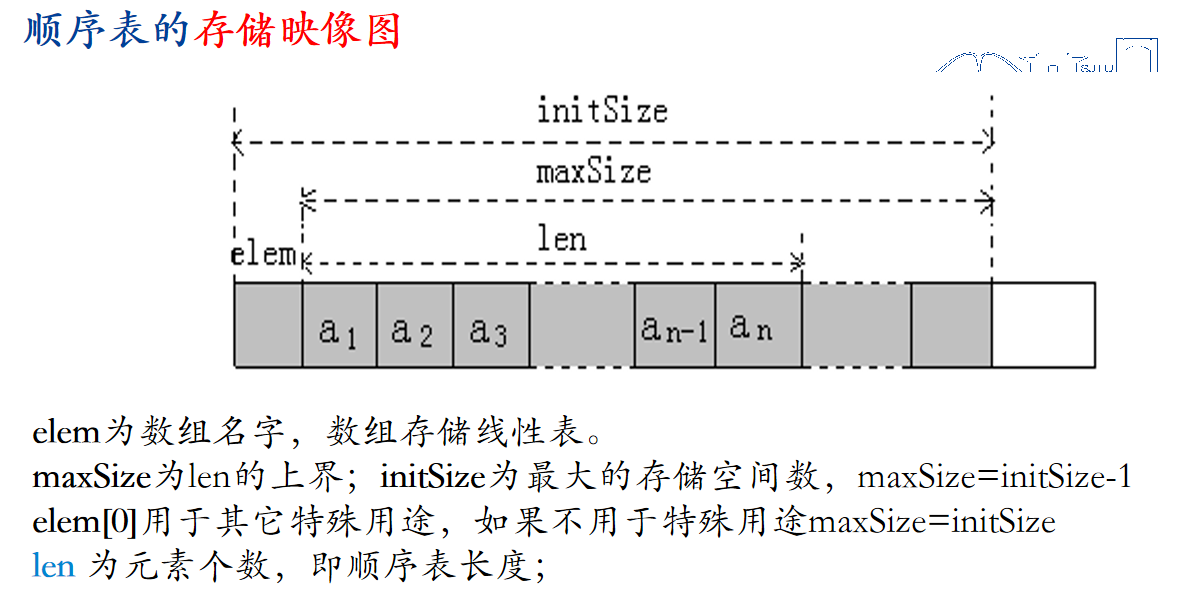
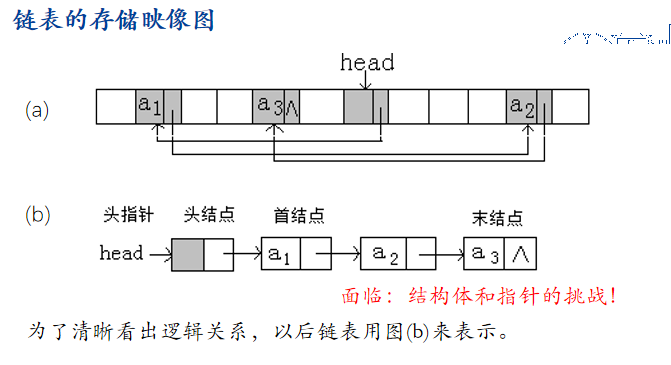
**数据结构复习笔记**

**线性表**

1. **顺序表**

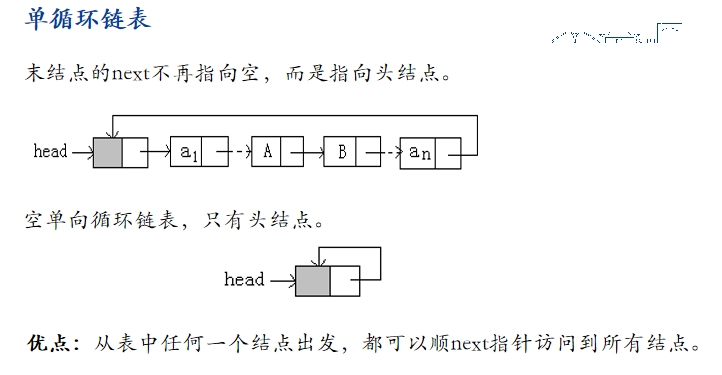
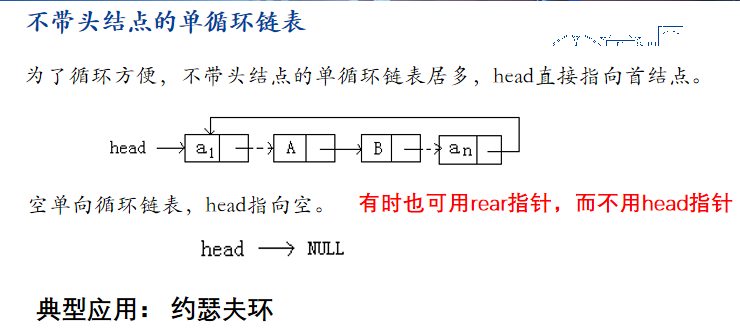
**（1）存储结构：**

1. **顺序表的基本操作&代码**
2. **链式表**
3. **存储结构：**

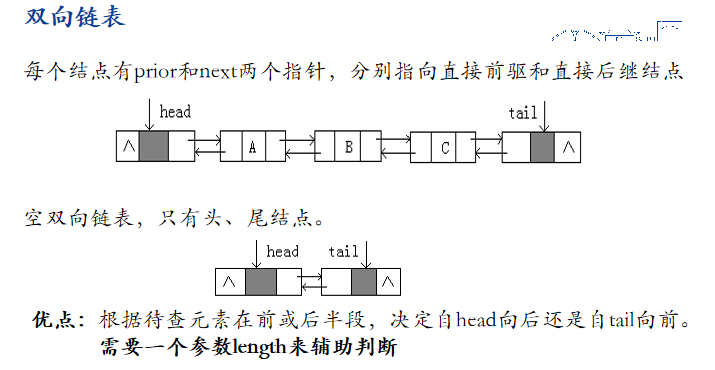


兄弟协同法”+“首席插入法”实现单链表的就地逆置？

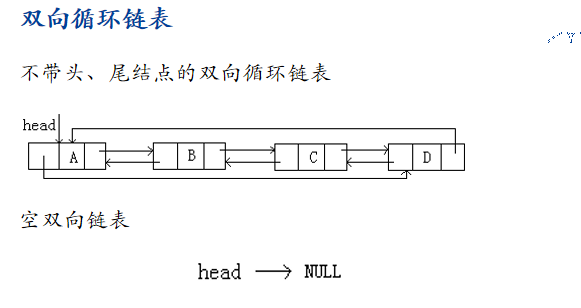
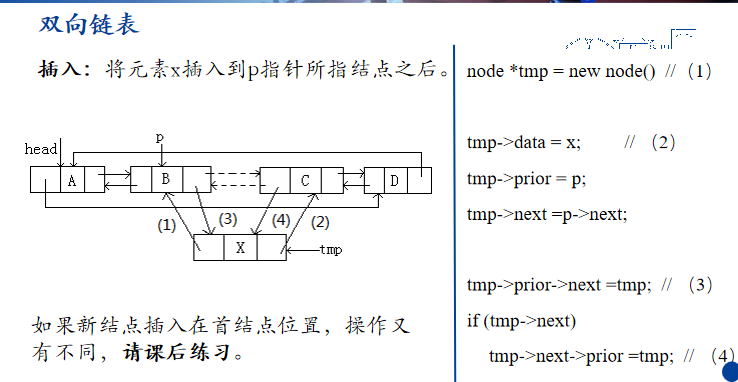
1. **单链表：**
2. **单循环链表：**



1. **双向链表：**

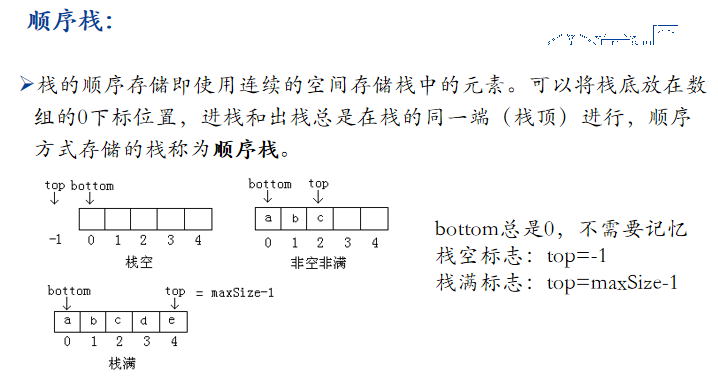


1. **双向循环链表：**

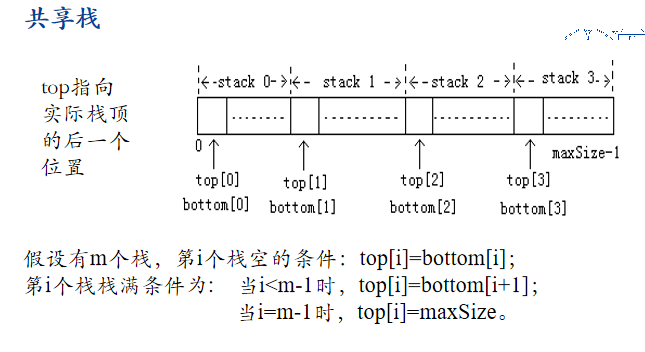
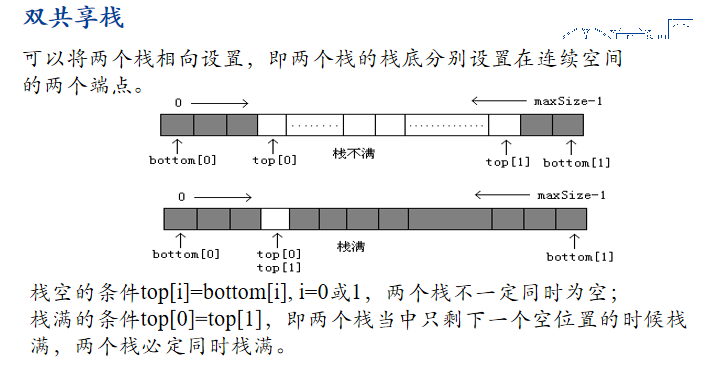


**栈**

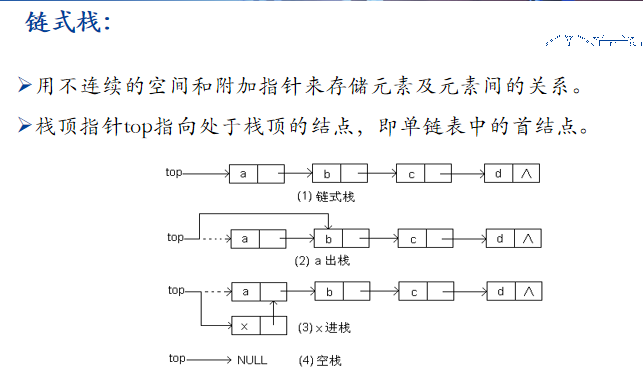
1. **顺序栈**



**1’、共享栈**

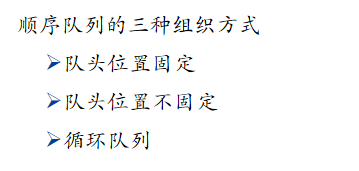


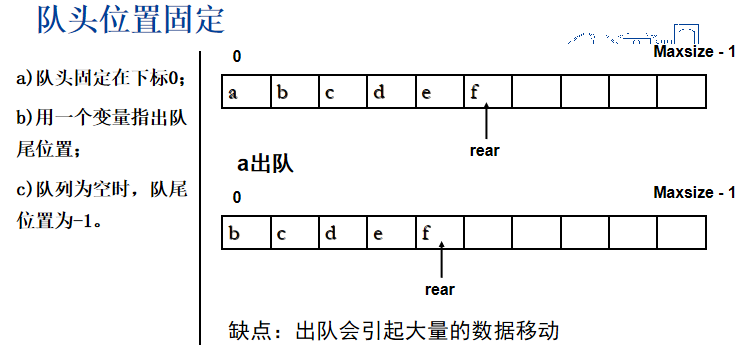
1. **链式栈**

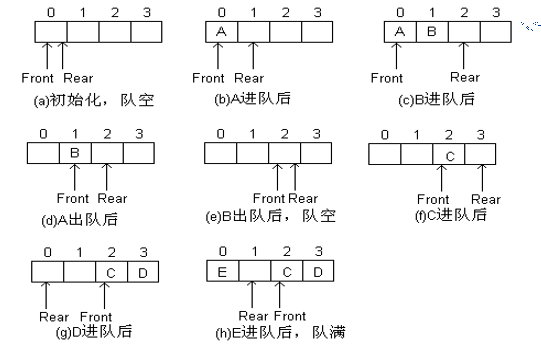


**队列**

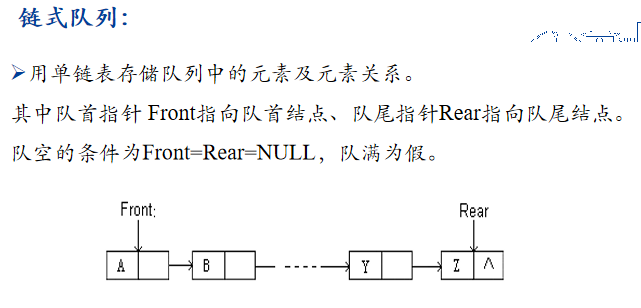
**1、顺序队列**

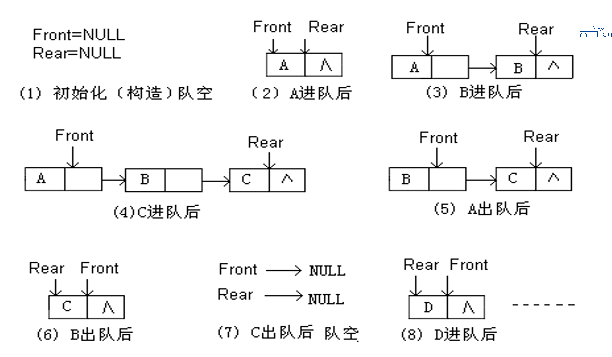






**2、链式队列**



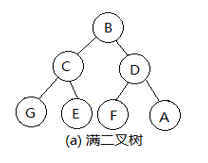


1. **优先队列**

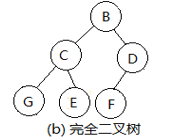
**树和二叉树**

1. **二叉树中结点个数可以为0，允许一棵空二叉树存在，这是作为工具能更方便地使用；而树中结点个数不能为0，必须至少是1。**
2. **二叉树中左右孩子要明确指出是左还是右，即便只有一个孩子，也要指明它是左子还是右子；有序树中孩子只是进行了排序，没有左右之分，当某个结点只有一个孩子时，只能说明它是大孩子，不需要确定其是左是右。**
3. **特殊二叉树**

**（1）满二叉树：如果一棵k层二叉树中每一层结点数量都达到了最大值，该二叉树称为满二叉树。**



**（2）完全二叉树：如果一棵二叉树有k层，其中k-1层都是满的，第k层可能缺少一些结点，但缺少的结点是自右向左的，这样的二叉树称为完全二叉树。**



1. **二叉树的性质**
2. **一棵非空二叉树的第i层上最多有2^(i-1)个结点（i≥1）**
3. **一棵高度为k的二叉树，最多具有2^k-1个结点。**
4. **对于一棵非空二叉树，如果叶子结点个数为n0，度数为2的结点个数为n2，则有: n0＝n2＋1。**

**(4)**

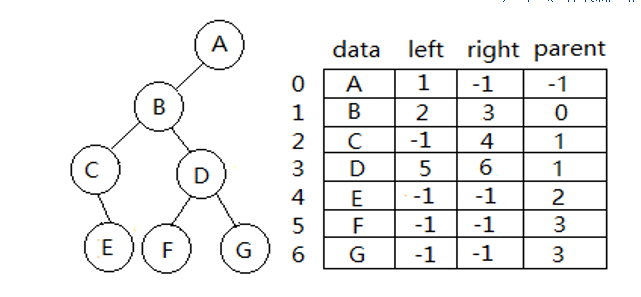
**(5)如果对一棵有n个结点的完全二叉树中的所有结点按层次自上而下、每一层自左而右依次对其编号。若根结点的编号为1，则编号为i的结点（1≤i≤n)，有以下性质：**

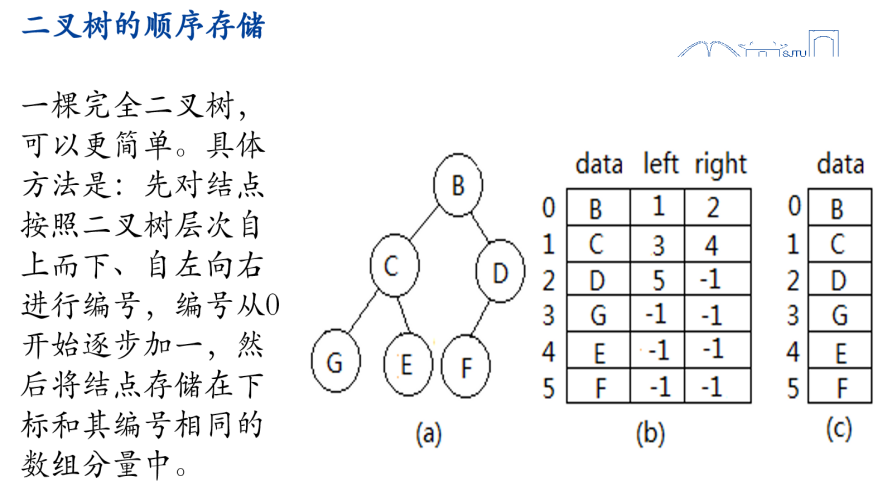
**A.如果i＝1，该结点是二叉树的根结点；如果i>1，其父亲结点的编号为i/2。**

**B.如果2i > n，编号为i的结点无左儿子；否则，其左儿子的编号为2i。**

**C.如果2i+1>n，编号为i的结点无右儿子；否则，其右儿子的编号为2i+1。**

**5、二叉树的顺序存储**

**（1）二叉树：**

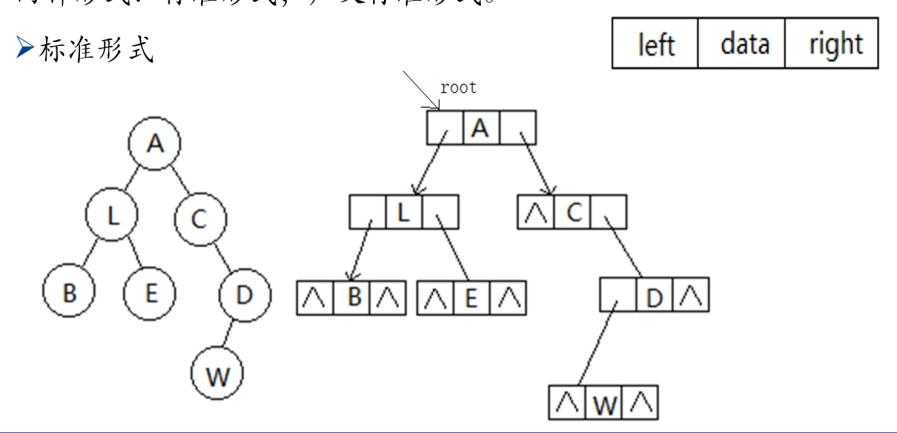
**（2）完全二叉树：**

**（3）优缺点：**

**好处是数组访问简单；**

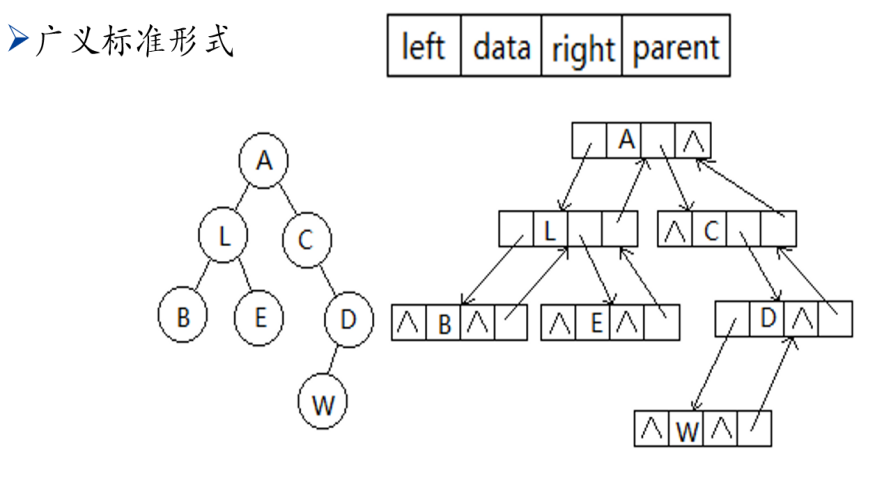
**坏处是它需要事先预估出数据的最大规模。**

**6、二叉树的链式存储**

**（1）标准模式：**

标准模式的链式存储模式最为直观

**（2）广义标准模式：**



1. **二叉树的遍历**
2. **、层次遍历：如果二叉树为空，遍历操作为空。否则，从第一层开始，从上而下，逐层访问每一层结点。对同一层结点，自左向右逐一访问。---思路和建立二叉树相同**

**基于递归的方法：**

1. **前序遍历：如果二叉树为空，遍历操作为空。否则，先访问根结点，然后前序遍历根的左子树，再前序遍历根的右子树。可简记为：“根左右”。**
2. **中序遍历：如果二叉树为空，遍历操作为空。否则，先中序遍历根的左子树，然后访问根结点，最后中序遍历根的右子树。可简记为：“左根右”。**
3. **后序遍历：如果二叉树为空，遍历操作为空。否则，先后序遍历根的左子树，然后后序遍历根的右子树，最后访问根结点。可简记为：“左右根”。**
4. **遍历序列确定二叉树**

**（1）已知一棵完全二叉树的层次遍历，能唯一确定这个完全二叉树。**

**（2）已知一棵满二叉树的前序、中序、后序遍历之一，**

**能唯一确定这棵满二叉树。**

**（3）已知一个一般二叉树的前序、中序、后序遍历之一，**

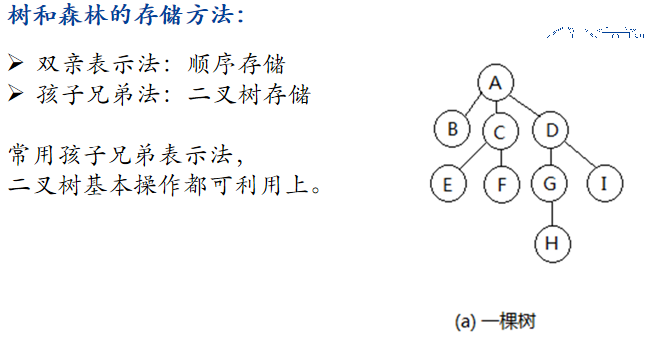
**是不能唯一确定这棵二叉树的。**

**当给了一棵二叉树的前序和中序遍历序列，能唯一确定一棵二叉树；当给了一棵二叉树的后序和中序遍历序列，能唯一确定一棵二叉树；当给了一棵二叉树的前序和后序遍历序列，不能唯一确定一棵二叉树。**

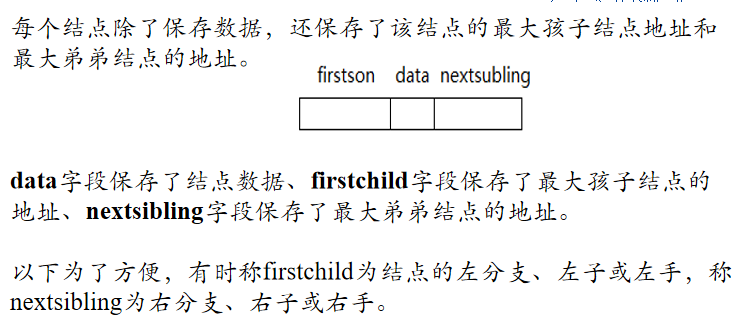
1. **二叉线索树**

**树和森林**

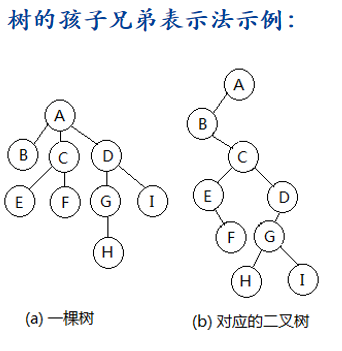
**1、**



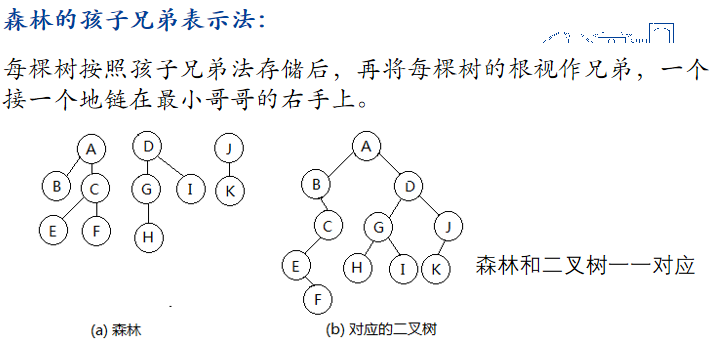
1. **孩子兄弟表示法**



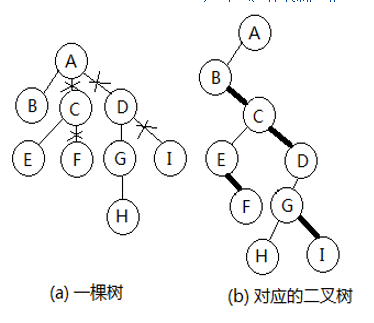
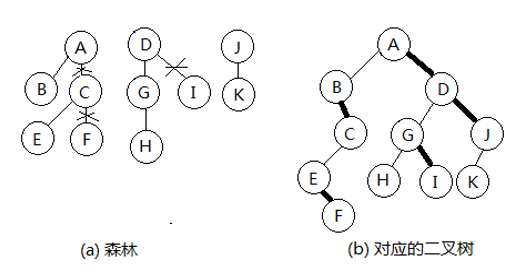
1. **树的孩子兄弟表示法**



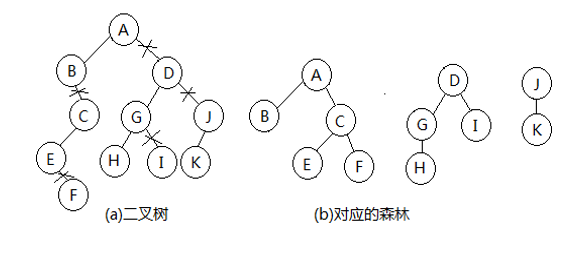
1. **森林的孩子兄弟表示法**



1. **树/森林转化为二叉树**



1. **二叉树转化为树/森林**



**2、树的先根/后根遍历**



1. **森林的先序遍历和后序遍历**
2. **优先级队列**
3. **最优二叉树**

**（1）定义：二叉树中任意两个结点间的路径长度为其路径上的分支总数。二叉树的路径长度为根到树中各个结点的路径长度之和。**

**特别地：如果二叉树中叶子结点上带有权值，二叉树的加权路径长度特指从根结点到各个叶子结点路径上的分支数乘以该叶子的权值之和。**

**当一组叶子的权值确定后，假设分别为{w1,w2,⋯,wn}，**

**将这些叶子以何种策略挂在一棵二叉树上，或者说这棵二叉树是怎样的形态，才能使其带权路径WPL达到最小？**

**把能使WPL达到最小的二叉树，称为最优二叉树。**

1. **哈夫曼算法：**
2. **哈夫曼算法就是利用了权值越大，越靠近根的思想，逐步比较结点的权值，构造出了哈夫曼树，哈夫曼树是一棵最优二叉树。**
3. **哈夫曼算法的具体步骤为：**

**对于给定的一个带权结点集合U**

**（1）如果U中只有一个结点，操作结束，否则转向2）。**

**（2）在集合中选取两个权值最小的结点x、y，构造一个新的结点z。新结点z的权值为结点x、y的权值之和。在集合U中删除结点x和y并加入新结点z，然后转向1）。**

**C、哈夫曼算法的实现**

**（1）二叉树用顺序存储法：用一个数组存储哈夫曼结点。**

**哈夫曼结点（叶子、中间结点）**

**HuffmanNode：包含5个字段：**

**data为结点的值**

**weight为结点的权值**

**parent为结点的父结点地址（下标）**

**left和right为结点的左、右孩子的地址（下标）。**

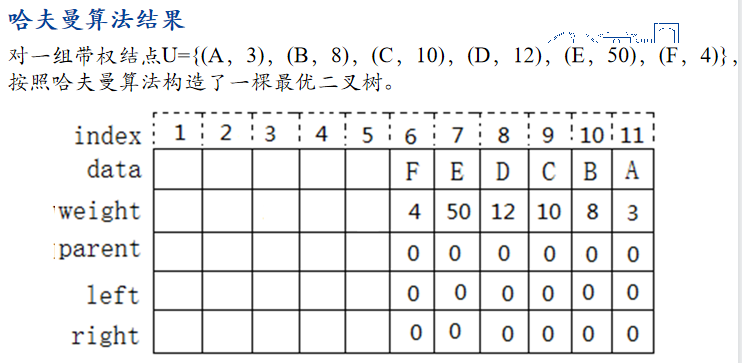
**特殊地：数组的0下标分量空出来不用，从下标为1的数组分量开始存储数据。**

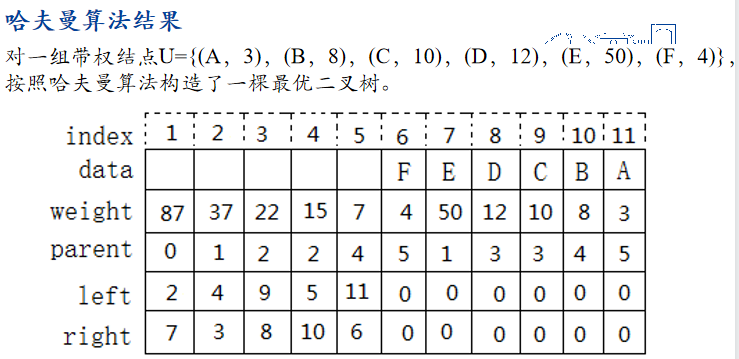
**（2）假定树中叶子结点有n个，中间结点都是由两个结点构造而成的，度均为2。按照n2=n0-1性质，最优二叉树中结点总数为：n+n-1=2n-1个。**

**开辟动态数组时，考虑到0下标分量不用，需要为数组申请2n个连续的结点空间。**

**初始时，数组中只有叶子结点，其parent、left、right都设置为0。**

**叶子结点在数组中从后往前依次存储，前面空余的n-1个数组分量作为存储即将构造的中间结点用。**

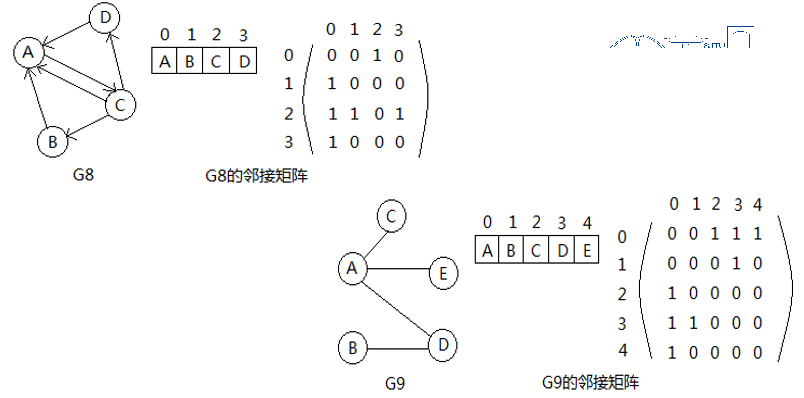




1. **哈夫曼编码**

**图**

1. **图的概念**
2. **有向图/无向图**
3. **出度、入度/度**
4. **有向完全图（n（n-1））/无向完全图（n（n-1）/2）**
5. **网络：加权有向图/加权无向图**
6. **路径 路径长度（是顶点i到顶点j之间的无向边或有向边的条数；如果边上有权重，路径长度也可以用路径上所有边的权重之和来表示。）**
7. **简单路径：没有点相同。**
8. **简单回路/简单环：简单路径上如果第一个顶点和最后一个顶点相同。**
9. **子图：假设有两个图G = (V,E)，G’ = (V’,E’)，且V’是V的子集，E’是E的子集，则称G’是G的子图。**
10. **联通：有路径存在**
11. **强联通图/联通图**
12. **强联通分量/联通分量（极大联通子图）**
13. **极小联通子图：生成树 该连通子图包含连通图的所有n个顶点，但只含它的n-1条边。如果去掉一条边，这个子图将不连通；如果增加一条新的边(vi,vj)，因顶点vi和vj之间原本连通，即存在一条路径，加上新加的这条边便形成了回路，有回路就不再是树。**
14. **一个联通图的生成树并不唯一**
15. **图的存储及操作实现**
16. **邻接矩阵表示法：顶点由一个一维数组存储，边由一个二维数组存储，这种存储方式。**

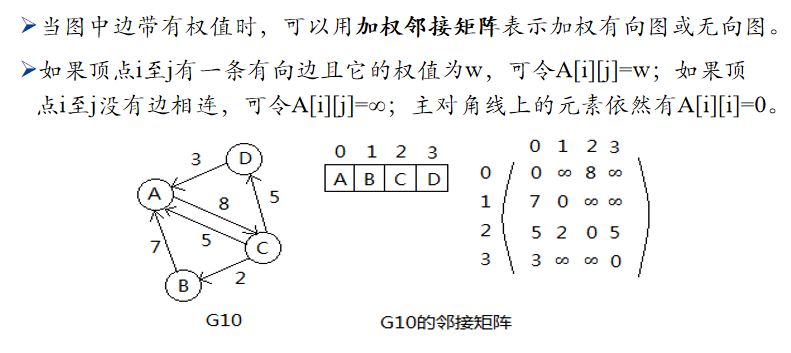


**A、在有向图中，其邻接矩阵某一行中所有1的个数，就是相应行顶点的出度；而某一列中所有1的个数，就是相应列顶点的入度。在无向图中，某一行中所有1的个数或者某一列中所有1的个数，就是相应顶点的度。**

**B、无向图中，同一条边在邻接矩阵中出现两次，无向图的邻接矩阵是以主对角线为轴对称的，主对角线全为零，因此在存储无向图时可以只存储它的上三角矩阵或下三角矩阵。（三角矩阵可特殊存储）**

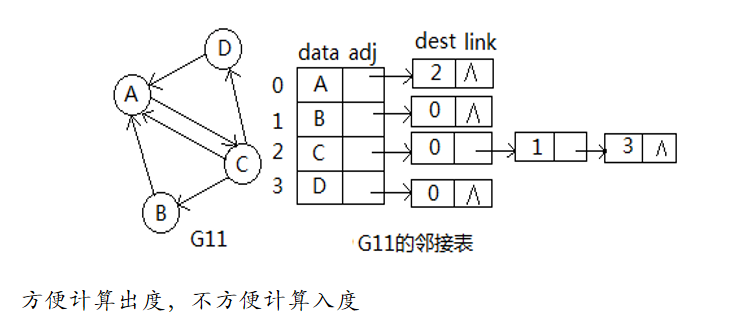
**C、一般来说，边的总数即便远远小于n2 ，也需n2 个内存单元来存储边的信息，空间消耗大。**

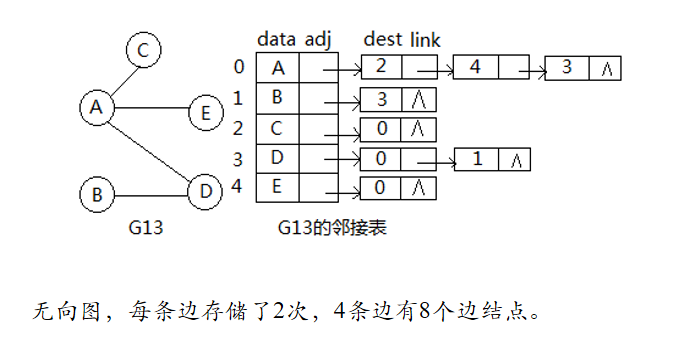
**(2)加权邻接矩阵**

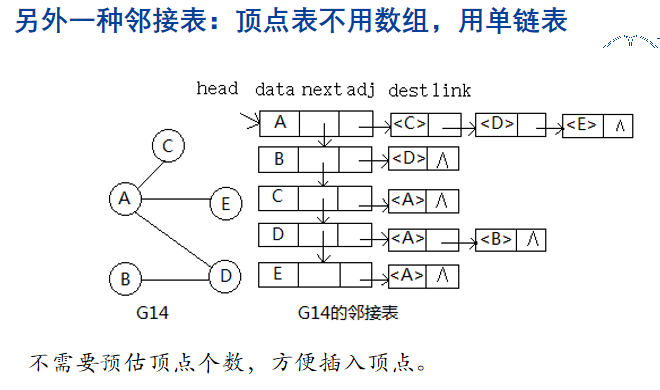


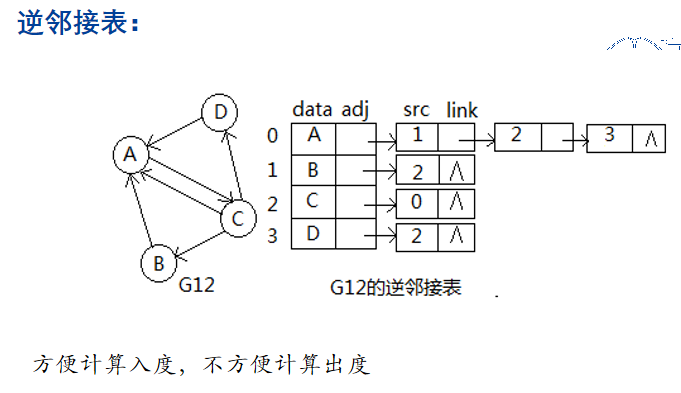
**（3）邻接表**

**A、定义：对于无向图，邻接于同一个顶点的所有边形成一条单链表；对于有向图，自同一个顶点出发的所有边形成一条单链表。顶点信息可以用一个一维数组来存储，这个数组称为顶点表，保存边信息的单链表称为边表。一个图可以由顶点表和边表共同表示，这种方法称为邻接表表示法。**

**B、**

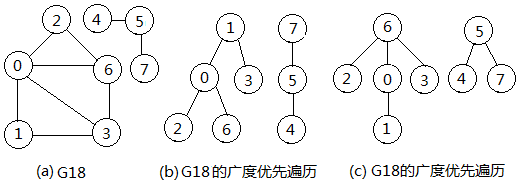




1. **逆邻接表**
2. **定义：对于有向图，射向同一个顶点的所有边形成一条单链表。顶点信息可以用一个一维数组来存储，这个数组称为顶点表，保存边信息的单链表称为边表。一个图可以由顶点表和边表共同表示，这种方法称为逆邻接表表示法。**
3. 
4. **图的遍历**
5. **广度优先遍历（借助队列）（DFS）**

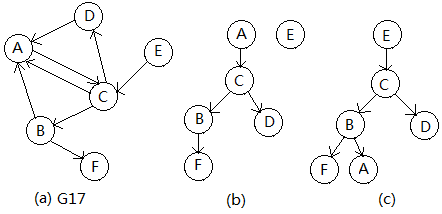
**A、定义：访问方式类似于二叉树的层次访问，访问方式如下：**

1. **选中第一个未被访问过的顶点。**
2. **访问、对顶点加已访问标志。**
3. **依次从顶点的未被访问过的第一个、第二个、第三个…… 邻接顶点出发，依次进行深度优先搜索。即转向2。**
4. **如果还有顶点未被访问过，选中其中一个作为起始顶点，转向2。如果所有的顶点都被访问到，结束。**

**B、同一个图的广度优先遍历结果并不唯一，图中就是对G18的两种不同的深度遍历结果。**

**C、广度优先遍历的算法思想：**

1. **程序首先将所有顶点的访问标志初始化为false，然后进入外层for循环。**
2. **在外循环中，顺序找未被访问过的顶点作起始顶点，将起始顶点进队，然后反复执行以下循环：顶点出队，如果未访问过，访问之并将它所有未被访问过的邻接点进队，反复循环，直到队空。**
3. **继续下一轮外循环，直到所有的顶点都被检查过。**
4. **深度优先遍历（栈和递归）（BFS）**
5. **定义：访问方式类似于二叉树的前序访问，访问方式如下：**
6. **选中第一个未被访问过的顶点。**
7. **访问、对顶点加已访问标志。**
8. **依次从顶点的未被访问过的第一个、第二个、第三个…… 邻接顶点出发，依次进行深度优先搜索。即转向2。**
9. **如果还有顶点未被访问过，选中其中一个作为起始顶点，转向2。如果所有的顶点都被访问到，结束。**

**B、同一个图的深度优先遍历结果并不唯一，图中就是对G17的两种不同的深度遍历结果。**

**C、递归算法：**

**图可能不连通，从一个顶点开始做深度优先遍历可能只能访问到部分顶点，此时需要重新选择尚未访问的顶点，从它开始再次开始深度优先遍历。**

**一个顶点可能和其他多个顶点邻接，故以它为起始顶点做深度优先遍历前需检查是否已经访问过。如果未访问过，遍历才能进行。**

**D、非递归算法：**

**图的深度优先遍历，建立一个栈，选一个顶点进栈，然后反复进行以下操作：如果栈不空，弹出访问，第一个未被访问的邻接点进栈，第二个未被访问的邻接点进栈，⋯，最后一个未被访问的邻接点进栈。**

1. **图的连通性**
2. **无向图的联通性**

**A、如果无向图是连通的，那么选定图中任何一个顶点，从该顶点出发，通过遍历，就能到达图中其他所有顶点。**

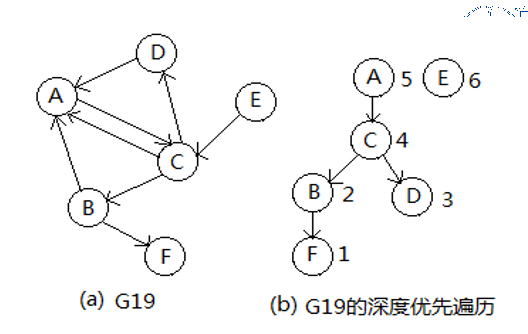
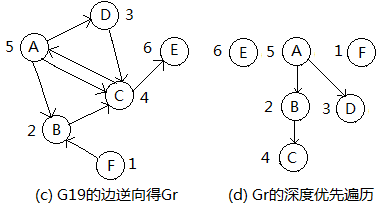
**这在以上的深度优先、广度优先遍历算法实现中增加一个计数器，记录外循环体中，进入内循环的次数，根据次数可以判断出该图是否连通、如果不连通有几个连通分量、每个连通分量包含哪些顶点。**

**B、连通性和连通分量算法实现：**

1. **有向图的连通性**
2. **有向图的连通性和其强连通分量是同一个问题：当有向图的强连通分量只有一个时，说明它是强连通图。当有向图的强连通分量不止一个时，说明它不是强连通图。**

**有向图的强连通分量问题解决起来比较复杂：对一个强连通分量来说，要求每一对顶点间都有路径可达，比如顶点i和j，不光要从i能到j，还要求从j能到i。**

1. **算法思想：**
2. **对有向图G进行深度优先遍历，按照遍历中回退顶点的次序给每个顶点进行编号。最先回退的顶点的编号为1，其它顶点的编号按回退先后逐次增大1。**
3. **将有向图G的所有有向边反向，构造新的有向图Gr。**
4. **根据步骤1对顶点进行的编号，选取未访问过的最大编号顶点。以该顶点为起始点在有向图Gr上进行深度优先遍历。如果没有访问到所有的顶点，则从剩余的那些未被访问过的顶点中选取编号最大的顶点，以该顶点为起始点再进行深度优先遍历，如此反复，直至所有的顶点都被访问到。**



**在Gr的生成森林中，每一棵生成树的顶点集都是和一个强连通分量的顶点集一一对应的。**

1. **欧拉回路**
2. **相关概念：**

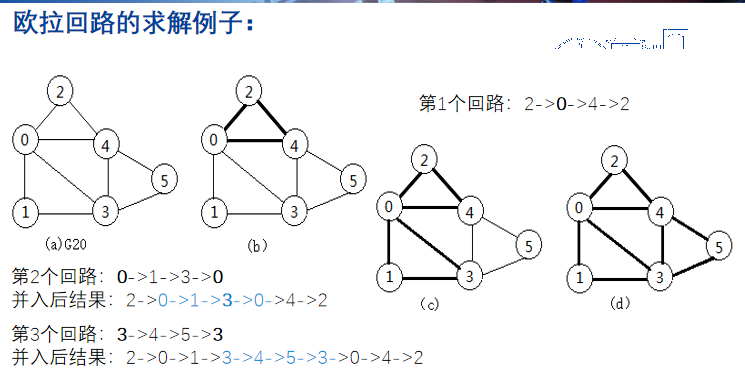
**如果图中一条路径经过了每条边一次且仅一次，这条路径称欧拉路径。**

**如果一条欧拉路径的起点和终点相同，是一个回路，称欧拉回路，具有欧拉回路的图称欧拉图(简称E图)。**

**具有欧拉路径但不具有欧拉回路的图称半欧拉图。**

**欧拉回路及一笔画问题。**

1. **欧拉定理：**
2. **一个无向连通图中，如果度为奇数的顶点超过了2个，则欧拉路径是不存在的。**
3. **一个无向连通图中，如果除了两个顶点的度是奇数而其他顶点的度都是偶数，则从一个度为奇数的顶点出发一定能找到一条经过每条边一次且仅一次的路径回到另外一个度为奇数的顶点。**
4. **一个无向连通图中，如果顶点的度都是偶数，则从任意一个顶点出发都能经过每条边一次且仅一次并回到原来的顶点。**
5. **求欧回路的算法：**
6. **任选一个顶点v，从该顶点出发开始深度优先搜索，搜索路径上都是由未访问过的边构成，搜索中访问这些边，最后直到回到顶点v且v没有尚未被访问的边，此时便得到了一个回路，此回路为当前结果回路。**
7. **在搜索路径上另外找一个尚有未访问边的顶点，继续如上操作，找到另外一个回路，将该回路拼接在当前结果回路上，形成一个大的、新的结果回路。**
8. **如果在新的结果回路中，还有中间某结点有尚未访问的边，回到2）；如果没有任何中间顶点尚余未访问的边，访问结束，当前结果回路即欧拉回路。**



1. **AOV网和AOE网**

**（1）有向无环图的应用通常分为两种：一种是AOV(Activity On Vertex NetWork)网，一种是AOE(Activity on Edge Network)网。**

**（2）AOV网（拓扑排序）**

**A、定义：将活动赋予顶点之上，顶点间的有向边表示活动发生的先后顺序，表达了活动之间的前后关系。 AOV网的一个典型应用是课程的先修关系。**



**B、偏序和全序关系：**

**在一个集合X中，若关系R有如下特点： 关系R是传递的、自反的、反对称的，就称R是集合X上的偏序关系。**

**若集合X上关系R是一个偏序关系，且对于每个a, b∈X，必有aRb或bRa，就称R是集合X上的全序关系。**

**Eg.实数轴上的实数集合，以及集合上的≤关系，是实数集合上的全序关系。**

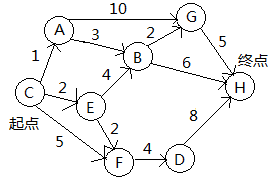
**C、拓扑排序：对集合X上的一个偏序关系R，通过将集合中原本不满足R关系的所有元素对人为地补充设定拥有R关系，从而将R改变为集合X上的一个全序关系，并按照此全序关系将元素排成一个线性序列。**

**在这个线性序列a1、a2、…、an中，如果偏序关系中aiRaj，必有个i≤j，这个序列称为拓扑序列，获得拓扑序列的操作称为拓扑排序(Topological Sort )。**

1. **拓扑排序算法：**
2. **首先在图中，找到入度为0的顶点，将这些顶点全部入栈，然后反复循环判断栈是否空，非空则执行以下操作：**
3. **顶点出栈，如果由该顶点射出了m条有向边，射入的这m个邻接点的入度减一（相当于该顶点对其m个邻接顶点的先修约束已经消失），在各邻接点入度减一的过程中，一旦发现哪个邻接点的入度变为0，将它进栈，然后再次回到循环，直到栈空。**
4. **在这个方法中，也可以使用队列来代替栈。**
5. **一个AOV网的拓扑序列不一定唯一。**
6. **利用拓扑排序算法可以判断一个有向图是否存在有环。**

**（3）AOE网（关键路径）**

**A、定义：将活动赋予边之上，顶点表达了活动发生后到达的某种状态或事件。某个状态或事件既意味着前面所有的活动结束，也意味着后面的活动可以开始。AOE网的一个典型应用是工程问题。**



**B、关键路径问题：**

1. **一个工程通常由若干个子工程构成。**
2. **大多子工程在开始实施时既要有前期子工程完成为条件，自身也需要一定的时间来完成。**
3. **如何根据这些信息求得工程的总工期？在整个工程项目中哪些子工程是关键的子工程？所有的关键子工程必须在可以开始时马上开始，中间不得拖延工期，必须按照计划如期完成，否则将影响整个工程工期。每个不是关键子工程的工程有多少时间余量？这些问题都是工程施工前要精心计算的。**
4. **关键子工程即关键活动会形成一条从总体工程开始和完工之间的路径，这条路径便是关键路径。**

**C、利用AOE网求工程中的关键活动的方法：**

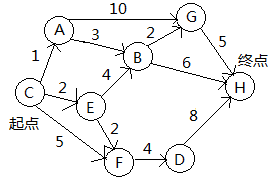
**求每个顶点事件的最早发生时间，即从起点到达顶点所需要的最短时间。**

**求每个顶点事件的最迟发生时间，即从起点到达顶点所能容忍的最长时间。**

**求每个活动的最早开始时间，即每个边表示的活动最早何时能开始。**

**求每个活动的最迟开始时间，即每个边表示的活动最晚何时必须开始。**

**⭐当某活动的最早开始时间和最迟开始时间相同时，这些活动便是关键活动。**



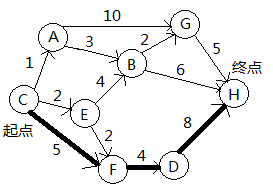
对于AOE网中的一个活动<u,v>，一旦顶点u事件发生，由u射出的边<u,v>所表示的活动就可以进行了，因此活动<u,v>的最早发生时间是顶点u事件的最早发生时间。

而活动<u,v>的最迟进行（发生）时间是顶点v事件的最迟发生时间减去边<u,v>的权值。

**D、求关键路径：**

当活动的最早发生时间和最迟发生时间一致时，表示该活动为关键活动，这些关键活动组成的由起点到终点的路径为关键路径。

关键活动在最早发生时间时就必须马上开始，不得延缓。

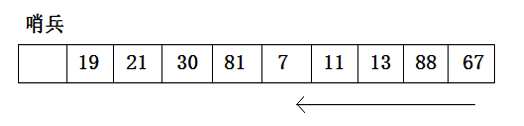


1. **查找**

**静态查找**

1. **顺序查找**

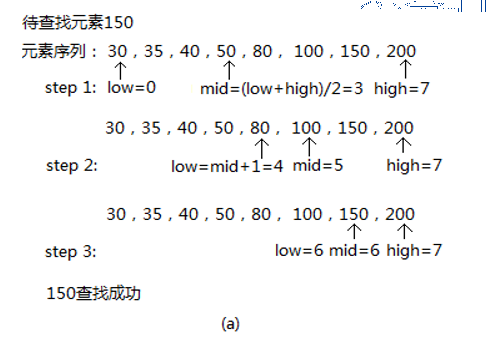
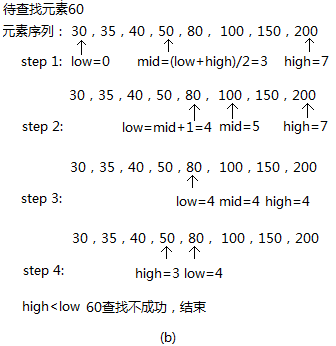
**（1）方法：把0下标位置设置成一个哨兵位，当要查找数据x时，首先将x存入哨兵位，然后从n下标开始向前一直找到0下标。假设下标m的数据是要查找的x，则当m大于0时，表示查找成功；m等于0时，表示查找不成功。加哨兵位的好处是避免了逐个检查数组元素时还要首先检测下标是否越界，用浪费一个空间的方式换取时间，提高了效率。**



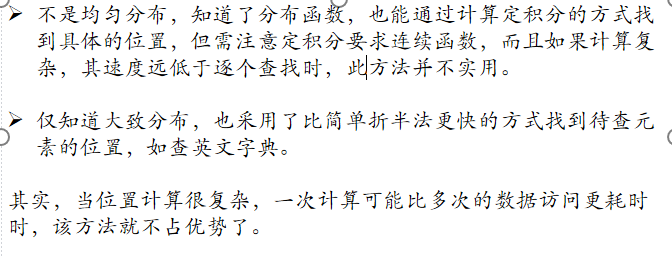
1. **折半查找（元素有序时）**

**（1）方法：首先比较中间位置元素，比较成功，查找结束；**

**比较不成功，1)如果待查找元素小于中间位置元素，在前半段使用上述同样方法继续查找。特殊地，如果前半段没有了，即长度为0，说明不存在待查找元素；2)如果待查找元素大于中间位置元素，在后半段使用上述同样方法继续查找。特殊地，如果后半段没有了，也说明不存在待查找元素。**

1. **插值查找（不仅数据是有序的，而且这组数据值的分布是均匀的。）**
2. **方法：折半查找每次简单地找中间位置，插值查找是根据待查元素值和两个端点即最大最小元素值的距离来计算或估算下次查找的位置: mid=low+(high−low) ((x−a[low])/(a[high]−a[low]))。理想的插值查找的时间复杂度为O(1)。**

**比如一组值为{1，3，5，7，9，11，13，15，17}，数组有序且分布非常均匀，查找11只需要通过计算mid=0+(8-0)(11-1)/(17-1)= 5便可以得到11的下标位置。**



**分块查找是把一个大的线性表分解成若干块，块中数据可以无序，任意存放，但块间必须有序。**

**要建立一个索引表，把每块中最大关键字值作为索引表的关键字，从小到大顺序存放在一个辅助数组中。查找时，可以先在索引表中折半查找，确定要找的元素所在的块，然后在块中采用顺序查找，即可找到对应的元素。故该方法也称“索引顺序查找”。**

**一般适用于数据量很大，无法一次调入内存的情况。特别适合于元素动态变化的情况。当增加或减少元素时，只需将该元素调整到所在的块即可。**

1. **在时间复杂度上，分块查找的速度虽然不如折半查找算法，但比顺序查找算法快得多。**

**动态查找**

**数据不稳定、频繁地插入、删除。链式存储**

1. **二叉查找树**
2. **定义：对二叉树中的每个结点而言，其左子树上的所有结点都比它小，其右子树上的所有结点都比它大。**
3. **查找：**
4. **递归：**

**根为空，查找失败。**

**否则和根结点值比较，相同则查找成功；不相同但比根结点值小，在以其左子为根的二叉查找树中继续查找；不相同但比根结点值大，在以其右子为根的二叉查找树中继续查找。**

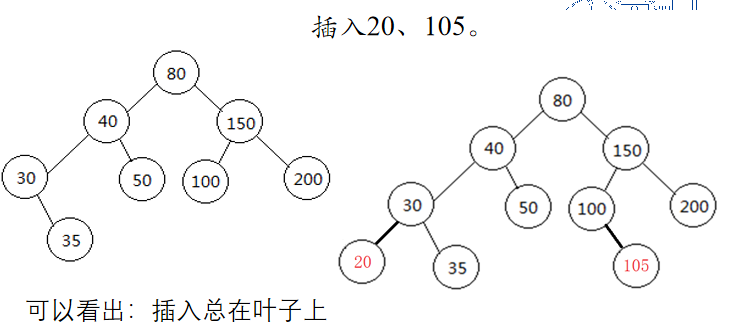
**B、非递归：**

**以根为当前结点。如果当前结点不为空，待查结点和当前结点比较，如果值相等则查找成功；如果比当前结点值小，设其左子为当前结点；如果比当前结点值大，设其右子为当前结点；反复如此，直到当前结点为空。**

**C、时间复杂度：**

**从根开始逐层比较，比较次数最多为二叉树的高度，故时间复杂度为二叉树高度。**

1. **插入**



**A、递归：根为空，直接插入。**

**否则和根结点值比较，相同则无需插入；**

**不相同但比根结点值小，在以其左子为根的二叉查找树中继续插入；**

**不相同但比根结点值大，在以其右子为根的二叉查找树中继续插入。**

1. **非递归：**
2. **如果根为空，创建新结点作为根结点。如果根不为空，设置当前结点为根结点。将待插入元素和当前结点比较，值相等则无需插入；若值小于当前结点，且当前结点无左子，创建新结点作为其左子，否则将其左子作为当前结点，继续比较；若值大于当前结点，且当前结点无右子，创建新结点作为其右子，否则将其右子作为当前结点，继续比较。**
3. **设置一个父结点指针，设根为当前结点，父结点指针为空。反复进行以下操作，直到当前结点为空：待插入元素和当前结点比较，值相等则无需插入；若值小于当前结点，父结点设为当前结点，当前结点改为其左子；若值大于当前结点，父结点设为当前结点，当前结点改为其右子。**

**创建新结点。**

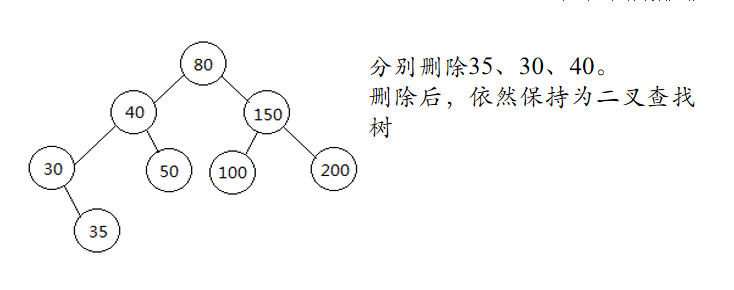
**若父结点为空，根指针置为新结点。**

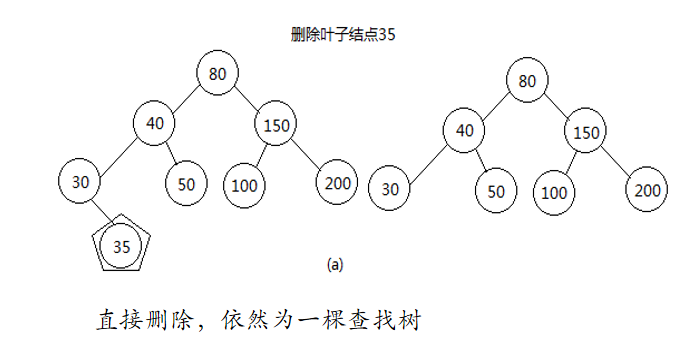
**若x小于父结点的值，新结点作为父结点的左子链入。**

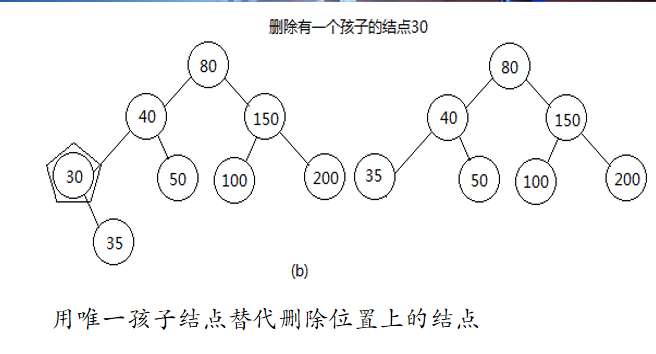
**若x大于于父结点的值，新结点作为父结点的右子链入。**

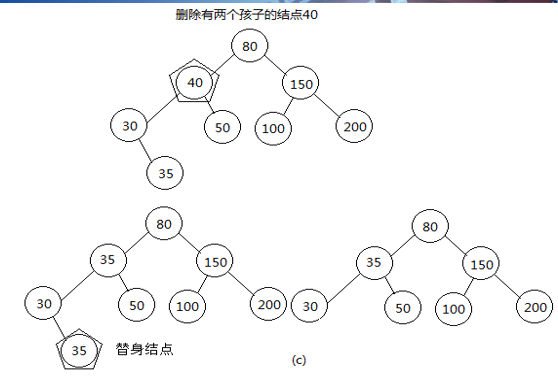
**C、事件复杂度：二叉树的高度**

1. **删除**







**找替身结点:**

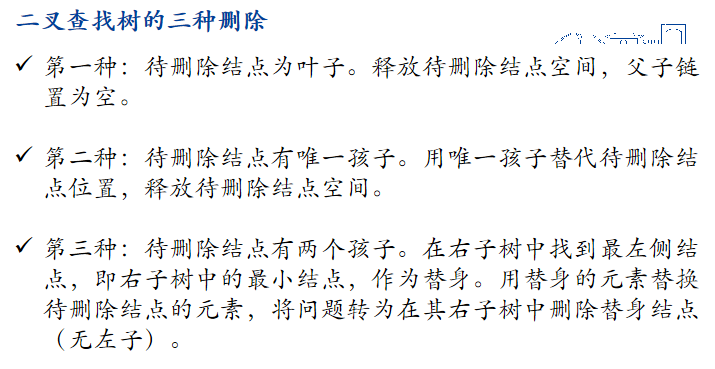
**左子树上的最大结点；即沿左子一路右子找下去。或右子树上最小结点。即沿右子一路左子找下去。**

**最后变为删替身结点。它最多只有一个孩子。问题转化为前两种。**

1. **递归：**

**根为空，删除结束。**

**否则和根结点值比较，相同则实施三种删除；**



**不相同但比根结点值小，在以其左子为根的二叉查找树中继续删除；不相同但比根结点值大，在以其右子为根的二叉查找树中继续删除。**

1. **非递归：**

**a：非递归方法一：**

**由根向下查找，每层有一个结点参与。用p指向当前结点，parent指向其父结点。父一路跟随其子下行。根据parent完成对父结点孩子字段的修改任务。**

1. **时间最多为树高的两倍**
2. **平衡二叉查找树**
3. **定义：**

**结点的平衡因子：一个结点的平衡因子等于其左子树的高度减去其右子树的高度。如果一棵二叉树中所有结点的平衡因子的绝对值不超过1，即为-1，0，1这三种情况，这棵二叉树就称为平衡二叉树。平衡二叉树的目标是限制二叉树的高度**

**⭐平衡二叉查找树并不能直接和树高最矮划等号，但平衡二叉查找树已经接近于最矮了。**

1. **查找：同普通查找树一样**
2. **插入：新插入结点的平衡因子为0，一路自下而上往祖先结点传导。如果传导来自于左子树，说明左子树高度增加了1，父结点平衡因子加1；如果传导来自于右子树，说明右子树高度增加了1，父结点平衡因子减1。**

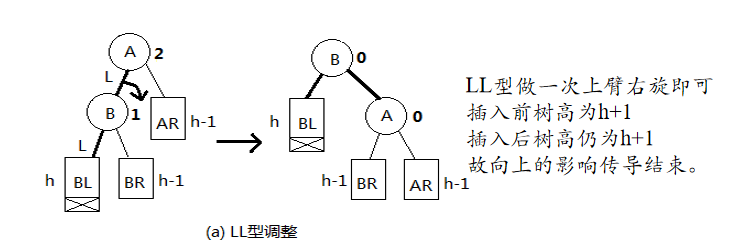
**父结点平衡因子变化后，如果结果变为0，说明原本的左右子树一边高、一边低，现在低的长高了，变得和高的一样了，以父结点为根的子树高度没有变化，自下而上的传导行为停止，祖父包括更上层祖先结点的平衡因子保持不变。**

**父结点平衡因子变化后，如果结果变为非0，依然按照传导来自左子树加1、右子树减1的原则向祖父结点传导，直到某一层祖先结点的平衡因子变为0、+2或者-2、或者到达根结点。**

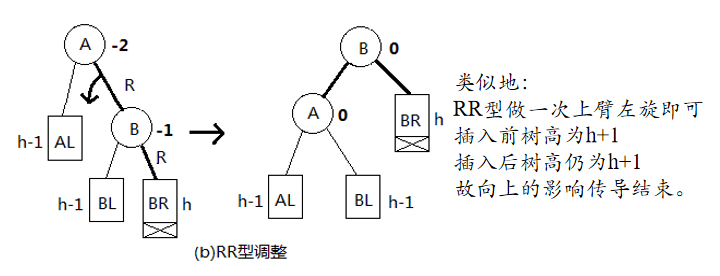
**冲突结点：二叉查找树中一旦有一个结点的平衡因子不在-1，0，1的范围内，二叉树就不再平衡。在向上的传导过程中，平衡因子第一个超过-1，0，1范围的结点称为冲突结点。一旦发现冲突结点，暂停沿祖先向上的传导，先对二叉树在冲突结点附近实施调整，直到它变得平衡。**

**⭐调整冲突结点**

1. **LL**



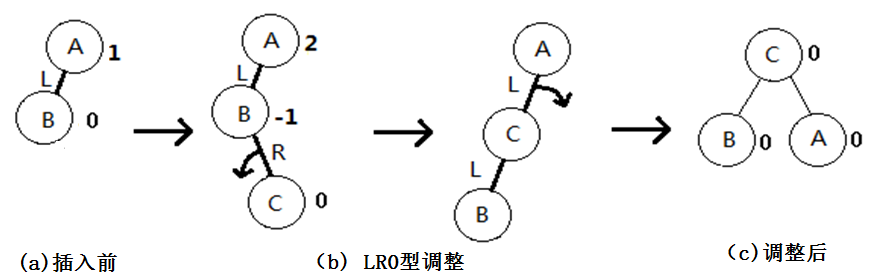
1. **RR**



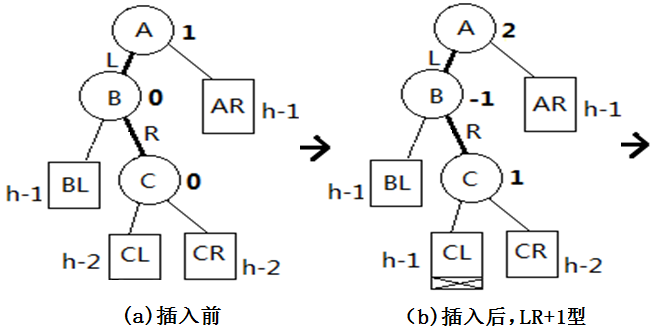
1. **LR**

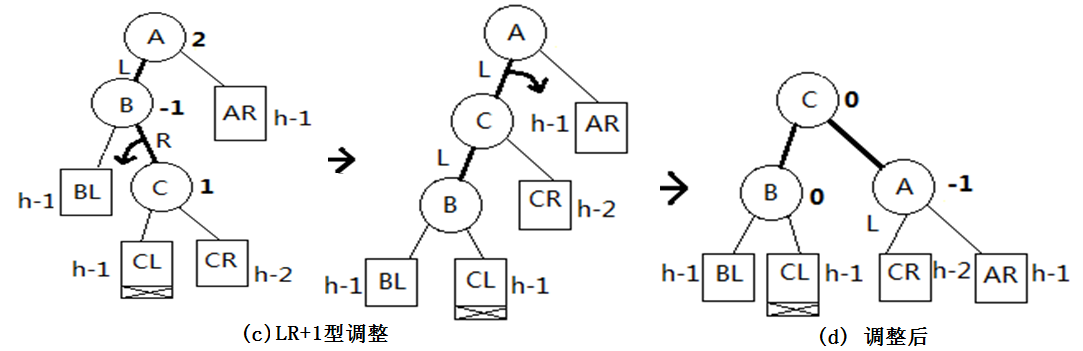
**LR有三种形态：LR0、LR+1、LR-1。**

**LR0型做双旋调整：一次下臂左旋，变LL型，一次上臂右旋，因插入前后二叉树高度均为2，向上影响传导结束**



**LR+1型做双旋调整：一次下臂的左旋，变LL型，一次上臂的右旋，插入前后高度均为h+1, 影响向上的传导结束。**





**LR-1：旋转方法如LR+1，但调整好后结点A平衡因子不同。插入前后二叉树高度均为h+1，影响向上的传导结束。**





LR0、LR+1、LR-1处理方法相同之处：

做一次双旋，先下臂左旋，变LL型，再上臂右旋。

不同之处在于双旋之后，局部根的左、右子结点平衡因子不同：

LR0： 左、右子平衡因子均为0

LR+1： 左、右子平衡因子分别为0、-1

LR-1： 左、右子平衡因子分别为1、0

1. **RL**

**也有三种形态：RL0、RL+1、RL-1**

**先下臂右旋，变RR型，再上臂左旋。**

**不同之处在于双旋之后，局部根的左、右子结点平衡因子不同：**

**RL0： 左、右子平衡因子分别为0、0**

**RL+1： 左、右子平衡因子分别为0、-1**

**RL-1： 左、右子平衡因子分别为1、0**

1. **删除**

**A、在一个平衡二叉树中删除结点，也可能造成冲突结点的出现。**

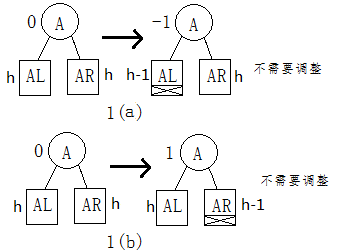
**删除分为叶子结点、只有一个孩子的结点、有两个孩子的结点。**

**有两个孩子的结点在删除时，可以通过找替身结点，最终转化为删除叶子或只有一个孩子结点的情况。**

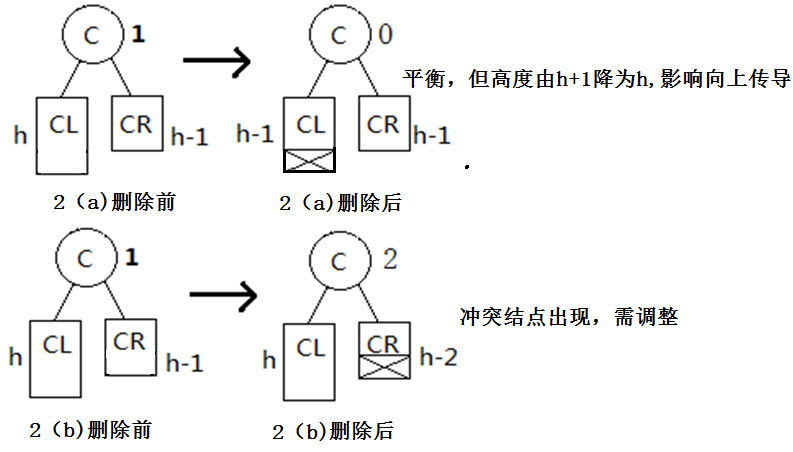
**无论是删除叶子还是只有一个孩子的结点，删除都可能引起其所属子树高度降低，自下而上各级祖先结点的平衡因子就可能发生变化。**

**B、**

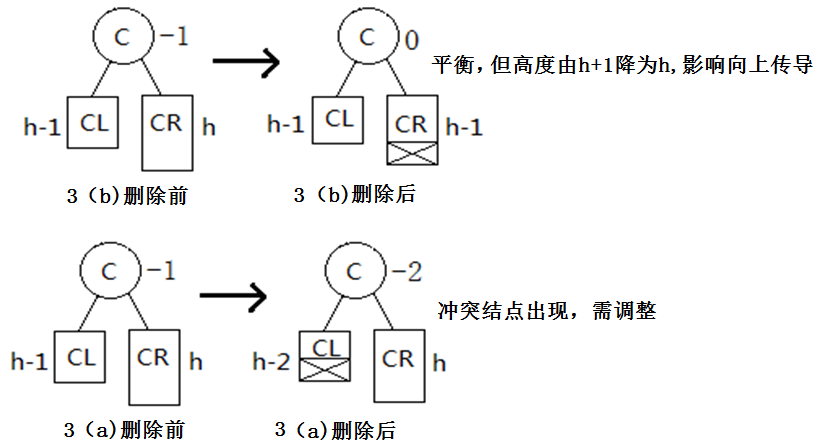
**a、当待删除结点的父结点平衡因子原本为0，左右子树一样高为h。**



**b、当待删除结点的父结点平衡因子原本为1**



**C、当待删除结点的父结点平衡因子原本为-1**



**一个祖先结点的平衡因子由0变为非0，传导结束；由非0变为0时，继续向上传导；由非0变为+2或者-2时，冲突结点确定。根据LL、RR、LR、RL四种形态分别做和插入时一样的旋转调整、做同样的平衡因子修正，调整后继续向上传导。**

**调整的分支不是自冲突结点向下到删除结点的分支，而是自冲突结点向下到不含删除结点的分支。**

**外部查找**

**1、在实际应用中，数据量往往很大，数据存储在外存储器上的文件中，且无法一次性载入内存。数据的存储常是按其写入时间的先后顺序存储，为了加快数据的查找速度，建立索引文件是最普遍的做法。**

**2、B树：**

**（1）定义：B树是一棵可以用来建立索引的多路查找树，也称多路搜索树。**

**M阶的B树须满足如下定义：**

**或者为空、或者只有一个根结点、或者除了根还有多个结点。**

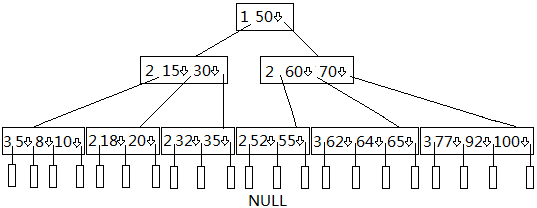
**根结点如果有子，则至少有两个儿子、至多有M个儿子。**

**除了根结点外，每个非叶子结点至少有⌈M/2⌉个儿子（向上取整），至多有M个儿子。**

**非叶子结点结构如下：**

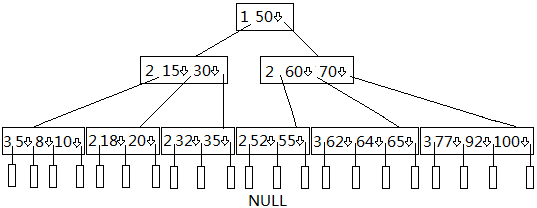
**(n,A\_0,K\_1,R\_1,A\_1,K\_2,R\_2,A\_2,⋯,K\_n,R\_n,A\_n)**

**其中：n为结点中关键字的个数，K\_i为关键字，R\_i为关键字为K\_i的数据在原始文件中的地址，A\_i−1为在树中关键字值小于K\_i的结点的地址，A\_i为在树中关键字值大于K\_i的结点的地址。n个关键字就意味着该结点有n+1个孩子**



1. **查找：找30**

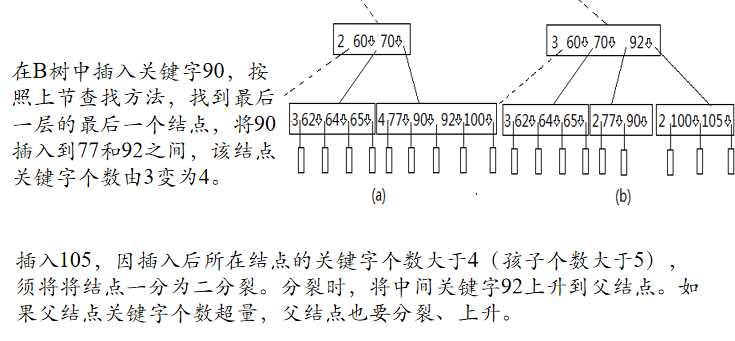
**首先从B树的根结点入手，关键字30不在根结点中，值小于50，因此顺50左侧指针走向第二层的第一个结点；30为该孩子结点的第二个关键字。关键字30找到后，其右侧空心箭头即标识其数据在原始数据文件中的具体地址。**



**B树每层只查找一个结点，如果B树作为原始数据文件的索引文件驻留在外存储器上，走向下一层时，只需根据指向的地址将B树中的一个结点读入内存，即对应着一次磁盘的访问，因此B树中一个结点的大小通常也取一次磁盘读取的数据量（称一个数据块）。**

**外存储器访问速度比内存访问速度要慢得多，降低B树的层次就能减少读取外存储器的次数。B树因为是多路搜索树，孩子的数量大于2，所以它比二叉查找树高度要少**

1. **插入**



**新建一棵B树：是一个结点逐一插入的过程**

**⭐插入操作总结如下：首先找到最后一层空结点位置，在其父结点处插入一个关键字，如果父结点中关键字个数在插入前已达上限（M-1），就需逐级向上进行结点分裂，直到某层结点中关键字个数少于上限。树的高度，就是在所有结点逐个插入的过程中分裂结点并向上建立新结点形成的。**

**⭐：算法性能分析：首先从根结点逐层向下移动查找，需要比较的次数最差为B树的高度。插入后如果引起结点分裂，最差情况是每一层都分裂、上移，直到根结点。所以总的时间消耗是B树高度的两倍。**

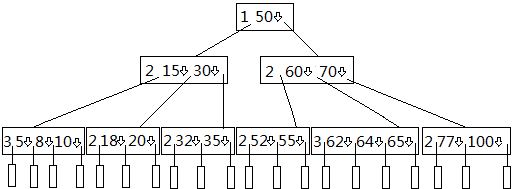
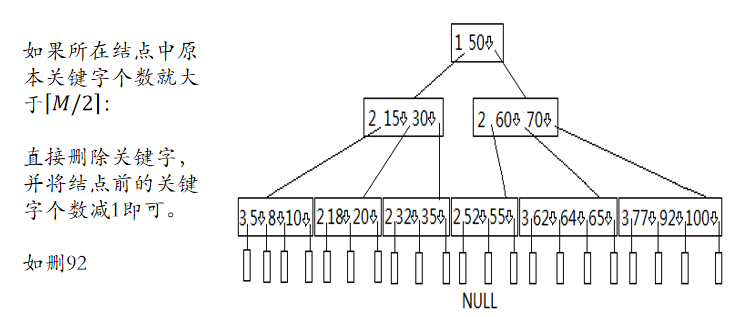
1. **删除**

**在原始数据文件中删除数据，只需要在其B树索引文件中删除该数据关键字。原始文件中数据可以暂时不做处理，留在后面做定期批量数据维护时清理。**

**删除首先也要进行查找，查找待删数据关键字在B树中的哪个结点上。然后根据结点的情况，将删除按照以下几种情况分别处理：**

1. **待删关键字在最下非叶子结点层，再下层就是空结点了：**

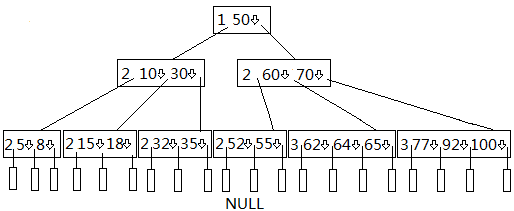
**a、关键字大于[M/2]-1**



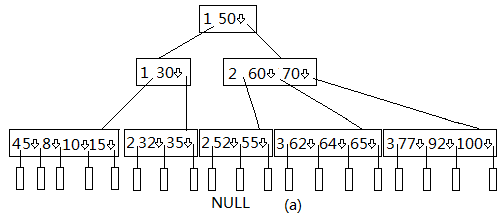
1. **关键字等于[M/2]-1**

**如果所在结点中原本关键字个数等于⌈M/2⌉-1，且左、右兄弟结点中有孩子个数非最小值:**

**从关键字个数非下限的兄弟处借过来一个。如果从左边兄弟结点处借，借最大关键字；如果从右边兄弟结点处借，借最小值。借来的关键字和父结点的一个关键字交换，将换来的父结点关键字追加到删除关键字的结点中。如删20后**

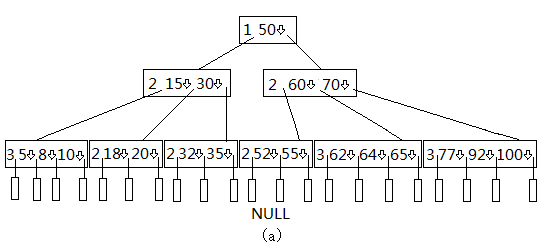


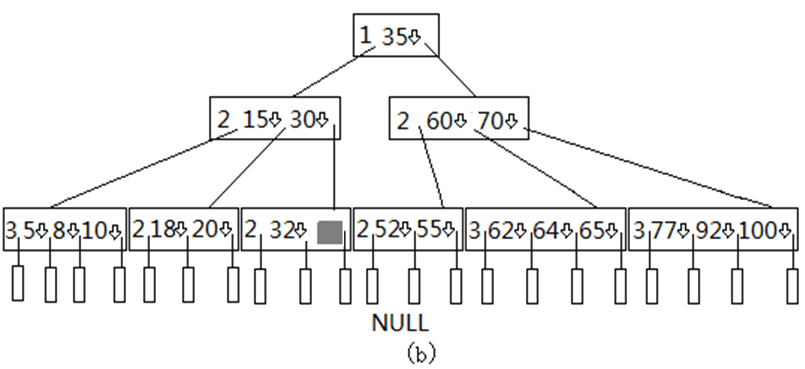
**兄弟都为下限：如果所在结点中原本关键字个数等于⌈M/2⌉-1，且左、右兄弟结点孩子个数均为最小值：*删除关键字的结点和左兄弟或者右兄弟结点合并，将父结点中介于两个合并孩子间的关键字下移加入合并结点。*如果父结点中关键字下移后，关键字个数少于⌈M/2⌉−1个，调整继续上移。如删18后**



1. **待删的关键字在中间层，下层仍为非叶子结点层：**

**在B树中找到待删关键字，顺关键字左侧子树中找到最大关键字或者顺着关键字右侧子树中找到最小关键字作为替身替代之，显然替身都在最后一层非叶子结点中，删除变为情况1。**





**（5）**

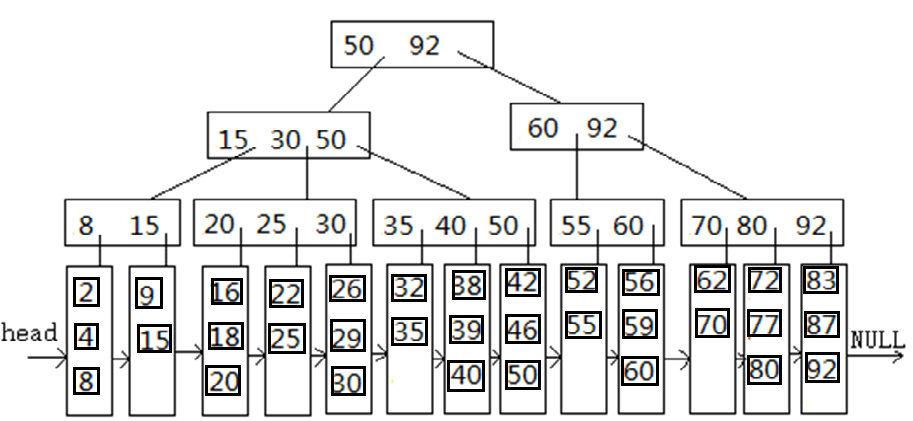
**优点：B树是一棵多线索查找树，所有的数据关键字都挂在了树的非叶子结点中，每个关键字右侧字段（空心箭头）标明了其对应数据记录在原始数据文件中的具体地址，因此根据关键字和作为索引的B树的帮助，就能非常方便地在数据文件中找到它。**

**缺点：B树的缺点也是明显的，因每个关键字都在B树上，造成B树过于庞大；另外如果要按关键字大小顺序访问所有数据，B树没有任何优势。为了解决这个问题，引入了B+树的概念。**

1. **B+树**

**（1）定义：M阶的B+树定义如下：**

1. **或者为空、或者只有一个根结点、或者除了根还有多个结点。**
2. **每个非叶子结点至多有M个儿子。**
3. **除了根结点外，每个非叶子结点至少有⌈M/2⌉儿子。**
4. **非叶子结点结构如下：(n,K\_1,A\_1,K\_2,A\_2,⋯,K\_n,A\_n)，n为关键字个数，A\_i为关键字小于等于K\_i的孩子结点地址。显然，有k个孩子的结点有k个关键字，非叶子结点并不包含所有数据的关键字。**
5. **叶子结点和中间结点不同，叶子结点存储了包含关键字在内的整个数据记录且包含了全部关键字信息，它的关键字个数最少⌈L/2⌉个，最多L个。**



**（2）查找**

**找到情况：如找30，第一层从根结点找，30小于等于50，顺50右侧指针向下；第二层最左结点中，30小于等于30，顺30右侧指针向下；第三层第2个结点中，30小于等于30，顺30右侧指针向下，找到第5个叶子结点，在该叶子中找到30，查找成功。**

**找不到情况：如找33，同样顺以上过程找到第5个叶子结点，在该叶子中找不到33，查找失败。**

**（3）插入、删除：**

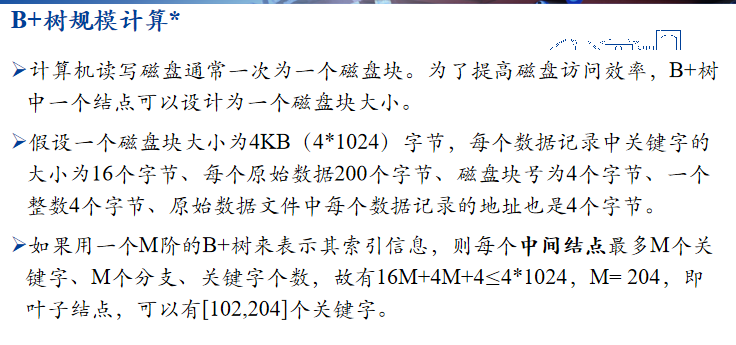
**B+树的插入和删除关键字的操作和B树类似：**

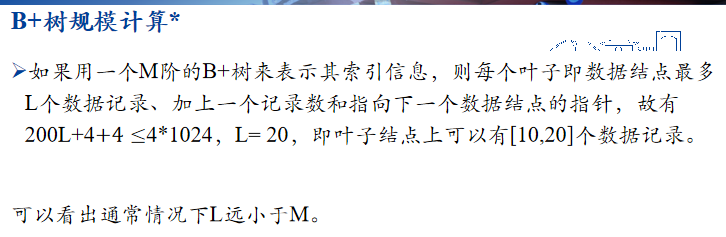
**先查找，后进行插入、删除。**

**插入总在叶子结点上进行，在插入过程中也面临着结点分裂的可能。**

**在删除的过程中也面临着借用、合并的可能，各种情况的处理方式和B树类似。**

1. **B树和B+树比较**
2. **B树关键字和对应数据记录地址信息可能分布在非叶子结点的任何结点中，所以查找可能停止在任何一层，时间消耗小于等于树的高度，最差为树的高度。而B+树查找必须从根到达叶子结点，时间消耗一定是树的高度。**
3. **B+树插入和删除操作的时间消耗，因B树和B+树中的查找时间不同而不同，但B树和B+树的插入、删除，最差时间消耗都是树高度的2倍。**
4. **B+树非叶子结点为索引层，叶子结点为数据层，索引层只包含关键字，不包含其对应数据在原始数据文件中的地址，因此可容纳更多的关键字。**
5. **B+树叶子结点层因包含了含关键字在内的整个数据记录，能容纳的数据记录最大个数L远小于非叶子结点层中关键字的个数M（B+树阶数）。**
6. **实际上利用B树和B+树建立索引的手段不仅仅用于外部查找，也可用于内部查找。**
7. **B+树索引层比B树规模小，且支持按照关键字大小从小到大顺序遍历。故B+树在建立索引文件时更加常用**
8. **B+树非叶子结点的子树指针与关键字个数相同；为所有的叶子结点增加一个链指针；所有的关键字都在叶子结点出现。**
9. **B树规模的计算？**





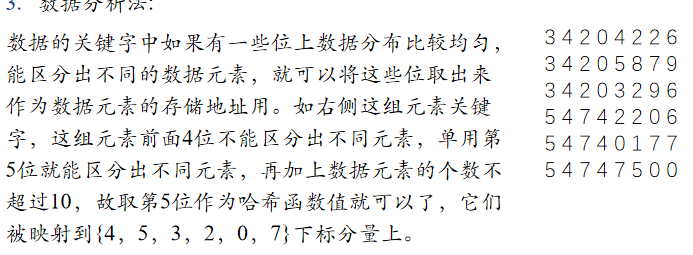
1. **哈希表**
2. **相关概念：**
3. **以一个可以存储m个数据的连续存储空间（数组）为基础，对数据关键字k\_1,k\_2,⋯,k\_n中的任意一个关键字k\_i，函数H(k\_i)的值为0到m-1之间的整数，且m≥n。**
4. **这段连续的存储空间称为哈希表，m为哈希表的大小。**
5. **函数H称为哈希函数或散列函数。**
6. **实际存储的元素和哈希表大小的比值∂=m∕n称为哈希表的负载因子或者负载系数。**
7. **负载因子越大，空间利用率越高，哈希函数就越难找。负载因子越小，冲突的可能性越小，但空间利用率越低。**
8. **好的哈希函数，须满足：计算速度快；哈希到的地址均匀、冲突少；哈希表的负载因子高，尽量减少空间的浪费。但这些要求常常互相矛盾。**
9. **一些常见的散列函数选取方法：**
10. **直接寻址法： 哈希函数为H(key) = a\*key+b。**

**如元素关键字序列{100，200，330，520，600，815}， 通过函数H(key) = key/100-1，将以上序列映射到{0，1，2，4，5，7}下标分量中。哈希表的大小可以取m=8，负载因子为∂=6/8=0.75。**

1. **除留余数法：哈希函数为H(key) = key mod p。**

**通过对关键字除以p取余数计算出地址。如元素关键字序列{35，192，64，5，76，653}， 通过函数H(key) = key mod 7，将以上序列映射到{0，3，1，5，6，2}下标分量中。此例哈希表的大小可以取m=7，负载因子为∂=6/7=0.86。函数中p通常取大于元素个数n的最小的素数，因p大于n，能保障元素全部入表，p取素数是为了尽量减少规律性的空间浪费。如当p和数据元素的关键字都是5的倍数时，映射出来的地址就会是0、5、10、15、20等等，这样会有4/5的空间浪费。**

1. **数据分析法:**



1. **平方取中法:**

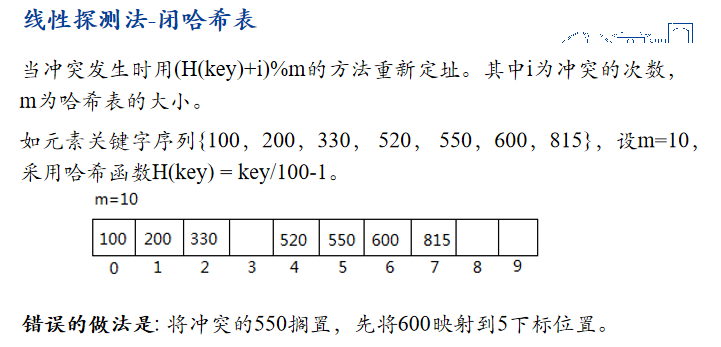
**如果关键字分布不均匀，也可以将关键字首先平方，有时平方后的结果中的几位就会变得均匀，这时再用直接定址或者数据分析法获得合适的哈希地址。如关键字136，平方后为18496，取其中的第2、3位得49，49便为关键字136的数据元素的哈希地址。**

1. **折叠法:**

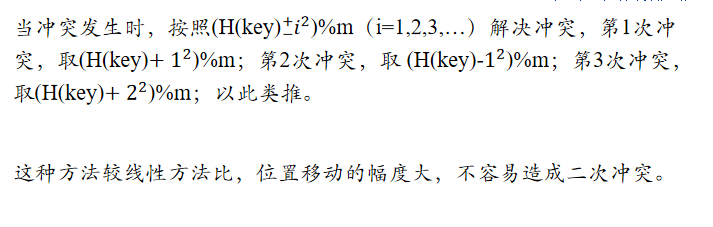
**当关键字位数相比于数据元素的个数大得多，也可以将其按照哈希表大小分割成若干段，并将这些段相加，得到的和作为哈希地址。**

1. **冲突解决**
2. **相关概念：**
3. **映射到同一地址上的不同关键字称同义词。**
4. **线性探测法、二次探测法：冲突时重新计算哈希地址，再计算出的地址仍然在原来的哈希表中，因此称为闭哈希表。**
5. **链地址法：如果冲突时在原来的哈希表外寻找一个存储地址，则称为开哈希表**

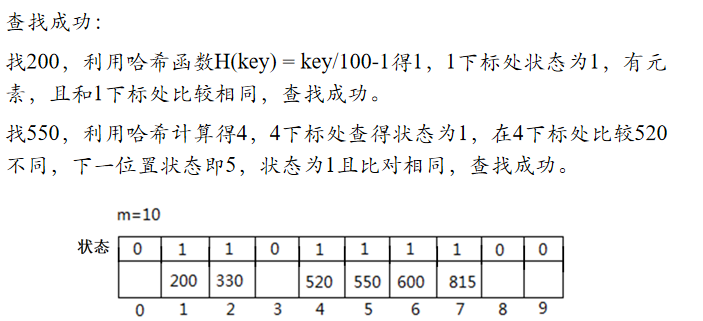
**B、线性探测法：**

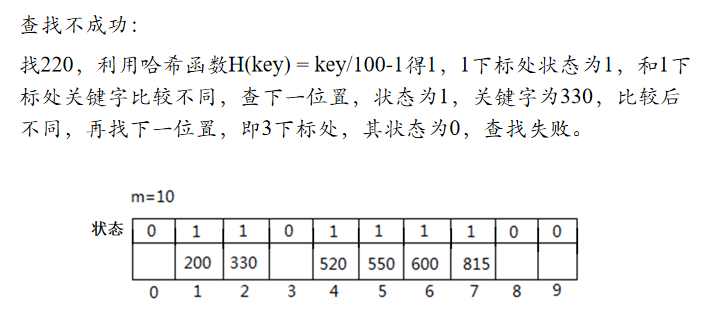


**C、二次探测法：**



1. **查找**

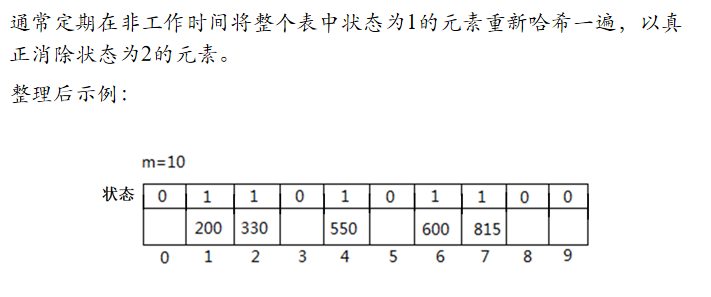


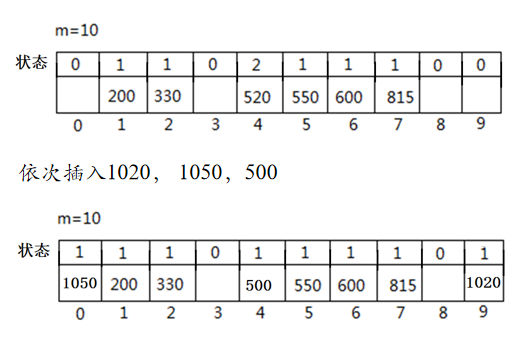


1. **删除**



1. **哈希表定期整理**

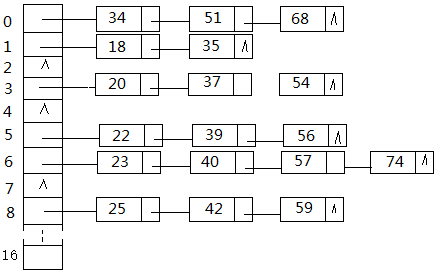


1. **插入**

**1050映射到9，下一个位置为0下标，可看出用了顺序循环方法。**

**500映射到了4，因4状态为2， 直接写入，改状态为1.**

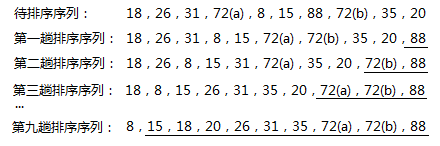
**D、链地址法-开哈希表：不在原来的哈希表中找地址，哈希表中根本就不存储实际的元素，而是存储一个单链表的首结点地址，元素存储在单链表中。**



1. **排序**

**内排序**

1. **冒泡排序：两两比较为基础；每一趟排序都有一个最值放在最端；直至前面剩下的元素个数为1个。**

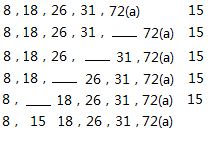


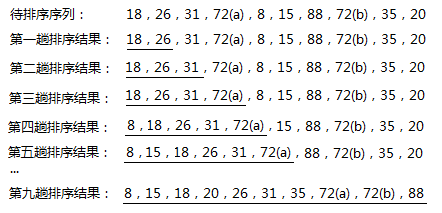
**稳定排序**

**O(n2)**

1. **插入排序**

**以两两比较为基础，将一个元素插入到有序序列**





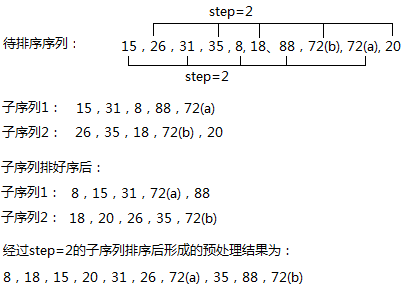
**稳定**

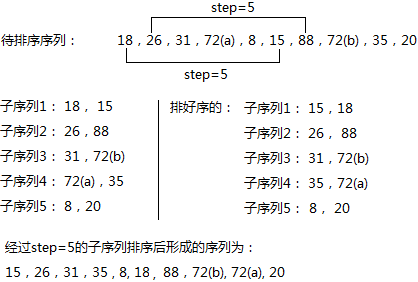
**O（n2）**

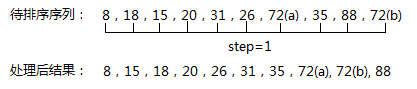
1. **希尔排序**

**预处理，使序列比较有序，然后进行插入排序**

**预处理方法：**

**将原始序列按不同步长分成若干子序列，分别进行插入排序，使序列变比较有序，降低插入排序时间消耗。**





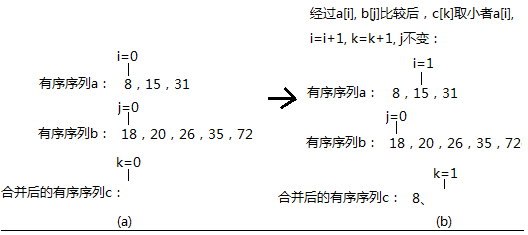
**一般来说，step的取值从n/2开始，之后逐次减半直到1，step将会取到log2n个值。**

**不稳定**

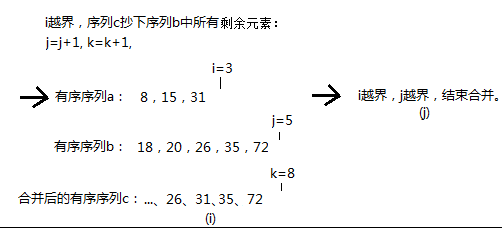
**O（n1.3）**

1. **归并排序**

**基于将两个有序序列归并为一个有序序列的方法。**



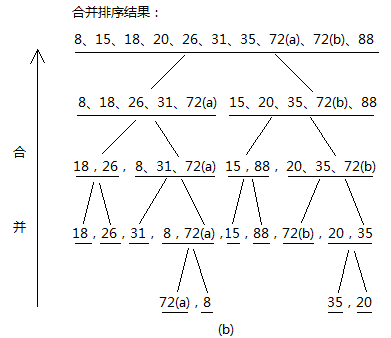


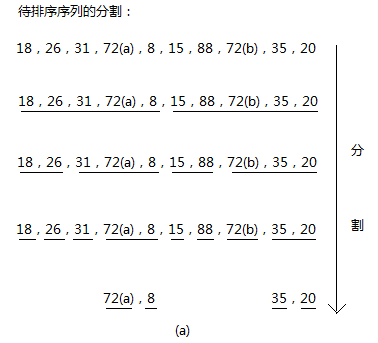


**归并排序的思想：**

**基于将两个有序序列归并为一个有序序列的方法。**

**原始序列中，每个元素可以看作是一个长度为1的有序序列，经过两两归并，形成多个长度为2的有序序列；**

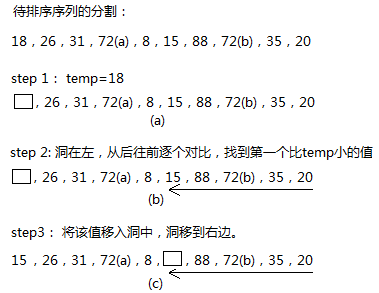
**再经过相邻两个有序序列归并，形成多个长度为4的有序序列；反复如此，最后形成一个长度为n的有序序列。**

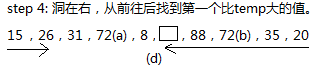
**稳定**

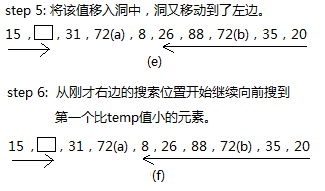
**O（nlog2n）**

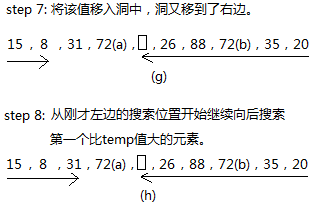
1. **快速排序**

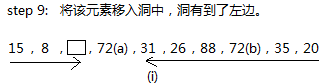
**快速排序的思想：选择一个元素作为标杆。所有小于它的元素移到它的前面，大于等于它的元素移到它的后面。对标杆前后两个子序列分别排序后，整个序列就有序了。**

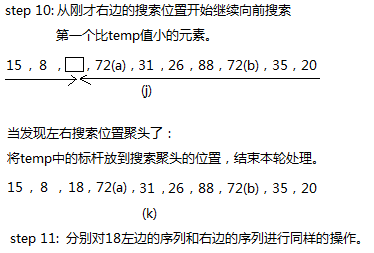




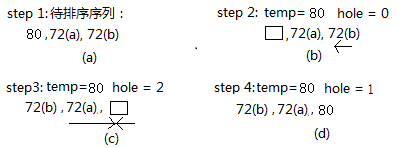








**不稳定**



**好：nlog2n 坏：n2**

**几种改进办法:**

**第一，取中间位置的值作为标杆；**

**第二，在首、尾、中间三个位置的值中找到中间值，将其作为标杆。**

**⭐无论标杆取什么位置上的元素，在排序前，需首先将该标杆元素换到首位置上，保证第一个洞一定在最左侧。当然，也可以将首洞位置定在最右侧，前面的处理就先从左向右搜索，而不是先从右向左搜索。**

1. **选择排序**

**选择排序的思想：以两两比较为基础**

**从左到右，为有序序列中每个位置选择合适的元素。**

**具体为：**

**在下标0-n-1范围找出最小值，换到0下标位置；**

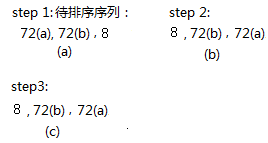
**在下标1-n-1范围找出最小值，换到1下标位置；**

**…**

**在下标n-2-n-1范围找出最小值，换到n-2下标位置.**

**不稳定**

**当找到最小值后，交换可能发生在不相邻元素之间，破环了原本的顺序，故是不稳定排序。**

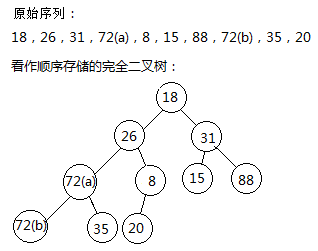


**O（n2）**

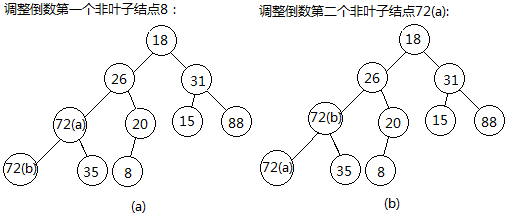
1. **堆排序**
2. **概念：**
3. **一个完全二叉树中，任意一个结点的值比其左右子结点值都大，称为大顶堆。**
4. **一个完全二叉树中，任意一个结点的值比其左右子结点值都小，称为小顶堆。**
5. **大顶堆和小顶堆都称为堆。**
6. **堆排序思想：**

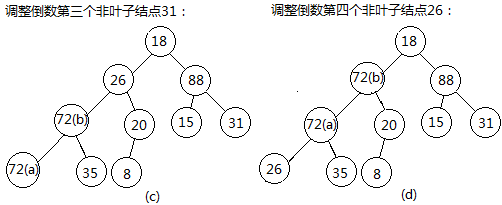
**将存于数组中的序列看作是一棵完全二叉树的顺序存储。**

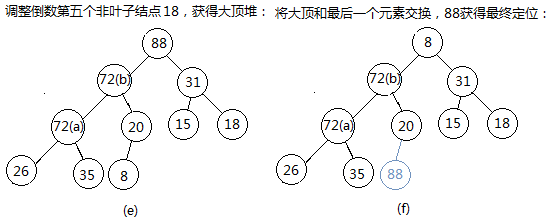
**按照堆的概念调整之，使之成为一个大顶堆。摘取大顶，换到待处理元素最后位置。继续调整新的根使之满足大顶堆概念，得到次大元素，继续后移，直到序列中元素全部有序。**

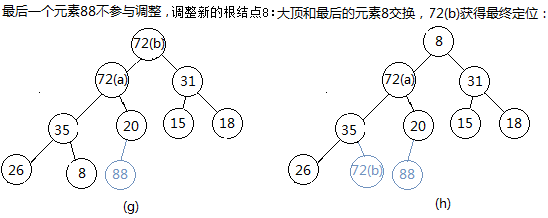


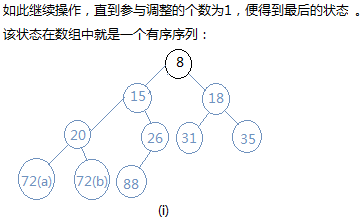
**对序列从后往前逐一（从后往前第一个非叶子结点）做元素检查、调整使得以该元素为根的二叉树满足大顶堆的定义。**











**数组中元素为：8、15、18、20、26、31、35、72(a)、72(b)、88， 有序！**

**不稳定排序**

**堆排序时间消耗由两部分组成：**

**初次建堆的时间消耗和；n-1次取大顶的时间消耗。**

**前者从形式上看时间复杂度是O(nlog2n),但实际可达O(n)。后者时间复杂度也是O(nlog2n)。**

1. **各种内部排序方法的比较：**

**冒泡排序、插入排序、选择排序、快速排序的最坏情况下时间复杂度都是O(n^2)，特别地在数据原本正序的基础上冒泡排序和插入排序时间复杂度可以达到O(n)。基数排序在一般情况下时间复杂度就能达到O(n)。归并排序、堆排序、快速排序最好情况时间复杂度能达到O(nlog2n)。**

**冒泡排序、插入排序、归并排序、快速排序都是稳定排序。希尔排序、选择排序、堆排序都是不稳定排序。**

**外排序**

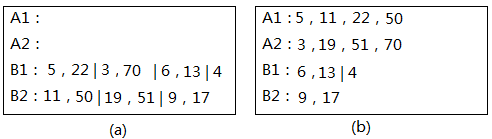
1. **概念：根据内存容量的大小一次调入一定量的数据，形成一个数据序列，该序列在内存中可以按照某种内排序的方法进行排序，然后将排好的序列写入外存，之后再调入其他未排序的数据进入，以此类推。最终在外存上原始的待排序序列分割成了多个有序序列，之后再设法将数据分段逐步调入内存，进行有序数据段的归并。**
2. **数据规模为900M，内存一次只能存储100M，由此可以把外存上的数据分为9段，每段读入内存排好序并再次放入外存后，后面就要进行归并。**

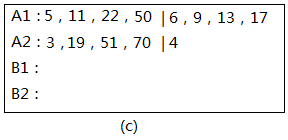
**归并：现在在内存中开辟10个10M的缓冲区，其中9个用于读入9个归并段的部分数据，作为输入缓冲区，1个10M用于存储归并结果，即作为输出缓冲区。**

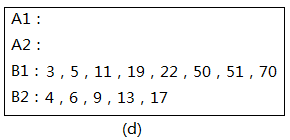
**假如9个归并段都是非递减有序的，在归并中，最小的元素必然出现在9个归并段的第一个元素中，输出该最小元素到输出缓冲区，以此类推，找到次小元素，归并过程中，如果输出缓冲区满，可将数据移出内存，清空缓冲区，继续存储后面的归并数据，最终在外存上形成了一个规模为900M的非递减有序序列。**

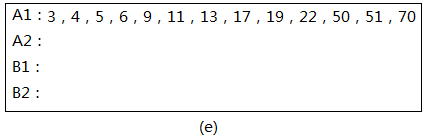
1. **K路排序（如果归并采用k路，除了可以用2k个磁带也可以仅用k+1个磁带来解决。）**

**最简单的归并是二路归并，二路归并是将两个有序序列归并为一个有序序列。在外排序中，二路归并需要4条磁带。最初无序序列在一条磁带上，如A1，分段入内存排序后放到B1、B2两条磁带上，后面逐次进行二路归并。**



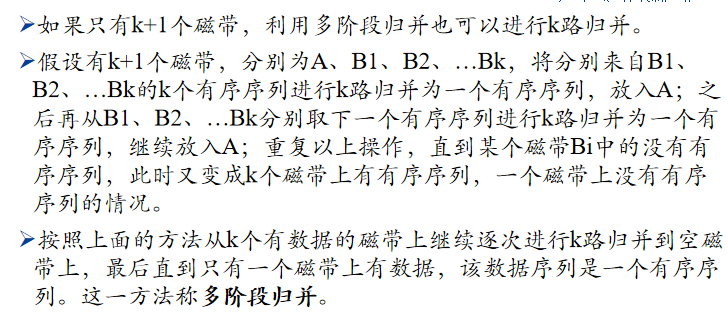




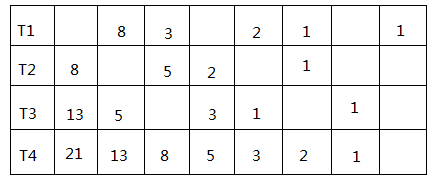


**如果有2K个磁带，就可以实现k路归并。**

1. **多阶段归并**



如果只有3+1个磁带，利用多阶段归并也可以进行3路归并。



上面的k+1磁带中，k条磁带上各放置多少初始归并段是最优的？

平均分配？差！

悬殊分配？差！

结论：

按照相邻的斐波那契数分配，不足的补空的初始规定段。

1. **置换选择**

**初始归并段优化：置换选择**

**对有m个初始归并段的k路归并，其时间消耗为logkm，m越小时间花费越小。**

**置换选择，通过拉长每个有序序列的长度来减少初始归并段的个数。**



