ArrayStack stack = new ArrayStack(4);

        String key = "";
        boolean loop = true; //控制是否退出菜单
        Scanner scanner = new Scanner(System.in);

        while(loop) {
            System.out.println("show: 表示显示栈");
            System.out.println("exit: 退出程序");
    
```



```
System.out.println("push: 表示添加数据到栈(入栈)");
System.out.println("pop: 表示从栈取出数据(出栈)");
System.out.println("请输入你的选择");
key = scanner.nextInt();
switch (key) {
    case "show":
        stack.list();
        break;
    case "push":
        System.out.println("请输入一个数");
        int value = scanner.nextInt();
        stack.push(value);
        break;
    case "pop":
        try {
            int res = stack.pop();
            System.out.printf("出栈的数据是 %d\n", res);
        } catch (Exception e) {
            // TODO: handle exception
            System.out.println(e.getMessage());
        }
        break;
    case "exit":
        scanner.close();
        loop = false;
        break;
}
```



```
default:  
    break;  
}  
  
}  
  
System.out.println("程序退出~~~");  
}  
  
}  
  
//定义一个 ArrayStack 表示栈  
class ArrayStack {  
    private int maxSize; // 栈的大小  
    private int[] stack; // 数组，数组模拟栈，数据就放在该数组  
    private int top = -1;// top 表示栈顶，初始化为-1  
  
    //构造器  
    public ArrayStack(int maxSize) {  
        this.maxSize = maxSize;  
        stack = new int[this.maxSize];  
    }  
  
    //栈满  
    public boolean isFull() {  
        return top == maxSize - 1;  
    }
```



```
//栈空
public boolean isEmpty() {
    return top == -1;
}

//入栈-push
public void push(int value) {
    //先判断栈是否满
    if(isFull()) {
        System.out.println("栈满");
        return;
    }
    top++;
    stack[top] = value;
}

//出栈-pop, 将栈顶的数据返回
public int pop() {
    //先判断栈是否空
    if(isEmpty()) {
        //抛出异常
        throw new RuntimeException("栈空, 没有数据~");
    }
    int value = stack[top];
    top--;
    return value;
}

//显示栈的情况[遍历栈], 遍历时, 需要从栈顶开始显示数据
```



```
public void list() {  
    if(isEmpty()) {  
        System.out.println("栈空， 没有数据~~");  
        return;  
    }  
    //需要从栈顶开始显示数据  
    for(int i = top; i >= 0 ; i--) {  
        System.out.printf("stack[%d]=%d\n", i, stack[i]);  
    }  
}
```

4) 关于栈的一个小练习

课堂练习，将老师写的程序改成使用链表来模拟栈

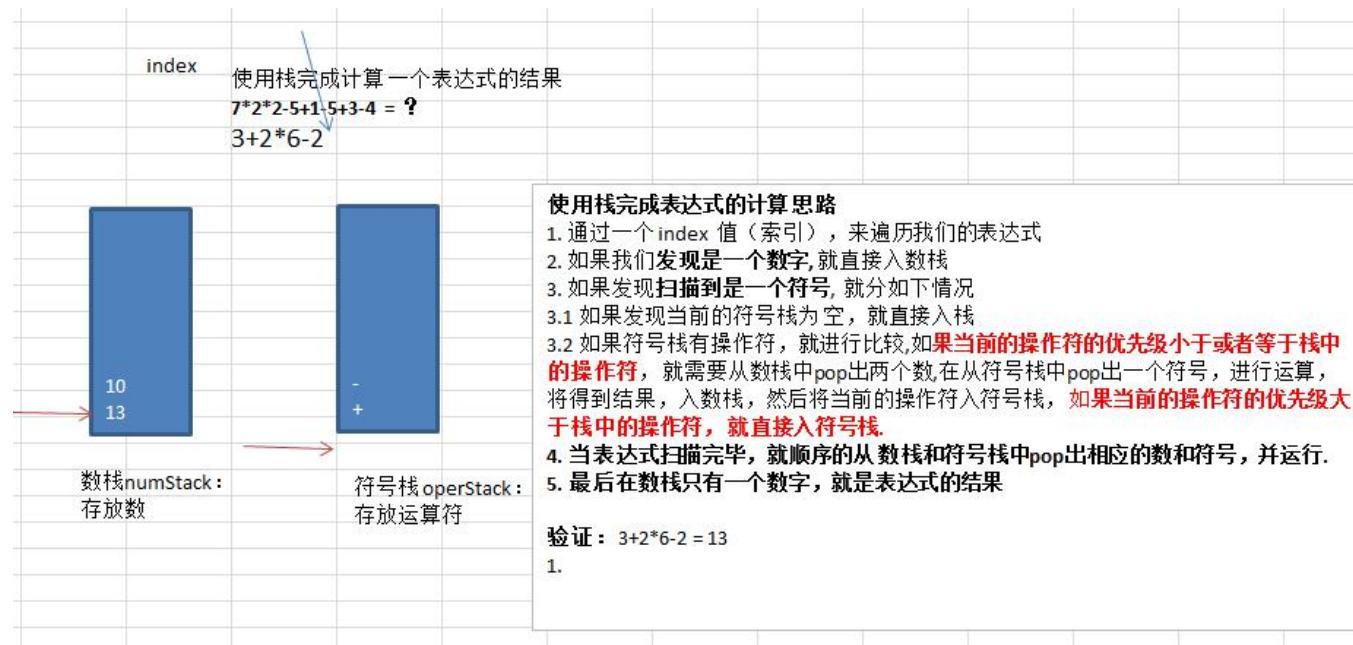
5.5 栈实现综合计算器(中缀表达式)

➤ 使用栈来实现综合计算器-

请输入一个表达式

计算式:[7*2*2-5+1-5+3-3] [点击计算](#)

➤ 思路分析(图解)



- 代价实现[1. 先实现一位数的运算， 2. 扩展到多位数的运算]

```
package com.atguigu.stack;
```

```
public class Calculator {
```

```
  public static void main(String[] args) {
```

```
    //根据前面老师思路，完成表达式的运算
```

```
    String expression = "7*2*2-5+1-5+3-4"; // 15//如何处理多位数的问题？
```

```
    //创建两个栈，数栈，一个符号栈
```

```
    ArrayStack2 numStack = new ArrayStack2(10);
```

```
    ArrayStack2 operStack = new ArrayStack2(10);
```

```
    //定义需要的相关变量
```

```
    int index = 0;//用于扫描
```

```
    int num1 = 0;
```

```
    int num2 = 0;
```



```
int oper = 0;
int res = 0;
char ch = '';//将每次扫描得到 char 保存到 ch
String keepNum = "";//用于拼接 多位数
//开始 while 循环的扫描 expression
while(true) {
    //依次得到 expression 的每一个字符
    ch = expression.substring(index, index+1).charAt(0);
    //判断 ch 是什么，然后做相应的处理
    if(operStack.isOper(ch)) {//如果是运算符
        //判断当前的符号栈是否为空
        if(!operStack.isEmpty()) {
            //如果符号栈有操作符，就进行比较,如果当前的操作符的优先级小于或者等于栈中的操作符,
           就需要从数栈中 pop 出两个数,
            //在从符号栈中 pop 出一个符号，进行运算，将得到结果，入数栈，然后将当前的操作符入符
            号栈
            if(operStack.priority(ch) <= operStack.priority(operStack.peek())) {
                num1 = numStack.pop();
                num2 = numStack.pop();
                oper = operStack.pop();
                res = numStack.cal(num1, num2, oper);
                //把运算的结果入数栈
                numStack.push(res);
                //然后将当前的操作符入符号栈
                operStack.push(ch);
            } else {
    
```



```
//如果当前的操作符的优先级大于栈中的操作符， 就直接入符号栈.  
operStack.push(ch);  
}  
}else {  
    //如果为空直接入符号栈..  
    operStack.push(ch); // 1 + 3  
}  
} else { //如果是数，则直接入数栈  
  
//numStack.push(ch - 48); //? "1+3" '1' => 1  
//分析思路  
//1. 当处理多位数时，不能发现是一个数就立即入栈，因为他可能是多位数  
//2. 在处理数，需要向 expression 的表达式的 index 后再看一位,如果是数就进行扫描，如果是符号  
才入栈  
//3. 因此我们需要定义一个变量 字符串，用于拼接  
  
//处理多位数  
keepNum += ch;  
  
//如果 ch 已经是 expression 的最后一位，就直接入栈  
if (index == expression.length() - 1) {  
    numStack.push(Integer.parseInt(keepNum));  
}else{  
  
    //判断下一个字符是不是数字，如果是数字，就继续扫描，如果是运算符，则入栈  
    //注意是看后一位，不是 index++
```



```
if (operStack.isOper(expression.substring(index+1,index+2).charAt(0))) {  
    //如果后一位是运算符，则入栈 keepNum = "1" 或者 "123"  
    numStack.push(Integer.parseInt(keepNum));  
    //重要的!!!!!, keepNum 清空  
    keepNum = "";  
  
}  
}  
}  
}  
//让 index + 1，并判断是否扫描到 expression 最后.  
index++;  
if (index >= expression.length()) {  
    break;  
}  
}  
  
}  
  
//当表达式扫描完毕，就顺序的从 数栈和符号栈中 pop 出相应的数和符号，并运行.  
while(true) {  
    //如果符号栈为空，则计算到最后的结果，数栈中只有一个数字【结果】  
    if(operStack.isEmpty()) {  
        break;  
    }  
    num1 = numStack.pop();  
    num2 = numStack.pop();  
    oper = operStack.pop();  
    res = numStack.cal(num1, num2, oper);
```



```
        numStack.push(res);//入栈  
    }  
    //将数栈的最后数, pop 出, 就是结果  
    int res2 = numStack.pop();  
    System.out.printf("表达式 %s = %d", expression, res2);  
}  
  
}  
  
//先创建一个栈,直接使用前面创建好  
//定义一个 ArrayStack2 表示栈, 需要扩展功能  
class ArrayStack2 {  
    private int maxSize; // 栈的大小  
    private int[] stack; // 数组, 数组模拟栈, 数据就放在该数组  
    private int top = -1;// top 表示栈顶, 初始化为-1  
  
    //构造器  
    public ArrayStack2(int maxSize) {  
        this.maxSize = maxSize;  
        stack = new int[this.maxSize];  
    }  
  
    //增加一个方法, 可以返回当前栈顶的值, 但是不是真正的 pop  
    public int peek() {  
        return stack[top];  
    }
```



```
//栈满
public boolean isFull() {
    return top == maxSize - 1;
}

//栈空
public boolean isEmpty() {
    return top == -1;
}

//入栈-push
public void push(int value) {
    //先判断栈是否满
    if(isFull()) {
        System.out.println("栈满");
        return;
    }
    top++;
    stack[top] = value;
}

//出栈-pop, 将栈顶的数据返回
public int pop() {
    //先判断栈是否空
    if(isEmpty()) {
        //抛出异常
        throw new RuntimeException("栈空, 没有数据~");
    }
}
```



```
int value = stack[top];
top--;
return value;
}

//显示栈的情况[遍历栈], 遍历时, 需要从栈顶开始显示数据

public void list() {
    if(isEmpty()) {
        System.out.println("栈空, 没有数据~~");
        return;
    }

    //需要从栈顶开始显示数据
    for(int i = top; i >= 0 ; i--) {
        System.out.printf("stack[%d]=%d\n", i, stack[i]);
    }
}

//返回运算符的优先级, 优先级是程序员来确定, 优先级使用数字表示
//数字越大, 则优先级就越高.

public int priority(int oper) {
    if(oper == '*' || oper == '/') {
        return 1;
    } else if (oper == '+' || oper == '-') {
        return 0;
    } else {
        return -1; // 假定目前的表达式只有 +, -, *, /
    }
}
```



```
//判断是不是一个运算符
public boolean isOper(char val) {
    return val == '+' || val == '-' || val == '*' || val == '/';
}

//计算方法
public int cal(int num1, int num2, int oper) {
    int res = 0; // res 用于存放计算的结果
    switch (oper) {
        case '+':
            res = num1 + num2;
            break;
        case '-':
            res = num2 - num1;// 注意顺序
            break;
        case '*':
            res = num1 * num2;
            break;
        case '/':
            res = num2 / num1;
            break;
        default:
            break;
    }
    return res;
}
```



{}

➤ 课后的练习-给表达式加入小括号

5.6 逆波兰计算器

我们完成一个逆波兰计算器，要求完成如下任务：

- 1) 输入一个逆波兰表达式(后缀表达式)，使用栈(Stack)，计算其结果
- 2) 支持小括号和多位数整数，因为这里我们主要讲的是数据结构，因此计算器进行简化，只支持对整数的计算。
- 3) 思路分析

例如: $(3+4) \times 5 - 6$ 对应的后缀表达式就是 $3\ 4\ +\ 5\ \times\ 6\ -$ ，针对后缀表达式求值步骤如下:

1. 从左至右扫描，将 3 和 4 压入堆栈；
2. 遇到+运算符，因此弹出 4 和 3（4 为栈顶元素，3 为次顶元素），计算出 $3+4$ 的值，得 7，再将 7 入栈；
3. 将 5 入栈；
4. 接下来是×运算符，因此弹出 5 和 7，计算出 $7 \times 5 = 35$ ，将 35 入栈；
5. 将 6 入栈；
6. 最后是-运算符，计算出 $35 - 6$ 的值，即 29，由此得出最终结果

4) 代码完成

```
package com.atguigu.stack;

import java.util.ArrayList;
import java.util.List;
import java.util.Stack;
```



```
public class PolandNotation {  
  
    public static void main(String[] args) {  
        //先定义给逆波兰表达式  
        // $(30+4) \times 5 - 6 \Rightarrow 30\ 4\ +\ 5\ \times\ 6\ - \Rightarrow 164$   
        // $4 * 5 - 8 + 60 + 8 / 2 \Rightarrow 4\ 5\ *\ 8\ -\ 60\ +\ 8\ 2\ / \Rightarrow 76$   
        //测试  
        //说明为了方便，逆波兰表达式 的数字和符号使用空格隔开  
        //String suffixExpression = "30 4 + 5 * 6 -";  
        String suffixExpression = "4 5 * 8 - 60 + 8 2 / +"; // 76  
        //思路  
        //1. 先将 "3 4 + 5 × 6 -" => 放到 ArrayList 中  
        //2. 将 ArrayList 传递给一个方法，遍历 ArrayList 配合栈 完成计算  
  
        List<String> list = getListString(suffixExpression);  
        System.out.println("rpnList=" + list);  
        int res = calculate(list);  
        System.out.println("计算的结果是=" + res);  
    }  
  
    //将一个逆波兰表达式， 依次将数据和运算符 放入到 ArrayList 中  
    public static List<String> getListString(String suffixExpression) {  
        //将 suffixExpression 分割  
        String[] split = suffixExpression.split(" ");  
        List<String> list = new ArrayList<String>();  
        ...  
    }  
}
```



```
for(String ele: split) {  
    list.add(ele);  
}  
return list;  
  
}  
  
//完成对逆波兰表达式的运算  
/*  
 * 1)从左至右扫描，将 3 和 4 压入堆栈；  
 * 2)遇到+运算符，因此弹出 4 和 3（4 为栈顶元素，3 为次顶元素），计算出 3+4 的值，得 7，再将 7 入栈；  
 * 3)将 5 入栈；  
 * 4)接下来是×运算符，因此弹出 5 和 7，计算出 7×5=35，将 35 入栈；  
 * 5)将 6 入栈；  
 * 6)最后是-运算符，计算出 35-6 的值，即 29，由此得出最终结果  
 */  
  
public static int calculate(List<String> ls) {  
    // 创建栈，只需要一个栈即可  
    Stack<String> stack = new Stack<String>();  
    // 遍历 ls  
    for (String item : ls) {  
        // 这里使用正则表达式来取出数  
        if (item.matches("\\d+")) { // 匹配的是多位数  
            // 入栈  
            stack.push(item);  
        }  
    }  
    // 栈中剩余的元素即为逆波兰表达式，从栈底到栈顶依次为 3 4 + 5 × 6 -  
    // 计算表达式  
    int result = 0;  
    while (!stack.isEmpty()) {  
        String item = stack.pop();  
        if (item.equals("+")) {  
            int num1 = stack.pop();  
            int num2 = stack.pop();  
            stack.push((int) (num1 + num2));  
        } else if (item.equals("-")) {  
            int num1 = stack.pop();  
            int num2 = stack.pop();  
            stack.push((int) (num2 - num1));  
        } else if (item.equals("*")) {  
            int num1 = stack.pop();  
            int num2 = stack.pop();  
            stack.push((int) (num1 * num2));  
        } else {  
            stack.push(Integer.parseInt(item));  
        }  
    }  
    return result;  
}
```



```
    } else {  
        // pop 出两个数，并运算，再入栈  
        int num2 = Integer.parseInt(stack.pop());  
        int num1 = Integer.parseInt(stack.pop());  
        int res = 0;  
        if (item.equals("+")) {  
            res = num1 + num2;  
        } else if (item.equals("-")) {  
            res = num1 - num2;  
        } else if (item.equals("*")) {  
            res = num1 * num2;  
        } else if (item.equals("/")) {  
            res = num1 / num2;  
        } else {  
            throw new RuntimeException("运算符有误");  
        }  
        // 把 res 入栈  
        stack.push("'" + res);  
    }  
}  
  
// 最后留在 stack 中的数据是运算结果  
return Integer.parseInt(stack.pop());  
}  
}
```



5.7 中缀表达式转换为后缀表达式

大家看到，后缀表达式适合计算式进行运算，但是人却不太容易写出来，尤其是表达式很长的情况下，因此在开发中，我们需要将 中缀表达式转成后缀表达式。

5.7.1 具体步骤如下：

- 1) 初始化两个栈：运算符栈 s1 和储存中间结果的栈 s2;
- 2) 从左至右扫描中缀表达式;
- 3) 遇到操作数时，将其压 s2;
- 4) 遇到运算符时，比较其与 s1 栈顶运算符的优先级：
 - 1.如果 s1 为空，或栈顶运算符为左括号 “(”，则直接将此运算符入栈;
 - 2.否则，若优先级比栈顶运算符的高，也将运算符压入 s1;
 - 3.否则，将 s1 栈顶的运算符弹出并压入到 s2 中，再次转到(4-1)与 s1 中新的栈顶运算符相比较;
- 5) 遇到括号时：
 - (1) 如果是左括号 “(”，则直接压入 s1
 - (2) 如果是右括号 “)” ，则依次弹出 s1 栈顶的运算符，并压入 s2，直到遇到左括号为止，此时将这一对括号丢弃
- 6) 重复步骤 2 至 5，直到表达式的最右边
- 7) 将 s1 中剩余的运算符依次弹出并压入 s2
- 8) 依次弹出 s2 中的元素并输出，结果的逆序即为中缀表达式对应的后缀表达式

5.7.2 举例说明：

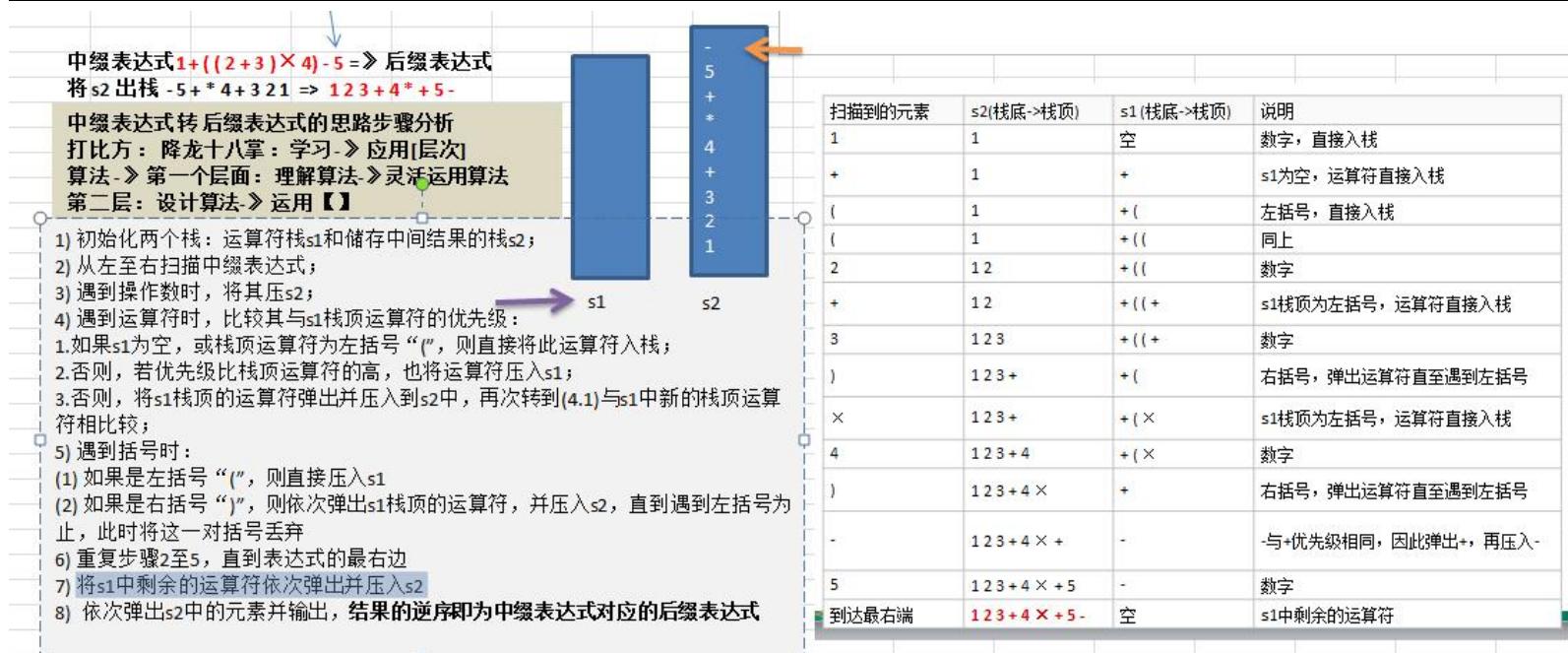
将中缀表达式 “ $1+((2+3)\times 4)-5$ ” 转换为后缀表达式的过程如下

因此结果为：“1 2 3 + 4 × + 5 - ”

扫描到的元素	s2(栈底->栈顶)	s1(栈底->栈顶)	说明
1	1	空	数字，直接入栈
+	1	+	s1为空，运算符直接入栈
(1	+()	左括号，直接入栈
(1	+((同上
2	1 2	+((数字
+	1 2	+((+	s1栈顶为左括号，运算符直接入栈
3	1 2 3	+((+	数字
)	1 2 3 +	+()	右括号，弹出运算符直至遇到左括号
*	1 2 3 +	+(*)	s1栈顶为左括号，运算符直接入栈
4	1 2 3 + 4	+(*)	数字
)	1 2 3 + 4 *	+	右括号，弹出运算符直至遇到左括号
-	1 2 3 + 4 * +	-	-与+优先级相同，因此弹出+，再压入-
5	1 2 3 + 4 * + 5	-	数字
到达最右端	1 2 3 + 4 * + 5 -	空	s1中剩余的运算符

5.7.3 代码实现中缀表达式转为后缀表达式

➤ 思路分析



➤ 代码实现

```

package com.atguigu.stack;

import java.util.ArrayList;
import java.util.List;
import java.util.Stack;

public class PolandNotation {

    public static void main(String[] args) {
        //完成将一个中缀表达式转成后缀表达式的功能
        //说明
        //1. 1+((2+3)*4)-5 => 转成 1 2 3 + 4 * + 5 - 
    }
}

```



```
//2. 因为直接对 str 进行操作，不方便，因此 先将 "1+((2+3)×4)-5" => 中缀的表达式对应的 List  
// 即 "1+((2+3)×4)-5" => ArrayList [1,+,(,(2,+3),*,4,),-,5]  
//3. 将得到的中缀表达式对应的 List => 后缀表达式对应的 List  
// 即 ArrayList [1,+,(,(2,+3),*,4,),-,5] => ArrayList [1,2,3,+4,*,+,5, - ]
```

```
String expression = "1+((2+3)*4)-5"; //注意表达式  
  
List<String> infixExpressionList = toInfixExpressionList(expression);  
System.out.println("中缀表达式对应的 List=" + infixExpressionList); // ArrayList [1,+,(,(2,+3),*,4,),-,5]  
  
List<String> suffixExpresionList = parseSuffixExpresionList(infixExpressionList);  
System.out.println("后缀表达式对应的 List" + suffixExpresionList); //ArrayList [1,2,3,+4,*,+,5, - ]
```

```
System.out.printf("expression=%d", calculate(suffixExpresionList)); // ?
```

```
/*  
  
//先定义给逆波兰表达式  
//(30+4)×5-6 => 30 4 + 5 × 6 - => 164  
// 4 * 5 - 8 + 60 + 8 / 2 => 4 5 * 8 - 60 + 8 2 / +  
//测试  
//说明为了方便，逆波兰表达式 的数字和符号使用空格隔开  
//String suffixExpression = "30 4 + 5 * 6 -";  
String suffixExpression = "4 5 * 8 - 60 + 8 2 / +"; // 76  
//思路  
//1. 先将 "3 4 + 5 × 6 - " => 放到 ArrayList 中
```



//2. 将 ArrayList 传递给一个方法，遍历 ArrayList 配合栈 完成计算

```
List<String> list = getListString(suffixExpression);
System.out.println("rpnList=" + list);
int res = calculate(list);
System.out.println("计算的结果是=" + res);

*/
```

```
//即 ArrayList [1,+,(,(2,+3,),*,4),-,5]  =》 ArrayList [1,2,3,+4,*,+,5, - ]
//方法：将得到的中缀表达式对应的 List => 后缀表达式对应的 List
public static List<String> parseSuffixExpresionList(List<String> ls) {
    //定义两个栈
    Stack<String> s1 = new Stack<String>(); // 符号栈
    //说明：因为 s2 这个栈，在整个转换过程中，没有 pop 操作，而且后面我们还需要逆序输出
    //因此比较麻烦，这里我们就不用 Stack<String> 直接使用 List<String> s2
    //Stack<String> s2 = new Stack<String>(); // 储存中间结果的栈 s2
    List<String> s2 = new ArrayList<String>(); // 储存中间结果的 Lists2

    //遍历 ls
    for(String item: ls) {
        //如果是一个数，加入 s2
        if(item.matches("\\d+")) {
```



```
s2.add(item);

} else if (item.equals("(")) {
    s1.push(item);
} else if (item.equals(")")) {
    //如果是右括号 “)”， 则依次弹出 s1 栈顶的运算符，并压入 s2，直到遇到左括号为止，此时将这一对括号丢弃
    while(!s1.peek().equals("(")) {
        s2.add(s1.pop());
    }
    s1.pop(); //!!! 将 ( 弹出 s1 栈， 消除小括号
} else {
    //当 item 的优先级小于等于 s1 栈顶运算符，将 s1 栈顶的运算符弹出并加入到 s2 中，再次转到(4.1)
    //与 s1 中新的栈顶运算符相比较
    //问题：我们缺少一个比较优先级高低的方法
    while(s1.size() != 0 && Operation.getValue(s1.peek()) >= Operation.getValue(item) ) {
        s2.add(s1.pop());
    }
    //还需要将 item 压入栈
    s1.push(item);
}

//将 s1 中剩余的运算符依次弹出并加入 s2
while(s1.size() != 0) {
    s2.add(s1.pop());
}
```



```
return s2; //注意因为是存放到 List, 因此按顺序输出就是对应的后缀表达式对应的 List

}

//方法: 将 中缀表达式转成对应的 List
// s="1+((2+3)×4)-5";
public static List<String> toInfixExpressionList(String s) {
    //定义一个 List,存放中缀表达式 对应的内容
    List<String> ls = new ArrayList<String>();
    int i = 0; //这时是一个指针, 用于遍历 中缀表达式字符串
    String str; // 对多位数的拼接
    char c; // 每遍历到一个字符, 就放入到 c
    do {
        //如果 c 是一个非数字, 我需要加入到 ls
        if((c=s.charAt(i)) < 48 || (c=s.charAt(i)) > 57) {
            ls.add(""+ c);
            i++; //i 需要后移
        } else { //如果是一个数, 需要考虑多位数
            str = ""; //先将 str 置成"" '0'[48]->'9'[57]
            while(i < s.length() && (c=s.charAt(i)) >= 48 && (c=s.charAt(i)) <= 57) {
                str += c;//拼接
                i++;
            }
            ls.add(str);
        }
    }
}
```



```
        }while(i < s.length());  
        return ls;//返回  
    }  
  
    //将一个逆波兰表达式， 依次将数据和运算符 放入到 ArrayList 中  
    public static List<String> getListString(String suffixExpression) {  
        //将 suffixExpression 分割  
        String[] split = suffixExpression.split(" ");  
        List<String> list = new ArrayList<String>();  
        for(String ele: split) {  
            list.add(ele);  
        }  
        return list;  
    }  
  
    //完成对逆波兰表达式的运算  
    /*  
     * 1)从左至右扫描，将 3 和 4 压入堆栈；  
     * 2)遇到+运算符，因此弹出 4 和 3 (4 为栈顶元素，3 为次顶元素)，计算出 3+4 的值，得 7，再将 7 入栈；  
     * 3)将 5 入栈；  
     * 4)接下来是×运算符，因此弹出 5 和 7，计算出 7×5=35，将 35 入栈；  
     * 5)将 6 入栈；  
     * 6)最后是-运算符，计算出 35-6 的值，即 29，由此得出最终结果  
     */
```



```
public static int calculate(List<String> ls) {  
    // 创建栈，只需要一个栈即可  
    Stack<String> stack = new Stack<String>();  
    // 遍历 ls  
    for (String item : ls) {  
        // 这里使用正则表达式来取出数  
        if (item.matches("\\d+")) { // 匹配的是多位数  
            // 入栈  
            stack.push(item);  
        } else {  
            // pop 出两个数，并运算，再入栈  
            int num2 = Integer.parseInt(stack.pop());  
            int num1 = Integer.parseInt(stack.pop());  
            int res = 0;  
            if (item.equals("+")) {  
                res = num1 + num2;  
            } else if (item.equals("-")) {  
                res = num1 - num2;  
            } else if (item.equals("*")) {  
                res = num1 * num2;  
            } else if (item.equals("/")) {  
                res = num1 / num2;  
            } else {  
                throw new RuntimeException("运算符有误");  
            }  
            // 把 res 入栈  
        }  
    }  
}
```



```
        stack.push("'" + res);

    }

}

//最后留在 stack 中的数据是运算结果
return Integer.parseInt(stack.pop());

}

}

//编写一个类 Operation 可以返回一个运算符 对应的优先级
class Operation {

    private static int ADD = 1;
    private static int SUB = 1;
    private static int MUL = 2;
    private static int DIV = 2;

    //写一个方法， 返回对应的优先级数字
    public static int getValue(String operation) {
        int result = 0;
        switch (operation) {
            case "+":
                result = ADD;
                break;
            case "-":
                result = SUB;
        }
    }
}
```



```
        break;

    case "*":
        result = MUL;
        break;

    case "/":
        result = DIV;
        break;

    default:
        System.out.println("不存在该运算符");
        break;
    }

    return result;
}

}
```

5.8 逆波兰计算器完整版

5.8.1 完整版的逆波兰计算器，功能包括

- 1) 支持 + - * / ()
- 2) 多位数，支持小数，
- 3) 兼容处理，过滤任何空白字符，包括空格、制表符、换页符

说明：逆波兰计算器完整版考虑的因素较多，下面给出完整版代码供同学们学习，其基本思路和前面一样，也是使用到：中缀表达式转后缀表达式。

代码实现：



```
package com.atguigu.reversepolishcal;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;
import java.util.Stack;
import java.util.regex.Pattern;

public class ReversePolishMultiCalc {

    /**
     * 匹配 + - * / ( ) 运算符
     */

    static final String SYMBOL = "\\+|-|\\*|/|\\(|\\)|\\|";
    static final String LEFT = "(";
    static final String RIGHT = ")";
    static final String ADD = "+";
    static final String MINUS = "-";
    static final String TIMES = "*";
    static final String DIVISION = "/";

    /**
     * 加减 + -
     */

    static final int LEVEL_01 = 1;
```



```
/*
 * 乘除 */
static final int LEVEL_02 = 2;

/*
 * 括号 */
static final int LEVEL_HIGH = Integer.MAX_VALUE;

static Stack<String> stack = new Stack<>();
static List<String> data = Collections.synchronizedList(new ArrayList<String>());

/*
 * 去除所有空白符
 * @param s
 * @return
 */
public static String replaceAllBlank(String s ){
    // \s+ 匹配任何空白字符，包括空格、制表符、换页符等等，等价于[ \f\n\r\t\v]
    return s.replaceAll("\s+", "");
}

/*
 * 判断是不是数字 int double long float
```



```
* @param s
* @return
*/
public static boolean isNumber(String s){
    Pattern pattern = Pattern.compile("^-\\+]?[.\\d]*$");
    return pattern.matcher(s).matches();
}

/**
* 判断是不是运算符
* @param s
* @return
*/
public static boolean isSymbol(String s){
    return s.matches(SYMBOL);
}

/**
* 匹配运算等级
* @param s
* @return
*/
public static int calcLevel(String s){
    if("+".equals(s) || "-".equals(s)){
        return LEVEL_01;
    } else if("*".equals(s) || "/".equals(s)){

```



```
        return LEVEL_02;
    }
    return LEVEL_HIGH;
}

/**
 * 匹配
 * @param s
 * @throws Exception
 */
public static List<String> doMatch (String s) throws Exception{
    if(s == null || "".equals(s.trim())) throw new RuntimeException("data is empty");
    if(!isNumber(s.charAt(0)+"")) throw new RuntimeException("data illeagle,start not with a number");

    s = replaceAllBlank(s);

    String each;
    int start = 0;

    for (int i = 0; i < s.length(); i++) {
        if(isSymbol(s.charAt(i)+"")){
            each = s.charAt(i)+"";
            //栈为空, (操作符, 或者 操作符优先级大于栈顶优先级 && 操作符优先级不是() 的优先级 及是 ) 不能直接入栈
            if(stack.isEmpty() || LEFT.equals(each)
                || ((calcLevel(each) > calcLevel(stack.peek())) && calcLevel(each) < LEVEL_HIGH)){
```



```
stack.push(each);

}else if( !stack.isEmpty() && calcLevel(each) <= calcLevel(stack.peek())){
    //栈非空，操作符优先级小于等于栈顶优先级时出栈入列，直到栈为空，或者遇到了(，最后
操作符入栈

    while (!stack.isEmpty() && calcLevel(each) <= calcLevel(stack.peek()) ){
        if(calcLevel(stack.peek()) == LEVEL_HIGH){
            break;
        }
        data.add(stack.pop());
    }

    stack.push(each);

}else if(RIGHT.equals(each)){
    // ) 操作符，依次出栈入列直到空栈或者遇到了第一个)操作符，此时)出栈
    while (!stack.isEmpty() && LEVEL_HIGH >= calcLevel(stack.peek())){
        if(LEVEL_HIGH == calcLevel(stack.peek())){
            stack.pop();
            break;
        }
        data.add(stack.pop());
    }

}

start = i ;      //前一个运算符的位置

}else if( i == s.length()-1 || isSymbol(s.charAt(i+1)+"") ){
    each = start == 0 ? s.substring(start,i+1) : s.substring(start+1,i+1);
    if(isNumber(each)) {
        data.add(each);
    }
}
```



```
        continue;

    }

    throw new RuntimeException("data not match number");

}

//如果栈里还有元素，此时元素需要依次出栈入列，可以想象栈里剩下栈顶为/，栈底为+，应该依次出栈
入列，可以直接翻转整个 stack 添加到队列

    Collections.reverse(stack);

    data.addAll(new ArrayList<>(stack));

    System.out.println(data);

    return data;

}

/***
 * 算出结果
 * @param list
 * @return
 */

public static Double doCalc(List<String> list){

    Double d = 0d;

    if(list == null || list.isEmpty()){

        return null;

    }

    if (list.size() == 1){

        System.out.println(list);

    }
```



```
d = Double.valueOf(list.get(0));

return d;

}

ArrayList<String> list1 = new ArrayList<>();

for (int i = 0; i < list.size(); i++) {

    list1.add(list.get(i));

    if(isSymbol(list.get(i))){

        Double d1 = doTheMath(list.get(i - 2), list.get(i - 1), list.get(i));

        list1.remove(i);

        list1.remove(i-1);

        list1.set(i-2,d1+"");

        list1.addAll(list.subList(i+1,list.size()));

        break;

    }

}

doCalc(list1);

return d;

}

/***
 * 运算
 * @param s1
 * @param s2
 * @param symbol
 * @return
 */
```

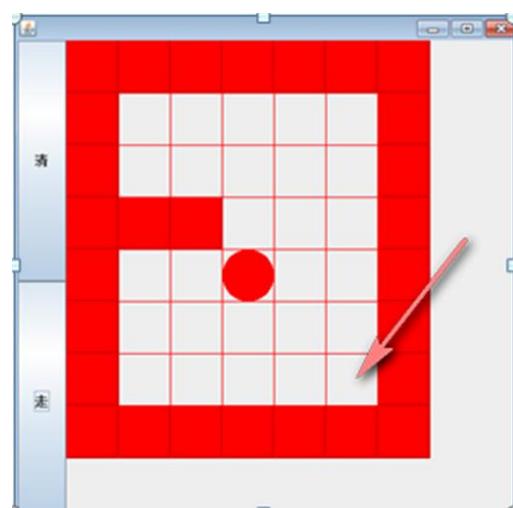


```
public static Double doTheMath(String s1, String s2, String symbol){  
    Double result ;  
    switch (symbol){  
        case ADD : result = Double.valueOf(s1) + Double.valueOf(s2); break;  
        case MINUS : result = Double.valueOf(s1) - Double.valueOf(s2); break;  
        case TIMES : result = Double.valueOf(s1) * Double.valueOf(s2); break;  
        case DIVISION : result = Double.valueOf(s1) / Double.valueOf(s2); break;  
        default : result = null;  
    }  
    return result;  
  
}  
  
public static void main(String[] args) {  
    //String math = "9+(3-1)*3+10/2";  
    String math = "12.8 + (2 - 3.55)*4+10/5.0";  
    try {  
        doCalc(doMatch(math));  
    } catch (Exception e) {  
        e.printStackTrace();  
    }  
}  
  
}
```

第 6 章 递归

6.1 递归应用场景

看个实际应用场景，迷宫问题(回溯)， 递归(Recursion)



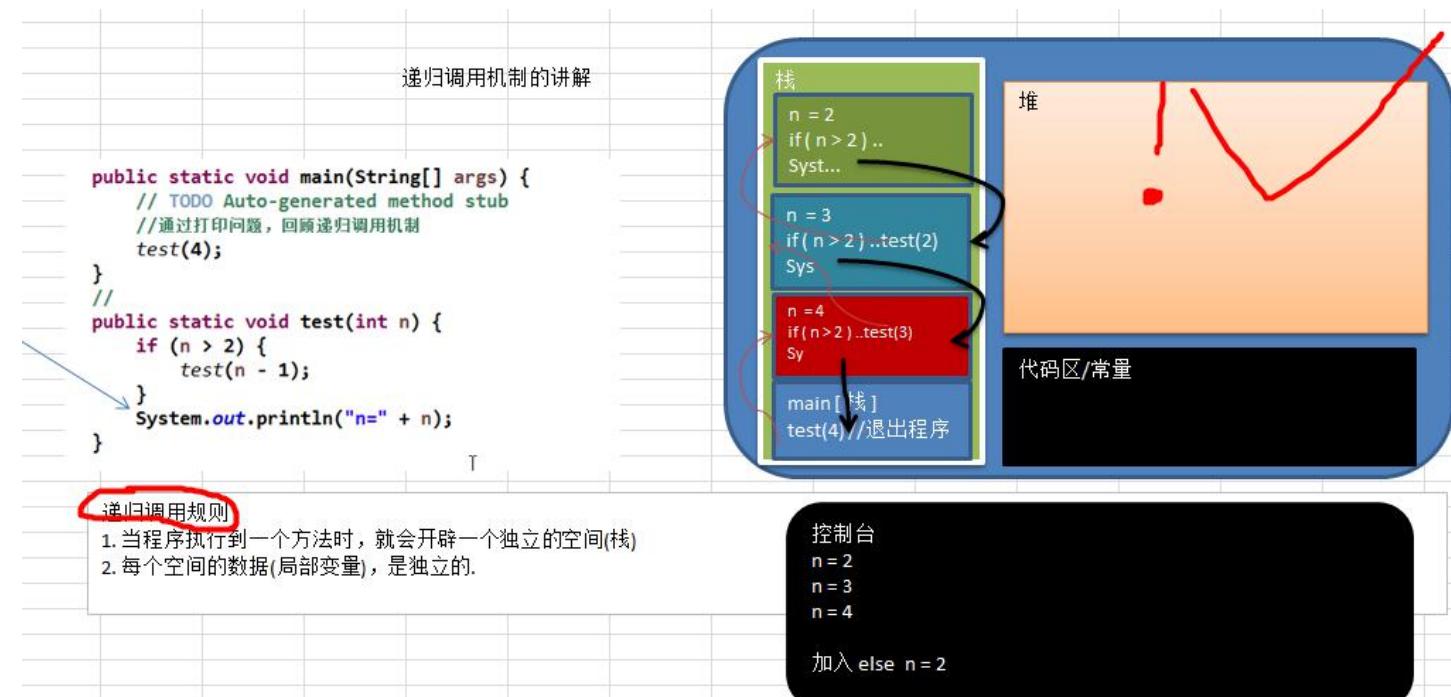
6.2 递归的概念

简单的说：递归就是方法自己调用自己,每次调用时传入不同的变量.递归有助于编程者解决复杂的问题,同时可以让代码变得简洁。

6.3 递归调用机制

我列举两个小案例,来帮助大家理解递归，部分学员已经学习过递归了，这里在给大家回顾一下递归调用机制

- 1) 打印问题
- 2) 阶乘问题
- 3) 使用图解方式说明了递归的调用机制



4) 代码演示

```

package com.atguigu.recursion;

public class RecursionTest {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        //通过打印问题，回顾递归调用机制
        //test(4);

        int res = factorial(3);
        System.out.println("res=" + res);
    }
    //打印问题.
}

```



```
public static void test(int n) {  
    if (n > 2) {  
        test(n - 1);  
    } //else {  
        System.out.println("n=" + n);  
    } // }  
}  
//阶乘问题  
public static int factorial(int n) {  
    if (n == 1) {  
        return 1;  
    } else {  
        return factorial(n - 1) * n; // 1 * 2 * 3  
    }  
}  
}
```

6.4 递归能解决什么样的问题

递归用于解决什么样的问题

- 1) 各种数学问题如: 8 皇后问题 , 汉诺塔, 阶乘问题, 迷宫问题, 球和篮子的问题(google 编程大赛)
- 2) 各种算法中也会使用到递归, 比如快排, 归并排序, 二分查找, 分治算法等.
- 3) 将用栈解决的问题-->第归代码比较简洁

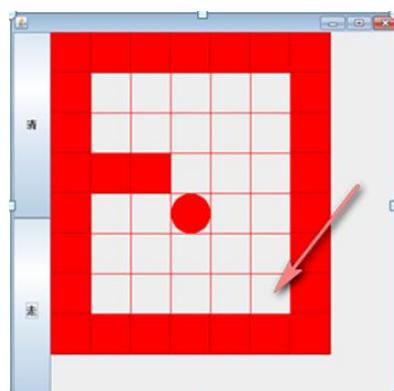
6.5 递归需要遵守的重要规则

递归需要遵守的重要规则

- 1) 执行一个方法时，就创建一个新的受保护的独立空间(栈空间)
- 2) 方法的局部变量是独立的，不会相互影响，比如 n 变量
- 3) 如果方法中使用的是引用类型变量(比如数组)，就会共享该引用类型的数据.
- 4) 递归必须向退出递归的条件逼近，否则就是无限递归,出现 StackOverflowError，死龟了:)
- 5) 当一个方法执行完毕，或者遇到 return，就会返回，遵守谁调用，就将结果返回给谁，同时当方法执行完毕或者返回时，该方法也就执行完毕

6.6 递归-迷宫问题

6.6.1 迷宫问题



6.6.2 代码实现:

```
package com.atguigu.recursion;

public class MiGong {

    public static void main(String[] args) {
```



```
// 先创建一个二维数组，模拟迷宫
// 地图
int[][] map = new int[8][7];
// 使用 1 表示墙
// 上下全部置为 1
for (int i = 0; i < 7; i++) {
    map[0][i] = 1;
    map[7][i] = 1;
}

// 左右全部置为 1
for (int i = 0; i < 8; i++) {
    map[i][0] = 1;
    map[i][6] = 1;
}

//设置挡板, 1 表示
map[3][1] = 1;
map[3][2] = 1;
// map[1][2] = 1;
// map[2][2] = 1;

// 输出地图
System.out.println("地图的情况");
for (int i = 0; i < 8; i++) {
    for (int j = 0; j < 7; j++) {
        System.out.print(map[i][j] + " ");
    }
}
```



```
}

System.out.println();

}

//使用递归回溯给小球找路

//setWay(map, 1, 1);
setWay2(map, 1, 1);

//输出新的地图，小球走过，并标识过的递归
System.out.println("小球走过，并标识过的 地图的情况");
for (int i = 0; i < 8; i++) {
    for (int j = 0; j < 7; j++) {
        System.out.print(map[i][j] + " ");
    }
    System.out.println();
}

}

//使用递归回溯来给小球找路
//说明
//1. map 表示地图
//2. i,j 表示从地图的哪个位置开始出发 (1,1)
//3. 如果小球能到 map[6][5] 位置，则说明通路找到。
//4. 约定：当 map[i][j] 为 0 表示该点没有走过 当为 1 表示墙； 2 表示通路可以走； 3 表示该点已经走过，但是走不通
```



//5. 在走迷宫时，需要确定一个策略(方法) 下->右->上->左，如果该点走不通，再回溯

```
/**  
 *  
 * @param map 表示地图  
 * @param i 从哪个位置开始找  
 * @param j  
 * @return 如果找到通路，就返回 true，否则返回 false  
 */  
  
public static boolean setWay(int[][] map, int i, int j) {  
    if(map[6][5] == 2) { // 通路已经找到 ok  
        return true;  
    } else {  
        if(map[i][j] == 0) { //如果当前这个点还没有走过  
            //按照策略 下->右->上->左 走  
            map[i][j] = 2; // 假定该点是可以走通.  
            if(setWay(map, i+1, j)) { //向下走  
                return true;  
            } else if (setWay(map, i, j+1)) { //向右走  
                return true;  
            } else if (setWay(map, i-1, j)) { //向上  
                return true;  
            } else if (setWay(map, i, j-1)){ // 向左走  
                return true;  
            } else {  
                //说明该点是走不通，是死路  
                map[i][j] = 3;  
            }  
        }  
    }  
}
```



```
        return false;  
    }  
} else { // 如果 map[i][j] != 0 , 可能是 1, 2, 3  
    return false;  
}  
}  
}  
}
```

//修改找路的策略，改成 上->右->下->左

```
public static boolean setWay2(int[][] map, int i, int j) {  
    if(map[6][5] == 2) { // 通路已经找到 ok  
        return true;  
    } else {  
        if(map[i][j] == 0) { //如果当前这个点还没有走过  
            //按照策略 上->右->下->左  
            map[i][j] = 2; // 假定该点是可以走通.  
            if(setWay2(map, i-1, j)) { //向上走  
                return true;  
            } else if (setWay2(map, i, j+1)) { //向右走  
                return true;  
            } else if (setWay2(map, i+1, j)) { //向下走  
                return true;  
            } else if (setWay2(map, i, j-1)){ // 向左走  
                return true;  
            } else {  
                //说明该点是走不通，是死路  
            }  
        }  
    }  
}
```



```
        map[i][j] = 3;  
        return false;  
    }  
    } else { // 如果 map[i][j] != 0 , 可能是 1, 2, 3  
        return false;  
    }  
}  
}  
}
```

6.6.3 对迷宫问题的讨论

- 1) 小球得到的路径，和程序员设置的**找路策略**有关即：找路的上下左右的顺序相关
- 2) 再得到小球路径时，可以先使用(下右上左)，再改成(上右下左)，看看路径是不是有变化
- 3) 测试回溯现象
- 4) 思考：如何求出最短路径？思路-》代码实现.

6.7 递归-八皇后问题(回溯算法)

6.7.1 八皇后问题介绍

八皇后问题，是一个古老而著名的问题，是回溯算法的典型案例。该问题是国际西洋棋棋手马克斯·贝瑟尔于1848年提出：在 8×8 格的国际象棋上摆放八个皇后，使其不能互相攻击，即：任意两个皇后都不能处于同一行、同一列或同一斜线上，问有多少种摆法(92)。



6.7.2 八皇后问题算法思路分析

- 1) 第一个皇后先放第一行第一列
- 2) 第二个皇后放在第二行第一列、然后判断是否 OK，如果不 OK，继续放在第二列、第三列、依次把所有列都放完，找到一个合适
- 3) 继续第三个皇后，还是第一列、第二列……直到第 8 个皇后也能放在一个不冲突的位置，算是找到了一个正确解
- 4) 当得到一个正确解时，在栈回退到上一个栈时，就会开始回溯，即将第一个皇后，放到第一列的所有正确解，全部得到。
- 5) 然后回头继续第一个皇后放第二列，后面继续循环执行 1,2,3,4 的步骤
- 6) 示意图：



- 1) 第一个皇后先放第一行第一列
- 2) 第二个皇后放在第二行第一列、然后判断是否OK，如果不OK，继续放在第二列、第三列、依次把所有列都放完，找到一个合适
- 3) 继续第三个皇后，还是第一列、第二列.....直到第8个皇后也能放在一个不冲突的位置，算是找到了一个正确解
- 4) 当得到一个正确解时，在找回退到上一个栈时，就会开始回溯，即将第一个皇后，放到第一列的所有正确解，全部得到。
- 5) 然后回头继续第一个皇后放第二列，后面继续循环执行1,2,3,4的步骤

➤ 说明：

理论上应该创建一个二维数组来表示棋盘，但是实际上可以通过算法，用一个一维数组即可解决问题. arr[8] = {0, 4, 7, 5, 2, 6, 1, 3} //对应 arr 下标 表示第几行，即第几个皇后，arr[i] = val , val 表示第 i+1 个皇后，放在第 i+1 行的第 val+1 列

6.7.3 八皇后问题算法代码实现

➤ 说明：看老师的代码演示

```
package com.atguigu.recursion;

public class Queue8 {

    //定义一个 max 表示共有多少个皇后
    int max = 8;
    //定义数组 array, 保存皇后放置位置的结果,比如 arr = {0, 4, 7, 5, 2, 6, 1, 3}
    int[] array = new int[max];
```



```
static int count = 0;

static int judgeCount = 0;

public static void main(String[] args) {
    //测试一把， 8 皇后是否正确
    Queue8 queue8 = new Queue8();
    queue8.check(0);
    System.out.printf("一共有%d 解法", count);
    System.out.printf("一共判断冲突的次数%d 次", judgeCount); // 1.5w

}

//编写一个方法，放置第 n 个皇后
//特别注意： check 是 每一次递归时，进入到 check 中都有 for(int i = 0; i < max; i++), 因此会有回溯
private void check(int n) {
    if(n == max) { //n = 8 , 其实 8 个皇后就既然放好
        print();
        return;
    }

    //依次放入皇后，并判断是否冲突
    for(int i = 0; i < max; i++) {
        //先把当前这个皇后 n, 放到该行的第 1 列
        array[n] = i;
        //判断当放置第 n 个皇后到 i 列时，是否冲突
    }
}
```



```
if(judge(n)) { // 不冲突
    //接着放 n+1 个皇后,即开始递归
    check(n+1); //
}

//如果冲突, 就继续执行 array[n] = i; 即将第 n 个皇后, 放置在本行得 后移的一个位置
}

}

//查看当我们放置第 n 个皇后, 就去检测该皇后是否和前面已经摆放的皇后冲突
/***
 *
 * @param n 表示第 n 个皇后
 * @return
 */
private boolean judge(int n) {
    judgeCount++;
    for(int i = 0; i < n; i++) {
        // 说明
        //1. array[i] == array[n] 表示判断 第 n 个皇后是否和前面的 n-1 个皇后在同一列
        //2. Math.abs(n-i) == Math.abs(array[n] - array[i]) 表示判断第 n 个皇后是否和第 i 皇后是否在同一斜
线
        // n = 1 放置第 2 列 1 n = 1 array[1] = 1
        // Math.abs(1-0) == 1 Math.abs(array[n] - array[i]) = Math.abs(1-0) = 1
        //3. 判断是否在同一行, 没有必要, n 每次都在递增
        if(array[i] == array[n] || Math.abs(n-i) == Math.abs(array[n] - array[i])) {
            return false;
        }
    }
}
```



```
    }  
}  
return true;  
}  
  
//写一个方法，可以将皇后摆放的位置输出  
private void print() {  
    count++;  
    for (int i = 0; i < array.length; i++) {  
        System.out.print(array[i] + " ");  
    }  
    System.out.println();  
}  
  
}
```

第 7 章 排序算法

7.1 排序算法的介绍

排序也称排序算法(Sort Algorithm)，排序是将一组数据，依指定的顺序进行排列的过程。

7.2 排序的分类：

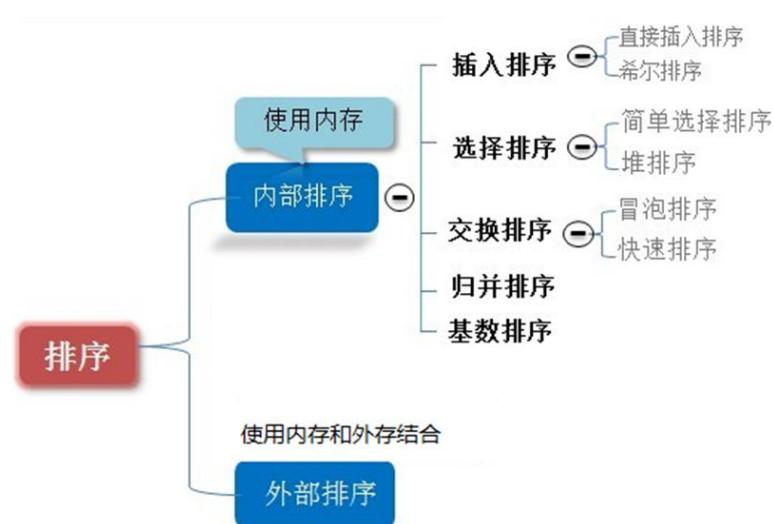
1) 内部排序：

指将需要处理的所有数据都加载到内部存储器(内存)中进行排序。

2) 外部排序法：

数据量过大，无法全部加载到内存中，需要借助外部存储(文件等)进行排序。

3) 常见的排序算法分类(见右图)：



7.3 算法的时间复杂度

7.3.1 度量一个程序(算法)执行时间的两种方法

1) 事后统计的方法

这种方法可行，但是有两个问题：一是要想对设计的算法的运行性能进行评测，需要实际运行该程序；二是所得时间的统计量依赖于计算机的硬件、软件等环境因素，这种方式，要在同一台计算机的相同状态下运行，才能比



较那个算法速度更快。

2) 事前估算的方法

通过分析某个算法的**时间复杂度**来判断哪个算法更优.

7.3.2 时间频度

➤ 基本介绍

时间频度：一个算法花费的时间与算法中语句的执行次数成正比例，哪个算法中语句执行次数多，它花费时间就多。一个算法中的语句执行次数称为语句频度或时间频度。记为 $T(n)$ 。[举例说明]

➤ 举例说明-基本案例

比如计算 1-100 所有数字之和，我们设计两种算法：

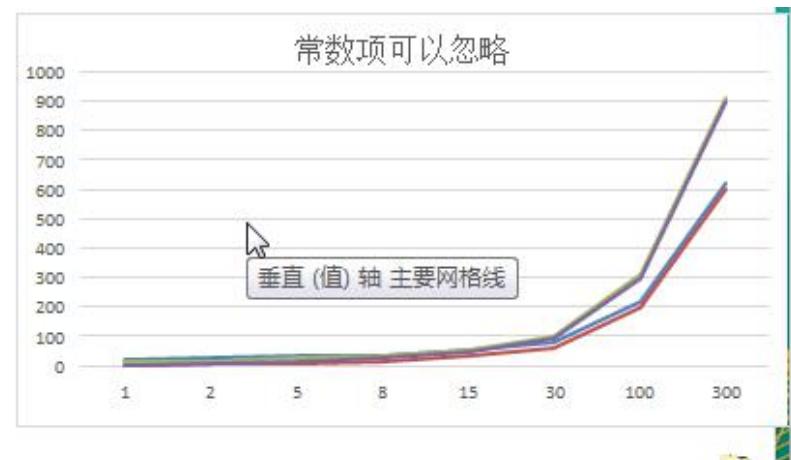
```
int total=0;
int end=100;
//使用for循环计算
for(int i=1;i<=end;i++){
    total+=i;
}
```

```
T(n)=n+1;
//直接计算
total=(1+end)*end/2;
```

$T(n)=1$

➤ 举例说明-忽略常数项

	$T(n)=2n+20$	$T(n)=2^n$	$T(3n+10)$	$T(3n)$
1	22	2	13	3
2	24	4	16	6
5	30	10	25	15
8	36	16	34	24
15	50	30	55	45
30	80	60	100	90
100	220	200	310	300
300	620	600	910	900



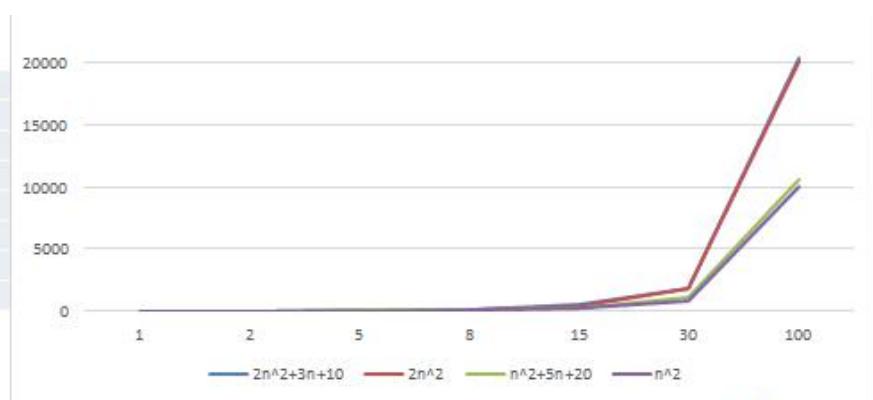
结论:

1) $2n+20$ 和 2^n 随着 n 变大, 执行曲线无限接近, 20 可以忽略

2) $3n+10$ 和 $3n$ 随着 n 变大, 执行曲线无限接近, 10 可以忽略

➤ 举例说明-忽略低次项

	$T(n)=2n^2+3n+10$	$T(2n^2)$	$T(n^2+5n+20)$	$T(n^2)$
1	15	2	26	1
2	24	8	34	4
5	75	50	70	25
8	162	128	124	64
15	505	450	320	225
30	1900	1800	1070	900
100	20310	20000	10520	10000

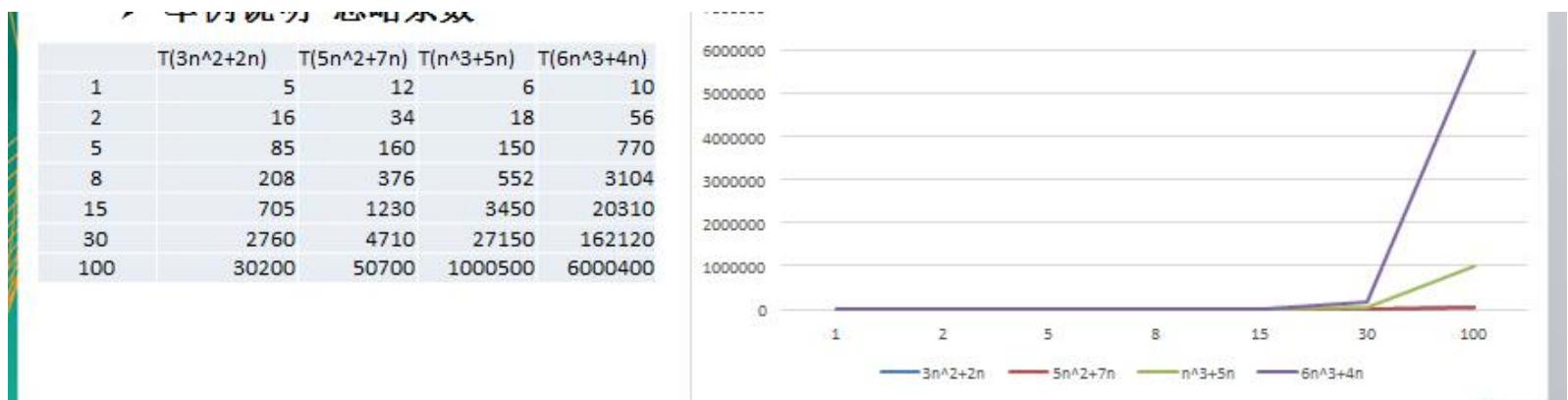


结论:

1) $2n^2+3n+10$ 和 $2n^2$ 随着 n 变大, 执行曲线无限接近, 可以忽略 $3n+10$

2) $n^2+5n+20$ 和 n^2 随着 n 变大, 执行曲线无限接近, 可以忽略 $5n+20$

➤ 举例说明-忽略系数



结论:

- 1) 随着 n 值变大, $5n^2+7n$ 和 $3n^2+2n$, 执行曲线重合, 说明这种情况下, 5 和 3 可以忽略。
- 2) 而 n^3+5n 和 $6n^3+4n$, 执行曲线分离, 说明多少次方式关键

7.3.3 时间复杂度

1) 一般情况下, 算法中的基本操作语句的重复执行次数是问题规模 n 的某个函数, 用 $T(n)$ 表示, 若有某个辅助函数 $f(n)$, 使得当 n 趋近于无穷大时, $T(n) / f(n)$ 的极限值为不等于零的常数, 则称 $f(n)$ 是 $T(n)$ 的同数量级函数。记作 $T(n)=O(f(n))$, 称 $O(f(n))$ 为算法的渐进时间复杂度, 简称时间复杂度。

2) $T(n)$ 不同, 但时间复杂度可能相同。如: $T(n)=n^2+7n+6$ 与 $T(n)=3n^2+2n+2$ 它们的 $T(n)$ 不同, 但时间复杂度相同, 都为 $O(n^2)$ 。

3) 计算时间复杂度的方法:

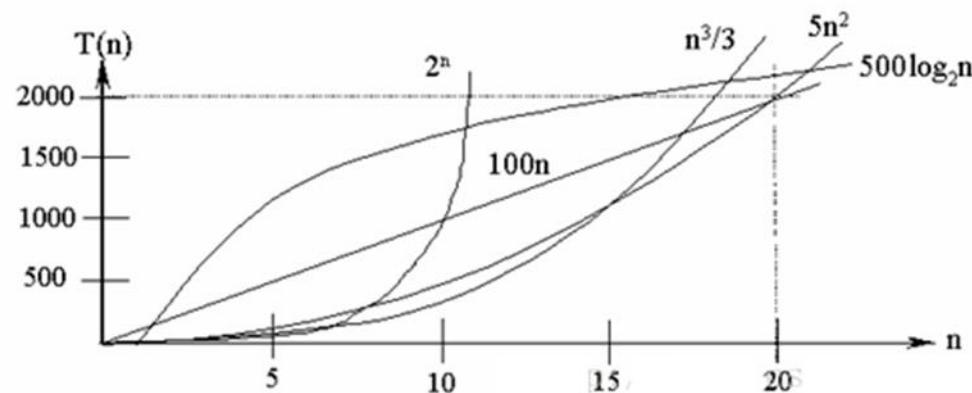
- 用常数 1 代替运行时间中的所有加法常数 $T(n)=n^2+7n+6 \Rightarrow T(n)=n^2+7n+1$
- 修改后的运行次数函数中, 只保留最高阶项 $T(n)=n^2+7n+1 \Rightarrow T(n)=n^2$
- 去除最高阶项的系数 $T(n)=n^2 \Rightarrow T(n)=n^2 \Rightarrow O(n^2)$

7.3.4 常见的时间复杂度

- 1) 常数阶 $O(1)$
- 2) 对数阶 $O(\log_2 n)$
- 3) 线性阶 $O(n)$
- 4) 线性对数阶 $O(n \log_2 n)$

- 5) 平方阶 $O(n^2)$
- 6) 立方阶 $O(n^3)$
- 7) k 次方阶 $O(n^k)$
- 8) 指数阶 $O(2^n)$

常见的时间复杂度对应的图:



说明:

- 1) 常见的算法时间复杂度由小到大依次为: $O(1) < O(\log_2 n) < O(n) < O(n \log_2 n) < O(n^2) < O(n^3) < O(n^k) < O(2^n)$, 随着问题规模 n 的不断增大, 上述时间复杂度不断增大, 算法的执行效率越低
- 2) 从图中可见, 我们应该尽可能避免使用指数阶的算法

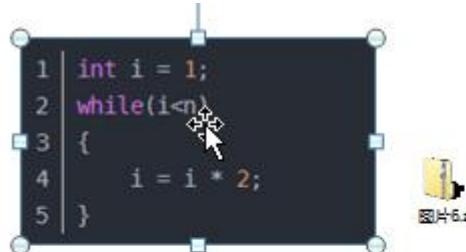
- 1) 常数阶 $O(1)$

无论代码执行了多少行, 只要是没有循环等复杂结构, 那这个代码的时间复杂度就都是 $O(1)$

```
1 | int i = 1;
2 | int j = 2;
3 | ++i;
4 | j++;
5 | int m = i + j;
```



上述代码在执行的时候, 它消耗的时候并不随着某个变量的增长而增长, 那么无论这类代码有多长, 即使有几万几十万行, 都可以用 $O(1)$ 来表示它的时间复杂度。

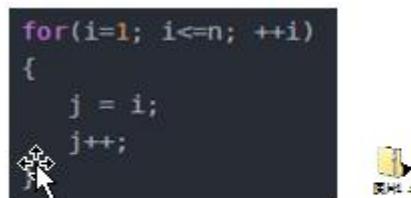
2) 对数阶 $O(\log_2 n)$ 

```
1 int i = 1;
2 while(i<n)
3 {
4     i = i * 2;
5 }
```

A screenshot of a code editor showing a simple Java code snippet. It contains a while loop where the variable 'i' is doubled in each iteration. A cursor is visible over the loop condition 'i < n'. The code is highlighted in purple and orange.

说明: 在while循环里面，每次都将 i 乘以2，乘完之后， i 距离 n 就越来越近了。假设循环 x 次之后， i 就大于 n 了，此时这个循环就退出了，也就是说 2 的 x 次方等于 n ，那么 $x = \log_2 n$ 也就是说当循环 $\log_2 n$ 次以后，这个代码就结束了。因此这个代码的时间复杂度为： $O(\log_2 n)$ 。 $O(\log_2 n)$ 的时间上是根据代码变化的， $i = i * 3$ ，则是 $O(\log_3 n)$.

如果 $N = a^x (a > 0, a \neq 1)$ ，即 a 的 x 次方等于 N （ $a>0$, 且 $a\neq 1$ ），那么数 x 叫做以 a 为底 N 的对数（logarithm），记作 $x = \log_a N$ 。其中， a 叫做对数的底数， N 叫做真数， x 叫做“以 a 为底 N 的对数”。

3) 线性阶 $O(n)$ 

```
for(i=1; i<=n; ++i)
{
    j = i;
    j++;
}
```

A screenshot of a code editor showing a for loop that runs from 1 to n. Inside the loop, there are two statements: 'j = i;' and 'j++'. A cursor is positioned before the first statement. The code is highlighted in purple and orange.

说明: 这段代码，for循环里面的代码会执行 n 遍，因此它消耗的时间是随着 n 的变化而变化的，因此这类代码都可以用 $O(n)$ 来表示它的时间复杂度

4) 线性对数阶 $O(n \log N)$

```
for(m=1; m<n; m++)
{
    i = 1;
    while(i<n)
    {
        i = i * 2;
    }
}
```



说明：线性对数阶 $O(n \log N)$ 其实非常容易理解，将时间复杂度为 $O(\log n)$ 的代码循环 N 遍的话，那么它的时间复杂度就是 $n * O(\log n)$ ，也就是 $O(n \log N)$

5) 平方阶 $O(n^2)$

```
for(x=1; i<=n; x++)
{
    for(i=1; i<=n; i++)
    {
        j = i;
        j++;
    }
}
```



说明：平方阶 $O(n^2)$ 就更容易理解了，如果把 $O(n)$ 的代码再嵌套循环一遍，它的时间复杂度就是 $O(n^2)$ ，这段代码其实就是嵌套了 2 层 n 循环，它的时间复杂度就是 $O(n * n)$ ，即 $O(n^2)$ 。如果将其中一层循环的 n 改成 m ，那它的时间复杂度就变成了 $O(m * n)$

6) 立方阶 $O(n^3)$ 、 K 次方阶 $O(n^k)$

说明：参考上面的 $O(n^2)$ 去理解就好了， $O(n^3)$ 相当于三层 n 循环，其它的类似

7.3.5 平均时间复杂度和最坏时间复杂度

- 1) 平均时间复杂度是指所有可能的输入实例均以等概率出现的情况下，该算法的运行时间。
- 2) 最坏情况下的时间复杂度称最坏时间复杂度。一般讨论的时间复杂度均是最坏情况下的时间复杂度。这样做的原因是：最坏情况下的时间复杂度是算法在任何输入实例上运行时间的界限，这就保证了算法的运行时间不会比最坏情况更长。
- 3) 平均时间复杂度和最坏时间复杂度是否一致，和算法有关(如图:)。

排序法	平均时间	最差情形	稳定度	额外空间	备注
冒泡	$O(n^2)$	$O(n^2)$	稳定	$O(1)$	n小时较好
交换	$O(n^2)$	$O(n^2)$	不稳定	$O(1)$	n小时较好
选择	$O(n^2)$	$O(n^2)$	不稳定	$O(1)$	n小时较好
插入	$O(n^2)$	$O(n^2)$	稳定	$O(1)$	大部分已排序时较好
基数	$O(\log_R B)$	$O(\log_R B)$	稳定	$O(n)$	B是真数(0-9), R是基数(个十百)
Shell	$O(n \log n)$	$O(n^s)$ $1 < s < 2$	不稳定	$O(1)$	s是所选分组
快速	$O(n \log n)$	$O(n^2)$	不稳定	$O(n \log n)$	n大时较好
归并	$O(n \log n)$	$O(n \log n)$	稳定	$O(1)$	n大时较好
堆	$O(n \log n)$	$O(n \log n)$	不稳定	$O(1)$	n大时较好

7.4 算法的空间复杂度简介

7.4.1 基本介绍

- 类似于时间复杂度的讨论，一个算法的空间复杂度(Space Complexity)定义为该算法所耗费的存储空间，它也是问题规模 n 的函数。
- 空间复杂度(Space Complexity)是对一个算法在运行过程中临时占用存储空间大小的量度。有的算法需要占用的临时工作单元数与解决问题的规模 n 有关，它随着 n 的增大而增大，当 n 较大时，将占用较多的存储单元，例如快速排序和归并排序算法，基数排序就属于这种情况
- 在做算法分析时，主要讨论的是时间复杂度。从用户使用体验上看，更看重的程序执行的速度。一些缓存产品(redis, memcache)和算法(基数排序)本质就是用空间换时间。

7.5 冒泡排序

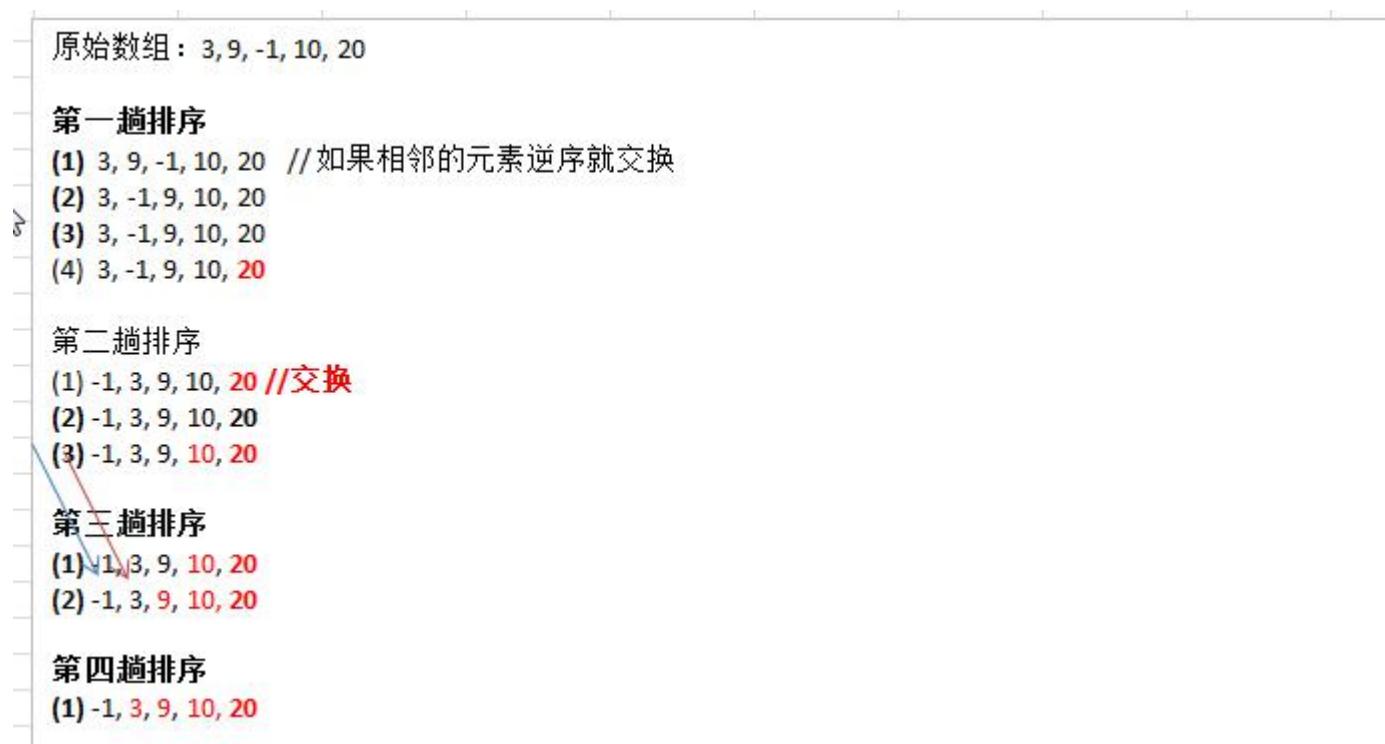
7.5.1 基本介绍

冒泡排序 (Bubble Sorting) 的基本思想是：通过对待排序序列从前向后（从下标较小的元素开始），依次比较相邻元素的值，若发现逆序则交换，使值较大的元素逐渐从前移向后部，就象水底下的气泡一样逐渐向上冒。

优化：

因为排序的过程中，各元素不断接近自己的位置，如果一趟比较下来没有进行过交换，就说明序列有序，因此要在排序过程中设置一个标志 flag 判断元素是否进行过交换。从而减少不必要的比较。(这里说的优化，可以在冒泡排序写好后，在进行)

7.5.2 演示冒泡过程的例子(图解)



小结上面的图解过程：

- (1) 一共进行 数组的大小-1 次 大的循环
- (2)每一趟排序的次数在逐渐的减少
- (3) 如果我们发现在某趟排序中，没有发生一次交换， 可以提前结束冒泡排序。这个就是优化

7.5.3 冒泡排序应用实例



我们举一个具体的案例来说明冒泡法。我们将五个无序的数：3, 9, -1, 10, -2 使用冒泡排序法将其排成一个从小到大的有序数列。

代码实现：

```
package com.atguigu.sort;

import java.text.SimpleDateFormat;
import java.util.Arrays;
import java.util.Date;

public class BubbleSort {

    public static void main(String[] args) {
        //    int arr[] = {3, 9, -1, 10, 20};
        //
        //    System.out.println("排序前");
        //    System.out.println(Arrays.toString(arr));

        //为了容易理解，我们把冒泡排序的演变过程，给大家展示

        //测试一下冒泡排序的速度 O(n^2)，给 80000 个数据，测试
        //创建要给 80000 个的随机的数组
        int[] arr = new int[80000];
```



```
for(int i = 0; i < 80000;i++) {  
    arr[i] = (int)(Math.random() * 8000000); //生成一个[0, 8000000) 数  
}  
  
Date data1 = new Date();  
SimpleDateFormat simpleDateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");  
String date1Str = simpleDateFormat.format(data1);  
System.out.println("排序前的时间是=" + date1Str);  
  
//测试冒泡排序  
bubbleSort(arr);  
  
Date data2 = new Date();  
String date2Str = simpleDateFormat.format(data2);  
System.out.println("排序后的时间是=" + date2Str);  
  
//System.out.println("排序后");  
//System.out.println(Arrays.toString(arr));  
  
/*  
 // 第二趟排序，就是将第二大的数排在倒数第二位  
  
 for (int j = 0; j < arr.length - 1 - 1 ;j++) {  
     // 如果前面的数比后面的数大，则交换
```



```
if (arr[j] > arr[j + 1]) {  
    temp = arr[j];  
    arr[j] = arr[j + 1];  
    arr[j + 1] = temp;  
}  
}
```

```
System.out.println("第二趟排序后的数组");  
System.out.println(Arrays.toString(arr));
```

```
// 第三趟排序，就是将第三大的数排在倒数第三位
```

```
for (int j = 0; j < arr.length - 1 - 2; j++) {  
    // 如果前面的数比后面的数大，则交换  
    if (arr[j] > arr[j + 1]) {  
        temp = arr[j];  
        arr[j] = arr[j + 1];  
        arr[j + 1] = temp;  
    }  
}
```

```
System.out.println("第三趟排序后的数组");  
System.out.println(Arrays.toString(arr));
```

```
// 第四趟排序，就是将第 4 大的数排在倒数第 4 位
```



```
for (int j = 0; j < arr.length - 1 - 3; j++) {  
    // 如果前面的数比后面的数大，则交换  
    if (arr[j] > arr[j + 1]) {  
        temp = arr[j];  
        arr[j] = arr[j + 1];  
        arr[j + 1] = temp;  
    }  
}  
  
System.out.println("第四趟排序后的数组");  
System.out.println(Arrays.toString(arr)); */  
  
}  
  
// 将前面的冒泡排序算法，封装成一个方法  
public static void bubbleSort(int[] arr) {  
    // 冒泡排序 的时间复杂度 O(n^2)，自己写出  
    int temp = 0; // 临时变量  
    boolean flag = false; // 标识变量，表示是否进行过交换  
    for (int i = 0; i < arr.length - 1; i++) {  
  
        for (int j = 0; j < arr.length - 1 - i; j++) {  
            // 如果前面的数比后面的数大，则交换  
            if (arr[j] > arr[j + 1]) {  
                flag = true;  
            }  
        }  
        if (flag) {  
            flag = false;  
            for (int j = 0; j < arr.length - 1 - i; j++) {  
                if (arr[j] > arr[j + 1]) {  
                    temp = arr[j];  
                    arr[j] = arr[j + 1];  
                    arr[j + 1] = temp;  
                }  
            }  
        }  
    }  
}
```



```
temp = arr[j];
arr[j] = arr[j + 1];
arr[j + 1] = temp;
}

}

//System.out.println("第" + (i + 1) + "趟排序后的数组");
//System.out.println(Arrays.toString(arr));

if (!flag) { // 在一趟排序中，一次交换都没有发生过
    break;
} else {
    flag = false; // 重置 flag!!!, 进行下次判断
}

}

}

}
```

7.6 选择排序

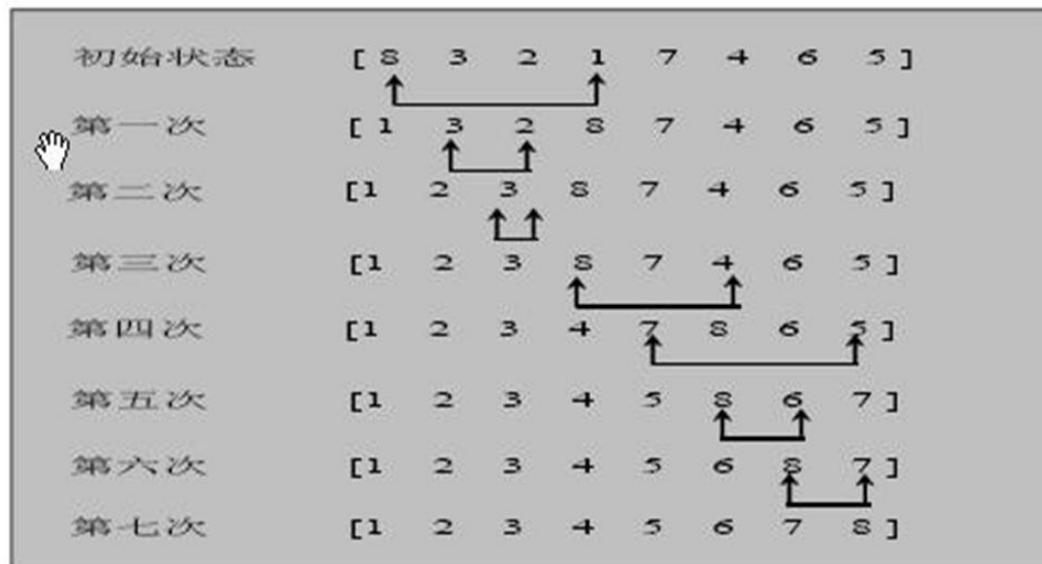
7.6.1 基本介绍

选择式排序也属于内部排序法，是从欲排序的数据中，按指定的规则选出某一元素，再依规定交换位置后达到排序的目的。

7.6.2 选择排序思想：

选择排序 (select sorting) 也是一种简单的排序方法。它的基本思想是：第一次从 $\text{arr}[0] \sim \text{arr}[n-1]$ 中选取最小值，与 $\text{arr}[0]$ 交换，第二次从 $\text{arr}[1] \sim \text{arr}[n-1]$ 中选取最小值，与 $\text{arr}[1]$ 交换，第三次从 $\text{arr}[2] \sim \text{arr}[n-1]$ 中选取最小值，与 $\text{arr}[2]$ 交换，…，第 i 次从 $\text{arr}[i-1] \sim \text{arr}[n-1]$ 中选取最小值，与 $\text{arr}[i-1]$ 交换，…，第 $n-1$ 次从 $\text{arr}[n-2] \sim \text{arr}[n-1]$ 中选取最小值，与 $\text{arr}[n-2]$ 交换，总共通过 $n-1$ 次，得到一个按排序码从小到大排列的有序序列。

7.6.3 选择排序思路分析图：

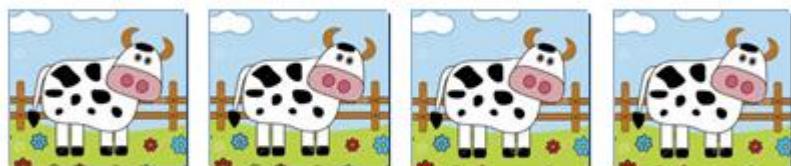


➤ 对一个数组的选择排序再进行讲解



7.6.4 选择排序应用实例：

有一群牛，颜值分别是 101, 34, 119, 1 请使用选择排序从低到高进行排序 [101, 34, 119, 1]



代码实现

```
package com.atguigu.sort;

import java.text.SimpleDateFormat;
import java.util.Arrays;
import java.util.Date;
//选择排序
public class SelectSort {

    public static void main(String[] args) {
        //int [] arr = {101, 34, 119, 1, -1, 90, 123};

        //创建要给 80000 个的随机的数组， 在我的机器是 2-3 秒， 比冒泡快.
        int[] arr = new int[80000];
        for (int i = 0; i < 80000; i++) {
            arr[i] = (int) (Math.random() * 8000000); // 生成一个[0, 8000000) 数
        }

        System.out.println("排序前");
```



```
//System.out.println(Arrays.toString(arr));\n\nDate data1 = new Date();\nSimpleDateFormat simpleDateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");\nString date1Str = simpleDateFormat.format(data1);\nSystem.out.println("排序前的时间是=" + date1Str);\n\nselectSort(arr);\n\nDate data2 = new Date();\nString date2Str = simpleDateFormat.format(data2);\nSystem.out.println("排序前的时间是=" + date2Str);\n\n//System.out.println("排序后");\n//System.out.println(Arrays.toString(arr));\n\n}\n\n//选择排序\npublic static void selectSort(int[] arr) {\n\n    //在推导的过程，我们发现了规律，因此，可以使用 for 来解决\n}
```



```
//选择排序时间复杂度是 O(n^2)

for (int i = 0; i < arr.length - 1; i++) {
    int minIndex = i;
    int min = arr[i];
    for (int j = i + 1; j < arr.length; j++) {
        if (min > arr[j]) { // 说明假定的最小值，并不是最小
            min = arr[j]; // 重置 min
            minIndex = j; // 重置 minIndex
        }
    }

    // 将最小值，放在 arr[0]，即交换
    if (minIndex != i) {
        arr[minIndex] = arr[i];
        arr[i] = min;
    }
}

//System.out.println("第"+(i+1)+"轮后~~");
//System.out.println(Arrays.toString(arr)); // 1, 34, 119, 101
}

/*
//使用逐步推导的方式来，讲解选择排序
//第 1 轮
```



```
//原始的数组 : 101, 34, 119, 1  
//第一轮排序 : 1, 34, 119, 101  
//算法 先简单--> 做复杂， 就是可以把一个复杂的算法， 拆分成简单的问题-> 逐步解决
```

```
//第 1 轮  
  
int minIndex = 0;  
int min = arr[0];  
for(int j = 0 + 1; j < arr.length; j++) {  
    if (min > arr[j]) { //说明假定的最小值，并不是最小  
        min = arr[j]; //重置 min  
        minIndex = j; //重置 minIndex  
    }  
}
```

```
//将最小值，放在 arr[0]，即交换  
  
if(minIndex != 0) {  
    arr[minIndex] = arr[0];  
    arr[0] = min;  
}
```

```
System.out.println("第 1 轮后~");  
System.out.println(Arrays.toString(arr)); // 1, 34, 119, 101
```

```
//第 2 轮
```



```
minIndex = 1;  
min = arr[1];  
for (int j = 1 + 1; j < arr.length; j++) {  
    if (min > arr[j]) { // 说明假定的最小值，并不是最小  
        min = arr[j]; // 重置 min  
        minIndex = j; // 重置 minIndex  
    }  
}  
  
// 将最小值，放在 arr[0]，即交换  
if(minIndex != 1) {  
    arr[minIndex] = arr[1];  
    arr[1] = min;  
}  
  
System.out.println("第 2 轮后~~");  
System.out.println(Arrays.toString(arr)); // 1, 34, 119, 101  
  
// 第 3 轮  
minIndex = 2;  
min = arr[2];  
for (int j = 2 + 1; j < arr.length; j++) {  
    if (min > arr[j]) { // 说明假定的最小值，并不是最小  
        min = arr[j]; // 重置 min  
        minIndex = j; // 重置 minIndex  
    }  
}
```



```
}
```

```
// 将最小值，放在 arr[0]，即交换
if (minIndex != 2) {
    arr[minIndex] = arr[2];
    arr[2] = min;
}

System.out.println("第 3 轮后~~");
System.out.println(Arrays.toString(arr)); // 1, 34, 101, 119 */
```

```
}
```

```
}
```

7.7 插入排序

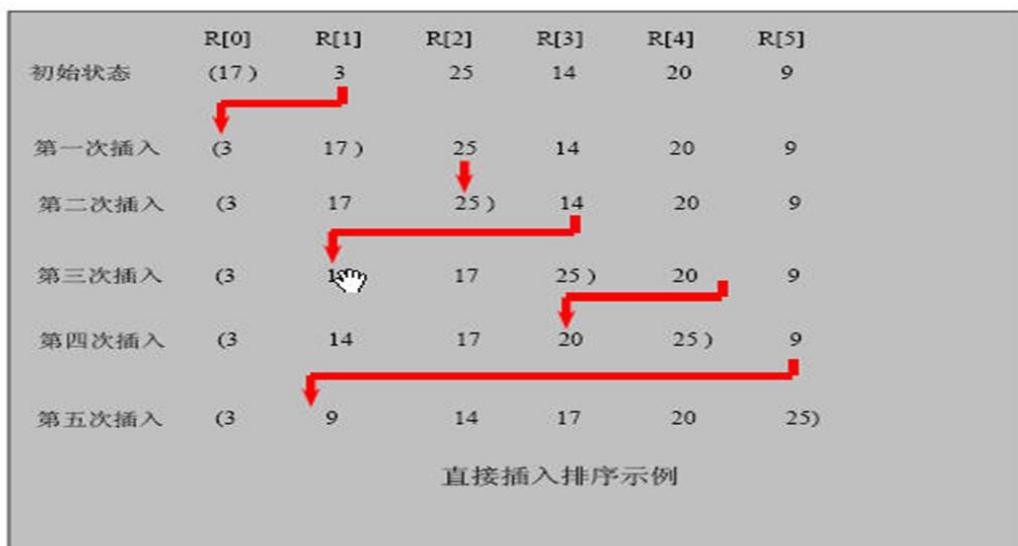
7.7.1 插入排序法介绍：

插入式排序属于内部排序法，是对于欲排序的元素以插入的方式找寻该元素的适当位置，以达到排序的目的。

7.7.2 插入排序法思想：

插入排序（Insertion Sorting）的基本思想是：把 **n** 个待排序的元素看成为一个有序表和一个无序表，开始时有序表中只包含一个元素，无序表中包含有 **n-1** 个元素，排序过程中每次从无序表中取出第一个元素，把它的排序码依次与有序表元素的排序码进行比较，将它插入到有序表中的适当位置，使之成为新的有序表。

7.7.3 插入排序思路图：



7.7.4 插入排序法应用实例：

有一群小牛，考试成绩分别是 101, 34, 119, 1 请从小到大排序

代码实现：

```
package com.atguigu.sort;

import java.text.SimpleDateFormat;
import java.util.Arrays;
import java.util.Date;

public class InsertSort {

    public static void main(String[] args) {
        //int[] arr = {101, 34, 119, 1, -1, 89};
        // 创建要给 80000 个的随机的数组
        int[] arr = new int[80000];
        for (int i = 0; i < 80000; i++) {
```



```
arr[i] = (int) (Math.random() * 8000000); // 生成一个[0, 8000000) 数  
}  
  
System.out.println("排序前");  
Date data1 = new Date();  
SimpleDateFormat simpleDateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");  
String date1Str = simpleDateFormat.format(data1);  
System.out.println("排序前的时间是=" + date1Str);  
  
insertSort(arr); //调用插入排序算法  
  
Date data2 = new Date();  
String date2Str = simpleDateFormat.format(data2);  
System.out.println("排序前的时间是=" + date2Str);  
  
//System.out.println(Arrays.toString(arr));  
  
}  
  
//插入排序  
public static void insertSort(int[] arr) {  
    int insertVal = 0;  
    int insertIndex = 0;
```



```
//使用 for 循环来把代码简化

for(int i = 1; i < arr.length; i++) {

    //定义待插入的数
    insertVal = arr[i];
    insertIndex = i - 1; // 即 arr[1]的前面这个数的下标

    // 给 insertVal 找到插入的位置
    // 说明
    // 1. insertIndex >= 0 保证在给 insertVal 找插入位置，不越界
    // 2. insertVal < arr[insertIndex] 待插入的数，还没有找到插入位置
    // 3. 就需要将 arr[insertIndex] 后移
    while (insertIndex >= 0 && insertVal < arr[insertIndex]) {

        arr[insertIndex + 1] = arr[insertIndex];// arr[insertIndex]
        insertIndex--;
    }

    // 当退出 while 循环时，说明插入的位置找到, insertIndex + 1
    // 举例：理解不了，我们一会 debug
    //这里我们判断是否需要赋值
    if(insertIndex + 1 != i) {

        arr[insertIndex + 1] = insertVal;
    }

    //System.out.println("第"+i+"轮插入");
    //System.out.println(Arrays.toString(arr));
}

}
```



```
/*
//使用逐步推导的方式来讲解，便利理解
//第 1 轮 {101, 34, 119, 1}; => {34, 101, 119, 1}

//{101, 34, 119, 1};=> {101,101,119,1}
//定义待插入的数
int insertVal = arr[1];
int insertIndex = 1 - 1; //即 arr[1]的前面这个数的下标

//给 insertVal 找到插入的位置
//说明
//1. insertIndex >= 0 保证在给 insertVal 找插入位置，不越界
//2. insertVal < arr[insertIndex] 待插入的数，还没有找到插入位置
//3. 就需要将 arr[insertIndex] 后移
while(insertIndex >= 0 && insertVal < arr[insertIndex] ) {
    arr[insertIndex + 1] = arr[insertIndex];// arr[insertIndex]
    insertIndex--;
}

//当退出 while 循环时，说明插入的位置找到, insertIndex + 1
//举例：理解不了，我们一会 debug
arr[insertIndex + 1] = insertVal;
```



```
System.out.println("第 1 轮插入");

System.out.println(Arrays.toString(arr));

//第 2 轮
insertVal = arr[2];
insertIndex = 2 - 1;

while(insertIndex >= 0 && insertVal < arr[insertIndex] ) {
    arr[insertIndex + 1] = arr[insertIndex];// arr[insertIndex]
    insertIndex--;
}

arr[insertIndex + 1] = insertVal;
System.out.println("第 2 轮插入");
System.out.println(Arrays.toString(arr));

//第 3 轮
insertVal = arr[3];
insertIndex = 3 - 1;

while (insertIndex >= 0 && insertVal < arr[insertIndex]) {
    arr[insertIndex + 1] = arr[insertIndex];// arr[insertIndex]
    insertIndex--;
}
```



```
arr[insertIndex + 1] = insertVal;  
System.out.println("第 3 轮插入");  
System.out.println(Arrays.toString(arr)); */  
  
}  
  
}
```

7.8 希尔排序

7.8.1 简单插入排序存在的问题

我们看简单的插入排序可能存在的问题.

数组 arr = {2,3,4,5,6,1} 这时需要插入的数 1(最小), 这样的过程是:

```
{2,3,4,5,6,6}  
{2,3,4,5,5,6}  
{2,3,4,4,5,6}  
{2,3,3,4,5,6}  
{2,2,3,4,5,6}  
{1,2,3,4,5,6}
```

结论: 当需要插入的数是较小的数时, 后移的次数明显增多, 对效率有影响.

7.8.2 希尔排序法介绍

希尔排序是希尔 (Donald Shell) 于 1959 年提出的一种排序算法。希尔排序也是一种插入排序, 它是简单插入

排序经过改进之后的一个更高效的版本，也称为缩小增量排序。

7.8.3 希尔排序法基本思想

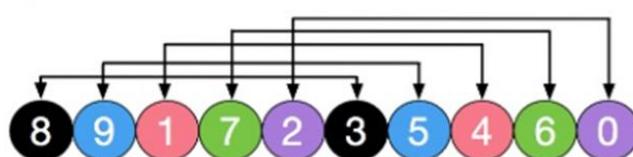
希尔排序是把记录按下标的一定增量分组，对每组使用直接插入排序算法排序；随着增量逐渐减少，每组包含的关键词越来越多，当增量减至 1 时，整个文件恰被分成一组，算法便终止

7.8.4 希尔排序法的示意图

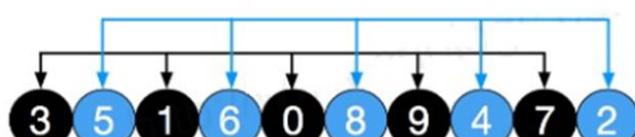
原始数组 以下数据元素颜色相同为一组



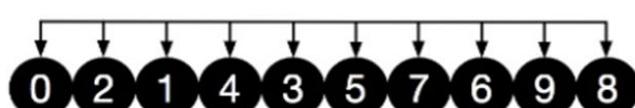
初始增量 $gap = \text{length}/2 = 5$ ，意味着整个数组被分为5组， $[8,3][9,5][1,4][7,6][2,0]$



对这5组分别进行直接插入排序，结果如下，可以看到，像3, 5, 6这些小元素都被调到前面了，然后缩小增量 $gap=5/2=2$ ，数组被分为2组 $[3,1,0,9,7][5,6,8,4,2]$



对以上2组再分别进行直接插入排序，结果如下，可以看到，此时整个数组的有序程度更进一步啦。
再缩小增量 $gap=2/2=1$ ，此时，整个数组为1组 $[0,2,1,4,3,5,7,6,9,8]$ ，如下



经过上面的“宏观调控”，整个数组的有序化程度成果喜人。

此时，仅仅需要对以上数列简单微调，无需大量移动操作即可完成整个数组的排序。



7.8.5 希尔排序法应用实例：

有一群小牛，考试成绩分别是 {8,9,1,7,2,3,5,4,6,0} 请从小到大排序。请分别使用

- 1) 希尔排序时， 对有序序列在插入时采用**交换法**，并测试排序速度。
- 2) 希尔排序时， 对有序序列在插入时采用**移动法**，并测试排序速度
- 3) 代码实现

```
package com.atguigu.sort;

import java.text.SimpleDateFormat;
import java.util.Arrays;
import java.util.Date;

public class ShellSort {

    public static void main(String[] args) {
        //int[] arr = { 8, 9, 1, 7, 2, 3, 5, 4, 6, 0 };

        // 创建要给 80000 个的随机的数组
        int[] arr = new int[80000];
        for (int i = 0; i < 80000; i++) {
```



```
arr[i] = (int) (Math.random() * 8000000); // 生成一个[0, 8000000) 数  
}  
  
System.out.println("排序前");  
Date data1 = new Date();  
SimpleDateFormat simpleDateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");  
String date1Str = simpleDateFormat.format(data1);  
System.out.println("排序前的时间是=" + date1Str);  
  
//shellSort(arr); //交换式  
shellSort2(arr); //移位方式  
  
Date data2 = new Date();  
String date2Str = simpleDateFormat.format(data2);  
System.out.println("排序前的时间是=" + date2Str);  
  
//System.out.println(Arrays.toString(arr));  
}  
  
// 使用逐步推导的方式来编写希尔排序  
// 希尔排序时， 对有序序列在插入时采用交换法，  
// 思路(算法)==> 代码  
public static void shellSort(int[] arr) {  
  
    int temp = 0;  
    int count = 0;
```



```
// 根据前面的逐步分析，使用循环处理

for (int gap = arr.length / 2; gap > 0; gap /= 2) {
    for (int i = gap; i < arr.length; i++) {
        // 遍历各组中所有的元素(共 gap 组，每组有个元素)，步长 gap
        for (int j = i - gap; j >= 0; j -= gap) {
            // 如果当前元素大于加上步长后的那个元素，说明交换
            if (arr[j] > arr[j + gap]) {
                temp = arr[j];
                arr[j] = arr[j + gap];
                arr[j + gap] = temp;
            }
        }
    }
}

//System.out.println("希尔排序第" + (++count) + "轮 =" + Arrays.toString(arr));
}

/*
// 希尔排序的第 1 轮排序
// 因为第 1 轮排序，是将 10 个数据分成了 5 组
for (int i = 5; i < arr.length; i++) {
    // 遍历各组中所有的元素(共 5 组，每组有 2 个元素)，步长 5
    for (int j = i - 5; j >= 0; j -= 5) {
        // 如果当前元素大于加上步长后的那个元素，说明交换
        if (arr[j] > arr[j + 5]) {
            temp = arr[j];
            arr[j] = arr[j + 5];
            arr[j + 5] = temp;
        }
    }
}
```



```
arr[j] = arr[j + 5];
arr[j + 5] = temp;
}

}

System.out.println("希尔排序 1 轮后=" + Arrays.toString(arr));//



// 希尔排序的第 2 轮排序
// 因为第 2 轮排序，是将 10 个数据分成了  $5/2 = 2$  组
for (int i = 2; i < arr.length; i++) {
    // 遍历各组中所有的元素(共 5 组，每组有 2 个元素)，步长 5
    for (int j = i - 2; j >= 0; j -= 2) {
        // 如果当前元素大于加上步长后的那个元素，说明交换
        if (arr[j] > arr[j + 2]) {
            temp = arr[j];
            arr[j] = arr[j + 2];
            arr[j + 2] = temp;
        }
    }
}

System.out.println("希尔排序 2 轮后=" + Arrays.toString(arr));//


// 希尔排序的第 3 轮排序
```



```
// 因为第 3 轮排序，是将 10 个数据分成了 2/2 = 1 组
for (int i = 1; i < arr.length; i++) {
    // 遍历各组中所有的元素(共 5 组，每组有 2 个元素)，步长 5
    for (int j = i - 1; j >= 0; j -= 1) {
        // 如果当前元素大于加上步长后的那个元素，说明交换
        if (arr[j] > arr[j + 1]) {
            temp = arr[j];
            arr[j] = arr[j + 1];
            arr[j + 1] = temp;
        }
    }
}

System.out.println("希尔排序 3 轮后=" + Arrays.toString(arr));//*
}
```

```
//对交换式的希尔排序进行优化->移位法
public static void shellSort2(int[] arr) {

    // 增量 gap，并逐步的缩小增量
    for (int gap = arr.length / 2; gap > 0; gap /= 2) {
        // 从第 gap 个元素，逐个对其所在的组进行直接插入排序
        for (int i = gap; i < arr.length; i++) {
            int j = i;
            int temp = arr[j];

```



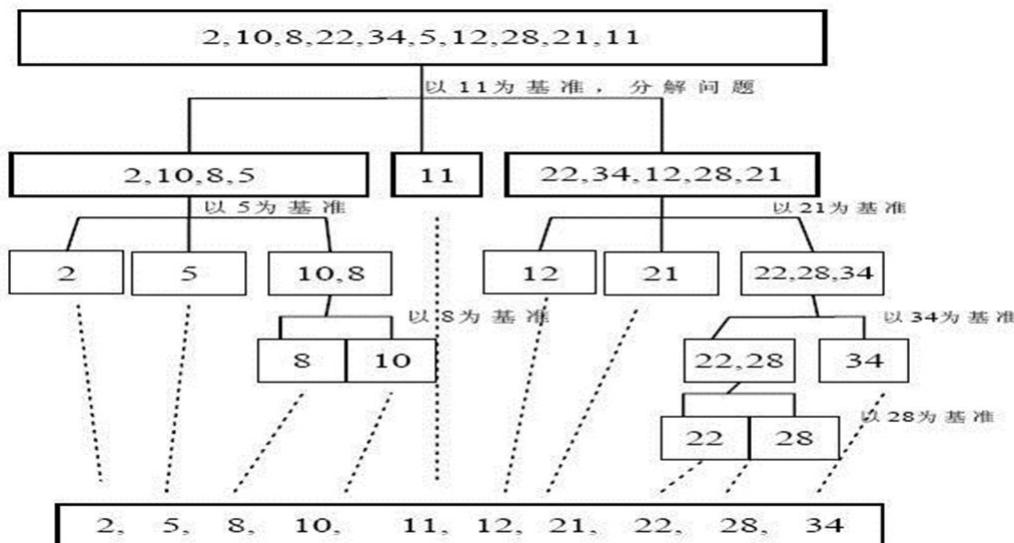
```
if (arr[j] < arr[j - gap]) {  
    while (j - gap >= 0 && temp < arr[j - gap]) {  
        //移动  
        arr[j] = arr[j-gap];  
        j -= gap;  
    }  
    //当退出 while 后，就给 temp 找到插入的位置  
    arr[j] = temp;  
}  
  
}  
}  
}  
}
```

7.9 快速排序

7.9.1 快速排序法介绍：

快速排序（Quicksort）是对冒泡排序的一种改进。基本思想是：通过一趟排序将要排序的数据分割成独立的两部分，其中一部分的所有数据都比另外一部分的所有数据都要小，然后再按此方法对这两部分数据分别进行快速排序，整个排序过程可以递归进行，以此达到整个数据变成有序序列。

7.9.2 快速排序法示意图：



7.9.3 快速排序法应用实例:

要求: 对 [-9, 78, 0, 23, -567, 70] 进行从小到大的排序, 要求使用快速排序法。【测试 8w 和 800w】

说明[验证分析]:

- 1) 如果取消左右递归, 结果是 -9 -567 0 23 78 70



- 2) 如果取消右递归,结果是 -567 -9 0 23 78 70
- 3) 如果取消左递归,结果是 -9 -567 0 23 70 78
- 4) 代码实现

```
package com.atguigu.sort;

import java.text.SimpleDateFormat;
import java.util.Arrays;
import java.util.Date;

public class QuickSort {

    public static void main(String[] args) {
        //int[] arr = {-9,78,0,23,-567,70, -1,900, 4561};

        //测试快排的执行速度
        // 创建要给 80000 个的随机的数组
        int[] arr = new int[8000000];
        for (int i = 0; i < 8000000; i++) {
            arr[i] = (int) (Math.random() * 8000000); // 生成一个[0, 8000000) 数
        }

        System.out.println("排序前");
        Date data1 = new Date();
        SimpleDateFormat simpleDateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        String date1Str = simpleDateFormat.format(data1);
        System.out.println("排序前的时间是=" + date1Str);
    }
}
```



```
quickSort(arr, 0, arr.length-1);

Date data2 = new Date();
String date2Str = simpleDateFormat.format(data2);
System.out.println("排序前的时间是=" + date2Str);
//System.out.println("arr=" + Arrays.toString(arr));
}

public static void quickSort(int[] arr,int left, int right) {
    int l = left; //左下标
    int r = right; //右下标
    //pivot 中轴值
    int pivot = arr[(left + right) / 2];
    int temp = 0; //临时变量，作为交换时使用
    //while 循环的目的是让比 pivot 值小放到左边
    //比 pivot 值大放到右边
    while( l < r) {
        //在 pivot 的左边一直找,找到大于等于 pivot 值,才退出
        while( arr[l] < pivot) {
            l += 1;
        }
        //在 pivot 的右边一直找,找到小于等于 pivot 值,才退出
        while(arr[r] > pivot) {
            r -= 1;
        }
    }
}
```



```
//如果 l >= r 说明 pivot 的左右两的值，已经按照左边全部是
//小于等于 pivot 值，右边全部是大于等于 pivot 值
if( l >= r) {
    break;
}

//交换
temp = arr[l];
arr[l] = arr[r];
arr[r] = temp;

//如果交换完后，发现这个 arr[l] == pivot 值 相等 r--, 前移
if(arr[l] == pivot) {
    r -= 1;
}

//如果交换完后，发现这个 arr[r] == pivot 值 相等 l++, 后移
if(arr[r] == pivot) {
    l += 1;
}

// 如果 l == r, 必须 l++, r--, 否则为出现栈溢出
if(l == r) {
    l += 1;
    r -= 1;
}
```



```
//向左递归
if(left < r) {
    quickSort(arr, left, r);
}

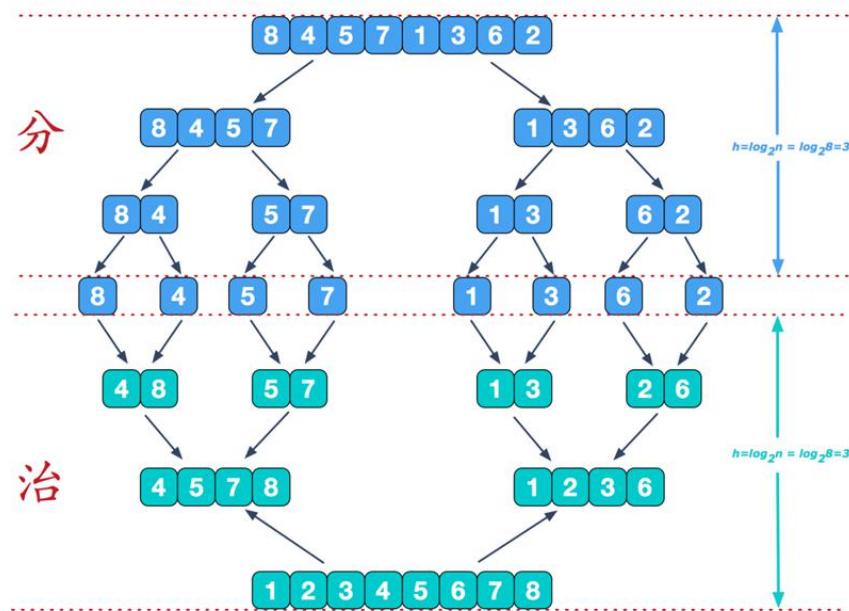
//向右递归
if(right > l) {
    quickSort(arr, l, right);
}
}
```

7.10 归并排序

7.10.1 归并排序介绍:

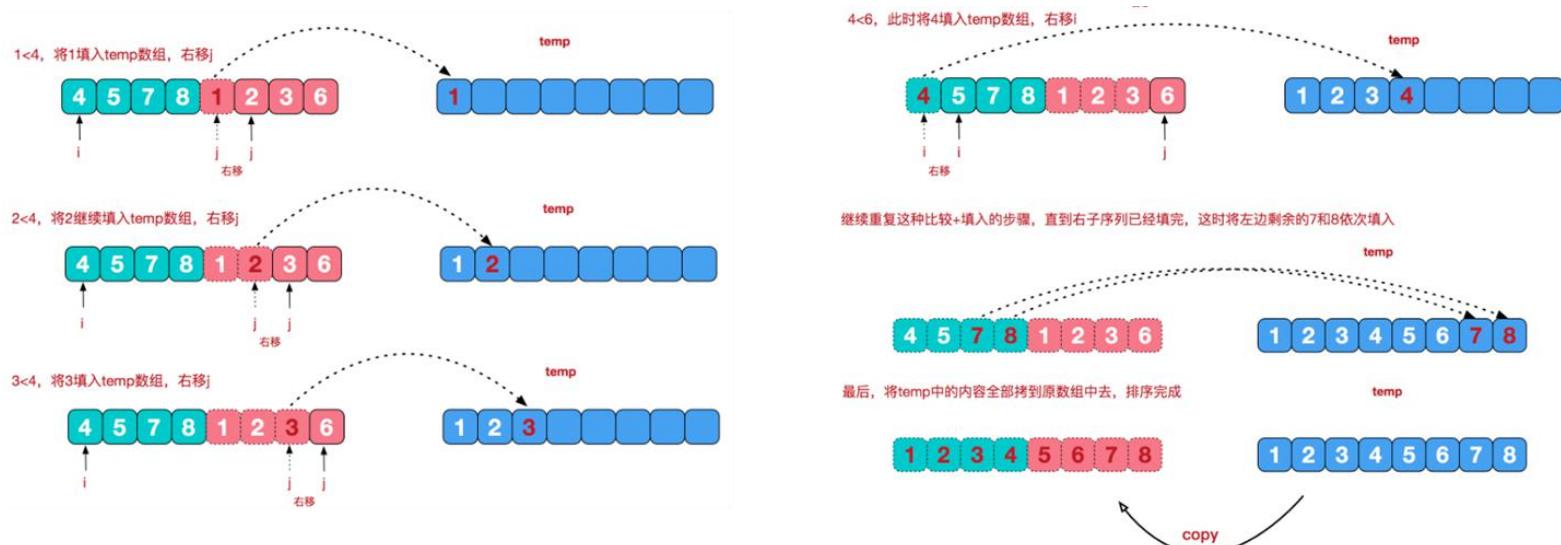
归并排序（MERGE-SORT）是利用归并的思想实现的排序方法，该算法采用经典的**分治（divide-and-conquer）策略**（分治法将问题分(divide)成一些小的问题然后递归求解，而治(conquer)的阶段则将分的阶段得到的各答案"修补"在一起，即分而治之）。

7.10.2 归并排序思想示意图 1-基本思想:



7.10.3 归并排序思想示意图 2-合并相邻有序子序列:

再来看看治阶段，我们需要将两个已经有序的子序列合并成一个有序序列，比如上图中的最后一次合并，要将[4,5,7,8]和[1,2,3,6]两个已经有序的子序列，合并为最终序列[1,2,3,4,5,6,7,8]，来看下实现步骤



7.10.4 归并排序的应用实例:

给你一个数组, val arr = Array(8, 4, 5, 7, 1, 3, 6, 2), 请使用归并排序完成排序。



代码演示：

```
package com.atguigu.sort;

import java.text.SimpleDateFormat;
import java.util.Arrays;
import java.util.Date;

public class MergetSort {

    public static void main(String[] args) {
        //int arr[] = { 8, 4, 5, 7, 1, 3, 6, 2 }; //

        //测试快排的执行速度
        // 创建要给 80000 个的随机的数组
        int[] arr = new int[8000000];
        for (int i = 0; i < 8000000; i++) {
            arr[i] = (int) (Math.random() * 8000000); // 生成一个[0, 8000000) 数
        }
        System.out.println("排序前");

        Date data1 = new Date();
        SimpleDateFormat simpleDateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
        String date1Str = simpleDateFormat.format(data1);
        System.out.println("排序前的时间是=" + date1Str);

        int temp[] = new int[arr.length]; //归并排序需要一个额外空间
        mergeSort(arr, 0, arr.length - 1, temp);
```



```
Date data2 = new Date();
String date2Str = simpleDateFormat.format(data2);
System.out.println("排序前的时间是=" + date2Str);

//System.out.println("归并排序后=" + Arrays.toString(arr));
}

//分+合方法
public static void mergeSort(int[] arr, int left, int right, int[] temp) {
    if(left < right) {
        int mid = (left + right) / 2; //中间索引
        //向左递归进行分解
        mergeSort(arr, left, mid, temp);
        //向右递归进行分解
        mergeSort(arr, mid + 1, right, temp);
        //合并
        merge(arr, left, mid, right, temp);
    }
}

//合并的方法
/**
 *
```



```
* @param arr 排序的原始数组
* @param left 左边有序序列的初始索引
* @param mid 中间索引
* @param right 右边索引
* @param temp 做中转的数组
*/
public static void merge(int[] arr, int left, int mid, int right, int[] temp) {

    int i = left; // 初始化 i, 左边有序序列的初始索引
    int j = mid + 1; // 初始化 j, 右边有序序列的初始索引
    int t = 0; // 指向 temp 数组的当前索引

    //((一))
    //先把左右两边(有序)的数据按照规则填充到 temp 数组
    //直到左右两边的有序序列，有一边处理完毕为止
    while (i <= mid && j <= right) { //继续
        //如果左边的有序序列的当前元素，小于等于右边有序序列的当前元素
        //即将左边的当前元素，填充到 temp 数组
        //然后 t++, i++
        if(arr[i] <= arr[j]) {
            temp[t] = arr[i];
            t += 1;
            i += 1;
        } else { //反之,将右边有序序列的当前元素，填充到 temp 数组
            temp[t] = arr[j];
            t += 1;
        }
    }
}
```



```
j += 1;  
}  
}  
  
//(二)  
//把有剩余数据的一边的数据依次全部填充到 temp  
while( i <= mid) { //左边的有序序列还有剩余的元素，就全部填充到 temp  
    temp[t] = arr[i];  
    t += 1;  
    i += 1;  
}  
  
while( j <= right) { //右边的有序序列还有剩余的元素，就全部填充到 temp  
    temp[t] = arr[j];  
    t += 1;  
    j += 1;  
}  
  
//(三)  
//将 temp 数组的元素拷贝到 arr  
//注意，并不是每次都拷贝所有  
t = 0;  
int tempLeft = left; //  
//第一次合并 tempLeft = 0 , right = 1 //  tempLeft = 2  right = 3 // tL=0 ri=3  
//最后一次 tempLeft = 0  right = 7
```



```
while(tempLeft <= right) {  
    arr[tempLeft] = temp[t];  
    t += 1;  
    tempLeft += 1;  
}  
  
}  
}
```

7.11 基数排序

7.11.1 基数排序(桶排序)介绍:

- 1) 基数排序 (radix sort) 属于“分配式排序” (distribution sort)，又称“桶子法” (bucket sort) 或 bin sort，顾名思义，它是通过键值的各个位的值，将要排序的元素分配至某些“桶”中，达到排序的作用
- 2) 基数排序法是属于稳定性的排序，基数排序法的是效率高的**稳定性排序法**
- 3) 基数排序(Radix Sort)是桶排序的扩展
- 4) 基数排序是 1887 年赫尔曼 · 何乐礼发明的。它是这样实现的：将整数按位数切割成不同的数字，然后按每个位数分别比较。

7.11.2 基数排序基本思想

- 1) 将所有待比较数值统一为同样的数位长度，数位较短的数前面补零。然后，从最低位开始，依次进行一次排序。这样从最低位排序一直到最高位排序完成以后，数列就变成一个有序序列。

2) 这样说明, 比较难理解, 下面我们看一个图文解释, 理解基数排序的步骤

7.11.3 基数排序图文说明

将数组 $\{53, 3, 542, 748, 14, 214\}$ 使用基数排序, 进行升序排序





7.11.4 基数排序代码实现

要求：将数组 {53, 3, 542, 748, 14, 214} 使用基数排序，进行升序排序

- 1) 思路分析：前面的图文已经讲明确
- 2) 代码实现：看老师演示

```
package com.atguigu.sort;

import java.text.SimpleDateFormat;
import java.util.Arrays;
import java.util.Date;

public class RadixSort {

    public static void main(String[] args) {
        //int arr[] = { 53, 3, 542, 748, 14, 214};
```



```
// 80000000 * 11 * 4 / 1024 / 1024 / 1024 =3.3G

int[] arr = new int[8000000];
for (int i = 0; i < 8000000; i++) {
    arr[i] = (int) (Math.random() * 8000000); // 生成一个[0, 8000000) 数
}
System.out.println("排序前");

Date data1 = new Date();
SimpleDateFormat simpleDateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");
String date1Str = simpleDateFormat.format(data1);
System.out.println("排序前的时间是=" + date1Str);

radixSort(arr);

Date data2 = new Date();
String date2Str = simpleDateFormat.format(data2);
System.out.println("排序前的时间是=" + date2Str);

//System.out.println("基数排序后 " + Arrays.toString(arr));

}

//基数排序方法
public static void radixSort(int[] arr) {

    //根据前面的推导过程，我们可以得到最终的基数排序代码
}
```



```
//1. 得到数组中最大的数的位数
```

```
int max = arr[0]; //假设第一数就是最大数
for(int i = 1; i < arr.length; i++) {
    if (arr[i] > max) {
        max = arr[i];
    }
}
```

```
//得到最大数是几位数
```

```
int maxLength = (max + "").length();
```

```
//定义一个二维数组，表示 10 个桶，每个桶就是一个一维数组
```

```
//说明
```

```
//1. 二维数组包含 10 个一维数组
```

```
//2. 为了防止在放入数的时候，数据溢出，则每个一维数组(桶)，大小定为 arr.length
```

```
//3. 明确，基数排序是使用空间换时间的经典算法
```

```
int[][] bucket = new int[10][arr.length];
```

```
//为了记录每个桶中，实际存放了多少个数据，我们定义一个一维数组来记录各个桶的每次放入的数据个数
```

```
//可以这里理解
```

```
//比如：bucketElementCounts[0]，记录的就是 bucket[0] 桶的放入数据个数
```

```
int[] bucketElementCounts = new int[10];
```

```
//这里我们使用循环将代码处理
```



```
for(int i = 0 , n = 1; i < maxLength; i++, n *= 10) {  
    //对每个元素的对应位进行排序处理), 第一次是个位, 第二次是十位, 第三次是百位..  
    for(int j = 0; j < arr.length; j++) {  
        //取出每个元素的对应位的值  
        int digitOfElement = arr[j] / n % 10;  
        //放入到对应的桶中  
        bucket[digitOfElement][bucketElementCounts[digitOfElement]] = arr[j];  
        bucketElementCounts[digitOfElement]++;  
    }  
    //按照这个桶的顺序(一维数组的下标依次取出数据, 放入原来数组)  
    int index = 0;  
    //遍历每一桶, 并将桶中是数据, 放入到原数组  
    for(int k = 0; k < bucketElementCounts.length; k++) {  
        //如果桶中, 有数据, 我们才放入到原数组  
        if(bucketElementCounts[k] != 0) {  
            //循环该桶即第 k 个桶(即第 k 个一维数组), 放入  
            for(int l = 0; l < bucketElementCounts[k]; l++) {  
                //取出元素放到 arr  
                arr[index++] = bucket[k][l];  
            }  
        }  
        //第 i+1 轮处理后, 需要将每个 bucketElementCounts[k] = 0 ! ! ! !  
        bucketElementCounts[k] = 0;  
    }  
    //System.out.println("第"+(i+1)+"轮, 对个位的排序处理 arr =" + Arrays.toString(arr));
```



```
}
```

```
/*
```

```
//第 1 轮(针对每个元素的个位进行排序处理)
for(int j = 0; j < arr.length; j++) {
    //取出每个元素的个位的值
    int digitOfElement = arr[j] / 1 % 10;
    //放入到对应的桶中
    bucket[digitOfElement][bucketElementCounts[digitOfElement]] = arr[j];
    bucketElementCounts[digitOfElement]++;
}

//按照这个桶的顺序(一维数组的下标依次取出数据，放入原来数组)
int index = 0;
//遍历每一桶，并将桶中是数据，放入到原数组
for(int k = 0; k < bucketElementCounts.length; k++) {
    //如果桶中，有数据，我们才放入到原数组
    if(bucketElementCounts[k] != 0) {
        //循环该桶即第 k 个桶(即第 k 个一维数组)，放入
        for(int l = 0; l < bucketElementCounts[k]; l++) {
            //取出元素放入到 arr
            arr[index++] = bucket[k][l];
        }
    }
}

//第 1 轮处理后，需要将每个 bucketElementCounts[k] = 0 ! ! ! !
```



```
bucketElementCounts[k] = 0;

}

System.out.println("第 1 轮， 对个位的排序处理 arr =" + Arrays.toString(arr));

=====

//第 2 轮(针对每个元素的十位进行排序处理)
for (int j = 0; j < arr.length; j++) {
    // 取出每个元素的十位的值
    int digitOfElement = arr[j] / 10 % 10; //748 / 10 => 74 % 10 => 4
    // 放入到对应的桶中
    bucket[digitOfElement][bucketElementCounts[digitOfElement]] = arr[j];
    bucketElementCounts[digitOfElement]++;
}

// 按照这个桶的顺序(一维数组的下标依次取出数据， 放入原来数组)
index = 0;
// 遍历每一桶， 并将桶中是数据， 放入到原数组
for (int k = 0; k < bucketElementCounts.length; k++) {
    // 如果桶中， 有数据， 我们才放入到原数组
    if (bucketElementCounts[k] != 0) {
        // 循环该桶即第 k 个桶(即第 k 个一维数组)， 放入
        for (int l = 0; l < bucketElementCounts[k]; l++) {
            // 取出元素放入到 arr
            arr[index++] = bucket[k][l];
        }
    }
}
```



```
        }

    }

//第 2 轮处理后，需要将每个 bucketElementCounts[k] = 0 ! ! ! !
bucketElementCounts[k] = 0;

}

System.out.println("第 2 轮，对个位的排序处理 arr =" + Arrays.toString(arr));




//第 3 轮(针对每个元素的百位进行排序处理)
for (int j = 0; j < arr.length; j++) {
    // 取出每个元素的百位的值
    int digitOfElement = arr[j] / 100 % 10; // 748 / 100 => 7 % 10 = 7
    // 放入到对应的桶中
    bucket[digitOfElement][bucketElementCounts[digitOfElement]] = arr[j];
    bucketElementCounts[digitOfElement]++;
}

// 按照这个桶的顺序(一维数组的下标依次取出数据，放入原来数组)
index = 0;
// 遍历每一桶，并将桶中是数据，放入到原数组
for (int k = 0; k < bucketElementCounts.length; k++) {
    // 如果桶中，有数据，我们才放入到原数组
    if (bucketElementCounts[k] != 0) {
        // 循环该桶即第 k 个桶(即第 k 个一维数组)，放入
        for (int l = 0; l < bucketElementCounts[k]; l++) {
            // 取出元素放入到 arr
            arr[index++] = bucket[k][l];
        }
    }
}
```

```
        }

    }

//第 3 轮处理后，需要将每个 bucketElementCounts[k] = 0 ! ! ! !
bucketElementCounts[k] = 0;

}

System.out.println("第 3 轮，对个位的排序处理 arr =" + Arrays.toString(arr)); */

}

}
```

7.11.5 基数排序的说明：

- 1) 基数排序是对传统桶排序的扩展，速度很快。
- 2) 基数排序是经典的空间换时间的方式，占用内存很大，当对海量数据排序时，容易造成 OutOfMemoryError。
- 3) 基数排序时稳定的。[注：假定在待排序的记录序列中，存在多个具有相同的关键字的记录，若经过排序，这些记录的相对次序保持不变，即在原序列中， $r[i]=r[j]$ ，且 $r[i]$ 在 $r[j]$ 之前，而在排序后的序列中， $r[i]$ 仍在 $r[j]$ 之前，则称这种排序算法是稳定的；否则称为不稳定的]
- 4) 有负数的数组，我们不用基数排序来进行排序，如果要支持负数，参考：<https://code.i-harness.com/zh-CN/q/e98fa9>

7.12 常用排序算法总结和对比

7.12.1 一张排序算法的比较图



排序算法	平均时间复杂度	最好情况	最坏情况	空间复杂度	排序方式	稳定性
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	In-place	不稳定
插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
希尔排序	$O(n \log n)$	$O(n \log^2 n)$	$O(n \log^2 n)$	$O(1)$	In-place	不稳定
归并排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Out-place	稳定
快速排序	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	In-place	不稳定
堆排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	In-place	不稳定
计数排序	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(k)$	Out-place	稳定
桶排序	$O(n + k)$	$O(n + k)$	$O(n^2)$	$O(n + k)$	Out-place	稳定
基数排序	$O(n \times k)$	$O(n \times k)$	$O(n \times k)$	$O(n + k)$	Out-place	稳定

7.12.2 相关术语解释：

- 1) 稳定：如果 a 原本在 b 前面，而 $a=b$ ，排序之后 a 仍然在 b 的前面；
- 2) 不稳定：如果 a 原本在 b 的前面，而 $a=b$ ，排序之后 a 可能会出现在 b 的后面；
- 3) 内排序：所有排序操作都在内存中完成；
- 4) 外排序：由于数据太大，因此把数据放在磁盘中，而排序通过磁盘和内存的数据传输才能进行；

- 5) 时间复杂度：一个算法执行所耗费的时间。
- 6) 空间复杂度：运行完一个程序所需内存的大小。
- 7) n : 数据规模
- 8) k : “桶”的个数
- 9) In-place: 不占用额外内存
- 10) Out-place: 占用额外内存



第 8 章 查找算法

8.1 查找算法介绍

在 java 中，我们常用的查找有四种：

- 1) 顺序(线性)查找
- 2) 二分查找/折半查找
- 3) 插值查找
- 4) 斐波那契查找

8.2 线性查找算法

有一个数列： {1,8, 10, 89, 1000, 1234}，判断数列中是否包含此名称【顺序查找】 要求：如果找到了，就提示找到，并给出下标值。

代码实现：

```
package com.atguigu.search;

public class SeqSearch {

    public static void main(String[] args) {
        int arr[] = { 1, 9, 11, -1, 34, 89 };// 没有序的数组
        int index = seqSearch(arr, -11);
        if(index == -1) {
            System.out.println("没有找到到");
        } else {
            System.out.println("找到， 下标为=" + index);
        }
    }
}
```



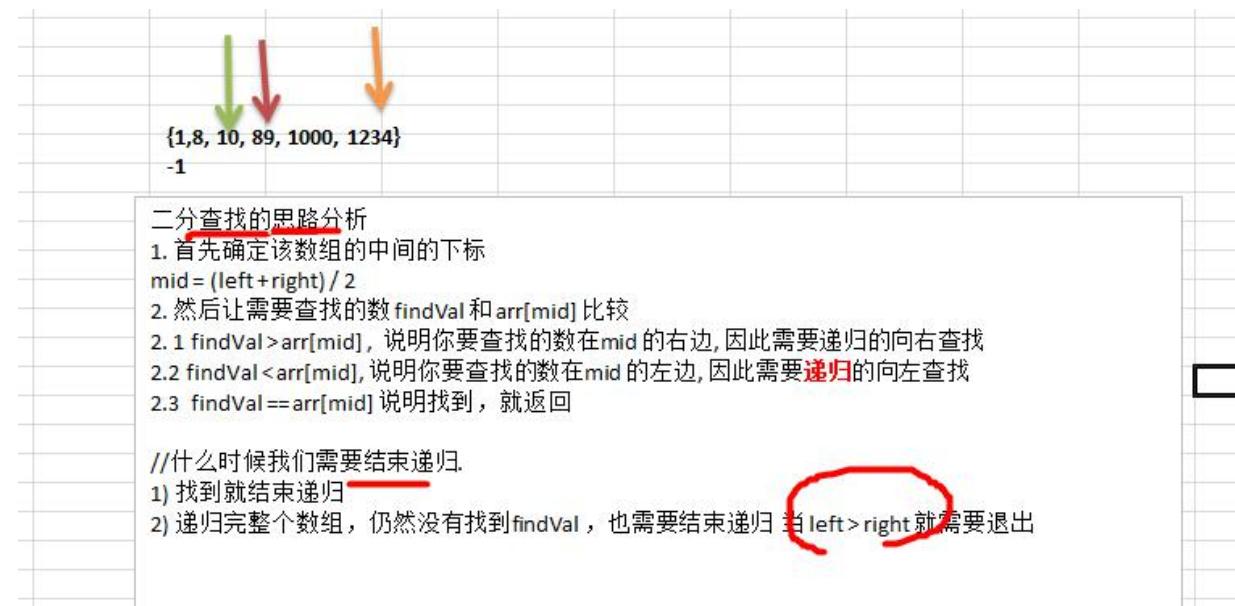
```
/**  
 * 这里我们实现的线性查找是找到一个满足条件的值，就返回  
 * @param arr  
 * @param value  
 * @return  
 */  
  
public static int seqSearch(int[] arr, int value) {  
    // 线性查找是逐一比对，发现有相同值，就返回下标  
    for (int i = 0; i < arr.length; i++) {  
        if(arr[i] == value) {  
            return i;  
        }  
    }  
    return -1;  
}  
  
}
```

8.3 二分查找算法

8.3.1 二分查找：

请对一个有序数组进行二分查找 `{1,8,10,89,1000,1234}`，输入一个数看看该数组是否存在此数，并且求出下标，如果没有就提示“没有这个数”。

8.3.2 二分查找算法的思路



8.3.3 二分查找的代码

说明：增加了找到所有的满足条件的元素下标：

课后思考题： {1,8, 10, 89, 1000, 1000, 1234} 当一个有序数组中，有多个相同的数值时，如何将所有的数值都查找到，比如这里的 1000.

```
package com.atguigu.search;

import java.util.ArrayList;
import java.util.List;

//注意：使用二分查找的前提是 该数组是有序的.

public class BinarySearch {

    public static void main(String[] args) {
        int arr[] = { 1, 8, 10, 89, 1000, 1000, 1234 };
```



```
//  
//    int resIndex = binarySearch(arr, 0, arr.length - 1, 1000);  
//    System.out.println("resIndex=" + resIndex);  
  
List<Integer> resIndexList = binarySearch2(arr, 0, arr.length - 1, 1000);  
System.out.println("resIndexList=" + resIndexList);  
}  
  
// 二分查找算法  
/**  
 *  
 * @param arr  
 *      数组  
 * @param left  
 *      左边的索引  
 * @param right  
 *      右边的索引  
 * @param findVal  
 *      要查找的值  
 * @return 如果找到就返回下标, 如果没有找到, 就返回 -1  
 */  
public static int binarySearch(int[] arr, int left, int right, int findVal) {  
  
    // 当 left > right 时, 说明递归整个数组, 但是没有找到  
    if (left > right) {  
        return -1;
```



```
}

int mid = (left + right) / 2;

int midVal = arr[mid];

if (findVal > midVal) { // 向 右递归
    return binarySearch(arr, mid + 1, right, findVal);
} else if (findVal < midVal) { // 向左递归
    return binarySearch(arr, left, mid - 1, findVal);
} else {

    return mid;
}

}

//完成一个课后思考题:

/*
 * 课后思考题: {1,8, 10, 89, 1000, 1000, 1234} 当一个有序数组中,
 * 有多个相同的数值时, 如何将所有的数值都查找到, 比如这里的 1000
 *
 * 思路分析
 * 1. 在找到 mid 索引值, 不要马上返回
 * 2. 向 mid 索引值的左边扫描, 将所有满足 1000, 的元素的下标, 加入到集合 ArrayList
 * 3. 向 mid 索引值的右边扫描, 将所有满足 1000, 的元素的下标, 加入到集合 ArrayList
 * 4. 将 ArrayList 返回
*/
```



```
public static List<Integer> binarySearch2(int[] arr, int left, int right, int findVal) {  
  
    // 当 left > right 时，说明递归整个数组，但是没有找到  
    if (left > right) {  
        return new ArrayList<Integer>();  
    }  
    int mid = (left + right) / 2;  
    int midVal = arr[mid];  
  
    if (findVal > midVal) { // 向右递归  
        return binarySearch2(arr, mid + 1, right, findVal);  
    } else if (findVal < midVal) { // 向左递归  
        return binarySearch2(arr, left, mid - 1, findVal);  
    } else {  
        // * 思路分析  
        // * 1. 在找到 mid 索引值，不要马上返回  
        // * 2. 向 mid 索引值的左边扫描，将所有满足 1000, 的元素的下标，加入到集合 ArrayList  
        // * 3. 向 mid 索引值的右边扫描，将所有满足 1000, 的元素的下标，加入到集合 ArrayList  
        // * 4. 将 ArrayList 返回  
  
        List<Integer> resIndexlist = new ArrayList<Integer>();  
        // 向 mid 索引值的左边扫描，将所有满足 1000, 的元素的下标，加入到集合 ArrayList  
        int temp = mid - 1;  
        while(true) {  
            if (temp < 0 || arr[temp] != findVal) { // 退出  
                break;  
            }  
            resIndexlist.add(temp);  
            temp--;  
        }  
        // 向 mid 索引值的右边扫描，将所有满足 1000, 的元素的下标，加入到集合 ArrayList  
        temp = mid + 1;  
        while(true) {  
            if (temp >= arr.length || arr[temp] != findVal) { // 退出  
                break;  
            }  
            resIndexlist.add(temp);  
            temp++;  
        }  
        return resIndexlist;  
    }  
}
```



```
        break;

    }

    //否则，就 temp 放入到 resIndexlist
    resIndexlist.add(temp);
    temp -= 1; //temp 左移

}

resIndexlist.add(mid); //



//向 mid 索引值的右边扫描，将所有满足 1000, 的元素的下标，加入到集合 ArrayList
temp = mid + 1;
while(true) {
    if (temp > arr.length - 1 || arr[temp] != findVal) {//退出
        break;
    }

    //否则，就 temp 放入到 resIndexlist
    resIndexlist.add(temp);
    temp += 1; //temp 右移
}

return resIndexlist;
}

}
```

8.4 插值查找算法

1) 插值查找原理介绍:

插值查找算法类似于二分查找，不同的是插值查找每次从自适应 mid 处开始查找。

2) 将折半查找中的求 mid 索引的公式，low 表示左边索引 left, high 表示右边索引 right.

key 就是前面我们讲的 findVal

$$mid = \frac{low + high}{2} = low + \frac{1}{2}(high - low)$$
 改成 $mid = low + \frac{key - arr[low]}{arr[high] - arr[low]}(high - low)$

3) int mid = low + (high - low) * (key - arr[low]) / (arr[high] - arr[low]); /*插值索引*/

对应前面的代码公式：

```
int mid = left + (right - left) * (findVal - arr[left]) / (arr[right] - arr[left])
```

4) 举例说明插值查找算法 1-100 的数组

插值查找算法的举例说明

数组 arr = [1, 2, 3, ..., 100]

假如我们需要查找的值 1

使用二分查找的话，我们需要多次递归，才能找到 1

使用插值查找算法

```
int mid = left + (right - left) * (findVal - arr[left]) / (arr[right] - arr[left])
```

```
int mid = 0 + (99 - 0) * (1 - 1) / (100 - 1) = 0 + 99 * 0 / 99 = 0
```

比如我们查找的值 100

```
int mid = 0 + (99 - 0) * (100 - 1) / (100 - 1) = 0 + 99 * 99 / 99 = 0 + 99 = 99
```

8.4.1 插值查找应用案例：

请对一个有序数组进行插值查找 {1, 8, 10, 89, 1000, 1234}，输入一个数看看该数组是否存在此数，并且求出下标，如果没有就提示“没有这个数”。

代码实现：



```
package com.atguigu.search;

import java.util.Arrays;

public class InsertValueSearch {

    public static void main(String[] args) {

        // int [] arr = new int[100];
        // for(int i = 0; i < 100; i++) {
        //     arr[i] = i + 1;
        // }

        int arr[] = { 1, 8, 10, 89, 1000, 1000, 1234 };

        int index = insertValueSearch(arr, 0, arr.length - 1, 1234);
        //int index = binarySearch(arr, 0, arr.length, 1);
        System.out.println("index = " + index);

        //System.out.println(Arrays.toString(arr));
    }

    public static int binarySearch(int[] arr, int left, int right, int findVal) {
        System.out.println("二分查找被调用~");
        // 当 left > right 时，说明递归整个数组，但是没有找到
        if (left > right) {
```



```
        return -1;

    }

    int mid = (left + right) / 2;

    int midVal = arr[mid];

    if (findVal > midVal) { // 向右递归
        return binarySearch(arr, mid + 1, right, findVal);
    } else if (findVal < midVal) { // 向左递归
        return binarySearch(arr, left, mid - 1, findVal);
    } else {

        return mid;
    }

}

//编写插值查找算法
//说明：插值查找算法，也要求数组是有序的

/**
 *
 * @param arr 数组
 * @param left 左边索引
 * @param right 右边索引
 * @param findVal 查找值
 * @return 如果找到，就返回对应的下标，如果没有找到，返回-1
 */
```



```
public static int insertValueSearch(int[] arr, int left, int right, int findVal) {  
  
    System.out.println("插值查找次数~~");  
  
    //注意： findVal < arr[0] 和 findVal > arr[arr.length - 1] 必须需要  
    //否则我们得到的 mid 可能越界  
    if (left > right || findVal < arr[0] || findVal > arr[arr.length - 1]) {  
        return -1;  
    }  
  
    // 求出 mid, 自适应  
    int mid = left + (right - left) * (findVal - arr[left]) / (arr[right] - arr[left]);  
    int midVal = arr[mid];  
    if (findVal > midVal) { // 说明应该向右边递归  
        return insertValueSearch(arr, mid + 1, right, findVal);  
    } else if (findVal < midVal) { // 说明向左递归查找  
        return insertValueSearch(arr, left, mid - 1, findVal);  
    } else {  
        return mid;  
    }  
}  
}
```

8.4.2 插值查找注意事项：

- 1) 对于数据量较大，关键字分布比较均匀的查找表来说，采用插值查找，速度较快。
- 2) 关键字分布不均匀的情况下，该方法不一定比折半查找要好

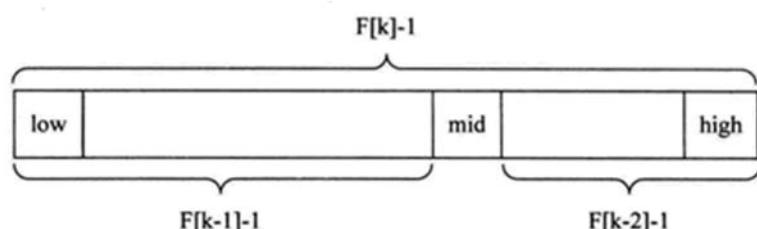
8.5 斐波那契(黄金分割法)查找算法

8.5.1 斐波那契(黄金分割法)查找基本介绍：

- 1) 黄金分割点是指把一条线段分割为两部分，使其中一部分与全长之比等于另一部分与这部分之比。取其前三位数字的近似值是 0.618。由于按此比例设计的造型十分美丽，因此称为黄金分割，也称为中外比。这是一个神奇的数字，会带来意向不大的效果。
- 2) 斐波那契数列 {1, 1, 2, 3, 5, 8, 13, 21, 34, 55} 发现斐波那契数列的两个相邻数 的比例，无限接近 黄金分割值 0.618

8.5.2 斐波那契(黄金分割法)原理：

斐波那契查找原理与前两种相似，仅仅改变了中间结点 (mid) 的位置，mid 不再是中间或插值得到，而是位于黄金分割点附近，即 $mid=low+F(k-1)-1$ (F 代表斐波那契数列)，如下图所示



➤ 对 $F(k-1)$ 的理解：

- 1) 由斐波那契数列 $F[k]=F[k-1]+F[k-2]$ 的性质，可以得到 $(F[k]-1) = (F[k-1]-1) + (F[k-2]-1) + 1$ 。该式说明：只要顺序表的长度为 $F[k]-1$ ，则可以将该表分成长度为 $F[k-1]-1$ 和 $F[k-2]-1$ 的两段，即如上图所示。从而中间



位置为 $mid=low+F(k-1)-1$

- 2) 类似的，每一子段也可以用相同的方式分割
- 3) 但顺序表长度 n 不一定刚好等于 $F[k]-1$ ，所以需要将原来的顺序表长度 n 增加至 $F[k]-1$ 。这里的 k 值只要能使得 $F[k]-1$ 恰好大于或等于 n 即可，由以下代码得到，顺序表长度增加后，新增的位置（从 $n+1$ 到 $F[k]-1$ 位置），都赋为 n 位置的值即可。

```
while(n>fib(k)-1)
```

```
    k++;
```

8.5.3 斐波那契查找应用案例：

请对一个**有序数组**进行斐波那契查找 {1,8,10,89,1000,1234}，输入一个数看看该数组是否存在此数，并且求出下标，如果没有就提示“没有这个数”。

代码实现：

```
package com.atguigu.search;

import java.util.Arrays;

public class FibonacciSearch {

    public static int maxSize = 20;

    public static void main(String[] args) {
        int [] arr = {1,8,10,89,1000,1234};

        System.out.println("index=" + fibSearch(arr, 189));// 0
    }

    public static int fibSearch(int[] arr, int target) {
        if (arr == null || arr.length == 0) {
            return -1;
        }
        int len = arr.length;
        int f1 = 0;
        int f2 = 1;
        int f3 = f1 + f2;
        int index = -1;
        while (f3 < len) {
            if (target < arr[f3]) {
                index = f2;
                break;
            } else if (target > arr[f3]) {
                f1 = f2;
                f2 = f3;
                f3 = f1 + f2;
            } else {
                index = f3;
                break;
            }
        }
        return index;
    }
}
```



```
}
```

```
//因为后面我们 mid=low+F(k-1)-1, 需要使用到斐波那契数列, 因此我们需要先获取到一个斐波那契数列  
//非递归方法得到一个斐波那契数列
```

```
public static int[] fib() {  
    int[] f = new int[maxSize];  
    f[0] = 1;  
    f[1] = 1;  
    for (int i = 2; i < maxSize; i++) {  
        f[i] = f[i - 1] + f[i - 2];  
    }  
    return f;  
}
```

```
//编写斐波那契查找算法
```

```
//使用非递归的方式编写算法
```

```
/**  
 *  
 * @param a 数组  
 * @param key 我们需要查找的关键码(值)  
 * @return 返回对应的下标, 如果没有-1  
 */
```

```
public static int fibSearch(int[] a, int key) {  
    int low = 0;  
    int high = a.length - 1;  
    int k = 0; //表示斐波那契分割数值的下标
```



```
int mid = 0; //存放 mid 值  
int f[] = fib(); //获取到斐波那契数列  
//获取到斐波那契分割数值的下标  
while(high > f[k] - 1) {  
    k++;  
}  
  
//因为 f[k] 值 可能大于 a 的 长度, 因此我们需要使用 Arrays 类, 构造一个新的数组, 并指向 temp[]  
//不足的部分会使用 0 填充  
int[] temp = Arrays.copyOf(a, f[k]);  
//实际上需求使用 a 数组最后的数填充 temp  
//举例:  
//temp = {1,8, 10, 89, 1000, 1234, 0, 0}  => {1,8, 10, 89, 1000, 1234, 1234, 1234,}  
for(int i = high + 1; i < temp.length; i++) {  
    temp[i] = a[high];  
}  
  
// 使用 while 来循环处理, 找到我们的数 key  
while (low <= high) { // 只要这个条件满足, 就可以找  
    mid = low + f[k - 1] - 1;  
    if(key < temp[mid]) { //我们应该继续向数组的前面查找(左边)  
        high = mid - 1;  
        //为甚是 k--  
        //说明  
        //1. 全部元素 = 前面的元素 + 后边元素  
        //2. f[k] = f[k-1] + f[k-2]  
        //因为 前面有 f[k-1]个元素,所以可以继续拆分 f[k-1] = f[k-2] + f[k-3]
```



```
//即 在 f[k-1] 的前面继续查找 k--  
//即下次循环 mid = f[k-1]-1  
k--;  
} else if ( key > temp[mid]) { // 我们应该继续向数组的后面查找(右边)  
    low = mid + 1;  
    //为什么是 k -=2  
    //说明  
    //1. 全部元素 = 前面的元素 + 后边元素  
    //2. f[k] = f[k-1] + f[k-2]  
    //3. 因为后面我们有 f[k-2] 所以可以继续拆分 f[k-1] = f[k-3] + f[k-4]  
    //4. 即在 f[k-2] 的前面进行查找 k -=2  
    //5. 即下次循环 mid = f[k - 1 - 2] - 1  
    k -= 2;  
} else { //找到  
    //需要确定，返回的是哪个下标  
    if(mid <= high) {  
        return mid;  
    } else {  
        return high;  
    }  
}  
return -1;  
}  
}
```

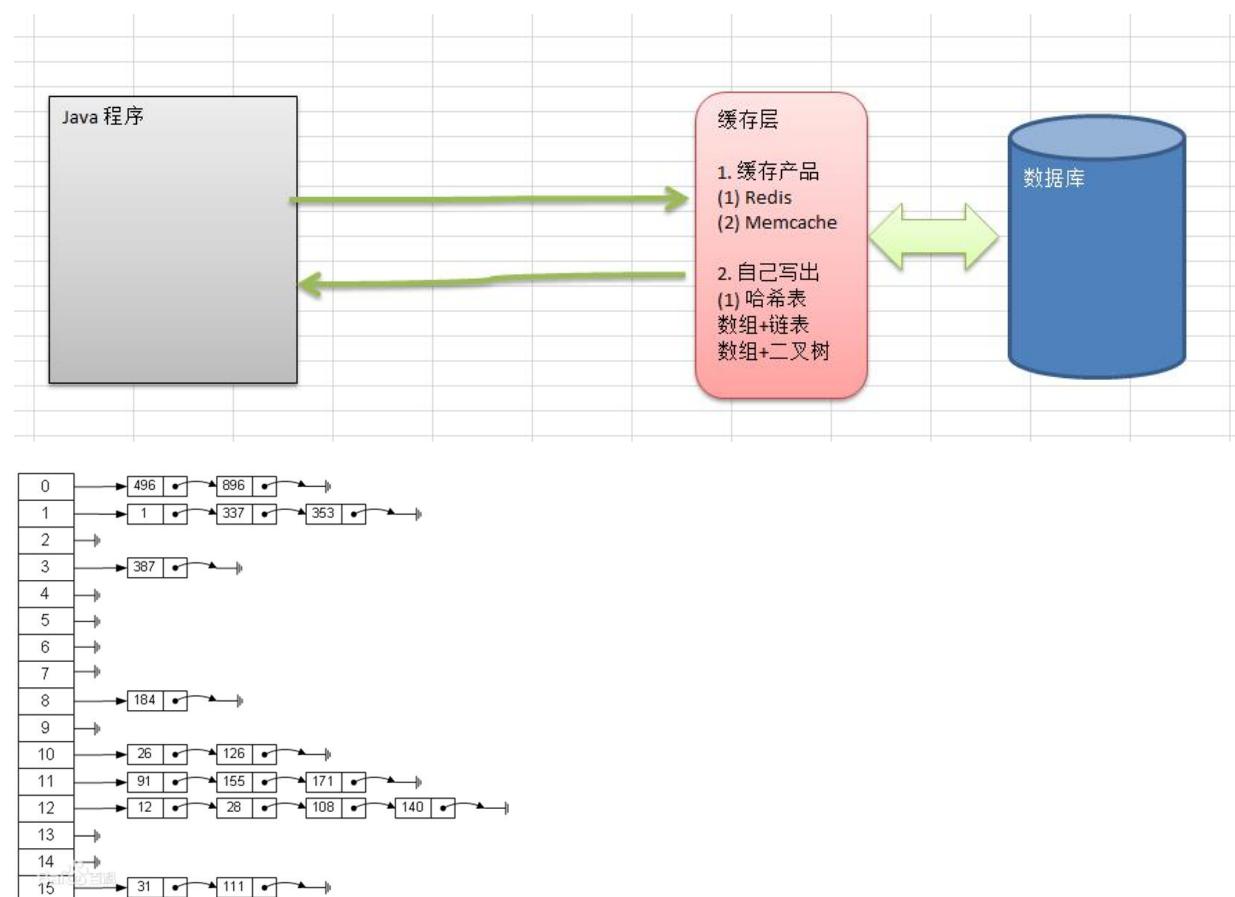
第 9 章 哈希表

9.1 哈希表(散列)-Google 上机题

- 1) 看一个实际需求, google 公司的一个上机题:
- 2) 有一个公司,当有新的员工来报道时,要求将该员工的信息加入(id,性别,年龄,住址..),当输入该员工的 id 时,要求查找到该员工的 所有信息.
- 3) 要求: 不使用数据库,尽量节省内存,速度越快越好=>哈希表(散列)

9.2 哈希表的基本介绍

散列表 (Hash table, 也叫哈希表) , 是根据关键码值(Key value)而直接进行访问的数据结构。也就是说, 它通过把关键码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, 存放记录的数组叫做散列表。

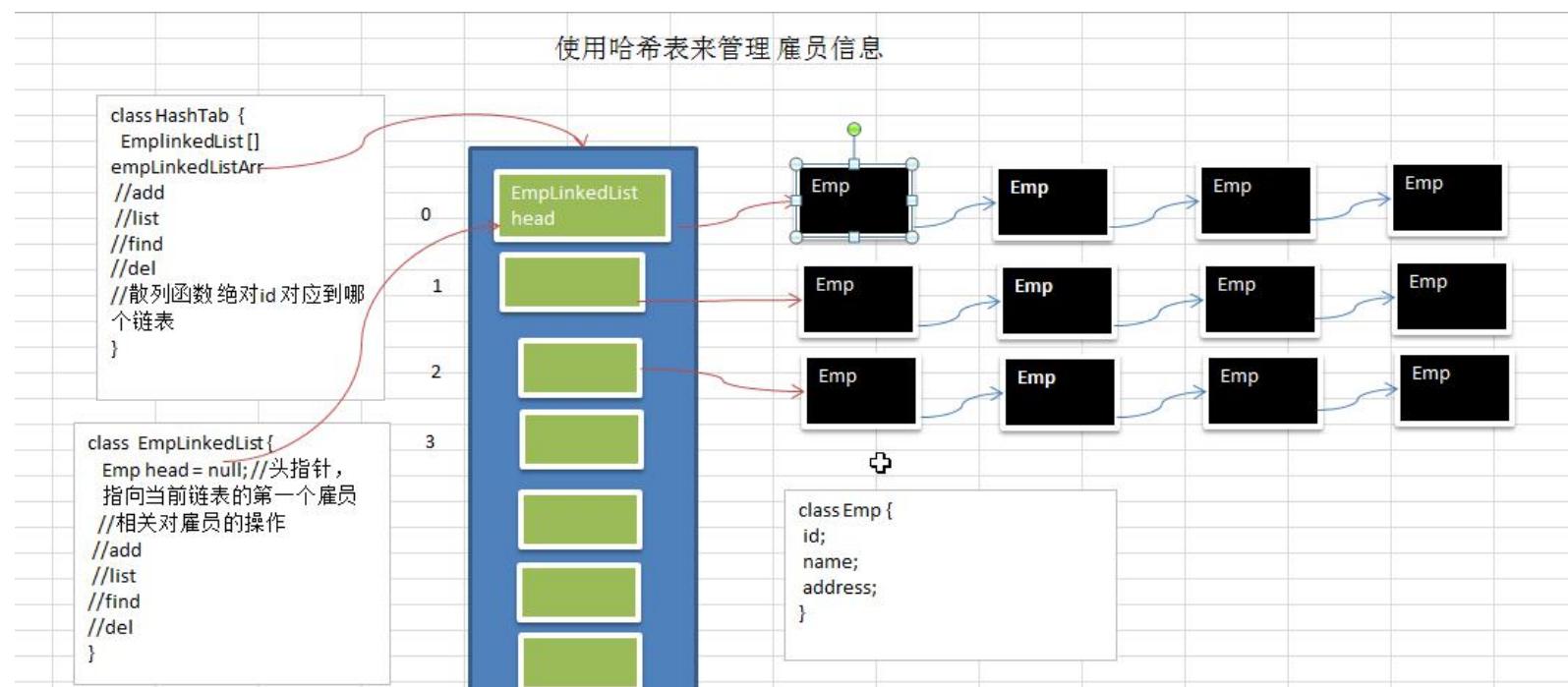


9.3 google 公司的一个上机题:

有一个公司,当有新的员工来报道时,要求将该员工的信息加入(id,性别,年龄,名字,住址..),当输入该员工的 id 时,要求查找到该员工的 所有信息.

要求:

- 1) 不使用数据库,,速度越快越好=>哈希表(散列)
- 2) 添加时, 保证按照 id 从低到高插入 [课后思考: 如果 id 不是从低到高插入, 但要求各条链表仍是从低到高, 怎么解决?]
- 3) 使用链表来实现哈希表, 该链表不带表头[即: 链表的第一个结点就存放雇员信息]
- 4) 思路分析并画出示意图



5) 代码实现

```
package com.atguigu.hashtab;
```



```
import java.util.Scanner;

public class HashTabDemo {

    public static void main(String[] args) {

        //创建哈希表
        HashTab hashTab = new HashTab(7);

        //写一个简单的菜单
        String key = "";
        Scanner scanner = new Scanner(System.in);
        while(true) {
            System.out.println("add: 添加雇员");
            System.out.println("list: 显示雇员");
            System.out.println("find: 查找雇员");
            System.out.println("exit: 退出系统");

            key = scanner.next();
            switch (key) {
                case "add":
                    System.out.println("输入 id");
                    int id = scanner.nextInt();
                    System.out.println("输入名字");
                    String name = scanner.next();
                    //创建 雇员

```



```
Emp emp = new Emp(id, name);
hashTab.add(emp);
break;
case "list":
    hashTab.list();
    break;
case "find":
    System.out.println("请输入要查找的 id");
    id = scanner.nextInt();
    hashTab.findEmpById(id);
    break;
case "exit":
    scanner.close();
    System.exit(0);
default:
    break;
}
}

}

//创建 HashTab 管理多条链表
class HashTab {
    private EmpLinkedList[] empLinkedListArray;
```



```
private int size; //表示有多少条链表

//构造器
public HashTab(int size) {
    this.size = size;
    //初始化 empLinkedListArray
    empLinkedListArray = new EmpLinkedList[size];
    //? 留一个坑, 这时不要分别初始化每个链表
    for(int i = 0; i < size; i++) {
        empLinkedListArray[i] = new EmpLinkedList();
    }
}

//添加雇员
public void add(Emp emp) {
    //根据员工的 id ,得到该员工应当添加到哪条链表
    int empLinkedListNO = hashFun(emp.id);
    //将 emp 添加到对应的链表中
    empLinkedListArray[empLinkedListNO].add(emp);
}

//遍历所有的链表,遍历 hashtab
public void list() {
    for(int i = 0; i < size; i++) {
        empLinkedListArray[i].list(i);
    }
}
```



```
}
```

```
//根据输入的 id,查找雇员
```

```
public void findEmpById(int id) {
```

```
    //使用散列函数确定到哪条链表查找
```

```
    int empLinkedListNO = hashFun(id);
```

```
    Emp emp = empLinkedListArray[empLinkedListNO].findEmpById(id);
```

```
    if(emp != null) { //找到
```

```
        System.out.printf("在第%d 条链表中找到 雇员 id = %d\n", (empLinkedListNO + 1), id);
```

```
    } else {
```

```
        System.out.println("在哈希表中， 没有找到该雇员~");
```

```
}
```

```
}
```

```
//编写散列函数, 使用一个简单取模法
```

```
public int hashFun(int id) {
```

```
    return id % size;
```

```
}
```

```
}
```

```
//表示一个雇员
```

```
class Emp {
```

```
    public int id;
```

```
    public String name;
```



```
public Emp next; //next 默认为 null  
public Emp(int id, String name) {  
    super();  
    this.id = id;  
    this.name = name;  
}  
  
//创建 EmpLinkedList ,表示链表  
class EmpLinkedList {  
    //头指针, 执行第一个 Emp,因此我们这个链表的 head 是直接指向第一个 Emp  
    private Emp head; //默认 null  
  
    //添加雇员到链表  
    //说明  
    //1. 假定, 当添加雇员时, id 是自增长, 即 id 的分配总是从小到大  
    // 因此我们将该雇员直接加入到本链表的最后即可  
    public void add(Emp emp) {  
        //如果是添加第一个雇员  
        if(head == null) {  
            head = emp;  
            return;  
        }  
        //如果不是第一个雇员, 则使用一个辅助的指针, 帮助定位到最后  
        Emp curEmp = head;  
        while(true) {
```



```
if(curEmp.next == null) {//说明到链表最后
    break;
}
curEmp = curEmp.next; //后移
}

//退出时直接将 emp 加入链表
curEmp.next = emp;
}

//遍历链表的雇员信息
public void list(int no) {
    if(head == null) { //说明链表为空
        System.out.println("第 "+(no+1)+" 链表为空");
        return;
    }
    System.out.print("第 "+(no+1)+" 链表的信息为");
    Emp curEmp = head; //辅助指针
    while(true) {
        System.out.printf(" => id=%d name=%s\t", curEmp.id, curEmp.name);
        if(curEmp.next == null) {//说明 curEmp 已经是最后结点
            break;
        }
        curEmp = curEmp.next; //后移， 遍历
    }
    System.out.println();
}
```



```
//根据 id 查找雇员
//如果查找到，就返回 Emp, 如果没有找到，就返回 null
public Emp findEmpById(int id) {
    //判断链表是否为空
    if(head == null) {
        System.out.println("链表为空");
        return null;
    }
    //辅助指针
    Emp curEmp = head;
    while(true) {
        if(curEmp.id == id) {//找到
            break;//这时 curEmp 就指向要查找的雇员
        }
        //退出
        if(curEmp.next == null) {//说明遍历当前链表没有找到该雇员
            curEmp = null;
            break;
        }
        curEmp = curEmp.next;//以后
    }

    return curEmp;
}
```



}

第 10 章 树结构的基础部分

10.1 二叉树

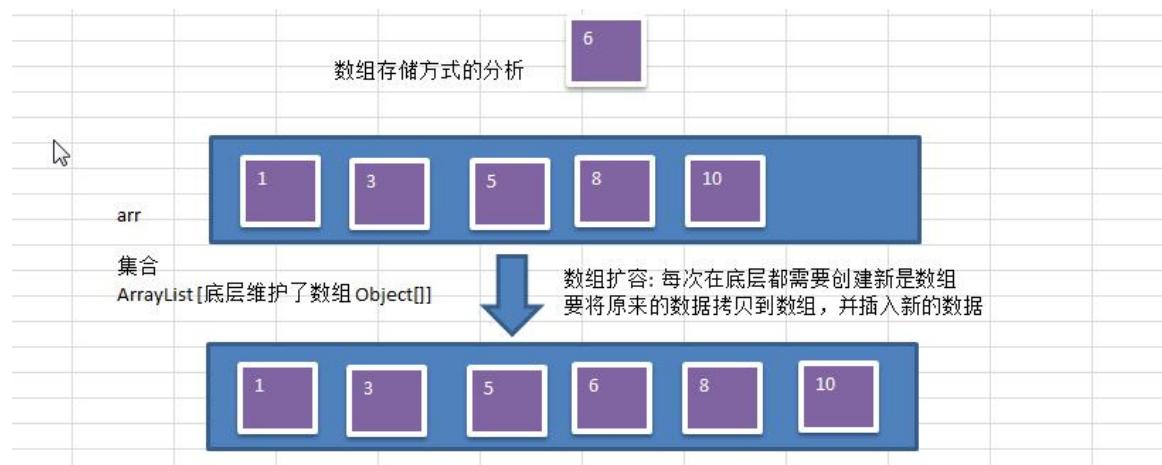
10.1.1 为什么需要树这种数据结构

1) 数组存储方式的分析

优点：通过下标方式访问元素，速度快。对于有序数组，还可使用二分查找提高检索速度。

缺点：如果要检索具体某个值，或者插入值(按一定顺序)会整体移动，效率较低 [示意图]

画出操作示意图：

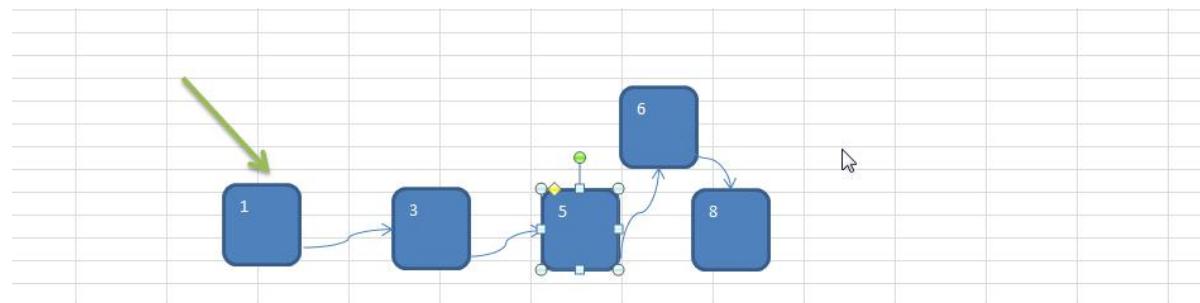


2) 链式存储方式的分析

优点：在一定程度上对数组存储方式有优化(比如：插入一个数值节点，只需要将插入节点，链接到链表中即可，删除效率也很好)。

缺点：在进行检索时，效率仍然较低，比如(检索某个值，需要从头节点开始遍历) 【示意图】

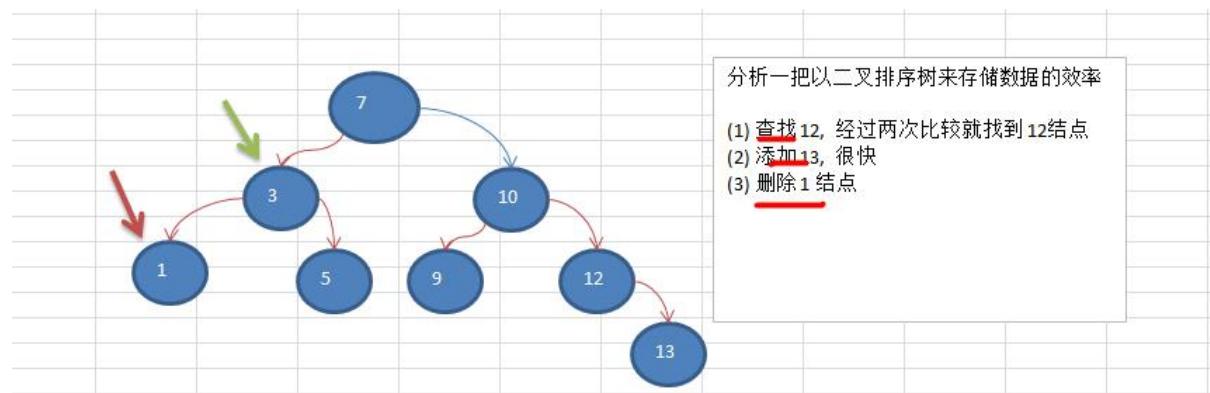
操作示意图：



3) 树存储方式的分析

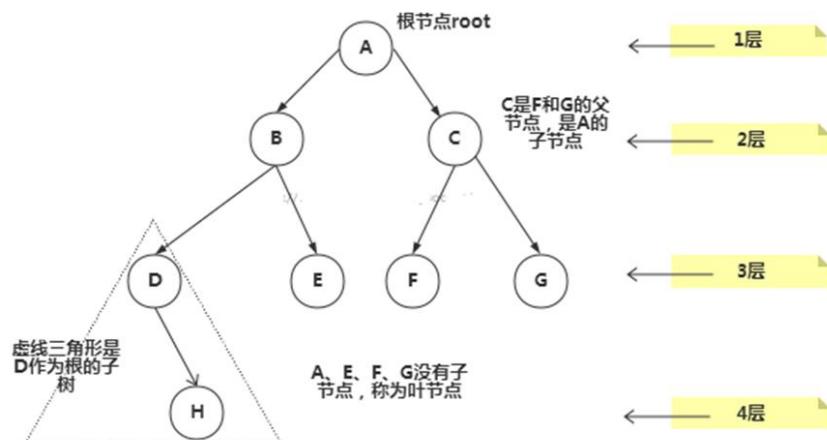
能提高数据存储，读取的效率，比如利用二叉排序树(Binary Sort Tree)，既可以保证数据的检索速度，同时也可保证数据的插入，删除，修改的速度。【示意图,后面详讲】

案例: [7, 3, 10, 1, 5, 9, 12]



10.1.2 树示意图

树的常用术语



树的常用术语(结合示意图理解):

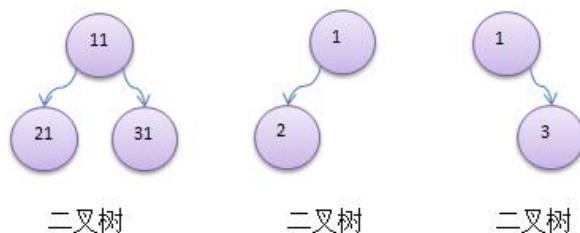
- 1) 节点
- 2) 根节点
- 3) 父节点
- 4) 子节点
- 5) 叶子节点 (没有子节点的节点)
- 6) 节点的权(节点值)
- 7) 路径(从 root 节点找到该节点的路线)
- 8) 层
- 9) 子树
- 10) 树的高度(最大层数)
- 11) 森林 :多颗子树构成森林

10.1.3 二叉树的概念

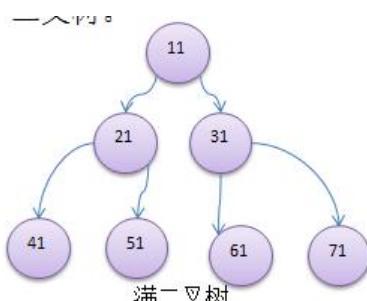
1) 树有很多种，每个节点最多只能有两个子节点的一种形式称为二叉树。

2) 二叉树的子节点分为左节点和右节点

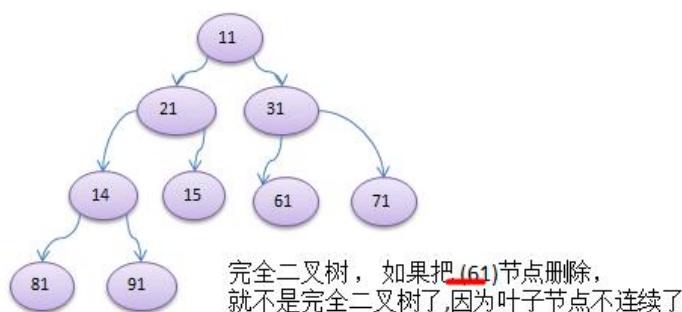
3) 示意图



4) 如果该二叉树的所有叶子节点都在最后一层，并且结点总数= $2^n - 1$, n 为层数，则我们称为满二叉树。



5) 如果该二叉树的所有叶子节点都在最后一层或者倒数第二层，而且最后一层的叶子节点在左边连续，倒数第二层的叶子节点在右边连续，我们称为完全二叉树



10.1.4 二叉树遍历的说明

使用前序，中序和后序对下面的二叉树进行遍历。

1) 前序遍历：先输出父节点，再遍历左子树和右子树

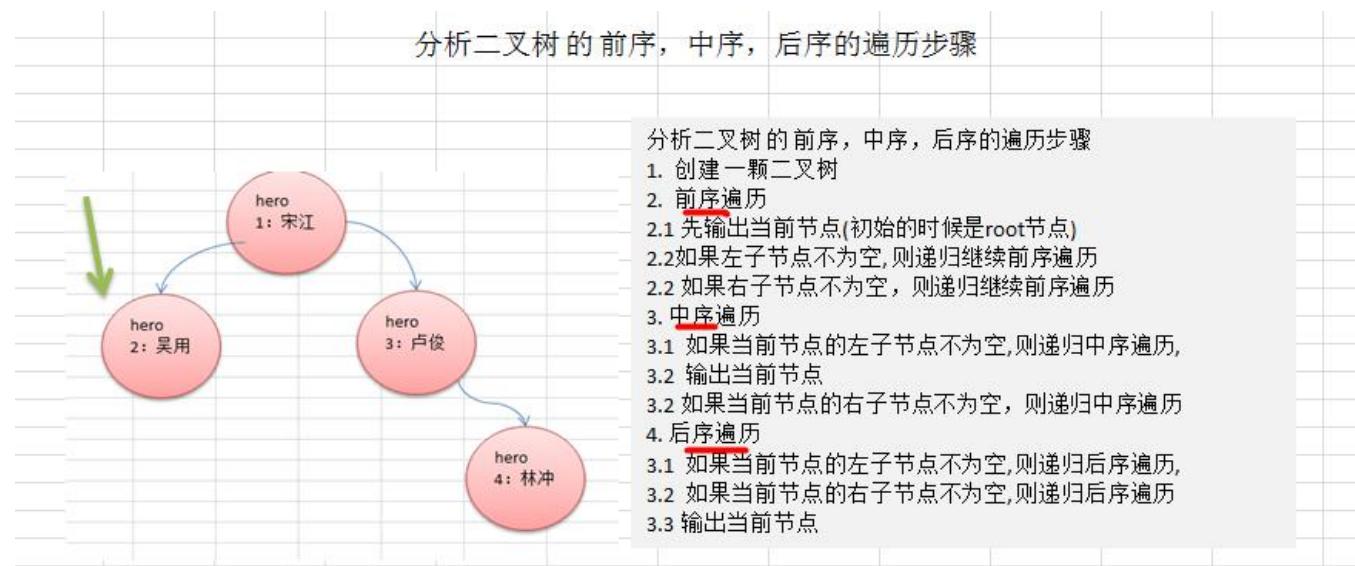
2) 中序遍历：先遍历左子树，再输出父节点，再遍历右子树

3) 后序遍历: 先遍历左子树, 再遍历右子树, 最后输出父节点

4) 小结: 看输出父节点的顺序, 就确定是前序, 中序还是后序

10.1.5 二叉树遍历应用实例(前序,中序,后序)

➤ 应用实例的说明和思路



➤ 代码实现

```
package com.atguigu.tree;

public class BinaryTreeDemo {

    public static void main(String[] args) {
        //先需要创建一颗二叉树
        BinaryTree binaryTree = new BinaryTree();
        //创建需要的结点
        HeroNode root = new HeroNode(1, "宋江");
        HeroNode node2 = new HeroNode(2, "吴用");
        HeroNode node3 = new HeroNode(3, "卢俊义");
    }
}
```



```
HeroNode node4 = new HeroNode(4, "林冲");
HeroNode node5 = new HeroNode(5, "关胜");

//说明，我们先手动创建该二叉树，后面我们学习递归的方式创建二叉树
root.setLeft(node2);
root.setRight(node3);
node3.setRight(node4);
node3.setLeft(node5);
binaryTree.setRoot(root);

//测试
System.out.println("前序遍历"); // 1,2,3,5,4
binaryTree.preOrder();

//测试
System.out.println("中序遍历");
binaryTree.infixOrder(); // 2,1,5,3,4
//

System.out.println("后序遍历");
binaryTree.postOrder(); // 2,5,4,3,1

}

}

//定义 BinaryTree 二叉树
```



```
class BinaryTree {  
    private HeroNode root;  
  
    public void setRoot(HeroNode root) {  
        this.root = root;  
    }  
  
    //前序遍历  
    public void preOrder() {  
        if(this.root != null) {  
            this.root.preOrder();  
        }else {  
            System.out.println("二叉树为空，无法遍历");  
        }  
    }  
  
    //中序遍历  
    public void infixOrder() {  
        if(this.root != null) {  
            this.root.infixOrder();  
        }else {  
            System.out.println("二叉树为空，无法遍历");  
        }  
    }  
    //后序遍历  
    public void postOrder() {  
    }
```



```
if(this.root != null) {  
    this.root.postOrder();  
}  
else {  
    System.out.println("二叉树为空，无法遍历");  
}  
}  
}  
  
//先创建 HeroNode 结点  
class HeroNode {  
    private int no;  
    private String name;  
    private HeroNode left; //默认 null  
    private HeroNode right; //默认 null  
    public HeroNode(int no, String name) {  
        this.no = no;  
        this.name = name;  
    }  
    public int getNo() {  
        return no;  
    }  
    public void setNo(int no) {  
        this.no = no;  
    }  
    public String getName() {  
        return name;  
    }  
}
```



```
}

public void setName(String name) {
    this.name = name;
}

public HeroNode getLeft() {
    return left;
}

public void setLeft(HeroNode left) {
    this.left = left;
}

public HeroNode getRight() {
    return right;
}

public void setRight(HeroNode right) {
    this.right = right;
}

@Override
public String toString() {
    return "HeroNode [no=" + no + ", name=" + name + "]";
}

//编写前序遍历的方法

public void preOrder() {
    System.out.println(this); //先输出父结点
    //递归向左子树前序遍历
    if(this.left != null) {
        this.left.preOrder();
    }
}
```



```
}

//递归向右子树前序遍历
if(this.right != null) {
    this.right.preOrder();
}

}

//中序遍历
public void infixOrder() {
    //递归向左子树中序遍历
    if(this.left != null) {
        this.left.infixOrder();
    }

    //输出父结点
    System.out.println(this);

    //递归向右子树中序遍历
    if(this.right != null) {
        this.right.infixOrder();
    }
}

//后序遍历
public void postOrder() {
    if(this.left != null) {
        this.left.postOrder();
    }

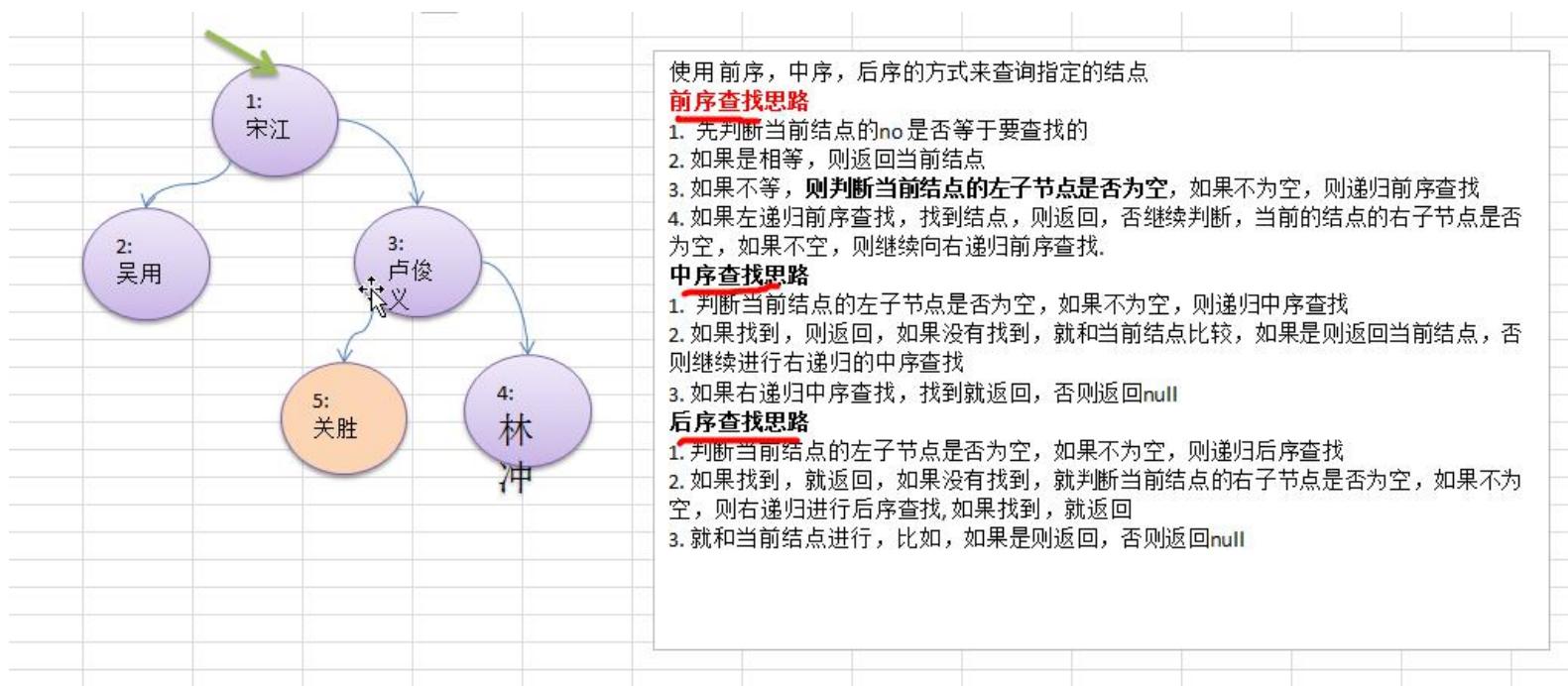
    if(this.right != null) {
        this.right.postOrder();
    }
}
```

```
        }  
        System.out.println(this);  
    }  
  
}  
  
//
```

10.1.6 二叉树-查找指定节点

要求

- 1) 请编写前序查找，中序查找和后序查找的方法。
- 2) 并分别使用三种查找方式，查找 heroNO = 5 的节点
- 3) 并分析各种查找方式，分别比较了多少次
- 4) 思路分析图解



- 5) 代码实现



```
package com.atguigu.tree;

public class BinaryTreeDemo {

    public static void main(String[] args) {
        //先需要创建一颗二叉树
        BinaryTree binaryTree = new BinaryTree();
        //创建需要的结点
        HeroNode root = new HeroNode(1, "宋江");
        HeroNode node2 = new HeroNode(2, "吴用");
        HeroNode node3 = new HeroNode(3, "卢俊义");
        HeroNode node4 = new HeroNode(4, "林冲");
        HeroNode node5 = new HeroNode(5, "关胜");

        //说明，我们先手动创建该二叉树，后面我们学习递归的方式创建二叉树
        root.setLeft(node2);
        root.setRight(node3);
        node3.setRight(node4);
        node3.setLeft(node5);
        binaryTree.setRoot(root);

        //测试
        System.out.println("前序遍历"); // 1,2,3,5,4
        binaryTree.preOrder();

        //测试
    }
}
```



```
System.out.println("中序遍历");
binaryTree.infixOrder() // 2,1,5,3,4
//

System.out.println("后序遍历");
binaryTree.postOrder() // 2,5,4,3,1

//前序遍历
//前序遍历的次数 : 4
//
System.out.println("前序遍历方式~~~");
// HeroNode resNode = binaryTree.preOrderSearch(5);
// if (resNode != null) {
//     System.out.printf("找到了, 信息为 no=%d name=%s", resNode.getNo(), resNode.getName());
// } else {
//     System.out.printf("没有找到 no = %d 的英雄", 5);
// }

//中序遍历查找
//中序遍历 3 次
//
System.out.println("中序遍历方式~~~");
// HeroNode resNode = binaryTree.infixOrderSearch(5);
// if (resNode != null) {
//     System.out.printf("找到了, 信息为 no=%d name=%s", resNode.getNo(), resNode.getName());
// } else {
//     System.out.printf("没有找到 no = %d 的英雄", 5);
// }
```



```
//后序遍历查找
//后序遍历查找的次数 2 次
System.out.println("后序遍历方式~~~");
HeroNode resNode = binaryTree.postOrderSearch(5);
if (resNode != null) {
    System.out.printf("找到了, 信息为 no=%d name=%s", resNode.getNo(), resNode.getName());
} else {
    System.out.printf("没有找到 no = %d 的英雄", 5);
}

}

}

//定义 BinaryTree 二叉树
class BinaryTree {
    private HeroNode root;

    public void setRoot(HeroNode root) {
        this.root = root;
    }

    //前序遍历
    public void preOrder() {
        if(this.root != null) {
            this.root.preOrder();
        }
    }
}
```



```
    }else {
        System.out.println("二叉树为空， 无法遍历");
    }
}

//中序遍历
public void infixOrder() {
    if(this.root != null) {
        this.root.infixOrder();
    }else {
        System.out.println("二叉树为空， 无法遍历");
    }
}

//后序遍历
public void postOrder() {
    if(this.root != null) {
        this.root.postOrder();
    }else {
        System.out.println("二叉树为空， 无法遍历");
    }
}

//前序遍历
public HeroNode preOrderSearch(int no) {
    if(root != null) {
        return root.preOrderSearch(no);
    }
}
```



```
    } else {
        return null;
    }
}

//中序遍历

public HeroNode infixOrderSearch(int no) {
    if(root != null) {
        return root.infixOrderSearch(no);
    }else {
        return null;
    }
}

//后序遍历

public HeroNode postOrderSearch(int no) {
    if(root != null) {
        return this.root.postOrderSearch(no);
    }else {
        return null;
    }
}

}

//先创建 HeroNode 结点

class HeroNode {
    private int no;
    private String name;
```



```
private HeroNode left; //默认 null  
private HeroNode right; //默认 null  
public HeroNode(int no, String name) {  
    this.no = no;  
    this.name = name;  
}  
public int getNo() {  
    return no;  
}  
public void setNo(int no) {  
    this.no = no;  
}  
public String getName() {  
    return name;  
}  
public void setName(String name) {  
    this.name = name;  
}  
public HeroNode getLeft() {  
    return left;  
}  
public void setLeft(HeroNode left) {  
    this.left = left;  
}  
public HeroNode getRight() {  
    return right;
```



```
}

public void setRight(HeroNode right) {
    this.right = right;
}

@Override
public String toString() {
    return "HeroNode [no=" + no + ", name=" + name + "]";
}

//编写前序遍历的方法
public void preOrder() {
    System.out.println(this); //先输出父结点
    //递归向左子树前序遍历
    if(this.left != null) {
        this.left.preOrder();
    }
    //递归向右子树前序遍历
    if(this.right != null) {
        this.right.preOrder();
    }
}

//中序遍历
public void infixOrder() {

    //递归向左子树中序遍历
    if(this.left != null) {
        this.left.infixOrder();
    }
}
```



```
}

//输出父结点
System.out.println(this);

//递归向右子树中序遍历
if(this.right != null) {
    this.right.infixOrder();
}

}

//后序遍历
public void postOrder() {
    if(this.left != null) {
        this.left.postOrder();
    }

    if(this.right != null) {
        this.right.postOrder();
    }

    System.out.println(this);
}

//前序遍历查找
/**
 *
 * @param no 查找 no
 * @return 如果找到就返回该 Node ,如果没有找到返回 null
 */
public HeroNode preOrderSearch(int no) {
```



```
System.out.println("进入前序遍历");

//比较当前结点是不是
if(this.no == no) {
    return this;
}

//1.则判断当前结点的左子节点是否为空，如果不为空，则递归前序查找
//2.如果左递归前序查找，找到结点，则返回
HeroNode resNode = null;
if(this.left != null) {
    resNode = this.left.preOrderSearch(no);
}
if(resNode != null) {//说明我们左子树找到
    return resNode;
}

//1.左递归前序查找，找到结点，则返回，否继续判断，
//2.当前的结点的右子节点是否为空，如果不空，则继续向右递归前序查找
if(this.right != null) {
    resNode = this.right.preOrderSearch(no);
}
return resNode;
}

//中序遍历查找
public HeroNode infixOrderSearch(int no) {
    //判断当前结点的左子节点是否为空，如果不为空，则递归中序查找
    HeroNode resNode = null;
```



```
if(this.left != null) {
    resNode = this.left.infixOrderSearch(no);
}

if(resNode != null) {
    return resNode;
}

System.out.println("进入中序查找");
//如果找到，则返回，如果没有找到，就和当前结点比较，如果是则返回当前结点
if(this.no == no) {
    return this;
}

//否则继续进行右递归的中序查找
if(this.right != null) {
    resNode = this.right.infixOrderSearch(no);
}

return resNode;
}

//后序遍历查找
public HeroNode postOrderSearch(int no) {

    //判断当前结点的左子节点是否为空，如果不为空，则递归后序查找
    HeroNode resNode = null;
    if(this.left != null) {
        resNode = this.left.postOrderSearch(no);
    }

    if(this.right != null) {
        resNode = this.right.postOrderSearch(no);
    }

    return resNode;
}
```



```
}

if(resNode != null) {//说明在左子树找到
    return resNode;
}

//如果左子树没有找到，则向右子树递归进行后序遍历查找
if(this.right != null) {
    resNode = this.right.postOrderSearch(no);
}

if(resNode != null) {
    return resNode;
}

System.out.println("进入后序查找");
//如果左右子树都没有找到，就比较当前结点是不是
if(this.no == no) {
    return this;
}

return resNode;
}

}
```

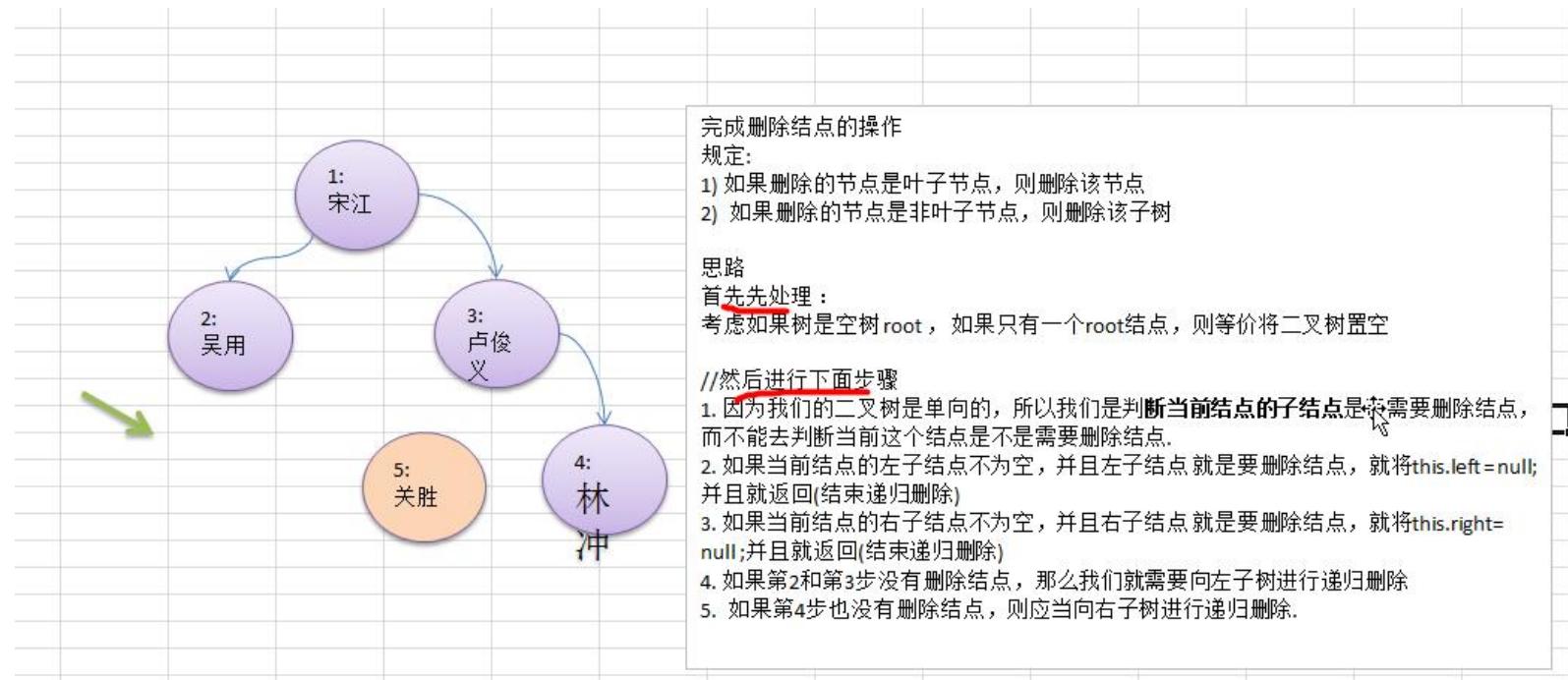
10.1.7 二叉树-删除节点

➤ 要求

- 1) 如果删除的节点是叶子节点，则删除该节点
- 2) 如果删除的节点是非叶子节点，则删除该子树.

3) 测试，删除掉 5 号叶子节点 和 3 号子树.

4) 完成删除思路分析



5) 代码实现

```
//HeroNode 类增加方法
```

```
//递归删除结点
```

```
//1.如果删除的节点是叶子节点，则删除该节点
//2.如果删除的节点是非叶子节点，则删除该子树
```

```
public void delNode(int no) {
```

```
//思路
```

```
/*
```

* 1. 因为我们的二叉树是单向的，所以我们是判断当前结点的子结点是否需要删除结点，而不能去判断当前这个结点是不是需要删除结点。



2. 如果当前结点的左子结点不为空，并且左子结点就是要删除结点，就将 this.left = null; 并且就返回(结束递归删除)
3. 如果当前结点的右子结点不为空，并且右子结点就是要删除结点，就将 this.right= null ;并且就返回(结束递归删除)
4. 如果第 2 和第 3 步没有删除结点，那么我们就需要向左子树进行递归删除
5. 如果第 4 步也没有删除结点，则应当向右子树进行递归删除.

*/

//2. 如果当前结点的左子结点不为空，并且左子结点就是要删除结点，就将 this.left = null; 并且就返回(结束递归删除)

```
if(this.left != null && this.left.no == no) {  
    this.left = null;  
    return;  
}
```

//3.如果当前结点的右子结点不为空，并且右子结点就是要删除结点，就将 this.right= null ;并且就返回(结束递归删除)

```
if(this.right != null && this.right.no == no) {  
    this.right = null;  
    return;  
}
```

//4.我们就需要向左子树进行递归删除

```
if(this.left != null) {  
    this.left.delNode(no);  
}
```

//5.则应当向右子树进行递归删除

```
if(this.right != null) {
```



```
this.right.delNode(no);  
}  
}  
}
```

//在 BinaryTree 类增加方法

//删除结点

```
public void delNode(int no) {  
    if(root != null) {  
        //如果只有一个 root 结点, 这里立即判断 root 是不是就是要删除结点  
        if(root.getNo() == no) {  
            root = null;  
        } else {  
            //递归删除  
            root.delNode(no);  
        }  
    } else {  
        System.out.println("空树, 不能删除~");  
    }  
}
```

//在 BinaryTreeDemo 类增加测试代码:

//测试一把删除结点

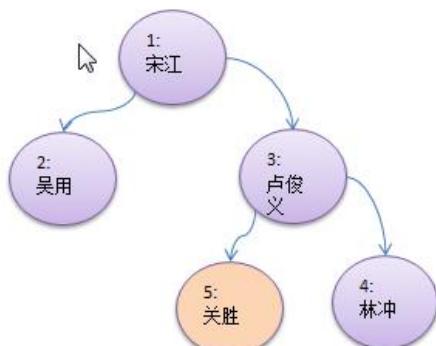
```
System.out.println("删除前,前序遍历");
```

```
binaryTree.preOrder(); // 1,2,3,5,4  
binaryTree.delNode(5);  
//binaryTree.delNode(3);  
System.out.println("删除后, 前序遍历");  
binaryTree.preOrder(); // 1,2,3,4
```

10.1.8 二叉树-删除节点

思考题(课后练习)

- 1) 如果要删除的节点是非叶子节点，现在我们不希望将该非叶子节点为根节点的子树删除，需要指定规则，假如规定如下：
- 2) 如果该非叶子节点 A 只有一个子节点 B，则子节点 B 替代节点 A
- 3) 如果该非叶子节点 A 有左子节点 B 和右子节点 C，则让左子节点 B 替代节点 A。
- 4) 请大家思考，如何完成该删除功能，老师给出提示。(课后练习)
- 5) 后面在讲解 二叉排序树时，在给大家讲解具体的删除方法

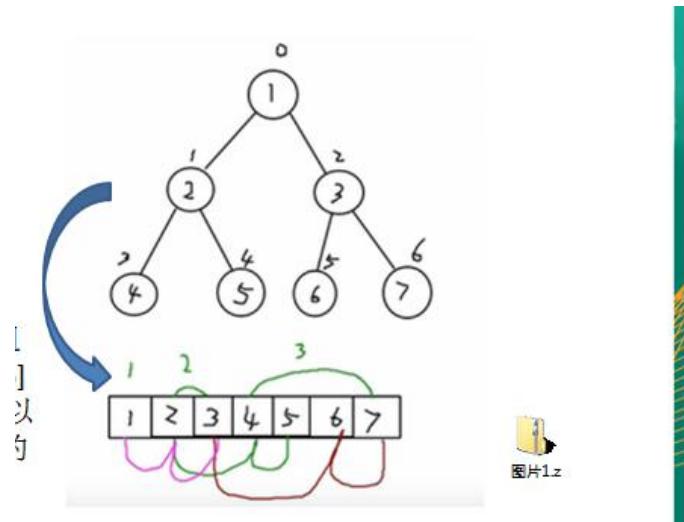


10.2 顺序存储二叉树

10.2.1 顺序存储二叉树的概念

➤ 基本说明

从数据存储来看，数组存储方式和树的存储方式可以相互转换，即数组可以转换成树，树也可以转换成数组，看右面的示意图。



➤ 要求：

- 1) 右图的二叉树的结点，要求以数组的方式来存放 $\text{arr} : [1, 2, 3, 4, 5, 6, 7]$
- 2) 要求在遍历数组 arr 时，仍然可以以前序遍历，中序遍历和后序遍历的方式完成结点的遍历

➤ 顺序存储二叉树的特点：

- 1) 顺序二叉树通常只考虑完全二叉树
- 2) 第 n 个元素的左子节点为 $2 * n + 1$
- 3) 第 n 个元素的右子节点为 $2 * n + 2$
- 4) 第 n 个元素的父节点为 $(n-1) / 2$
- 5) n ：表示二叉树中的第几个元素(按 0 开始编号如图所示)

10.2.2 顺序存储二叉树遍历



需求：给你一个数组 {1,2,3,4,5,6,7}，要求以二叉树前序遍历的方式进行遍历。 前序遍历的结果应当为 1,2,4,5,3,6,7

代码实现：

```
package com.atguigu.tree;

public class ArrBinaryTreeDemo {

    public static void main(String[] args) {
        int[] arr = { 1, 2, 3, 4, 5, 6, 7 };
        //创建一个 ArrBinaryTree
        ArrBinaryTree arrBinaryTree = new ArrBinaryTree(arr);
        arrBinaryTree.preOrder(); // 1,2,4,5,3,6,7
    }
}

//编写一个 ArrayBinaryTree, 实现顺序存储二叉树遍历

class ArrBinaryTree {
    private int[] arr;//存储数据结点的数组

    public ArrBinaryTree(int[] arr) {
        this.arr = arr;
    }
}
```



```
//重载 preOrder
public void preOrder() {
    this.preOrder(0);
}

//编写一个方法，完成顺序存储二叉树的前序遍历
/**
 *
 * @param index 数组的下标
 */
public void preOrder(int index) {
    //如果数组为空，或者 arr.length = 0
    if(arr == null || arr.length == 0) {
        System.out.println("数组为空，不能按照二叉树的前序遍历");
    }
    //输出当前这个元素
    System.out.println(arr[index]);
    //向左递归遍历
    if((index * 2 + 1) < arr.length) {
        preOrder(2 * index + 1 );
    }
    //向右递归遍历
    if((index * 2 + 2) < arr.length) {
        preOrder(2 * index + 2);
    }
}
```

{}

作业:

课后练习：请同学们完成对数组以二叉树中序，后序遍历方式的代码.

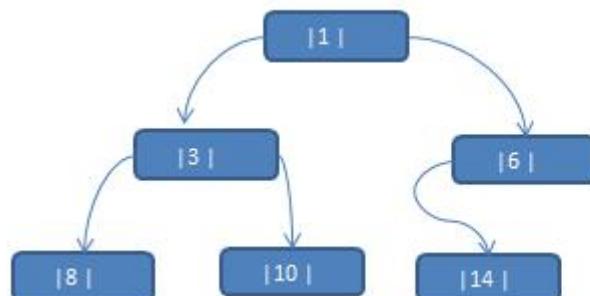
10.2.3 顺序存储二叉树应用实例

八大排序算法中的堆排序，就会使用到顺序存储二叉树，关于堆排序，我们放在<<树结构实际应用>> 章节讲解。

10.3 线索化二叉树

10.3.1 先看一个问题

将数列 {1, 3, 6, 8, 10, 14} 构建成一颗二叉树。 $n+1=7$



问题分析:

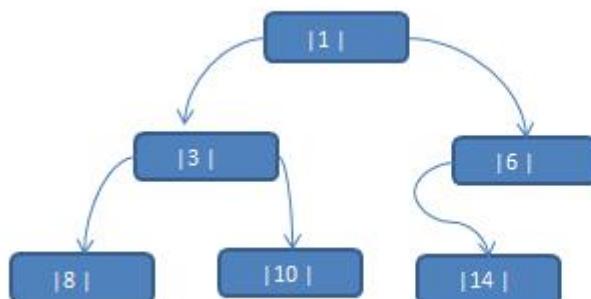
- 1) 当我们对上面的二叉树进行中序遍历时，数列为 {8, 3, 10, 1, 6, 14}
- 2) 但是 6, 8, 10, 14 这几个节点的 左右指针，并没有完全的利用上.
- 3) 如果我们希望充分的利用 各个节点的左右指针， 让各个节点可以指向自己的前后节点,怎么办?
- 4) 解决方案-线索二叉树

10.3.2 线索二叉树基本介绍

- 1) n 个结点的二叉链表中含有 $n+1$ 【公式 $2n-(n-1)=n+1$ 】 个空指针域。利用二叉链表中的空指针域，存放指向该结点在某种遍历次序下的前驱和后继结点的指针（这种附加的指针称为“线索”）
- 2) 这种加上了线索的二叉链表称为线索链表，相应的二叉树称为线索二叉树(Threaded Binary Tree)。根据线索性质的不同，线索二叉树可分为前序线索二叉树、中序线索二叉树和后序线索二叉树三种
- 3) 一个结点的前一个结点，称为前驱结点
- 4) 一个结点的后一个结点，称为后继结点

10.3.3 线索二叉树应用案例

应用案例说明：将下面的二叉树，进行中序线索二叉树。中序遍历的数列为 {8, 3, 10, 1, 14, 6}



思路分析： 中序遍历的结果： {8, 3, 10, 1, 14, 6}



➤ 说明: 当线索化二叉树后, Node 节点的 属性 left 和 right , 有如下情况:

- 1) left 指向的是左子树, 也可能是指向的前驱节点. 比如 ① 节点 left 指向的左子树, 而 ⑩ 节点的 left 指向的就是前驱节点.
- 2) right 指向的是右子树, 也可能是指向后继节点, 比如 ① 节点 right 指向的是右子树, 而⑩ 节点的 right 指向的是后继节点.

➤ 代码实现:

```
package com.atguigu.tree.threadedbinarytree;

import java.util.concurrent.SynchronousQueue;

public class ThreadedBinaryTreeDemo {

    public static void main(String[] args) {
        //测试一把中序线索二叉树的功能
        HeroNode root = new HeroNode(1, "tom");
        HeroNode node2 = new HeroNode(3, "jack");
        HeroNode node3 = new HeroNode(6, "smith");
        HeroNode node4 = new HeroNode(8, "mary");
    }
}
```



```
HeroNode node5 = new HeroNode(10, "king");
HeroNode node6 = new HeroNode(14, "dim");

//二叉树， 后面我们要递归创建，现在简单处理使用手动创建
root.setLeft(node2);
root.setRight(node3);
node2.setLeft(node4);
node2.setRight(node5);
node3.setLeft(node6);

//测试中序线索化
ThreadedBinaryTree threadedBinaryTree = new ThreadedBinaryTree();
threadedBinaryTree.setRoot(root);
threadedBinaryTree.threadedNodes();

//测试：以 10 号节点测试
HeroNode leftNode = node5.getLeft();
HeroNode rightNode = node5.getRight();
System.out.println("10 号结点的前驱结点是 = " + leftNode); //3
System.out.println("10 号结点的后继结点是 = " + rightNode); //1

//当线索化二叉树后，能在使用原来的遍历方法
//threadedBinaryTree.infixOrder();
System.out.println("使用线索化的方式遍历 线索化二叉树");
threadedBinaryTree.threadedList(); // 8, 3, 10, 1, 14, 6
```



```
}
```

```
}
```

```
//定义 ThreadedBinaryTree 实现了线索化功能的二叉树
```

```
class ThreadedBinaryTree {
```

```
    private HeroNode root;
```

```
//为了实现线索化，需要创建要给指向当前结点的前驱结点的指针
```

```
//在递归进行线索化时，pre 总是保留前一个结点
```

```
    private HeroNode pre = null;
```

```
    public void setRoot(HeroNode root) {
```

```
        this.root = root;
```

```
}
```

```
//重载一把 threadedNodes 方法
```

```
    public void threadedNodes() {
```

```
        this.threadedNodes(root);
```

```
}
```

```
//遍历线索化二叉树的方法
```

```
    public void threadedList() {
```



```
//定义一个变量，存储当前遍历的结点，从 root 开始  
  
HeroNode node = root;  
  
while(node != null) {  
    //循环的找到 leftType == 1 的结点，第一个找到就是 8 结点  
    //后面随着遍历而变化,因为当 leftType==1 时，说明该结点是按照线索化  
    //处理后的有效结点  
    while(node.getLeftType() == 0) {  
        node = node.getLeft();  
    }  
  
    //打印当前这个结点  
    System.out.println(node);  
    //如果当前结点的右指针指向的是后继结点,就一直输出  
    while(node.getRightType() == 1) {  
        //获取到当前结点的后继结点  
        node = node.getRight();  
        System.out.println(node);  
    }  
    //替换这个遍历的结点  
    node = node.getRight();  
  
}  
  
}  
  
//编写对二叉树进行中序线索化的方法  
/**
```



```
*  
* @param node 就是当前需要线索化的结点  
*/  
  
public void threadedNodes(HeroNode node) {  
  
    //如果 node==null, 不能线索化  
    if(node == null) {  
        return;  
    }  
  
    //((一)先线索化左子树  
    threadedNodes(node.getLeft());  
    //((二)线索化当前结点[有难度]  
  
    //处理当前结点的前驱结点  
    //以 8 结点来理解  
    //8 结点的.left = null , 8 结点的.leftType = 1  
    if(node.getLeft() == null) {  
        //让当前结点的左指针指向前驱结点  
        node.setLeft(pre);  
        //修改当前结点的左指针的类型,指向前驱结点  
        node.setLeftType(1);  
    }  
  
    //处理后继结点  
    if(pre != null && pre.getRight() == null) {
```



```
//让前驱结点的右指针指向当前结点
    pre.setRight(node);
    //修改前驱结点的右指针类型
    pre.setRightType(1);
}

//!!! 每处理一个结点后，让当前结点是下一个结点的前驱结点
pre = node;

//(三)在线索化右子树
threadedNodes(node.getRight());

}

//删除结点
public void delNode(int no) {
    if(root != null) {
        //如果只有一个 root 结点，这里立即判断 root 是不是就是要删除结点
        if(root.getNo() == no) {
            root = null;
        } else {
            //递归删除
            root.delNode(no);
        }
    } else{
        System.out.println("空树， 不能删除~");
    }
}
```



```
}

}

//前序遍历

public void preOrder() {
    if(this.root != null) {
        this.root.preOrder();
    }else {
        System.out.println("二叉树为空，无法遍历");
    }
}

//中序遍历

public void infixOrder() {
    if(this.root != null) {
        this.root.infixOrder();
    }else {
        System.out.println("二叉树为空，无法遍历");
    }
}

//后序遍历

public void postOrder() {
    if(this.root != null) {
        this.root.postOrder();
    }else {
        System.out.println("二叉树为空，无法遍历");
    }
}
```



```
}
```

```
//前序遍历
```

```
public HeroNode preOrderSearch(int no) {
```

```
    if(root != null) {
```

```
        return root.preOrderSearch(no);
```

```
    } else {
```

```
        return null;
```

```
}
```

```
}
```

```
//中序遍历
```

```
public HeroNode infixOrderSearch(int no) {
```

```
    if(root != null) {
```

```
        return root.infixOrderSearch(no);
```

```
    } else {
```

```
        return null;
```

```
}
```

```
}
```

```
//后序遍历
```

```
public HeroNode postOrderSearch(int no) {
```

```
    if(root != null) {
```

```
        return this.root.postOrderSearch(no);
```

```
    } else {
```

```
        return null;
```

```
}
```

```
}
```



```
}
```

```
//先创建 HeroNode 结点

class HeroNode {

    private int no;
    private String name;
    private HeroNode left; //默认 null
    private HeroNode right; //默认 null

    //说明
    //1. 如果 leftType == 0 表示指向的是左子树, 如果 1 则表示指向前驱结点
    //2. 如果 rightType == 0 表示指向是右子树, 如果 1 表示指向后继结点

    private int leftType;
    private int rightType;

    public int getLeftType() {
        return leftType;
    }

    public void setLeftType(int leftType) {
        this.leftType = leftType;
    }

    public int getRightType() {
        return rightType;
    }

    public void setRightType(int rightType) {
```



```
this.rightType = rightType;  
}  
  
public HeroNode(int no, String name) {  
    this.no = no;  
    this.name = name;  
}  
  
public int getNo() {  
    return no;  
}  
  
public void setNo(int no) {  
    this.no = no;  
}  
  
public String getName() {  
    return name;  
}  
  
public void setName(String name) {  
    this.name = name;  
}  
  
public HeroNode getLeft() {  
    return left;  
}  
  
public void setLeft(HeroNode left) {  
    this.left = left;  
}  
  
public HeroNode getRight() {  
    return right;  
}
```



```
}

public void setRight(HeroNode right) {
    this.right = right;
}

@Override
public String toString() {
    return "HeroNode [no=" + no + ", name=" + name + "]";
}

//递归删除结点
//1.如果删除的节点是叶子节点，则删除该节点
//2.如果删除的节点是非叶子节点，则删除该子树
public void delNode(int no) {

    //思路
    /*
     * 1. 因为我们的二叉树是单向的，所以我们是判断当前结点的子结点是否需要删除结点，而不能去判断
     * 当前这个结点是不是需要删除结点。
     *
     * 2. 如果当前结点的左子结点不为空，并且左子结点 就是要删除结点，就将 this.left = null; 并且就返回
     * (结束递归删除)
     *
     * 3. 如果当前结点的右子结点不为空，并且右子结点 就是要删除结点，就将 this.right= null ;并且就返回
     * (结束递归删除)
     *
     * 4. 如果第 2 和第 3 步没有删除结点，那么我们就需要向左子树进行递归删除
     *
     * 5. 如果第 4 步也没有删除结点，则应当向右子树进行递归删除.
     */

}
```



//2. 如果当前结点的左子结点不为空，并且左子结点就是要删除结点，就将 this.left = null; 并且就返回(结束递归删除)

```
if(this.left != null && this.left.no == no) {  
    this.left = null;  
    return;  
}
```

//3.如果当前结点的右子结点不为空，并且右子结点就是要删除结点，就将 this.right= null ;并且就返回(结束递归删除)

```
if(this.right != null && this.right.no == no) {  
    this.right = null;  
    return;  
}
```

//4.我们就需要向左子树进行递归删除

```
if(this.left != null) {  
    this.left.delNode(no);  
}
```

//5.则应当向右子树进行递归删除

```
if(this.right != null) {  
    this.right.delNode(no);  
}  
}
```

//编写前序遍历的方法

```
public void preOrder() {  
    System.out.println(this); //先输出父结点  
    //递归向左子树前序遍历
```



```
if(this.left != null) {  
    this.left.preOrder();  
}  
  
//递归向右子树前序遍历  
if(this.right != null) {  
    this.right.preOrder();  
}  
}  
  
//中序遍历  
public void infixOrder() {  
  
    //递归向左子树中序遍历  
    if(this.left != null) {  
        this.left.infixOrder();  
    }  
  
    //输出父结点  
    System.out.println(this);  
  
    //递归向右子树中序遍历  
    if(this.right != null) {  
        this.right.infixOrder();  
    }  
}  
  
//后序遍历  
public void postOrder() {  
    if(this.left != null) {  
        this.left.postOrder();  
    }
```



```
}

if(this.right != null) {
    this.right.postOrder();
}

System.out.println(this);

}

//前序遍历查找

/**
 *
 * @param no 查找 no
 * @return 如果找到就返回该 Node ,如果没有找到返回 null
 */

public HeroNode preOrderSearch(int no) {
    System.out.println("进入前序遍历");

    //比较当前结点是不是
    if(this.no == no) {
        return this;
    }

    //1.则判断当前结点的左子节点是否为空，如果不为空，则递归前序查找
    //2.如果左递归前序查找，找到结点，则返回
    HeroNode resNode = null;
    if(this.left != null) {
        resNode = this.left.preOrderSearch(no);
    }
    if(resNode != null) {//说明我们左子树找到
```



```
        return resNode;

    }

    //1.左递归前序查找，找到结点，则返回，否继续判断，

    //2.当前的结点的右子节点是否为空，如果不空，则继续向右递归前序查找
    if(this.right != null) {

        resNode = this.right.preOrderSearch(no);

    }

    return resNode;

}

//中序遍历查找

public HeroNode infixOrderSearch(int no) {

    //判断当前结点的左子节点是否为空，如果不为空，则递归中序查找
    HeroNode resNode = null;

    if(this.left != null) {

        resNode = this.left.infixOrderSearch(no);

    }

    if(resNode != null) {

        return resNode;

    }

    System.out.println("进入中序查找");

    //如果找到，则返回，如果没有找到，就和当前结点比较，如果是则返回当前结点
    if(this.no == no) {

        return this;

    }

    //否则继续进行右递归的中序查找
}
```



```
if(this.right != null) {  
    resNode = this.right.infixOrderSearch(no);  
}  
return resNode;  
  
}  
  
//后序遍历查找  
public HeroNode postOrderSearch(int no) {  
  
    //判断当前结点的左子节点是否为空，如果不为空，则递归后序查找  
    HeroNode resNode = null;  
    if(this.left != null) {  
        resNode = this.left.postOrderSearch(no);  
    }  
    if(resNode != null) {//说明在左子树找到  
        return resNode;  
    }  
  
    //如果左子树没有找到，则向右子树递归进行后序遍历查找  
    if(this.right != null) {  
        resNode = this.right.postOrderSearch(no);  
    }  
    if(resNode != null) {  
        return resNode;  
    }  
}
```



```
System.out.println("进入后序查找");

//如果左右子树都没有找到，就比较当前结点是不是
if(this.no == no) {

    return this;

}

return resNode;

}

}
```

10.3.4 遍历线索化二叉树

- 1) 说明：对前面的中序线索化的二叉树， 进行遍历
- 2) 分析：因为线索化后， **各个结点指向有变化，因此原来的遍历方式不能使用**，这时需要使用新的方式遍历线索化二叉树，各个节点可以通过线型方式遍历，因此无需使用递归方式，这样也提高了遍历的效率。 遍历的次序应当和中序遍历保持一致。
- 3) 代码：

```
//ThreadedBinaryTree 类

//遍历线索化二叉树的方法

public void threadedList() {

    //定义一个变量，存储当前遍历的结点，从 root 开始
    HeroNode node = root;

    while(node != null) {
```



```
//循环的找到 leftType == 1 的结点，第一个找到就是 8 结点
//后面随着遍历而变化,因为当 leftType==1 时，说明该结点是按照线索化
//处理后的有效结点
while(node.getLeftType() == 0) {
    node = node.getLeft();
}

//打印当前这个结点
System.out.println(node);
//如果当前结点的右指针指向的是后继结点,就一直输出
while(node.getRightType() == 1) {
    //获取到当前结点的后继结点
    node = node.getRight();
    System.out.println(node);
}

//替换这个遍历的结点
node = node.getRight();

}
```

10.3.5 线索化二叉树的课后作业:

我这里讲解了中序线索化二叉树，前序线索化二叉树和后序线索化二叉树的分析思路类似，同学们作为课后作业完成。

第 11 章 树结构实际应用

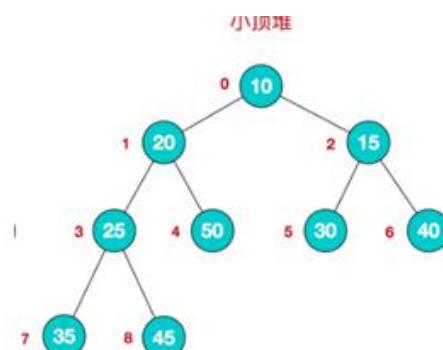
11.1 堆排序

11.1.1 堆排序基本介绍

- 1) 堆排序是利用堆这种数据结构而设计的一种排序算法，堆排序是一种选择排序，它的最坏，最好，平均时间复杂度均为 **$O(n \log n)$** ，它也是不稳定排序。
- 2) 堆是具有以下性质的完全二叉树：每个结点的值都大于或等于其左右孩子结点的值，称为大顶堆，注意：没有要求结点的左孩子的值和右孩子的值的大小关系。
- 3) 每个结点的值都小于或等于其左右孩子结点的值，称为小顶堆
- 4) 大顶堆举例说明



- 5) 小顶堆举例说明



小顶堆： ~~$arr[i] \leq arr[2*i+1] \&& arr[i] \leq arr[2*i+2]$~~ // i 对应第几个节点，i 从0开始编号

- 6) 一般升序采用大顶堆，降序采用小顶堆



11.1.2 堆排序基本思想

堆排序的基本思想是：

- 1) 将待排序序列构造成一个大顶堆
- 2) 此时，整个序列的最大值就是堆顶的根节点。
- 3) 将其与末尾元素进行交换，此时末尾就为最大值。
- 4) 然后将剩余 $n-1$ 个元素重新构造成一个堆，这样会得到 n 个元素的次小值。如此反复执行，便能得到一个有序序列了。

可以看到在构建大顶堆的过程中，元素的个数逐渐减少，最后就得到一个有序序列了。

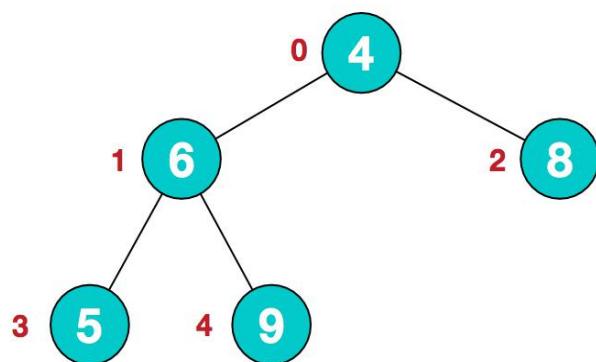
11.1.3 堆排序步骤图解说明

要求：给你一个数组 $\{4,6,8,5,9\}$ ，要求使用堆排序法，将数组升序排序。

步骤一 构造初始堆。将给定无序序列构造成一个大顶堆（一般升序采用大顶堆，降序采用小顶堆）。

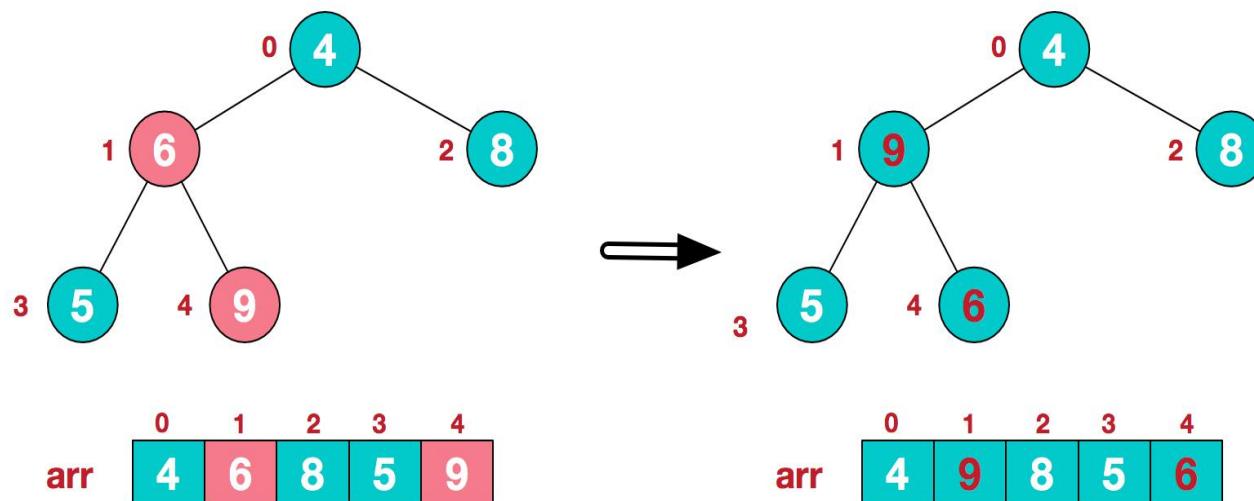
原始的数组 [4, 6, 8, 5, 9]

- 1) 假设给定无序序列结构如下

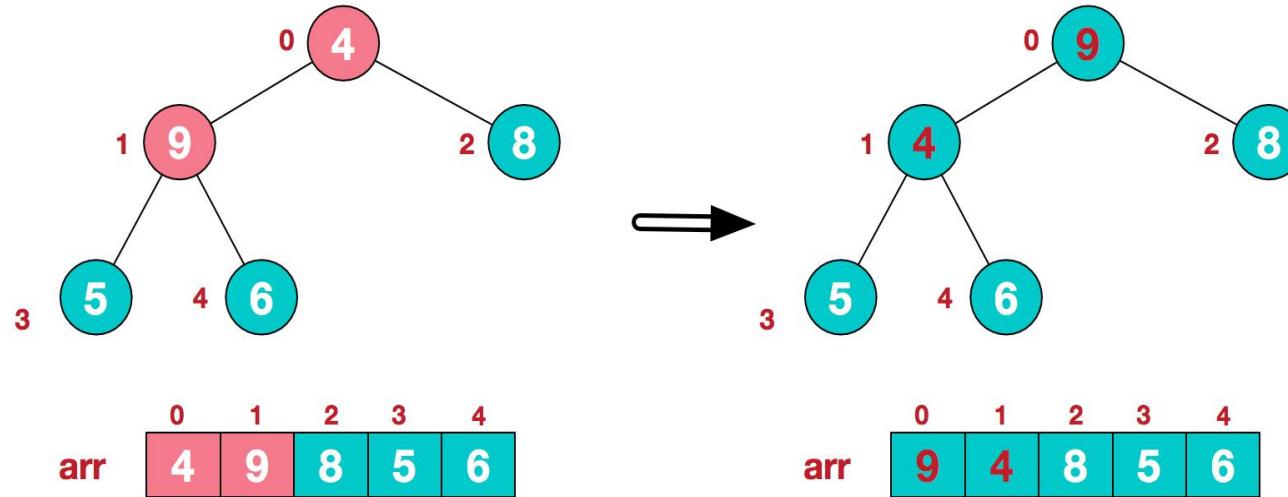


arr	0	1	2	3	4
	4	6	8	5	9

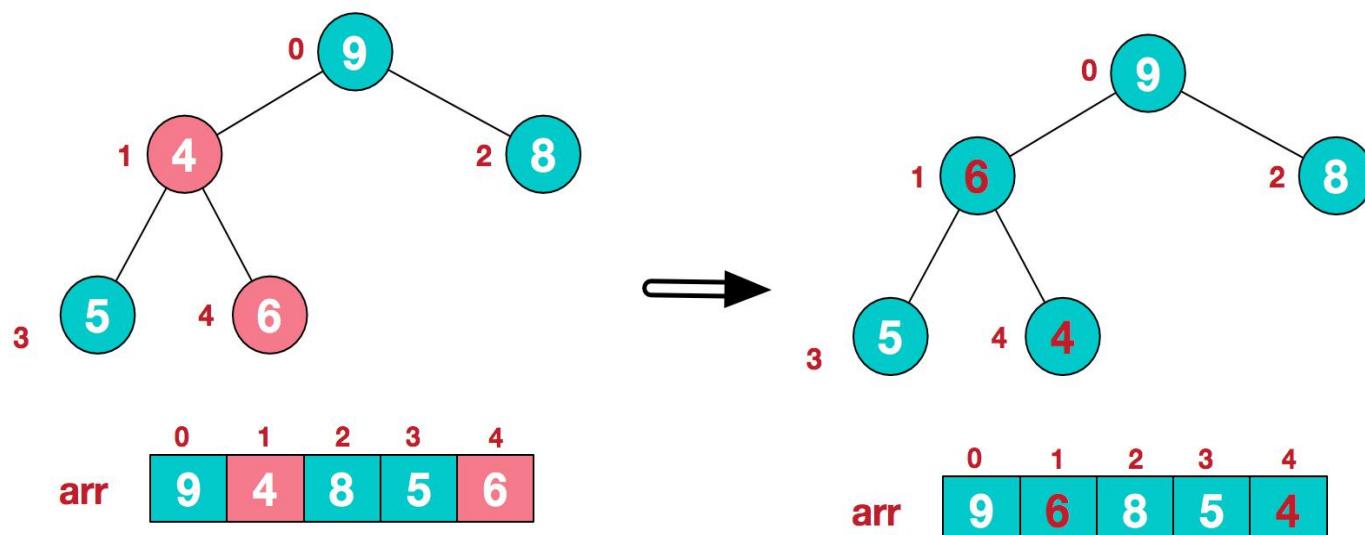
2) 此时我们从最后一个非叶子结点开始（叶结点自然不用调整，第一个非叶子结点
 $\text{arr.length}/2-1=5/2-1=1$ ，也就是下面的 6 结点），从左至右，从下至上进行调整。



3) 找到第二个非叶节点 4，由于[4,9,8]中 9 元素最大，4 和 9 交换。



4) 这时，交换导致了子根[4,5,6]结构混乱，继续调整，[4,5,6]中 6 最大，交换 4 和 6。

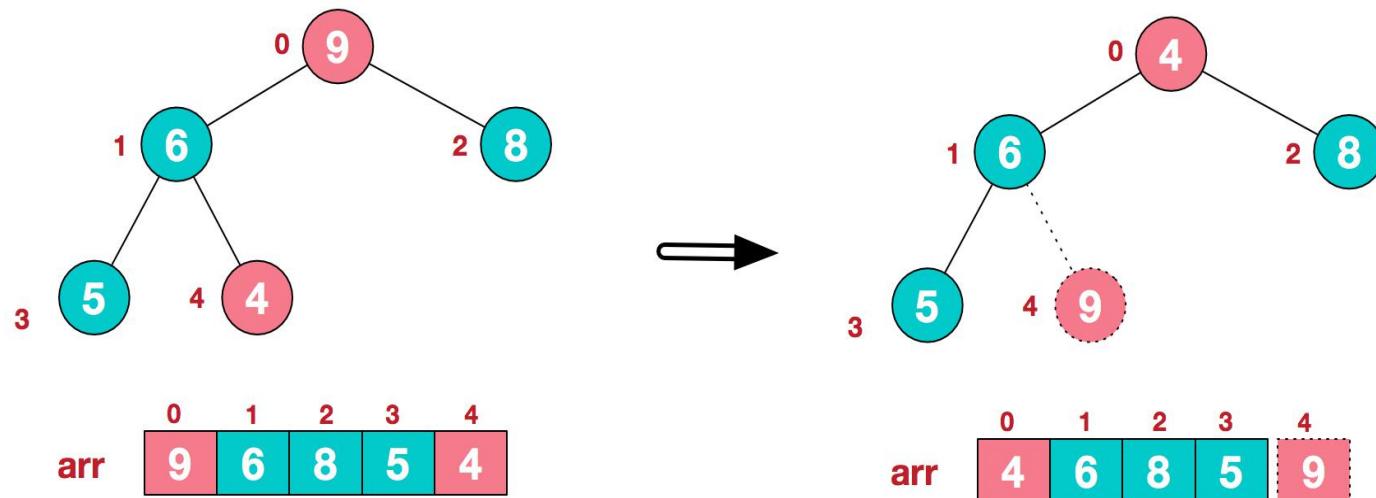


此时，我们就将一个无序序列构造成了一个大顶堆。

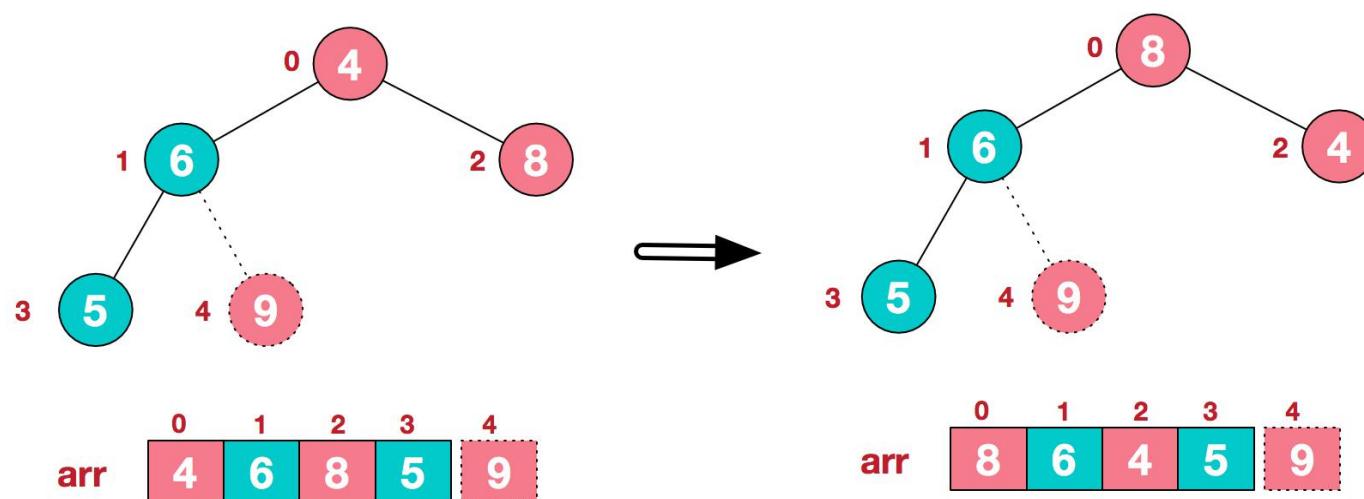
步骤二 将堆顶元素与末尾元素进行交换，使末尾元素最大。然后继续调整堆，再将堆顶元素与末尾元素交换，

得到第二大元素。如此反复进行交换、重建、交换。

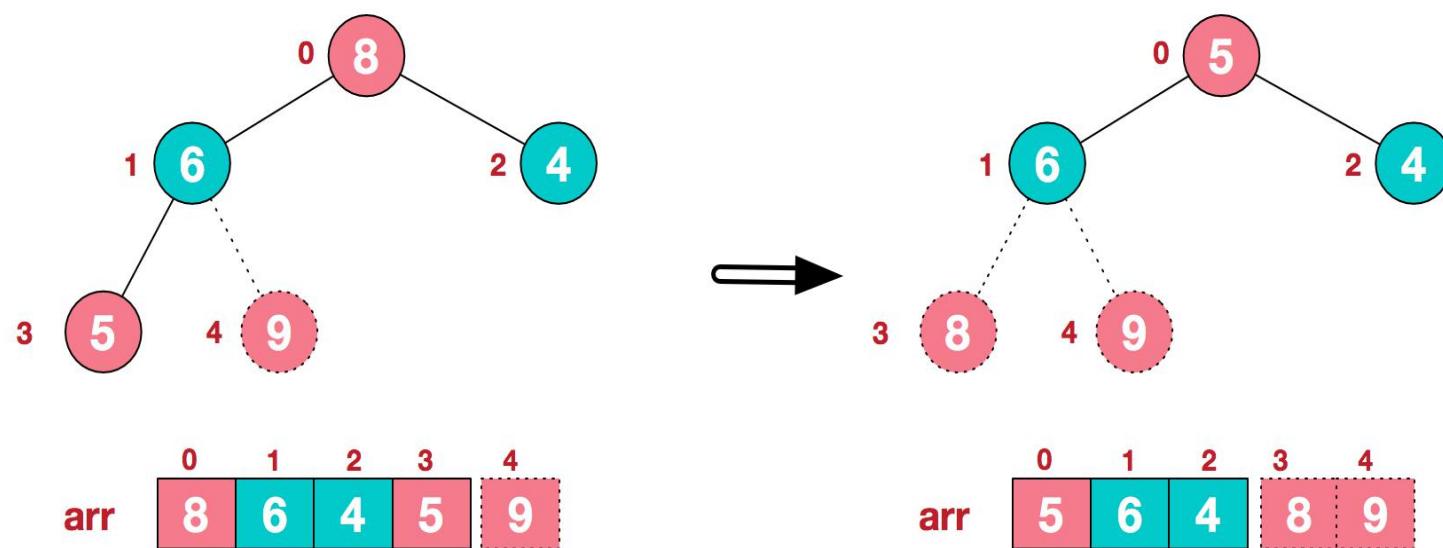
1) 将堆顶元素 9 和末尾元素 4 进行交换



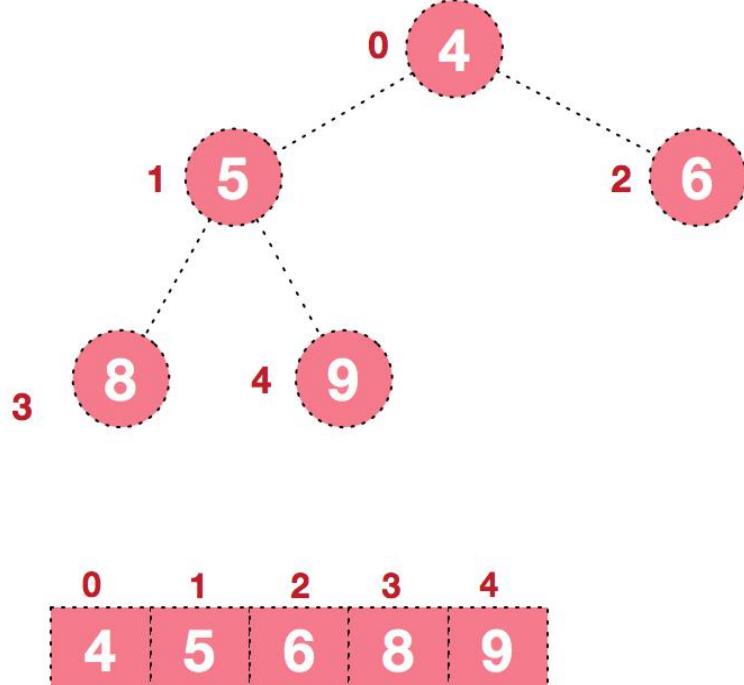
2) 重新调整结构，使其继续满足堆定义



3) 再将堆顶元素 8 与末尾元素 5 进行交换，得到第二大元素 8.



4) 后续过程，继续进行调整，交换，如此反复进行，最终使得整个序列有序





再简单总结下堆排序的基本思路：

- 1). 将无序序列构建成一个堆，根据升序降序需求选择大顶堆或小顶堆；
- 2). 将堆顶元素与末尾元素交换，将最大元素“沉”到数组末端；
- 3). 重新调整结构，使其满足堆定义，然后继续交换堆顶元素与当前末尾元素，反复执行调整+交换步骤，直到整个序列有序。

11.1.4 堆排序代码实现

要求：给你一个数组 {4,6,8,5,9}，要求使用堆排序法，将数组升序排序。

代码实现：看老师演示：

说明：

- 1) 堆排序不是很好理解，老师通过 Debug 帮助大家理解堆排序
- 2) 堆排序的速度非常快，在我的机器上 8 百万数据 3 秒左右。O(nlogn)
- 3) 代码实现

```
package com.atguigu.tree;

import java.text.SimpleDateFormat;
import java.util.Arrays;
import java.util.Date;
```



```
public class HeapSort {  
  
    public static void main(String[] args) {  
        //要求将数组进行升序排序  
        //int arr[] = {4, 6, 8, 5, 9};  
        // 创建要给 80000 个的随机的数组  
        int[] arr = new int[8000000];  
        for (int i = 0; i < 8000000; i++) {  
            arr[i] = (int) (Math.random() * 8000000); // 生成一个[0, 8000000) 数  
        }  
  
        System.out.println("排序前");  
        Date data1 = new Date();  
        SimpleDateFormat simpleDateFormat = new SimpleDateFormat("yyyy-MM-dd HH:mm:ss");  
        String date1Str = simpleDateFormat.format(data1);  
        System.out.println("排序前的时间是=" + date1Str);  
  
        heapSort(arr);  
  
        Date data2 = new Date();  
        String date2Str = simpleDateFormat.format(data2);  
        System.out.println("排序前的时间是=" + date2Str);  
        //System.out.println("排序后=" + Arrays.toString(arr));  
    }  
  
    //编写一个堆排序的方法
```



```
public static void heapSort(int arr[]) {  
    int temp = 0;  
    System.out.println("堆排序!!");  
  
    // //分步完成  
    // adjustHeap(arr, 1, arr.length);  
    // System.out.println("第一次" + Arrays.toString(arr)); // 4, 9, 8, 5, 6  
    //  
    // adjustHeap(arr, 0, arr.length);  
    // System.out.println("第 2 次" + Arrays.toString(arr)); // 9,6,8,5,4  
  
    //完成我们最终代码  
    //将无序序列构建成一个堆，根据升序降序需求选择大顶堆或小顶堆  
    for(int i = arr.length / 2 -1; i >=0; i--) {  
        adjustHeap(arr, i, arr.length);  
    }  
  
    /*  
     * 2).将堆顶元素与末尾元素交换，将最大元素"沉"到数组末端;  
     * 3).重新调整结构，使其满足堆定义，然后继续交换堆顶元素与当前末尾元素，反复执行调整+交换  
     * 步骤，直到整个序列有序。  
     */  
    for(int j = arr.length-1;j >0; j--) {  
        //交换  
        temp = arr[j];  
        arr[j] = arr[0];  
        arr[0] = temp;  
    }  
}
```



```
arr[0] = temp;
adjustHeap(arr, 0, j);
}

//System.out.println("数组=" + Arrays.toString(arr));

}

//将一个数组(二叉树), 调整成一个大顶堆
/**
 * 功能: 完成 将 以 i 对应的非叶子结点的树调整成大顶堆
 * 举例 int arr[] = {4, 6, 8, 5, 9}; => i = 1 => adjustHeap => 得到 {4, 9, 8, 5, 6}
 * 如果我们再次调用 adjustHeap 传入的是 i = 0 => 得到 {4, 9, 8, 5, 6} => {9,6,8,5, 4}
 * @param arr 待调整的数组
 * @param i 表示非叶子结点在数组中索引
 * @param lenght 表示对多少个元素继续调整, length 是在逐渐的减少
 */
public static void adjustHeap(int arr[], int i, int lenght) {

    int temp = arr[i];//先取出当前元素的值, 保存在临时变量
    //开始调整
    //说明
    //1. k = i * 2 + 1 k 是 i 结点的左子结点
    for(int k = i * 2 + 1; k < lenght; k = k * 2 + 1) {
        if(k+1 < lenght && arr[k] < arr[k+1]) { //说明左子结点的值小于右子结点的值
            k++; // k 指向右子结点
    }
}
```



```
    }

    if(arr[k] > temp) { //如果子结点大于父结点
        arr[i] = arr[k]; //把较大的值赋给当前结点
        i = k; //!!! i 指向 k,继续循环比较
    } else {
        break;//!
    }

}

//当 for 循环结束后，我们已经将以 i 为父结点的树的最大值，放在了 最顶(局部)
arr[i] = temp;//将 temp 值放到调整后的位置

}

}
```

11.2 赫夫曼树

11.2.1 基本介绍

- 1) 给定 n 个权值作为 n 个叶子结点，构造一棵二叉树，若该树的带权路径长度(wpl)达到最小，称这样的二叉树为最优二叉树，也称为哈夫曼树(Huffman Tree)，还有的书翻译为霍夫曼树。
- 2) 赫夫曼树是带权路径长度最短的树，权值较大的结点离根较近

11.2.2 赫夫曼树几个重要概念和举例说明

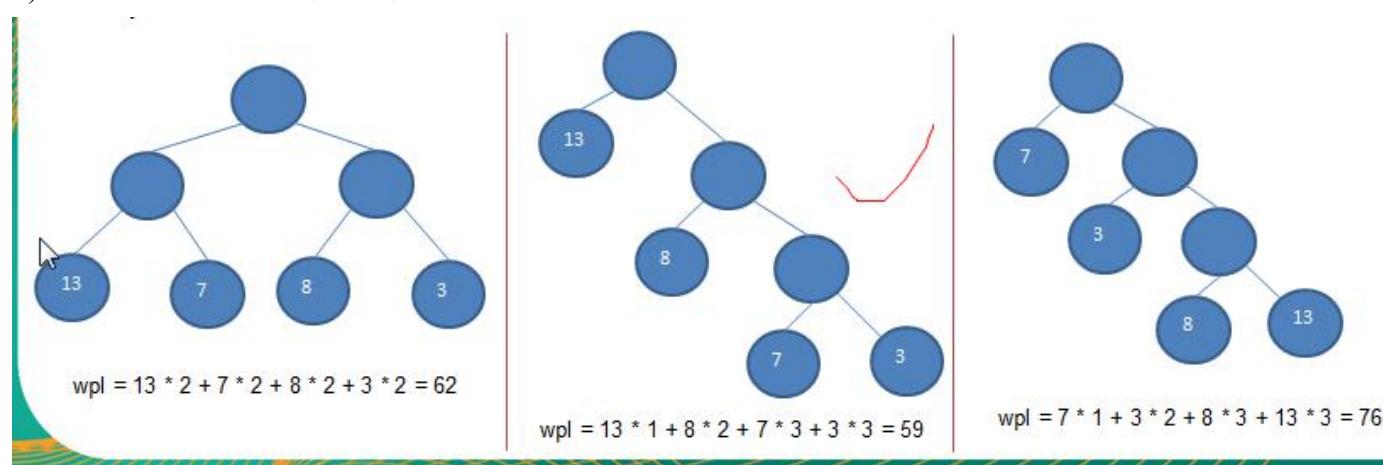
- 1) **路径和路径长度：**在一棵树中，从一个结点往下可以达到的孩子或孙子结点之间的通路，称为路径。通路

中分支的数目称为路径长度。若规定根结点的层数为 1，则从根结点到第 L 层结点的路径长度为 L-1

2) 结点的权及带权路径长度：若将树中结点赋给一个有着某种含义的数值，则这个数值称为该结点的权。结点的带权路径长度为：从根结点到该结点之间的路径长度与该结点的权的乘积

3) 树的带权路径长度：树的带权路径长度规定为所有叶子结点的带权路径长度之和，记为 WPL(weighted path length), 权值越大的结点离根结点越近的二叉树才是最优二叉树。

4) WPL 最小的就是赫夫曼树



11.2.3 赫夫曼树创建思路图解

给你一个数列 {13, 7, 8, 3, 29, 6, 1}，要求转成一颗赫夫曼树。

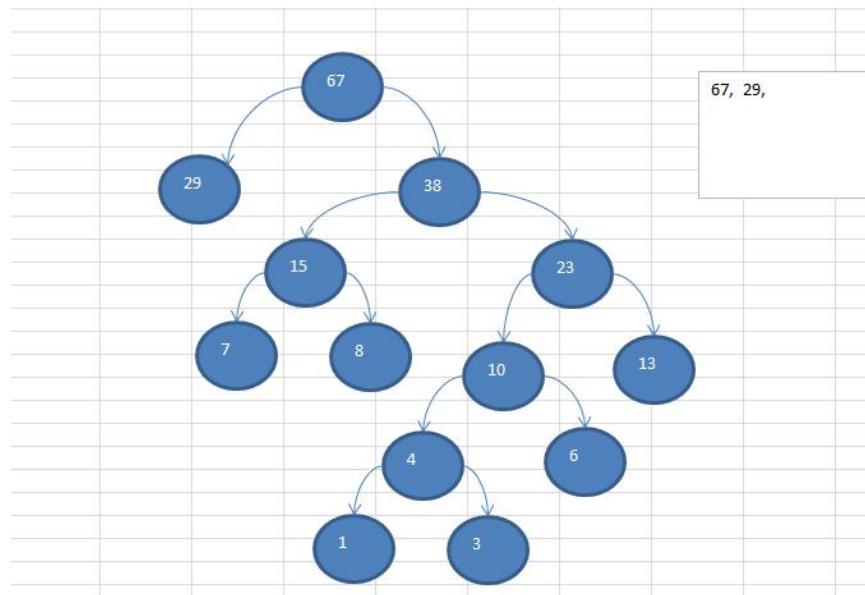
➤ 思路分析(示意图):

{13, 7, 8, 3, 29, 6, 1}

构成赫夫曼树的步骤：

- 1) 从小到大进行排序，将每一个数据，每个数据都是一个节点， 每个节点可以看成是一颗最简单的二叉树
- 2) 取出根节点权值最小的两颗二叉树
- 3) 组成一颗新的二叉树，该新的二叉树的根节点的权值是前面两颗二叉树根节点权值的和

- 4) 再将这颗新的二叉树，以根节点的权值大小 再次排序，不断重复 1-2-3-4 的步骤，直到数列中，所有的数据都被处理，就得到一颗赫夫曼树
- 5) 图解：



11.2.4 赫夫曼树的代码实现

代码实现：

```
package com.atguigu.huffmanTree;

import java.util.ArrayList;
import java.util.Collections;
import java.util.List;

public class HuffmanTree {

    public static void main(String[] args) {
        int arr[] = { 13, 7, 8, 3, 29, 6, 1 };
```



```
Node root = createHuffmanTree(arr);

//测试一把
preOrder(root); //


}

//编写一个前序遍历的方法
public static void preOrder(Node root) {
    if(root != null) {
        root.preOrder();
    }else{
        System.out.println("是空树， 不能遍历~~");
    }
}

// 创建赫夫曼树的方法
/**
 *
 * @param arr 需要创建成哈夫曼树的数组
 * @return 创建好后的赫夫曼树的 root 结点
 */
public static Node createHuffmanTree(int[] arr) {
    // 第一步为了操作方便
    // 1. 遍历 arr 数组
    // 2. 将 arr 的每个元素构成一个 Node
```



```
// 3. 将 Node 放入到 ArrayList 中

List<Node> nodes = new ArrayList<Node>();
for (int value : arr) {
    nodes.add(new Node(value));
}

//我们处理的过程是一个循环的过程

while(nodes.size() > 1) {

    //排序 从小到大
    Collections.sort(nodes);

    System.out.println("nodes =" + nodes);

    //取出根节点权值最小的两颗二叉树
    //(1) 取出权值最小的结点（二叉树）
    Node leftNode = nodes.get(0);
    //(2) 取出权值第二小的结点（二叉树）
    Node rightNode = nodes.get(1);

    //(3)构建一颗新的二叉树
    Node parent = new Node(leftNode.value + rightNode.value);
    parent.left = leftNode;
    parent.right = rightNode;
```



```
//(4)从 ArrayList 删除处理过的二叉树
nodes.remove(leftNode);
nodes.remove(rightNode);
//(5)将 parent 加入到 nodes
nodes.add(parent);

}

//返回哈夫曼树的 root 结点
return nodes.get(0);

}

// 创建结点类
// 为了让 Node 对象持续排序 Collections 集合排序
// 让 Node 实现 Comparable 接口
class Node implements Comparable<Node> {
    int value; // 结点权值
    Node left; // 指向左子结点
    Node right; // 指向右子结点

    //写一个前序遍历
    public void preOrder() {
        System.out.println(this);
        if(this.left != null) {
```



```
this.left.preOrder();

}

if(this.right != null) {

    this.right.preOrder();

}

}

public Node(int value) {

    this.value = value;

}

@Override

public String toString() {

    return "Node [value=" + value + "]";

}

@Override

public int compareTo(Node o) {

    // TODO Auto-generated method stub

    // 表示从小到大排序

    return this.value - o.value;

}

}
```



11.3 赫夫曼编码

11.3.1 基本介绍

- 1) 赫夫曼编码也翻译为 哈夫曼编码(Huffman Coding), 又称霍夫曼编码, 是一种编码方式, 属于一种程序算法
- 2) 赫夫曼编码是赫哈夫曼树在电讯通信中的经典的应用之一。
- 3) 赫夫曼编码广泛地用于数据文件压缩。其压缩率通常在 20%~90%之间
- 4) 赫夫曼码是可变字长编码(VLC)的一种。Huffman 于 1952 年提出一种编码方法, 称之为最佳编码

11.3.2 原理剖析

➤ 通信领域中信息的处理方式 1-定长编码

- i like like java do you like a java // 共40个字符(包括空格)
- 105 32 108 105 107 101 32 108 105 107 101 32 108 105 107 101 32 106 97 118 97
32 100 111 32 121 111 117 32 108 105 101 32 97 32 106 97 118 97 //对应Ascii码
- 01101001 00100000 01101100 01101001 01101011 01100101 00100000 01101100
01101001 01101011 01100101 00100000 01101100 01101001 01101011 01100101
00100000 01101010 01100001 01110110 01100001 00100000 01100100 01101111
00100000 01111001 01101111 01110101 00100000 01101100 01101001 01101011
01100101 00100000 01100001 00100000 01101010 01100001 01110110 01100001 //
对应的二进制
- 按照二进制来传递信息, 总的长度是 359 (包括空格)
- 在线转码工具: <https://www.mokuge.com/tool/asciito16/>

➤ 通信领域中信息的处理方式 2-变长编码

- i like like like java do you like a java // 共40个字符(包括空格)
- d:1 y:1 u:1 j:2 v:2 o:2 l:4 k:4 e:4 i:5 a:5 :9 // 各个字符对应的个数
- 0= , 1=a, 10=i, 11=e, 100=k, 101=l, 110=o, 111=v, 1000=j, 1001=u, 1010=y, 1011=d
说明：按照各个字符出现的次数进行编码，原则是出现次数越多的，则编码越小，比如空格出现了9次，编码为0，其它依次类推。
- 按照上面给各个字符规定的编码，则我们在传输 "i like like like java do you like a java" 数据时，编码就是
10010110100...
- 字符的编码都不能是其他字符编码的前缀，符合此要求的编码叫做前缀编码，即不能匹配到重复的编码(这个在赫夫曼编码中，我们还要进行举例说明，不捉急)

➤ 通信领域中信息的处理方式 3-赫夫曼编码

步骤如下：

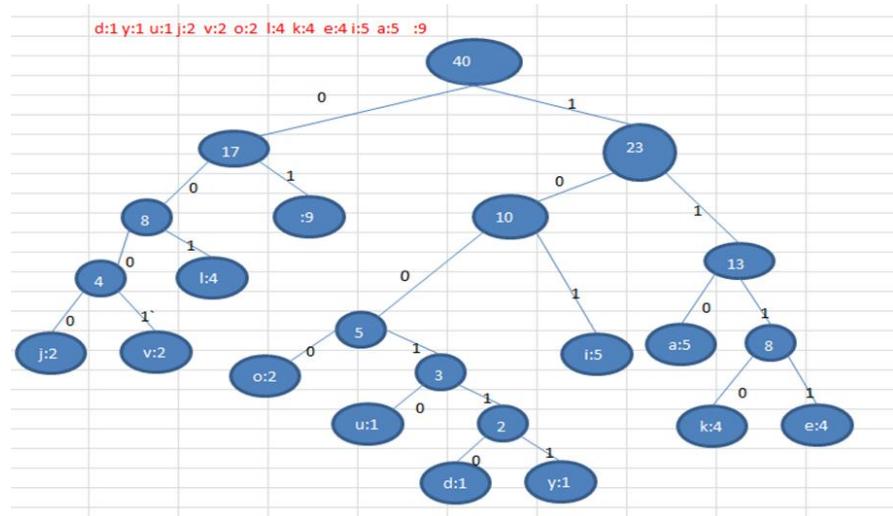
传输的 字符串

- 1) i like like like java do you like a java
- 2) d:1 y:1 u:1 j:2 v:2 o:2 l:4 k:4 e:4 i:5 a:5 :9 // 各个字符对应的个数
- 3) 按照上面字符出现的次数构建一颗赫夫曼树，次数作为权值

步骤：

构成赫夫曼树的步骤：

- 1) 从小到大进行排序，将每一个数据，每个数据都是一个节点，每个节点可以看成是一颗最简单的二叉树
- 2) 取出根节点权值最小的两颗二叉树
- 3) 组成一颗新的二叉树，该新的二叉树的根节点的权值是前面两颗二叉树根节点权值的和
- 4) 再将这颗新的二叉树，以根节点的权值大小再次排序，不断重复 1-2-3-4 的步骤，直到数列中，所有的数据都被处理，就得到一颗赫夫曼树



- 4) 根据赫夫曼树, 给各个字符, 规定编码 (前缀编码), 向左的路径为 0 向右的路径为 1 , 编码如下:

o: 1000 u: 10010 d: 100110 y: 100111 i: 101
a : 110 k: 1110 e: 1111 j: 0000 v: 0001
l: 001 : 01

- 5) 按照上面的赫夫曼编码，我们的 "i like like java do you like a java" 字符串对应的编码为 (注意这里我们使用的无损压缩)

10101001101111011110100110111101111010011011110111101000011000011100110011110000110
01111000100100100110111101111011100100001100001110 通过赫夫曼编码处理 长度为 133

- 6) 长度为 : 133

说明:

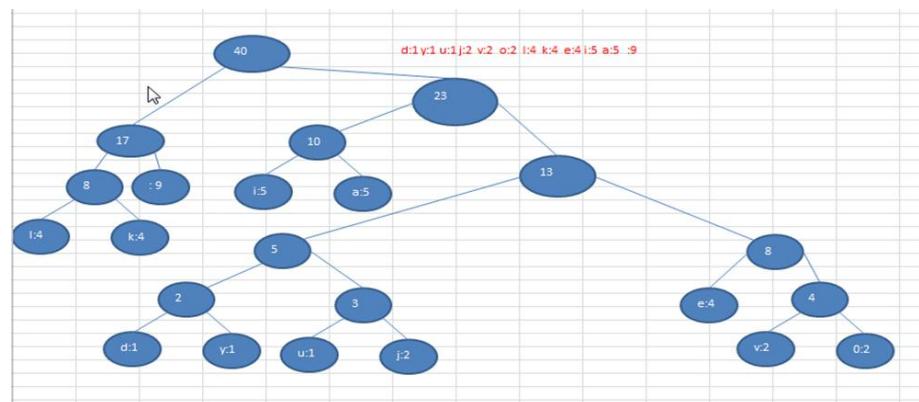
原来长度是 359，压缩了 $(359-133) / 359 = 62.9\%$

此编码满足前缀编码，即字符的编码都不能是其他字符编码的前缀。不会造成匹配的多义性

赫夫曼编码是无损处理方案

➤ 注意事项

注意，这个赫夫曼树根据排序方法不同，也可能不太一样，这样对应的赫夫曼编码也不完全一样，但是 wpl 是一样的，都是最小的，最后生成的赫夫曼编码的长度是一样，比如：如果我们让每次生成的新的二叉树总是排在权值相同的二叉树的最后一个，则生成的二叉树为：



11.3.3 最佳实践-数据压缩(创建赫夫曼树)

将给出的一段文本，比如 "i like like like java do you like a java"，根据前面的讲的赫夫曼编码原理，对其进行数
据 压 缩 处 理 ， 形 式 如

"1010100110111101111010011011110111101001101111011110100001100001110011001111000011001111000100100100

110111101111011100100001100001110

"

步骤 1：根据赫夫曼编码压缩数据的原理，需要创建 "i like like like java do you like a java" 对应的赫夫曼树。



思路：前面已经分析过了，而且我们已然讲过了构建赫夫曼树的具体实现。

代码实现：看老师演示：

```
//可以通过 List 创建对应的赫夫曼树

private static Node createHuffmanTree(List<Node> nodes) {

    while(nodes.size() > 1) {
        //排序, 从小到大
        Collections.sort(nodes);
        //取出第一颗最小的二叉树
        Node leftNode = nodes.get(0);
        //取出第二颗最小的二叉树
        Node rightNode = nodes.get(1);
        //创建一颗新的二叉树,它的根节点 没有 data, 只有权值
        Node parent = new Node(null, leftNode.weight + rightNode.weight);
        parent.left = leftNode;
        parent.right = rightNode;

        //将已经处理的两颗二叉树从 nodes 删除
        nodes.remove(leftNode);
        nodes.remove(rightNode);
        //将新的二叉树, 加入到 nodes
        nodes.add(parent);

    }

    //nodes 最后的结点, 就是赫夫曼树的根结点
    return nodes.get(0);
}
```



{}

11.3.4 最佳实践-数据压缩(生成赫夫曼编码和赫夫曼编码后的数据)

我们已经生成了 赫夫曼树，下面我们继续完成任务

- 1) 生成赫夫曼树对应的赫夫曼编码，如下表：

=01 a=100 d=11000 u=11001 e=1110 v=11011 i=101 y=11010 j=0010 k=1111 l=000 o=0011

- 2) 使用赫夫曼编码来生成赫夫曼编码数据，即按照上面的赫夫曼编码，将 "i like like like java do you like a java" 字符串生成对应的编码数据，形式如下。

1010100010111111001000101111111001000101111111001001010011011100011100000110111010001111001010
00101111111001100010010011011100

- 3) 思路：前面已经分析过了，而且我们讲过了生成赫夫曼编码的具体实现。

- 4) 代码实现：看老师演示：

```
//为了调用方便，我们重载 getCodes
private static Map<Byte, String> getCodes(Node root) {
    if(root == null) {
        return null;
    }
    //处理 root 的左子树
    getCodes(root.left, "0", stringBuilder);
    //处理 root 的右子树
    getCodes(root.right, "1", stringBuilder);
    return huffmanCodes;
```



```
}
```

```
/**
```

```
* 功能：将传入的 node 结点的所有叶子结点的赫夫曼编码得到，并放入到 huffmanCodes 集合
```

```
* @param node 传入结点
```

```
* @param code 路径：左子结点是 0，右子结点 1
```

```
* @param stringBuilder 用于拼接路径
```

```
*/
```

```
private static void getCodes(Node node, String code, StringBuilder stringBuilder) {
```

```
    StringBuilder stringBuilder2 = new StringBuilder(stringBuilder);
```

```
    //将 code 加入到 stringBuilder2
```

```
    stringBuilder2.append(code);
```

```
    if(node != null) { //如果 node == null 不处理
```

```
        //判断当前 node 是叶子结点还是非叶子结点
```

```
        if(node.data == null) { //非叶子结点
```

```
            //递归处理
```

```
            //向左递归
```

```
            getCodes(node.left, "0", stringBuilder2);
```

```
            //向右递归
```

```
            getCodes(node.right, "1", stringBuilder2);
```

```
        } else { //说明是一个叶子结点
```

```
            //就表示找到某个叶子结点的最后
```

```
            huffmanCodes.put(node.data, stringBuilder2.toString());
```

```
        }
```

```
}
```

```
}
```



11.3.5 最佳实践-数据解压(使用赫夫曼编码解码)

使用赫夫曼编码来解码数据，具体要求是

- 1) 前面我们得到了赫夫曼编码和对应的编码

byte[]，即: [-88, -65, -56, -65, -56, -65, -55, 77
, -57, 6, -24, -14, -117, -4, -60, -90, 28]

- 2) 现在要求使用赫夫曼编码，进行解码，又

重新得到原来的字符串 "i like like like java do you like a java"

- 3) 思路：解码过程，就是编码的一个逆向操作。

- 4) 代码实现：看老师演示：

```
/**  
 * 将一个 byte 转成一个二进制的字符串，如果看不懂，可以参考我讲的 Java 基础 二进制的原码，反码，补  
码  
 * @param b 传入的 byte  
 * @param flag 标志是否需要补高位如果是 true ， 表示需要补高位，如果是 false 表示不补，如果是最后一个  
字节，无需补高位  
 * @return 是该 b 对应的二进制的字符串，（注意是按补码返回）  
 */  
  
private static String byteToBitString(boolean flag, byte b) {  
    //使用变量保存 b  
    int temp = b; //将 b 转成 int  
    //如果是正数我们还存在补高位  
    if(flag) {  
        temp |= 256; //按位与 256 1 0000 0000 | 0000 0001 => 1 0000 0001  
    }  
    String str = Integer.toBinaryString(temp); //返回的是 temp 对应的二进制的补码
```



```
if(flag) {  
    return str.substring(str.length() - 8);  
} else {  
    return str;  
}  
}
```

//编写一个方法，完成对压缩数据的解码

```
/**  
 *  
 * @param huffmanCodes 赫夫曼编码表 map  
 * @param huffmanBytes 赫夫曼编码得到的字节数组  
 * @return 就是原来的字符串对应的数组  
 */
```

```
private static byte[] decode(Map<Byte, String> huffmanCodes, byte[] huffmanBytes) {
```

//1. 先得到 huffmanBytes 对应的 二进制的字符串， 形式 1010100010111...

```
StringBuilder stringBuilder = new StringBuilder();
```

//将 byte 数组转成二进制的字符串

```
for(int i = 0; i < huffmanBytes.length; i++) {
```

```
    byte b = huffmanBytes[i];
```

//判断是不是最后一个字节

```
    boolean flag = (i == huffmanBytes.length - 1);
```

```
    stringBuilder.append(byteToBitString(!flag, b));
```

```
}
```

//把字符串安装指定的赫夫曼编码进行解码



```
//把赫夫曼编码表进行调换，因为反向查询 a->100 100->a
Map<String, Byte> map = new HashMap<String, Byte>();
for(Map.Entry<Byte, String> entry: huffmanCodes.entrySet()) {
    map.put(entry.getValue(), entry.getKey());
}

//创建要给集合，存放 byte
List<Byte> list = new ArrayList<>();
//i 可以理解成就是索引,扫描 stringBuilder
for(int i = 0; i < stringBuilder.length(); ) {
    int count = 1; // 小的计数器
    boolean flag = true;
    Byte b = null;

    while(flag) {
        //101010001011...
        //递增的取出 key 1
        String key = stringBuilder.substring(i, i+count); //i 不动，让 count 移动，指定匹配到一个字符
        b = map.get(key);
        if(b == null) { //说明没有匹配到
            count++;
        } else {
            //匹配到
            flag = false;
        }
    }
}
```



```
list.add(b);

i += count;//i 直接移动到 count

}

//当 for 循环结束后，我们 list 中就存放了所有的字符 "i like like like java do you like a java"

//把 list 中的数据放入到 byte[] 并返回

byte b[] = new byte[list.size()];

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

    b[i] = list.get(i);

}

return b;

}
```

11.3.6 最佳实践-文件压缩

我们学习了通过赫夫曼编码对一个字符串进行编码和解码，下面我们来完成对文件的压缩和解压，具体要求：给你一个图片文件，要求对其进行无损压缩，看看压缩效果如何。

- 1) 思路：读取文件-> 得到赫夫曼编码表 -> 完成压缩
- 2) 代码实现：

```
//编写方法，将一个文件进行压缩

/**
 *
 * @param srcFile 你传入的希望压缩的文件的全路径
 * @param dstFile 我们压缩后将压缩文件放到哪个目录
 */
```



```
public static void zipFile(String srcFile, String dstFile) {  
  
    //创建输出流  
    OutputStream os = null;  
    ObjectOutputStream oos = null;  
    //创建文件的输入流  
    FileInputStream is = null;  
    try {  
        //创建文件的输入流  
        is = new FileInputStream(srcFile);  
        //创建一个和源文件大小一样的 byte[]  
        byte[] b = new byte[is.available()];  
        //读取文件  
        is.read(b);  
        //直接对源文件压缩  
        byte[] huffmanBytes = huffmanZip(b);  
        //创建文件的输出流，存放压缩文件  
        os = new FileOutputStream(dstFile);  
        //创建一个和文件输出流关联的 ObjectOutputStream  
        oos = new ObjectOutputStream(os);  
        //把 赫夫曼编码后的字节数组写入压缩文件  
        oos.writeObject(huffmanBytes); //我们是把  
        //这里我们以对象流的方式写入 赫夫曼编码，是为了以后我们恢复源文件时使用  
        //注意一定要把赫夫曼编码 写入压缩文件  
        oos.writeObject(huffmanCodes);  
    } catch (Exception e) {  
        e.printStackTrace();  
    } finally {  
        if (os != null) {  
            try {  
                os.close();  
            } catch (IOException e) {  
                e.printStackTrace();  
            }  
        }  
        if (oos != null) {  
            try {  
                oos.close();  
            } catch (IOException e) {  
                e.printStackTrace();  
            }  
        }  
        if (is != null) {  
            try {  
                is.close();  
            } catch (IOException e) {  
                e.printStackTrace();  
            }  
        }  
    }  
}
```



```
        }catch (Exception e) {  
            // TODO: handle exception  
            System.out.println(e.getMessage());  
        }finally {  
            try {  
                is.close();  
                oos.close();  
                os.close();  
            }catch (Exception e) {  
                // TODO: handle exception  
                System.out.println(e.getMessage());  
            }  
        }  
    }  
}
```

11.3.7 最佳实践-文件解压(文件恢复)

具体要求：将前面压缩的文件，重新恢复成原来的文件。

- 1) 思路：读取压缩文件(数据和赫夫曼编码表)-> 完成解压(文件恢复)
- 2) 代码实现：

```
//编写一个方法，完成对压缩文件的解压  
/**  
 *  
 */
```



```
* @param zipFile 准备解压的文件
* @param dstFile 将文件解压到哪个路径
*/
public static void unZipFile(String zipFile, String dstFile) {

    //定义文件输入流
    InputStream is = null;
    //定义一个对象输入流
    ObjectInputStream ois = null;
    //定义文件的输出流
    OutputStream os = null;
    try {
        //创建文件输入流
        is = new FileInputStream(zipFile);
        //创建一个和 is 关联的对象输入流
        ois = new ObjectInputStream(is);
        //读取 byte 数组 huffmanBytes
        byte[] huffmanBytes = (byte[])ois.readObject();
        //读取赫夫曼编码表
        Map<Byte,String> huffmanCodes = (Map<Byte,String>)ois.readObject();

        //解码
        byte[] bytes = decode(huffmanCodes, huffmanBytes);
        //将 bytes 数组写入到目标文件
        os = new FileOutputStream(dstFile);
        //写数据到 dstFile 文件
    } catch (Exception e) {
        e.printStackTrace();
    } finally {
        if (os != null) {
            try {
                os.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        if (ois != null) {
            try {
                ois.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
        if (is != null) {
            try {
                is.close();
            } catch (IOException e) {
                e.printStackTrace();
            }
        }
    }
}
```



```
        os.write(bytes);

    } catch (Exception e) {
        // TODO: handle exception
        System.out.println(e.getMessage());
    } finally {

        try {
            os.close();
            ois.close();
            is.close();
        } catch (Exception e2) {
            // TODO: handle exception
            System.out.println(e2.getMessage());
        }
    }
}
```

11.3.8 代码汇总，把前面所有的方法放在一起

```
package com.atguigu.huffmancode;

import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.InputStream;
```



```
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.OutputStream;
import java.util.ArrayList;
import java.util.Arrays;
import java.util.Collections;
import java.util.HashMap;
import java.util.List;
import java.util.Map;

public class HuffmanCode {

    public static void main(String[] args) {

        //测试压缩文件
        //    String srcFile = "d://Uninstall.xml";
        //    String dstFile = "d://Uninstall.zip";
        //
        //    zipFile(srcFile, dstFile);
        //    System.out.println("压缩文件 ok~~");

        //测试解压文件
        String zipFile = "d://Uninstall.zip";
        String dstFile = "d://Uninstall2.xml";
        unZipFile(zipFile, dstFile);
    }
}
```



```
System.out.println("解压成功!");

/*
String content = "i like like like java do you like a java";
byte[] contentBytes = content.getBytes();
System.out.println(contentBytes.length); //40

byte[] huffmanCodesBytes= huffmanZip(contentBytes);
System.out.println(" 压 缩 后 的 结 果 是 :" + Arrays.toString(huffmanCodesBytes) + " 长 度 = " +
huffmanCodesBytes.length);

//测试一把 byteToBitString 方法
//System.out.println(byteToBitString((byte)1));
byte[] sourceBytes = decode(huffmanCodes, huffmanCodesBytes);

System.out.println("原来的字符串=" + new String(sourceBytes)); // "i like like like java do you like a java"
*/

//如何将 数据进行解压(解码)
//分步过程
/*
List<Node> nodes = getNodes(contentBytes);
System.out.println("nodes=" + nodes);
```



```
//测试一把，创建的赫夫曼树
System.out.println("赫夫曼树");
Node huffmanTreeRoot = createHuffmanTree(nodes);
System.out.println("前序遍历");
huffmanTreeRoot.preOrder();

//测试一把是否生成了对应的赫夫曼编码
Map<Byte, String> huffmanCodes = getCode(huffmanTreeRoot);
System.out.println("~生成的赫夫曼编码表= " + huffmanCodes);

//测试
byte[] huffmanCodeBytes = zip(contentBytes, huffmanCodes);
System.out.println("huffmanCodeBytes=" + Arrays.toString(huffmanCodeBytes)); //17

//发送 huffmanCodeBytes 数组 */

}

//编写一个方法，完成对压缩文件的解压
/**
 *
 * @param zipFile 准备解压的文件
 * @param dstFile 将文件解压到哪个路径
 */

```



```
public static void unZipFile(String zipFile, String dstFile) {  
  
    //定义文件输入流  
    InputStream is = null;  
    //定义一个对象输入流  
    ObjectInputStream ois = null;  
    //定义文件的输出流  
    OutputStream os = null;  
    try {  
        //创建文件输入流  
        is = new FileInputStream(zipFile);  
        //创建一个和 is 关联的对象输入流  
        ois = new ObjectInputStream(is);  
        //读取 byte 数组 huffmanBytes  
        byte[] huffmanBytes = (byte[])ois.readObject();  
        //读取赫夫曼编码表  
        Map<Byte,String> huffmanCodes = (Map<Byte,String>)ois.readObject();  
  
        //解码  
        byte[] bytes = decode(huffmanCodes, huffmanBytes);  
        //将 bytes 数组写入到目标文件  
        os = new FileOutputStream(dstFile);  
        //写数据到 dstFile 文件  
        os.write(bytes);  
    } catch (Exception e) {  
        // TODO: handle exception  
    }
```



```
System.out.println(e.getMessage());  
} finally {  
  
    try {  
        os.close();  
        ois.close();  
        is.close();  
    } catch (Exception e2) {  
        // TODO: handle exception  
        System.out.println(e2.getMessage());  
    }  
  
}  
  
}  
  
//编写方法，将一个文件进行压缩  
/**  
 *  
 * @param srcFile 你传入的希望压缩的文件的全路径  
 * @param dstFile 我们压缩后将压缩文件放到哪个目录  
 */  
public static void zipFile(String srcFile, String dstFile) {  
  
    //创建输出流  
    OutputStream os = null;  
    ObjectOutputStream oos = null;
```



```
//创建文件的输入流
FileInputStream is = null;
try {
    //创建文件的输入流
    is = new FileInputStream(srcFile);
    //创建一个和源文件大小一样的 byte[]
    byte[] b = new byte[is.available()];
    //读取文件
    is.read(b);
    //直接对源文件压缩
    byte[] huffmanBytes = huffmanZip(b);
    //创建文件的输出流，存放压缩文件
    os = new FileOutputStream(dstFile);
    //创建一个和文件输出流关联的 ObjectOutputStream
    oos = new ObjectOutputStream(os);
    //把 赫夫曼编码后的字节数组写入压缩文件
    oos.writeObject(huffmanBytes); //我们是把
    //这里我们以对象流的方式写入 赫夫曼编码，是为了以后我们恢复源文件时使用
    //注意一定要把赫夫曼编码 写入压缩文件
    oos.writeObject(huffmanCodes);

} catch (Exception e) {
    // TODO: handle exception
    System.out.println(e.getMessage());
} finally {
```



```
try {
    is.close();
    oos.close();
    os.close();
} catch (Exception e) {
    // TODO: handle exception
    System.out.println(e.getMessage());
}

}

//完成数据的解压
//思路
//1. 将 huffmanCodeBytes [-88, -65, -56, -65, -56, -65, -55, 77, -57, 6, -24, -14, -117, -4, -60, -90, 28]
//   重写先转成 赫夫曼编码对应的二进制的字符串 "1010100010111..."
//2. 赫夫曼编码对应的二进制的字符串 "1010100010111..." => 对照 赫夫曼编码 => "i like like like
java do you like a java"

//编写一个方法，完成对压缩数据的解码
/**
 *
 * @param huffmanCodes 赫夫曼编码表 map
 * @param huffmanBytes 赫夫曼编码得到的字节数组
 * @return 就是原来的字符串对应的数组
```



```
*/
```

```
private static byte[] decode(Map<Byte, String> huffmanCodes, byte[] huffmanBytes) {  
  
    //1. 先得到 huffmanBytes 对应的 二进制的字符串 , 形式 1010100010111...  
    StringBuilder stringBuilder = new StringBuilder();  
    //将 byte 数组转成二进制的字符串  
    for(int i = 0; i < huffmanBytes.length; i++) {  
        byte b = huffmanBytes[i];  
        //判断是不是最后一个字节  
        boolean flag = (i == huffmanBytes.length - 1);  
        stringBuilder.append(byteToBitString(!flag, b));  
    }  
    //把字符串安装指定的赫夫曼编码进行解码  
    //把赫夫曼编码表进行调换, 因为反向查询 a->100 100->a  
    Map<String, Byte> map = new HashMap<String, Byte>();  
    for(Map.Entry<Byte, String> entry: huffmanCodes.entrySet()) {  
        map.put(entry.getValue(), entry.getKey());  
    }  
  
    //创建要给集合, 存放 byte  
    List<Byte> list = new ArrayList<>();  
    //i 可以理解成就是索引,扫描 stringBuilder  
    for(int i = 0; i < stringBuilder.length(); ) {  
        int count = 1; // 小的计数器  
        boolean flag = true;  
        Byte b = null;
```



```
while(flag) {  
    //1010100010111...  
    //递增的取出 key 1  
    String key = stringBuilder.substring(i, i+count); //i 不动，让 count 移动，指定匹配到一个字符  
    b = map.get(key);  
    if(b == null) {//说明没有匹配到  
        count++;  
    } else {  
        //匹配到  
        flag = false;  
    }  
    list.add(b);  
    i += count; //i 直接移动到 count  
}  
  
//当 for 循环结束后，我们 list 中就存放了所有的字符 "i like like like java do you like a java"  
//把 list 中的数据放入到 byte[] 并返回  
byte b[] = new byte[list.size()];  
for(int i = 0;i < b.length; i++) {  
    b[i] = list.get(i);  
}  
return b;  
}
```



```
/**  
 * 将一个 byte 转成一个二进制的字符串，如果看不懂，可以参考我讲的 Java 基础 二进制的原码，反码，  
补码  
 * @param b 传入的 byte  
 * @param flag 标志是否需要补高位如果是 true ，表示需要补高位，如果是 false 表示不补，如果是最后一个字节，无需补高位  
 * @return 是该 b 对应的二进制的字符串，（注意是按补码返回）  
 */  
  
private static String byteToBitString(boolean flag, byte b) {  
    //使用变量保存 b  
    int temp = b; //将 b 转成 int  
    //如果是正数我们还存在补高位  
    if(flag) {  
        temp |= 256; //按位与 256 1 0000 0000 | 0000 0001 => 1 0000 0001  
    }  
    String str = Integer.toBinaryString(temp); //返回的是 temp 对应的二进制的补码  
    if(flag) {  
        return str.substring(str.length() - 8);  
    } else {  
        return str;  
    }  
}  
  
//使用一个方法，将前面的方法封装起来，便于我们的调用.  
/**  
 *
```



```
* @param bytes 原始的字符串对应的字节数组
* @return 是经过 赫夫曼编码处理后的字节数组(压缩后的数组)
*/
private static byte[] huffmanZip(byte[] bytes) {
    List<Node> nodes = getNodes(bytes);
    //根据 nodes 创建的赫夫曼树
    Node huffmanTreeRoot = createHuffmanTree(nodes);
    //对应的赫夫曼编码(根据 赫夫曼树)
    Map<Byte, String> huffmanCodes = getCodeMap(huffmanTreeRoot);
    //根据生成的赫夫曼编码，压缩得到压缩后的赫夫曼编码字节数组
    byte[] huffmanCodeBytes = zip(bytes, huffmanCodes);
    return huffmanCodeBytes;
}
```

```
//编写一个方法，将字符串对应的 byte[] 数组，通过生成的赫夫曼编码表，返回一个赫夫曼编码 压缩后的
byte[]
/**
*
* @param bytes 这时原始的字符串对应的 byte[]
* @param huffmanCodes 生成的赫夫曼编码 map
* @return 返回赫夫曼编码处理后的 byte[]
* 举例： String content = "i like like like java do you like a java"; => byte[] contentBytes = content.getBytes();
*          返      回      的      是      字      符      串
"101010001011111100100010111111100100010111111100100101001101110001110000011011101000111100101000
101111111100110001001010011011100"
```



```
* => 对应的 byte[] huffmanCodeBytes , 即 8 位对应一个 byte, 放入到 huffmanCodeBytes
* huffmanCodeBytes[0] = 10101000(补码) => byte [推导 10101000=> 10101000 - 1 => 10100111(反
码)=> 11011000= -88 ]
* huffmanCodeBytes[1] = -88
*/
private static byte[] zip(byte[] bytes, Map<Byte, String> huffmanCodes) {

    //1.利用 huffmanCodes 将 bytes 转成 赫夫曼编码对应的字符串
    StringBuilder stringBuilder = new StringBuilder();
    //遍历 bytes 数组
    for(byte b: bytes) {
        stringBuilder.append(huffmanCodes.get(b));
    }

    //System.out.println("测试 stringBuilder~~~=" + stringBuilder.toString());

    //将 "10101000101111110..." 转成 byte[]
    //统计返回 byte[] huffmanCodeBytes 长度
    //一句话 int len = (stringBuilder.length() + 7) / 8;
    int len;
    if(stringBuilder.length() % 8 == 0) {
        len = stringBuilder.length() / 8;
    } else {
        len = stringBuilder.length() / 8 + 1;
    }
}
```



```
//创建 存储压缩后的 byte 数组
byte[] huffmanCodeBytes = new byte[len];
int index = 0;//记录是第几个 byte
for (int i = 0; i < stringBuilder.length(); i += 8) { //因为是每 8 位对应一个 byte,所以步长 +8
    String strByte;
    if(i+8 > stringBuilder.length()) {//不够 8 位
        strByte = stringBuilder.substring(i);
    } else{
        strByte = stringBuilder.substring(i, i + 8);
    }
    //将 strByte 转成一个 byte,放入到 huffmanCodeBytes
    huffmanCodeBytes[index] = (byte)Integer.parseInt(strByte, 2);
    index++;
}
return huffmanCodeBytes;
}

//生成赫夫曼树对应的赫夫曼编码
//思路:
//1. 将赫夫曼编码表存放在 Map<Byte, String> 形式
//   生成的赫夫曼编码表 {32=01, 97=100, 100=11000, 117=11001, 101=1110, 118=11011, 105=101,
121=11010, 106=0010, 107=1111, 108=000, 111=0011}
static Map<Byte, String> huffmanCodes = new HashMap<Byte, String>();
//2. 在生成赫夫曼编码表示, 需要去拼接路径, 定义一个 StringBuilder 存储某个叶子结点的路径
static StringBuilder stringBuilder = new StringBuilder();
```



```
//为了调用方便，我们重载 getCodes
private static Map<Byte, String> getCodes(Node root) {
    if(root == null) {
        return null;
    }
    //处理 root 的左子树
    getCodes(root.left, "0", stringBuilder);
    //处理 root 的右子树
    getCodes(root.right, "1", stringBuilder);
    return huffmanCodes;
}

/**
 * 功能：将传入的 node 结点的所有叶子结点的赫夫曼编码得到，并放入到 huffmanCodes 集合
 * @param node 传入结点
 * @param code 路径： 左子结点是 0, 右子结点 1
 * @param stringBuilder 用于拼接路径
 */
private static void getCodes(Node node, String code, StringBuilder stringBuilder) {
    StringBuilder stringBuilder2 = new StringBuilder(stringBuilder);
    //将 code 加入到 stringBuilder2
    stringBuilder2.append(code);
    if(node != null) { //如果 node == null 不处理
        //判断当前 node 是叶子结点还是非叶子结点
        if(node.data == null) { //非叶子结点
            if(node.left != null) {
                getCodes(node.left, code + "0", stringBuilder2);
            }
            if(node.right != null) {
                getCodes(node.right, code + "1", stringBuilder2);
            }
        } else {
            huffmanCodes.put((byte) node.data, stringBuilder2.toString());
        }
    }
}
```



```
//递归处理
    //向左递归
    getCodes(node.left, "0", stringBuilder2);
    //向右递归
    getCodes(node.right, "1", stringBuilder2);
} else { //说明是一个叶子结点
    //就表示找到某个叶子结点的最后
    huffmanCodes.put(node.data, stringBuilder2.toString());
}
}

}

//前序遍历的方法
private static void preOrder(Node root) {
    if(root != null) {
        root.preOrder();
    } else {
        System.out.println("赫夫曼树为空");
    }
}

/**
 *
 * @param bytes 接收字节数组
 * @return 返回的就是 List 形式 [Node[date=97 ,weight = 5], Node[]date=32,weight = 9].....],
 */

```



```
private static List<Node> getNodes(byte[] bytes) {  
  
    //1 创建一个 ArrayList  
    ArrayList<Node> nodes = new ArrayList<Node>();  
  
    //遍历 bytes，统计 每一个 byte 出现的次数->map[key,value]  
    Map<Byte, Integer> counts = new HashMap<>();  
    for (byte b : bytes) {  
        Integer count = counts.get(b);  
        if (count == null) { // Map 还没有这个字符数据,第一次  
            counts.put(b, 1);  
        } else {  
            counts.put(b, count + 1);  
        }  
    }  
  
    //把每一个键值对转成一个 Node 对象，并加入到 nodes 集合  
    //遍历 map  
    for(Map.Entry<Byte, Integer> entry: counts.entrySet()) {  
        nodes.add(new Node(entry.getKey(), entry.getValue()));  
    }  
    return nodes;  
}  
  
//可以通过 List 创建对应的赫夫曼树
```



```
private static Node createHuffmanTree(List<Node> nodes) {  
  
    while(nodes.size() > 1) {  
        //排序，从小到大  
        Collections.sort(nodes);  
        //取出第一颗最小的二叉树  
        Node leftNode = nodes.get(0);  
        //取出第二颗最小的二叉树  
        Node rightNode = nodes.get(1);  
        //创建一颗新的二叉树,它的根节点 没有 data, 只有权值  
        Node parent = new Node(null, leftNode.weight + rightNode.weight);  
        parent.left = leftNode;  
        parent.right = rightNode;  
  
        //将已经处理的两颗二叉树从 nodes 删除  
        nodes.remove(leftNode);  
        nodes.remove(rightNode);  
        //将新的二叉树，加入到 nodes  
        nodes.add(parent);  
  
    }  
    //nodes 最后的结点，就是赫夫曼树的根结点  
    return nodes.get(0);  
}
```



```
}
```

```
//创建 Node ,待数据和权值
class Node implements Comparable<Node> {
    Byte data; // 存放数据(字符)本身, 比如'a'=>97 ''=>32
    int weight; //权值, 表示字符出现的次数
    Node left;//
    Node right;
    public Node(Byte data, int weight) {

        this.data = data;
        this.weight = weight;
    }
    @Override
    public int compareTo(Node o) {
        // 从小到大排序
        return this.weight - o.weight;
    }

    public String toString() {
        return "Node [data = " + data + " weight=" + weight + "]";
    }
}
```



```
//前序遍历  
public void preOrder() {  
    System.out.println(this);  
    if(this.left != null) {  
        this.left.preOrder();  
    }  
    if(this.right != null) {  
        this.right.preOrder();  
    }  
}
```

11.3.9 赫夫曼编码压缩文件注意事项

- 1) 如果文件本身就是经过压缩处理的，那么使用赫夫曼编码再压缩效率不会有明显变化，比如视频,ppt 等等文件
[举例压一个 .ppt]
- 2) 赫夫曼编码是按字节来处理的，因此可以处理所有的文件(二进制文件、文本文件) [举例压一个.xml 文件]
- 3) 如果一个文件中的内容，重复的数据不多，压缩效果也不会很明显。

11.4 二叉排序树

11.4.1 先看一个需求

给你一个数列 (7, 3, 10, 12, 5, 1, 9)，要求能够高效的完成对数据的查询和添加

11.4.2 解决方案分析

➤ 使用数组

数组未排序，优点：直接在数组尾添加，速度快。缺点：查找速度慢。[示意图]

数组排序，优点：可以使用二分查找，查找速度快，缺点：为了保证数组有序，在添加新数据时，找到插入位置后，后面的数据需整体移动，速度慢。[示意图]

➤ 使用链式存储-链表

不管链表是否有序，查找速度都慢，添加数据速度比数组快，不需要数据整体移动。[示意图]

➤ 使用二叉排序树

11.4.3 二叉排序树介绍

二叉排序树： BST: (Binary Sort(Search) Tree)，对于二叉排序树的任何一个非叶子节点，要求左子节点的值比当前节点的值小，右子节点的值比当前节点的值大。

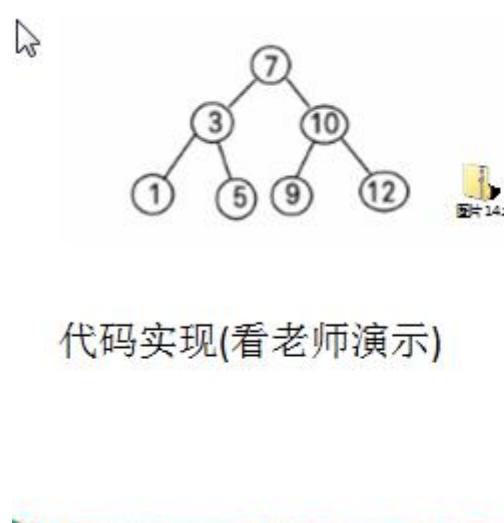
特别说明：如果有相同的值，可以将该节点放在左子节点或右子节点

比如针对前面的数据 (7, 3, 10, 12, 5, 1, 9) ，对应的二叉排序树为：



11.4.4 二叉排序树创建和遍历

一个数组创建成对应的二叉排序树，并使用中序遍历二叉排序树，比如：数组为 Array(7, 3, 10, 12, 5, 1, 9)，创建成对应的二叉排序树为：

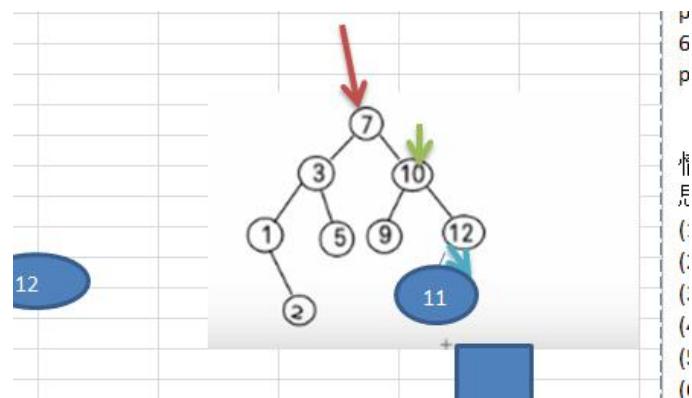


```
public void add(Node node) {           // 向二叉排序树添加节点
    if (node == null) {
        return;
    }
    if (node.value < this.value) {       // 如果左节点为空
        if (this.left == null) {
            this.left = node;
        } else {                         // 如果不为空,递归添加
            this.left.add(node);
        }
    } else {                           // 如果右节点为空
        if (this.right == null) {
            this.right = node;
        } else {                         // 如果不为空,递归添加
            this.right.add(node);
        }
    }
}
```

11.4.5 二叉排序树的删除

二叉排序树的删除情况比较复杂，有下面三种情况需要考虑

- 1) 删除叶子节点 (比如： 2, 5, 9, 12)
- 2) 删除只有一颗子树的节点 (比如： 1)
- 3) 删除有两颗子树的节点 (比如： 7, 3, 10)
- 4) 操作的思路分析



//对删除结点的各种情况的思路分析:

第一种情况:

删除叶子节点 (比如: 2, 5, 9, 12)

思路

- (1) 需求先去找到要删除的结点 targetNode
- (2) 找到 targetNode 的 父结点 parent
- (3) 确定 targetNode 是 parent 的左子结点 还是右子结点
- (4) 根据前面的情况来对应删除

左子结点 parent.left = null

右子结点 parent.right = null;

第二种情况: 删除只有一颗子树的节点 比如 1

思路

- (1) 需求先去找到要删除的结点 targetNode
 - (2) 找到 targetNode 的 父结点 parent
 - (3) 确定 targetNode 的子结点是左子结点还是右子结点
 - (4) targetNode 是 parent 的左子结点还是右子结点
 - (5) 如果 targetNode 有左子结点
5. 1 如果 targetNode 是 parent 的左子结点



```
parent.left = targetNode.left;
```

5.2 如果 targetNode 是 parent 的右子结点

```
parent.right = targetNode.left;
```

(6) 如果 targetNode 有右子结点

6.1 如果 targetNode 是 parent 的左子结点

```
parent.left = targetNode.right;
```

6.2 如果 targetNode 是 parent 的右子结点

```
parent.right = targetNode.right
```

情况三：删除有两颗子树的节点。(比如：7, 3, 10)

思路

- (1) 需求先去找到要删除的结点 targetNode
- (2) 找到 targetNode 的父结点 parent
- (3) 从 targetNode 的右子树找到最小的结点
- (4) 用一个临时变量，将 最小结点的值保存 temp = 11
- (5) 删除该最小结点
- (6) targetNode.value = temp

11.4.6 二叉排序树删除结点的代码实现：

```
package com.atguigu.binarysorttree;
```



```
public class BinarySortTreeDemo {  
  
    public static void main(String[] args) {  
        int[] arr = {7, 3, 10, 12, 5, 1, 9, 2};  
        BinarySortTree binarySortTree = new BinarySortTree();  
        //循环的添加结点到二叉排序树  
        for(int i = 0; i < arr.length; i++) {  
            binarySortTree.add(new Node(arr[i]));  
        }  
  
        //中序遍历二叉排序树  
        System.out.println("中序遍历二叉排序树~");  
        binarySortTree.infixOrder(); // 1, 3, 5, 7, 9, 10, 12  
  
        //测试一下删除叶子结点  
  
        binarySortTree.delNode(12);  
  
        binarySortTree.delNode(5);  
        binarySortTree.delNode(10);  
        binarySortTree.delNode(2);  
        binarySortTree.delNode(3);  
  
        binarySortTree.delNode(9);  
    }  
}
```



```
binarySortTree.delNode(1);
binarySortTree.delNode(7);

System.out.println("root=" + binarySortTree.getRoot());

System.out.println("删除结点后");
binarySortTree.infixOrder();
}

}

//创建二叉排序树
class BinarySortTree {
    private Node root;

    public Node getRoot() {
        return root;
    }

    //查找到要删除的结点
    public Node search(int value) {
```



```
if(root == null) {  
    return null;  
}  
else {  
    return root.search(value);  
}  
  
}  
  
//查找父结点  
public Node searchParent(int value) {  
    if(root == null) {  
        return null;  
    } else {  
        return root.searchParent(value);  
    }  
}  
  
//编写方法:  
//1. 返回的 以 node 为根结点的二叉排序树的最小结点的值  
//2. 删除 node 为根结点的二叉排序树的最小结点  
/**  
 *  
 * @param node 传入的结点(当做二叉排序树的根结点)  
 * @return 返回的 以 node 为根结点的二叉排序树的最小结点的值  
 */  
public int delRightTreeMin(Node node) {  
    Node target = node;
```



```
//循环的查找左子节点，就会找到最小值
while(target.left != null) {
    target = target.left;
}

//这时 target 就指向了最小结点
//删除最小结点
delNode(target.value);
return target.value;
}

//删除结点
public void delNode(int value) {
    if(root == null) {
        return;
    }else {
        //1.需求先去找到要删除的结点 targetNode
        Node targetNode = search(value);
        //如果没有找到要删除的结点
        if(targetNode == null) {
            return;
        }
        //如果我们发现当前这颗二叉排序树只有一个结点
        if(root.left == null && root.right == null) {
            root = null;
            return;
        }
    }
}
```



```
}
```

```
//去找到 targetNode 的父结点
Node parent = searchParent(value);

//如果要删除的结点是叶子结点
if(targetNode.left == null && targetNode.right == null) {
    //判断 targetNode 是父结点的左子结点，还是右子结点
    if(parent.left != null && parent.left.value == value) { //是左子结点
        parent.left = null;
    } else if (parent.right != null && parent.right.value == value) { //是右子结点
        parent.right = null;
    }
} else if (targetNode.left != null && targetNode.right != null) { //删除有两颗子树的节点
    int minVal = delRightTreeMin(targetNode.right);
    targetNode.value = minVal;
}

} else { // 删除只有一颗子树的结点
    //如果要删除的结点有左子结点
    if(targetNode.left != null) {
        if(parent != null) {
            //如果 targetNode 是 parent 的左子结点
            if(parent.left.value == value) {
                parent.left = targetNode.left;
            } else { // targetNode 是 parent 的右子结点
                parent.right = targetNode.left;
            }
        }
    }
}
```



```
        }

    } else {
        root = targetNode.left;
    }

} else { //如果要删除的结点有右子结点
    if(parent != null) {
        //如果 targetNode 是 parent 的左子结点
        if(parent.left.value == value) {
            parent.left = targetNode.right;
        } else { //如果 targetNode 是 parent 的右子结点
            parent.right = targetNode.right;
        }
    } else {
        root = targetNode.right;
    }
}

}

//添加结点的方法
public void add(Node node) {
    if(root == null) {
        root = node; //如果 root 为空则直接让 root 指向 node
    }
}
```



```
    } else {
        root.add(node);
    }
}

//中序遍历

public void infixOrder() {
    if(root != null) {
        root.infixOrder();
    } else {
        System.out.println("二叉排序树为空，不能遍历");
    }
}

//创建 Node 结点

class Node {
    int value;
    Node left;
    Node right;
    public Node(int value) {
        this.value = value;
    }
}

//查找要删除的结点
```



```
/**  
 *  
 * @param value 希望删除的结点的值  
 * @return 如果找到返回该结点, 否则返回 null  
 */  
  
public Node search(int value) {  
    if(value == this.value) { //找到就是该结点  
        return this;  
    } else if(value < this.value) { //如果查找的值小于当前结点, 向左子树递归查找  
        //如果左子结点为空  
        if(this.left == null) {  
            return null;  
        }  
        return this.left.search(value);  
    } else { //如果查找的值不小于当前结点, 向右子树递归查找  
        if(this.right == null) {  
            return null;  
        }  
        return this.right.search(value);  
    }  
}  
  
//查找到要删除结点的父结点  
/**  
 *  
 * @param value 要找到的结点的值  
 */
```



```
* @return 返回的是要删除的结点的父结点，如果没有就返回 null
*/
public Node searchParent(int value) {
    //如果当前结点就是要删除的结点的父结点，就返回
    if((this.left != null && this.left.value == value) ||
        (this.right != null && this.right.value == value)) {
        return this;
    } else {
        //如果查找的值小于当前结点的值，并且当前结点的左子结点不为空
        if(value < this.value && this.left != null) {
            return this.left.searchParent(value); //向左子树递归查找
        } else if (value >= this.value && this.right != null) {
            return this.right.searchParent(value); //向右子树递归查找
        } else {
            return null; // 没有找到父结点
        }
    }
}

@Override
public String toString() {
    return "Node [value=" + value + "]";
}
```



```
//添加结点的方法  
//递归的形式添加结点，注意需要满足二叉排序树的要求  
public void add(Node node) {  
    if(node == null) {  
        return;  
    }  
  
    //判断传入的结点的值， 和当前子树的根结点的值关系  
    if(node.value < this.value) {  
        //如果当前结点左子结点为 null  
        if(this.left == null) {  
            this.left = node;  
        } else {  
            //递归的向左子树添加  
            this.left.add(node);  
        }  
    } else { //添加的结点的值大于 当前结点的值  
        if(this.right == null) {  
            this.right = node;  
        } else {  
            //递归的向右子树添加  
            this.right.add(node);  
        }  
    }  
}
```



```
//中序遍历
public void infixOrder() {
    if(this.left != null) {
        this.left.infixOrder();
    }
    System.out.println(this);
    if(this.right != null) {
        this.right.infixOrder();
    }
}
```

11.4.7 课后练习：完成老师代码，并使用第二种方式来解决

如果我们从左子树找到最大的结点，然后前面的思路完成.

11.5 平衡二叉树(AVL 树)

11.5.1 看一个案例(说明二叉排序树可能的问题)

给你一个数列{1,2,3,4,5,6}，要求创建一颗二叉排序树(BST)，并分析问题所在.

➤ 左边 BST 存在的问题分析：

- 1) 左子树全部为空，从形式上看，更像一个单链表.
- 2) 插入速度没有影响
- 3) 查询速度明显降低(因为需要依次比较)，不能发挥 BST

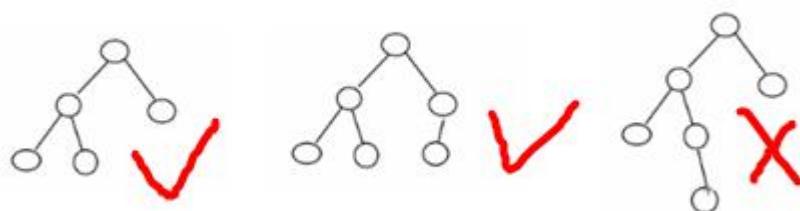
的优势，因为每次还需要比较左子树，其查询速度比

单链表还慢

4) 解决方案-平衡二叉树(AVL)

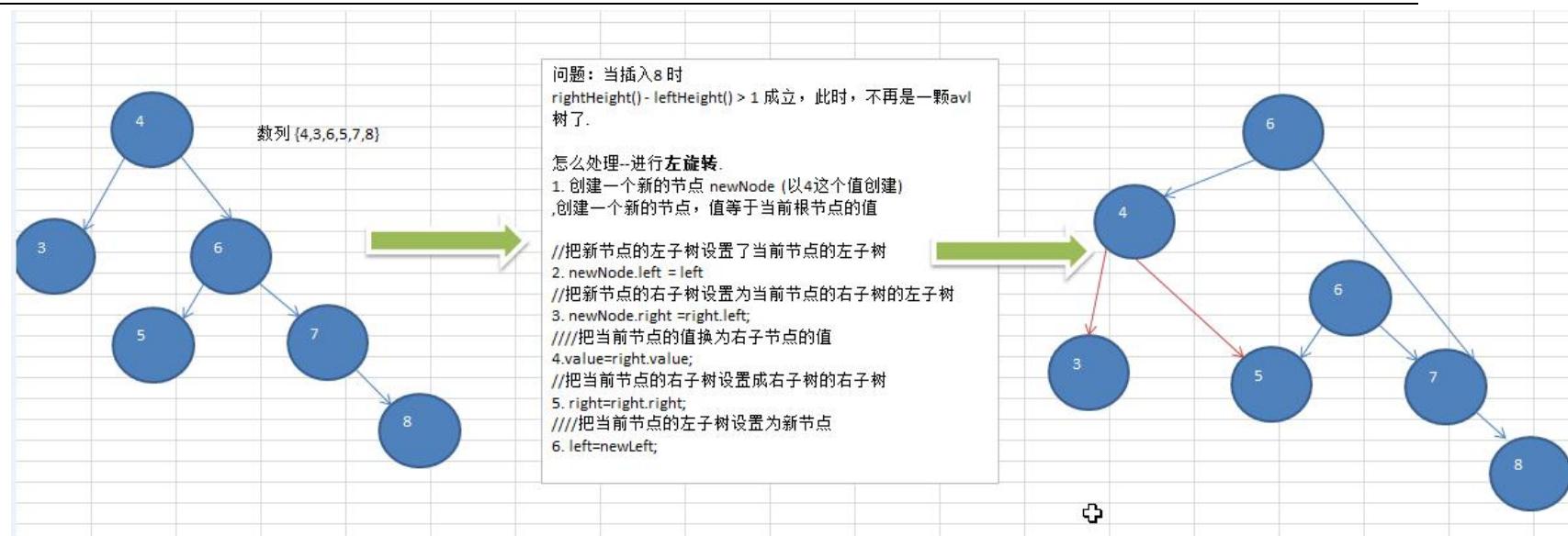
11.5.2 基本介绍

- 1) 平衡二叉树也叫平衡二叉搜索树 (Self-balancing binary search tree) 又被称为 AVL 树，可以保证查询效率较高。
- 2) 具有以下特点：它是一棵空树或它的左右两个子树的高度差的绝对值不超过 1，并且左右两个子树都是一棵平衡二叉树。平衡二叉树的常用实现方法有红黑树、AVL、替罪羊树、Treap、伸展树等。
- 3) 举例说明，看看下面哪些 AVL 树，为什么？



11.5.3 应用案例-单旋转(左旋转)

- 1) 要求：给你一个数列，创建出对应的平衡二叉树. 数列 {4,3,6,5,7,8}
- 2) 思路分析(示意图)



3) 代码实现

```
//左旋转方法

private void leftRotate() {

    //创建新的结点，以当前根结点的值
    Node newNode = new Node(value);

    //把新的结点的左子树设置成当前结点的左子树
    newNode.left = left;

    //把新的结点的右子树设置成带你过去结点的右子树的左子树
    newNode.right = right.left;

    //把当前结点的值替换成右子结点的值
    value = right.value;

    //把当前结点的右子树设置成当前结点右子树的右子树
    right = right.right;

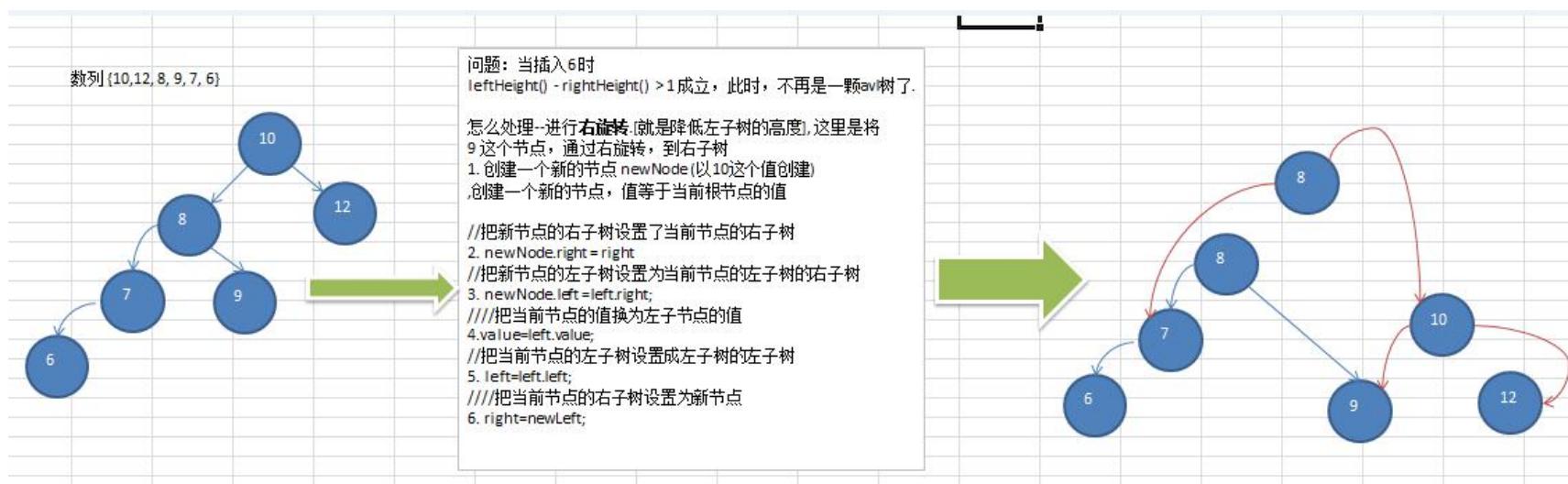
    //把当前结点的左子树(左子结点)设置成新的结点
    left = newNode;
}
```

```
}
```

11.5.4 应用案例-单旋转(右旋转)

1) 要求: 给你一个数列, 创建出对应的平衡二叉树. 数列 {10,12, 8, 9, 7, 6}

2) 思路分析(示意图)



3) 代码实现

```
//右旋转
private void rightRotate() {
    Node newNode = new Node(value);
    newNode.right = right;
    newNode.left = left.right;
    value = left.value;
    left = left.left;
    right = newNode;
}
```

}

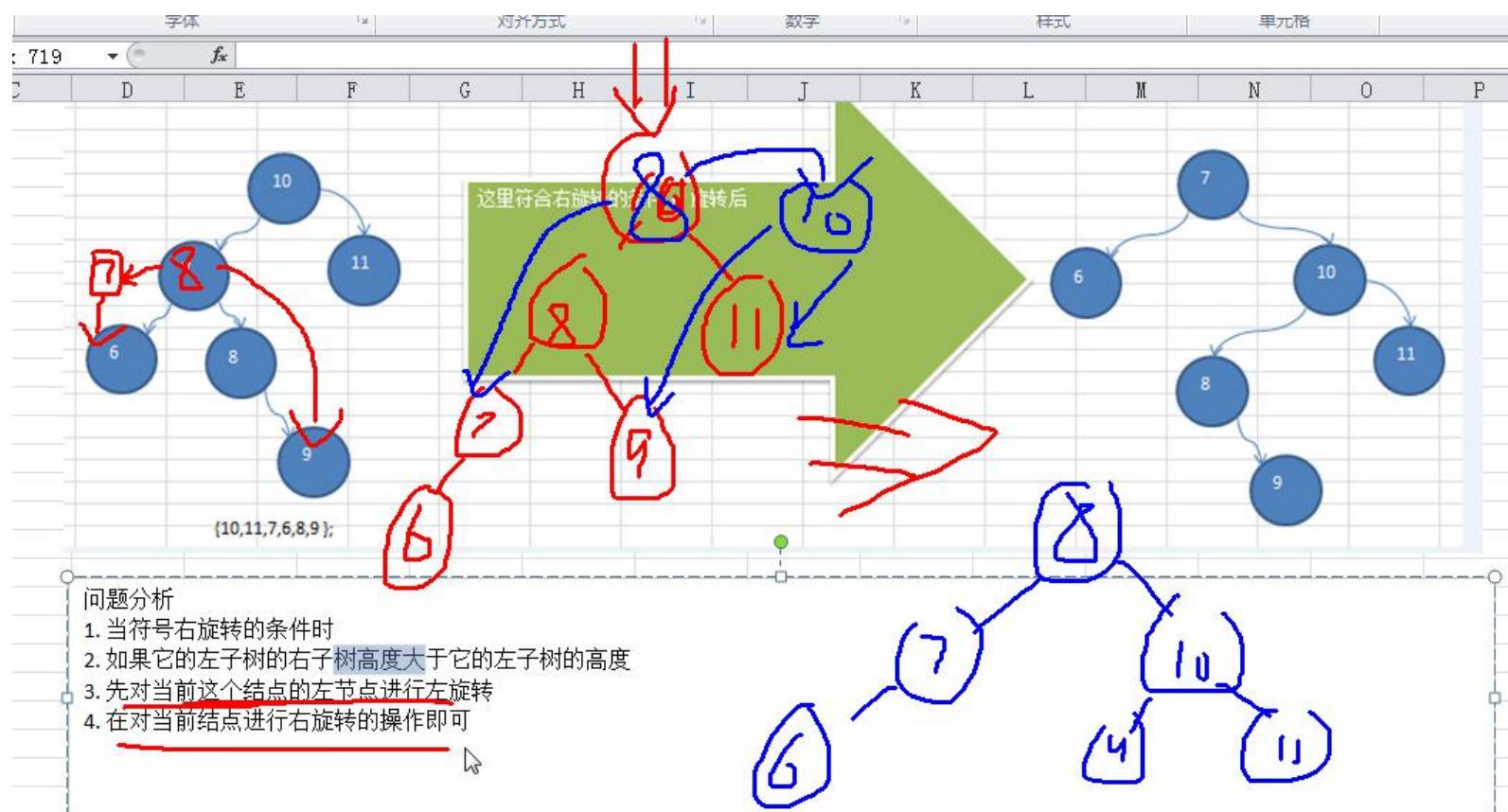
11.5.5 应用案例-双旋转

前面的两个数列，进行单旋转(即一次旋转)就可以将非平衡二叉树转成平衡二叉树,但是在某些情况下，单旋转不能完成平衡二叉树的转换。比如数列

```
int[] arr = { 10, 11, 7, 6, 8, 9 }; 运行原来的代码可以看到，并没有转成 AVL 树.
```

```
int[] arr = {2,1,6,5,7,3}; // 运行原来的代码可以看到，并没有转成 AVL 树
```

1) 问题分析



2) 解决思路分析

1. 当符号右旋转的条件时
2. 如果它的左子树的右子树高度大于它的左子树的高度



3. 先对当前这个结点的左节点进行左旋转
 4. 在对当前结点进行右旋转的操作即可
- 3) 代码实现[AVL 树的汇总代码(完整代码)]

```
package com.atguigu.avl;

public class AVLTreeDemo {

    public static void main(String[] args) {
        //int[] arr = {4,3,6,5,7,8};
        //int[] arr = { 10, 12, 8, 9, 7, 6 };
        int[] arr = { 10, 11, 7, 6, 8, 9 };
        //创建一个 AVLTree 对象
        AVLTree avlTree = new AVLTree();
        //添加结点
        for(int i=0; i < arr.length; i++) {
            avlTree.add(new Node(arr[i]));
        }

        //遍历
        System.out.println("中序遍历");
        avlTree.infixOrder();

        System.out.println("在平衡处理~");
        System.out.println("树的高度=" + avlTree.getRoot().height()); //3
        System.out.println("树的左子树高度=" + avlTree.getRoot().leftHeight()); // 2
        System.out.println("树的右子树高度=" + avlTree.getRoot().rightHeight()); // 2
        System.out.println("当前的根结点=" + avlTree.getRoot());//8
    }
}
```



```
}

}

// 创建 AVLTree
class AVLTree {
    private Node root;

    public Node getRoot() {
        return root;
    }

    // 查找要删除的结点
    public Node search(int value) {
        if (root == null) {
            return null;
        } else {
            return root.search(value);
        }
    }

    // 查找父结点
    public Node searchParent(int value) {
        if (root == null) {
```



```
        return null;  
    } else {  
        return root.searchParent(value);  
    }  
}  
  
// 编写方法:  
// 1. 返回的 以 node 为根结点的二叉排序树的最小结点的值  
// 2. 删除 node 为根结点的二叉排序树的最小结点  
  
/**  
 *  
 * @param node  
 *      传入的结点(当做二叉排序树的根结点)  
 * @return 返回的 以 node 为根结点的二叉排序树的最小结点的值  
 */  
  
public int delRightTreeMin(Node node) {  
    Node target = node;  
    // 循环的查找左子节点，就会找到最小值  
    while (target.left != null) {  
        target = target.left;  
    }  
    // 这时 target 就指向了最小结点  
    // 删除最小结点  
    delNode(target.value);  
    return target.value;  
}
```



```
// 删除结点
public void delNode(int value) {
    if (root == null) {
        return;
    } else {
        // 1.需求先去找到要删除的结点 targetNode
        Node targetNode = search(value);
        // 如果没有找到要删除的结点
        if (targetNode == null) {
            return;
        }
        // 如果我们发现当前这颗二叉排序树只有一个结点
        if (root.left == null && root.right == null) {
            root = null;
            return;
        }

        // 去找到 targetNode 的父结点
        Node parent = searchParent(value);
        // 如果要删除的结点是叶子结点
        if (targetNode.left == null && targetNode.right == null) {
            // 判断 targetNode 是父结点的左子结点，还是右子结点
            if (parent.left != null && parent.left.value == value) { // 是左子结点
                parent.left = null;
            } else if (parent.right != null && parent.right.value == value) { // 是右子结点
                parent.right = null;
            }
        }
    }
}
```



```
parent.right = null;  
}  
} else if (targetNode.left != null && targetNode.right != null) { // 删除有两颗子树的节点  
    int minVal = delRightTreeMin(targetNode.right);  
    targetNode.value = minVal;  
  
} else { // 删除只有一颗子树的结点  
    // 如果要删除的结点有左子结点  
    if (targetNode.left != null) {  
        if (parent != null) {  
            // 如果 targetNode 是 parent 的左子结点  
            if (parent.left.value == value) {  
                parent.left = targetNode.left;  
            } else { // targetNode 是 parent 的右子结点  
                parent.right = targetNode.left;  
            }  
        } else {  
            root = targetNode.left;  
        }  
    } else { // 如果要删除的结点有右子结点  
        if (parent != null) {  
            // 如果 targetNode 是 parent 的左子结点  
            if (parent.left.value == value) {  
                parent.left = targetNode.right;  
            } else { // 如果 targetNode 是 parent 的右子结点  
                parent.right = targetNode.right;  
            }  
        }  
    }  
}
```





```
        }

    }

}

// 创建 Node 结点

class Node {

    int value;
    Node left;
    Node right;

    public Node(int value) {

        this.value = value;
    }

    // 返回左子树的高度
    public int leftHeight() {
        if (left == null) {
            return 0;
        }
        return left.height();
    }

    // 返回右子树的高度
    public int rightHeight() {
        if (right == null) {
```



```
        return 0;  
    }  
    return right.height();  
}  
  
// 返回 以该结点为根结点的树的高度  
public int height() {  
    return Math.max(left == null ? 0 : left.height(), right == null ? 0 : right.height()) + 1;  
}  
  
//左旋转方法  
private void leftRotate() {  
  
    //创建新的结点， 以当前根结点的值  
    Node newNode = new Node(value);  
    //把新的结点的左子树设置成当前结点的左子树  
    newNode.left = left;  
    //把新的结点的右子树设置成带你过去结点的右子树的左子树  
    newNode.right = right.left;  
    //把当前结点的值替换成右子结点的值  
    value = right.value;  
    //把当前结点的右子树设置成当前结点右子树的右子树  
    right = right.right;  
    //把当前结点的左子树(左子结点)设置成新的结点  
    left = newNode;
```



```
}
```

```
//右旋转
```

```
private void rightRotate() {  
    Node newNode = new Node(value);  
    newNode.right = right;  
    newNode.left = left.right;  
    value = left.value;  
    left = left.left;  
    right = newNode;  
}
```

```
// 查找要删除的结点
```

```
/**  
 *  
 * @param value  
 *      希望删除的结点的值  
 * @return 如果找到返回该结点，否则返回 null  
 */
```

```
public Node search(int value) {  
    if (value == this.value) { // 找到就是该结点  
        return this;  
    } else if (value < this.value) { // 如果查找的值小于当前结点，向左子树递归查找  
        // 如果左子结点为空  
        if (this.left == null) {
```



```
        return null;
    }

    return this.left.search(value);

} else { // 如果查找的值不小于当前结点，向右子树递归查找
    if (this.right == null) {
        return null;
    }

    return this.right.search(value);
}

}

// 查找要删除结点的父结点
/**
 *
 * @param value
 *      要找到的结点的值
 * @return 返回的是要删除的结点的父结点，如果没有就返回 null
 */
public Node searchParent(int value) {
    // 如果当前结点就是要删除的结点的父结点，就返回
    if ((this.left != null && this.left.value == value) || (this.right != null && this.right.value == value)) {
        return this;
    } else {
        // 如果查找的值小于当前结点的值，并且当前结点的左子结点不为空
        if (value < this.value && this.left != null) {

```



```
        return this.left.searchParent(value); // 向左子树递归查找
    } else if (value >= this.value && this.right != null) {
        return this.right.searchParent(value); // 向右子树递归查找
    } else {
        return null; // 没有找到父结点
    }
}

@Override
public String toString() {
    return "Node [value=" + value + "]";
}

// 添加结点的方法
// 递归的形式添加结点，注意需要满足二叉排序树的要求
public void add(Node node) {
    if (node == null) {
        return;
    }

    // 判断传入的结点的值，和当前子树的根结点的值关系
    if (node.value < this.value) {
        // 如果当前结点左子结点为 null
        if (this.left == null) {
```



```
this.left = node;  
} else {  
    // 递归的向左子树添加  
    this.left.add(node);  
}  
} else { // 添加的结点的值大于 当前结点的值  
    if (this.right == null) {  
        this.right = node;  
    } else {  
        // 递归的向右子树添加  
        this.right.add(node);  
    }  
}  
  
//当添加完一个结点后，如果: (右子树的高度-左子树的高度) > 1， 左旋转  
if(rightHeight() - leftHeight() > 1) {  
    //如果它的右子树的左子树的高度大于它的右子树的右子树的高度  
    if(right != null && right.leftHeight() > right.rightHeight()) {  
        //先对右子结点进行右旋转  
        right.rightRotate();  
        //然后在对当前结点进行左旋转  
        leftRotate(); //左旋转..  
    } else {  
        //直接进行左旋转即可  
        leftRotate();  
    }  
}
```



```
}

return ; //必须要!!!

}

//当添加完一个结点后，如果 (左子树的高度 - 右子树的高度) > 1, 右旋转
if(leftHeight() - rightHeight() > 1) {
    //如果它的左子树的右子树高度大于它的左子树的高度
    if(left != null && left.rightHeight() > left.leftHeight()) {
        //先对当前结点的左结点(左子树)->左旋转
        left.leftRotate();
        //再对当前结点进行右旋转
        rightRotate();
    } else {
        //直接进行右旋转即可
        rightRotate();
    }
}

}

// 中序遍历
public void infixOrder() {
    if (this.left != null) {
        this.left.infixOrder();
    }
    System.out.println(this);
    if (this.right != null) {
```



```
this.right.infixOrder();  
}  
}  
}
```

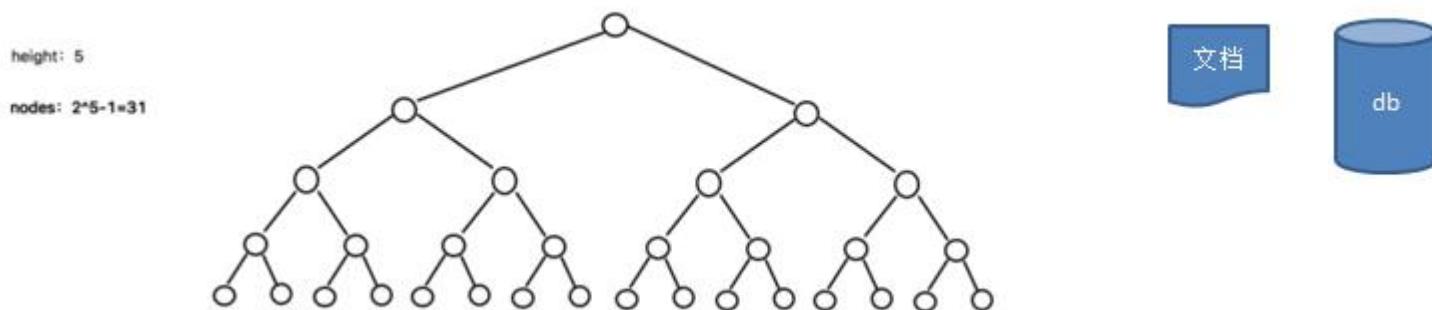
第 12 章 多路查找树

12.1 二叉树与 B 树

12.1.1 二叉树的问题分析

二叉树的操作效率较高，但是也存在问题，请看下面的二叉树

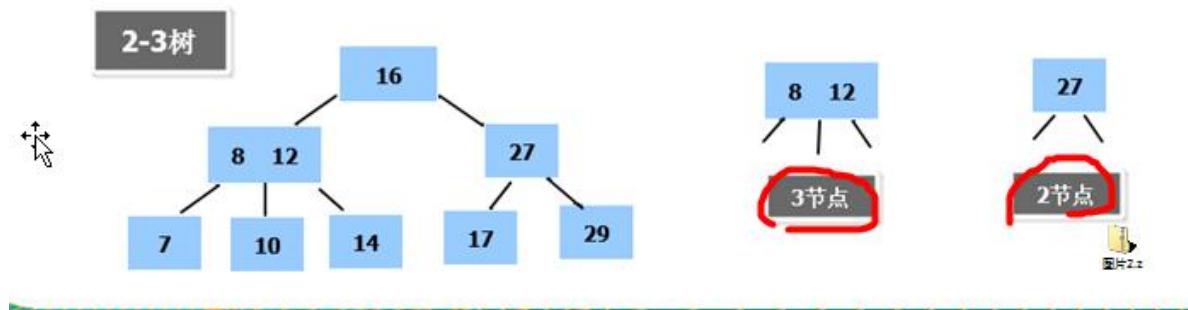
——二叉树的深度很大，因此查询速度慢，而且占用内存大



- 1) 二叉树需要加载到内存的，如果二叉树的节点少，没有什么问题，但是如果二叉树的节点很多(比如 1 亿)，就存在如下问题：
- 2) 问题 1：在构建二叉树时，需要多次进行 i/o 操作(海量数据存在数据库或文件中)，节点海量，构建二叉树时，速度有影响
- 3) 问题 2：节点海量，也会造成二叉树的高度很大，会降低操作速度.

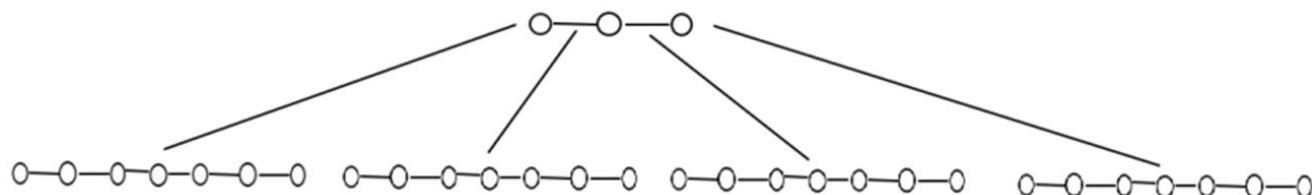
12.1.2 多叉树

- 1) 在二叉树中，每个节点有数据项，最多有两个子节点。如果允许每个节点可以有更多的数据项和更多的子节点，就是多叉树（multiway tree）
- 2) 后面我们讲解的 2-3 树，2-3-4 树就是多叉树，多叉树通过重新组织节点，减少树的高度，能对二叉树进行优化。
- 3) 举例说明(下面 2-3 树就是一颗多叉树)



12.1.3 B 树的基本介绍

B 树通过重新组织节点，降低树的高度，并且减少 i/o 读写次数来提升效率。



- 1) 如图 B 树通过重新组织节点，降低了树的高度。
- 2) 文件系统及数据库系统的设计者利用了磁盘预读原理，将一个节点的大小设为等于一个页(页得大小通常为 4k)，这样每个节点只需要一次 I/O 就可以完全载入
- 3) 将树的度 M 设置为 1024，在 600 亿个元素中最多只需要 4 次 I/O 操作就可以读取到想要的元素，B 树(B+)广泛应用于文件存储系统以及数据库系统中

12.2 2-3 树

12.2.1 2-3 树是最简单的 B 树结构，具有如下特点：

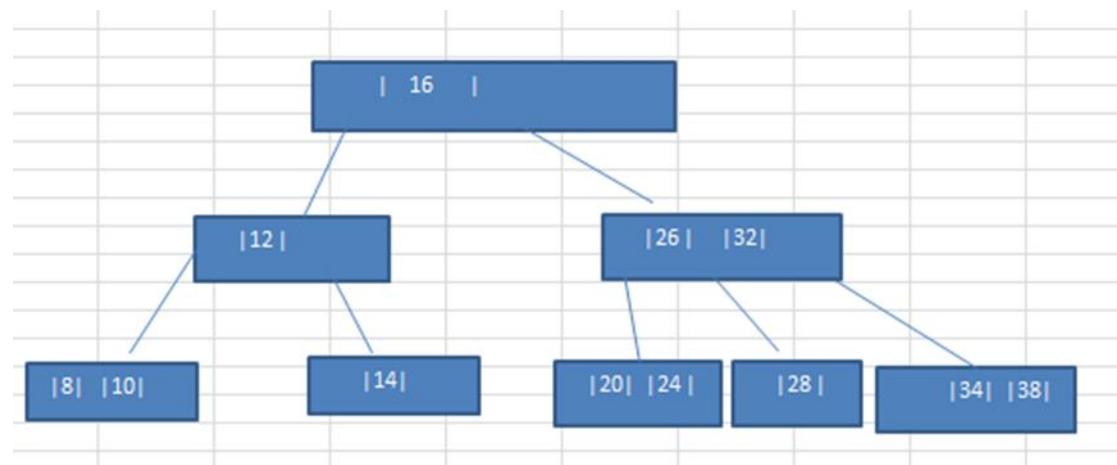
- 1) 2-3 树的所有叶子节点都在同一层。(只要是 B 树都满足这个条件)
- 2) 有两个子节点的节点叫二节点，二节点要么没有子节点，要么有两个子节点。

3) 有三个子节点的节点叫三节点，三节点要么没有子节点，要么有三个子节点。

4) 2-3 树是由二节点和三节点构成的树。

12.2.2 2-3 树应用案例

将数列{16, 24, 12, 32, 14, 26, 34, 10, 8, 28, 38, 20} 构建成 2-3 树，并保证数据插入的大小顺序。(演示一下构建 2-3 树的过程。)

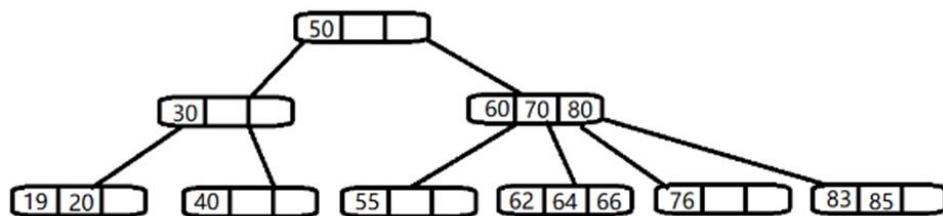


插入规则：

- 1) 2-3 树的所有叶子节点都在同一层。(只要是 B 树都满足这个条件)
- 2) 有两个子节点的节点叫二节点，二节点要么没有子节点，要么有两个子节点。
- 3) 有三个子节点的节点叫三节点，三节点要么没有子节点，要么有三个子节点
- 4) 当按照规则插入一个数到某个节点时，不能满足上面三个要求，就需要拆，先向上拆，如果上层满，则拆本层，拆后仍然需要满足上面 3 个条件。
- 5) 对于三节点的子树的值大小仍然遵守(BST 二叉排序树)的规则

12.2.3 其它说明

除了 23 树，还有 234 树等，概念和 23 树类似，也是一种 B 树。如图：



一棵2-3-4树

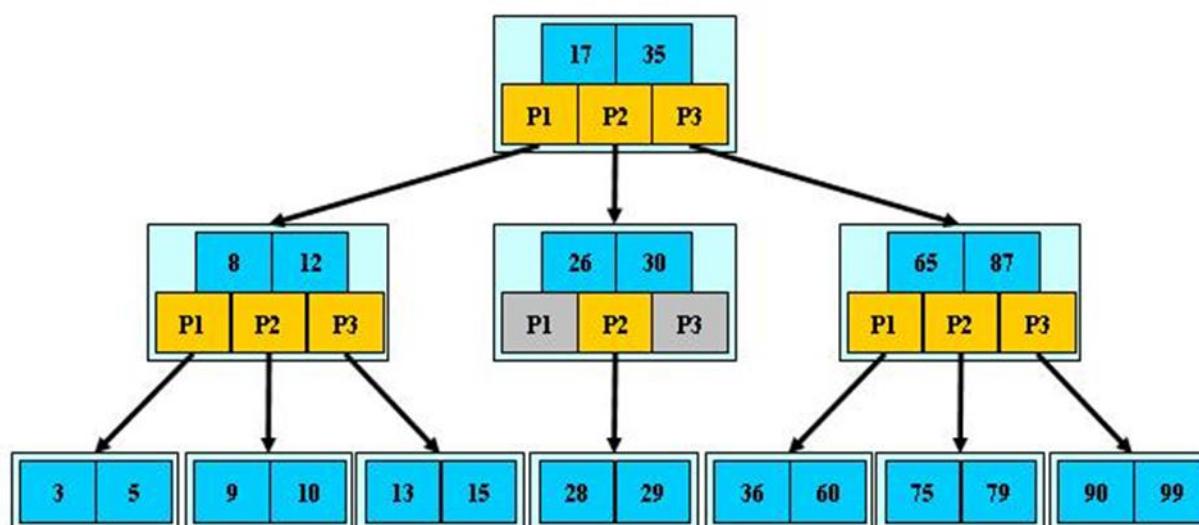
12.3 B 树、B+树和 B*树

12.3.1 B 树的介绍

B-tree 树即 B 树，B 即 Balanced，平衡的意思。有人把 B-tree 翻译成 B-树，容易让人产生误解。会以为 B-树是一种树，而 B 树又是另一种树。实际上，B-tree 就是指的 B 树。

12.3.2 B 树的介绍

前面已经介绍了 2-3 树和 2-3-4 树，他们就是 B 树(英语：B-tree 也写成 B-树)，这里我们再做一个说明，我们在学习 Mysql 时，经常听到说某种类型的索引是基于 B 树或者 B+树的，如图：



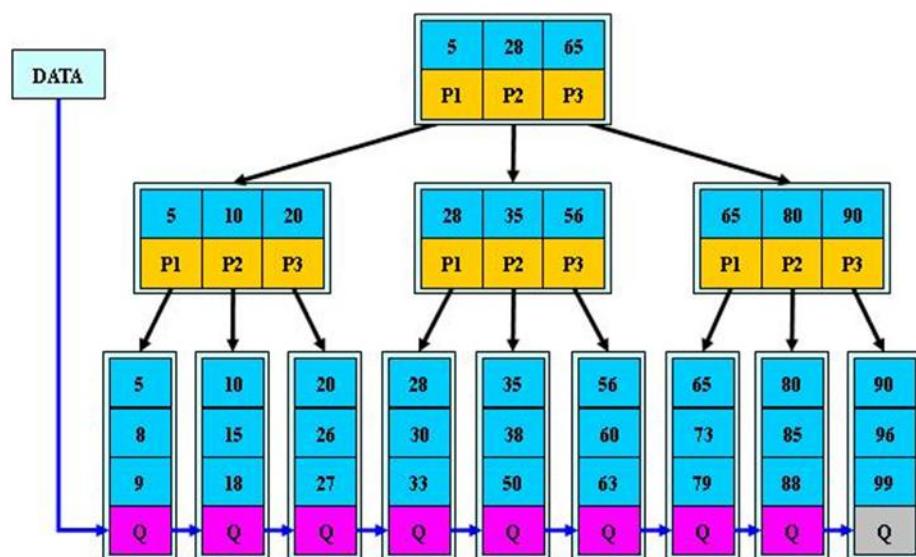
对上图的说明：

- 1) B 树的阶：节点的最多子节点个数。比如 2-3 树的阶是 3，2-3-4 树的阶是 4

- 2) B-树的搜索，从根结点开始，对结点内的关键字（有序）序列进行二分查找，如果命中则结束，否则进入查询关键字所属范围的儿子结点；重复，直到所对应的儿子指针为空，或已经是叶子结点
- 3) 关键字集合分布在整颗树中，即叶子节点和非叶子节点都存放数据。
- 4) 搜索有可能在非叶子结点结束
- 5) 其搜索性能等价于在关键字全集内做一次二分查找

12.3.3 B+树的介绍

B+树是 B 树的变体，也是一种多路搜索树。



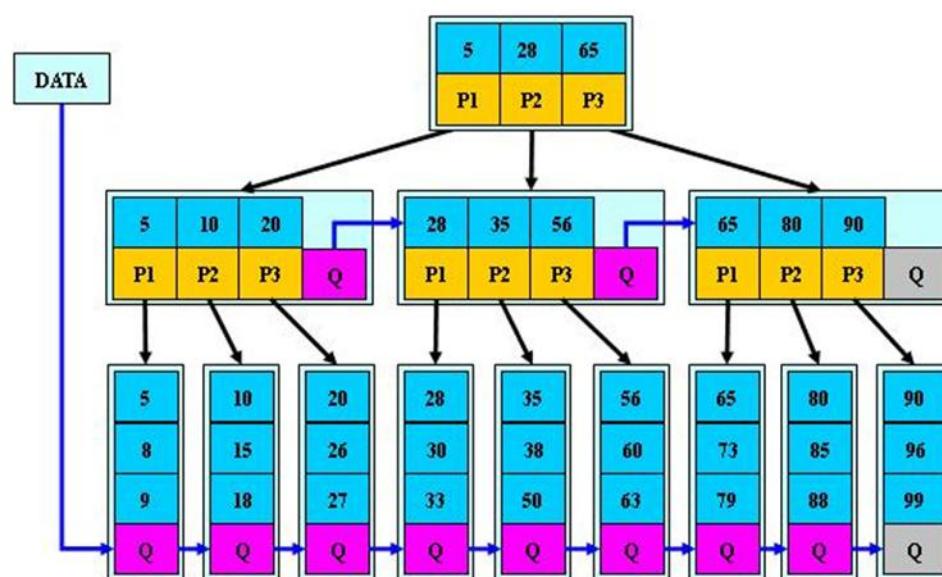
对上图的说明：

- 1) B+树的搜索与 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
- 2) 所有关键字都出现在叶子结点的链表中（即数据只能在叶子节点【也叫稠密索引】），且链表中的关键字(数据)恰好是有序的。
- 3) 不可能在非叶子结点命中
- 4) 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

- 5) 更适合文件索引系统
- 6) B 树和 B+树各有自己的应用场景，不能说 B+树完全比 B 树好，反之亦然。

12.3.4 B*树的介绍

B*树是 B+树的变体，在 B+树的非根和非叶子结点再增加指向兄弟的指针。



➤ B*树的说明:

- 1) B*树定义了非叶子结点关键字个数至少为 $(2/3)M$ ，即块的最低使用率为 2/3，而 B+树的块的最低使用率为的 1/2。
- 2) 从第 1 个特点我们可以看出，B*树分配新结点的概率比 B+树要低，空间使用率更高

第 13 章图

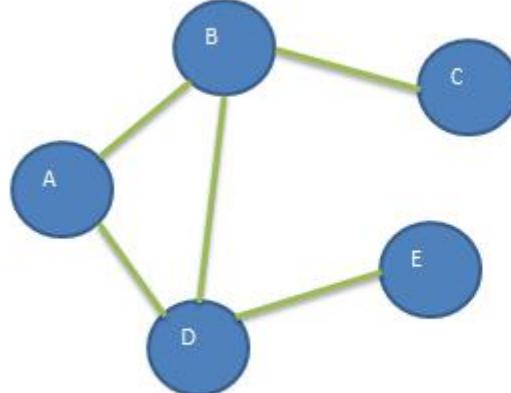
13.1 图基本介绍

13.1.1 为什么要有图

- 1) 前面我们学了线性表和树
- 2) 线性表局限于一个直接前驱和一个直接后继的关系
- 3) 树也只能有一个直接前驱也就是父节点
- 4) 当我们需要表示多对多的关系时， 这里我们就用到了图。

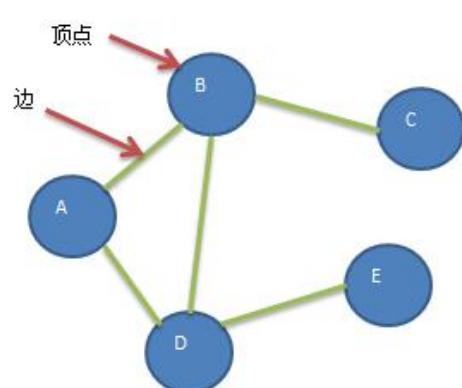
13.1.2 图的举例说明

图是一种数据结构，其中结点可以具有零个或多个相邻元素。两个结点之间的连接称为边。结点也可以称为顶点。如图：



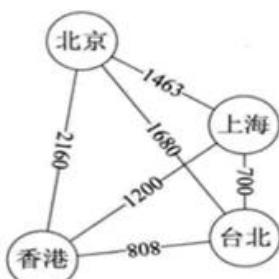
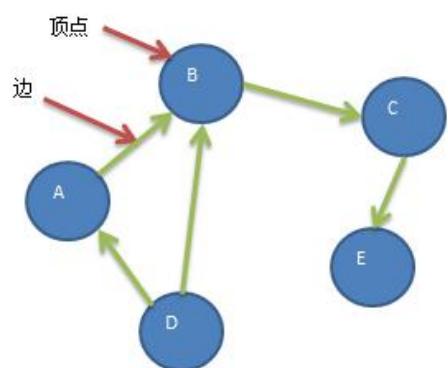
13.1.3 图的常用概念

- 1) 顶点(vertex)
- 2) 边(edge)
- 3) 路径
- 4) 无向图(右图)



无向图：顶点之间的连接没有方向，比如A-B，即可以是 A->B 也可以 B->A。
路径：比如从 D->C 的路径有
 1) D->B->C
 2) D->A->B->C

- 5) 有向图
- 6) 带权图



有向图：顶点之间的连接有方向，比如A-B，只能是 A->B 不能是 B->A。

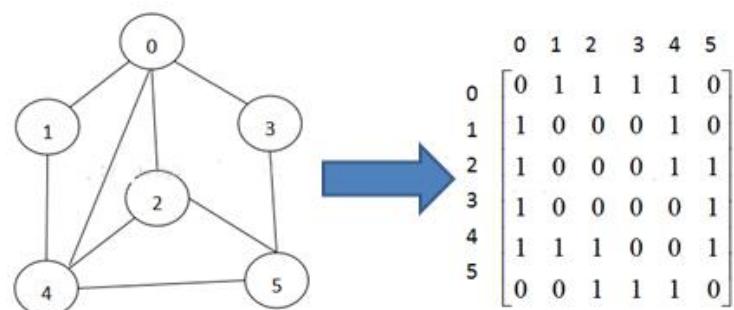
带权图：这种边带权值的图也叫网。

13.2 图的表示方式

图的表示方式有两种：二维数组表示（邻接矩阵）；链表表示（邻接表）。

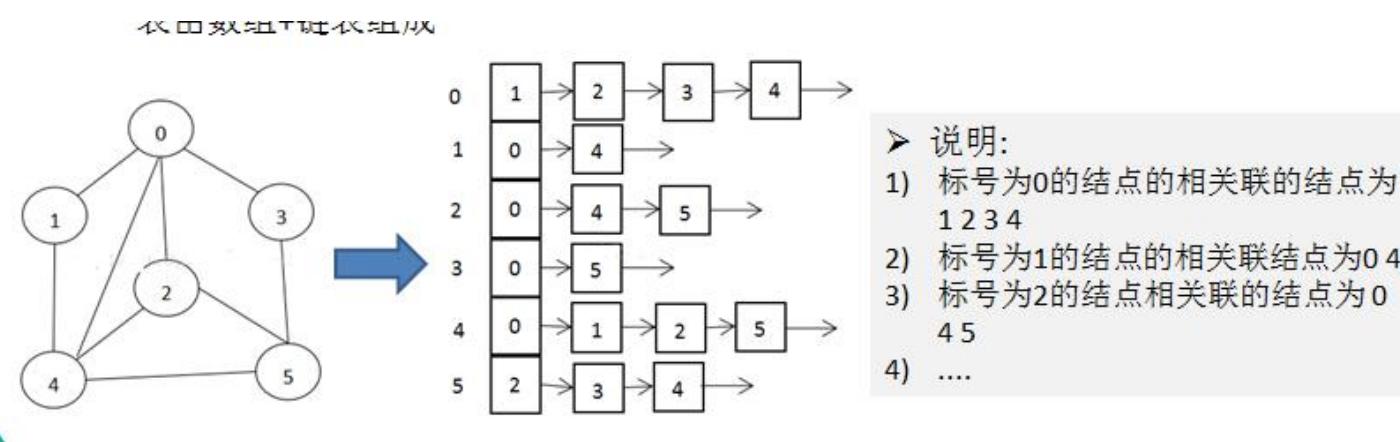
13.2.1 邻接矩阵

邻接矩阵是表示图形中顶点之间相邻关系的矩阵，对于 n 个顶点的图而言，矩阵是的 row 和 col 表示的是 1....n 个点。



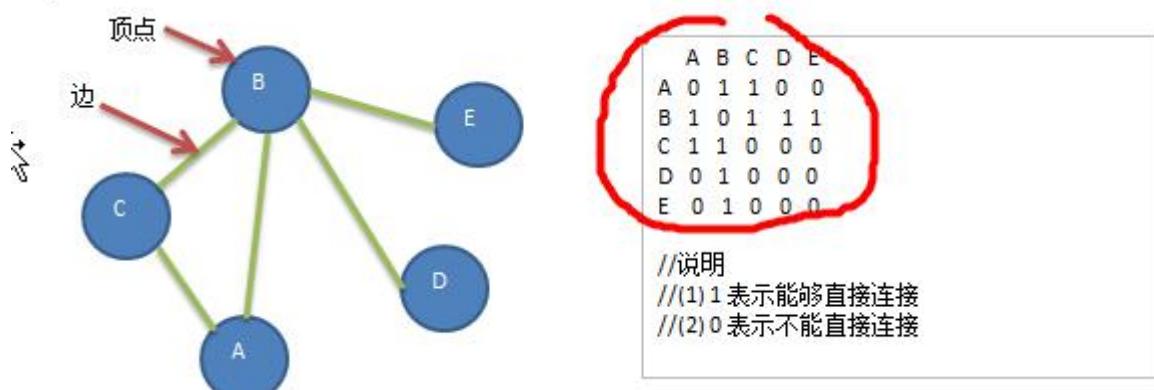
13.2.2 邻接表

- 1) 邻接矩阵需要为每个顶点都分配 n 个边的空间，其实有很多边都是不存在，会造成空间的一定损失。
- 2) 邻接表的实现只关心存在的边，不关心不存在的边。因此没有空间浪费，邻接表由数组+链表组成
- 3) 举例说明



13.3 图的快速入门案例

1) 要求：代码实现如下图结构。



2) 思路分析 (1) 存储顶点 String 使用 ArrayList (2) 保存矩阵 int[][] edges

3) 代码实现

```
//核心代码，汇总在后面
```

```
//插入结点
```

```
public void insertVertex(String vertex) {  
    vertexList.add(vertex);  
}  
  
//添加边  
  
/**  
 *  
 * @param v1 表示点的下标即使第几个顶点 "A"->"B" "A"->0 "B"->1  
 * @param v2 第二个顶点对应的下标  
 * @param weight 表示  
 */  
  
public void insertEdge(int v1, int v2, int weight) {  
    edges[v1][v2] = weight;  
    edges[v2][v1] = weight;
```



```
    numOfEdges++;
```

```
}
```

13.4 图的深度优先遍历介绍

13.4.1 图遍历介绍

所谓图的遍历，即是对结点的访问。一个图有那么多个结点，如何遍历这些结点，需要特定策略，一般有两种访问策略：(1)深度优先遍历 (2)广度优先遍历

13.4.2 深度优先遍历基本思想

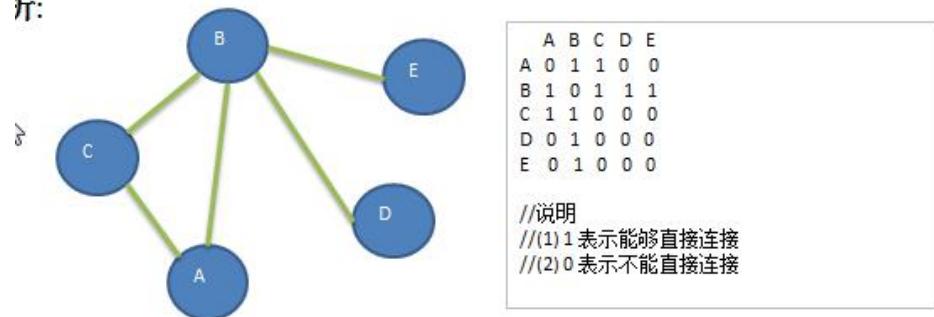
图的深度优先搜索(**Depth First Search**)。

- 1) 深度优先遍历，从初始访问结点出发，初始访问结点可能有多个邻接结点，深度优先遍历的策略就是首先访问第一个邻接结点，然后再以这个被访问的邻接结点作为初始结点，访问它的第一个邻接结点，可以这样理解：每次都在访问完当前结点后首先访问当前结点的第一个邻接结点。
- 2) 我们可以看到，这样的访问策略是优先往纵向挖掘深入，而不是对一个结点的所有邻接结点进行横向访问。
- 3) 显然，深度优先搜索是一个递归的过程

13.4.3 深度优先遍历算法步骤

- 1) 访问初始结点 v，并标记结点 v 为已访问。
- 2) 查找结点 v 的第一个邻接结点 w。
- 3) 若 w 存在，则继续执行 4，如果 w 不存在，则回到第 1 步，将从 v 的下一个结点继续。
- 4) 若 w 未被访问，对 w 进行深度优先遍历递归（即把 w 当做另一个 v，然后进行步骤 123）。
- 5) 查找结点 v 的 w 邻接结点的下一个邻接结点，转到步骤 3。
- 6) 分析图

斤:



13.4.4 深度优先算法的代码实现

```
//核心代码
```

```
//深度优先遍历算法
```

```
//i 第一次就是 0
```

```
private void dfs(boolean[] isVisited, int i) {  
    //首先我们访问该结点,输出  
    System.out.print(getValueByIndex(i) + "->");  
    //将结点设置为已经访问  
    isVisited[i] = true;  
    //查找结点 i 的第一个邻接结点 w  
    int w = getFirstNeighbor(i);  
    while(w != -1) {  
        //说明有  
        if(!isVisited[w]) {  
            dfs(isVisited, w);  
        }  
        //如果 w 结点已经被访问过  
        w = getNextNeighbor(i, w);  
    }  
}
```

```
}

//对 dfs 进行一个重载, 遍历我们所有的结点, 并进行 dfs
public void dfs() {
    isVisited = new boolean[vertexList.size()];
    //遍历所有的结点, 进行 dfs[回溯]
    for(int i = 0; i < getNumOfVertex(); i++) {
        if(!isVisited[i]) {
            dfs(isVisited, i);
        }
    }
}
```

13.5 图的广度优先遍历

13.5.1 广度优先遍历基本思想

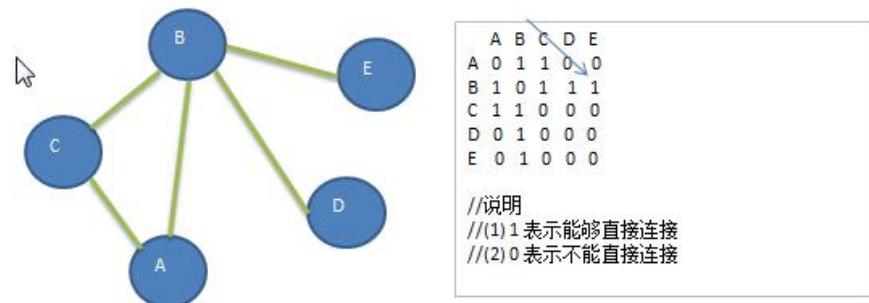
- 1) 图的广度优先搜索(Broad First Search)。
- 2) 类似于一个**分层搜索**的过程, 广度优先遍历需要使用一个队列以保持访问过的结点的顺序, 以便按这个顺序来访问这些结点的邻接结点

13.5.2 广度优先遍历算法步骤

- 1) 访问初始结点 v 并标记结点 v 为已访问。
- 2) 结点 v 入队列

- 3) 当队列非空时，继续执行，否则算法结束。
- 4) 出队列，取得队头结点 u。
- 5) 查找结点 u 的第一个邻接结点 w。
- 6) 若结点 u 的邻接结点 w 不存在，则转到步骤 3；否则循环执行以下三个步骤：
 - 6.1 若结点 w 尚未被访问，则访问结点 w 并标记为已访问。
 - 6.2 结点 w 入队列
 - 6.3 查找结点 u 的继 w 邻接结点后的下一个邻接结点 w，转到步骤 6。

13.5.3 广度优先算法的图示



13.6 广度优先算法的代码实现

```
//对一个结点进行广度优先遍历的方法
private void bfs(boolean[] isVisited, int i) {
    int u; // 表示队列的头结点对应下标
    int w; // 邻接结点 w
    //队列，记录结点访问的顺序
    LinkedList queue = new LinkedList();
    //访问结点，输出结点信息
    System.out.print(getValueByIndex(i) + "=>");
    //将结点 i 放入队列
    queue.add(i);
    while (!queue.isEmpty()) {
        //取出队首元素
        u = (int) queue.remove(0);
        //遍历与 u 相邻的所有结点
        for (w = 0; w < vertexs.length; w++) {
            if (vertexs[u][w] == 1 && !isVisited[w]) {
                //将结点 w 放入队列
                queue.add(w);
                //访问结点 w，并输出
                System.out.print(getValueByIndex(w) + "=>");
                //将结点 w 标记为已访问
                isVisited[w] = true;
            }
        }
    }
}
```



```
//标记为已访问
isVisited[i] = true;
//将结点加入队列
queue.addLast(i);

while( !queue.isEmpty() ) {
    //取出队列的头结点下标
    u = (Integer)queue.removeFirst();
    //得到第一个邻接结点的下标 w
    w = getFirstNeighbor(u);
    while(w != -1) { //找到
        //是否访问过
        if(!isVisited[w]) {
            System.out.print(getValueByIndex(w) + "=>");
            //标记已经访问
            isVisited[w] = true;
            //入队
            queue.addLast(w);
        }
        //以 u 为前驱点，找 w 后面的下一个邻结点
        w = getNextNeighbor(u, w); //体现出我们的广度优先
    }
}
```



```
//遍历所有的结点，都进行广度优先搜索
public void bfs() {
    isVisited = new boolean[vertexList.size()];
    for(int i = 0; i < getNumOfVertex(); i++) {
        if(!isVisited[i]) {
            bfs(isVisited, i);
        }
    }
}
```

13.7 图的代码汇总

```
package com.atguigu.graph;

import java.util.ArrayList;
import java.util.Arrays;
import java.util.LinkedList;

public class Graph {

    private ArrayList<String> vertexList; //存储顶点集合
    private int[][] edges; //存储图对应的邻接矩阵
    private int numOfEdges; //表示边的数目
    //定义给数组 boolean[], 记录某个结点是否被访问
    private boolean[] isVisited;
```



```
public static void main(String[] args) {  
    //测试一把图是否创建 ok  
    int n = 8; //结点的个数  
  
    //String Vertexs[] = {"A", "B", "C", "D", "E"};  
    String Vertexs[] = {"1", "2", "3", "4", "5", "6", "7", "8"};  
  
    //创建图对象  
    Graph graph = new Graph(n);  
    //循环的添加顶点  
    for(String vertex: Vertexs) {  
        graph.insertVertex(vertex);  
    }  
  
    //添加边  
    //A-B A-C B-C B-D B-E  
    //    graph.insertEdge(0, 1, 1); // A-B  
    //    graph.insertEdge(0, 2, 1); //  
    //    graph.insertEdge(1, 2, 1); //  
    //    graph.insertEdge(1, 3, 1); //  
    //    graph.insertEdge(1, 4, 1); //  
  
    //更新边的关系  
    graph.insertEdge(0, 1, 1);  
    graph.insertEdge(0, 2, 1);  
    graph.insertEdge(1, 3, 1);  
    graph.insertEdge(1, 4, 1);
```



```
graph.insertEdge(3, 7, 1);
graph.insertEdge(4, 7, 1);
graph.insertEdge(2, 5, 1);
graph.insertEdge(2, 6, 1);
graph.insertEdge(5, 6, 1);

//显示一把邻接矩阵
graph.showGraph();

//测试一把，我们的 dfs 遍历是否 ok
System.out.println("深度遍历");
graph.dfs(); // A->B->C->D->E [1->2->4->8->5->3->6->7]
//
System.out.println();
System.out.println("广度优先!");
graph.bfs(); // A->B->C->D-E [1->2->3->4->5->6->7->8]

}

//构造器
public Graph(int n) {
    //初始化矩阵和 vertexList
    edges = new int[n][n];
    vertexList = new ArrayList<String>(n);
    numOfEdges = 0;
```



```
}
```

```
//得到第一个邻接结点的下标 w
/**
 *
 * @param index
 * @return 如果存在就返回对应的下标，否则返回-1
 */
public int getFirstNeighbor(int index) {
    for(int j = 0; j < vertexList.size(); j++) {
        if(edges[index][j] > 0) {
            return j;
        }
    }
    return -1;
}

//根据前一个邻接结点的下标来获取下一个邻接结点
public int getNextNeighbor(int v1, int v2) {
    for(int j = v2 + 1; j < vertexList.size(); j++) {
        if(edges[v1][j] > 0) {
            return j;
        }
    }
    return -1;
}
```



```
//深度优先遍历算法
//i 第一次就是 0
private void dfs(boolean[] isVisited, int i) {
    //首先我们访问该结点,输出
    System.out.print(getValueByIndex(i) + "->");
    //将结点设置为已经访问
    isVisited[i] = true;
    //查找结点 i 的第一个邻接结点 w
    int w = getFirstNeighbor(i);
    while(w != -1) {//说明有
        if(!isVisited[w]) {
            dfs(isVisited, w);
        }
        //如果 w 结点已经被访问过
        w = getNextNeighbor(i, w);
    }
}

//对 dfs 进行一个重载, 遍历我们所有的结点, 并进行 dfs
public void dfs() {
    isVisited = new boolean[vertexList.size()];
    //遍历所有的结点, 进行 dfs[回溯]
    for(int i = 0; i < getNumOfVertex(); i++) {
        if(!isVisited[i]) {
```



```
dfs(isVisited, i);

}

}

}

//对一个结点进行广度优先遍历的方法

private void bfs(boolean[] isVisited, int i) {

    int u; // 表示队列的头结点对应下标

    int w; // 邻接结点 w

    //队列，记录结点访问的顺序

    LinkedList queue = new LinkedList();

    //访问结点，输出结点信息

    System.out.print(getValueByIndex(i) + "=>");

    //标记为已访问

    isVisited[i] = true;

    //将结点加入队列

    queue.addLast(i);

    while( !queue.isEmpty() ) {

        //取出队列的头结点下标

        u = (Integer)queue.removeFirst();

        //得到第一个邻接结点的下标 w

        w = getFirstNeighbor(u);

        while(w != -1) { //找到

            //是否访问过

            if(!isVisited[w]) {

                //访问结点，输出结点信息

                System.out.print(getValueByIndex(w) + "=>");

                //标记为已访问

                isVisited[w] = true;

                //将结点加入队列

                queue.addLast(w);
            }
        }
    }
}
```



```
System.out.print(getValueByIndex(w) + ">");

//标记已经访问
isVisited[w] = true;
//入队
queue.addLast(w);

}

//以 u 为前驱点, 找 w 后面的下一个邻结点
w = getNextNeighbor(u, w); //体现出我们的广度优先

}

}

}

//遍历所有的结点, 都进行广度优先搜索
public void bfs() {

    isVisited = new boolean[vertexList.size()];
    for(int i = 0; i < getNumOfVertex(); i++) {
        if(!isVisited[i]) {
            bfs(isVisited, i);
        }
    }
}

//图中常用的方法
//返回结点的个数
public int getNumOfVertex() {
```



```
return vertexList.size();

}

//显示图对应的矩阵

public void showGraph() {
    for(int[] link : edges) {
        System.out.println(Arrays.toString(link));
    }
}

//得到边的数目

public int getNumOfEdges() {
    return numOfEdges;
}

//返回结点 i(下标)对应的数据 0->"A" 1->"B" 2->"C"

public String getValueByIndex(int i) {
    return vertexList.get(i);
}

//返回 v1 和 v2 的权值

public int getWeight(int v1, int v2) {
    return edges[v1][v2];
}

//插入结点

public void insertVertex(String vertex) {
    vertexList.add(vertex);
}

//添加边

/**
```

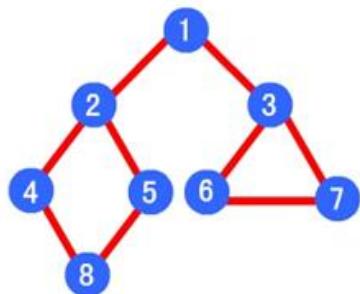
```

*
* @param v1 表示点的下标即使第几个顶点 "A"->"B" "A"->0 "B"->1
* @param v2 第二个顶点对应的下标
* @param weight 表示
*/
public void insertEdge(int v1, int v2, int weight) {
    edges[v1][v2] = weight;
    edges[v2][v1] = weight;
    numOfEdges++;
}
}

```

13.8 图的深度优先 VS 广度优先

应用实例



```

graph.insertEdge(0, 1, 1);
graph.insertEdge(0, 2, 1);
graph.insertEdge(1, 3, 1);
graph.insertEdge(1, 4, 1);
graph.insertEdge(3, 7, 1);
graph.insertEdge(4, 7, 1);
graph.insertEdge(2, 5, 1);
graph.insertEdge(2, 6, 1);
graph.insertEdge(5, 6, 1);

```

- 1) 深度优先遍历顺序为 1->2->4->8->5->3->6->7
- 2) 广度优先算法的遍历顺序为： 1->2->3->4->5->6->7->8



第 14 章程序员常用 10 种算法

14.1 二分查找算法(非递归)

14.1.1 二分查找算法(非递归)介绍

- 1) 前面我们讲过了二分查找算法，是使用递归的方式，下面我们讲解二分查找算法的非递归方式
- 2) 二分查找法只适用于从有序的数列中进行查找(比如数字和字母等)，将数列排序后再进行查找
- 3) 二分查找法的运行时间为对数时间 $O(\log_2 n)$ ，即查找到需要的目标位置最多只需要 $\log_2 n$ 步，假设从[0,99]的队列(100 个数，即 $n=100$)中寻到目标数 30，则需要查找步数为 $\log_2 100$ ，即最多需要查找 7 次($2^6 < 100 < 2^7$)

14.1.2 二分查找算法(非递归)代码实现

数组 {1,3, 8, 10, 11, 67, 100}，编程实现二分查找，要求使用非递归的方式完成。

1) 思路分析：

2) 代码实现：

```
package com.atguigu.binarysearchnorecursion;

public class BinarySearchNoRecur {

    public static void main(String[] args) {
        //测试
        int[] arr = {1,3, 8, 10, 11, 67, 100};
        int index = binarySearch(arr, 100);
        System.out.println("index=" + index);
```



```
}
```

```
//二分查找的非递归实现

/**
 *
 * @param arr 待查找的数组, arr 是升序排序
 * @param target 需要查找的数
 * @return 返回对应下标, -1 表示没有找到
 */

public static int binarySearch(int[] arr, int target) {

    int left = 0;
    int right = arr.length - 1;
    while(left <= right) { //说明继续查找
        int mid = (left + right) / 2;
        if(arr[mid] == target) {
            return mid;
        } else if ( arr[mid] > target) {
            right = mid - 1;//需要向左边查找
        } else {
            left = mid + 1; //需要向右边查找
        }
    }
    return -1;
}
```



{}

14.2 分治算法

14.2.1 分治算法介绍

- 1) 分治法是一种很重要的算法。字面上的解释是“分而治之”，就是把一个复杂的问题分成两个或更多的相同或相似的子问题，再把子问题分成更小的子问题……直到最后子问题可以简单的直接求解，原问题的解即子问题的解的合并。这个技巧是很多高效算法的基础，如排序算法(快速排序，归并排序)，傅立叶变换(快速傅立叶变换)……
- 2) 分治算法可以求解的一些经典问题
 - ✓ 二分搜索
 - ✓ 大整数乘法
 - ✓ 棋盘覆盖
 - ✓ 合并排序
 - ✓ 快速排序
 - ✓ 线性时间选择
 - ✓ 最接近点对问题
 - ✓ 循环赛日程表
 - ✓ 汉诺塔

14.2.2 分治算法的基本步骤

分治法在每一层递归上都有三个步骤：

- 1) 分解：将原问题分解为若干个规模较小，相互独立，与原问题形式相同的子问题
- 2) 解决：若子问题规模较小而容易被解决则直接解，否则递归地解各个子问题

- 3) 合并：将各个子问题的解合并为原问题的解。

14.2.3 分治(Divide-and-Conquer(P))算法设计模式如下：

```
if |P|≤n0
    then return(ADHOC(P))
//将P分解为较小的子问题 P1,P2,...,Pk
for i←1 to k
do yi ← Divide-and-Conquer(Pi) 递归解决Pi
T ← MERGE(y1,y2,...,yk) 合并子问题
return(T)
```

其中 $|P|$ 表示问题P的规模； n_0 为一阈值，表示当问题P的规模不超过 n_0 时，问题已容易直接解出，不必再继续分解。 $ADHOC(P)$ 是该分治法中的基本子算法，用于直接解小规模的问题P。因此，当P的规模不超过 n_0 时直接用算法 $ADHOC(P)$ 求解。算法 $MERGE(y_1,y_2,\dots,y_k)$ 是该分治法中的合并子算法，用于将P的子问题 P_1,P_2,\dots,P_k 的相应的解 y_1,y_2,\dots,y_k 合并为P的解。

14.2.4 分治算法最佳实践-汉诺塔

➤ 汉诺塔的传说

汉诺塔：汉诺塔（又称河内塔）问题是源于印度一个古老传说的益智玩具。大梵天创造世界的时候做了三根金刚石柱子，在一根柱子上从下往上按照大小顺序摞着 64 片黄金圆盘。大梵天命令婆罗门把圆盘从下面开始按大小顺序重新摆放在另一根柱子上。并且规定，在小圆盘上不能放大圆盘，在三根柱子之间一次只能移动一个圆盘。

假如每秒钟一次，共需多长时间呢？移完这些金片需要 5845.54 亿年以上，太阳系的预期寿命据说也就是数百亿年。真的过了 5845.54 亿年，地球上的一切生命，连同梵塔、庙宇等，都早已经灰飞烟灭。

➤ 汉诺塔游戏的演示和思路分析：



1) 如果是有一个盘, A->C

如果我们有 $n \geq 2$ 情况, 我们总是可以看做是两个盘 1.最下边的盘 2. 上面的盘

2) 先把 最上面的盘 A->B

3) 把最下边的盘 A->C

4) 把 B 塔的所有盘 从 B->C

➤ 汉诺塔游戏的代码实现:

看老师代码演示:

```
package com.atguigu.dac;

public class Hanoitower {

    public static void main(String[] args) {
        hanoiTower(5, 'A', 'B', 'C');
    }

    //汉诺塔的移动的方法
    //使用分治算法

    public static void hanoiTower(int num, char a, char b, char c) {
        //如果只有一个盘
        if(num == 1) {
            System.out.println("第 1 个盘从 " + a + "->" + c);
        }
    }
}
```

```
    } else {  
        //如果我们有 n >= 2 情况，我们总是可以看做是两个盘 1.最下边的一个盘 2. 上面的所有盘  
        //1. 先把 最上面的所有盘 A->B， 移动过程会使用到 c  
        hanoiTower(num - 1, a, c, b);  
        //2. 把最下边的盘 A->C  
        System.out.println("第" + num + "个盘从 " + a + "->" + c);  
        //3. 把 B 塔的所有盘 从 B->C， 移动过程使用到 a 塔  
        hanoiTower(num - 1, b, a, c);  
  
    }  
}  
}
```

14.3 动态规划算法

14.3.1 应用场景-背包问题

背包问题：有一个背包，容量为 4 磅， 现有如下物品

物品	重量	价格
吉他(G)	1	1500
音响(S)	4	3000
电脑(L)	3	2000

- 1) 要求达到的目标为装入的背包的总价值最大，并且重量不超出
- 2) 要求装入的物品不能重复

14.3.2 动态规划算法介绍

- 1) 动态规划(Dynamic Programming)算法的核心思想是：将大问题划分为小问题进行解决，从而一步步获取最优解的处理算法
- 2) 动态规划算法与分治算法类似，其基本思想也是将待求解问题分解成若干个子问题，先求解子问题，然后从这些子问题的解得到原问题的解。
- 3) 与分治法不同的是，适合于用动态规划求解的问题，经分解得到子问题往往不是互相独立的。（即下一个子阶段的求解是建立在上一个子阶段的解的基础上，进行进一步的求解）
- 4) 动态规划可以通过填表的方式来逐步推进，得到最优解.

14.3.3 动态规划算法最佳实践-背包问题

背包问题：有一个背包，容量为 4 磅，现有如下物品

物品	重量	价格
吉他(G)	1	1500
音响(S)	4	3000
电脑(L)	3	2000

- 1) 要求达到的目标为装入的背包的总价值最大，并且重量不超出
- 2) 要求装入的物品不能重复



思路分析和图解

- 3) 背包问题主要是指一个给定容量的背包、若干具有一定价值和重量的物品，如何选择物品放入背包使物品的价值最大。其中又分 **01 背包** 和 **完全背包**(完全背包指的是：每种物品都有无限件可用)
- 4) 这里的问题属于 **01 背包**，即每个物品最多放一个。而无限背包可以转化为 01 背包。
- 5) 算法的主要思想，利用动态规划来解决。每次遍历到的第 i 个物品，根据 $w[i]$ 和 $v[i]$ 来确定是否需要将该物品放入背包中。即对于给定的 n 个物品，设 $v[i]$ 、 $w[i]$ 分别为第 i 个物品的价值和重量，C 为背包的容量。再令 $v[i][j]$ 表示在前 i 个物品中能够装入容量为 j 的背包中的最大价值。则我们有下面的结果：

```
(1) v[i][0]=v[0][j]=0; //表示 填入表 第一行和第一列是 0  
(2) 当 w[i]>j 时: v[i][j]=v[i-1][j] // 当准备加入新增的商品的容量大于 当前背包的容量时，就直接使用上一个单元格的装入策略  
(3) 当 j>=w[i]时: v[i][j]=max {v[i-1][j], v[i]+v[i-1][j-w[i]]}  
// 当 准备加入的新增的商品的容量小于等于当前背包的容量，  
// 装入的方式:  
v[i-1][j]: 就是上一个单元格的装入的最大值  
v[i]: 表示当前商品的价值  
v[i-1][j-w[i]] : 装入 i-1 商品，到剩余空间 j-w[i]的最大值  
当 j>=w[i]时: v[i][j]=max {v[i-1][j], v[i]+v[i-1][j-w[i]]} :
```

6) 图解的分析



物品	重量	价格		物品	0 磅	1磅	2磅	3磅	4磅
吉他(G)	1	1500			0	1500(G)	1500(G)	1500(G)	
音响(S)	4	3000			0	1500(G)	1500(G)	3000(S)	
电脑(L)	3	2000	→		0	1500(G)	1500(G)	2000(L)	2000(L)+1500(G)

(1) $v[i][0]=v[0][j]=0$; //表示填入表第一行和第一列是0
(2) 当 $w[i]>j$ 时: $v[i][j]=v[i-1][j]$ //当准备加入新增的商品的容量时, 就直接使用上一个单元格的装入策略
(3) 当 $j>w[i]$ 时: $v[i][j]=\max\{v[i-1][j], v[i]+v[i-1][j-w[i]]\}$ //当准备加入的新商品的容量小于等于当前背包的容量

验证公式:
 $v[1][1]=1500$
1. $i=1, j=1$
2. $w[1]=w[1]=1$
 $w[1]=1, j=1, v[1][j]=\max\{v[1][1], v[1]+v[0][1-1]\}=1500$
 $v[1][1]=\max\{v[0][1], v[1]+v[0][1-1]\}=\max\{0, 1500+0\}=1500$
 $v[3][4]=3560$
1. $i=3, j=4$
 $w[3]=3, j=4$
 $w[3]=3, j=4$

14.3.4 动态规划-背包问题的代码实现

```
package com.atguigu.dynamic;

public class KnapsackProblem {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        int[] w = {1, 4, 3}; // 物品的重量
        int[] val = {1500, 3000, 2000}; // 物品的价值 这里 val[i] 就是前面讲的 v[i]
        int m = 4; // 背包的容量
        int n = val.length; // 物品的个数
    }
}
```



```
//创建二维数组，  
//v[i][j] 表示在前 i 个物品中能够装入容量为 j 的背包中的最大价值  
int[][] v = new int[n+1][m+1];  
//为了记录放入商品的情况，我们定一个二维数组  
int[][] path = new int[n+1][m+1];  
  
//初始化第一行和第一列，这里在本程序中，可以不去处理，因为默认就是 0  
for(int i = 0; i < v.length; i++) {  
    v[i][0] = 0; //将第一列设置为 0  
}  
for(int i=0; i < v[0].length; i++) {  
    v[0][i] = 0; //将第一行设置 0  
}  
  
//根据前面得到公式来动态规划处理  
for(int i = 1; i < v.length; i++) { //不处理第一行 i 是从 1 开始的  
    for(int j=1; j < v[0].length; j++) { //不处理第一列, j 是从 1 开始的  
        //公式  
        if(w[i-1]>j) { // 因为我们程序 i 是从 1 开始的，因此原来公式中的 w[i] 修改成 w[i-1]  
            v[i][j]=v[i-1][j];  
        } else {  
            //说明：  
            //因为我们的 i 从 1 开始的， 因此公式需要调整成  
            //v[i][j]=Math.max(v[i-1][j], val[i-1]+v[i-1][j-w[i-1]]);
```



```
//v[i][j] = Math.max(v[i - 1][j], val[i - 1] + v[i - 1][j - w[i - 1]]);  
  
//为了记录商品存放到背包的情况，我们不能直接的使用上面的公式，需要使用 if-else 来体现公式  
  
if(v[i - 1][j] < val[i - 1] + v[i - 1][j - w[i - 1]]) {  
    v[i][j] = val[i - 1] + v[i - 1][j - w[i - 1]];  
    //把当前的情况记录到 path  
    path[i][j] = 1;  
} else {  
    v[i][j] = v[i - 1][j];  
}  
  
}  
}  
}  
  
}  
  
//输出一下 v 看看目前的情况  
for(int i = 0; i < v.length; i++) {  
    for(int j = 0; j < v[i].length; j++) {  
        System.out.print(v[i][j] + " ");  
    }  
    System.out.println();  
}  
  
System.out.println("=====");  
//输出最后我们是放入的哪些商品  
//遍历 path，这样输出会把所有的放入情况都得到，其实我们只需要最后的放入
```



```
//     for(int i = 0; i < path.length; i++) {  
//         for(int j=0; j < path[i].length; j++) {  
//             if(path[i][j] == 1) {  
//                 System.out.printf("第%d 个商品放入到背包\n", i);  
//             }  
//         }  
//     }  
  
//动脑筋  
int i = path.length - 1; //行的最大下标  
int j = path[0].length - 1; //列的最大下标  
while(i > 0 && j > 0 ) { //从 path 的最后开始找  
    if(path[i][j] == 1) {  
        System.out.printf("第%d 个商品放入到背包\n", i);  
        j -= w[i-1]; //w[i-1]  
    }  
    i--;  
}  
  
}  
}
```

14.4 KMP 算法



14.4.1 应用场景-字符串匹配问题

- 字符串匹配问题：：
- 1) 有一个字符串 str1= ""硅硅谷 尚硅谷你尚硅 尚硅谷你尚硅谷你尚硅你好""，和一个子串 str2="尚硅谷你尚硅 你"
 - 2) 现在要判断 str1 是否含有 str2, 如果存在，就返回第一次出现的位置，如果没有，则返回-1

14.4.2 暴力匹配算法

如果用暴力匹配的思路，并假设现在 str1 匹配到 i 位置，子串 str2 匹配到 j 位置，则有：

- 1) 如果当前字符匹配成功（即 $\text{str1}[i] == \text{str2}[j]$ ），则 $i++$, $j++$, 继续匹配下一个字符
- 2) 如果失配（即 $\text{str1}[i] != \text{str2}[j]$ ），令 $i = i - (j - 1)$, $j = 0$ 。相当于每次匹配失败时， i 回溯， j 被置为 0。
- 3) 用暴力方法解决的话就会有大量的回溯，每次只移动一位，若是不匹配，移动到下一位接着判断，浪费了大量的时间。（不可行！）
- 4) 暴力匹配算法实现。
- 5) 代码

```
package com.atguigu.kmp;

public class ViolenceMatch {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        // 测试暴力匹配算法
    }
}
```



```
String str1 = "硅硅谷 尚硅谷你尚硅 尚硅谷你尚硅谷你尚硅你好";  
String str2 = "尚硅谷你尚硅你~";  
int index = violenceMatch(str1, str2);  
System.out.println("index=" + index);  
  
}  
  
// 暴力匹配算法实现  
public static int violenceMatch(String str1, String str2) {  
    char[] s1 = str1.toCharArray();  
    char[] s2 = str2.toCharArray();  
  
    int s1Len = s1.length;  
    int s2Len = s2.length;  
  
    int i = 0; // i 索引指向 s1  
    int j = 0; // j 索引指向 s2  
    while (i < s1Len && j < s2Len) {// 保证匹配时，不越界  
  
        if(s1[i] == s2[j]) {//匹配 ok  
            i++;  
            j++;  
        } else { //没有匹配成功  
            //如果失配（即 str1[i] != str2[j]），令 i = i - (j - 1), j = 0。  
            i = i - (j - 1);  
            j = 0;  
        }  
    }  
}
```



```
    }  
    }  
  
    //判断是否匹配成功  
    if(j == s2Len) {  
        return i - j;  
    } else {  
        return -1;  
    }  
}  
  
}
```

14.4.3 KMP 算法介绍

- 1) KMP 是一个解决模式串在文本串是否出现过，如果出现过，最早出现的位置的经典算法
- 2) Knuth-Morris-Pratt 字符串查找算法，简称为“KMP 算法”，常用于在一个文本串 S 内查找一个模式串 P 的出现位置，这个算法由 Donald Knuth、Vaughan Pratt、James H. Morris 三人于 1977 年联合发表，故取这 3 人的姓氏命名此算法。
- 3) KMP 方法算法就利用之前判断过信息，通过一个 next 数组，保存模式串中前后最长公共子序列的长度，每次回溯时，通过 next 数组找到，前面匹配过的位置，省去了大量的计算时间
- 4) 参考资料：<https://www.cnblogs.com/ZuoAndFutureGirl/p/9028287.html>

14.4.4 KMP 算法最佳应用-字符串匹配问题

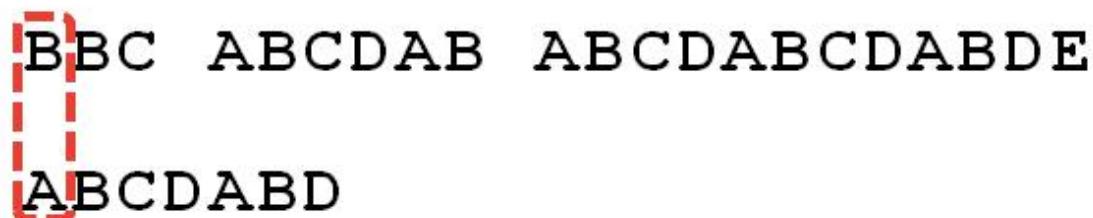
➤ 字符串匹配问题：：

- 1) 有一个字符串 str1= "BBC ABCDAB ABCDABCDABDE"，和一个子串 str2="ABCDABD"
- 2) 现在要判断 str1 是否含有 str2, 如果存在, 就返回第一次出现的位置, 如果没有, 则返回-1
- 3) 要求: 使用 **KMP 算法完成判断**, 不能使用简单的暴力匹配算法.

➤ 思路分析图解

举例来说, 有一个字符串 Str1 = “BBC ABCDAB ABCDABCDABDE”，判断, 里面是否包含另一个字符串 Str2 = “ABCDABD”？

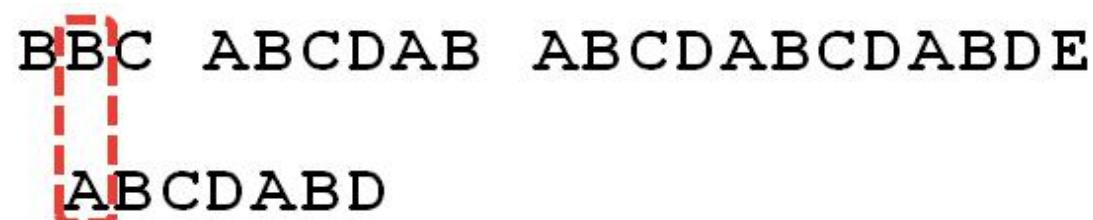
1.首先, 用 Str1 的第一个字符和 Str2 的第一个字符去比较, 不符合, 关键词向后移动一位



BBC ABCDAB ABCDABCDABDE

ABCDABD

2. 重复第一步, 还是不符合, 再后移



BBC ABCDAB ABCDABCDABDE

ABCDABD

3. 一直重复, 直到 Str1 有一个字符与 Str2 的第一个字符符合为止

BBC ABCDAB ABCDABCDABDE
ABCDABD

4. 接着比较字符串和搜索词的下一个字符，还是符合。

BBC ABCDAB ABCDABCDABDE
ABCDABD

5. 遇到 Str1 有一个字符与 Str2 对应的字符不符合。

BBC ABCDAB ABCDABCDABDE
ABCDABD

6. 这时候，想到的是继续遍历 Str1 的下一个字符，重复第 1 步。(其实是很不明智的，因为此时 BCD 已经比较过了，没有必要再做重复的工作，一个基本事实是，当空格与 D 不匹配时，你其实知道前面六个字符是“ABCDAB”。KMP 算法的想法是，设法利用这个已知信息，不要把“搜索位置”移回已经比较过的位置，继续把它向后移，这样就提高了效率。)

BBC ABCDAB ABCDABCDABDE
ABCDABD

7. 怎么做到把刚刚重复的步骤省略掉？可以对 Str2 计算出一张《部分匹配表》，这张表的产生在后面介绍

搜索词

A B C D A B D

部分匹配值

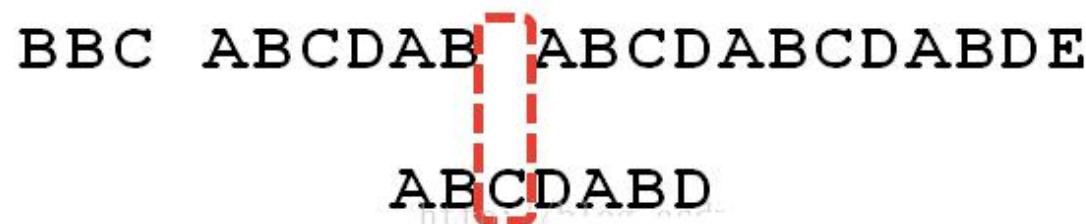
0 0 0 0 1 2 0

8. 已知空格与 D 不匹配时，前面六个字符”ABCDAB”是匹配的。查表可知，最后一个匹配字符 B 对应的”部分匹配值”为 2，因此按照下面的公式算出向后移动的位数：

移动位数 = 已匹配的字符数 - 对应的部分匹配值

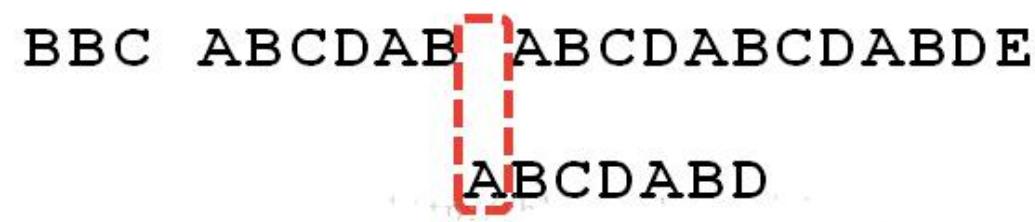
因为 $6 - 2$ 等于 4，所以将搜索词向后移动 4 位。

9. 因为空格与 C 不匹配，搜索词还要继续往后移。这时，已匹配的字符数为 2（”AB”），对应的”部分匹配值”为 0。所以，移动位数 = $2 - 0$ ，结果为 2，于是将搜索词向后移 2 位。



该图展示了字符串匹配的过程。上方文本为“BBC ABCDAB ABCDABCDABDE”，下方文本为“ABCDABD”。上方文本中从第4个字符开始（即“CDAB”）到第8个字符（即“ABDE”）被红色虚线框选中，表示当前正在比较的子串。下方文本完全重叠在上方文本的“ABDE”部分之下。

10. 因为空格与 A 不匹配，继续后移一位。



该图展示了字符串匹配的过程。上方文本为“BBC ABCDAB ABCDABCDABDE”，下方文本为“ABCDABD”。上方文本中从第4个字符开始（即“CDAB”）到第7个字符（即“ABD”）被红色虚线框选中，表示当前正在比较的子串。下方文本完全重叠在上方文本的“ABD”部分之下。

11. 逐位比较，直到发现 C 与 D 不匹配。于是，移动位数 = $6 - 2$ ，继续将搜索词向后移动 4 位。

BBC ABCDAB ABCDABCDABDE
ABCDABD

12.逐位比较，直到搜索词的最后一位，发现完全匹配，于是搜索完成。如果还要继续搜索（即找出全部匹配），移动位数 = 7 - 0，再将搜索词向后移动 7 位，这里就不再重复了。

BBC ABCDAB ABCDABCDABDE
ABCDABD

13.介绍《部分匹配表》怎么产生的
先介绍前缀，后缀是什么

字符串： “bread”

前缀： b , br , bre , brea

后缀： read , ead , ad , d

“部分匹配值”就是”前缀”和”后缀”的最长的共有元素的长度。以”ABCDABD”为例，
— ”A”的前缀和后缀都为空集，共有元素的长度为 0；
— ”AB”的前缀为[A]，后缀为[B]，共有元素的长度为 0；
— ”ABC”的前缀为[A, AB]，后缀为[BC, C]，共有元素的长度 0；
— ”ABCD”的前缀为[A, AB, ABC]，后缀为[BCD, CD, D]，共有元素的长度为 0；
— ”ABCDA”的前缀为[A, AB, ABC, ABCD]，后缀为[BCDA, CDA, DA, A]，共有元素为”A”，长度为 1；
— ”ABCDAB”的前缀为[A, AB, ABC, ABCD, ABCDA]，后缀为[BCDAB, CDAB, DAB, AB, B]，共有元素为”AB”，
长度为 2；
— ”ABCDABD”的前缀为[A, AB, ABC, ABCD, ABCDA, ABCDAB]，后缀为[BCDABD, CDABD, DABD, ABD, BD,
D]，共有元素的长度为 0。

14.“部分匹配”的实质是，有时候，字符串头部和尾部会有重复。比如，“ABCDAB”之中有两个“AB”，那么它的“部分匹配值”就是2（“AB”的长度）。搜索词移动的时候，第一个“AB”向后移动4位（字符串长度-部分匹配值），就可以来到第二个“AB”的位置。

搜索词	A B C D A B D
部分匹配值	0 0 0 0 1 2 0

到此 KMP 算法思想分析完毕！

➤ 看老师代码实现

```
package com.atguigu.kmp;

import java.util.Arrays;

public class KMPAlgorithm {

    public static void main(String[] args) {
        // TODO Auto-generated method stub
        String str1 = "BBC ABCDAB ABCDABCDABDE";
        String str2 = "ABCDABD";
        //String str2 = "BBC";

        int[] next = kmpNext("ABCDABD"); // [0, 1, 2, 0]
        System.out.println("next=" + Arrays.toString(next));
    }
}
```



```
int index = kmpSearch(str1, str2, next);
System.out.println("index=" + index); // 15 了

}

//写出我们的 kmp 搜索算法
/**
 *
 * @param str1 源字符串
 * @param str2 子串
 * @param next 部分匹配表，是子串对应的部分匹配表
 * @return 如果是-1 就是没有匹配到，否则返回第一个匹配的位置
 */
public static int kmpSearch(String str1, String str2, int[] next) {

    //遍历
    for(int i = 0, j = 0; i < str1.length(); i++) {

        //需要处理 str1.charAt(i) != str2.charAt(j), 去调整 j 的大小
        //KMP 算法核心点，可以验证...
        while( j > 0 && str1.charAt(i) != str2.charAt(j)) {
            j = next[j-1];
        }
    }
}
```



```
if(str1.charAt(i) == str2.charAt(j)) {  
    j++;  
}  
if(j == str2.length()) {//找到了 // j = 3 i  
    return i - j + 1;  
}  
}  
return -1;  
}  
  
//获取到一个字符串(子串) 的部分匹配值表  
public static int[] kmpNext(String dest) {  
    //创建一个 next 数组保存部分匹配值  
    int[] next = new int[dest.length()];  
    next[0] = 0; //如果字符串是长度为 1 部分匹配值就是 0  
    for(int i = 1, j = 0; i < dest.length(); i++) {  
        //当 dest.charAt(i) != dest.charAt(j) , 我们需要从 next[j-1]获取新的 j  
        //直到我们发现 有 dest.charAt(i) == dest.charAt(j)成立才退出  
        //这时 kmp 算法的核心点  
        while(j > 0 && dest.charAt(i) != dest.charAt(j)) {  
            j = next[j-1];  
        }  
        //当 dest.charAt(i) == dest.charAt(j) 满足时, 部分匹配值就是+1  
        if(dest.charAt(i) == dest.charAt(j)) {  
            j++;  
        }  
    }  
}
```

```
    }  
  
    next[i] = j;  
  
}  
  
return next;  
  
}  
  
}
```

14.5 贪心算法

14.5.1 应用场景-集合覆盖问题

假设存在下面需要付费的广播台，以及广播台信号可以覆盖的地区。 如何选择最少的广播台，让所有的地区都可以接收到信号

广播台	覆盖地区
K1	"北京", "上海", "天津"
K2	"广州", "北京", "深圳"
K3	"成都", "上海", "杭州"
K4	"上海", "天津"
K5	"杭州", "大连"

14.5.2 贪心算法介绍

- 1) 贪婪算法(贪心算法)是指在对问题进行求解时，在每一步选择中都采取最好或者最优(即最有利)的选择，从而希望能够导致结果是最好或者最优的算法

2) 贪婪算法所得到的结果不一定是最优的结果(有时候会是最优解), 但是都是相对近似(接近)最优解的结果

14.5.3 贪心算法最佳应用-集合覆盖

1) 假设存在如下表的需要付费的广播台, 以及广播台信号可以覆盖的地区。 如何选择最少的广播台, 让所有的地区都可以接收到信号

广播台	覆盖地区
K1	"北京", "上海", "天津"
K2	"广州", "北京", "深圳"
K3	"成都", "上海", "杭州"
K4	"上海", "天津"
K5	"杭州", "大连"

2) 思路分析:

➤ 如何找出覆盖所有地区的广播台的集合呢, 使用穷举法实现, 列出每个可能的广播台的集合, 这被称为幂集。假设总的有 n 个广播台, 则广播台的组合总共有

$2^n - 1$ 个, 假设每秒可以计算 10 个子集, 如图:

广播台数量n	子集总数 2^n	需要的时间
5	32	3.2秒
10	1024	102.4秒
32	4294967296	13.6年
100	1.26×10^{30}	4×10^{23} 年

➤ 使用贪婪算法，效率高：

- 1) 目前并没有算法可以快速计算得到准备的值， 使用贪婪算法，则可以得到非常接近的解，并且效率高。选择策略上，因为需要覆盖全部地区的最小集合：
- 2) 遍历所有的广播电台，找到一个覆盖了最多未覆盖的地区的电台(此电台可能包含一些已覆盖的地区，但没关系)
- 3) 将这个电台加入到一个集合中(比如 ArrayList)，想办法把该电台覆盖的地区在下次比较时去掉。
- 4) 重复第 1 步直到覆盖了全部的地区

分析的图解：



3) 代码实现

```
package com.atguigu.greedy;

import java.util.ArrayList;
import java.util.HashMap;
import java.util.HashSet;
```



```
public class GreedyAlgorithm {  
  
    public static void main(String[] args) {  
        //创建广播电台,放入到 Map  
        HashMap<String,HashSet<String>> broadcasts = new HashMap<String, HashSet<String>>();  
        //将各个电台放入到 broadcasts  
        HashSet<String> hashSet1 = new HashSet<String>();  
        hashSet1.add("北京");  
        hashSet1.add("上海");  
        hashSet1.add("天津");  
  
        HashSet<String> hashSet2 = new HashSet<String>();  
        hashSet2.add("广州");  
        hashSet2.add("北京");  
        hashSet2.add("深圳");  
  
        HashSet<String> hashSet3 = new HashSet<String>();  
        hashSet3.add("成都");  
        hashSet3.add("上海");  
        hashSet3.add("杭州");  
  
        HashSet<String> hashSet4 = new HashSet<String>();  
        hashSet4.add("上海");  
        hashSet4.add("天津");  
    }  
}
```



```
HashSet<String> hashSet5 = new HashSet<String>();  
hashSet5.add("杭州");  
hashSet5.add("大连");  
  
//加入到 map  
broadcasts.put("K1", hashSet1);  
broadcasts.put("K2", hashSet2);  
broadcasts.put("K3", hashSet3);  
broadcasts.put("K4", hashSet4);  
broadcasts.put("K5", hashSet5);  
  
//allAreas 存放所有的地区  
HashSet<String> allAreas = new HashSet<String>();  
allAreas.add("北京");  
allAreas.add("上海");  
allAreas.add("天津");  
allAreas.add("广州");  
allAreas.add("深圳");  
allAreas.add("成都");  
allAreas.add("杭州");  
allAreas.add("大连");  
  
//创建 ArrayList, 存放选择的电台集合  
ArrayList<String> selects = new ArrayList<String>();  
  
//定义一个临时的集合, 在遍历的过程中, 存放遍历过程中的电台覆盖的地区和当前还没有覆盖的地区的
```



交集

```
HashSet<String> tempSet = new HashSet<String>();

//定义给 maxKey , 保存在一次遍历过程中，能够覆盖最大未覆盖的地区对应的电台的 key
//如果 maxKey 不为 null，则会加入到 selects

String maxKey = null;

while(allAreas.size() != 0) { // 如果 allAreas 不为 0，则表示还没有覆盖到所有的地区
    //每进行一次 while,需要
    maxKey = null;

    //遍历 broadcasts, 取出对应 key
    for(String key : broadcasts.keySet()) {
        //每进行一次 for
        tempSet.clear();
        //当前这个 key 能够覆盖的地区
        HashSet<String> areas = broadcasts.get(key);
        tempSet.addAll(areas);
        //求出 tempSet 和 allAreas 集合的交集, 交集会赋给 tempSet
        tempSet.retainAll(allAreas);
        //如果当前这个集合包含的未覆盖地区的数量，比 maxKey 指向的集合地区还多
        //就需要重置 maxKey
        // tempSet.size() > broadcasts.get(maxKey).size() 体现出贪心算法的特点,每次都选择最优的
        if(tempSet.size() > 0 &&
            (maxKey == null || tempSet.size() > broadcasts.get(maxKey).size())){
            maxKey = key;
        }
    }
}
```



```
}

//maxKey != null, 就应该将 maxKey 加入 selects
if(maxKey != null) {

    selects.add(maxKey);
    //将 maxKey 指向的广播电台覆盖的地区，从 allAreas 去掉
    allAreas.removeAll(broadcasts.get(maxKey));

}

System.out.println("得到的选择结果是" + selects);//[K1,K2,K3,K5]

}

}
```

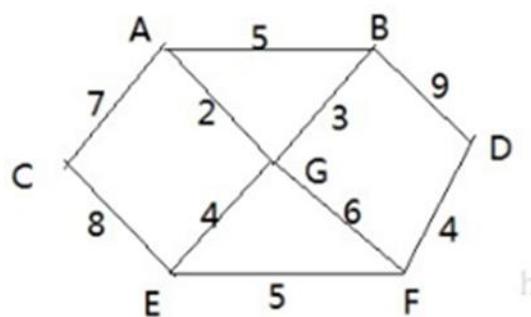
14.5.4 贪心算法注意事项和细节

- 1) 贪婪算法所得到的结果不一定是最优的结果(有时候会是最优解)，但是都是相对近似(接近)最优解的结果
- 2) 比如上题的算法选出的是 K1, K2, K3, K5，符合覆盖了全部的地区
- 3) 但是我们发现 K2, K3, K4, K5 也可以覆盖全部地区，如果 K2 的使用成本低于 K1,那么我们上题的 K1, K2, K3, K5 虽然是满足条件，但是并不是最优的.

14.6 普里姆算法

14.6.1 应用场景-修路问题

➤ 看一个应用场景和问题：



- 1) 有胜利乡有 7 个村庄(A, B, C, D, E, F, G) , 现在需要修路把 7 个村庄连通
- 2) 各个村庄的距离用边线表示(权) , 比如 A - B 距离 5 公里
- 3) 问：如何修路保证各个村庄都能连通，并且总的修建公路总里程最短？

思路：将 10 条边，连接即可，但是总的里程数不是最小。

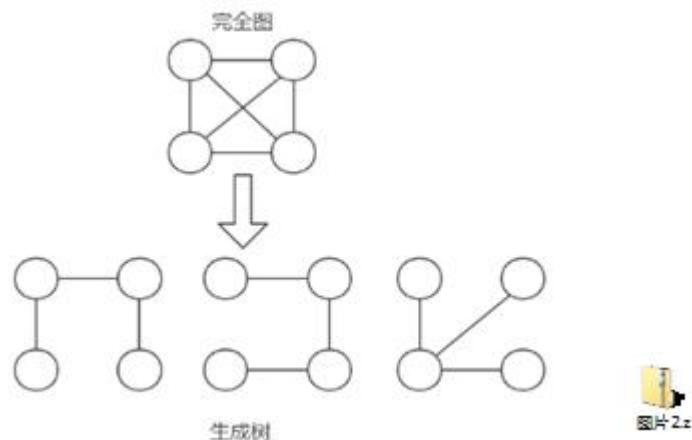
正确的思路，就是尽可能的选择少的路线，并且每条路线最小，保证总里程数最少。

14.6.2 最小生成树

修路问题本质就是就是**最小生成树问题**，先介绍一下**最小生成树(Minimum Cost Spanning Tree)**，简称 **MST**。给定一个带权的无向连通图,如何选取一棵生成树,使树上所有边上权的总和为最小,这叫**最小生成树**

- 1) N 个顶点，一定有 N-1 条边
- 2) 包含全部顶点
- 3) N-1 条边都在图中
- 4) 举例说明(如图:)

5) 求最小生成树的算法主要是普里姆算法和克鲁斯卡尔算法

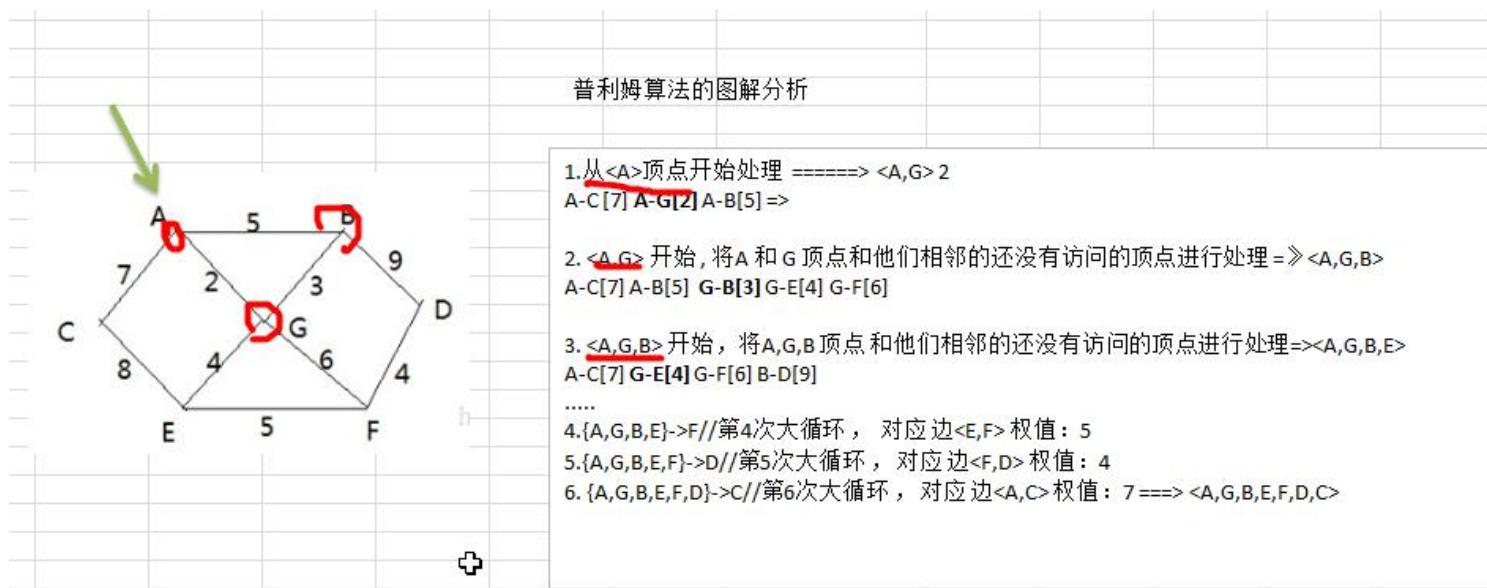


14.6.3 普里姆算法介绍

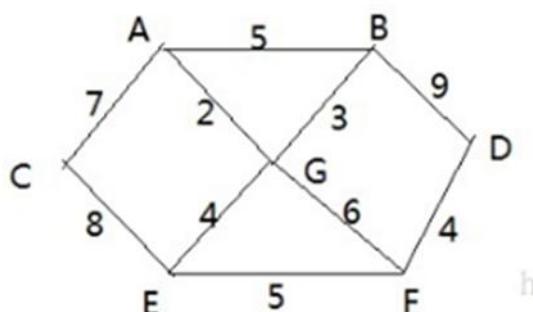
普利姆(Prim)算法求最小生成树，也就是在包含 n 个顶点的连通图中，找出只有 $(n-1)$ 条边包含所有 n 个顶点的连通子图，也就是所谓的极小连通子图

普利姆的算法如下：

- 1) 设 $G=(V,E)$ 是连通网， $T=(U,D)$ 是最小生成树， V,U 是顶点集合， E,D 是边的集合
- 2) 若从顶点 u 开始构造最小生成树，则从集合 V 中取出顶点 u 放入集合 U 中， 标记顶点 v 的 $\text{visited}[v]=1$
- 3) 若集合 U 中顶点 ui 与集合 $V-U$ 中的顶点 vj 之间存在边，则寻找这些边中权值最小的边，但不能构成回路，将顶点 vj 加入集合 U 中， 将边 (ui,vj) 加入集合 D 中， 标记 $\text{visited}[vj]=1$
- 4) 重复步骤②， 直到 U 与 V 相等， 即所有顶点都被标记为访问过， 此时 D 中有 $n-1$ 条边
- 5) 提示：单独看步骤很难理解，我们通过代码来讲解，比较好理解。
- 6) 图解普利姆算法



14.6.4 普里姆算法最佳实践(修路问题)



- 1) 有胜利乡有 7 个村庄(A, B, C, D, E, F, G) , 现在需要修路把 7 个村庄连通
- 2) 各个村庄的距离用边线表示(权) , 比如 A - B 距离 5 公里
- 3) 问: 如何修路保证各个村庄都能连通, 并且总的修建公路总里程最短?
- 4) 看老师思路分析+代码演示:

```
package com.atguigu.prim;
```

```
import java.util.Arrays;
```



```
public class PrimAlgorithm {  
  
    public static void main(String[] args) {  
        //测试看看图是否创建 ok  
        char[] data = new char[]{'A','B','C','D','E','F','G'};  
        int verxs = data.length;  
        //邻接矩阵的关系使用二维数组表示,10000 这个大数, 表示两个点不联通  
        int [][]weight=new int[][]{  
            {10000,5,7,10000,10000,10000,2},  
            {5,10000,10000,9,10000,10000,3},  
            {7,10000,10000,10000,8,10000,10000},  
            {10000,9,10000,10000,10000,4,10000},  
            {10000,10000,8,10000,10000,5,4},  
            {10000,10000,10000,4,5,10000,6},  
            {2,3,10000,10000,4,6,10000},};  
  
        //创建 MGraph 对象  
        MGraph graph = new MGraph(verxs);  
        //创建一个 MinTree 对象  
        MinTree minTree = new MinTree();  
        minTree.createGraph(graph, verxs, data, weight);  
        //输出  
        minTree.showGraph(graph);  
        //测试普利姆算法  
        minTree.prim(graph, 1);  
    }  
}
```



```
}

}

//创建最小生成树->村庄的图

class MinTree {

    //创建图的邻接矩阵

    /**
     *
     * @param graph 图对象
     * @param verxs 图对应的顶点个数
     * @param data 图的各个顶点的值
     * @param weight 图的邻接矩阵
     */

    public void createGraph(MGraph graph, int verxs, char data[], int[][] weight) {

        int i, j;
        for(i = 0; i < verxs; i++) {//顶点
            graph.data[i] = data[i];
            for(j = 0; j < verxs; j++) {
                graph.weight[i][j] = weight[i][j];
            }
        }
    }

    //显示图的邻接矩阵

    public void showGraph(MGraph graph) {
```



```
for(int[] link: graph.weight) {  
    System.out.println(Arrays.toString(link));  
}  
}  
  
//编写 prim 算法，得到最小生成树  
/**  
 *  
 * @param graph 图  
 * @param v 表示从图的第几个顶点开始生成'A'->0 'B'->1...  
 */  
public void prim(MGraph graph, int v) {  
    //visited[] 标记结点(顶点)是否被访问过  
    int visited[] = new int[graph.verxs];  
    //visited[] 默认元素的值都是 0, 表示没有访问过  
    //    for(int i=0; i<graph.verxs; i++) {  
    //        visited[i] = 0;  
    //    }  
  
    //把当前这个结点标记为已访问  
    visited[v] = 1;  
    //h1 和 h2 记录两个顶点的下标  
    int h1 = -1;  
    int h2 = -1;  
    int minWeight = 10000; //将 minWeight 初始成一个大数，后面在遍历过程中，会被替换  
    for(int k = 1; k < graph.verxs; k++) {//因为有 graph.verxs 顶点，普利姆算法结束后，有 graph.verxs-1 边
```



```
//这个是确定每一次生成的子图， 和哪个结点的距离最近
for(int i = 0; i < graph.verxs; i++) {// i 结点表示被访问过的结点
    for(int j = 0; j < graph.verxs; j++) {//j 结点表示还没有访问过的结点
        if(visited[i] == 1 && visited[j] == 0 && graph.weight[i][j] < minWeight) {
            //替换 minWeight(寻找已经访问过的结点和未访问过的结点间的权值最小的边)
            minWeight = graph.weight[i][j];
            h1 = i;
            h2 = j;
        }
    }
}
//找到一条边是最小
System.out.println("边<" + graph.data[h1] + "," + graph.data[h2] + "> 权值:" + minWeight);
//将当前这个结点标记为已经访问
visited[h2] = 1;
//minWeight 重新设置为最大值 10000
minWeight = 10000;
}

}

class MGraph {
    int verxs; //表示图的节点个数
    char[] data;//存放结点数据
}
```

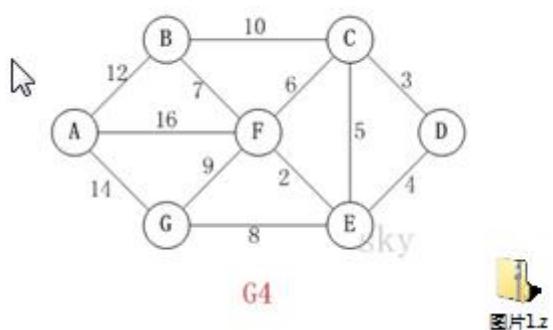
```
int[][] weight; //存放边，就是我们的邻接矩阵
```

```
public MGraph(int verxs) {  
    this.verxs = verxs;  
    data = new char[verxs];  
    weight = new int[verxs][verxs];  
}  
}
```

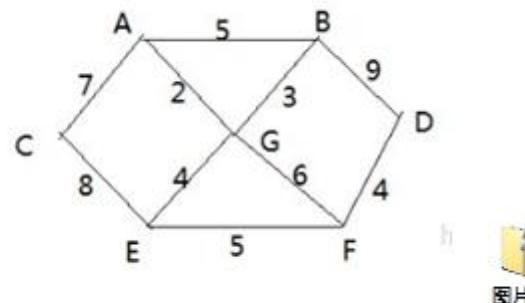
14.7 克鲁斯卡尔算法

14.7.1 应用场景-公交站问题

看一个应用场景和问题：



类似的



- 1) 某城市新增 7 个站点(A, B, C, D, E, F, G) , 现在需要修路把 7 个站点连通
- 2) 各个站点的距离用边线表示(权) , 比如 A - B 距离 12 公里
- 3) 问：如何修路保证各个站点都能连通，并且总的修建公路总里程最短？

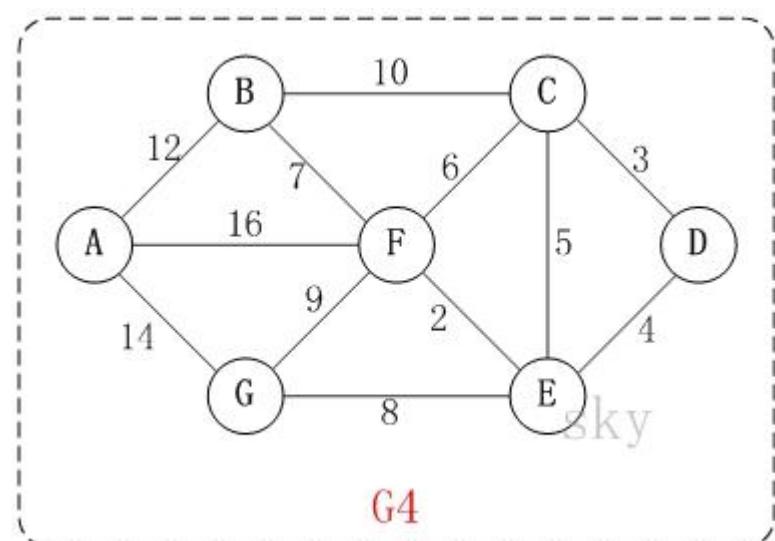
14.7.2 克鲁斯卡尔算法介绍

- 1) 克鲁斯卡尔(Kruskal)算法，是用来求加权连通图的最小生成树的算法。
- 2) 基本思想：按照权值从小到大的顺序选择 $n-1$ 条边，并保证这 $n-1$ 条边不构成回路
- 3) 具体做法：首先构造一个只含 n 个顶点的森林，然后依权值从小到大从连通网中选择边加入到森林中，并使森林中不产生回路，直至森林变成一棵树为止

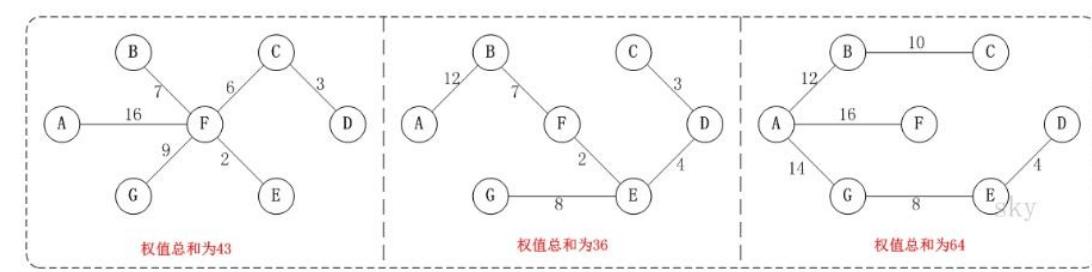
14.7.3 克鲁斯卡尔算法图解说明

以城市公交站问题来图解说明 克鲁斯卡尔算法的原理和步骤：

在含有 n 个顶点的连通图中选择 $n-1$ 条边，构成一棵极小连通子图，并使该连通子图中 $n-1$ 条边上权值之和达到最小，则称其为连通网的最小生成树。



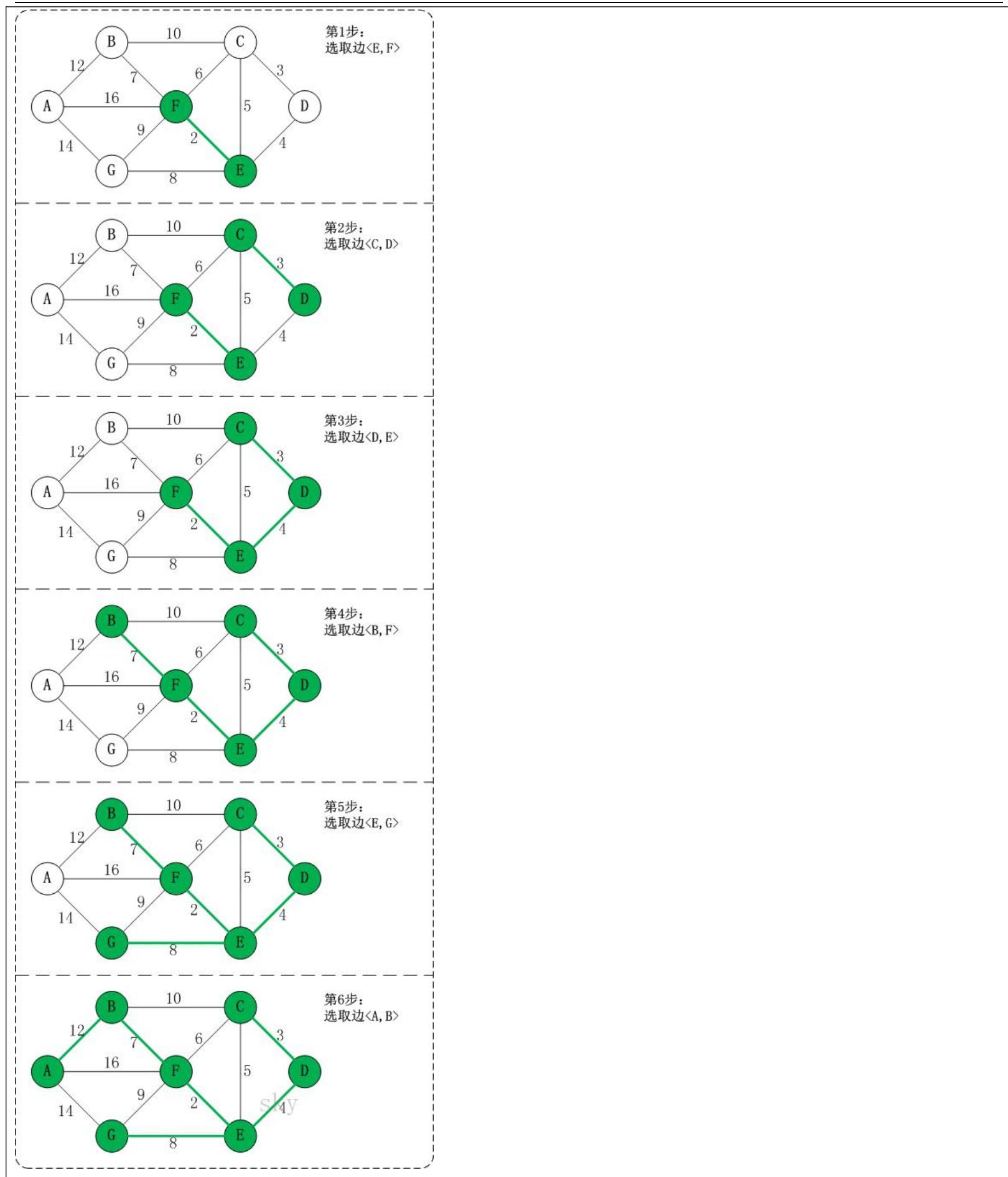
例如，对于如上图 G4 所示的连通网可以有多棵权值总和不相同的生成树。





3.1.1 克鲁斯卡尔算法图解

以上图 G4 为例，来对克鲁斯卡尔进行演示(假设，用数组 R 保存最小生成树结果)。





第 1 步：将边 $\langle E, F \rangle$ 加入 R 中。

边 $\langle E, F \rangle$ 的权值最小，因此将它加入到最小生成树结果 R 中。

第 2 步：将边 $\langle C, D \rangle$ 加入 R 中。

上一步操作之后，边 $\langle C, D \rangle$ 的权值最小，因此将它加入到最小生成树结果 R 中。

第 3 步：将边 $\langle D, E \rangle$ 加入 R 中。

上一步操作之后，边 $\langle D, E \rangle$ 的权值最小，因此将它加入到最小生成树结果 R 中。

第 4 步：将边 $\langle B, F \rangle$ 加入 R 中。

上一步操作之后，边 $\langle C, E \rangle$ 的权值最小，但 $\langle C, E \rangle$ 会和已有的边构成回路；因此，跳过边 $\langle C, E \rangle$ 。同理，跳过边 $\langle C, F \rangle$ 。将边 $\langle B, F \rangle$ 加入到最小生成树结果 R 中。

第 5 步：将边 $\langle E, G \rangle$ 加入 R 中。

上一步操作之后，边 $\langle E, G \rangle$ 的权值最小，因此将它加入到最小生成树结果 R 中。

第 6 步：将边 $\langle A, B \rangle$ 加入 R 中。

上一步操作之后，边 $\langle F, G \rangle$ 的权值最小，但 $\langle F, G \rangle$ 会和已有的边构成回路；因此，跳过边 $\langle F, G \rangle$ 。同理，跳过边 $\langle B, C \rangle$ 。将边 $\langle A, B \rangle$ 加入到最小生成树结果 R 中。

此时，最小生成树构造完成！它包括的边依次是： $\langle E, F \rangle \langle C, D \rangle \langle D, E \rangle \langle B, F \rangle \langle E, G \rangle \langle A, B \rangle$ 。

3.1.2 克鲁斯卡尔算法分析

根据前面介绍的克鲁斯卡尔算法的基本思想和做法，我们能够了解到，克鲁斯卡尔算法重点需要解决的以下两个问题：

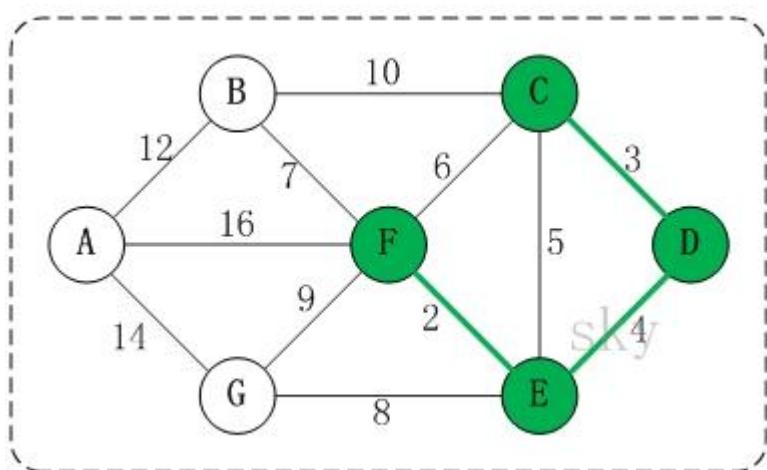
问题一 对图的所有边按照权值大小进行排序。

问题二 将边添加到最小生成树中时，怎么样判断是否形成了回路。

问题一很好解决，采用排序算法进行排序即可。

问题二，处理方式是：记录顶点在“最小生成树”中的终点，顶点的终点是“在最小生成树中与它连通的最大顶点”。然后每次需要将一条边添加到最小生存树时，判断该边的两个顶点的终点是否重合，重合的话则会构成回路。

3.1.3 如何判断是否构成回路-举例说明(如图)



在将 $\langle E, F \rangle$ $\langle C, D \rangle$ $\langle D, E \rangle$ 加入到最小生成树 R 中之后，这几条边的顶点就都有了终点：

- (01) C 的终点是 F。
- (02) D 的终点是 F。
- (03) E 的终点是 F。
- (04) F 的终点是 F。

关于终点的说明：

- 1) 就是将所有顶点按照从小到大的顺序排列好之后；某个顶点的终点就是“与它连通的最大顶点”。
- 2) 因此，接下来，虽然 $\langle C, E \rangle$ 是权值最小的边。但是 C 和 E 的终点都是 F，即它们的终点相同，因此，将 $\langle C, E \rangle$ 加入最小生成树的话，会形成回路。这就是判断回路的方式。也就是说，我们加入的边的两个顶点不能都指向同一个终点，否则将构成回路。【后面有代码说明】

3.1.4 克鲁斯卡尔算法的代码说明

14.7.4 克鲁斯卡尔最佳实践-公交站问题

看一个公交站问题：



- 1) 有北京有新增 7 个站点(A, B, C, D, E, F, G) , 现在需要修路把 7 个站点连通
- 2) 各个站点的距离用边线表示(权) , 比如 A - B 距离 12 公里
- 3) 问: 如何修路保证各个站点都能连通, 并且总的修建公路总里程最短?
- 4) 代码实现和注解

```
package com.atguigu.kruskal;

import java.util.Arrays;

public class KruskalCase {

    private int edgeNum; //边的个数
    private char[] vertexs; //顶点数组
    private int[][] matrix; //邻接矩阵
    //使用 INF 表示两个顶点不能连通
    private static final int INF = Integer.MAX_VALUE;

    public static void main(String[] args) {
        char[] vertexs = {'A', 'B', 'C', 'D', 'E', 'F', 'G'};
        //克鲁斯卡尔算法的邻接矩阵
        int matrix[][] = {
            /*A*//*B*//*C*//*D*//*E*//*F*//*G*/
            /*A*/ { 0, 12, INF, INF, INF, 16, 14 },
            /*B*/ { 12, 0, 10, INF, INF, 7, INF },
            /*C*/ { INF, 10, 0, 3, 5, 6, INF },
            /*D*/ { INF, INF, 3, 0, 4, INF, INF },
            /*E*/ { INF, INF, INF, 4, 0, 2, 1 },
            /*F*/ { INF, INF, INF, INF, 2, 0, 6 },
            /*G*/ { INF, INF, INF, INF, INF, 6, 0 }
        };
    }
}
```



```
/*E*/ { INF, INF,    5,    4,    0,    2,    8},  
/*F*/ {  16,    7,    6, INF,    2,    0,    9},  
/*G*/ {  14, INF, INF, INF,    8,    9,    0}};  
//大家可以在去测试其它的邻接矩阵，结果都可以得到最小生成树.
```

//创建 KruskalCase 对象实例

```
KruskalCase kruskalCase = new KruskalCase(vertexs, matrix);
```

//输出构建的

```
kruskalCase.print();
```

```
kruskalCase.kruskal();
```

```
}
```

//构造器

```
public KruskalCase(char[] vertexs, int[][] matrix) {
```

//初始化顶点数和边的个数

```
int vlen = vertexs.length;
```

//初始化顶点，复制拷贝的方式

```
this.vertexs = new char[vlen];
```

```
for(int i = 0; i < vertexs.length; i++) {
```

```
    this.vertexs[i] = vertexs[i];
```

```
}
```

//初始化边，使用的是复制拷贝的方式

```
this.matrix = new int[vlen][vlen];
```



```
for(int i = 0; i < vlen; i++) {  
    for(int j= 0; j < vlen; j++) {  
        this.matrix[i][j] = matrix[i][j];  
    }  
}  
  
//统计边的条数  
  
for(int i =0; i < vlen; i++) {  
    for(int j = i+1; j < vlen; j++) {  
        if(this.matrix[i][j] != INF) {  
            edgeNum++;  
        }  
    }  
}  
  
}  
  
public void kruskal() {  
    int index = 0; //表示最后结果数组的索引  
    int[] ends = new int[edgeNum]; //用于保存"已有最小生成树" 中的每个顶点在最小生成树中的终点  
    //创建结果数组, 保存最后的最小生成树  
    EData[] rets = new EData[edgeNum];  
  
    //获取图中 所有的边的集合 , 一共有 12 边  
    EData[] edges = getEdges();  
    System.out.println("图的边的集合=" + Arrays.toString(edges) + " 共"+ edges.length); //12  
  
    //按照边的权值大小进行排序(从小到大)
```



```
sortEdges(edges);
```

//遍历 edges 数组, 将边添加到最小生成树中时, 判断是准备加入的边否形成了回路, 如果没有, 就加入 rets, 否则不能加入

```
for(int i=0; i < edgeNum; i++) {
```

//获取到第 i 条边的第一个顶点(起点)

```
int p1 = getPosition(edges[i].start); //p1=4
```

//获取到第 i 条边的第 2 个顶点

```
int p2 = getPosition(edges[i].end); //p2 = 5
```

//获取 p1 这个顶点在已有最小生成树中的终点

```
int m = getEnd(ends, p1); //m = 4
```

//获取 p2 这个顶点在已有最小生成树中的终点

```
int n = getEnd(ends, p2); // n = 5
```

//是否构成回路

```
if(m != n) { //没有构成回路
```

```
    ends[m] = n; // 设置 m 在"已有最小生成树"中的终点 <E,F> [0,0,0,0,5,0,0,0,0,0,0]
```

```
    rets[index++] = edges[i]; //有一条边加入到 rets 数组
```

```
}
```

```
}
```

//<E,F> <C,D> <D,E> <B,F> <E,G> <A,B>。

//统计并打印 "最小生成树", 输出 rets

```
System.out.println("最小生成树为");
```

```
for(int i = 0; i < index; i++) {
```

```
    System.out.println(rets[i]);
```

```
}
```



```
}
```

```
//打印邻接矩阵
```

```
public void print() {
```

```
    System.out.println("邻接矩阵为: \n");
```

```
    for(int i = 0; i < vertexs.length; i++) {
```

```
        for(int j=0; j < vertexs.length; j++) {
```

```
            System.out.printf("%12d", matrix[i][j]);
```

```
        }
```

```
        System.out.println();//换行
```

```
}
```

```
}
```

```
/**
```

```
* 功能：对边进行排序处理，冒泡排序
```

```
* @param edges 边的集合
```

```
*/
```

```
private void sortEdges(EData[] edges) {
```

```
    for(int i = 0; i < edges.length - 1; i++) {
```

```
        for(int j = 0; j < edges.length - 1 - i; j++) {
```

```
            if(edges[j].weight > edges[j+1].weight) {//交换
```

```
                EData tmp = edges[j];
```

```
                edges[j] = edges[j+1];
```

```
                edges[j+1] = tmp;
```



```
        }

    }

}

/**

 *

 * @param ch 顶点的值, 比如'A','B'

 * @return 返回 ch 顶点对应的下标, 如果找不到, 返回-1

 */

private int getPosition(char ch) {

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

        if(vertexs[i] == ch) {//找到

            return i;

        }

    }

    //找不到,返回-1

    return -1;

}

/**

 * 功能: 获取图中边, 放到 EData[] 数组中, 后面我们需要遍历该数组

 * 是通过 matrix 邻接矩阵来获取

 * EData[] 形式 [[A,B,12],[B,F,7],.....]

 * @return

 */

private EData[] getEdges() {

    int index = 0;
```



```
EData[] edges = new EData[edgeNum];  
  
for(int i = 0; i < vertexs.length; i++) {  
  
    for(int j=i+1; j < vertexs.length; j++) {  
  
        if(matrix[i][j] != INF) {  
  
            edges[index++] = new EData(vertexs[i], vertexs[j], matrix[i][j]);  
        }  
    }  
}  
  
return edges;  
}  
  
/**  
 * 功能: 获取下标为 i 的顶点的终点(), 用于后面判断两个顶点的终点是否相同  
 * @param ends : 数组就是记录了各个顶点对应的终点是哪个,ends 数组是在遍历过程中, 逐步形成  
 * @param i : 表示传入的顶点对应的下标  
 * @return 返回的就是 下标为 i 的这个顶点对应的终点的下标, 一会回头还有来理解  
 */  
  
private int getEnd(int[] ends, int i) { // i = 4 [0,0,0,0,5,0,0,0,0,0,0]  
  
    while(ends[i] != 0) {  
  
        i = ends[i];  
    }  
  
    return i;  
}  
  
}  
  
//创建一个类 EData , 它的对象实例就表示一条边
```

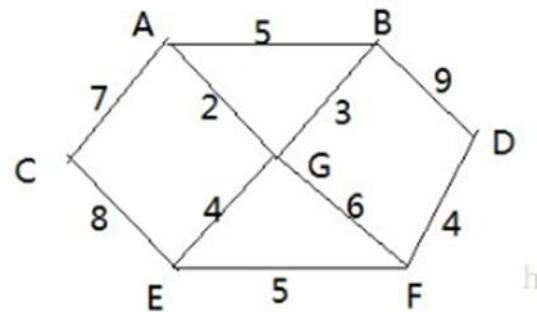


```
class EData {  
    char start; //边的一个点  
    char end; //边的另外一个点  
    int weight; //边的权值  
    //构造器  
    public EData(char start, char end, int weight) {  
        this.start = start;  
        this.end = end;  
        this.weight = weight;  
    }  
    //重写 toString, 便于输出边信息  
    @Override  
    public String toString() {  
        return "EData [<" + start + ", " + end + ">= " + weight + "]";  
    }  
}
```

14.8 迪杰斯特拉算法

14.8.1 应用场景-最短路径问题

看一个应用场景和问题：



- 1) 战争时期，胜利乡有 7 个村庄(A, B, C, D, E, F, G) , 现在有六个邮差，从 G 点出发，需要分别把邮件分别送到 A, B, C, D, E, F 六个村庄
- 2) 各个村庄的距离用边线表示(权) , 比如 A - B 距离 5 公里
- 3) 问：如何计算出 G 村庄到 其它各个村庄的最短距离?
- 4) 如果从其它点出发到各个点的最短距离又是多少?

14.8.2 迪杰斯特拉(Dijkstra)算法介绍

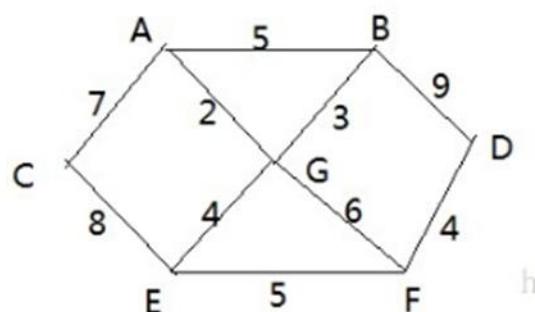
迪杰斯特拉(Dijkstra)算法是典型最短路径算法，用于计算一个结点到其他结点的最短路径。它的主要特点是以起始点为中心向外层层扩展(广度优先搜索思想)，直到扩展到终点为止。

14.8.3 迪杰斯特拉(Dijkstra)算法过程

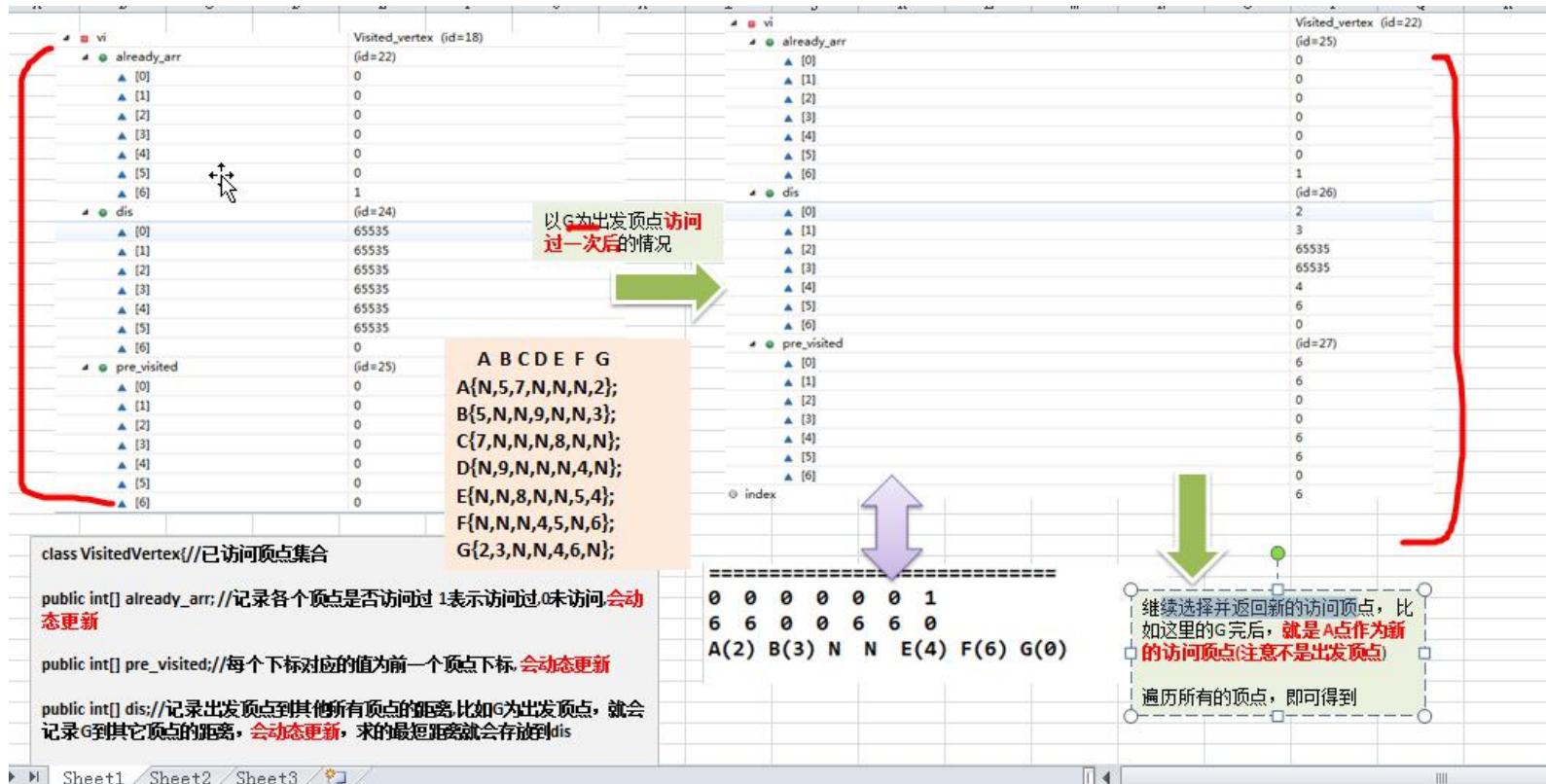
- 1) 设置出发顶点为 v，顶点集合 $V\{v_1, v_2, v_i, \dots\}$ ， v 到 V 中各顶点的距离构成距离集合 Dis, $Dis\{d_1, d_2, d_i, \dots\}$ ，Dis 集合记录着 v 到图中各顶点的距离(到自身可以看作 0, v 到 v_i 距离对应为 d_i)
- 2) 从 Dis 中选择值最小的 d_i 并移出 Dis 集合，同时移出 V 集合中对应的顶点 v_i ，此时的 v 到 v_i 即为最短路径

- 3) 更新 Dis 集合，更新规则为：比较 v 到 V 集合中顶点的距离值，与 v 通过 vi 到 V 集合中顶点的距离值，保留值较小的一个(同时也应该更新顶点的前驱节点为 vi，表明是通过 vi 到达的)
- 4) 重复执行两步骤，直到最短路径顶点为目标顶点即可结束

14.8.4 迪杰斯特拉(Dijkstra)算法最佳应用-最短路径



- 1) 战争时期，胜利乡有 7 个村庄(A, B, C, D, E, F, G) ，现在有六个邮差，从 G 点出发，需要分别把邮件分别送到 A, B, C , D, E, F 六个村庄
- 2) 各个村庄的距离用边线表示(权) ，比如 A - B 距离 5 公里
- 3) 问：如何计算出 G 村庄到 其它各个村庄的最短距离？
- 4) 如果从其它点出发到各个点的最短距离又是多少？
- 5) 使用图解的方式分析了迪杰斯特拉(Dijkstra)算法 思路



6) 代码实现

```

package com.atguigu.dijkstra;

import java.util.Arrays;

public class DijkstraAlgorithm {

    public static void main(String[] args) {
        char[] vertex = { 'A', 'B', 'C', 'D', 'E', 'F', 'G' };
        //邻接矩阵
        int[][] matrix = new int[vertex.length][vertex.length];
        final int N = 65535; // 表示不可以连接
    }
}

```



```
matrix[0]=new int[]{N,5,7,N,N,N,2};  
matrix[1]=new int[]{5,N,N,9,N,N,3};  
matrix[2]=new int[]{7,N,N,N,8,N,N};  
matrix[3]=new int[]{N,9,N,N,N,4,N};  
matrix[4]=new int[]{N,N,8,N,N,5,4};  
matrix[5]=new int[]{N,N,N,4,5,N,6};  
matrix[6]=new int[]{2,3,N,N,4,6,N};  
//创建 Graph 对象  
  
Graph graph = new Graph(vertex, matrix);  
//测试，看看图的邻接矩阵是否 ok  
graph.showGraph();  
//测试迪杰斯特拉算法  
graph.dsj(2);//C  
graph.showDijkstra();  
  
}  
  
}  
  
class Graph {  
    private char[] vertex; // 顶点数组  
    private int[][] matrix; // 邻接矩阵  
    private VisitedVertex vv; //已经访问的顶点的集合  
  
    // 构造器
```



```
public Graph(char[] vertex, int[][] matrix) {  
    this.vertex = vertex;  
    this.matrix = matrix;  
}  
  
//显示结果  
public void showDijkstra() {  
    vv.show();  
}  
  
// 显示图  
public void showGraph() {  
    for (int[] link : matrix) {  
        System.out.println(Arrays.toString(link));  
    }  
}  
  
//迪杰斯特拉算法实现  
/**  
 *  
 * @param index 表示出发顶点对应的下标  
 */  
public void dsj(int index) {  
    vv = new VisitedVertex(vertex.length, index);  
    update(index);//更新 index 顶点到周围顶点的距离和前驱顶点  
    for(int j = 1; j < vertex.length; j++) {
```



```
index = vv.updateArr(); // 选择并返回新的访问顶点
update(index); // 更新 index 顶点到周围顶点的距离和前驱顶点
}

}

//更新 index 下标顶点到周围顶点的距离和周围顶点的前驱顶点,
private void update(int index) {
    int len = 0;
    //根据遍历我们的邻接矩阵的 matrix[index]行
    for(int j = 0; j < matrix[index].length; j++) {
        // len 含义是 : 出发顶点到 index 顶点的距离 + 从 index 顶点到 j 顶点的距离的和
        len = vv.getDis(index) + matrix[index][j];
        // 如果 j 顶点没有被访问过, 并且 len 小于出发顶点到 j 顶点的距离, 就需要更新
        if(!vv.in(j) && len < vv.getDis(j)) {
            vv.updatePre(j, index); //更新 j 顶点的前驱为 index 顶点
            vv.updateDis(j, len); //更新出发顶点到 j 顶点的距离
        }
    }
}

// 已访问顶点集合
class VisitedVertex {
    // 记录各个顶点是否访问过 1 表示访问过,0 未访问,会动态更新
```



```
public int[] already_arr;  
// 每个下标对应的值为前一个顶点下标, 会动态更新  
public int[] pre_visited;  
// 记录出发顶点到其他所有顶点的距离,比如 G 为出发顶点, 就会记录 G 到其它顶点的距离, 会动态更新, 求的最短距离就会存放到 dis  
public int[] dis;  
  
//构造器  
/**  
 *  
 * @param length :表示顶点的个数  
 * @param index: 出发顶点对应的下标, 比如 G 顶点, 下标就是 6  
 */  
public VisitedVertex(int length, int index) {  
    this.already_arr = new int[length];  
    this.pre_visited = new int[length];  
    this.dis = new int[length];  
    //初始化 dis 数组  
    Arrays.fill(dis, 65535);  
    this.already_arr[index] = 1; //设置出发顶点被访问过  
    this.dis[index] = 0;//设置出发顶点的访问距离为 0  
  
}  
/**  
 * 功能: 判断 index 顶点是否被访问过  
 * @param index
```



```
* @return 如果访问过，就返回 true, 否则访问 false
*/
public boolean in(int index) {
    return already_arr[index] == 1;
}

/**
 * 功能: 更新出发顶点到 index 顶点的距离
 * @param index
 * @param len
 */
public void updateDis(int index, int len) {
    dis[index] = len;
}

/**
 * 功能: 更新 pre 这个顶点的前驱顶点为 index 顶点
 * @param pre
 * @param index
 */
public void updatePre(int pre, int index) {
    pre_visited[pre] = index;
}

/**
 * 功能:返回出发顶点到 index 顶点的距离
 * @param index
 */
```



```
public int getDis(int index) {  
    return dis[index];  
}  
  
/**  
 * 继续选择并返回新的访问顶点， 比如这里的 G 完后，就是 A 点作为新的访问顶点(注意不是出发顶点)  
 * @return  
 */  
  
public int updateArr() {  
    int min = 65535, index = 0;  
    for(int i = 0; i < already_arr.length; i++) {  
        if(already_arr[i] == 0 && dis[i] < min ) {  
            min = dis[i];  
            index = i;  
        }  
    }  
    //更新 index 顶点被访问过  
    already_arr[index] = 1;  
    return index;  
}  
  
//显示最后的结果  
//即将三个数组的情况输出  
public void show() {
```



```
System.out.println("=====");
//输出 already_arr
for(int i : already_arr) {
    System.out.print(i + " ");
}
System.out.println();
//输出 pre_visited
for(int i : pre_visited) {
    System.out.print(i + " ");
}
System.out.println();
//输出 dis
for(int i : dis) {
    System.out.print(i + " ");
}
System.out.println();
//为了好看最后的最短距离，我们处理
char[] vertex = { 'A', 'B', 'C', 'D', 'E', 'F', 'G' };
int count = 0;
for (int i : dis) {
    if (i != 65535) {
        System.out.print(vertex[count] + "(" + i + ") ");
    } else {
        System.out.println("N ");
    }
    count++;
}
```



```
}

System.out.println();

}

}
```

14.9 弗洛伊德算法

14.9.1 弗洛伊德(Floyd)算法介绍

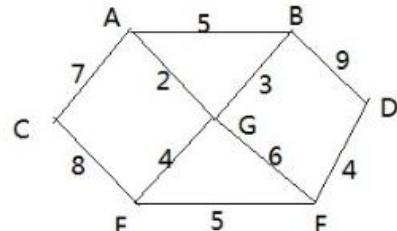
- 1) 和 Dijkstra 算法一样，弗洛伊德(Floyd)算法也是一种用于寻找给定的加权图中顶点间最短路径的算法。该算法名称以创始人之一、1978 年图灵奖获得者、斯坦福大学计算机科学系教授罗伯特·弗洛伊德命名
- 2) 弗洛伊德算法(Floyd)计算图中各个顶点之间的最短路径
- 3) 迪杰斯特拉算法用于计算图中某一个顶点到其他顶点的最短路径。
- 4) 弗洛伊德算法 VS 迪杰斯特拉算法：迪杰斯特拉算法通过选定的被访问顶点，求出从出发访问顶点到其他顶点的最短路径；弗洛伊德算法中每一个顶点都是出发访问点，所以需要将每一个顶点看做被访问顶点，求出从每一个顶点到其他顶点的最短路径。

14.9.2 弗洛伊德(Floyd)算法图解分析

- 1) 设置顶点 v_i 到顶点 v_k 的最短路径已知为 L_{ik} ，顶点 v_k 到 v_j 的最短路径已知为 L_{kj} ，顶点 v_i 到 v_j 的路径为 L_{ij} ，则 v_i 到 v_j 的最短路径为： $\min((L_{ik}+L_{kj}), L_{ij})$ ， v_k 的取值为图中所有顶点，则可获得 v_i 到 v_j 的最短路径
- 2) 至于 v_i 到 v_k 的最短路径 L_{ik} 或者 v_k 到 v_j 的最短路径 L_{kj} ，是以同样的方式获得

3) 弗洛伊德(Floyd)算法图解分析-举例说明

示例：求最短路径为例说明



示意图

	A	B	C	D	E	F	G
0	0	1	2	3	4	5	6
A	0						
B	5	0	N	9	N	N	3
C	7	N	0	N	8	N	N
D	N	9	N	0	N	4	N
E	N	N	8	N	0	5	4
F	N	N	N	4	5	0	6
G	2	3	N	N	4	6	0

初始各顶点之间距离表

	A	B	C	D	E	F	G
A	A	A	A	A	A	A	A
B	B	B	B	B	B	B	B
C	C	C	C	C	C	C	C
D	D	D	D	D	D	D	D
E	E	E	E	E	E	E	E
F	F	F	F	F	F	F	F
G	G	G	G	G	G	G	G

初始顶点前驱关系

弗洛伊德算法的步骤：

第一轮循环中，以 A(下标为: 0)作为中间顶点【即把 A 作为中间顶点的所有情况都进行遍历，就会得到更新距离表 和 前驱关系】，距离表和前驱关系更新为：

	A	B	C	D	E	F	G
0	0	1	2	3	4	5	6
A	0	0	5	7	N	N	N
B	1	5	0	12	9	N	N
C	2	7	12	0	N	8	N
D	3	N	9	N	0	N	4
E	4	N	N	8	N	0	5
F	5	N	N	N	4	5	0
G	6	2	3	9	N	4	6

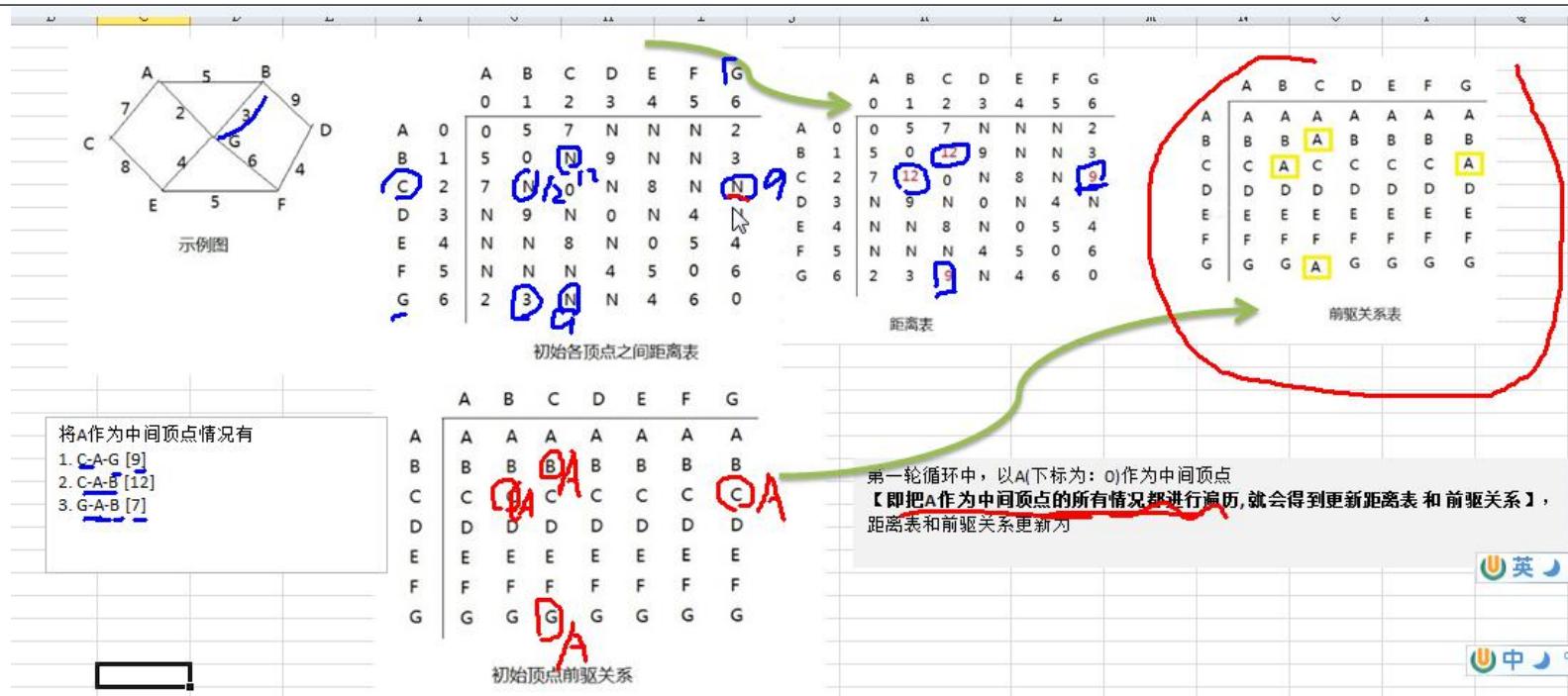
距离表

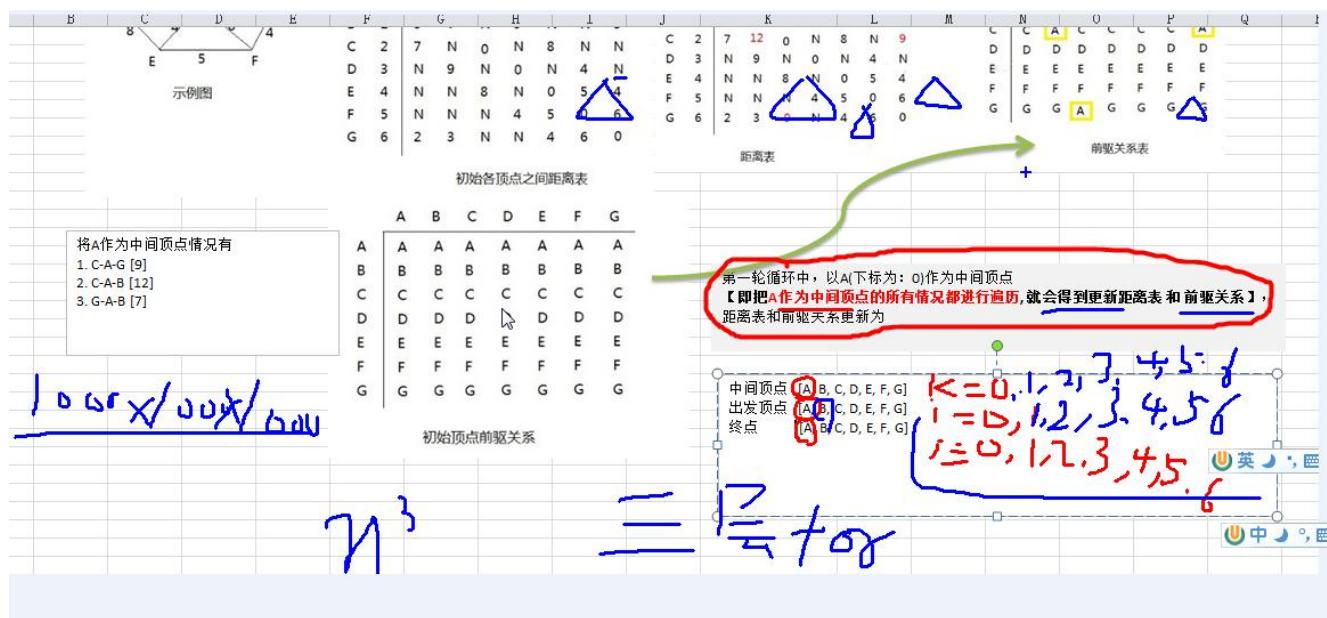
	A	B	C	D	E	F	G
A	A	A	A	A	A	A	A
B	B	B	A	B	B	B	B
C	C	A	C	C	C	C	A
D	D	D	D	D	D	D	D
E	E	E	E	E	E	E	E
F	F	F	F	F	F	F	F
G	G	G	A	G	G	G	G

前驱关系表

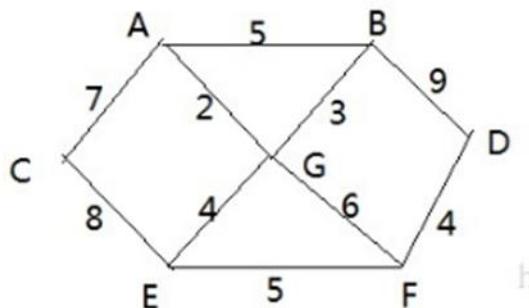
分析如下：

- 1) 以 A 顶点作为中间顶点是，B->A->C 的距离由 N->9，同理 C 到 B；C->A->G 的距离由 N->12，同理 G 到 C
- 2) 更换中间顶点，循环执行操作，直到所有顶点都作为中间顶点更新后，计算结束





14.9.3 弗洛伊德(Floyd)算法最佳应用-最短路径



- 1) 胜利乡有 7 个村庄(A, B, C, D, E, F, G)
- 2) 各个村庄的距离用边线表示(权)，比如 A - B 距离 5 公里
- 3) 问：如何计算出各村庄到其它各村庄的最短距离？
- 4) 代码实现

```
package com.atguigu.floyd;
```



```
import java.util.Arrays;

public class FloydAlgorithm {

    public static void main(String[] args) {
        // 测试看看图是否创建成功
        char[] vertex = { 'A', 'B', 'C', 'D', 'E', 'F', 'G' };
        // 创建邻接矩阵
        int[][] matrix = new int[vertex.length][vertex.length];
        final int N = 65535;
        matrix[0] = new int[] { 0, 5, 7, N, N, N, 2 };
        matrix[1] = new int[] { 5, 0, N, 9, N, N, 3 };
        matrix[2] = new int[] { 7, N, 0, N, 8, N, N };
        matrix[3] = new int[] { N, 9, N, 0, N, 4, N };
        matrix[4] = new int[] { N, N, 8, N, 0, 5, 4 };
        matrix[5] = new int[] { N, N, N, 4, 5, 0, 6 };
        matrix[6] = new int[] { 2, 3, N, N, 4, 6, 0 };

        // 创建 Graph 对象
        Graph graph = new Graph(vertex.length, matrix, vertex);
        // 调用弗洛伊德算法
        graph.floyd();
        graph.show();
    }
}
```



```
// 创建图

class Graph {

    private char[] vertex; // 存放顶点的数组

    private int[][] dis; // 保存，从各个顶点出发到其它顶点的距离，最后的结果，也是保留在该数组

    private int[][] pre; // 保存到达目标顶点的前驱顶点


    // 构造器

    /**
     *
     * @param length
     *          大小
     * @param matrix
     *          邻接矩阵
     * @param vertex
     *          顶点数组
     */

    public Graph(int length, int[][] matrix, char[] vertex) {
        this.vertex = vertex;
        this.dis = matrix;
        this.pre = new int[length][length];
        // 对 pre 数组初始化，注意存放的是前驱顶点的下标
        for (int i = 0; i < length; i++) {
            Arrays.fill(pre[i], i);
        }
    }
}
```



```
// 显示 pre 数组和 dis 数组
public void show() {

    //为了显示便于阅读，我们优化一下输出
    char[] vertex = { 'A', 'B', 'C', 'D', 'E', 'F', 'G' };
    for (int k = 0; k < dis.length; k++) {
        // 先将 pre 数组输出的一行
        for (int i = 0; i < dis.length; i++) {
            System.out.print(vertex[pre[k][i]] + " ");
        }
        System.out.println();
        // 输出 dis 数组的一行数据
        for (int i = 0; i < dis.length; i++) {
            System.out.print("(" + vertex[k] + "到" + vertex[i] + "的最短路径是" + dis[k][i] + ") ");
        }
        System.out.println();
        System.out.println();
    }

}

//弗洛伊德算法，比较容易理解，而且容易实现
public void floyd() {
    int len = 0; //变量保存距离
```

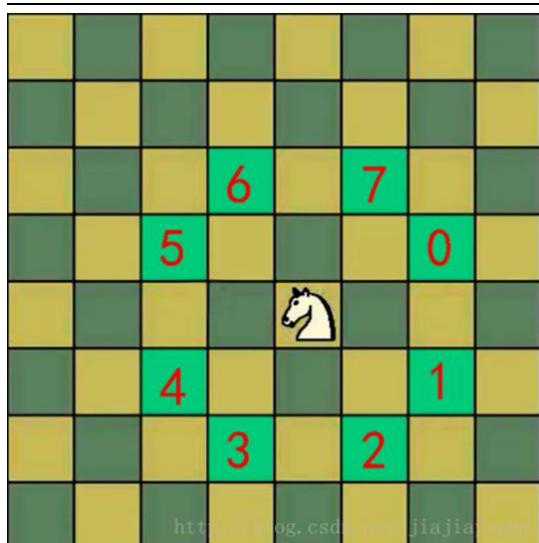


```
//对中间顶点遍历， k 就是中间顶点的下标 [A, B, C, D, E, F, G]
for(int k = 0; k < dis.length; k++) { //
    //从 i 顶点开始出发 [A, B, C, D, E, F, G]
    for(int i = 0; i < dis.length; i++) {
        //到达 j 顶点 // [A, B, C, D, E, F, G]
        for(int j = 0; j < dis.length; j++) {
            len = dis[i][k] + dis[k][j];//=> 求出从 i 顶点出发， 经过 k 中间顶点， 到达 j 顶点距离
            if(len < dis[i][j]) {//如果 len 小于 dis[i][j]
                dis[i][j] = len;//更新距离
                pre[i][j] = pre[k][j];//更新前驱顶点
            }
        }
    }
}
```

14.10 马踏棋盘算法

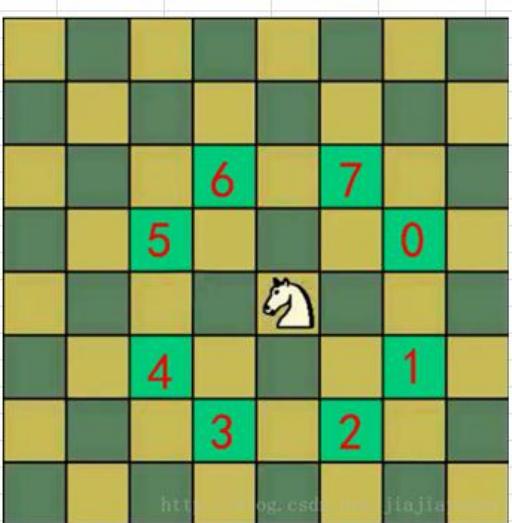
14.10.1 马踏棋盘算法介绍和游戏演示

- 1) 马踏棋盘算法也被称为骑士周游问题
- 2) 将马随机放在国际象棋的 8×8 棋盘 $\text{Board}[0 \sim 7][0 \sim 7]$ 的某个方格中，马按走棋规则(马走日字)进行移动。要求每个方格只进入一次，走遍棋盘上全部 64 个方格
- 3) 游戏演示: http://www.4399.com/flash/146267_2.htm



14.10.2 马踏棋盘游戏代码实现

- 1) 马踏棋盘问题(骑士周游问题)实际上是图的深度优先搜索(DFS)的应用。
 - 2) 如果使用回溯（就是深度优先搜索）来解决，假如马儿踏了 53 个点，如图：走到了第 53 个，坐标 (1,0)，发现已经走到尽头，没办法，那就只能回退了，查看其他的路径，就在棋盘上不停的回溯……，思路分析+代码实现
- 对第一种实现方式的思路图解



骑士周游问题的解决步骤和思路

1. 创建棋盘 chessBoard, 是一个二维数组
2. 将当前位置设置为已经访问, 然后根据当前位置, 计算马儿还能走哪些位置, 并放入到一个集合中(ArrayList), 最多有8个位置, 每走一步, 就使用step+1
3. 遍历ArrayList中存放的所有位置, 看看哪个可以走通, 如果走通, 就继续, 走不通, 就回溯.
4. 判断马儿是否完成了任务, 使用 step 和应该走的步数比较, 如果没有达到数量, 则表示没有完成任务, 将整个棋盘置0

注意: 马儿不同的走法(策略), 会得到不同的结果, 效率也会有影响(优化)

```
//创建一个Point
Point p1 = new Point();
if((p1.x=curPoint.x - 2) >= 0 && (p1.y = curPoint.y -1) >= 0) {
    ps.add(new Point(p1));
}
```

3) 分析第一种方式的问题, 并使用贪心算法(greedy algorithm)进行优化。解决马踏棋盘问题。

使用贪心算法对原来的算法优化

1. 我们获取当前位置, 可以走的下一个位置的集合

```
//获取当前位置可以走的下一个位置的集合
ArrayList<Point> ps = next(new Point(column, row));
```

2. 我们需要对ps 中所有的point 的下一步的所有集合的数目, 进行非递减排序, 就ok,

```
9, 7, 6, 5, 3, 2, 1 //递减排序
1, 2, 3, 4, 5, 6, 10 //递增排序
```

```
1, 2, 2, 2, 3, 3, 4, 5, 6 //非递减
9, 7, 6, 6, 5, 5, 3, 2, 1 //非递增
```

4) 使用前面的游戏来验证算法是否正确。

5) 代码实现

```
package com.atguigu.horse;

import java.awt.Point;
import java.util.ArrayList;
import java.util.Comparator;
```



```
public class HorseChessboard {  
  
    private static int X; // 棋盘的列数  
    private static int Y; // 棋盘的行数  
    //创建一个数组，标记棋盘的各个位置是否被访问过  
    private static boolean visited[];  
    //使用一个属性，标记是否棋盘的所有位置都被访问  
    private static boolean finished; // 如果为 true,表示成功  
  
    public static void main(String[] args) {  
        System.out.println("骑士周游算法，开始运行~~");  
        //测试骑士周游算法是否正确  
        X = 8;  
        Y = 8;  
        int row = 1; //马儿初始位置的行，从 1 开始编号  
        int column = 1; //马儿初始位置的列，从 1 开始编号  
        //创建棋盘  
        int[][] chessboard = new int[X][Y];  
        visited = new boolean[X * Y];//初始值都是 false  
        //测试一下耗时  
        long start = System.currentTimeMillis();  
        traversalChessboard(chessboard, row - 1, column - 1, 1);  
        long end = System.currentTimeMillis();  
        System.out.println("共耗时：" + (end - start) + " 毫秒");  
    }  
}
```



```
//输出棋盘的最后情况
for(int[] rows : chessboard) {
    for(int step: rows) {
        System.out.print(step + "\t");
    }
    System.out.println();
}

/**
 * 完成骑士周游问题的算法
 * @param chessboard 棋盘
 * @param row 马儿当前的位置的行 从 0 开始
 * @param column 马儿当前的位置的列 从 0 开始
 * @param step 是第几步 ,初始位置就是第 1 步
 */
public static void traversalChessboard(int[][] chessboard, int row, int column, int step) {
    chessboard[row][column] = step;
    //row = 4 X = 8 column = 4 = 4 * 8 + 4 = 36
    visited[row * X + column] = true; //标记该位置已经访问
    //获取当前位置可以走的下一个位置的集合
    ArrayList<Point> ps = next(new Point(column, row));
    //对 ps 进行排序,排序的规则就是对 ps 的所有的 Point 对象的下一步的位置的数目, 进行非递减排序
    sort(ps);
    //遍历 ps
    while(!ps.isEmpty()) {
```



```
Point p = ps.remove(0); //取出下一个可以走的位置  
//判断该点是否已经访问过  
if(!visited[p.y * X + p.x]) { //说明还没有访问过  
    traversalChessboard(chessboard, p.y, p.x, step + 1);  
}  
}  
  
//判断马儿是否完成了任务，使用 step 和应该走的步数比较，  
//如果没有达到数量，则表示没有完成任务，将整个棋盘置 0  
//说明：step < X * Y 成立的情况有两种  
//1. 棋盘到当前位置，仍然没有走完  
//2. 棋盘处于一个回溯过程  
if(step < X * Y && !finished) {  
    chessboard[row][column] = 0;  
    visited[row * X + column] = false;  
} else {  
    finished = true;  
}  
  
}  
  
/**  
 * 功能：根据当前位置(Point 对象)，计算马儿还能走哪些位置(Point)，并放入到一个集合中(ArrayList)，最多  
有 8 个位置  
 * @param curPoint  
 * @return  
 */
```



```
public static ArrayList<Point> next(Point curPoint) {  
    //创建一个 ArrayList  
    ArrayList<Point> ps = new ArrayList<Point>();  
    //创建一个 Point  
    Point p1 = new Point();  
    //表示马儿可以走 5 这个位置  
    if((p1.x = curPoint.x - 2) >= 0 && (p1.y = curPoint.y -1) >= 0) {  
        ps.add(new Point(p1));  
    }  
    //判断马儿可以走 6 这个位置  
    if((p1.x = curPoint.x - 1) >=0 && (p1.y=curPoint.y-2)>=0) {  
        ps.add(new Point(p1));  
    }  
    //判断马儿可以走 7 这个位置  
    if ((p1.x = curPoint.x + 1) < X && (p1.y = curPoint.y - 2) >= 0) {  
        ps.add(new Point(p1));  
    }  
    //判断马儿可以走 0 这个位置  
    if ((p1.x = curPoint.x + 2) < X && (p1.y = curPoint.y - 1) >= 0) {  
        ps.add(new Point(p1));  
    }  
    //判断马儿可以走 1 这个位置  
    if ((p1.x = curPoint.x + 2) < X && (p1.y = curPoint.y + 1) < Y) {  
        ps.add(new Point(p1));  
    }  
    //判断马儿可以走 2 这个位置
```



```
if ((p1.x = curPoint.x + 1) < X && (p1.y = curPoint.y + 2) < Y) {  
    ps.add(new Point(p1));  
}  
  
//判断马儿可以走 3 这个位置  
if ((p1.x = curPoint.x - 1) >= 0 && (p1.y = curPoint.y + 2) < Y) {  
    ps.add(new Point(p1));  
}  
  
//判断马儿可以走 4 这个位置  
if ((p1.x = curPoint.x - 2) >= 0 && (p1.y = curPoint.y + 1) < Y) {  
    ps.add(new Point(p1));  
}  
  
return ps;  
}  
  
//根据当前这个一步的所有的下一步的选择位置，进行非递减排序，减少回溯的次数  
public static void sort(ArrayList<Point> ps) {  
    ps.sort(new Comparator<Point>() {  
  
        @Override  
        public int compare(Point o1, Point o2) {  
            // TODO Auto-generated method stub  
            //获取到 o1 的下一步的所有位置个数  
            int count1 = next(o1).size();  
            //获取到 o2 的下一步的所有位置个数  
            int count2 = next(o2).size();  
            if(count1 < count2) {  
                return 1;  
            } else if(count1 > count2) {  
                return -1;  
            } else {  
                return 0;  
            }  
        }  
    });  
}
```



```
        return -1;  
    } else if (count1 == count2) {  
        return 0;  
    } else {  
        return 1;  
    }  
}  
});  
}
```

