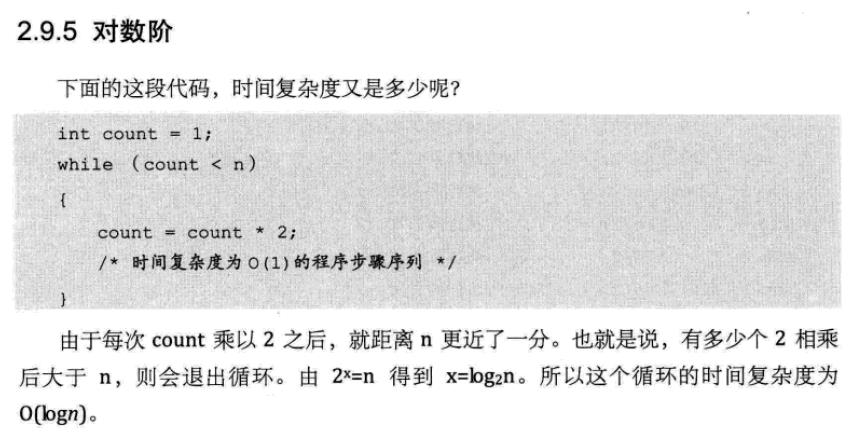
# **【抽象数据类型】**

体现了程序设计中问题分解、抽象和信息隐藏的特性。

# 【算法】

及解决特定问题的步骤描述

# 【算法复杂度】



对数阶： count = count \*2\*2\*2\*2…

线性阶： count = count +1+1+1+1…

【复杂操作由基本操作的组合实现】

# 【线性表】list

【线性表基本操作】

初始化

判断是否为空

清空

获取某位置的值

获取某值的位置

插入

删除

计算长度

## 【线性表的两种物理结构】

### 【顺序存储结构】（三个属性）

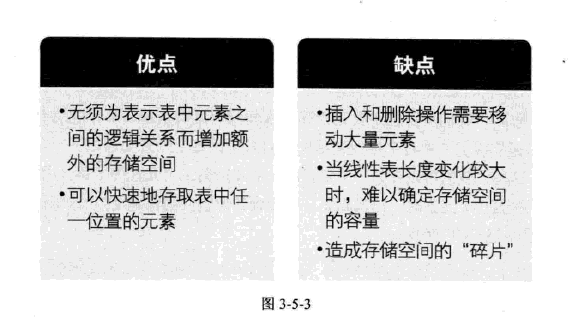
起始位置

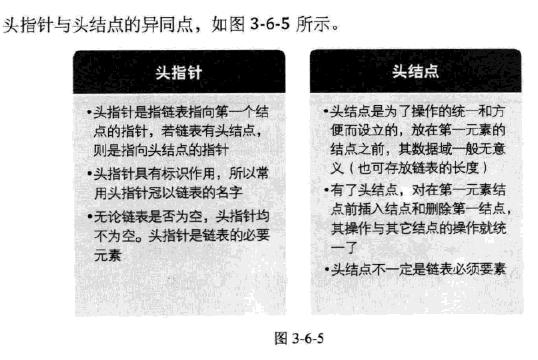
最大容量

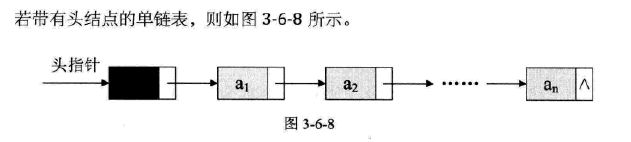
当前长度

【随机存取结构】

存储空间连续，元素所占空间一样——下标操作复杂度为O(1)：一步计算出任意位置的地址







### 【链式存储结构】

如果不知道第i个元素的指针，链式和顺序在插入和删除操作上没有优势，都是O(n)

顺序存储结构：n-i次赋值，移动元素

链式存储结构：i次赋值，寻找i的指针



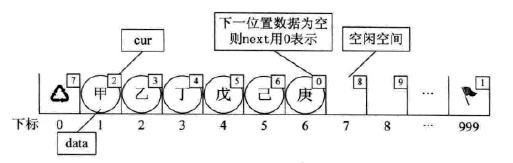
查找的意思是随机存取

#### 【静态链表】用数组描述的链表

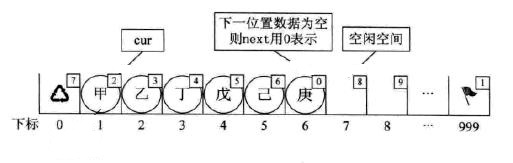
静态链表是为了给没有指针的高级语言实现单链表能力的方法

**原理是在一个数组中巧妙设计了两个相互错开的单向链表**

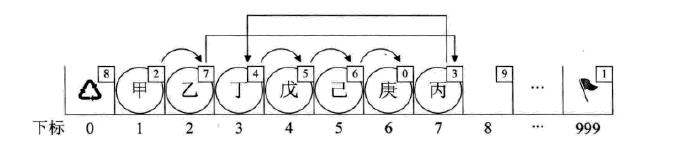
【初始化】左边是数据链，右边是备用链



【首尾元素】尾元素标记数据链头部，首元素标记备用链头部



【插入元素】备用链头部减少一个位置，更新首元素提供的

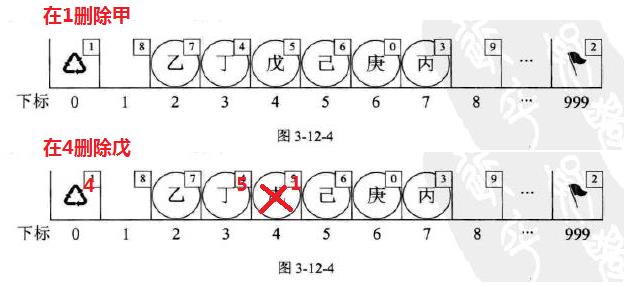


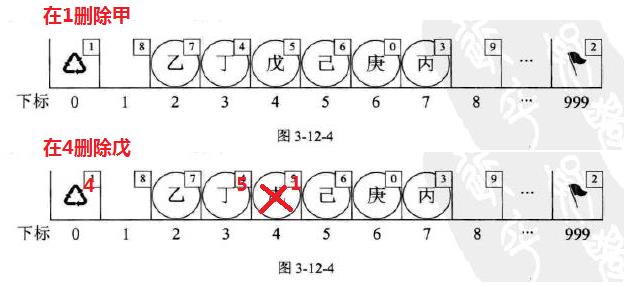
【删除元素】备用链头部增加一个位置

删除1元素



删除4元素





#### 【循环链表】

#### 【双向链表】

# 【栈】stack

先进后出

栈是线性表的特例——只能在表头插入和删除元素

作为线性表的特例，栈也可以有顺序存储结构和链式存储结构

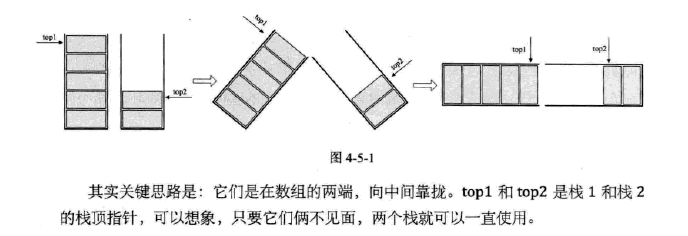
## 【顺序存储结构】预设空间大小

初始化时需要预设栈空间的大小

### 【两栈共享空间】两个栈使用同一个数组

左边的栈已经满了 左边的栈还没满

↓ ↓



适合两个数据此消彼长的模型

## 【链式存储结构】基本无空间限制

用单向链表实现的栈可以不断地入栈，直到内存不足

## 【栈与递归】

编译器使用栈实现递归

对每一层递归，函数的局部变量、参数值、以及返回地址都被压入栈中。在退回阶段逐一弹出。

# 【队列】queue

先进先出

队列作为特殊的线性表同样有顺序存储结构和链式存储结构两种实现方式。

## 【顺序存储结构】

对于队列，顺序存储结构的不足是要频繁移动元素，循环队列可以解决这个问题。

### 【循环队列】

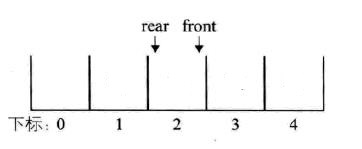
几个要点：

①front指针：指向头部

②rear指针：指向尾部的下一个位置

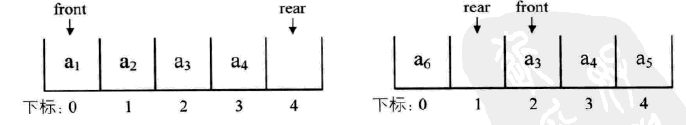
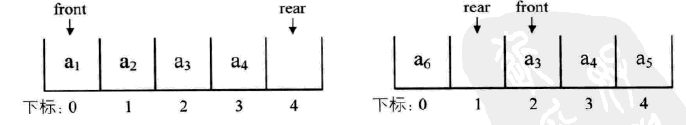
③空队列：front与rear指向同一个位置

rear == front



④满队列：front与rear相差一，或者刚好相差一圈

( rear + 1 ) % SZIE == front

⑤队列长度：

通用公式 ( rear – front + SIZE) % SIZE

## 【链式存储结构】

用链式存储结构实现队列，用的是只能尾部插入、头部删除的单链表

## 【栈与队列的总结】



# 【字符串】string

同样有顺序存储结构和链式存储结构匹配

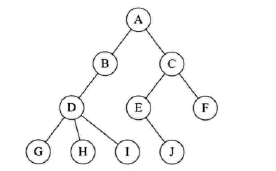
## 【朴素模式匹配算法】

逐一匹配主字符串和子字符串

## 【KMP模式匹配算法】

# 【树】tree

## 【相关概念】



### 【节点】

节点包括**数据元素**和指向其子树的**分支**



### 【度】degree

【节点的度】

节点拥有的子树数量称为节点的**度**，上图节点D的度为3

【树的度】

树的度是各个节点的度中的最大值

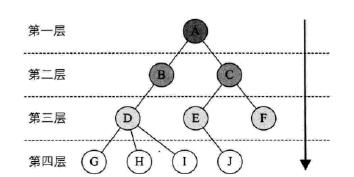
【根节点】

【内部节点】

【叶节点】终端节点，度为0

### 【层次】level

节点的层次从上往下定义



### 【有序树】

树中节点的各子树从左往有排列，是有序的，不能互换，称为有序树

### 【森林】forest

n棵互不相交的树的集合

### 【抽象数据类型】

构造

析构

按定义来构造树（？）

清空

判断是否为空

返回树的深度

返回根节点

返回节点值

给节点赋值

返回父节点

返回最左的子节点

返回右兄弟

插入子树

删除子树

## 【存储】

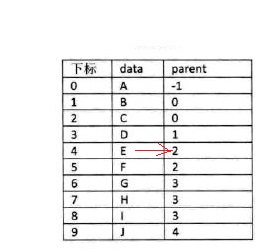
### 【parent表示法】





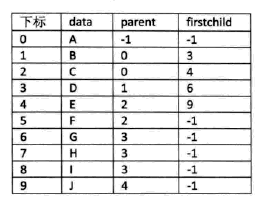
已知child找parent：O(1)

已知parent找child：要遍历树

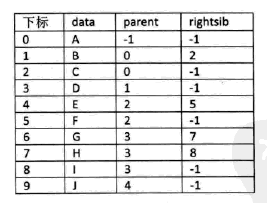


寻找父节点 寻找子节点

【解决与子节点的关系】增加长子域（firstchild）



【解决与兄弟节点的关系】增加兄弟域（rightsib）



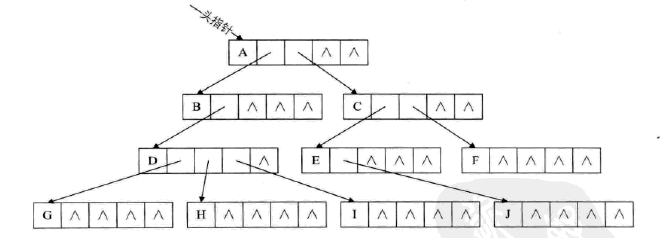
### 【child表示法】

指针域储存子节点

【方案一】指针域个数 = 树的度（节点的度的最大值）

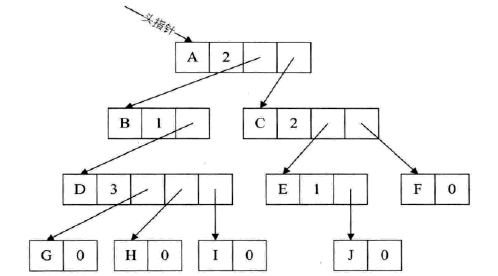


如果节点的度相差很多，就会有很多指针域空出浪费掉。^是空的指针域



【方案二】指针域个数动态分配（使用链表）



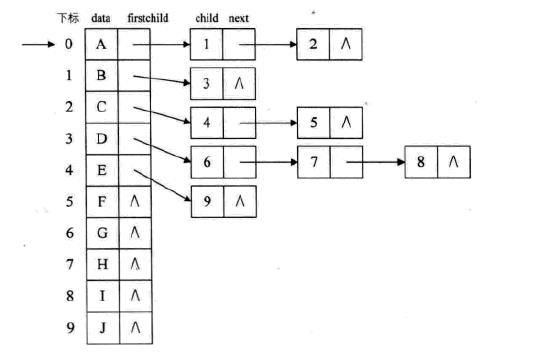


节点结构不一样，有一个空间记录节点的度

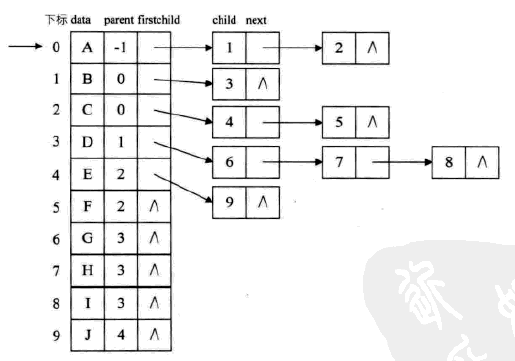
【方案三】顺序结构与链式结构相结合，这才是真正的child表示法，上面两种比较鸡肋



数据元素用顺序结构存储，节点的的子节点位置信息用单链表存储



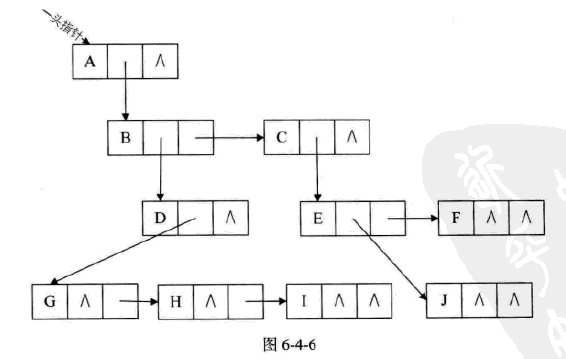
以上三种方案容易找节点的孩子，但是要找父节点还是要靠遍历，以下是结合parent表示法和child表示法：



### 【child\_sibling表示法】

存储节点的长子节点右边的第一个兄弟，这个方法最大的好处是**把一棵复杂的树变成了二叉树**。





便于找到孩子和兄弟，如果需要找父节点，需要遍历或是增加一个parent域

## 【二叉树】binary tree

【二叉树的特点】

①度：节点的度小于等于2，也就是每个节点拥有小于等于2个子节点

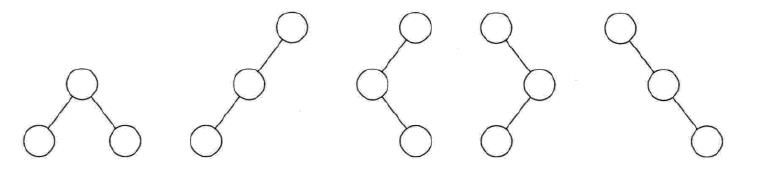
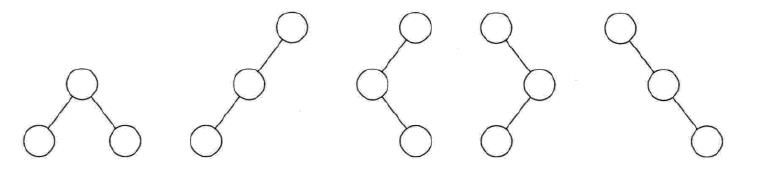
②序：左右节点有顺序，即使只有一个节点也要区别是左节点还是右节点

### 【特殊二叉树】

#### 【斜树】

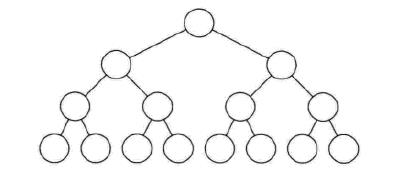
左斜树：所有节点都只有左树，节点个数与树的深度相同（线性表结构）

右斜树：所有节点都只有右树

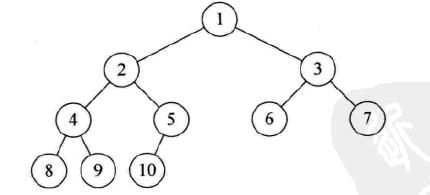


#### 【满二叉树】

所有节点都有左树和右树，并且所有叶子都在同一层上



#### 【完全二叉树】



节点按从上到下、从左到右的顺序，紧密编排的树（编号连续）

（如果上图缺了10，还是完全二叉树，缺了其他编号都不是）

性质：

在完全二叉树中，按层序遍历编号，i的左孩子是2i，右孩子是2i+1（除非没有孩子）

### 【二叉树的性质】

节点总数 = 度为1的节点数 + 度为2的节点数 + 终端节点数

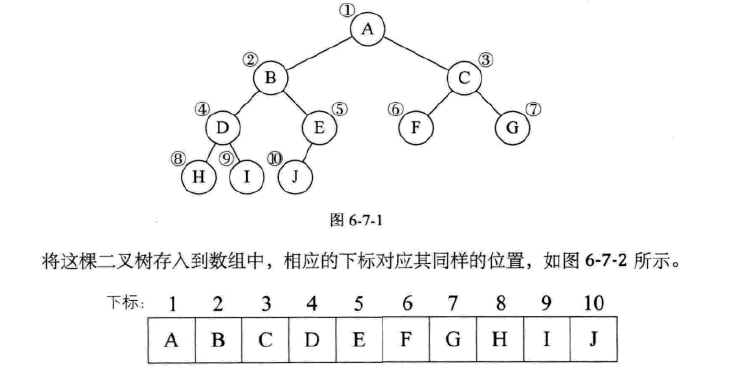
分支总数 = 度为1的节点数\*1 + 度为2的节点数\*2 + 0

分支总数 = 节点总数 – 1

终端节点数 = 度为2的节点数 + 1

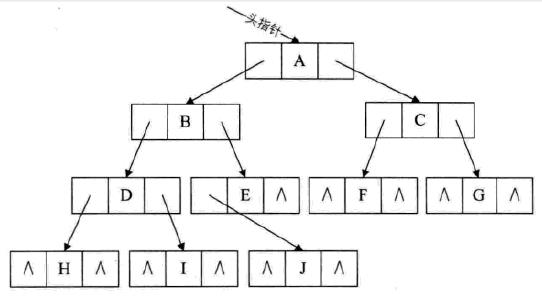
### 【二叉树的存储结构】

#### 【顺序存储结构】完全二叉树



#### 【二叉链表】





增加指向父节点的指针域，变成三叉链表

### 【遍历二叉树】

#### 【前序遍历】printf在前

void PreOrderTraverse(BiTree T)

{

if (T == nullptr)

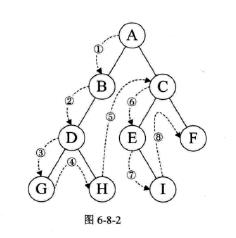
return;

printf("c%", T->data);

PreOrderTraverse(T->lchild);

PreOrderTraverse(T->rchild);

}



#### 【中序遍历】printf在中

void InOrderTraverse(BiTree T)

{

if (T == nullptr)

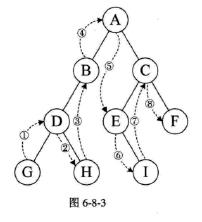
return;

PreOrderTraverse(T->lchild);

printf("c%", T->data);

PreOrderTraverse(T->rchild);

}



#### 【后序遍历】printf在后

void PostOrderTraverse(BiTree T)

{

if (T == nullptr)

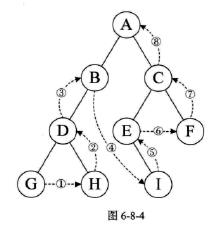
return;

PreOrderTraverse(T->lchild);

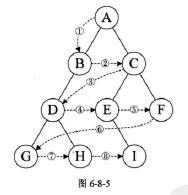
PreOrderTraverse(T->rchild);

printf("c%", T->data);

}



#### 【层序遍历】



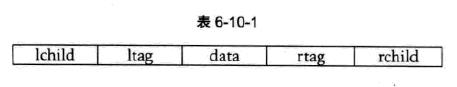
### 【构建二叉树】

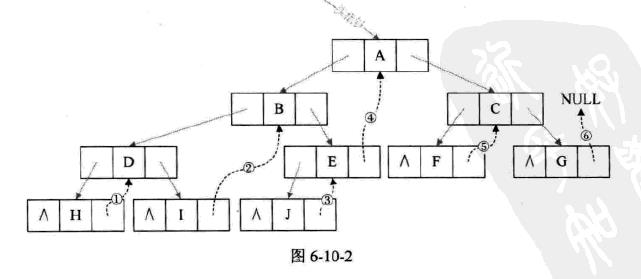
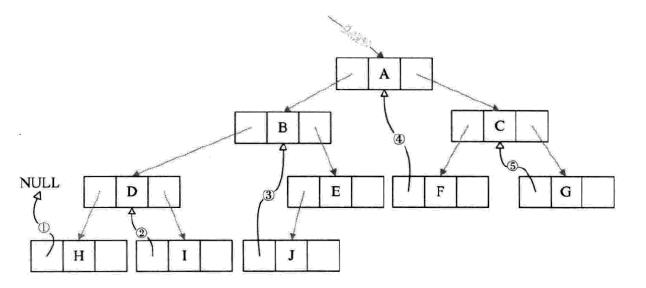
### 【线索二叉树】

生成二叉树后，按某种方式遍历会有排序：HDIBJEAFCG（中序排列）

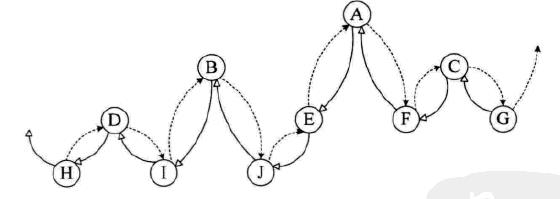
线索化的实质是将二叉链表中的空指针改为指向前驱或后继的线索。

利用空的指针域，左边指向前驱，右边指向后继，还需要两位标记位来标记是线索指针，还是子节点指针。





线索指针让二叉树成为双线链表（但是线索并不全？）

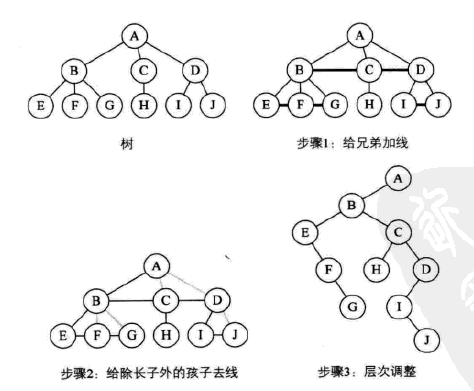


### 【树、森林与二叉树之间的转化】

#### 【树转化为二叉树】

child\_sibling表示法可以将所有的树转化为二叉树

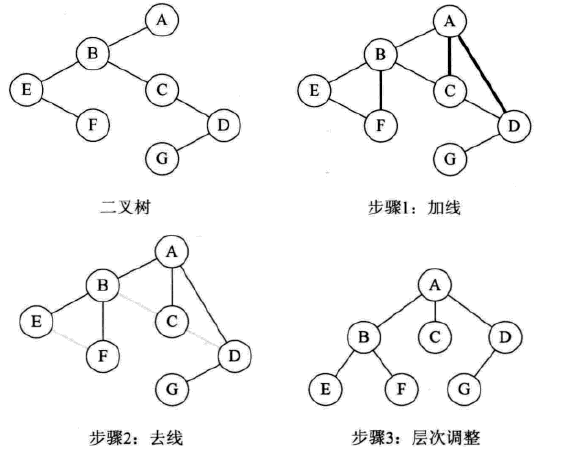
加线、去线、旋转



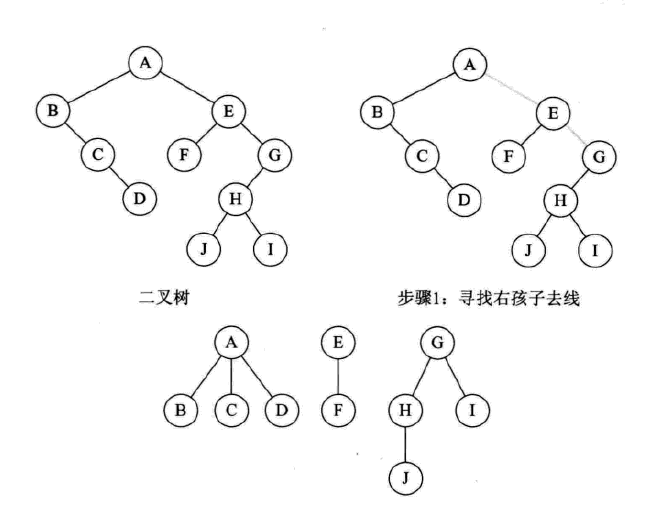
#### 【森林转化为二叉树】

森林有多棵不相交的二叉树，把这些树之间的关系看作是兄弟

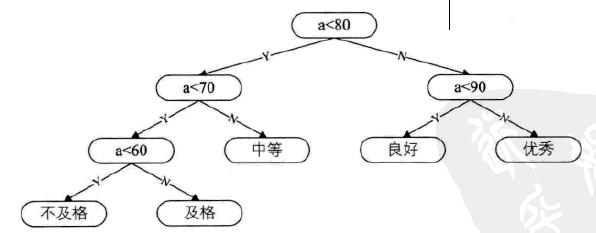
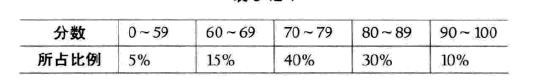
#### 【二叉树转化为树】



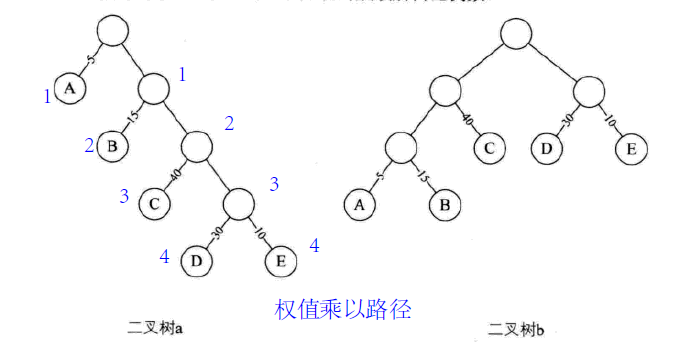
#### 【二叉树转化为森林】

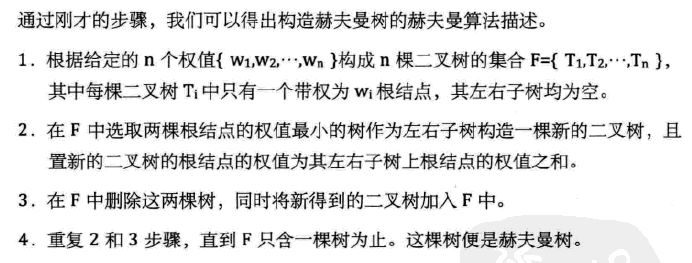


### 【霍夫曼树】



加权路径最小的树（路径是连线数，权值是概率）

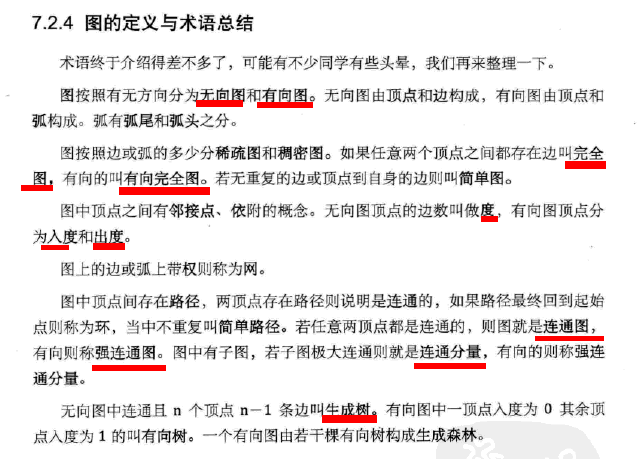




# 【图】graph

图graph、顶点vertex、边edge

表示：G(V,E) G表示图，V表示图中顶点的集合，E表示图中边的集合

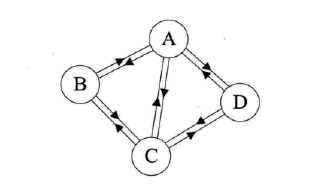


边、

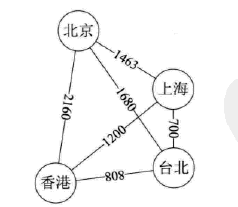
无向边

有向边

有向完全图

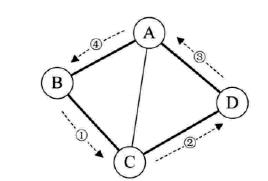


权



顶点的度：与顶点相关联的边的数目

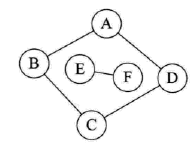
简单回路



连通图

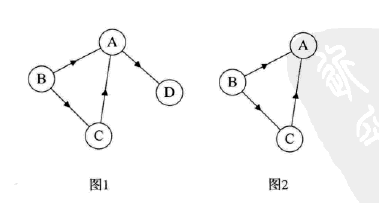
两个顶点之间有路径到达

无向非连通图，下图有两个连通分量



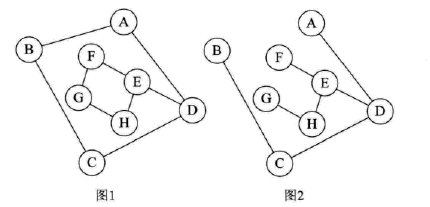
强联通图：有向图中，没每两个顶点之间都有路径

图2是强联通图，是图一的极大强联通子图

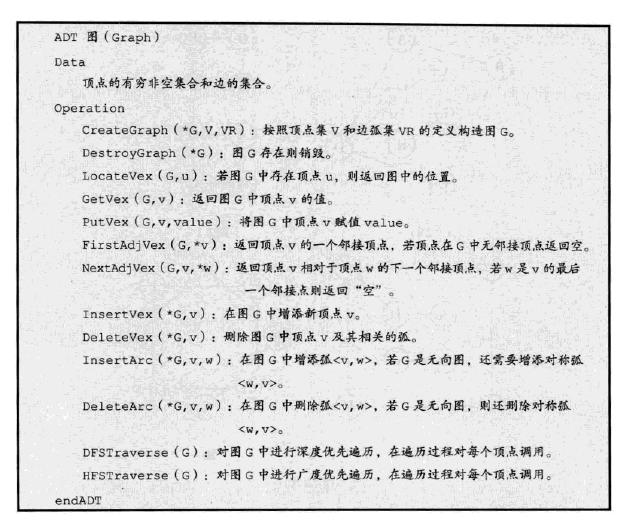


生成树：包含图中的n个顶点，但只有构成树的n-1条边

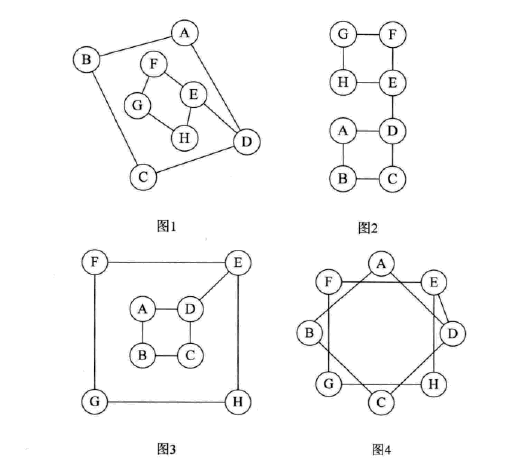
图二是图一的生成树



## 【图的ADT】



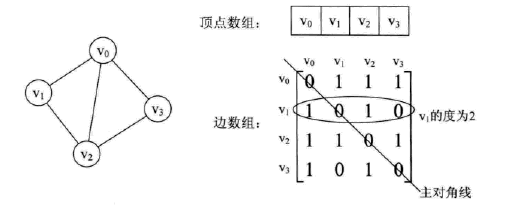
同一个图因顶点不同所造成的假象



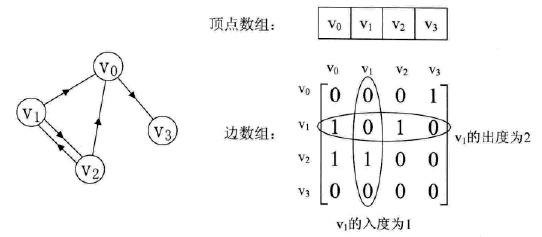
## 【存储结构】

### 【邻接矩阵】Adjacency Matrix

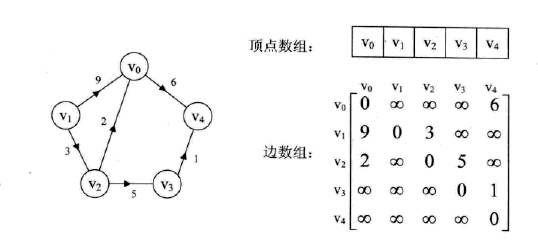
无向图的边数组是一个对称矩阵



有向图



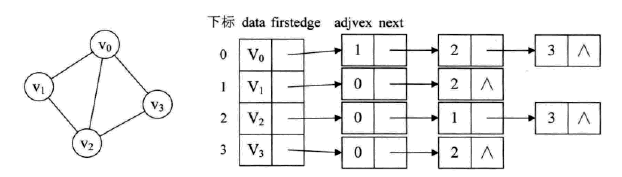
网（边上有权的图）



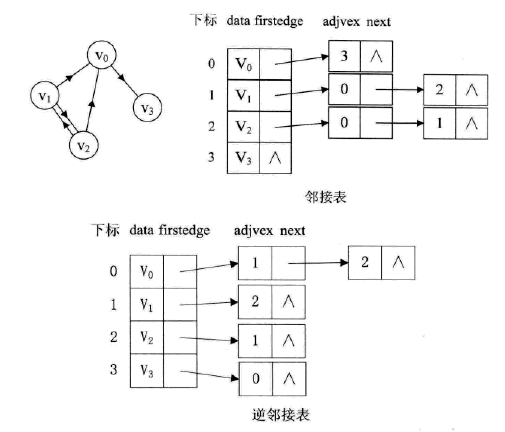
矩阵值就是权值，∞无穷大代表两顶点之间没有连接

### 【邻接表】Adjacency List

无向图（5条边却有10个表示边的节点，因为从0找到1的同时，也要能够从1找到0）



有向图（5条边有5个表示边的节点，因为是单向）

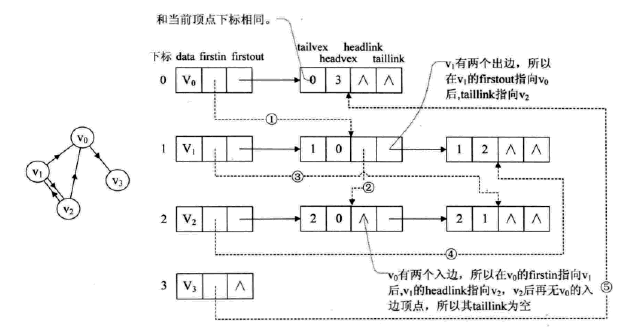


（一般记录出度，记录入度时可以增加一个逆邻接表）

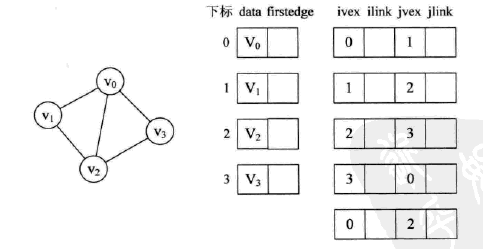
### 【十字链表】Orthogonal List（有向图）

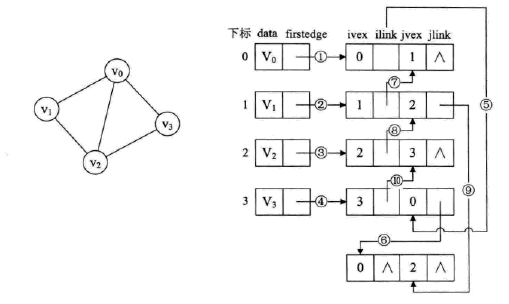
|  |  |  |
| --- | --- | --- |
| 数据 | 入边指针 | 出边指针 |

|  |  |  |  |
| --- | --- | --- | --- |
| 弧尾 | 弧头 | 下一个入边指针 | 下一个出边指针 |



### 【邻接多重表】无向图







### 【边集数组】最简单的形式

两个一维数组组成（一个存储顶点，一个存储边）

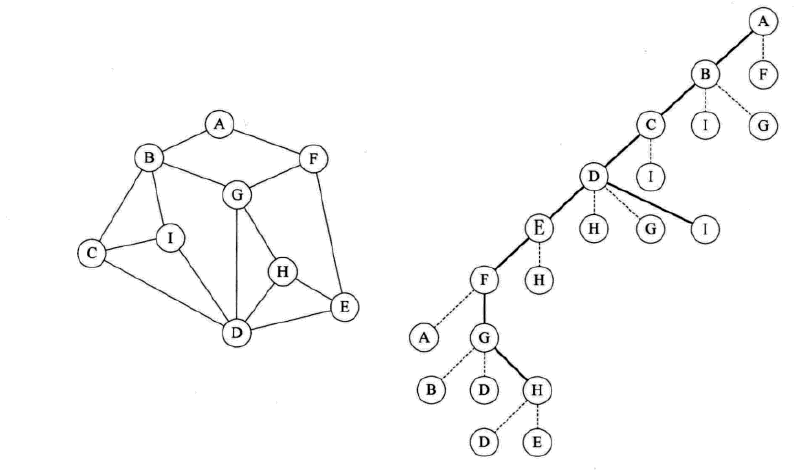


适合对边依次进行处理的操作，不适合对顶点的操作。

## 【图的遍历】

### 【深度优先】彻查

递归：访问A后，访问所有与A联通的顶点



类似于一棵树的前序遍历（递归）

void traverse(Vertex A)

{

if A已被访问

结束递归

printf(A)标记A已被遍历过

traverse(与A连通的顶点1)

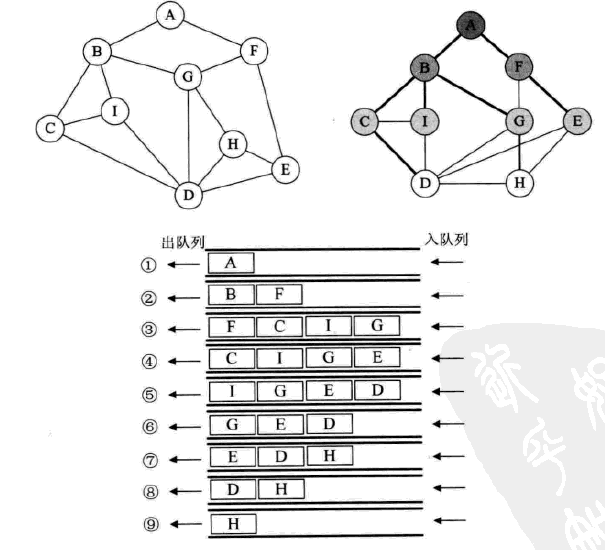
traverse(与A连通的顶点2)

traverse(与A连通的顶点n)

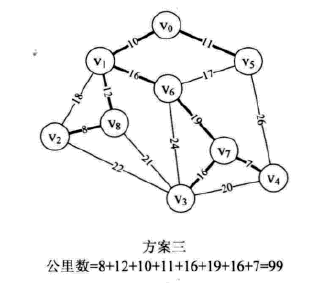
}

### 【广度优先】

类似于一棵树的层序遍历（队列：顶点出队列，顶点的联通点进队列，当然进队列的点是未被访问的）



## 【最小生成树】



### 【普里姆算法】Prim

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **图例** | **说明** | **不可选** | **可选** | **已选（Vnew）** |
| C:\Users\knight\Desktop\2012073015154494.png | 此为原始的加权连通图。每条边一侧的数字代表其权值。 | - | - | - |
| C:\Users\knight\Desktop\2012073015175038.png | 顶点**D**被任意选为起始点。顶点**A**、**B**、**E**和**F**通过单条边与**D**相连。**A**是距离**D**最近的顶点，因此将**A**及对应边**AD**以高亮表示。 | C, G | A, B, E, F | D |
| C:\Users\knight\Desktop\2012073016090032.png | 下一个顶点为距离**D**或**A**最近的顶点。**B**距**D**为9，距**A**为7，**E**为15，**F**为6。因此，**F**距**D**或**A**最近，因此将顶点**F**与相应边**DF**以高亮表示。 | C, G | B, E, F | A, D |
| C:\Users\knight\Desktop\2012073016130394.png | 算法继续重复上面的步骤。距离**A**为7的顶点**B**被高亮表示。 | C | B, E, G | A, D, F |
| C:\Users\knight\Desktop\2012073016143177.png | 在当前情况下，可以在**C**、**E**与**G**间进行选择。**C**距**B**为8，**E**距**B**为7，**G**距**F**为11。**E**最近，因此将顶点**E**与相应边**BE**高亮表示。 | 无 | C, E, G | A, D, F, B |
| C:\Users\knight\Desktop\2012073016154616.png | 这里，可供选择的顶点只有**C**和**G**。**C**距**E**为5，**G**距**E**为9，故选取**C**，并与边**EC**一同高亮表示。 | 无 | C, G | A, D, F, B, E |
| C:\Users\knight\Desktop\2012073016114494.png | 顶点**G**是唯一剩下的顶点，它距**F**为11，距**E**为9，**E**最近，故高亮表示**G**及相应边**EG**。 | 无 | G | A, D, F, B, E, C |
| C:\Users\knight\Desktop\2012073016100874.png | 现在，所有顶点均已被选取，图中绿色部分即为连通图的最小生成树。在此例中，最小生成树的权值之和为39。 | 无 | 无 | A, D, F, B, E, C, G |

### 【克鲁斯卡尔算法】Kruskal

具体流程：

(1)将图G看做一个森林，每个顶点为一棵独立的树

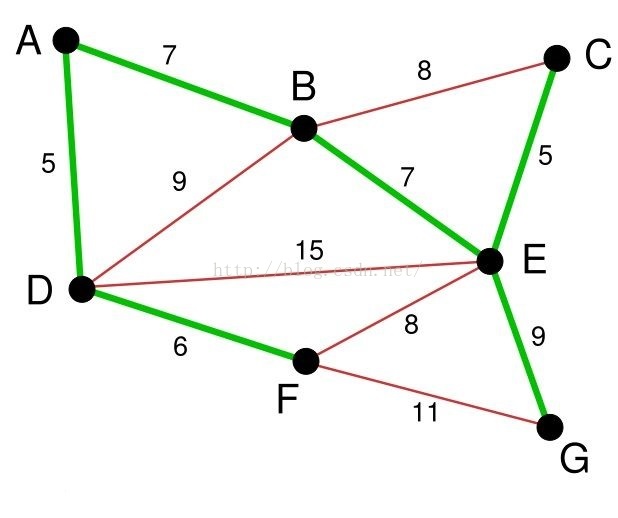
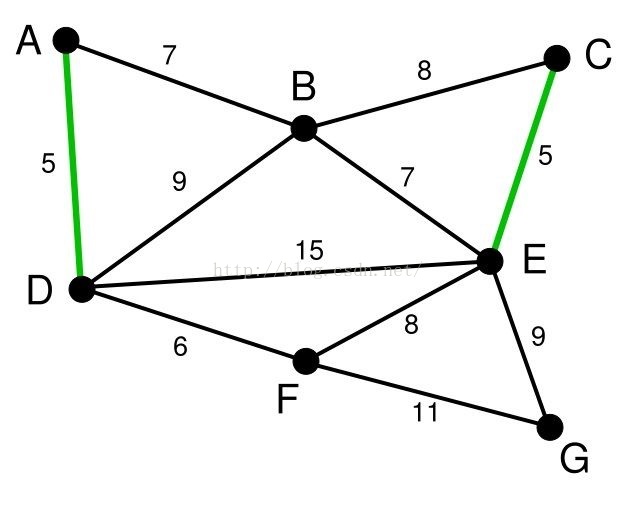
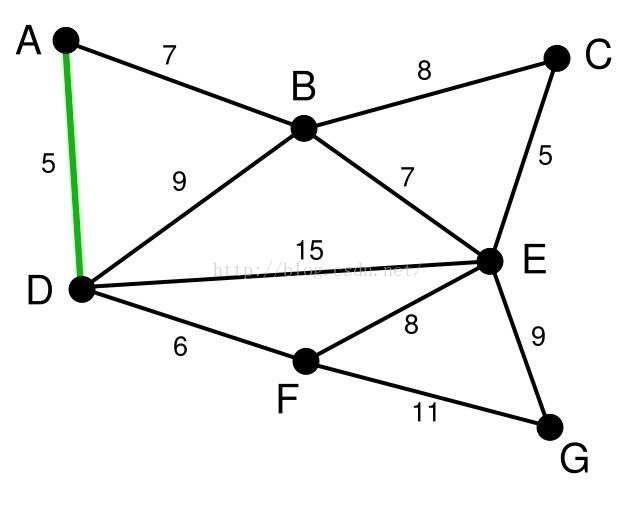
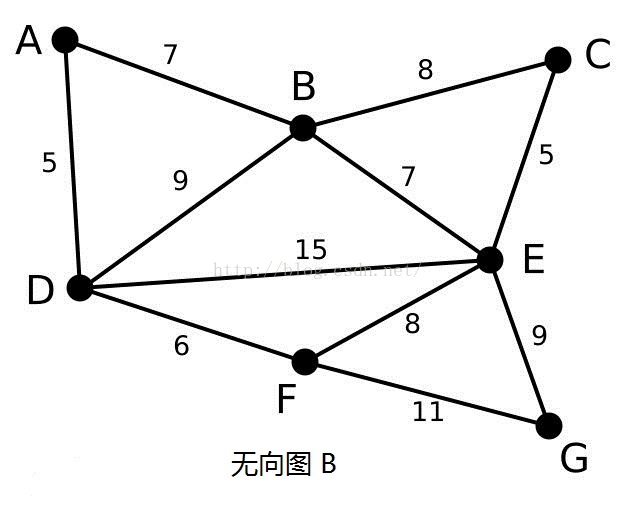
(2)将所有的边加入集合S，生成树的边集Edge一开始为空

(3)从S中拿出一条最短的边(u,v)，如果(u,v)不在同一棵树内，则连接u,v合并这两棵树，同时将(u,v)加入生成树的边集Edge

(4)重复(3)直到所有点属于同一棵树，边集E就是一棵最小生成树

从小到大排序

|  |  |  |
| --- | --- | --- |
| AD | 5 | AD不在同一棵树，加入Edge |
| CE | 5 | CE不在同一棵树，加入Edge |
| DF | 6 | DF不在同一棵树，加入Edge |
| AB | 7 | AB不在同一棵树，加入Edge |
| BE | 7 | BE不在同一棵树，加入Edge |
| BC | 8 | BC已在一棵树中 |
| EF | 8 | EF已在一棵树中 |
| BD | 9 | BD已在一棵树中 |
| EG | 9 | EG不在同一棵树，加入Edge |
| FG | 11 | FG已在一棵树中 |
| DE | 15 | DE已在一棵树中 |



## 【最短路径】

### 【迪杰斯特拉算法】Dijkstra

**dijkstra算法原理：**最优子路径存在。假设从S→E存在一条最短路径SE，且该路径经过点A，那么可以确定SA子路径一定是S→A的最短路径。证明：反证法。如果子路径SA不是最短的，那么就必然存在一条更短的'SA，从而SE路径也就不是最短，与原假设矛盾。

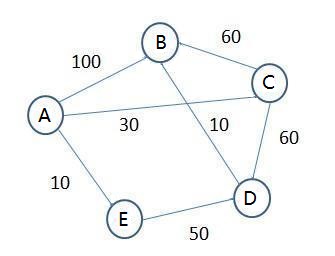
**dijkstra算法缺点：**此算法能够求出从起点到其余每个结点的最短路径，所以需要遍历所有的路径和结点，计算复杂度比较大。

**dijkstra算法流程：**

**迭代：**

**①从已知距离中选择最短的，确认为到某点的最短路径**

**②根据确认的最短路径，更新已知距离**



**step1：**首先建立两个集合

S=｛｝：表示已经找到最短路径的结点

U=｛｝：表示尚未找到最短路径的结点

**step2：**

**建立一个数组dist[i]，用于存放起点0到该结点i的最短路径**

**然后再建立一个布尔数组s[i]（初值均为0），用于表示该结点是否已经找到最短路径  
step3：具体执行部分：  
A：初始点设定。**

结点0到结点0：首先将其纳入到S集合中，dist[0]=0

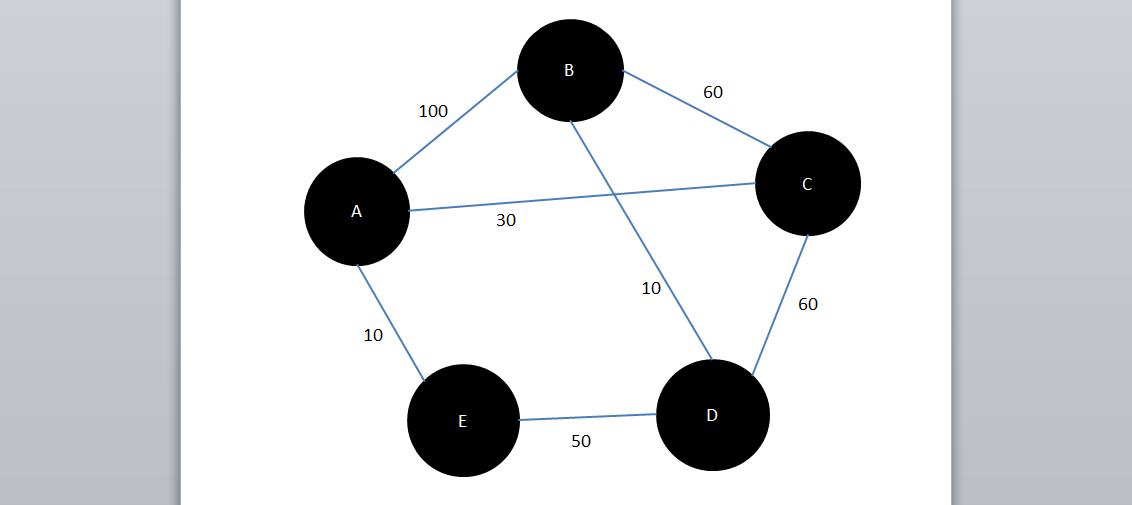
结点0到直接联通的结点：dist[1]=100,dist[2]=30,dist[4]=10

结点0到不能直接联通结点：dist[3]= ∞

然后使s[0]=1，表示已经遍历过该结点。  
**B：选取最小dist[i]。**比较dist[1],dist[3],dist[4]的长度，选择长度最短的dist[4]，并将结点4纳入到S集合中，令s[4]=1，表明0到4的最短路径已经找到，且值为10。**原因：最优子路径存在原理。由于dist[1],dist[3]均大于dist[4]，所以若选择走经过结点**1、3**到达结点4路径，无论如何也不可能找到一条小于直接从结点0到结点4的路径！(利用这个原理避免了遍历所有路径)**这个结论非常非常非常重要，是理解这个算法的关键！后面会反复用到，每一轮循环都要比较并选取最小的dist[i]。  
**C：更新dist[i]。**现在，我们开始以结点4为中心向外扩展（广度优先）。现在，结点4可以到达结点3了，也表明从结点0可以通过结点4到达结点3了。

**step4：**重复上述步骤B、C，直到U集合清空，s[i]中所有值均为1。这就表明图中所有点都找到了最短路径。

图的结构

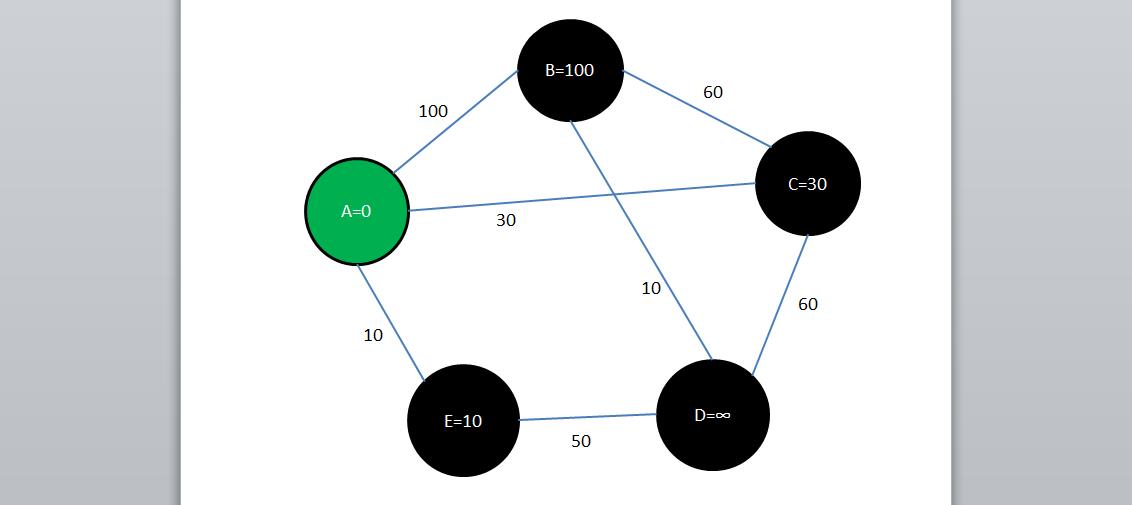


A与自己距离为0，可以确定为最短距离，s[0]=1

与A直接相连的BCE距离为:dist[1]=100,dist[2]=300,dist[4]=10

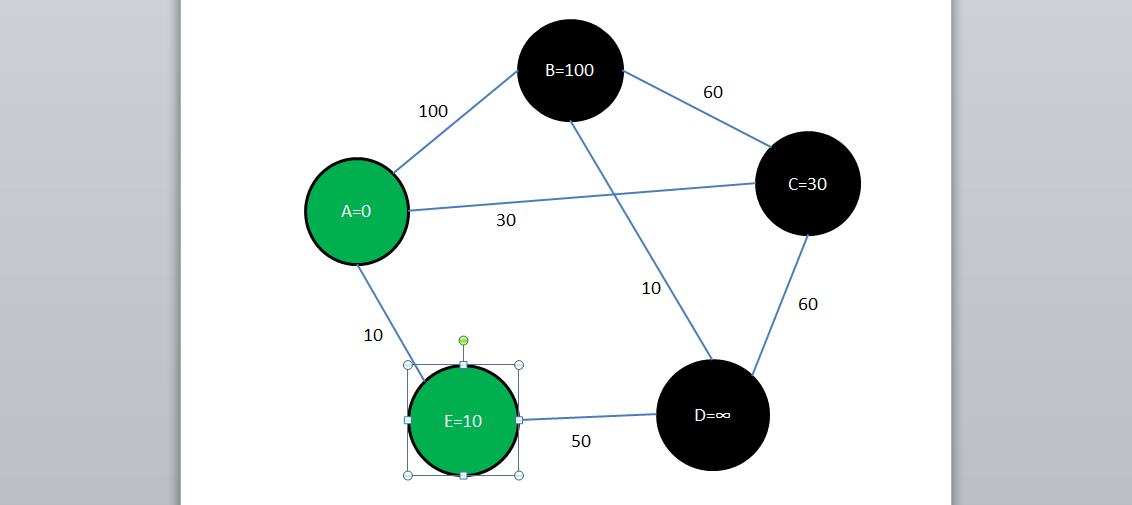
与A间接相连的D距离为：dist[3]=∞

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | A | B | C | D | E |
| dist | 0 | 100 | 30 | ∞ | 10 |
| s | **1** | 0 | 0 | 0 | 0 |



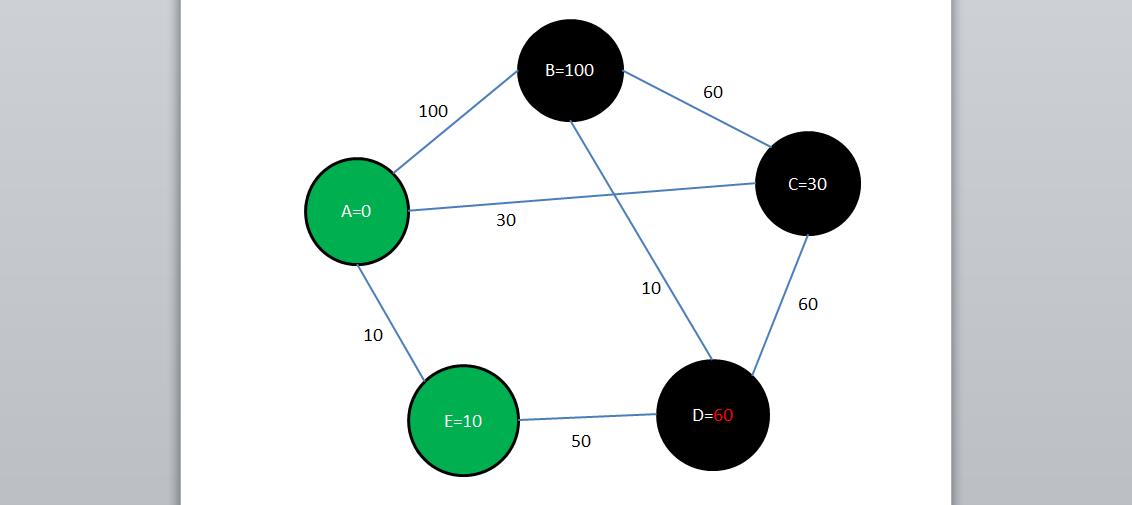
在BCDE中选择距离最短的dist[4]=10，确定为A到E的最短距离，令s[4]=1

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | A | B | C | D | E |
| dist | 0 | 100 | 30 | ∞ | 10 |
| s | 1 | 0 | 0 | 0 | **1** |



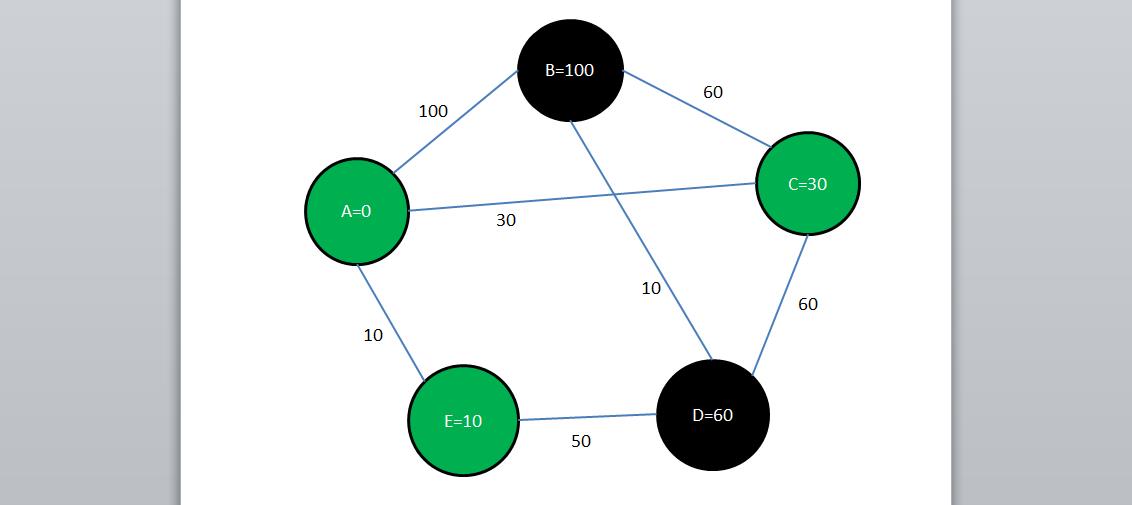
E点已通，与E直接相连的未确定的有D，将dist[3] 更新到目前已知的最小值60

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | A | B | C | D | E |
| dist | 0 | 100 | 30 | **60** | 10 |
| s | 1 | 0 | 0 | 0 | 1 |



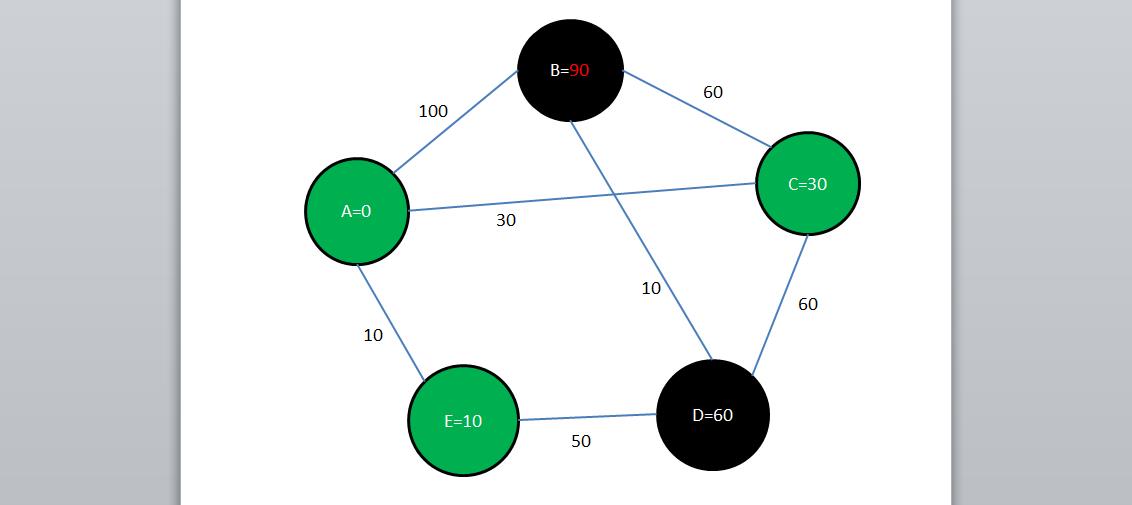
在BCD中选择距离最短的dist[2]=30，确定为A到C的最短距离，令s[2]=1

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | A | B | C | D | E |
| dist | 0 | 100 | 30 | 60 | 10 |
| s | 1 | 0 | **1** | 0 | 1 |



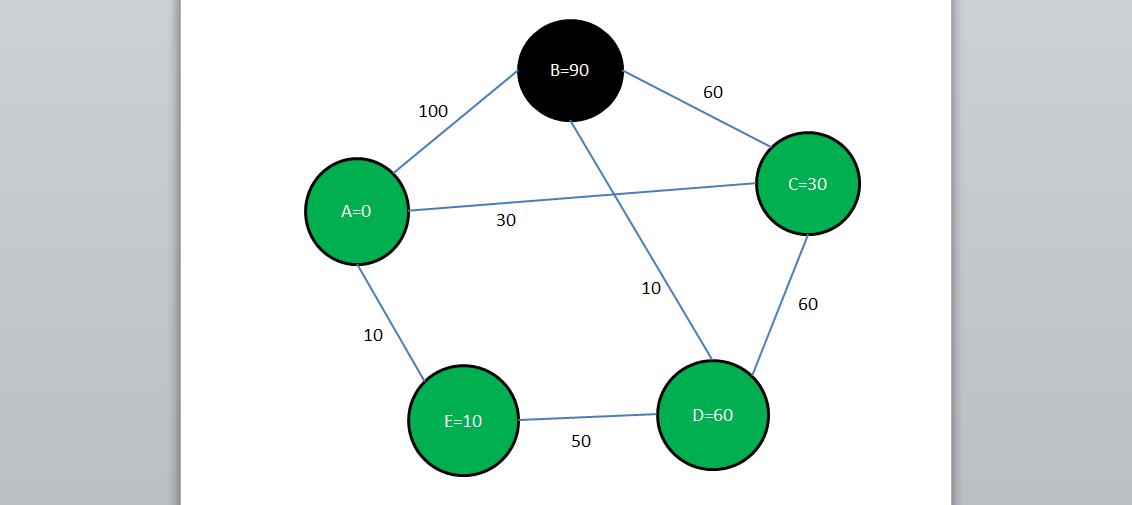
C点已通，与C直接相连的未确定的有B，将dist[1] 更新到目前已知的最小值90

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | A | B | C | D | E |
| dist | 0 | **90** | 30 | 60 | 10 |
| s | 1 | 0 | 1 | 0 | 1 |



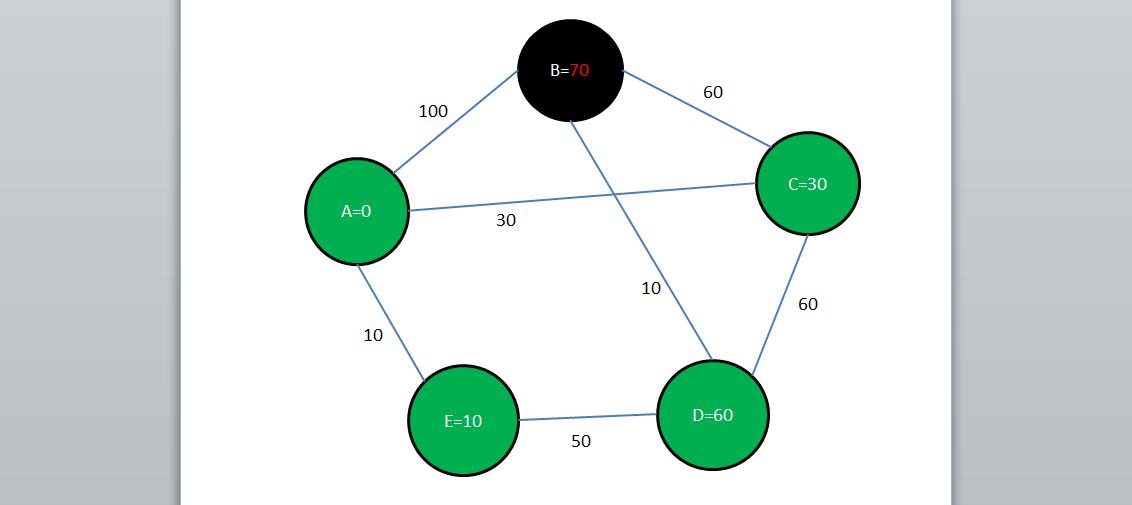
在BD中选择距离最短的dist[3]=30，可以确定为A到D的最短距离，令s[3]=1

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | A | B | C | D | E |
| dist | 0 | 90 | 30 | 60 | 10 |
| s | 1 | 0 | 1 | **1** | 1 |



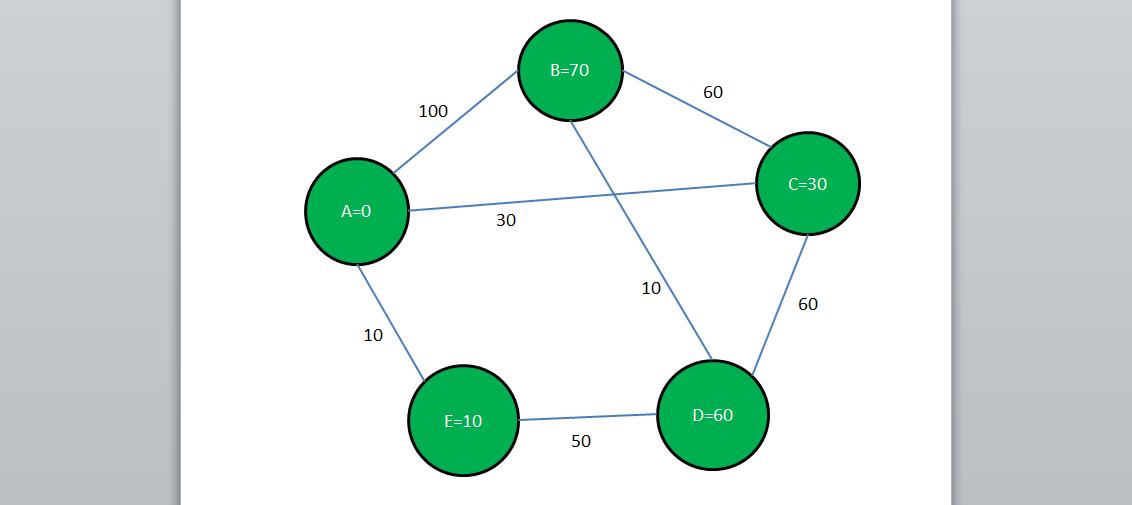
D点已通，与D直接相连的未确定的有B，将dist[1]更新到目前已知的最小值70

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | A | B | C | D | E |
| dist | 0 | **70** | 30 | 60 | 10 |
| s | 1 | 0 | 1 | 1 | 1 |



在B中选择距离最短的dist[1]=70，可以确定为A到D的最短距离，令s[1]=1

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
|  | A | B | C | D | E |
| dist | 0 | 70 | 30 | 60 | 10 |
| s | 1 | **1** | 1 | 1 | 1 |



### 【弗洛伊德算法】Floyd

for(k=1;k<=n;k++)

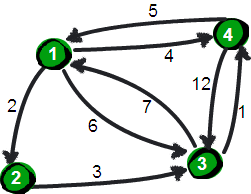
for(i=1;i<=n;i++)

for(j=1;j<=n;j++)

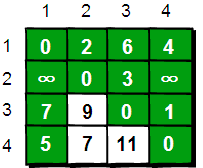
if(e[i][j]>e[i][k]+e[k][j])

e[i][j]=e[i][k]+e[k][j];

这段代码的基本思想就是：最开始只允许经过 1 号顶点进行中转，接下来只允许经过 1 和 2 号顶点进行中转……允许经过 1~n 号所有顶点进行中转，求任意两点之间的最短路程。

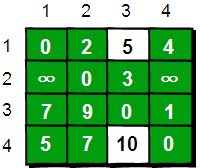
 

k=1



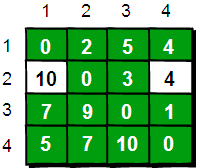
|  |  |  |  |
| --- | --- | --- | --- |
|  |  |  |  |
|  |  |  |  |
|  | 3-1-2 |  |  |
|  | 4-1-2 | 4-1-3 |  |

k=2

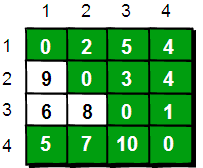


|  |  |  |  |
| --- | --- | --- | --- |
|  |  | 1-2-3 |  |
|  |  |  |  |
|  |  |  |  |
|  |  | 4-1-2-3 |  |

k=3



k=4



## 【拓扑排序】

前提：有向无环图

要求：

①每个顶点只出现一次

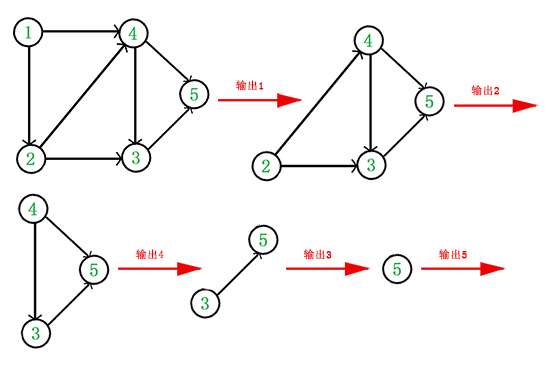
②若存在从A到B的路径，排序中A必在B前面

步骤：

①选择一个入度为零的顶点并输出

②删除该顶点，并删除以其作为起点的有向边

③重复①②直到输出全部顶点，或不存在入度为0的顶点



拓扑排序结果：1、2、4、3、5

上图每次删除顶点之后，都只有一个入度为0的点，但如果遇到多个入度为0的顶点的情况如何处理？

拓扑排序主要解决一个工程能否顺序进行的问题

AOV网：表示工程依赖顺序的网

AOE网：AOV网的边带权表示时间

## 【关键路径】

**关键路径**通常（但并非总是）是决定项目工期的进度活动序列。 它是项目中最长的**路径**，即使很小浮动也可能直接影响整个项目的最早完成时间。

顶点是事件，代表时间点，弧是活动；权值代表时间段。

事件最早发生时间etv

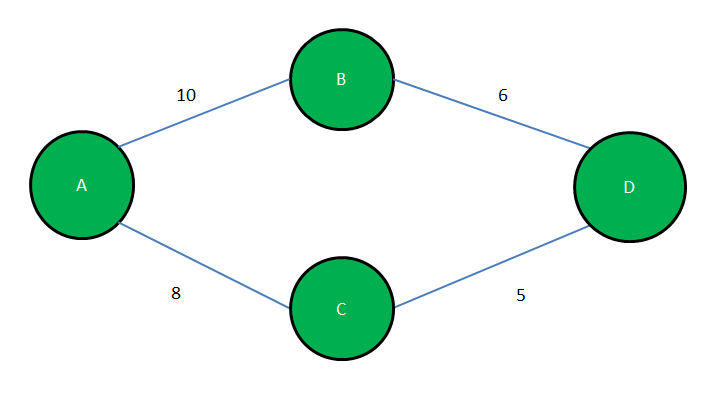
事件最晚发生时间ltv

活动最早开工时间ete

活动最晚开工时间lte

算法思路：

在求关键路径的算法中，先求出每个事件的最早完成时间。在事件最早完成的时间集合中，工程最后一步完成的时间就是我们工程完成的最优时间。**然后在工程时间最优的情况下求出每个事件最晚完成时间。**如果某个时间最早的完成时间与最晚的完成时间相同，那么该事件就是我们的关键事件，该事件就位于我们关键路径中。



最早完成时间：

|  |  |  |  |
| --- | --- | --- | --- |
| A | B | C | D |
| 0 | 10 | 8 | 16 |

可知整个工程最优的完成时间是16，从16往前倒推不影响工期的最晚完成时间：

|  |  |  |  |
| --- | --- | --- | --- |
| A | B | C | D |
| 0 | 10 | 11 | 16 |

可得关键事件是A、B、D

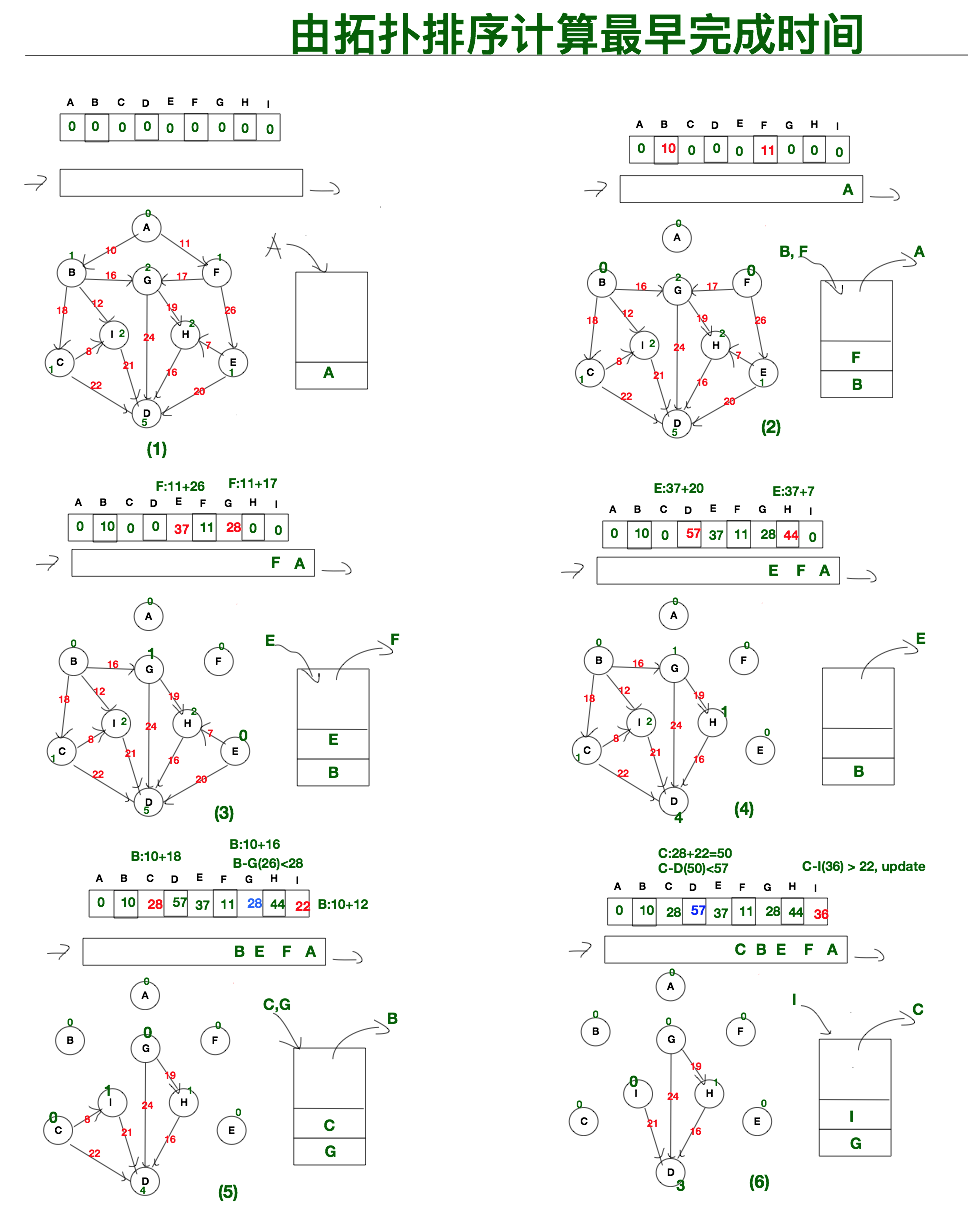
关键路径为：A→B→D

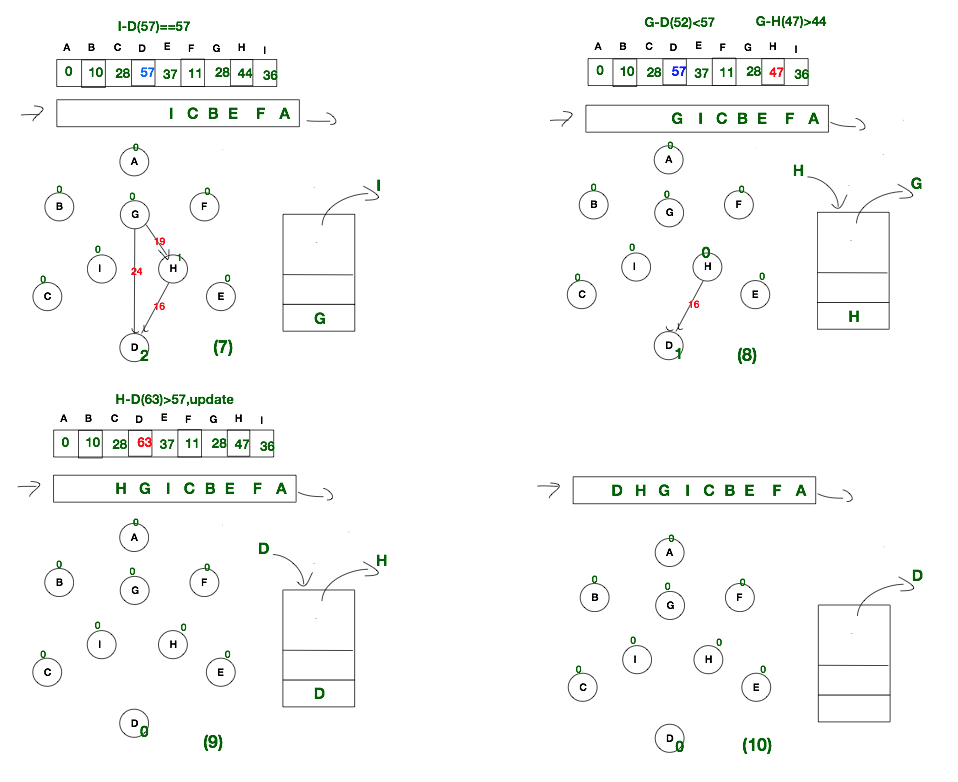
算法步骤：

①由拓扑排序计算最早完成时间：

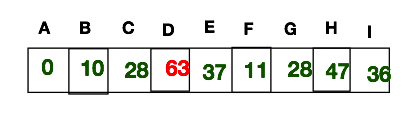
A从0开始，一直往下加照弧的权值

注意：有的点会被遍历多次，比如G，A-F-G是28，先保存28；等到A-B-G遍历时是26，与28比较，保存较大的数值。（因为事件有前后依赖关系，所以要选最大值）





最早完成时间：



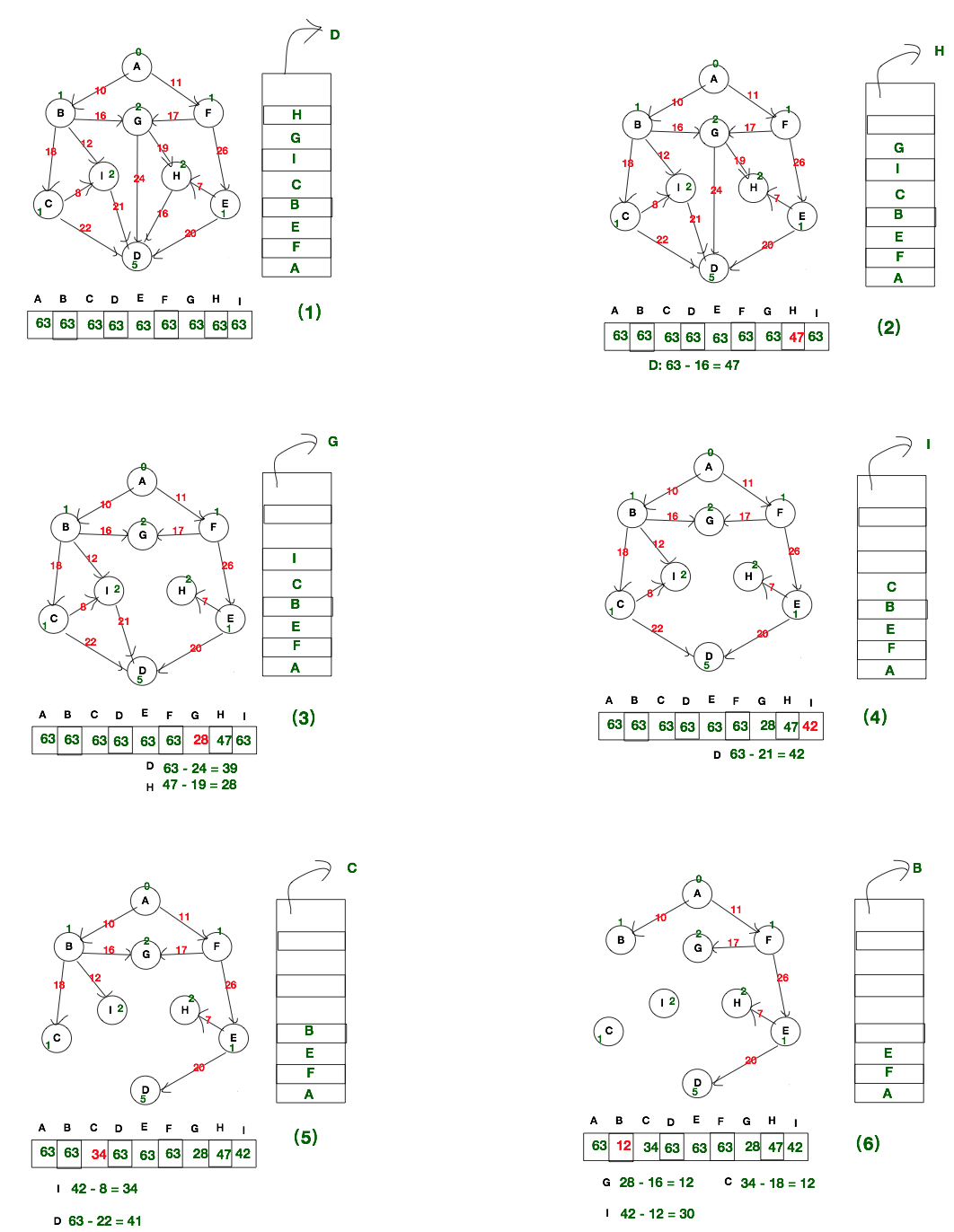
②倒推最晚完成时间：

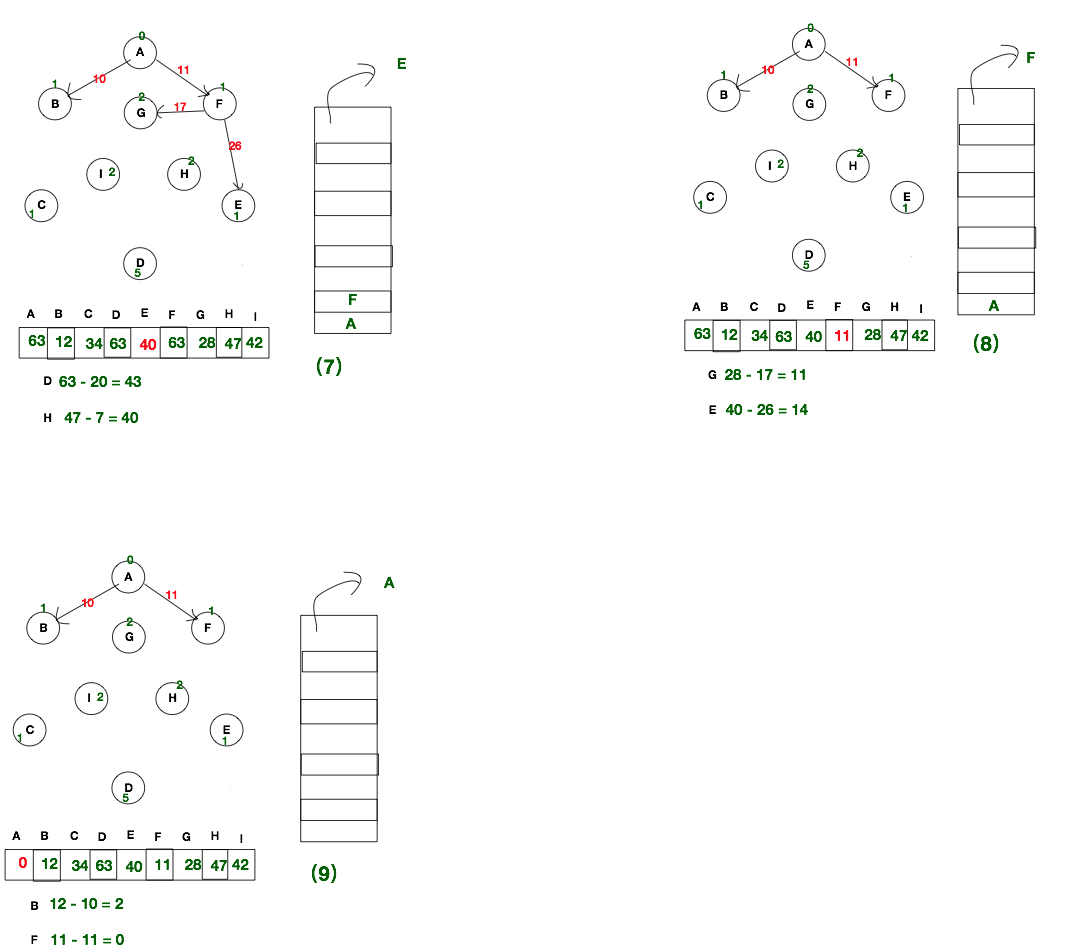
D从63开始，一直往上减去弧的权值

注意：有的点会被遍历多次，比如E：

D→E是63-20=43，先保存43；等到D→H→E遍历时是63-16-7=40，与43比较，保存较小的数值40。

因为有事件有前后依赖关系，如果保存D→E的43，E→H→D就没有足够事件完成了。

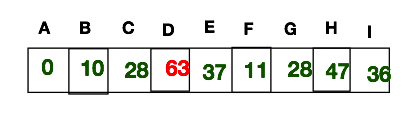




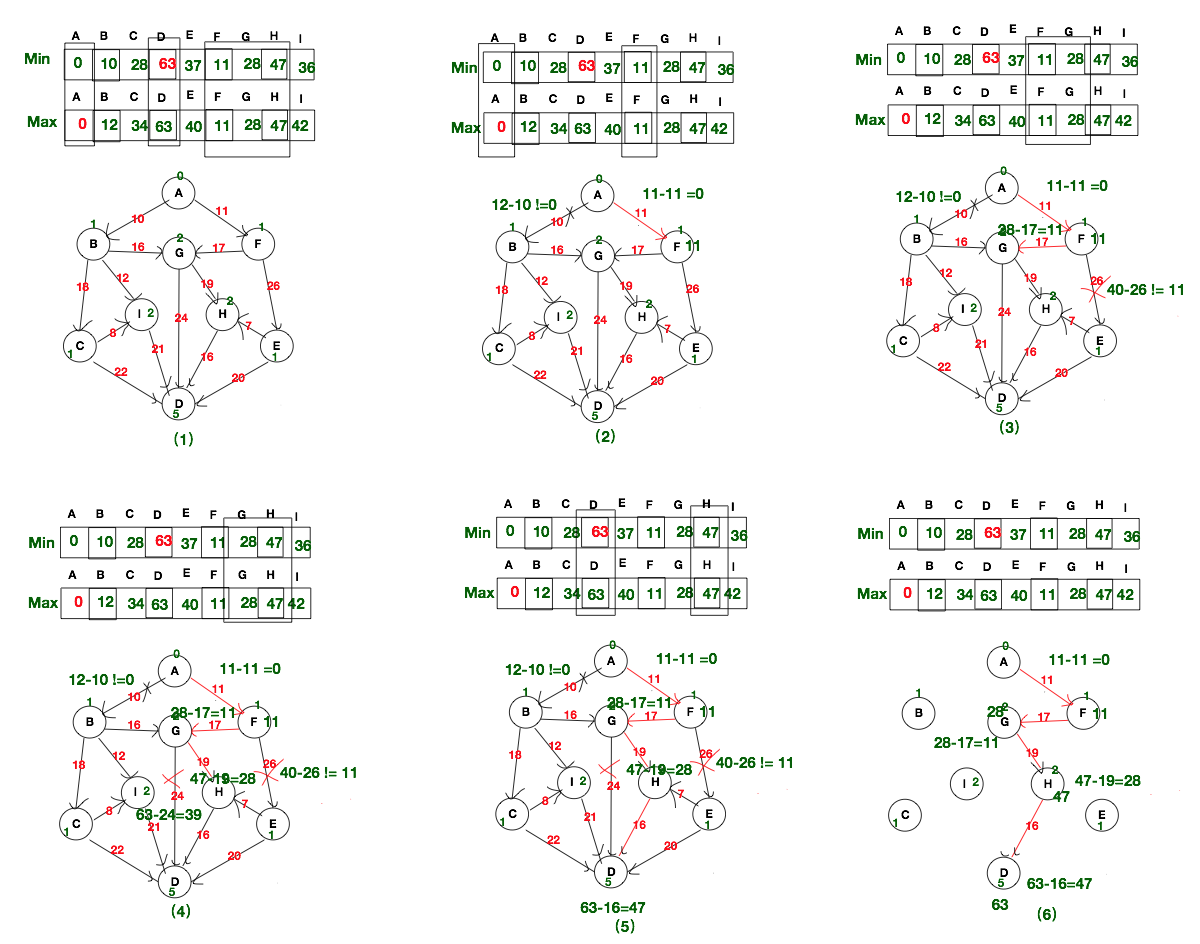
最晚完成时间：



③对比最早完成时间和最晚完成时间，找出关键路径







# 【查找】

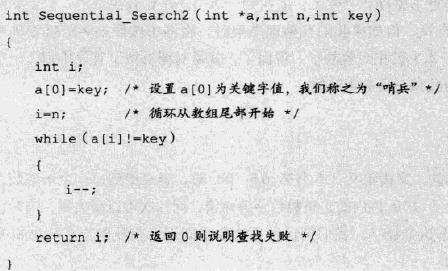
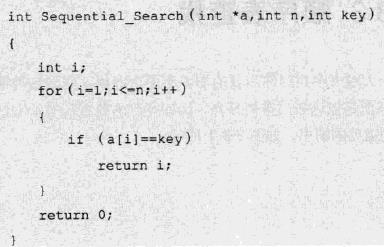
【静态查找】只有查找操作

【动态查找】查找时插入、删除元素

## 【无序表查找】

### 【顺序查找】

对比两种方法，第二种通过设置“哨兵”，避免每次判断是否越界。



## 【有序表查找】

### 【折半查找】binary search(一分为二的查找)

在有序表中，不断取中间元素作为比较对象。

复杂度：O(log2n)

//输入有序数组,查找目标

//输出下标，没有找到输出-1

int binary\_search(vector<int> array, int aim)

{

int low = 0;

int high = array.size();

int mid;

while (low<=high)//没有找到就结束循环

{

mid = (low + high) / 2;//不用太精确，可能会前后移一个数，但不会遗漏

if (aim < array[mid])

high = mid - 1;//注意-1与+1

else if (aim > array[mid])

low = mid + 1;

else

return mid;

}

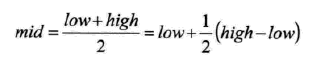
return -1;

}

### 【插值查找】interpolation search(按比例查找)

在有序表中，如果数值分布比较均匀，查找较大的数值时，从较大的下标开始查找，找到的概率更高。

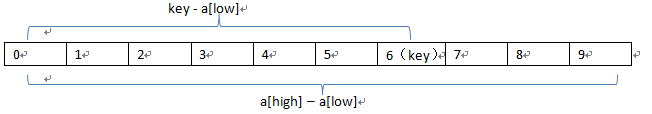
在折半查找中：



就是mid等于low加上高低下标之差的一半，现在对系数1/2进行改进：



原理是假设数值是均匀分布:



当数值接近均匀分布时有较好的查找效果

### 【斐波那契查找】fibonacci search(只涉及加减法的查找)

斐波那契数列（后一个数与前一个数的比值越来越接近黄金比例0.618）：

1 1 2 3 5 8 13 21 34 55 89 …

与折半查找相似，但是每次“折半”的比例不是1/2而是≈0.618。例如89可以分成55和34两段查找，55可以分成34和21两段……这些数字都在斐波那契数列中，只需要加减法就可以得出，不需要运用乘除法。

分割示意图：

|  |  |  |  |
| --- | --- | --- | --- |
| 89 | | | |
| 55 | | 34 | |
| 34 | 21 | 21 | 13 |

……

【算法要点】

①生成斐波那契数列

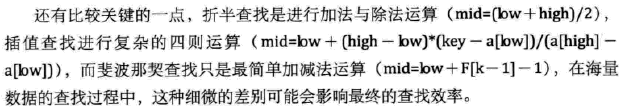
②找到斐波那契数列中最接近且大于有序表长度的数值

③把有序表长度补全到那个数值

④利用斐波那契数列进行分割查找

⑤通过判断下标是否大于有有序表的长度，可知元素数补全的还是属于有序表的

【总结】



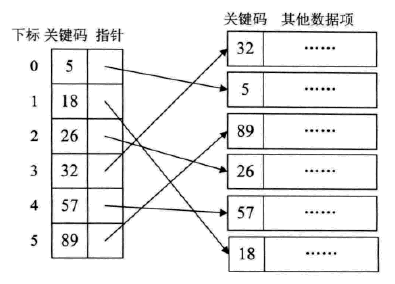
## 【索引查找】

【索引】把一个关键字与它对应的记录相关联的过程

一个索引由若干个索引项构成，每个索引项至少应包含关键字以及它对应记录的位置。索引技术是组织大型数据库以及磁盘文件的一项重要技术。

### 【线性索引】将索引项集合组织为线性结构，也称为索引表

#### 【稠密索引】将数据集的每一个记录，对应一个索引项



稠密索引要对应的可能是成千上万的数据，索引项有序排列，就能使用各种有序表的查找方法。

#### 【分块索引】将数据集分块，使分块有序，每一块对应一个索引项，减少索引项的数量。

特点：

①块内无序：有序将会付出大量时间与空间代价

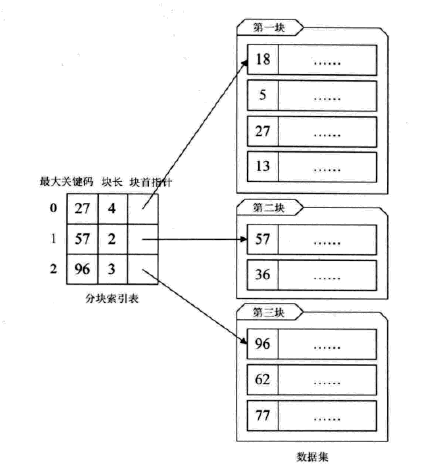
②块间有序：例如第一块内的关键字要小于第二块内的关键字

索引项的数据有：

①最大关键码

②块中记录个数

③块首数据元素指针



#### 【倒排索引】





一个未经处理的数据库中，一般是以文档ID作为索引，以文档内容作为记录。而Inverted index 指的是将单词或记录作为索引，将文档ID作为记录，这样便可以方便地通过单词或记录查找到其所在的文档。

正排：通过ID找内容

倒排：通过内容找ID

### 【树形索引】

#### 【二叉排序树】binary sort tree（二叉查找树、二叉搜索树）

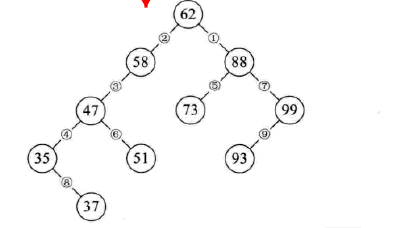
【特点】

①左子树上所有值均小于根节点

②右子树上所有值均大于根节点

③左右子树也是二叉排序树

构建过程：62 88 58 47 35 73 51 99 37 93



对它进行中序遍历，能够得到从小到大的有序排列

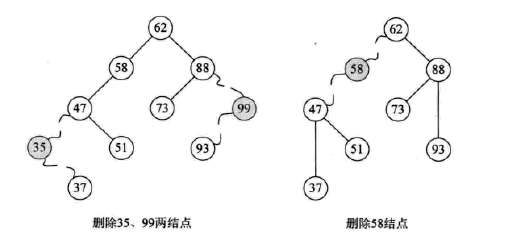
【插入操作】

如果要加入59，放到58的右子树；如果要加入57，放到51的右子树（插入一定会成为叶子）

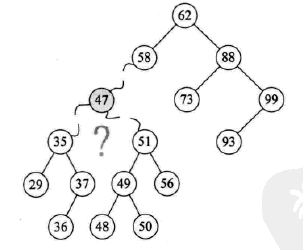
【删除操作】

①叶子——直接删除

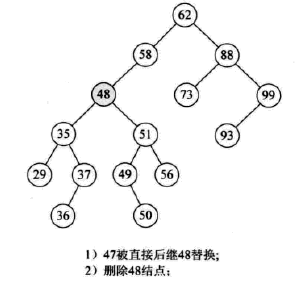
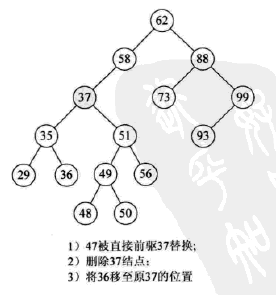
②节点只有左子树或者只有右子树——独子继承



③节点既有左子树又有右子树

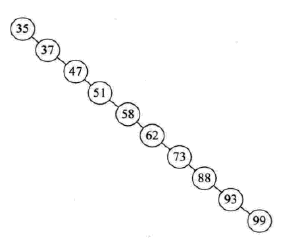


考虑到不要对树结构进行太大改变，寻找可以代替47的点——37、48，这是它在中序遍历中的前驱或后继。



【优点】插入删除方便，查找次数等于节点所在层数，最多不会超过树的深度。

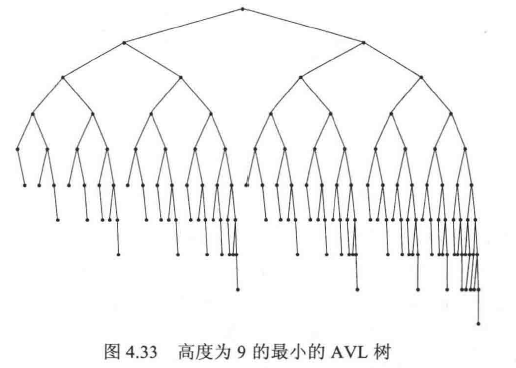
【缺点】二叉排序树的形状不确定，有可能出现极端情况：

插入顺序一直是从小到大，出现了极端右斜的树。

#### 【平衡二叉树】AVL

【特点】

AVL是二叉排序树的一种，其每一个节点的左右子树高度最多相差1。

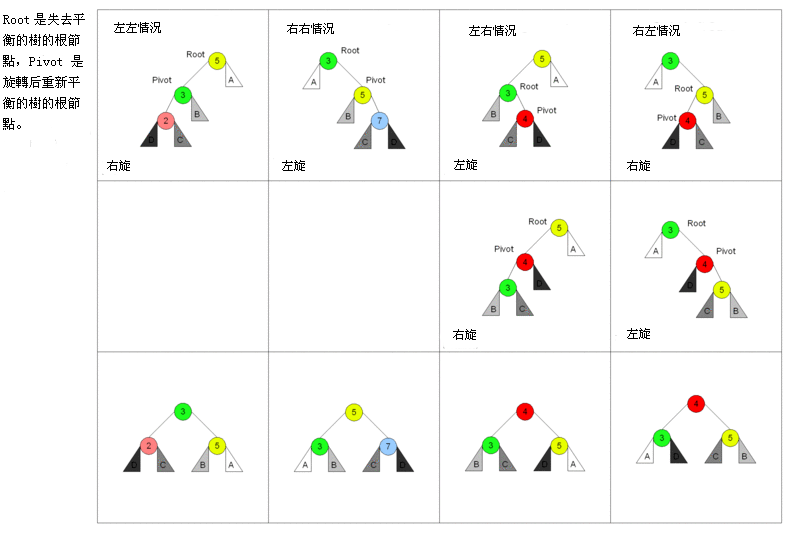


【平衡因子】balance factor BF

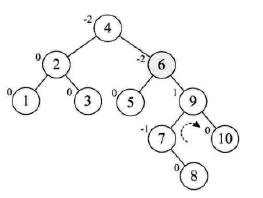
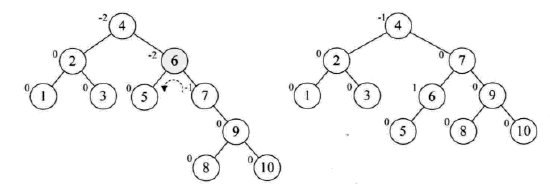
左子树高减右子树高等于结点的平衡因子，平衡二叉树每个结点的平衡因子是1、0、-1。

【插入】

每插入一个元素后，都检测是否破坏二叉树的平衡性。如果是，找最小不平衡子树并对其进行调整。



例如右左情况：

【删除】

从AVL树中删除，可以通过把要删除的节点向下旋转成一个叶子节点，接着直接移除这个叶子节点来完成。因为在旋转成叶子节点期间最多有log n个节点被旋转，而每次AVL旋转耗费固定的时间，所以删除处理在整体上耗费O(log n) 时间。

#### 【B树】自平衡多路查找树

【特点】保持平衡

每个节点的孩子可以多于2个，每个节点可以存储多个元素。由于它是查找树，元素之间存在某种特定的排序关系。

##### 【2-3树】每个节点都具有2或3个孩子（或0个）

2结点：包含1个元素和2个孩子（或者0个孩子）。左子树元素小于该元素，右子树元素大于该元素）

3结点：包含一大一小2个元素和3个孩子（或者0个孩子）。左子树元素小于较小的元素，右子树元素大于较大的元素，中间子树元素在较小元素和较大元素之间。

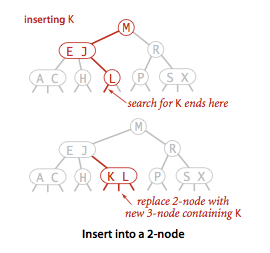
所有叶子都在同一层次上。

【插入】不论插入什么总会插到叶子中——要么比结点元素大，要么比结点元素小，所以会插到叶子中

情况总结：

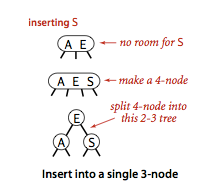
①插入到2结点

无论2节点的父节点是什么，2结点都可以变成3结点



②插入到3结点

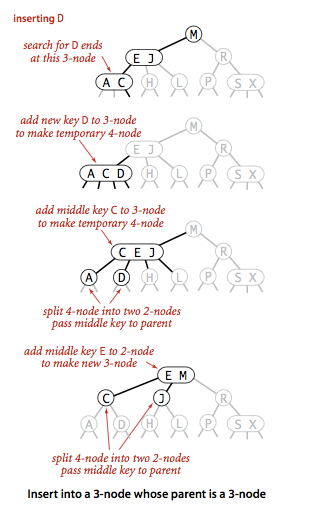
没有父节点



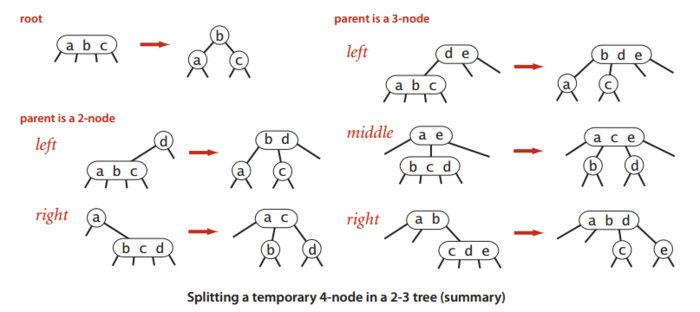
父结点是2结点

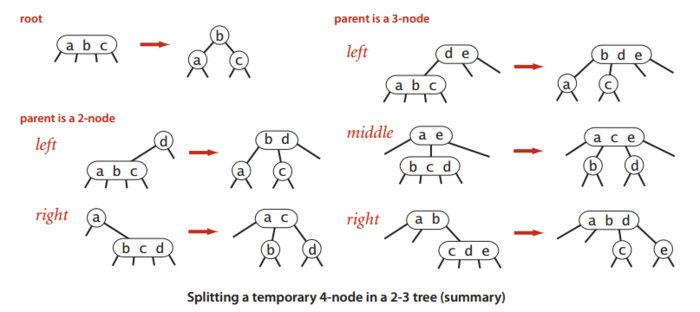


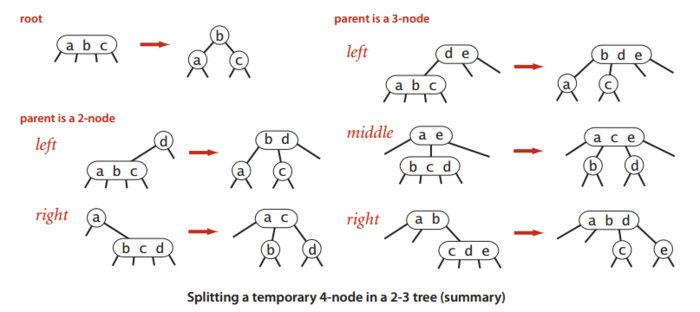
父结点是3结点



【四结点的分解】当插入到3结点时，不可避免会出现4结点的分解，处理都是把元素提到上面的结点中。



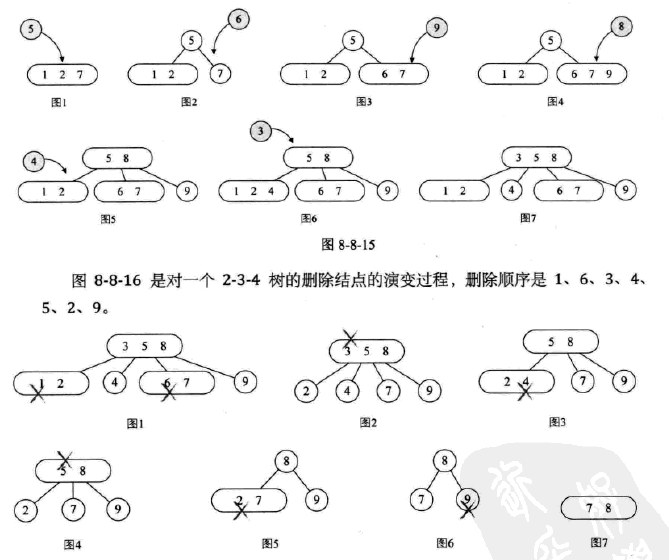




【删除】

没有完全弄懂

##### 【234树】

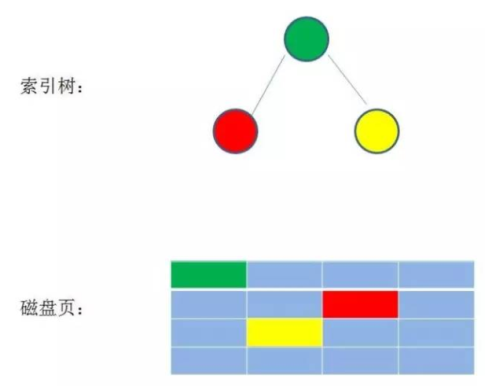


##### 【B树】内存与外存

B树是一种自平衡的多路查找树，23树和234树都是B树的特例，23树是3阶B树，234树是4阶B树

B树的插入和23树、234树类似，不过可能阶数很大。

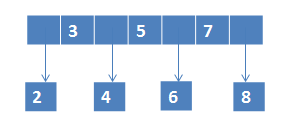
可以说，B树的数据结构就是为内外存的数据交互而设的，这种数据结构常被应用在[数据库](https://zh.wikipedia.org/wiki/%E6%95%B0%E6%8D%AE%E5%BA%93)和[文件系统](https://zh.wikipedia.org/wiki/%E6%96%87%E4%BB%B6%E7%B3%BB%E7%BB%9F)的实现上。

在数据库或者文件系统中，数据量比较大，不能把整个索引从磁盘加载大奥内存，所以只能逐一加载磁盘页，磁盘页对应着索引树的结点。

最坏情况下，磁盘IO次数等于树的高度，B树的特征之一是结构“矮胖”，减少加载磁盘数据的次数。相比磁盘IO速度内存中的耗时几乎可以忽略。

磁盘储存

页面1



页面2 页面3 页面4 页面5

加入不同节点在硬盘的不同页面，为了遍历所有元素，就要往返与不同页面，中序遍历：

2 3 4 5 6 7 8

页面2 页面1 页面3 页面1 页面4 页面1 页面5

为了解决B树重复遍历的情况，设计了B+树

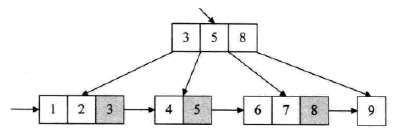
##### 【B+树】

B+树严格来说已经不是树结构了

【特点】

所有叶子结点就已经包含了全部信息

分支结点可以看成是索引



这个结构既有利于查找，又有利于遍历，但是会牺牲内存空间。

这个结构特别适合带有范围的查找（如查找2~5）

### 【散列表查找】哈希表 hash

【散列存储技术】

存储位置 = f（关键字）

关键在于在存储位置和关键字之间建立一个确定的关系f，把它称为散列函数，又称哈希函数。

【步骤】

①存储：通过散列函数计算散列地址，存储该记录。（输入记录，输出地址）

②查找：通过散列函数计算散列地址，读取该记录。（输入记录，输出地址）

两步用的都是同一个散列函数，所以说，散列技术既是一种存储方法，也是一种查找方法。

【关键问题】

设计一个简单、均匀、存储利用率高的函数，以及解决冲突（不同关键字却生成相同的地址，这种情况是不能完全避免的，只能让冲突减少。不同关键词在散列函数的作用下生成相同的地址，称之为“同义词”。）

【负载因子】

负载因子 = 填入表中记录个数 / 哈希表长度

负载因子表示哈希表中的记录密度，负载因子越大，产生冲突的可能性越大。

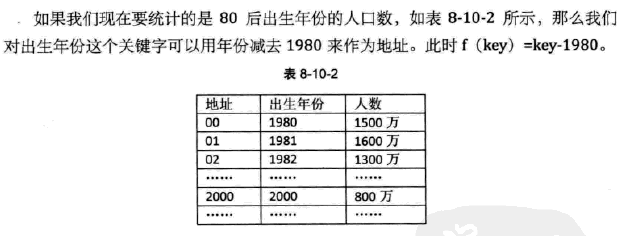
【复杂度】

如果没有冲突，时间复杂度是O(1)。有冲突的话因解决冲突的方法而异。

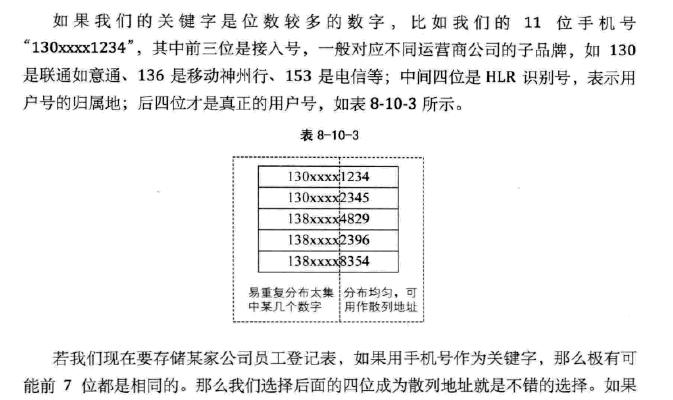
散列表的平均查找长度取决于负载因子，而不是记录的数量。不管记录个数有多大，总能找到适合的负载因子，以便限制平均查找长度，使得时间复杂度是O(1)。通常会设置散列表空间比查找集大，牺牲空间换取时间。

#### 【构建散列函数】

【直接定值法】



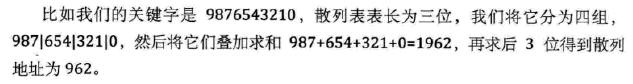
【数字分析法】



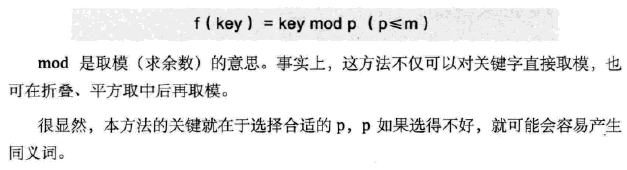
【平方区中法】



【折叠法】



【除留余数法】



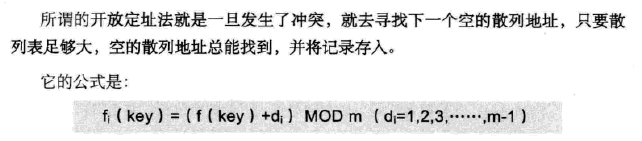


【随机数法】



#### 【处理散列冲突】

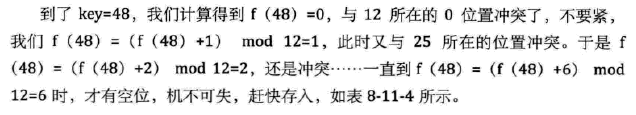
【开放定值法】线性探测



线性探测找的处理为：从发生冲突的位置开始，依次继续向后探测，直到找到空位置为止。

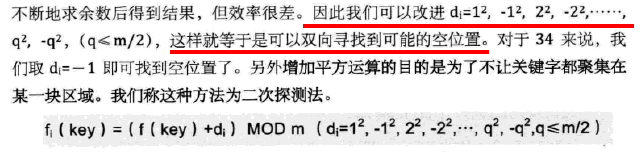
如：





一旦发生哈希冲突，所有的冲突连在一起，容易产生数据“堆积”，即：不是同义词却争夺同一个地址。

【二次探测法】双向寻找空位置

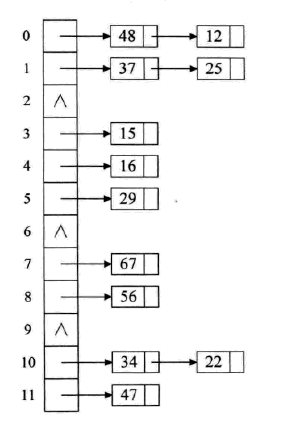


【散列函数再造法】

实现准备多个散列函数，每当发生冲突，就换一个散列函数计算。

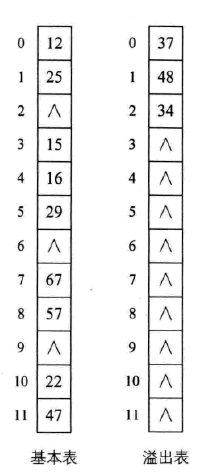
【链地址法】

所有关键词为同义词的记录，都存在一个单链表中。



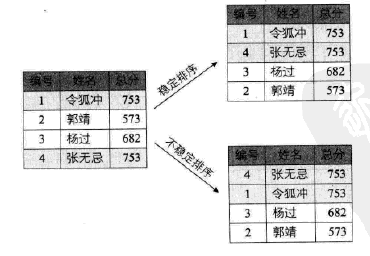
【公共溢出区法】

同义词存到另一个表中，查找时，先与基本表比对，匹配则返回，不匹配则到溢出表进行顺序查找。



# 【排序】

【稳定排序与不稳定排序】

未排序时，令狐冲在前，张无忌在后。

按总分排序之后，两人分数相等，令狐冲仍然在张无忌的前面，就是稳定排序。

【内排序与外排序】

内排序：这个排序过程在内存中进行

外排序：记录个数过多无法一次性加载到内存，排序过程中要在内外存之间进行数据交换。

## 【错误使用循环排序的一个例子】

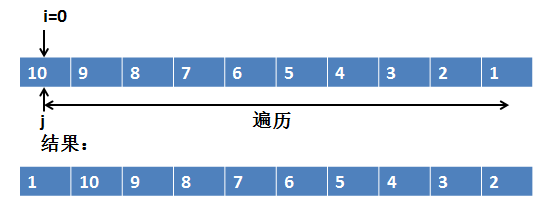
vector<int> array = { 10, 9, 8, 7, 6, 5, 4, 3, 2, 1 };

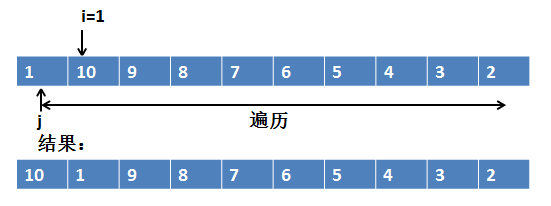
for (int i = 0; i < array.size(); i++)//i从0开始

for (int j = 0; j < array.size(); j++)//j从0开始是错误的，应该j=i+1

if (array[i] > array[j])

swap(array[i], array[j]);





/\*

当i=0时，从0开始遍历j会把最小值移到数组首位

当i=1时，j=0，就会把之前的操作又换回来，错误由此产生

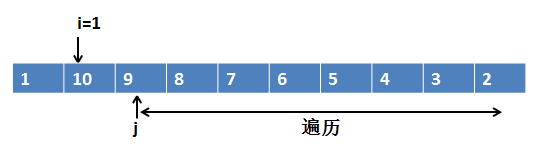
所遍历j时，每次应从j=i+1开始

因为i之前已经排好序了

\*/

【这个错误告诉我们，要正确处理好已排好序的元素，不要再去遍历他们】

【正确做法】int j = i + 1



## 【交换排序】（i++，j++）

【思想】通过遍历比较，把小的值提到前面

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

for (int j = i+1; j < array.size(); j++)//j从i+1开始

if (array[i] > array[j])

swap(array[i], array[j]);

【时间复杂度】O(n2)

## 【冒泡排序】bubble sort （i++，j--）

【思想】两两比较相邻的数值，如果反序则交换。

for (int i = 0; i < array.size() - 1; i++)

//i遍历0到8，不用到9是因为只剩最后一位时已经排好序了

for (int j = array.size() - 1; j > i; j--)

//j是从大到小，注意如果j >= i，那么j=i时，array[j - 1]会越界

if (array[j] < array[j - 1])

swap(array[j], array[j - 1]);

【优化】如果还没有遍历完，就已经排好序了，那么后面的遍历、比较都是浪费的

//输入待排序数组

//顺序按下标从小到大

void BubbleSort(vector<int> &array)

{

bool flag = true; //

for (int i = 0; i < array.size() - 1 && flag; i++)//增加对flag的判断

{

flag = false; //

for (int j = array.size() - 1; j > i; j--)

{

if (array[j] < array[j - 1])

{

swap(array[j], array[j - 1]);

flag = true; //

}

//如果在某一轮中完全没有交换，说明已经排好了，就不需要后面的遍历了。

}

}

}

【时间复杂度】O(n2)

## 【简单选择排序】记录下标，确定之后再交换数据

【思想】减少频繁交换，比较大小后只是记录下标，不作交换，确定位置之后，只移动一次就完成相关元素的排序。

//输入待排序数组

//顺序按下标从小到大

void SelectSort(vector<int> &array)

{

int min;

for (int i = 0; i < array.size() - 1; i++)//只剩一个元素的时候不需要排序

{

min = i;

for (int j = i+1; j < array.size(); j++)

{

if (array[i] > array[j])

min = j;

}

if (min != i )

swap(array[i], array[min]); //只有确定了位置才进行交换，最多9次

}

}

【时间复杂度】O(n2)

## 【直接插入排序法】straight insertion sort

【思想】将新纪录插入到有序表中，有序表长度加一。

//输入待排序数组

//顺序按下标从小到大

void InsertSort(vector<int> &array)

{

int temp;//辅助存储空间

for (int i = 1; i < array.size(); i++)//i前面是已经排好序的，i从1开始

{

temp = array[i];

for (int j = i; j >= 0; j--)

{

if ( j == 0 || array[j - 1] < temp )//与当前位置j的前一位比较大小

{

array[j] = temp;//将temp插入当前位置j

break;//完成插入后可以结束循环

}

else

array[j] = array[j - 1];//移一位

}

}

}

【时间复杂度】O(n2)

在基本有序时能获得较高的效率

## 【希尔排序】缩小增量排序（增量指的是步长）

【思想】

如果一个数列“基本有序”，插入排序效率高，即可以达到[线性排序](https://zh.wikipedia.org/w/index.php?title=%E7%BA%BF%E6%80%A7%E6%8E%92%E5%BA%8F&action=edit&redlink=1)的效率。那么就先让数列基本有序。

数列相隔某个增量，分成多个子序列，分别进行插入排序；增量是不断缩小的，直到等于1。

如果一个数列以步长5进行了排序，然后再以步长3进行排序，那么该数列不仅是以步长3有序，而且是以步长5有序。最后以步长1进行了排序，保证了整体有序。

【步长序列】

步长的选择是希尔排序的重要部分，只要最终步长为1任何步长序列都可以工作。

作者最初建议步长选择为n/2并且对步长取半直到步长达到1。

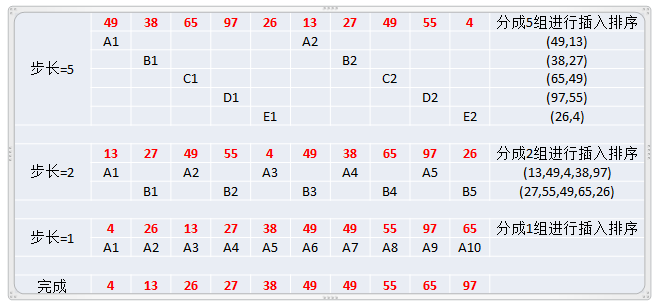
比如有长度为10的数组，步长序列为：

10/2 5

5/2 2

2/1 1

例如待排序数组：49   38   65   97   26   13   27   49   55   4



步长的选择目前还是一个数学难题，但研究表明以下公式计算步长可以获得不错的效果：



【时间复杂度】O(n2/3)，优于O(n2)

插入排序是按遍历减小到1

希尔排序是按步长序列减小到1

## 【堆排序】heap sort

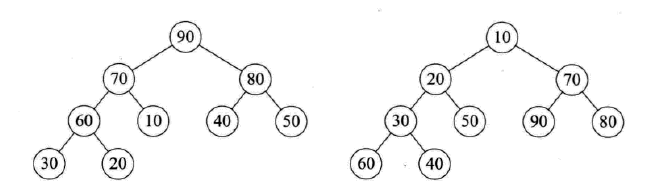
【思想】从n个数中找到最小的数，要比较n-1次，可惜这样的操作没有把所有比较结果记录下来，在后一趟的比较中，其实有许多比较在前一趟就已经比较过了。

### 【堆】数据结构

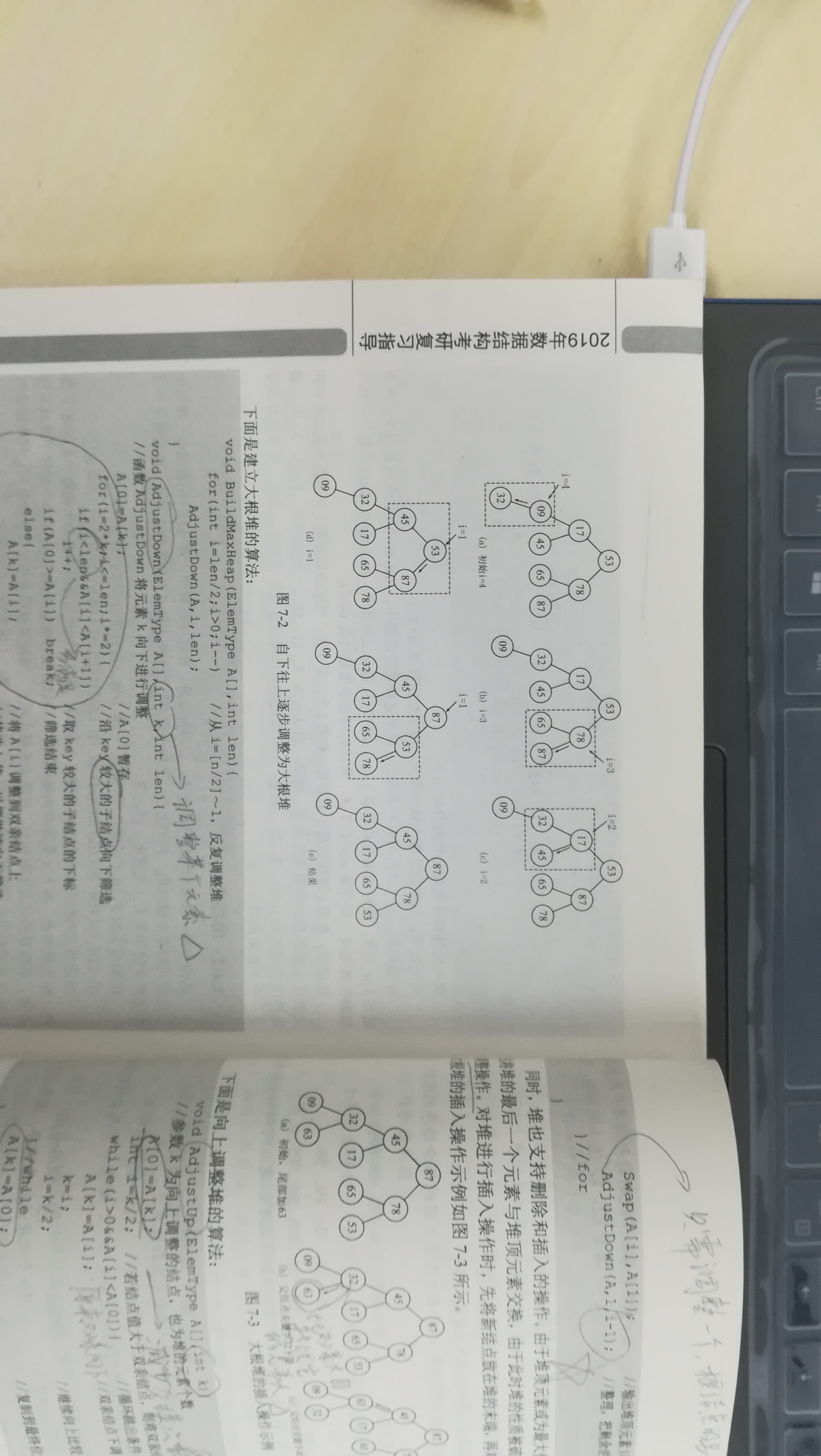
堆是具有以下性质的完全二叉树：

每个节点的值都大于或等于左右孩子的值——大顶堆、最大堆、大根堆

每个节点的值都小于或等于左右孩子的值——小顶堆



### 【构造初始堆】无序数组→大顶堆



①层序遍历构建完全二叉树

②找到不是叶子的最后一个节点（小于n/2的最大整数，如图9/2=4）

③与子女比较，把大的交换当前节点（交换后可能破坏下一级的堆，于是采用红字的方法构建下一级的堆，直到以该节点为根的子树成堆为止）

④向前依次对各节点（9/2 ~ 1）重复操作

//向前依次遍历（n/2 ~ 1）节点

void BuildMaxHeap(vector<int> &array)

{

//在数组头插入一个元素，让数组下标和层序遍历的序号相等

array.insert(array.begin(), 0);

for (int i = (array.size() - 1) / 2; i > 0; i--)//从非叶子的最后一个节点开始

AdjustDown(array, i);

//去掉数组头的占位元素

array.erase(array.begin());

}

//当前节点i与较大的一个子节点比较，不断向下调整，使i为根节点的子树成为大根堆

void AdjustDown(vector<int> &array, int i)

{

//child = i \* 2 意思是child是i的左孩子

for (int child = i \* 2; child < array.size(); )

{

//选取两个孩子中较大的一个，要保证 child + 1 没有越界

if (child < array.size() - 1 && array[child] < array[child + 1])

child++;

if (array[i] >= array[child])//与较大的那一个子节点比较大小

break;

else

{

swap(array[i], array[child]);

i = child;//向下更新节点

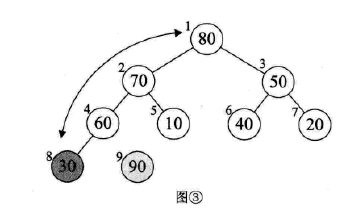
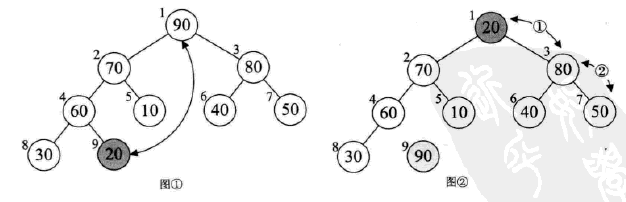
child \*= 2;//向下更新子节点

}

}

}

### 【堆排序】对已成大顶堆的数组进行排序



【思想】将待排序序列构造成一个大顶堆，此时，整个序列的最大值就是堆顶的根节点。将其与末尾元素进行交换，此时末尾就为最大值（已排好）。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。一般升序采用大顶堆，降序采用小顶堆。

void HeapSort(vector<int> &array)

{

BuildMaxHeap(array); //把数组建成大顶堆

//在数组头插入一个元素，让数组下标和层序遍历的序号相等

array.insert(array.begin(), 0);

vector<int> temp;

for (int i = array.size()-1; i > 1; i--)

{

swap(array[1], array[i]);

temp.push\_back(array.back());//temp存储array抛出的最大值

array.pop\_back();//array删除最大值

AdjustDown(array, 1);//重新调整大顶堆

//这里写成pop\_back这么麻烦，是因为放在后面的元素时已经排好序的

//不能再输入AdjustDown中

}

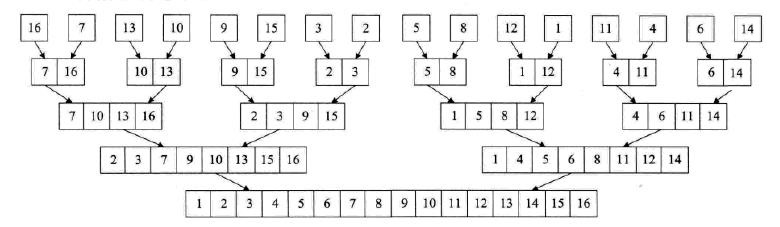
//清空array后，把temp倒序复制给array

array.clear();

array.insert(array.begin(), temp.rbegin(), temp.rend());

}

## 【归并排序】merging sort



【思想】左子序列排序，右子序列排序，归并子序列；递归调用前三步。

左子序列排序

右子序列排序

归并子序列

左子序列排序

右子序列排序

归并子序列

左子序列排序

右子序列排序

归并子序列



## 【快速排序】quick sort

**希尔排序**相当于**直接插入排序**的升级版——插入类排序

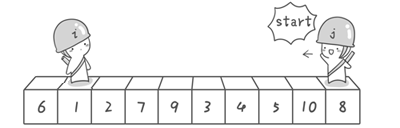
**堆排序**相当于**简单选择排序**的升级版——选择类排序

**快速排序**相当于**冒泡排序**的升级版——交换类排序

【思想】分治法

【步骤】

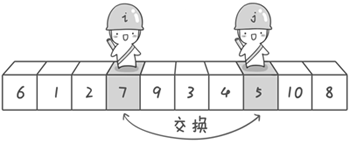
①选择数组第一个数字作为基准数



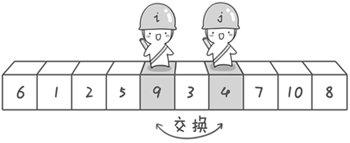
②j从右向左遍历，寻找一个小于基准数的数

i从左向右遍历，寻找一个大于基准数的数

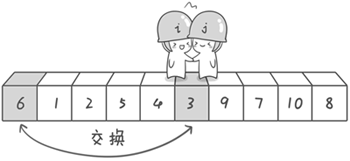
交换i和j的数字



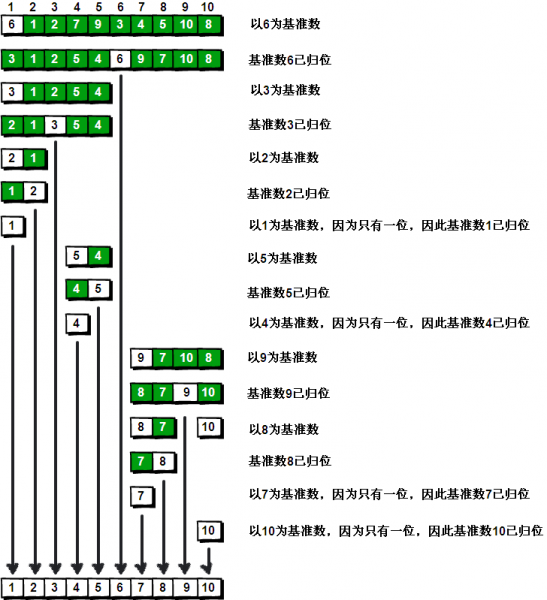
③重复上一步



④直到i=j，将6交换到i和j相遇的地方



⑤将左右两个子数组分别执行上述步骤

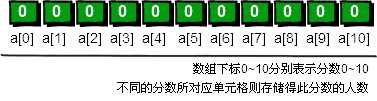


【时间复杂度】

快速排序之所比较快，因为相比冒泡排序，每次交换是跳跃式的。每次排序的时候设置一个基准点，将小于等于基准点的数全部放到基准点的左边，将大于等于基准点的数全部放到基准点的右边。这样在每次交换的时候就不会像冒泡排序一样每次只能在相邻的数之间进行交换，交换的距离就大的多了。因此总的比较和交换次数就少了，速度自然就提高了。当然在最坏的情况下，仍可能是相邻的两个数进行了交换。因此快速排序的最差时间复杂度和冒泡排序是一样的都是 **O(N2)**，它的平均时间复杂度为 **O(NlogN)**。

## 【桶排序】需要额外的储存空间（桶）

【思想】准备一个额外的“桶”，用于统计待排序的序列中各元素出现的次数，遍历桶复现经过排序的序列。



比如排序： 5 3 5 2 8

在“桶”中记录5

C:\Users\knight\Desktop\1.4.png

在“桶”中记录3

C:\Users\knight\Desktop\1.5.png

在“桶”中记录5

C:\Users\knight\Desktop\1.6.png

在“桶”中记录2、8

C:\Users\knight\Desktop\1.7.png

遍历“桶”，输出： 2 3 5 5 8

所需桶的最小空间是能容纳2~8的空间大小