# 第8章　查找

# 目录

# 8.1 查找的基本概念

查找，也称为检索。在我们日常生活中，随处可见查找的实例。如查找某人的地址、电话号码；查某单位45岁以上职工等，都属于查找范畴。本书中，我们规定查找是按关键字进行的，所谓关键字(key)是数据元素(或记录)中某个数据项的值，用它可以标识(或识别)一个数据元素。例如，描述一个考生的信息，可以包含：考号、姓名、性别、年龄、家庭住址、电话号码、成绩等关键字。但有些关键字不能唯一标识一个数据元素，而有的关键字可以唯一标识一个数据元素。如刚才的考生信息中，姓名不能唯一标识一个数据元素(因有同名同姓的人)，而考号可以唯一标识一个数据元素(每个考生考号是唯一的，不能相同)。我们将能唯一标识一个数据元素的关键字称为主关键字，而其它关键字称为辅助关键字或从关键字。

有了主关键字及关键字后，我们可以给查找下一个完整的定义。所谓查找，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。

因为查找是对已存入计算机中的数据所进行的操作，所以采用何种查找方法，首先取决于使用哪种数据结构来表示“表”，即表中结点是按何种方式组织的。为了提高查找速度，我们经常使用某些特殊的数据结构来组织表。因此在研究各种查找算法时，我们首先必须弄清这些算法所要求的数据结构，特别是存储结构。

查找有内查找和外查找之分。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。我们仅介绍内查找。

要衡量一种查找算法的优劣，主要是看要找的值与关键字的比较次数，但我们将找到给定值与关键字的比较次数的平均值来作为衡量一个查找算法好坏的标准，对于一个含有n个元素的表，查找成功时的平均查找长度可表示为ASL= $\sum_{i=1}^{n} p_i c_i$ ，其中Ｐi为查找第i个元素的概率，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，Ｃi为查找第i个元素所用到的比较次数。

# 8．2 线性表的查找

## 8.2.1 顺序查找

### 1．顺序查找的基本思想

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。

下面以顺序表的形式来描述算法。

## 2．顺序查找算法实现

```
struct    node
{  …;
   int   key;                                  //key为关键字，类型设定为整型
}; typedef struct node NODE;
 int  seq_search (NODE  arrry[ ],int n,int k)  //在表中查找关键字值
 { int i;                                                为K的元素, n是表的长度
     for (i=0; i<n && array[i].key!=k;i++);    //从表尾开始向前扫描
      if(i<n) return i;
     else retur(-1);
}
```

## 3.顺序查找性能分析

顺序查找的优点是算法简单，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。顺序查找的缺点是查找效率低，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。

## 8.2.2二分查找

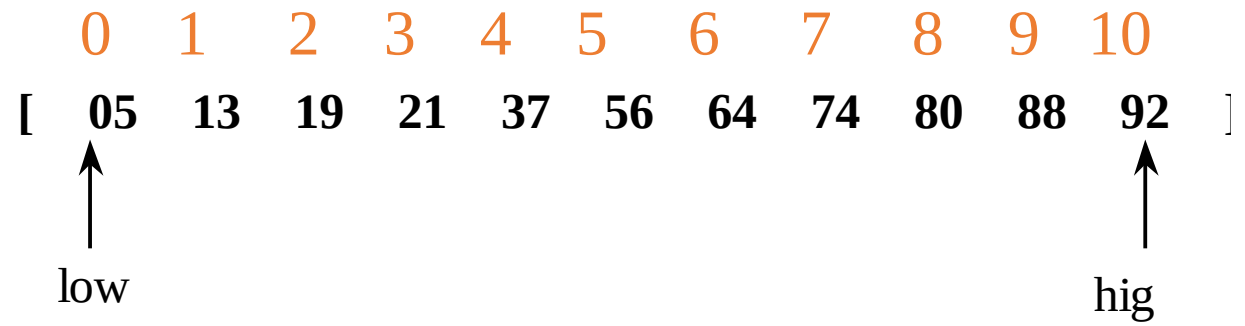
### 1.二分查找的基本思想

二分查找，也称折半查找，是一种高效率的查找方法。但要求表中元素必须按关键字有序(升序或降序)。不妨假设表中元素为升序排列。二分查找的基本思想是：首先将待查值K与有序表array[0]到array[n-1]的中点mid上的关键字array[mid].key进行比较，若相等，则查找成功；否则，若array[mid].key>k ， 则在array[1]到array[mid-1]中继续查找，若有array[mid].key<k ， 则在array[mid+1]到array[n-1]中继续查找。每通过一次关键字的比较，区间的长度就缩小一半，区间的个数就增加一倍，如此不断进行下去，直到找到关键字为K的元素；若当前的查找区间为空(表示查找失败)。

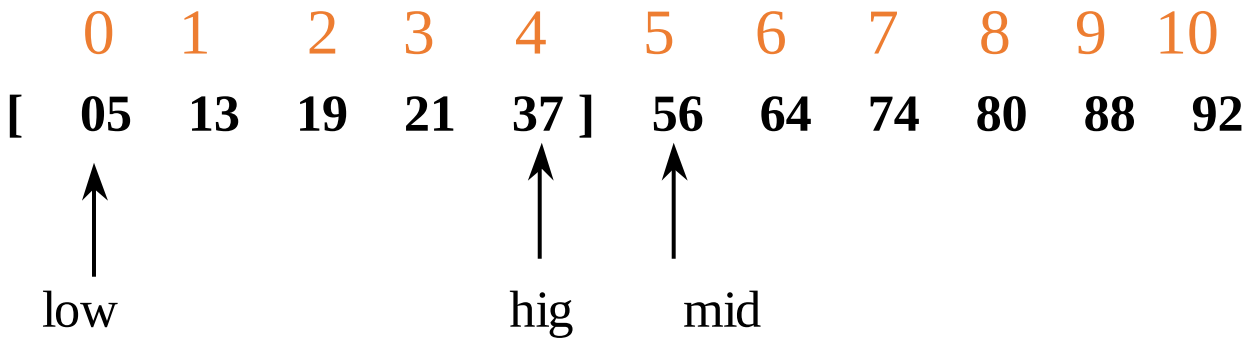
## 2．二分查找算法实现

```
int    bin_search (NODE array[ ],int n,int k)
{ int low=0,hig=n-1,mid;
 while(low<=hig)
  { mid=(low +hig)/2;                          //取区间中点
    if (array[mid].key= =k) return(mid);      //查找成功
    if (array[mid].key>k)
       hig=mid-1;                             //在左子区间中查找
        else  low=mid+1;  }                    //在右子区间中查找
       return(-1); }                           //查找失败
```

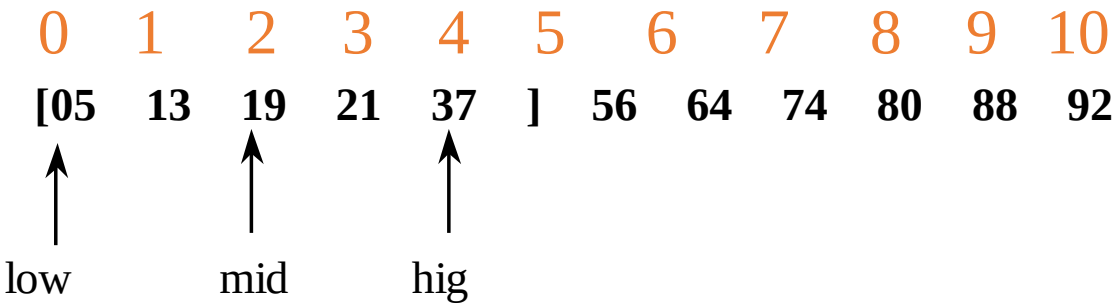
例如，假设给定有序表中关键字为05,13,19,21,37,56,64,74,80,88,92将查找K=21和K=85的情况描述为图8-1及图8-2形式。




**(a) 初始情形**

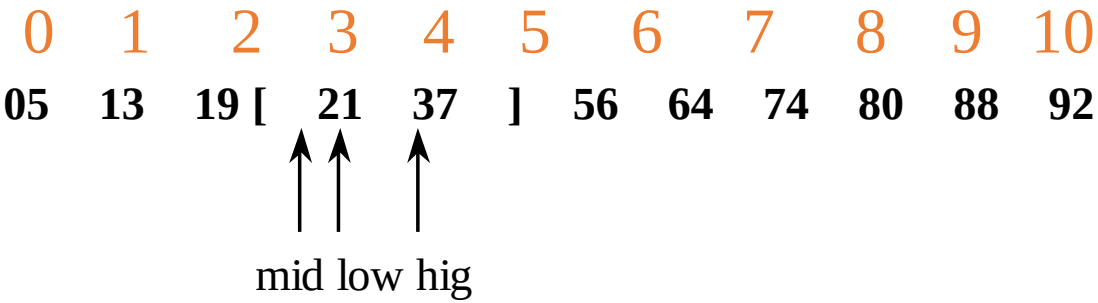



**(b) 经过一次比较后的情形**



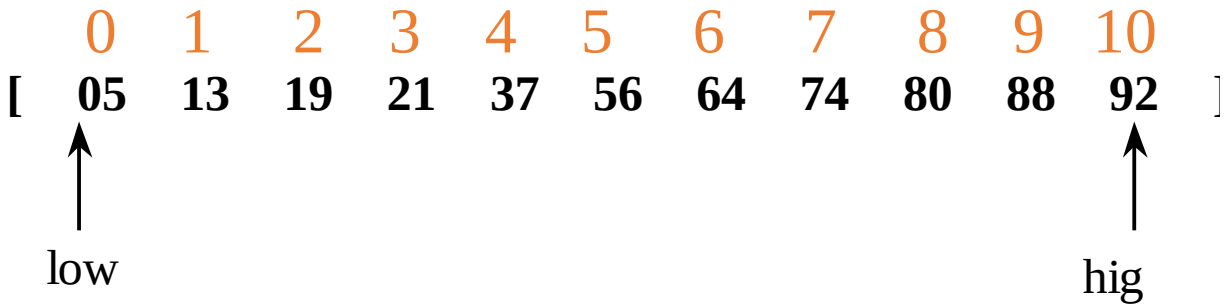

**(c ) 经过二次比较后的情形 (array[mid].key=19)**

**图 8-1 查找 K=21 的示意图**

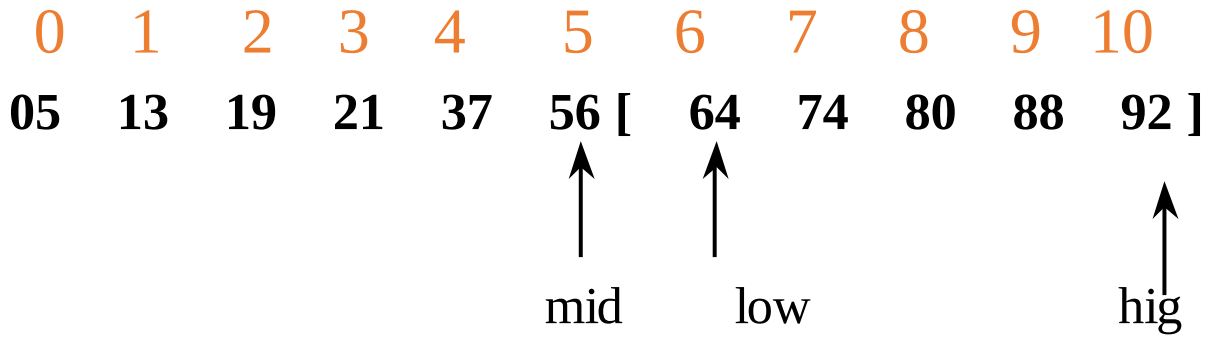



**(d ) 经过三次比较后的情形 (array[mid].key=21)**

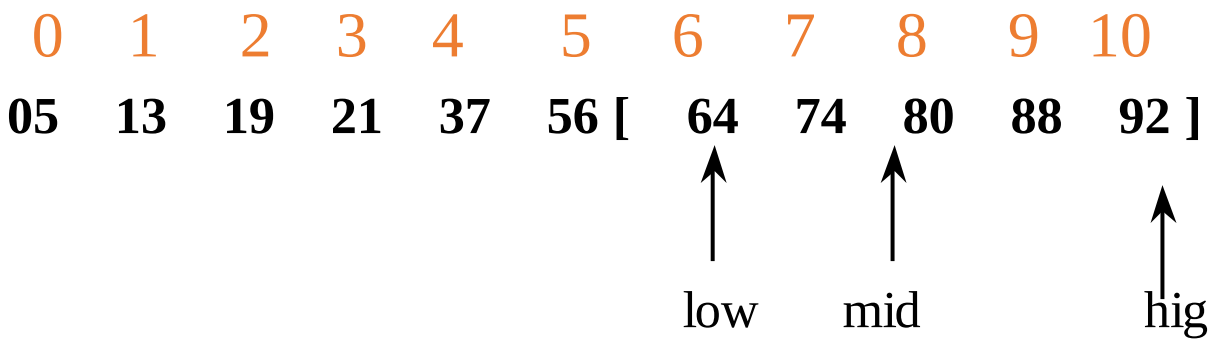
**图 8-1 查找 K=21 的示意图 (查找成功)**



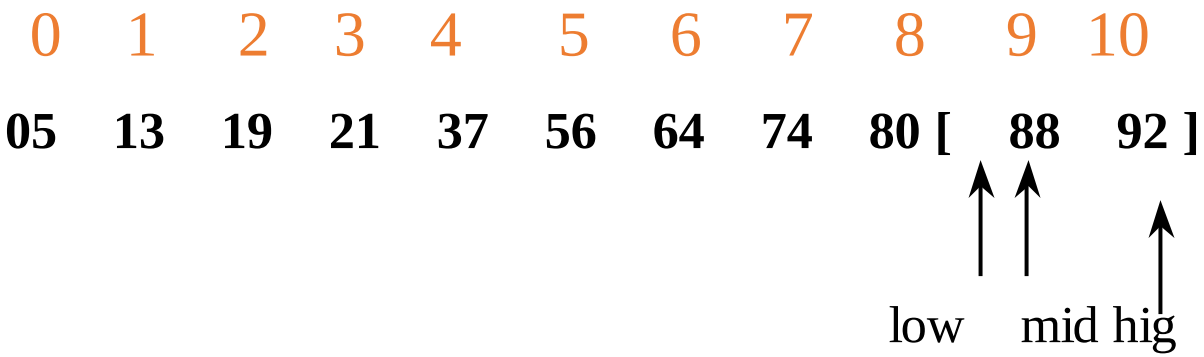

**(a) 初始情形**



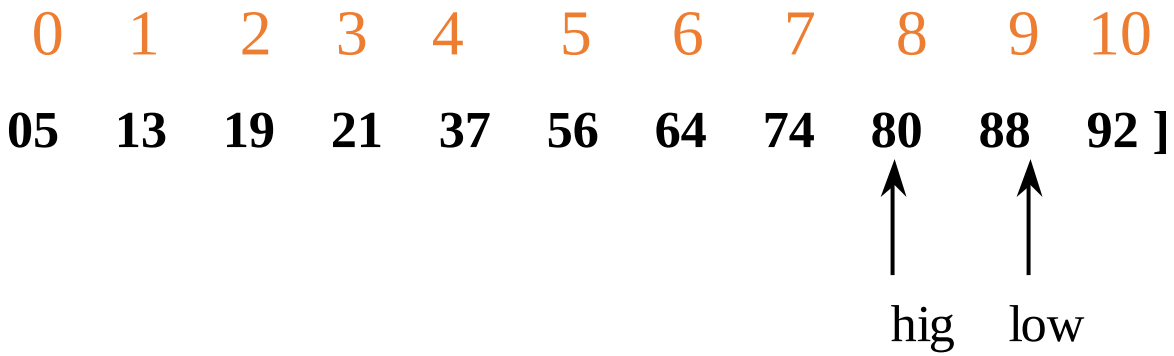

**(b) 经过一次比较后的情形**




**(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(hig<low)**

**图8-2　查找K=85 的示意图 (查找不成功)**

## 3.二分查找的性能分析

为了分析二分查找的性能，可以用二叉树来描述二分查找过程。把当前查找区间的中点作为根结点，左子区间和右子区间分别作为根的左子树和右子树，左子区间和右子区间再按类似的方法，由此得到的二叉树称为二分查找的判定树。例如，图8-1给定的关键字序列05,13,19,21,37,56,64,74,80,88,92，的判定树见图8-3。

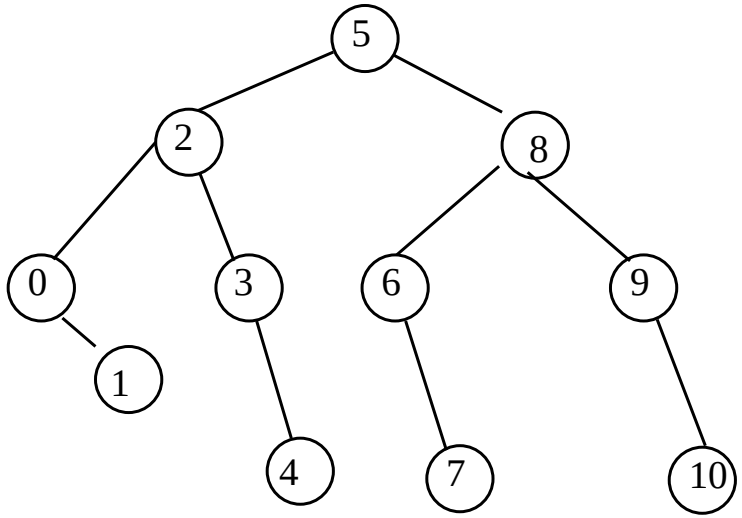



**图 8-3 具有 11 个关键字序列的二分查找判定树**

从图8-3 可知，查找21的过程是从根到结点3的路径，查找85,应在结点9的左子树上,但其左子树为空,故失败。

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。则二分查找的时间复杂度为$O(\log_2 n)$。

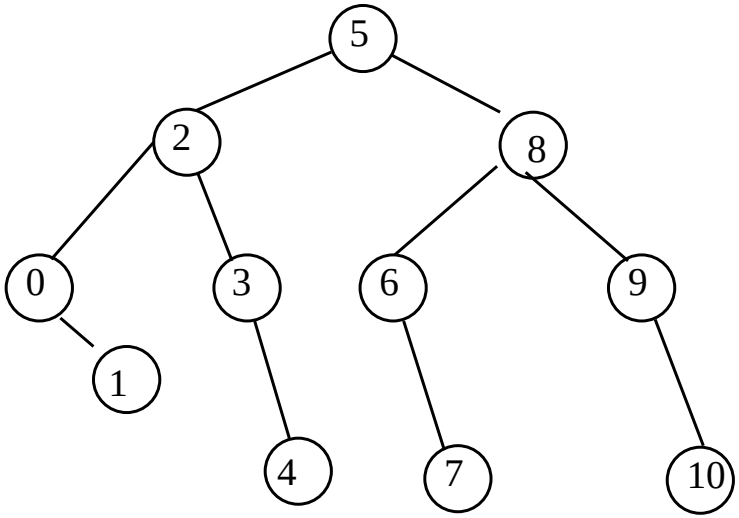



**图 8-3 具有 11 个关键字序列的二分查找判定树**

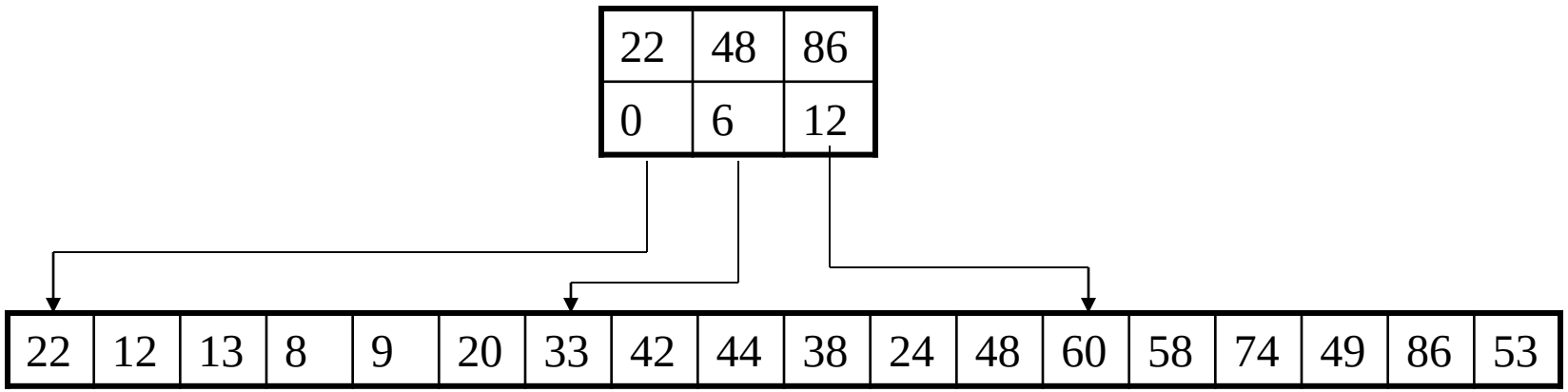
## 8.2.3分块查找

### 1.分块查找 的思想

分块查找是顺序查找的一种改进方法，又称索引顺序查找，具体实现如下：将一个主表分成n个子表，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，用每个子表最大关键字和指示块中第一个记录在表中位置建立索引表。

```
typedef struct
  { int key;
     …;
   } NODE;
typedef struct
  { int key, pos;
   } INDEX;
```

例如，给定关键字序列如下：22,12,13,8,9,20,33,42,44,38,24,48,60,58,74,49,86,53, 假设n=3，即将该序列分成3个子表，每个子表有6个元素，则得到的主表和索引表如图所 示。




分块查找的过程是先确定待查记录所在的块，然后在块中顺序查找。假设给定k=38, 则先在索引表中按顺序比较，因为22<k<48，则可确定38应该在第二块中，从第二块的第一个记录 array[6] 顺序查起.

分块查找的时间复杂度是O（m+n) m是块长度，n是索引表长度。

# 8．3 树表的查找

## 8.3.1二叉排序树查找

### 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。

## 2．二叉排序树的数据类型描述

和第六章类似，可以用一个二叉链表来描述一棵二叉排序树，具体为：

```
struct  node
    {   int key;                    //代表关键字
        …
          struct  node *lch,*rch;       //代表左、右孩子
};
```

## 4．二叉排序 树上的查找

### （1）二叉排序 树的查找思想

若二叉排树为空，则查找 失败，否则，先拿根结点值与待查值进行比较，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

```
NODE  * search(int k, NODE *root)
  //在以root为根指针的二叉排序树中查找关键值为x的结点
 {NODE *p;
   p=root;
   while(p!= =NULL)
   { if  (p->key= =k) return(p);                    //查找成功
      else if (p->key>k)  p=p->lch ;        //进入左子树查找
          else p=p->rch ;                  //进入右子树查找
     }     return(NULL);
   }
```

## 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，但比二分查找要差。

## 8.3.2平衡二叉树查找

### 1．平衡二叉树的概念

平衡二叉树(balanced binary tree)是由阿德尔森一维尔斯和兰迪斯(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。如图8-8所示二叉排序树，是一棵平衡二叉树，而如图8-9所示二叉 排序树，就不是一棵平衡二叉树。

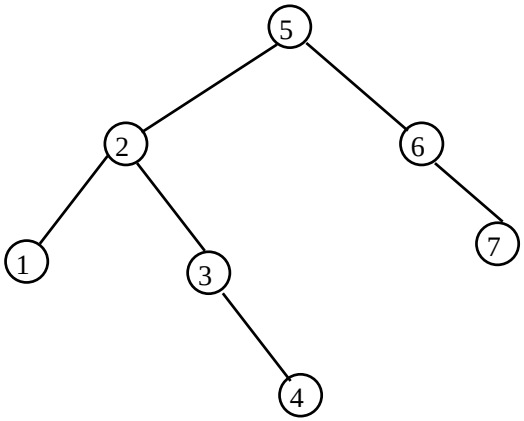



图 8-8 一棵平衡二叉树

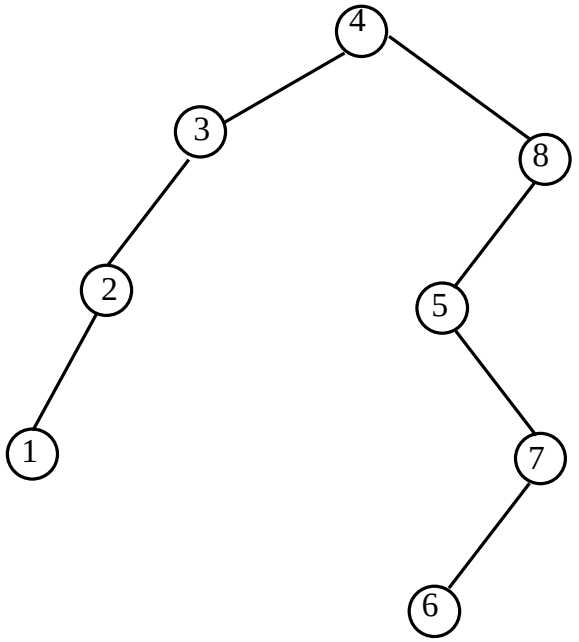



图 8-9 一棵非平衡二叉树

## 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8-10所示，在A的左孩子B上插入一个左孩子结点C，使A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A顺时针旋转，成为B的右子树，而原来B的右子树则变成A的左子树，待插入结点C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)

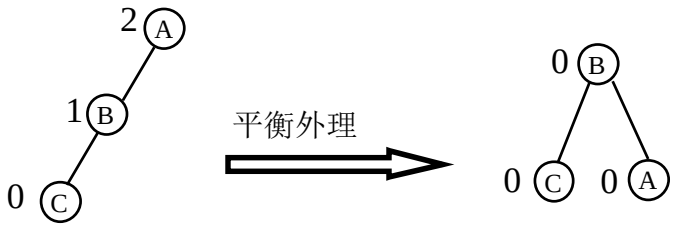



图 8-10　　LL 型平衡外理

## （2）LR型的处理(左右型)

如图8-11所示，在A的左孩子B上插入一个右孩子C，使的A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C变到B与A 之间，使之成为LL型，然后按第(1)种情形LL型处理。

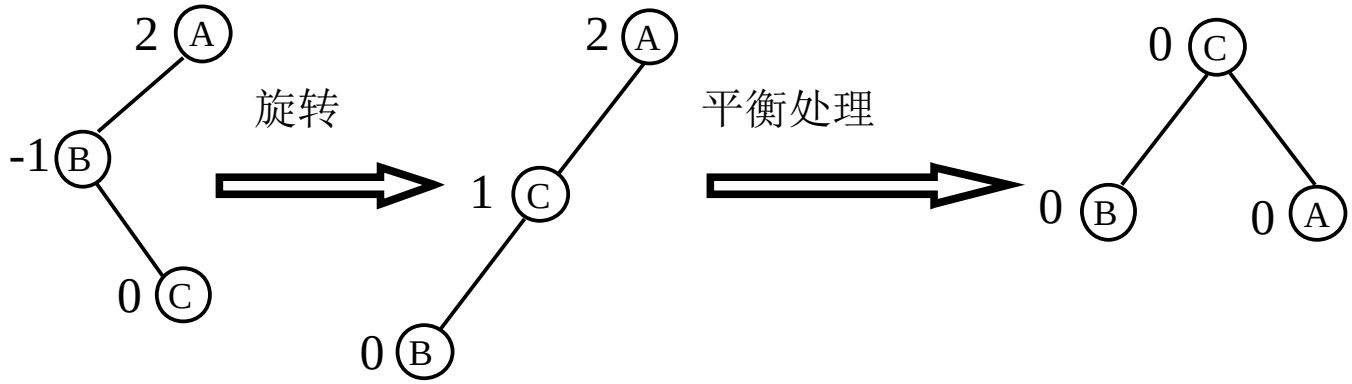



图 8-11　　LR 型平衡处理

## （3）RR型的处理(右右型)

如图8-12所示，在A的右孩子B上插入一个右孩子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，待插入结点C成为B的右子树。

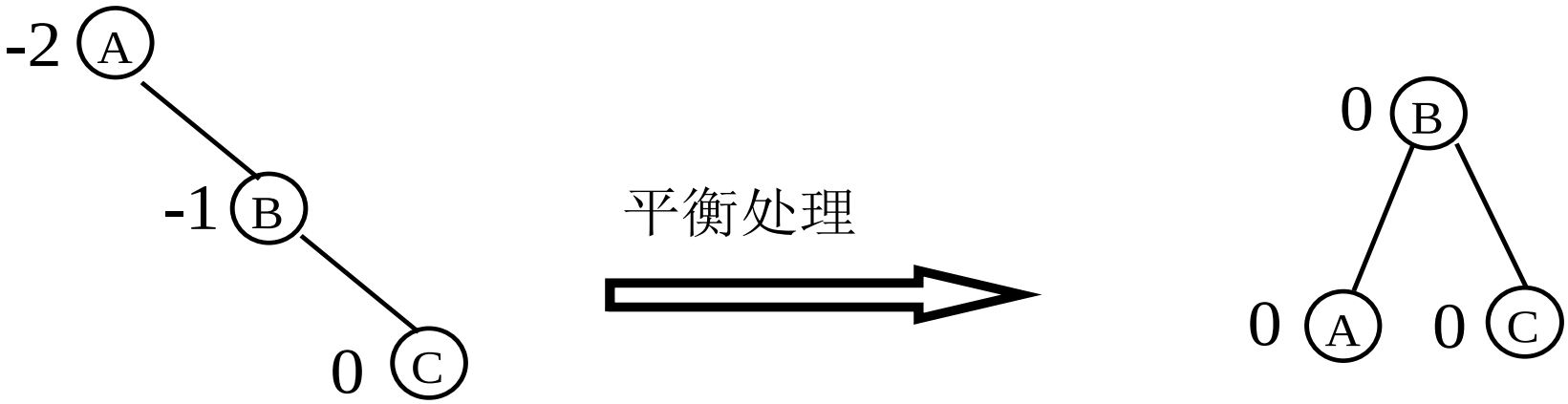



图 8-12　　RR 型平衡处理

## （4）RL型的处理(右左型)

如 图8-13所示，在A的右孩子B上插入一个左孩子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C变到A与B之间，使之成为RR型，然后按第(3)种情形RR型处理。

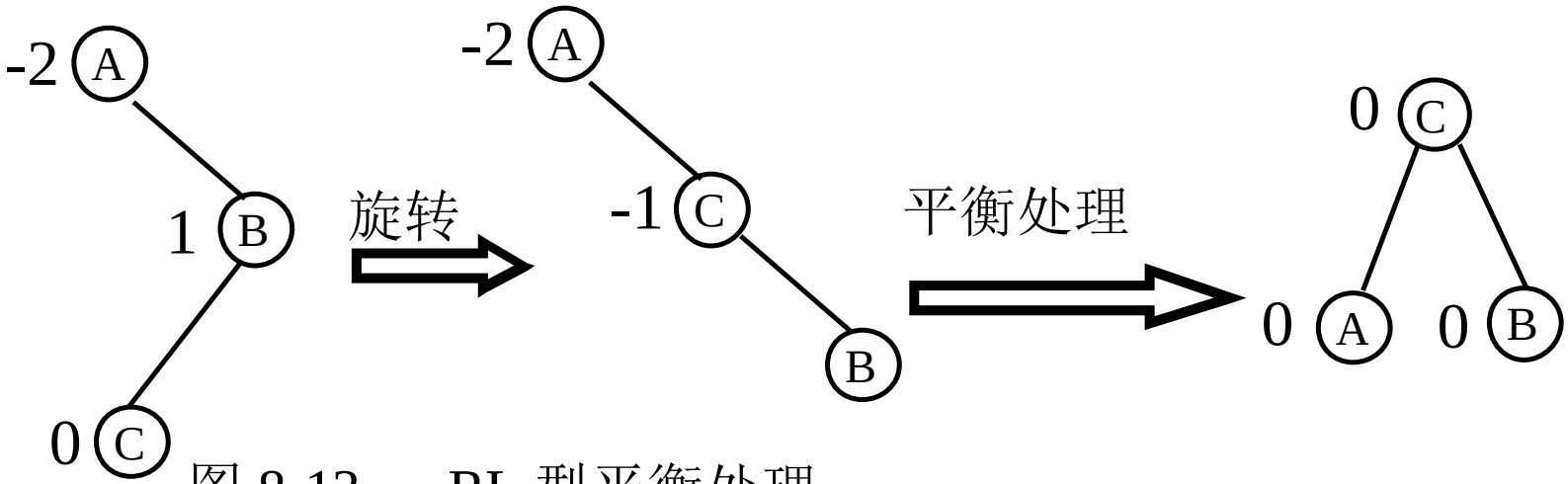
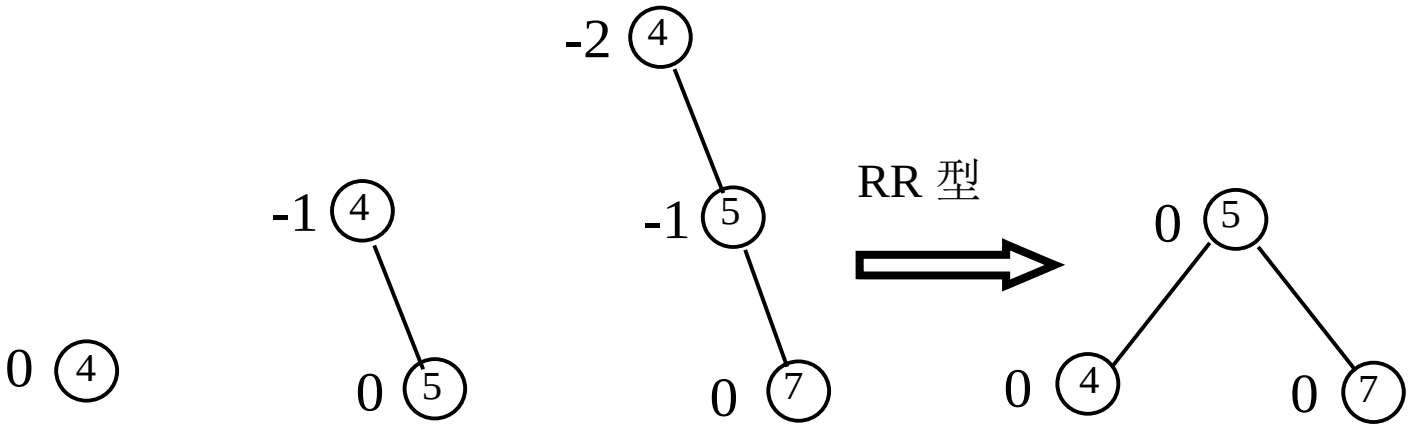


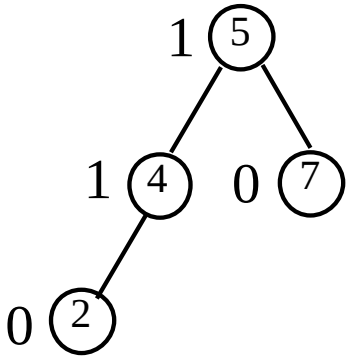

图 8-13　　RL 型平衡处理

例8-2，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。

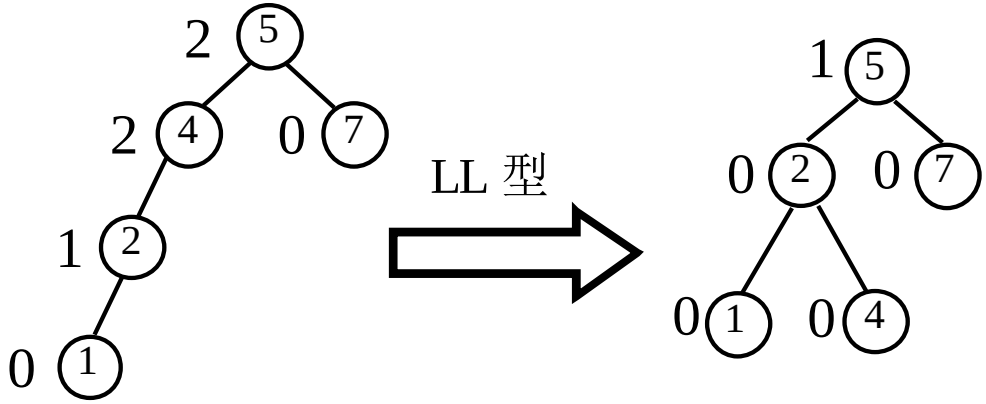
分析：平衡二叉树实际上也是一棵二叉排序树，故可以按建立二叉排序树的思想建立，在建立的过程中，若遇到不平衡，则进行相应平衡处理，最后就可以建成一棵平衡二叉树。具体生成过程见图8-14。



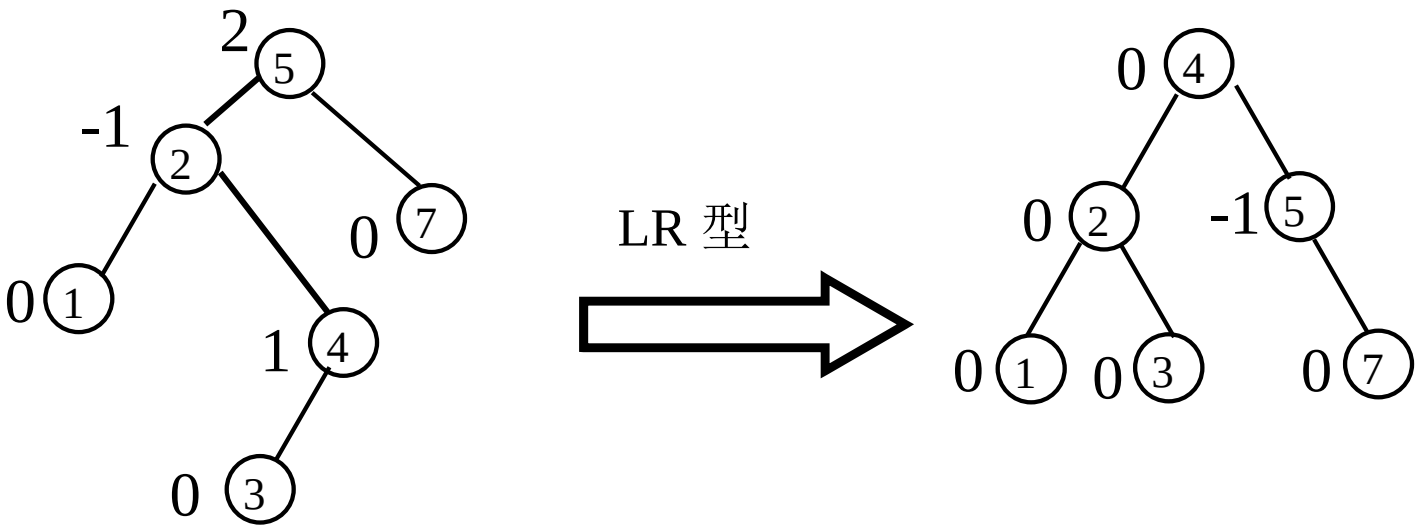

(a)插入 4　　(b)插入 5　　(c)插入 7



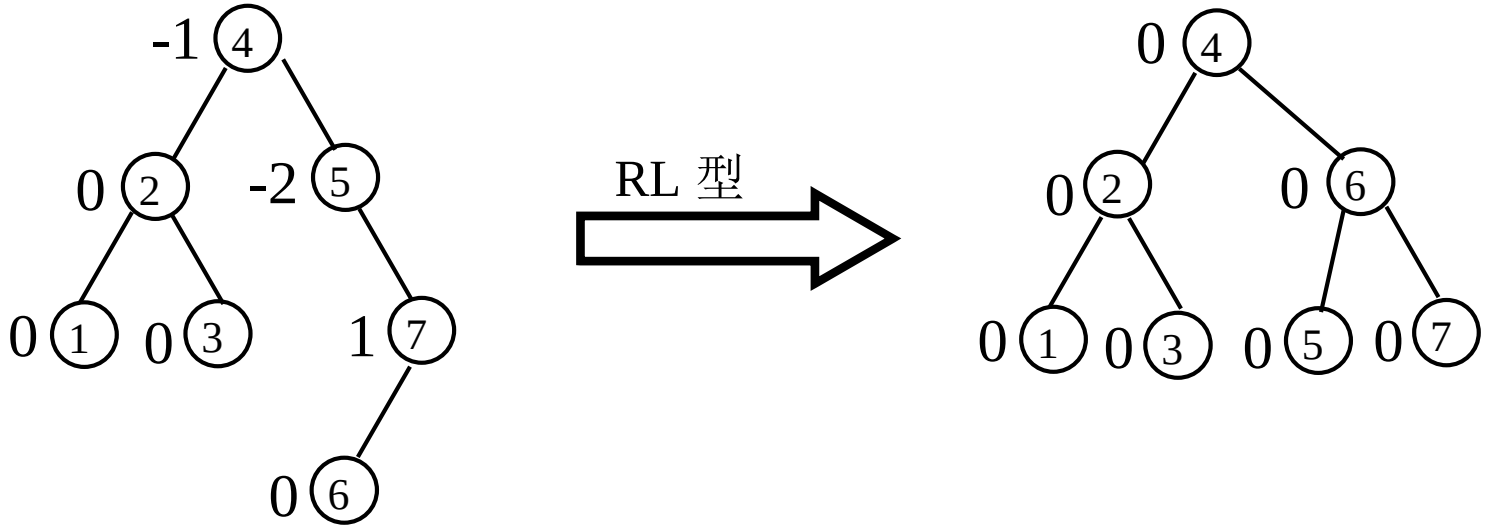

(d)插入 2




(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相同，都为$O(\log_2 n)$。

例8-3，对例8-2给定的关键字序列4,5,7,2,1,3,6，试用二叉排序树和平衡二叉树两种方法查找，给出查找6的次数及成功的平均查找长度。

分析：由于关键字序列的顺序已经确定，故得到的二叉排序树和平衡二叉树都是唯一的。得到的平衡二叉树见图8-14，得到的二叉排序树见图8-15。

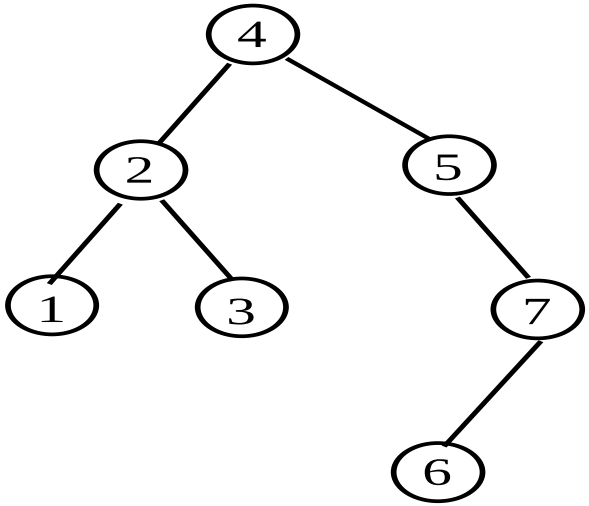



图 8-15 由关键字序列 4,5,7,2,1 ,3,6 生成的二叉排序树

从图8-15的二叉排序树可知，查找6需4次，平均查找长度ASL=(1+2+2+3+3+3+4)/7=18/7≈2.57。

从图8-14的平衡二叉树可知，查找6需2次，平均查找长度

ASL=(1+2+2+3+3+3+3)=17/7≈2.43。

从结果可知，平衡二叉树的查找性能优于二叉排序树。

## 8.3.3 B- 树

B-树是一种平衡的多路查找树，在文件系统中很有用，定义如下：

一棵m阶的B-树，或是空树，或是满足以下条件的m叉树：

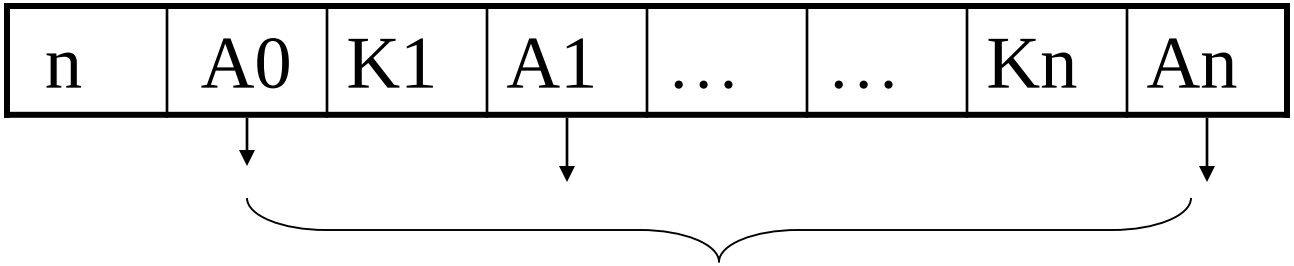
（1）树中每个结点至多有m棵子树

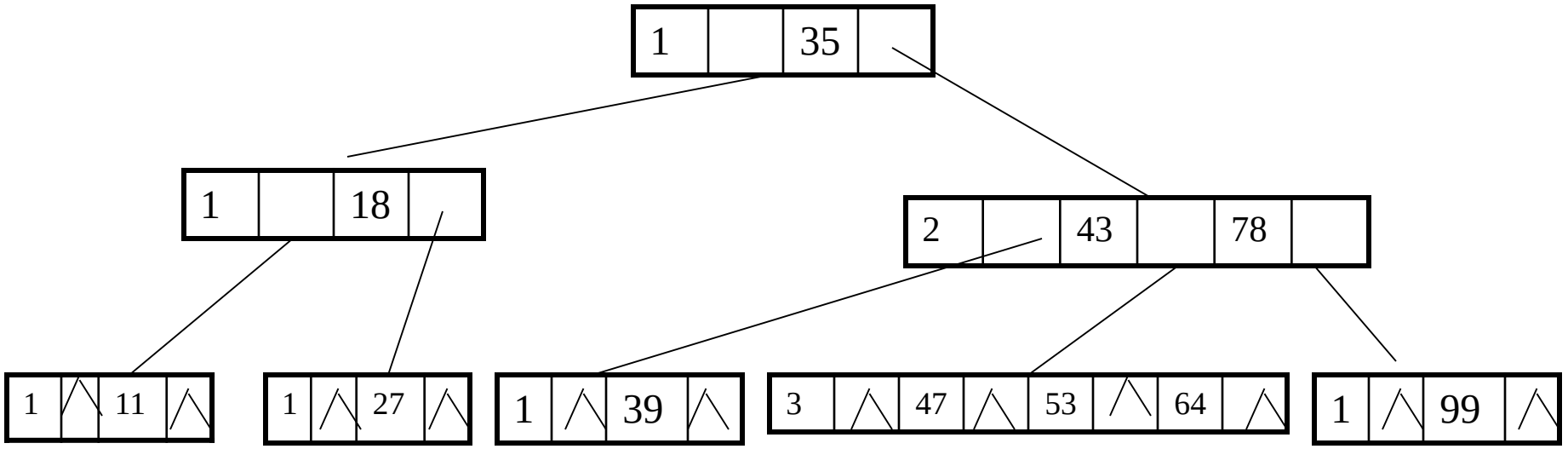
（2）若根结点不是叶子结点，则至少有二棵子树。

（3）除根结点外的所有非终端结点至少有int(m/2)棵子树

（4）所有结点包含信息（n,A0,K1,A1,…Kn,An)其中Ki为关键字且有序。Ai为指向子树根结点的指针，Ai所指子树中所有结点的关键字均小于Ki+1,An所指子树中所有结点的关键字均大于Kn

（5）所有叶子结点都出现在同一层次上，即所有空的指针出现在同一层上。


n
A0
K1
A1
…
…
Kn
An
n+1个分支


1
35
1
18
2
43
78
1
11
1
27
1
39
3
47
53
64
1
99

## B-树的操作

（1）查找

要查找关键字k的记录，首先从根结点开始，若找到则找所对应的记录，否则沿p所指的子树继续查找，其中

$$P = \begin{cases} A0 & k<K1 \\ An & k>Kn \\ Ai & Ki<k<Ki+1 \end{cases}$$

若直到叶子结点还未找到，则查找失败。

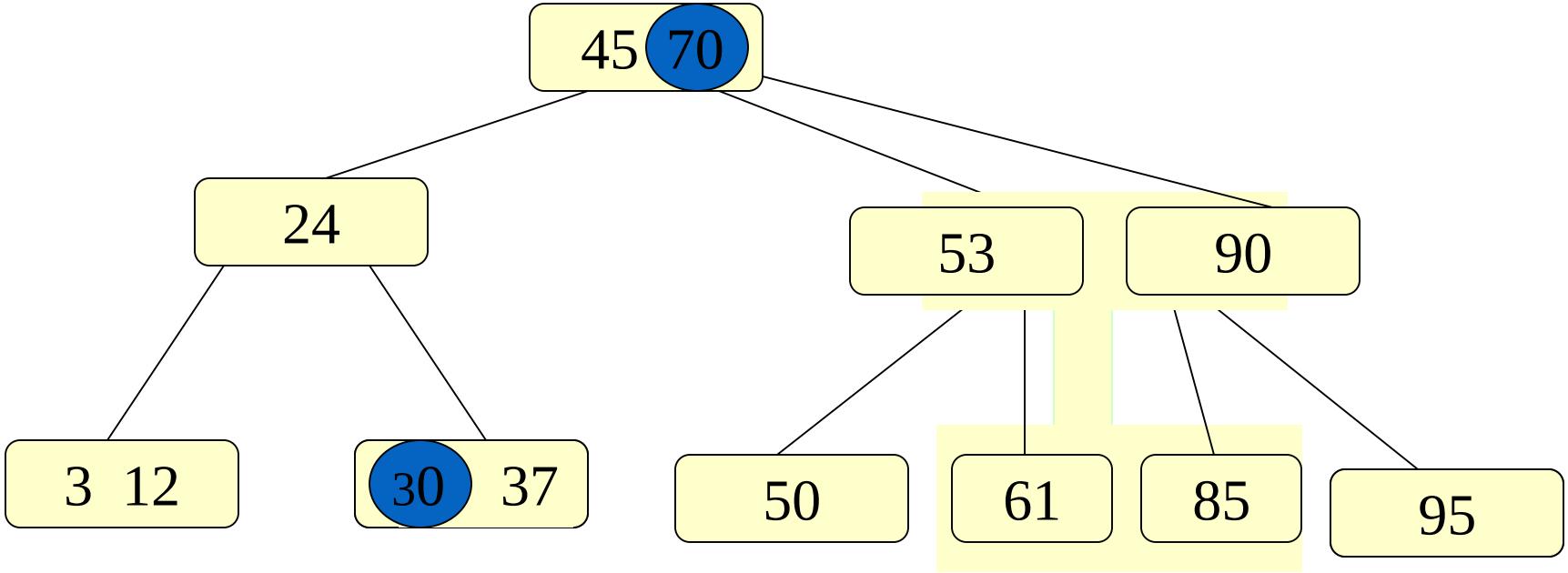
## B-树的操作

（2）插入

设要插入关键值为k的记录，指向k所在记录的指针为p。

首先找到k应插入的叶子结点，将 k和p插入。然后，判断被插入结点是否满足m叉B-树的定义，即插入后结点的分支数是否大于m（结点的关键字数是否大于m-1)，若不大于，则插入结束；否则，要把该结点分裂成两个。方法是：

申请一个新结点，由指针p’指向，将插入后的结点按照关键字的值大小分成左、中、右三部分，中间只含一项，左边的留在原结点，右边的移入新结点，中间的构成新的插入项，插入到它们的双亲结点中，若双亲结点在插入后也要分裂，则在分裂后再往上插入。




插入30

插入85

# 8．4　哈希表（散列查找）

## 8.4.1 基本概念

散列查找，也称为哈希查找。它既是一种查找方法，又是一种存贮方法，称为散列存贮。散列存贮的内存存放形式也称为哈希表或散列表。

散列查找，与前面介绍的查找方法完全不同，前面介绍的所有查找都是基于待查关键字与表中元素进行比较而实现的查找方法，而散列查找是通过构造哈希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。

例如，要找关键字为k的元素，则只需求出函数值H（k），H（k）为给定的哈希函数，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，假设有一批关键字序列18,75,60,43,54,90,46，给定哈希函数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

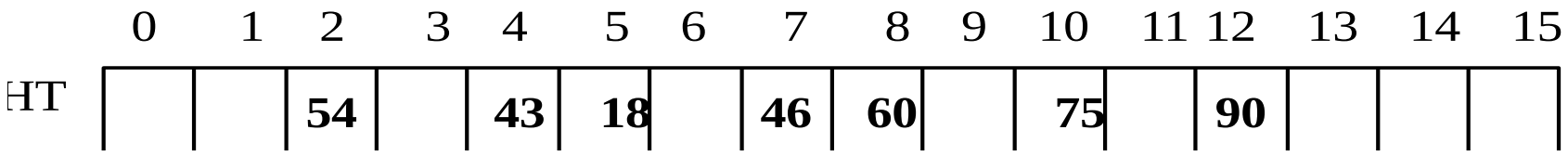
H（43）=43%13=4

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（哈希表或散列表）中，具体表示为：




其中HT就是散列存贮的表，称为散列表或哈希表。从哈希表中查找一个元素相当方便，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。

为了保证哈希表查找得以实现，必须使记录的存放规则和查找规则一致，即：使用同样的哈希函数。在存储时，以每个记录的关键字为自变量，通过哈希函数计算出存储地址，将该记录存放在存储地址对应的存储单元中。在查找时，以查找值为自变量，通过哈希函数计算出地址，从该地址所对应的存储单元中取出记录数据。

关键字序列为19，14，23，1，68，20，84，27，55，11，10，79，

哈希函数H（k）=k%13

上面讨论的哈希表是一种理想的情形，即每一个关键字对应一个唯一的地址。但是有可能出现这样的情形，两个不同的关键字有可能对应同一个内存地址，即两个记录的关键值不等，但它们的哈希函数的值相同，这样，将导致后放的关键字无法存贮，我们把这种现象叫做冲突（collision）。在散列存贮中，冲突是很难避免的，除非构造出的哈希函数为线性函数。哈希函数选得比较差，则发生冲突的可能性越大。我们把相互发生冲突的关键字互称为“同义词”。

使用哈希方法，首先要选择一个好的哈希函数，使一组关键值所得到的哈希地址能均匀分布在整个地址空间中，并且冲突次数尽可能地少。

在哈希存贮中，若发生冲突，则必须采取特殊的方法来解决冲突问题，才能使哈希查找能顺利进行。虽然冲突不可避免，但发生冲突的可能性却与三个方面因素有关。第一是与装填因子α有关，所谓装填因子是指哈希表中已存入的元素个数n与哈希表的大小m的比值，即α=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。第二是与所构造的哈希函数有关(前面已介绍)。第三是与解决冲突的方法有关，这些内容后面将再作进一步介绍。

## 8.4.2 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散列空间上，同时使计算尽可能简单，冲突次数少。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：

**9 9 3 4 6 5 3 2**

**9 9 3 7 2 2 4 2**

**9 9 3 8 7 4 3 3**

**9 9 3 0 1 3 6 7**

**9 9 3 2 2 8 1 7**

**9 9 3 3 8 9 6 7**

**9 9 3 5 4 1 5 7**

**9 9 3 6 8 5 3 7**

**9 9 3 6 8 5 3 2**

通过对上述关键字序列分析，发现前3位相同，第8位只可取2、3、7，因此，这四位不可取。中间的四位的数字变化多些，可看成是随机的，若规定地址取3位，则哈希函数可取它的第4、5、6位。于是有：

H（99346532）＝465

H（99372242）＝722

H（99387433）＝874

H（99301367）＝016

H（99322817 ）＝228

## 3．平方取中法

取关键字平方后的中间几位为哈希函数地址。这是一种比较常用的哈希函数构造方法，但在选定哈希函数时不一定知道关键字的全部信息，取其中哪几位也不一定合适，而一个数平方后的中间几位数和数的每一位都相关，因此，可以使用随机分布的关键字得到哈希函数地址。

如图中，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 | 0010000 | 010 |
| 1100 | 1210000 | 210 |
| 1200 | 1440000 | 440 |
| 1160 | 1370400 | 370 |
| 2061 | 4310541 | 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）作为哈希函数地址，称为折叠法。

例如，假设关键字为某人身份证号码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哈希函数地址。

## 5．除留余数法

该方法是用关键字序列中的关键字k除以一个整数p所得余数作为哈希函数的地址，即

H（k）＝k％p 。　P<=m　m为哈希表长度

除留余数法计算简单，适用范围广，是一种最常使用的方法。这种方法的关键是选取较理想的p值，使得每一个关键字通过该函数转换后映射到散列空间上任一地址的概率都相等，从而尽可能减少发生冲突的可能性。一般情形下，p 取为一个素数较理想，并且要求装填因子α最好是在0.6∽0.9之间，所以p 最好取1.1n∽1.7n之间的一个素数较好，其中n为哈希表中待装元素个数。

**6. 随机法**

选择一个随机函数，取关键值的随机函数值为它的哈希地址，即　　H(key)=random(key)

random()不能是一般的随机函数，固定的参数必须返回确定的值。

## 8.4.3 解决冲突的方法

由于哈希存贮中选取的哈希函数不是线性函数，将大的关键值的取值空间映射到小的地址空间中，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，从哈希表中找出一个空闲的存储单元，把发生冲突的待插入关键字存储到该单元中，从而解决冲突的发生。在哈希表未满时，处理冲突需要的“下一个”空地址在哈希表中解决。

开放定址法利用下列公式求“下一个”空地址

$$Hi=(H(key)+di)\ MOD\ m \quad i=1,2,\ldots K(K<=m-1)$$

其中H（key)为哈希函数，m为哈希表长度，di为增量序列

根据di的取法，解决冲突时具体使用下面一些方法。

## （1）线性探查法

假设哈希表的地址为0∽m-1，则哈希表的长度为m。若一个关键字在地址d处发生冲突，则依次探查d+1，d+2，…，m-1(当达到表尾m-1时，又从0，1，2，….开始探查)等地址，直到找到一个空闲位置来装冲突处的关键字，将这一种方法称为线性探查法。假设发生冲突时的地址为$d_0$=H(k)，则探查下一位置的公式为$d_i=(d_{i-1}+1)\%m$ $(1\le i\le m-1)$，最后将冲突位置的关键字存入$d_i$地址中。

例8-5　给定关键字序列为19，14，23，1，68，20，84，27，55，11，10，79，哈希函数H（k）=k%13 ，哈希表空间地址为0∽12，试用线性探查法建立哈希存贮(哈希表)结构。

得到的哈希表如图8-17所示

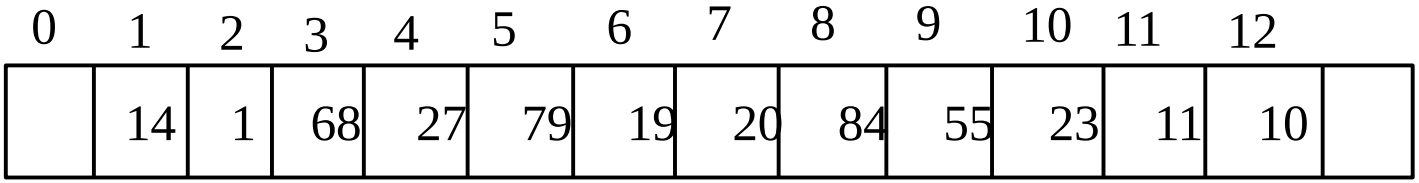



图 8-17　用线性探查建立的哈希表

## (2) 二次方探查法

该方法规定，若在d地址发生冲突，下一次探查位置为$d+1^2$，$d-1^2$,$d+2^2$，$d-2^2$，…，直到找到一个空闲位置为止。

开放地址法充分利用了哈希表的空间，但在解决一个冲突时，可能造成下一个冲突。另外，用开放地址法解决冲突不能随便对结点进行删除。

## 2. 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：对给定的关键字序列19,14,23,1,68,20,84,27,55,11,10,79，给定哈希函数为H(k)=k%13，试用拉链法解决冲突建立哈希表。

图8-18为用尾插法建立的关于例8-6的拉链法解决冲突的哈希表。

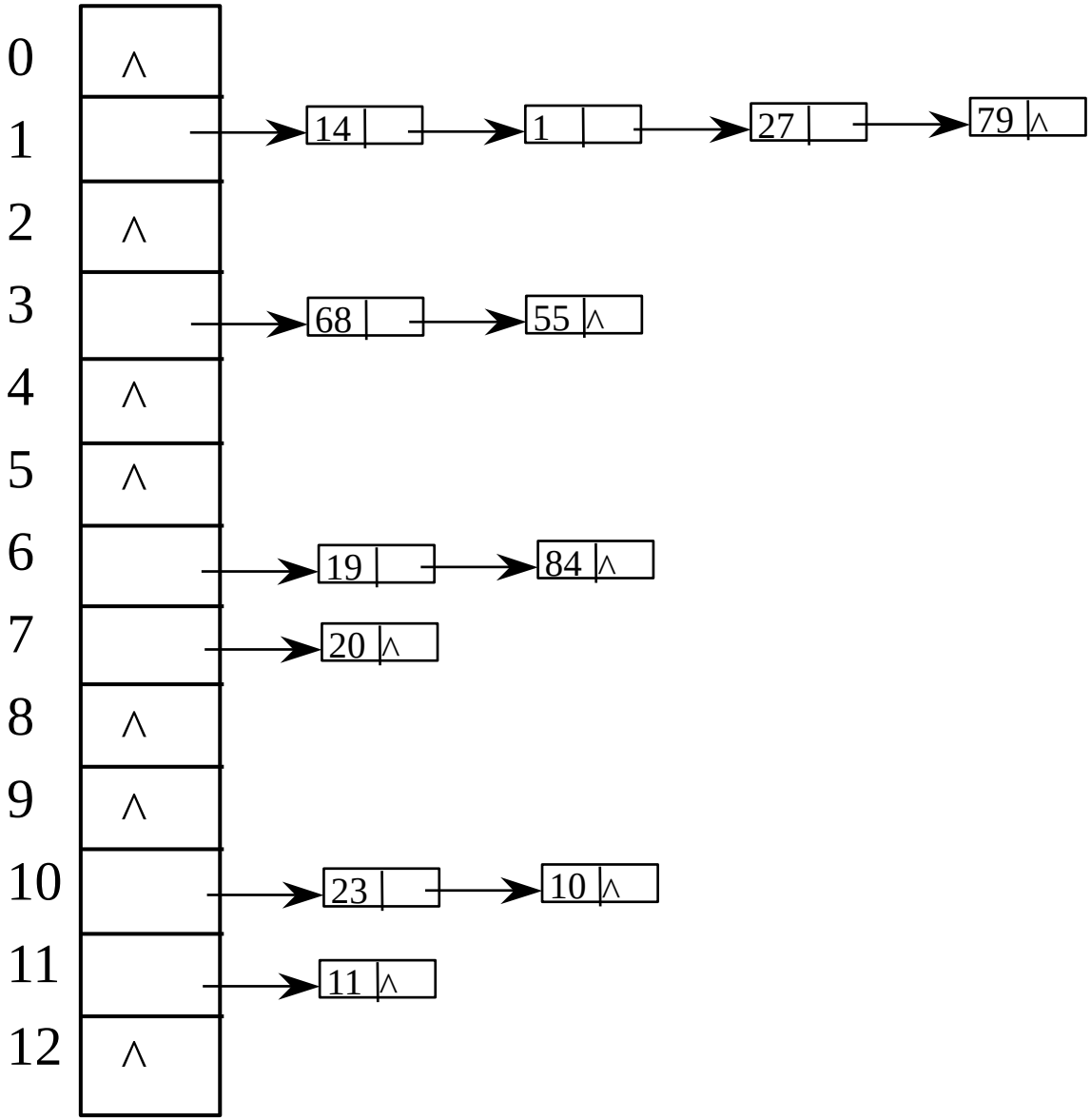



图 8-18　拉链法解决冲突的散列表

## 8.4.5 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。下面将分析几种方法的平均查找长度。

### 1.线性探查法的性能分析

由于线性探查法解决冲突是线性地查找空闲位置的，平均查找长度与表的大小m无关，只与所选取的哈希函数H及装填因子α的值和该处理方法有关，这时的成功的平均查找长度为ASL=1/2 (1+1/(1- α)) 。

### 2.拉链法查找的性能分析

由于拉链法查找就是在单链表上查找，查找单链表中第一个结点的次数为1，第二个结点次数为2，其余依次类推。它的平均查找长度ASL=1+α/2。

例8-7 给定关键字序列11，78，10，1，3，2，4，21，试分别用顺序查找、二分查找、二叉排序树查找、平衡二叉树查找、哈希查找(用线性探查法和拉链法)来实现查找，试画出它们的对应存储形式(顺序查找的顺序表，二分查找的判定树，二叉排序树查找的二叉排序树及平衡二叉树查找的平衡二叉树，两种哈希查找的哈希表)，并求出每一种查找的成功平均查找长度。哈希函数H(k)=k%11。

顺序查找的顺序表（一维数组）如图8-19所示，

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| 11 | 78 | 10 | 1 | 3 | 2 | 4 | 21 | | | |

图 8-19 顺序存储的顺序表

从图8-19可以得到顺序查找的成功平均查找长度为：

ASL=(1+2+3+4+5+6+7+8)/8=4.5；

二分查找的判定树（中序序列为从小到大排列的有序序列）如图8-20所示，

11，78，10，1，3，2，4，21

1,2,3,4,10,11,21,78

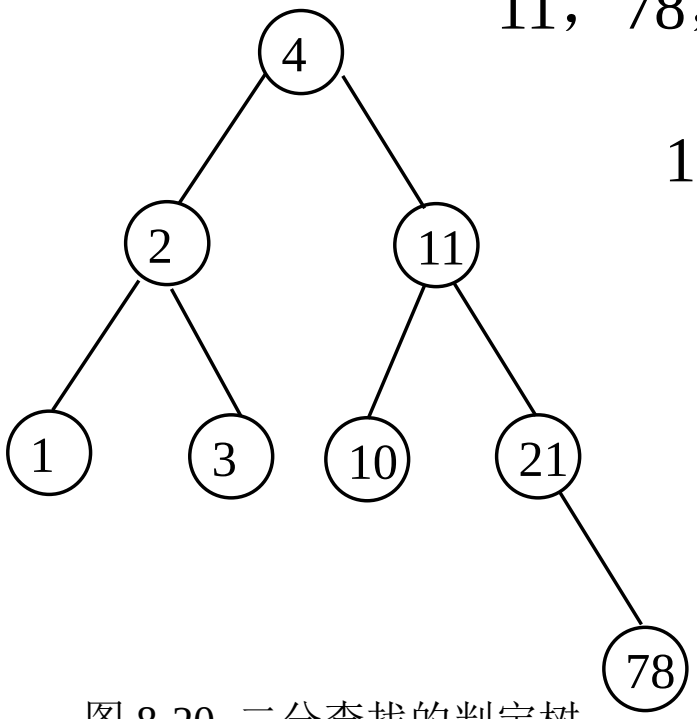
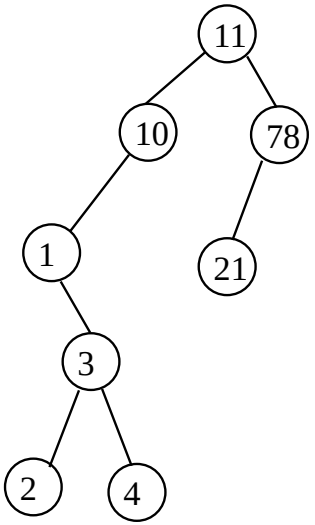



图 8-20 二分查找的判定树

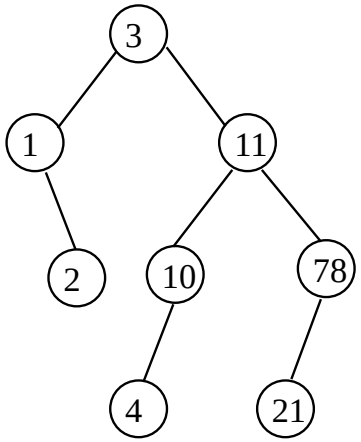
从图8-20可以得到二分查找的成功平均查找长度为：

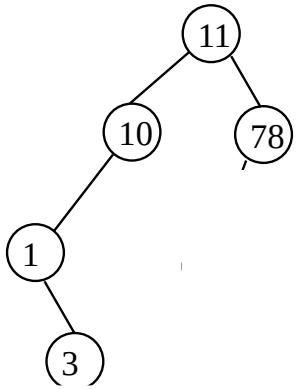
ASL=(1+2*2+3*4+4)/8=2.625；

二叉排序树（关键字顺序已确定，该二叉排序树应唯一）如图 8-21(a)所示，平衡二叉树（关键字顺序已确定，该平衡二叉树也应该是唯一的），如图8-21(b)所示，




(a)二叉排序树




11，78，10，1，3，2，4，21

(b)平衡二叉树

图 8-21 二叉排序树及平衡二叉树

从图8-21(a)可以得到二叉排序树查找的成功平均查找长度为：

ASL=(1+2*2+3*2+4+5*2)/8=3.125；

从图8-21(b)可以得到平衡二叉树的成功平均查找长度为：

ASL=(1+2*2+3*3+4*2)/8=2.75；

线性探查法解决冲突的哈希表如图8-22所示，

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|
| 11 | 78 | 1 | 3 | 2 | 4 | 21 | | | | 10 |

图 8-22 线性探查的散列表

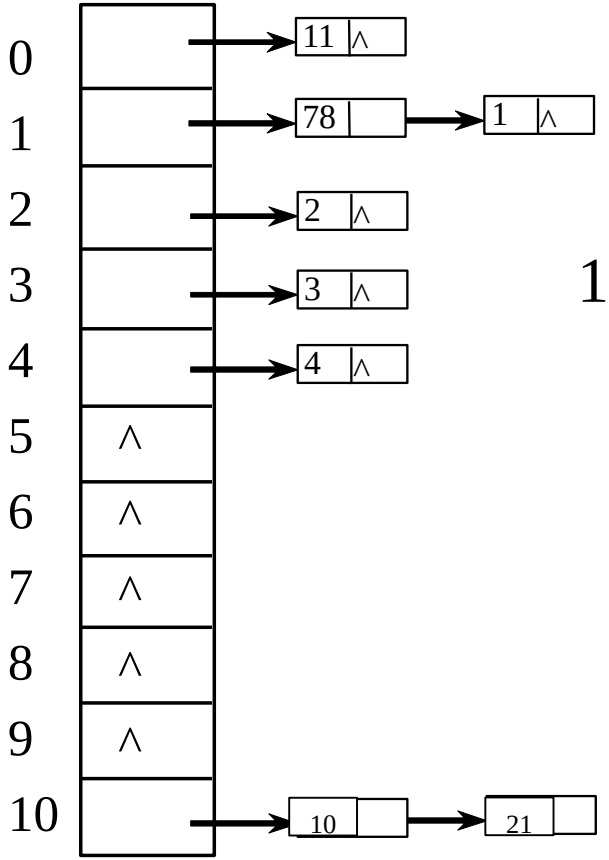
从图8-22可以得到线性探查法的成功平均查找长度为：

ASL=（1+1+2+1+3+2+1+8）/8=2.375；

11，78，10，1，3，2，4，21

1+1+1+2+1+3+2+8

拉链法解决冲突的哈希表如图8-23所示。




11，78，10，1，3，2，4，21

图 8-23 拉链法的哈希表

从图8-23可以得到拉链法的成功平均查找长度为：

ASL=(1*6+2*2)/8=1.25。