**1.基本概念**

**查找表(Search Table)**：**相同类型的**数据元素(或记录)组成的**集合**，每个**数据元素**通常**由若干数据项**构成

**关键字(Key，码)**：数据元素中某个(或几个)数据项的值，它可以标识一个数据元素

* + **主关键字(Primary Key)**：能唯一标识一个数据元素的关键字
  + **次关键字(Secondary Key)**：能标识若干个数据元素的关键字

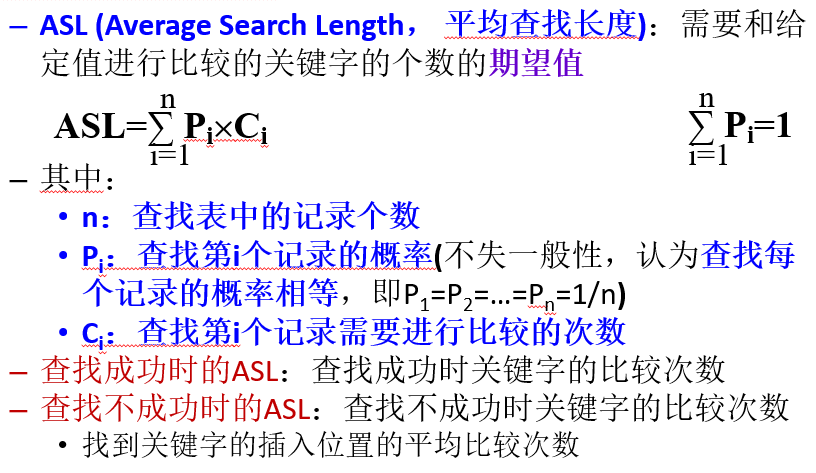
关键字通常是可比较的数值



**查找的分类：**

按存储结构的不同：**线性表查找、树表查找、哈希查找**

按查找过程是否在内存完成：内查找、外查找（查找过程需要访问外存）

按数据元素的数量是否变化：**静态查找/静态查找表、动态查找/动态查找表（表的结构本身是在查找过程中动态生成的）**

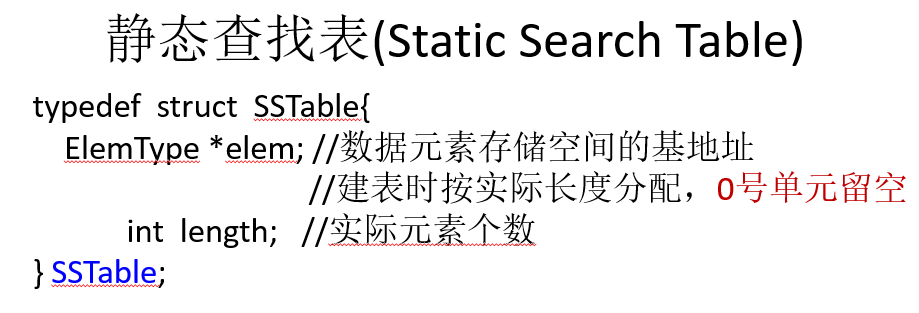
**查找方法的评价指标**

（1）查找算法的时间复杂度，所需的附加存储空间

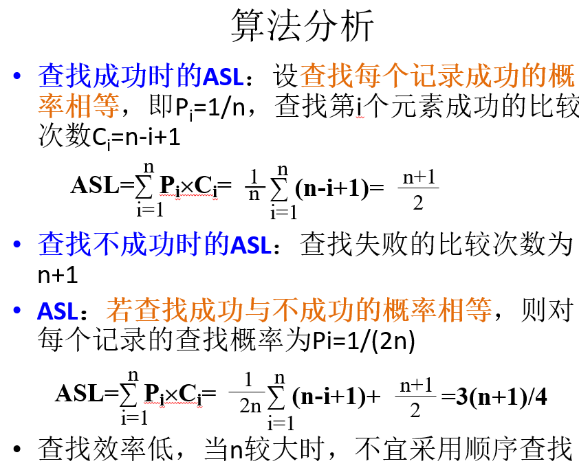
（2）**关键字的平均比较次数**

**2.静态查找表的查找方法**





**2.1 顺序表的查找：顺序查找**

逐个比较：

改进：按数据元素被查找的概率进行升序排序

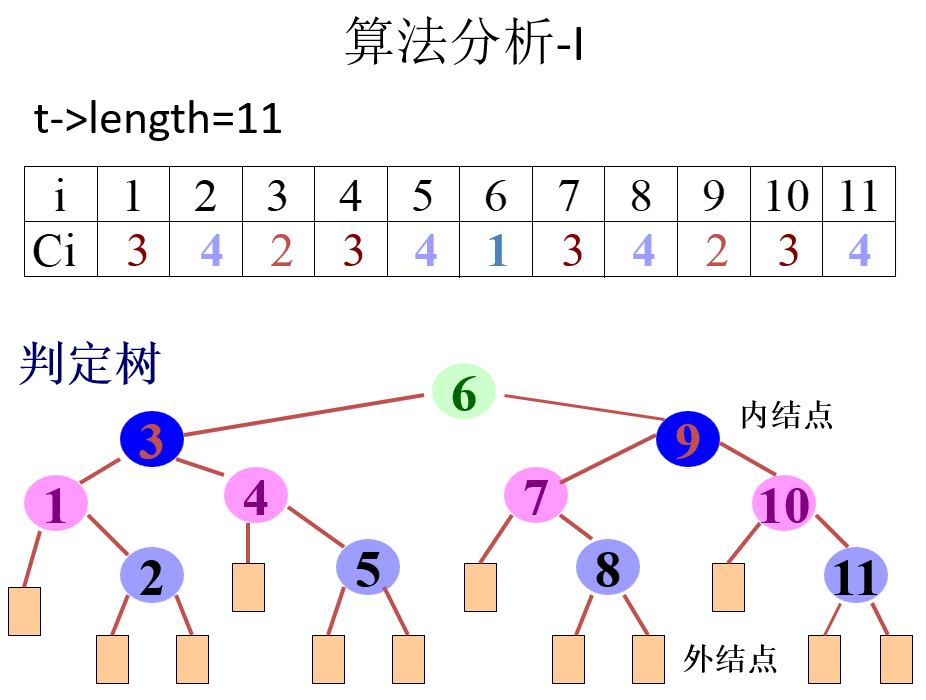
在数据元素中增加一数据项，用于记录对该数据元素的访问次数

在每次查找后，维护查找表使得其按照数据元素的访问次数升序排序

**2.2 有序顺序表的查找：折半查找**

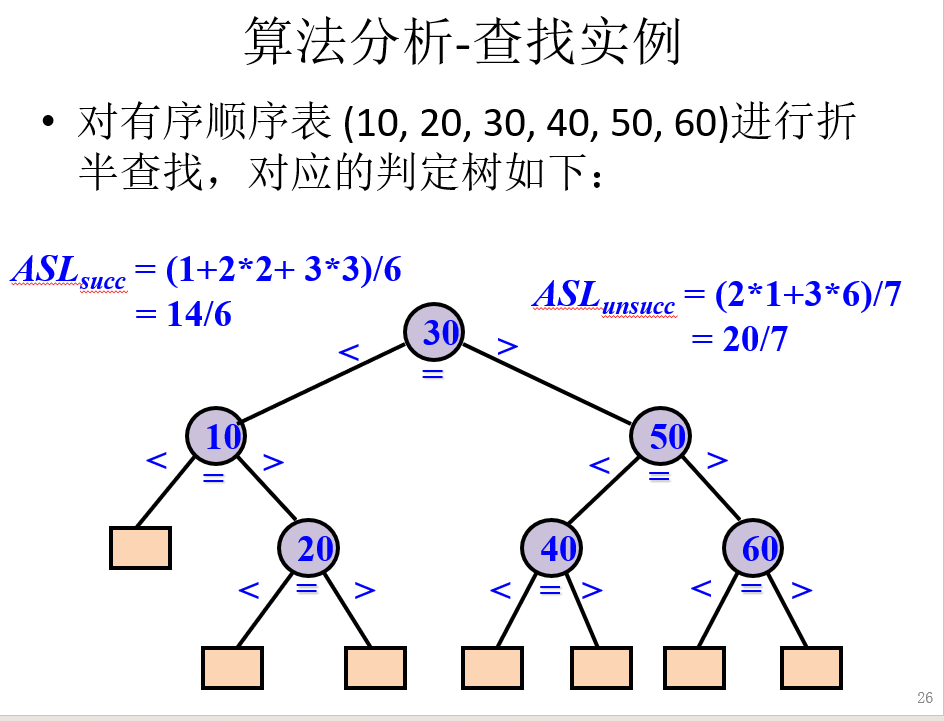


每个查找过程可以看作判定树**(Decision Tree)**上（到叶节点或外结点）的一条路径：

每经过一次比较，查找范围就缩小一半

当前查找区间的中点作为根结点

* + **内路径**：走到内节点的路径



折半查找优点：速度很快

局限：查找前需要对顺序表进行排序操作

缺点：1.无法应用于链表；

2.在不等概率查找的情况下不一定最好

3.当查找表的长度不大时，也许折半查找的效率不如顺序查找

**2.3 有序顺序表的查找：Fibonacci查找**

每次根据Fibonacci数列的特点对查找表进行分割



**2.4 索引顺序表的查找：分块查找**

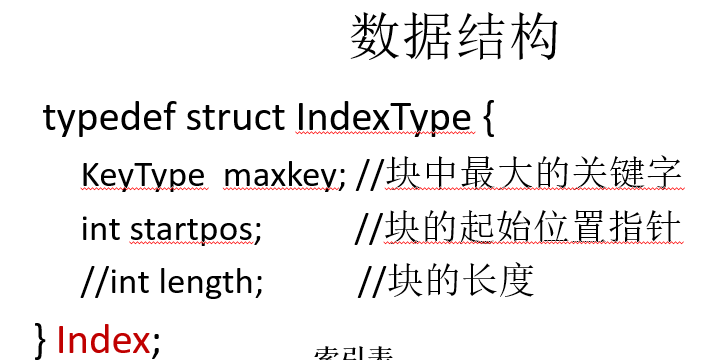
查找表的组织：将查找表分成几块

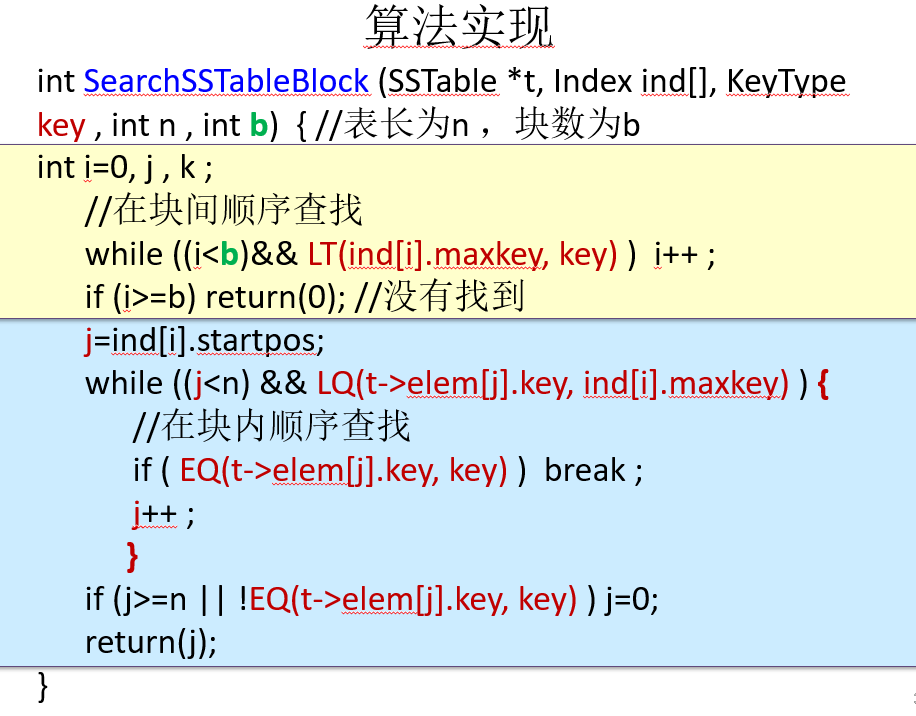
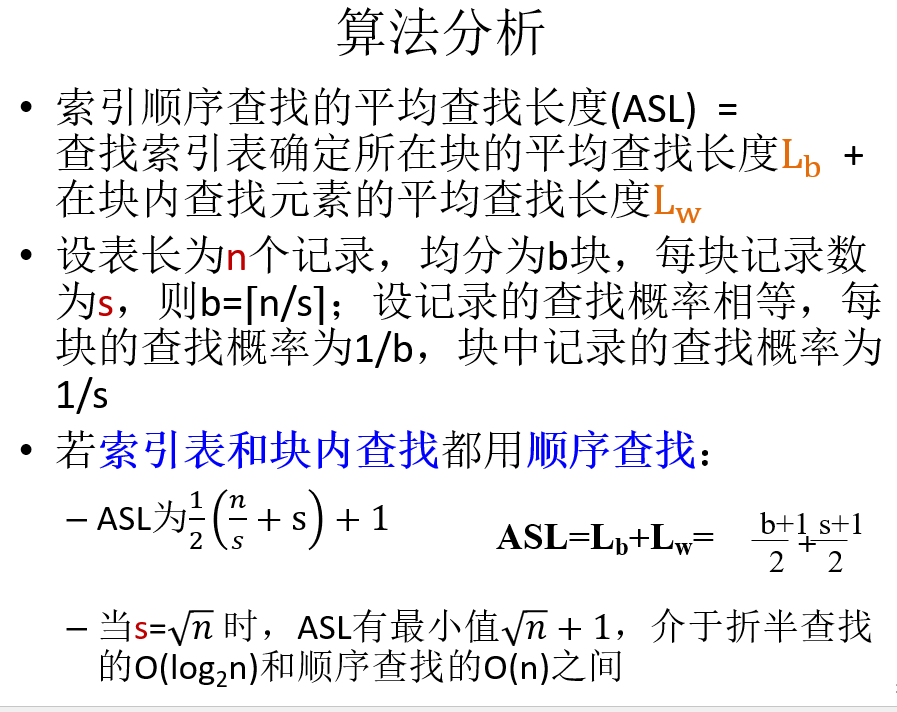
块间有序，即第i+1块的**所有记录关键字均大于(或小于)**第i块记录关键字

 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：

先(用顺序查找或折半查找)确定待查记录所在块，再在块内查找(顺序查找)



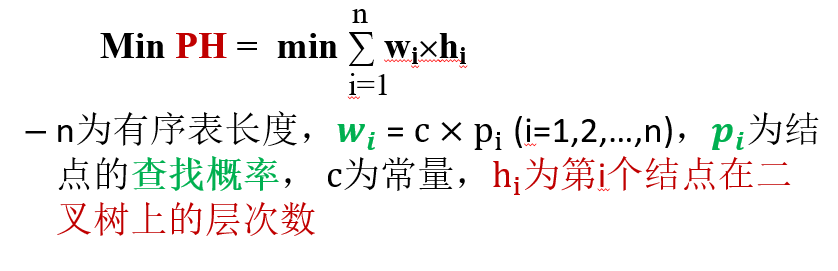


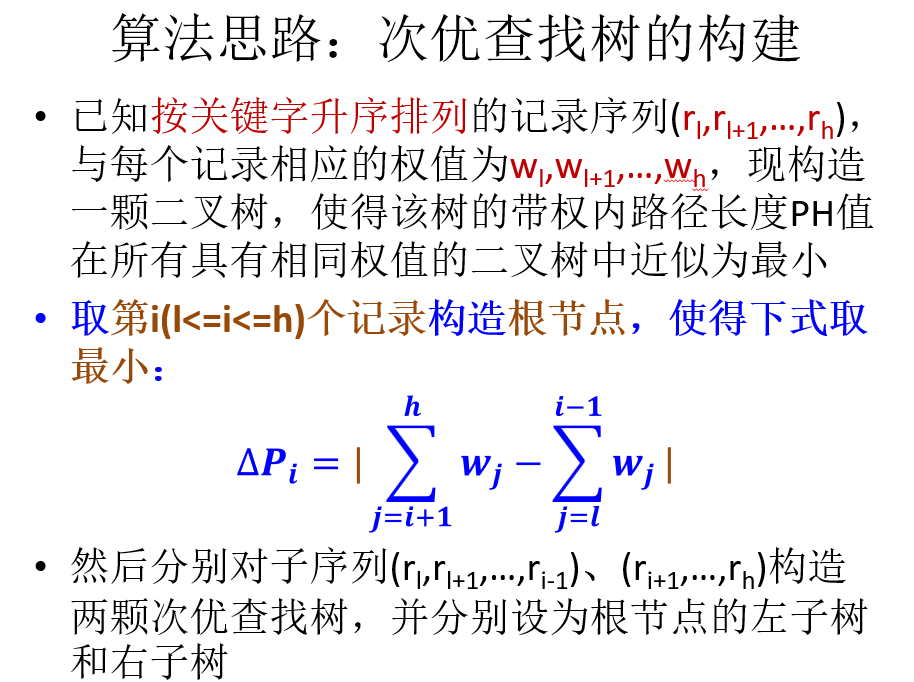


**2.5 静态树表：静态次优查找树的查找**

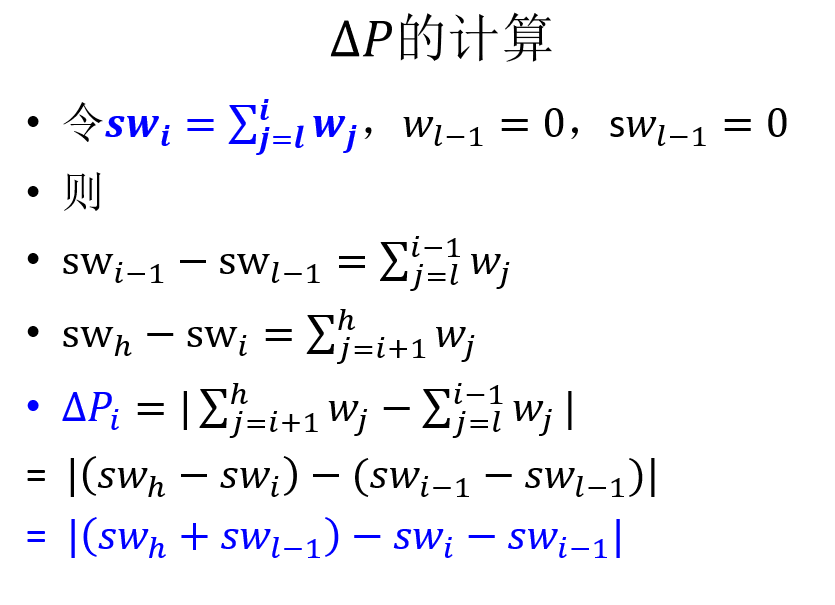
针对各个元素的查找概率不等的情况

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





pretask：构建部分和数组：sw[l]~sw[h]



注意：sw和w的0号位置不用！！！设置为0

**3. 动态查找表**

**3.1二叉排序树BST**

二叉排序树或者是空树，或者是满足下列性质的二叉树：

若左子树不为空，则左子树上所有结点的值(关键字)都小于根结点的值；

若右子树不为空，则右子树上所有结点的值(关键字)都大于根结点的值；

左、右子树都是二叉排序树

每个结点的Key互不相同

BST可以用二叉链表来存储

typedef struct RecType{

KeyType key; // 节点值，可用于比较大小

//Others

} ElemType;

typedef struct BiTreeNode {

ElemType data; // 数据

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

} \*BiTree;

若按中序遍历(LDR/RDL)一棵二叉排序树，所得到的结点序列是一个有序(递增/递减)序列

**BST的构造**

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**

**查找算法(+构造算法)**

**指针p指向当前访问的结点T；指针f指向T的双亲，其初始调用值为NULL**

BiTree SearchBST (BiTree T, KeyType key , BiTree f, BiTree p) {

if (!T) { p = f; return FALSE; } // 查找不成功，将p设置为指向其双亲结点

else if (EQ(key, T->data.key)) {p = T; return TRUE; } //查找成功 else if (LT(key, T->data.key))

return SearchBST(T->lchild, key); //在左子树中查找

else return SearchBST(T->rchild, key); //在右子树中查找

} // SearchBST

**插入算法**

Status InsertBST(BiTree T, ElemType e) {

BiTree p,s;

if (!SearchBST(T, e.key, NULL, p)) { //查找不成功

s = (BiTree)malloc(sizeof(BiTreeNode));

s->data = e; s->lchild = s->rchild = NULL;

if (!p) T = s; // 插入 s 为新的根结点

else if (LT(e.key, p->data.key)) p->lchild=s; //插入s为左孩子

else p->rchild=s; //插入 s 为右孩子

return TRUE;

} else return FALSE; // 树中已有关键字相同的结点，不再插入

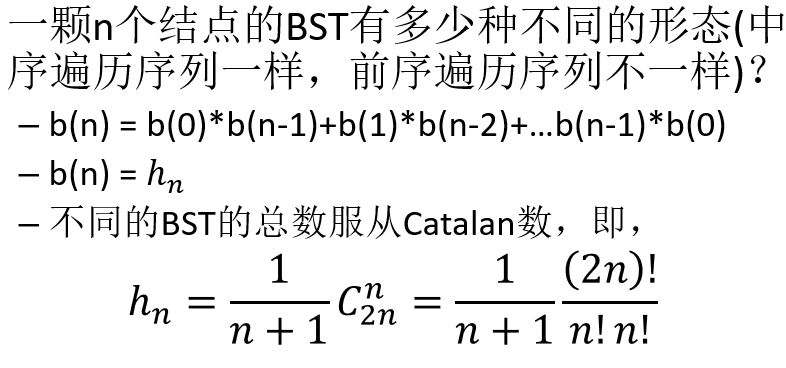
} // Insert BST

**构造出的树的形态与BST的插入顺序相关，并直接影响ASL**

**BST的性质**

一颗有n个结点(n>=0)的BST的最小高度为⌈log2 (𝑛+1)⌉(丰满树)，最大高度为n(单支树)

BST的最小元素一定没有左子女，最大元素一定没有右子女



**BST上结点的删除**

设被删除结点为p，其父结点为f

(1)若p是叶子结点：直接删除p

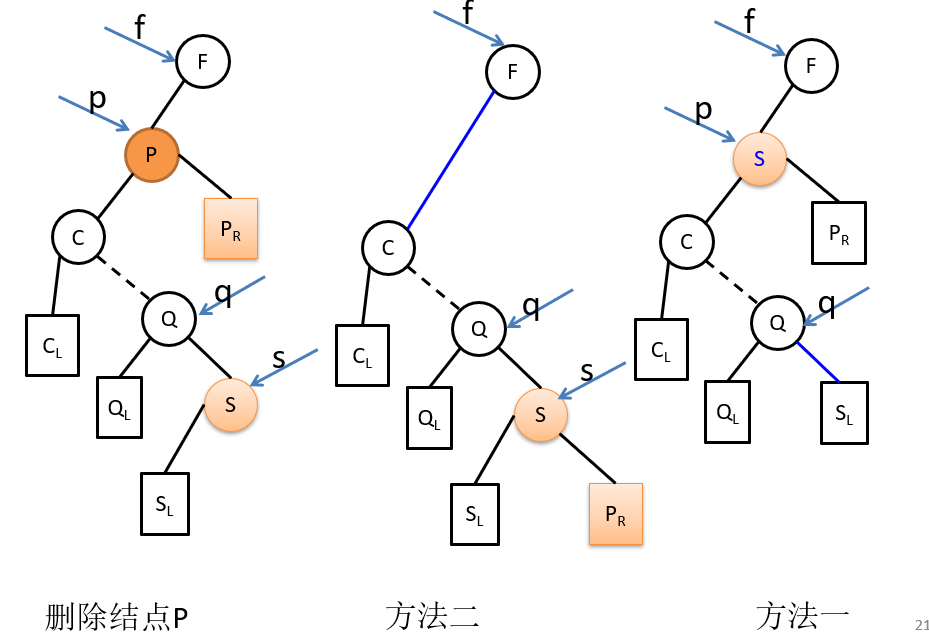
(2)若p只有一棵子树(左子树或右子树) ：直接用p的左子树(或右子树)取代p的位置而成为f的一棵子树

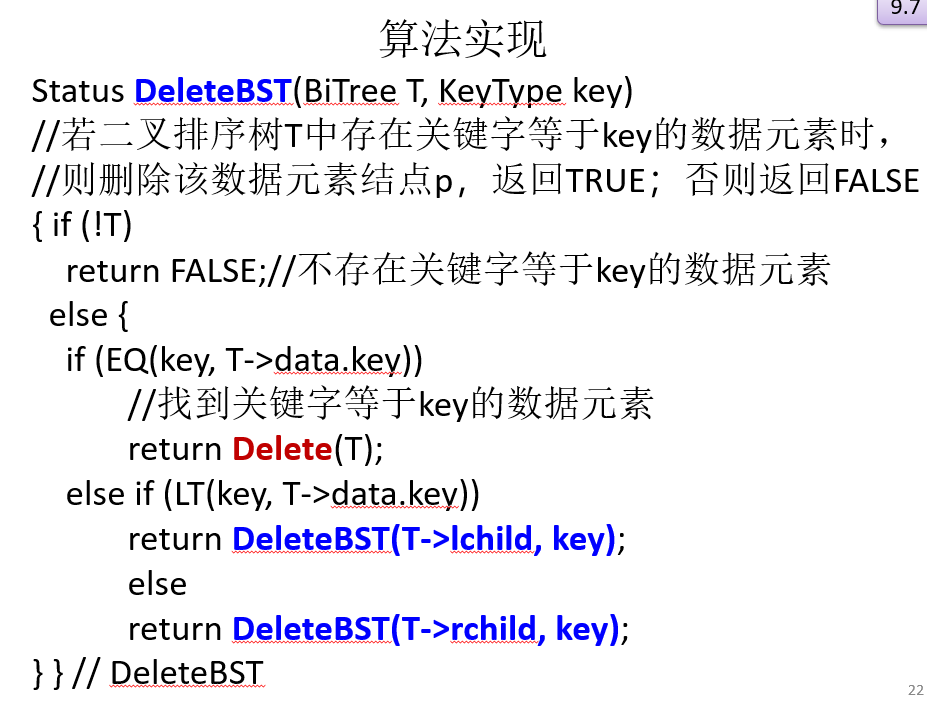
(3)若p既有左子树又有右子树：下面方法改用（中序遍历的）直接后继s也可

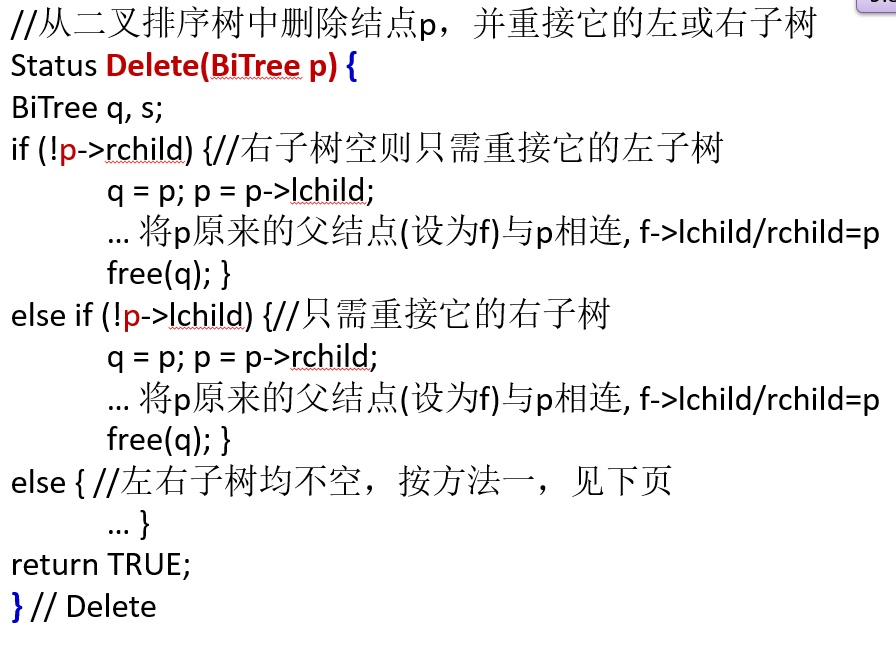
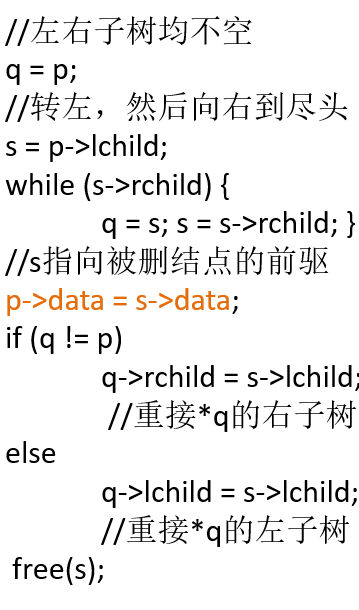
方法一：用p的直接前驱结点s代替p，即从p的左子树中选择值最大的结点s放在p的位置(用结点s的**内容替换**结点p内容)，然后**删除结点s**，**并将s的左子树作为s的父结点的右子树**

注：s是p的左子树中的最右边的结点且没有右子树

方法二：找到s，s是p的左子树中的最右边的结点且没有右子树，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**(也就是将p的左子树为f的左子树)







**BST的查找性能分析**

二叉排序树上成功的查找次数不会超过二叉树的深度，而具有n个结点的二叉排序树的深度，最好为⌈log\_2⁡〖(𝑛+1)〗 ⌉，最坏为n。因此，二叉排序树查找的最好时间复杂度为O(log2n)，最坏的时间复杂度为O(n)

ASL=

**3.2 平衡二叉树(AVL树)**

平衡二叉树(Balanced Binary Tree) : 或者是空树，或者是满足下列性质的二叉树

—左子树和右子树深度之差的绝对值不大于1

—左子树和右子树也都是平衡二叉树

平衡二叉排序树(Balanced Binary Sort Tree)：一棵二叉树既是二叉排序树又是平衡二叉树

**结点的平衡因子**(Balance Factor) ：该结点的左子树的深度减右子树深度，只能是0,1,-1

AVL树最终显示的形态即为不要求缺失结点位置的完全二叉树

typedef struct BSTNode {

ElemType data;

int bf; //平衡因子

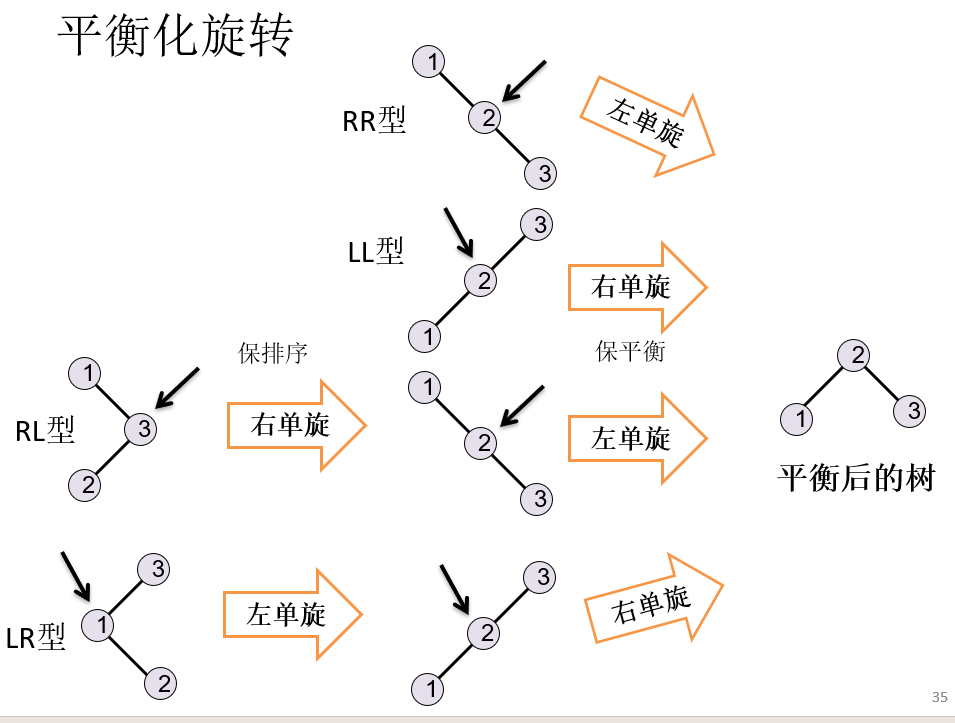
struct BSTNode \*lchild , \*rchild;

} BSTNode, \*BSTree;

平衡化旋转

每插入一个新结点时（按排序树方式插入），AVL 树中相关结点的平衡状态会发生改变。因此，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，沿刚才回溯的路径取直接下两层的结点，进行平衡化旋转



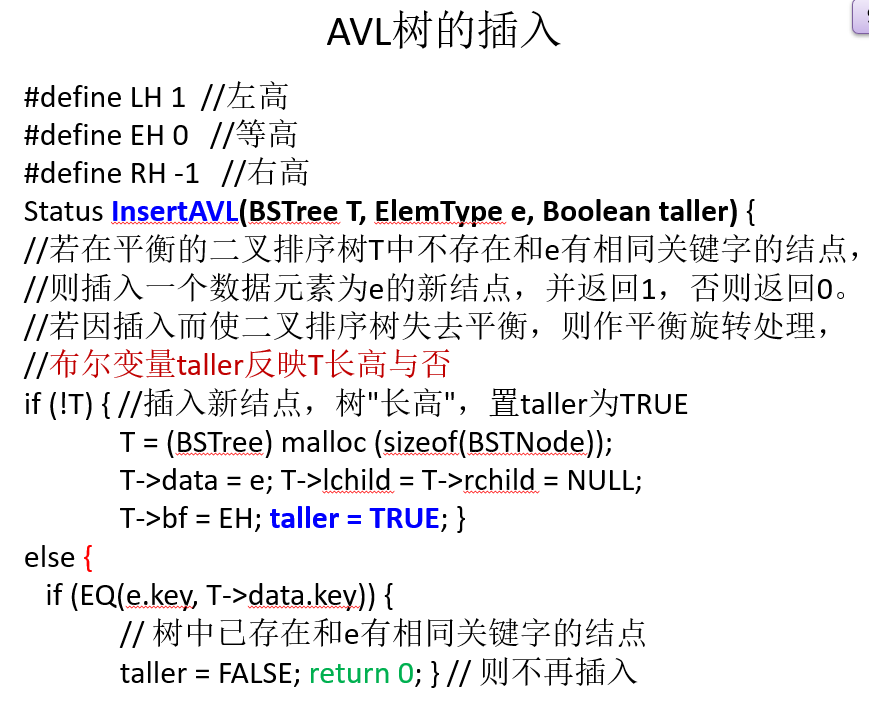
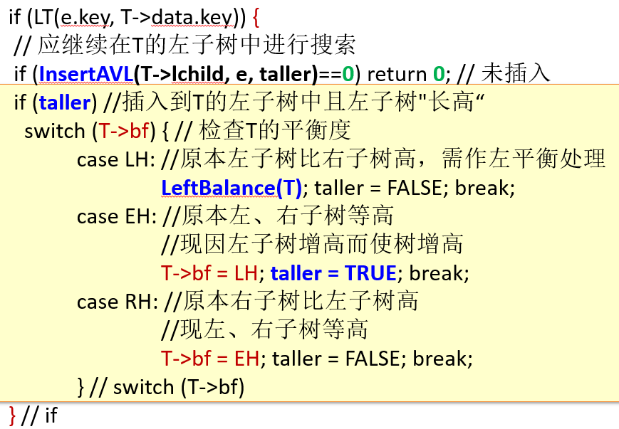
结点插入对平衡因子的影响

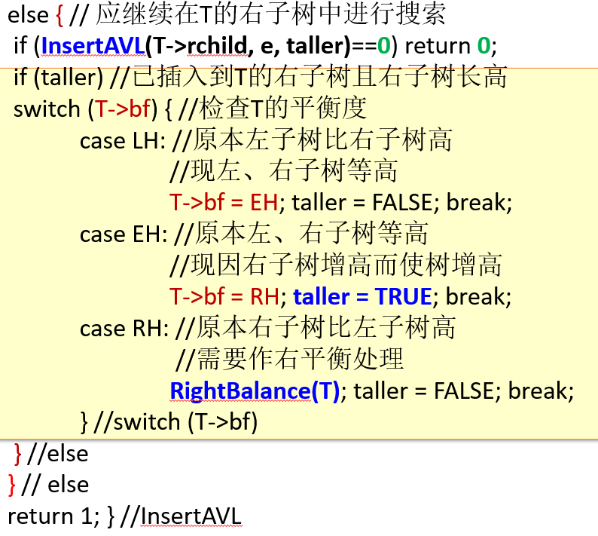
设新结点p的平衡因子为0，其父结点为pr，更新pr的bf值

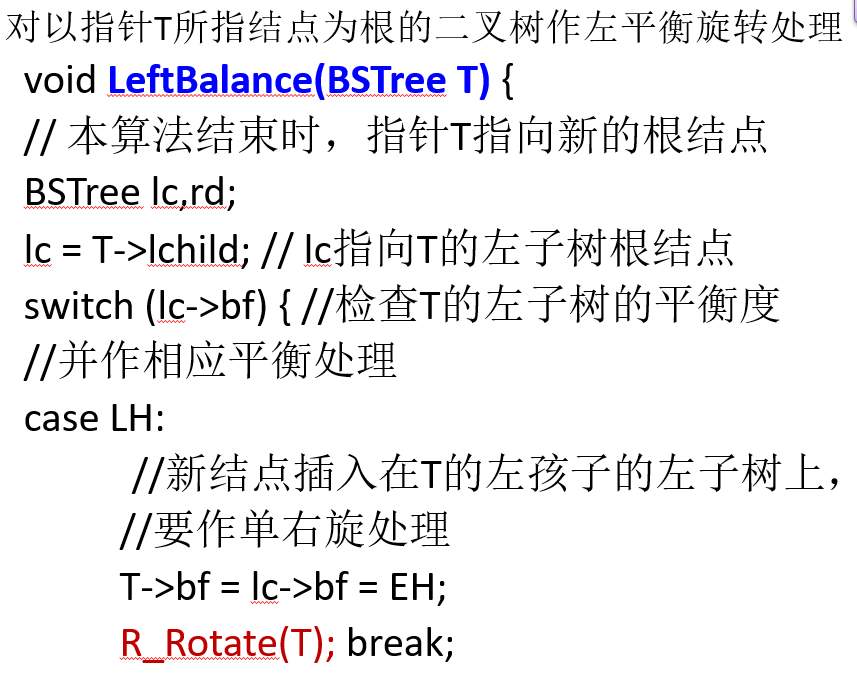
（1）结点pr的平衡因子变为0，退出

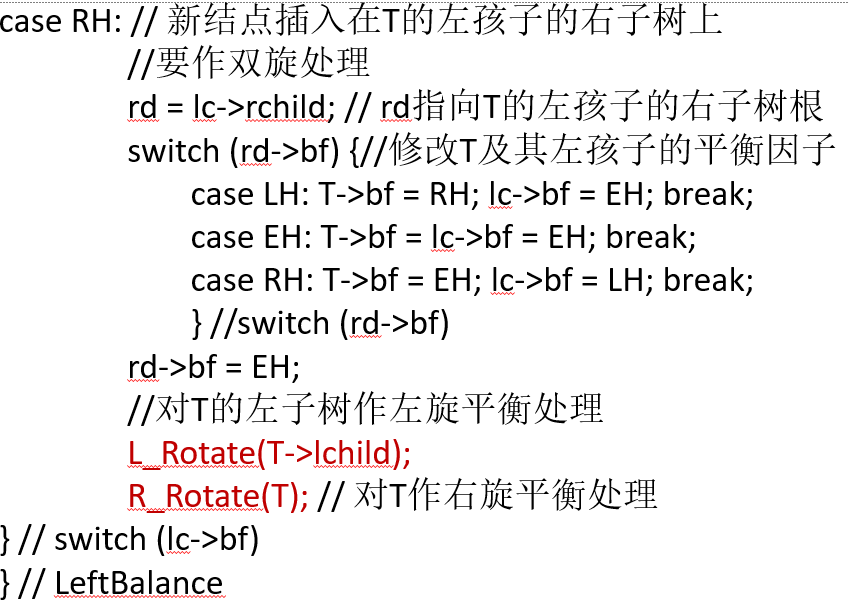
（2）结点pr的平衡因子的绝对值|bf| = 1，不需平衡化旋转，但需从结点pr向根方向回溯

（3）结点pr的平衡因子的绝对值|bf| = 2，需要做平衡化旋转









**AVL树的删除**

（1）被删结点x最多只有一个子女：将结点x从树中删去；把x的双亲中原来指向x的指针改指到x的这个子女结点（或NULL）；将原来以结点x为根的子树的高度减1

（2）被删结点 x 有两个子女：搜索 x 在中序次序下的直接前驱 y，把结点 y 的内容传送给结点 x，现在问题转移到删除结点 y。把结点 y 当作被删结点x，因为结点 y 最多有一个子女，可以简单地用 前一页给出的方法进行删除。沿结点 x 通向根的路径反向追踪高度的变化对路径上各个结点的影响

**AVL上的查找**

深度为 h 的AVL树的最小结点数：𝑁ℎ = 𝐹ℎ+2 -1

有 n 个结点的AVL树的深度不超过𝐥𝐨𝐠𝝋(√𝟓 (𝒏+𝟏))−𝟐

𝐹ℎ为Fibonacci数列，约为𝜑ℎ/√5，其中𝝋=(𝟏+√𝟓)/𝟐

**3.3 B树和B+树**

用于内、外存之间数据交换。一种能尽可能降低磁盘I/O次数的索引组织方式：树结点的大小尽可能地接近磁盘页的大小

**m阶B树(a balanced tree of order m)**

或者是空树，或者是满足以下性质的m叉树：

1.每个结点至多有m棵子树

2.**根结点或者是叶子结点，或者至少有两棵子树**

3.除根结点外，所有非终端结点至少有⌈𝒎∕𝟐⌉棵子树

4.所有非终端结点应包含如下信息

(n，A0，K1，A1，K2，A2，… ，Kn，An)

n是结点中关键字的个数，且⌈𝑚∕2⌉ -1≤ n ≤ m-1，n+1为子树的棵数

Ki(1≤i≤n)是关键字，且Ki<Ki+1 (1≤i≤n-1)；

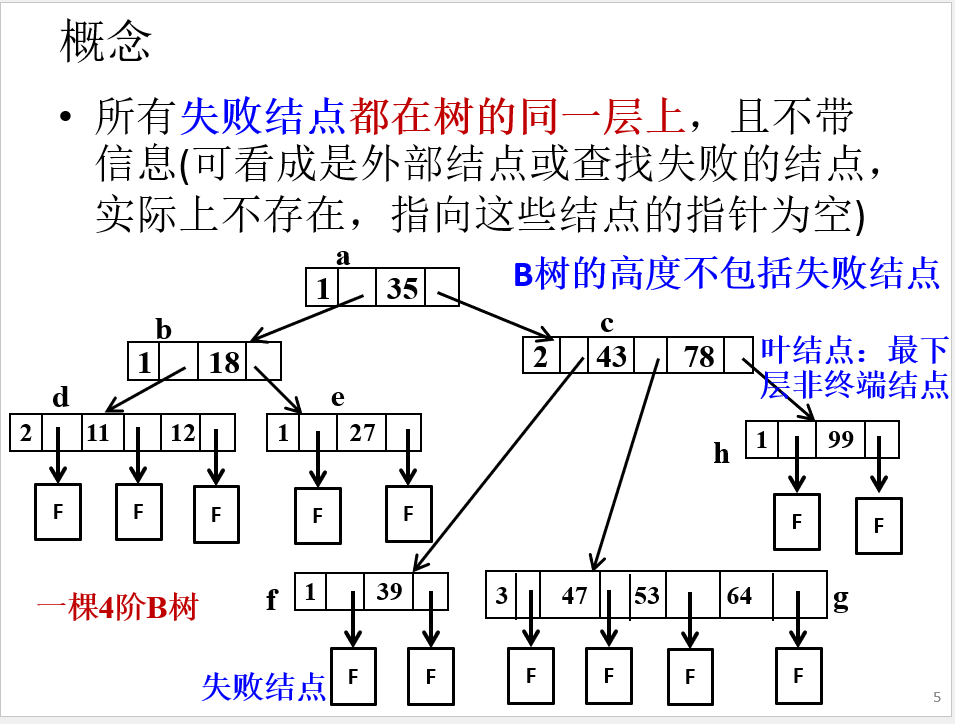
Ai (i=0，1，… ，n)为指向孩子结点的指针，且Ai-1所指向的子树中所有结点的 关键字都小于Ki ，Ai所指向的子树中所有结点的关键字都大于Ki

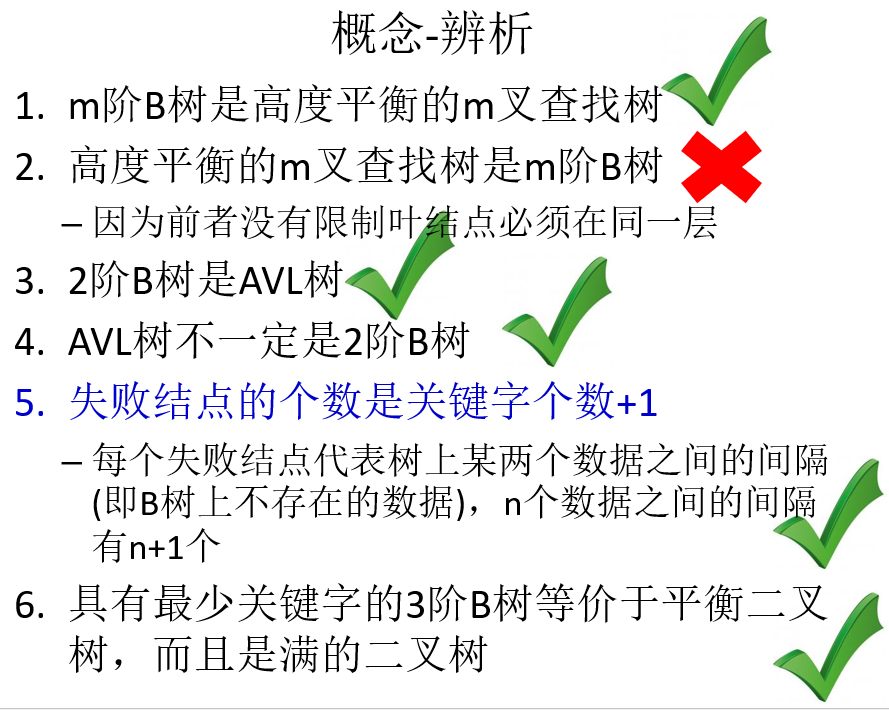
非叶节点的结构：关键字数n，指针0，关键字1，指针1…，指针n-1，关键字n，指针n；

至少⌈𝑚∕2⌉个子树：n+1≥⌈𝑚∕2⌉；

最多m个子树：n+1≤m ，最多有m-1个关键字

5. 所有失败结点都在树的同一层上，且不带信息（实际上不存在，指向这些结点的指 针为空）B树的高度不包括失败结点。





**m阶B树的性质**

**m阶B树实现**

typedef struct BTNode {

int keynum; //结点中关键字个数，即结点大小

struct BTNode \*parent; //指向父结点的指针

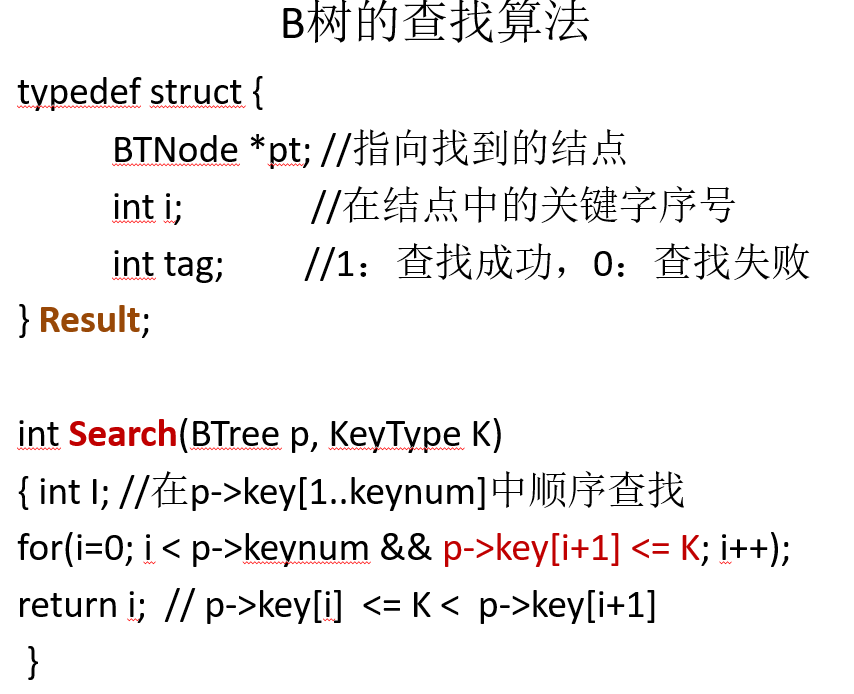
KeyType key[m+1]; //关键字，0号单元不用

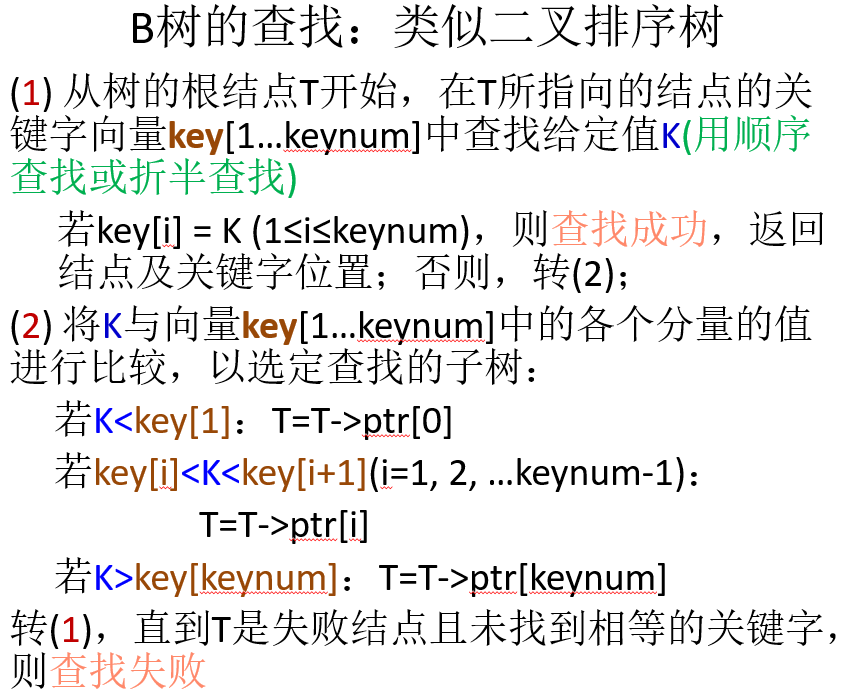
//Record \*recptr[m+1]; //记录指针向量

//0号单元不用

struct BTNode \*ptr[m+1]; //子树指针向量

} BTNode, \*BTree; //B树结点和B树的类型





**1. 查找（Search）**

过程类似于多叉查找：

* 从根结点开始，
* 在当前结点中二分查找合适范围（判断在哪两个关键字之间），
* 然后进入对应的子树递归查找，
* 最终找到或失败。

时间复杂度：

* 每层查找时间：O(log m)（用二分）
* 总层数：O(logₘ n)  
  ⇒ 总复杂度：**O(logₘ n)**，非常适合磁盘读写优化。

**2. 插入（Insert）**

步骤：

1. 按查找方式找到应该插入的位置；（查找失败的叶节点）
2. 插入后，如果关键字个数超过最大值（m−1），则发生**结点分裂**；
3. 分裂将中间关键字提升到父结点，左右两边形成两个新结点；
4. 若父结点也满了，继续向上分裂，可能一直传递到根；
5. 若根也分裂，则树高加 1。

**3. 删除（Delete）**

**注意移动时指针也跟着关键字移动!**

**情况分类：**

删除的是**非叶节点**：找到**前驱/后继**关键字替代，再从叶子节点中删除该替代关键字。

前驱：左子树最右边的结点； 后继：右子树最左边的结点

删除叶子节点：

若结点N中的关键字个数>⌈𝑚∕2⌉ -1：在结点中直接删除关键字K。

若结点N中的关键字个数=⌈𝑚∕2⌉ -1：

若结点N的左(右)兄弟结点中的关键字个 数>⌈𝑚∕2⌉-1，则将结点N的左(或右)兄 弟结点中的最大(或最小)关键字上移到其父结点 中，而父结点中大于(或小于)且 紧靠上移关键字的关键字下移到结点N

若结点N和其兄弟结点中的关键字数=⌈𝑚∕2⌉-1，删除N中的该关键字

(1) 将结点N、其兄弟结点q以及父结点中分割二 者的关键字Ki(N<Ki<q)，合并 为一个结点，合并后的结点有(⌈𝑚∕2⌉-2)+(⌈𝑚∕2⌉-1)+1 =2⌈𝑚∕2⌉-2 个关键字(替代结 点q的位置)

(2) 删除结点N

(3) 若父结点中的关键字个数不足，

如果父节点是根节点：将其删除

如果父节点是非根节点且只有⌈𝑚∕2⌉-2个关键字：该父结点与其最近邻兄弟、 以及父父节点进行(1)；该合并操作可以沿B树一直向上进行

**3.4 键树**

Keyword Tree/Digital Search Tree是关键字(**字符串/数字串**)的一种组织方式

（1）树的结点包含组成关键字的符号：**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。键树的深度和关键字集合的大小无关，取决于关键字中字符或数位的个数。

（2）度大于2的树：每个结点的最大度与关键字的“基”有关

（3）键树被约定为是一棵有序树，即**同一层中兄弟结点之间依所含符号自左至右有序**，并约定结束符‘$’小于任何其它符号

两种结点：叶子和分支

typedef enum { LEAF, BRANCH } NodeKind;

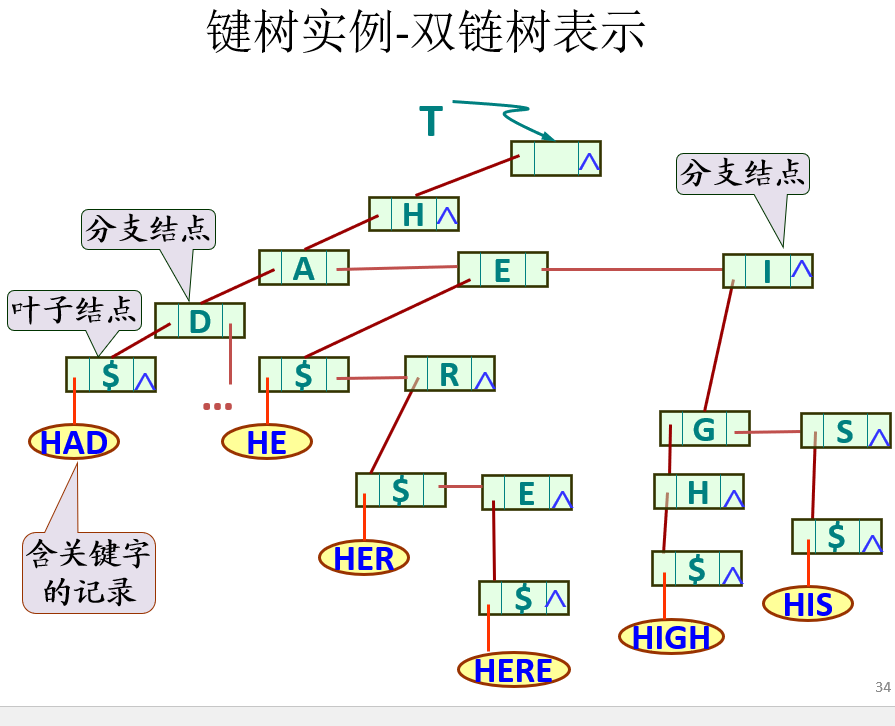
孩子兄弟表示法

typedef struct DLTNode {

NodeKind kind;

char symbol;

struct DLTNode \*next; //指向兄弟结点的指针

 union {

//叶子结点内的记录指针

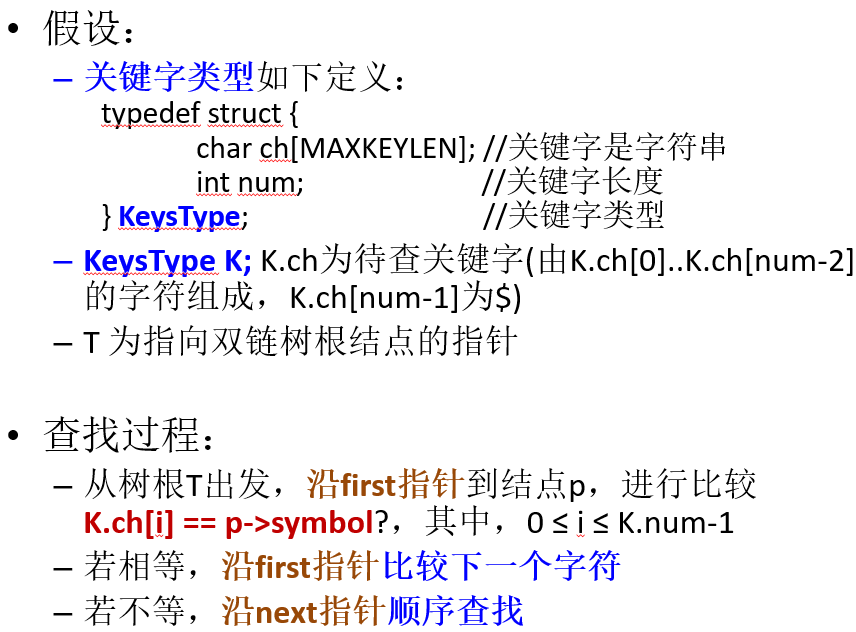
Record \*infoptr;

//分支结点内的孩子链指针

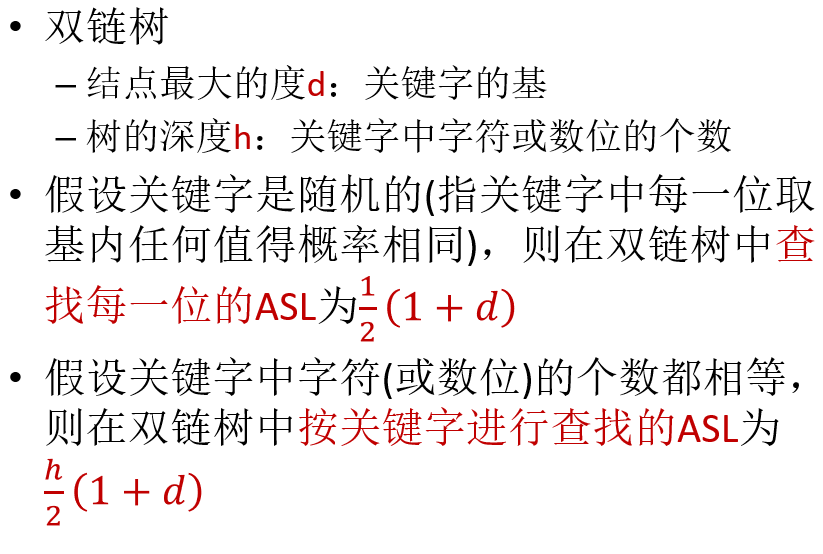
struct DLTNode \*first;

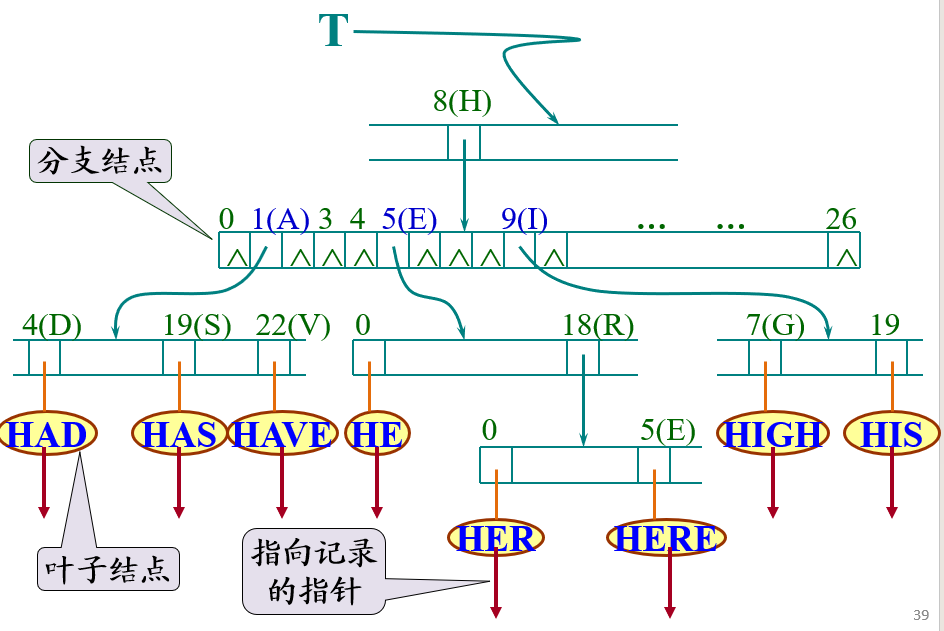
}

**在双链树中查找记录**



**双链树的查找性能**



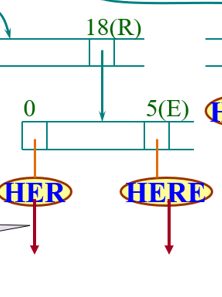
**实际实现：Trie树表示**

typedef struct TrieNode {

NodeKind kind; //结点类型

union {

//叶子结点

 struct { KeysType K; Record \*infoptr } lf;

//分支结点(27个指向下一层结点的指针)

struct { TrieNode \*ptr[27]; int num } bh; //0号位置指向不拓展字母的记录， 其他位置分别拓展26个字母

};

} TrieNode, \*TrieTree; // 键树类型

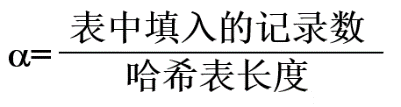
建立规则（插入算法）

对于一个输入字符串，循环处理，为每个字符生成一个分支节点。当处理到最后一个字符时：如果曾经没有创建过该字符的分支节点（即倒数第二层分支节点的对应ptr==NULL），则先创建一个分支节点，再创建一个叶子节点挂载到分支节点的ptr[0]上；如果曾创建过该字符的分支节点，字需要在该层分支节点的ptr[0]处创建一个叶子节点存储该字符串key。

删除算法

**4. 哈希表**

表长习惯设置为素数

哈希函数H：在记录的关键字与记录的存储地址之间建立了一种确定的对应关系

哈希表的装填因子α：

冲突(collision)：对于关键字k\_i、k\_j，若k\_i ≠ k\_j，但H(k\_i)=H(k\_j) 的现象叫哈希冲突

同义词：具有相同哈希函数值的两个不同的关键字，称为该哈希函数的同义词

**哈希表设计要素**

1. 确定哈希函数的定义域和值域

2. 构造合适的哈希函数：单值函数、均匀映射到值域（使冲突尽可能少）

3. 给出处理冲突的方法

**哈希函数的构造**

(1)直接定址法：取关键字或关键字的某个线性函数作哈希地址

特点：直接定址法所得地址集合与关键字集合大小相等，不会发生冲突，但实际中很少使用

(2)数字分析法：若关键字为以r为基的数，取关键字的若干位或组合作为哈希地址

特点：适用于关键字位数比哈希地址位数大，且事先知道可能出现的关键字的情况(如频度)

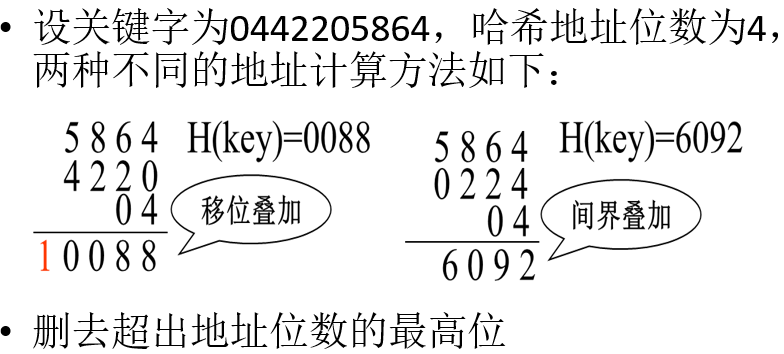
(3)平方取中法：将关键字平方后取中间几位作为哈希地址

特点：一个数平方后中间几位和数的每一位都有关，则由随机分布的关键字得到的哈希地址也是随机的。适用于：不知道全部关键字情况；适用于：关键字中的每一位都有某些数字重复出现

(4)折叠法(folding)：将关键字分割成位数相同的几部分(最后一部分可以不同)，然后取这几部分的叠加**和**作为哈希地址

移位叠加：将分割后的几部分低位对齐相加

间界叠加：从一端到另一端沿分割界来回折迭，然后对齐相加



(5)除留余数法：取关键字被某个不大于哈希表表长m、但最接近于或等于m的质数p除后所得余数作哈希地址，即：H(key) = key MOD p (p≤m)

或 H(key) =(a\*key+b) MOD p (a>0, b>0, a MOD p !=0, p 为素数)

特点：简单、常用的哈希函数构造方法

(6) 随机数法：取关键字的随机函数值作哈希地址，即H(key)=random(key)

当哈希表中关键字长度不等时，该方法比较合适

**冲突处理的方法**

(1)开放定址法

当冲突发生时，形成某个探测序列；按此序列逐个探测哈希表中的其他地址，直到为给定的关键字找到一个空地址(开放的地址)为止

哈希地址的计算公式是：

H0(key) = H(key)

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

di是第i次探测时的增量序列

①线性探测法(Linear Probing)

增量序列为：di=1, 2, 3, …, m-1

优点：只要哈希表未满，总能找到一个不冲突的哈希地址

缺点：二次聚集(两个第一次哈希地址不同的记录争夺同一个后继哈希地址)

②二次探测法(Quadratic Probing)

增量序列为：di=1², -1², 2², -2², 3², ……+k², - k² (k≤⌊m/2⌋)

优点：探测序列**跳跃式地哈希到整个表中**，不易产生冲突的聚集现象

缺点：不能保证探测到哈希表的所有地址

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,…)时，可以保证查找链的前m项均互异**

③伪随机探测法

用伪随机函数获得伪随机数列，即di是一个随机数

(2)多哈希法/双哈希法

构造若干个哈希函数

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

优点：不易产生冲突的聚集现象

缺点：计算时间增加

(3)链地址法

将所有关键字为同义词(哈希地址相同)的记录存储在一个单链表中，并用一维数组存放链表的头指针

优点：不易产生冲突的聚集；删除记录也很简单

(4)建立公共溢出区

在基本哈希表之外，另外设立一个溢出表保存与基本表中记录冲突的所有记录

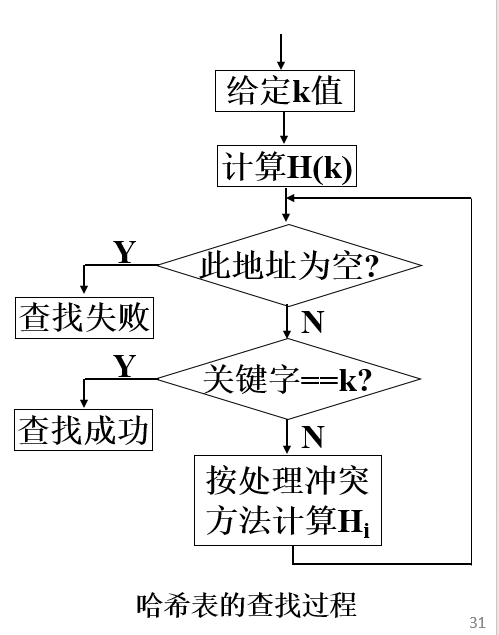
基本哈希表Hashtable[m]，每个分量保存一个记录；

溢出表Overtable[m]：一旦某个记录的哈希地址发生冲突，填入溢出表当前空位中

（用于冲突较少的情况）

哈希查找过程

通过检测当前位置是否为空。决定是否要按冲突处理方法生成下一个哈希地址



**开放定址哈希表的结构**

int hashsize[] = { 997, ... };

typedef struct {

HElemType \*elem; //HElemType 中含key

int count; //当前数据元素个数

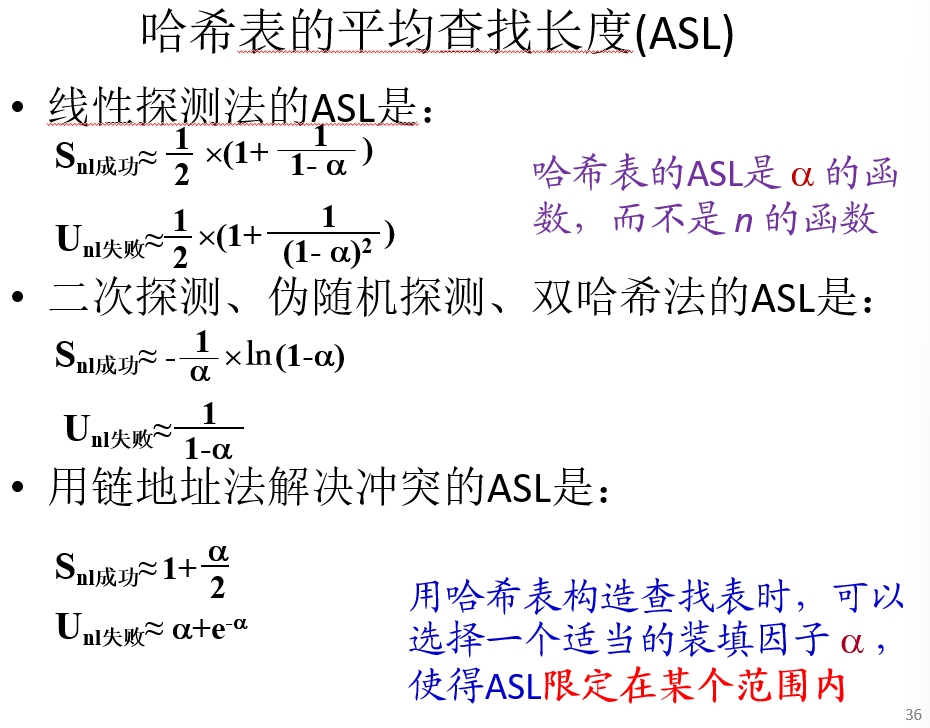
int sizeindex; //hashsize[sizeindex]为当前容量

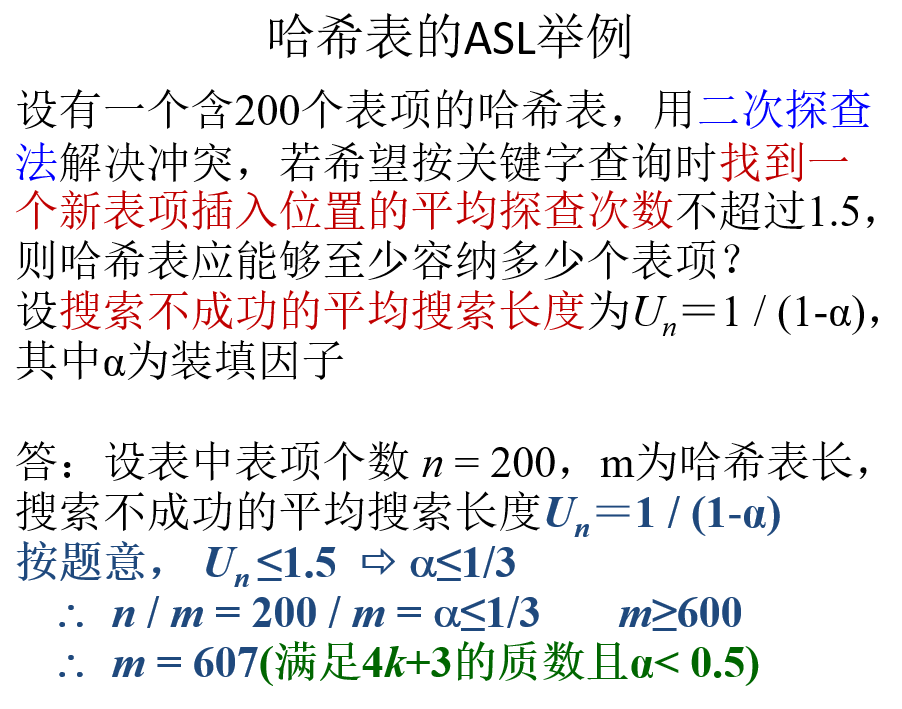
} HashTable;

#define SUCCESS 1

#define UNSUCCESS 0

#define DUPLICATE -1





注意表长要是质数！！