



数据结构与算法

Data Structure and Algorithm

第9章 查找



目 录

9.1 查找的基本概念

9. 2 静态查找表

9. 3 动态查找表

9. 4哈希表

9.1 查找的基本概念

(1) 关键字 主关键字

关键字是数据元素中某个数据项的值，可以标志一个数据元素。

主关键字是可唯一标志一个数据元素的某个数据项的值

学 号	姓 名	性 别	出生日期			来 源	总分	录取专业
			年	月	日			
20010983	赵剑平	男	1982	11	05	石家庄一中	593	计算机
20010984	蒋伟峰	男	1982	09	12	保定三中	601	计算机
20010985	郭 娜	女	1983	01	25	易县中学	598	计算机
20010989	郭 娜	女	1982	05	2	昌平一中	578	计算机



(2) 查找

也称为检索，就是根据给定的值，在一个表中查找出其关键字等于给定值的数据元素，若表中有这样的元素，则称查找是成功的，此时查找的信息为给定整个数据元素的输出或指出该元素在表中的位置；若表中不存在这样的记录，则称查找是不成功的，或称查找失败，并可给出相应的提示。

(3) 内部查找 外部查找

若整个查找过程全部在内存进行，则称这样的查找为内查找；反之，若在查找过程中还需要访问外存，则称之为外查找。我们仅介绍内部查找。



(4) 查找表

由同一类型的数据元素构成的集合。可分为静态查找表和动态查找表。

(5) 静态查找 动态查找

查找过程中表中数据不会再增加、删除、修改，如查找数据元素是否在表中，或检查某个数据的属性这些一般查找都属于静态查找。

查找过程中数据元素会增加、删除和修改操作的属于动态查找

。




9. 2 静态表的查找

9. 2. 1 顺序表的查找

9. 2. 2 有序表的查找

9. 2. 3 索引顺序表的查找



9.2.1 顺序表的查找

1. 顺序查找的基本思想(Sequential Search)

顺序查找是一种最简单的查找方法，它的基本思想是：从表的一端开始，顺序扫描线性表，依次将扫描到的结点关键字和待找的值 K 相比较，若相等，则查找成功，若整个表扫描完毕，仍未找到关键字等于 K 的元素，则查找失败。

顺序查找既适用于顺序表，也适用于链表。若用顺序表，查找可从前往后扫描，也可从后往前扫描，但若采用单链表，则只能从前往后扫描。另外，顺序查找的表中元素可以是无序的。

此处以顺序表为例，从后往前扫描。



2. 顺序查找算法实现

```
typedef struct{
    ElemType *elem;
    int length;
}SStable;

int Search_Seq(SStable ST, KeyType key){
    //在顺序表ST中顺序查找其关键字等于Key的数据元素。若找到，则函数值
    //为该元素在表中的位置，否则为0。

    ST.elem[0].key = key; //“哨兵” ， 提高查找效率
    for(i=ST.length; !EQ(ST.elem[i].key,key); --i);
    return i;
} //Search_Seq
```




ST.elem[0]为监视哨，起两个作用：

- 为了省去循环中下标越界的条件 $i \geq 1$ ，从而节省比较时间；若算法中不设立监视哨，程序花费的时间将会增加，例如：

```
for(i=ST.length; !EQ(ST.elem[i].key,key)&&(i>=1); --i);
```

- 保存要找值的副本，若查找时，遇到它，表示查找不成功。

3. 顺序查找性能分析

1、平均查找程度ASL (Average Search Length)

为确定记录在查找表中的位置，需和给定值进行比较的关键字个数的期望值称为查找算法在查找成功时的平均查找长度。**ASL**越小越好。

对于含有n个记录的表，查找成功时的平均查找长度为

$$ASL = \sum_{i=1}^n p_i c_i$$

其中：P_i为查找表中第i个记录的概率，且 $\sum_{i=1}^n p_i = 1$;

一般情形下我们认为查找每个元素的概率相等，

C_i为查找第i个元素所用到的比较次数。



2、 ASL_{ss}

假设在每个位置查找的概率相等，即有 $p_i=1/n$ ，由于查找是从后往前扫描，则有每个位置的查找比较次数 $C_n=1$ ， $C_{n-1}=2$ ， \dots ， $C_1=n$ ，则 $C_i=n-i+1$ 。于是，查找成功的平均查找

$$ASL_{ss} = \sum_{i=1}^n [\frac{1}{n}(n-i+1)] = \frac{n+1}{2}$$

即它的时间复杂度为 $O(n)$ 。这就是说，查找成功的平均比较次数约为表长的一半。若 k 值不在表中，则必须进行 $n+1$ 次比较之后才能确定查找失败。另处，从 ASL 可知，当 n 较大时， ASL 值较大，查找的效率较低。



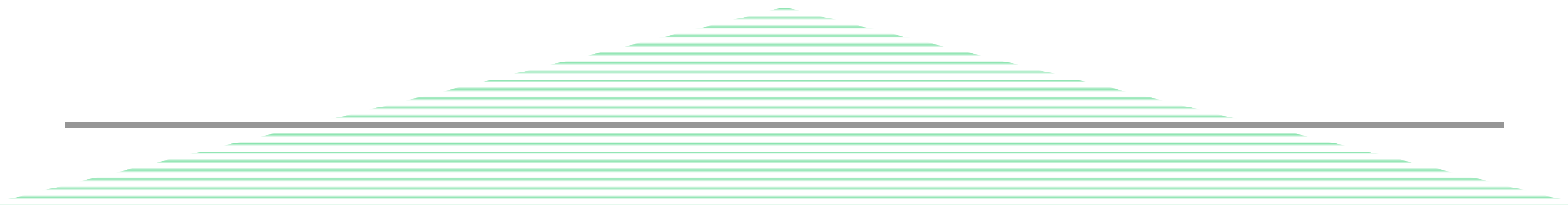
4. 优缺点

优点:

算法简单，对表结构无任何要求，无论是用向量还是用链表来存放结点，也无论结点之间是否按关键字有序或无序，它都同样适用。

缺点:

查找效率低，当 n 较大时，不宜采用顺序查找，而必须寻求更好的查找方法。



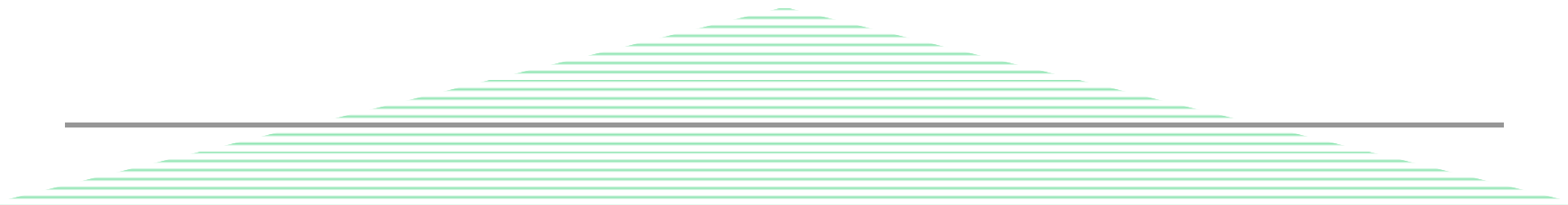


9.2.2二分查找

1. 二分查找的基本思想（Binary Search）


二分查找，也称折半查找，它是一种高效率的查找方法。但二分查找有**条件限制**：要求表必须用向量作存贮结构，且表中元素必须按关键字有序(升序或降序均可)。

二分查找的基本思想是：用于查找一个数据已按大小次序排列的文件的方法，查找时依数据中某个键值进行，先确定待查找所在的范围，然后每次缩小一半范围，直到找到或找不到记录为止。

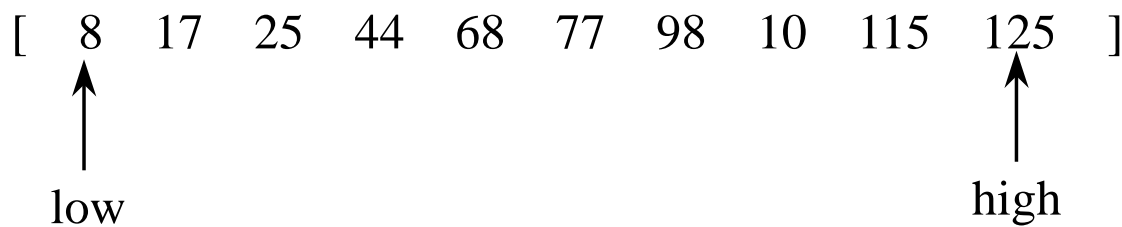


2. 二分查找算法实现

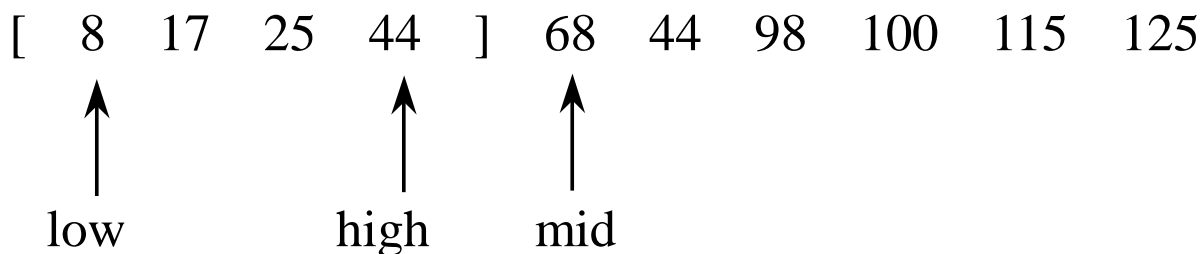
```
int Search_Bin(SSTable ST, KeyType key){  
    //在有序表ST中折半查找其关键字等于key的数据元素。若找到，  
    //则函数值为该元素在表中的位置，否则为0  
    low=1; high=ST.length;  
    while (low<= high){  
        mid=(low +high)/2;           //取区间中点  
        if ( EQ(key,ST.elem[mid].key)) return mid;    //查找成功  
        else if ( LT( key,ST.elem[mid].key ) ) high = mid -1;//在左子区间中查找  
        else low=mid+1;    //在右子区间中查找  
    }  
    return 0; //查找失败  
} //Search_Bin
```



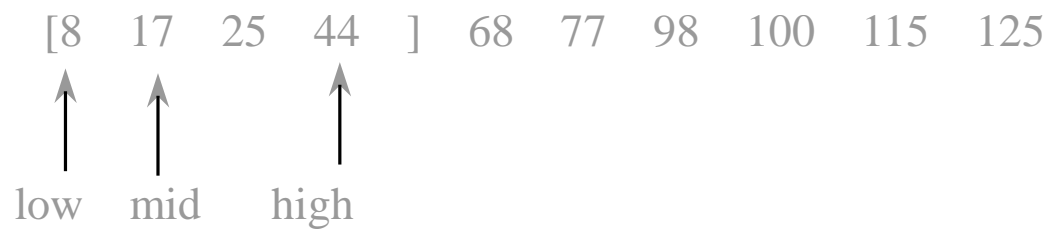
例如，假设给定有序表中关键字为8,17,25,44,68,77,98,100,115,125，将查找K=17和K=120的情况描述为图8-1及图8-2形式。



(a) 初始情形

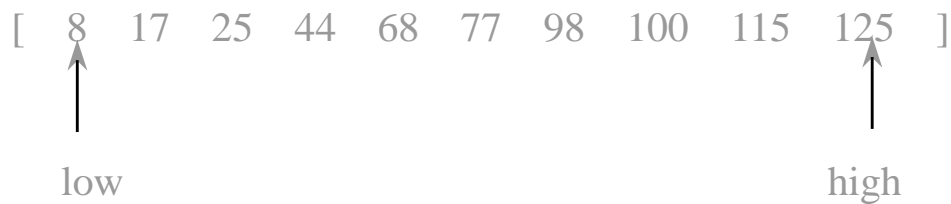


(b) 经过一次比较后的情形

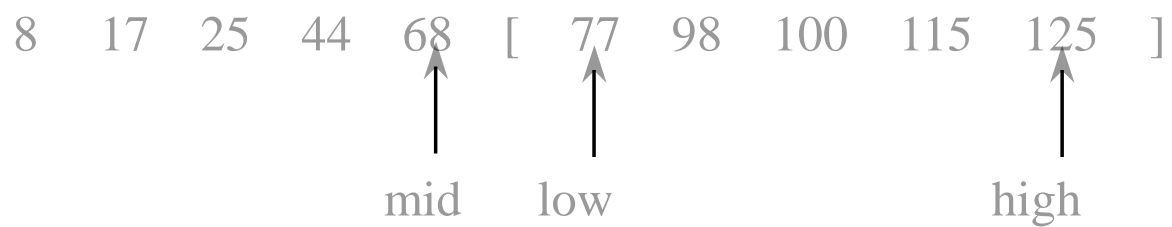


(c) 经过二次比较后的情形 (R[mid].key=17)

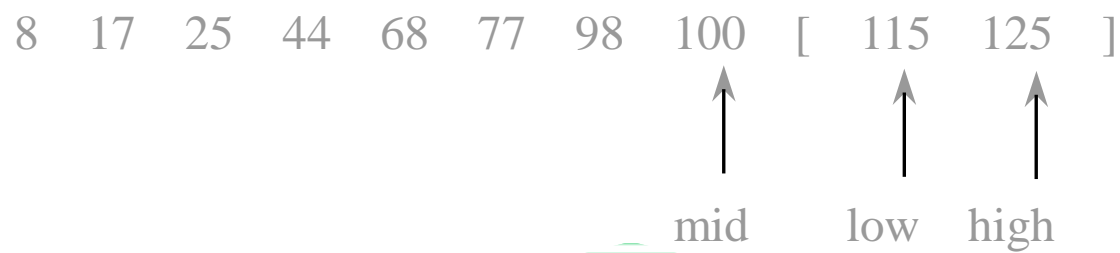
图 8-1 查找 K=17 的示意图 (查找成功)



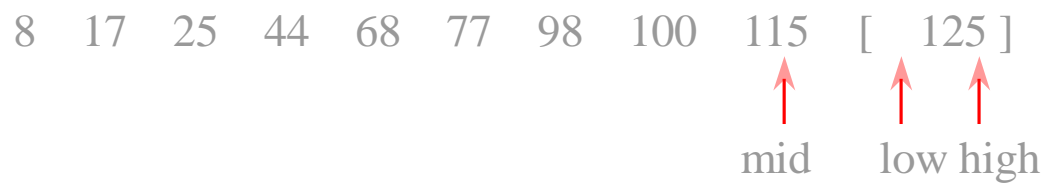
(a) 初始情形



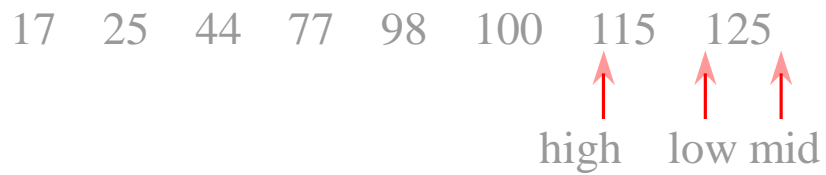
(b) 经过一次比较后的情形



(c) 经过二次比较后的情形



(d) 经过三次比较后的情形



(e) 经过四次比较后的情形($high < low$)

图 8-2 查找 K-120 的示意图 (查找不成功)

3.二分查找的性能分析

1.判定树

树中的每个结点表示表中一个记录，结点中的值为该记录在表中的位置，通常称这个描述查找过程的二叉树为判定树。

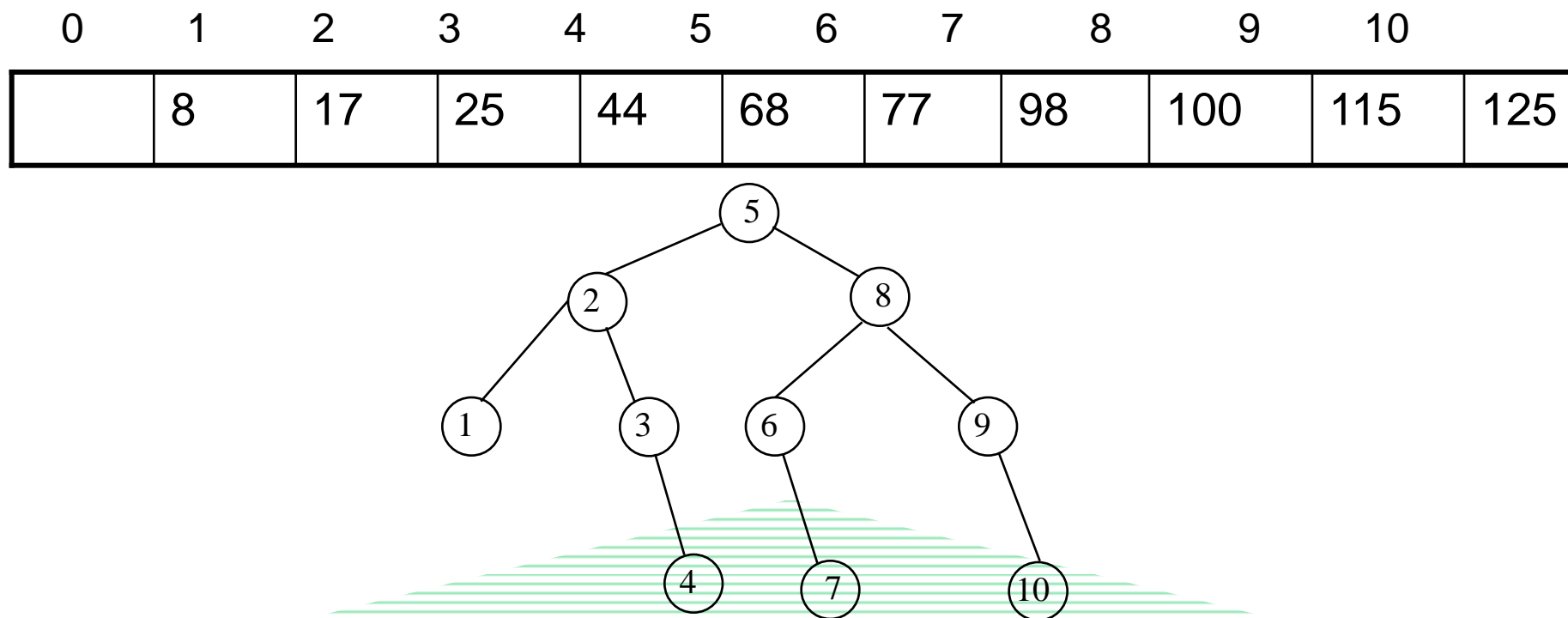


图 8-3 具有 10 个关键字序列的二分查找判定树



2.折半查找的 ASL_{bs}

可以得到结论：二叉树第K层结点的查找次数各为k次(根结点为第1层)，而第k层结点数最多为 2^{k-1} 个。假设该二叉树的深度为h，则二分查找的成功的平均查找长度为（假设每个结点的查找概率相等）：

$$ASL = \sum_{i=1}^n p_i c_i = 1/n \sum_{i=1}^n c_i \leq 1/n (1 + 2 \times 2 + 3 \times 2^2 + \dots + h \times 2^{h-1}),$$

因此，在最坏情形下，上面的不等号将会成立，并根据二叉树的性质，最大的结点数 $n=2^h-1$ ， $h=\log_2(n+1)$ ，于是可以得到平均查找长度 $ASL=(n+1)/n \log_2(n+1)-1$ 。

当n很大时， $ASL \approx \log_2(n+1)-1$ 可以作为二分查找成功时的平均查找长度，它的时间复杂度为 $O(\log_2 n)$ 。



9.2.3 索引顺序表的查找

1.基本思想

是顺序查找的改进方法。

基本思想：将一个含有 n 个元素的主表分成 m 个子表，但要求子表之间元素是按关键字有序排列的，而子表中元素可以无序的，然后建立索引表，索引表中索引域的值用每个子表最大关键字代替，则可以按索引查找思想找到表中元素。

例如，给定关键字序列如下：

18,7,26,34,15,42,36,70,60,55,83,90,78,72,74，假设 $m=3$ ， $s=5$ ，即将该序列分成3个子表，每个子表有5个元素，则得到的主表和索引表如图8-5所示。

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
18	7	26	34	15	42	36	70	60	55	83	90	78	72	74

(a) 15 个关键字序列得到的主表

index	start	length
34	0	5
70	5	5
90	10	5

(b) 按关键字序列递增得到的索引表

图 8-5 分块查找的主表和索引表



静态查找表的比较

A: 顺序表的查找 B: 有序表的查找 C: 索引顺序表的查找

- 平均查找长度: $B < C < A$
- 记录顺序: A灵活性最好
- 表的存储结构: A: 无要求, 可为顺序表, 也可为链表
B: 一定为顺序表
C: 都可以, 但要有一个索引表



作业

9.1、9.3、9.26

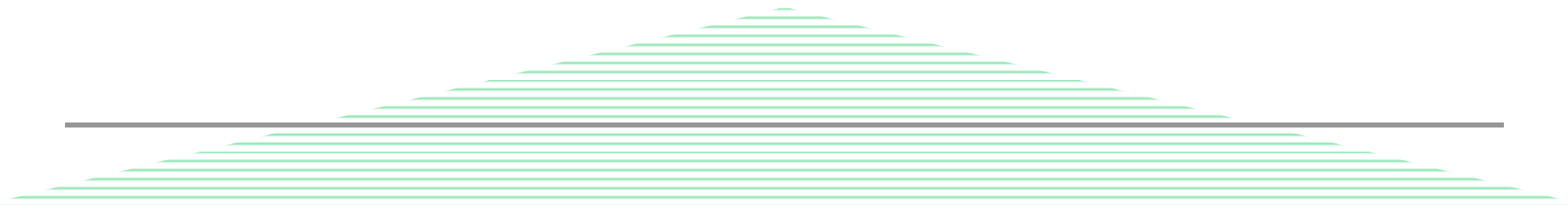


9. 3 动态查找表

1.特点

表结构本身是在查找过程中动态生成的，对于给定的**KEY**，若表中存在其关键字等于**KEY**的记录，则查找成功返回，否则插入关键字等于**KEY**的记录。

2.方法（各种树结构表示时的实现方法）

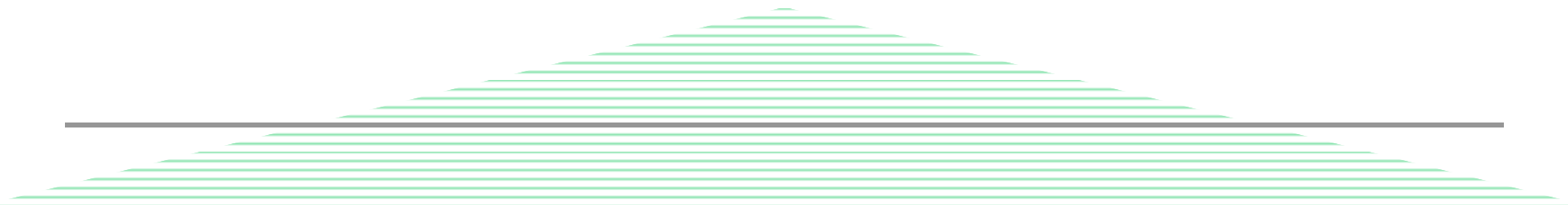
- a. 二叉排序树
 - b. 平衡二叉树
 - c. B树
 - d. B⁺树
- 



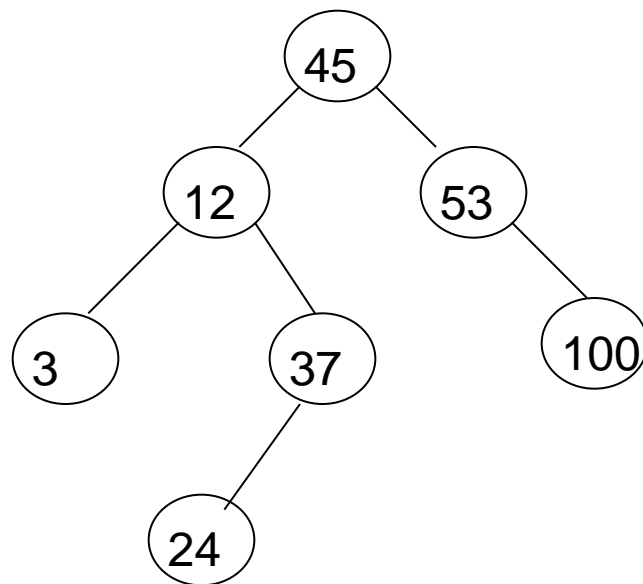
9.3.1 二叉排序树查找

1. 什么是二叉排序树

二叉排序树（Binary Sorting Tree），它或者是一棵空树，或者是一棵具有如下特征的非空二叉树：

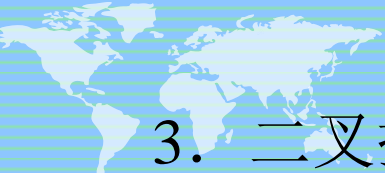
- (1) 若它的左子树非空，则左子树上所有结点的关键字均小于根结点的关键字；
 - (2) 若它的右子树非空，则右子树上所有结点的关键字均大于等于根结点的关键字；
 - (3) 左、右子树本身又都是一棵二叉排序树。
- 

2. 二叉排序树的特点



中序序列：3，12，24，37，45，53，100

结论：中序遍历二叉树可得到一个关键字的有序序列



3. 二叉排序 树上的查找

(1) 二叉排序 树的查找思想

当二叉排序树不为空的时候，若相等，则查找成功，若根结点值大于待查值，则进入左子树重复此步骤，否则，进入右子树重复此步骤，若在查找过程中遇到二叉排序树的叶子结点时，还没有找到待找结点，则查找不成功。

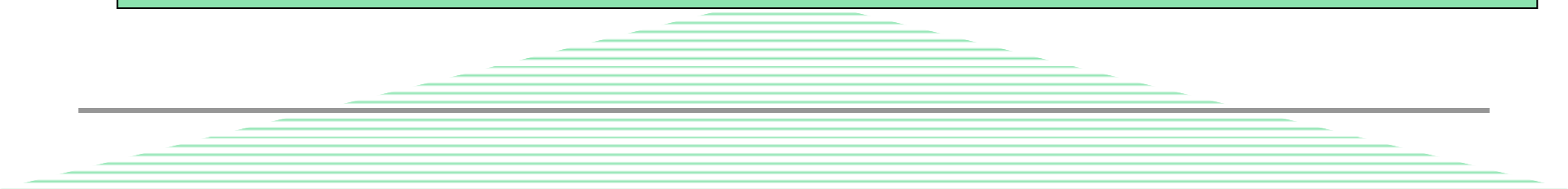
(2) 二叉排序树查找的算法实现





算法一

```
BiTree SearchBST(BiTree T, KeyType key){  
    //在根结点T所指二叉排序树中递归地查找某关键字等于key的数据元素，  
    //若查找成功，则返回指向该数据元素结点的指针，否则返回空指针  
    if (!T||EQ ( key,T->data.key)) return( T ); //查找结束  
    else if LT( key,T->data.key) return( SearchBST(T->lchild, key));  
                                                //在左子树中继续查找  
    else return( SearchBST(T->rchild, key)); //在右子树中继续查找  
} //SearchBST
```





算法二

```
Status SearchBST(BiTree T, KeyType key, BiTree f, BiTree &p){  
    //在根结点T所指二叉排序树中递归地查找某关键字等于key的数据元素，  
    //若查找成功，则指针P指向该数据元素结点，并返回TURE，否则指针p  
    指  
    //向查找路径上访问的最后一个结点并返回FALSE，指针f指向T的双亲，  
    //其初始调用值为NULL。  
    if (!T) {p=f; return FALSE;} //查找不成功  
    else if EQ( key,T->data.key) {p=T; return True;} //查找成功  
    else if LT ( key,T->data.key ) SearchBST(T->lchild, key,T,p);  
                                     //在左子树中继续查找  
    else SearchBST(T->rchild, key,T,p); //在右子树中继续查找  
}  
//SearchBST
```

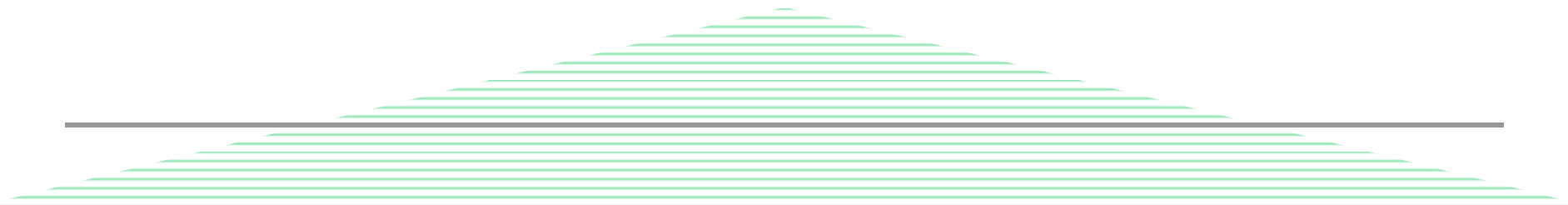


4. 二叉排序树的基本运算

(1) 二叉排序树的插入

若二叉排序树为空，则作为根结点插入，否则，若待插入的值小于根结点值，则作为左子树插入，否则作为右子树插入。

新插入的结点一定是新添加的叶子结点，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。



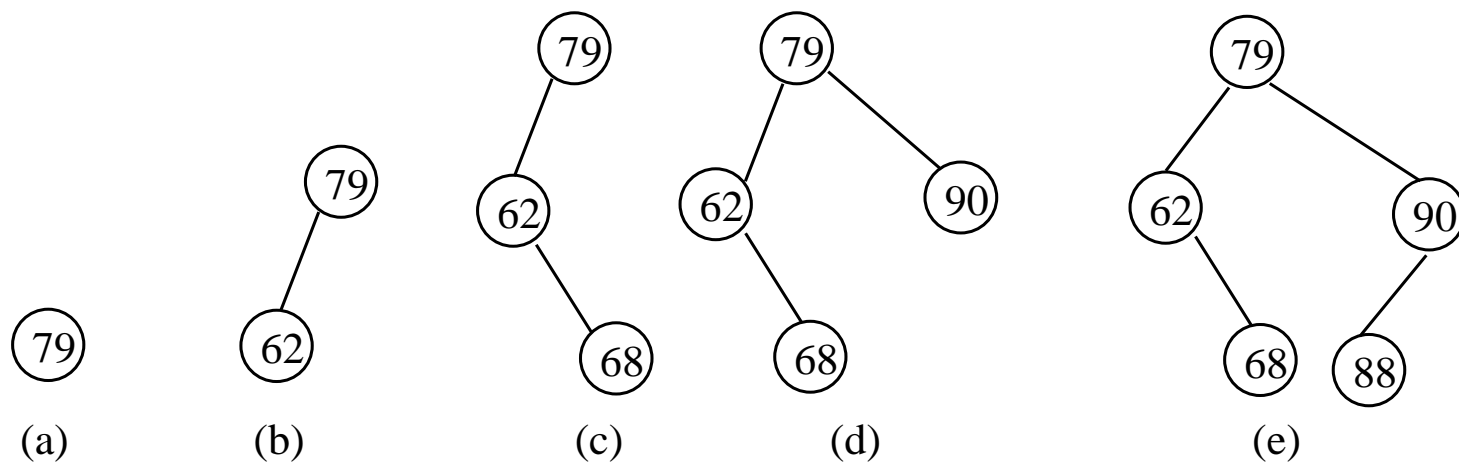


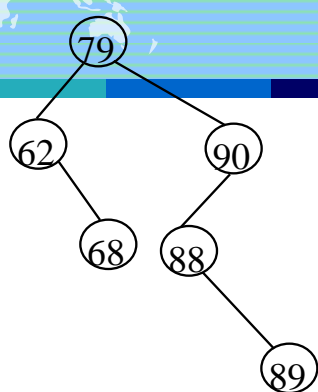
```
Status Insert_BST(BiTree &T,ElemType e){  
    //当二叉排序树T中不存在关键字等于e.key的数据元素时，插入e并返回True，  
    //否则返回False  
    if(!SearchBST(T, e.key,NULL,p)){           //查找不成功  
        s=(Bitree)malloc(sizeof(BiTNode));  
        s->data=e; s->lchild=s->rchild=NULL;  
        if(!p) T=s;           //原来树为空树，则被插结点*s为新的根结点  
        else if(LT(e.key,p->data.key))    p->lchild=s; //为左孩子  
        else p->rchild = s;           //为右孩子  
        return TRUE;  
    }  
    else retrun FLASE;           //树中已有关键字相同的结点，不再插入  
}  
} //Insert_BST
```


(2) 二叉排序树的建立

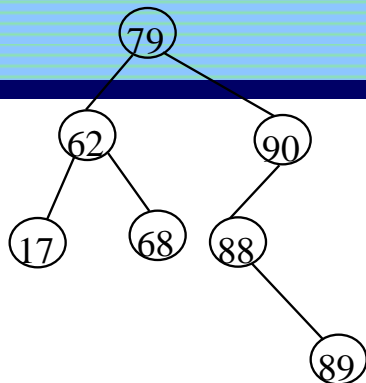
基本思想：只要反复调用二叉排序树的插入算法即可。

例如，给定关键字序列79，62，68，90，88，89，17，5，100，120，生成二叉排序树过程如图8-6所示。

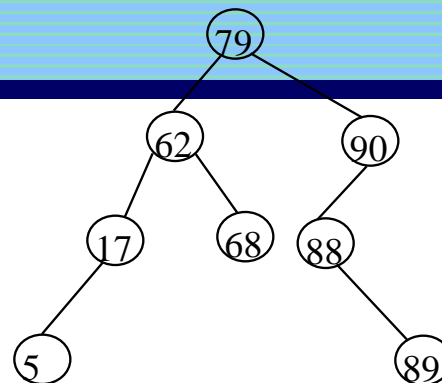




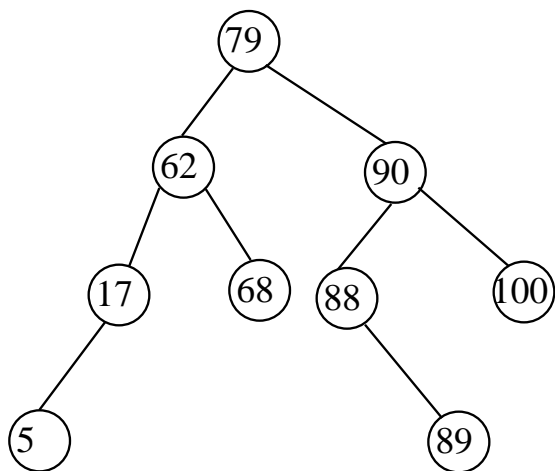
(f)



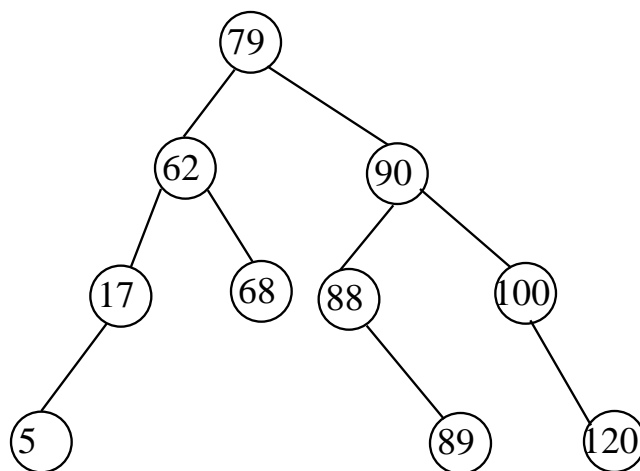
(g)



(h)



(i)



(j)

二叉排序树的生成过程

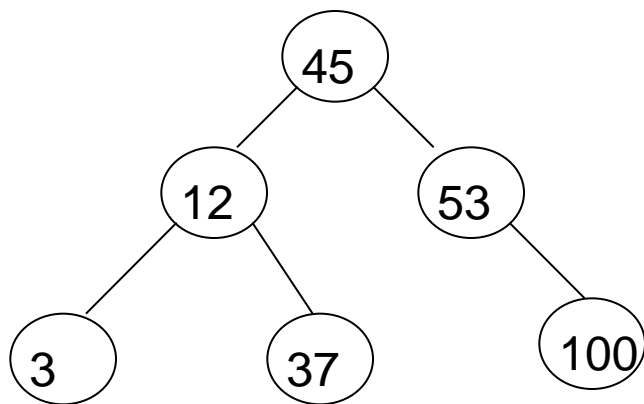


结论

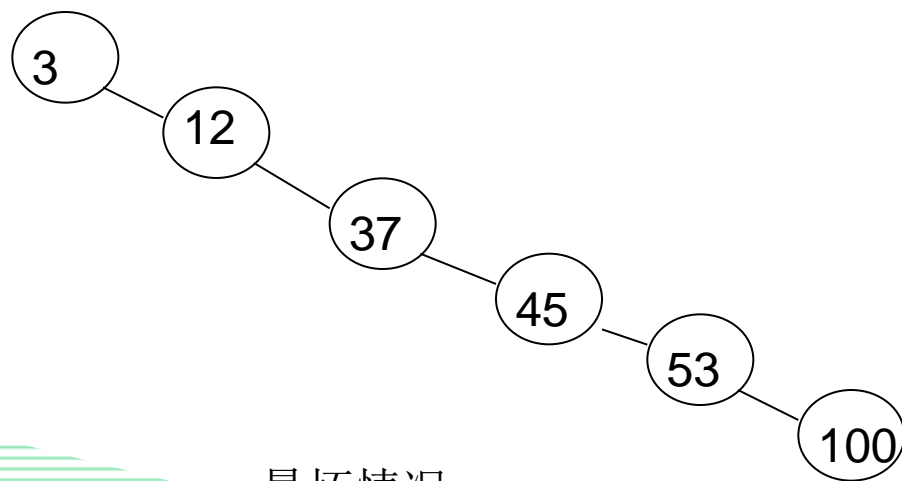
- 二叉排序树与关键字排列顺序有关，排列顺序不一样，得到的二叉排序树也不一样
- 一个无序序列可以通过构造一棵二叉排序树变成一个有序序列。
- 二叉排序树类似于折半查找，有采用了链表做存储结构。

5. 二叉排序树查找的性能分析

在二叉排序树查找中，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 $\log_2 n$ ，最坏为 n 。因此，二叉排序树查找的最好时间复杂度为 $O(\log_2 n)$ ，最坏的时间复杂度为 $O(n)$ ，一般情形下，其时间复杂度大致可看成 $O(\log_2 n)$ ，比顺序查找效率要好，但比折半查找要差。



最好情况



最坏情况



9.3.2平衡二叉树查找

1. 平衡二叉树的概念

平衡二叉树(balanced binary tree)是由阿德尔森-维尔斯和兰迪斯(Adelson-Velskii and Landis)于1962年首先提出的, 所以又称为AVL树。

若一棵二叉树中每个结点的左、右子树的深度之差的绝对值不超过1, 则称这样的二叉树为平衡二叉树。将该结点的左子树深度减去右子树深度的值, 称为该结点的平衡因子(balance factor)。也就是说, 一棵二叉排序树中, 所有结点的平衡因子只能为0、1、-1时, 则该二叉排序树就是一棵平衡二叉树, 否则就不是一棵平衡二叉树。

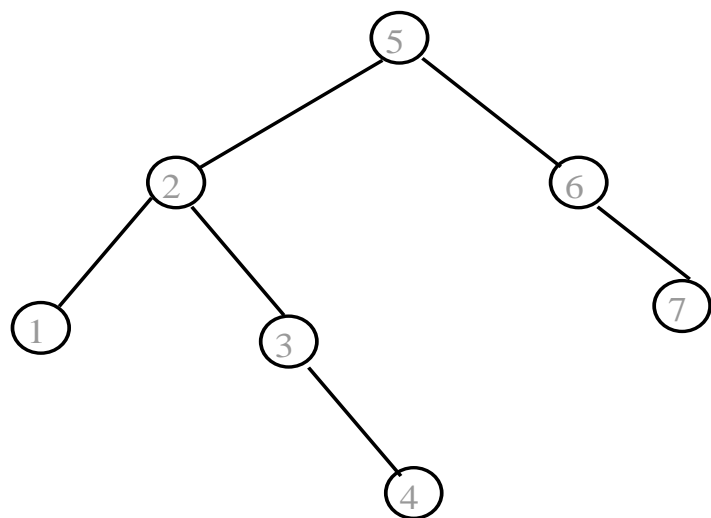


图 8-8 一棵平衡二叉树

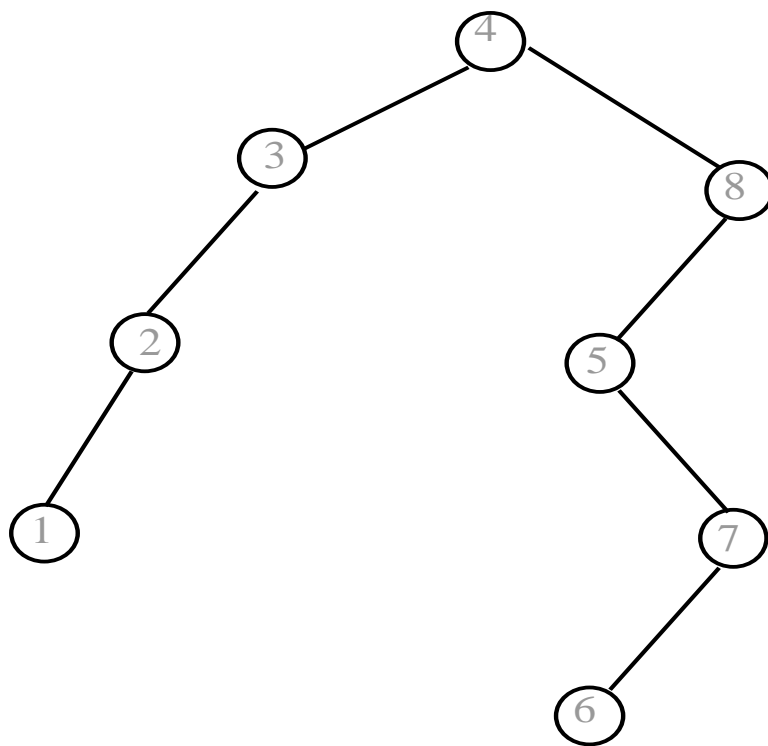


图 8-9 一棵非平衡二叉树



2.非平衡二叉树的平衡处理

若一棵二叉排序树是平衡二叉树，插入某个结点后，可能会变成非平衡二叉树，这时，就可以对该二叉树进行平衡处理，使其变成一棵平衡二叉树。

处理的原则应该是处理与插入点最近的、而平衡因子又比1大或比-1小的结点。下面将分四种情况讨论平衡处理。

(1) LL型 的处理(左左型)

如图9-10所示，在C的左孩子B上扞入一个左孩子结点A，使C的平衡因子由1变成了2，成为不平衡的二叉树序树。这时的平衡处理为：将C顺时针旋转，成为B的右子树，而原来B的右子树则变成C的左子树，待扞入结点A作为B的左子树。(注：图中结点旁边的数字表示该结点的平衡因子)

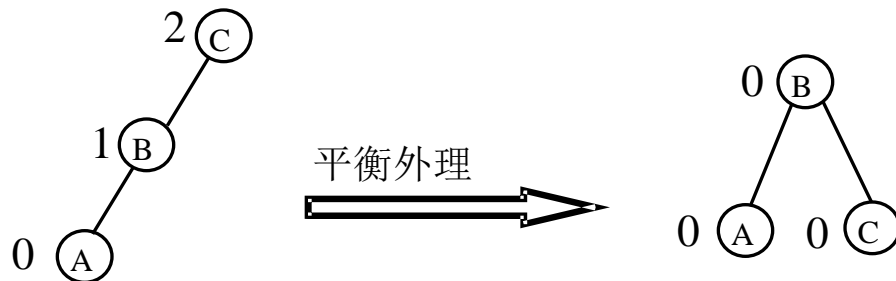


图 9-10 LL 型平衡外理

(2) LR型的处理(左右型)

如图8-11所示，在C的左孩子A上扞入一个右孩子B，使的C的平衡因子由1变成了2，成为不平衡的二叉排序树。这是的平衡处理为：将B变到A与C 之间，使之成为LL型，然后按第(1)种情形LL型处理。

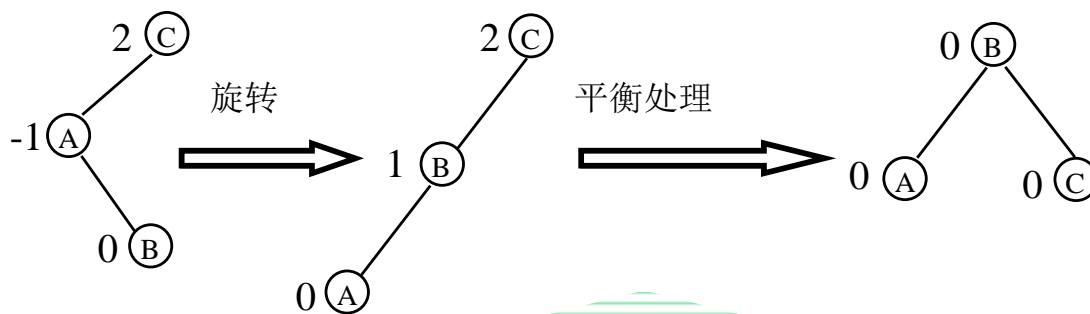


图 8-11 LR 型平衡处理

(3) **RR型的处理(右右型)**

如图9-12所示，在A的右孩子B上扞入一个右孩子C，使A的平衡因子由-1变成-2，成为不平衡的二叉排序树。这时的平衡处理为：将A逆时针旋转，成为B的左子树，而原来B的左子树则变成A的右子树，待扞入结点C成为B的右子树。

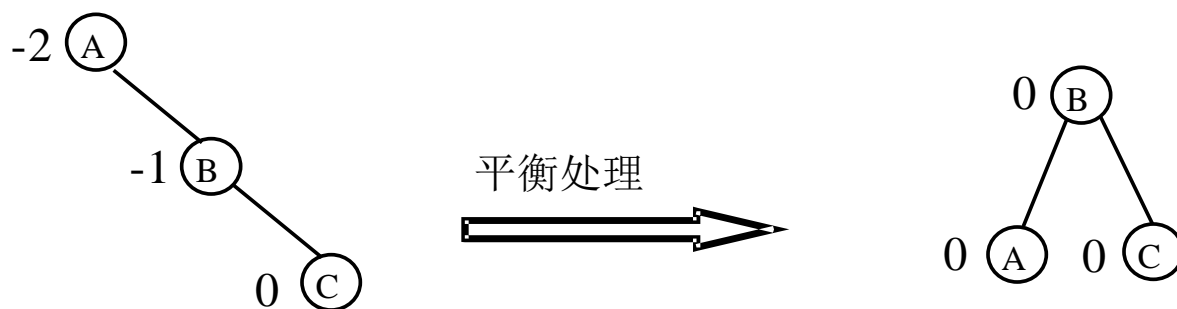


图 9-12 RR 型平衡处理

(4) RL型的处理(右左型)

如图8-13所示，在A的右孩子C上扞入一个左孩子B，使A的平衡因子由-1变成-2，成为不平衡的二叉排序树。这时的平衡处理为：将B变到A与C之间，使之成为RR型，然后按第(3)种情形RR型处理。

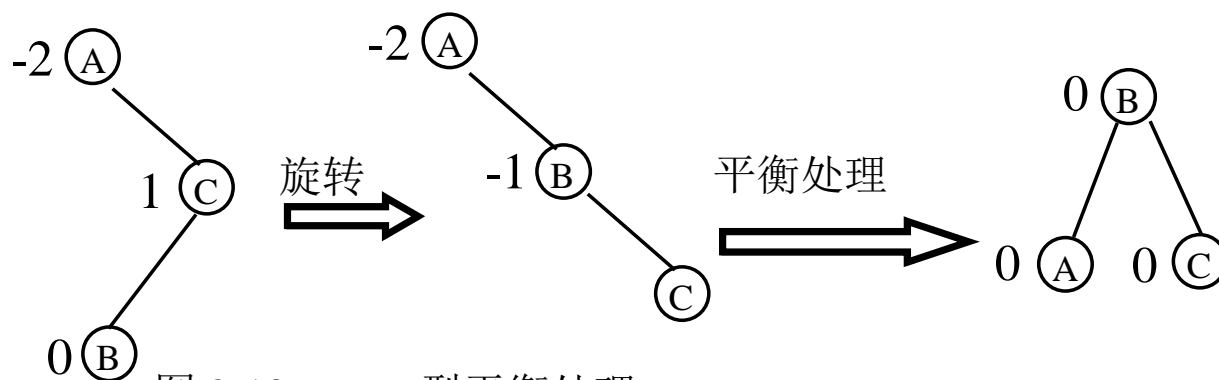
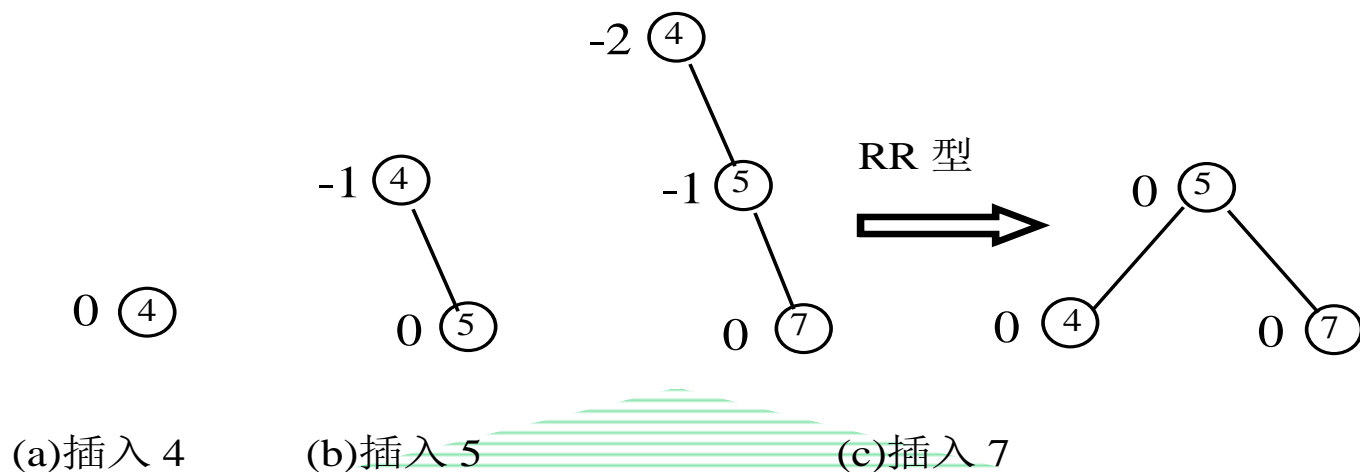


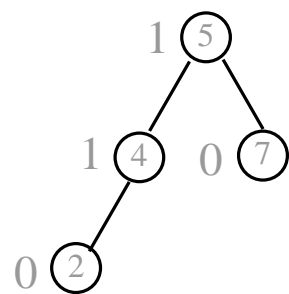
图 8-13 RL 型平衡处理



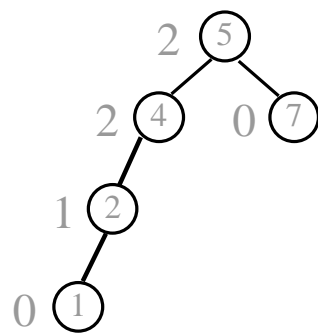
例9-2，给定一个关键字序列4,5,7,2 ,1,3,6，试生成一棵平衡二叉树。

分析：平衡二叉树实际上也是一棵二叉排序树，故可以按建立二叉排序树的思想建立，在建立的过程中，若遇到不平衡，则进行相应平衡处理，最后就可以建成一棵平衡二叉树。具体生成过程见图8-14。

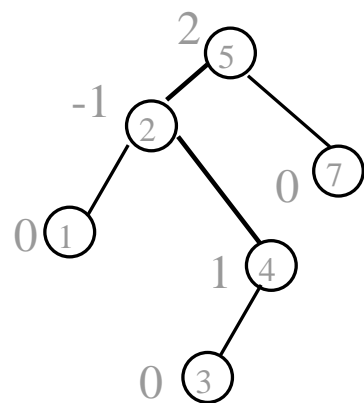
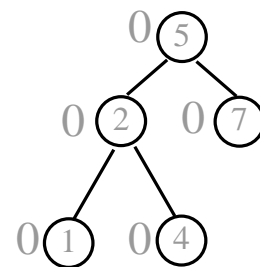




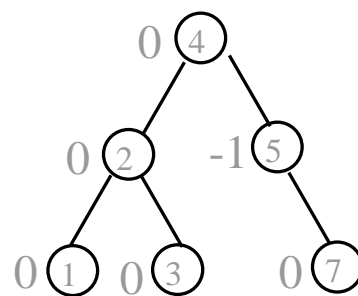
(d)插入 2

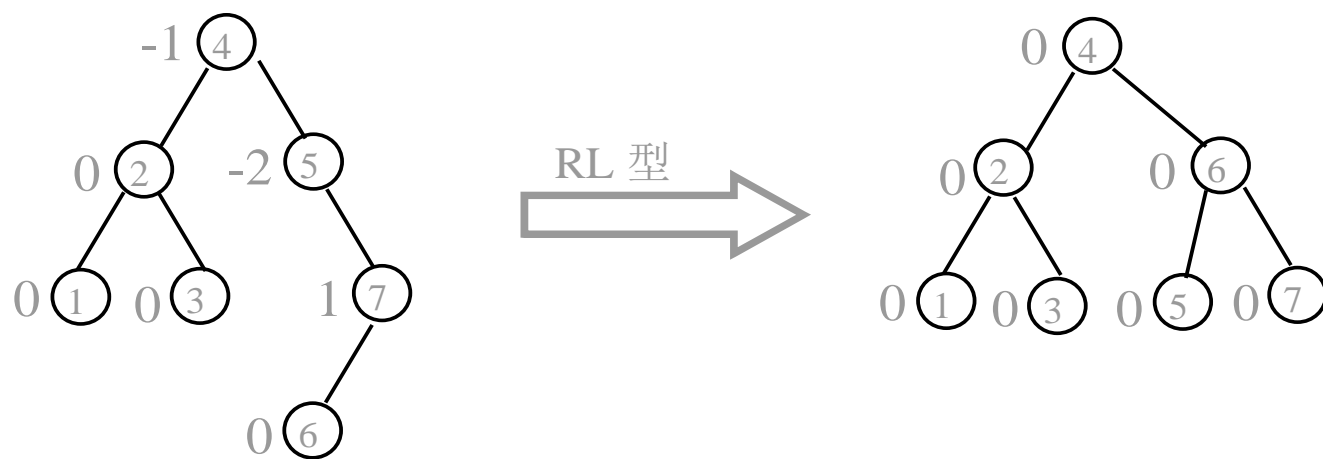


(e)插入 1




(f)插入 3





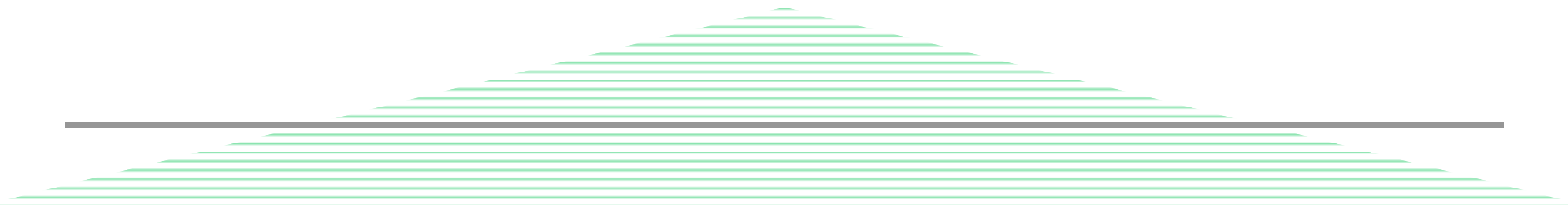
(g) 插入 6

图 8-14 平衡二叉树的生成过程



3. 平衡二叉树的查找及性能分析

平衡二叉树本身就是一棵二叉排序树，故它的查找与二叉排序树完全相同。但它的查找性能优于二叉排序树，不像二叉排序树一样，会出现最坏的时间复杂度 $O(n)$ ，它的时间复杂度与二叉排序树的最好时间复杂度相同，都为 $O(\log_2 n)$ 。

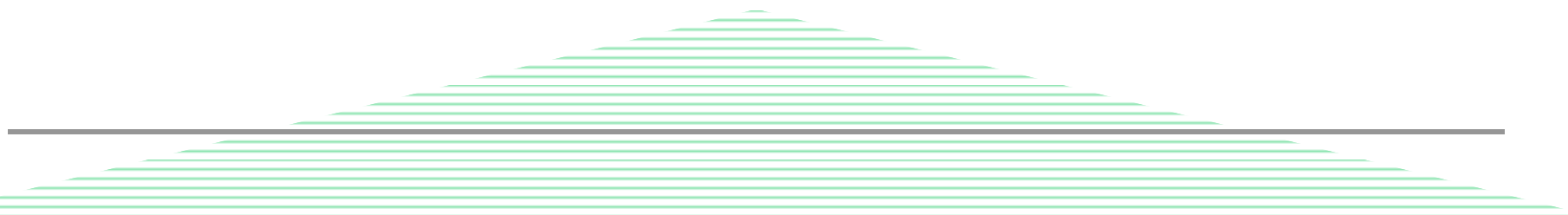




作 业

9.9

9.11





9.3.3 B-树查找

1. B-树的概念

B-树是一种平衡的多路查找树，与二叉排序树的查找类似，它在文件系统中很有用。

定义：一棵m阶的B-树，或者为空树，或为满足下列特性的m叉树：

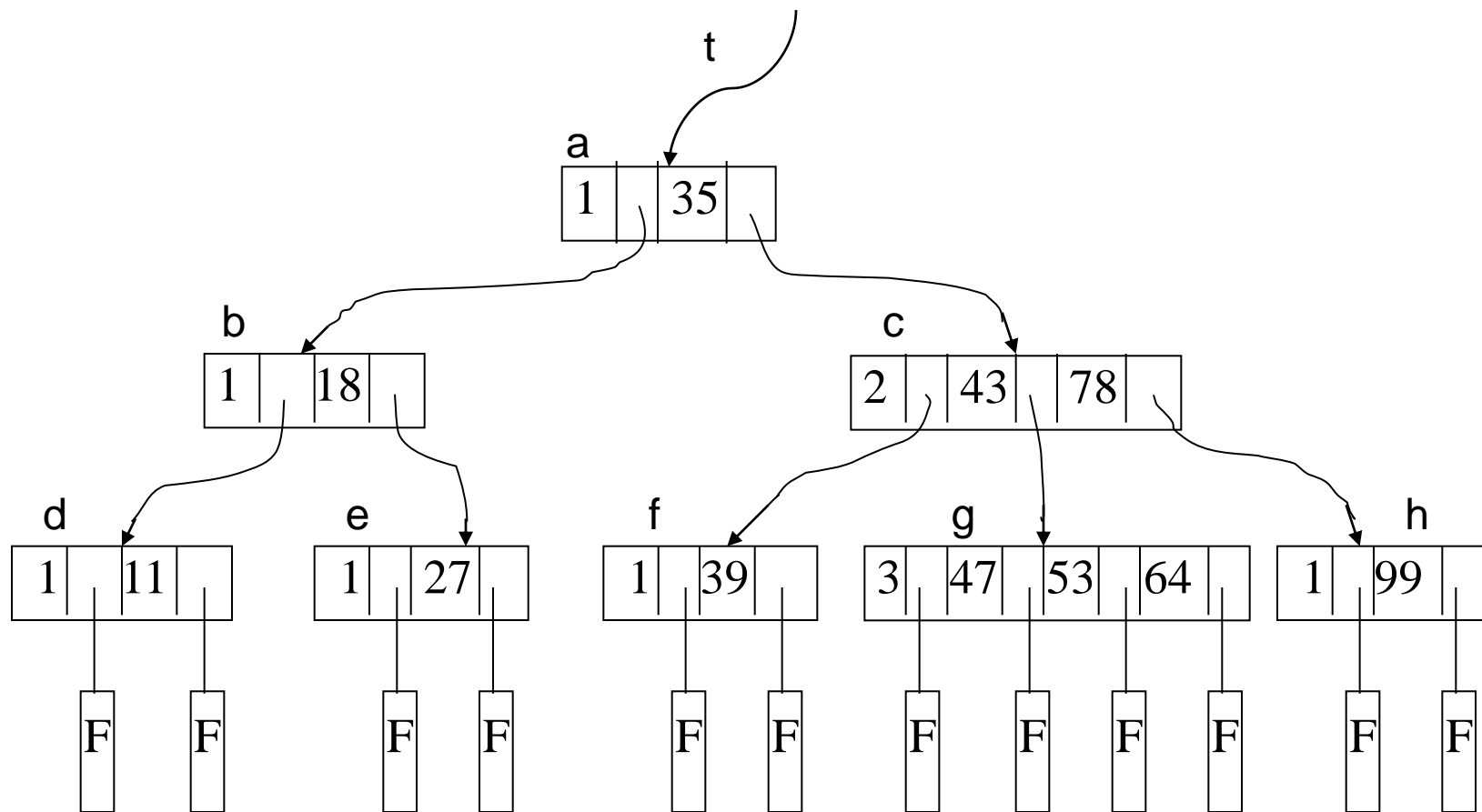
- (1) 树中每个结点至多有m棵子树；
- (2) 若根结点不是叶子结点，则至少有两棵子树；
- (3) 除根结点之外的所有非终端结点至少有 $\lceil m/2 \rceil$ 棵子树；
- (4) 所有的非终端结点中包含以下信息数据：

$(n, A_0, K_1, A_1, K_2, \dots, K_n, A_n)$

其中： K_i ($i=1,2,\dots,n$) 为关键码，且 $K_i < K_{i+1}$ ，

A_i 为指向子树根结点的指针 ($i=0,1,\dots,n$)，且指针 A_{i-1} 所指子树中所有结点的关键码均小于 K_i ($i=1,2,\dots,n$)， A_n 所指子树中所有结点的关键码均大于 K_n ， $\lceil m/2 \rceil - 1 \leq n \leq m - 1$ ， n 为关键码的个数。

- (5) 所有的叶子结点都出现在同一层次上，并且不带信息（可以看作是外部结点或查找失败的结点，实际上这些结点不存在，指向这些结点的指针为空）。



一棵4阶B树



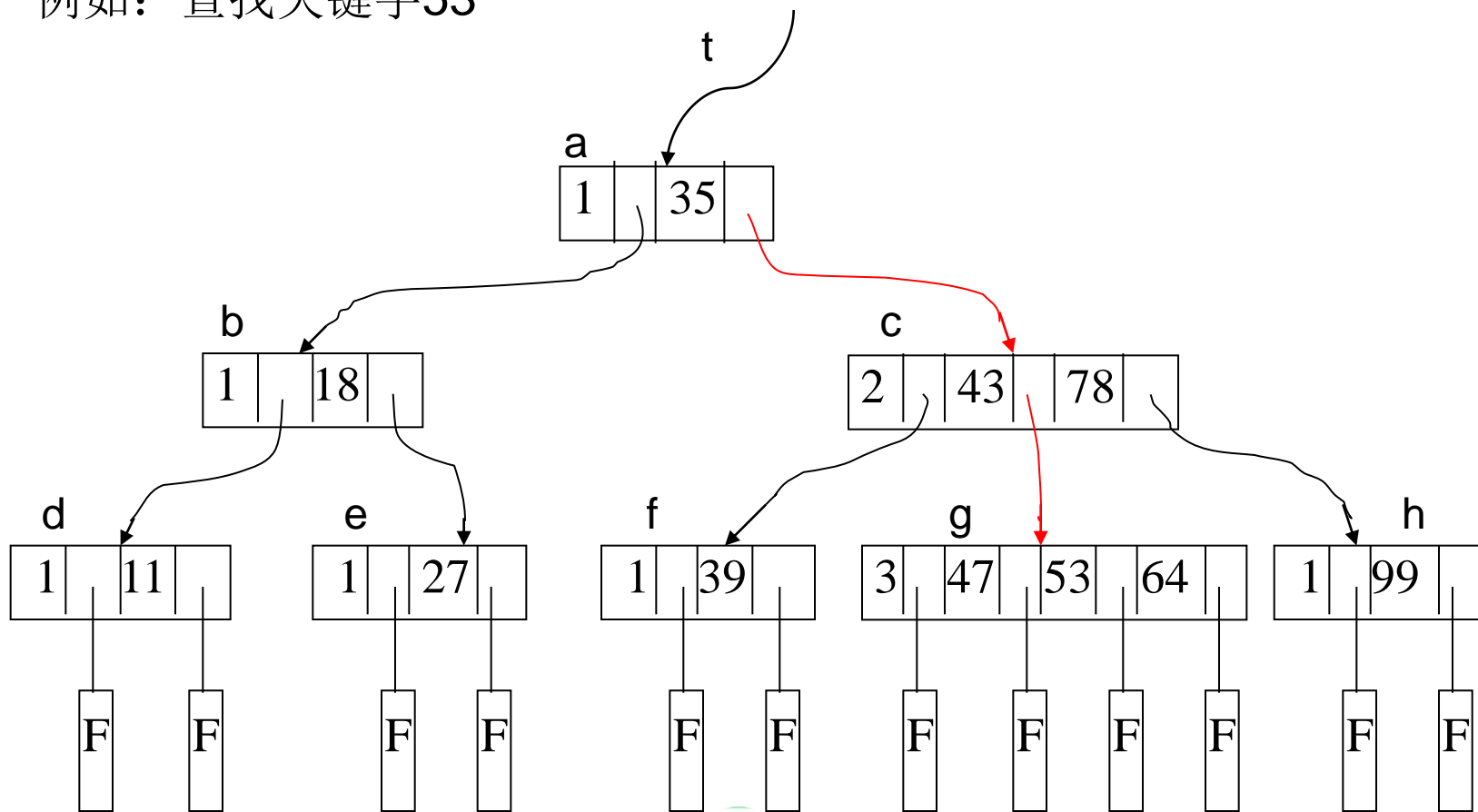
由定义可知：

	子树数	关键字数
根结点	$2 \sim m$	$1 \sim m-1$
非终端结点	$\lceil m/2 \rceil \sim m$	$\lceil m/2 \rceil - 1 \sim m-1$

例如：3阶B树，子树个数为2或3，又称2~3树

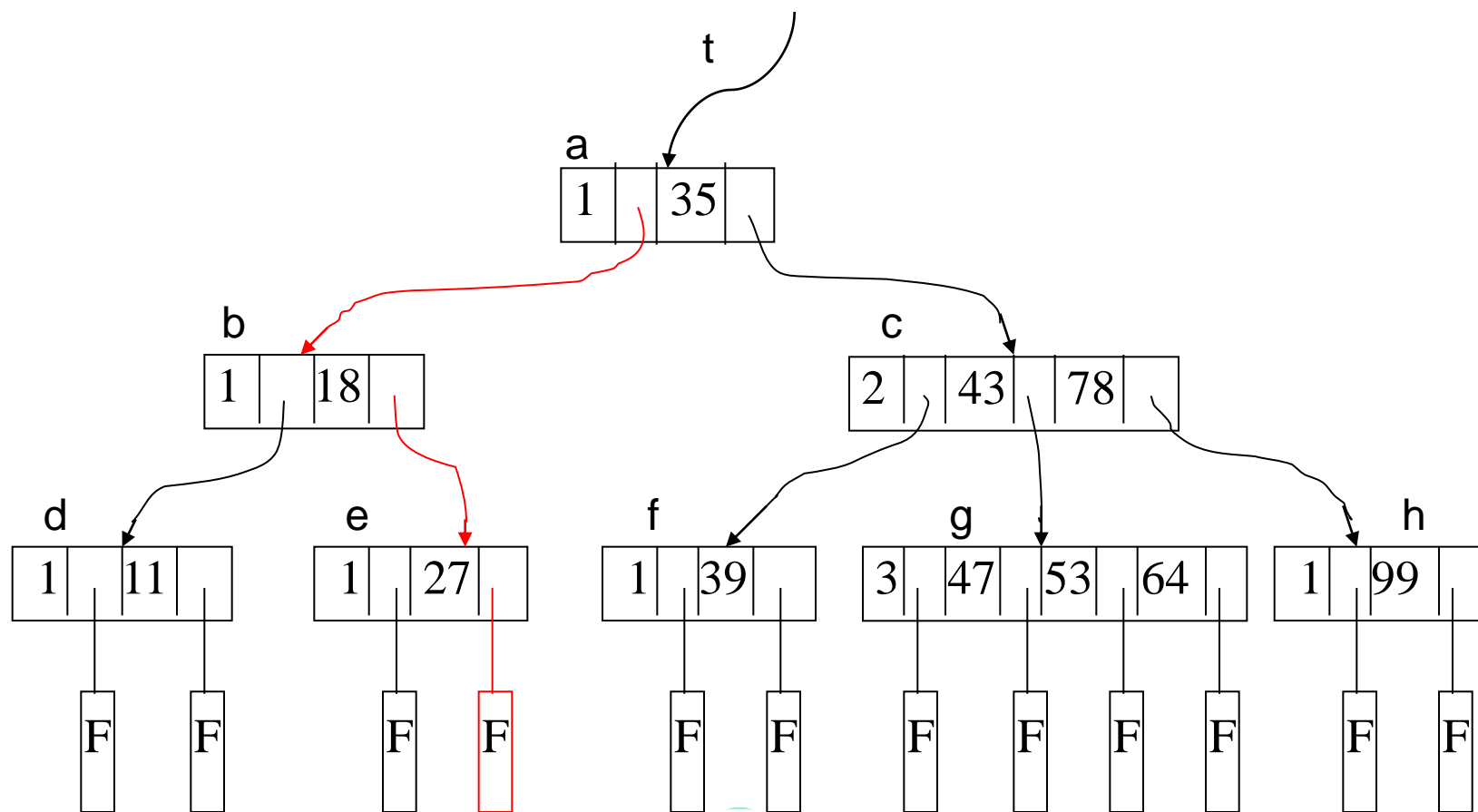
2、B-树的查找

例如：查找关键字53



查找成功

例如：查找关键字33



查找不成功



B-树的查找类似二叉排序树的查找，所不同的是**B-树**每个结点上是多关键码的有序表，在到达某个结点时，先在有序表中查找，若找到，则查找成功；否则，到按照对应的指针信息指向的子树中去查找，当到达叶子结点时，则说明树中没有对应的关键码，查找失败。

B-树的查找是由两个基本操作交叉进行的过程，即

- (1)在**B-树**上找结点；（在磁盘中）
- (2)在结点中找关键码。（在内存中）



查找算法：


```
#define    m    3                //B树的阶，暂设为3

typedef struct BTreeNode{
    int    keynum;                //结点中关键码的个数，即结点的大小
    struct BTreeNode *parent;    //指向双亲结点
    KeyType key[m+1];            //关键字向量，0号单元未用
    struct BTreeNode *ptr[m+1];  //子树指针向量
    Record *recptr[m+1];        //记录指针向量, 0号单元未用
} BTreeNode, * BTree;

typedef struct{
    BTreeNode *pt;               //指向找到的结点
    int    i;                    //在结点中的关键码序号，结点序号区间[1...m]
    int    tag;                  // 1:查找成功，0:查找失败
}Result;                        //B树的查找结果类型
```



```
Result SearchBTree( BTree T, KeyType k){  
    //在m阶B树T上查找关键码k，返回结果(pt, i, tag)。若查找成功，则特征值tag=1，  
    //指针pt所指结点中第i个关键码等于k；否则特征值tag=0,等于k的关键码记录  
    //应插入在指针pt所指结点中第i个和第i+1个关键码之间  
    p=T; q=NULL; found=FALSE; i=0; //初始化，p指向待查结点，q指向p的双亲  
    while( p&&! found){  
        n=p->keynum; i=Search( p,k);           //在p->key[1...keynum]中查找  
                                                //i使得： p->key[i]<=k<p->key[i+1]  
        if(i>0&&p->key[i]= =k)   found=TRUE; //找到待查关键字  
        else {q=p; p=p->ptr[i]; }  
    }  
    if( found)      return (p,i,1);           //查找成功  
    else return (q,i,0);   //查找不成功，返回k的插入位置信息  
} //SearchBTree
```

3. B-树的性能分析

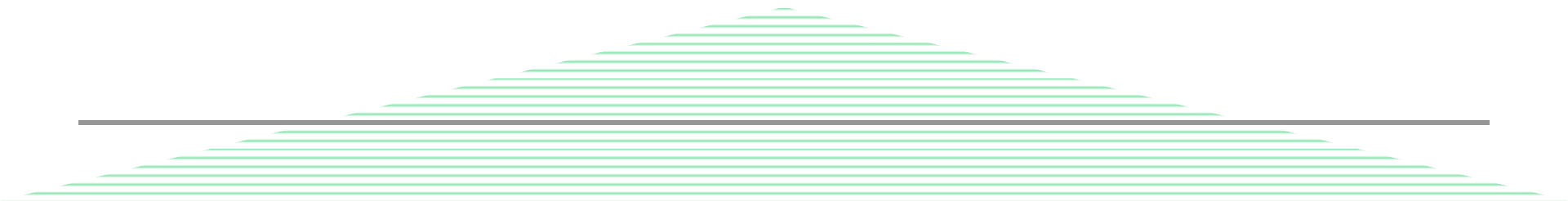
在磁盘上读取结点信息的次数，即B-树的层次树是决定B-树查找效率的首要因素。在含有n个关键码的B-树上进行查找时，从根结点到关键码所在结点的路径上涉及的结点数不超过

$$\log_{\lceil m/2 \rceil} \left(\frac{n+1}{2} \right) + 1$$



作 业

思考：9.13





9.4 哈希查找

9.4.1 基本概念

1. 哈希查找 哈希表

哈希查找，也称为散列查找，或杂凑查找法。它既是一种查找方法，又是一种存贮方法，称为哈希存贮。哈希存贮的内存存放形式也称为哈希表（hash table）。

散列查找是通过构造散列函数来得到待查关键字的地址，按理论分析真正不需要用到比较的一种查找方法。

2. 哈希函数（hash function）

将关键项转换成数组下标值的函数，表示为 $h(\text{key})$ 或 $f(\text{key})$ 。代表关键字 k 在存贮区中的地址，而存贮区为一块连续的内存单元，可用一个一维数组(或链表)来表示。



例9-4，假设有一批关键字序列18,75,60,43,54,90,46，给定散列函数

$H(k) = k \% 13$ ，存贮区的内存地址从0到15，则可以得到每个关键字的散列地址为：

$$H(18) = 18 \% 13 = 5$$

$$H(75) = 75 \% 13 = 10$$

$$H(60) = 60 \% 13 = 8$$

$$H(43) = 43 \% 13 = 4$$

$$H(54) = 54 \% 13 = 2$$

$$H(90) = 90 \% 13 = 12$$

$$H(46) = 46 \% 13 = 7$$

于是，根据散列地址，可以将上述7个关键字序列存贮到一个一维数组HT（散列表）中，具体表示为：

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
HT			54		43	18		46	60		75		90			



3. 查找

例如，查找75，只需计算出 $H(75) = 75 \% 13 = 10$ ，则可以在HT[10]中找到75。

4. 同义词冲突

对两个不同的关键字有可能对应同一个哈希地址，既 $key1 \neq key2$ 而 $f(key1) = f(key2)$ ，将导致后放的关键字无法存贮，我们把这种现象叫做同义词冲突（collision）。Key1和key2为同义词。

例如：关键词28， $H(28) = 28 \% 13 = 2$

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
HT			54		43	18		46	60		75		90			

28



发生冲突的可能性与三个方面因素有关。

1.与装填因子 α 有关, $\alpha=n/m$

所谓装填因子是指散列表中已存入的元素个数 n 与散列表的大小 m 的比值, 即 $\alpha=n/m$ 。当 α 越小时, 发生冲突的可能性越小, α 越大(最大为1)时, 发生冲突的可能性就越大。但是, 为了减少冲突的发生, 不能将 α 变得太小, 这样将会造成大量存贮空间的浪费, 因此必须兼顾存储空间和冲突两个方面。

2.与所构造的哈希函数有关

3.与解决冲突的方法有关



9.4.2 哈希函数的构造

哈希函数的构造目标是使哈希地址尽可能均匀地分布在哈希空间上，同时使计算尽可能简单。具体常用的构造方法有如下几种：

1. 直接定址法

可表示为 $H(k) = a.k + b$ ，其中 a 、 b 均为常数。

这种方法计算特别简单，并且不会发生冲突，但当关键字分布不连续时，会出现很多空闲单元，将造成大量存贮单元的浪费。

2. 数字分析法

对关键字序列进行分析，取那些位上数字变化多的、频率大的作为哈希函数地址。



例如，对如下的关键字序列
：

4 3 0 1 0 4 6 8 1 0 1 5 3 5 5

4 3 0 1 0 1 7 0 1 1 2 8 3 5 2

4 3 0 1 0 3 7 2 0 8 1 8 3 5 0

4 3 0 1 0 2 6 9 0 6 0 5 3 5 1

4 3 0 1 0 5 8 0 1 2 2 6 3 5 6

1 2 3 4 5 6 7 8 9 10 11 12 13 14 15

通过对上述关键字序列分析，发现前5位相同，第6、8、10位上的数字变化多些，若规定地址取3位，则散列函数可取它的第6、8、10位。于是有：

$$H(430104681015355) = 480$$

$$H(430101701128352) = 101$$

$$H(430103620805351) = 328$$

$$H(430102690605351) = 296$$

$$H(430105801226356) = 502$$

3. 平方取中法（较常用）

取关键字平方后的中间几位为散列函数地址。这是一种比较常用的散列函数构造方法。

如图8-16中，随机给出一些关键字，并取平方后的第2到4位为函数地址。

关键字	(关键字) ²	函数地址
0100	00 <u>10</u> 000	010
1100	12 <u>10</u> 000	210
1200	14 <u>40</u> 000	440
1160	13 <u>70</u> 400	370
2061	43 <u>10</u> 541	310

图 8-16 利用平方取中法得到散列函数地址



4. 折叠法

将关键字分割成位数相同的几部分（最后一部分的位数可以不同），然后取这几部分的叠加和（舍去进位）作为散列函数地址，称为折叠法。

例如，假设关键字为某人的身份证号码430104681015355，则可以用4位为一组进行叠加，即有 $5355 + 8101 + 1046 + 430 = 14932$ ，舍去高位，则有 $H(430104681015355) = 4932$ 为该身份证关键字的散列函数地址。

5. 除留余数法(最常用)

该方法是用关键字序列中的关键字 k 除以散列表长度 m 所得余数作为散列函数的地址，即有 $H(k) = k \% m$ 。

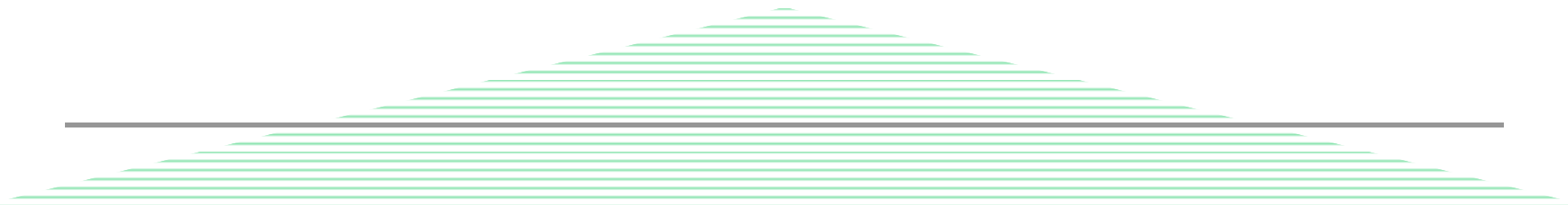


9.4.3 解决冲突的方法

由于散列存贮中选取的散列函数不是线性函数，故不可避免地会产生冲突，下面给出常见的解决冲突方法。

1. 开放定址法

开放定址法就是从发生冲突的那个单元开始，按照一定的次序，从散列表中找出一个空闲的存储单元，把发生冲突的待扞入关键字存储到该单元中，从而解决冲突的发生。





$$H_i = (H(\text{key}) + d_i) \text{ MOD } m$$

$$i=1 \sim k, k \leq m-1$$

其中：i为冲突的次数

H(key) 哈希函数

m为哈希表表长，即数组大小

d_i 递增序列

- 线性探测 $d_i=1,2,3,\dots,m-1$
- 平方探测(二次探测) $d_i=1^2, -1^2, 2^2, -2^2, \dots, \pm k^2 (k \leq m/2)$
- 随机探测 d_i 为伪随机序列

例9-4，假设有一批关键字序列20,11,14,68,19,23,10,1,84,55,27,79，给定散列函数 $H(k) = k \% 13$ ，存贮区的内存地址从0到14，处理冲突：

(1) 线性探测 (2) 二次探测

解：可以得到每个关键字的散列地址为：

$$H(20) = 20 \% 13 = 7$$

$$H(11) = 11 \% 13 = 11$$

$$H(14) = 14 \% 13 = 1$$

$$H(68) = 68 \% 13 = 3$$

$$H(19) = 19 \% 13 = 6$$

$$H(23) = 23 \% 13 = 10$$

$$H(10) = 10 \% 13 = 10$$

$$H(1) = 1 \% 13 = 1$$

$$H(84) = 84 \% 13 = 6$$

$$H(55) = 55 \% 13 = 3$$

$$H(27) = 27 \% 13 = 1$$

$$H(79) = 79 \% 13 = 1$$

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
HT1		14	1	68	55	27	19	20	84	79	23	11	10		

	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
HT2	27	14	1	68	55	84	19	20		10	23	11		79	

非同义词冲突： $key1 \neq key2$ ，而 $f(key1) \neq f(key2)$ 且 $H1 = H2$



2. 再哈希法

$$H_i = R H_i(\text{key}) \quad i=1, 2, \dots, k$$

$R H_i$ 均是不同哈希函数，即在同义词产生地址冲突时计算另一个哈希函数地址，直到冲突不再发生。这种方法不易产生“聚集”，但增加了计算时间。

3. 链地址法

链地址法也称拉链法，是把相互发生冲突的同义词用一个单链表链接起来，若干组同义词可以组成若干个单链表

例10-6 对给定的关键字序列19,14,23,1,68,20,84,27,55,11,10,79，给定散列函数为 $H(k)=k\%13$ ，试用拉链法解决冲突建立散列表。

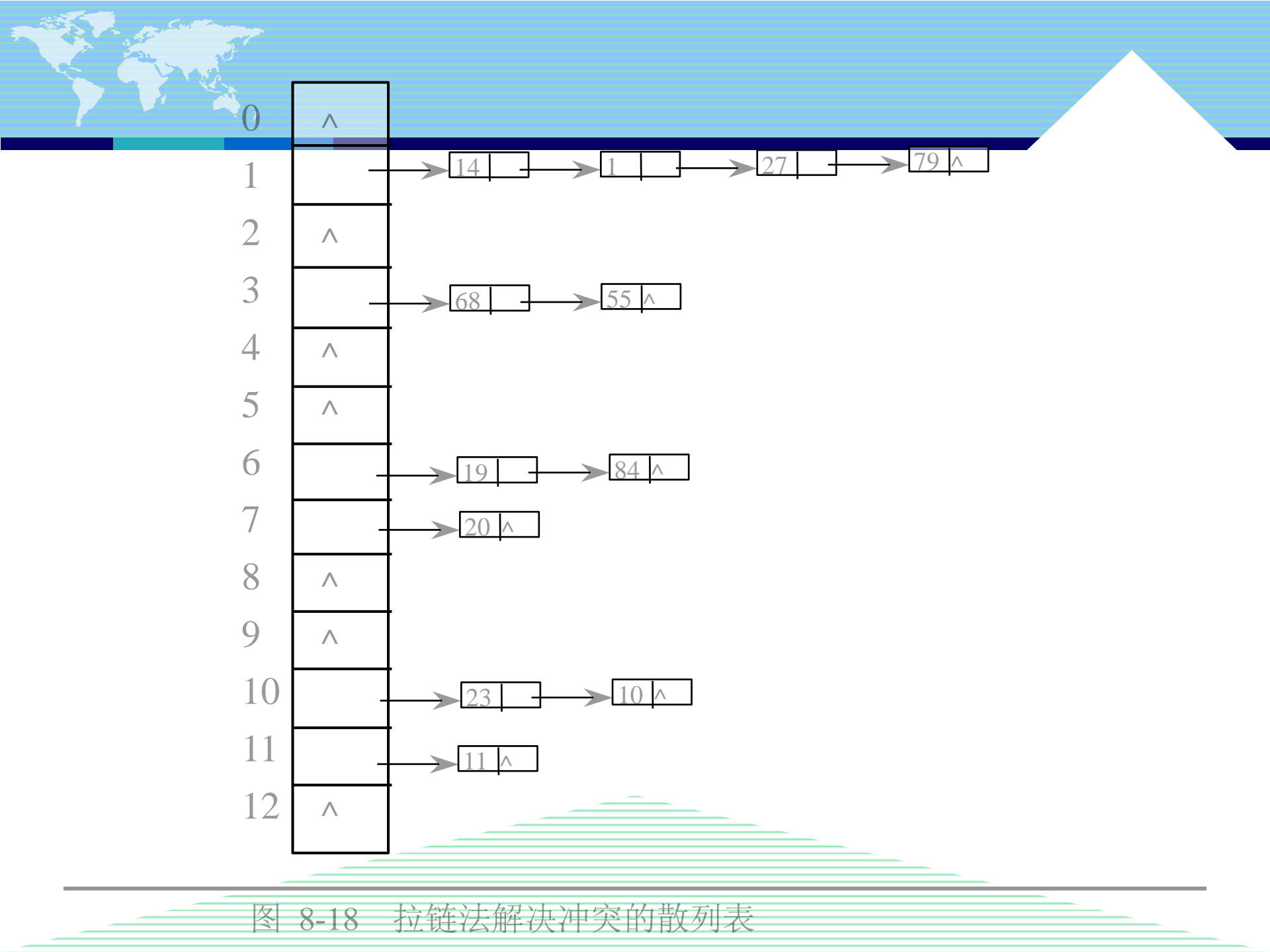



图 8-18 拉链法解决冲突的散列表



9.4.5 哈希查找的性能分析

哈希查找按理论分析，它的时间复杂度应为 $O(1)$ ，它的平均查找长度应为 $ASL=1$ ，但实际上由于冲突的存在，它的平均查找长度将会比1大。

作业：9.19

