【线性数据结构】

全序性（全部结点两两可比先后）

单索性（每个结点有唯一的前驱和后继）

【存储密度】有用数据/该结构整个存储空间大小

【复杂度记号】

O(n)：小于等于（最坏情况）

Ω(n)：大于等于（最好情况）

Θ(n)：正好等于（所有情况）

【栈】

1. 如何判断出栈顺序是否合法？

Eg. 1423不合法，因为3作为最后一个出栈的元素将序列分为了两半，入栈先于3的元素全部在3入栈前出栈，即出栈早于入栈后于3的全部元素。

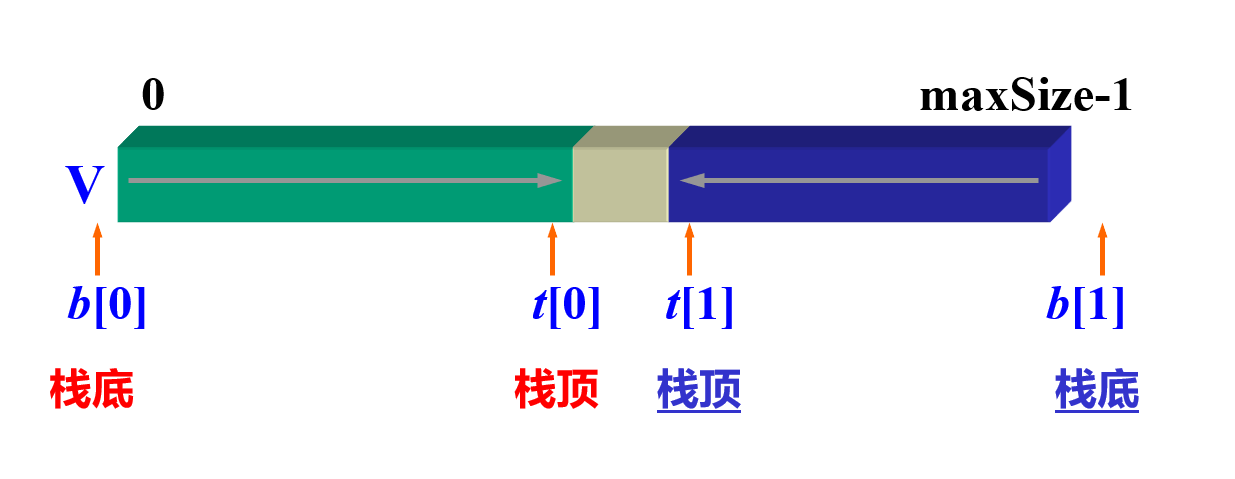
2. 长度为n的序列有多少种出栈顺序？

卡特兰数。

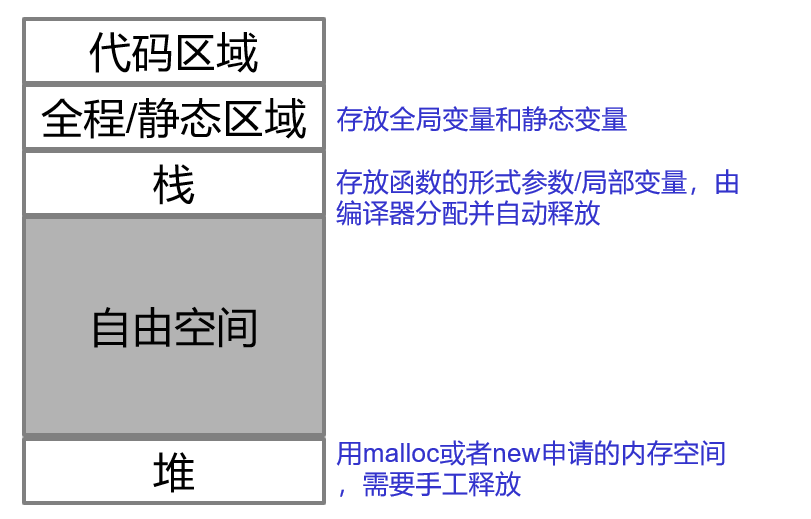
设长度为x的有f(x)种出栈顺序，则f(n) = f(n-1) + f(1)\*f(n-2) + f(2)\*f(n-3) + …

数学上推导等于1/(n+1) \* C2n(n)，也等于C2n(n) – C2n(n-1)

3. 双栈共享栈空间，用于两栈元素总数稳定时。

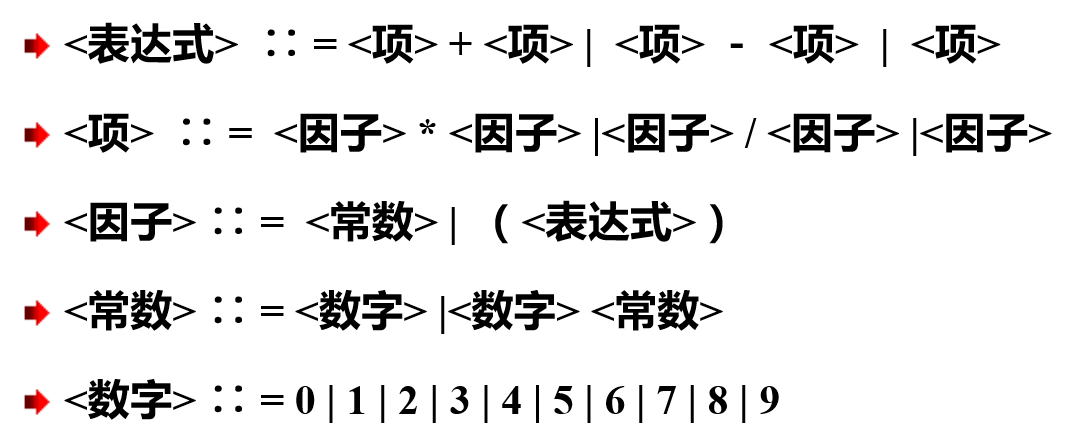


4. 栈的应用场景

动态存储分配

表达式求值

中缀表达式的递归定义（十分复杂）



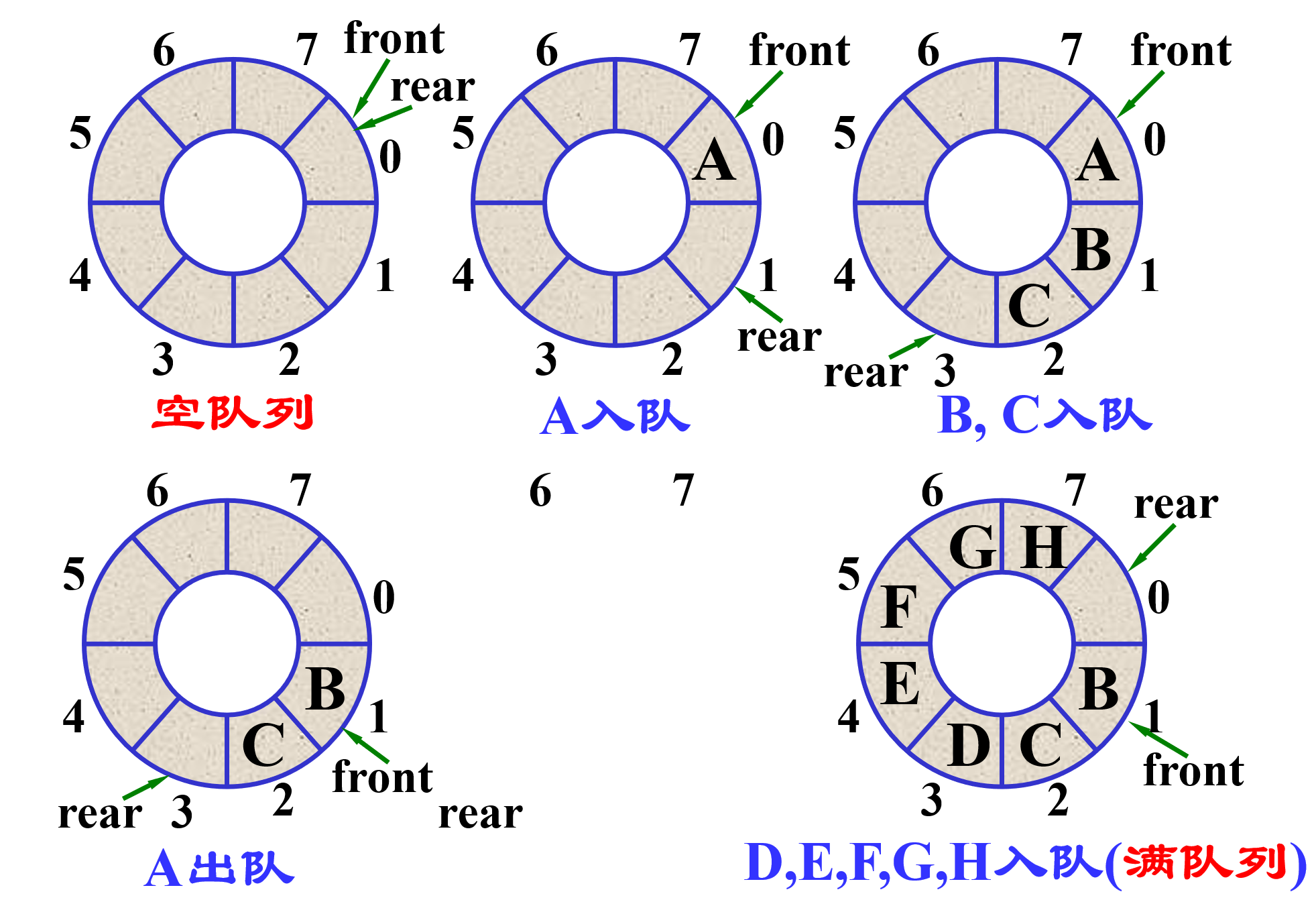
所以一般采用后缀表达式（逆波兰式）。

\***中缀转后缀**：操作数直接写，左括号直接压栈，右括号弹出栈顶元素直 到左括号为止，运算符与栈顶运算符优先级相比较、弹出栈顶元素直 到运算符优先级大于栈顶运算符后将当前运算符压栈。

【队列】

1. 假溢出：再插入时溢出了，但其实前面有空的位置。

解决方法：变为循环队列（通过取模实现）。

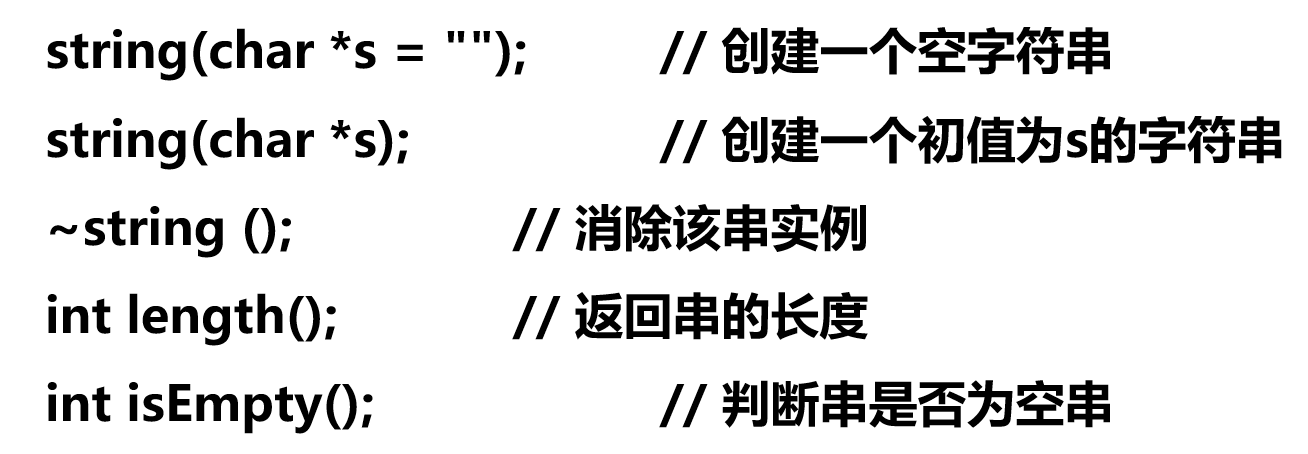


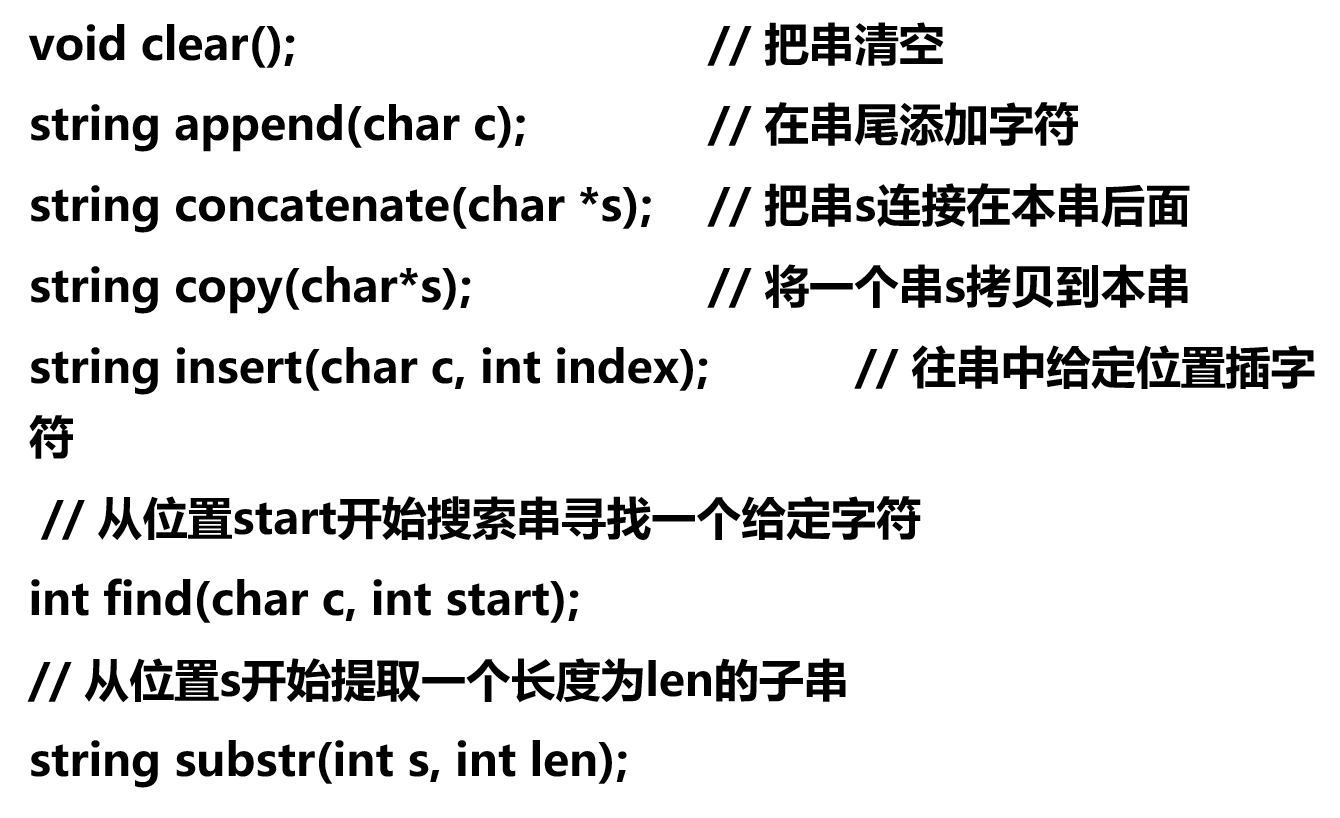
\*浪费一个空间使得判断满/空/队内元素个数更加容易

【字符串】

1. 一些常用函数

·String版本







·char s[]版本



2. 字符串匹配

·近似匹配：看需要更改（改/增/删）多少个字符，用dp实现

·精确匹配：kmp算法

kmp算法复杂度计算：

j只增不减，所以最多执行LEN次。i只有i=next[i]会动，假设i=next[i]执行y次、循环结束时i=x、x属于[-1, len)。那么i最多加了LEN（和j一起加的），最少减了y（假设一次next只往后退一格），那么LEN-y≥x。即y≤LEN-x，即y≤LEN+1。所以最多执行2\*LEN+1次。所以为O(n)。

同理可得计算next时复杂度为O(m)。所以总复杂度为O(n+m)。

【二叉树】

1. N个节点的二叉树有几种形态？

卡特兰数。（拎一个头起来分为左右两边）

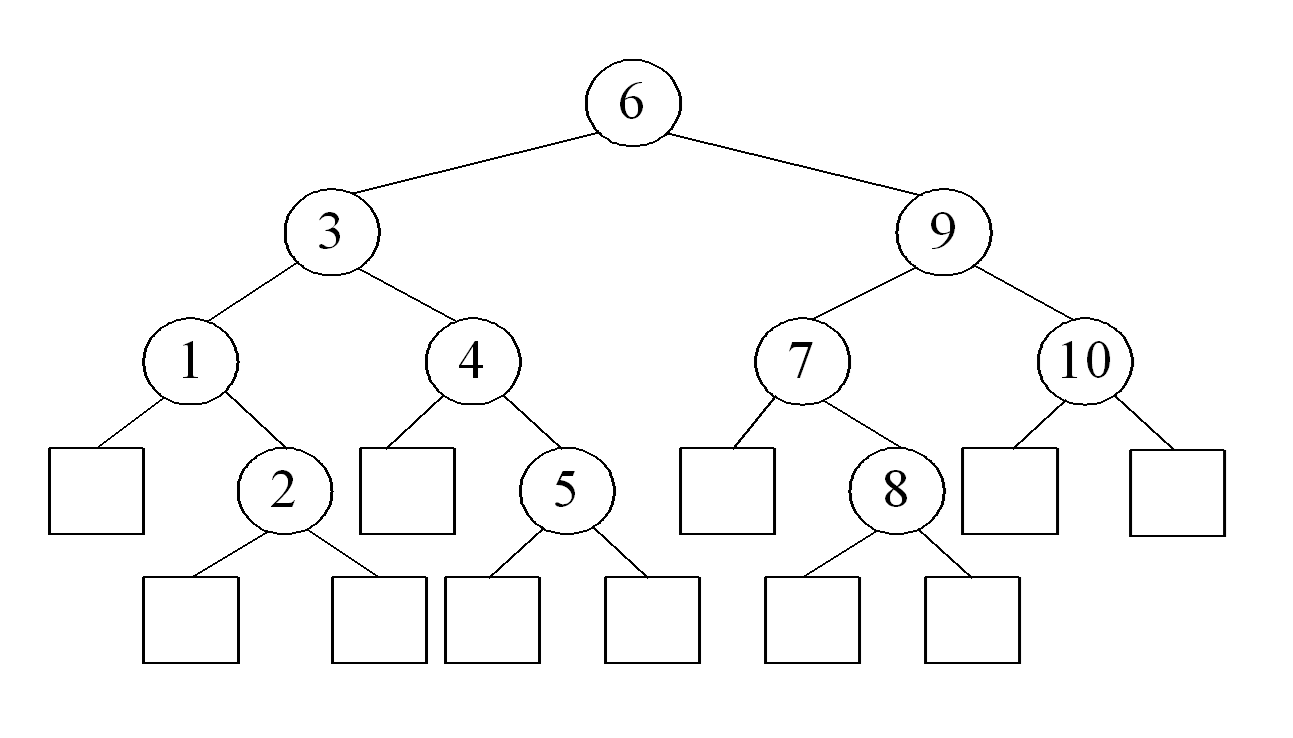
2. 完全二叉树：摆不满的向左对齐。

\*特性：路径长度和最短（由根节点到各个节点的路径长度总和最短）

3. 满二叉树：所有有儿子的都有两个儿子。

\*特性：叶子结点数 = 分支结点数+1

4. 扩充二叉树：全部补一层叶子。



\*特性：扩充结点数 = 原结点数+1 （可由数学归纳法/满二叉树性质证明）

外部路径长度E = 内部路径长度I + 2n （可由数学归纳法证明）

E: 根到所有外部节点路径的长度

I: 根到所有内部节点路径的长度

N: 内部节点的个数

5. 二叉树的性质

-任何一颗二叉树，度为0的结点比度为2的结点多1个 （可由方程解出）（可由此证明满二叉树性质）

-二叉树高度定义为二叉树中层数最大的叶子结点层数（即深度）加1

6. 二叉树的周游

-非递归周游：手动入栈

前序：访问自己后右孩子入栈，访问左孩子；

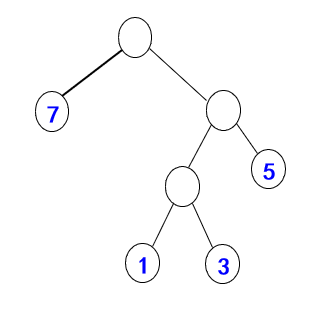
中序：自己入栈后访问左孩子，出栈后访问右孩子；

后序：将自己分为L和R两次入栈。

7. 二叉搜索树（BST）：中序遍历有序。

BST的删除：若删除节点无左子树，则右孩子代替该结点；若删除结点有左子树，则左子树中最右下的结点提上代替被删除节点，最右下结点的左孩子代替该结点。

8. Huffman树（最优二叉树）：具有最小带权路径长度的二叉树。



-动机：希望构建不定长编码（which要求任何字符编码不是另一个的前缀），根据字符频率来编码，从而减少总编码长度。

-建Huffman树：取权重最小的两个，用结点联结起来，将新建的节点权重记为俩节点权重之和，放入原数据中，递归地操作。（用堆实现）

-证明最优：非常复杂，考虑数学归纳法。

【堆】

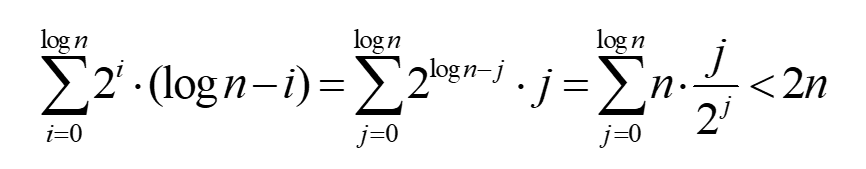
1. 最小堆（/最大堆）：父亲永远小于儿子，兄弟无约束。用于解决取最小值的问 题，是一个可用数组表示的完全二叉树。

2. 建堆：先随意放，然后从第一个非叶子节点开始向下调整，即和两个儿子中 较小的那个换，直至比两个儿子都大。（这样最下面i层就会递归地满足最小 堆）。

3. 插入：放在最后一位，然后向上调整，即比父亲小就交换。

4. 移除最小值（根节点）：和最后一个节点交换之后根节点向下调整。

5. 移除特定值：和最后一个结点交换之后向上调整、向下调整。（两个都写着但 是其实只会执行一个）

6. 建堆复杂度：

其他操作复杂度均为O(logn)

7. 堆是优先队列的一种自然实现方法。

【树】

1. 有序树：儿子有序的区别，大儿子删除后二儿子成为大儿子。

//与二叉树不同。二叉树：严格区分左儿子与右儿子

2. 森林：引入一个空根即可表示为一棵树。

3. 森林和二叉树一一对应。

对应方法：建立空根后，对于森林的每一个节点，只存其大儿子和右兄弟。

4. 周游：先根周游、后根周游。

先根周游对应相应二叉树前序周游，后根周游对应相应二叉树中序周游。 因为： 树先根 二叉前序 （后根---中序周游同理）

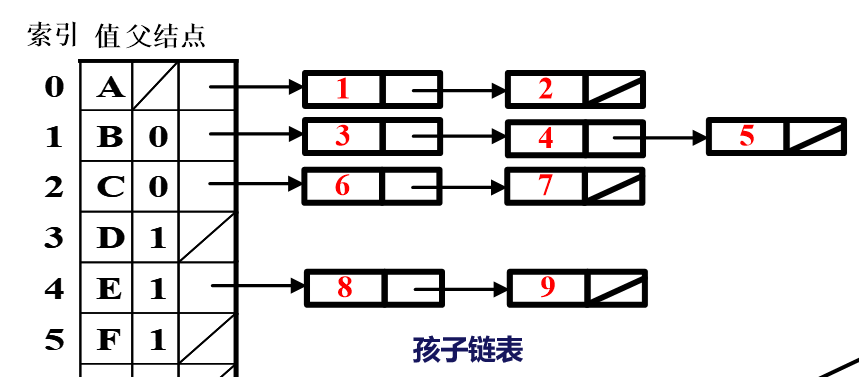
先访问n1 --- 先访问n1

再n1子树 --- 再n1左儿子

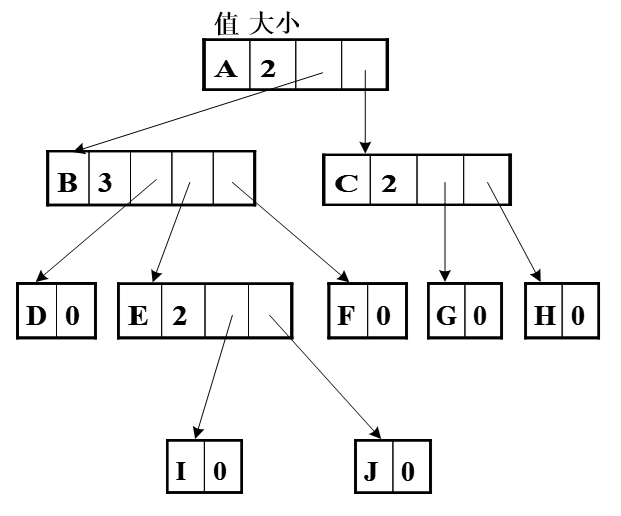
再n1右兄弟 --- 再n1右儿子

5. 存储方法：

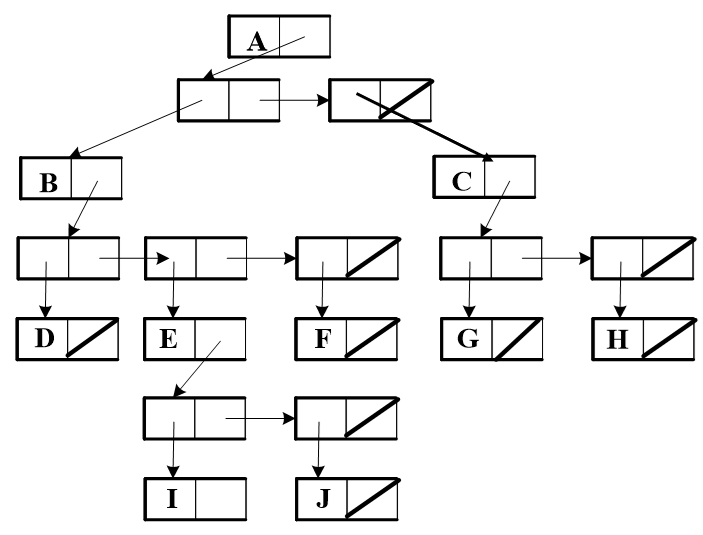
-链表：数组存索引，链表存儿子，相当于以图的方式存树。找兄弟不方便。



-指针数组法：完全用指针还原树。但不知道每个结点应该开多大的内存来存儿子的地址。

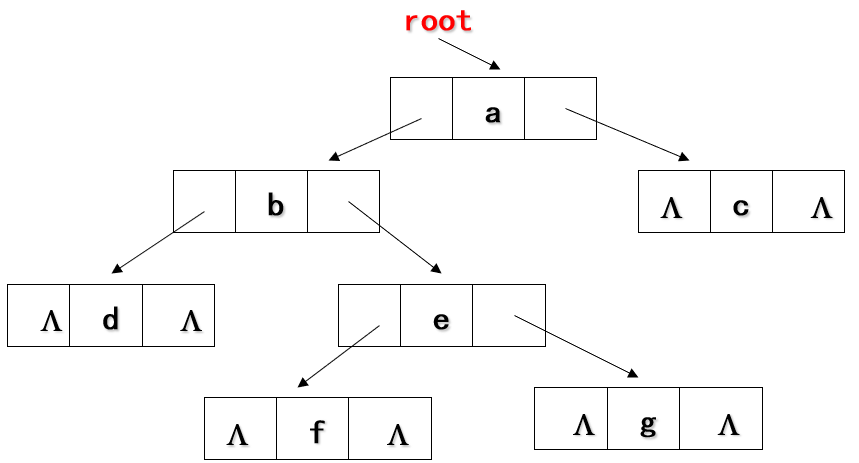


-指针链表法：omg这好蠢



-左孩子右兄弟数组表示法：空间效率高。

-左孩子右兄弟二叉链表表示法：（竟然这个是最常用的6.）

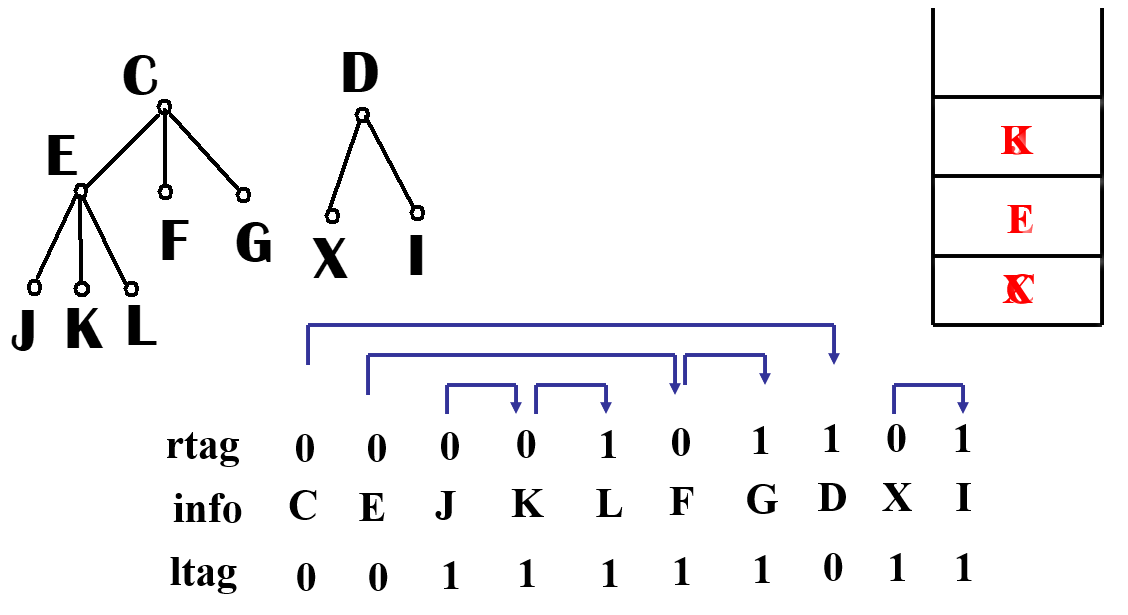


-顺序存储方式：按照树遍历的次序进行顺序存储，重点是如何还原。

带右链的先根次序表示法：bool ltag =是否有子节点，int\* rlink = \*rson.

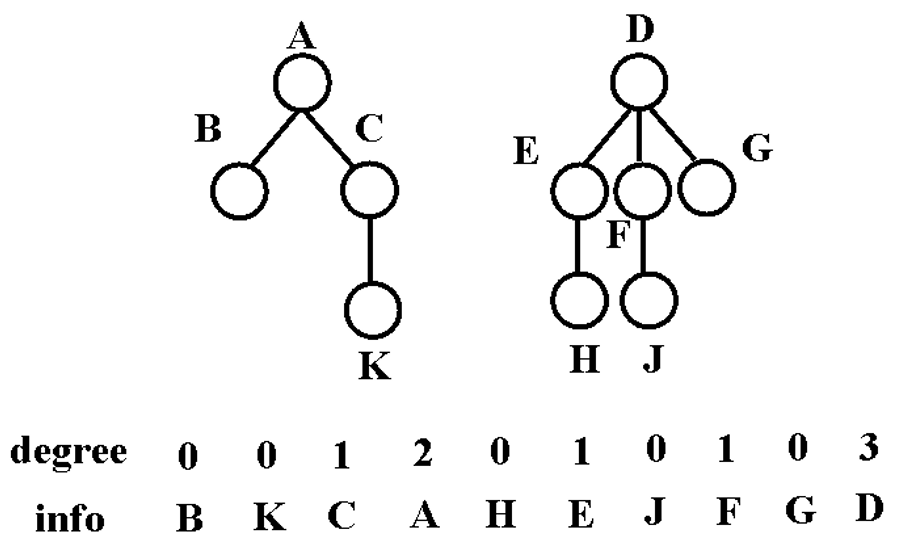
因为如果有儿子，必然是后一个。但显然rlink还可以优化。所以↓

带双标记位的先根次序表示法：bool ltag =是否有子节点，bool rtag = 是否有右兄弟。之后使用栈来还原，有兄弟就入栈，无孩子就出栈。



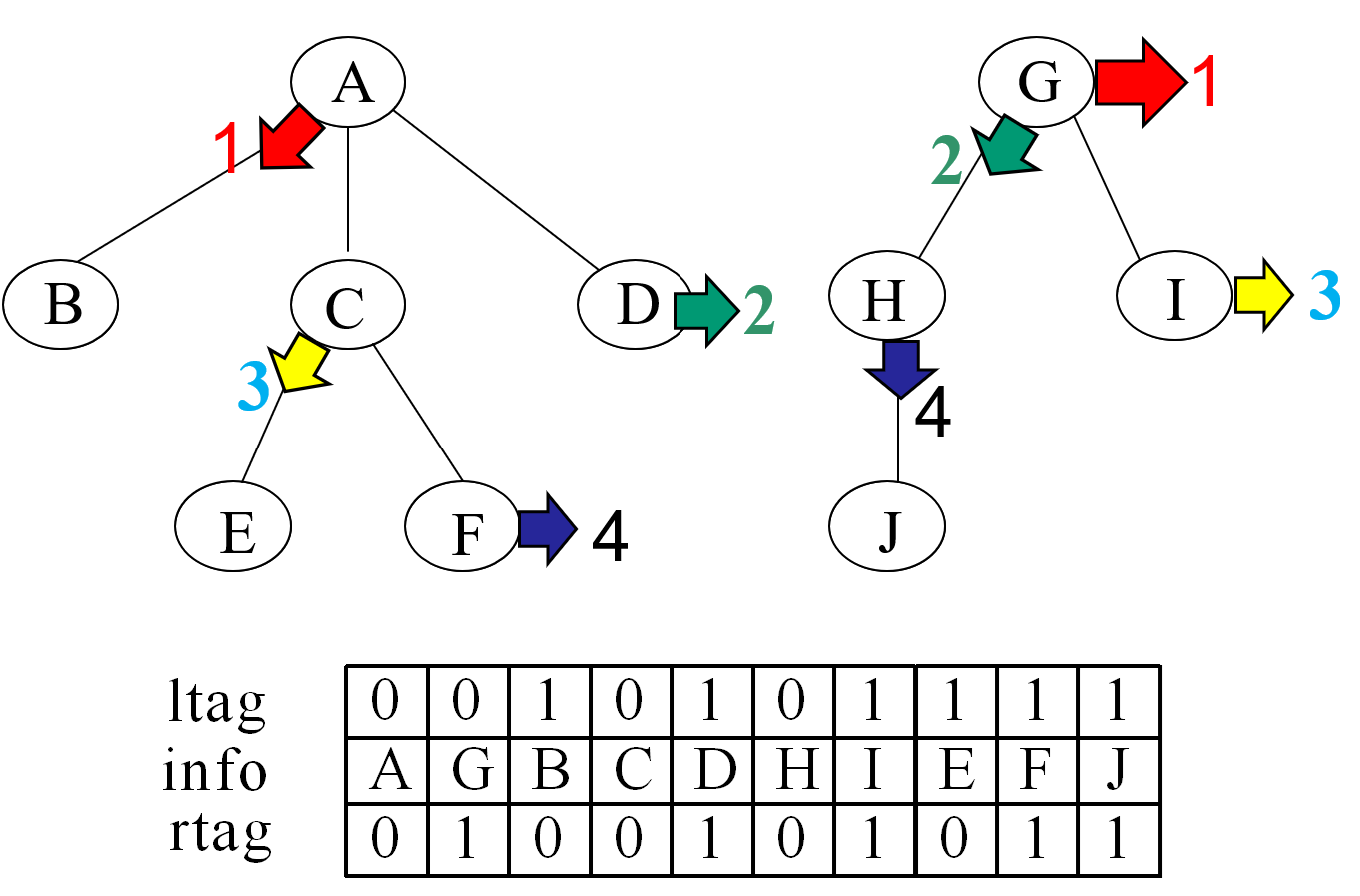
带度数的后根次序表示法

结点按后根次序顺序存储在一片连续的存储单元中，存储结点编号以及度数。还原：遇到零度顶点就入栈；遇到非零k度顶点就从栈中弹出k个节点作为其子节点，然后将该非零顶点入栈；持续扫描，直至序列扫描完毕。



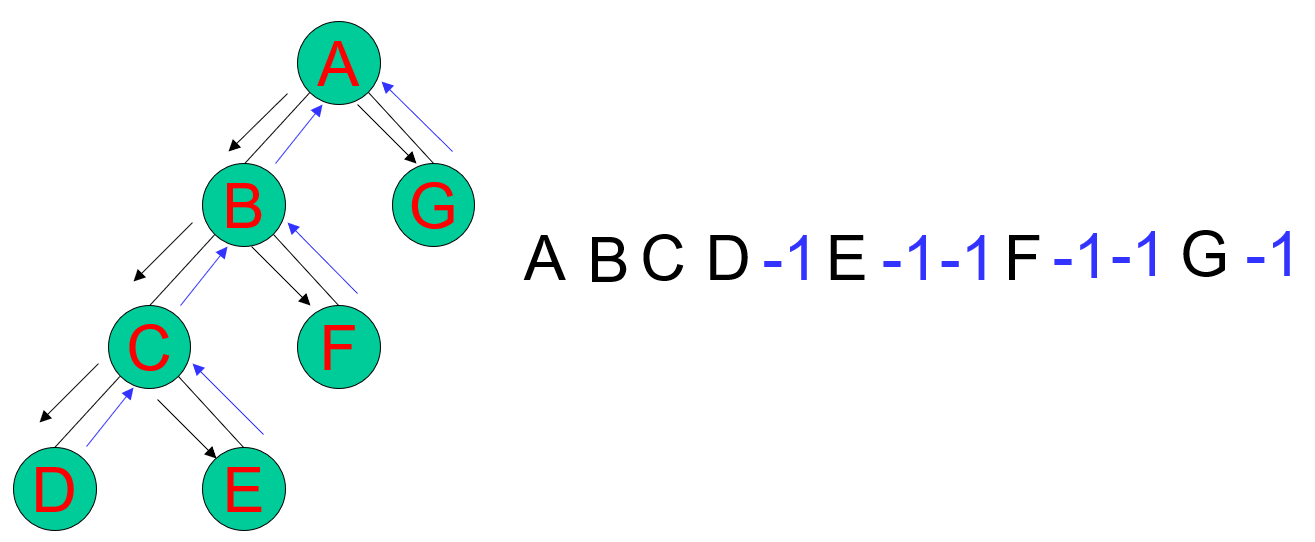
带双标记的层次次序表示法：

类似于带双标记的先根层次次序，ltag、rtag分别表示有无左孩子右 兄弟。如果有孩子就存入队列，接下来一个必定是孩子，如果孩子有 兄弟那接下来一个一定是右兄弟，如果孩子无兄弟就出队列。



前序字符串表示法：

按照先根排列依次写字母，如果回溯则添加“-1”。



6. 并查集：判断两元素是否处于同一个集合，采用父指针表示法。

优化：重量权衡合并原则：合并两棵树时，将矮的那棵接到高的那棵上。

路径压缩：在递归返回时把沿路的所有结点挂到最祖先的位置上。

优化后复杂度可以达到Find: O(1) Unison: O(1) Space: O(n)

7. k叉树

同样有满k叉树、完全k叉树等，性质可由二叉树推广。

【图】

1. 一些概念

团/完全图：有所有边，是最密集的图。

标号图：标了号的图。

带权图：带权的图。

简单图：无重边，无自环。

简单路径：除了起始结尾点可相同外其他结点不出现两次（即路径中无环）。

有根图：有向图中，若从一个顶点V0有路径可以到达其它所有顶点，则为 有根图，V0称为根。

强连通：有向图中如果任意节点到任意另一个节点都有路径可以到达，则为 强连通图。

强连通分量：强连通子图。

\*如果把所有强连通分量condense为一个点，则新的图没有环。

网络：带权的连通图。

2. 图的存储

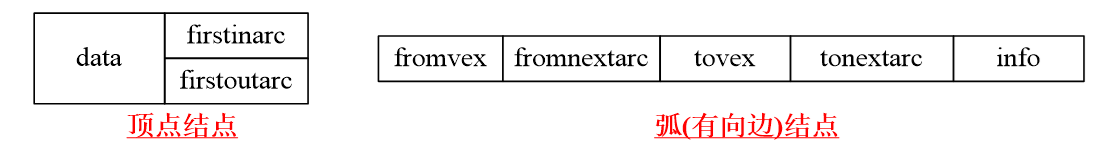
相邻矩阵：空间复杂度为O(n2), a[i][j]表示i与j是否相连。

\*加权矩阵：即把权重记录在a[i][j]上。

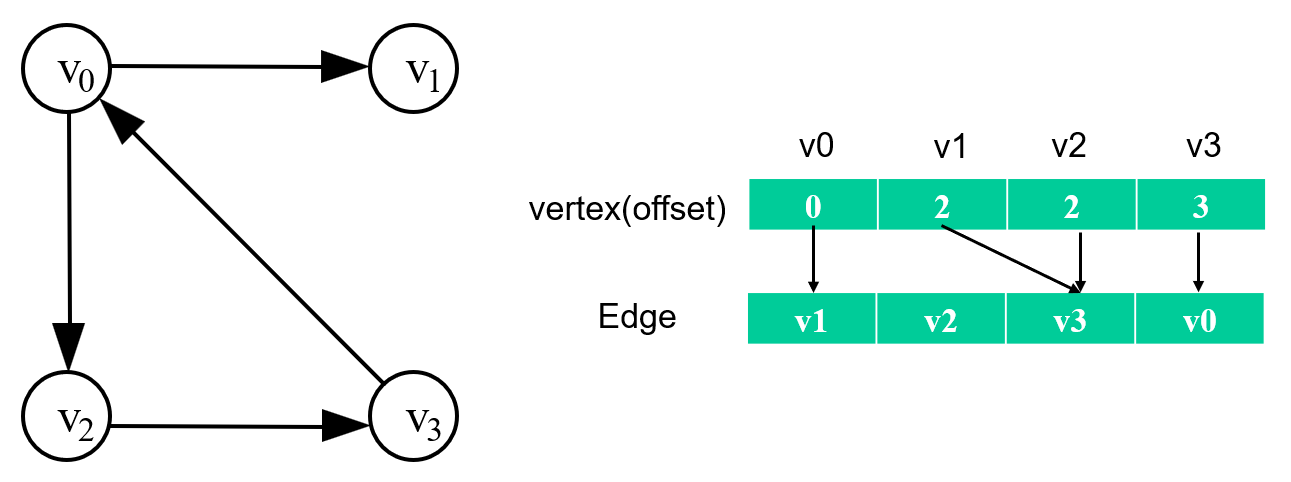
邻接表：顶点表+边链表。

\*有向图可保存出边表与入边表，但修改时两个表分开操作可能会有电脑 突发状况导致一个改了一个没改结果不匹配，因此需要[原子操作]！

十字链表：每个点连接出边表和入边表。



压缩稀疏矩阵(CSR)(compressed sparse representation)



\*仅用于顶点顺序存储的情况

存储两个数组，vertex数组的下标即顶点下标，存的是该顶点的出边从第几条边开始。edge数组按照顶点顺序存边(01, 02, 23, 30)，因为从vertex数组出发就完全知道第x条边的起始点是谁了，所以只用记录终止点。

空间存储复杂度为严格的n+e。

\*不适合增删。

3. dfs遍历

-基于dfs的边分类



4. 拓扑排序

针对有向无环图（eg.先修课），给出一个合理的拓扑排序（上课顺序）。

拓扑排序并不唯一。

-bfs法

建立入度表，挑选入度为0的点加入排序，更新入度表，直到没有入度为0的点了。此时如果还有点没有排序，则图有环。

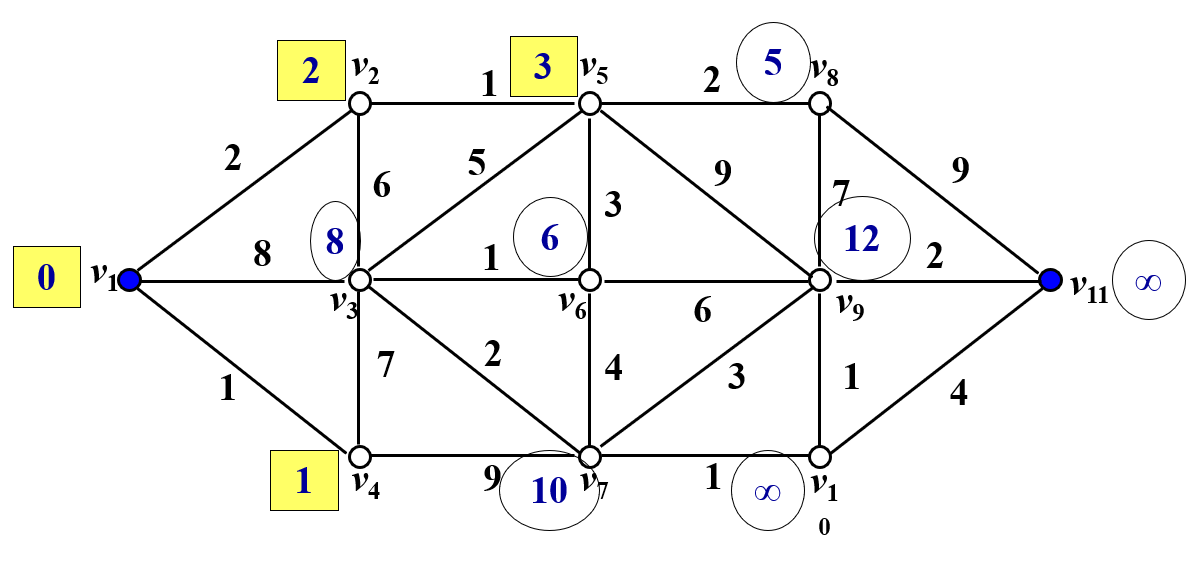
-dfs法

采用后根遍历，则可得到一个逆序的拓扑排序，再逆序输出即可。

缺陷：无法判断环。需要加入backward edge检测才能判断环。

5. 单源最短路径：Dijkstra算法（贪心）

从起点开始，把起点开始所有能到达的点的距离加入最小堆，取出最短距离的点，加入“能到达的点集合”，使用该点的距离更新其他点并将距离加入最小堆，重复操作直到所有点可达。



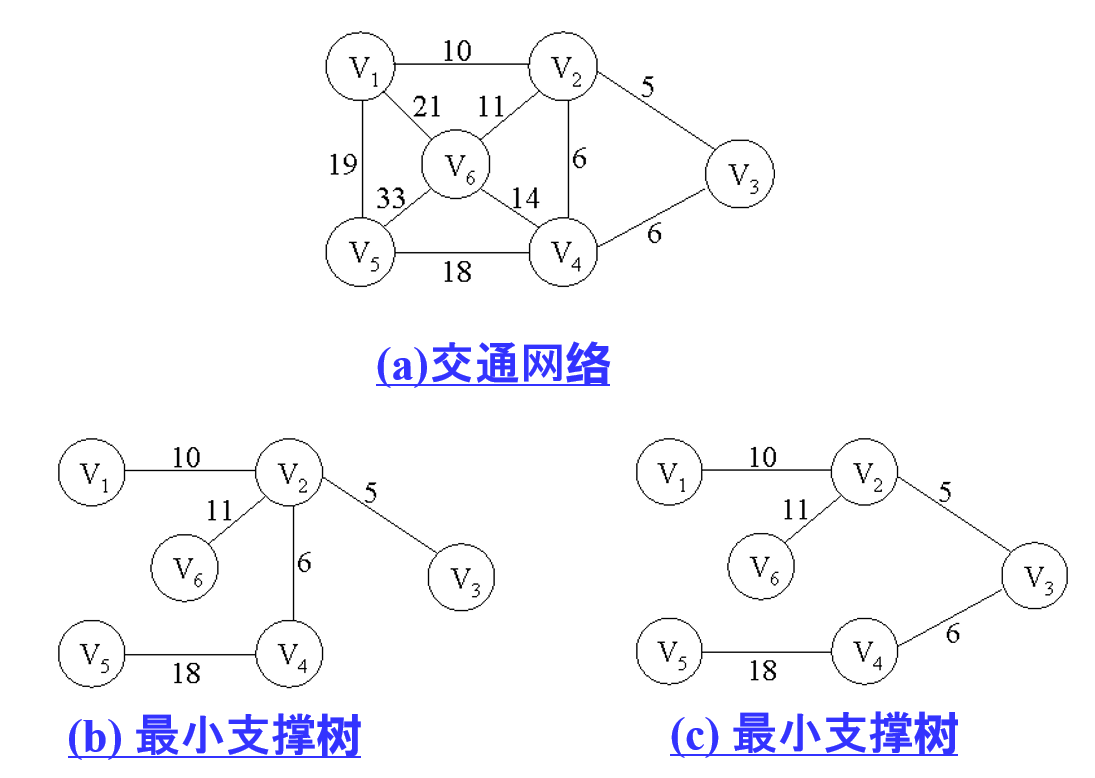
时间复杂度：nlogn

6. 每对顶点间最短路径：Floyd算法（动规）

d[i][j] = min( d[i][j], d[i][k]+d[k][j] ). 从小到大遍历k。

时间复杂度：n3。（适合稠密图）

7. 最小生成树/最小支撑树（MST）(minimal-cost spanning tree)



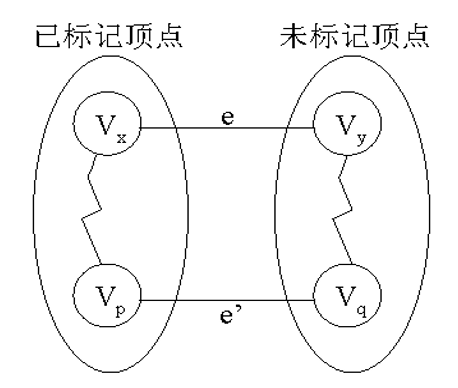
可见不唯一。

Prim算法

将随机一个点加入集合，找一个距离最近的集合外点加入集合，连边加入树，从该点出发更新未在集合内的点的距离，加入最小堆。重复。

复杂度：nlogn。适用于稠密图。

证明正确性：反证法。若当前最短边e不应该在MST中，则有另一条边e’要将已在集合内的点集和不在集合内的点集相连，而在使用e’构造的MST中加入e，删除e’，则仍为树且代价更小，与MST代价最小矛盾。



Kruskal算法

选择最短的边加入集合，如果已连通就不加（通过并查集判断），直到所有点连通。

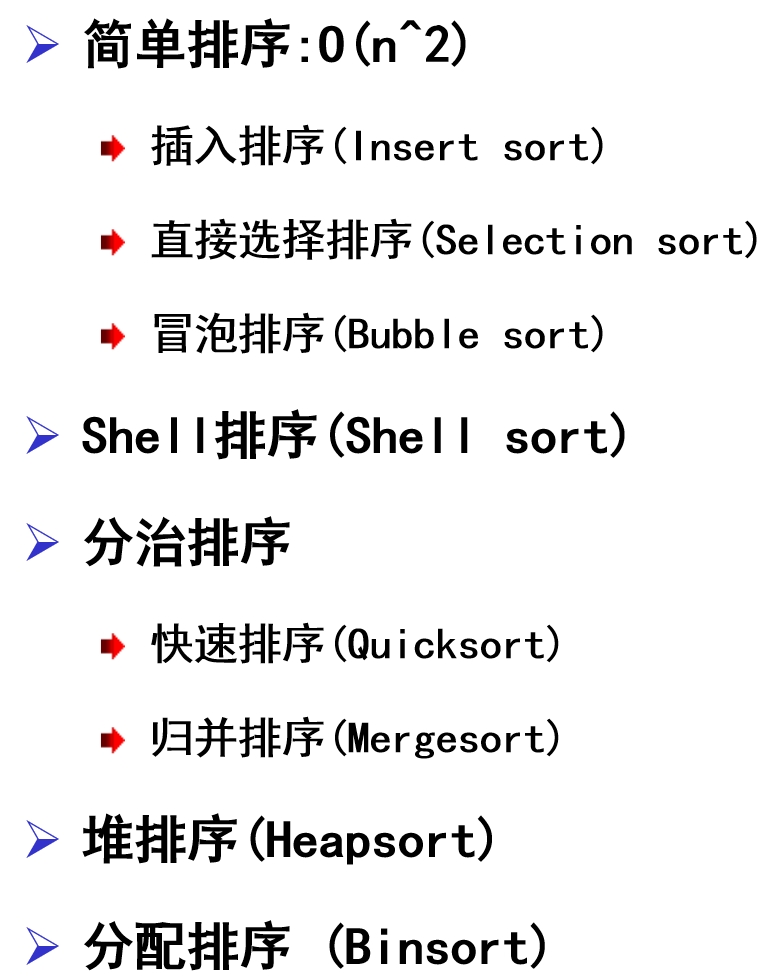
复杂度：eloge。适用于稀疏图。

【内排序】

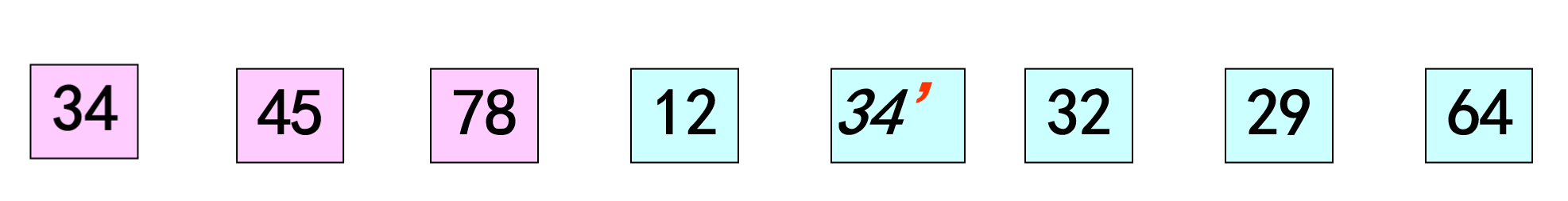
·内排序：待排序记录少，可以放在内存进行。

外排序：待排序记录多，内存放不下，排序过程中还需访问外存。

·分类



1. 插入排序



时间复杂度O(n2), 空间复杂度O(1)，稳定。

二分优化仍然是O(n2)，因为虽然比较次数降了，但移动次数不变。

2. 冒泡

时间复杂度O(n2), 空间复杂度O(1)，稳定。

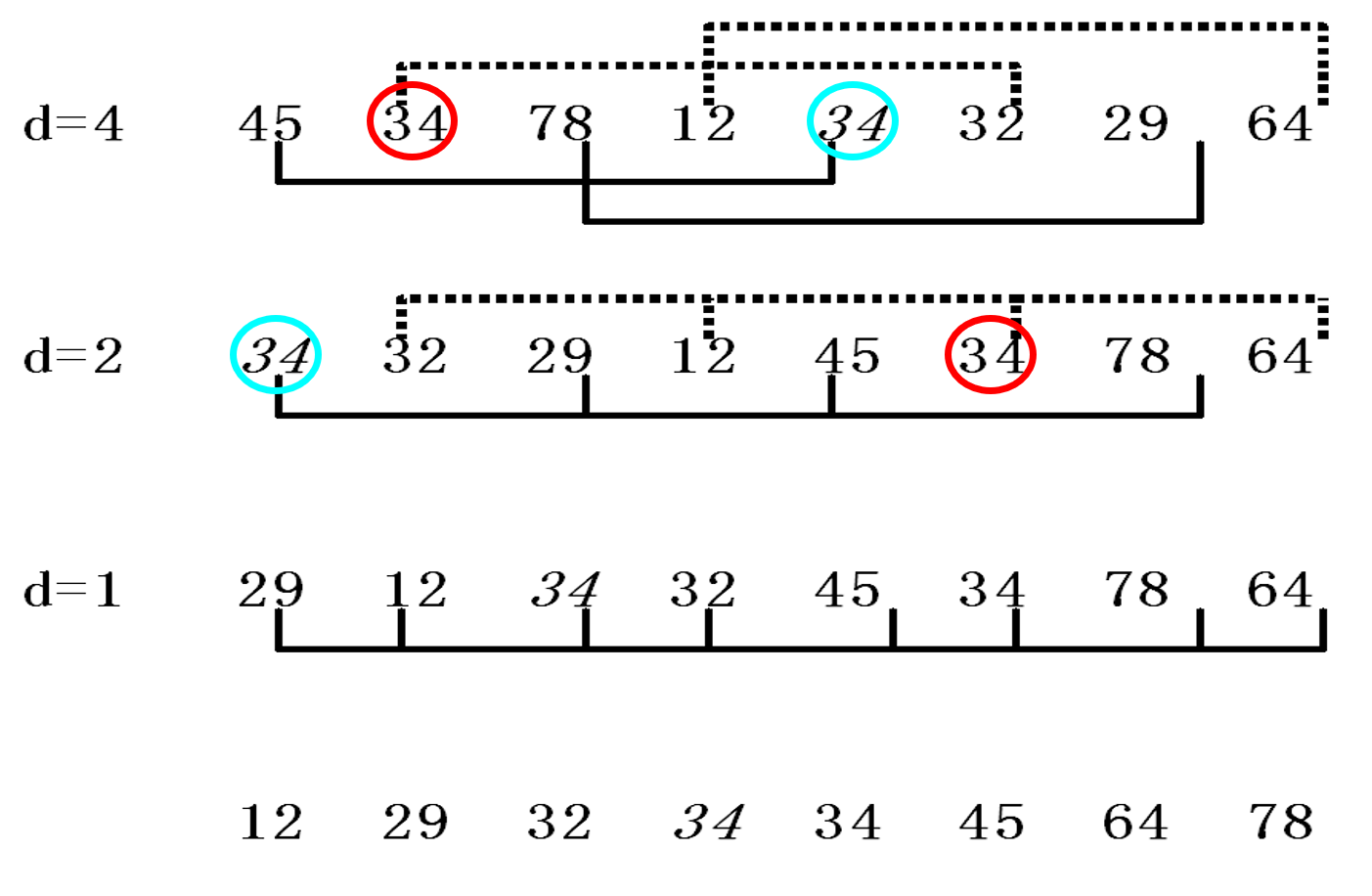
3. 直接选择排序

选择后直接交换。移动次数减少，比较次数仍多。

时间复杂度O(n2), 空间复杂度O(1)，不稳定。

4. Shell sort

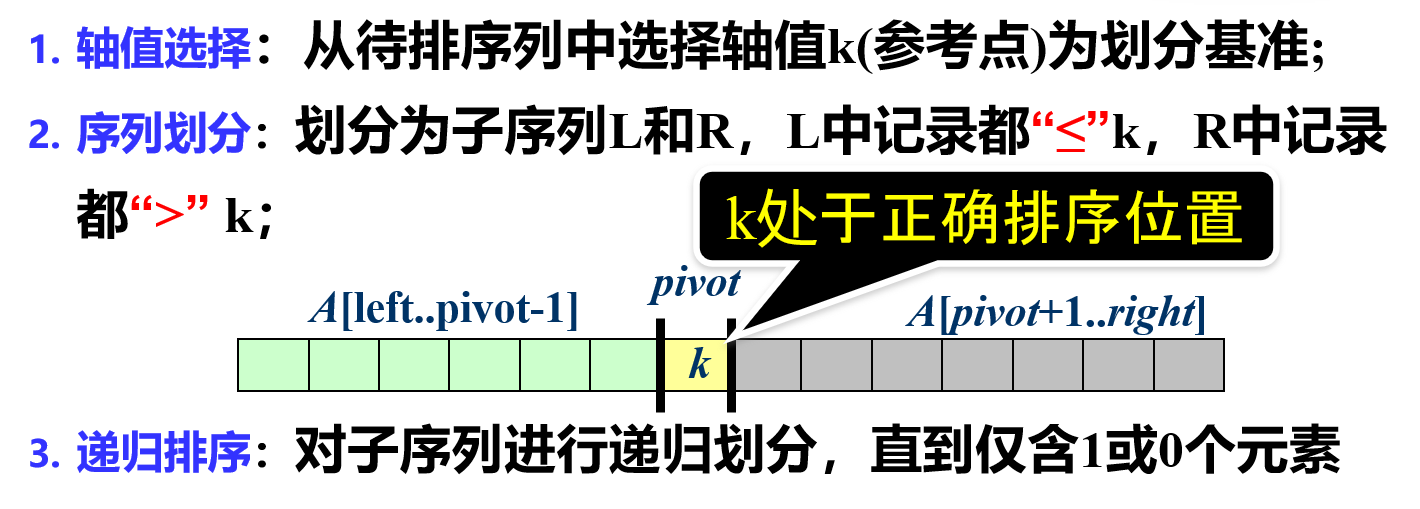
选一个递减的“间隔增量”序列，如下图，则第一次按照间隔为4比较，直接交换，第二次按照间隔为2比较，直接交换……直到间隔为1。任何正整数的递减序列d1, d2, …dt, 只要 d1<n, dt=1, 原则上都可作为希尔排序的增量序列。



\* Hibbard增量序列{2k -1，2k-1 -1，…，7，3，1}的Shell排序的效率可以达到O(n3/2)。

\*选取其他增量序列还可以更进一步减少时间代价。

5. 快排

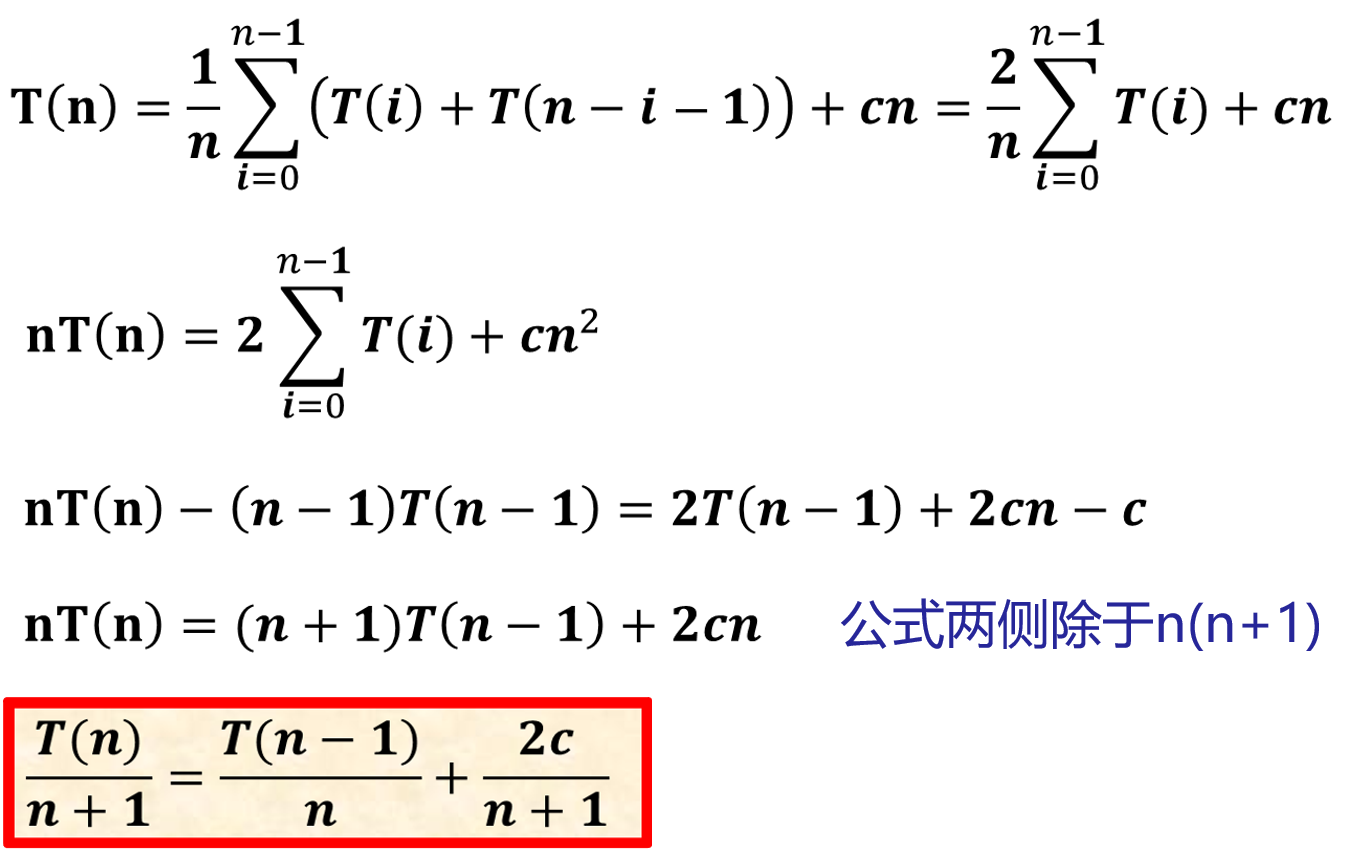


划分时可采用左右指针交替换，两指针重合位置即为轴值位置。

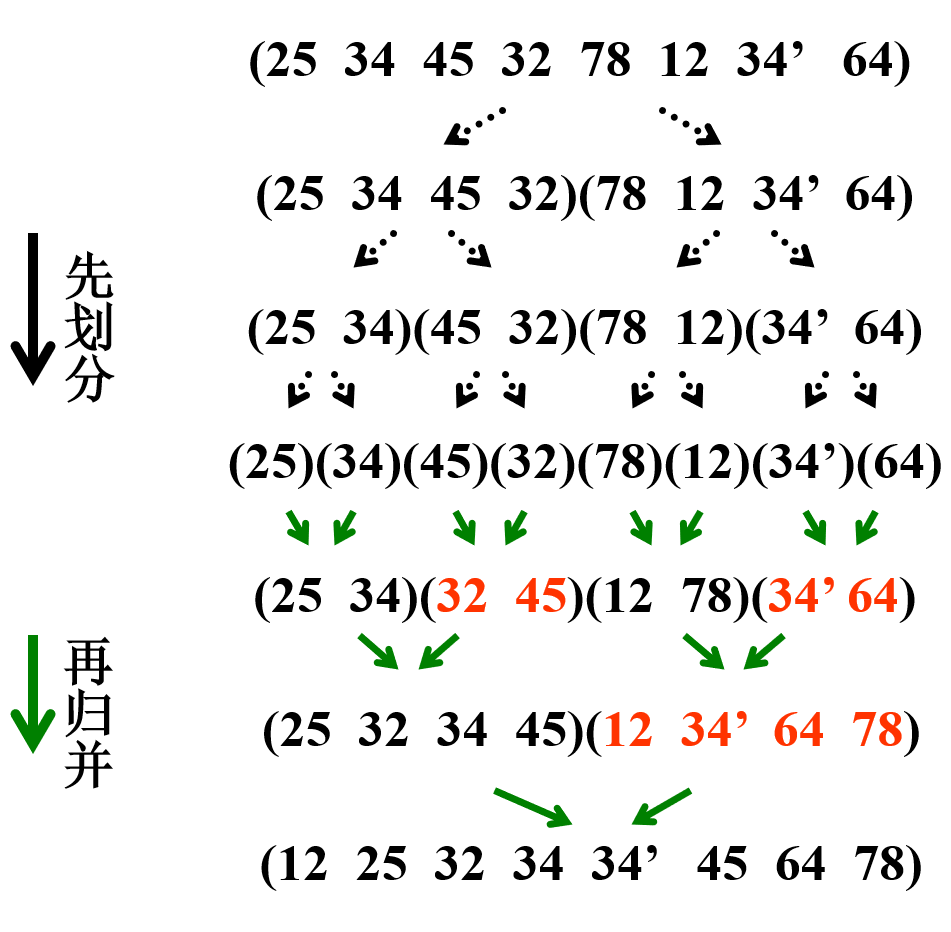
最佳情况O(nlogn)，最差情况O(n2)，平均情况O(nlogn)。

\*快排是内排序中最好的算法

平均复杂度计算↓



6. 归并排序



时间复杂度O(nlogn), 空间复杂度O(n)，稳定。

\*辅助存储量最多

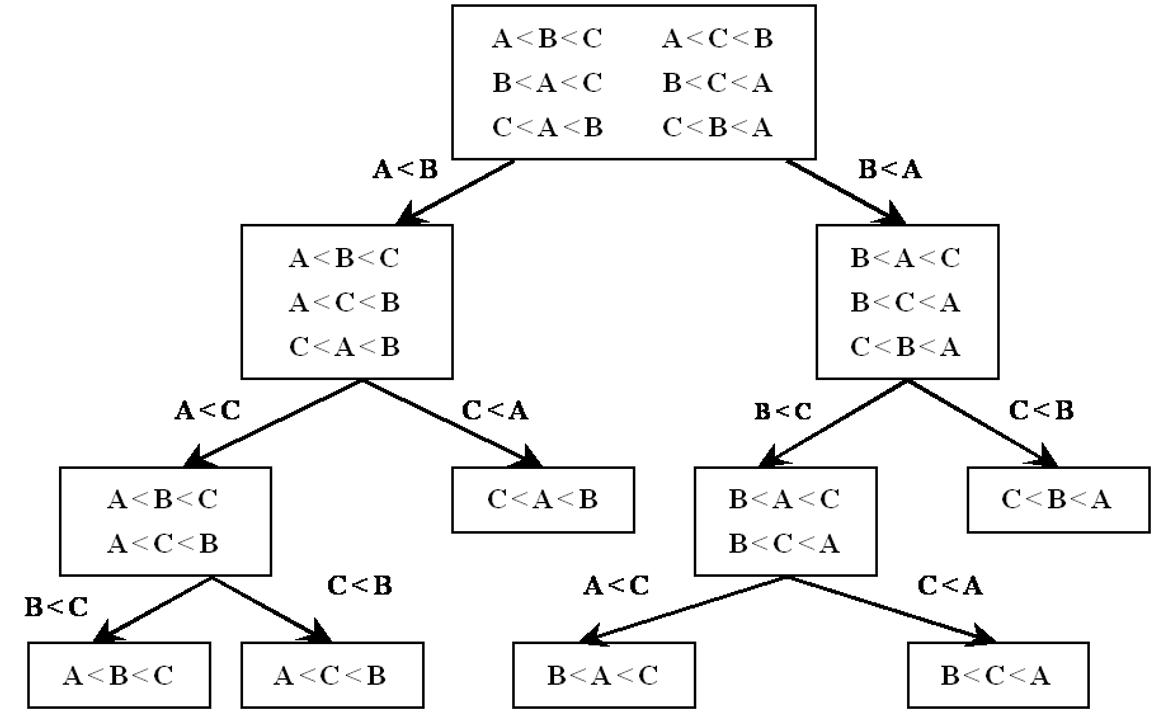
7. 堆排序

建堆，然后直接依次从堆中取出最大值/最小值排序。

时间复杂度O(nlogn)，空间复杂度O(1)。

\*以上是基于比较的算法。

基于比较的算法的理论下界（可能到达的最小复杂度）是nlogn。因为一共有n!种排序可能，即一棵二叉树有n!个叶子结点，则最小深度为log(n!)，化简后得到nlogn，即为可能达到的最小复杂度。



8. 分配排序

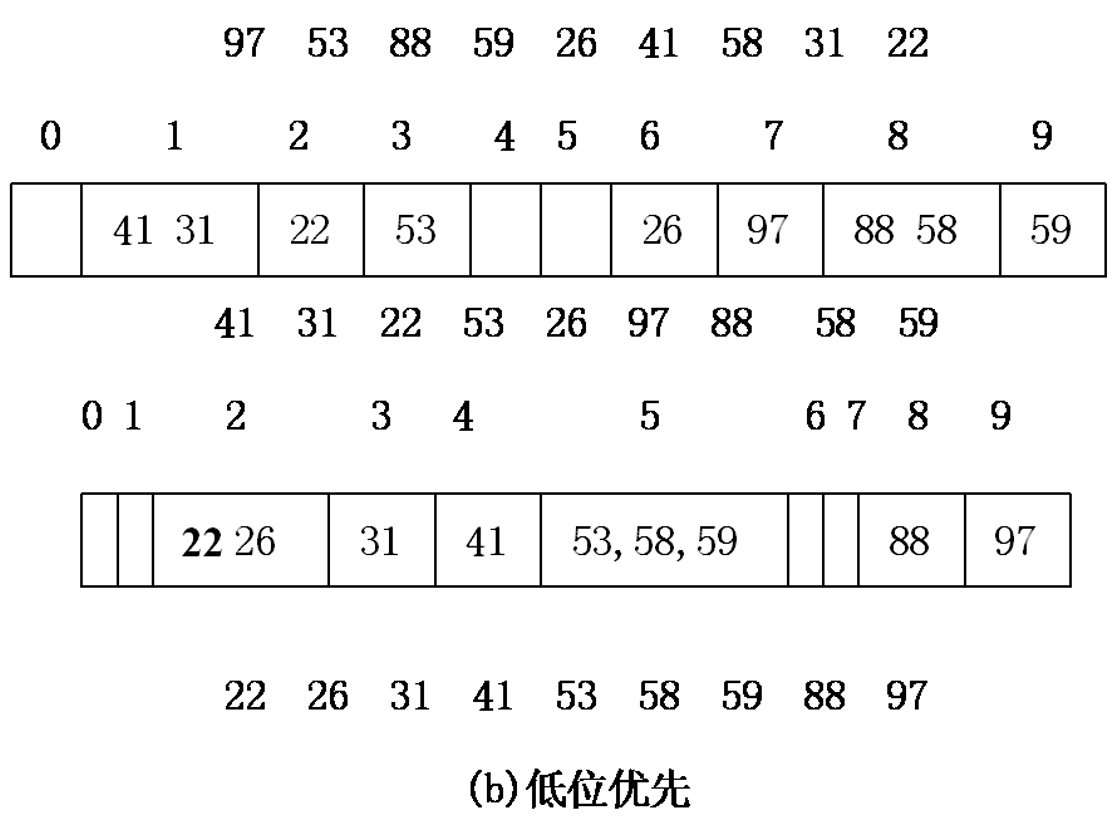
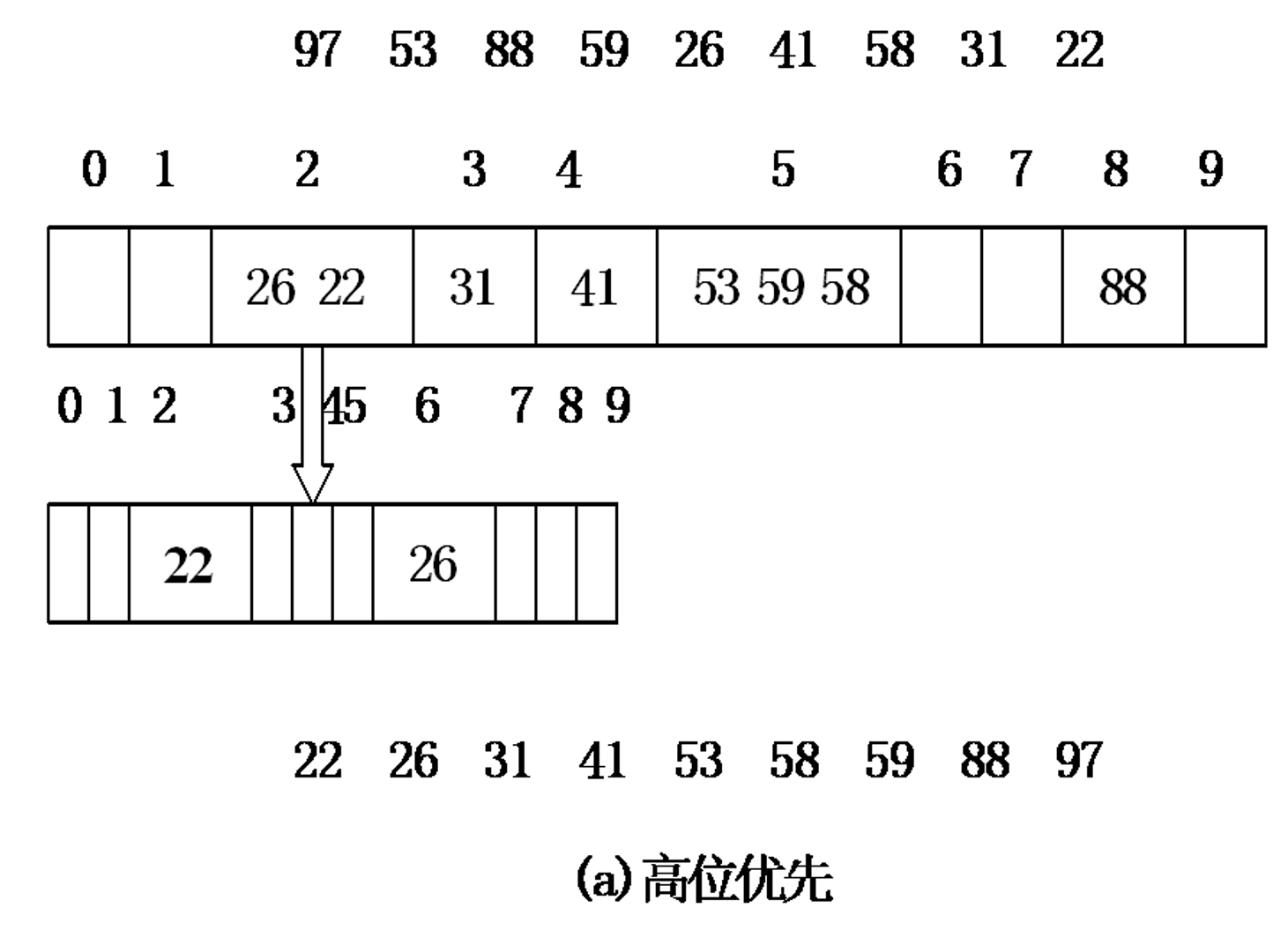
桶排序：时间O(m+n)，空间O(m+n)，稳定。m为桶的数量。

\*适用于m约等于n的情况

基数排序：

高位优先法MSD：递归分治问题。

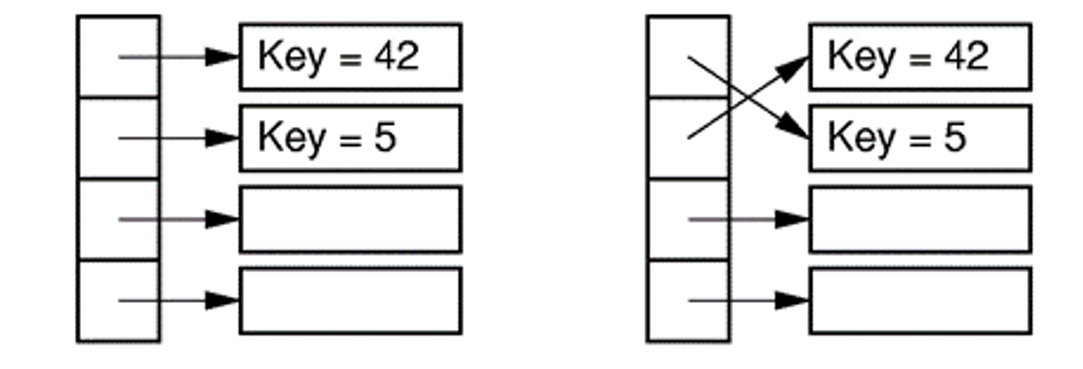
低位优先法LSD：直接排好，较简单，计算机常用。



时间O(d\*(n+r))，d为次数，r为基数数量，本质上还是nlogn。空间O(n+r)。

9. 索引排序

当数据规模很大时，不移动数据，用索引排完序后标识顺序。



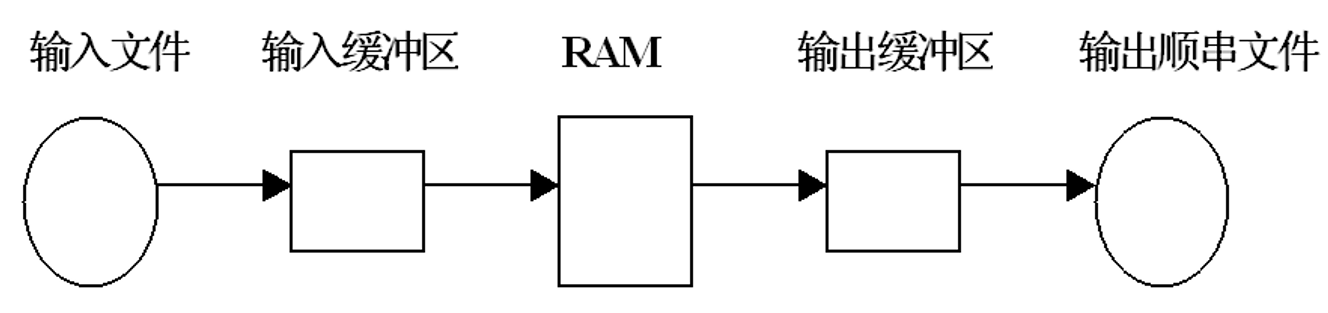
·总结



【外排序】

访问外存(10-3秒)比访问内存(10-9秒)慢5~6个数量级

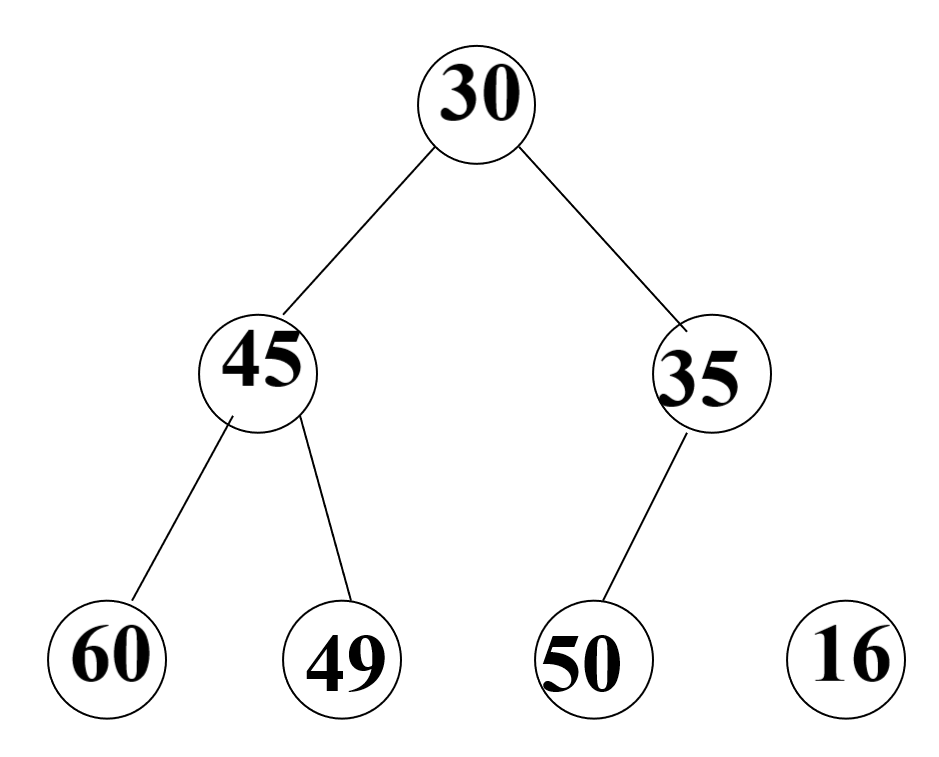
数据读取模式：



目标：1. 产生初始顺串（内排序）2. 归并排序（需要I/O读写）

1. 内排序采用置换选择排序：

在RAM中使用最小堆。为了使顺串尽可能长，每选出一个数后就将下一个数递补进来，如果下一个数比已选择的数小，则将它置于最后一位，堆的大小--。直到堆的大小为0后一个顺串生成完毕，废的部分正好构成一个新的堆，开始下一个顺串的生成。这样平均可产生的顺串长度是2\*堆大小(假如数据完全随机，由期望计算所得)。

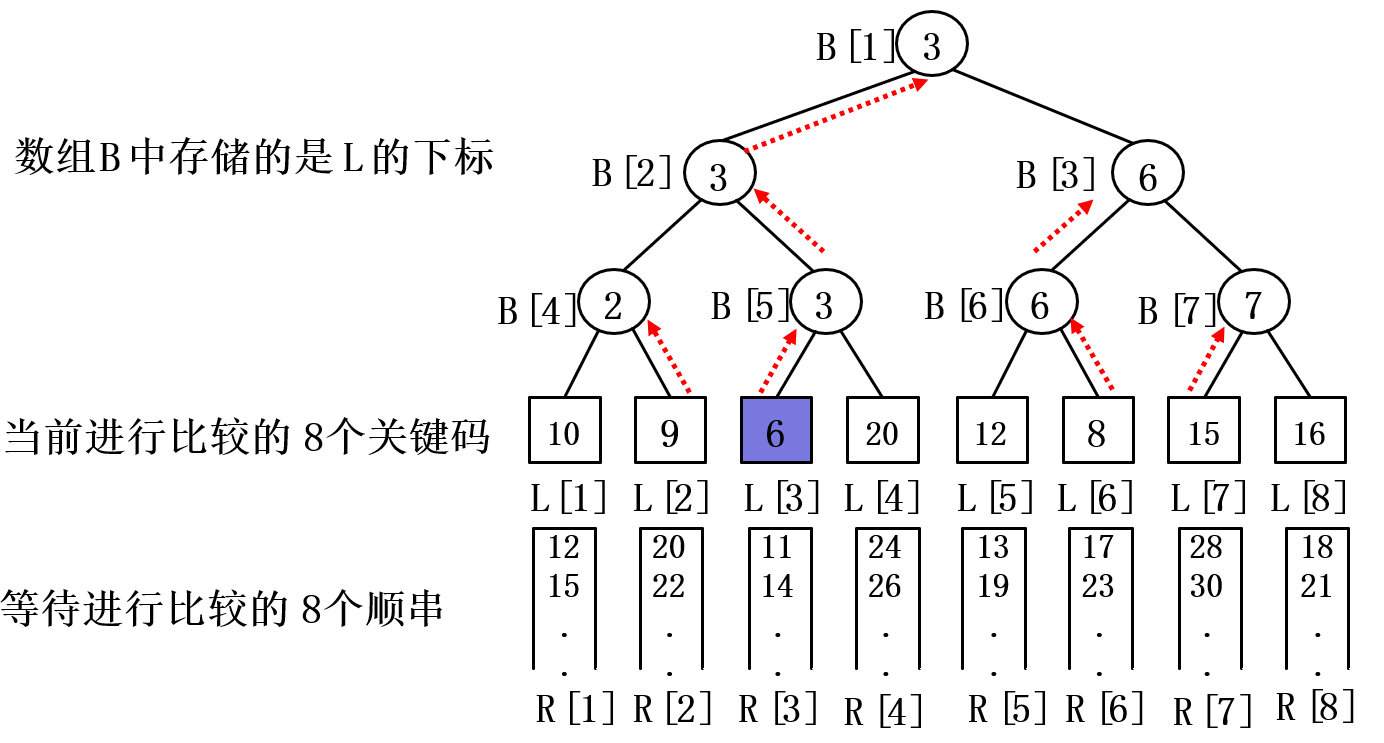


1. 归并排序

若采用二路归并，树高为log2(顺串个数)，即要进行log2(顺串个数)次读写，太慢了。于是采用多路归并。

多路归并时使用堆/赢者树/败者树来排k个数。

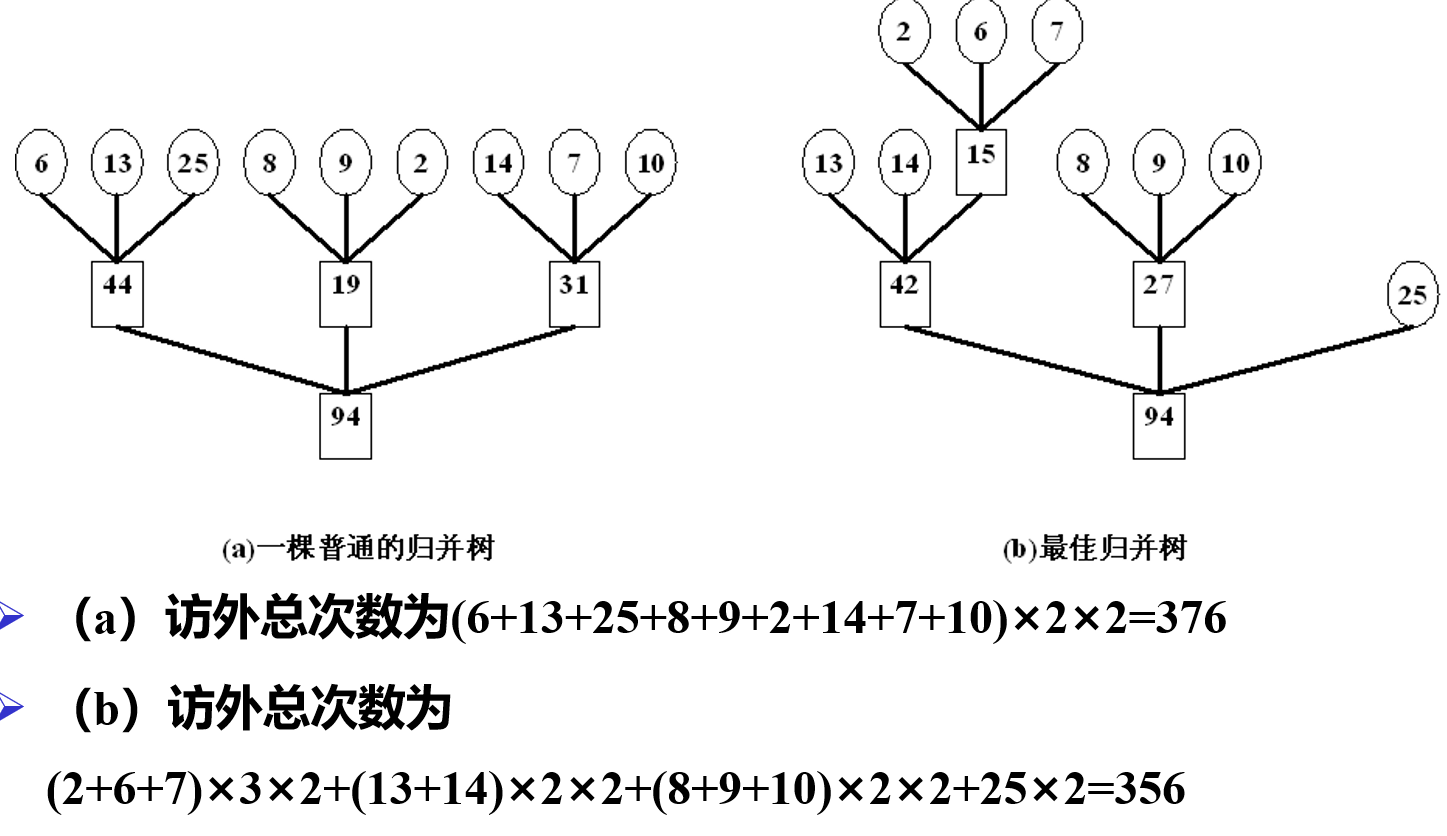
赢者树：



败者树类似。

堆更新时需要和两个儿子进行比较，赢者树只需要从叶子出发和兄弟比较、再往上，败者树只需要和父亲比较。败者树的复杂度为(k+n\*logk)，k为k路归并，n为顺串长度。

此外，由于各顺串长度不同，对外存扫描的次数会对执行时间产生影响。可以使用k叉Huffman树（最佳归并树）优化归并路径。

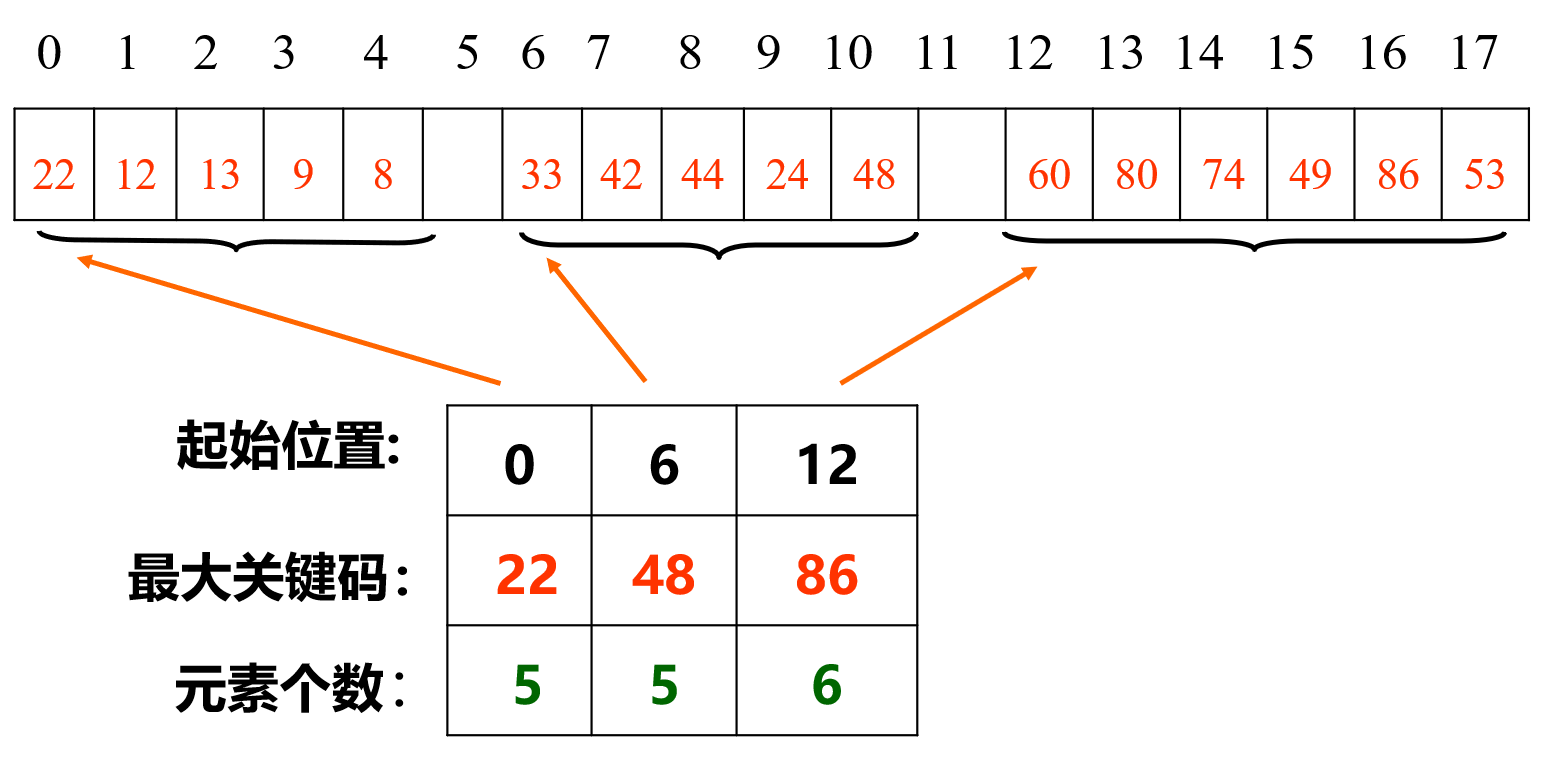


【检索】

衡量检索算法：平均检索长度（Average Search Length）= Σ找这个数据的概率\*找到这个数据需要的步数

·基于线性表的检索

1. 顺序检索：挨个儿找。不用提前排序、增删方便，但检索慢。
2. 二分检索：增删不方便。
3. 分块检索：顺序与二分的折中。分块有序，块内无序。

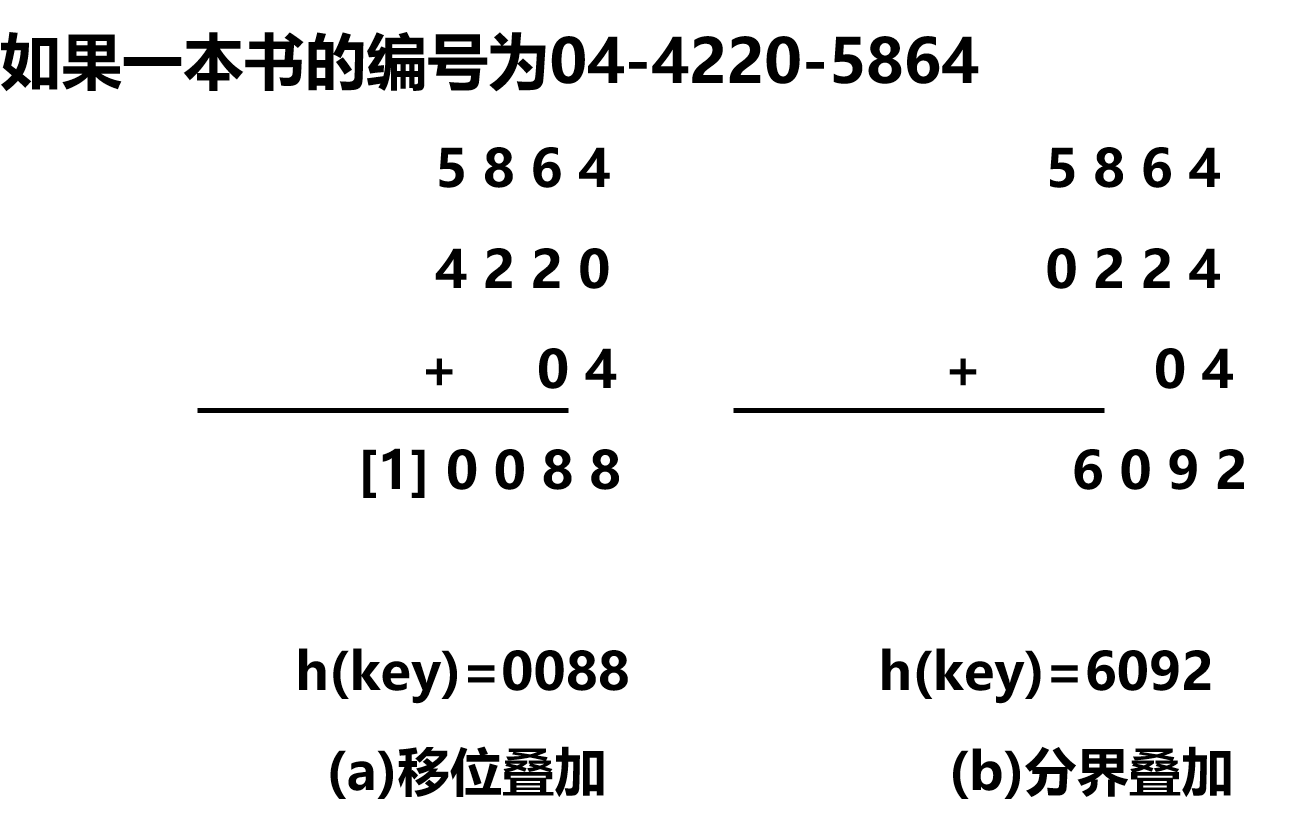


·散列检索（哈希）

目标：尽可能均匀分布，减少冲突

常用散列函数选取方法：

1. 取余法：缺点是容易形成连续的存储单元
2. 乘余取整法：key\*小数常数，然后取小数部分乘以一个整数常数向下取余
3. 平方取中法：将key平方后（放大差距）取中间几位。随机化最好。
4. 数字分析法：已知表中会有哪些数据后计算每一位是否均匀分布（如学号前几位就大家都一样），取计算出的较均匀分布的几位作为哈希值。
5. 基数转换法：如每位依次乘以130, 131, 132…再取其中若干位。
6. 折叠法：将关键码分割成尽量位数相同的几部分，然后取这几部分的叠加和（舍去进位）作为散列地址。



·哈希冲突的解决办法

1. 开散列方法

拉链法：加链表。（可以根据输入顺序、访问频率顺序、值的顺序）

但如果数据存在外存中，链表元素可能存在不同的page里，引发多次磁盘访问，浪费时间，于是引入↓

桶式散列：散列文件记录分为若干桶，每个桶包含若干页块。桶内各页块用指针链接，每个页块包含若干记录。

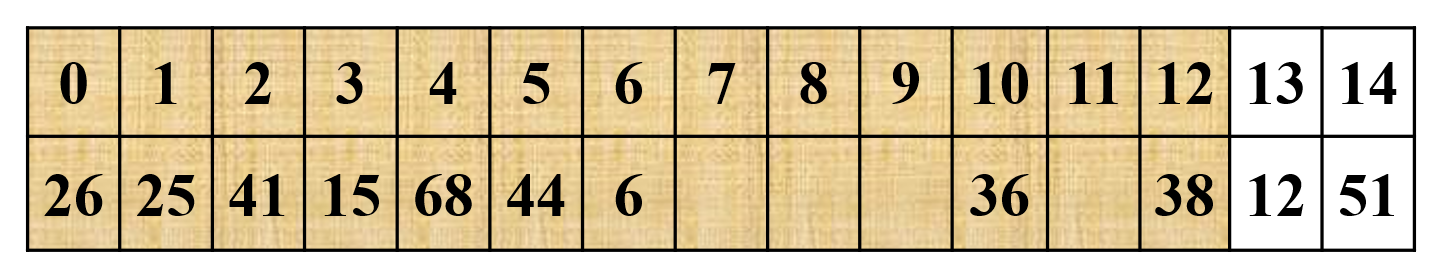
1. 闭散列方法



若所有后继散列地址都不空闲，说明该闭散列表已满，报告溢出。

探查方法：

线性探查法：如果这个位置有人了就放在下一个。优点是不会出现伪溢出，缺点是容易产生聚集，导致很长的探查序列。



//此时下一条记录放在7的概率是9/13，放在11的概率是2/13。

\*改进线性探查法：如果这个位置有人了就放在+C的地方，则基位置相邻的记录不会进入同一个探查序列，但隔C的key们仍然会产生聚集。

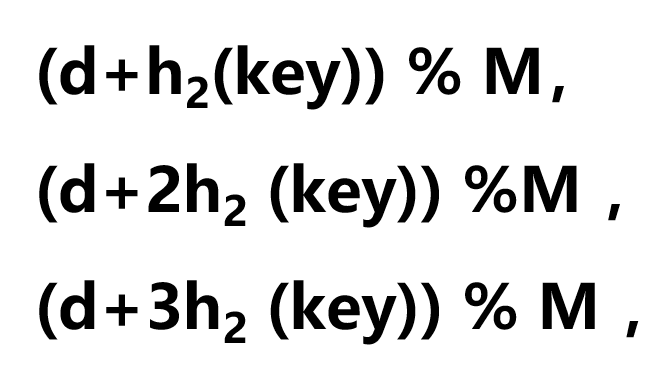
二次探查法：探查序列为 ( h(k) + c1​\*i + c2​\*i2 ) mod m。采用二次函数，使得探查序列不会重合。

伪随机探查法：探查序列使用一个乱序的permutation数组，例子如下。则也可以使探查序列不重合。



然而，虽然二次探查法和伪随机探查法可以消除基本聚集（基地址不一的key的聚集），但确无法消除二次聚集（基地址一致的k们产生的聚集）。

双散列探测法：探查序列是原来关键码值的函数。如↓

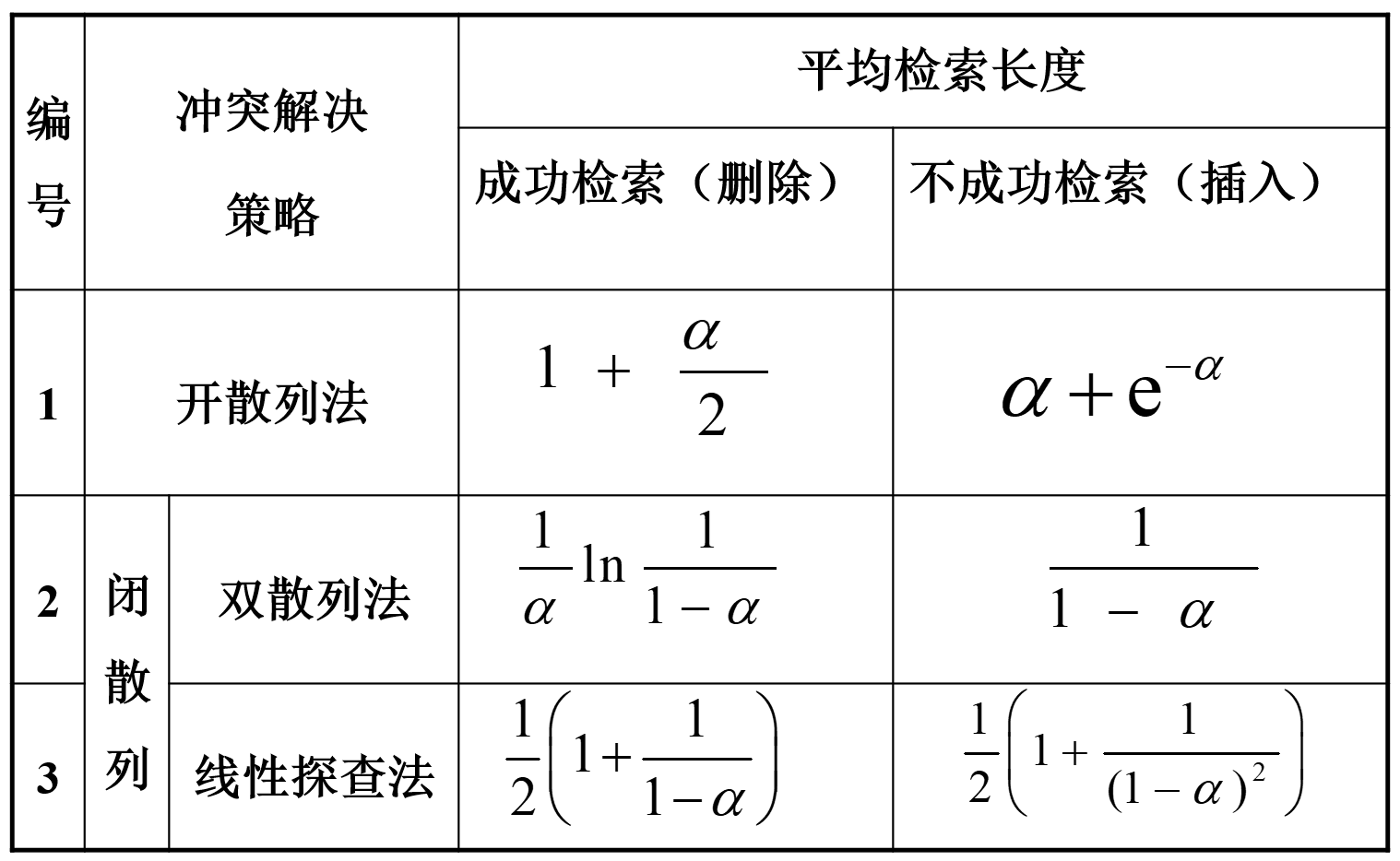


优点是不易产生聚集，缺点是计算量较大。

整体执行：

* 插入
* 检索
* 删除：注意闭散列中只能标记为墓碑，不能真正删除，空间未再次分配之前不可用，以防影响后续探查序列中所存数据。插入新数据时要确认整个探查序列中没有该数据，才可覆盖墓碑。

总结：（α为负载因子(装填量/总空间)）



\*负载因子的临界值是0.5（将近半满）。大于这个临界值，性能就会急剧下降。

α< 0.5时，大部分操作的分析预期代价都小于2，远远好于二分检索。

\*如果插入和删除操作很频繁，散列操作检索效率将受到影响（负载因子增加、墓碑过多）。实际应用中，对于插入和删除操作比较频繁的散列表，可以定期对表进行重新散列，把所有记录重新插入到一个新的表中，最频繁访问的记录放到其基地址。

【索引】

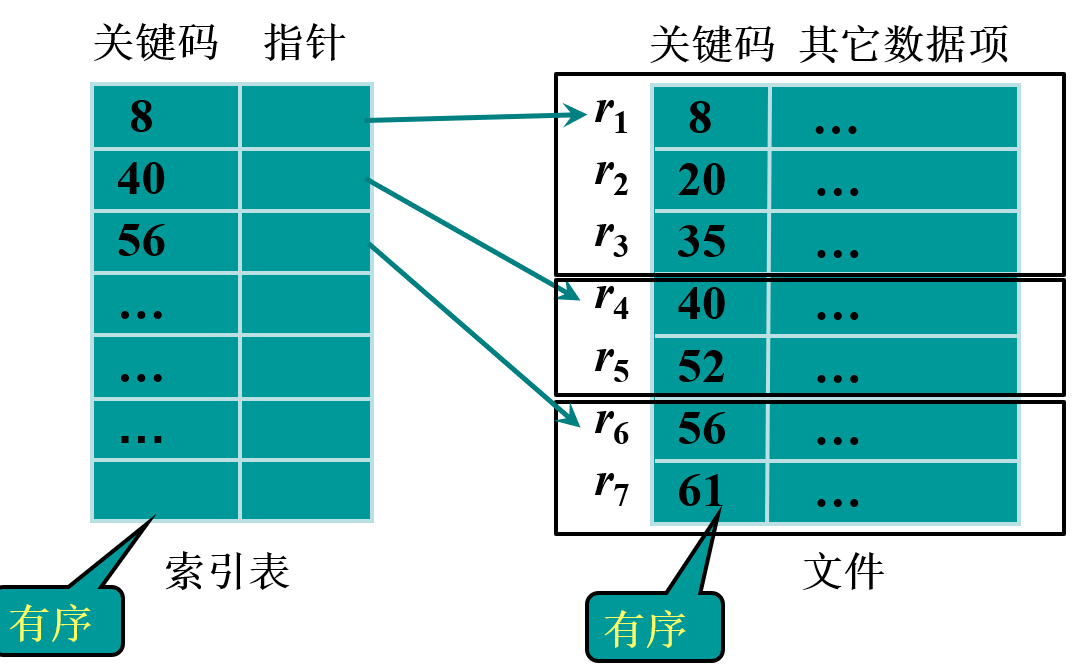
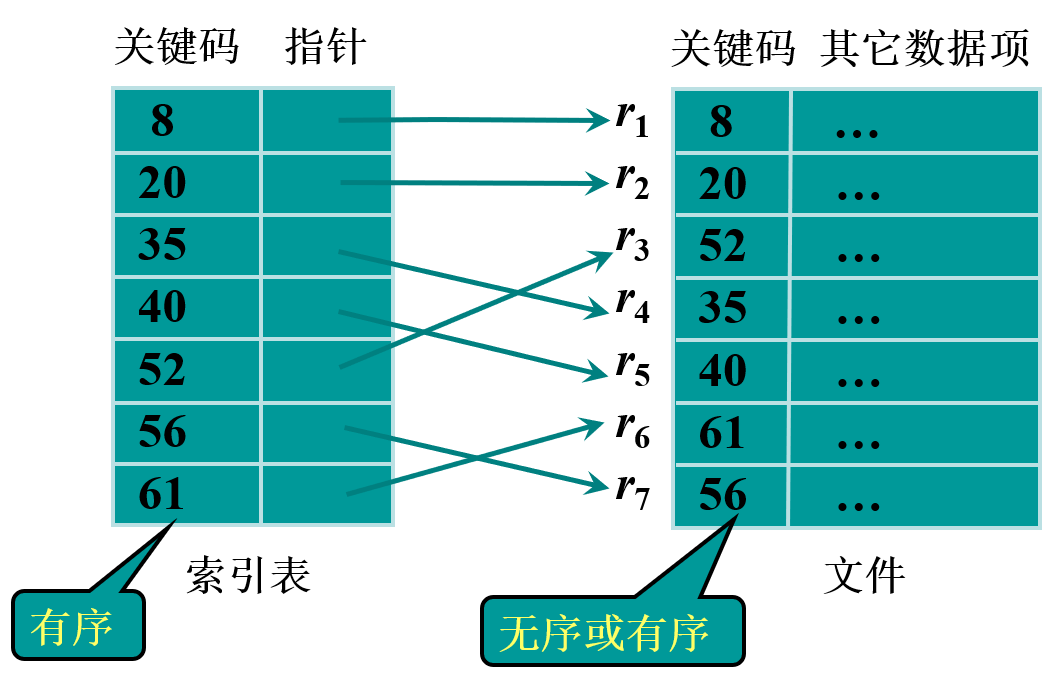
·主码：是数据库中的每条记录的唯一标识

辅码：假如编号是主码，但要按照名字查找，就需要辅码索引。

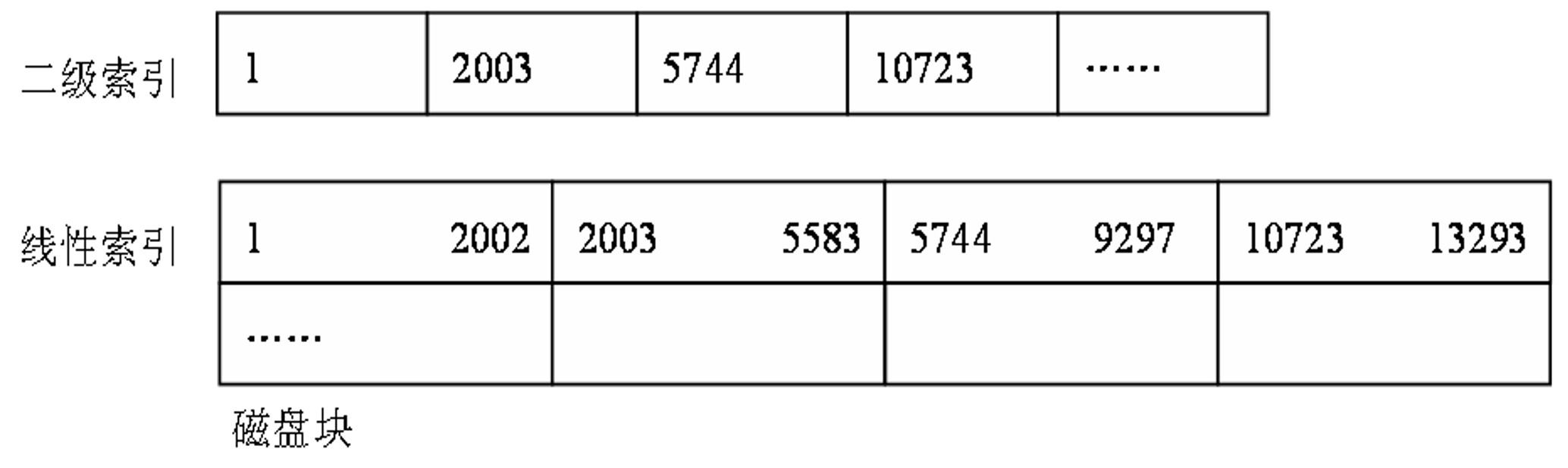
一个文件可以有多个索引文件，索引是 (关键码，指针) 对。

·稠密索引（下左）：对每个记录都有索引项

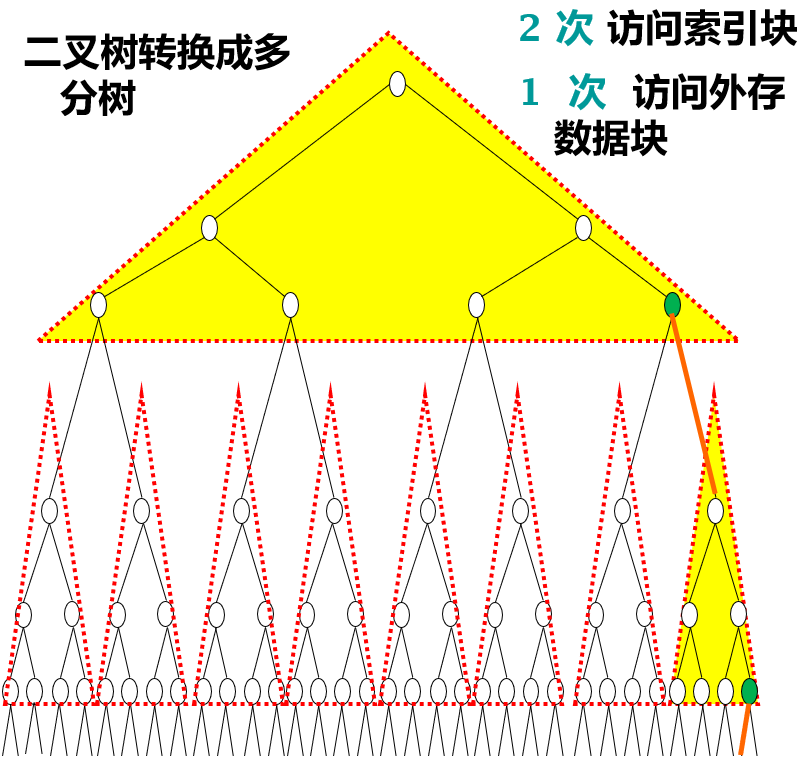
稀疏索引（下右）：对一组记录记录一个索引项



1. 线性索引
2. 二级线性索引



\*二叉树转多分树检索

（多分树的结点数量由数据块大小决定）

\*ISAM树：对新插入的数据采用溢出页来存，就是该存放的位置伸出去一块未排序的表。

3. 倒排索引

Eg. 

在文本索引中使用较多。

* 词索引（问题：如何抽词干、切词）

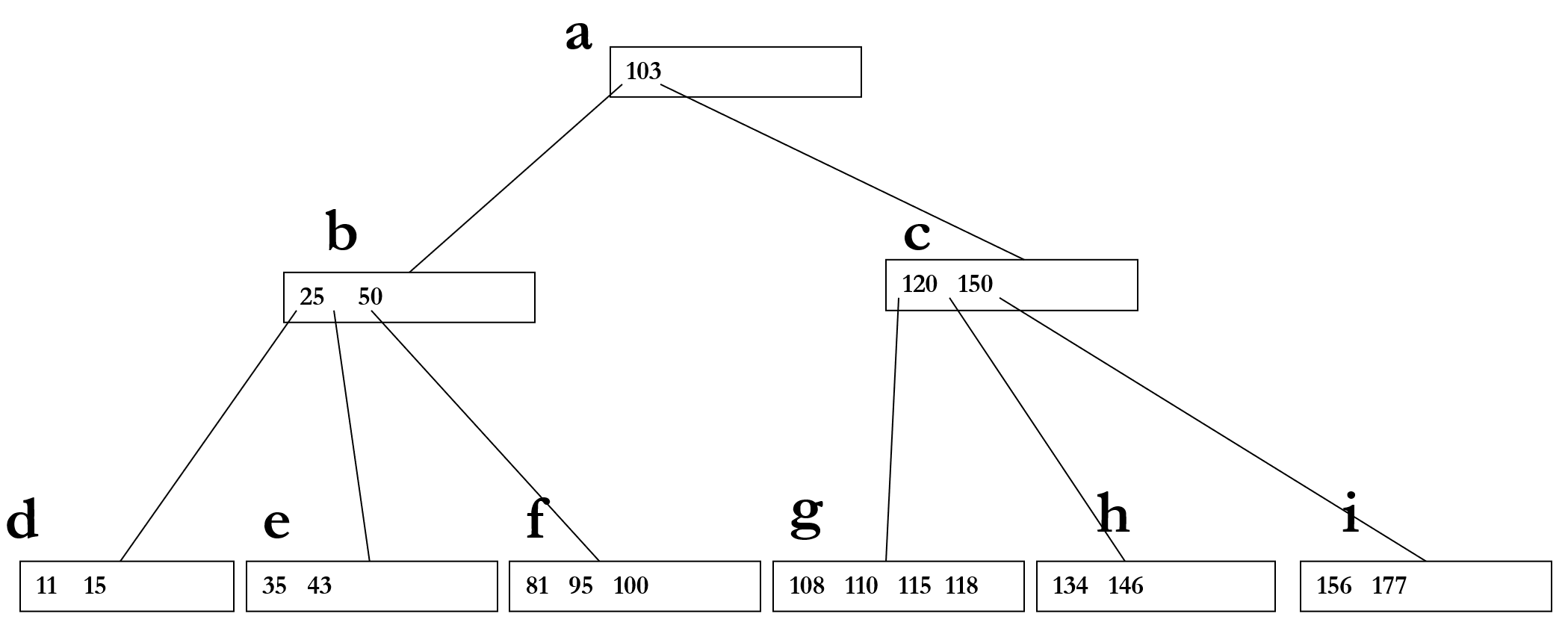


* 全文索引：可以对每个字符串（n-gram，比如3-gram就是三个三个切）建立索引，从而使查询词不再限于关键词。Eg. DNA查询。（如果在3-gram中要找更长的单词，可以找出Pek, eki, kin, ing后求交集）

1. B树（balanced tree）

m 阶B-树是一棵 m 路查找树，或者为空，或者树中每个非根结点的儿子数在[ **⎡*m*/2⎤ , *m***] 之间，根结点至少有2棵子树。有k棵子树的结点有k-1个关键码，即关键码个数在[ **⎡*m*/2⎤ -1, *m*-1**]之间。

Eg. 5阶B树

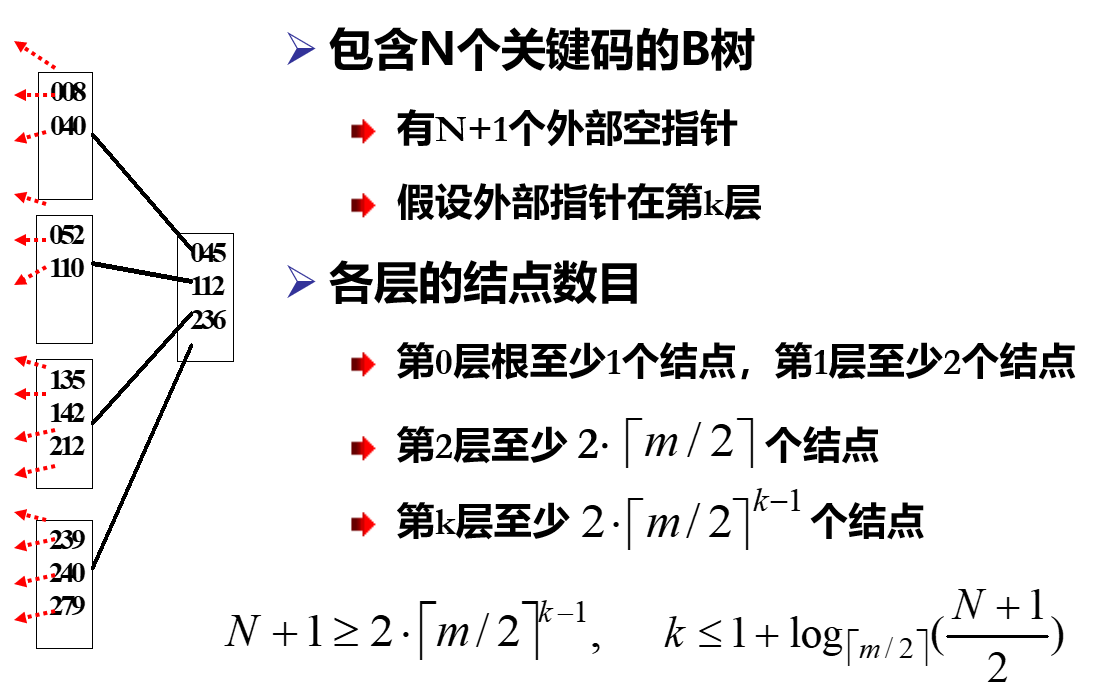


插入：搜索到叶子结点，发现没有，于是插入到叶子结点中。如果叶子节点到了m个，则将中间值上提到父亲节点，左右值分裂为中间值的左右子树。如果父亲节点此时满了，则重复该操作，直到根节点。如果根节点满了，则相同操作，树高+1。

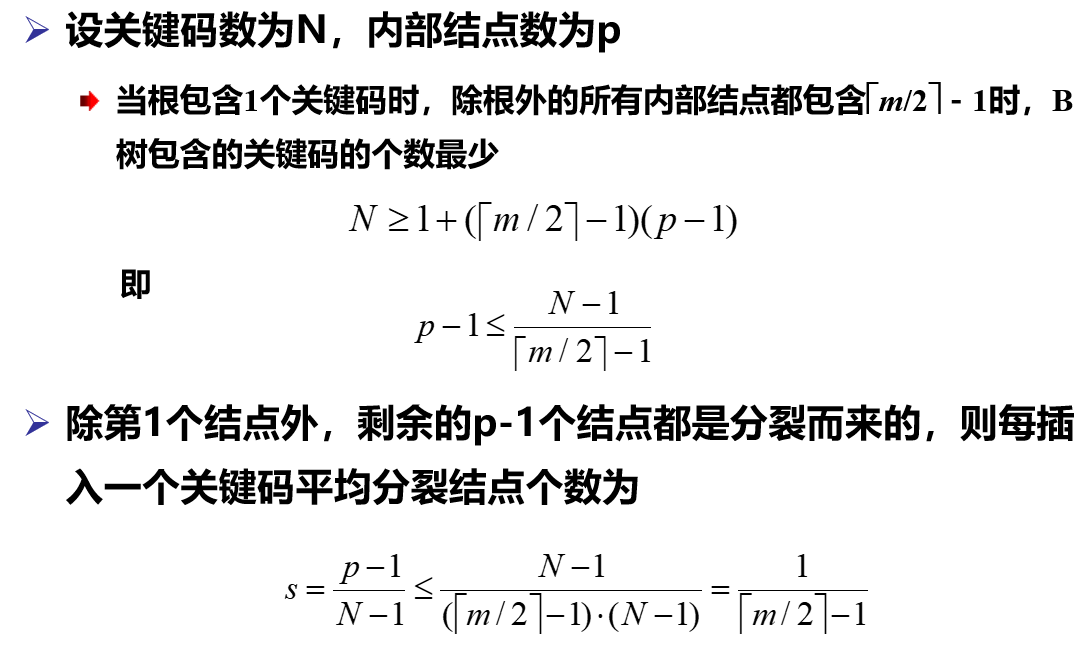
删除：若删除元素不在叶子结点，则将删除元素和它在树中的后继交换（右子树中最小的值），交换完后再删除。若删除操作导致结点关键码过少，则试图向左边的邻居借一个关键码（左边邻居上去，父亲下来到右边）。若左边邻居也卡在临界点无法借给你，则两个结点和父亲下来合并。如果父亲结点此时过少了，重复该操作。

性能分析：好！

层高少

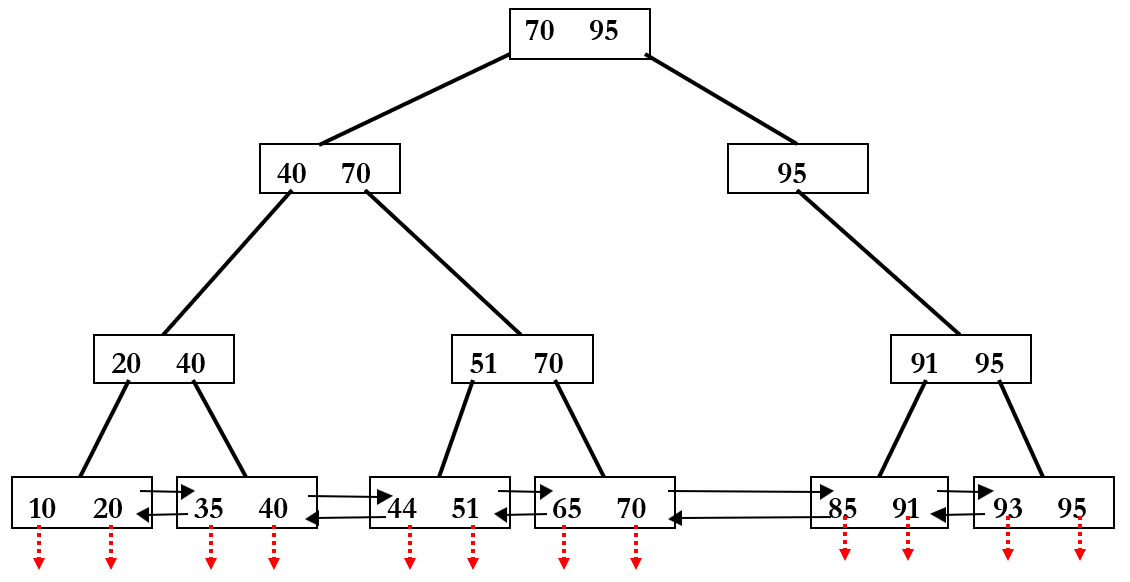


分裂次数少



1. B+树

把所有的关键码都放在叶子节点，中间结点不起到指向文件的作用。其他和B树差不多。叶子节点间一般有双向链表，方便区间查询。现在B+树较为常用。



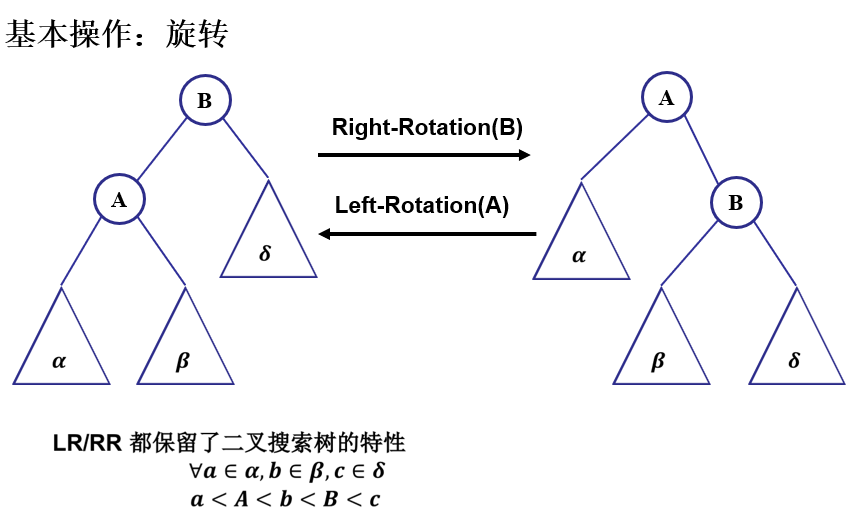
1. 红黑树（内排序索引）

基本定义：一种BST树，每个结点为红或黑，树根和树叶为黑，父子结点不允许都为红色，任意节点到其每个叶节点的路径上包含相同数目的黑结点。

(结点，叶子节点] 上的黑色节点数量称为结点的阶，根结点的阶称为该树的阶。

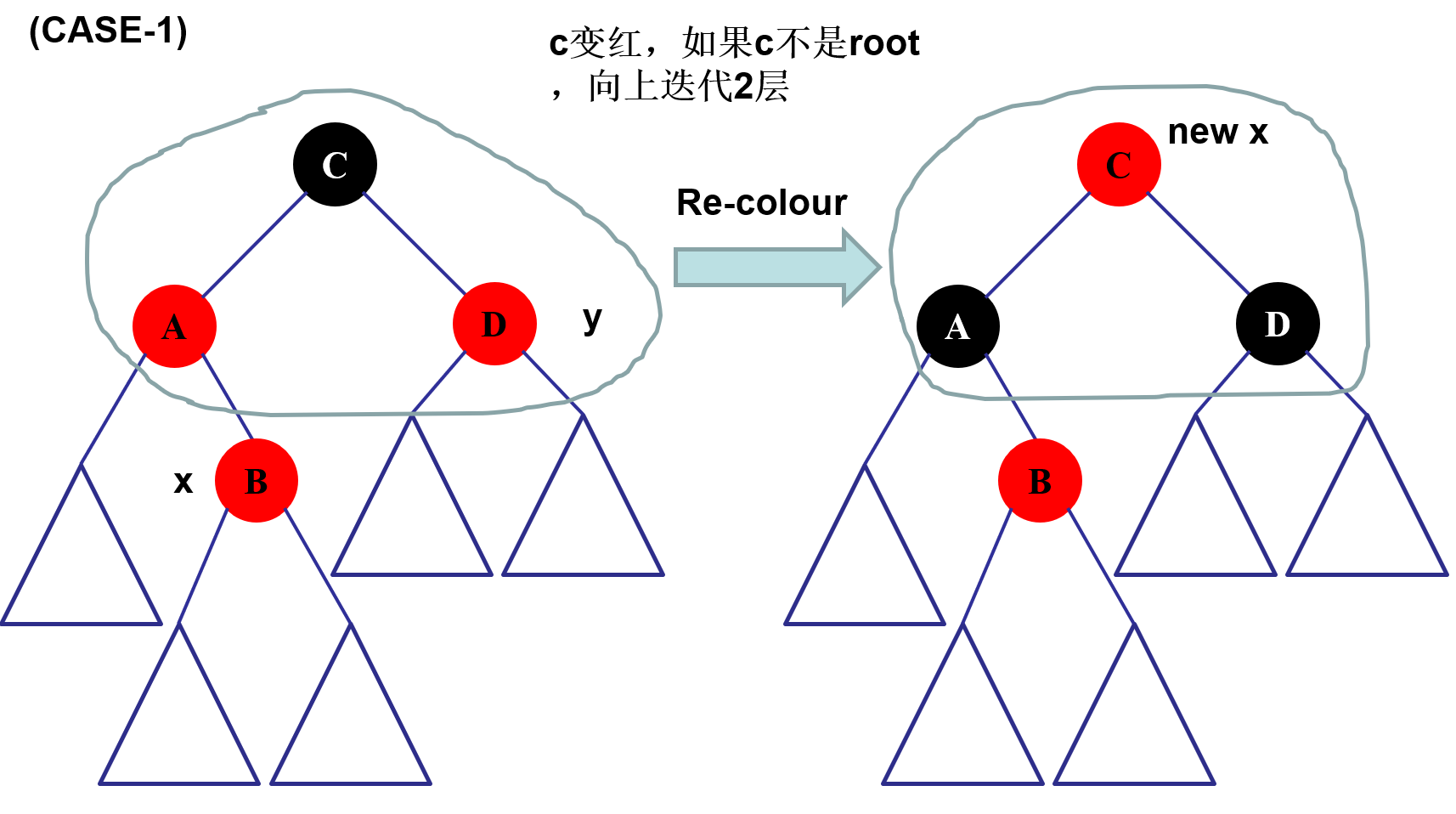
易知有n个内部节点的红黑树高度为logn级别，即操作均为logn级别。（因为限制了最长路径最多为最短路径的两倍）

基本操作：旋转（想象以AB为跷跷板）

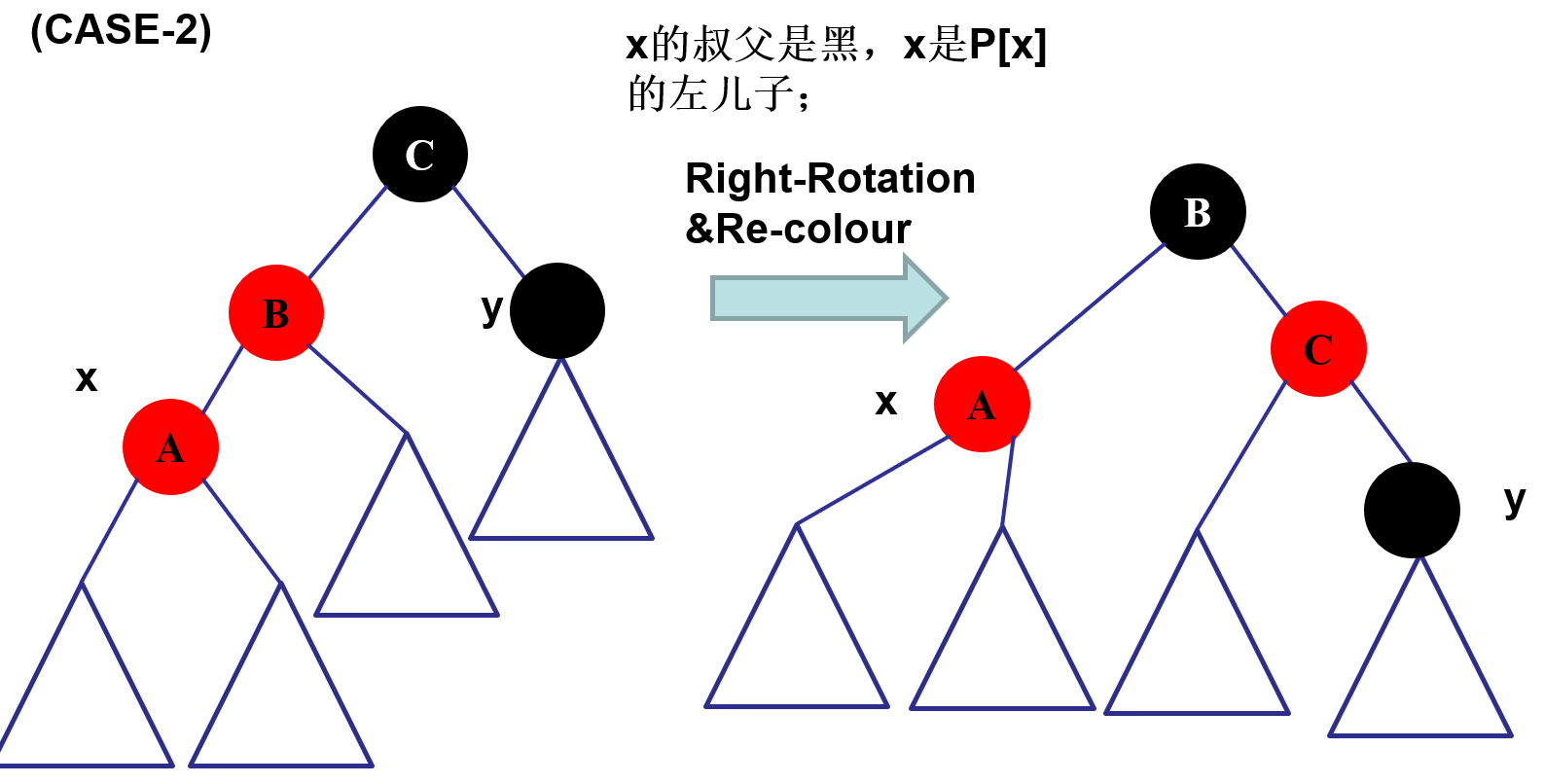


插入：将插入的结点标为红色。若与父亲产生红红矛盾，则调整。

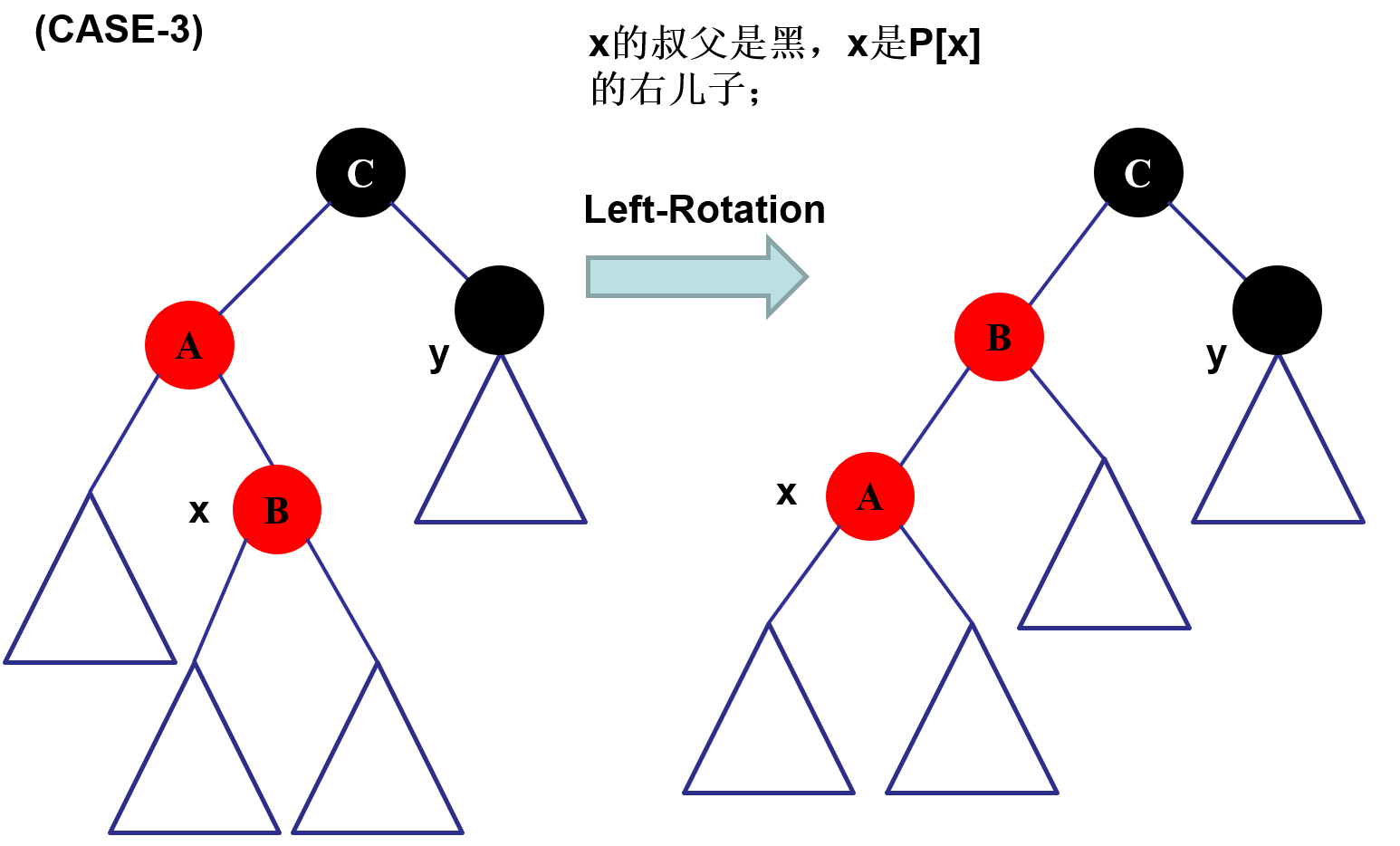
如果爸爸和叔叔都是红的，进行recolor，将颜色传递给爷爷，再看爷爷是否和太爷爷有红红矛盾。



如果叔叔不是红的且自己、父亲、爷爷形成一条直线，则进行父亲-爷爷rotate，再把两者颜色互换，问题解决。



如果自己、父亲、爷爷成之字形，发现按照2的情况没法解决问题（红的换不掉），所以先调整到2（通过自己-父亲rotate）再按照2的情况换。



插入操作是logn的，因为2和3直接常数操作完成调整，1向上两层两层迭代。

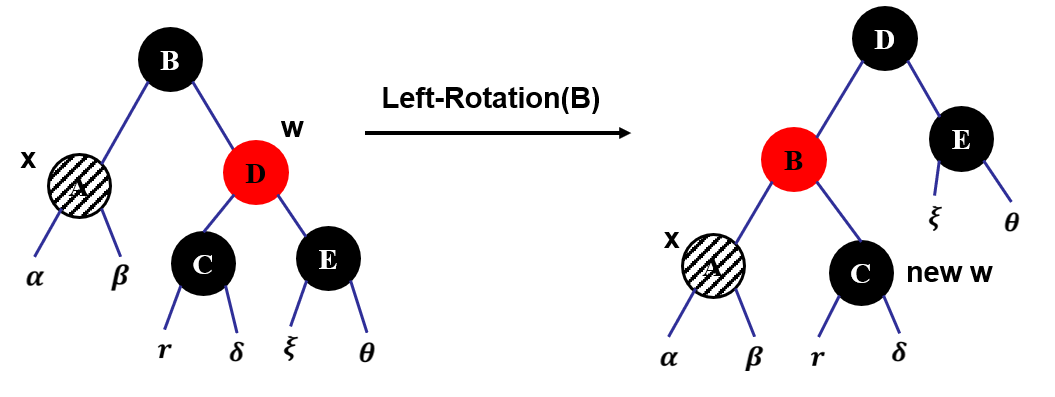
删除：

首先回顾BST的删除。若删除结点为叶子结点，则直接删除；若删除结点有一个儿子，则将儿子与爷爷相连，再删除自己；若删除结点有两个儿子，则在结点子树中找到结点后继，将后继与结点互换，再删除结点。

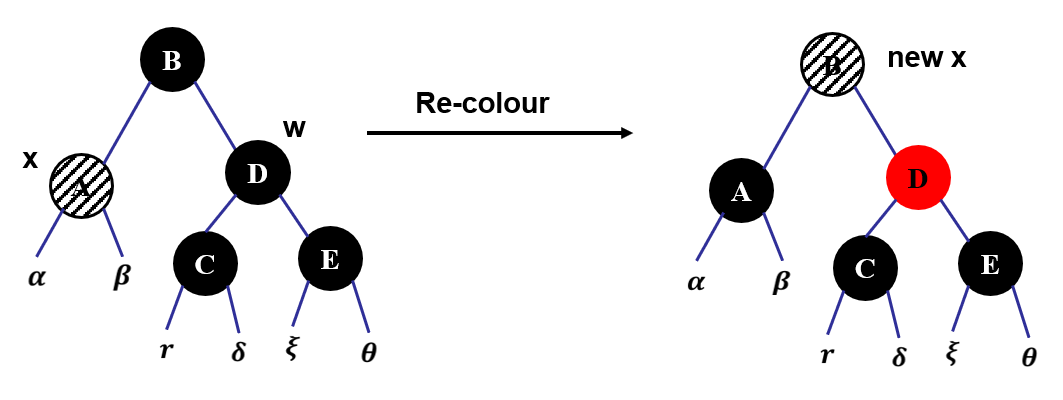
红黑树中同理。（若互换，颜色不换）

若要删除的结点为红色，则直接删除就好。若要删除的结点为黑色，则将黑色传递给儿子。若儿子为红色，直接变为黑色，结束。若儿子为黑色，标记儿子为双重黑色，然后想办法调整。

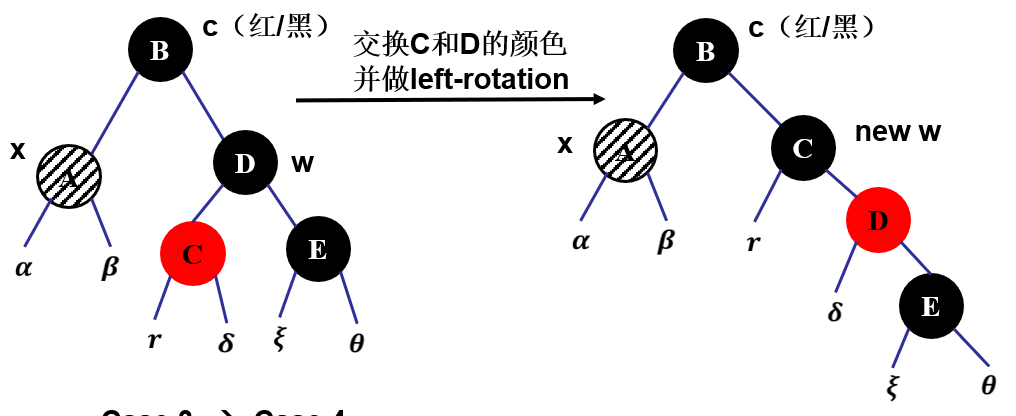
若兄弟是红色的，进行如下调整，使兄弟为黑色。



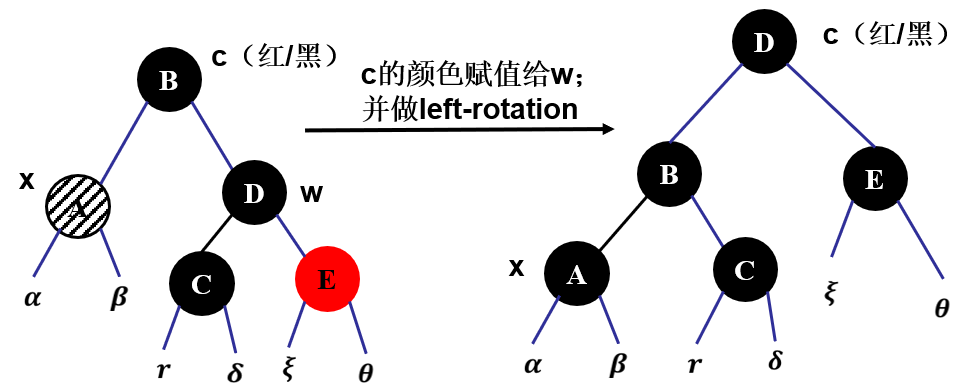
此时，若兄弟的两个儿子都是黑色的，则可以将自己与兄弟都剥去一层黑色给父亲，使得黑色可以往上传递迭代（如果父亲为红色，即可调整为黑色，结束）。



若两个儿子不都是黑色的，和插入过程类似的将之字形转化为一字形。



一字形后即可进行终结操作。



删除操作是logn的，因为1只是一个初始化操作，3和4都是常数操作终结调整，只有2会向上迭代。

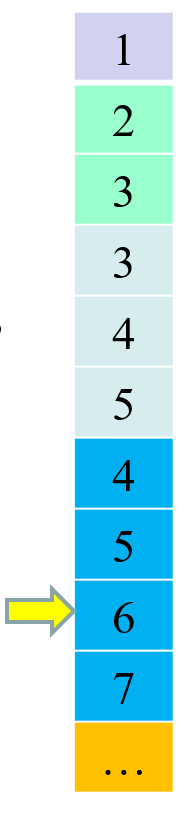
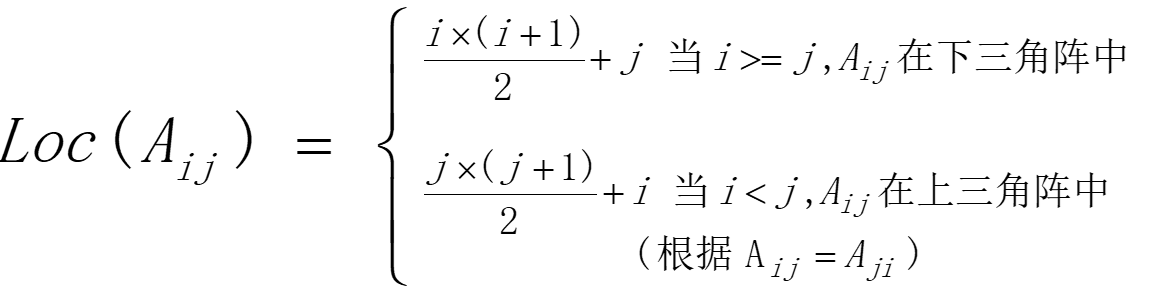
【高级数据结构】

1. 多维数组

\*特殊矩阵：存储有一定规律，可以压缩存储。

Eg. 对称矩阵、三角矩阵（均只需要一半的空间）

对称矩阵：

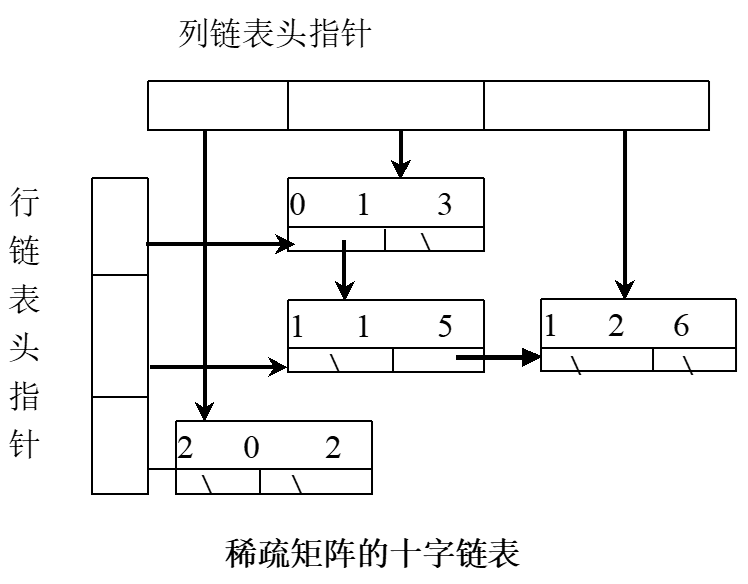


Eg. 稀疏矩阵（只存非零元）

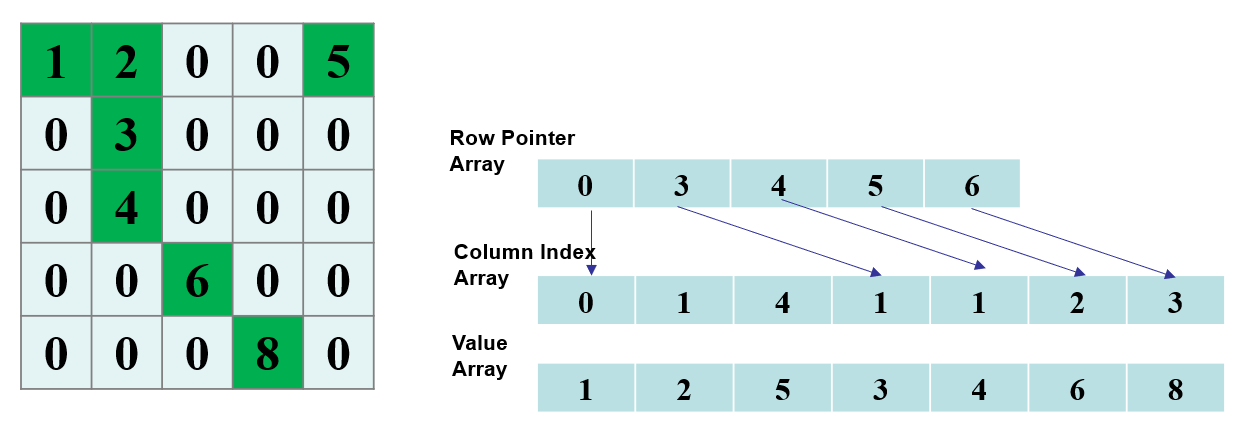
三元组表：



十字链表：



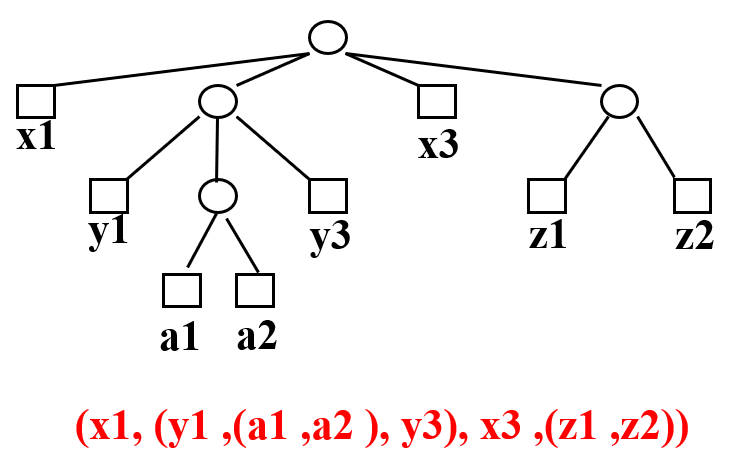
CSR（压缩稀疏矩阵表示）：



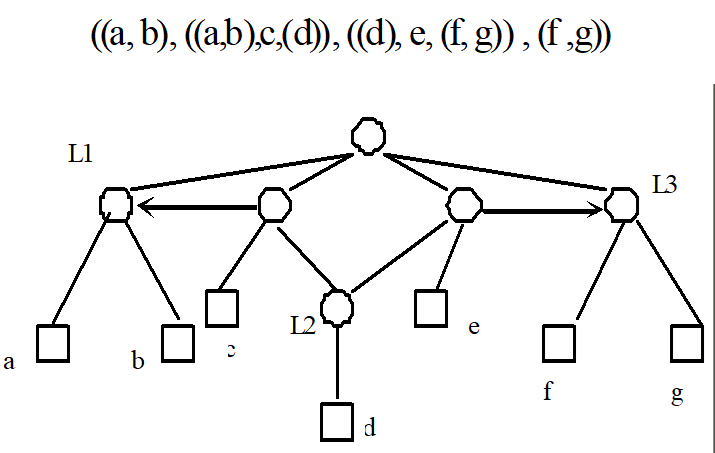
1. 广义表：元素可以是单个原子或子表。

广义表深度：括号层数

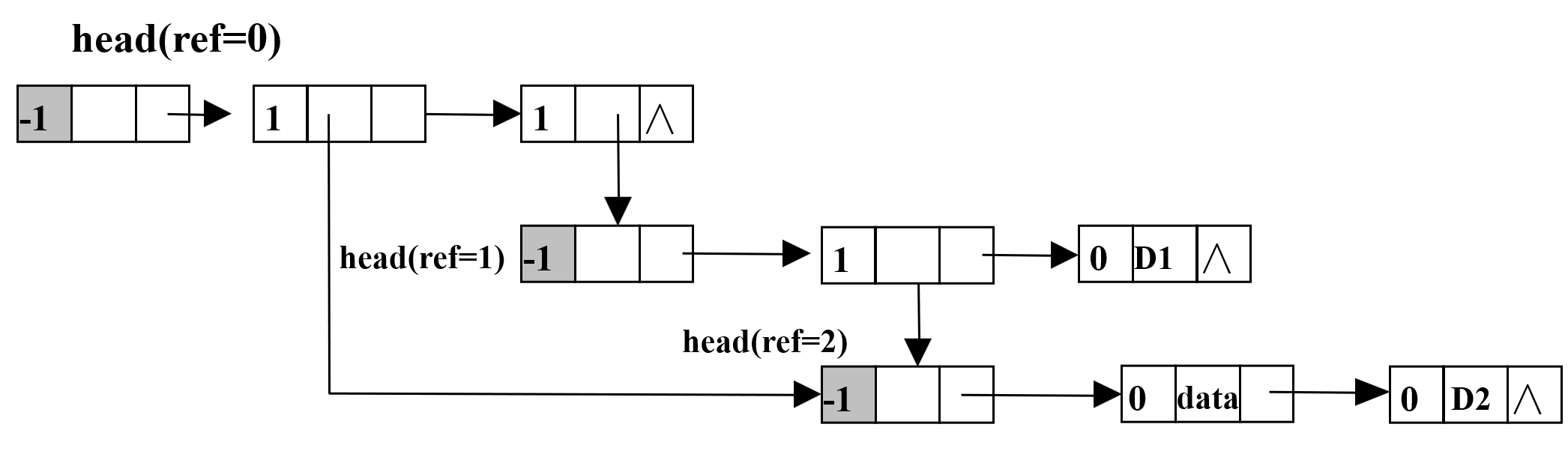
·纯表：任何元素（原子、子表）只能在广义表中出现一次。对应树。



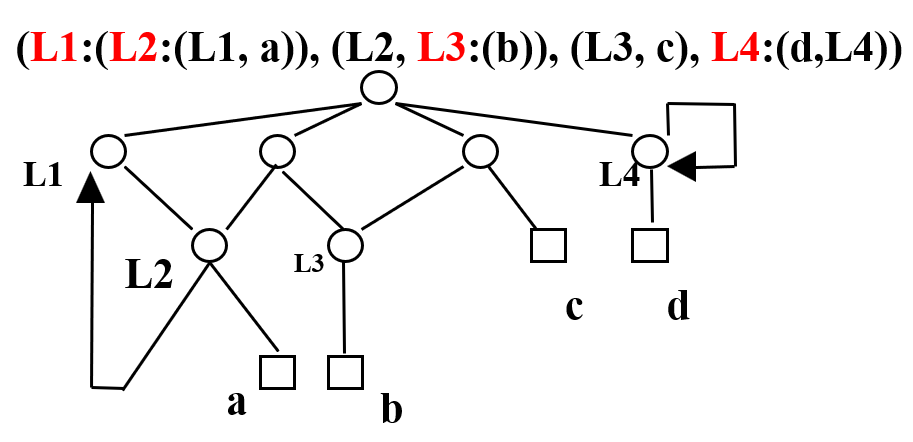
·可重入表（再入表）：元素可重复出现。对应DAG（有向无环图）。



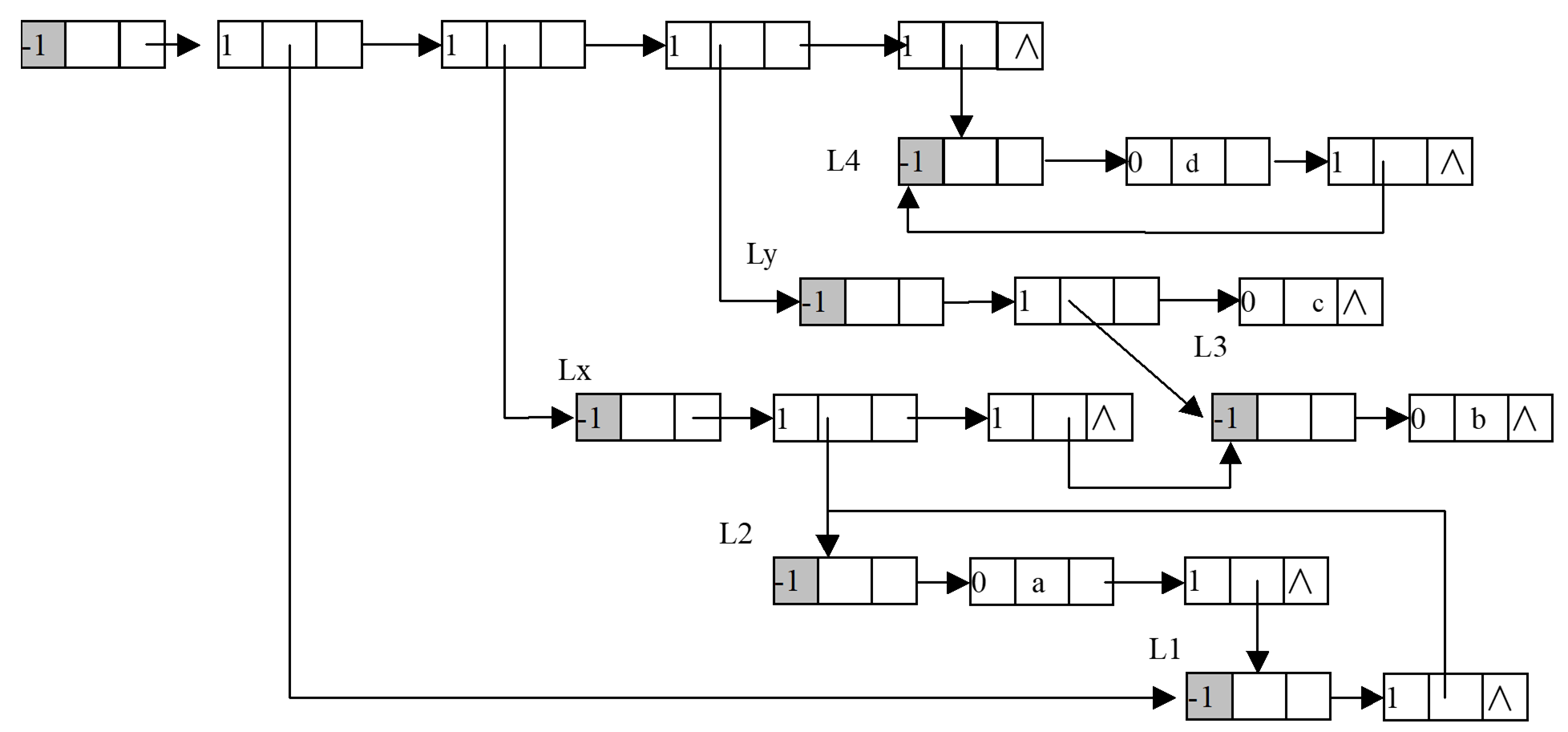
存储：



·循环表：包含回路，深度为无穷大。

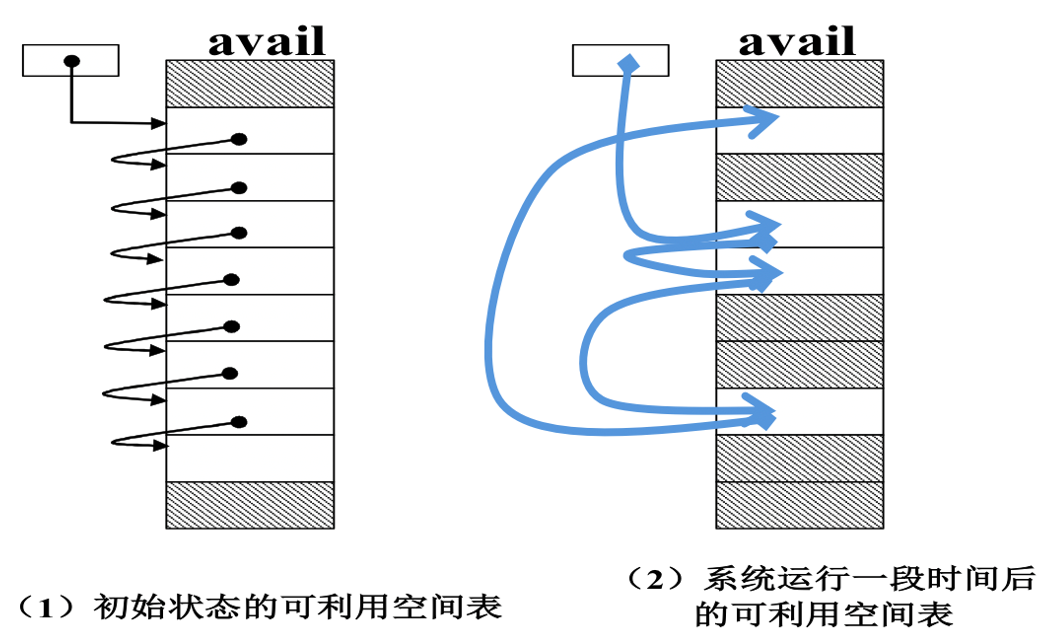


存储：

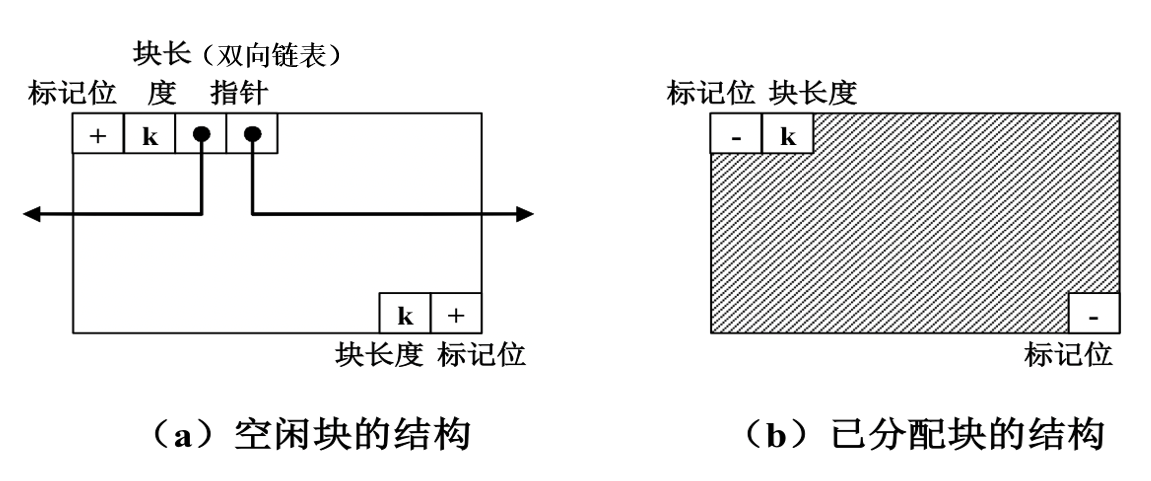


1. 存储分配与回收

：可利用空间表



对于不定长的内存块，需要考虑申请时找到其长度大于等于申请长度的结点 截取合适的长度 被删除的结点空间能否与邻接合并



\*外部碎片：将找到的空闲块中的N个字节分配出去，同时将剩下的K-N个字节作为新的空闲块。

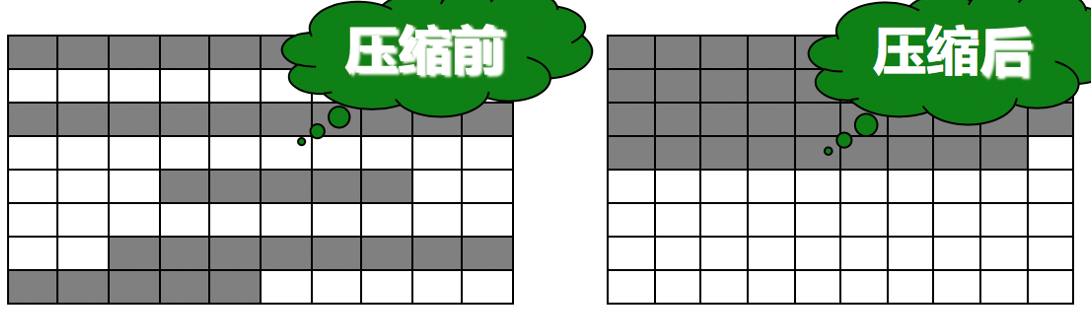
\*内部碎片：将整个空闲块分配。

分配策略：首先适配（遇到能放的就放）、最佳适配（找放得下的最小的内存块）、最差适配（找放不下的最大的内存块）。

如果遇到因内存不足而无法满足一个存储请求，存储管理器可以有两种行为：什么都不做，直接返回一个系统错误信息；或使用失败处理策略 (failure policy) 来满足请求。

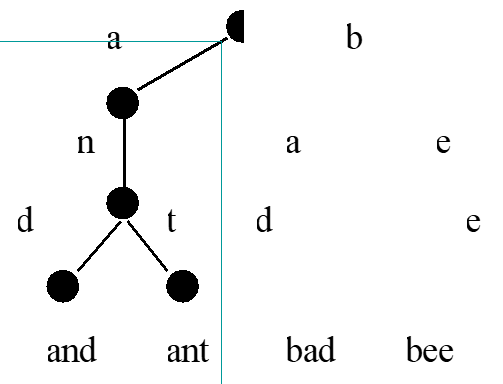
失败处理策略：

内存压缩



无用单元收集（最彻底的失败策略）：普查内存，标记把那些不属于任何链的结，将它们收集到可利用空间表中。

1. Trie树（前缀树）

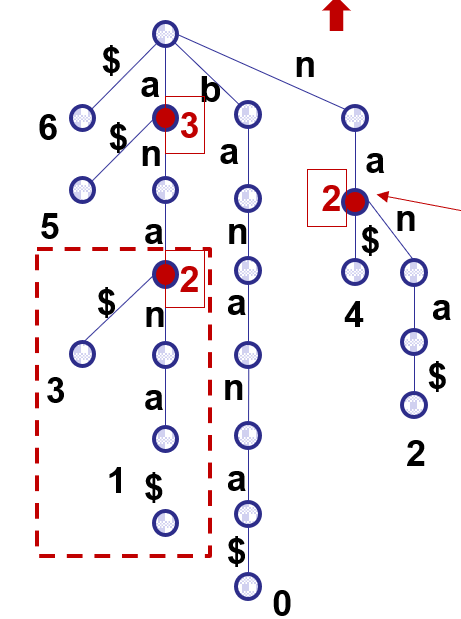


（最后可加上&符号表示单词结束来防止某单词是另一单词前缀的情况）

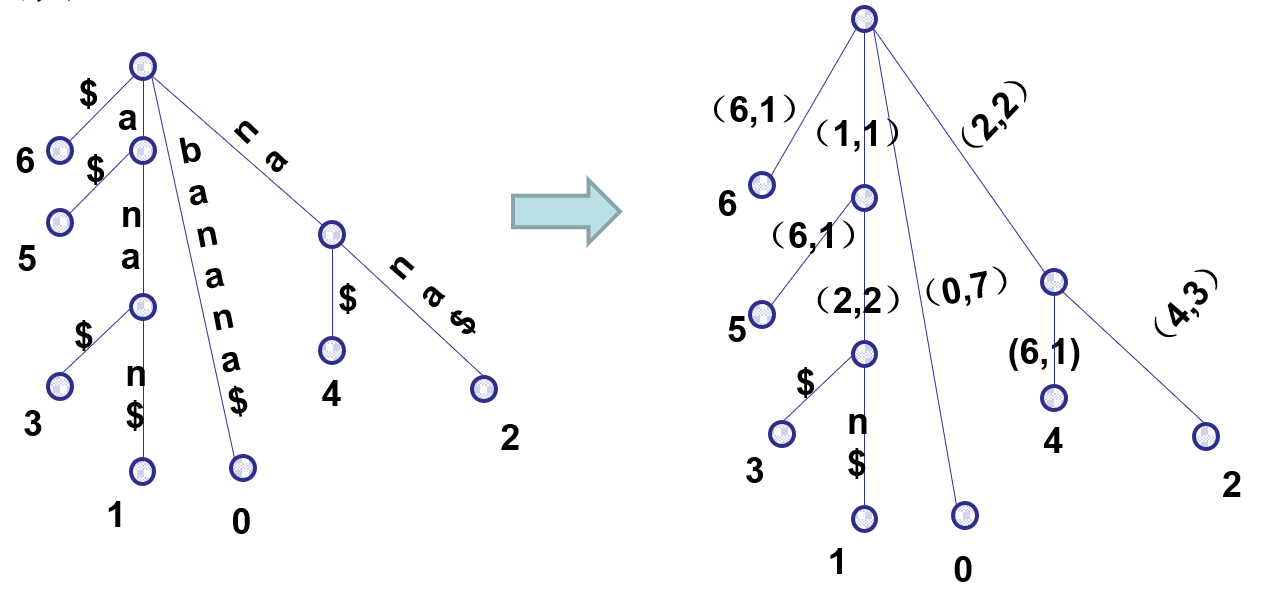
查找复杂度是O(字符串长度)，与集合内有多少元素无关。

\*后缀trie树：对一个特定字符串，用所有后缀子串建立一棵trie树，即可用这棵后缀trie树判断是否含有某个子串。（也可判断子串出现次数，比如ana，看ana后(红框)有几个叶子即可）（也可判断最长重复子串，看最深的分叉结点即可）

Eg. banana

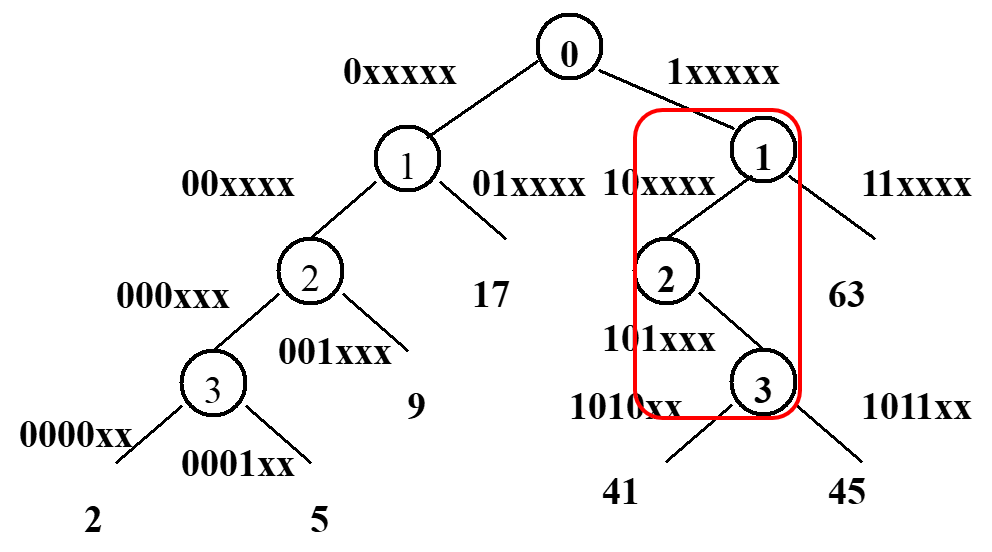


可见空间需求大：O(n2)。可压缩：将没有分叉的路径压缩成一条边（压缩完后非叶子节点<叶子节点数-1）每条边上的label，按照（offset，length）的方式来表达，这样对于字母只需将存储字符串本身存好即可。



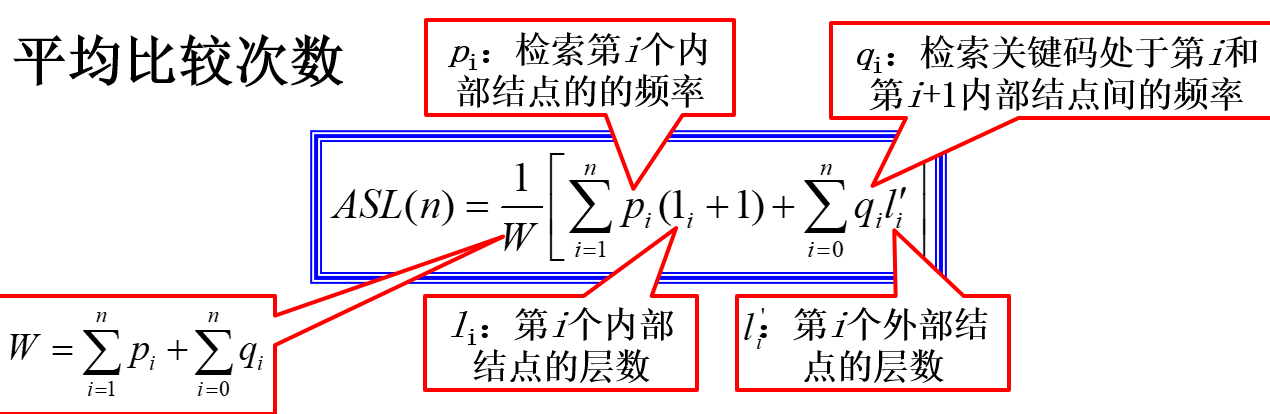
1. PATRICIA树（PAT-tree）

把整数存成二进制字符串，然后成为二叉trie树。

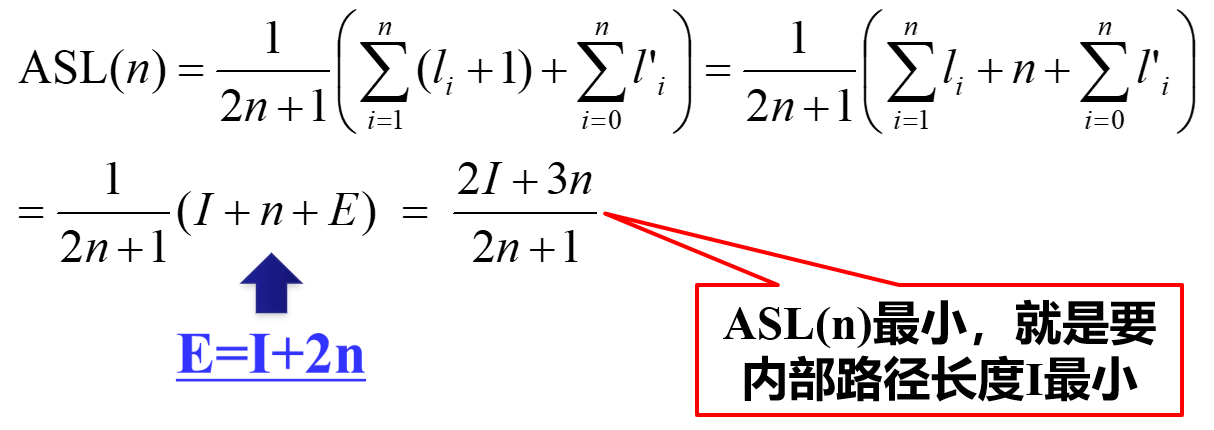
（也可压缩）

1. 最佳BST树

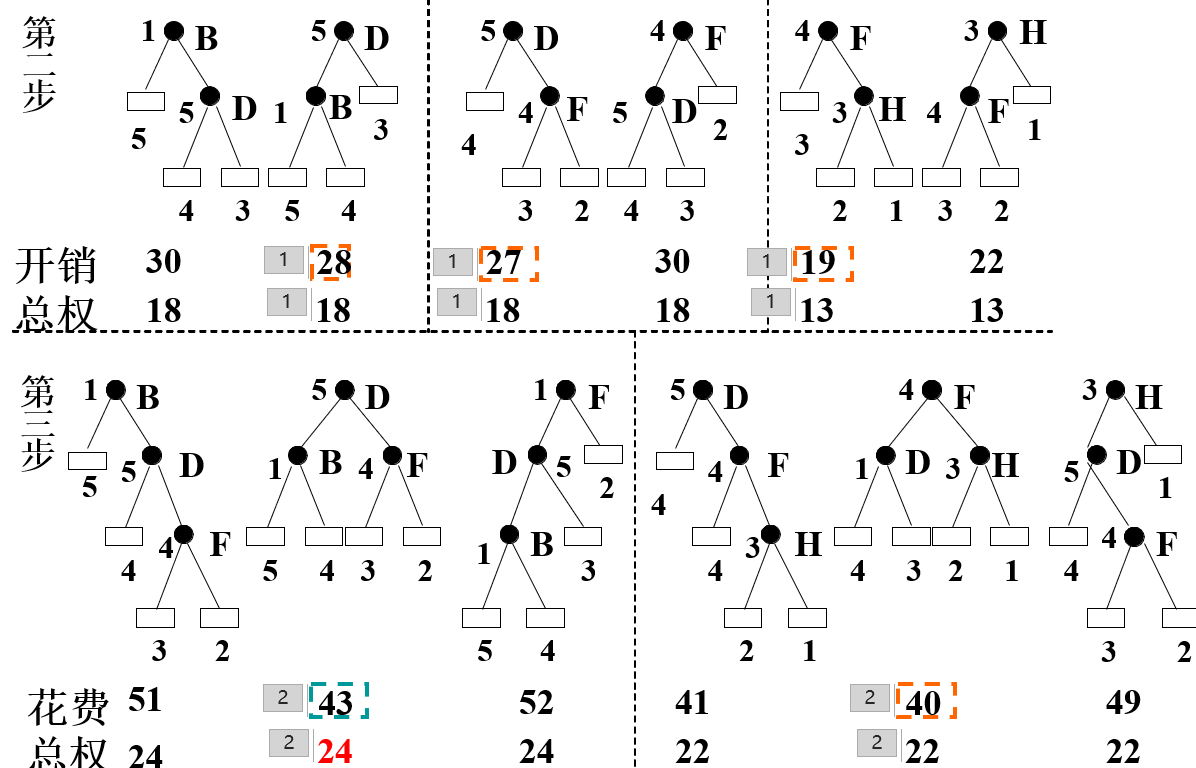
评判标准：



对于等概率访问：



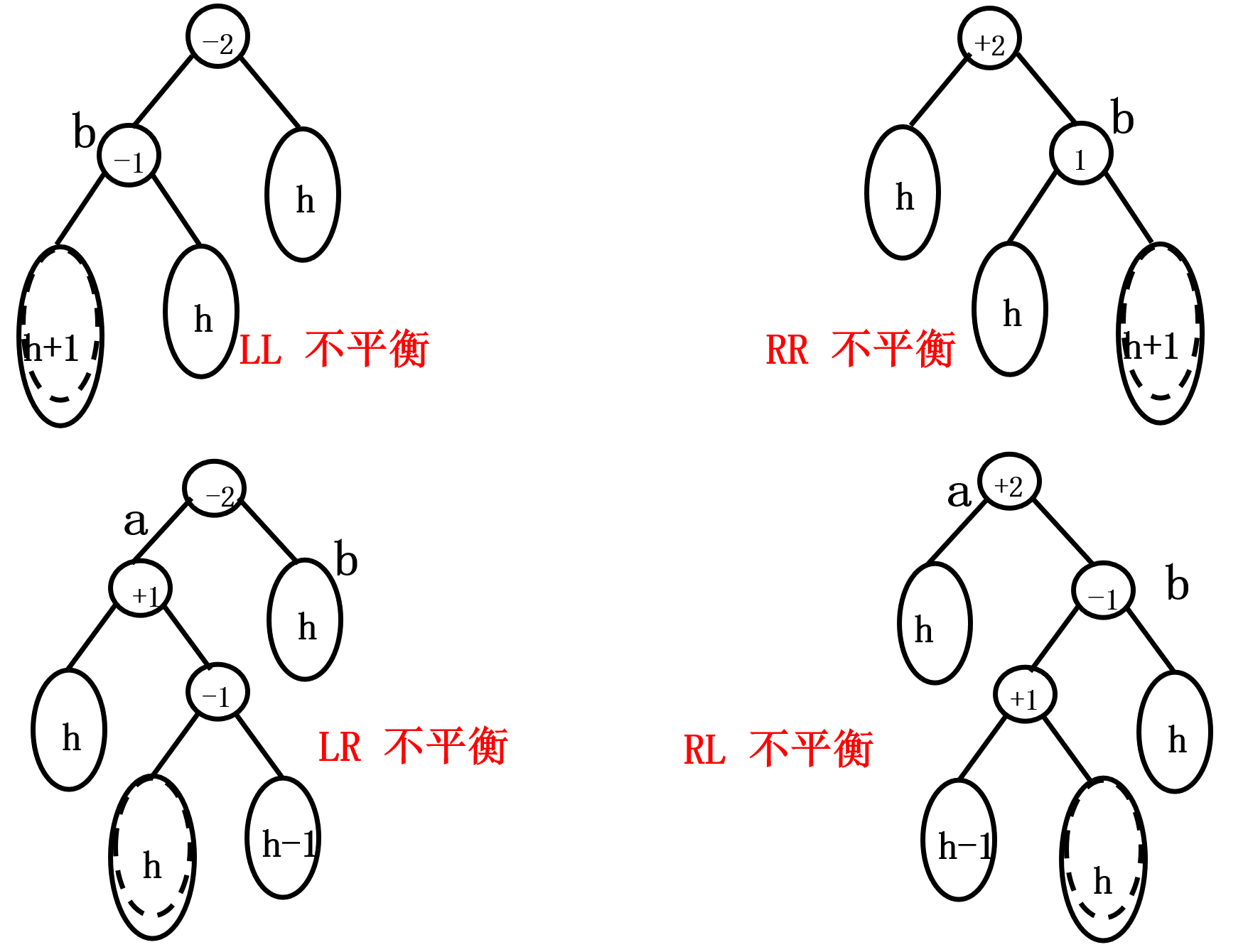
对于不等概率访问：可以从底层递归生成一棵最佳二叉搜索树（因为最佳二叉树任何子树都是最佳二叉搜索树）。



1. 平衡二叉搜索树（AVL）负责动态处理，不考虑访问频率

空二叉树是AVL树；若T是AVL树，则它的左右子树是AVL树，且左右子树高度最多相差1。（则树高整体是logn的）

插入（可能导致）：

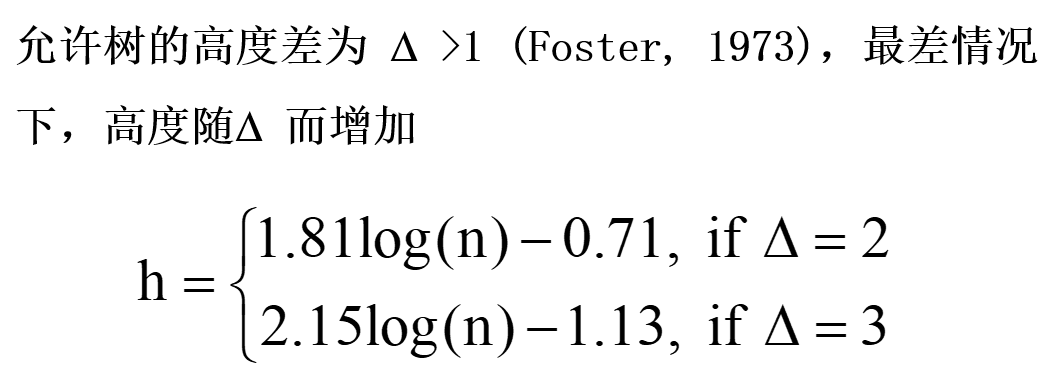


LL和RR需要一次旋转，LR和RL需要两次旋转。（均为常数操作）

建树：正常挨个儿插入。

删除：若子树高度被缩短，则需继续向上调整。

\*AVL树扩展：



1. 伸展树（Splaying Tree）

一种自组织的数据结构，数据随检索而调整位置。Eg. 汉字输入法的词表

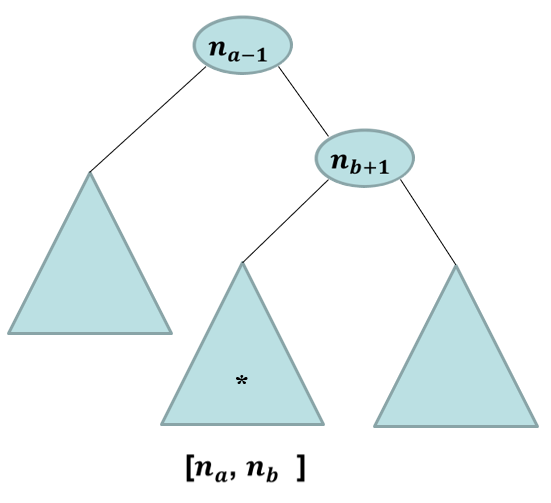
插入/检索x时，通过旋转把x移到根节点。

删除x时，把x的父节点移到根节点。

不能保证单个操作有效率（有可能是O(n)的），但能保证m次操作是O(mlogn)。

应用：区间删除

假设要删除区间[a, b]，则将a-1转到树根，b+1转到树根右儿子，则[a, b]成为b+1的左子树，直接删除即可。



区间插入、区间翻转同理。

1. 半伸展树

把父亲往上转一次，再把父亲的父亲往上转一次……（？）

可以比伸展树节省调整步骤。

