传智播客 – C++学院

高级数据结构

# 图(Graph)

## 1.1图的相关概念

### 1.1.1图的定义和术语

**图的定义：图是由顶点的有穷非空集合和顶点之间的边的集合组成，通常表示为：G = (V,E)，其中，G表示一个图，V是图G中顶点的集合，E是图G中边的集合。**

**无向边**：若顶点Vi到Vj的边没有方向，则称这条边为无向边，用无序偶对

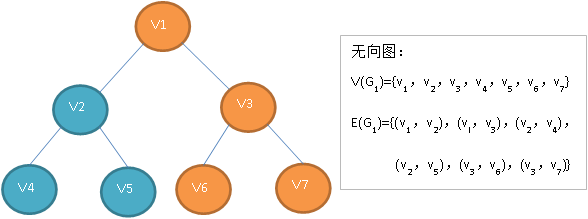
（Vi ，Vj）来表示。

**有向边**：若从顶点Vi 到Vj的边有方向，则称这条边为有向边，也称为弧(Arc)。用有序偶对<Vi ，Vj>来表示。Vi称为弧尾(Tail)或初始点，Vj称为弧头(Head)或终端点。

**注意: < Vi ，Vj >和< Vj ，Vi >是两条不同的有向边。**

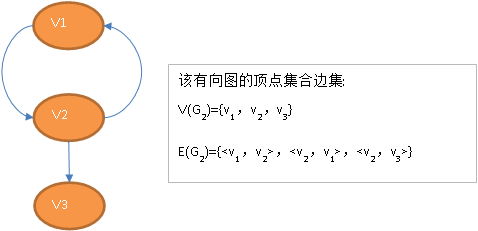
**无向图**: 如果图中任意两个顶点之间的边都是无向边，则称该图为无向图。

图例(G1):



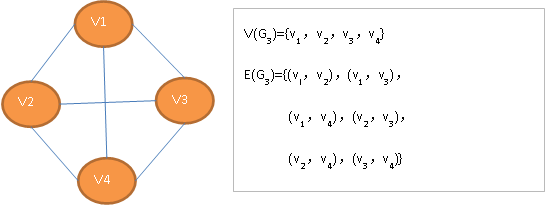
**有向图:** 如果图中任意顶点之间的边都是有向边，则称该图为有向图。

图例(G2):

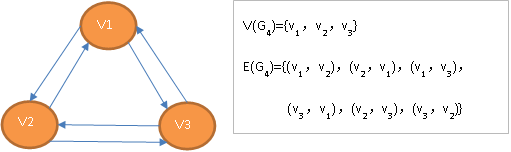


**无向完全图：**在无向图中，如果任意两个顶点之间都存在边，则称该图为无向完全图。含有n个顶点的无向完全图有**n(n-1)/2**条边。

图例(G3)



**有向完全图：**在有向图中，如果任意两个顶点之间都存在方向互为相反的两条弧，则称该图为有向完全图。含有n个顶点的有向完全图有**n(n-1)**条边。



**总结: 图G的顶点数n和边数e的关系**

**（1）若G是无向图，则0≤e≤n(n-1)/2**

**恰有n(n-1)/2条边的无向图称无向完全图(Undireet-ed Complete Graph)**

**（2）若G是有向图，则0≤e≤n(n-1)。**

**恰有n(n-1)条边的有向图称为有向完全图(Directed Complete Graph)。**

**注意：**

**完全图具有最多的边数。任意一对顶点间均有边相连。**

**稀疏图:** 有很少条边或弧的图。

**稠密图:** 有很多条边或弧的图。

**权:** 有时图的边或弧具有与它相关的数,这种**与图的边或弧相关的数**叫做权。

**网：带权的图**通常称为网。

**度：**顶点的度是指**和该顶点关联的边的数目**。

**入度：**有向图中以顶点（v）为头的弧的数目，称为（v）的入度。

**出度：**有向图中以顶点（v）为尾的弧的数目，称为（v）的出度。

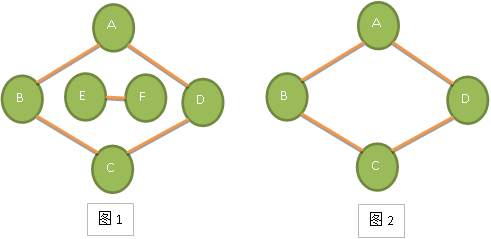
**邻接点**：对于无向图，同一边上的两个顶点称为邻接点。

**子图:** 假设两个图G=(V,E)和G1=(V1,E1),如果V1⊆V且E1⊆E则G1为G的子图

**路径的长度:** 路径上的边或弧的数目。

### 1.1.2连通图相关术语

在无向图G=(V,E)中，如果从顶点v到顶点w有路径，则称v和w是相通的。**如果对图中任意两个顶点Vi和Vj 属于E，则两个顶点是连通的**，则称G是连通图。如下图1，它的顶点A都顶点B、C、D都是连通的，但显然顶点A与顶点E或F就无路径，因此不能算是连通图。而图2，顶点A、B、C、D相互都是连通的，所以它本身是连通图**。**

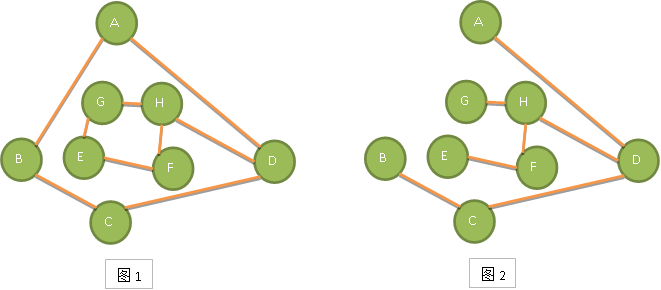
****

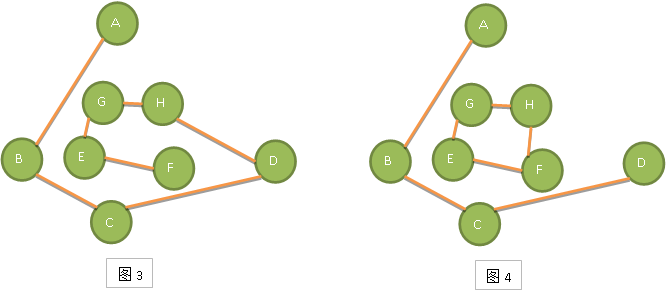
#### 连通图生成树

连通图的生成树**是一个极小的连通子图**，**它含有图中全部的n个顶点，但只有足以构成一棵树的n-1条边。**

**极小连通子图是相对于连通图来说的。**

比如下图的图1是一个普通图，但显然它不是生成树，当去掉两条构成环的边后，比如图2或图3，就满足n个顶点n-1条边且连通的定义了。它们都是一棵生成树。从这里也知道，如果一个图有n个顶点和小于n-1条边，则是非连通图，如果它多于n-1条边，必定构成一个环，因为这条边使得它依附的那两个顶点之间有了第二条路径。比如图2和图3，随便加哪两顶点的边都将构成环。**不过有n-1条边并不一定是生成树**，比如图4。





### 1.1.3总结

**图**按照**有无方向**分为**无向图**和**有向图**。

无向图由**顶点**和**边**构成。

有向图由**顶点**和**弧**构成，弧有**弧头**和**弧尾**之分。

图按照**边或弧的多少**分为**稀疏图**和**稠密图**。

如果**任意两个顶点之间都存在边**叫**完全图**，有向的叫有向完全图。

与顶点相关联的边的条数叫做度，有向图顶点分为入度和出度。  
图上的边或弧带权则称为网。

无向图中连通且n个顶点n-1条边叫生成树。

## 1.2、图的存储结构

### 1.2.1邻接矩阵

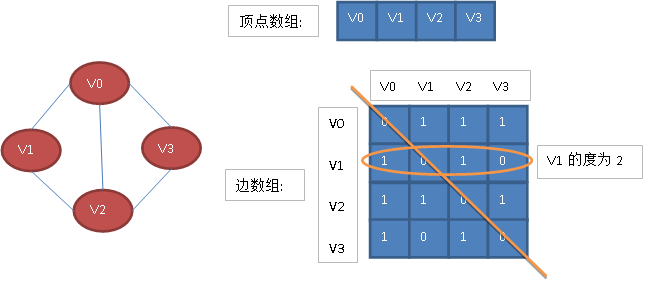
图的邻接矩阵存储方式是**用两个数组来表示图**。

* **一个一维数组存储图中顶点信息，**
* **一个二维数组（邻接矩阵）存储图中的边或弧的信息。**

设图G有n个顶点，则邻接矩阵是一个n\*n的方阵，定义为：

****

看一个实例，下图左就是一个**无向图**。



从上面可以看出，**无向图的边数组是一个对称矩阵**。所谓对称矩阵就是n阶矩阵的元满足aij= aji。即从矩阵的左上角到右下角的主对角线为轴，右上角的元和左下角相对应的元全都是相等的。

    从这个矩阵中，很容易知道图中的信息。

    （1）**判断任意两顶点是否有边无边；**

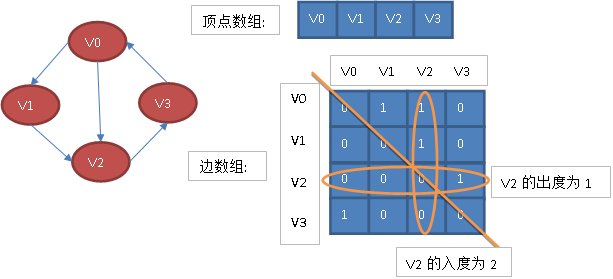
（2）**某个顶点的度，其实就是这个顶点vi在邻接矩阵中第i行或**

**（第i列）的元素之和；**

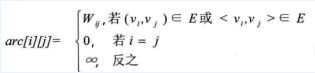
（3）**求顶点vi的所有邻接点就是将矩阵中第i行元素扫描一遍，**

**arc[i][j]为1就是邻接点；**

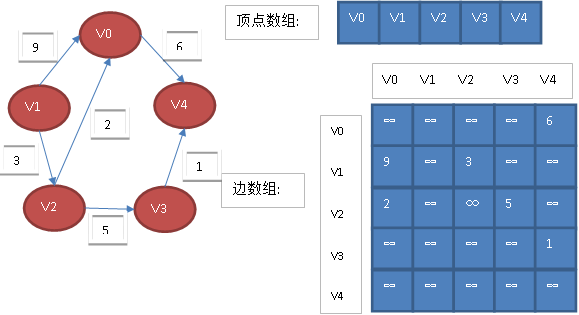
而**有向图讲究入度和出度**，顶点v2的入度为2，正好是第i列各数之和。顶点v2的出度为1，即第i行的各数之和。



若图G是**网图**，有n个顶点，则邻接矩阵是一个n\*n的方阵，定义为：



这里的wij表示(vi,vj)上的权值。无穷大表示一个计算机允许的、大于所有边上权值的值，也就是一个不可能的极限值。下面左图就是一个有向网图，下图就是它的邻接矩阵。



### 1.2.2邻接表

邻接矩阵是不错的一种图存储结构，但是，对于边数相对顶点较少的图，这种结构存在对存储空间的极大浪费。因此，找到一种**数组与链表相结合的存储方法**称为邻接表。

邻接表的存储方式是这样的：

1. 图中**顶点用一个一维数组存储**，当然，**顶点也可以用单链表来存储**，

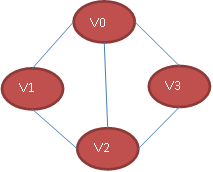
不过，数组可以较容易的读取顶点的信息，更加方便。

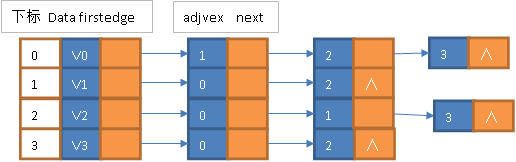
1. 图中**每个顶点vi的所有邻接点构成一个线性表**，由于邻接点的个数不定，所以，用单链表存储，无向图称为顶点vi的边表，有向图则称为顶点vi作为弧尾的出边表。

数据结构定义:



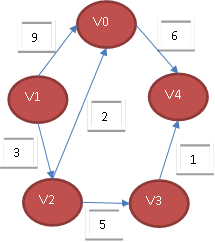
例如，下图就是一个无向图的邻接表的结构。

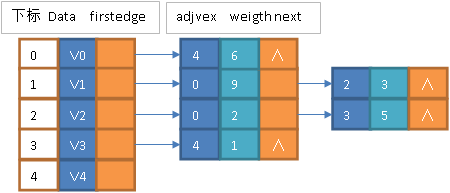




从图中可以看出，顶点表的各个结点由data和firstedge两个域表示，data是数据域，存储顶点的信息，firstedge是指针域，指向边表的第一个结点，即此顶点的第一个邻接点。边表结点由adjvex和next两个域组成。adjvex是邻接点域，存储某顶点的邻接点在顶点表中的下标，next则存储指向边表中下一个结点的指针。

对于带权值的网图，可以在边表结点定义中再增加一个weight的数据域，存储权值信息即可。如下图所示。





## 1.3、图的遍历

图的遍历和树的遍历类似，希望从图中某一顶点出发访遍图中其余顶点，且使每一个顶点仅被访问一次，这一过程就叫图的遍历。

对于图的遍历来说，如何避免因回路陷入死循环，就需要科学地设计遍历方案，通常有两种遍历次序方案：**深度优先遍历**和**广度优先遍历**。

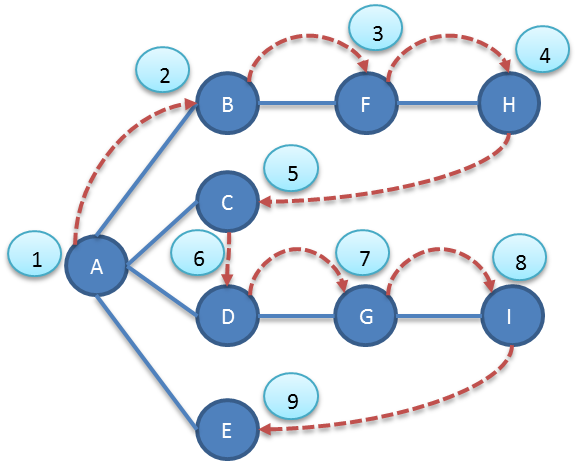
### 1.3.1深度优先遍历

**深度优先遍历，也有称为深度优先搜索，简称DFS**。其实，**就像是一棵树的前序遍历**。

它从图中某个结点v出发，访问此顶点，然后从v的未被访问的邻接点出发深度优先遍历图，直至图中所有和v有路径相通的顶点都被访问到。若图中尚有顶点未被访问，则另选图中一个未曾被访问的顶点作起始点，重复上述过程，直至图中的所有顶点都被访问到为止。

**深度优先搜索是通过栈来实现的。**

下图中的数字显示了深度优先搜索顶点被访问的顺序



为了实现深度优先搜索，首先选择一个起始顶点并需要遵守三个规则：

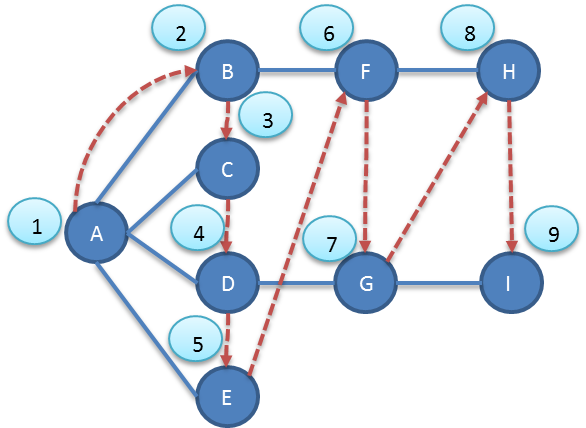
* 如果可能，访问一个邻接的未访问顶点，标记它，并把它放入栈中。
* 当不能执行规则1时，如果栈不空，就从栈中弹出一个顶点。
* 如果不能执行规则1和规则2，就完成了整个搜索过程。

### 1.3.2广度优先遍历

**广度优先遍历，又称为广度优先搜索，简称BFS。图的广度优先遍历就类似于树的层序遍历了。**

在深度优先搜索中，算法表现得好像要尽快地远离起始点似的。相反，在广度优先搜索中，**算法好像要尽可能地靠近起始点**。**它首先访问起始顶点的所有邻接点，然后再访问较远的区域。它是用队列来实现的**。

下面图中的数字显示了广度优先搜索顶点被访问的顺序。



实现广度优先搜索，也要遵守三个规则：

* 访问下一个未来访问的邻接点，这个顶点必须是当前顶点的邻接点，标记它，并把它插入到队列中。
* 如果因为已经没有未访问顶点而不能执行规则1时，那么从队列头取一个顶点，并使其成为当前顶点。
* 如果因为队列为空而不能执行规则2，则搜索结束。

### 1.4 最小生成树

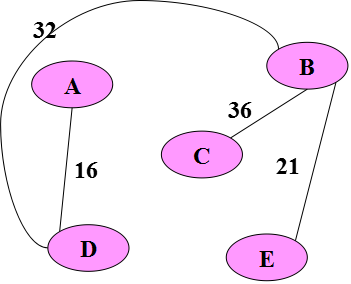
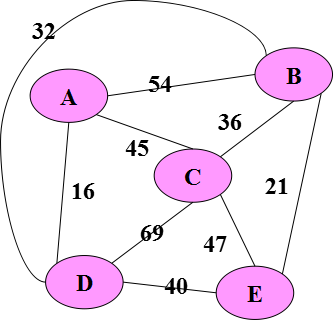
* 相关概念

**生成树**：一个**连通图的生成树是它的极小连通子图**，在**n个顶点的情形下，有n-1条边**。生成树是对连通图而言的，是连同图的极小连通子图，包含图中的所有顶点，有且仅有n-1条边。

**最小生成树：**在图论中，常常将树定义为一个无回路连通图。对于一个带权的无向连通图，其每个生成树所有边上的权值之和可能不同，我们把**所有边上权值之和最小的生成树称为图的最小生成树**。

* 案例（造价最优问题就是一个最小生成树问题）

铺设煤气管道问题（图形结构）假设要在某个城市的n个居民区之间铺设煤气管道，则在这n个居民区之间只要铺设n-1条管道即可。假设任意两个居民区之间都可以架设管道，管道铺设方案。在众多可选边中，如何选择n-1条边，使总代价最小？这就是求该网络最小生成树问题。

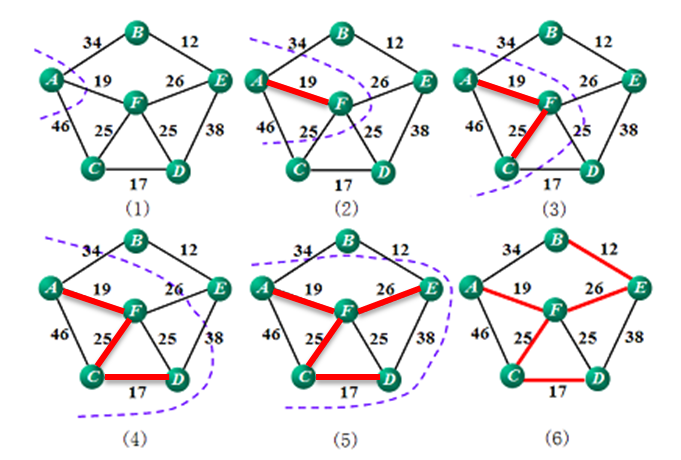


* **普里姆（Prim）算法**
  + 基本思想：设G=(V, E)是具有n个顶点的连通网，T=(U, TE)是G的最小生成树，T的初始状态为U={u0}（u0∈V），TE={}，重复执行下述操作：**在所有u∈U，v∈V-U的边中找一条代价最小的边(u, v)并入集合TE，同时v并入U，直至U=V**。即：
    - 从连通网络 G = { V, E }中的某一顶点 u0 出发，选择与它关联的具有最小权值的边(u0, v)，将其顶点加入到生成树的顶点集合U中。
    - 以后每一步从**一个顶点在U中，而另一个顶点不在U中的各条边中选择权值最小的边(**u, v),把它的顶点加入到集合U中。如此继续下去，直到网络中的所有顶点都加入到生成树顶点集合U中为止。

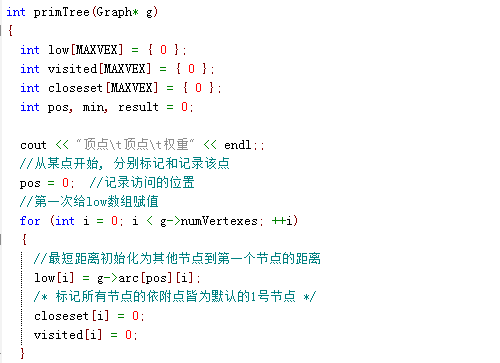
此时，**TE中必有n-1条边，T=(V，TE)为G的最小生成树。**

**Prim算法的核心:始终保持TE中的边集构成一棵生成树。**

* + 示例



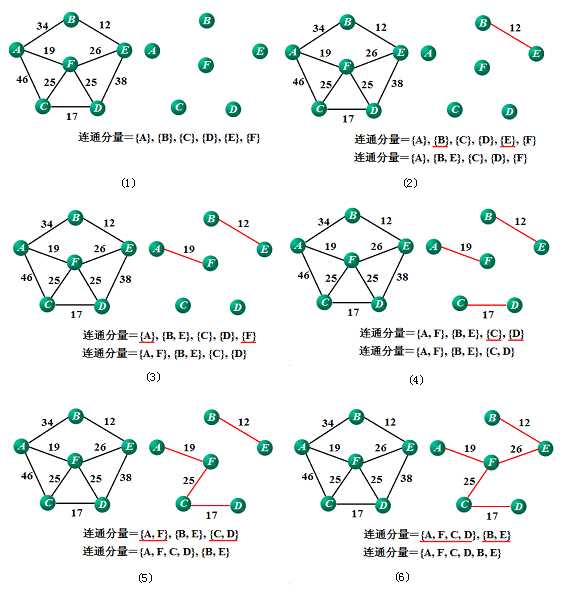
算法实现:







* **克鲁斯卡尔（Kruskal）算法** 
  + 基本思想：设无向连通网为G＝(V, E)，令G的最小生成树为T＝(U, TE)，其初态为U＝V，TE＝{ }，然后，按照边的权值由小到大的顺序，考察G的边集E中的各条边。若被考察的边的两个顶点属于T的两个不同的连通分量，则将此边作为最小生成树的边加入到T中，同时把两个连通分量连接为一个连通分量；若被考察边的两个顶点属于同一个连通分量，则舍去此边，以免造成回路，如此下去，当T中的连通分量个数为1时，此连通分量便为G的一棵最小生成树。
  + 示例



**两种算法总结:**

**prim算法适合稠密图，其时间复杂度为O(n2)，其时间复杂度与边得数目无关，而kruskal算法的时间复杂度为O(nlogn)跟边的数目有关，适合稀疏图。**

### 1.5 最短路径

* 基本概念

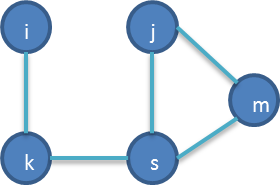
**最短路径**：从图中某一顶点（源点）到达另一顶点（终点）找到一条路径，沿此路径上各边的权值总和（称为路径长度）达到最小。   
**单源最短路径**：已知有向带权图(简称有向网)G=(V，E)，找出从某个源点s∈V到V中其余各顶点的最短路径。

习惯上称路径开始顶点为源点，路径的最后一个顶点为终点。

* 最短路径的最优子结构性质

该性质描述为：如果P(i,j)={Vi....Vk..Vs...Vj}是从顶点i到j的最短路径，k和s是这条路径上的一个中间顶点，那么P(k,s)必定是从k到s的最短路径。下面证明该性质的正确性。

假设P(i,j)={Vi....Vk..Vs...Vj}是从顶点i到j的最短路径，则有P(i,j)=P(i,k)+P(k,s)+P(s,j)。而P(k,s)不是从k到s的最短距离，那么必定存在另一条从k到s的最短路径P'(k,s)，那么P'(i,j)=P(i,k)+P'(k,s)+P(s,j)<P(i,j)。则与P(i,j)是从i到j的最短路径相矛盾。因此该性质得证。(如图)



* 迪杰斯特拉(Dijkstra)算法

由上述性质可知，如果存在一条从i到j的最短路径(Vi.....Vk,Vj)，Vk是Vj前面的一顶点。那么(Vi...Vk)也必定是从i到k的最短路径。为了求出最短路径，Dijkstra就提出了**以最短路径长度递增，逐次生成最短路径的算法**。譬如对于源顶点V0，首先选择其直接相邻的顶点中长度最短的顶点Vi，那么当前已知可得从V0到达Vj顶点的最短距离dist[j]=min{dist[j],dist[i]+matrix[i][j]}。根据这种思路，假设存在G=<V,E>，源顶点为V0，U={V0},dist[i]记录V0到i的最短距离，path[i]记录从V0到i路径上的i前面的一个顶点。

* 从V-U中选择使dist[i]值最小的顶点i，将i加入到U中；
* 更新与i直接相邻顶点的dist值。

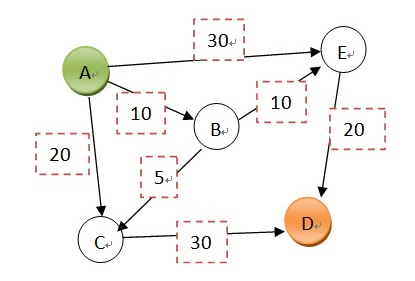
(dist[j]=min{dist[j],dist[i]+matrix[i][j]})

* 直到U=V，停止。
* 案例

交通网络可以画成带权的图，图中顶点表示城市，边代表城市间的公路，边上的权表示公路的长度。对于这样的交通网络常常提出这样的问题：两地之间是否有公路可通？在有几条路可通的情况下，哪一条路最短？以上提出的问题就是在带权图中求最短路径问题，此时路径的长度不是路径上边的数目，而是路径上的边所带权值的总和。

在下面图中找出 A 城到 D 城最近的一条公路。

本次用于分析的拓扑图如下：(A为起点，D为终点，边上的数字为权值)



### 1.6 Prim算法和Dijkstra算法的异同

主要有以下几点：

* 1

Prim是计算最小生成树的算法，比如为N个村庄修路，怎么修花销最少。

Dijkstra是计算最短路径的算法，比如从a村庄走到其他任意村庄的距离。

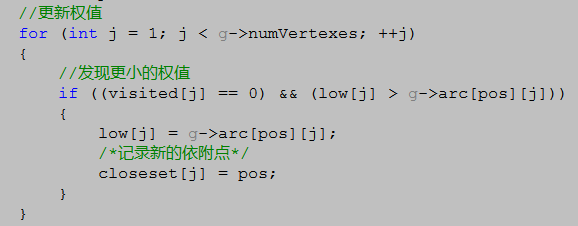
* 2

Prim算法中有一个统计总len的变量，每次都要把到下一点的距离加到len中；

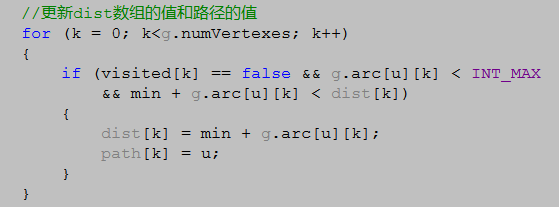
Dijkstra算法中却没有，只需要把到下一点的距离加到cls数组中即可；

* 3

Prim算法的更新操作更新的low是已访问集合到未访问集合中各点的距离；



Dijkstra算法的更新操作更新的dist是源点到未访问集合中各点的距离，已经访问过的相当于已经找到源点到它的最短距离了；



# 2. 查找

## 2.1 二叉排序树

### 2.1.1 二叉排序树定义

* 简介

二叉排序树(Binary Sort Tree)又称二叉查找(搜索)树(Binary Search Tree)。

* 满足条件

它是一颗空树，或者是满足如下性质的二叉树：

**①若它的左子树非空，则左子树上所有结点的值均小于根结点的值；**

**②若它的右子树非空，则右子树上所有结点的值均大于根结点的值；**

**③左、右子树本身又各是一棵二叉排序树。**

上述性质简称二叉排序树性质(BST性质)，故二叉排序树实际上是满足BST性质的二叉树。

### 2.1.2 二叉排序树的特点

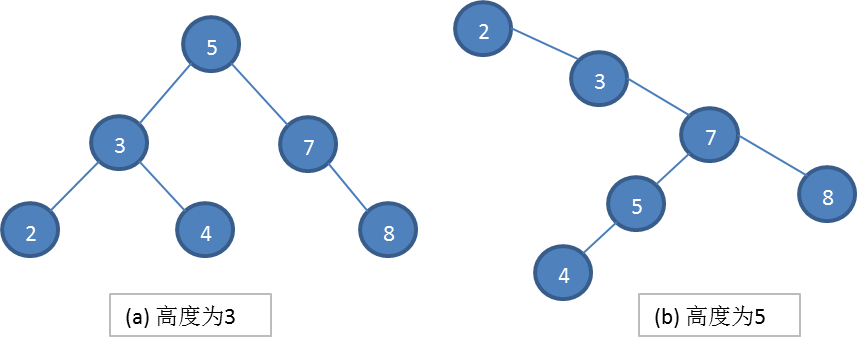
* 特点（BST性质）
  + **二叉排序树中任一结点x，其左(右)子树中任一结点y(若存在)的关键字必小(大)于x的关键字。**
  + **二叉排序树中，各结点关键字是惟一的。**

注意：

实际应用中，不能保证被查找的数据集中各元素的关键字互不相同，所以可将二叉排序树定义中BST性质(1)里的"小于"改为"大于等于"，或将BST性质(2)里的"大于"改为"小于等于"，甚至可同时修改这两个性质。

* + **按中序遍历该树所得到的中序序列是一个递增有序序列。**
* **图例**

下图所示两棵树均是二叉排序树,他们的中序序列均为有序序列:2,3,4,5,6,7,8

****

### 2.1.3 二叉排序树操作

中序遍历二叉排序树可得到一个依据关键字的有序序列，一个无序序列可以通过构造一棵二叉排序树变成一个有序序列，**构造树的过程即是对无序序列进行排序的过程**。每次插入的新的结点都是二叉排序树上新的叶子结点，在进行插入操作时，不必移动其它结点，只需改动某个结点的指针，由空变为非空即可。搜索、插入、删除的时间复杂度等于树高，期望O(logn)，最坏O(n)（数列有序，树退化成线性表，如右斜树）。虽然二叉排序树的最坏效率是O(n)，但它支持动态查找，且有很多改进版的二叉排序树可以使树高为O(logn)，如AVL、红黑树等。

#### 2.1.3.1 存储结构

* 存储结构

二叉排序树通常采用二叉链表作为存储结构。

* 结构定义

/\* 二叉树的二叉链表结点结构定义 \*/

typedef struct BiTNode /\* 结点结构 \*/

{

int data; /\* 结点数据 \*/

struct BiTNode \*lchild, \*rchild; /\* 左右孩子指针 \*/

} BiTNode, \*BiTree;

#### 2.1.3.2 插入算法

* 插入新节点的过程
  + 若二叉排序树T为空，则为待插入的关键字key申请一个新结点，并令其为根；
  + 若二叉排序树T不为空，则将key和根的关键字比较：
    - 若二者相等，则说明树中已有此关键字key，无须插入。
    - 若key<T→key，则将key插入根的左子树中。
    - 若key>T→key，则将它插入根的右子树中。

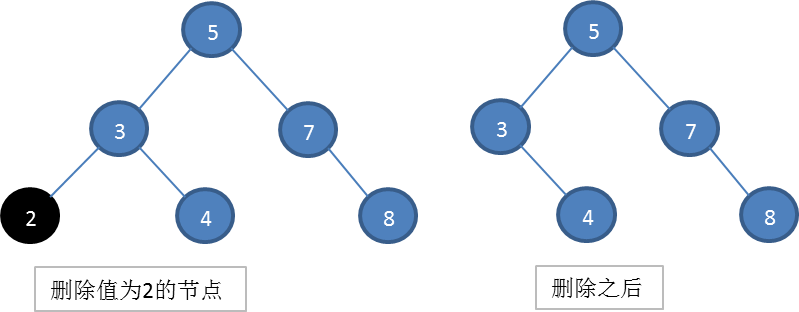
子树中的插入过程与上述的树中插入过程相同。如此进行下去，直到将key作为一个新的叶结点的关键字插入到二叉排序树中，或者直到发现树中已有此关键字为止。

#### 2.1.3.3 查找算法

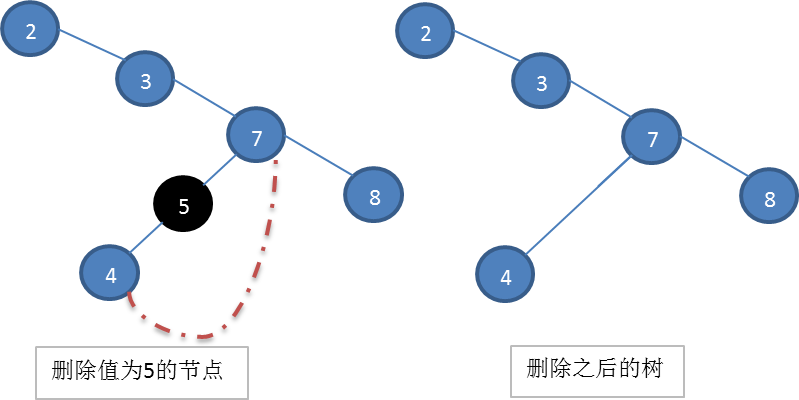
* 查找步骤
  + 若二叉树T为空树,则搜索失败,否则:
  + 若查找的数x等于T根节点的数据域的值,则查找成功,否则:
  + 若查找的数x小于T根节点的数据域的值,则搜索左子树,否则:
  + 查找右子树

#### 2.1.3.4 删除算法

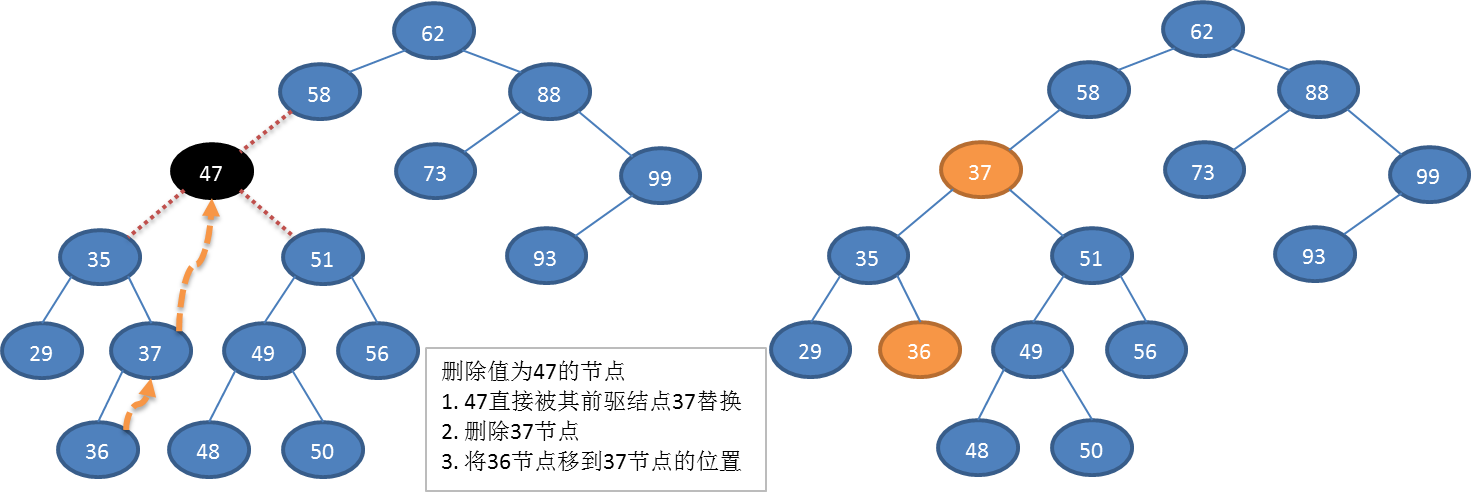
* 删除步骤（分三种情况）
* 若p结点为叶子结点，即该节点左子树PL和右子树PR均为空树。由于删去叶子结点不破坏整棵树的结构，则只需修改其双亲结点的指针即可。



* 若p结点只有左子树PL或右子树PR，此时只要令PL或PR直接成为其双亲结点f的左子树（当p是左子树）或右子树（当p是右子树）即可，作此修改也不破坏二叉排序树的特性。



* 若p结点的左子树和右子树均不空。在删去p之后，为保持其它元素之间的相对位置不变，可按中序遍历保持有序进行调整。比较好的做法是，找到p的直接前驱（或直接后继）s，用s来替换结点p，然后再删除结点s。



### 2.1.4二叉排序树性能分析

每个结点的Ci为该结点的层次数。最好的情况是二叉排序树的形态和折半查找的判定树相同，其平均查找长度和logn成正比（O(log2(n))）。最坏情况下，当先后插入的关键字有序时，构成的二叉排序树为一棵斜树，树的深度为n，其平均查找长度为(n + 1) / 2。也就是时间复杂度为O(n)，等同于顺序查找。因此，如果希望对一个集合按二叉排序树查找，最好是把它构建成一棵平衡的二叉排序树（平衡二叉树）。

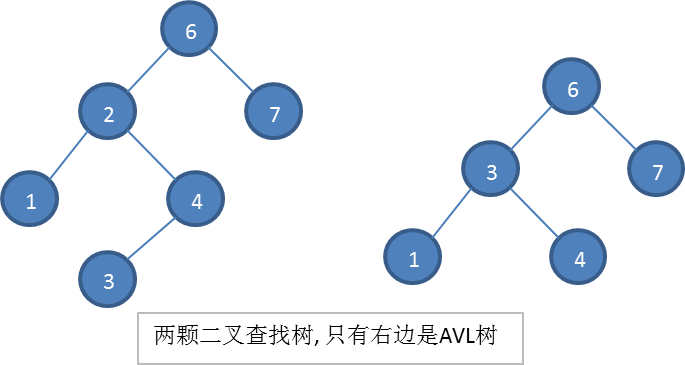
衡二叉树（Balanced Binary Tree）是二叉查找树的一个进化体，也是第一个引入平衡概念的二叉树。1962年，G.M. Adelson-Velsky 和 E.M. Landis发明了这棵树，所以它又叫AVL树。平衡二叉树要求对于每一个节点来说，它的左右子树的高度之差不能超过1，如果插入或者删除一个节点使得高度之差大于1，就要进行节点之间的旋转，将二叉树重新维持在一个平衡状态。这个方案很好的解决了二叉查找树退化成链表的问题，把插入，查找，删除的时间复杂度最好情况和最坏情况都维持在O(logN)。但是频繁旋转会使插入和删除牺牲掉O(logN)左右的时间，不过相对二叉查找树来说，时间上稳定了很多。

## 2.2 平衡二叉树

### 2.2.1基本概念

* 概念

平衡二叉树（Balanced Binary Tree）是二叉查找树的一个进化体，也是第一个引入平衡概念的二叉树。1962年，G.M. Adelson-Velsky 和 E.M. Landis发明了这棵树，所以它又叫AVL树。平衡二叉树要求对于每一个节点来说，**它的左右子树的高度之差不能超过1**，如果插入或者删除一个节点使得高度之差大于1，就要进行节点之间的旋转，将二叉树重新维持在一个平衡状态。这个方案很好的解决了二叉查找树退化成链表的问题，把插入，查找，删除的时间复杂度最好情况和最坏情况都维持在O(logN)。但是频繁旋转会使插入和删除牺牲掉O(logN)左右的时间，不过相对二叉查找树来说，时间上稳定了很多。



平衡二叉树实现的大部分过程和二叉查找树是一样的（学平衡二叉树之前一定要会二叉查找树），**区别就在于插入和删除之后要写一个旋转算法去维持平衡，维持平衡需要借助一个节点高度的属性。**

* 满足平衡二叉树的条件

一棵空树是平衡二叉树；

若 T 是一棵非空二叉树，其左、右子树为 TL 和 TR ，令 hl 和 hr 分别为左、右子树的深度。当且仅当

* + **TL 、 TR 都是平衡二叉树；**
  + **｜ hl － hr ｜≤ 1；**

时，则 T 是平衡二叉树。

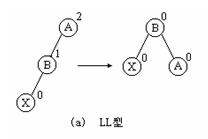
相应地定义 hl － hr 为二叉平衡树的平衡因子 (balance factor) 。因此，平衡二叉树上所有结点的平衡因子可能是 -1，0 ，1 。换言之，若一棵二叉树上任一结点的平衡因子的绝对值都不大于 1 ，则该树是就平衡二叉树。

* 最小不平衡子树

**以离插入结点最近、且平衡因子绝对值大于 1 的结点作根结点的子树**。为了简化讨论，不妨假设二叉排序树的最小不平衡子树的根结点为 A ，则调整该子树的规律可归纳为下列四种情况：

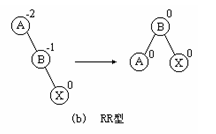
* + LL型

新结点 X 插在 A 的左孩子的左子树里。调整方法见下图 (a) 。图中以 B 为轴心，将 A 结点从 B 的右上方转到 B 的右下侧，使 A 成为 B 的右孩子。



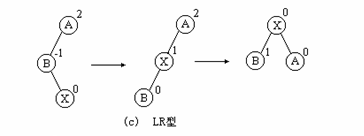
* + RR型

新结点 X 插在 A 的右孩子的右子树里。调整方法见下图 (b) 。图中以 B 为轴心，将 A 结点从 B 的左上方转到 B 的左下侧，使 A 成为 B 的左孩子。



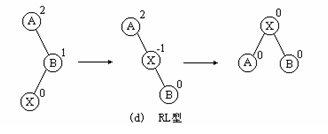
* + LR型

新结点 X 插在 A 的左孩子的右子树里。调整方法见图 (c) 。分为两步进行：第一步以 X 为轴心，将 B 从 X 的左上方转到 X 的左下侧，使 B 成为 X 的左孩子， X 成为 A 的左孩子。第二步跟 LL 型一样处理 ( 应以 X 为轴心 ) 。



* + RL型

新结点 X 插在 A 的右孩子的左子树里。调整方法见图 (d) 。分为两步进行：第一步以 X 为轴心，将 B 从 X 的右上方转到 X 的右下侧，使 B 成为 X 的右孩子， X 成为 A 的右孩子。第二步跟 RR 型一样处理 ( 应以 X 为轴心 ) 。



### 2.2.2 平衡二叉树的创建

创建平衡二叉树，我们采用依次插入节点的方式进行。而平衡二叉树上插入节点采用递归的方式进行。递归算法如下：

* 若该树为一空树，那么插入一个数据元素为e的新节点作为平衡二叉树的根节点，树的高度增加1。
* 若待插入的数据元素e和平衡二叉树的根节点的关键字相等，那么就不需要进行插入操作。
* 若待插入的元素e比平衡二叉树的根节点的关键字小，而且在平衡二叉树的左子树中也不存在和e有相同关键字的节点，则将e插入在平衡二叉树的左子树上，并且当插入之后的左子树深度增加1时，分别就下列情况处理之：
  + 平衡二叉树的根节点的平衡因子为-1（右子树的深度大于左子树的深度）：则将根节点的平衡因子更改为0，平衡二叉树的深度不变；
  + 平衡二叉树的根节点的平衡因子为0（左右子树的深度相等）：则将根节点的平衡因子修改为1，平衡二叉树的深度增加1；
  + 平衡二叉树的根节点的平衡因子为1（左子树的深度大于右子树的深度）：若平衡二叉树的左子树根节点的平衡因子为1，则需要进行单向右旋转平衡处理，并且在右旋处理后，将根节点和其右子树根节点的平衡因子更改为0，树的深度不变；
  + 若平衡二叉树的左子树根节点的平衡因子为-1，则需进行先向左，后向右的双向旋转平衡处理，并且在旋转处理之后，修改根节点和其左，右子树根节点的平衡因子，树的深度不变；
* 若e的关键字大于平衡二叉树的根节点的关键字，而且在平衡二叉树的右子树中不存在和e有相同关键字的节点，则将e插入到平衡二叉树的右子树上，并且当插入之后的右子树深度加1时，分别就不同的情况处理之：
  + 平衡二叉树的根节点的平衡因子是1（左子树的深度大于右子树的深度）：则将根节点的平衡因子修改为0，平衡二叉树的深度不变；
  + 平衡二叉树的根节点的平衡因子是0（左右子树的深度相等）：则将根节点的平衡因子修改为-1，树的深度加1；
  + 平衡二叉树的根节点的平衡因子为-1（右子树的深度大于左子树的深度）：若平衡二叉树的右子树根节点的平衡因子为1，则需要进行两次选择，第一次先向右旋转，再向左旋转处理，并且在旋转处理之后，修改根节点和其左，右子树根节点的平衡因子，树的深度不变；
  + 若平衡二叉树的右子树根节点的平衡因子为1，则需要进行一次向左的旋转处理，并且在左旋之后，更新根节点和其左，右子树根节点的平衡因子，树的深度不变；

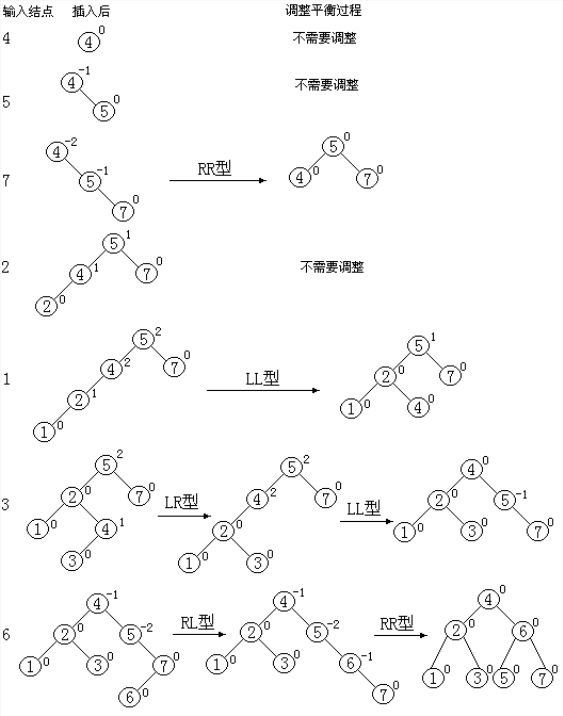
### 课后练习

* 案例

设一组记录的关键字按以下次序进行插入： 4 、 5 、 7 ， 2 、 1 、 3 、 6 ，创建平衡二叉树。

* 分析

在下图中，当插入关键字为 3 的结点后，由于离结点 3 最近的平衡因子为 2 的祖先是根结点 5 。所以，第一次旋转应以结点 4 为轴心，把结点 2 从结点 4 的左上方转到左下侧，从而结点 5 的左孩子是结点 4 ，结点 4 的左孩子是结点 2 ，原结点 4 的左孩子变成了结点 2 的右孩子。第二步再以结点 4 为轴心，按 LL 类型进行转换。



## 2.3 红黑树

* 红黑树的介绍

红黑树，一种二叉查找树，但在每个结点上增加一个存储位表示结点的颜色，可以是Red或Black。

通过对任何一条从根到叶子的路径上各个结点着色方式的限制，红黑树确保没有一条路径会比其他路径长出俩倍，因而是接近平衡的。

* 红黑树节点的组成

红黑树上每个结点内含五个域，color，key，left，right，parent。如果相应的指针域没有，则设为NULL。

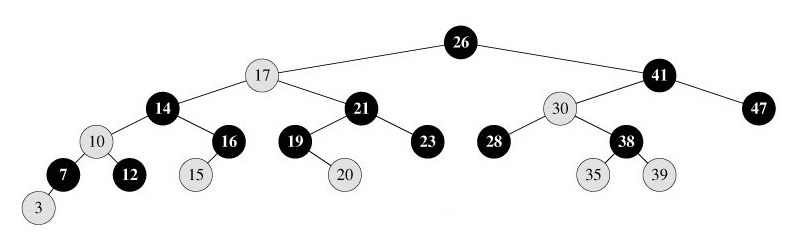


* 红黑树的性质

一般的，红黑树，满足以下性质，即只有满足以下全部性质的树，我们才称之为红黑树：

* + **每个结点要么是红的，要么是黑的。**
  + **根结点是黑的。**
  + **每个叶结点，即空结点（NULL）是黑的。**
  + **如果一个结点是红的，那么它的俩个儿子都是黑的。**
  + **从根到叶节点的每条路径,必须包含相同数目的黑色节点。**

下图所示,即为一颗红黑树



* 红黑树的操作

当我们在对红黑树进行插入和删除等操作时，对树做了修改，那么可能会违背红黑树的性质。

**为了保持红黑树的性质，我们可以通过对树进行旋转，即修改树种某些结点的颜色及指针结构**，以达到对红黑树进行插入、删除结点等操作时，红黑树依然能保持它特有的性质（如上文所述的，五点性质）。

**修正方式:**

* + **改变节点的颜色**
  + **旋转**

# 3.排序

现实生活中排序很重要，例如：



* 概念

排序是计算机内经常进行的一种操作，其目的是将一组“无序”的数据元素调整为“有序”的数据元素。

* 排序数学定义：

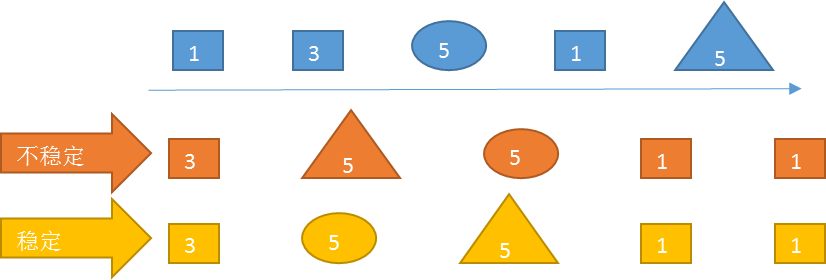
假设含n个数据元素的序列为{ R1, R2, …, Rn}，其相应的关键字序列为{ K1, K2, …, Kn}这些关键字相互之间可以进行比较，即在它们之间存在着这样一个关系 ：

Kp1≤Kp2≤…≤Kpn

按此固有关系将上式记录序列重新排列为{ Rp1, Rp2, …，Rpn}的操作称作排序

* 排序的稳定性

如果在序列中有两个数据元素r[i]和r[j]，它们的关键字k[i] == k [j]，且在排序之前，对象r[i]排在r[j]前面。如果在排序之后，对象r[i]仍在r[j]前面，则称这个排序方法是稳定的，否则称这个排序方法是不稳定的。



* 多关键字排序

排序时需要比较的关键字多余一个，排序结果首先按关键字1进行排序，当关键字1相同时按关键字2进行排序，当关键字n-1相同时按关键字n进行排序，对于多关键字排序，只需要在比较操作时同时考虑多个关键字即可！

* 排序中的关键操作
  + 比较：任意两个数据元素通过比较操作确定先后次序
  + 交换：数据元素之间需要交换才能得到预期结果
* 内排序和外排序
  + 内排序：整个排序过程不需要访问外存便能完成
  + 外排序：待排序的数据元素数量很大，整个序列的排序过程不可能在内存中完成
* 排序的审判
  + 时间性能：关键性能差异体现在比较和交换的数量
  + 辅助存储空间：为完成排序操作需要的额外的存储空间，必要时可以“空间换时间”
  + 算法的实现复杂性：过于复杂的排序法会影响代码的可读性和可维护性，也可能影响排序的性能
* 总结
  + 排序是数据元素从无序到有序的过程
  + 排序具有稳定性，是选择排序算法的因素之一
  + 比较和交换是排序的基本操作
  + 多关键字排序与单关键字排序无本质区别
  + 排序的时间性能是区分排序算法好坏的主要因素

## 3.1选择法

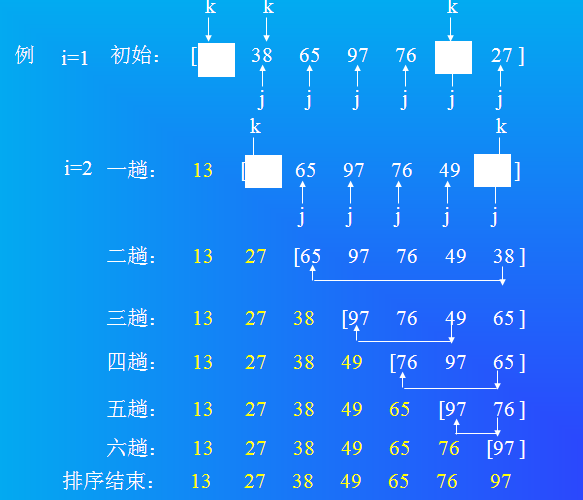
* 算法介绍

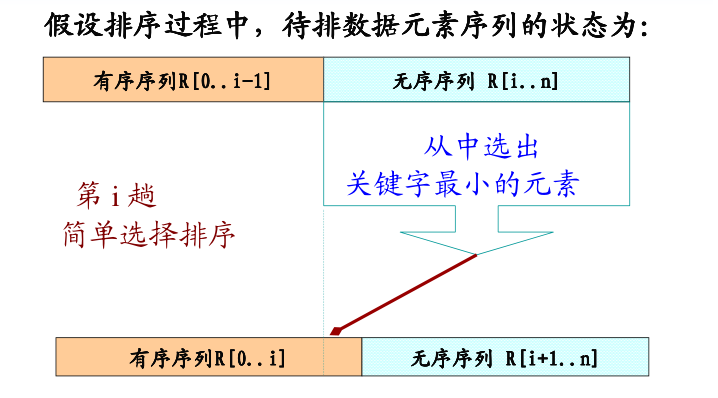
每一次从待排序的数据元素中选出最小（或最大）的一个元素，存放在序列的起始位置，直到全部待排序的数据元素排完。

* 基本思想

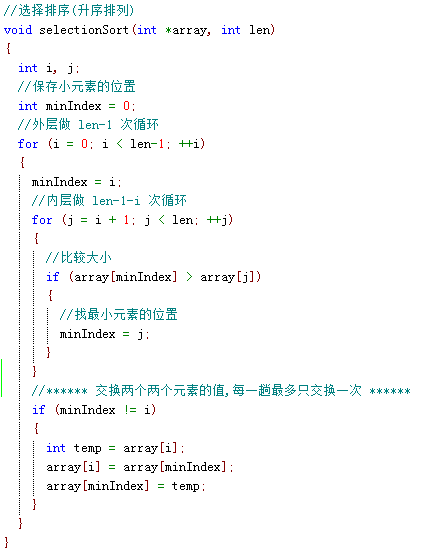
设数组为a[0…n-1]。

* 初始时，数组全为无序区为a[0..n-1]。令i=0
* 在无序区a[i…n-1]中选取一个最小的元素，将其与a[i]交换。交换之后a[0…i]就形成了一个有序区。
* i++并重复第二步直到i==n-1。排序完成。
* 稳定性
  + **选择排序是不稳定的排序方法**
  + **选择排序效率：O（n²）**
* 排序过程





* 算法实现



## 3.2冒泡排序

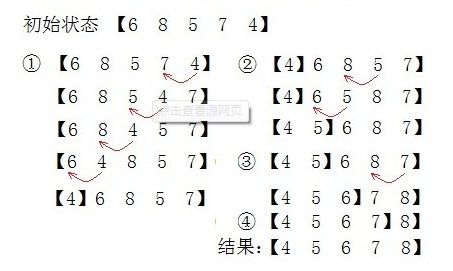
* 算法介绍

冒泡排序算法的运作如下：（从后往前）

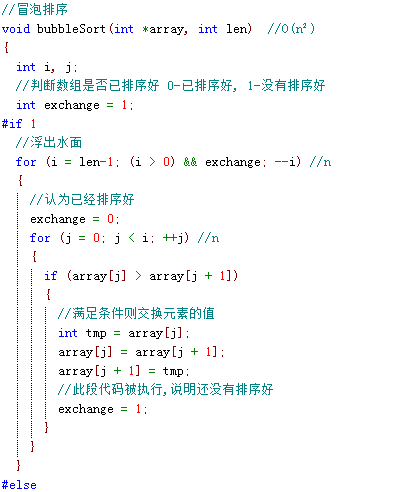
* + **比较相邻的元素**。如果第一个比第二个大，就交换他们两个。
  + 对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对。在这一点，最后的元素应该会是最大的数。
  + 针对所有的元素重复以上的步骤，除了最后一个。
  + 持续每次对越来越少的元素重复上面的步骤，直到没有任何一对数字需要比较。
* 基本思想

设数组长度为N。

* + **比较相邻的前后两个数据**，如果前面数据大于后面的数据，就将二个数据交换。
  + 这样对数组的第0个数据到N-1个数据进行一次遍历后，最大的一个数据就“升”到数组第N-1个位置。
  + N=N-1，如果N不为0就重复前面二步，否则排序完成。
* 稳定性
  + **冒泡排序是一种稳定的排序算法**
  + **冒泡排序的效率：O（n²）**
* 排序过程



* 算法实现





* 冒泡总结：
  + 冒泡排序是一种效率低下的排序方法，在数据规模很小时，可以采用。数据规模比较大时，最好用其它排序方法。
  + 上述例子总对冒泡做了优化，添加了exchange作为标记，记录序列是否已经有序，减少循环次数。

## 3.3插入排序

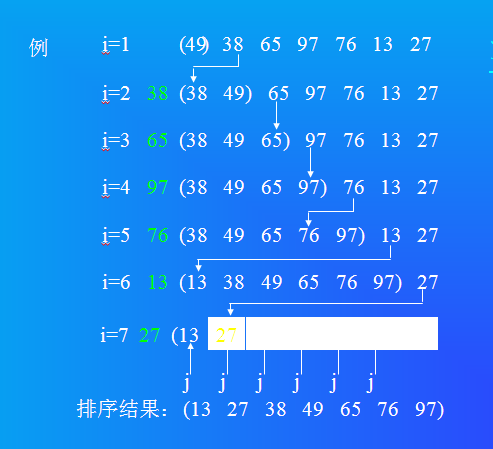
* 算法介绍

每次将一个待排序的记录，按其关键字大小插入到前面已经排好序的子序列中的适当位置，直到全部记录插入完成为止。

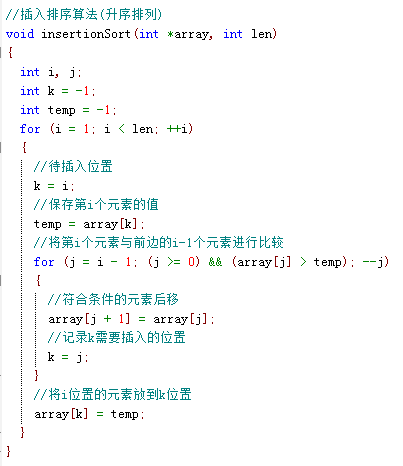
* 基本思想

设数组为a[0…n-1]。

* + 初始时，a[0]自成1个有序区，无序区为a[1..n-1]。令i=1
  + 将a[i]并入当前的有序区a[0…i-1]中形成a[0…i]的有序区间。
  + i++并重复第二步直到i==n-1。排序完成。
* 稳定性
  + **插入排序是稳定的排序算法**
  + **插入排序效率：O（n²）**
* 排序过程



* 算法实现



## 3.4希尔排序

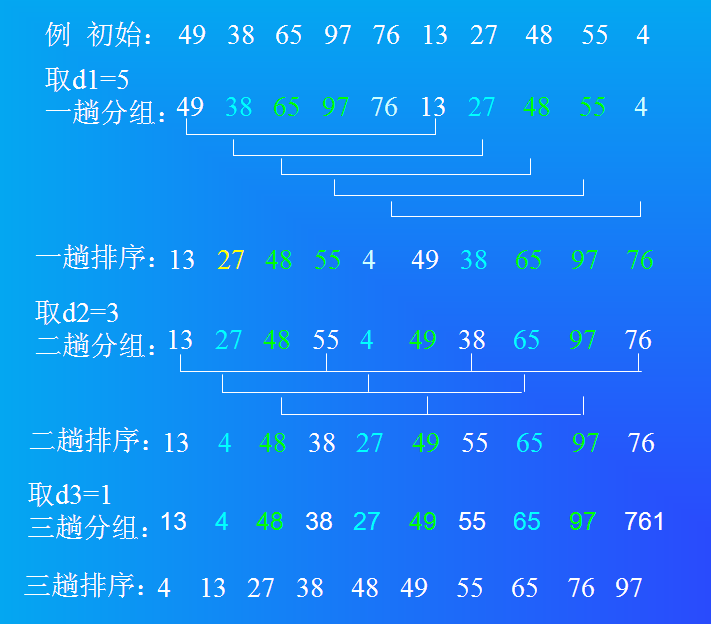
* 算法介绍

**希尔排序的实质就是分组插入排序**，该方法又称缩小增量排序，因DL．Shell于1959年提出而得名。

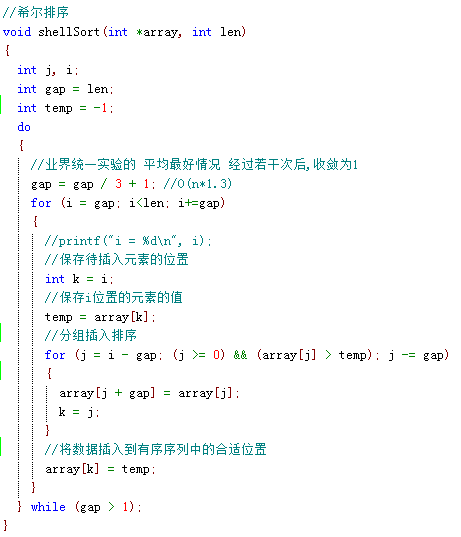
* 基本思想

**先将整个待排元素序列分割成若干个子序列（由相隔某个“增量”的元素组成的）分别进行直接插入排序，然后依次缩减增量再进行排序，待整个序列中的元素基本有序（增量足够小）时，再对全体元素进行一次直接插入排序。因为直接插入排序在元素基本有序的情况下（接近最好情况），效率是很高的，因此希尔排序在时间效率上比前三种方法有较大提高。**

* 稳定性
  + **希尔排序是不稳定的排序算法。**
  + **希尔排序的效率：O（n\*logn）≈ O（1.3\*n）**
* 排序过程



* 算法实现



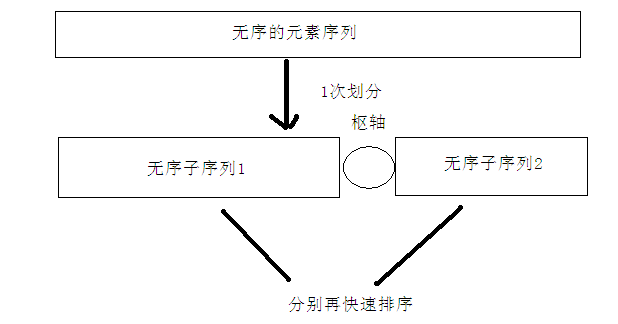
* 关于希尔排序步长的说明
  + 步长的计算公式可以自行制定，最后步长 == 1即可。
  + 通过大量测试得出的结论：步长 = 步长 / 3 + 1

## 3.5快速排序

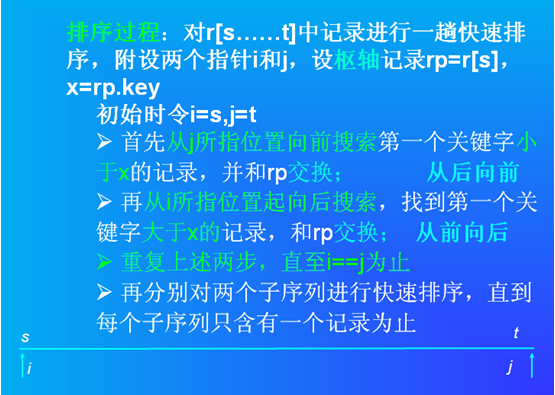
* 算法介绍

快速排序是C.R.A.Hoare于1962年提出的**一种划分交换排序**。它**采用了一种分治的策略**，通常称其为分治法(Divide-and-ConquerMethod)。

* 分治法基本思想
  + 先从数列中取出一个数作为基准数（枢轴）。
  + **分区过程将比这个数大的数全放到它的右边，小于或等于它的数全放到它的左边。**
  + 再对左右区间重复第二步，直到各区间只有一个数。

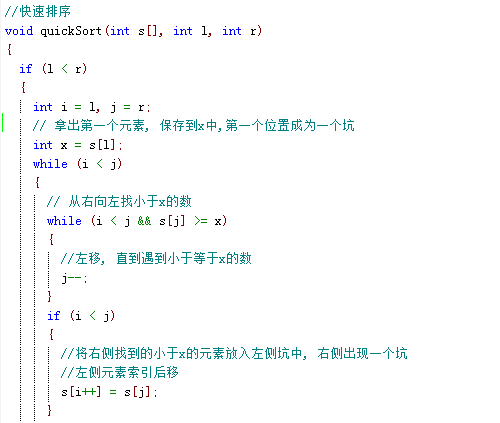


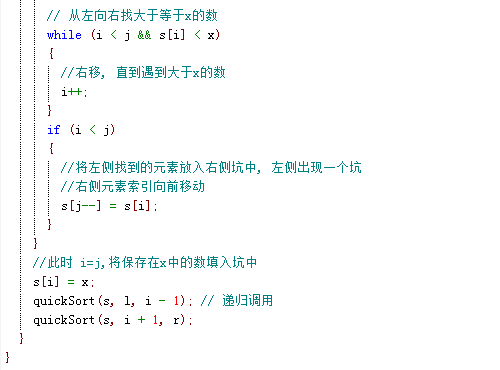
* 稳定性
  + **快速排序是一种不稳定的排序算法。**
  + **排序效率： O(N\*logN)**
* 排序过程





* 算法实现





## 3.6归并排序

* 算法介绍

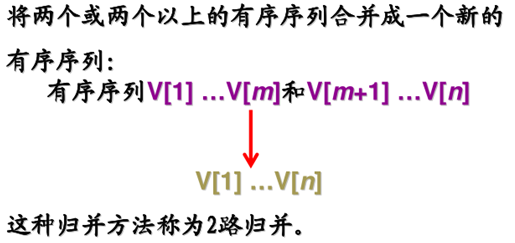
归并排序是建立在归并操作上的一种有效的排序算法。该算法是采用分治法（Divide and Conquer）的一个非常典型的应用。

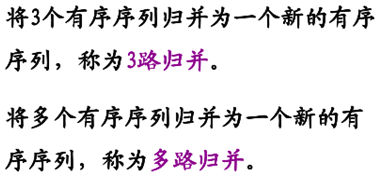
* 基本思想

**基本思路就是将数组分成二组A，B，如果这二组组内的数据都是有序的，那么就可以很方便的将这二组数据进行排序**。如何让这二组组内数据有序了？

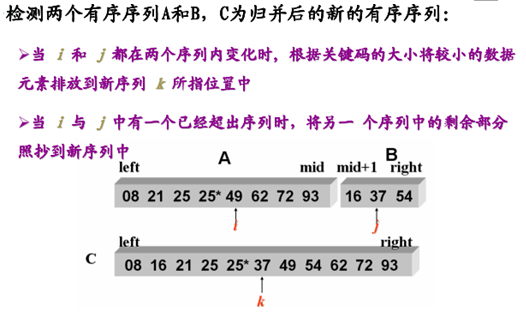
**可以将A，B组各自再分成二组。依次类推，当分出来的小组只有一个数据时，可以认为这个小组组内已经达到了有序，然后再合并相邻的二个小组就可以了。这样通过先递归的分解数列，再合并数列就完成了归并排序。**

* **归并的定义**

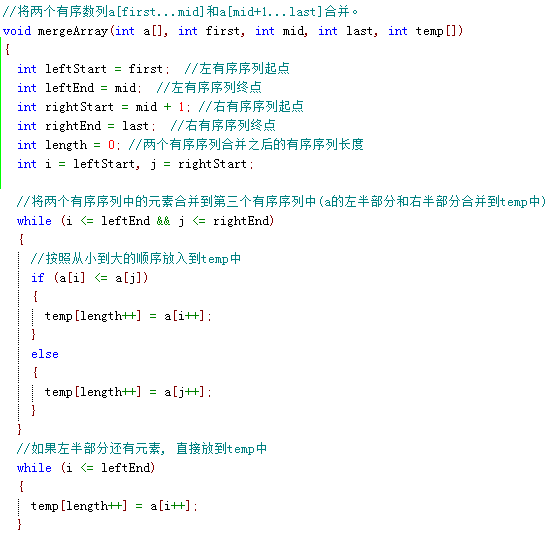




* 稳定性
  + **归并排序是一种稳定的排序算法。**
  + **排序效率： O(N\*logN)**
* **如何合并连个有序序列？？？**
  + 只要从比较二个数列的第一个数，谁小就先取谁，取了后就在对应数列中删除这个数。然后再进行比较，如果有数列为空，那直接将另一个数列的数据依次取出即可。
  + 图例分析



* + 代码实现

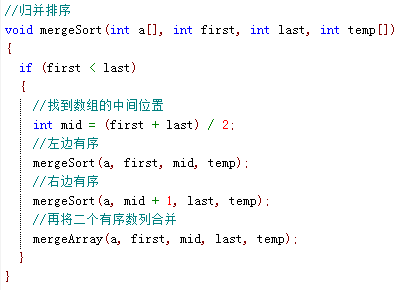




* 排序过程



* 算法实现



## 3.7 堆排序

* 二叉堆
  + 定义

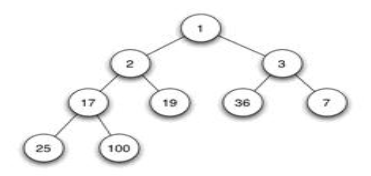
**二叉堆是完全二叉树或者是近似完全二叉树。**

* + 二叉堆满足二个特性：
    - 父结点的键值总是大于或等于（小于或等于）任何一个子节点的键值。
    - 每个结点的左子树和右子树都是一个二叉堆（都是最大堆或最小堆）。

**当父结点的键值总是大于或等于任何一个子节点的键值时为最大堆。**

**当父结点的键值总是小于或等于任何一个子节点的键值时为最小堆。**

下图展示一个最小堆：

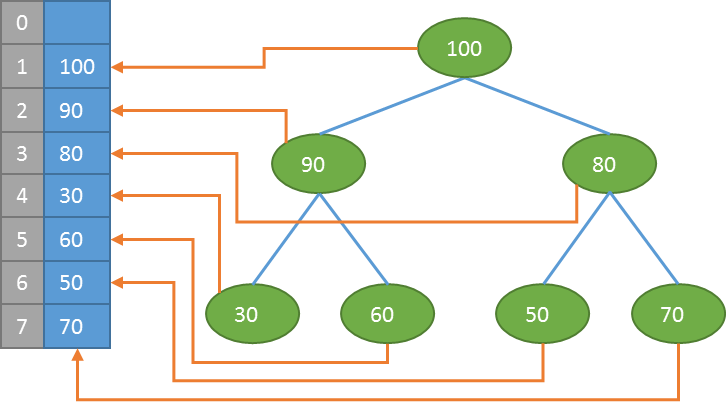


由于其它几种堆（二项式堆，斐波纳契堆等）用的较少，一般将二叉堆就

简称为堆。

* 堆的存储

**一般都用数组来表示堆**，从数组的1号位置开始存储树的根节点,i结点的父结点下标就为i / 2。它的左右子结点下标分别为2 \* i 和2 \* i + 1。如第1个结点左右子结点下标分别为2和3。数组的零号位置一般空闲不用。



* 堆的插入（向上渗透）

每次插入都是将新数据放在数组最后。可以发现从这个新数据的父结点到根结点必然为一个有序的数列，现在的任务是将这个新数据插入到这个有序数据中——这就类似于直接插入排序中将一个数据并入到有序区间中

* 堆的删除（向下渗透）

按定义，堆中每次都只能删除第0个数据。为了便于重建堆，实际的操作是将最后一个数据的值赋给根结点，然后再从根结点开始进行一次从上向下的调整。调整时先在左右儿子结点中找最小的，如果父结点比这个最小的子结点还小说明不需要调整了，反之将父结点和它交换后再考虑后面的结点。相当于从根结点将一个数据的“下沉”过程。

## 3.8排序总结

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **排序算法** | **平均时间**  **复杂度** | **最坏时间**  **复杂度** | **平均空间**  **复杂度** | **稳定性** |
| 选择排序 | O(n2) | O(n2) | O(1) | 不稳定 |
| 冒泡排序 | O(n2) | O(n2) | O(1) | 稳定 |
| 直接插入排序 | O(n2) | O(n2) | O(1) | 稳定 |
| 希尔排序 | O(nlogn) | O(n2) | O(log2n) | 不稳定 |
| 快速排序 | O(nlogn) | O(n2) | O(1) | 不稳定 |
| 归并排序 | O(nlogn) | O(nlogn) | O(n) | 稳定 |
| 堆排序 | O(nlogn) | O(nlogn) | O(1) | 不稳定 |