JAVA数据结构P178

目录

[1.导言 2](#_Toc49439265)

[2.队列 3](#_Toc49439266)

[3.链表 3](#_Toc49439267)

[4.栈 (Stack) 4](#_Toc49439268)

[5.递归 6](#_Toc49439269)

[6.排序算法 6](#_Toc49439270)

[(4)希尔排序 8](#_Toc49439271)

[(6)归并排序 9](#_Toc49439272)

[(9)算法比较 11](#_Toc49439273)

[8.查找算法 11](#_Toc49439274)

[9.哈希表 12](#_Toc49439275)

[10.树结构 13](#_Toc49439276)

[一.堆排序 15](#_Toc49439277)

[二.赫夫曼树 15](#_Toc49439278)

[11.二叉排序树 16](#_Toc49439279)

[删除结点： 16](#_Toc49439280)

[一.比较复杂 16](#_Toc49439281)

[二.比较简单 17](#_Toc49439282)

[11.5.平衡二叉树 17](#_Toc49439283)

[12.多路查找树 19](#_Toc49439284)

[13.图论 21](#_Toc49439285)

[14.十大算法 23](#_Toc49439286)

[14.1二分查找算法(非递归) 23](#_Toc49439287)

[14.2分治算法 24](#_Toc49439288)

[14.3动态规划算法 24](#_Toc49439289)

[14.4 KMP算法[字符串匹配] 25](#_Toc49439290)

[14.5贪心算法 27](#_Toc49439291)

[14.6普利姆算法 27](#_Toc49439292)

[最小生成树 27](#_Toc49439293)

[14.7克鲁斯卡尔算法 28](#_Toc49439294)

[14.8 Dijkstra算法 29](#_Toc49439295)

[14.9 Floyd算法 29](#_Toc49439296)

[14.10马踏棋盘算法 30](#_Toc49439297)

# 1.导言

1.树 图论 排序 程序=数据结构+算法 算法是灵魂

马踏棋盘算法 KMP算法 二分查找 好玩

2.思路分析 代码实现 测试程序 -->实际遇到的问题 如何解决？

3.单链表/ 二维数组/稀疏数组/约瑟夫（单向环形链表）/普利姆算法/分治算法/回溯法/树

（快速排序/堆排序）较高效

4.线性结构/非线性结构

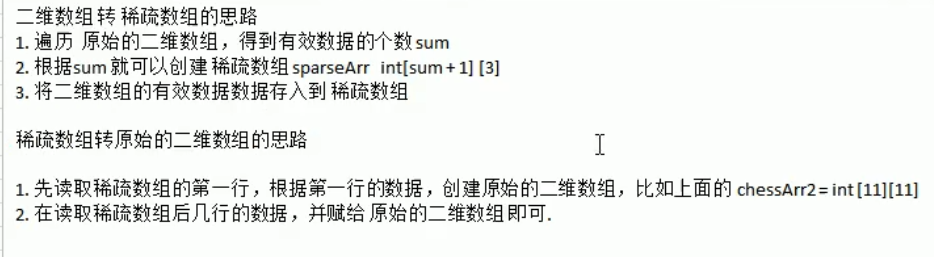
线性： 1对1（顺序/链式） 链表元素不一定连续

【数组，队列，链表，栈】

非线性： 二维数组/多维数组/广义表/树结构/图结构

5.**稀疏数组**：当一个数组中大部分的元素为0或者为同一值时，可以它来保存该数组

 [0]元素:记录 行/列/多少个值

先压缩 再转二维数组 

# 2.队列

（银行排号系统 如：4窗口）/列车进站出站

先入先出 后存入.要后取出

【可用 数组/链表 模拟】

0.1.2.3 令数组[4] maxsize=3 [3]是预留的空间

思路如下: (入队rear+1,出队front+1) rear队尾 front队头

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.

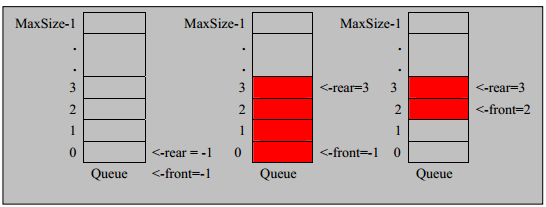
rear 的初始值 = 0

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 = 3 front = 1 有效2



6. 我们就可以在原来的队列上修改得到，一个环形队列. (循环队列)

# 3.链表

1.节点存储，不一定是连续存储（data/next）

2.带头节点/无头结点 的链表 （类似C语言但无malloc动态分配内存）

一.单向链表【增删改查】

删：⒈两个变量 temp指向删除的，front指向删除的前一个节点

⒉一个temp. temp.next.no==del.no; //temp的下一个节点

temp.next=temp.next.next;

被删除的节点没有引用指向会被垃圾回收机制回收 (Java特性)

一定要判断节点是否为空，链表是否为空（break）

插入：p=head; p=p.next; (直到找到要插入的位置)

Insnode.next=p.next; p.next=Insnode;

二.查找单链表中的倒数第n个节点

1.先遍历全部，记录size 2.从头开始遍历size-n个节点

三.逆序从尾到头打印单链表

1.可利用栈 将各个节点压入栈中，然后利用栈的先进后出的特点

实现了逆序打印的效果 [ Stack ]

2.用链表的头插法将链表的数据装入新的链表，再将新链表的值复制到原链表 输出

四.单链表反转 【有难度】 用新链表(从头结点开始加入节点) [头插法]\*\*\*

五.双向链表

1.next 指向下一个节点. pre指向上一个节点 [默认为NULL]

2.遍历方法和单链表一样，可以向前，也可以向后查找

3.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 = newNode;

(3) newNode.pre = temp;

4.**修改** 思路和 原来的单向链表一样. [先找到再修改]

5.**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

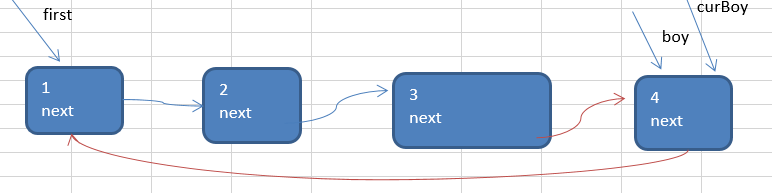
(3) temp.pre.next = temp.next;

(4) if(temp.next!=null){

temp.next.pre = temp.pre;} 如果是最后一个节点不需要写这个

六.单向环形链表（约瑟夫问题）

1.出圈顺序24153 出圈后节点删除

2.

**构建**一个单向的环形链表思路

1. 先创建第一个节点, 让 first 指向该节点，并形成环形

2. 后面当我们每创建一个新的节点，就把该节点，加入到已有的环形链表中即可.

**遍历**环形链表

1. 先让一个辅助指针(变量) curBoy，指向first节点

2. 然后通过一个while循环遍历该环形链表即可 curBoy.next == first结束

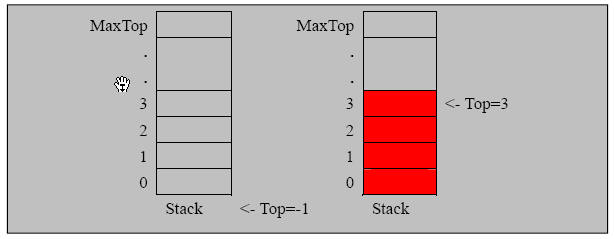
# 4.栈 (Stack)

两个栈实现计算器运算 /二叉树的遍历 /处理递归调用

1.栈是一个先入后出的有序列表。

2.栈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)。

3.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



(用数组模拟)

**一**.逆波兰计算器 [只用一个栈实现]

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，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)×5-6对应的**后缀表达式**就是3 4+5×6- ,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7×5=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(留在栈最后的一个数)

:需要手动将中缀表达式转为后缀表达式 从左向右扫描 还要注意哪个在前35-6

二.中缀表达式转换为后缀表达式 (复杂)

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(消除括号)

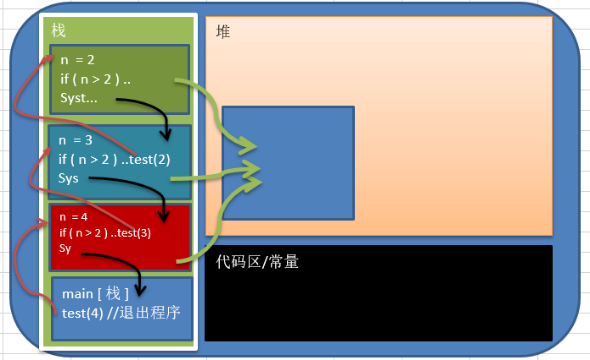
6)重复步骤2至5，直到表达式的最右边

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

注:s2也可用队列实现，就不需要逆序输出

# 5.递归

递归调用规则：

1. 当程序执行到一个方法时，就会开辟一个独立的空间(栈)

2. 每个空间的数据(局部变量)，是独立的.

if (n > 2) {

test(n - 1);

}

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

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

1. 迷宫回溯 （下->右->上->左）设计策略求最短路径

4个判断，再调用自己。

1. 八皇后问题

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)

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

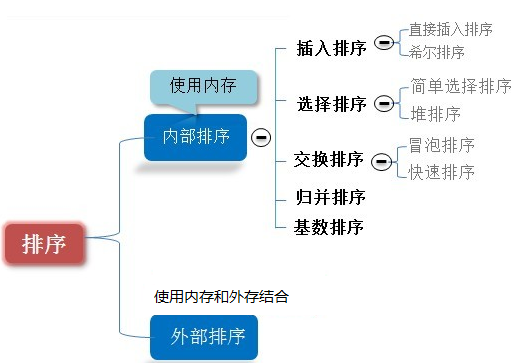
4)当得到一个正确解时，在栈回退到上一个栈时，就会开始回溯，即将第一个皇后，放到第一列的所有正确解，全部得到.

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

得到的结果就是第一行的皇后从0到7的位置都试过,最后判断(judge)的函数调用了1.5w次

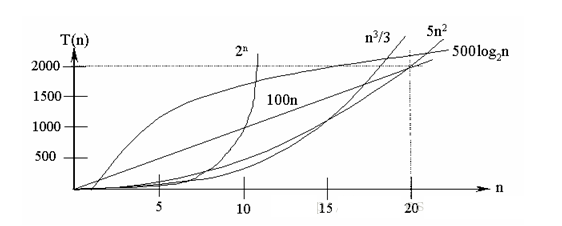
# 6.排序算法

8大排序算法



**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

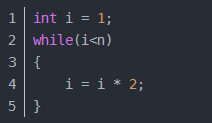
常见的算法时间复杂度由小到大依次为：

**Ο(1)＜Ο(log*2n*)＜Ο(n)＜Ο(nlog*2n*)＜Ο(*n*2)＜Ο(*n*3)＜ Ο(*n*k) ＜Ο(*2*n)** ，

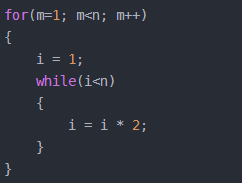
随着问题规模n的不断增大，上述时间复杂度不断增大，算法的执行效率越低

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

对数阶O(**log*2n***)

 2^x=n x为循环的次数，x= **log*2n***

**线性对数阶O(nlogN)**

 在**log*2n*** 的基础上再套个for(n)的循环

**平方阶O(n²)** 两个for(n)循环嵌套

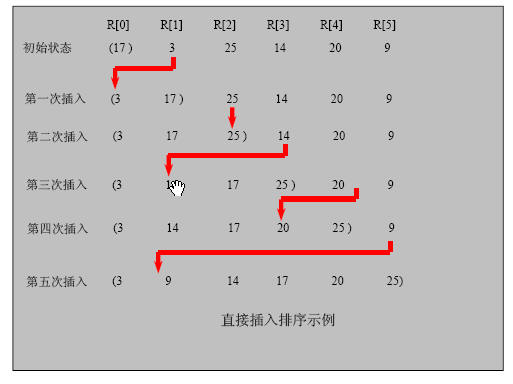


(1)选择排序交换的次数少，比(2)**冒泡排序**快【8w个随机数测试】

选择排序：记录下最小值的下标，每次都把后面最小的值与a[i]交换位置，交换的次数

少，冒泡需要交换的次数多，需要的时间就多，两种都是O(n^2);

(3)插入排序 就像打斗地主牌，将后面的牌插入左边合适的位置，值找位置，找到要插入的位置，在其后的数都要往后移

每次需要移动数组元素的位置

## (4)希尔排序

希尔排序基于插入排序，而插入排序需要大量的移动复制，标记的元素左边都是有序的，而右边都是无序的，要把标记符所指的元素插入左边的有序数据中，就需要将中间数据依次右移一位，这样就会**移动复制太多次**。而我们可不可以通过增量来进行插入排序呢？答案是可以的。我们可以先取一个最大增量作为间隔，把以增量大小为间距的数据划为一组，在组内进行直接插入排序，这样数据就**可以大跨度的移动**，而不是直接排序那样只移动一位（ps：步子小，当然走的慢。。。）。然后不断减小增量，直到增量为1，此时只需要很小的移动就可以完成最终排序。



希尔排序时1)对有序序列在插入时采用交换法,并测试排序速度.(速度很慢)

2)对有序序列在插入时采用移动法，移动法更快速

如12个数第一轮12/2=6 6组2个数比较

6/2=3 3组4个数比较

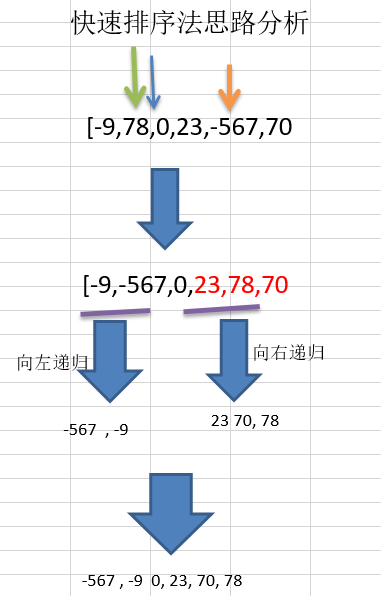
3/2=1 1组12个数比较[这时就跟插入排序类似，但是交换的次数少]

(5)**快速排序**

冒泡的升级 用递归完成，以空间换时间

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。

↓此图是以a[len-1]为基准的，



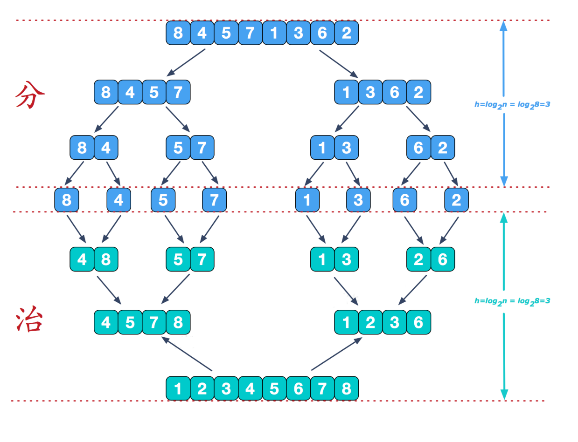
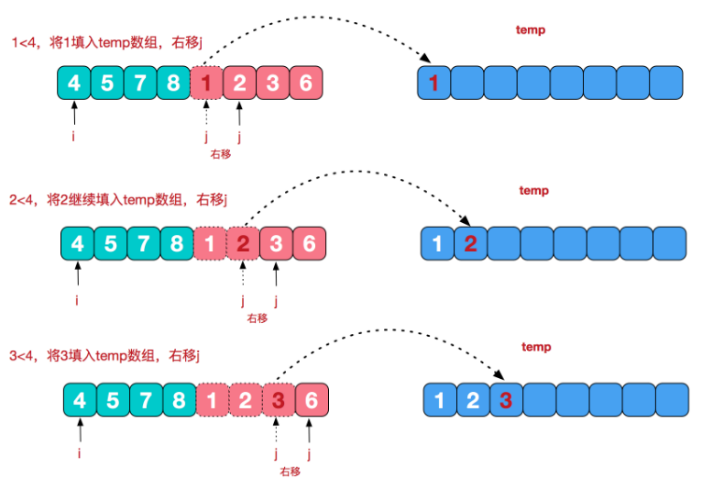
也可以以别的下标为基准:a[(left+right)/2]

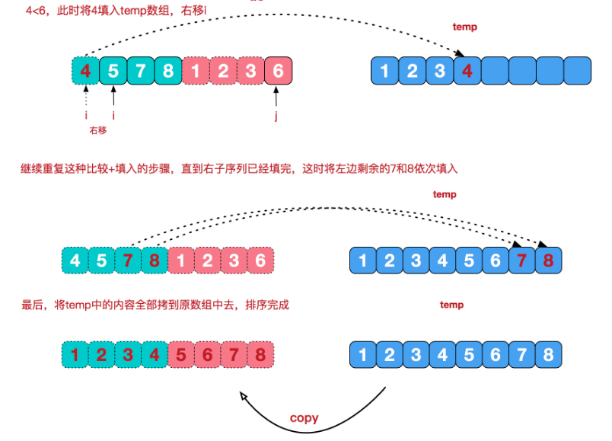
即中间的那个数为基准

以中间为基准就要注意左递归和右递归

## (6)归并排序

分治算法 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。

最后将temp数组 copy到原数组中 [每次]

**合并相邻有序子序列** 左右两个索引来比较,谁小丢谁进去，要注意有一边的索引走完了，但是另一边还有数据时需要将剩余的元素丢进temp中, 用while循环.

用到递归也是空间换时间,速度跟快排差不多

(7)**基数排序**

基数排序是对传统桶排序的扩展，速度很快. 基数排序（radixsort）属于“分配式排序”（distributionsort），又称“桶子法”（bucketsort）或binsort，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 (个->十->百->千->万~~)

1. 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 ，且只能对整数数组排序，小数点不行。
2. 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**]
3. **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**](https://code.i-harness.com/zh-CN/q/e98fa9)

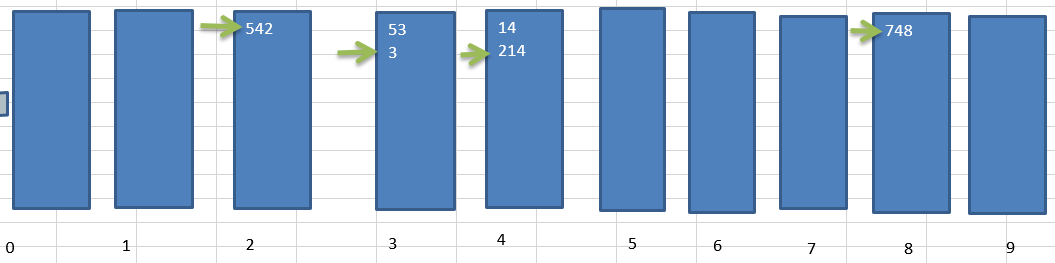
{53, 3, 542, 748, 14, 214}

1. {542 ,53 ,3 ,14 ,214 ,748 } 个

2. {3 ,14 ,214 ,542 ,748 ,53 } 十

3. {3 14 53 214 542 748 } 百 最大的数有多少位就排多少次

分十个桶，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,取到对应的位数后，按位数的值丢进对应的桶中,再将桶内的数据从左往右放入数组中



内存可能不够,当数据量为8000w时已经占了3G内存，就会运行不了,速度是比快排快。

## (9)算法比较



**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

# 8.查找算法

**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

1.线性顺序查找(遍历)，很慢

2.**二分查找** [数组必须为有序]

1). 首先确定该数组的中间的下标

mid = (left + right) / 2

2). 然后让需要查找的数 findVal 和 arr[mid] 比较

2).1 findVal > arr[mid] , 说明你要查找的数在mid 的右边, 因此需要**递归**的向右查找

2).2 findVal < arr[mid], 说明你要查找的数在mid 的左边, 因此需要**递归**的向左查找

2).3 findVal == arr[mid] 说明找到，就返回 / **else**有可能left/right的值就是findVal

//什么时候我们需要结束递归.

(1) 找到就结束递归

(2) 递归完整个数组，仍然没有找到findVal,也需要结束递归 当 left > right 就需要退出

\*\*\*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,这是考虑到数组中如果有多个重复的值要把他们都找到，从找到的下标开始，先从左遍历，再进行右遍历把所有值找到

3.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。[改变mid的计算公式]
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
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])

**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

4.斐波那契查找(复杂)

# 9.哈希表

数组+链表 组成，也叫散列表

HashMap 1.8为数组+链表+红黑树 组成

19 01 23 14 55 68 11 86 37

要存储在表长11的数组中，其中H（key）=key MOD 11 从左往右遍历

产生hash冲突后在存储数据后面加一个指针，指向后面冲突的数据 .next

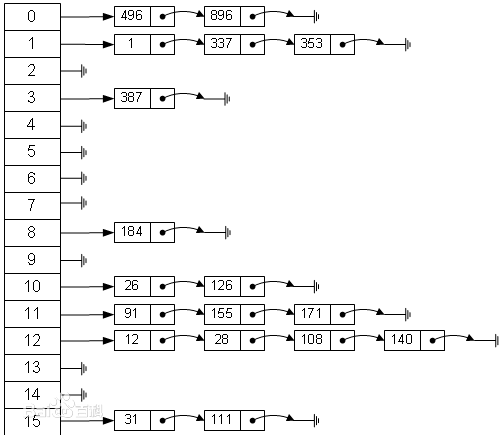
第一列为数组[11]，后面为链表，元素为id

通过找到id的位置，再将id的其他信息找到，或者修改/删除该信息

H（key）=key MOD p （p<=m ,m为表长）,自己找合适的p来减少**冲突**[计算方法可多样]

\*\*随机数法\*\* H（key）=Random（key） 取关键字的随机函数值为它的散列地址

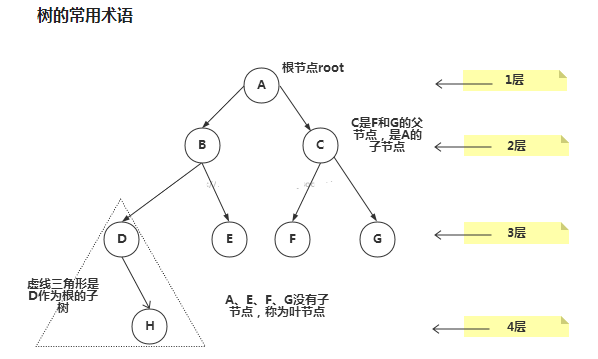
// \*\*平方探测再散列 \*\*线性探测再散列 \*\*随机探测在散列（双探测再散列）\*\*

 p=16 H（key）=key MOD 16

# 10.树结构

**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。

能提高效率 二叉树 红黑树



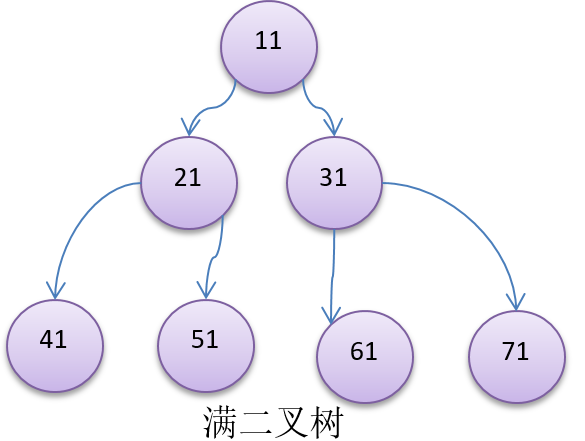
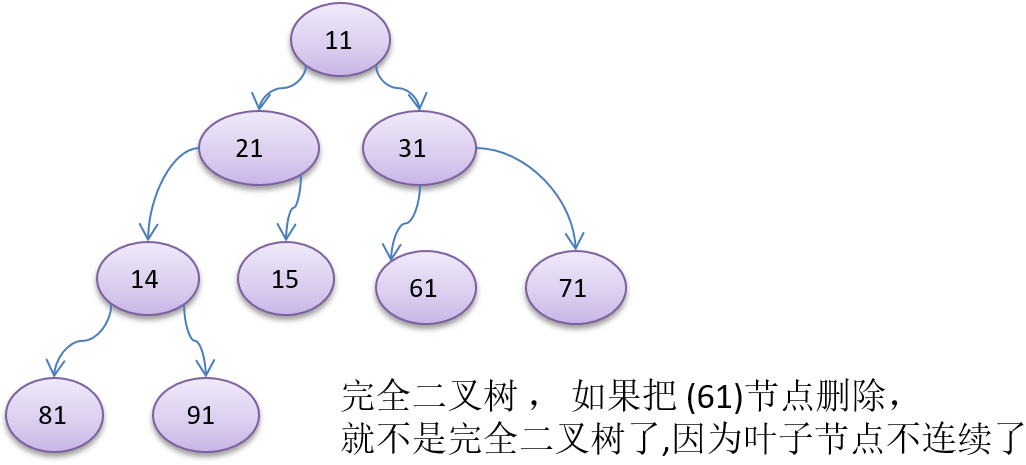
1. 叶子节点 (没有子节点的节点)
2. 节点的权(节点值)
3. 路径(从root节点找到该节点的路线)

4）树的高度(最大层数)

1.二叉树

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。

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。

二叉树遍历：

前序遍历: **先输出父节点**，再遍历左子树和右子树

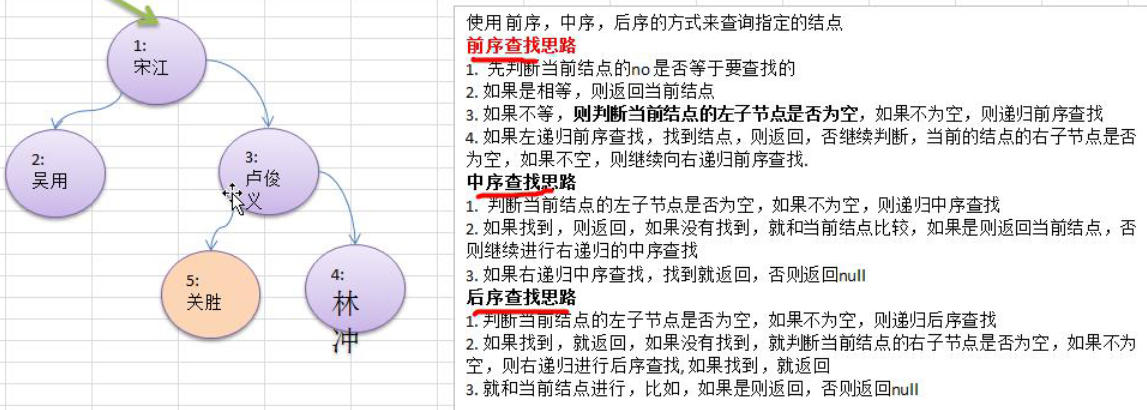
中序遍历: 先遍历左子树，**再输出父节点**，再遍历右子树

后序遍历: 先遍历左子树，再遍历右子树，**最后输出父节点**

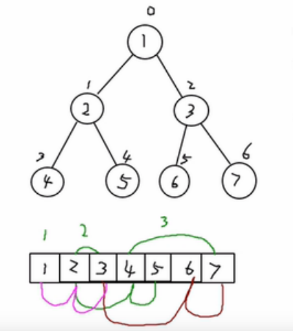
**小结**: 看输出父节点的顺序，就确定是前序，中序还是后序

反复递归调用，直到节点的左右子节点均为空时结束

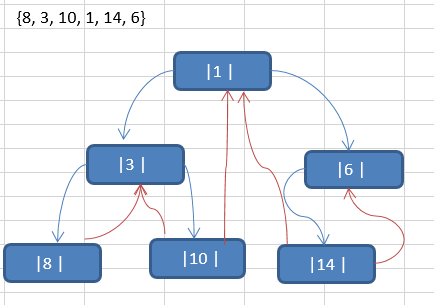
二叉树查找节点:



2.顺序存储二叉树

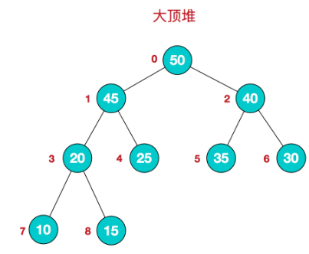
 从数据存储来看，数组存储方式和树的存储方式可以相互转换，即数组可以转换成树，树也可以转换成数组

3.线索化二叉树

 红色线为添加的线，使其线索化

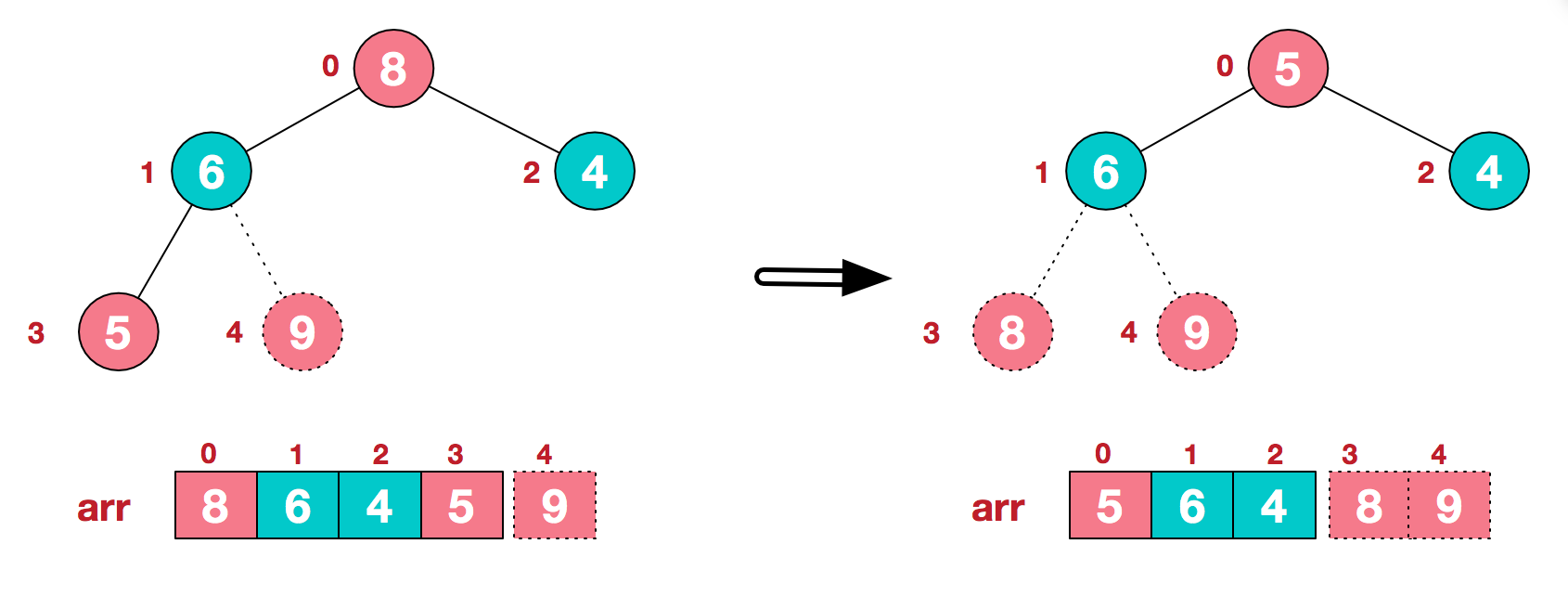
## 一.堆排序

自己写一遍堆排序的代码

 大顶堆:用来升序排序 小顶堆:用来降序排序

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。



每次大顶堆操作将最大值放到根节点，再与数组中末尾元素交换

大顶堆调整:我们从最后一个非叶子结点开始（叶子结点自然不用调整，第一个非叶子结点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“三角形”

将最大的值放在小三角形的顶

堆排序的速度非常快 O(nlogn)

## 二.赫夫曼树

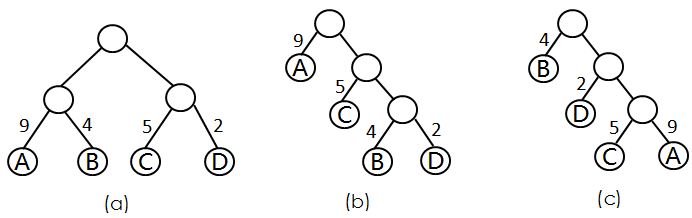
又称作最优二叉树，它是n个带权叶子结点构成的所有二叉树中，带权路径长度最小的二叉树。

“路径”就是从树中的一个结点到另一个结点之间的分支构成的部分，而分支的数目就是路径长度。

树的路径长度：就是从树根到每一结点的路径长度之和。

考虑带权的结点，结点的带权路径长度为：从该结点到树根之间的路径长度与结点上权的乘积。

树的带权路径长度WPL(weighted path length)：树中所有叶子结点的带权路径长度之和。



三棵二叉树的带权路径长度为：

（a）WPL=9x2+4x2+5x2+2x2=18+8+10+4=40

（b）WPL=9x1+5x2+4x3+2x3=9+10+12+6=37

（c）WPL=4x1+2x2+5x3+9x3=4+4+15+27=50

其中（b）所示的二叉树的WPL最小，此树是哈夫曼树。

由上图可知：由n个带权叶子结点所构成的二叉树中，满二叉树或完全二叉树

不一定是最优二叉树。权值越大的结点离根结点越近的二叉树才是最优二叉树。

2、赫夫曼树的构造方法

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

**赫夫曼编码**：了解一下即可，代码实现很复杂

:实现对二进制文件进行压缩，然后可以再解压，解压后与原文件相同 jpg/ppt文件不行

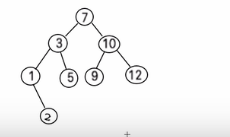
属于无损压缩，如一段英语短文，经过压缩后，比之前的二进制的位数少了很多，压缩率60%左右，且是无损的压缩

# 11.二叉排序树

**二叉排序树**：BST: (Binary Sort(Search) Tree)/二叉查找树, 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

 按照中序遍历出的结果就是排序好的数组

完成增删改查：

增：从根节点开始对比,比node.value ? this.value

小就向左递归直到this.left==null <

大就向右递归直到this.right==null >

查：找到该点(类似遍历,直到找到最后一个节点的左/右子节点为空，此时说明没找到)

改：找到该点将该点的其他信息修改

## **删除结点**：

### 一.比较复杂

[添加一个父指针，node.parent]

还需要在这前提判断他的parent是否为空 两种情况

细分总共2^3=8种 左/右是否为空 /parent是否为空

先找到要删除的结点,再找到target.parent，确定它是parent的左/右子节点

1)叶子节点(left==right==null)

parent.left=null; / parent.right=null;

2)只有一颗子树的节点(left==null或right==null)

①t.left空 将target换成 target.right

parent.left=t.right / parent.right=t.right

②t.right空 将target换成 target.left

parent.left=t.left / parent.right=t.left

3)有两颗子树的节点(left&&right都!=null)

调用一个新函数findMAX /findMIN [递归查找最大/最小的结点]

t.left 左子树去找最大的结点 / t.right 右子树去找最小的结点

用个临时变量temp来保存最小结点的值 temp=11;

删除该最小结点 递归调用del(最小结点);

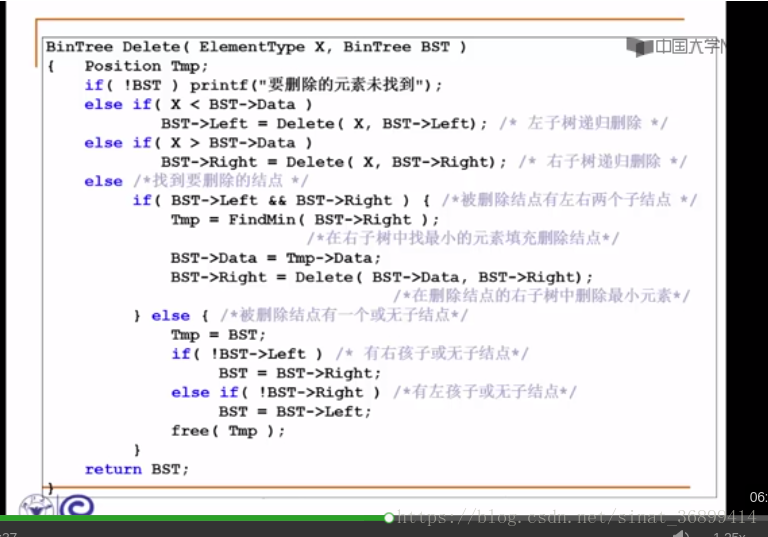
target.value=temp;

注意：要做连续的删除时，每次需要将parent换上新的 或者用个函数来找它的parent (新的结点)

### 二.比较简单

要删除点就分成三种情况1.左子节点为空2.右子节点为空3.左右子节点都不为空

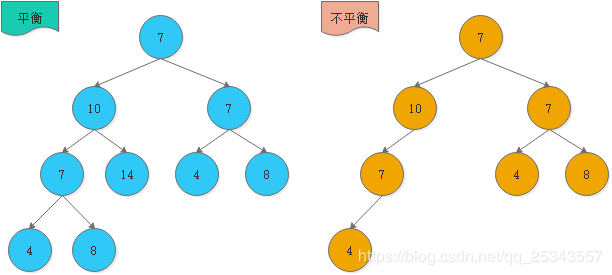
用C语言

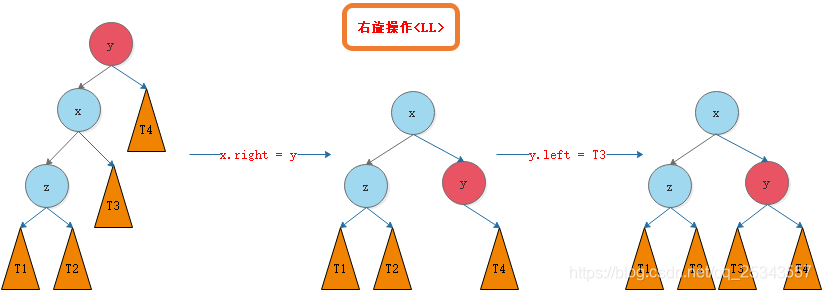
 等待验证

## 11.5.平衡二叉树

平衡二叉树也叫平衡二叉搜索树（Self-balancingbinarysearchtree）又被称为AVL树，可以保证查询效率较高。

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





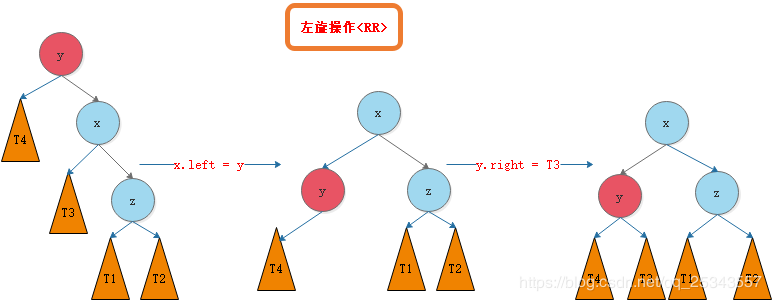
对xy进行大改动 其他小改 将x.right=y; y.left=t3;

Node x = y.left;

Node t3 = x.right;

x.right = y;

y.left = t3; return x;



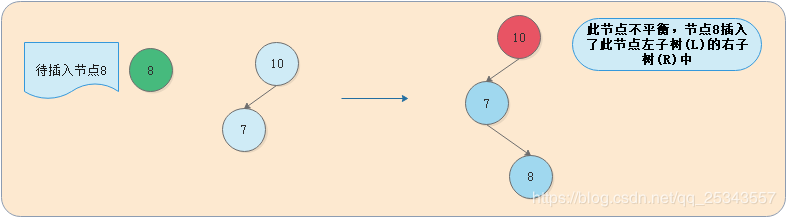
对xy进行大改动 其他小改 将x.left=y; y.right=t3;

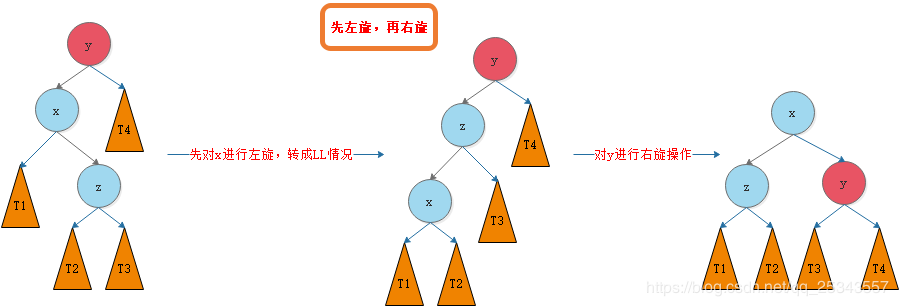
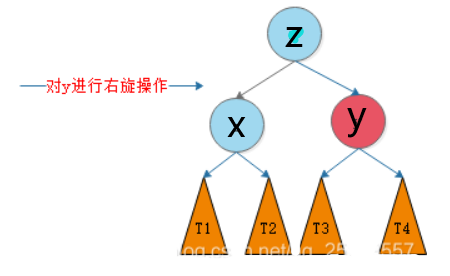
Node x = y.right;

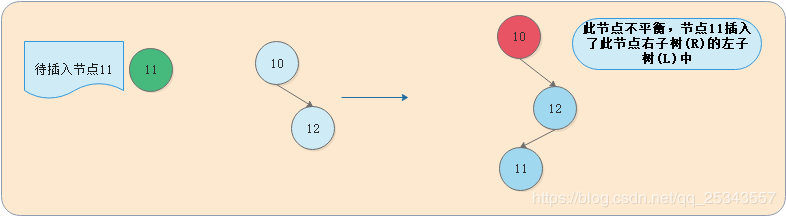
Node t3 = x.left;

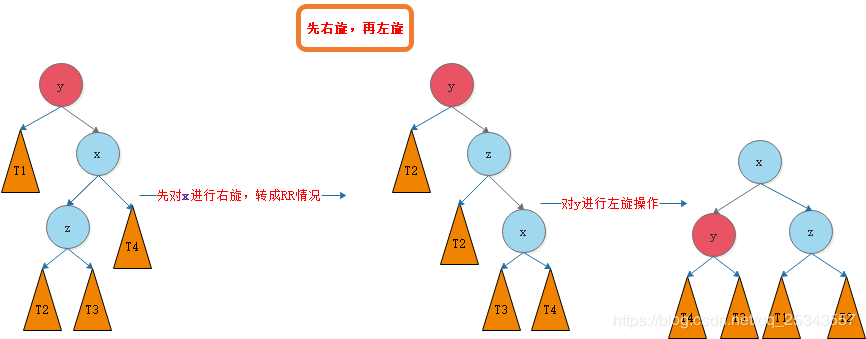
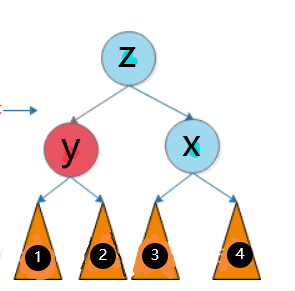
x.left = y;

y.right = t3; return x;

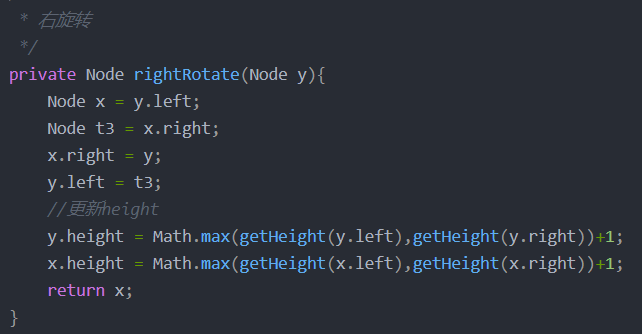
 ①



 ②



代码实现：右旋转



1. 单旋转 左/右旋
2. 双旋转 问题分析出来: 在满足右旋转条件时，要判断
3. (1)如果 是 左子树的 右子树高度 大于左子树的左子树时：
4. (2)就是对当前根节点的左子树 先进行左旋转，
5. (3)然后，再对当前根节点进行右旋转即可
6. 否则，直接对当前节点（根节点）进行右旋转.即可.

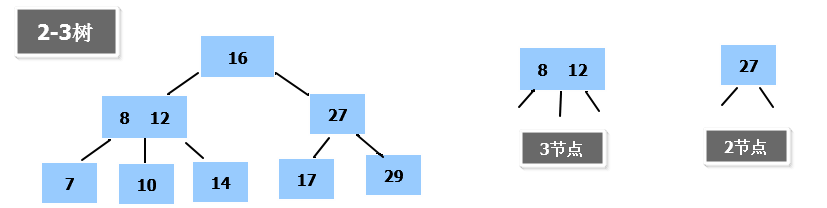
# 12.多路查找树

（2-3树/2-3-4树）B树 B+树 B\*树 代码实现一般不需要

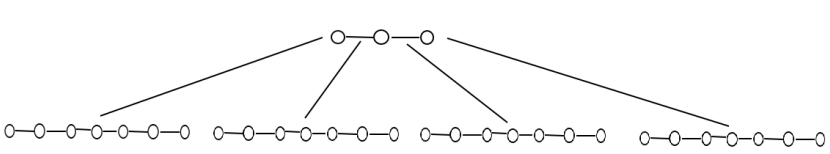
二叉树需要加载到内存的，如果二叉树的节点少，没有什么问题，但是如果二叉树的节点很多(比如1亿)， 就存在如下问题:

**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

**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



B树



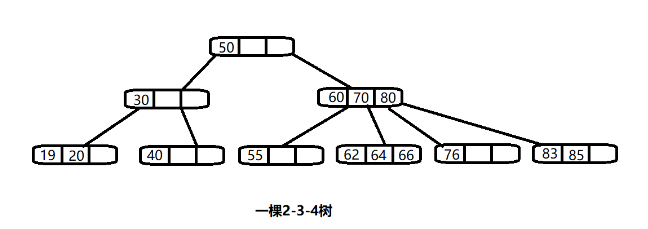
1. **如图B树通过重新组织节点， 降低了树的高度.**
2. 文件系统及数据库系统的设计者利用了磁盘预读原理，将一个节点的大小设为等于一个页**(**页得大小通常为4k**)，这样每个节点只需要一次I/O就可以完全载入**
3. 将树的度M设置为1024，在600亿个元素中最多只需要4次I/O操作就可以读取到想要的元素, B树(B+)广泛应用于文件存储系统以及数据库系统中

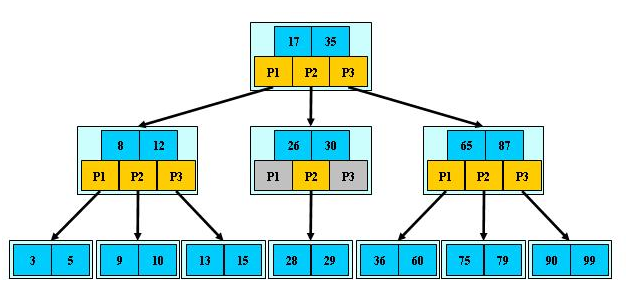
2-3树是最简单的B树结构, 具有如下特点:

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.

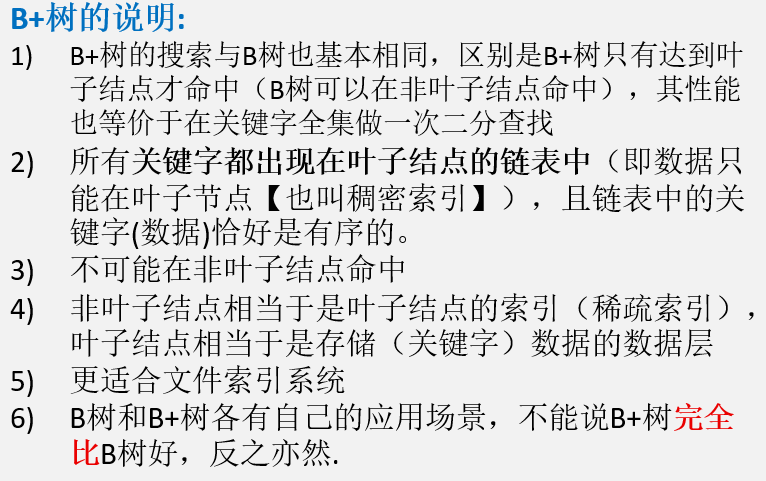
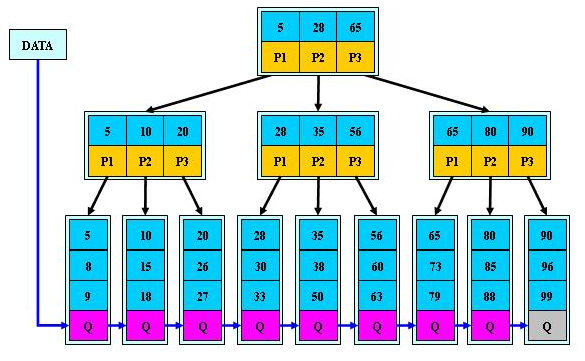
**2-3树是由二节点和三节点构成的树**

1. 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。
2. 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

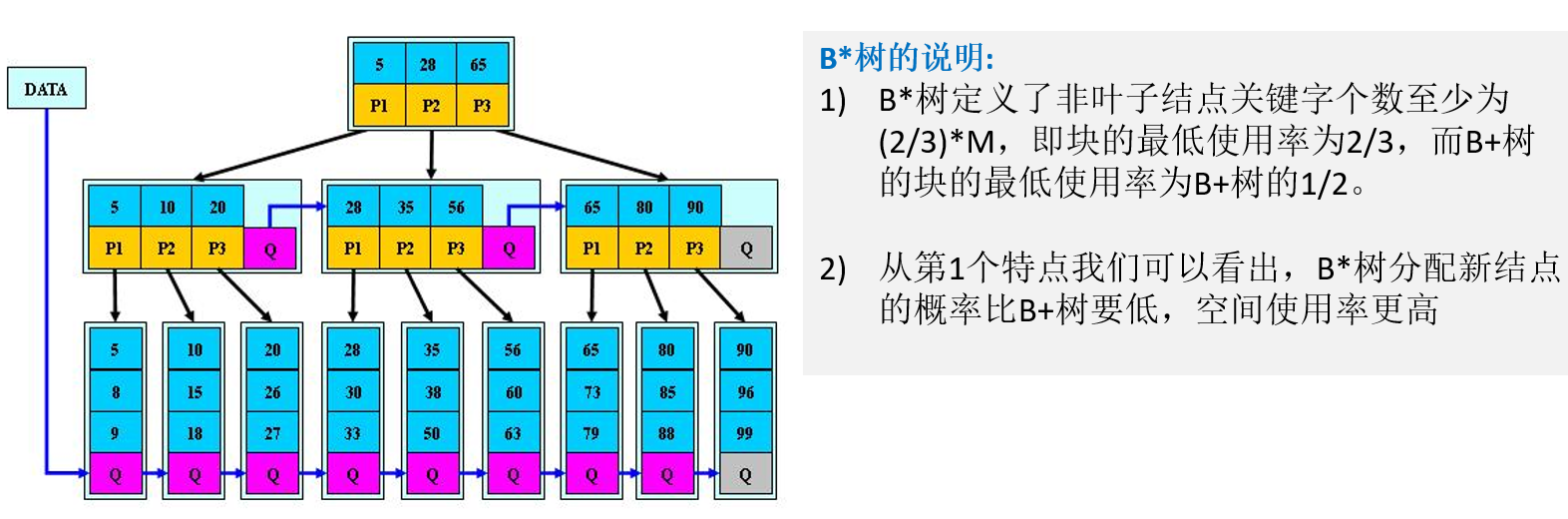
**** 2-3-4树



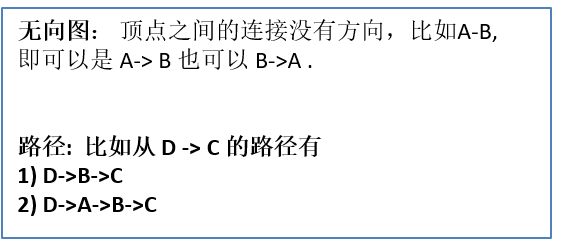
**B+树**是B树的变体，也是一种多路搜索树

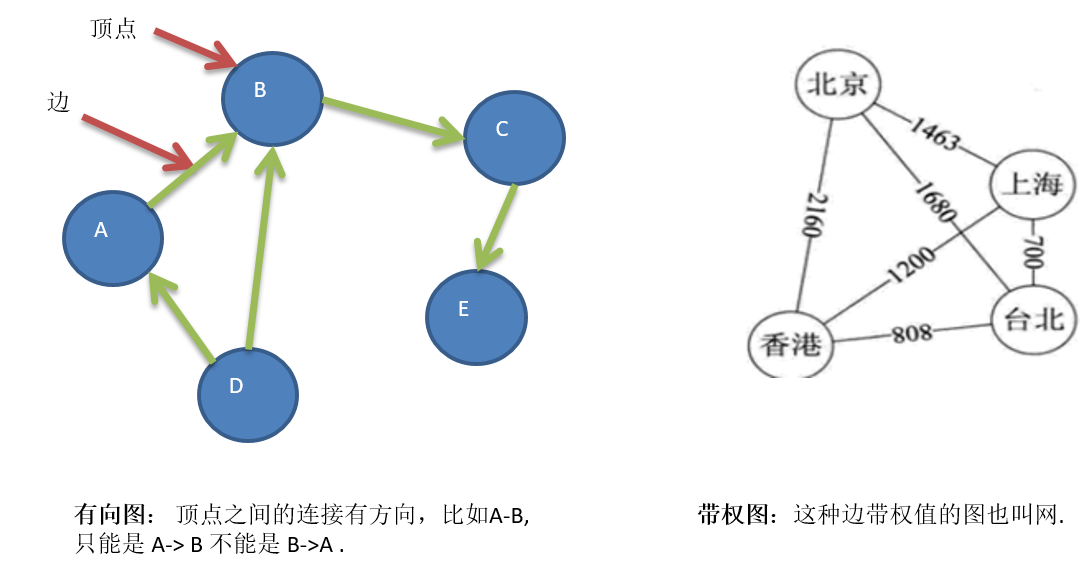


B\*树是B+树的变体，在B+树的非根和非叶子结点再增加指向兄弟的指针



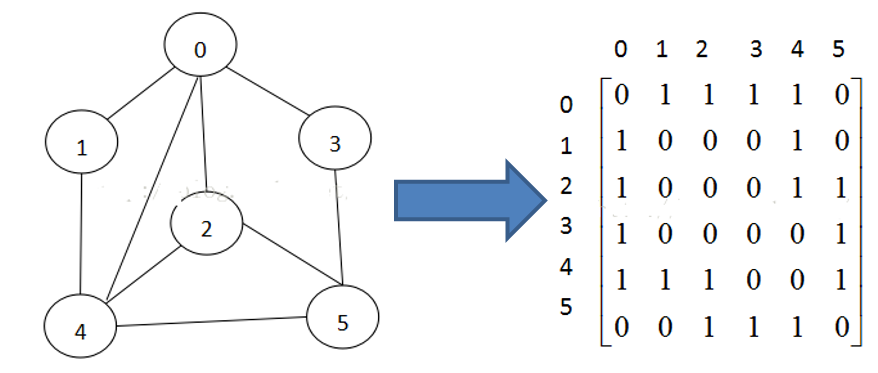
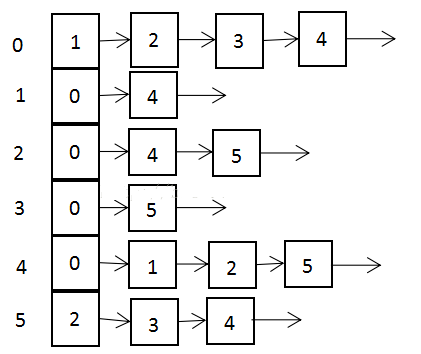
# 13.图论



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）【类似哈希表】数组+链表

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

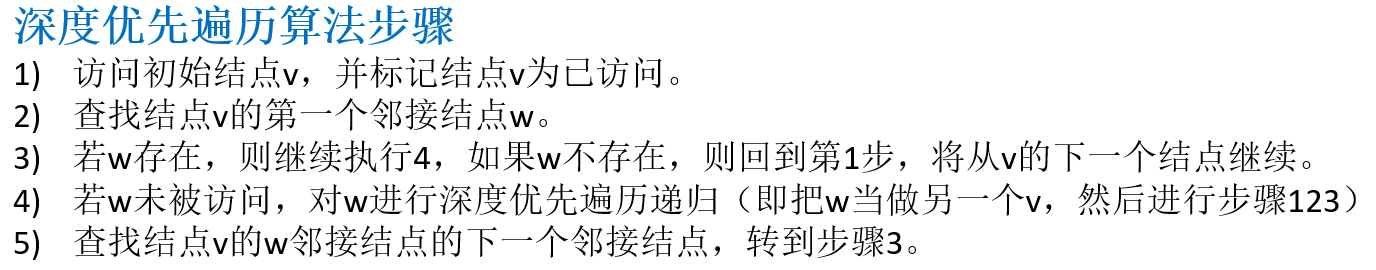
**图遍历介绍**

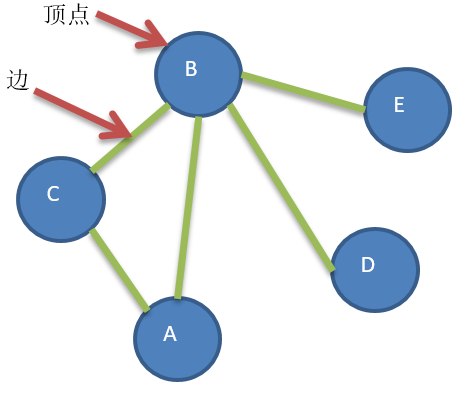
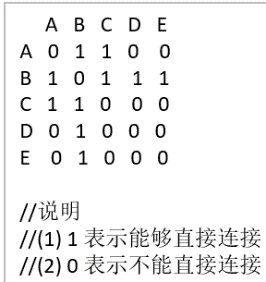
所谓图的遍历，即是对结点的访问。一个图有那么多个结点，如何遍历这些结点，需要特定策略，一般有两种访问策略: (1)深度优先遍历 (2)广度优先遍历

深度优先 纵向的 【往下走】类似前序遍历

广度优先 横向的 检查完第一层再往下检查第二层

1. 深度优先遍历Depth First Search
2. 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**。
3. 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。
4. 显然，深度优先搜索是一个递归的过程。



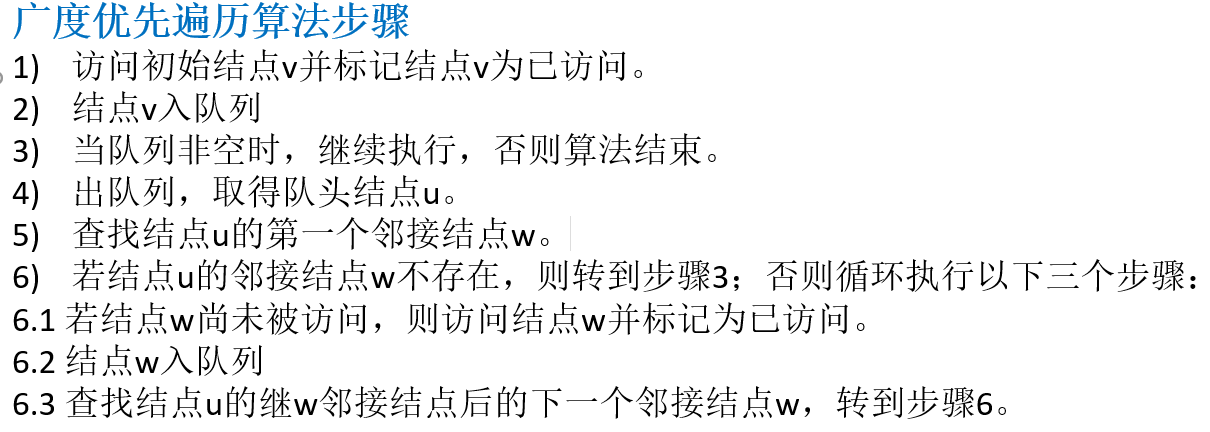


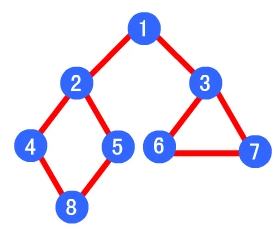
深度：A开始1.AB 2.BA BC 3.CA CB BD 4.DB BE 5.EB

广度：A开始1.AB 2.AC 3.BA BC BD 4. BE

广度优先遍历 (搜索)

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





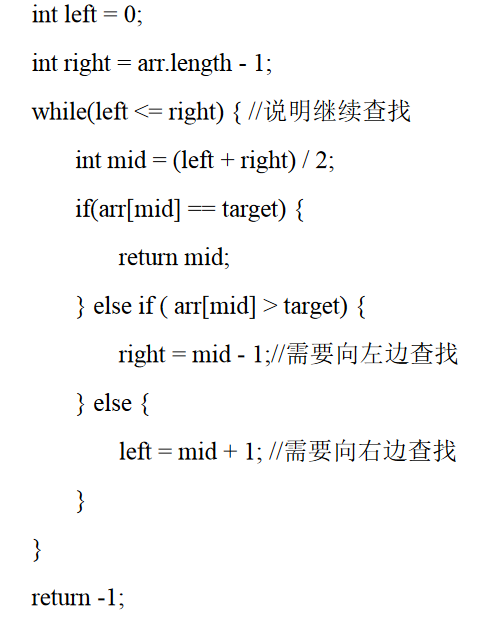
1. 深度优先遍历顺序为 1->2->4->8->5->3->6->7
2. 广度优先算法的遍历顺序为：1->2->3->4->5->6->7->8

# 14.十大算法

## 14.1二分查找算法(非递归)

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

1)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)

 重要的是while的条件 left<=right 当left==right时继续

## 14.2分治算法

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

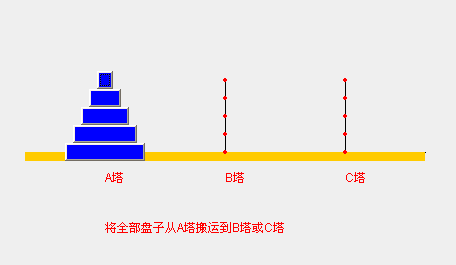
分治法在每一层递归上都有三个步骤：

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

汉诺塔



1. 如果是有一个盘， A->C
2. 如果我们有 n >= 2 情况，我们总是可以看做是两个盘 1.最下边的盘 2. 上面的盘
3. 先把最上面的盘 A->B
4. 把最下边的盘 A->C
5. 再把B塔的所有盘从 B->C 就把复杂的问题分成3步就完成

* 代码都是递归调用

## 14.3动态规划算法



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

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. 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。

* 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]]}

算法了解一下就好，记不住也没事

## 14.4 KMP算法[字符串匹配]

引：大多数人都是暴力匹配两个字符串 效率低

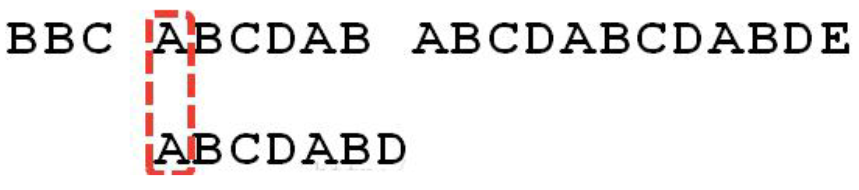
字符串匹配问题：：

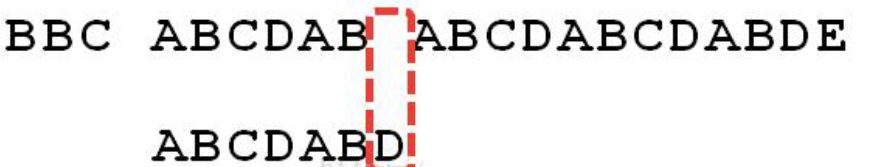
1)有一个字符串str1=""硅硅谷尚硅谷你尚硅尚硅谷你尚硅谷你尚硅你好""，和一个子串str2="尚硅谷你尚硅你"

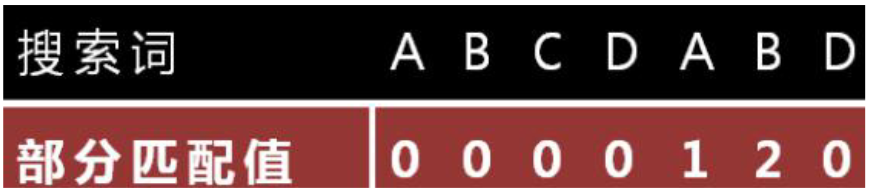
2)现在要判断str1是否含有str2,如果存在，就返回第一次出现的位置,如果没有，则返回-1

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

有一个字符串str1="BBCABCDABABCDABCDABDE"，和一个子串str2="ABCDABD"





 上即为str2 下为出现重复的次数

A1 B2 D0 该str2中出现重复的字符开始记 遇到不是连续就为0

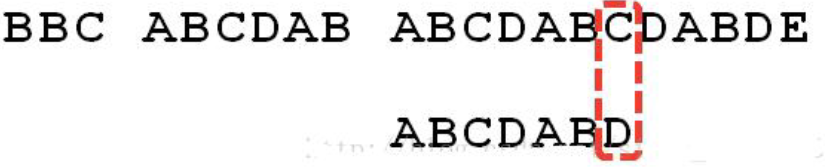
.已知空格与D不匹配时，前面六个字符”ABCDAB”是匹配的。查表可知，最后一个匹配字符B对应的”部分匹配值”为2，因此按照下面的公式算出向后移动的位数：

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

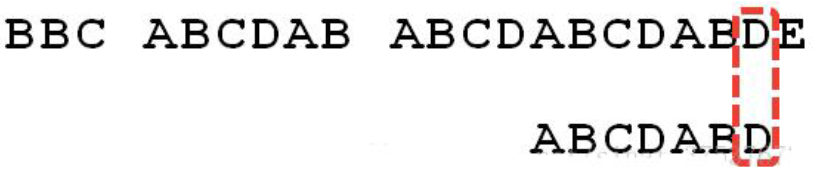


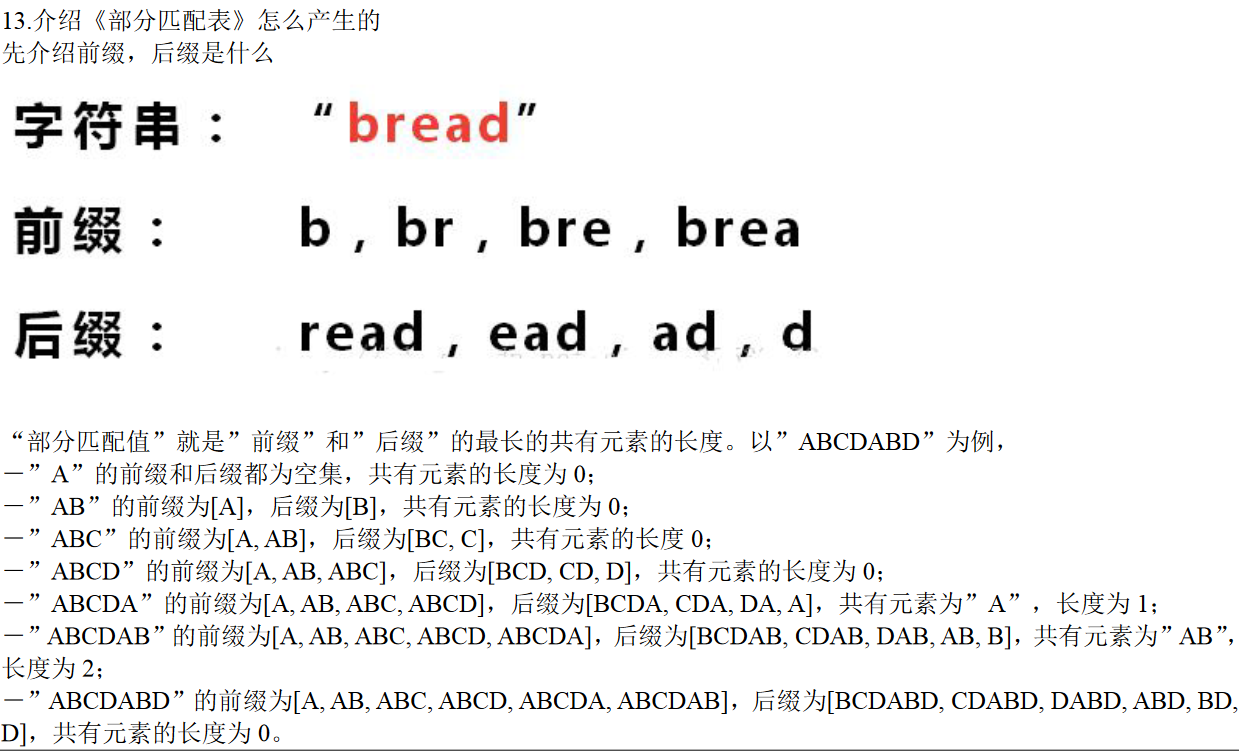
.因为空格与Ｃ不匹配，搜索词还要继续往后移。这时，已匹配的字符数为2（”AB”），对应的”部分匹配值”为0。所以，移动位数=2-0，结果为2，于是将搜索词向后移2位。

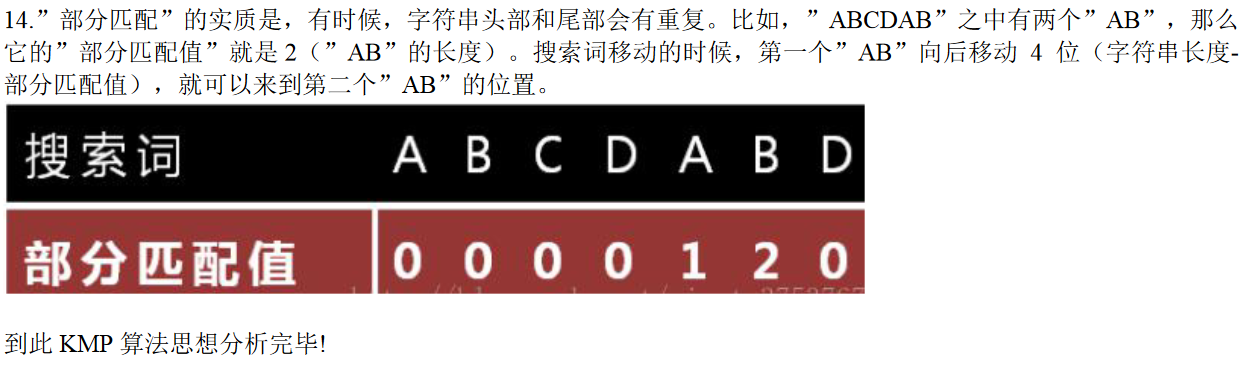




.逐位比较，直到发现C与D不匹配。于是，移动位数=6-2，继续将搜索词向后移动4位。





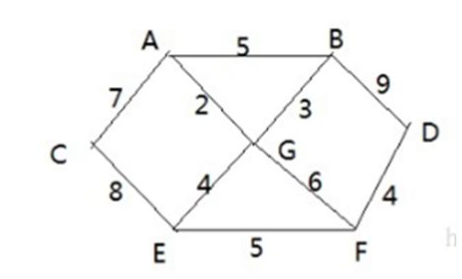


KMP在大量数据或str2中有重复字符时会很高效。

## 14.5贪心算法

## 14.6普利姆算法

应用场景-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

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

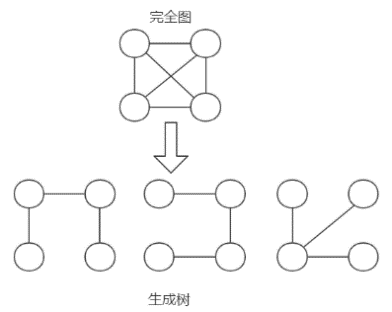
### 最小生成树

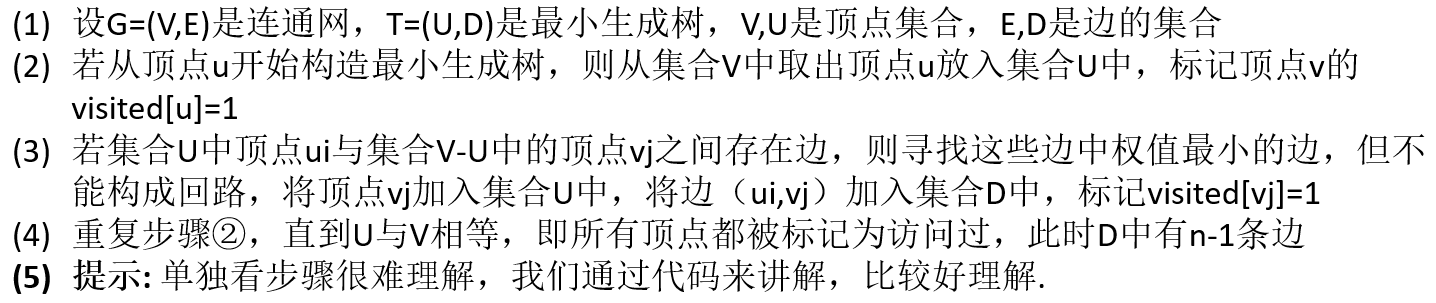
修路问题本质就是就是最小生成树问题，先介绍一下最小生成树(MinimumCostSpanningTree)，简称MST。给定一个带权的无向连通图,如何选取一棵生成树,使树上所有边上权的总和为最小,这叫最小生成树

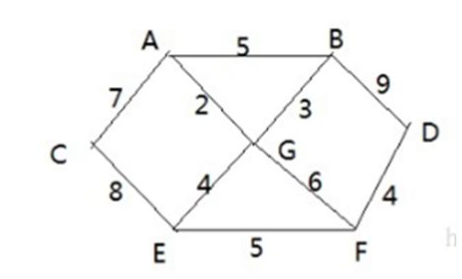
1)N个顶点，一定有N-1条边

2)包含全部顶点

3)N-1条边都在图中







1.从<A>顶点开始处理===> <A,G>2

A-C[7] A-G[2] A-B[5]

2.<AG>开始，将A和G顶点和他们相邻的还没有访问的顶点进行处理=><A,G,B> GB3

A-C[7] A-B[5] G-B[3] G-E[4] G-F[6]

3.<A,6.B>开始，将A,G,B顶点和他们相邻的还没有访问的顶点进行处理=><A,G,B,E> GE4

A-C[7] G-E[4] G-F[6] B-D[9]

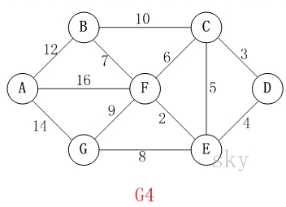
4.{A,G,B,E}->F //第4次大循环，对应边<E,F>权值:5

5.{A,G,B,E,F}->D//第5次大循环，对应边<F,D>权值:4

6.{A,G,B,E,F,D}->C//第6次大循环，对应边<A,C>权值:7 =><A,G,B,E,F,D,C>

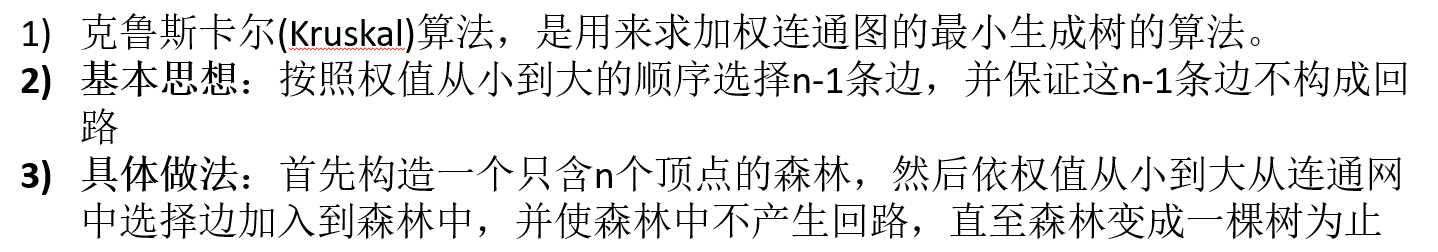
每次循环都找到权重最小的边 最后n-1次循环 找到n-1条边

## 14.7克鲁斯卡尔算法

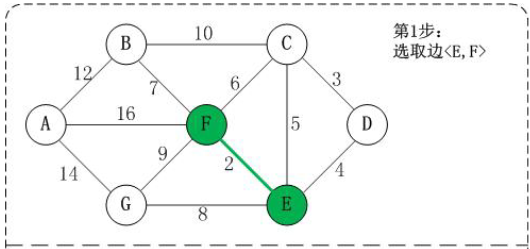
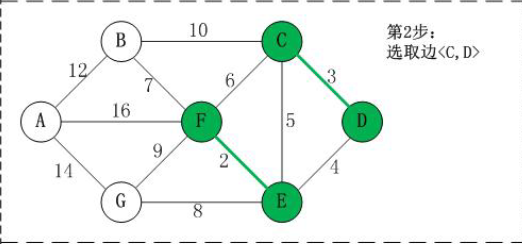


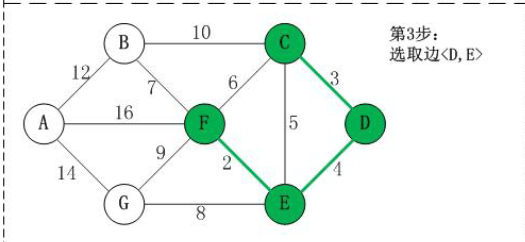
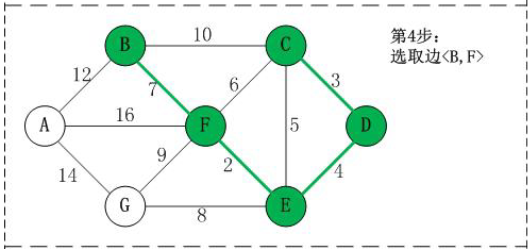
1. 有北京有新增7个站点(A, B, C, D, E, F, G) ，现在需要修路把7个站点连通
2. 各个站点的距离用边线表示(权) ，比如 A – B 距离 12公里
3. 问：如何修路保证各个站点都能连通，并且总的修建公路总里程最短?

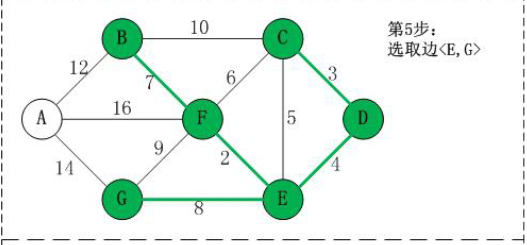
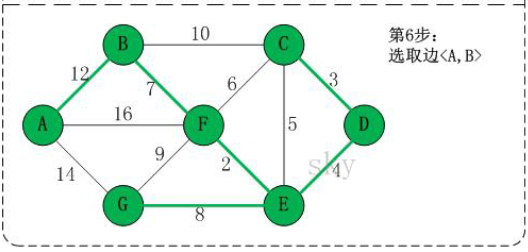
**Kruskal**算法

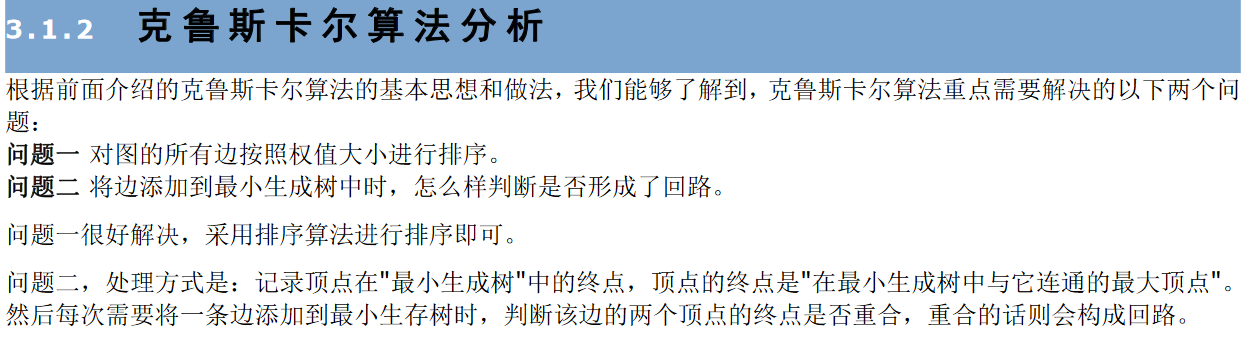


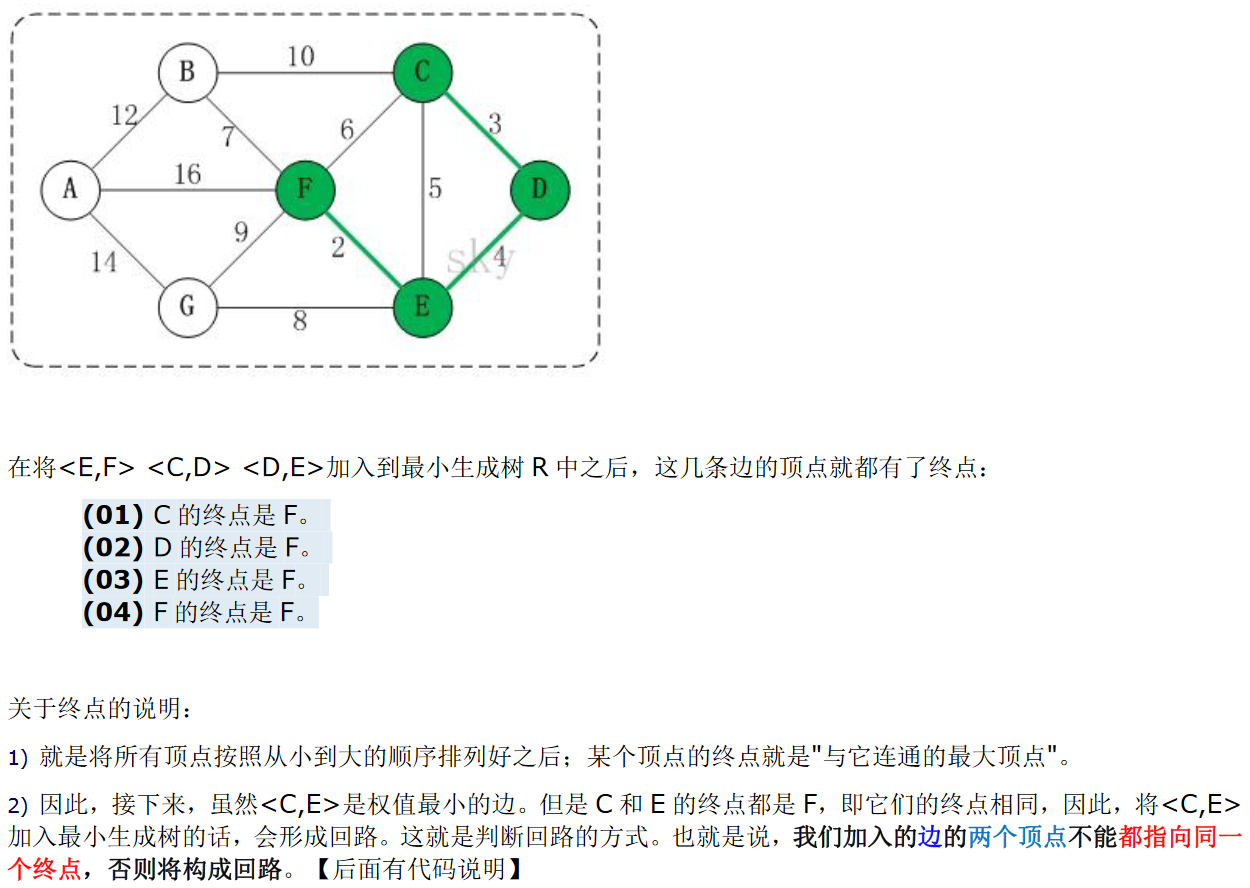
将所有的权都加入一个数组中排序 从最小的开始连，连的线不能产生回路，直到连完所有点。





## 14.8 Dijkstra算法

## 14.9 Floyd算法

## 14.10马踏棋盘算法