邻接表中的表结点存放的不是结点数据，而是结点在表头数组中的位置！！（画图时注意！）

**第7章 图**

**图G=(V,E)** V：顶点集，**非空**有穷集 **故不存在空图**

**简单图(simple graph)**：无自环、无重边

**完全图(complete graph)**：**每对顶点**间**都有边**的简单图

**多重图(multigraph)**：有**多重边**连接到**同一对顶点**的图

**简单回路(简单环)：**若除第一个与最后一个顶点之外，其余顶点不重复出现的回路

**连通图：**任何两个顶点都是连通的（有向/无向）

**连通分量：**非连通的无向图中的极大连通子图

**无向图的极小连通子图：无向连通图的生成树**

**有向图的生成森林：**由若干棵有向树组成，[ 含有图中**全部顶点，但只有足以构成若干棵不相交的有向树（只有一个顶点的入度为0 ，其余顶点的入度均为1的有向图）的弧 ]**

**关节点/割点（用于判定重连通图）：**若连通图中某个顶点和其相关联的边被删去之后，该连通图被**分割成两个或两个以上的连通分量**

无向图的度：

**握手定理：**所有顶点的度的和是图中边的2倍，即：∑TD(vi)=2e，e为图的边数

**定理：**在任何图中，所有度数之和必为偶数，度数为奇数的顶点必定是偶数个

有向图的度：

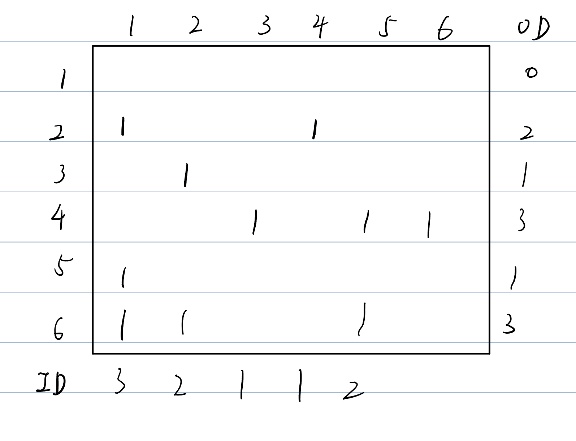
i的出度与入度之和称为**vi的度**，记为TD(vi)，即，TD(vi)=OD(vi)+ID(vi)

**握手定理：**所有顶点的度的和是图中边的2倍

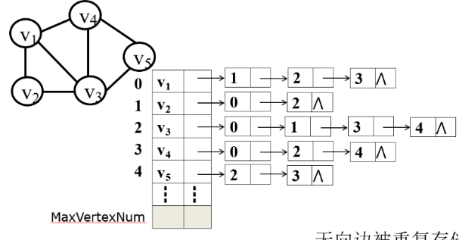
**定理：**在任何有向图中，**所有顶点的入度之和等于所有顶点的出度之和** **∑OD(vi)= ∑ID(vi)**

**图的存储结构**

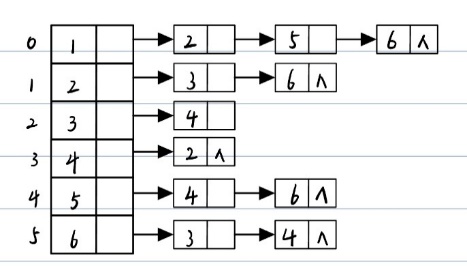
**邻接矩阵**

****无向图的邻接矩阵：**对称矩阵；第i行/列的非0元素的个数为vi的度数**

有向图的邻接矩阵：**行为出度，列为入度**

**邻接表法**

无向图



有向图：正邻接表（降序）、负邻接表（升序）

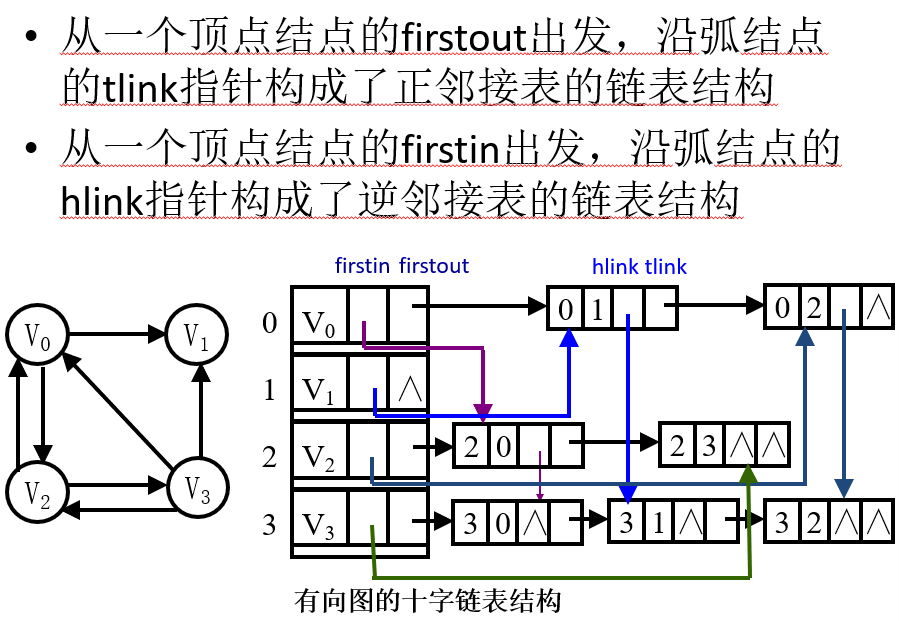
（数组下标写在外面，节点内容写在里面）

**十字链表法（只用于有向图）**

链表指针数组每个元素代表一个顶点

链表中每个节点代表一个弧

hlink：指向下一个有相同弧头的弧结点 tlink：指向下一个有相同弧尾的弧结点

****

**（弧尾-->弧头）！！**

**邻接多重表（只用于无向图）**

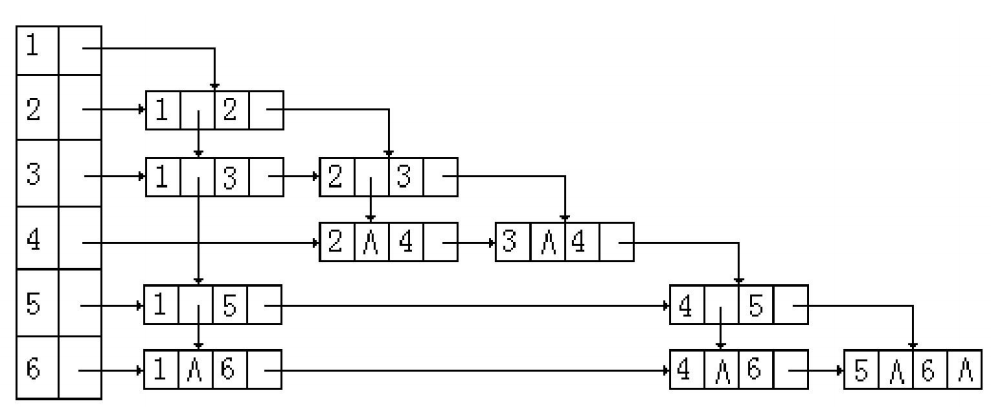
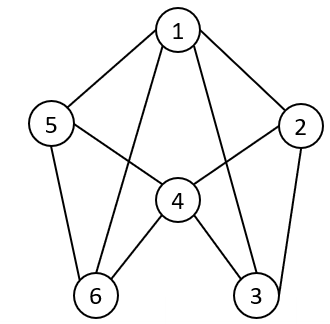
链表指针数组每个元素代编一个顶点

链表中每个节点代表一个弧

**顶点结点VexBox：** data顶点信息 firstedge：指向依附于该节点的第一条边

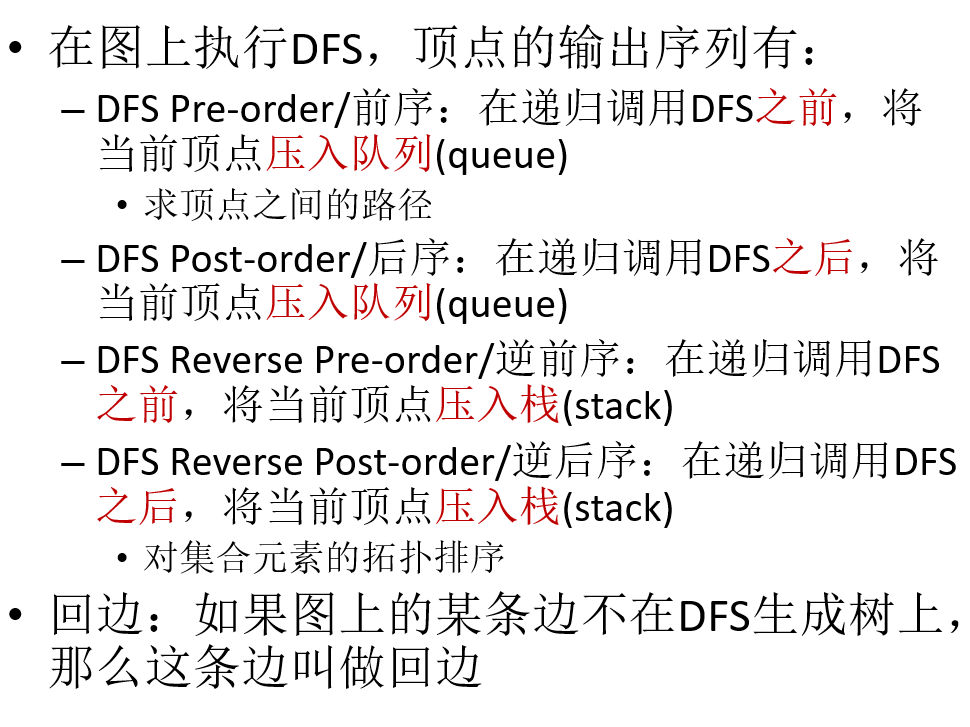
**边结点EBox：**ilink：指向i依附于顶点i的下一条边、jlink：指向j依附于顶点j的下一条边

（画起来有点像邻接矩阵，但i为列号，j为行号）



**图的遍历（按升序生成后继）**

DFS：访问顶点v0，然后依次从V0的未被访问的邻接点出发深度优先遍历图

****

BFS：访问顶点V0 ，然后依次访问V0的所有未被访问过的邻接点，之后按这些顶点被访问的先后次序依次访问它们的邻接点（按优先级队列进行递归）

**拓扑排序（有向图）**

有环时拓扑排序不存在

* 选择一个**入度为0**(没有前驱)的顶点并输出

– 删除该顶点以及从该顶点出发的(以该顶点为尾的弧)所有有向边

– 重复执行前两步，直到：图中全部顶点都已输出(图中无环) 或 图中不存在无前驱的顶点(图中必有环)

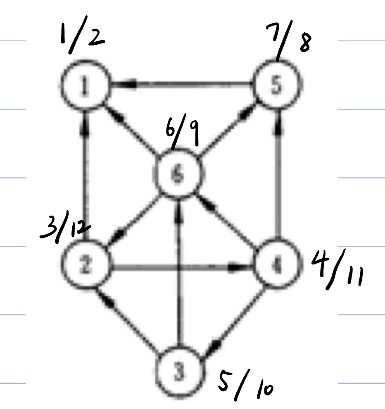
**拓扑序列不唯一：与初始时选择的入度为0的点有关。**

对于有向无环图，也可以使用DFS+栈获得拓扑排序序列（每次DFS调用结束时将当前顶点压入栈）拓扑排序=逆后序

**无向连通图的生成树**

顶点集V+图遍历过程中所经过的边的集合 = 无向图的生成树（包括深度/广度优先生成树）

**有向图的强连通分量**

Kosaraju算法：对G进行DFS，生成G的深度优先生成森林T，顶点**按退出DFS函数的顺序**进行编号，即，对顶点进行逆后序排序（拓扑排序）。按照编号顺序，对G的逆图G’做DFS，每次得到的生成树的顶点就是一个强连通分量的所有顶点。

①DFS遍历G，如图，按访问顺序生成逆后序：**243651**

（标注数对为：发现顺序/访问顺序）

②按逆后序出发对逆向图G’进行DFS得到以下强 连通分量：**1、5、2346**

**重连通图的判定**（不能有关节点）

关节点：删掉一个点及其相关联的边后G不再是一个连通图

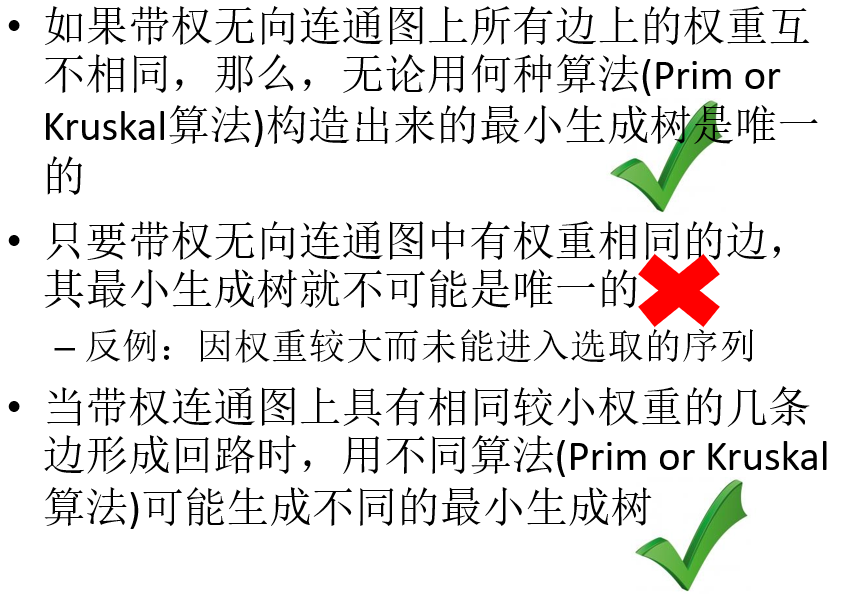
判定方法：对于G的深度优先生成树，若**根结点**有**两个及以上的分支**，必为关节点；对内部结点v，若v的**某棵子树的根或子树中的其它结点没有和v祖先相通**的（图上的）**回边**，则该结点v必为关节点

**最小生成树**

最小代价生成树：生成树各边权重之和最小

**Prim算法**：从v0出发，纳入v0，每次从纳入点到未纳入点的弧中找到权值最小的一条，纳入这条弧和相连的顶点。（起点寻路式构造）

**Kruskal算法**：从只有所有顶点的初始子图开始，每次按权值从小到大顺序尝试添加边：若不会产生回路，则添加上这条边；否则舍弃该边。直至加上 n-1 条边为止（散点加边式构造）



权重较大而未能入选的边，其权值相同对于最小生成树的唯一性没有影响

拟阵：M=(S, l)，S有穷非空集，l是S的一个非空子集族，l中每个子集的各级（减少1、2…个元素）子子集都属于l（关于子集封闭）

图的拟阵（定理）： 若G=(V,E)是一个无向图，则是个拟阵，其中集合为E，即G的边集，为**无回路**的E的子集

**用贪心算法求解拟阵的组合优化问题，能获得最优解**

贪心算法能求得最优独立集（在独立系统上定义的组合优化问题）的充分必要条件是该独立系统是一个拟阵

**AOE网的关键路径**

顶点 最早发生时间ve(j) = Maxi { ve(i)+dut(<i, j>) } i为j的所有前驱

最迟发生时间vl(j) = Mink {vl(k)-dut(<j, k>) } k为j的所有后继

**起点、汇点**的**最早发生时间等于其最迟发生时间vl(s)=ve(s)=0 vl(t)=ve(t)**

设活动ai对应弧<j, k>

弧 最早开始时间e(i) = ve(j)

**最迟开始时间 l(i) = vl(k)-dut(<j,k>)**

**计算过程：（chap7 part4 ppt P20 ）**

1. 起点事件的最早发生时间设为0

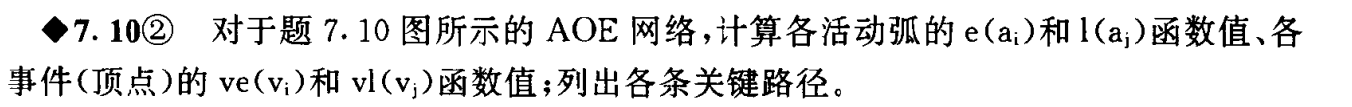
2. 对所有事件(顶点)进行拓扑排序，然后**按拓扑顺序，依次计算每个事件的最早发生时间**，**ve(j) = Maxi { ve(i)+dut(<i, j>) }**

3. 上述计算完成后，**按拓扑排序的逆序，依次计算每个事件的最晚发生时间**

**vl(j) = Mink {vl(k)-dut(<j, k>) }**

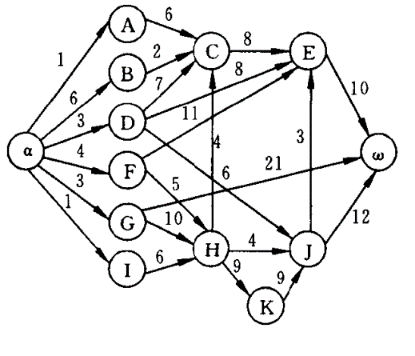
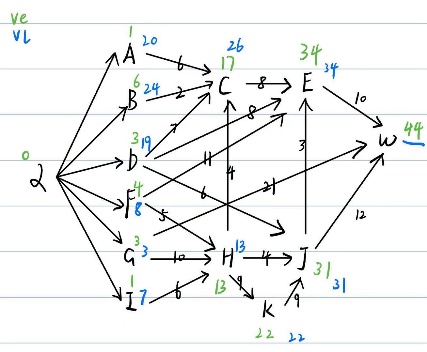
4. 计算**e(<j, k>) = ve(j)** 和 **l(<j, k>) = vl(k)-dut(<j,k>)**

5. 所有e=l的弧为关键活动，串起的路径为关键路径。



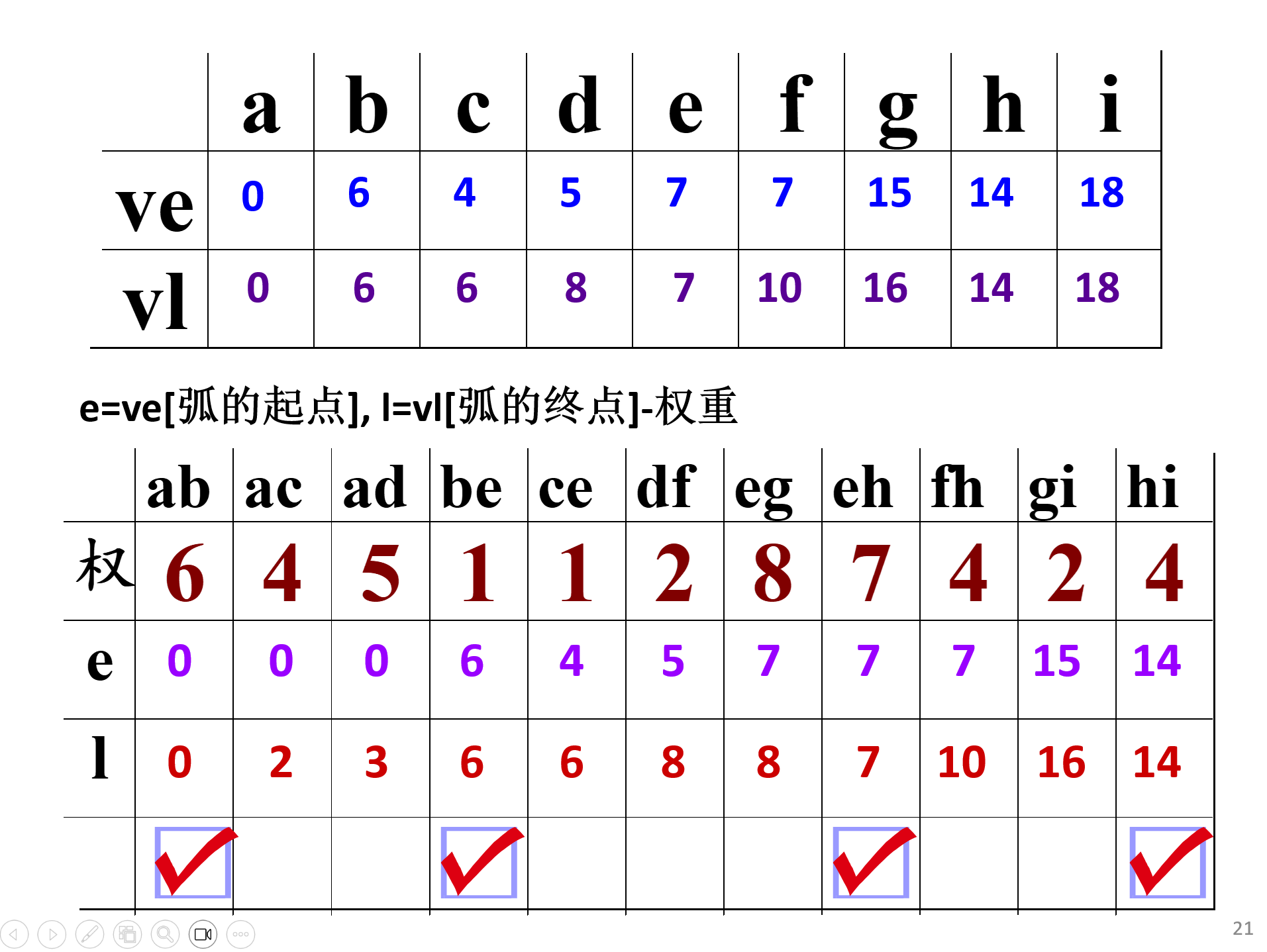
对顶点进行拓扑排序得到顶点序列：

α>A>B>D>F>G>I>H>C>K>J>E>ω



按拓扑顺序，依次计算每个事件的最早发生时间，再按拓扑排序的逆序，依次计算每个事件的最晚发生时间

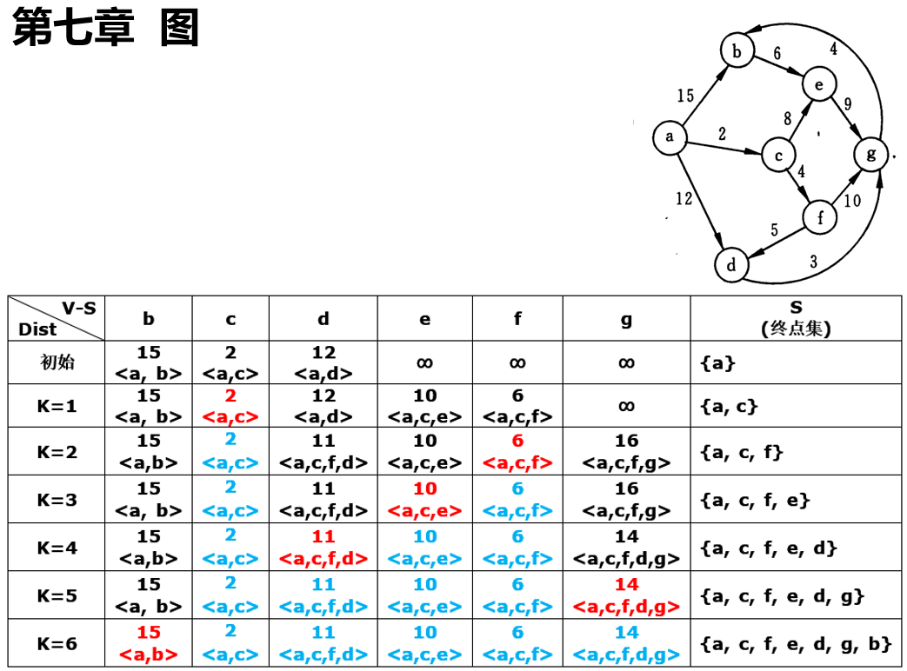
|  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **顶点v** | **α** | **A** | **B** | **D** | **F** | **G** | **I** | **H** | **C** | **K** | **J** | **E** | **ω** |
| **ve(v)** | **0** | **1** | **6** | **3** | **4** | **3** | **1** | **13** | **17** | **22** | **31** | **34** | **44** |
| **vl(v)** | **0** | **20** | **24** | **19** | **8** | **3** | **7** | **13** | **26** | **22** | **31** | **34** | **44** |

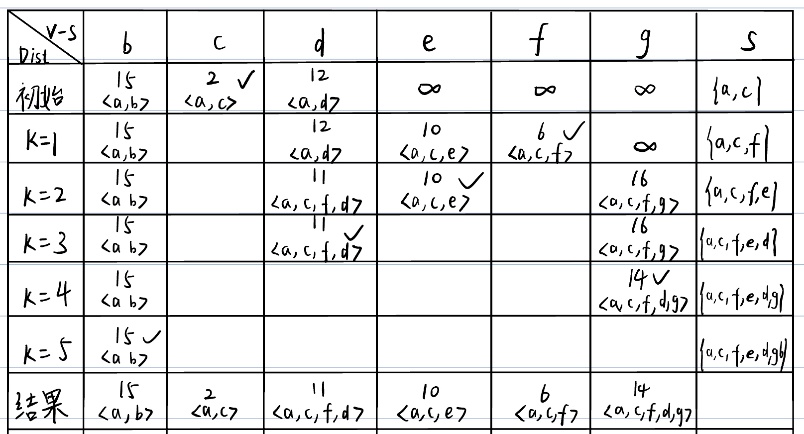


**最短路径（适用于有向图和无向图）**

**Dijkstra算法（权值非负 单源点，可用于带回路的情况）**

按路径长度递增的次序生成从源点到各顶点的最短路径



每次选择**从已求出最短路径的顶点到待求解顶点的**弧中权值最小的弧，将其弧头顶点标记为已求出。

初始时，除了起始点邻接弧外，其他弧均设为∞

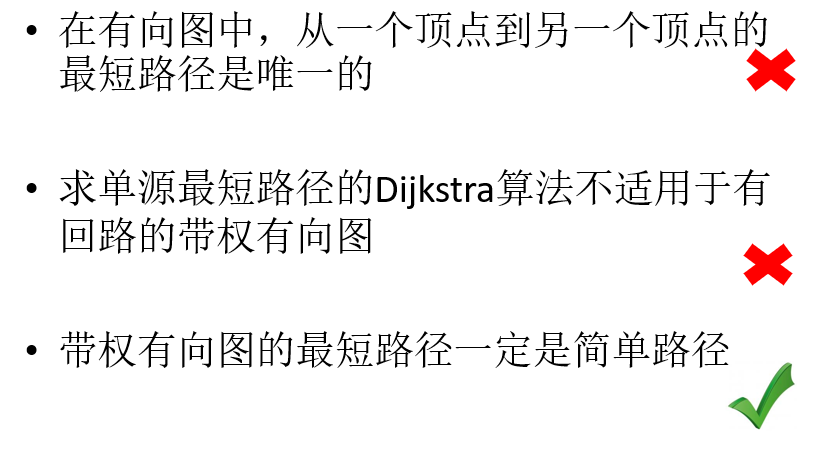
**Bellman-Ford算法（权值任意 单源点）**

**Floyd算法（权值为任意值 全源点，但不允许有负权值回路）**

初始时**D[i][j]对角线全设为0，有边的位置设为权值**，其他地方设为∞。设终点集S={}。

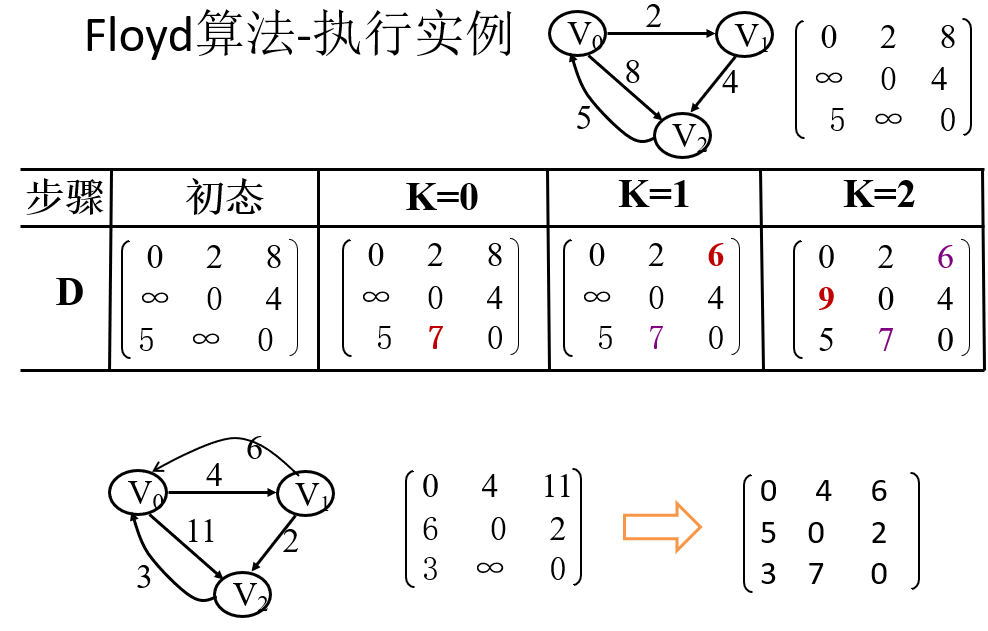
每次向S中加入一个顶点Vk，根据修改D[i][j]的值。

**使用二维数组P[i][j]记录所有路径，当某个顶点 𝑉\_𝑘 加入到S中后使D[i][j]变小时，令Path[i][j]=k （需要输出路径时递归搜索Path）**



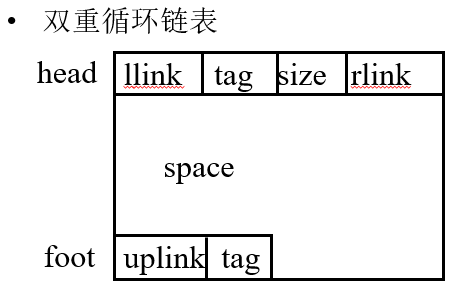
最短路径不唯一，最短路径长度唯一。

**只要求权值非负，不要求无回路。**



**第8章 动态存储管理**

**边界标识法**

数据结构Space：（WORD\*）

头部域：llink指向前驱节点（WORD\*）

rlink指后继节点

tag 空闲0或占用1

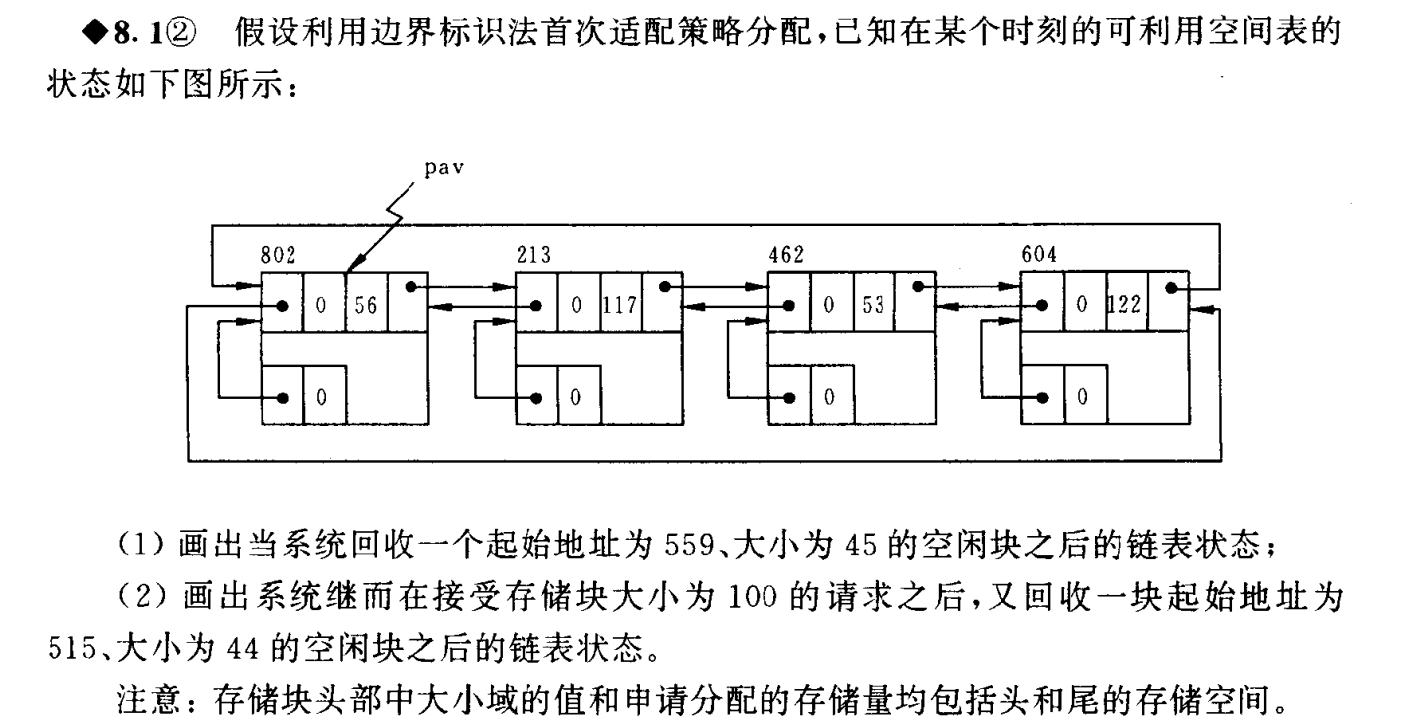
size 空间大小

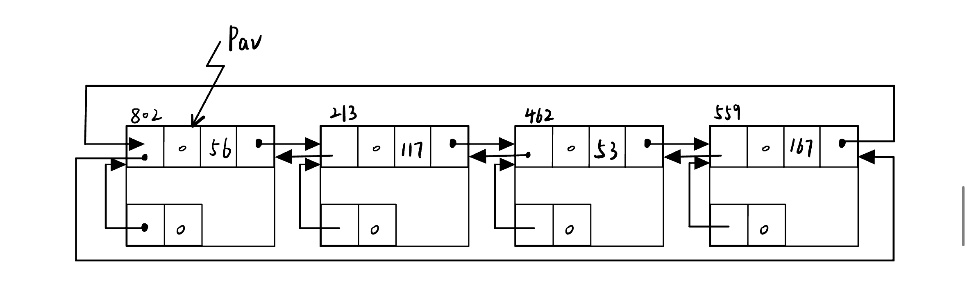
尾部域：uplink指向本结点的头部

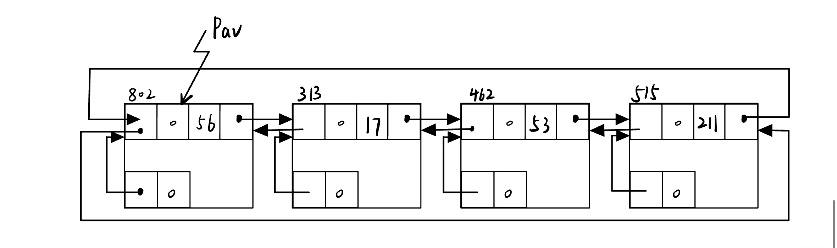
* **分配约定**：
  + 选定适当**常量e**，设待分配空闲块、请求分配空间的大小分别为m、n
  + **当m-n≤e时：将整个空闲块分配给用户**；
  + **当m-n>e时：则只分配请求的大小n给用户**；

尽量减少空闲块链表中出现小碎片(容量≤e) ，提高分配效率；减少维护工作量

* + 为了避免修改指针，约定**将高地址部分分配给用户**
* **查找约定：**
  + 每次需要查找空闲块时，从**上次刚分配结点的后继结点**开始查找空闲块（pav指针）
  + 作用：提高查找空闲块的速度，防止小容量结点聚集
* 分配算法：首次拟合法（且不保留小于等于e的剩余量）
* 回收算法：释放占用块时，检查左右紧邻块是否空闲，若空闲考虑合并

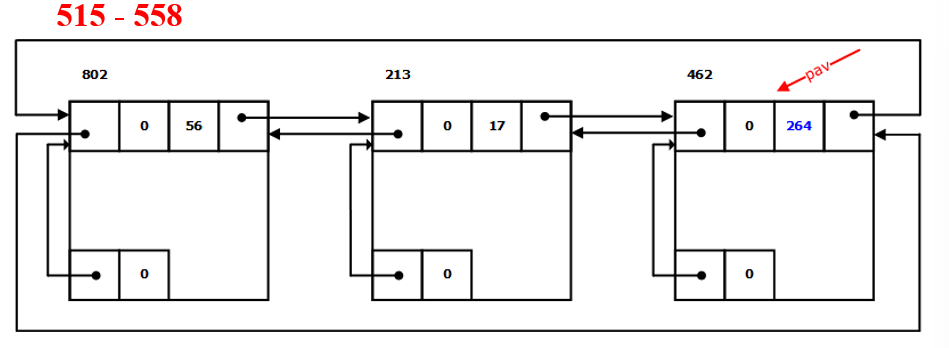
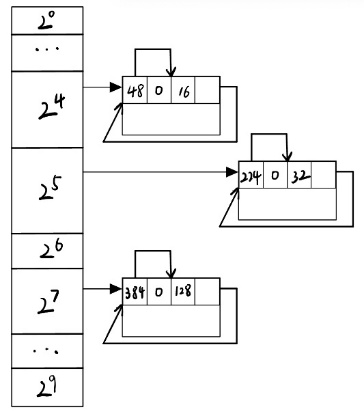


（1）



pav

（2）

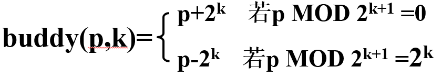


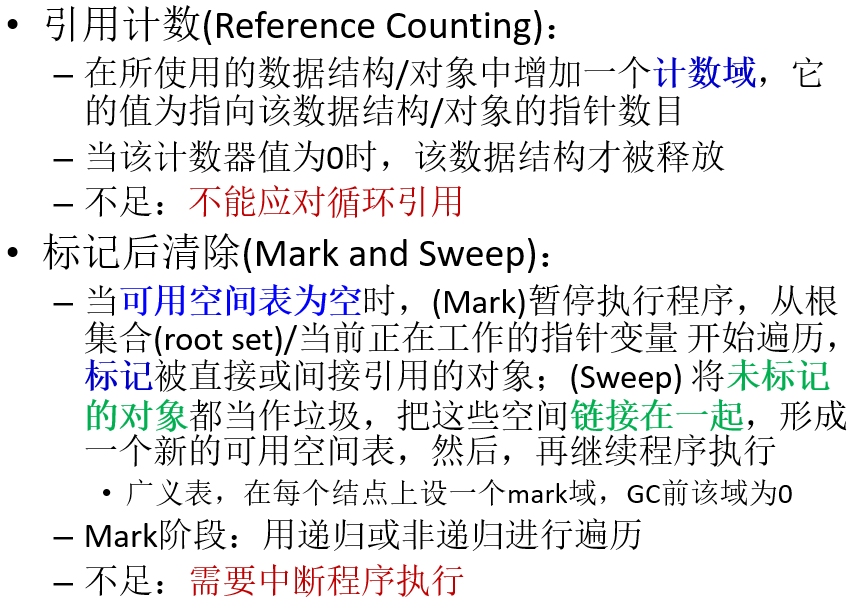
伙伴系统的分配算法：

* + 若2k-1<n≤2k-1：2k子表有空闲子表结点，则将子表中的任意一个结点分配之
  + 2k子表为空，则从结点大小为2k+1的子表中找到一个空闲结点，将其中一半分配给程序，剩余的一半插入到结点大小为2k的子表中

（若2k到2k+i都为空，则从2k+i+1取结点时，要依次给这些子表分配节点）

回收算法：只有互为伙伴的两个相邻空闲块可合并（即**从同一个大块分裂而来**）

伙伴的首地址：当前块是大块的前半还是后半



**第9章 查找/搜索**

查找方法的评价指标：（1）查找算法的时间复杂度，所需的附加存储空间（2）关键字的平均比较次数

**一、静态查找表**

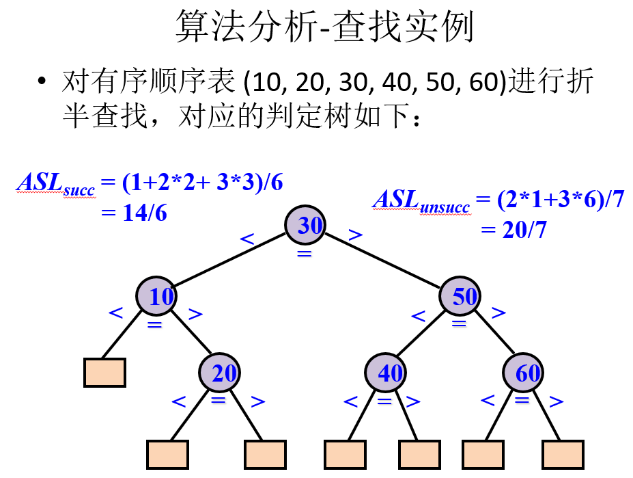
**（1）顺序查找：**依次将记录的关键字与给定Key值比较。**可在0号位置设置哨兵，从后往前查找失败时返回0**，省去了控制循环结束的条件判断。

成功时ASL=(n+1)/2 失败时ASL=n+1 **ASL=3(n+1)/4**

改进：①按数据元素**被查找的概率进行升序排序**，此时ASL取极小值 ②元素中增加**访问次数**数据项，每次查找后**维护访问次数升序**排序。

**（2）折半查找**：基于**有序**顺序表的查找

最多查找次数为⌊log2n⌋+1次



缺点：无法应用于链表、不等概率查找时不一定最优、长度不大时效率不如顺序查找

**（3）Fibonacci查找**：基于**有序**顺序表的查找

平均性能比折半查找好，但最坏情况下比折半查找差

**（4）索引顺序查找/分块查找**：将查找表分成几块，块间有序，块内无序。在查找表的基础上附加一个索引表，索引表是按关键字有序的，索引表中记录的构成是：(块内的)最大关键字、(块的)起始指针

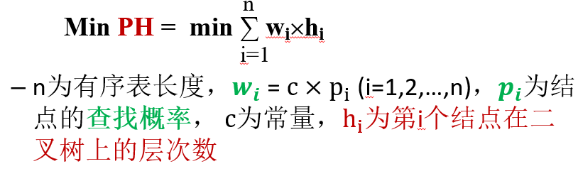
ASL = Lb + Lw 确定所在块的平均查找长度+在块内查找的平均查找长度



**静态树表/次优查找树的查找**

各个元素的查找概率不等

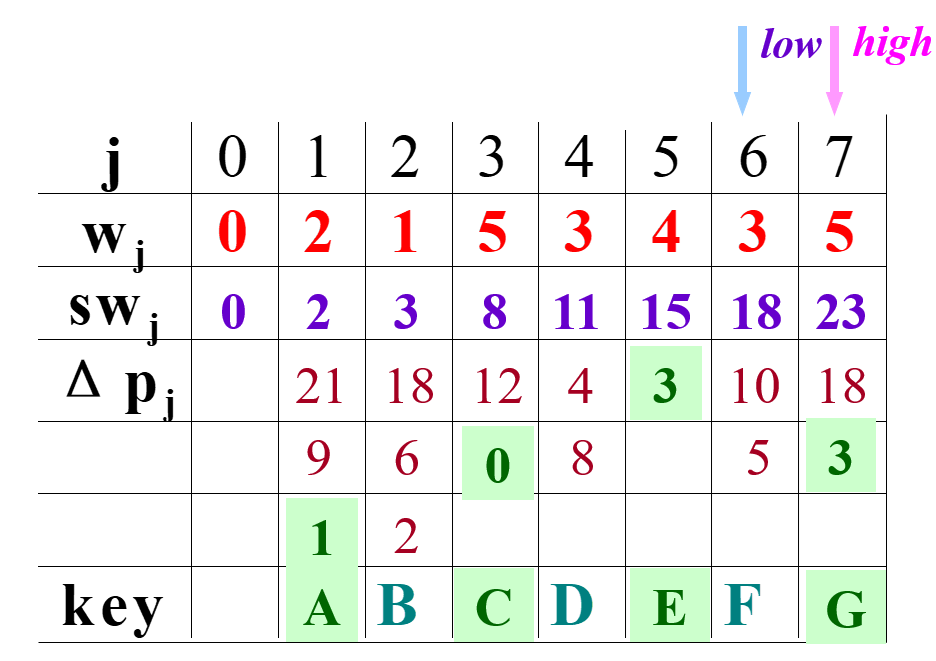
静态最优查找树(Static Optimal Tree)：该判定树的带权内路径长度之和最小

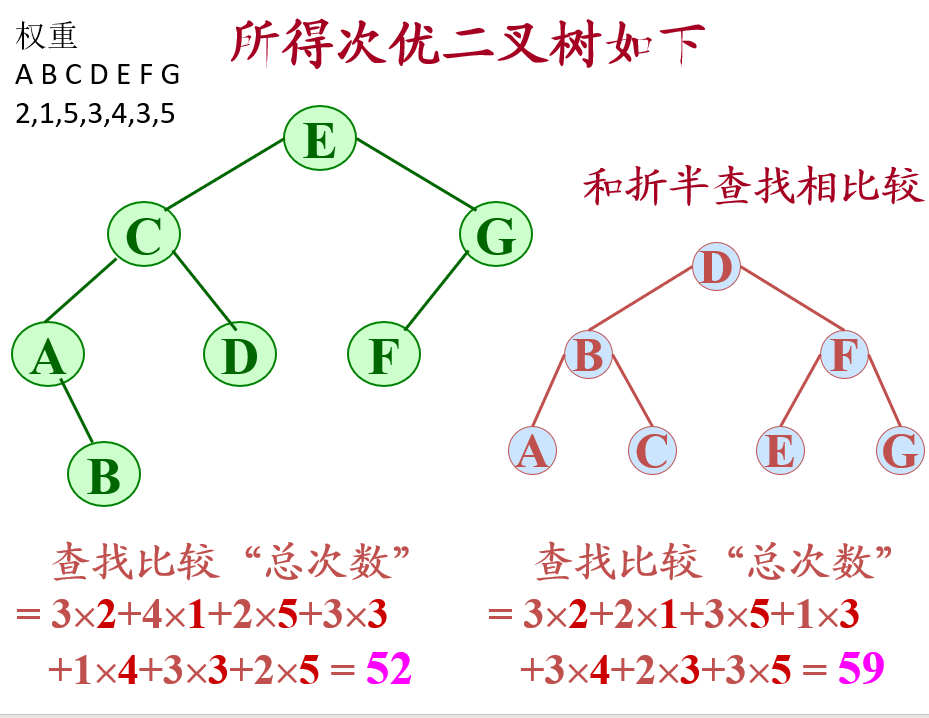
概率长度

但构造静态最优查找树的时间开销太大

构造次优查找树(Nearly Optimal Search Tree)：

对于升序序列(rl,rl+1,…,rh)，设对应权值为wi，选择一个根结点rj，使得其前后序列中结点的**权值和相差ΔP最小**，并分别为前半序列和后半序列构造次优查找树，作为rj的左右子树。





**二、动态查找表**

树的形式组织查找表，可以对查找表进行动态、高效的查找

**二叉排序树**

空树 或 左子树**所有结点**的值<根结点<右子树**所有结点**的值且左右子树都是二叉排序树

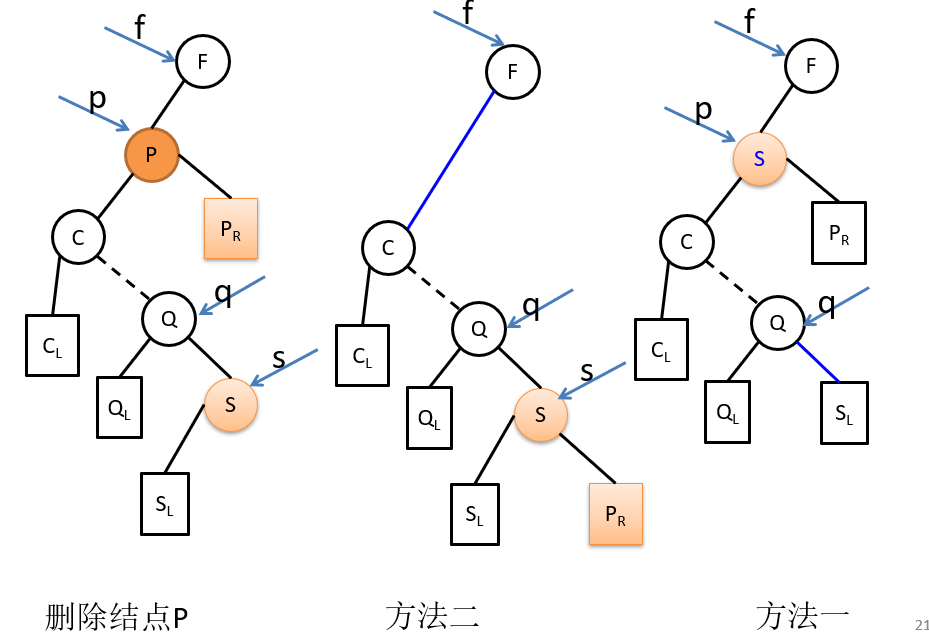
**注意：若从根结点到某个叶结点有一条路径，路径左边的结点的关键字不一定小于路径上的结点的关键字**

中序遍历可得有序序列。

构造：新插入的结点一定是BST的一个新的叶子结点，并且是查找不成功时查找路径上访问的最后一个结点的左孩子或右孩子

结点的删除：叶子节点直接删；只有一棵子树，子树根替代；有左右子树时，方法一：找到p的左子树中的最右边的结点s(直接前驱结点，没有右子树)，删除s，代替p，并将s 的左子树作为s父节点的右子树。方法二：将p的右子树作为s的右子树，然后用p的左子树顶替被删结点。

查找效率：依赖于树的形态



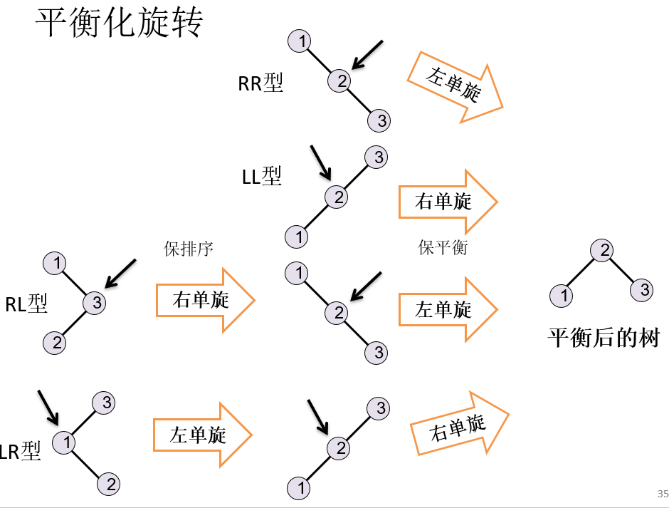
**AVL树（看ppt-part2熟悉流程！！）**

左子树和右子树深度之差的绝对值不大于1，且左右子树均为平衡二叉树

结点的平衡因子：左子树深度-右子树深度

平衡二叉排序树：既是二叉排序树，优势平衡二叉树

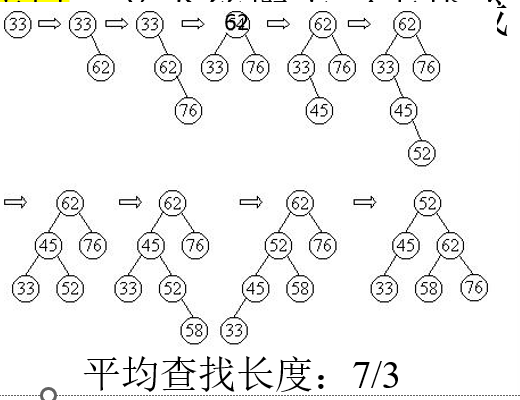
插入：在插入一个新结点后，需要从插入位置**沿通向根的路径回溯**，检查各结点的平衡因子。如果在某一结点发现不平衡，停止回溯。从发生**不平衡的结点**起，沿刚才**回溯的路径（向下）**取直接**下两层的结点**，进行平衡化旋转

三点直线：单旋转

三点折线：双旋转

右单旋：父节点与左子节点交换拉链，并成为左子节点的右节点

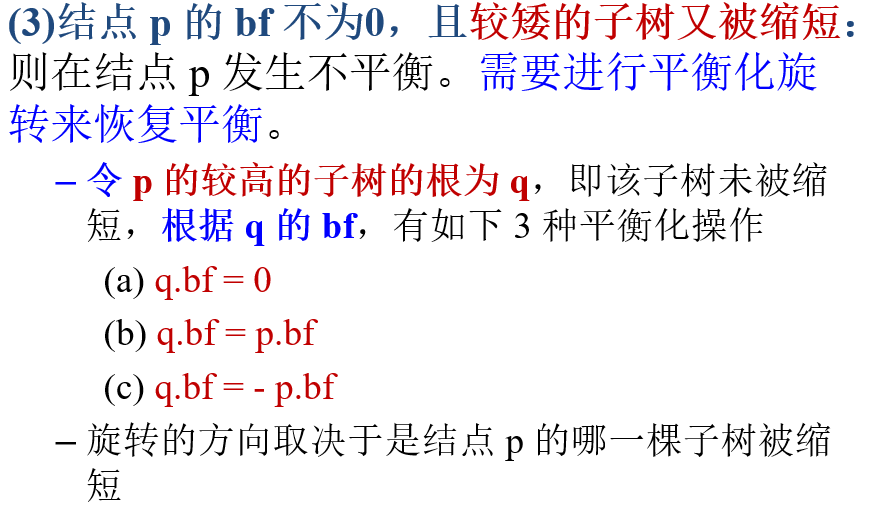
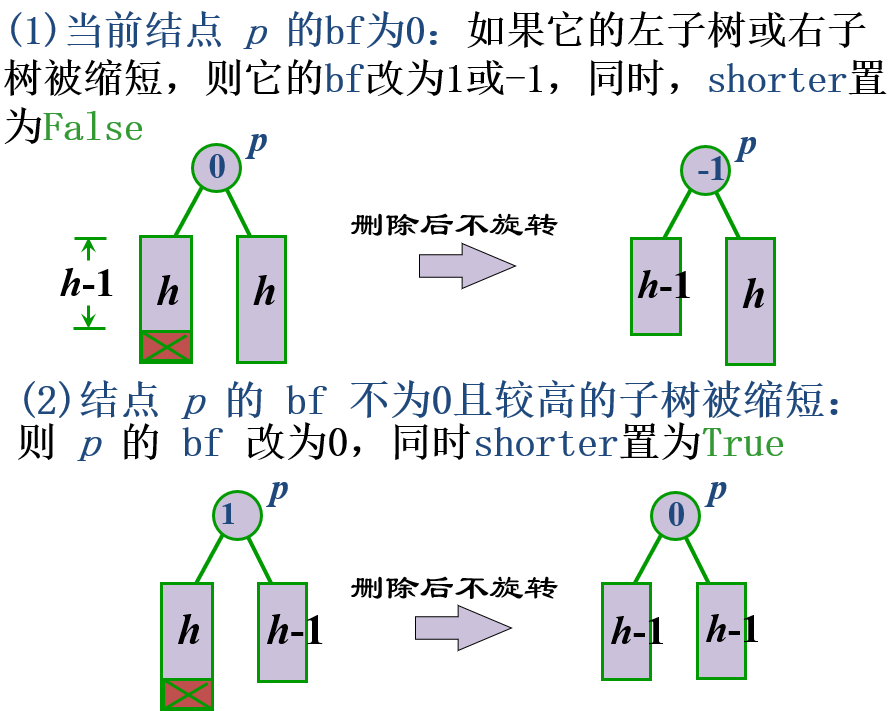
右单旋：父节点与右子节点交换拉链，并成为右子节点的左节点

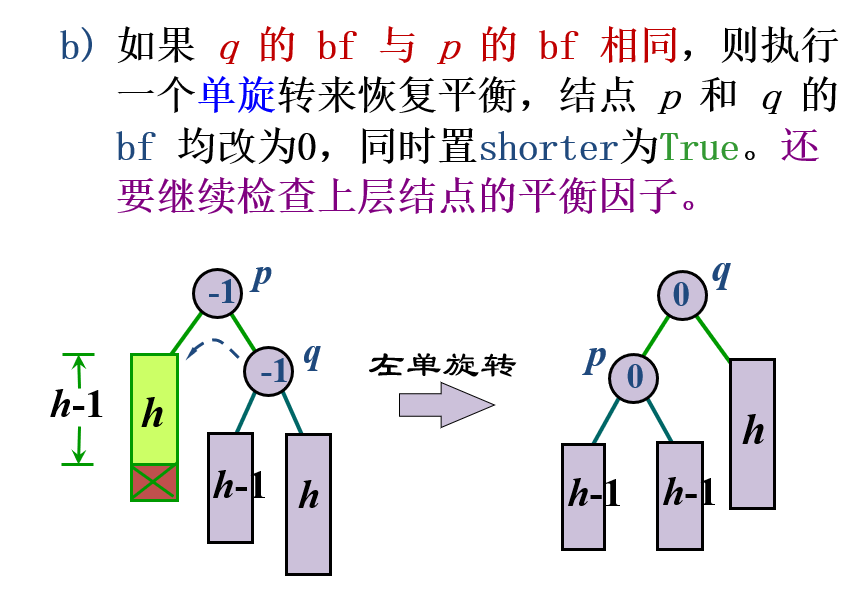
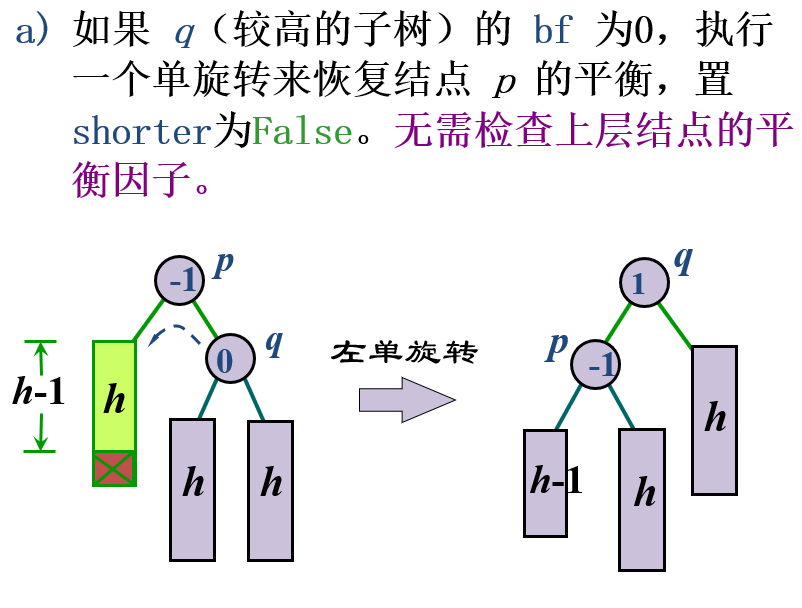


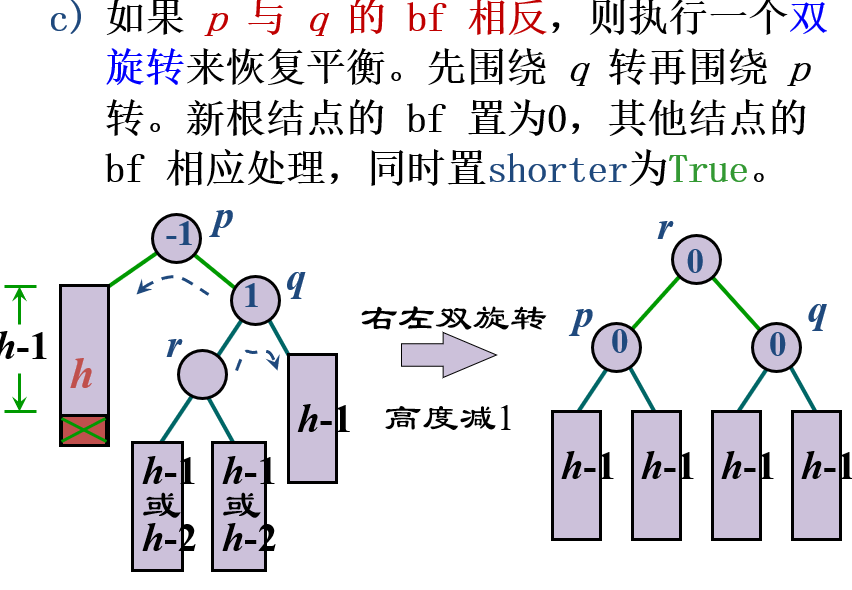
从9一路向上回溯检查，检查到的第一个不平衡结点为4，4与路径上的结点构成直线型，将4及其子树旋转下来，6的左孩子5插入到4所在的子树中

删除：叶子节点/单子女，直接删；两个子女，找到x的直接前驱y，将y的内容传给x后，转为对y结点的删除。**完成删除后，反向回溯检查**。

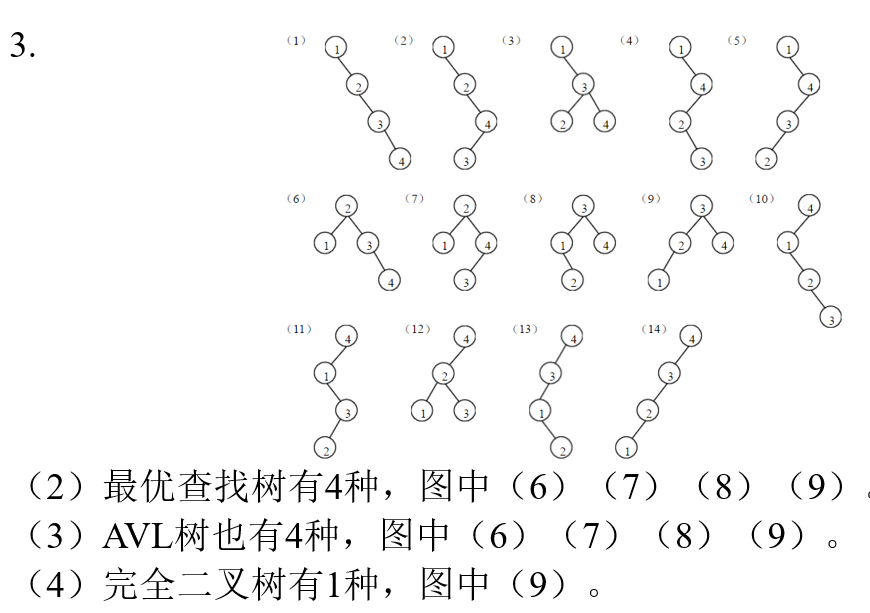
p为当前结点，q为较高子树

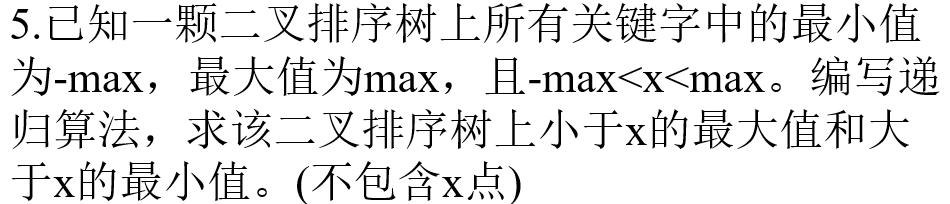


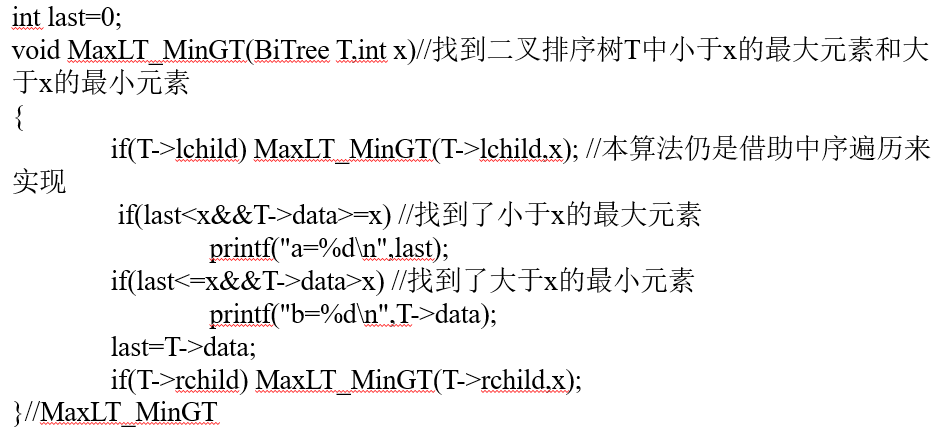




ASL = O(log2𝑛)







**B树（一种平衡的多路查找树）**

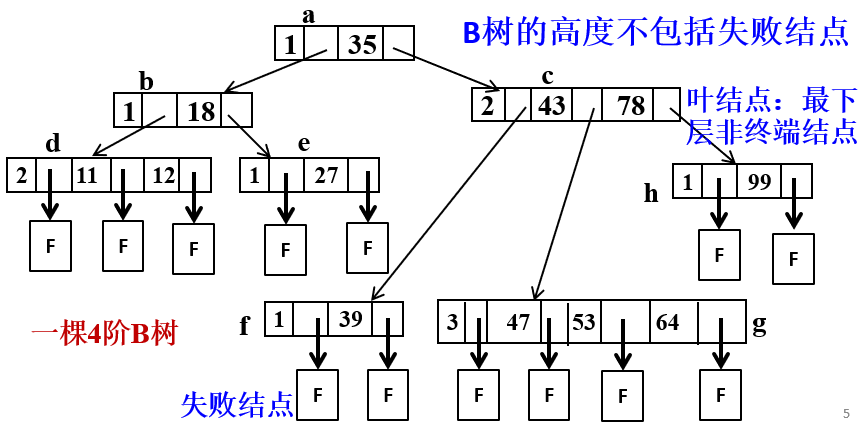
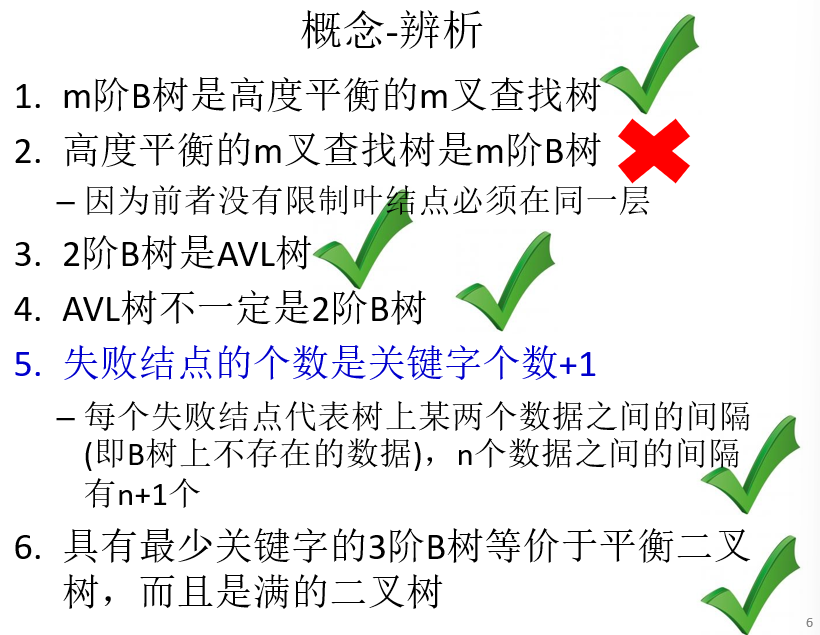
降低磁盘I/O次数：树结点的大小尽可能地接近磁盘页的大小

**m阶B树**：每个结点**至多**有**m棵子树**，根结点要么无子树要么至少有2棵子树，其他所有非终端结点**至少有⌈𝒎∕𝟐⌉棵子树**（关键字个数≥⌈𝒎∕𝟐⌉-1）

非终端结点：(n，A0，K1，A1，K2，A2，… ，Kn，An) n为关键字个数，Ki为关键字，Ai为指向孩子结点的指针，且Ai-1所指向的子树中所有结点的关键字都小于Ki

失败结点：都在树的同一层上，且不带信息。指向它们的指针为空

叶结点：最下层的非终端结点，处于同一层



查找：从树的根结点T开始，在T所指向的结点的关键字向量key[1…keynum]中查找给定值K，若成功，返回结点及关键字位置；否则转到对应大小范围的的子树进行查找，若找到的子树是失败结点（空指针）则查找失败。

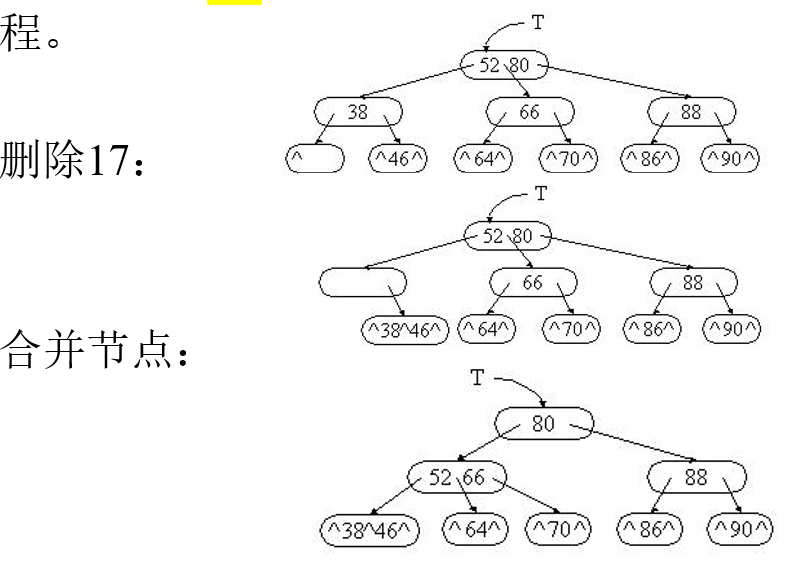
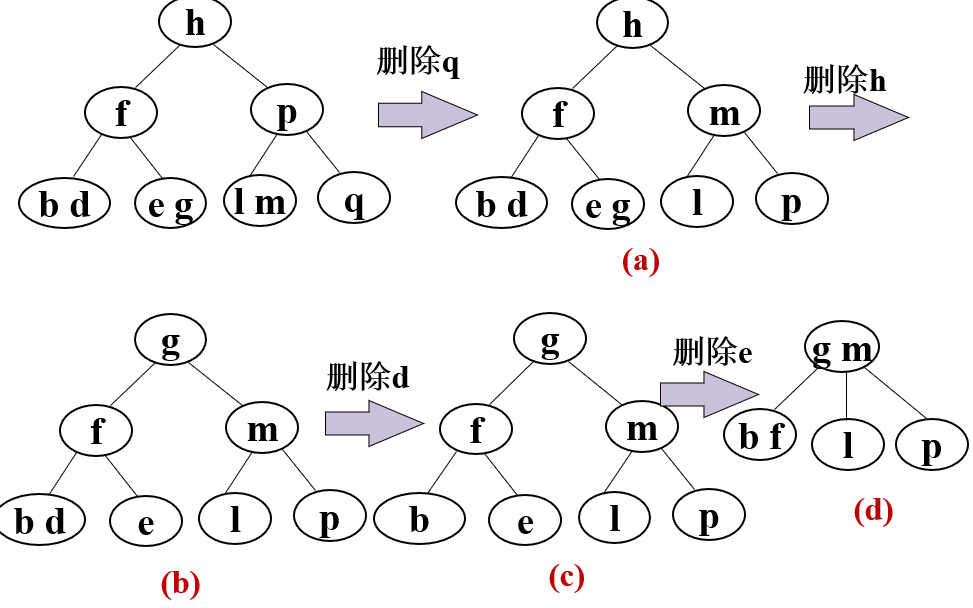
删除：删除结点N中的一个关键字K。

K在非叶子节点中：设K是N中的第i个关键字，则将指针所指**子树**中的**最大关键字K’**放在(K)的位置，然后删除叶子节点中的**K’**

K在叶子节点中：

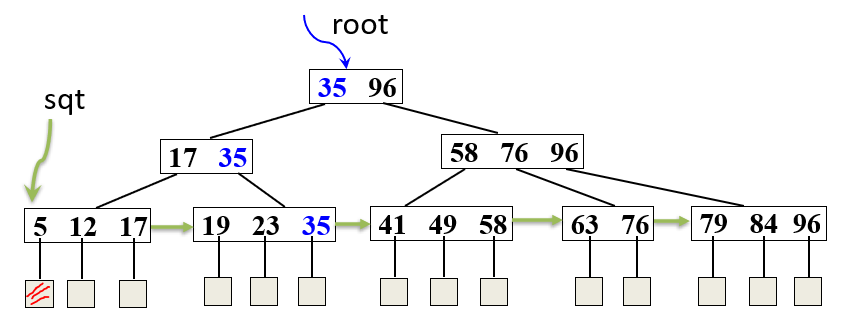
1. 若N中关键字个数＞⌈𝒎∕𝟐⌉-1，直接删；

2. 若=⌈𝒎∕𝟐⌉-1，（1）如果左/右兄弟结点关键字个数足够（＞⌈𝒎∕𝟐⌉-1）则将其中最大/小的关键字上移到父结点中，父结点中大于/小于且紧上移关键字的关键字下移到结点N；（2）如果兄弟结点中的关键字数=⌈𝒎∕𝟐⌉-1，将N与兄弟节点q+分割两者的父节点的某个关键字Ki 合并为一个结点（不包括被删关键字），注意父节点不动即使它变空。此后，若父节点的关键字不足时：如果是根结点，则直接删除；如果非根，且只有⌈𝒎∕𝟐⌉-2个关键字，则对父节点、其兄弟、父父节点做类似操作。

**B+树**

1. 每个结点至多有m棵子树，根结点要么无子树要么至少有2棵子树，其他所有非终端结点至少有⌈𝒎∕𝟐⌉棵子树
2. 若一个结点有n棵子树，则必含有n个关键字
3. **所有叶子结点中包含了全部记录的关键字信息以及这些关键字记录的指针**，而且叶子结点按关键字的大小从小到大顺序链接，构成一个**有序链表**
4. 所有的非叶子结点可以看成是索引，**结点中只含有其子树的根结点中的最大(或最小)关键字（及指向对应子树的指针）**



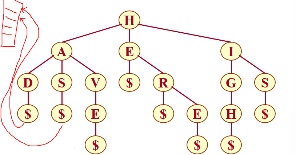
与B树相比的查找优势：树不仅可以从根结点开始按关键字随机查找，而且可以从最小关键字起，按叶子结点的链接顺序进行顺序查找。

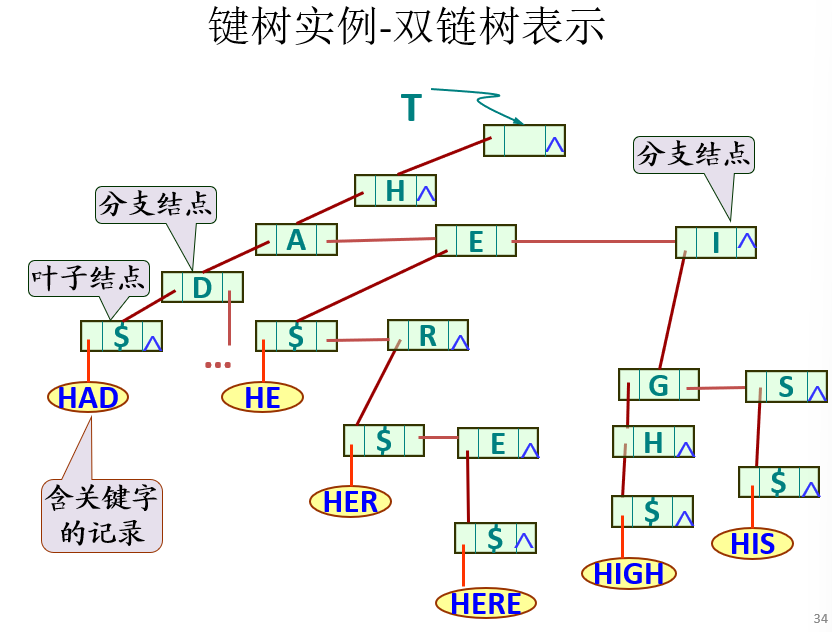
**在B+树上进行随机查找时，若非叶子结点的关键字等于给定的K值，并不终止，而是继续向下直到叶子结点(只有叶子结点才存储记录) ，即无论查找成功与否，都走了一条从根结点到叶子结点的路径**

插入：仅在叶子节点上执行，当叶子节点关键字数>m时，分裂为⌊(m+1)/2⌋ 和⌈(m+1)/2⌉两个节点，且**更新父节点对应位置的关键字（即两个新节点各自的最大关键字）**，当然这可以引起父节点继续分裂；当叶子节点关键字数=⌈m/2⌉时，要与兄弟结点合并。

**键树**

关键字中的各个符号分布在从根结点到叶的路径上，叶结点内的符号‘$’为结束的标志符。

是度数大于2的树。有序树：**同一层中的兄弟结点**之间所含符号从左到右有序，约定‘$’小于任何符号。 实际形态：



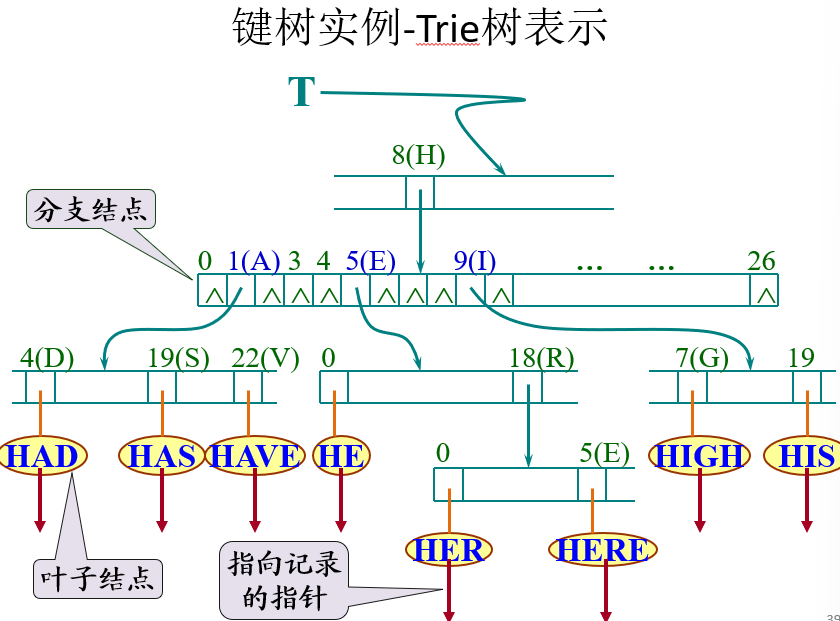
逻辑存储数据结构：双链树

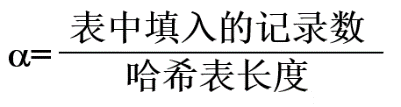
标识符、指向兄弟的指针、指向孩子/记录的指针、节点类型

ASL=h/2 \* (1+d)

d为最大的度，h为树的深度

逻辑存储数据结构：Trie树

节点类型、指向记录的指针/指向下一层结点的指针数组

**哈希表！！！**

哈希表的装填因子α：

(1)开放定址法：使用探测序列di生成新的哈希地址，直到不冲突为止。

H0(key) = H(key)

Hi(key) = (H(key)+di) MOD m，i=1, 2, …, m-1

序列选择：①线性探测法，di=1, 2, 3, …, m-1 ②二次探测法 （提前计算好的）di=1², -1², 2², -2², 3², ……+k², - k² (k≤⌊m/2⌋) ③伪随机探测法：伪随机序列

二次探测法的性质：当表长m是质数，且装填因子小于等于0.5，可以找出空闲地址；表长 m 是形如 **4j+3的质数** (如7, 11, 19, 23, 31, 43, … )时，可以保证**查找链的前m项均互异**

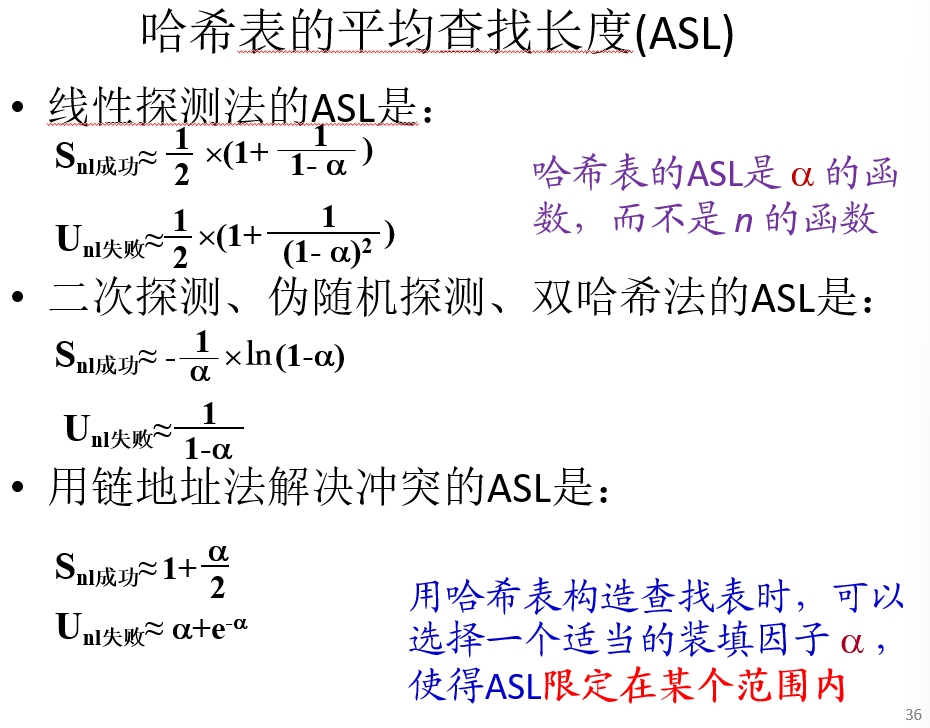
(2)多哈希法/双哈希法：Hi=RHi(key) i=1, 2, …, k。 RHi 为一组不同的哈希函数

(3)链地址法：同义词直接存储再一个单链表中，并用一维数组存放m个链表的头指针

(4)建立公共溢出区：发生冲突的数据填入溢出表当前空位中（用于冲突较少的情况）

查找：通过检测当前位置是否为空/是否等于关键字，决定是否要按冲突处理方法生成下一个哈希地址，直到找到关键字或超出查找次数。

**各种冲突处理方法下哈希表的ASL：**



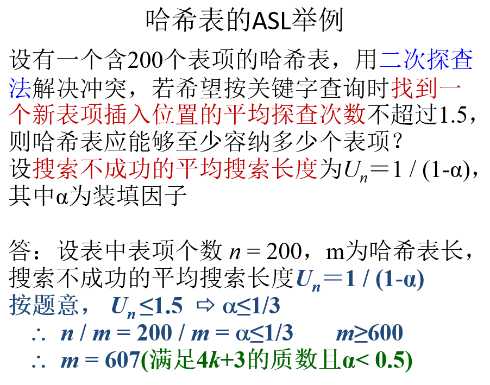
α：装填因子

线性探测：

ASL成功 = (1+1/(1-α))

ASL失败 = (1+1/(1-α)^2)

**插入操作的平均探查次数=ASL失败**



二次探查法：**表长 m 是形如 4j+3的质数 (如7, 11, 19,…)时，可以保证查找链的前m项均互异**

设计哈希表的方法：

1. 由预期的ASL，根据所选的处理冲突方法，求出装载因子α的上界

2. 由α值和记录数设计哈希表的长度m

3. 根据关键字的特性和表长m选定合适的哈希函数

**第10章 内部排序**

评估指标：1. 时间开销 2. 空间开销（就地排序：空间复杂度O(1) 、非就地排序：空间复杂度与n有关）3.稳定性 4. 输入敏感

排序方式：插入、交换、选择、归并、基数



**不稳定的排序方法：Shell、简单选择（最小值更新条件为≤）、快速、堆**

**（1）插入排序**：每次将一个待排序的元素插入到已经排序好的序列上

以下算法时间开销均为O(𝑛^2)

直接插入排序：数组前半为已排序好的部分，后半部分为未排序好的部分。初始时r[1]是排好序的，每趟排序时将r[0]设置为本次待排序的元素。

空间开销：辅助空间仅r[0] 时间开销：O(𝑛^2) 稳定的 输入敏感的

折半插入排序：插入已排序好的部分时，采用二分查找

稳定的 输入敏感的

2-路插入排序：新开设一个数组d当成循环向量，将r[1]赋给d[1]作为排好序数组的中间元素，设两个指针first和final分别指示排序过程中得到的有序序列中的第一个和最后一个记录。之后按照与d[1]的大小关系将r[i]插入到d[1]之前或之后的有序数组中

当L->r[1]是待排序记录中关键字最大或最小的记录时，2-路插入排序就完全失去了优越性

稳定的 输入敏感的

表插入排序：排序过程中不移动记录，只改变记录的链接顺序。为了方便查找，需要对结果进行重新排列求得一个有序数组

O(n)的辅助空间 稳定的

希尔排序 ：按增量序列d划分子序列，对每个子序列进行直接插入排序



**希尔排序是唯一一种不稳定的插入排序方法**

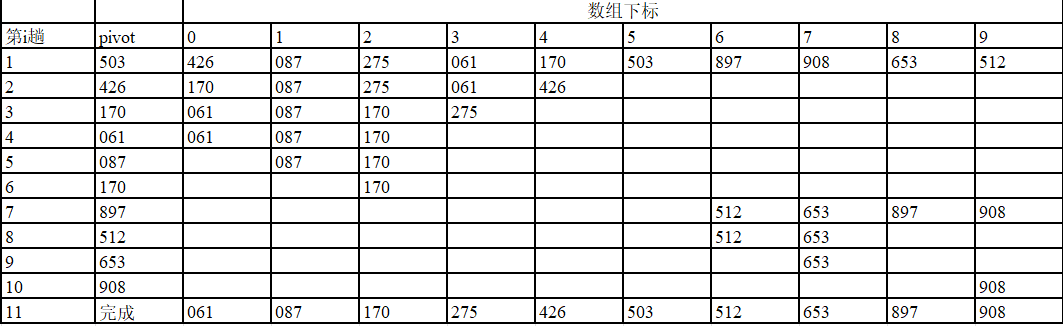
输入敏感的，实际运行的时间更多地取决于所取“增量/gap“

**（2）交换排序𝑂(𝑛^2)**

起泡排序：依次比较每一对相邻元素，如有必要，交换之。每扫描一趟后问题规模减一。若整趟扫描都没有进行交换，则排序完成

稳定的 最好O(n)

快速排序：任取序列S中的记录m作为轴点(pivot)，将序列分为两个子序列



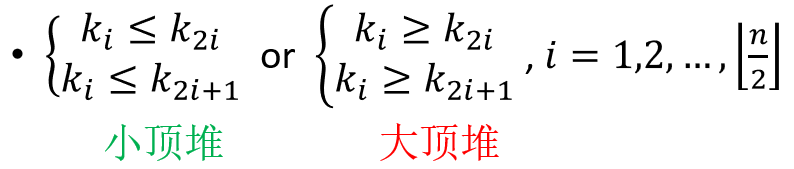
最好𝑂(𝑛 log\_2 𝑛) 空间开销：递归栈 不稳定

**（3）选择排序 ：**每一趟中选出关键字最小的元素，作为有序元素序列的第 i 个元素

简单选择排序：每次从后面 n-i +1个到最后逐一进行比较，将最小元素与第i个元素交换，没有利用上次比较的结果 **不稳定**

树形选择排序/锦标赛排序：构建胜者树（完全二叉树（结点按编号连续）），待排序元素作为叶子节点，每次两两比较的结果是把排序码小者作为优胜者上升到双亲结点。每次输出胜者后，将胜者所在位置改为\*（MAX），然后重新向上调整。

堆排序：父节点大于/小于左右子节点的**完全二叉树** **不稳定**



建（小顶）堆：自底向上调整堆，先找到最后一个非叶子节点**i=(n-2)/2**，依次从i到0号结点（i+1次循环）递归调用调整：每次调整将左右子节点中较小的与父节点比较，若小则上升，然后递归调整左右子树。（**调整是向下递归的调整**）

输出堆顶后，将堆顶与堆末的元素（非空位置）交换，然后将堆末置空（小顶堆置MAX），从堆顶递归调整。

**（4）归并排序：**将两个或两个以上的有序表合并成一个新的有序表

2-路归并：



**（5）基数排序**：

链式基数排序：按**个位-十位-百位**顺序（**从低位到高位顺序**） 交替进行分配和收集

radix基数：队列数 d：排序码位数 总时间复杂度为O(d(n+radix))

若基数radix相同, 对于**元素个数较多而排序码位数d较少**的情况, 使用链式基数排序较好

基于“比较关键字”排序方法的时间复杂度下限：基数排序不在此列

在最坏情况下达到的最好时间复杂度为 O(nlogn)

**第11章 外部排序**

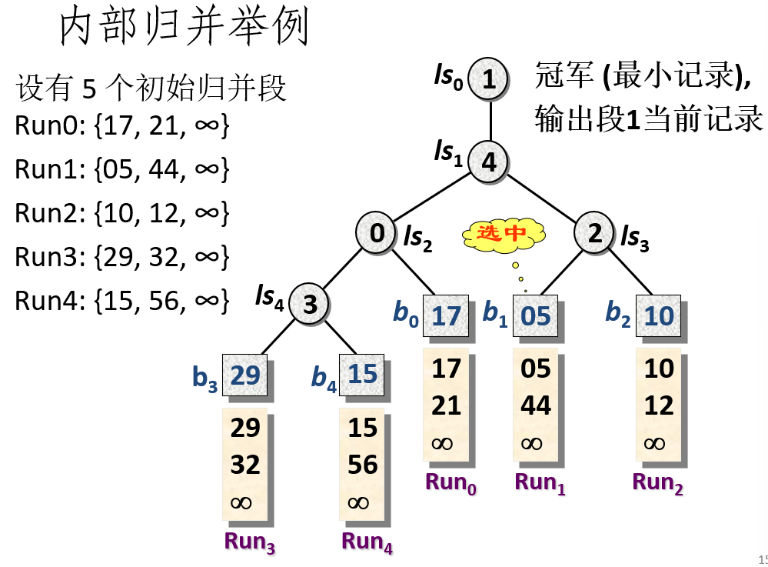
多使用归并排序

第一步：在内存缓冲区中将输入文件划分为若干初始归并段，生成后就写入到外存中

第二步：一趟趟对初始归并段加以归并

**k-路平衡归并（对初始归并段归并时）**

使用“败者树(tree of losers)”从k个归并段中选最小者

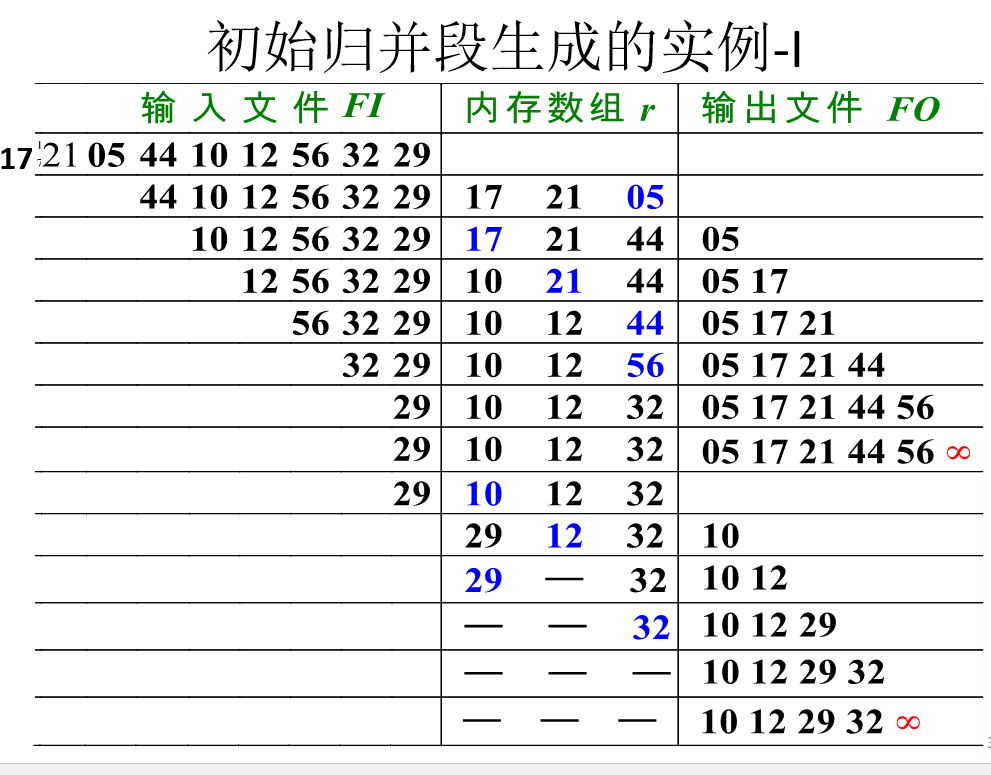
ls0记录每次比较的胜者（实时更新），最终是冠军。其他节点记录每次比较的败者的段序号。

**每次输出冠军后，对应的段下一元素参赛，从对应段结点开始从下向上调整，每次与父节点记录的败者比较，更新败者树。**

败者树重构时只需考虑父节点，无需关注兄弟结点。

**置换-选择排序**

**生成初始归并段：**

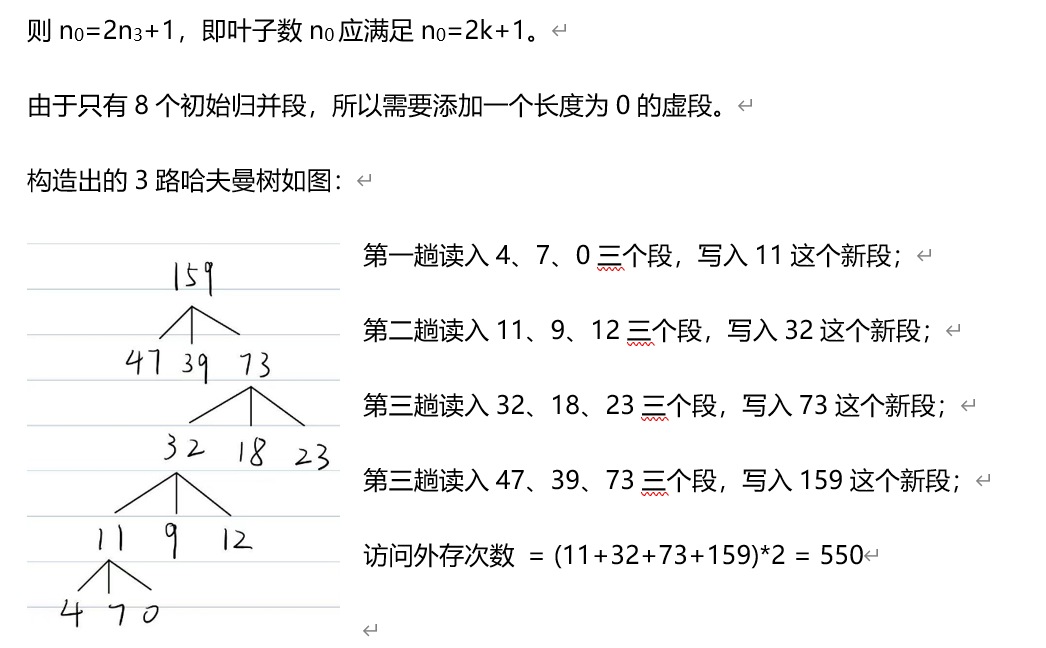


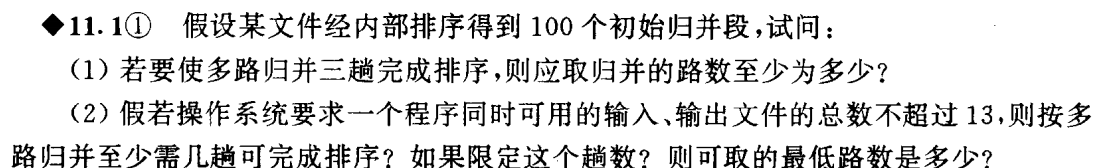
**（chap11 ppt P35）**

**最佳归并树（归并排序时归并初始段的顺序）**

以段大小为权值构造k路Huffman树（是满k叉树）

注意结点关系n0=(k-1)nk+1，必要时需要补充虚段（大小为0）





（1）由logk100 ≤ 3得k ≥ e^(1n(100) / 3)≈4.6

则归并路数至少为5。

（2）一次k路归并操作需要k个输入文件和1个输出文件，要求k+1≤13，则k≤12。设趟数为p，要求kp≥100。（k路p次I/O）

若取p=1，显然k不满足约束。若取p=2，则10≤k≤12

因此，至少需要2趟完成排序。如果限定2趟，可取的最低路数为10路。