# 数据结构

数据结构包括**线性结构**和**非线性结构**；

## 线性结构

1. 线性结构是最常见的数据结构，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，顺序存储结构和链式存储结构：

顺序存储的线性表成为顺序表，顺序表的存储元素是连续的，如数组：依次往下保存；

链式存储的线性表成为链表，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；

1. 线性结构常见的有数组、队列、链表和栈；

## 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

# 稀疏数组和队列

## 稀疏数组

当一个数组中大部分元素都为0，或者为同一值的数组时，可以使用稀疏数组来保存该数据；

1. 稀疏数组的处理方法：
2. 记录数组一共有几行几列，有多少个不同的值
3. 把具有不同值的元素的队列及值记录在一个规模数组中，从而缩小程序的规模；

第一列数据表示：6行7列8个值

PS：数组下标是从0开始的；

EG：需求保存五子棋中下棋的位置

1. 二维数组转为稀疏数组的思路：
2. 遍历原始的二维数组，得到有效数据的个数sum；
3. 根据sum创建稀疏数组sparseArray int[sum+1][3];
4. 将二维数组的有效数据存入稀疏数组；
5. 稀疏数组转为二维数组：
6. 先读取稀疏数组第一行，根据第一行创建原始的二位数组：

**int[][]** origin = **new int[**sparse**[**0**][**0**]][**sparse**[**0**][**1**]]**;

1. 读取稀疏数组最后几行的数据，并赋予原始的二位数组；

## 队列

1. 队列的介绍
2. 队列是一个有序列表，可以用数组或者链表来实现；
3. 遵循先入先出的原则；
4. 数组模拟队列：



maxSize为该队列的最大容量；front和rear作为两个指针，记录队列的前后端指标，初始位置是-1；

存入数据后，rear值为存放值的最大下标，front为已取出值下标+1；

存数据rear加，front不变；取数据rear不变，front加；

思路：

1. addQueue 添加队列
2. 当rear<maxSize-1时数据存入rear为下标的元素中，然后rear上移++；
3. 当rear=maxSize-1，数组存满；
4. fetchQueue 获取队列
5. 先将front指针上移+1，取出对应下标的值；
6. 如果front=rear，则队列取完；
7. 数组模拟环形队列

思路：

1. front：初始值0，指向队列的第一个元素的当前位置;
2. rear： 初始值0，指向队列后一个元素的后面一个位置，为了空出来一个空间做约定，发现要不没办法判断队列是否满，因为取模实现循环，判断只能使用==。所以导致实际队列存放的数据是数组maxSize-1;
3. 队列为空的条件：front==rear；
4. 队列为满的条件：(rear+1)%maxSize==front；
5. 队列中的有效数据的个数：(rear+ maxSize - front)%maxSize

# 链表(Linked List)

链表是有序的列表，内存中存储模型如下:

小结：

1. 链表是以节点存储的；
2. 每个节点包含data域和next域：指向下一个节点；
3. 链表的各个节点不是连续存储的，即next域记录内存地址不是连续的；
4. 链表分为带头节点的链表和无头节点的链表；
5. 头指针记录第一个数据的地址；
6. 第一个数据的地址在data域获取数据，在next域获取下一个数据的地址；依次往后，直到next域内容为null；

## 单链表

单链表：带有头结点，逻辑结构如下：



最后一个的next域为null，这只是逻辑结构，实际在内存中a1，a2...是不连续的；

1. 依次添加Node
2. 先创建一个head头节点，作用是表示单链表的头。初始情况下头节点的next域为null；
3. 后面每添加一个节点，就直接加入到链表的最后

此时需要遍历找到最后一个节点，头节点都是不能动的，需要引入一个临时节点tempNode来承接数据；

1. 然后tempNode一直赋予next，直到next == null;

CODE

节点的数据结构：

**class** HeroNode**{  
 public int no**;  
 **public** String **name**;  
 //自己当时理解的是存的地址的具体信息  
 //其实存的就是对象 他本身就是一个地址  
 **public** HeroNode **next**;  
 **public** HeroNode**(int** no, String name**) {  
 this**.**no** = no;  
 **this**.**name** = name;  
 **}  
}**

添加节点：

**private** HeroNode **headNode** = **new** HeroNode**(**0,**"")**; //头节点  
**public void** addNode**(**HeroNode newNode**){** HeroNode temp = **headNode**;  
 **while (true) {** //标志找到了当前队列的最后一个元素  
 **if (**temp.**next** == **null) {** temp.**next** = newNode;  
 **break**;  
 **}** //否则就一直往后走  
 //以为将当前元素的内存地址赋予了temp，那么更改temp就是更改对应的LinkedList的最后一个元素  
 temp = temp.**next**;   
 **}  
}**

1. 按顺序添加Node



1. 首先定位位置，仍然是需要借助临时节点：需要找到新节点的前一个节点，这样将新节点插入该节点后面即可；
2. 定位条件：
3. temp.next == null 即新节点是位于最后一个节点的位置；
4. temp.**next**.**no** > newNode.**no** temp刚好是新节点的前一个节点
5. newNode.next = temp.next;
6. temp.next = newNode;
7. 单链表面试题：
8. 求链表中有效节点的个数，参数为头节点

遍历即可，头节点要去掉

1. 查找倒数第K个节点

获取链表总长度length，然后遍历得到第length-k+1个节点；

1. 单链表翻转

思路：

1. 先定义一个headNode来承接翻转后获取的链表；
2. 从头到尾遍历原链表，每遍历一个，就放在新链表的headNode后面；

PS：此处需要单独保存原节点的next，如不然，因为需要将新链表headNode.next赋给该找到的节点的next域，这样这导致该节点.next.next的丢失

代码：

HeroNode current = headNode.**next**;  
HeroNode curNext;  
HeroNode newHeadNode = **new** HeroNode**(**0,**"")**;  
//自己之前没有保存current.next 导致current被重新赋值后其next的丢失  
//注意此处涉及到内存地址的相互指向  
//所以要注意赋值的形式  
**while (**current != **null){** curNext = current.**next**;  
 current.**next** = newHeadNode.**next**;  
 newHeadNode.**next** = current;  
 current = curNext;  
**}**headNode.**next** = newHeadNode.**next**;

1. 从尾到头打印链表

根据栈这个数据结构的特点：先进后出

遍历放入栈中，然后遍历栈：

Stack**<**HeroNode**>** stack = **new** Stack**<>()**;  
HeroNode temp = headNode.**next**;  
while **(**temp != **null){** stack.add**(**temp**)**;  
 temp = temp.**next**;  
**}  
while (**!stack.empty**()){** System.**out**.println**(**stack.pop**())**;  
**}**

1. 合并两个有序链表并整体保持有序

就是结合3的思路加上有序添加

## 双向链表

带有head域记录前面一个数据的位置

单向链表的缺点：

1. 单向链表的查找只能是一个方向，而双向链表可以向前、向后查找；
2. 单向链表不能自我删除，需要借助辅助节点；而双向链表可以自我删除；

双向链表的数据结构示意：



1. 双向链表的添加：
2. 通过辅助接点找到链表的最后一个节点；
3. temp.next = newNode
4. newNode.pre = temp;

**void** addNode**(**HeroNode newNode**){** HeroNode temp = **this**.**headNode**;  
 **while (**temp.**next** != **null) {** temp = temp.**next**;  
 **}** temp.**next** = newNode;  
 newNode.**pre** = temp;  
**}**

1. 双向链表的修改：

同单向链表

1. 双向链表的删除

双向链表可以自我删除

直接找到待删除节点temp：

temp.pre.next = temp.next;

temp.next.pre = temp.pre;

**void** deleteNode**(**HeroNode delNode**){** HeroNode temp = **this**.**headNode**.**next**;  
 **if (**temp == **null){** System.**out**.println**("该链表为空，无法删除")**;  
 **return**;  
 **}  
 while (true){** //相比单链的一个额外判断 要不就temp.next.pre = temp.pre报空指针  
 **if (**temp.**no** == delNode.**no** && temp.**next** == **null){** System.**out**.println**("待删除节点为最后一个节点")**;  
 temp.**pre**.**next** = temp.**next**;  
 **break**;  
 **}  
 if (**temp.**no** == delNode.**no){** temp.**pre**.**next** = temp.**next**;  
 temp.**next**.**pre** = temp.**pre**;  
 **break**;  
 **}  
 if (**temp.**next** == **null){** System.**out**.println**("未找到待删除的节点")**;  
 **break**;  
 **}** temp = temp.**next**;  
 **}  
}**

1. 双链表的按顺序添加节点：

思路同单节点，关键还在于节点的定位

**void** addNodeByOrder**(**HeroNode newNode**){** HeroNode current = **this**.**headNode**;  
 HeroNode currentNext;  
 HeroNode currentPre;  
 **while (true){  
 if (**current.**next** == **null){** System.**out**.println**("新插入节点为最后一位....")**;  
 newNode.**pre** = current;  
 current.**next** = newNode;  
 **break**;  
 **}  
 if (**current.**next**.**no** > newNode.**no){** currentNext = current.**next**;  
 currentPre = current.**pre**;  
   
 current.**next** = newNode;  
 newNode.**next** = currentNext;  
   
 newNode.**pre** = current;  
 currentPre.**pre** = newNode;  
 **break**;  
 **}** current = current.**next**;  
 **}  
}**

## 环形链表和约瑟夫问题

介绍：一个环形链表，有n个元素，约定编号为k(1<=k<=n)的人从1开始报数，数到m出列，他的下一位继续从1开始数，到m出列，依次直到所有人出列，由此产生一个出队编号的序列；

1. 构建单向环形链表的思路：
2. 先构件第一个节点，让first指向自己形成一个环形；
3. 后续每添加一个节点，就把该节点加入到已有的环形链表中；
4. 单向环形链表的遍历
5. 先让指针（辅助变量）currentNode指想first节点；
6. 然后一个个遍历直到currentNode.next == first;
7. 约瑟夫问题思路：
8. 需要创建一个辅助指针变量pre，事先指向环形量表的最后一个节点，记录要数的变量的前一个变量；

PS：报数前需要先移动k-1次至指定位置；

1. 报数时，first和pre同时迁移m-1次；
2. 此时first节点出圈：

**first** = **first**.**next**;  
helper.**next** = **first**;

原first就会没有引用而被删除

# 数据结构 --栈 stack

## 栈的介绍

1. 栈是一个先入后出(FILO -- First In Last Out)的有序列表；
2. 栈是限制限制线性表中的元素的插入和删除只能在线性表的同一端进行的特殊线性表。允许插入和删除的一段为变化的一端，成为栈顶(Top)；另一端为固定的一端，称为栈底(Bottom)；
3. 最先放入栈的元素放入栈底，最后放入栈的元素放入栈顶，而删除元素刚好相反，最后放入的先被删除，最先放入的后被删除；
4. 出栈成为pop，入栈为push；

## 栈的应用场景：

1. 子程序的调用：在调往子程序之前，会先将下个指令的地址存入栈堆中，直到子程序执行完后再将地址去除，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、区域变量扥数据存入栈堆中；
3. 表达式的转换[中缀转为后缀表达式]与求值；
4. 二叉树的遍历；
5. 图形的深度优先(depth-first)搜索法；

## 用数组实现栈的模拟

1. 定义一个top来表示栈顶，初始化为-1；
2. 入栈：有数据加入到栈，top++;stack[top]=data;
3. 出栈：value=stack[top];top--;

## 栈实现计算器：

1. 通过一个index来遍历需要计算的表达式；
2. 如果是一个数字则直接入数栈；
3. 如果是一个符号则分情况考虑：
4. 如果当前符号栈为空，则可以直接入栈；
5. 如果符号栈不为空，则进行比较：

如果当前的操作符优先级小于于或者等于栈中的操作符(\* /操作大于+-)，就需要去数栈中pop出两个数字；

如果当前的操作符优先级大于栈中的操作符，就直接放入符号栈；

1. 扫描件完毕，就顺序从数栈和符号栈pop相应数和操作符并运行；
2. 最后数栈中只有一个数字，就是计算结果；

# 前缀、中缀、后缀表达式（逆波兰拨打时）

## 前缀表达式(波兰表达式)

前缀表达式的运算符位于操作数之前；

EG：(3+4)\*5-6 对应的前缀表达式是 - \* + 3 4 5 6

从右至左扫描表达式，遇到数字压入栈堆，遇到运算符时，弹出栈顶的两个数，用运算符对它们做相应的计算(栈顶元素和次顶的元素)，并将结果入栈：重复上述过程指导表达式的最左端，最后计算的值即为表达式的结果；

## 中缀表达式

中缀表达式就是常见的运算表达式，如(3+4)\*5-6

中缀表达式对计算机不好操作，一般转为后缀表达式进行计算；

## 后缀表达式(逆波兰表达式)

运算符位于操作数之后

EG：(3+4)\*5-6 对应的后缀表达式是 3 4 + 5 \* 6 -

从左至右扫描表达式，遇到数字时，将数字压入栈，遇到运算符时，弹出栈顶的两个数，用运算符对它们做相应的运算（次顶元素和栈顶元素），并将结果入栈；重复上述操作直到最右端；

## 中缀表达式转换为后缀表达式

1. 初始化两个栈，运算符栈s1和存储中间计算结果的栈s2；
2. 从左至右扫描中缀表达式；
3. 遇到操作数时，将其压入s2；
4. 遇到运算符时，比较其与s1栈顶运算符的优先级(可以用一个enum来做个int的优先级映射)：
5. 若s1为空，或者栈顶运算符为左括号(,则直接将此运算符入栈；
6. 若优先级比栈顶运算符的高，也将运算符压入s1；(只有高才执行，同级不执行)
7. 否则将s1栈顶的运算符弹出并压入s2中，再次转到4-1步骤中与s1中新的栈顶运算符进行比较；
8. 遇到括号时：
9. 如果是左括号(,则直接压入s1；
10. 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
11. 重复步骤2到5，直到表达式的最右端；
12. 将s1中的运算符依次弹出并压入s2；
13. 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；

# 递归 recursion

主要理解一下递归调用的执行顺序

## 递归调用规则

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；
3. 如果方法中使用的是引用变量，就会共享改引用类型的数据；
4. 递归必须向退出递归的条件逼近，否则就会无限递归，出现StackOverflowError;
5. 当一个方法执行完毕，或遇到return就会返回，遵守谁调用结果归谁，同时方法执行完毕或返回时，该方法也就执行完毕；

## 递归的常见应用场景

1. 各种数学问题，eg：8皇后问题、汉诺塔、阶乘、迷宫问题、数独、球和篮子问题；
2. 各种算法：快排、归并排序、二分查找、分治算法等；
3. 将用栈解决问题递归的代码更简洁；
4. 迷宫寻址问题：

这部分查看代码

1. 八皇后问题

前后左右和相邻的对角线不能放置两个皇后，不能放在同一行，同一列，相同斜线上 ；

PS：这个解题思路和数独很像

1. 第一个皇后放在第一行第一列；
2. 第二个皇后放在第二行第一列，然后判断是否ok；若不行依次向后放置；
3. 继续第三个皇后放置在第三行第一列，同上；直到第八个；算是找到了第一个正确的解；
4. 当第一个正确之后，再栈后退至上一个栈，就会开始回溯(就是换个位置看能不能得出新解)，直到得到第一列开始的所有的解；
5. 然后将第一个换后放在第二列，依次执行1 2 3 4，直到获取所有的解；
6. 说明：理论上应创建二维数组表示棋盘，但实际可以一一维数组来解决：

arr \={0,4,7,5,2,6,1,3} //对应arr的下标表示第几行，及第几个皇后

arr[i]=val val表示第i+1个皇后，放在了第i+1行的val+1列；

{0,4,7,5,2,6,1,3} 表示第一个皇后放在第一行 0+1列，第二个：第二行4+1列...

# 时间复杂度和空间复杂度

排序的分类：

1. 内部排序：将需要处理的所有数据都加载到内部存储器中进行排序；

内部排序细分为:

1. 插入排序：直接插入排序；希尔排序；
2. 选择排序：简单选择排序；堆排序；
3. 交换排序：冒泡排序；快速排序‘
4. 归并排序；
5. 基数排序；’
6. 外部排序：数据量很大，无法全部加载到内存中，需要借助外部存储进行排序；

## 时间复杂度

度量一个算法执行时间

1. 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；

一个算法中语句执行此时称为语句频度或者时间频度。记为T(n);

时间频度忽略常数项、低次项、系数；

1. 时间复杂度的概念：
2. 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0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；
3. T(n)不同，但是时间复杂度可能相同，eg：T(n)=n2+7n+6与T(n)=3n2+2n+2,但时间复杂度是一样的，都记为O(n2);这里的f(n)= n2;

忽略常数项、低次项、系数；

1. 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运算；

修改后 的运行次数函数中，只保留最高阶项；

去除最高阶项的系数；

1. 常见的时间复杂度：

执行次数与条件n的关系

1. 常数阶O(1);

没有循环，eg:i+j；

1. 对数阶O(log2n);

比如：

**int** i = **1**;  
**while** (i<n){i = i \* **2**;}

假设执行m次之后推出，执行次数的方程：2m=n 🡪 m = log2n

所以时间复杂度记为O(log2n);

1. 线性阶O(n)；

单层的for循环，i<n；i++

for循环里面的代码会执行n次；

1. 线性对数阶O(nlog2n)；

时间复杂度为O(log2n)循环n遍，比如for循环里面放的是上述的while循环；

1. 平方阶O(n2)；

双层for循环；

1. 立方阶O(n3)；
2. K次方阶O(nk)；
3. 指数阶O(2n);

上述时间复杂度从小到大；时间复杂度应该越低越好，效率越高 ；

1. 平均时间复杂度和最坏时间复杂度：
2. 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。
3. 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。
4. 平均时间复杂度和最坏时间复杂度是否一致，和算法有关(如上图)

## 空间复杂度

1. 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。
2. 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就属于这种情况;
3. 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;

# 排序算法

## 冒泡排序：

通过对待排序序列从前向后(从下表较小的元素开始)，依次比较相邻元素的值，发现逆序则交换，是值较大的元素从前移植后部；

PS：因为排序的过程中，各元素不断接近自己的位置，若一趟比较下来没有进行排序，则说明有序，可以设置一个标志flag来判断是否进行了排序，从而减小不必要的比较；

**private static void** bubbleTeacher(){  
 booleanflag = **false**;  
 **for** (**int** i=**0**;i<*array*.length-**1**;i++){  
 **for** (**int** j = **0**; j < *array*.length-**1**-i; j++) {  
 **if** (*array*[j] < *array*[j+**1**]){  
 flag = **true**;  
 **int** temp = *array*[j];  
 *array*[j] = *array*[j+**1**];  
 *array*[j+**1**] = temp;  
 }  
 }  
 **if** (!flag){ **break**; } **else** { flag = **false**; }  
 }  
 }

## 选择排序 select sort

选择排序是从欲排序的数据中，按照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；

基本思路：

第一次从arr[0]到arr[n-1]中选取最小值，与arr[0]进行交换；

第二次从arr[1]到arr[n-1]中选取最小值，与arr[1]进行交换；

依次类推...

第n-1次从arr[n-2]到arr[n-1]中选取最小值，与arr[n-2]进行交换；

总共n-1次；

说明：

1. 一共有length-1轮排序；
2. 每一轮排序，又是一个循环：

先假定这个数是最小数；

然后和后面比较，如果有更小的则获取最小数并记录下标；

当遍历到数组最后时，得到本轮的最小数和下标，交换arr[i]和arr[index]；

**private void** selectSort(){  
 **for** (**int** i=**0**;i<*array*.length;i++){  
 **int** min = *array*[i];  
 **int** index = i;  
 for(**int** j = i+**1**; j < *array*.length; j++) {  
 **if** (min > *array*[j]){  
 min = *array*[j];  
 index = j;  
 }  
 }  
 **if**(index != i){  
 **int** temp = *array*[i];  
 *array*[i] = min;  
 *array*[index] = temp;  
 }  
 }  
}

## 插入排序 insertion sort

基本思想：

把n个待排序的元素看成是一个有序表和无序表，开始时有序表只有一个元素，无序表中有n你-1个元素，排序过程实每次从无序表中取出一个元素，把他的排序码依次与有序表中的排序码进行比较，将他插入合适的位置，使之成为新的有序表；

insertVal = arr[i];  
insertIndex = i - **1**;

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

//比较完成

**if**(insertIndex + **1** != i) {  
 arr[insertIndex + **1**] = insertVal;  
}

进入当前循环的说明满足两个条件：

insertIndex >= **0** && insertVal < arr[insertIndex]，这里可以理解为：

先记录下待比较的数的值，和他前边数(已经有序)的index，比较时若满足条件，就把后者赋给待比较数这个位置，依次满足条件后移，直到index=0或者当前数已经比待比较数小，跳出循环，insertIndex--用于记录数组索引前移了几次；

**private static void** myInsert(){  
 //arr[0]默认可以看做是有序列表  
 **for** (**int** i = **1**; i < *array*.length; i++) {  
 **int** compareValue = *array*[i];  
 //因为当前数与他前面的数进行比较并用于记录当前比较的索引  
 **int** compareIndex = i-**1**;  
 **while** (compareIndex >=**0** && compareValue < *array*[compareIndex]){  
 //把后面的数依次前移 最前面的数已经被记录了所以不用交换  
 *array*[compareIndex + **1**] = *array*[compareIndex];  
 compareIndex--;  
 System.*out*.println(Arrays.*toString*(*array*));  
 }  
 **if** (i != compareIndex + **1**){  
 *array*[compareIndex + **1**] = compareValue;  
 }  
 }  
}

## 希尔(shell)排序

插入排序存在的问题：当最后一个数很小时，必须后移很多次，影响效率；

shell排序也是一种插入排序，是对插入排序的一种优化，也称为缩小增量排序；

基本思想：

希尔排序是把记录按照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少，当增量减至一时，整个文件恰好分为一组，排序终止；

PS：仔细思考第二组for的作用

1. 交换法：

第一个for的作用是分组；

里面的代码执行的就是冒泡排序；

所以交换法效率较低



**private static void** shell(){  
 **for** (**int** gap = *array*.length / **2**; gap > **0**; gap = gap / **2**) {  
 //在分组里面进行了冒泡排序  
 **for** (**int** i = gap; i < *array*.length; i++) {  
 **for** (**int** j = i-gap; j >= **0**; j-=gap) {  
 **if** (*array*[j] > *array*[j+gap]){  
 **int** temp = *array*[j];  
 *array*[j] = *array*[j+gap];  
 *array*[j+gap] = temp;  
 }  
 }  
 }  
 }  
}

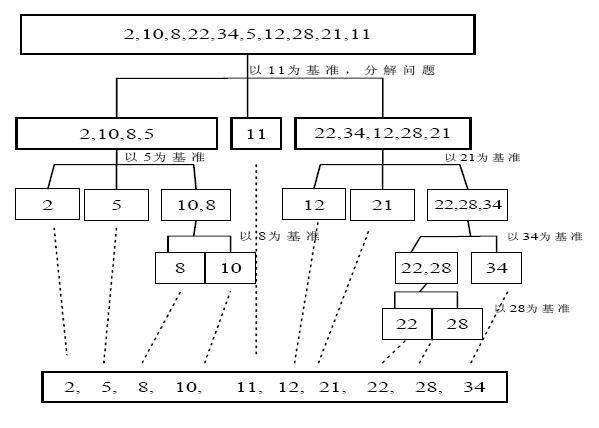
1. 移动法：

就是对所在的分组进行插入排序

**private static void** shellInsert(){  
 **for** (**int** gap=*array*.length/**2**; gap> **0**; gap = gap / **2**) {  
 **for** (**int** i = gap; i < *array*.length; i++) {  
 **int** j = i;  
 **int** temp = *array*[j];  
 **while** (j-gap>=**0** && temp < *array*[j-gap]){  
 //当前arr[j-gap]移动导[j]  
 *array*[j] = *array*[j-gap];  
 j -= gap;  
 }  
 *array*[j] = temp;  
 }  
 }  
   
}

## 快速排序：

快速排序是对冒泡排序的一种改进。基本思想：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的数据进行快速排序，整个排序过程可以递归进行，一次达到整个数据变成有序序列；

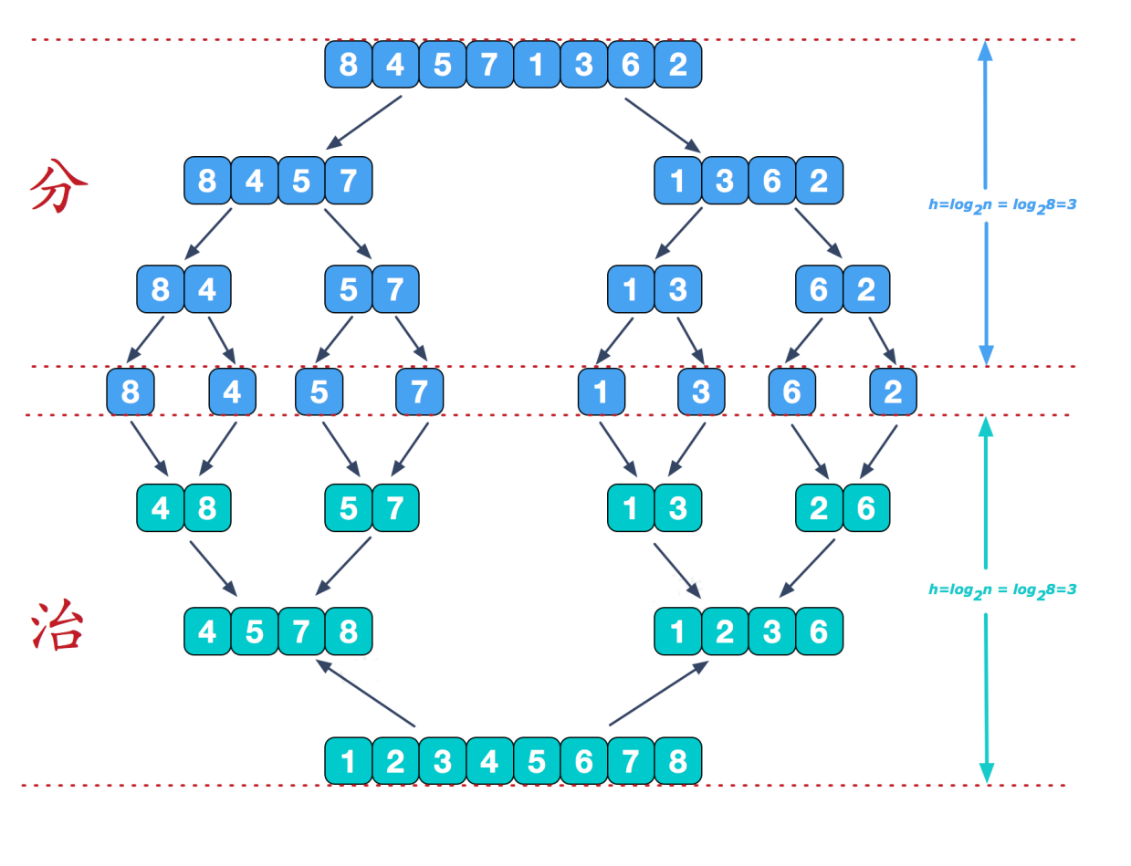


**private static void** quick(**int**[] array,**int** left,**int** right){  
 **int** l = left;  
 **int** r = right;  
 //中轴值 两侧与此值进行比较  
 **int** pivot = array[(left+right)/**2**];  
 //while循环的目的是将比pivot小的放在其左边，大的放右边  
 **while** (l<r){  
 //在pivot左边找>=pivot 并不一定是index也比其小  
 **while** (array[l] < pivot){ l+=**1**;}  
 //在pivot左边找<=pivot 跳出循环  
 **while** (array[r] > pivot){ r-=**1**; }  
 //l>=r说明pivot两侧的值已经排好了  
 **if** (l>=r) **break**;  
 //交换两者的值  
 **int** temp = array[l];  
 array[l] = array[r];  
 array[r] = temp;  
 //交换玩之后发现array[l] == pivot r-- 前移  
 **if** (array[l] == pivot) r-=**1**;  
 //交换玩之后发现array[r] == pivot l++ 后移  
 **if** (array[l] == pivot) l+=**1** ;  
 }  
  
 **if** (l==r){  
 l+=**1**;r-=**1**;  
 }  
 //向左递归  
 **if** (left<r){  
 *quick*(array,left,r);  
 }  
  
 //向右递归  
 **if** (right > l){  
 *quick*(array,l,right);  
 }  
   
}

## 归并排序 merge sort

归并排序是利用归并的思想实现的排序方法，采用了分治策略(divide-and-conquer)策略；

分治法是问题分divide成一些小问题，然后递归求解，而治conquer的阶段则是将分的阶段得到的各答案修补在一起，即分而治之；



**private static void** mergeSort(**int**[] array,**int** left,**int** right,**int**[] temp){  
 **if** (left < right){  
 **int** middle = (left+right)/**2**;  
 //向左递归拆解  
 *mergeSort*(array,left,middle,temp);  
 //向右递归拆解  
 *mergeSort*(array,middle+**1**,right,temp);  
 //合并  
 *merge*(array,left,middle,right,temp);  
 }  
}  
/\*\*  
 \* 先写归并的方法 先写治  
 \* @param *array* 待排序数组  
 \* @param *left* 左边有序数列的初始索引  
 \* @param *middle* 中间位置  
 \* @param *right* 右边有序数列的最右侧索引  
 \* @param *temp* 临时数组 搬到这里  
 \*/  
**private static void** merge(**int**[] array,**int** left,**int** middle,**int** right,**int**[] temp){  
 **int** i =left;  
 **int** j = middle + **1**; //右边的起始索引  
 **int** t = **0**;//temp的索引记录  
 /\* 第一步  
 \* 先把左右两边(已经有序)的数组按照规则填充到temp数组  
 \* 直到左右两边其中一边移动完成 (则另一边剩余的也是有序排列的)  
 \*/  
 **while** (i <= middle && j <= right){  
 **if** (array[i] <= array[j]){  
 //只用搬就行  
 temp[t] = array[i];  
 t++;  
 i++;  
 } **else** {  
 temp[t] = array[j];  
 t++;  
 j++;  
 }  
 }  
 /\* 第二步  
 \* 左边有剩余  
 \*/  
 **while** (i<=middle){  
 temp[t] = array[i];  
 t++;  
 i++;  
 }  
 /\* 第二步  
 \* 或者右边有剩余  
 \*/  
 **while** (j<=right){  
 temp[t] = array[j];  
 t++;  
 j++;  
 }  
 /\* 第三步  
 \* 将temp copy到array  
 \* 注意 此处并不是全部都需要copy  
 \*/  
 t=**0**;  
 **int** tempLeft = left;  
 **while**(tempLeft <= right) {  
 array[tempLeft] = temp[t];  
 t++;  
 tempLeft++;  
 }  
}

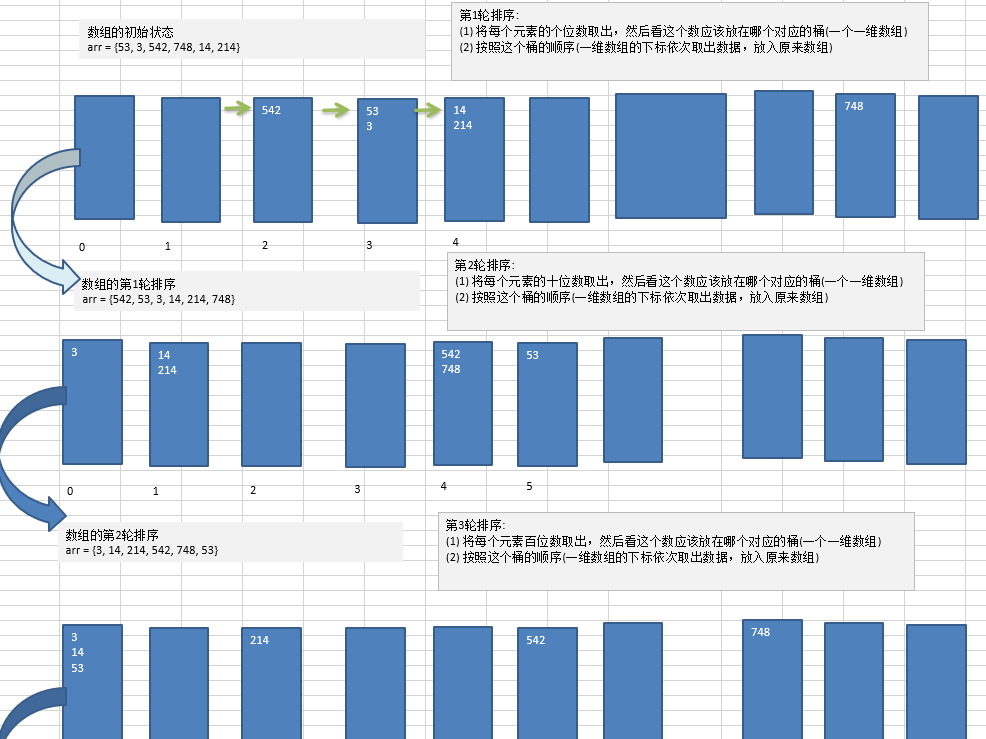
## 基数排序 raidx sort

1. 基数排序属于分配式排序(distribution sort)，又称桶子法，他是通过键值的各个位的值，将要排序的元素分配只某些桶中，达到排序的目的；
2. 基数排序是属于稳定性的排序，基数排序法是效率高的稳定性排序法；
3. 实现原理：将整数按位数切割成不同的数字，然后按照每个位数分别进行比较；

基本思想：

将所有带比较的数值统一为同样的数位长度，数位较短的前面补0，然后从低位开始，依次进行依次排序。

这样从最底位排序一直到最高位排序完成以后变成了一个有序序列；



基数排序的一些问题：

1. 基数排序是对桶排序的扩展，速度很快；
2. 基数排序是以空间换时间，占用内存很大(新开了很多数组)，数据量超大时可能OutOfMemoryError;
3. 基数排序是稳定的；

排序稳定性介绍：

假定待排序的记录序列中，存在多个而具有相同关键字的记录，若经过排序，这些记录的相对次序保持不变，即在原次序中，r[i]=r[j],且i在j之前，而在排序后的序列中，r[i]仍在r[j]之前；则称这种算法是稳定的，反之成为不稳定的；

**private static void** radixSort(**int**[] array){  
 //设置10个桶  
 **int**[][] bucket = **new int**[**10**][array.length];  
 //记录各个桶放置几个有效数字  
 **int**[] bucketSize = **new int**[**10**];  
 **int** max = **0**;  
 **for** (**int** a:array){  
 **if** (max < a) max = a;  
 }  
 **int** maxSize = (max + **""**).length();  
 **for** (**int** i=**0**,n=**1**;i<maxSize;i++,n=n\***10**){  
 **for** (**int** a:array){  
 //这一步存数是重点  
 **int** digital = a / n % **10**;  
 bucket[digital][bucketSize[digital]] = a;  
 bucketSize[digital]++;  
 }  
 //将桶中的元素一次取出  
 **int** index = **0**;  
 **for** (**int** k = **0**; k < **10**; k++) {  
 **int** size = bucketSize[k];  
 **if** (size > **0**){  
 **for** (**int** j = **0**; j < size; j++) {  
 array[index++] = bucket[k][j];  
 }  
 }  
 //这一步必须清除为0 防止后面被重复取  
 //而bucket不需要清除 应为只要bucketSize[i]==0就不用去对应的桶中取数据  
 bucketSize[k] = **0**;  
 }  
 }  
}

# 排序算法的相关术语

1. 稳定：如果a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在b前面，而a=b，排序之后a不一定在b的前面；
3. 内排序：所有排序操作都在内存中；
4. 外排序：由于数据量太大，需要将数据放在磁盘中，通过磁盘和内存实现排序；
5. 时间复杂度：一个算法所需要消耗的时间；
6. 空间复杂度：一个算法所需要消耗的内存；
7. In-space:不占用额外的内存；
8. Out-space：占用额外的内存；



# 查找算法

常用的查找方法有四种：

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；

## 线性查找

就是遍历然后查找到对应的值和下标；

没找到返回-1；

## 二分查找

二分查找只能是对有序数组的查找；

思路分析：

1. 首先确定数组中间的下标mid=(left+right)/2；
2. 然后对比待查找的值findVal与array[mid]的大小比较：

2.1. findVal > array[mid]，说明要往右边找；

2.2. findVal < array[mid]，说明要往左边找；

2.3. findVal = array[mid]，查找；

1. 递归的结束条件：
   1. 找到了对应的值；
   2. 未找到：则left > right退出；

但是这种算法存在如果有多个满足条件的值中的一个

**int** binary(**int**[] array, **int** left, **int** right, **int** find) {  
 **int** middle = (left + right) / **2**;  
 **if** (left > right) {  
 **return** -**1**;  
 }  
 //对left和right值要做相应的移动  
 //因为middle已经参与过比较，所以不需要重复，对其+-1即可  
 **if** (array[middle] == find) {  
 **return** middle;  
 } **else if** (array[middle] > find) {  
 **return** *binary*(array, left, middle-**1**, find);  
 } **else** {  
 **return** *binary*(array, middle+**1**, right, find);  
 }  
   
}

找到所有满足条件的值

/\*\*  
 \* 找到所有满足条件的值  
 \* 找到后想左/右分别进行扫描，满足则加入下标  
 \*/  
List<Integer> fiandAll(**int**[] array,**int** left,**int** right,**int** find){  
 List<Integer> indeies = **new** ArrayList<>();  
 **int** middle = (left + right) / **2**;  
 **if** (left > right) {  
 **return** indeies;  
 }  
 //对left和right值要做相应的移动  
 //因为middle已经参与过比较，所以不需要重复，对其+-1即可  
 **if** (array[middle] == find) {  
 indeies.add(middle);  
 **int** index = **1**;  
 **while** (array[middle - index] == find && middle-index>=**0**) {  
 indeies.add(middle - (index++));  
 }  
 index = **1**;  
 **while** (array[middle+index]==find&&middle+index<array.length){  
 indeies.add(middle + (index++));  
 }  
 **return** indeies;  
 } **else if** (array[middle] > find) {  
 **return** *fiandAll*(array, left, middle-**1**, find);  
 } **else** {  
 **return** *fiandAll*(array, middle+**1**, right, find);  
 }  
}

## 插值查找

1. 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;



等差数列 ... 根据差值与index差值的比例来确定每次查找的区间；

代码同二分查找，只是确定的middle不一样；

退出条件：

**if** (left > right || find < array[**0**] || find > array[array.length-**1**]) {  
 **return** indeies;  
}

注意事项：

1. 对于数据量大、关键字分布比较均匀的查找，采用差值查找较快；
2. 对于分布不均匀，不一定比二分查找更快；

## 斐波那契(黄金分割法)查找算法

同二分查找，只是更改了middle的位置，是其位于黄金分割的位置：

middle=low+F(k-1)-1; F(k-1) - 表示斐波那契数列

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。

**private static int** fiboSearch(**int**[] array,**int** key){  
 **int** low = **0**;**int** high = array.length - **1**;  
 **int** k = **0**;**int** middle;  
 **int**[] fibo = *fibo*();  
   
 // 获取满足条件的斐波那契数列的下标  
 // high <= fibo[k] -1 才满足条件找到下标  
 **while** (high > fibo[k] -**1**){  
 k++;  
 }  
   
 //因为fibo[k]的值可能大于array的长度，需要补长原有的array  
 //不足的部分用0填充  
 **int**[] temp = Arrays.*copyOf*(array,fibo[k]);  
 **for** (**int** i = high+**1**;i<temp.length;i++){  
 temp[i] = array[high];//用最大值来填充数组  
 }  
   
 //开始查找  
 **while** (low <= high){  
 middle = low + fibo[k-**1**] - **1**;  
 **if** (key < temp[middle]){  
 high = middle - **1**;  
 //全部元素 = 前面的元素 + 后边元素  
 //f[k] = f[k-1] + f[k-2]

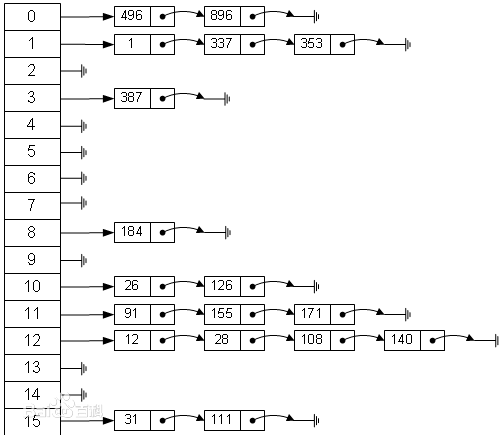
//因为 前面有 f[k-1]个元素,所以可以继续拆分 f[k-1] = f[k-2] + f[k-3]  
 //即 在 f[k-1] 的前面继续查找 k--  
 //即下次循环 mid = f[k-1-1]-1  
 k--;  
 } **else if** (key > temp[middle]) {  
 low = middle + **1**;  
 //全部元素 = 前面的元素 + 后边元素  
 //f[k] = f[k-1] + f[k-2]  
 //因为后面我们有f[k-2] 所以可以继续拆分 f[k-1] = f[k-3] + f[k-4]  
 //即在f[k-2] 的前面进行查找 k -=2  
 //即下次循环 mid = f[k - 1 - 2] - 1  
 k-=**2**;  
 } **else** {  
 //需要确定，返回的是哪个下标  
 **if**(middle <= high) {  
 **return** middle;  
 } **else** {  
 **return** high;  
 }  
 }  
 }  
 **return** -**1**;  
}

# 哈希表和内存布局

需求举例：新员工入职，加入信息（id，姓名，。。。等），输入id获取全部信息；不使用数据库；节省内存，越快越好 🡪 HASH

## HASH表的介绍

散列表(hash table)是根据关键码值(key value)而直接访问的数据结构；

也就是说它是通过把关键码值映射到表中的一个位置来访问记录，以加快查找的速度。这个映射函数成为散列函数；存放记录的**数组**叫做散列表，或哈希表；

数组 + 链表

数组里面放的是链表

查找对应链表时可以根据它的hash / length-1(模就是0-length-1 共length个)来快速获取其所在位置；

EG：使用hash管理雇员信息

三个类：  
 \* 散列函数  
**public class** HashTable {  
 **private static final int** *SIZE* = **7**;  
 //还要对数组初始化 要不空指针  
 **private** EmployerLinkList[] linkList;  
 **public** HashTable() {  
 linkList = **new** EmployerLinkList[*SIZE*];  
 **for** (**int** i = **0**; i < *SIZE*; i++) {  
 linkList[i] = **new** EmployerLinkList();  
 }  
 }  
 //根据id取模来放置  
 **public void** add(Employee emp) {  
 **int** index = emp.id % *SIZE*;  
 Employee byId = findById(emp.id);  
 **if** (byId != **null**) {  
 System.*out*.println(**"当前员工号 "** + emp.id + **"已存在"**);  
 **return**;  
 }  
 EmployerLinkList employerLinkList = linkList[index];  
 employerLinkList.add(emp);  
 }  
}

**public class** EmployerLinkList {  
 //此处头指针是有效，是该链表的第一个Employee  
 **private** Employee head;  
 //假定传进来的id是自增长的  
 **public void** add(Employee emp) {  
 **if** (head == **null**) {  
 //是第一个  
 head = emp;  
 **return**;  
 }  
 Employee temp = head;  
 **while** (temp.next != **null**) {  
 temp = temp.next;  
 }  
 temp.next = emp;  
 }  
}

**class** Employee{  
 **int** id;  
 String name;  
 Employee next;  
}

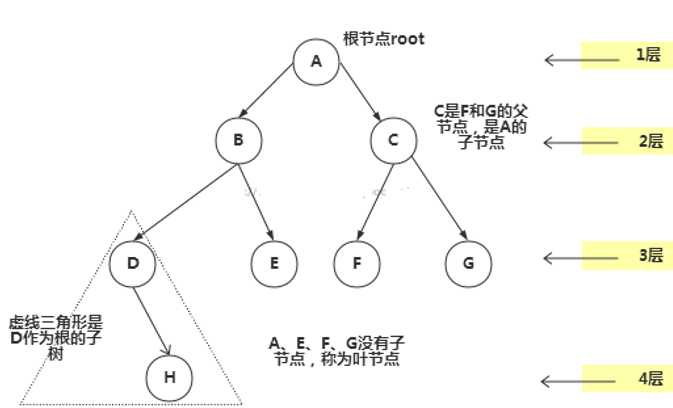
# 二叉树

## 为什么要用树结构

1. 数组存储方式  
   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;
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)
3. 树存储方式的分析:

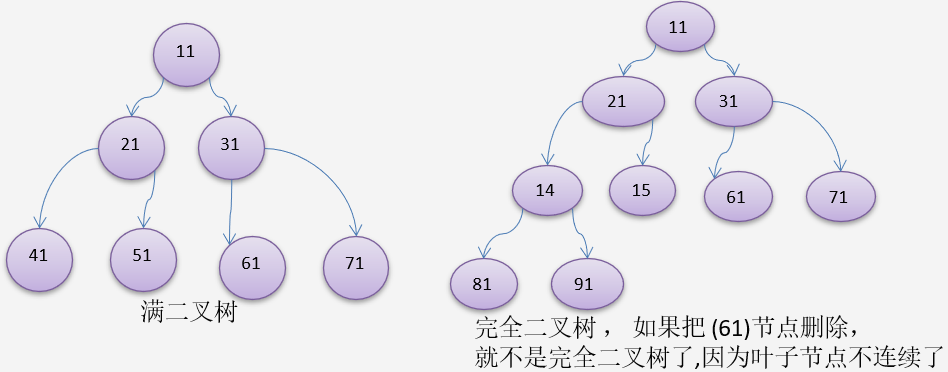
能提高数据存储，读取的效率, 比如利用 二叉排序树(Binary Sort Tree)，既可以保证数据的检索速度，同时也可以保证数据的插入，删除，修改的速度

## 树的基本概念

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到该节点路线)
8. 层
9. 子树
10. 树的高度(最大层数)
11. 森林：多颗子树构成

## 二叉树的概念

1. 每个节点最多只能有两个子节点的一种树成为二叉树；
2. 二叉树的子节点分为左节点、右节点；
3. 如果二叉树所有叶子节点都在最后一侧，节点总数为2n-1，n为层数，成为满二叉树；
4. 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；



## 二叉树的遍历

分为前序遍历、中序遍历和后序遍历；

都是先左后右，System.*out*.println(**this**);的位置取决于前中后序遍历；

看输出父节点的顺序，就确定是前序，中序还是后序；

1. 前序遍历:

先输出父节点，再遍历左子树和右子树

1. 先输出当前节点(初始的时候是root节点)；
2. 如果左子节点不为空，则递归继续前序遍历；
3. 如果右子节点不为空，则递归继续前序遍历；

**public void** preOrderIterator(){  
 System.*out*.println(**this**);  
 //递归向左遍历  
 **if** (**this**.left != **null**){  
 **this**.left.preOrderIterator();  
 }  
 //递归向右遍历  
 **if** (**this**.right != **null**){  
 **this**.right.preOrderIterator();  
 }  
}

1. 中序遍历:

先遍历左子树，再输出父节点，再遍历右子树

1. 如果当前节点的左子节点不为空，递归中序遍历；
2. 输出当前节点；
3. 如果当前节点的右子节点不为空，递归中序遍历；

**public void** infixOrderIterator(){  
 //递归向左遍历  
 **if** (**this**.left != **null**){  
 **this**.left.infixOrderIterator();  
 }  
 System.*out*.println(**this**);  
 //递归向右遍历  
 **if** (**this**.right != **null**){  
 **this**.right.infixOrderIterator();  
 }  
}

1. 后序遍历:

先遍历左子树，再遍历右子树，最后输出父节点；

1. 如果当前节点的左子节点不为空，递归后序遍历；
2. 如果当前节点的右子节点不为空，递归后序遍历；
3. 输出当前节点；

**public void** rightOrderIterator(){  
 //递归向左遍历  
 **if** (**this**.left != **null**){  
 **this**.left.rightOrderIterator();  
 }  
 //递归向右遍历  
 **if** (**this**.right != **null**){  
 **this**.right.rightOrderIterator();  
 }  
 System.*out*.println(**this**);  
}

## 二叉树的查找

分为前序遍历、中序遍历和后序遍历；

代码实现同遍历；

根据遍历的顺序来判断查找的次数；

1. 前序查找:
2. 先判断当前节点的值是否等于要查找；
3. 如果相等，则返回当前节点；
4. 如果不等，则判断当前节点的左子节点是否为空，不为空则递归前序查找；
5. 左子节点中若未找到，则判断当前节点的右子节点是否为空，不为空则向右递归查找；
6. 中序查找
7. 判断当前节点的左子节点是否为空，不为空则递归中序查找；
8. 如果找到，则返回；如果没有找到，就和当前节点比较，如果相等，返回当前节点；
9. 否则再右子节点递归中序查找，找到就返回，反之返回null；

**public** HeroNode infixOrderSearch(**int** no){   
 //递归向左遍历  
 HeroNode heroNode = **null**;  
 **if** (**this**.left != **null**){   
 heroNode = **this**.left.infixOrderSearch(no);  
 }

**//此处这个判断必须加 要不然在左子找到了 没有返回又在右支查了一下 返回了null 则最后还是返回为null**

**//其他查找同理**  
 **if** (heroNode != **null**){  
 **return** heroNode;  
 }  
 **if** (**this**.no == no){  
 **return this**;  
 }  
 //递归向右遍历  
 **if** (**this**.right != **null**){   
 heroNode = **this**.right.infixOrderSearch(no);  
 }  
 **return** heroNode;  
}

1. 后序查找
2. 判断当前节点的左子节点是否为空，不为空则递归后序查找；
3. 如果找到，则返回；否则再右子节点递归中序查找，找到就返回；
4. 和当前节点比较，是则返回，反之返回null;

## 二叉树删除节点

1. 简单删除

规定：

1） 如果是叶子节点，则直接删除；

2） 如果是非叶子节点，则删除该子树；

思路：

1. 因为二叉树是单向的，所以需要判断的是当前节点的子节点是否需要删除，而不能去判断当前节点是否需要删除；

因为当前节点设为null也没用，需要将其父节点对应的左/右子节点设为null；

1. 首先要判断root节点是否是待删除节点，是则将其置为null；
2. 如果当前节点的左子节点不为空，并且左子节点是要删除的节点，则**this**.left = **null**；并结束递归返回；
3. 如果当前节点的右子节点不为空，并且右子节点是要删除的节点，则**this**.right = **null**；并结束递归返回；
4. 如果3) 4)没有找到删除的节点，判断左子树不为null，则向左子树递归删除；
5. 如果5)没有找到删除的节点，就需要判断右子树不为null，则向右子树递归删除；

在BinaryTree中：

**public boolean** preOrderDelete(**int** no){  
 **if** (root == **null**){  
 System.*out*.println(**"该二叉树为空"**);  
 **return false**;  
 }  
 **if** (**this**.root.getNo() == no){  
 **this**.root = **null**;  
 **return true**;  
 }  
 **return this**.root.preOrderDelete(no);  
}

在HeroNode类：

**public boolean** preOrderDelete(**int** no){  
 **boolean** isDelete = **false**;  
 **if** (**this**.left != **null** && **this**.left.no == no){  
 **this**.left = **null**;  
 **return true**;  
 } **else if** (**this**.right != **null** && **this**.right.no == no){  
 **this**.right = **null**;  
 **return true**;  
 }  
 **if** (**this**.left != **null**){  
 isDelete = **this**.left.preOrderDelete(no);  
 }  
 **if** (isDelete) **return true**;  
   
 **if** (**this**.right != **null**){  
 isDelete = **this**.right.preOrderDelete(no);  
 }  
 **return** isDelete;  
}

## 顺序存储二叉树

1. 基本说明：

从数据存储来看，数组存储方式和树的存储方式是可以相互转换的；

即树可以转换为数组，数组可以转换为树；

应用场景是堆排序；

要求：

1. 右图中的二叉树的节点要求右图的二叉树的结点，要求以数组的方式来存放 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；
3. 顺序存储二叉树的特点：
4. 顺序二叉树通常只考虑完全二叉树；
5. 第n个元素的左子节点为 2 \* n + 1；
6. 第n个元素的右子节点为 2 \* n + 2；
7. 第n个元素的父节点为 (n-1) / 2；
8. n : 表示二叉树中的第几个元素(按0开始编号)；
9. 案例展示

给定数组[1, 2, 3, 4, 5, 6, 7]，要求以二叉树前序遍历：1,2,4,5,3,6,7；

参考前序、中序、后序 只需要交换打印语句的顺序即可；

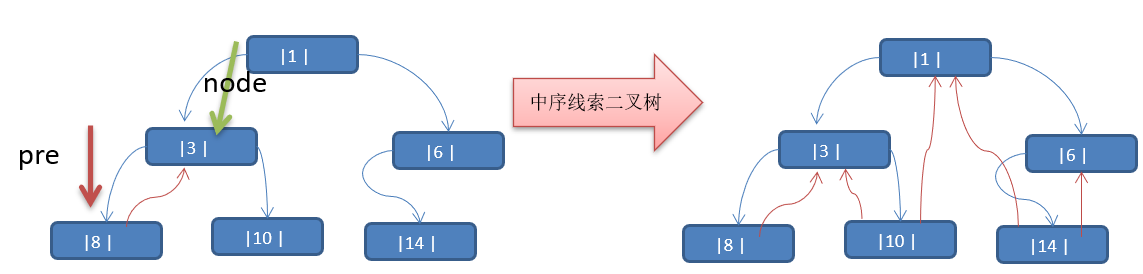
**class** ArrBinaryTree {  
 **private int**[] arr;//存储数据结点的数组  
 **public** ArrBinaryTree(**int**[] arr) {  
 **this**.arr = arr;  
 }  
 //重载preOrder  
 **public void** preOrder() {  
 **this**.preOrder(**0**);  
 }  
 //编写一个方法，完成顺序存储二叉树的前序遍历  
 **public void** preOrder(**int** index) {  
 //如果数组为空，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**);  
 }  
 }  
}

## 线索化二叉树

1. 线索化二叉树基本介绍
2. n个节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，存放指向节点在某种遍历次序下的前驱和后继节点的指针（这种附加的指针成为线索）；

1. 这种加上线索的二叉树链表称为线索链表，相应的二叉树成为线索二叉树(Threaded Binary Tree)。
2. 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（对应不同的遍历方式）；
3. 一个节点的前一个节点称为前驱节点；
4. 一个节点的后一个节点称为后继节点；



1. 二叉树线索化的实现：

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。

//对node节点的参数信息进行改造 方便遍历

**class** HeroNode {  
 **private int** no;  
 **private** String name;  
 **private** HeroNode left;  
 **private** HeroNode right;  
 //0 表示指向的是左子节点 1 表示指向的是前驱节点  
 **private int** leftType;  
 //0 表示指向的是右子节点 1 表示指向的是后继节点  
 **private int** rightType;

}

**public void** infixThreadedNode(){  
 **this**.infixThreadedNode(root);  
}  
/\*\*  
 \* 中序线索化二叉树  
 \* node 待线索化的节点  
 \*/  
**public void** infixThreadedNode(HeroNode node){  
 **if** (node == **null**){ **return**; }  
 //先线索化左子树  
 infixThreadedNode(node.getLeft());  
 /\* 1  
 \* 3 6  
 \* 8 10 14   
 \*/  
 //线索化当前节点 仅在当前节点的左/右子节点为null才处理  
 **if** (node.getLeft() == **null**){  
 //让当前左指针指向前驱节点  
 node.setLeft(pre);  
 //修改当前节点做指针的类型  
 node.setLeftType(**1**);  
 }  
 //处理后继节点 应该是在递归后的下一次才执行 的  
 //因为第一次进入这个判断是进不去的  
 //以8节点为例 执行到这时 是对于3节点的判断  
 //node ->HeroNode{no=3, name='FW', leftType=0, rightType=0}  
 //pre ->HeroNode{no=8, name='QP', leftType=1, rightType=0}  
 **if** (pre != **null** && pre.getRight() == **null**){  
 pre.setRight(node);  
 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;  
 //线索化右子树  
 infixThreadedNode(node.getRight());  
}

1. 遍历线索化二叉树

//HeroNode类内

//参考二叉树中序遍历

**public void** infixList(){  
 **if** (**this**.getLeft() != **null** && **this**.getLeftType() == **0**){  
 **this**.getLeft().infixList();  
 }  
 System.*out*.println(**this**);  
 **if** (**this**.getRight() != **null** && **this**.getRightType() == **0**){  
 **this**.getRight().infixList();  
 }  
}

//老师的参考

**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();  
 }  
}

## 树结构的应用 — 堆排序

1. 堆排序的介绍
2. 对排序是利用**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、平均时间复杂度都是O(nlogn)，是不稳定排序；
3. 堆是具有以下性质的完全二叉树：

每个节点的值都>=其左右子节点的值，称为大顶堆；

PS：左右子节点的值没有明确大小要求；

1. 每个节点的值都<=其左右子节点的值，称为小顶堆；

|  |  |
| --- | --- |
| 大顶堆 | 小顶堆 |
|  |  |
| 映射为数组：  [50,45,40,20,25,35,30,10,15]  大顶堆的特点：  arr[i] >= arr[2i+1] &&  arr[i] >= arr[2i+2]  其实应i节点的子节点为2i+1和2i+2; | 映射为数组  [10,20,15,35,50,30,40,35,45]  小顶堆的特点：  arr[i] <= arr[2i+1] &&  arr[i] <= arr[2i+2] |

1. 堆排序的思想：
2. 将待排序的序列构造成一个大顶堆；
3. 此时整个序列的最大值位于堆顶的根节点；
4. 将其与末尾元素交换，此时末尾为最大值；
5. 然后n-1个元素重复上述操作（实现从小到大）；

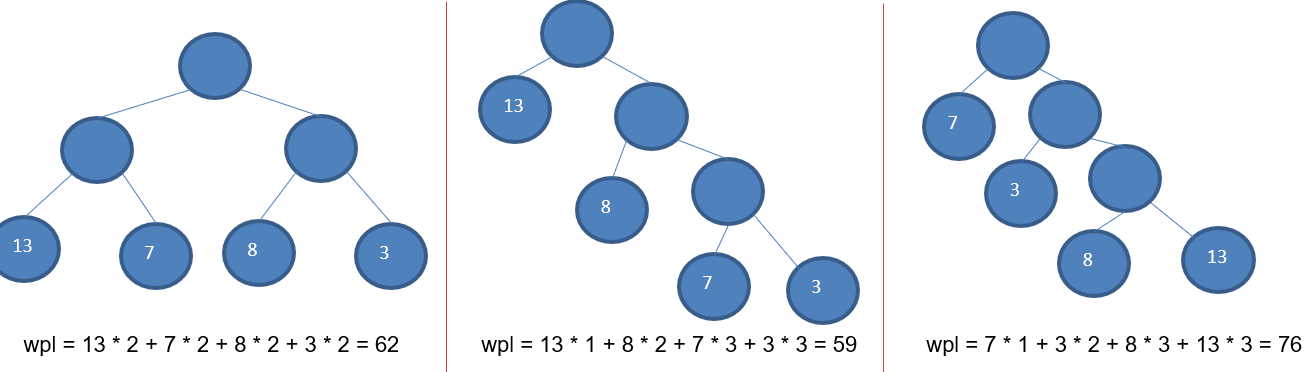
关键点：最后一个非叶子节点为i= array.length/**2**-**1**

**public static void** heapSort(**int**[] array){  
 //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=array.length/**2**-**1**;i>=**0**;i--){  
 *adjustHeap*(array,i,array.length);  
 }  
 /\*  
 \* 将堆顶元素与末尾元素交换，将最大元素"沉"到数组末端;  
 \* 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=array.length-**1**;j>**0**;j--){  
 **int** temp = array[**0**];  
 array[**0**] = array[j];  
 array[j] = temp;

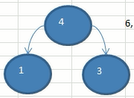
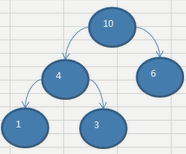
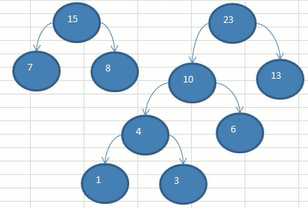
**//都是以0为顶构筑大顶堆**  
 *adjustHeap*(array,**0**,j);  
 }  
}  
/\*\*  
 \* 将以i为顶的对变成了大顶堆  
 \*/  
**public static void** adjustHeap(**int**[] arr,**int** i,**int** length){  
 **int** temp = arr[i];  
 //n=2\*i+1 根据当前指定节点来判断其子节点 2\*i+1 左子节点  
 **for** (**int** n=**2**\*i+**1**;n<length;n=n\***2**+**1**){  
 //存在右子节点 且右>左 替换为右子节点  
 **if** (n+**1** < length && arr[n] < arr[n+**1**]){  
 n++; //就让当前指向右子节点  
 }  
 **if** (arr[n] > temp){  
 arr[i] = arr[n]; //大的值移到父节点  
 i = n; //继续循环比较 确保后面的子节点也都满足条件  
 } **else** {  
 **break**; //使用此处的前提就是他下面的节点都已经比较过了  
 }  
 arr[i] = temp;  
 }  
}

## 树结构的应用 — 赫夫曼树

1. 赫夫曼树的基本介绍和相关概念
2. 给定n个权值作为n个叶子节点，构造一个二叉树，若该树的带权路径长度(wpl-weight path length)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);
3. 哈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；
4. 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；
5. 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；
6. 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；
7. WPL最小的就是赫夫曼树；



PS:!!!!!只能是叶子节点！只能是叶子节点！

1. 生成哈夫曼树的步骤：
2. 从小到大进行排序,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(左右节点为null)；
3. 取出根节点权值最小的两颗二叉树；
4. 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(取出最小两个值，两者之和作为权值成为他们的根节点。array=[1,3,6,7,8,13,29];
5. 取出待计算的数列最小值 1,3；两者为左右节点生成一个二叉树；他们的和为4仍然处于数组最小值；
6. 再取出数组中剩余数字的最小值6，与前面这个二叉树组成一个新的二叉树；仍然以左右子节点的和作为根节点的权；
7. 然后让左子节点指向权值较大的，右子节点指向权值较小的；
8. 这时10比7 8 大，所以7和8重复第i步，生成一个以15为顶点的二叉树;
9. 这时排序应为[10,13,15,29]，所以10和13组成二叉树，根节点变成了23；整体[15,23,29]；
10. 15与23再组成一个二叉树；
11. 以此类推；)；
12. 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；

**private static void** createHuffman(**int**[] array){  
 //将array每个值都构造成一个node，并将其放入List中  
 List<Node> nodes = **new** ArrayList<>();  
 Arrays.*stream*(array).forEach(a -> nodes.add(**new** Node(a)));  
 **while** (nodes.size() > **1**){  
 Collections.*sort*(nodes);  
 Node left = nodes.get(**0**);  
 Node right = nodes.get(**1**);  
 Node parent = **new** Node(left.value + right.value);  
 nodes.remove(left);  
 nodes.remove(right);  
 parent.left = left;  
 parent.right = right;  
 nodes.add(parent);  
 }  
}

1. 哈夫曼编码
2. 赫夫曼编码也翻译为哈夫曼编码(Huffman Coding)，又称哈夫曼编码，是一种编码方式, 属于一种程序算法；
3. 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。
4. 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；
5. 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；

应用举例：

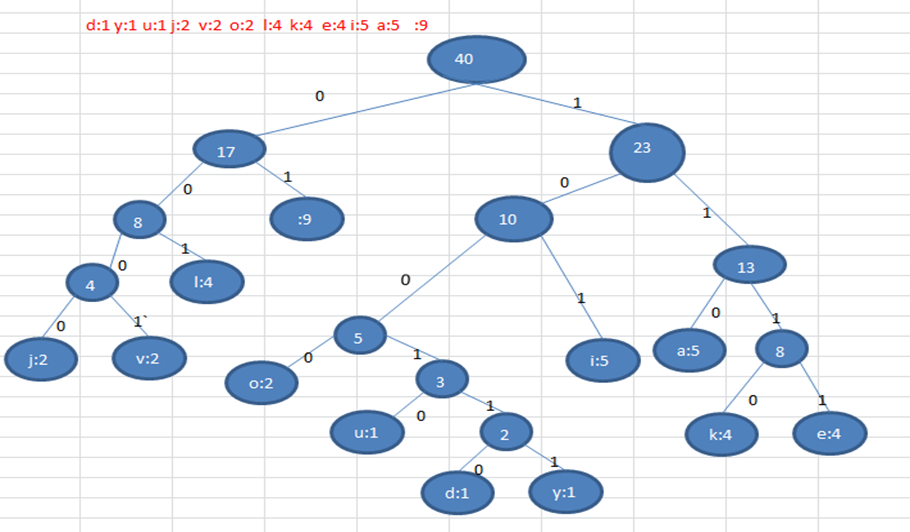
* 传输的字符串为：

i like like like java do you like a java

* 各个字符对应的个数

d:1 y:1 u:1 j:2 v:2 o:2 l:4 k:4 e:4 i:5 a:5 空格:9

* 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；



* 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

这个是前缀编码：任何一个编码都不会是另外一个编码的前缀，不会造成匹配的多样性；

* 按照上面的赫夫曼编码，我们的"i like 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)

1010100110111101111010011011110111101001101111011110100001100001110011001111000011001111000100100100110111101111011100100001100001110 通过赫夫曼编码处理长度：133

注意：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, 最后生产的赫夫曼编码长度时一样的；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；

1. 哈夫曼压缩的注意事项

* 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；
* 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)；
* 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；

# 二叉树重点 - 二叉排序树和平衡二叉树

## 二叉排序树(Binary Sort Tree BST)

1. 二叉排序树的介绍：

对于一个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**比当前的值**大**；

如果是相同的值，可以放在左子节点或者右子节点；

[7, 3, 10, 12, 5, 1, 9]:



1. 二叉树的添加

对一个二叉排序树中序输出刚好就是一个升序有序的序列；

code：

**public class** BinarySortTree {  
 **private** SortNode root;  
 **public** BinarySortTree(SortNode root) {  
 **this**.root = root;  
 }  
 **public void** add(SortNode newNode){  
 **if** (**this**.root == **null**) **return**;  
 **this**.root.createSortTree(newNode);  
 }  
}  
**class** SortNode{  
 **int** value;  
 SortNode left;  
 SortNode right;  
 **public** SortNode(**int** value) {  
 **this**.value = value;  
 }  
 **public void** createSortTree(SortNode newNode){  
 **if** (newNode == **null**) **return**;  
 //同当前节点比较  
 **if** (**this**.value >= newNode.value ){  
 **if** (**this**.left != **null**){  
 **this**.left.createSortTree(newNode);  
 } **else** {  
 **this**.left = newNode;  
 }  
 } **else** {  
 **if** (**this**.right != **null**){  
 **this**.right.createSortTree(newNode);  
 } **else** {  
 **this**.right = newNode;  
 }  
 }  
 }  
}

1. 二叉树的删除

删除节点的思路：

1. 先去找到要删除的节点；
2. 找到targetNode的父节点parent
3. 确定targetNode是parent的左子节点还是右子节点
4. 删除分三种情况考虑：
5. 删除的targetNode是叶子节点：直接删除；
6. 找到targetNode的父节点parent;
7. 确定targetNode是parent的左子节点还是右子节点

对应将左/右子节点设为null；

parent.left = null; /

parent.right = null;

1. 删除的targetNode是有一个子树的节点：
2. 如果targetNode有左子节点：
3. 如果targetNode是parent的左子节点，则：

parent.left = targetNode.left;

1. 如果targetNode是parent的右子节点，则：

parent.right = targetNode.left;

1. 如果targetNode有右子节点：
2. 如果targetNode是parent的左子节点，则：

parent.left = targetNode.right;

1. 如果targetNode是parent的右子节点，则：

parent.right = targetNode.right;

1. 删除的targetNode是有两个子树的节点：
2. 找到targetNode的父节点parent；
3. 从targetNode的右子树找到最小的节点(或者左子树最大的节点)[PS：此处肯定会找到这个分子的最左或者最右一个叶子节点]；
4. 用一个临时变量temp将该最小节点的只保存；
5. 删除这个临时节点；
6. targetNode.value=temp;
7. 删除节点的代码：

BinarySortTree

**public void** delete(**int** value){  
 **if** (root == **null**) **return**;  
 **if** (root.left==**null**&&root.right==**null**&&root.value == value){  
 **this**.root = **null**;  
 **return**;  
 }  
 SortNode target = search(value);  
 **if** (target == **null**){ **return**; }  
 SortNode parent = searchParent(value);  
 //删除的是叶子节点  
 **if** (target.left == **null** && target.right == **null**){  
 **if** (parent.left!=**null** && target.value==parent.left.value){  
 parent.left = **null**;  
 } **else** {  
 //已经确认父子关系 不是左必是右  
 parent.right = **null**;  
 }  
 }**else if** (target.left != **null** && target.right != **null**){  
 //删除的是有两个子节点  
 SortNode lastRightNode = target.getRightBranchSmallestNode();  
 **int** temp = lastRightNode.value;  
 **this**.delete(temp);  
 target.value = temp;  
 } **else if** (target.left != **null**){  
 //待删除节点还有一个左子节点  
 //假如剩余两个节点 删除root则存在其父节点不存在的情况   
 // 报空指针 加parent == null  
 **if** (parent == **null**){  
 **this**.root = target.left;  
 System.*out*.println(root);  
 } **else if** (target == parent.left){  
 parent.left = target.left;  
 } **else** {  
 parent.right = target.left;  
 }  
 } **else** {  
 **if** (parent == **null**){  
 **this**.root = target.right;  
 System.*out*.println(root);  
 }**else if** (target == parent.left){  
 parent.left = target.right;  
 } **else** {  
 parent.right = target.right;  
 }  
 }  
}

SortNode

**public** SortNode getRightBranchSmallestNode(){  
 **if** (**this**.right == **null**) **return null**;  
 **return this**.right.getSmallestNode();  
}  
/\*\*  
 \* 获取当前节点的最小节点  
 \*/  
**public** SortNode getSmallestNode(){  
 **if** (**this**.left != **null**) {  
 **return this**.left.getSmallestNode();  
 } **else** {  
 **return this**;  
 }  
}

1. 根据中序遍历获取排序后的数组

**private static int** *index* = **0**;  
**public void** infix(**int**[] array){  
 **if** (**this**.left != **null**) **this**.left.infix(array);  
 System.*out*.println(**this**);  
 array[*index*++] = **this**.value;  
 **if** (**this**.right != **null**) **this**.right.infix(array);  
}

## 平衡二叉树（AVL数）

对于二叉排序树，如果原始数组是有序的(升序或者降序)，则该二叉树只会有左侧分支或者右侧分支，就间接相当于一个链表，并且并链表还多一个节点，效率更低一点；提出用平衡二叉树解决这个缺陷；

他的构件是基于BST，所以基本方法是在其上的增加

1. 平衡二叉树基本介绍
2. 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(得名于其发明者)，可以保证查询效率较高；
3. 具有以下特点：
4. 它是一颗空树或它的左右两个子树的高度差绝对值不超过1，并且左右两个子树都是一个平衡二叉树；
5. 平衡二叉树的实现方法有：

红黑树、AVL、替罪羊树、Treap、伸展树等；

那边低往那边旋转

右比左高 左旋转 左比右高 右旋转

1. 核心算法：获取子树的高度

**public int** getLeftHight(){  
 **if** (**this**.left == **null**) **return 0**;  
 **return** left.getHight();  
}  
  
**public int** getRightHight(){  
 **if** (**this**.right == **null**) **return 0**;  
 **return** right.getHight();  
}  
/\*\*  
 \* 返回以当期节点为根节点的高度  
 \* 应该是左/右子树比较大的值  
 \*/  
**public int** getHight(){  
 //因为自身节点高度为1 所以+1  
 **return** Math.*max*(left==**null**?**0**:left.getHight(),

right==**null**?**0**:right.getHight()) + **1**;  
}

1. 左旋转

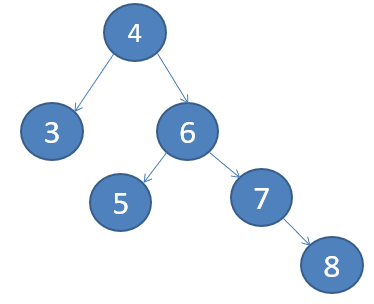
如[4,3,6,5,7]中插入8，就不满足高度差绝对值<=1;需要进行左旋转操作，

这个方法是写在Node对应的类中：

条件： 右 - 左 > 1;

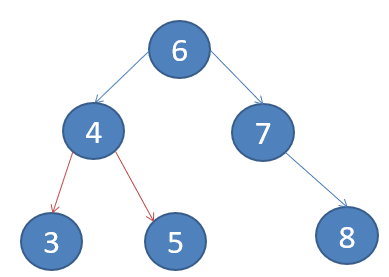
右右 - 右左 > 1;

(子树和子树的子树都是右侧高1 左旋)；

1. 创建一个新节点newNode(以4位置进行创建，即以当前节点的根节点进行创建)；
2. 把新节点newNode的左子树设置为当前节点(当前根节点)的左子树：

newNode.left = left；

1. 把新节点newNode的右子树设为为当前节点右子树的左子树：

newNode.right = right.left；

1. 把当前节点的值换成右子节点的值；

value= right.value;

1. 把当前节点的右子树(此时当前节点已变成以6为根节点)设置成右子树的右子树：

right= right.right;

1. 把当前节点的左子树设置为新节点

left = newNode;

PS：相当于将最初的相对根节点的第一个右子节点提上去作为新的相对root节点，就需要将其的左节点设成最初根节点第一个左子节点的右子节点；而新提上去的根节点原本的左子树还是自己的左子树；

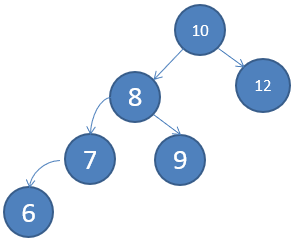
code：

**public void** leftRotate(){  
 AVLNode newNode = **new** AVLNode(**this**.value);  
 newNode.left = **this**.left;  
 newNode.right = **this**.right.left;  
 **this**.value= **this**.right.value;  
 **this**.right= **this**.right.right;  
 **this**.left = newNode;  
}

1. 右旋转

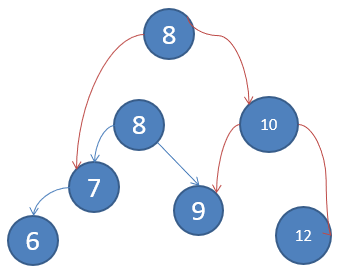
如[10,12, 8, 9, 7]中插入新值6，需要执行右旋转操作：

需要将9这个节点，通过右旋转到右子树

1. 创建一个新节点newNode(以10这个值创建，即值等于当前根节点的值因为要将他移下去)；
2. 把新节点的右子树设置成当前节点的右子树：

newNode.right= this.right；

1. 把新节点的左子树设置为当前节点的左子树的右子树：

newNode.left = this.left.right;

1. 把当前节点的值换成左子节点的值：

this.value= this.left.value;

1. 把当前节点的左子树设置成左子树的左子树：

this.left = left.left;

1. 把当前节点的右子树设置为新节点：

this.right = newNode;

PS：因为左子树高了，所以需要将左子树的第一个节点(原root的第一个左子节点)作为newNode提上去，而她的右子节点必然小于原root的第一个右子节点，所以要变成newNode的左子节点(3)),原有的右子树不动(2)), 把当前节点（10节点）的值换成左子节点的值(8)需要将要提上去的节点(8节点)的左子树变为当前的左子树(5)),把新生成的有右子树的newNode作为当前的右节点；

code：

**public void** rightRotate(){  
//1) 创建一个新节点newNode(以10这个值创建，即值等于当前根节点的值因为要将他移下去)；  
AVLNode newNode = **new** AVLNode(**this**.value);  
//2) 把新节点的右子树设置成当前节点的右子树：  
newNode.right= **this**.right;  
//3) 把新节点的左子树设置为当前节点的左子树的右子树：  
newNode.left = **this**.left.right;  
//4) 把当前节点的值换成左子节点的值：  
**this**.value= **this**.left.value;  
//5) 把当前节点的左子树设置成左子树的左子树：  
**this**.left = **this**.left.left;  
//6) 把当前节点的右子树设置为新节点：  
**this**.right = newNode;

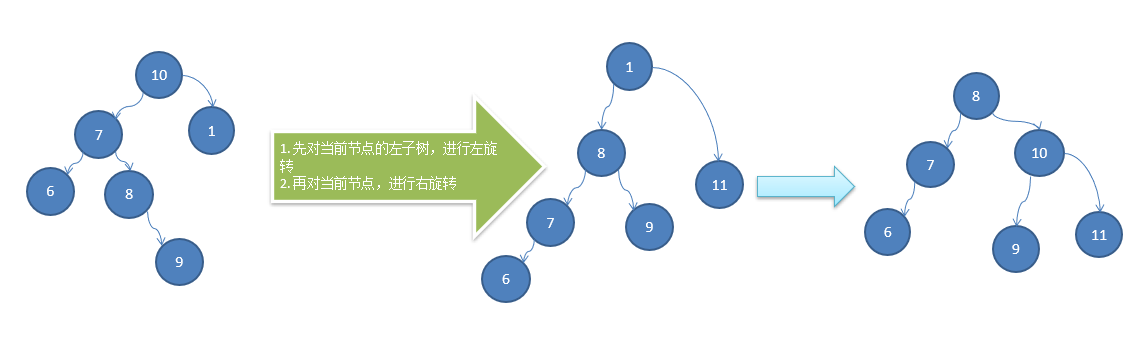
}

1. 双旋转



仍然不是一个AVL；

需要其左子树进行左旋转，然后对整体进行右旋转；

另外一种情况以此类推；

条件： 左 - 右 > 1 并且

左左 - 左右 > 1;

(子树和子树的子树都是左侧高1 右旋)；

左比右高，但是左子树的右比左高，

左子树左旋，整体右旋；

--------------------------------------------------------------------

右 - 左 > 1 并且

右右 - 右左 > 1;

(右子树和子树的子树都是右侧高1 左旋)；

右比左高，但是右子树的左比右高，

右子树右旋，整体左旋；

1. 平衡二叉树添加节点的代码实现

SelfBlanceTree 类：

**public void** add(AVLNode newNode){  
 **if** (**this**.root == **null**){  
 **this**.root = newNode;  
 **return**;  
 }  
 **this**.root.createSortTree(newNode);  
}

AVLNode类：

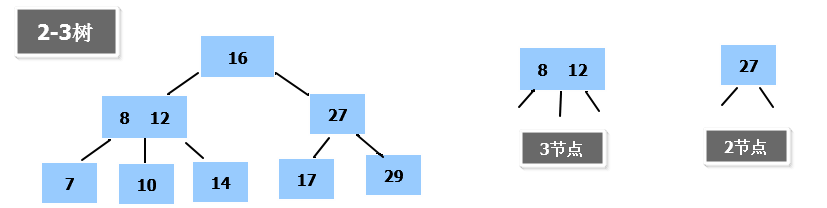
**public void** createSortTree(AVLNode newNode){  
 **if** (newNode == **null**) **return**;  
 //同当前节点比较  
 **if** (**this**.value >= newNode.value ){  
 **if** (**this**.left != **null**){  
 **this**.left.createSortTree(newNode);  
 } **else** {  
 **this**.left = newNode;  
 }  
 } **else** {  
 **if** (**this**.right != **null**){  
 **this**.right.createSortTree(newNode);  
 } **else** {  
 **this**.right = newNode;  
 }  
 }  
 //这个方法是被root调用的  
 // 右比左高 左旋转  
 **if** (getRightHight() - getLeftHight() > **1**){  
 **if** (right != **null** && right.getRightHight() <right.getLeftHight()){  
 //先对右子树进行右旋转  
 **this**.right.rightRotate();  
 //最整体左旋转  
 **this**.leftRotate();  
 } **else** {  
 leftRotate();  
 }  
 **return**;  
 }  
 **if** (getLeftHight() - getRightHight() > **1**){  
 **if** (**this**.left!= **null** && left.getLeftHight() < left.getRightHight()){  
 //先对左子树进行左旋转  
 **this**.left.leftRotate();  
 //最整体右旋转  
 **this**.rightRotate();  
 } **else** {  
 rightRotate();  
 }  
 }  
}

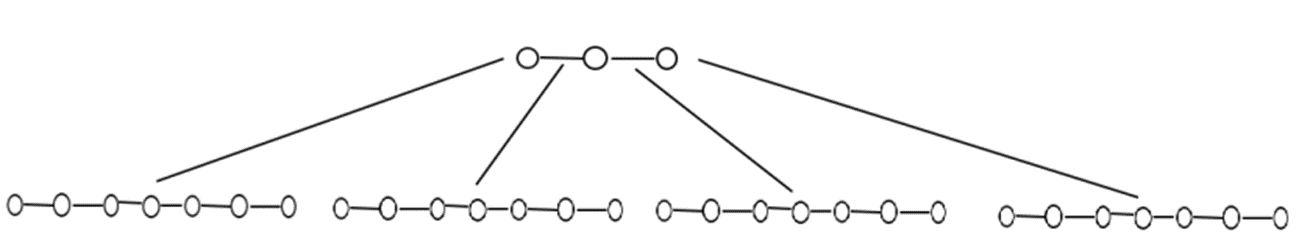
# 多路查找数

1. 二叉树存在的一些问题：

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:

1. 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
2. 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
3. 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）；
4. 比如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。





1. 如图B树(balance tree)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;

## 2-3树的基本介绍

1. 2-3树的基本特点

2-3树是最简单的B树结构, 具有如下特点:

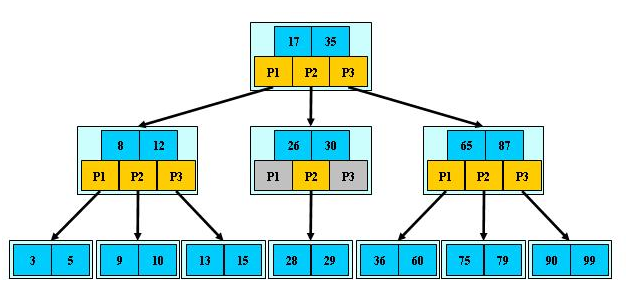
1. 2-3树的所有叶子节点都在同一层.(只要是B树都满足这个条件);
2. 有两个子节点的节点叫二节点，二节点要么没有子节点，要么有两个子节点;
3. 有三个子节点的节点叫三节点，三节点要么没有子节点，要么有三个子节点;
4. 2-3树是由二节点和三节点构成的树。
5. 2-3树的插入规则：
6. 2-3树的所有叶子节点都在同一层(只要是B树都满足这个条件);
7. 有两个子节点的节点叫二节点，二节点要么没有子节点，要么有两个子节点;
8. 有三个子节点的节点叫三节点，三节点要么没有子节点，要么有三个子节点
9. 当按照规则插入一个数到某个节点时，不能满足上面三个要求，就需要拆，先向上拆，如果上层满，则拆本层，拆后仍然需要满足上面3个条件;
10. 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：

左子节点比权值大，中间值介于两者之间，右子节点比权值大；

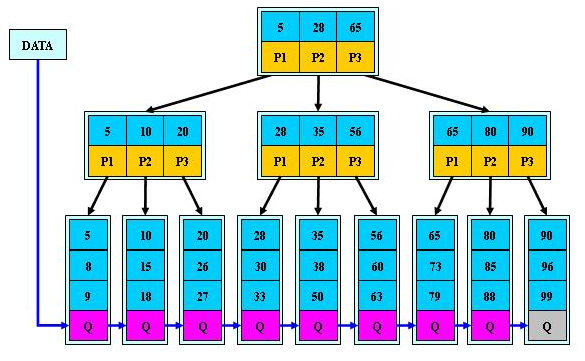
## B树

B树即Balance Tree,B-Tree指的就是B树；是一个平衡树AVL，所以也是一个搜索树；

1. B树的介绍：



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;
6. B+树的介绍

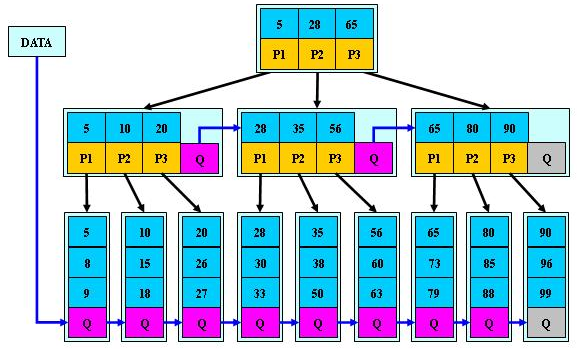


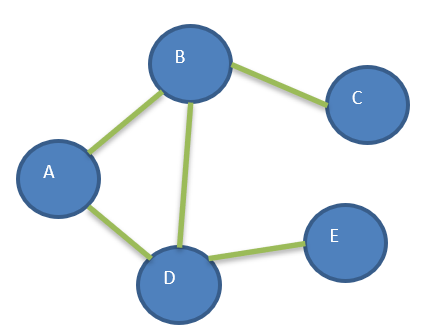
1. B+树的搜索与B树也基本相同，区别是B+树只有达到叶子结点才命中（B树可以在非叶子结点命中），其性能也等价于在关键字全集做一次二分查找;

非叶子节点上的值是索引，不是他的值；

1. 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。
2. 不可能在非叶子结点命中;
3. 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;
4. 更适合文件索引系统;
5. 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;
6. 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。



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B+树的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；
3. B 树和 B+树的区别， 为什么 MySQL 要使用 B+树

B 树：

1. 关键字集合分布在整颗树中；
2. 任何一个关键字出现且只出现在一个结点中；
3. 搜索有可能在非叶子结点结束；
4. 其搜索性能等价于在关键字全集内做一次二分查找；

B+树：

1. 有 n 棵子树的非叶子结点中含有 n 个关键字（ b 树是 n-1 个） ， 这些关键字不保存数据， 只用来索引， 所有数据都保存在叶子节点（b 树是每个关键字都保存数据）；
2. 所有的叶子结点中包含了全部关键字的信息， 及指向含这些关键字记录的指针，且叶子结点本身依关键字的大小自小而大顺序链接；
3. 所有的非叶子结点可以看成是索引部分， 结点中仅含其子树中的最大（或最小）关键字；
4. 通常在 b+树上有两个头指针， 一个指向根结点， 一个指向关键字最小的叶子结点；

（ 5） 同一个数字会在不同节点中重复出现， 根节点的最大元素就是 b+树的最大元

素。

B+树相比于 B 树的查询优势：

1. B+树的中间节点不保存数据， 所以磁盘页能容纳更多节点元素，更“矮胖”；
2. B+树查询必须查找到叶子节点， B 树只要匹配到即可不用管元素位置， 因此 B+树查找更稳定（并不慢） ；
3. 对于范围查找来说， B+树只需遍历叶子节点链表即可， B 树却需要重复地中序遍

历

# 图的基本介绍和存储形式

## 图的介绍

线性表和树都是只有一个前驱和一个直接后继的关系；

图满足多对多的关系；

图是一种数据结构，其中的节点可以具有零个或多个相邻元素。两个节点之间的连接称为边。节点称为顶点；

## 图的基本概念

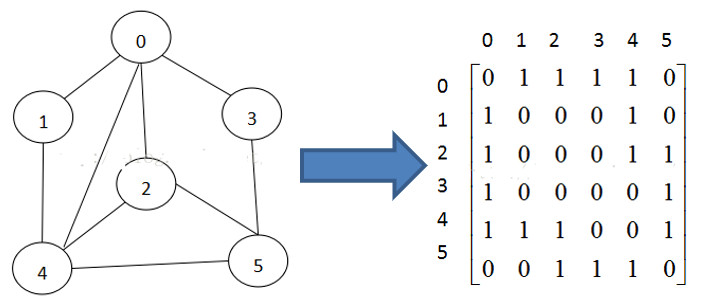
1. 顶点 vertex；
2. 边 edge
3. 路径
4. 无向图：节点之间没有方向
5. 有向图
6. 带权图

## 图的表示方式

二维数组(邻接矩阵)、链表表示(邻链表)；

## 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。

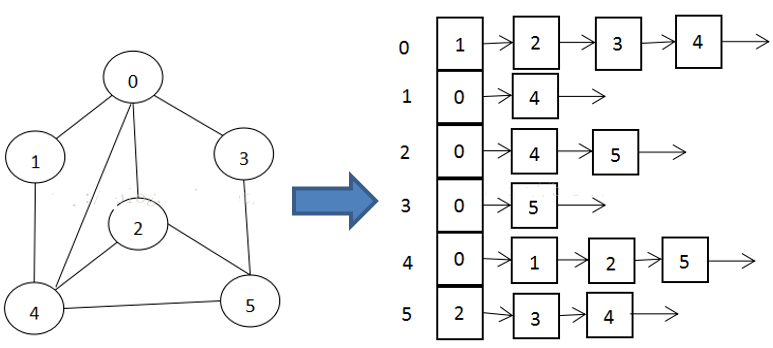


0与1联通，所以(0,1)处值为1，以此类推；

矩阵中0表示不连接，1表示连接

## 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;



标号为0的结点的相关联的结点为 1 2 3 4

标号为1的结点的相关联结点为0 4,以此类推...

# 常用的算法

## 二分查找算法

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(㏒₂n) ，即查找到需要的目标位置最多只需要㏒₂n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㏒₂100 , 即最多需要查找7次( 2^6 < 100 < 2^7)；

其实思路是一样的

**public static int** binarySearch(**int**[] array,**int** value){  
   
 **int** left = **0**;  
 **int** right = array.length - **1**;  
 **while** (left <= right){  
 **int** middle = (left + right) / **2**;  
 **if** (array[middle] == value){  
 **return** middle;  
 } **else if** (array[middle] < value){  
 left = middle + **1**;  
 } **else** {  
 right = middle - **1**;  
 }  
 }  
 **return** -**1**;  
}

## 分治算法

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

比如：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

1. 分治算法在每一层递归上都有三个步骤
2. 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;
3. 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;
4. 合并：将各个子问题的解合并为原问题的解;

分治(Divide-and-Conquer(P))算法设计模式

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；n0为一阈值，表示当问题P的规模不超过n0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n0时直接用算法ADHOC(P)求解。算法MERGE(y1,y2,…,yk)是该分治法中的合并子算法，用于将P的子问题P1 ,P2 ,…,Pk的相应的解y1,y2,…,yk合并为P的解。

/\*\*  
 \* @param *num* 总共几个盘子  
 \* @param *a* a柱  
 \* @param *b* b柱  
 \* @param *c* b柱  
 \*/  
**public static void** hanoi(**int** num,String a,String b,String c){  
 **if** (num == **1**){  
 System.*out*.println(**"第 1 个盘从 "** + a + **" -> "** + c);  
 *step*++;  
 } **else** {  
 //一直分到最后是两个盘来解决问题  
 //如果n>=2，总是将其看做事两个盘，上面是1，下面是2  
 //1 先把最上面所有盘 a->b 移动过程用到c  
 *hanoi*(num-**1**,a,c,b);  
 //2 把下边的盘a->c  
 System.*out*.println(**"第 "** + num + **" 个盘从 "** + a + **" -> "** + c);  
 *step*++;  
 //3 把B塔所有的盘 b -> c,移动过程中使用到a  
 *hanoi*(num-**1**,b,a,c);  
 }  
}

## 动态规划算法

EG：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；

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.

背包问题解析：

1. 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)
2. 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。
3. 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。

具体解析

动态规划问题就是一个填包的问题：

解决类似的问题可以分解成一个个的小问题进行解决：

假设存在背包容量大小分为1，2，3，4的各种容量的背包(分配容量的规则为最小重量的整数倍)；

物品的情况也分为没有物品、吉他、吉他+音箱、吉他+音箱+电脑

| **物品** | **0 磅** | **1磅** | **2磅** | **3磅** | **4磅** |
| --- | --- | --- | --- | --- | --- |
|  | 0 | 0 | 0 | 0 | 0 |
| 吉他(G) | 0 | 1500(G) | 1500(G) | 1500(G) | 1500(G) |
| 音响(S) | 0 |  |  |  |  |
| 电脑(L) | 0 |  |  |  |  |

对于第0行 没有任何物品可供选择；

对于第1行i=1， 当j=0，什么都不能装；

当j=1,2,3,4 也只能装一个吉他，如上图所示

对于第2行i=2，

当j=0，什么都不能装；

当j=1，放不下音箱，只能放吉他，所以**采用上面的方案**；

j=2,3，也是一样，所以**采用**对应的**上面的方案**；

| **物品** | **0 磅** | **1磅** | **2磅** | **3磅** | **4磅** |
| --- | --- | --- | --- | --- | --- |
|  | 0 | 0 | 0 | 0 | 0 |
| 吉他(G) | 0 | 1500(G) | 1500(G) | 1500(G) | 1500(G) |
| 音响(S) | 0 | 1500(G) | 1500(G) | 1500(G) | 3000(S) |
| 电脑(L) | 0 | 1500(G) | 1500(G) | 2000(L) | 2000(L)+1500(G) |

对于第3行i=3，

当j=0，什么都不能装；

当j=1，2，放不下音箱，只能放吉他，所以**采用上面的方案**；

当j=3可以放入本行的基本物品电脑；(优先被放入)

当j=4，可以放入本行的基本物品电脑+吉他；

得出公式：

1. v[i][0]=v[0][j]=0; //表示填入表第一行和第一列是0;
2. //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;

当w[i]> j 时：v[i][j]=v[i-1][j]

1. 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:

w[i] <= j 时：v[i][j]=max{v[i-1][j], v[i]+v[i-1][j-w[i]]}

v[i-1][j] //表示按照上一个方案来放置物品获得的值

v[i]+v[i-1][j-w[i]] //放入当前行基本物品v[i] 以及剩余匹配到的最优解

因为当前的步骤的之前步骤都已经是最优解了

代码实现

**public static void** main(String[] args) {  
 //记录各个物品的重量  
 **int**[] w = {**1**,**4**,**3**};  
 //记录对应的价值  
 **int**[] value = {**1500**,**3000**,**2000**};  
 **int** packageWeight = **4**; //背包容量  
 **int** n = value.length;  
 //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的背包的最大价值  
 **int**[][] v = **new int**[n+**1**][packageWeight+**1**];  
 //为了记录放如商品的情况 所以用二维数组来记录  
 **int**[][] store = **new int**[n+**1**][packageWeight+**1**];  
   
 //初始化第一行和第一列  
 //此处可以无需处理 默认为0 但是遇到有些情况可能需要处理  
   
 //不处理第一行 全为0  
 **for** (**int** i=**1**;i<v.length;i++){  
 **for** (**int** j = **1**; j < v[**0**].length; j++) {  
 //公式  
 //1 v[i][0]=v[0][j]=0; 已处理  
//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;  
 //当w[i]> j 时：v[i][j]=v[i-1][j]  
 //因为此处循环是从1开始的 所以-1  
 **if** (w[i-**1**] > j) {  
 v[i][j] = v[i-**1**][j];  
// store[i][j] = store[i-1][j];  
 } **else** {  
 //注意 此处因为从1循环 所以value[i-1]，v[i-1][j-w[i-1]]中的w[i-1]也要-1  
 //v[i][j]=max{v[i-1][j], v[i]+v[i-1][j-w[i]]}  
v[i][j]=v[i-**1**][j]>(value[i-**1**]+v[i-**1**][j-w[i-**1**]])?v[i-**1**][j]:(value[i-**1**]+v[i-**1**][j-w[i-**1**]]);   
 //此处是为了记录商品的存  
 **if** (v[i-**1**][j]<(value[i-**1**]+v[i-**1**][j-w[i-**1**]])){  
 //存放的是v[i-1][j] 只记录最有情况  
 store[i][j] = **1**;  
 }  
 }  
 }  
 }  
 Arrays.*stream*(v).forEach(vv -> System.*out*.println(Arrays.*toString*(vv)));  
 System.*out*.println(**"商品放入背包的情况"**);  
 **int** i = store.length - **1**,j=store[**0**].length-**1**;  
 **while** (i>**0** && j>**0**){  
 **if** (store[i][j] == **1**){  
 System.*out*.printf(**"第%d个商品放入背包**\n**"**,i);  
 j -= w[i-**1**];  
 }  
 i--;  
 }  
 }

## KMP算法

EG：对于字符串匹配问题

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

如果使用暴力匹配，效率很低

1. 如果当前字符匹配成功（即str1[i] == str2[j]），则i++，j++，继续匹配下一个字符
2. 如果失配（即str1[i]! = 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!)

暴力匹配算法实现：

**public static int** violenceMatchTeacher(String str,String match){  
 **char**[] char1 = str.toCharArray();  
 **char**[] char2 = match.toCharArray();  
 **int** i = **0**,j=**0**;  
 **while** (i<char1.length && j<char2.length){  
 **if** (char1[i] == char2[j]){  
 i++;j++;  
 } **else** {  
 i = i - (j-**1**);  
 j = **0**;  
 }  
 }  
 **return** j == char2.length?i-j:-**1**;  
}  
 /\*\*  
 \* 采用暴力匹配  
 \*/  
**public static int** violenceMatch(String str,String match){  
 String[] origin = str.split(**""**);  
 String[] mat = match.split(**""**);  
   
 **for** (**int** i=**0**;i<origin.length - mat.length;i++){  
   
 **if** (mat[**0**].equals(origin[i])){  
 **for** (**int** j=**0**;j<mat.length;j++){  
 String or = origin[i + j];  
 **if** (mat[j].equals(or)){  
 **if** (j == mat.length-**1** && mat[j].equals(or)){  
 System.*out*.println(**"完全匹配到了"**);  
 **return** i;  
 }  
 } **else** {  
 **break**;  
 }  
 }  
 }  
 }  
 **return** -**1**;  
}

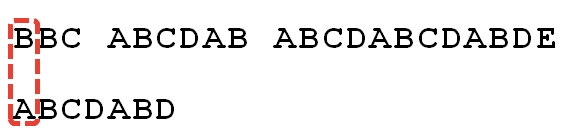
KMP算法介绍

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>

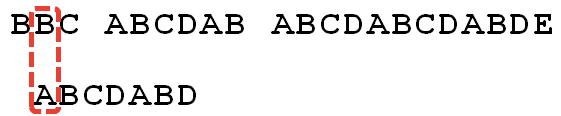
KMP算法举例

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”？

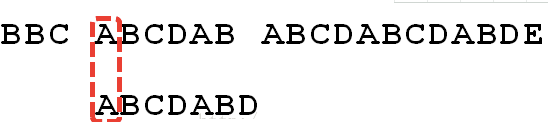
1. 首先，用Str1的第一个字符和Str2的第一个字符去比较，不符合，关键词向后移动一位



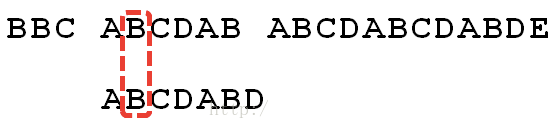
1. 重复第一步，还是不符合，再后移



1. 一直重复，直到Str1有一个字符与Str2的第一个字符符合为止



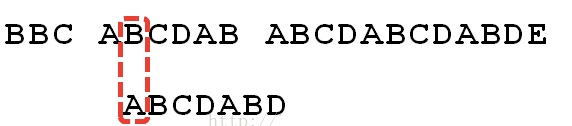
1. 接着比较字符串和搜索词的下一个字符，还是符合。



1. 遇到Str1有一个字符与Str2对应的字符不符合。



1. 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。)



1. 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

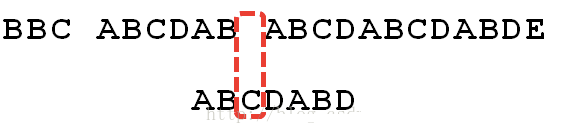


1. 已知空格与D不匹配时，前面六个字符”ABCDAB”是匹配的。查表可知，最后一个匹配字符B对应的”部分匹配值”为2，因此按照下面的公式算出向后移动的位数：

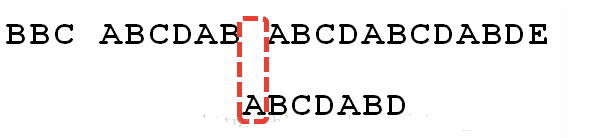
移动位数 = 已匹配的字符数 - 对应的部分匹配值

因为 6 - 2 等于4，所以将搜索词向后移动 4 位。

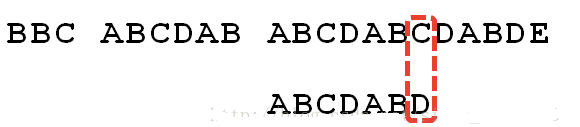
1. 因为空格与Ｃ不匹配，搜索词还要继续往后移。这时，已匹配的字符数为2（”AB”），对应的”部分匹配值”为0。所以，移动位数 = 2 - 0，结果为 2，于是将搜索词向后移 2 位。



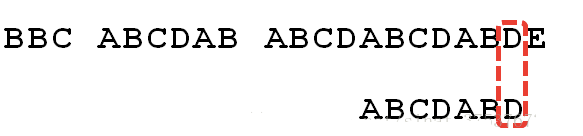
1. 因为空格与A不匹配，继续后移一位。



1. 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。

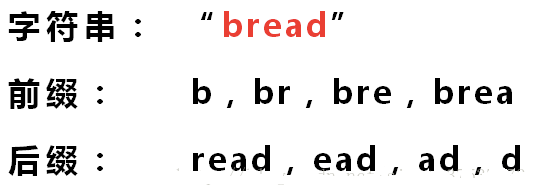


1. 逐位比较，直到搜索词的最后一位，发现完全匹配，于是搜索完成。如果还要继续搜索（即找出全部匹配），移动位数 = 7 - 0，再将搜索词向后移动 7 位，这里就不再重复了。



1. 介绍《部分匹配表》怎么产生的

先介绍前缀，后缀是什么



“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。

1. ”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。



代码 背住即可

**private static int** kmpMatch(String str1,String str2){  
 **int**[] part = *partMatchKMP*(str2);  
 **for** (**int** i = **0**,j=**0**; i < str1.length(); i++) {  
 //核心点  
 **while** (j>**0** && str1.charAt(i) != str2.charAt(j)){  
 j = part[j-**1**];  
 }  
 **if** (str1.charAt(i) == str2.charAt(j)){  
 j++;  
 }  
 //说明找到了  
 **if** (j == str2.length()){  
 **return** i-j+**1**;  
 }  
 }  
   
 **return** -**1**;  
}

**public static int**[] partMatchKMP(String dest){  
 **int**[] part = **new int**[dest.length()];  
 //如果字符串长度为1 则部分匹配值为0  
 part[**0**] = **0**;  
 //aac  
 **for** (**int** i=**1**,j=**0**;i<dest.length();i++){  
 //dest.charAt(i) != dest.charAt(j)  
 //需要从next[j-1]获取新的j  
 //dest.charAt(i) == dest.charAt(j) 成立时退出  
 **while** (j>**0** && dest.charAt(i) != dest.charAt(j)){  
 j = part[j-**1**];  
 }  
 //当满足相等时 部分匹配值+1  
 **if** (dest.charAt(i) == dest.charAt(j)){  
 j++;  
 }  
 part[i] = j;  
 }  
 **return** part;  
}

## 贪心算法

应用举例

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" |

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;

思路分析：

1. 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2ⁿ -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2ⁿ | 需要的时间 |
| 5 | 32 | 3.2秒 |
| 10 | 1024 | 102.4秒 |
| 32 | 4294967296 | 13.6年 |
| 100 | 1.26\*100³º | 4x10²³年 |

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，但没有关系）
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3步直到覆盖了全部的地区

代码实现

**public static void** greedy() {  
 //记录电台  
 Map<String, Set<String>> broadcasts = **new** HashMap<>();  
 //各电台覆盖的范围  
 Set<String> k1 = **new** HashSet<>();  
 ...  
   
 //记录所有需要覆盖的地区 随着计算是会变动的  
 Set<String> allAreas = **new** HashSet<>();  
  
// System.out.println(allAreas.toString());  
 //存放选择的电台的集合  
 List<String> select = **new** ArrayList<>();  
   
 //定义一个临时集合 在遍历过程中 存放被遍历过程中电台覆盖度的地区和未覆盖的地区  
 Set<String> tempSet = **new** HashSet<>();  
   
 //保存在一次遍历中 能够覆盖最多尚未覆盖地区对应的key  
 //如果maxKey 不为空就加入到select  
 String maxKey;  
   
 //allAreas.size() > 0 就说明还没有完全覆盖  
 **while** (allAreas.size() > **0**) {  
 //每进行一次 maxKey置空  
 maxKey = **null**;  
   
 **for** (String key:broadcasts.keySet()){  
 tempSet.clear();  
 //当前能够覆盖的区域  
 Set<String> area = broadcasts.get(key);  
 tempSet.addAll(area);  
   
 //求出 tempSet allAreas的交集，交集会被赋给tempSet  
 tempSet.retainAll(allAreas);  
 **if** (tempSet.size() > **0** //当前集合还为存值  
 //货值当前集合包含的未覆盖地区的数量比maxKey指向的集合地区还多  
 //后面这个选择条件体现了贪心算法  
 && (maxKey == **null** || tempSet.size() > broadcasts.get(maxKey).size())){  
 //说明当前key还存在未覆盖地区  
 maxKey = key;  
 }  
 }  
  
 **if** (maxKey != **null**) {  
 select.add(maxKey);  
 allAreas.removeAll(broadcasts.get(maxKey));  
 }  
 }  
 System.*out*.println(select.toString());  
 }