第 9 章 检索结构

检索（也称查找）是使用最频繁的一种基本操作。小到算法中符号表的维护，大到数据库系统的信息检索，都涉及到检索操作。因此，检索操作的实现效率变得十分重要，以至于常常要根据检索的需要，设置专门的数据结构——面向检索/查找的数据结构——**检索结构**。这就是本章要讨论的主要内容。

§9.1 概述

§9.1.1 检索的概念

广义地讲，检索(Search)是指在数据元素（记录）集合中求出满足某给定条件的记录。由于这里的“条件”可能是多种多样的，所以很难对这类检索给出通用的高效算法。为了提高检索效率，常常对“条件”加以限制或具体化为“匹配”，这后者对应的检索，就是在数据元素（记录）中确定某特定数据字段的值与给定值相匹配的记录，这个数字字段称为关键字字段，简称**关键字**。本章中主要讨论这种“狭义”检索。

一般情况下，关键字是指在某问题中被指定为记录标识的数据字段，对检索，它是搜索的目标，对排序，它是确定记录次序的根据。

若记录集合中找到了关键字字段与指定关键字值匹配的记录，则称该次检索成功，否则为不成功。一般情况下，检索成功时，返回一个成功标志，或返回所找到的记录的位置或值；不成功时返回一个不成功标志。

在某些问题中，当检索不成功时要插入不存在的数据记录，或在某种情况下删除所找到的记录，因此检索操作也涉及到记录的插入与删除操作。

检索算法（方法）有多种多样，可如下分类：

△按检索操作是否全部在内存进行：

- 内检索：被查对象全部在内存；
- 外检索：被查对象很多，不能一次全部存入内存，在检索操作进行中，有部分留在外存。

△按是否增删元素：

- 静态检索：单纯检索，不涉及插入与删除元素；
- 动态检索：在检索中，要插入新记录（元素），或删除某些老记录。

△按是否进行比较操作：
• 比较式检索：检索中要通过关系运算（比较）检索对象
• 非比较式检索：常为计算式检索，即根据给定关键字的特征计算被查对象的位置
△按关键字是否变化
• 原词检索：用原关键字进行检索
• 变词检索：将关键字变换后再用在检索中

§9.1.2 检索结构

一般而言，各种数据结构都会涉及到检索操作，如前面介绍的线性表（连续存贮结构、链式存贮结构）、特殊线性表（栈、队、串）、数组、广义表、树与图等均可能需要进行检索操作，这类检索操作并没有被做为主要操作考虑，它的实现服从于数据结构。但是，在某些应用中，检索操作上升到主要地位，为了提高检索效率，要专门为检索操作设置数据结构，这种数据结构面向检索操作，我们称之为检索结构。如语言编译程序中用到的符号表，数据库系统中用到的数据缓冲区等均主要面向检索操作。
常用的检索结构有若干类。若按数据元素集合中元素间结构关系分类，检索结构可分为下列 4 类
- 线性结构（含线性链结构）
- 线性索引结构
- 树形结构
- 散列（杂凑）结构

若按检索结构中数据元素是否会增加或减少分类，检索结构可分成下列两大类
• 静态检索结构：操作中不增加或减少元素
• 动态检索结构：操作中可能增加元素或减少元素

§9.1.3 检索算法的时间与空间复杂度分析

检索算法的空间复杂度的分析与其它算法相比，没有什么特别之处。对于为了方便实现检索算法而构造专门的数据结构的算法，它们的空间复杂度中应考虑那些专门构造出来的数据结构的存贮占用量，对于仅在现有结构上进行检索的算法，辅助存贮量的占用是很小的，一般不需考虑。
检索算法的时间复杂度分析则有所不同。首先，对比较型检索算法，其主要运行时间花在关键字比较上，所以，应以关键字的比较次数为主度量算法的时间复杂度。但比较次数又与哪些因素相关呢？显然，除与算法本身及问题规模（数据集的大小）相关外，还与待查关键字在数据集中的位置（包括不出现在数据集中的情况）相关。对同一数据集、同一算法，待查关键字所处位置不同，比较次数就不同。所以，时间复杂度应为问题规模与待查关键字在数据集中位置的函数，即 T(n,k).
然而，对检索算法，以个别关键字衡量速度性能是不完全的，有时也是很繁的。一般来

讲，我们关心的是它的整体性能，即对各个位置上的关键字的综合性能。但待查关键字的位置对算法本身而言是个随机量，所以要综合测定算法的检索次数，需使用检索次数的数学期望。我们将检索算法的检索次数的数学期望称为平均检索长度 ASL（Average Search Length）。对检索成功（关键字在表中）情况，其计算式为：

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

这里，n 为数据量（记录个数），p_i 为检索第 i 个元素的概率，其应满足 $\sum_{i=1}^{n} p_i =1$；c_i 为检索第 i 个关键字所需的检索（比较）次数。

显然，c_i 与算法密切相关，决定于算法，而 p_i 与算法无关，决定于具体应用。如果 p_i 是已知的，则 ASL 只是 n 的函数。

对检索不成功的情况（关键字不在数据元素集中），平均检索长度即为检索失败对应的比较次数。

检索算法的总的平均检索长度应为检索成功与失败两种情况的检索长度的平均。

§9.1.4 检索算法的判定树

为了便于分析检索算法的时间性能，引入一种称为判定树的方法（结构），来描述检索算法的活动过程。

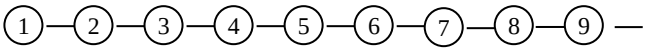
判定树中每个结点表示被查数据元素集中的一个元素（常用元素编号表示）。从树根到某一结点的路径，就表示检索该结点所经历的过程，路径上各结点为检索中测试（比较）过的结点。树结点的各个儿子，表示从该结点起下步检索的各选择分枝（根据上次比较结果选择某一儿子继续进行比较）。对无儿子结点，表示检索过程进行到它（即与它比较）后，不能继续进行，应终止。

若某检索算法 A 在数据元素集 D 上进行检索，则 A 对 D 的（检索）判定树定义为：

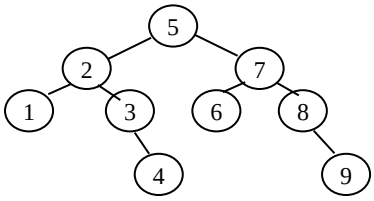
- 若 D=空，则 A 对 D 的判定树为空。
- 若第 1 次是与元素 i_1 比较（测试），则令 i_1 为判定树的根。
- 若与 i_1 比较后，下一步的检索操作是在数据集 $D_1,...,D_m$ 之一中进行（即可能在 $D_1,...,D_m$ 中任意一个上进行），则令算法 A 对 $D_1,...,D_m$ 的判定树（子树）为根结点 i_1 的子树。这里，$D_1,...,D_m$ 应为(D-{i_1})的一个划分。 （即 $D_1 \cup ... \cup D_m$=D-{i_1}，且 $D_j \cap D_k$=空，$j \neq k$，$j, k \in \{1,...,m\}$）。

例如，图 9-0（a）对应的是线性表的顺序检索算法的判定树。它实质上蜕化为一个线性结构，每个结点只有一个儿子，表示不进行分块检索，体现了顺序检索的特点。

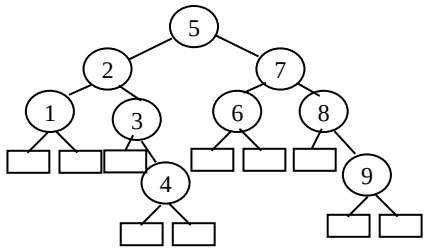
图 9-0（b）对应的是有序线性表的折半检索算法的判定树（数据集中有 9 个元素），树根为 5，表示算法第 1 次与 5 号元素比较，结点 5 有两个儿子 2 与 7，表示与结点 5 比较后，下步或者与结点 2 比较，或者与结点 7 比较，其它结点的含意类似。



（a）顺序查找判定树



（b）折半查找判定树



（c）折半查找判定树(带外结点)

图 9-1 判定树

显然，判定树的深度就表示检索算法的最大比较次数。树的分枝数越大（即数据集分块越多），树深度就越小。

检索某结点的比较次数，就是从树根到该结点的路径上的结点个数。有时，为了讨论检索不成功的情况，给判定树中的每个空链域分别增设一个结点，这种增设的结点称为**外结点**。为区别，一般用方框或双圆圈表示外结点。图 9-0（c）就是图 9-0（b）对应的带外结点的判定树。

显然，增设了外结点的判定树为完全二叉树，叶子均为外结点，叶子数目（外结点数目）等于内结点数目加 1。

为了分析检索速度，常引入内路径长与外路径长的概念。一棵判定树的**内路径**长定义为该树中各内结点的路径长（从根到某结点的路径长定义为该结点的内径长）之和；而**外路径**长定义为树中各外结点的路径长之和。

设 I 与 E 分别代表判定树的内与外路径长，则有

$$E=I+2n$$

这里 n 为内结点数目。该式可用数学归纳法证得。

内外路径长可分别用来表示检索成功与不成功的比较次数的算术合计。实质上对应的是一种等概率检索（每个关键字的被查概率均相等）的比较次数的统计量。但实际中，各关键字被查概率可能不同，为此，引入加权内外路径长。

设 w_i 为结点 i 的权值，l_i 为在树中的层次数（即结点 i 的路径长），则定义 w_i 与 l_i 的积为 i 的加权路径长。而**加权内路径长**定义为树中各内结点的加权路径长之和； **加权外路径长**定义树中各外结点的加权路径长之和，即

$$\text{加权内路径长 } I=\sum_{i=1}^{n} w_i l_i$$

加权外路径长 $E=\sum_{i=n+1}^{2n+1} w_i l_i$

这里，假设内结点编号为 1~n，外结点编号为(n+1)~(2n+1)。
对检索算法，内结点权 w_i 一般为结点的被查概率，而外结点 i 的 w_i 为外结点 i 对应的值集的被查概率。
由于结点的路径长即为自身的比较次数，又 w_i 代表被查概率，所以加权内外路径分别为检索成功与不成功的平均检索长度。

§9.2 线性结构的检索

本节讨论如何直接（即不构造专门的检索结构）在线性结构中进行检索。这属于静态检索。这种检索算法简单，但速度不高，主要用在对小型数据集的检索。
一般的线性表有两种存贮结构：连续式与链式。对它们的检索，一般采用顺序搜索的方式——称为顺序查找。对连续结构，若元素已按关键字排序，则可采用更高效的检索法，如折半查找。
在下面的叙述中，为了突出主题，我们对顺序（连续）结构的线性表，简单地假定其为一维整型数组。

§9.2.1 顺序检索

(一) 顺序结构的顺序检索

顺序检索方式是指依元素的存储次序逐个检查每个元素。这种检查可以是从左到右或从右到左。对于无序表，如果直接检索（即不构造附加的数据结构），一般只能按这种方式。无序表是指元素内容未排序的线性表。

在这种检索方式下，可以通过设置“哨兵”结点减少条件判断，从而提高速度。这里，哨兵就是待查的关键字值，它应该放置在检索方向的“尽头”处，使得在检索中不需要判断是否已检索到“尽头”（因为，此时的哨兵的值等于关键字的值，所以，检索到哨兵时，检索就自然地停了下来）。

下面的程序是关于顺序结构的线性表（用一维数组）的顺序检索的算法示意。它从左到右（从下标最小的元素起，依次向下标增大的方向检查）依次检查每个元素。哨兵设在最后一个元素的后面。

```
long SeqSearch(int a[], long n, int key)
{ //顺序存储结构的线性表的顺序查找
  // a[]——输入量，一维数组，充当线性表，n 为元素个数。要求 a[n]为空闲位
```

```
 // key——待查关键字
 //返回——成功时返回 key 在表中的序号，否则返回 n(超出范围)
     long i;
     a[n]=key;   //设置哨兵
     i=0;
     while ( a[i]!=key)    i++;
     return i;
}
```

分析一下上述算法的平均检索长度。在该算法中，检索表中 i 号元素所需的比较次数（循环次数）c_i 为(i+1)，这里 n 为表中元素个数。现假定每个记录被查的概率是相等的，则 p_i=1/n，从而由定义知，检索成功时的平均检索长度为:

$$ASL=\sum_{i=0}^{n-1} p_i c_i$$

$$=\sum_{i=0}^{n-1}\frac{1}{n}(i+1)=\frac{n+1}{2}=O(n)$$

显然，对于不成功的检索，比较次数为(n+1)，总的平均检索长度为

$$ASL_{总}= q_1(n+1)+ q_0(n+1)/2$$

这里 q_0 与 q_1 分别为检索成功与不成功的概率，若它们是等概率的，则

$$ASL_{总}=\frac{1}{2}(n+1)+\frac{1}{2}\frac{n+1}{2}=\frac{3(n+1)}{4}$$

(二) 线性链表的顺序检索

对线性链表的检索，一般只能用顺序方式。检索方法与连续结构的线性表类似。假定链表的首结点指针为 pHead，则在链表中检索关键字值为 key 的结点的过程为：

```
p=pHead;
while (p!=NULL && p→key!=key) p=p→next;
return p; //p 为 NULL 时失败，否则 p 指向所查到的结点
```

也可以设立一个监视哨结点，以免去对（p=!NULL）的测试。

§9.2.2 折半检索

如果表中元素已按关键字排了序，则称其为有序表。对这种表的针对排序关键字字段的检索，有一种更有效的算法——折半检索。

(一) 折半检索算法

折半检索（Binary Search）算法利用了元素按关键字有序排列的特点。它的基本思想是，先找出有序表中点位置上的元素（表空时无中点，表示检索不成功），用待查关键字与它比较，若相等，则已查到，结束，否则，若它大于待查关键字，则下步在有序表的前半部（值小于中点元素的部分）中按类似方法检索，若它的值小于待查关键字，则在表的后半部按类似方法检索。具体实现方法见下面的程序。

```
long BinSearch(int a[], long n, int key)
{//有序表（升序）的折半查找的非递归算法
//a[]——输入量，一维数组，充当有序线性表（连续结构），表中元素按关键字 key 的升序排列，
//元素个数为 n，0 号位置空置
//key——输入量，待查关键字
//返回——返回所找到的关键字的位置号，找不到时返回 0
long    low, high, mid;

low=1;high=n;
while(low<=high)
{
mid=(low+high)/2;   //求中点
if (key==a[mid] ) return mid;
else
      if ( key < a[mid]) high=mid-1;   //下次在前半部分查找
else low=mid+1; //下次在后半部分查找
}
return 0;     //查找失败
}//BinSearch
```

上面的算法是非递归的。现在考虑递归算法。递归算法很自然，可以直接依照折半查找的定义给出。但注意的是，为了递归，函数参数中需要有表示当前检索范围的参数。下面是具体的递归程序。

```
long BinSearch2(int a[], long low, long high, int key)
{//有序线性表（连续存储结构）的折半查找的递归算法
  //a[]----同前
  //long, high——输入量，指定查找范围，分别为有序表的查找范围的低端和高端的序号
//  返回——同前
   long mid;
```

```
 if (low>high)    return 0;          //查不到
 mid=(low+high)/2;

if (key==a[mid]) return mid;
if (key<a[mid])    return BinSearch2(a, low,mid-1, key);     //在前半部分查找
else return BinSearch2(a,mid+1,high, key); //在后半部查找
}//BinSearch2
```

(二) 折半检索判定树*

折半检索的判定树颇具特点，这里借助它分析一下折半检索算法。根据前面关于判定树的定义，折半检索判定树的根为第 1 次找到的中点结点，而左右子树分别为在前半部与后半部中进行折半检索对应的判定树，若前/后半部为空，则对应的子树为空。图 9-0（b）给出了一个具有 9 个结点的有序表的折半检索判定树。折半检索判定树具有下列性质：

①任一结点的左右子树中的结点数目最多相差 1

②任意两棵折半检索判定树，若它们的结点数目相同，则它们的结构完全等同（全等）

③具有 n 个结点的折半检索判定树的深度为$[\log_2 n]+1$，即深度与同结点数目的顺序二叉树相同。

④折半检索判定树是平衡二叉树，即任一结点的左右子树的深度最多相差 1。

⑤折半检索判定树中任两个叶子所处层次号之差不超过 1。

证：性质①与②可直接由折半检索判定树的定义推出。

对性质③，我们使用强归纳法证明。容易验证，当 n=1,2,3,4 时，性质③成立。现设性质③对所有的 n≤k 均成立，那么考查 n=k+1 的情况，由折半检索判定树定义知，这(k+1)个结点的判定树的左右子树的结点数目分别为[k/2]与[k/2]+1，它们均小于或等于 k，由归纳假设知，它们的深度分别为（注：这里用[x]表示不大于 x 的最大整数）$[\log_2[k/2]]+1$ 与 $[\log_2([k/2]+1)]+1$（注：请检查划线处是“[”还是“「」”？？？？？----答：最好的符号是方括号去掉上方的直角，这里采用方括号是因为在 Word 环境下无这种符号，所以，下面可以不变，或统一换为无上角的方括号）

其中后者大于或等于前者，故这(k+1)个结点的判定树的深度为：

$$[\log_2([k/2]+1)]+2$$

若 k 为奇数，则

$$[\log_2([k/2]+1)]+2=[\log_2(k+1)/2]+2$$
$$=[\log_2(k+1) - \log_2 2]+2$$
$$=[\log_2(k+1)]+1$$

若 k 为偶数，则

$$[\log_2([k/2]+1)]+2=[\log_2 (k/2)+1]+2$$
$$=[\log_2 k - \log_2 2+1]+2$$

$=[\log_2 k]+2$

$=[\log_2(k+1)]+1$

这表明当 n=k+1 时性质③亦成立，由归纳法知，对任何 n，性质③均成立。

在上式最后一步的推导中，用到了$[\log_2 k]=[\log_2(k+1)]$(k 为偶数)的结论。事实上，对任何整数 k，总存在整数 i，使

$2^i \leqslant k < 2^{i+1}$

因 k 为偶数，则(k+1)为奇数，故

$2^i < k+1 < 2^{i+1}$（注：若 k 为奇数，则可能有 $k+1=2^{i+1}$，从而不能推得此式）

从而

$i < \log_2(k+1) < i+1$

即

$[\log_2(k+1)]=i$

再从式 $2^i \leqslant k < 2^{i+1}$ 推知 $i \leqslant \log_2 k < i+1$ 即

$[\log_2 k]=i$

比较推得的$[\log_2(k+1)]=i$ 与$[\log_2 k]=i$，知

$[\log_2(k+1)]=[\log_2 k]$ (k 为偶数)

对于性质④，因为折半检索判定树中任一结点的左右子树中的结点个数最多相差 1，故左右子树的深度最多相差：

$[\log_2(n+1)]+1 - ([\log_2 n]+1) = [\log_2(n+1)]-[\log_2 n]$

因对任意 x，有 $x-1<[x]\leqslant x$，故上式的最大值为

$\log_2(n+1)-(\log_2 n-1) = \log_2((n+1)/n) + 1 = \log_2(1+1/n)+1 < \log_2 2+1=2$

即左右子树深度之差不超过 2。

至于性质⑤，可由性质④推得。证毕。

由性质③可知，折半检索最大比较次数为$[\log_2 n]+1$， 那么它的平均检索长度如何？

先假定有序表长度为 $n=2^h-1$（即 $h=\log_2(n+1)$）， 则相应的折半检索判定树是深度为 h 的满二叉树（因每次折半均恰好等分）；对满二叉树，第 k 层上结点数目为 2^{k-1}。检索第 k 层上任一结点均需进行 k 次比较，再假定检索每个元素的概率相等（$p_i=1/n$），从而，折半检索成功时的平均检索长度为

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

$$= \sum_{j=1}^{h} \frac{1}{n} j \cdot 2^{j-1}$$ (注: $j \cdot 2^{j-1}$ 为检索 j 层上结点比较次数之和)

$$= \frac{1}{n}\sum_{j=1}^{h} j \cdot 2^{j-1} = \frac{1}{2n}\sum_{j=1}^{h} j \cdot 2^{j}$$

$\sum_{j=1}^{h} j \cdot 2^j$ 是个有趣的级数，它的和可按下法推出：

$$\sum_{j=1}^{h} j \cdot 2^j = 1 \cdot 2^1 + 2 \cdot 2^2 + \cdots + h \cdot 2^h$$

$$= 2^1 + 2^2 + \ldots + 2^h +$$
$$2^2 + \cdots + 2^h +$$
$$\cdots\cdots$$
$$2^h$$

$$= \sum_{k=1}^{h} 2^k + \sum_{k=2}^{h} 2^k + \sum_{k=3}^{h} 2^k + \ldots + \sum_{k=h}^{h} 2^k$$

$$= \sum_{j=1}^{h} \sum_{k=j}^{h} 2^k$$

$$= \sum_{j=1}^{h} (2^{h+1} - 2^j) \quad \text{（等比级数求和公式）}$$

$$= \sum_{j=1}^{h} 2^{h+1} - \sum_{j=1}^{h} 2^j$$

$$= (h-1)2^{h+1} + 2$$

因此，

$$\text{ASL} = \frac{1}{2n}((h-1)2^{h+1} + 2)$$

$$= \frac{1}{n}((h-1)2^h + 1)$$

$$= \frac{1}{n}((\log_2(n+1)-1)(n+1)+1)$$

$$= \frac{n+1}{n}\log_2(n+1) - 1$$

当 n 较大时，(n+1)/n≈1，从而

$$\text{ASL} \approx \log_2(n+1) - 1 = O(\log_2 n)$$

折半检索算法尚有许多可改进的地方，如设法去掉变量 low 与 high 的使用， 免掉求 mid 的除法运算。按这种思想改进的算法有一致折半检索算法。

§9.2.3 斐波那契检索*

对有序表（顺序存储结构），还有其他的快速检索法，这里介绍其中两种：斐波那契法和插值法。

(一) 斐波那契数列

折半检索是用数据集的中点分割数据集的，而斐波那契(Fibonacci)检索是用斐波那契数列中的数分割数据集的。

一个 2 阶斐波那契数列定义如下：

$F_0=0$

$F_1=1$

$F_n=F_{n-1}+F_{n-2}$ $n>1$

例如，二阶斐波那契数列的前 14 项为：

n：	0	1	2	3	4	5	6	7	8	9	10	11	12	13
F_n：	0	1	1	2	3	5	8	13	21	34	55	89	144	233

斐波那契数列具有一些奇妙的性质，有相当一部分事物，如果服从斐波那契数列规律，就会达到最优状态或富有艺术魅力。如著名的黄金分割比例 0.618 …（即 $(\sqrt{5}-1)/2$）就与斐波那契数密切相关。有兴趣的读者可参阅进一步的资料。

(二) 斐波那契检索法

设待查数据集中元素个数 n 比二阶斐波那契数列中某数少 1（如果不是这样，则增设若干虚元素，使数据集长度满足该条件），即 $n=F_k-1$，那么，斐波那契检索是每次找相对序号为 F_{k-1} 的数据元素进行测试。设数据集中元素序号为 1~ F_k-1，那么对该数据集的斐波那契检索方法为：

(a) 若 n=0，则检索失败，结束

(b) 令关键字 key 与数据集中序号为 F_{k-1} 的元素 $R(F_{k-1})$比较。

(c) 若 key<$R(F_{k-1})$，则对元素序号为 1 到$(F_{k-1}-1)$ 的子集施行斐波那契检索。注意，此子集长度为$(F_{k-1}-1)$，比 F_{k-1} 少 1。

(d) 若 key=$R(F_{k-1})$，则检索成功地结束。

(e) 若 key>$R(F_{k-1})$，则继续对元素序号为$(F_{k-1}+1)$到 F_k-1 的子集进行斐波那契检索。注意，此子集长度为$(F_k-1)-(F_{k-1}+1)+1=F_k-F_{k-1}-1=F_{k-2}-1$。

对情况(c)，下次的比较点显然为序号为 F_{k-2} 的元素，但情况(e)如何？因子集长为$(F_{k-2}-1)$，故下个比较点的元素序号的偏移量为 F_{k-3}，它相对于起点 F_{k-1}，所以下个比较点应为 $F_{k-3}+F_{k-1}$。

(三) 斐波那契检索判定树

为了分析斐波那契检索算法，并构造斐波那契检索算法的实现程序，我们讨论斐波那契检索算法的判定树。下面根据上面给出的斐波那契检索算法讨论斐波那契树的构造。

先定义 m 阶斐波那契树 T_m，它有$(F_{m+1}-1)$个内部结点和 F_{m+1} 个外部结点（用双圆圈表示外部结点），它的构造方法为：

(a) 若 m=0 或 1，有 $T_0=T_1=$ ⓪

(b) 若 m≥2，则 T_m 之根为 F_m，左子树是 T_{m-1}，右子树为 $T_{m-2}+F_m$（符号 $T_{m-2}+F_m$ 表示将树 T_{m-2} 的结点序号分别加上数 F_m 后得到树）。图 9-0给出了 2、3、4 阶斐波那契树构造的例子。图 9-0是 6 阶斐波那契树的例子。

显然，这里定义的 m 阶斐波那契树完全描述了前面给出的斐波那契检索算法。

从该构造法推知，m 阶 Fibonacii 树的高为 m（即带外结点的高），（可由数学归纳法证得）。因此它中任一内结点的左右子树深度相差 1，为平衡二叉树。

若数据集长度为 $n=F_k-1$，则它的斐波那契检索判定树是一棵(k-1) 阶的斐波那契树，其中外结点表示检索失败时的终止点。图 9-0是关于长度为 $n=F_k-1=12$ 的数据集的斐波那契检索判定树。

在斐波那契树中，对任一有左右两个儿子的结点，它的编号与它的两个儿子编号之差的绝对值为同一个量，且这个量也是个斐波那契数。若它与它的二个儿子的差为 F_i，则它的左儿子与它的左孙子的差是 F_{i-1}，而它的右儿子与它的右孙子（它的右儿子的两个儿子）间的差是 F_{i-2}。这里，仅对儿子、孙子为内结点的情况。这些特殊情况可用在斐波那契检索算法的实现中。

(四) 斐波那契检索算法的实现

斐波那契检索算法的实现的关键是如何由当前记录（即当前参加比较的记录）号 i，

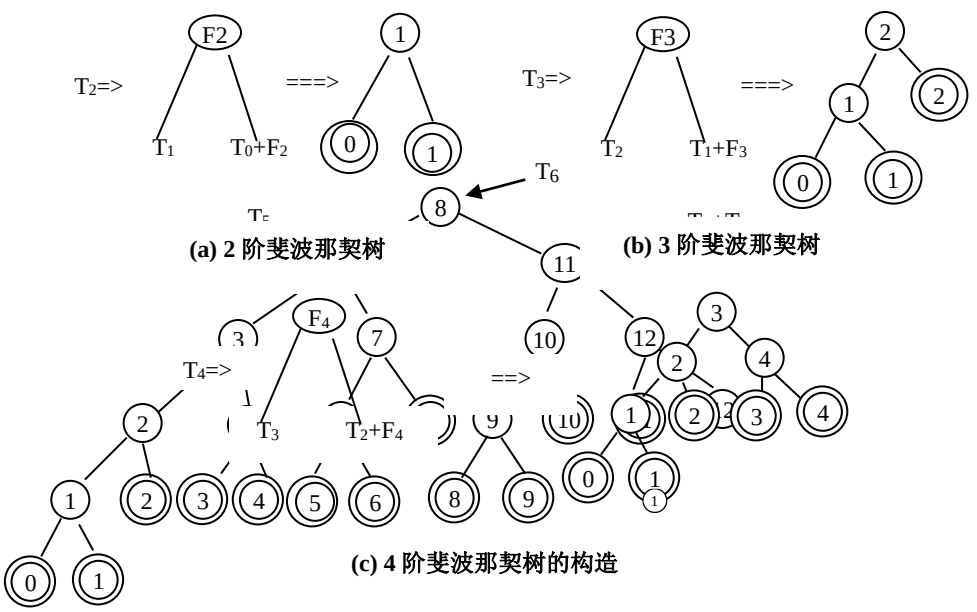


图 9-2 斐波那契树的构造

图 9-3

求得下次比较的记录号，即如何求得判定树中结点 i 的左右儿子。

由斐波那契检索树的构造知，斐波那契检索树中每个内结点，它的两个儿子的编号与它自己的编号的差相等，且为一个斐波那契检索数。若结点 i（其编号为 i）与它的儿子的编号差为 f_j，那么，i 的左儿子与自己的儿子的差为 f_{j-1}，而 i 的右儿子与自己的儿子的差为 f_{j-2}。据此，在算法中设置两个量 q 和 p，令它们一直是相邻的一对 Fabonacci 数，即若 i 为 T_{k-1} 的根，则 q 和 p 依次表示 F_{k-3} 和 F_{k-2}。利用 p 和 q 就可方便地求出 i，具体求法如见图 9-0。具体实现请见下面的程序。

```
long   FindFibonacii(long f2,long &f0,long &f1)
{//查找 Fibonacii 数 f2 的两项 f0 和 f1
 //f2—输入量，此函数目的是要求 f2 的前两项
 //f1，f0—输入量（地址）,返回时为 f2 的前两项
 //返回值—若 f2 恰为其二阶 Fibonacii 数，则返回 0；否则返回 f2 与不大于 f2 的最大 Fibonacii 数的差

 long t,f;
 f0 = 0;
 f1 =1;
 f=f0+f1;
 while(f<f2)
 {
  t=f+f1;
  f0=f1;
  f1=f;
  f=t;
 }

 if (f==f2) return 0;
 else return f2-f1;
}//FindFibonacii
```

```
long   FibonaciiSearch(int a[],long n, int key)
{//有序表（升序）的 Fibonacii 查找算法
 // a[]—输入量，一维数组，表示有序表，表中元素不减排列，0 号位置空闲，
 //n 为表中元素个数，其比 Fibonacii 数小 1
 //key—输入量，待查关键字
 //返回值—成功返回所查到的关键字的位置号，否则返回 0

 long i,p,q,t;
```

```
 char bDo;

 bDo=1;
 FindFibonacii(n+1,p,i);//p--i--n 为顺序三个 Fibonacii 数
 q=i-p; //q--p 为顺序 2 个 Fibonacii 数，下面的循环中，一直保持此关系
 while(bDo)
 {
  if(key==a[i])    return i; //成功
  if(key<a[i])
  {
   if (q==0) bDo=0; //已到叶子，结束检索
   else
   {
    i=i-q;
    t = p;
    p=q;
    q=t-q;
   }
  }
  else
  {
   if (p==1) bDo=0; //已到叶子，结束检索
   else
   {
    i=i+q;
    p=p-q;
    q=q-p;
   }
  }//else
 }//while
 return 0; //未发现
}// FibonaciiSearch
```

就平均而言，Fibonacci 检索算法要比折半检索算法快些。可以证明， 在检索成功下，Fibonacci 检索算法速度约为一致折半检索的 1.2 倍，普通折半检索算法的 2.4 倍。但在最坏情况下，比折半检索稍差。

§9.2.4 插值检索*

注意一下日常中我们查英文词典的方式。如果我们要查一个以 w 打头的词， 则先从词典中较后部分检索，而不是从中间或按 Fibonacii 数确定位置。这种方式实质上是根据被查关键字的大小确定比较点。这种思想可导致一种新的检索法——插值检索法。

具体地讲，插值检索法是按下式确定比较点 i：

$$i=(K-Ke)/(K_n-Ke)\cdot n$$

这里，K 为待查关键字，Ke 与 Kn 分别为当前检索子集中最小与最大关键字，n 为记录总数。

插值检索的实现方法与折半检索很类似。

§9.3 线性索引结构

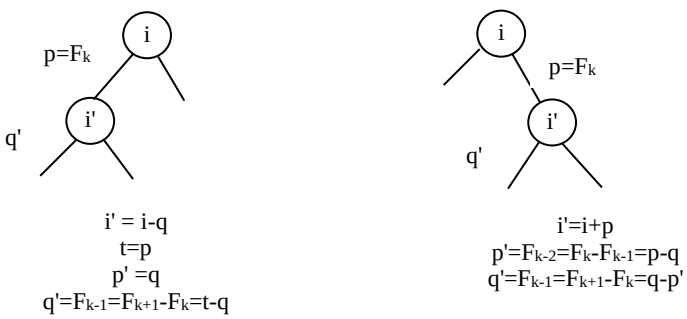


图 9-4 Fabonacci 检索点的计算（即当前检索点为 i，计算下次检索点 i'，并调整 p 与 q 的值为 p'与 q'

§9.3.1 概述

索引是为了加快检索速度而引进的一种数据结构。书的目录就是索引技术的一个应用实例。

索引技术中的关键对象是索引。一个**索引**(Index)针对（隶属于）某一数据记录集，它由若干索引项构成，**索引项**的结构为

关键字	关键字所在记录的位置的指示器

索引项集合可以组织成多种结构，一般有线性结构与树形结构。
若索引中的索引项组织为线性表，则称其为**线性索引（索引表）**。若线性索引中的索引项按关键字有序，则称这种索引为**顺序索引**（也称索引表）。索引所针对的数据集可以为任意结构，其驻留介质可以是内存，也可是外存。线性索引一般为静态结构。
若索引中的索引项组织为树形结构，则称为**树形索引**（目录树）。树形索引常为动态结构。本节介绍线性结构的索引，树形索引将在下节起介绍。
对一些大数据集，其索引表本身可能也很大，可对索引表建立另一个索引，这样就构成了多级索引。某级索引很大时，也可能要存贮在外存。
顺序索引技术可根据索引表的索引项与数据集中元素之间的对应关系，分为**稠密索引**与**非稠密索引**（非稠密索引常为**分块索引——**索引顺序结构）。
例如，在一本书中，正文对应数据集，而目录对应索引（为非稠密索引）。
数据集建有索引后，检索工作即可在索引上进行，所以，若索引组织得好，则可显著提高检索速度。
索引技术除涉及检索操作外，还涉及索引的创建与维护问题。

§9.3.2 稠密索引

如果数据元素集中的每个元素对应一个索引项，则这种索引称为稠密索引。这里我们仅讲线性稠密索引，即索引为一个按关键字有序的线性表（连续结构）。图 9-0给出了一个稠密索引的例子。
对这种索引，若索引已建立，则检索问题可通过对索引表直接调用上节所述的有序表检索算法而解决，所以，这里的主要问题是如何创建索引。
为简单起见，我们假定数据集为连续存贮结构的线性表。由于索引是按关键字有序的，所以，创建索引可以看作是在有序表中插入元素（保持有序）的过程。具体的算法可参照相关的排序算法。

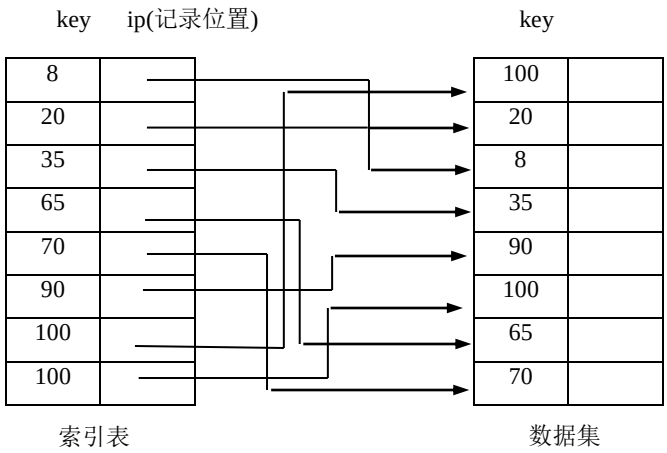


图 9-5 一个线性稠密索引的例子

§9.3.3 分块索引（索引顺序检索）

(一) 分块索引结构

显然，稠密索引的索引项的数目与对应的数据记录个数相同，所以空间代价很大。为了减少索引项数目，可以对数据级按序分块，使其分块有序。

这里的块是指数据元素集的子部分（连续出现的一部分元素）。**分块有序**是指：若数据集可以划分为若干块，则对任意两块 B1 和 B2，若 B1 在逻辑上位于 B2 之前，则 B1 中任一元素的关键字的值也在逻辑上位于 B2 中任一元素之前。分块有序并不要求块内一定有序。显然，有序集一定是分块有序的。

对于分块有序的数据元素集，就没有必要采用稠密索引，只需令每块对应一个索引项，这样就可大大减少索引项的数目。

分块结构的索引的构成方法是，每块对应一个索引项，每个索引项的结构为：

最大或最小关键字值	块长	块首地址或尾地址

各索引项按关键字值字段的升序或降序排列在一起，形成一个索引表。

在实际应用中，每块可以占用一个独立的存贮区（可以不按逻辑次序出现）。存贮区可以是内存，也可以是外存。图 9-0就是一个分块索引的例子。

(二) 索引顺序检索

这里讨论对分块索引结构的检索——索引顺序检索。

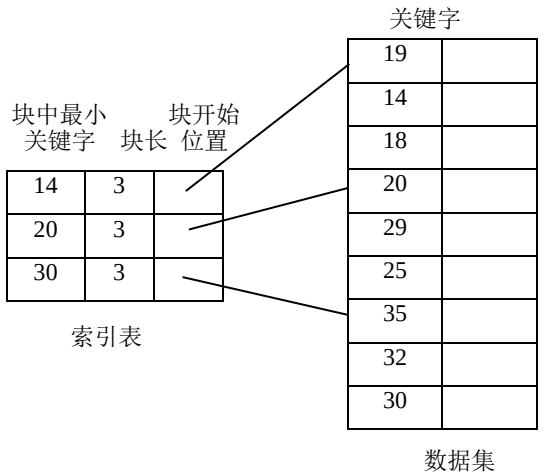


图 9-6 一个分块索引的例子

索引顺序检索分两个步骤：①在索引表中确定关键字所在块；②在块中检索关键字。
查索引表时，由于索引表是有序的，则可使用高效的有序表检索法（如折半检索，Fibonacii 检索等）。但这里由于是确定关键字所在块， 所以在检索不成功时，应指出关键字值位于索引表中哪两项之间（即指出位于数据集中那一块中），因此，前述的有序表检索算法要做适当扩充，使得当检索不成功时，应指出关键字值位于索引表中哪两项之间。这种扩充的实现留作练习。
在块内检索时，由于块内可能无序，所以不能使用高效的有序表检索法，只能使用顺序检索法。
分块检索的平均检索长度 ASL 应为

$$ASL=ASL_i+ASL_b$$

这里，ASL_i 为对索引表的平均检索长度，ASL_b 为块内平均检索长度。
现考查一下索引表与数据表均采用顺序检索法时的平均检索长度。设数据表长为 n，

块长为 s，块数则为 b=[n/s]注：是「」吗？+1，又设块与记录被检索的概率分别为 1/b 与 1/n（即等概率），则有

$$ASL=ASL_i+ASL_b$$

$$= \sum_{j=1}^{b}\frac{1}{b}j+\sum_{j=1}^{s}\frac{1}{s}j=\frac{b+1}{2}+\frac{s+1}{2}$$

$$= \frac{1}{2}([\frac{n}{s}]+1+s)$$ +1 （无错）

因此，平均检索长度不仅与表长 n 有关，而且与块长 s 有关，可以证明，当 s 取 $\sqrt{n}$ 时，上式（即 ASL）取极小值 $\sqrt{n}+1$，这个值已比顺序检索(n+1)/2 有了数量级的改进，但还不及折半检索（$\frac{n+1}{n}\log_2(n+1)-1$）。

§9.4 树形检索结构与二叉排序

二叉排序树是一种基本的树形检索结构（树目录），许多其他树形目录都是从它发展而来。

§9.4.1 树形检索结构概述

树形检索结构也称树目录，是一种树形结构的索引。树中的每一结点是一个索引项，它一般应包含它所对应的记录的关键字（对非稠密索引是关键字代表）和记录（元素）地址。由于树结构的高度一般小于同规模的线性结构的长度，所以对树结构的检索一般也快于线性结构。

与线性索引类似，树目录也常常用于驻留在外存上的数据集。这种用途的检索结构为外查找结构。不过，我们在这里并不打算涉及外存数据的操作，而为了突出主题，只针对驻留在内存的数据集。尽管如此，这里的讨论经适当扩充就可适合于外存数据集，有时甚至与数据集的驻留介质无关。

树目录多用作动态结构，即树目录中结点可动态地（即不是一次性地）增加或撤消，因此树目录常采用链式存贮结构。

树目录中父子关系用来指示结点间次序关系，一般别无它意。

二叉排序树是一种基本的树目录，由它可导出最优（次优）二叉排序树与平衡二叉树。然后导出各种 B 树。B 树可用来实现高效的外部文件索引，广泛地应用于数据库实现中。这几种树目录均将关键字做为一个整体处理。字符树则与此不同，它处理的基本成份不

是关键字整体，而是组成关键字的各个字符。但从结点值的次序方面讲，它仍然属于二叉排序树。

对树目录，由于检索就在树目录上进行，所以树目录的检索判定树就是树目录本身，因此关于判定树的许多结论与概念均适合树目录（如内外路径的概念含意等），但判定树只是个概念上的东西，而树目录是为支持检索而引入的一种结构。

§9.4.2 二叉排序树的概念

二叉排序树是一种特殊的二叉树，它的结点按结点中关键字次序分布，可认为是一种结点内容受限的二叉树。

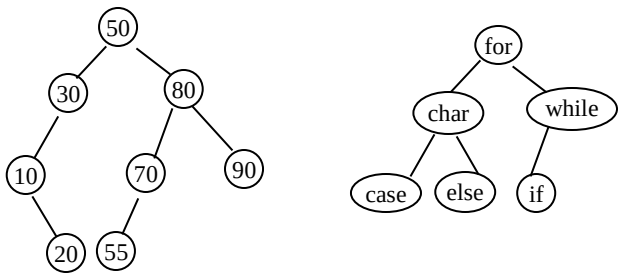
二叉排序树（也称二叉检索树）是一种结点含关键字的二叉树，它或为空二叉树，或满足下列性质：

若它的左子树不空， 则其左子树上所有结点的关键字均分别小于它的根结点关键字；

若它的右子树不空，则右子树上所有结点的关键字均分别大于它的根结点的关键字。

它的左右子树亦分别为二叉排序树。

例如，图 9-0所示的两棵树均为二叉排序树。



(a) **(b)**

图 9-7 二叉排序树

前面讨论的折半检索与 Fibonacii 检索判定树也都是二叉排序树。

显然，对二叉排序树进行中序遍历就会得到一个结点（关键字）的升序序列。因此，若已得到二叉排序树，就相当有了排序序列。

若给二叉排序树添加外结点，则外结点就可有具体的含意：某外结点代表这么一个值的集合，该集合中的值的大小由开区间(a,b)确定，这里 a 与 b 分别为该外结点在该二叉排序树（带外结点）的中序遍历序列中的前趋结点与后继结点的关键字值。例如，图 9-0 (a)中结点 20 的右外结点的前趋与后继分别为结点 20 与 30，所以它代表值集(20,30)。

由外结点的这个特性知，外结点可以代表分块有序数据集的数据块，而二叉排序树是数据集的索引。在实际中，数据块可以驻留在任意地方，如外存。

由于二叉排序树一般是动态的，故我们采用链式结构，它的结点的 C++ 描述同前面章节介绍的二叉树，即：

```
template <class TElem>
struct TBinSortTreeNode
{
  TElem info;
  TBinSortTreeNode *lc,*rc;
};
```

为简便，我们下面将 info 作为关键字。

§9.4.3 二叉排序树的检索

由二叉排序树的定义推知，在以 root 为根的二叉排序树中检索关键字 key 时，先检查 key 是否等于 root 的关键字，若是，则检索成功，否则，若 key 小于 root 的关键字，则按类似方式在 root 的左子树中检索 key， 否则按类似方法在 root 的右子树中检索。若某次检索时，被查树的根为空，则表示检索失败，返回。

二叉树检索的递归程序为：

```
template <class TElem>
TBinTreeNode *SearchBinSortTree(TBinTree *root, TElem key)
{//root 为被检索的树的根指针
if (root==NULL) return NULL;
    if (root->info == key) return root;
    if (key <root->info) return SearchBinSortTree(root->lc, key);
return SearchBinSortTree(root->rc, key);
}
```

但是，更有用的是找到指定结点（指定关键字对应的结点）的父亲。下面是具体的程序。这是个非递归程序。递归程序的编写留作练习。

```
template <class TElem>
TBinSortTreeNode<TElem> *SearchFatherBinSortTree(
TBinSortTreeNode<TElem> *root, TElem key, int &sonNo)
{//二叉排序树检索。root 为被检索的树的根指针，检索 key，
  //若找到，则返回其父结点的指针(当找到的是 root 时，返回 NULL,并置 sonNo 为 0)
  //当 key 在左儿子中时，置 sonNo 为 1，否则置 sonNo 为 2。
  //若找不到 key,返回 key 的插入位置的父亲指针,
  //且置 sonNo=-1(当 key 为插入点的左儿),或置 sonNo=-2(当 key 为插入点的右儿)
  TBinSortTreeNode<TElem> *p, *p0;
```

```
 sonNo=0;
 p0=NULL;
 p = root;
 if (key ==p->info) return NULL;
 while (p!=NULL)
 {
  if (key ==p->info)//假定 Telem 型支持恒等运算“==" 
     return p0;//已找到，返回其父亲
  p0 = p;
  if (key <p->info) {p = p->lc; sonNo=1;} //准备在 root 的左子树中找
  else {p = p->rc; sonNo=2; } //准备在 root 的右子树中找
 }

 if (key<p0->info) sonNo=-1;
 else sonNo=-2;
 return p0; //未找到
}
```

§9.4.4 二叉排序树的插入*

二叉排序树的插入是按结点关键字插入，即应保证插入后二叉树仍为二叉排序树。所以，插入位置要根据待插入的关键字的值决定。例如，若关键字序列为(70,20,30, 10, 80,50,90,55)，则依次从此序列中取关键字，将其插入当前树中的过程如图 9-0所示。注意，初始时，是在空树中插入，相当于创建新树。

至于插入位置的确定，可利用上面给出的检索程序 SearchFatherBinSortTree(). 下面是具体的程序。

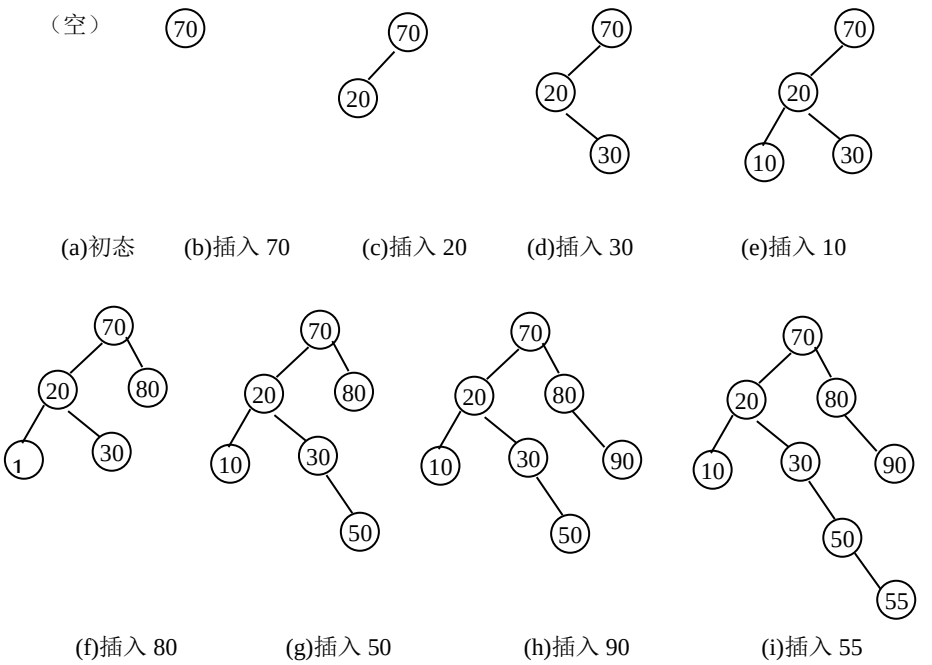
```
template <class TElem>
TBinSortTreeNode<TElem> *InsertBinSortTree(TBinSortTreeNode<TElem> * &root,
                                           TBinSortTreeNode<TElem> *pNode)
{ //将 pNode 所指结点插入到以 root 为根的二叉排序树中，返回插入后 pNode 的父亲.
   //如果 root 为空，则 pNode 被作为树根，root 直接指向 pNode
   //假定 pNode 的值不与已树中现有结点的值重复
   //若待插入结点与其他结点关键字重复,则不插入,返回 NULL

   TBinSortTreeNode<TElem> *p;
   int sonNo;
```

```
  pNode->lc=NULL; pNode->rc=NULL;
  if (root==NULL) {root=pNode; return NULL;}
  p = SearchFatherBinSortTree(root, pNode->info,sonNo);
  if (sonNo>=0) return NULL; //若待插入结点与其他结点关键字重复,则不插入,返回 NULL
  if (sonNo==-2) p->rc=pNode;
  else p->lc=pNode;

  return p;
}
```

利用上面的插入程序，可根据原始数据生成二叉排序树。下面是一个示例程序：



(a)初态 (b)插入 70 (c)插入 20 (d)插入 30 (e)插入 10

(f)插入 80 (g)插入 50 (h)插入 90 (i)插入 55

图 9-8 建立二叉排序过程示例

```
TBinSortTreeNode<int> *CreateBinSortTree(int a[], long n)
{//根据一维数组 a[]中的原始数据（个数为 n），创建二叉排序树，返回其树根。
 //假定 a[]中无重复元素

 long k;
 TBinSortTreeNode<int> *root, *p;
```

```
 root=NULL;
 for (k=0; k<n; k++)
 {
   p = new TBinSortTreeNode<int>;
   p->info=a[k];
   InsertBinSortTree(root, p);
 }
 return root;
}
```

显然，在插入过程中，插入关键字的次序不同，所建二叉树的形式也不同。图 9-0是二棵形式不同的二叉排序树，但它们对应同一个关键字集合。左图对应的插入次序为（3, 2, 5, 1, 4），而右图对应的插入次序是（5, 4, 3, 2, 1）

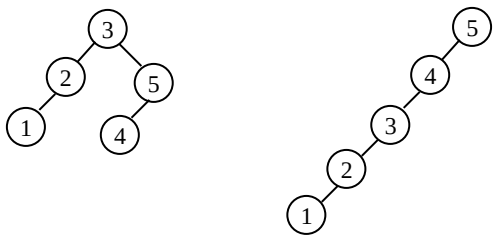


图 9-9 插入次序不同，树结构也不同

§9.4.5 二叉排序树的删除*

从二叉排序树中删除任一给定结点时，只删除该结点本身，并不删除它的子树，而且要保持二叉排序树的性质（排序性质）。

设待删结点的指针为 p，它的父结点的指针为 f，则从二叉排序树中删除 p，可分下述几种情况进行。

(1) 若 p 无左子树，则用 p 的右子树（若有的话）的根代替 p（见图 9-0 a）。

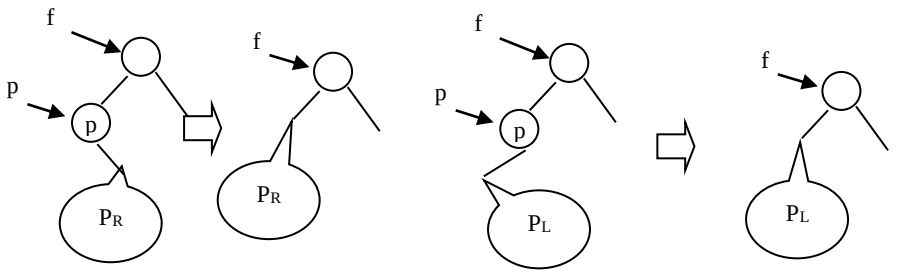
该方法显然是正确的，因为，此时 p 的右子树自然是二叉排序树，并且，用该右子树代替 p 后，与 p 的父亲 f 的大小次序不改变。该方法可用下列程序段描述：

```
if (p==f ->lc)    f ->lc=p ->rc;    //p 是 f 的左儿子
else f ->rc=p ->rc;
return    p;
```

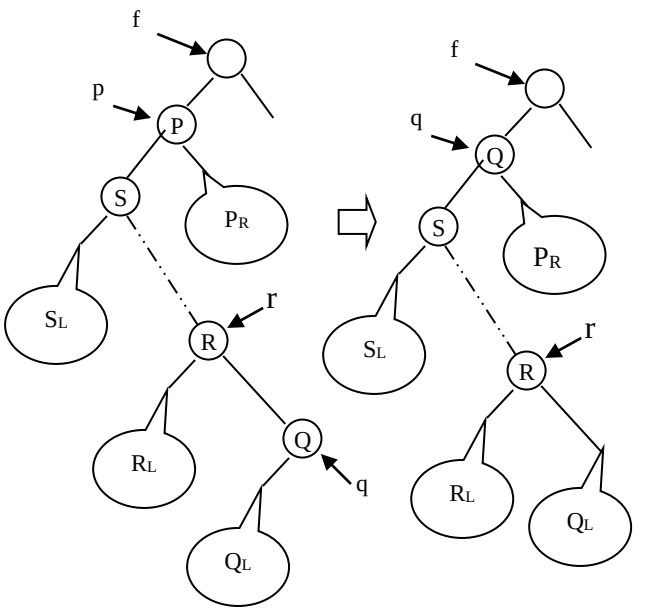
(2) 若 p 有左子树，但 p 无右子树，则用 p 的左子树代替 p（见图 9-0b）。实现方法为：

```
if (f ->lc==p) f ->lc=p->lc;
else f ->rc=p ->lc;
  return p;
```

(3) 若 p 有左子树，也有右子树，则用 p 的中序前驱（中序序列中 p 的直接前趋）q 代替



(a)p 无左子树 **(b)p 有左子树，但无右子树**



(c)p 有左、右子树

图 9-10 二叉排序树中删除 p

p，并将 q 的左子树转让给 q 的父亲（注：q 必无右子树），（见图 9-0c）。

事实上，因 q 为 p 的中序前趋，所以它的大小情况完全符合替代 p 的要求（p 也可用 p 的直接后继替代 p），且 q 无右子树（否则由中序遍历方法知，q 不是 p 的直接前趋），所以拿掉 q 后，只需处理 q 的左子树。因 q 与 q 的左子树同为 q 的父亲的左子树（或右

子树）结点，故用 q 的左子树（根）替代 q 的位置后，大小次序保持。

实现时，重点是要找到 p 的中序前趋 q。由中序遍历性质知，只需从 p 的左儿子起，顺着右链往下摸，直到右链为空，此时该空右链的父亲即为 q。 具体实现过程可描述为：

```
//下面查找 p 的中序前驱 q
   q=p->lc;
   r=p;   //使 r 一直为 q 的父亲
   while (q->rc!=NULL)
   {
    r = q ;
    q=q->rc;
   }

  //r==p 时表示 p 的中序前驱就是 p 的左儿子，这需要特殊处理
   if (r!=p)    r->rc=q->lc; //用 q 的左子树代替 q 的位置
   //下面用 q 置换 p
   if (r!=p) q->lc=p->lc;
   q->rc=p->rc;

   if (f->lc==p) f->lc=q;
else f->rc=q;retrun p;
```

显然，情况（3）包含了情况（2），故可取消情况（2），但用情况（3）处理情况（2）会降低速度。

上面讨论的是被删结点 p 的父亲 f 存在的情况，如果被删结点是根（无父亲），则上面的程序中，相关于 f 的操作都不需要进行。另外，在情况(3)中，当被删除的结点 p 的直接中序前驱 q 是 p 的儿子时，情况略有不同。

显然，该算法不论何种情况，删除后树的高度不会增加，而且最多减少 1， 该特性很有用，它使得算法方便地扩充为平衡二叉树的删除算法。

下面是完整的二叉排序树删除程序。

```
template <class TElem>
void DeleteNodeBinSortTree(TBinSortTreeNode<TElem> *p,
TBinSortTreeNode<TElem> *&f)
{//删除二叉排序树中由 p 所指向结点。
//进入时 f 指向待删结点 p 的父亲。若 f 为空（p 无父亲），则函数返回后，f 指向代替了 p 的结点
TBinSortTreeNode<TElem> *q,*r;

  if (p->lc==NULL) //p 无左子树
  {
```

```
  if (f==NULL) f=p->rc;
  else
     if (p==f->lc)    f->lc=p->rc;    //p 是 f 的左儿子
     else f->rc=p->rc;
 }
 else
 if (p->rc==NULL)
 {
  if (f==NULL) f=p->lc;
  else
     if (f->lc==p) f->lc=p->lc;
     else f->rc=p->lc;
 }
 else    //p 的左右子树均不为空
 {
  //下面查找 p 的中序前驱 q
  q=p->lc;
  r=p;    //使 r 一直为 q 的父亲
  while (q->rc!=NULL)
  {
    r = q ;
    q=q->rc;
  }

//r==p 时表示 p 的中序前驱就是 p 的左儿子，这需要特殊处理
  if (r!=p)    r->rc=q->lc; //用 q 的左子树代替 q 的位置
  //下面用 q 置换 p
  if (r!=p) q->lc=p->lc;
  q->rc=p->rc;

  if (f==NULL) f=q; //p 无父亲时,删除 p 后,q 成为新根,令 f 指向新根
  else //令 p 的父亲 f 指向 p 的替换者 q
     if (f->lc==p) f->lc=q;
     else f->rc=q;
 } //else

}//DeleteNodeBinSortTree
```

```
template <class TElem>
TBinSortTreeNode<TElem> *DeleteKeyBinSortKey(
TBinSortTreeNode<TElem> *&root, TElem key)
{//从二叉排序树中摘除指定关键字所在的第一个结点
//root--指向二叉排序树的根
//key--待删除关键字值
//返回--被删除结点的指针

   TBinSortTreeNode<TElem>    *p,*f;
   int sonNo;
   p=NULL;

   if (key==root->info)
   {//删除根结点 root
    f=root; //由于是删除根结点 root，故删除后的树的根改变了，新根为 p
    DeleteNodeBinSortTree(root,p);
    root=p;
    return f;
   }
   //找 key 所在结点的父亲（返回值）,无父亲时返回 NULL，且 sonNo==0
   p=SearchFatherBinSortTree(root,key,sonNo);
   if(p==NULL && sonNo!=0) return NULL ; //找不到 key
   if(sonNo==1)
   {//被删结点是 p 的左儿子
    DeleteNodeBinSortTree(p->lc,p);
    return p->lc;
   }
   else
   {//被删结点是 p 的右儿子
    DeleteNodeBinSortTree(p->rc,p);
    return p->rc;
   }
}
```

二叉排序树的删除也可以采用其它一些方法，例如，对情况(3)（被删结点 p 有左右子树），可以在摘除 p 后，用 p 的左子树（或右子树）代替 p 的位置， 而将 p 的右子树（或左子树）挂接到原 p 的直接前趋的右链域上（或挂接到原 p 的直接后继的左链域上）（见图 9-0）。但该方法不如前面介绍的方法好，因为它有可能使树升高。

§9.4.6 二叉排序树的分析与最优二叉排序树*

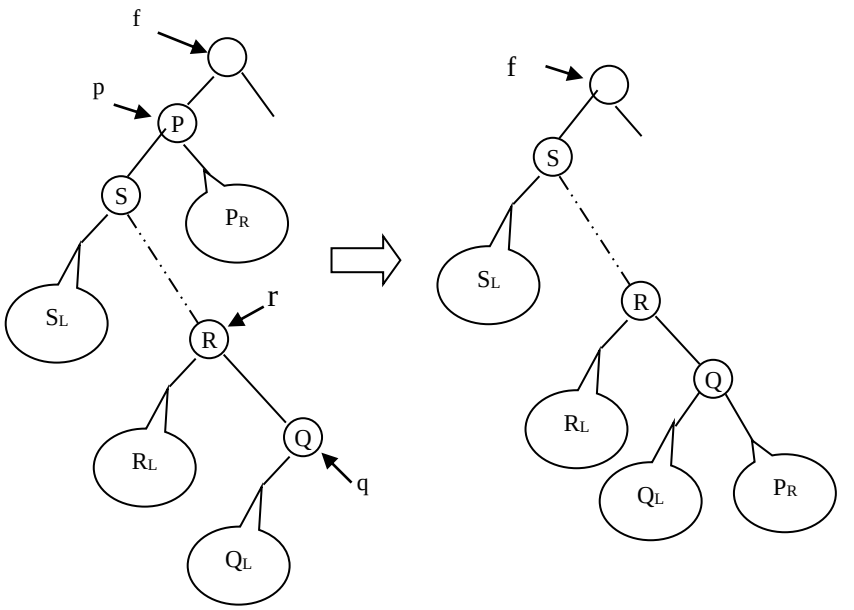


图 9-11 删除 p 的另一方法(用 p 左儿置换 p，p 的右子树挂接到 p 的直接前趋的右链)

二叉排序树的插入、删除均依赖于检索算法，所以只需分析二叉排序树的检索算法。

检索算法的效率与树的结构相关，而树结构又与关键字进入树中的次序有关。不同的进入次序，会导致不同的树。在极端情况下，二叉排序树可能变蜕化为一条链（当关键字按大小次序进入）。图 9-0就是一例。在这种情况下，检索算法的时间复杂度同线性表的顺序检索，即 O(n)。

这情况可能令人失望，但所幸的是，有人已证明，在等概率的情况下（即每个关键字被查概率相等），由 n 个关键字随机输入所构成的二叉排序树的检索查成功的平均检索长度 $P(n)\leqslant 1+4\log_2 n$。

在等概率下，显然树愈紧凑，平均检索长度愈小，最好应该是这样的树：树叶子只出现同一层上（顺序二叉树、满二叉树就属于这种树）。

若不是等概率，情况就复杂了，直观看，此种情况下，被查频度较高的结点应靠近树根，这与哈夫曼树（最优二叉树）类似，但哈夫曼树只针对叶子的权值，情况较简单。

给定 n 个关键字与它们的 n 个检索概率值，能否存在一棵平均检索长度（即加权内外路径长度）最小的二叉排序树呢？回答是肯定的。 这种树称为最优二叉排序树。

关于最优二叉排序树的构造，哈夫曼（Huffman D.）提供了一个极好的算法（迭代

构造），它可在阶为 $O(n\log_2 n)$的时间内构造出具有 n 个结点的最优二叉树。
然而，最优二叉树经动态地插入或删除后，就可能不是最优二叉树了，要保证一直最优，需扩充插入与删除算法，但这样时间代价很高，得不偿失，所以最优二叉树算法一般只用于静态结构或初始树构造。
对于需要经常进行插入与删除的情况，较好的选择是平衡二叉排序树，它是二叉排序树与最优二叉排序树之间的一种折衷。

§9.5 平衡二叉排序树*

高度平衡二叉树是一种结构受限的二叉树，它常与二叉排序树结合，称为平衡二叉排序树。该种树首先由艾德尔森•维乐斯基（Adelson-Velskii ）与兰迪斯（Landis）于 1962 年研究，故也称 AVL 树。

§9.5.1 基本概念

高度平衡二叉树（Height balanced binary Tree）：或是一棵空树， 或是具有下列性质的二叉树：它的左右子树均为平衡二叉树，且它们二者的深度之差不超过 1。这个定义与我们在讨论树结构的时候的第一实质上是相同的。
平衡因子：二叉树中的结点的平衡因子定义为该结点的左子树的深度与右子树深度之差。
平衡二叉排序树：若二叉排序树为平衡二叉树，则称为平衡二叉排序树。
在本节中，在不至于产生混淆的情况下， 我们常称高度平衡二叉树为平衡二叉树或平衡树。也常简称平衡二叉排序树为平衡二叉树。

§9.5.2 若干性质

平衡二叉树具有下列性质：
①平衡树中任一结点的平衡因子均为-1、0、1 三者之一，这是显然的。因为任一结点的左右子树深度之差不超过 1。
②任一棵二叉排序树，都可以转化为一棵平衡二叉排序树。
事实上，因折半检索判定树是平衡二叉树，所以，对任一棵二叉排序树，我们可以求出它的排序序列，然后根据排序序列按折半检索判定树的生成方法，组织成一棵二叉树，它显然是二叉排序树，且是平衡的。
该性质的意义在于指出了平衡二叉排序树的存在性，同时也指出了它的一种构造方法，但它不是一种好的构造方法，因为它是一种完全重构方法，不适合于动态构造，我们的目标是要找出一种动态平衡调整方法。
③一棵具有 n 个（内部）结点的平衡二叉树，其高度 h 满足

$$\log_2(n+1) \leqslant h \leqslant 1.4404\log_2(n+2)-0.328$$

该性质告诉我们，对具有 n 个结点的平衡二叉排序树的检索的最大比较次数为 $1.4404\log_2(n+2)-0.328$，即阶为 $O(\log_2 n)$，同时也可推知， $n= f_{n+2}-1$ 个结点的 Fibonacii 树是所有的具有 n 个结点的平衡二叉树中高度最大者，所以，若结点数为 $F_{27-1}=196417$，则对平衡二叉排序树的检索所需的比较次数决不会超过 25 次（即 h=27-2=25），这是相当可观的速度改善。

§9.5.3 局部平衡调整算法

对于平衡二叉树，经插入或删除结点后可能变得不平衡，所以插入或删除后，需调整原平衡树，使其保持平衡。这种操作称为**平衡调整**。这里我们先讨论对树中某一结点的平衡调整问题——局部平衡调整，这是树平衡调整的基础。

(一) 插入平衡调整

这里讨论某结点插入到某子树后，该子树的根结点的平衡调整问题，设 p 是一棵平衡二叉树（p 为根），则将某一结点插入它中后，有下列情况需要平衡调整：
(i) p 的平衡因子为-1（p 右重），若结点插入到 p 的右子树，且使 p 的右子树升高，则 p 变为右超重（p 平衡因子变为-2），此时需进行平衡调整。
(ii) p 的平衡因子为 1（p 左重），若结点插入到 p 的左子树，且使 p 左子树升高，则 p 变为左超重，需进行平衡调整。
其它情况均不需进行平衡调整。事实上，对情况(i)与(ii)的调整操作是对称的。
对平衡二叉排序树的调整，既要恢复平衡又要保持二叉排序树的性质。由二叉排序树定义知，对任意两个结点 A 与 B，若它们有父子关系，则可按图 9-0所示变换后，二叉排序树排序性质不变，这种变换称为旋转。

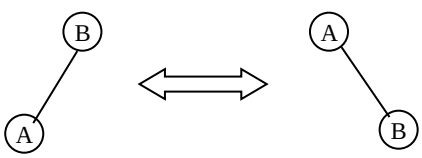
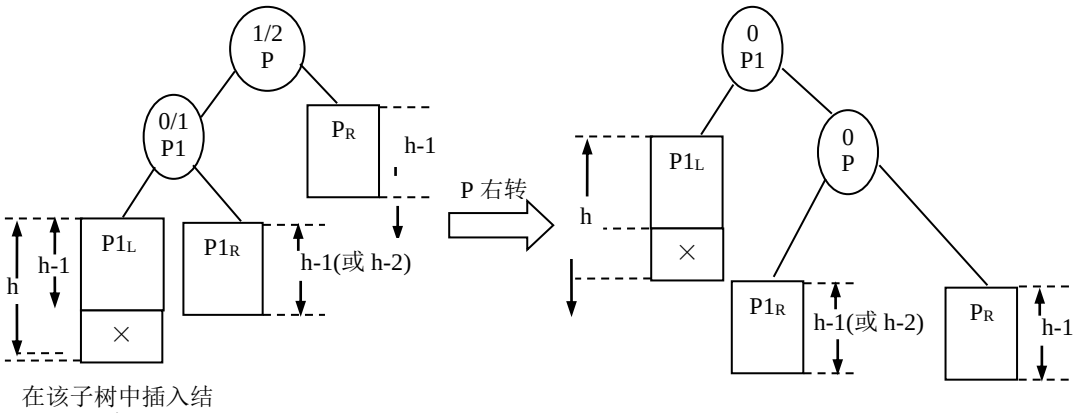


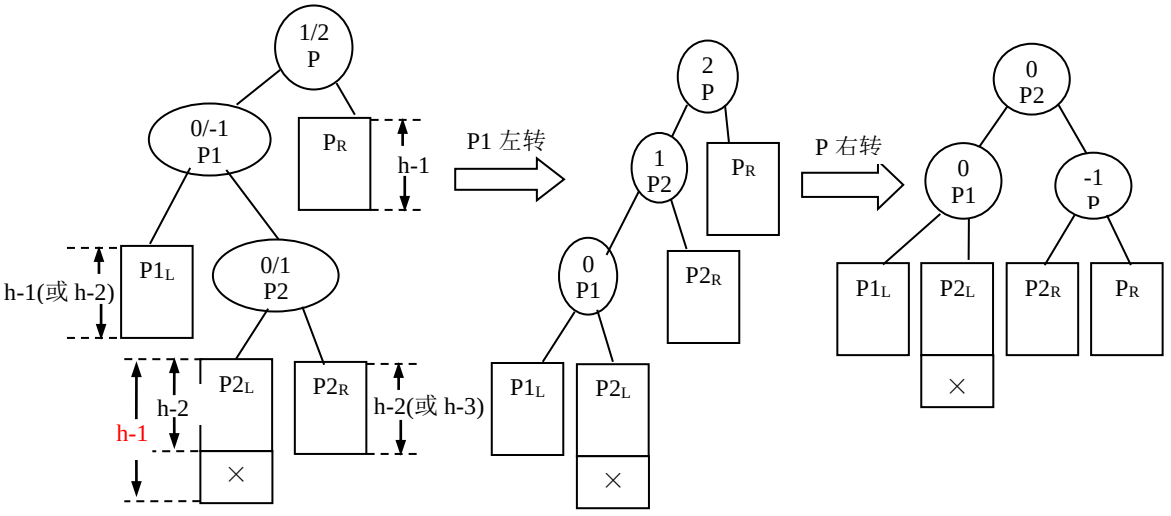
图 9-12 二叉排序树变换

左—〉右（B 右转）：以 B 为轴顺时针旋转，A 到 B 的位置，而 B 做为 A 的右子树
右—〉左（A 左转）：以 A 为轴递时针旋转，B 到 A 的位置，而 A 做为 B 的左子树

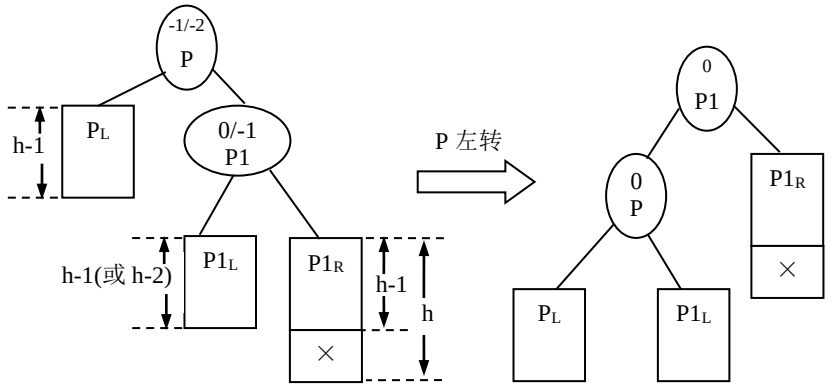
平衡调整就是通过使用这两种变换实现的。
对情况(i)，若插入到 P 的右子树的右子树，则称为 RR 型调整，否则（插入到 p 的右子树的左子树）称为 RL 型调整。对于情况(ii)，若结点插入到 p 的左子树的左子树，则称为 LL 型调整，否则（插入 p 的左子树的右子树）称为 LR 调整；这四种调整如图 9-0 所示。显然， 只有两种是独立的，即 RR 与 LL 对称；RL 与 LR 对称。



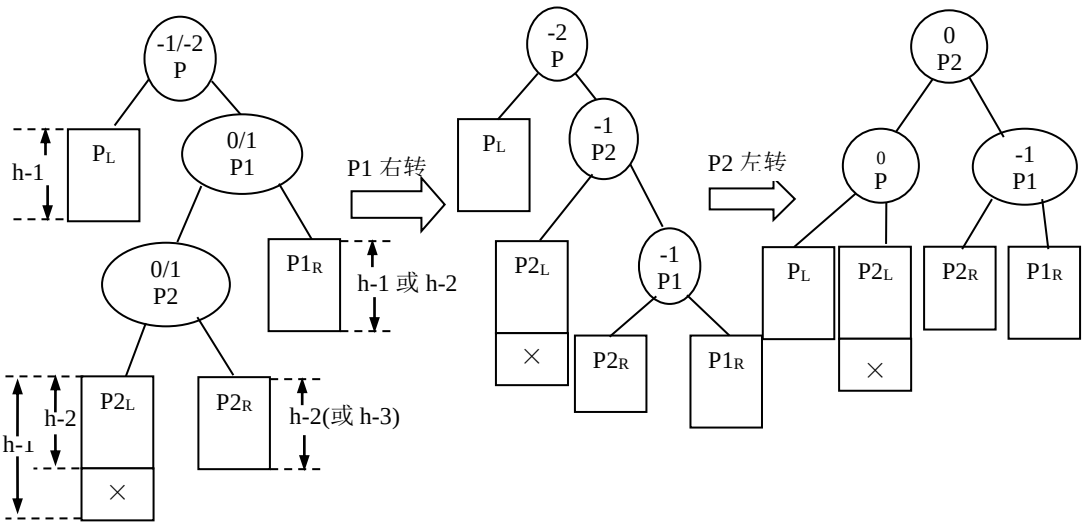
(a) LL 调整



答：原文正确。注：红字处是 **(b) LR 调整** ---?



(c) RR 调整



(d) RL 调整

图 9-13 在平衡二叉排序树中，插入接点后的平衡调整（结点内标注的 x/y 表示插入前与插入后的平衡因子分别为 x/y。图中，圆圈表示结点，方框表示子树

在图 9-0中，假定 p 为平衡二叉排序树，它的深度为 h+1， 插入一个结点后深度变为 h+2（若插入后深度未变，则无需平衡调整），成为最小非平衡子树（即它为不平衡树，但它的子树均为平衡树）。经过所示的四种平衡调整后，它变为一棵新的树（子树），高度恢复为 h+1，树根也已换成其它结点。 这四种调整具体说明如下：

LL 型：即 p 左重，结点插入到 p 的左子树的左子树 $p1_L$ 中，并使该子树升高（否则无需调整）。又由于已假定 p 深度为 h+1，则 p 右子树 p_R 深必为 h-1，p 左子树的左子树 $p1_L$ 深必为 h-1，结点插入它($p1_L$)后它的深变为 h。在后面将会看到，我们在调整 p 的平衡性的时候，总假定 p 起的插入路径（不含 p）上各点的平衡因子均为 0（这也是能办到的），对其它几种平衡类型情况也如此。对此种情况，以 p 为轴， 顺时针旋转 p1 与 p，然后将 p1 的右子树 $p1_R$ 转让给 p，这样可使该子树恢复平衡， 且保持排序性。实现这种调整的操作过程为：

```
p1=p→lc;   //p1 即为调整后的子树的根
p→lc=p1→rc;
p1→rc=p;
```

LR 型：即 p 左重，结点插入到 p 的左子树 p1 的右子树 p2 中，并使 p2（及 p1）升高。

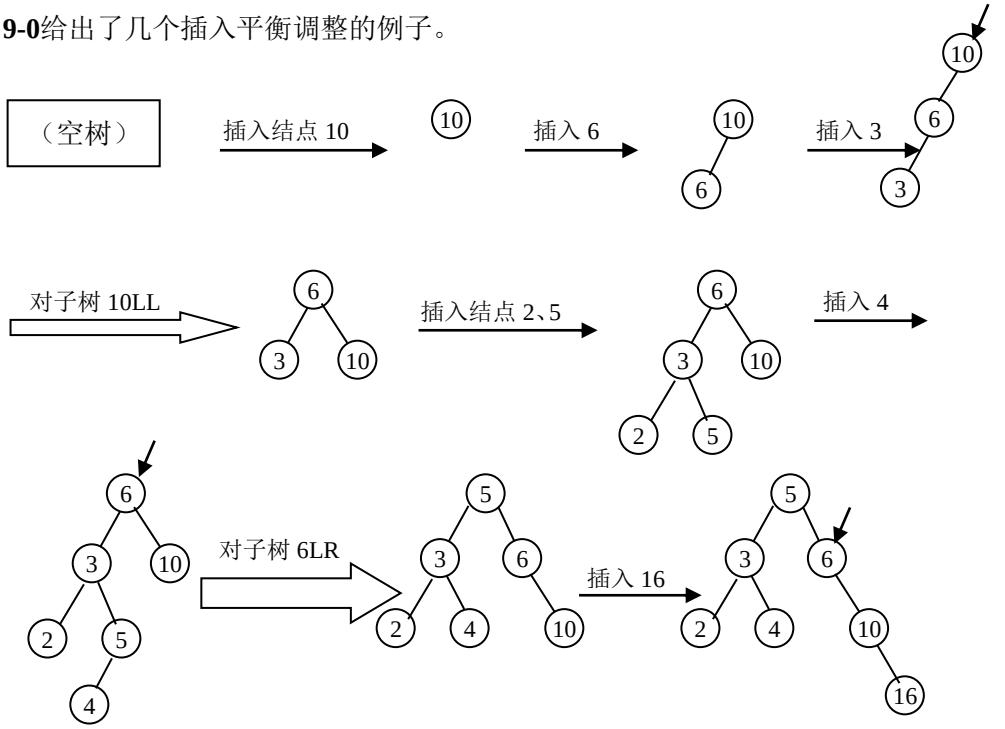
这里假定结点插入到 p2 的左子树（插入到 p2 右子树的处理方法与此类似）。在这种情况下，p2 左子树原为 h-2（否则不需平衡调整）， 插入后升高到 h-1，p 右子树高为 h-1，而 p2 右子树高为 h-2 或 h-3，p1 左子树高为 h-1 或 h-2， 这种情况的平衡调整，逻辑上需两次旋转：第 1 次以 p1 为轴，逆时针旋转 p1 与 p2；第 2 次以 p 为轴，顺时针旋转 p 与 p2。具体的实现过程如下：

```
p1=p→lc;
p2=p→lc→rc;
p1→rc=p2→lc;
p2→lc=p1;
p→lc=p2→rc;
p2→rc=p;
```

这里需注意一下 p1 的右子树原为空的情况，此时，原 p 子树实质上只含两个结点：p 与 p1（p1 为 p 之左子树），否则插入后 p 就无需调整了，或 p 就不是平衡树了。 这种情况下，p2 即为新插入结点，平衡调整也可按上面给出的过程进行。

RR 型与 RL 型：这两种类型的调整分别与 LL 型和 LR 型对称。分别将 LL 型与 LR 型的操作过程中的 rc 换为 lc，而 lc 换为 rc 即可成为 RR 型与 RL 型的操作，其它无需改动。

显然，这里给出的两种平衡调整过程，均会使被调整树恢复原高度（即插入前高度）。**图 9-0**给出了几个插入平衡调整的例子。



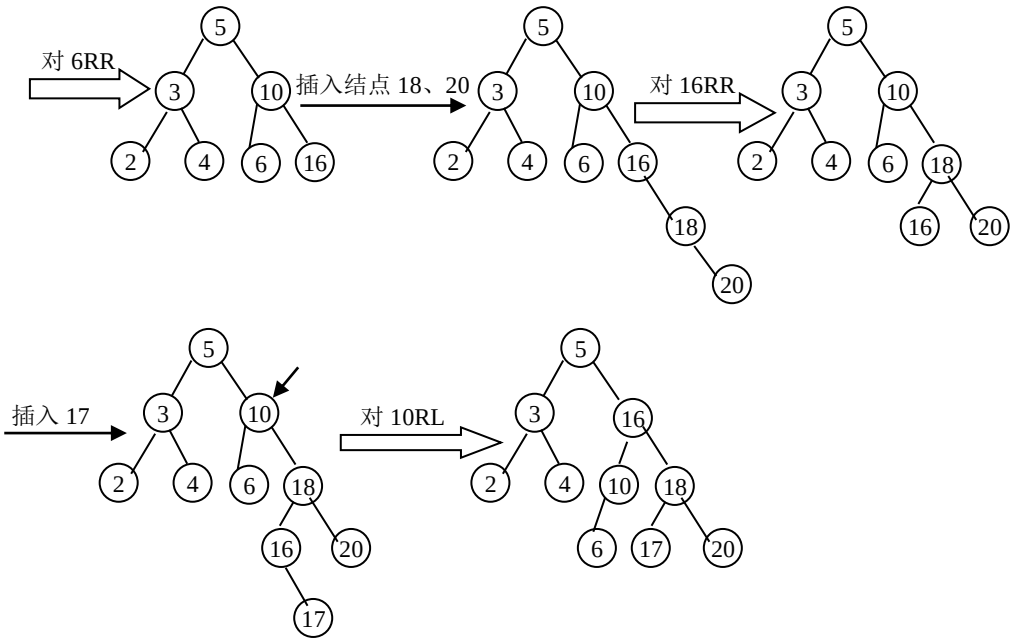


图 9-14 平衡二叉树的插入与平衡（箭头所指结点为最小不平衡子树的根）

(二) 删除平衡调整

这里讨论在某平衡二叉排序子树中删除一个结点后需进行的平衡性调整。在讨论二叉排序树中，我们给出一种二叉排序树删除算法，使得删除某结点 q 后，q 的剩余下属（各后代）构成的子树的高度恰好降低了 1， 所以我们这里只需讨论删除结点后高度降低 1 的情形。

某树 p 的左/右子树的高度降低 1，在平衡性上等价于 p 的右/左子树升高 1，所以， 删除结点的平衡调整算法应与插入结点类似。

设 p 是一棵平衡二叉排序树（子树），则当 p 的某子树高度被降低 1 后， 仅在下列情况下，p 才需进行平衡调整：

(i) p 左轻（即右重），且 p 左子树高度被降低

(ii) p 右轻（即左重），且 p 右子树高度被降低

显然，这里的情况(i)与(ii)分别对应于前面介绍的插入平衡调整中的情况(i)与(ii)。具体的操作方法也很类似。

图 9-0给出了几个删除平衡调整的例子。

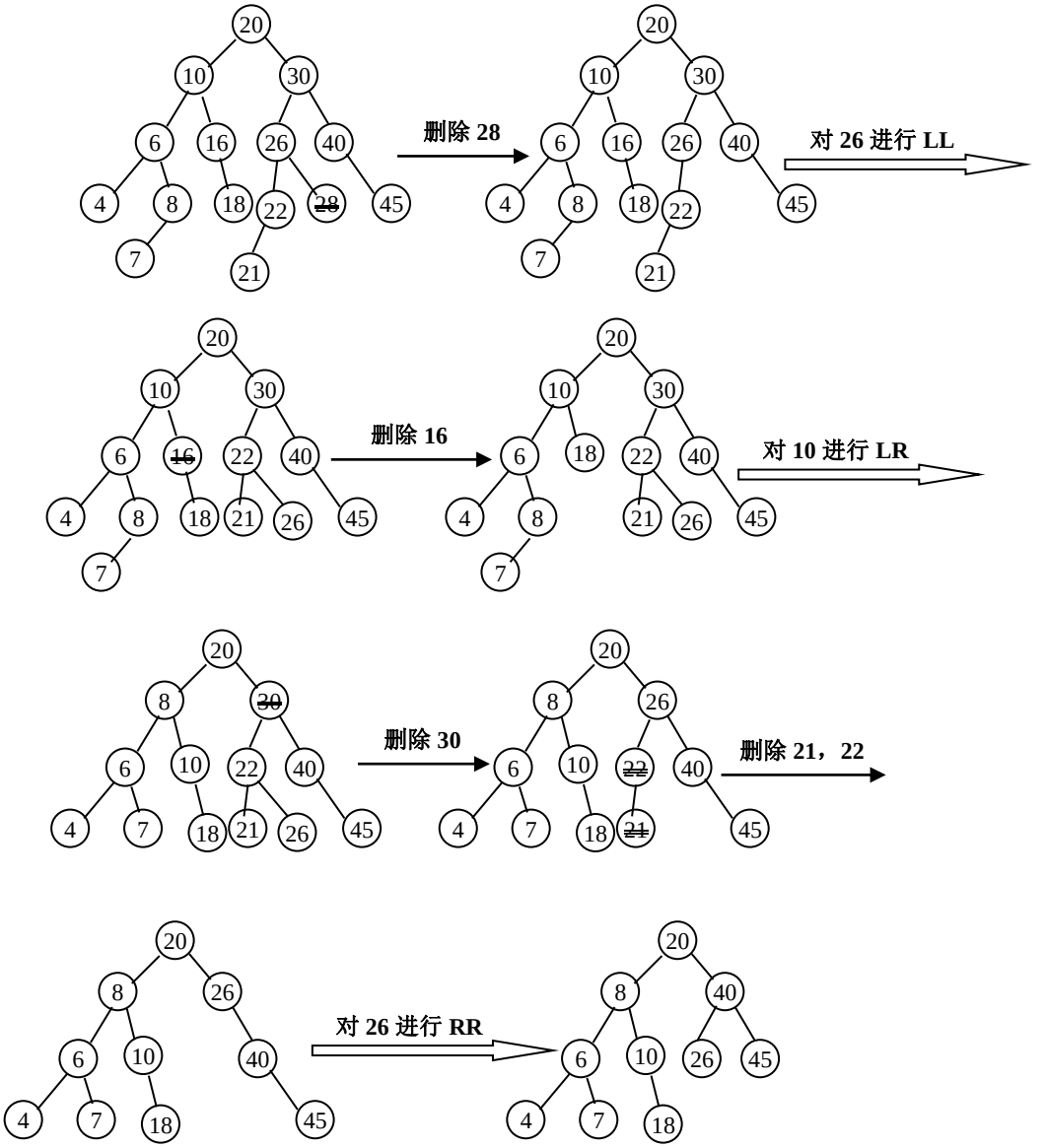


图 9-15 平衡二叉排序树的删除平衡调整

(三) 通用平衡调整操作

这里考虑将前述的插入平衡调整与删除平衡调整操作统一在一起。在进行平衡调整时，必需先知道是插入平衡还是删除平衡，是 L 型（含 LL 型与 LR 型，树的左子树升高，或右子树降低）还是 R 型（含 RR 型与 RL 型）。至于单旋（LL 型与 RR 型）还是双旋

（LR 型与 RL 型），可在内部由被平衡树的树根的平衡因子得知。因此，这种通用平调操作的一般形式应为

TreeBalance (p, left)

其中，p 指向待平衡的子树的根（过程结束后，p 指向平衡后的树的根），left 则指出是 L 型(LL,LR)还是 R 型(RL,RR 型)。

§9.5.4 平衡二叉排序树的插入

前面介绍的局部平衡算法，只是用于调整某棵子树的平衡性，但是，对插入操作，可能会使多棵子树失去平衡，这样就要多次调用局部平衡算法。但这不是一种好的做法。事实上，在这多棵失去平衡的子树中，必有这样一棵，它的根距原树根最远，并且它的各子树均未失去平衡，我们称其为**最小不平衡子树**。在插入中，最小不平衡子树的高度升高了 1，致使它的那些原已偏重的祖先结点不平衡， 只要将它恢复平衡，并使高度也恢复原高度，则由它引起的所有不平衡就消除了（即它的各祖先就恢复平衡了），从而也恢复了整棵树的平衡性。我们在前面给出的局部调整算法恰好能实现这点，所以，只要找到最小不平衡子树，则对它使用局部平衡调整算法，即可实现对整棵树的平衡恢复。因此，剩下的问题就是如何找到最小不平衡子树了。显然，若某结点的平衡因子为 0，则在它中插入一个结点后，不会使它失去平衡（即不会使它的平衡因子的绝对值大于 1），因此，最小不平衡子树的根， 只可能是距插入点最近（即距它所在树的根最远）（注意：新插入的结点总做为叶子）的原平衡因子不为 0 的结点。 至于它是否是真正的最小不平衡子树的根，还要看结点是否插入到它的偏重一端，若是插入到偏重端，则表明它即为所求的最小不平衡树，否则表明无最小不平衡树，即插入结点后并未破坏任何结点的平衡性。

找到最小不平衡子树后，还应修改它的根到插入点之间的各结点的平衡因子（修改为 1 或-1）， 然后调用前述的局部平衡调整算法即可。当然先应得知是插入方向（即插入到最小不平衡树的左子树还是右子树）。

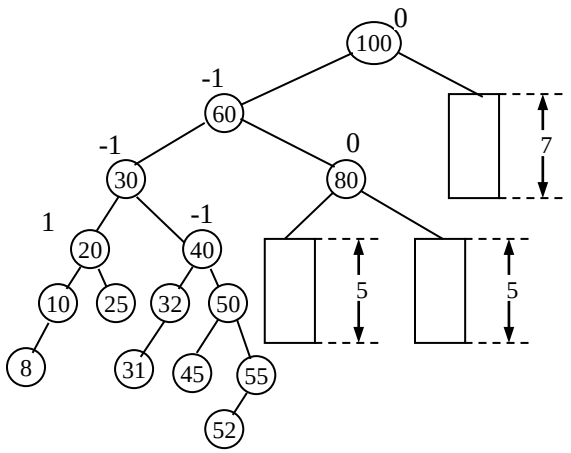
§9.5.5 平衡二叉树的删除

这里讨论删除平衡二叉排序树中某结点后对整棵树应做的平衡调整。

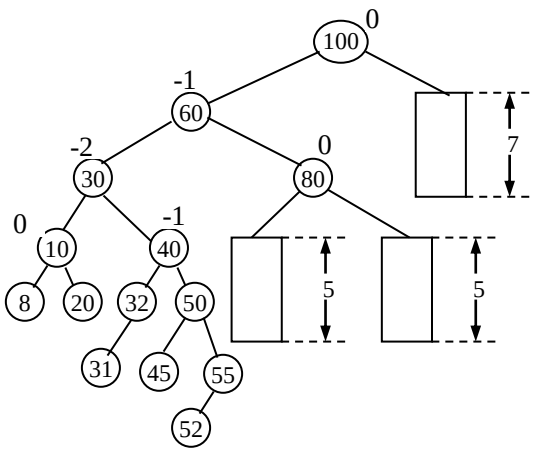
平衡二叉排序树的删除平衡调整与插入有所不同。对于插入，尽管他可能导致多棵子树失去平衡，但由于局部插入平衡算法会使被平衡的子树恢复原高度，所以，只要将插入导致的最小不平衡子树用局部平衡算法进行平衡，其它子树就会自动恢复平衡。但对于删除平衡，情况没有这样理想。这主要因为删除平衡算法不能保证在任何情况下都可将所平衡的树恢复原高度（事实上，它或者恢复原高度，或者使高度减 1），因此，用局部删除平衡算法平衡了某子树后，并不能保证使包含该子树的其它任何子树都自动恢复平衡，而可能带来新的不平衡，这种新产生的不平衡可能会随着局部平衡调整的进行而逐步向树根方向传播。图 9-0给出这种情况的一个例子。该例中，"失平衡"一直从结点 20 开始使传播到 60。该例中是对 60 进行 RR 调整，但若对 60 进行的是 RL 调整（当

80 的左子树高度大于右子树高度时就需要 RL），则原 60 子树高度会减 1，从而失平衡传向结点 100，但 100 的平衡因子为 0，则此传播被阻塞了。若 100 的平衡因子为-1，则 60 的降低会导致 100 的不平衡，调整 100 后使树的高度降 1。

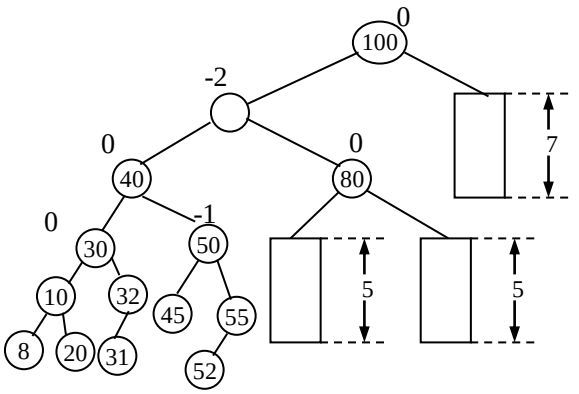
“失平衡”的这种传播也会被阻塞（终止）的。传播中，当遇到一个平衡因子为 0 的结点时，传播就会被终止。这是因为平衡因子为 0 的结点是等重的，一边高度降低 1 后不影响以该结点为根的子树的高度，对于传播路径（传播开始点到阻塞点之间的路径）上结点（它们的平衡因子不为 0），若“失平衡”是从它的偏轻分枝传来，则它本身也会失去平衡，需进行调整，我们称这种点为“失平衡点”；若“失平衡”是偏重端传来，则不影响它本身的平衡性，但使以它为根的子树的高度降低，因此“失平衡”可能“穿过”它向上传播，我们将这种点称为失平衡的传递点。



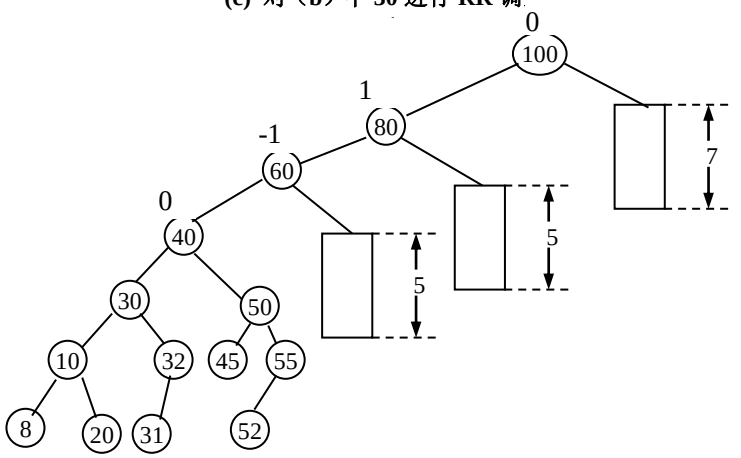
(a) 高为 8 的平衡二叉排序树



(b) 从（a）中删除结点 25 并对 20 进行 LL 调整后



(c) 对（b）中 30 进行 RR 调整



(d) 对（b）中 60 进行 RR 调整后

图 9-16 平衡二叉排序树删除平衡引起的失平衡传播（结点外标注的是结点的平衡因子）

因此，在平衡二叉排序树中删除一个结点后，要对失平衡传播路径上每个失平衡点调用局部平衡算法进行平衡调整。那么如何确定失平衡路径呢？让我们回忆一下二叉排序树的删除算法。若被删除结点 p 无左子树或无有子树，则情况比较简单，失平衡传播路径的起点就是该路径的父亲；若被删除结点既有左子树又有右子树，则可能有两条不同的传播路径，第一条起于被删除结点的父亲，第二条起于被删除结点的中序直接前驱。

至于阻塞点，它是距传播起点最近的平衡因子为 0 的点，它可在删除查找过程中求得，即在查找被删除结点中，每遇平衡因子为 0 的结点，就将它记下，这样，最后（查到被删除点时）所记录的为第一阻塞点，第二阻塞点可按同样方法在查被删除结点的中序直接前趋时记下。

对失平衡点的调整 ，可按先后代后祖先的次序调用局部平衡调整算法进行，因此，应将各失平衡点按代的次序记录在栈中。

§9.6 B 树

B 树是一种特殊的树，常用做索引，在文件系统和数据库系统中获得重要应用。

§9.6.1 B 树的概念

首先给出 B 树的定义：

一棵 m 阶 B 树，或为空树，或为满足下列条件的 m 叉树：

- 所有叶子结点都出现在同一层上。叶子不带信息。叶子的父亲称为**终端结点**。既非叶子又非终端结点的结点称为**内结点**；
- 每个非叶子结点，除根外，子树个数在$\lceil m/2 \rceil$到 m 之间。若根结点不是终端结点（根结点肯定不是叶结点），则其子树个数可以少到 2；这里形为$\lceil x \rceil$的式子表示不小于 x 的最小整数。
- 每个非叶子结点，除子树个数等信息外，还要包含下列形式的内容：

  （$P_1, K_1, P_2, K_2, \ \ldots, P_{n-1}\ K_{n-1}, P_n$）

  这里，n 为结点中实际具有的子树个数，K_i 为关键字（i=1, .., n-1），且 $K_i<K_{i+1}$（i=1, …, n-2）P_i(i=1, …, n)为指向子树根结点的指针，且指针 P_i 所指子树中所有结点的关键字值，均小于 K_i(i=1, …, n-1)，而均大于 K_{i-1} (i=2, …, n)；以后，我们为了说话方便，称 P_i 为 K_i 的**左子树**，P_{i+1} 为 K_i 的**右子树**。

由于叶子都出现在同一层，所以 B 树是一种平衡树。另外，每个结点中的关键字个数为子树个数减 1.

显然，B 树是二叉排序树的扩充，是一种多叉排序平衡树。图 9-0是一个 4 阶 B 树。

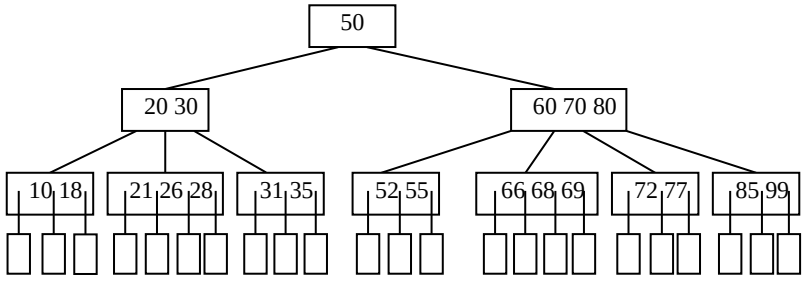


图 9-17 一棵 4 阶 B 树

在实际中，B 树的叶子可以有具体的含义。在一般情况下，B 树的叶子代表失败结点（外结点），它表示数据集范围。例如，图 9-0左起第二终端结点中的关键字 21 与 26 之间的叶子，可以代表关键字范围在 21 和 26 之间的数据集。一般情况下，B 树的叶子可以不画出。

§9.6.2 B 树的存储结构

B 树是一种多叉树，可采用普通的多叉树的存储结构。我们注意到，对 B 树，其主要操作是检索、插入和删除，而插入和删除都可能向上访问（在下面即将看到），故每个结点，设立父指针是需要的。其次，由于从某一结点访问它的儿子们时，一般是顺序访问，所以，每个结点只需设立指向大儿子的指针，而令各个儿子通过一个链指针链接为一个单链表。这种结构类似于树的儿子兄弟链存储法。

对 B 树结点内的内容，可认为是个具有 m-1 个元素的一维数组。我们称之为关键字数组。具体讲，B 树这种存储结构的树结点的结构为：

儿子个数	父亲指针	大儿子指针	兄弟子指针	关键字数组

对应的 C/C++描述为：

```
template <class TElem>
struct TBTreeNode
{
  long numSons; //儿子个数
  BTreeNode *father, //父亲指针
            *firstSon,   //第一个儿子的指针
            *nextBrother; //下个兄弟的指针，用于将各兄弟链为单链表
  TElem *keys; //关键字数组指针
  //下面是成员函数
…. …
};
```

当然，B 树也采用其他结构，例如，也可以采用多指针结构，即在结点中，设立 m 个儿子指针。当然也可以采用 m 叉树的其他存储结构。

§9.6.3 B 树的基本操作

B 树主要面向索引结构，而索引结构主要是面向检索操作。除检索外，结点的插入与删除在 B 树中具有特殊意义，涉及到保持平衡和语义的问题。

§9.6.4 B 树的检索方法

B 树的检索与二叉排序树类似。在某棵子树（包括根）中检索关键字 key 时，首先，在子树的根(记为 r)中搜索，若与某关键字 K_i 相等，则表示已找到，结束，否则，在 r 的子树 P_j 中，按类似方法继续检索。这里，j 满足

$$K_{j-1}<key<K_j$$

为了应用于插入操作，在检索不成功时，应返回关键字范围包括 key 的结点的指针，

该结点是叶子。

§9.6.5 B 树的插入方法

B 树的插入与删除都比较复杂，涉及到保持结构和保持语义问题。

为了处理方便，B 的插入，只在终端结点中插入。

假定要在 m 阶 B 树中插入关键字 key。设 n=m-1，即 n 为结点中关键字数目的最大值。首先，要查找插入位置（定位），由于是在终端结点中插入，所以要确定它属于那个终端结点，其方法与上面介绍的检索方法相同。定位的结果是返回了 key 所属叶结点的指针 p。定位后，就可以在 p 结点中插入 key。但由于 B 树结点的子树个数受限（不超过阶数 m），所以，若 p 中的关键字个数小于 n，则直接插入适当位置（同线性结构的插入），否则，插入点的关键字个数溢出，此时，我们将该结点 p 一分为二（称为“分裂”），设其前后部分分别是 p1 和 p2，然后，将中点关键字“上抽”到 p 的父亲，并令中点关键字的右指针指向 p2，其左指针 p1。分裂方法如图 9-0。

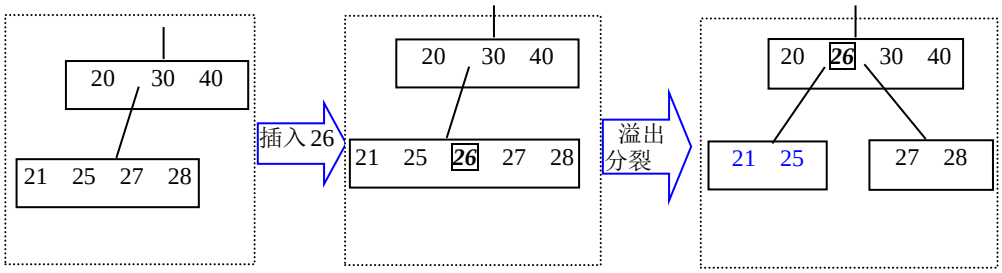


图 9-18 一个 5 阶 B 树插入的“分裂”

这里，“上抽”也相当于插入，即在 p 的父亲中插入一个关键字。如果父亲结点中关键字已满，就会导致父亲的分裂。所以，这种“上抽”引起的插入，也按类似过程进行。显然，这种分裂可能一直上上传，如果根也分裂了，则树长高了 1.

下面给出插入过程的严格描述。

1 [新建] 若在一个空结点（不存在的结点）中插入，则生成新结点，以 key 为关键字，结束。否则转下步。

2 [定位] 在 B 树中检索 key，若已存在 key，则特殊处理（转出），否则，得到 key 所属终端结点（key 的值介于该结点中的最小和最大关键字之间）的指针 p；

3 [插入] 将 key（及 key 的右子树）按序插入 p。

4 [判断] 若(插入后)p 不溢出（即 p 中关键字数目不超过 n），则结束，否则转下步；

5 [抽出] 将 p 中位于中点的关键字 K_i 取出（关键字数目为偶数时，中点为偏小的方向，下同）；

6 [分裂] 将 p 在中点处一分为二，所分得的两个结点分别记为 p1 和 p2，其中，p1

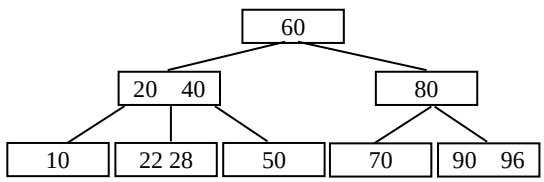
中关键字小于 p2；

7 [连接] 将 p1 作为原 p（即原指向 p 的指针指向 p1），将 p2 作为已抽出的 K_i 所对应的右子树 P_{i+1}，即令 P_{i+1} 指向 p2，（K_i 的左子树指针不变）。

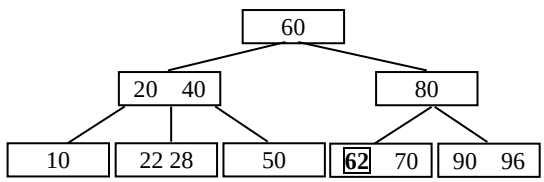
8 [上提] 若 p 是当前 B 树的根，则生成一个新结点，以 5 中取出的关键字 K_i 为其关键字，以 p1 和 p2 分别为 K_i 的左右子树，结束。若 p 不是当前 B 树的根，（则将 K_i 及其右子树 P_{i+1} 作为整体插入到 p 的父亲中），将 5 中取出的关键字 K_i，作为新的 key，将 p 的父结点作为新的 p，转 3。

上面的算法已很接近计算机程序了，具体的程序编制，留作练习。

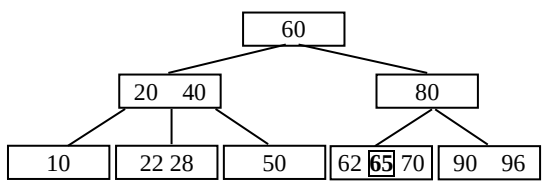
图 9-0给出了几个关于插入算法的完整例子。



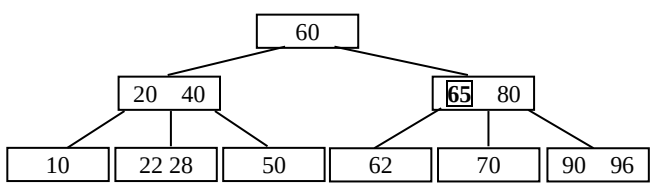
(a) 一棵 3 阶 B 树



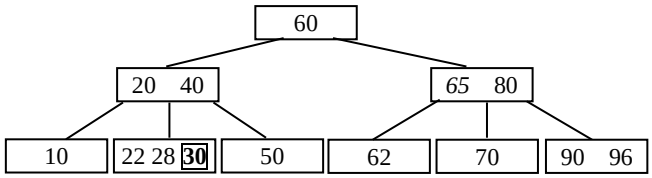
(b) 在 a 中插入 62



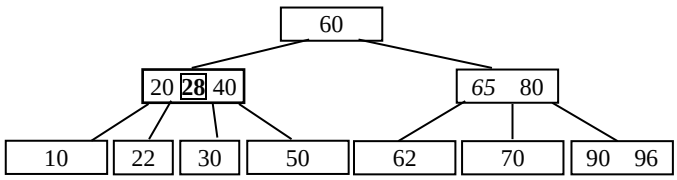
(c) 在 b 中插入 65，溢出，需要分裂



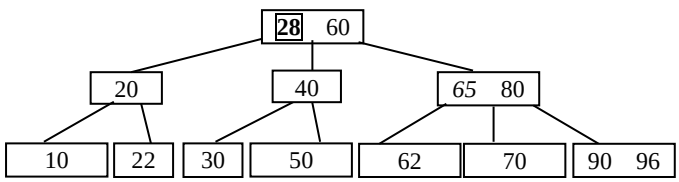
(d) 分裂 c 的(62,65,70)，65 被插入到它的父亲



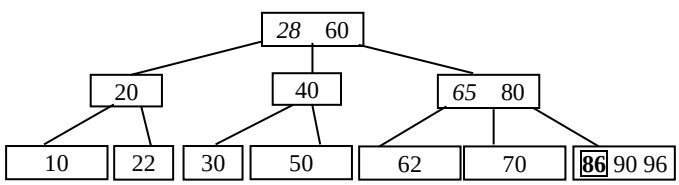
(e) 在 d 中插入 30，溢出，需分裂



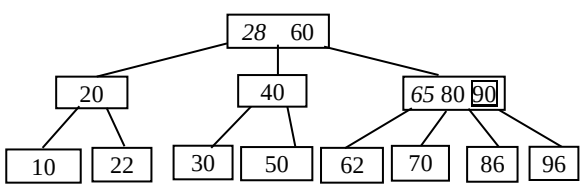
(f) 分裂 e 的(22,28,30)，28 被插入到它的父亲，再导致溢出



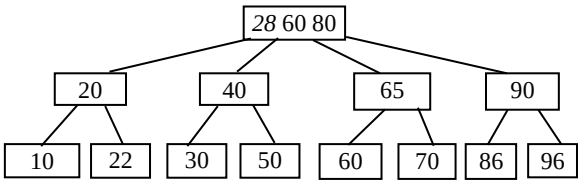
(g) 分裂 f 的(20,28,40)，28 被插入根



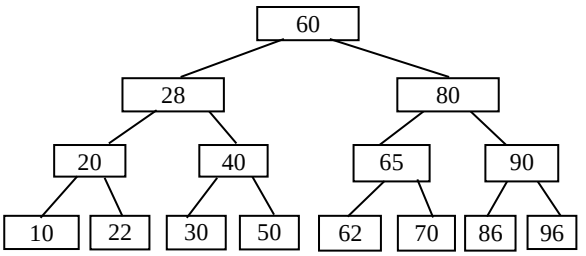
(h) 在 g 中插入 86，导致溢出



(i) 分裂 h 的(86,90,96)，90 被插入到它的父亲，导致父亲溢出



(j) 分裂 i 的(65,80,90)，80 被插入到它的父亲，导致父亲溢出



(k) 分裂 j 的(28,60,80)，导致树长高 1 级

图 9-19 一棵 3 阶 B 树(2-3 树)的插入示例

§9.6.6 B 树的删除方法

与插入操作类似，删除操作也是将删除归结到终端结点中进行。

设在 m 阶 B 树中删除关键字 key。首先需要查找 key 的位置，其方法与上面介绍的检索方法相同。找到 key 的位置后，即返回了 key 所属结点的指针 q，假定 key 是 q 中第 i 个关键字 K_i。若 q 不是终端结点，则直接删除 key，然后，用 P_i 所指子树中的最小键值 x 代替 K_i 的位置。这里，该 x 所在结点必为终端结点。这样，问题就归结为在终端中删除关键字。

如果在终端结点 p 中删除后，关键字数目为不足$\lceil m/2 \rceil-2$ ，则不符合 m 阶 B 树的要求，这需要借关键字，或合并结点。具体分两种情况：

(a) 兄弟够借：若 p 的左兄弟（或右兄弟）中关键字个数等于或大于$\lceil m/2 \rceil$，则将左兄弟（或右兄弟）中最大（或最小）的关键字（记为 x），上提到它的父亲结点中，代替父亲中的关键字 y，这里 y 满足：y 的左子树（或右子树）是 p。并且，将 y 下拉到 p 中最左（或右）。

(b) 兄弟不够借：即 p 的左右兄弟中关键字均不足$\lceil m/2 \rceil$，不能出借。此时需要将 p 的左兄弟（或右兄弟）合并。由于合并后 p 的父亲少了一个子树，所以也有将 p 的父亲中的一个关键字下拉到 p 中。这个关键字 x 应该是被合并的两个结点的“根”，即 x 的左右子树分别为被合并的两个结点。的具体的合并方法是，将 p 的左兄弟（或右兄弟）中

的所有关键字，连同从父亲下拉来的 x，依次加入到 p 中，并在 p 中删除 x 及指向被合并了的结点的指针。合并后，p 的父亲的关键字数目减少了 1，如果减少后 p 的父亲中关键字数目没有下溢，则该合并过程结束，否则，父亲结点也需要借结点或合并，该过程类似于在父亲中删除。显然，合并过程也可能一直上传到根，若使根变空，则需要删除该根，导致树高减 1.

删除引起的“借”和“合并”过程的图示如图 9-0。

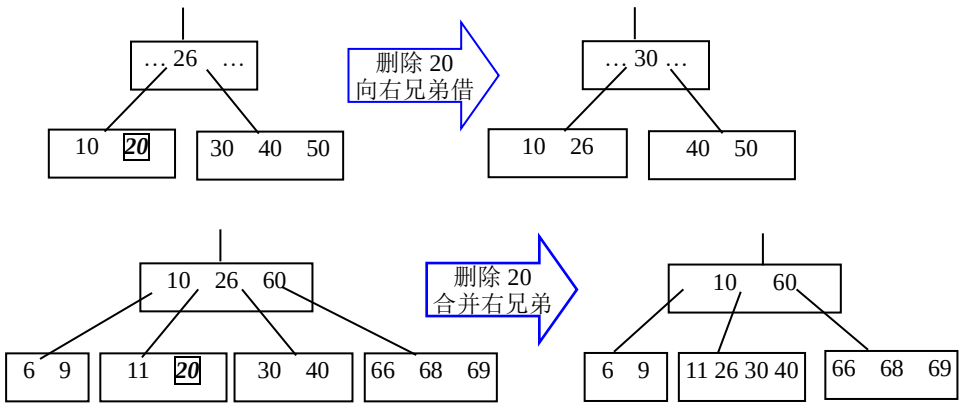


图 9-20 一个 5 阶级 B 树的删除引起的“合并”

删除操作的严格描述如下。

1 [定位] 在 B 树中检索 key，若不存在 key，则特殊处理（转出），否则，得到 key 所属结点的指针 q；

2 [替换] 若 q 不是终端结点，则找到 key 的右子树中最小关键字 x（其必在终端结点中，设该终端结点指针为 p，它的父亲为 p0），用 x 代替 q 中的 key 值。若 q 是终端结点，则也记 p 为该终端结点指针，记待删关键字为 x；转下步（删除 x）；

3 [删除] 若 p 是根，且关键字数目 n>1，或 p 不是根，但关键字数目 n>⌈m/2⌉-1，则直接在 p 中将 x 删除，结束。否则，转下步；

4 [平借]（此时，p 不是根且关键字数目 n<=⌈m/2⌉-1，或 p 是根，但关键字数目 n<=1）若与 p 相邻的右兄弟（或左兄弟）的关键字数目 n>⌈m/2⌉-1（兄弟够借），则按下面的子步骤调整 p 及 p 的相邻兄弟，否则（兄弟不够借）转 5；

4.1 [下拉] 将 p 的双亲结点 p0 中刚刚大于（或小于）x 的关键字 K_i，下拉到 p 中适当位置（末尾/开头）；

4.2 [上推] 将右兄弟（或左兄弟）中最小（或最大）关键字，上推到 p 的双亲结点中的 K_i 的位置；

4.3 [平移]将右兄弟（或左兄弟）中最左（或最右）指针（即上推到父亲的结点的左/右指针），平移到 p 中最后（或最前）子树指针位置。

4.4 [调整] 在右兄弟（或左兄弟）结点中，将被移走的关键字和指针位置用剩余的

关键字和指针填补、调整。转 6；

5 [合并]（兄弟不够借），则按下面的子步骤合并 p 和 p 的某个兄弟：

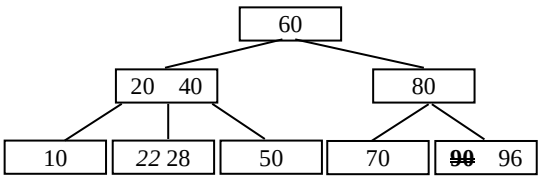
5.1 [上借] （p 或 p 的某兄弟向双亲结点借关键字，即将 p 的双亲结点 p0 的相应关键字下移到 p 或 p 的某兄弟）此时，可合并 p 和 p 的左兄弟（若存在），或合并 p 和 p 的右兄弟（若存在）。不论那种情况，我们假定选定合并的两个结点的（在 p0 中的）指针分别为 P_i 与 P_{i+1}，且逻辑上保留 P_i 所指结点，则把 p0 中 P_i 对应的关键字 K_i（它大于 P_i 所指结点中所有结点）的复制品下移到 P_i 所指的结点中适当位置（最右）；（保留 P_{i+1} 的情况类似）；

5.2 [复制] 将 P_{i+1} 所指结点中全部指针和关键字照搬到 P_i 所指结点的后面。删去 P_{i+1} 所指结点；

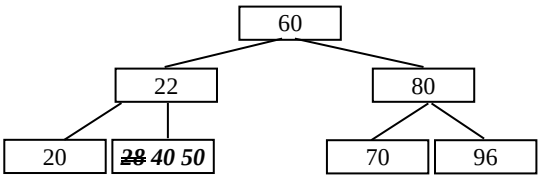
6 [重复]令 p 指向 p0 所指结点，同时令 p0 为新 p 的父亲，视 K_i（在 5.1 中被复制下移的关键字）为新的 x，转 3（继续在父亲中删除）。

该算法也已很接近计算机程序了，程序的编写留作练习。

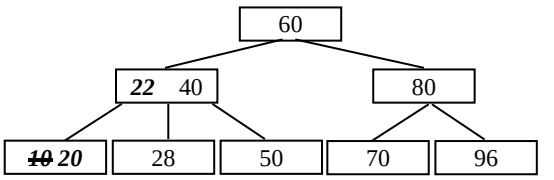
下面给出几个关于删除操作的具体例子，见图 9-0。



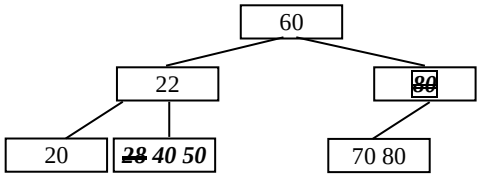
(a) 从图 9-19(a)中删除 90 后



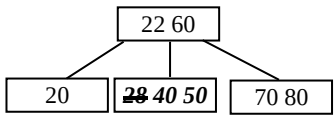
(c) 从(b)中删除 28，兄弟不够借，合并右兄弟



(b) 从(a)中删除 10，从右兄弟借



(d) 从(c)中删除 96，兄弟不够借，合并左兄弟，引起父亲下溢



(e) 处理 d 中的下溢，相当于在 60 的左儿删除 80,此时不够借，合并左兄弟，导致根空，故删除根结点，树高降 1

图 9-21 B 树删除示例(2-3 树)

§9.6.7 B^+树（注：是 B+树还是 B^+树？？？？----答：是 B^+）

B^+树是 B 树的变种。一棵 m 阶 B^+树，在结构上与 m 阶 B 树相同，但在结点内部的关键字安排不同。具体不同点（或需要强调的）如下：

- 具有 n 棵子树的结点含 n 个关键字，即每个关键字对应一棵子树；
- 结点内关键字仍然按序排列（与 B 树相同）
- 关键字 K_i 是它所对应的子树的根结点中的最大（最小）关键字。
- （关键字符合二叉排序树要求）对同一结点内的任意两个关键字 K_i 和 K_j，若 $K_i<K_j$，则 K_i 小于 K_j 对应的子树中的所有关键字（与二叉排序树及 B 树类似）。
- （由上面一点推知）所有终端结点中包含了所有的关键字；
- 终端结点中的关键字对应的子树（叶子），实际应用中一般代表纪录块。关键字 K_i 是它所代表的纪录块中的最大（最小）关键字。
- 各终端结点可以按关键字大小次序链接在一起（形成单链表），并设置链头指针。

图 9-0给出了一棵 3 阶 B^+树.

显然，B^+树中，某结点中的关键字，均是它的各儿子结点中最大（最小）关键字的复写，所以，如果叶子是分块有序记录集，则 B^+树相当于记录集的**多级索引**。

B^+树上有两个头指针，一个指向根结点，另一个指向关键字最小的终端结点。

B^+树有两种查找操作：一种是从最小关键字起顺序查找；另一种是从根结点开始进行随机查找。

在 B^+树上查找时，若非终端结点上的键值等于给定值，并不终止查找过程，而是继续向下查找直至终端。因此，在 B^+树中，不管查找成功与否，每次查找都是走了一条从根到终端结点的完整路径。

在 B^+树的插入和删除和在 B 树上类似，也仅在终端结点上进行。但由于关键字的安排有所不同，所以，具体算法也有所不同。

B^+树广泛地使用在包括 VSAM 文件在内的多种文件系统中。

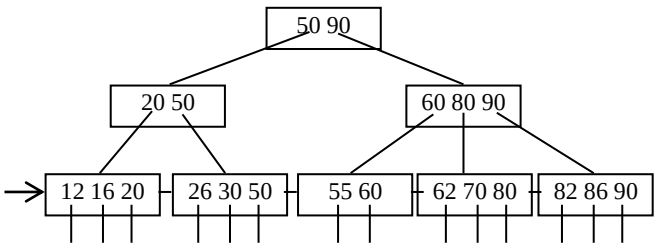


图 9-22 一棵 3 阶 B+树

§9.6.8 B 树对象模型

B 树是一种特殊的多叉树，所以，从对象关系方面讲，B 树的逻辑/抽象（存储结构无关的）模型应该是多叉树的派生，所需扩展或增加的方法是查找、插入和删除等。

但是，B 树的用途一般比较单一，主要面向查找、插入和删除等操作，所以，若为了提高效率，也可设立独立的 B 树结构类（与结点内的关键字无关），然后再在此基础上派生出具体的 B 树类和 B+树类。具体的做法，留作练习。

§9.7 散列结构

§9.7.1 概念

在第一章中，我们已初步介绍过散列，那里，我们是将散列作为一种存储结构介绍的。事实上，散列主要是作为面向检索的存储结构，即检索结构。

与其他检索结构不同，在散列结构上的检索，是基于“计算”的，而“比较”操作退居次要位置。

一个**散列结构**是一块连续的存储空间，它与一个称为**散列函数**的函数关联，该函数是数据记录的关键字到地址空间的映射：

hash ： 关键字→ 地址

这种存储空间的使用方式为：

(a)存储记录时，通过散列函数计算出记录的存储位置：

记录位置=hash (记录的关键字）

并按此存储位置存储记录；

(b)访问记录时，同样利用散列函数计算存储位置，然后根据所计算出的存储位置访问记录。

散列也称**杂凑**或**哈希**(Hash)。

从上面的说明看出，散列结构不是一种完整的数据存储结构，因为它只是通过数据元素的关键字定位记录，一般很难完整地表达数据元素间的逻辑关系，所以，一般也主要面向检索操作。

显然，由于记录的定位主要基于计算，所以，一般情况下，散列方法的检索速度要比前面介绍过的基于比较的方法高。但是，散列结构的存储利用率一般不高。

§9.7.2 散列技术中的主要问题

表面上看，如果只是为了检索操作，设置了散列函数后，只需简单的计算即可。事实上问题并没有这样简单。总结起来，有下列问题：

- 冲突处理：一般情况下，设计出散列函数很难是单调的，即不同关键字可能对应相同的函数值，这样，就造成地址**冲突**（**碰撞**）。
- 散列函数的设计：设计一个简单、均匀、存储利用高、冲突少的散列函数，是问题的关键。

§9.7.3 散列过程

散列方法的一般使用方法是，首先要根据具体问题的特点（记录集的特点、存储空间的限制等）设计散列函数。确定了散列函数和存储空间后，一般的进行过程为：

[1] 如果是存储（插入）记录，则提出记录的关键字，用散列函数计算出地址 addr；

[2] 如果 addr 处已被其他记录占用（**插入冲突**），则调用[5]，否则，直接将记录存入该位置，结束。

[3] 如果是访问记录（查找，删除、修改等），则提出记录的关键字，用散列函数计算出地址 addr；

[4] 检查 addr 处的记录是否为所要查找（定位）的记录，若不是（**定位冲突**），调用[5]；若是，则读出该记录，进行相应的处理，结束；

[5] (冲突处理)按既定的冲突处理策略处理冲突。如果是插入冲突，冲突处理是要找一个空位置（返回其）；如果是定位冲突，冲突处理是要继续查找所要找的记录，确定记录是否在散列结构中，若是，则返回其位置，否则返回特殊标记。

§9.7.4 散列函数的设计

散列函数的设计是个关键问题，也是个复杂问题。但是，散列函数设计一般没有什么严格的规则，只有一些可以遵从的原则。

(一) 设计原则：

根据散列函数的用途，设计散列函数时一般应遵循下列原则：

- 符合地址空间范围：函数值取值的范围，要在允许的存储空间的地址范围（地址空间）内。
- 函数值尽量分布均匀：函数值空间要尽量均匀散布在地址空间，即函数值之间的差要尽量小、尽量接近。这样才保证存储空间的有效使用，并减少冲突。
- 函数尽量单调（少冲突）：非单调的函数意味着不同的关键字（因变量），可能对应相同的函数值，这样会导致冲突，所以，要尽量使函数具有较少的极值点。
- 计算简单：散列函数不应该有很大的计算量，否则会降低效率。

下面就介绍几种常见的设计方法。

(二) 直接定址法

直接定址法的散列函数为关键字的线性函数:

Hash(key) = a*F(key) + b

这里，F 是个简单函数（变换），一般负责对关键字进行截取、组合、转换之类的操作。a 与 b 为常数，可用于调节范围和起点。

这种函数的特点是单调（无冲突）、均匀，且范围容易确定。

例如，若一个数据集如下所示

职工号	姓名	年龄	…
A01	李华	30	
A02	章宏	40	
A03	伊林		
B10	何光	30	
B11	李华	20	
…	…	…	…

在该数据集中，可取职工号为关键字，其后两位数字分布比较均匀，所以，如果记录总数不超过 100，则截取职工号的后两位做为地址，即散列函数为：

Hash(key) = StrRight(key,2)

这里，StrRight 充当变换函数，负责将 key 的最右两位取出。在实际中，还需要将字符串形式转换为数值形式。显然，如果职工记录超过 100，仅取两位是不够的，应该将首位的英文字母也取来（即全部取来），这时，需要将字母转化为数字。具体做法可以是：

(ASC(StrLeft(key,1))-65) * 100 + StrToInt(StrRight(key,2))

这里，ASC 表示求字符的 ASCII 编码（值），StrLeft 表示取前缀，StrtoInt 表示将串转化为整数。

(三) 除留余数法

除留余数法的基本思想是，利用求余数操作取关键字的末尾若干位做为地址值。散列函数的形式为：

Hash(key) = key MOD p

其中，MOD 表示求余数运算，p 是某个常数，其首先要满足地址空间的要求，即如果地址空间范围为[0，x]，则 p 的取值应该等于或大于 x。另外，p 最好为素数。通过分析可以知道，在 p 为素数时，函数值的分布比较均匀，较少冲突。

(四) 除余取整法

这种方法的散列函数形式为：

Hash(key) = n*⌊ (A*key % 1) ⌋

这里，0<A<1，A*key % 1 表示取 a*key 的小数部分，即

A*key % 1 = A*key -⌊ (A*key) ⌋

对于 A 的取值，Knuth 做了仔细分析，他认为，虽然 A 取 0 到 1 间任何值都可以，但取黄金分割数($\sqrt{5}$ -1)/2 = 0.6180339…最好。

(五) 平方取中法

这种方法的散列函数，是先求关键字的平方值，然后，在平方值中取中间几位（平方和截取）。之所以这样，是因为平方值中，靠近中间的几位与原值的各位相关度较大，所以，取中间位能保证关键字的敏感，即不同的关键字，较少有相同平方和截取，从而减少冲突。

(六) 数字分析法

数字分析法也是一种关键字截取的方法，它分析各个关键字的位的构成，截取其中若干位做为散列函数值，尽可能使关键字具有大的敏感度。

例如，已知各关键字如下：

key1: 3 3 4 8 9 2

key2: 3 1 8 7 1 1

key3: 3 1 2 9 2 3

key4: 3 2 3 8 1 2

…

分析它们的位的结构，发现第 3 和第 4 位对关键字的敏感度较大，所以可以取这两位为散列函数值。

显然，这种方法适合于关键字事先已知的情况。

(七) 折叠法

这种方法是将关键字值从左到右分成长度相等的若干段（最后一段可能短于其他段），然后将各段叠加（舍弃进位），取叠加和做为对应关键字的散列函数值。

这种方法方法显然与平方取中法类似。

§9.7.5 冲突解决

一般情况下，由于关键字的复杂性和随机性，很难有理想的散列函数存在，所以，冲突一般是很难避免的。因此需要有合适的**冲突解决方法**，即冲突发生时的应对措施。为了方便，首先引入桶的概念。

(一) 桶

若散列表有 n 个逻辑地址，则将其视为具有 n 个**桶**(Bucket)，且为每个桶编号（桶号），并令桶号与散列地址一一对应。设每个桶可存放 s 个表项，称它们为**同义词**（散列函数值相同的关键字称为同义词）。

因此，桶是用于装载冲突项的。当欲装入某桶中的同义词个数大于桶容量时，称为溢出。

对桶的处理不同，就对应于不同的冲突处理方法。

(二) 开放地址法

这种方法是，将所有的桶都设在地址空间内。每个桶大小为 1 个表项。一个地址，就是一个桶。

在插入时，若发现冲突，在其他位置寻找空桶。有下列几种寻找空桶法。

1．线性探测法

冲突时，从冲突位置的下个位置起，依次寻找空桶。

2．二次探测法

冲突时，下个被检查的桶的桶号，等于 $a+k^2$，这里，a 为冲突地址，k 为探测的次数，k=1, 2, …。

(三) 链地址法

这种冲突处理方法的基本思想是，每个桶为一个链表，其存放着对应的同义词。具体的操作方法类似于链表。

具体实现时，链表可以是动态结构，也可以是静态结构，特别是，在静态结构下，可以省略链指针，使每个桶成为一个简单的线性结构，如栈。

(四) 溢出区法

这种方法的基本思想是，将散列表分为两个区，一个称为基本区，另一个称为溢出区。基本区的基本单元数目为散列函数的（有效）取值个数，每个单元存放一个记录。当发生冲突时，将当前冲突记录顺序存入溢出区。

与该方法相比，链地址法中相当于有多个溢出区。

本章小结

检索结构一般是为了提高检索速度而专门引入的数据结构，充当索引。当数据集很大（特别是数据集在外存）时，更有必要引入结构。索引也是数据集，它中的每一项（每一索引纪录）对应于数据集中一条纪录（稠密索引），或若干条连续出现的记录（分块索引），用于指出所对应的数据记录（或记录块）的存储位置。在索引中，用数据记录的关键字标识数据记录。

索引一般有两大类：线性结构和树结构。线性索引中，记录关键字一般是按序排列的（线性索引是有序表），以提高检索速度（使用折半检索、斐波那契检索等）。树形索引有二叉排序树、平衡二叉排序树、B 树等。树形检索的时间复杂度与树的深度同级，所以一般是索引项数目的对数函数。

散列是一种特殊的检索结构。确切地讲，散列结构是面向检索的存储结构，没有单独的索引。数据的存储地址是根据数据的关键字值“计算”。负责这种计算的是散列函数。散列函数的设计是散列法的关键。如果散列函数不是单调的，则不同关键字就会对应相同的存储位置----导致冲突。所以必须有适当的冲突处理方法。冲突的存在，就导致存储地址不能均匀使用。即使没有冲突，散列函数的取值也可能与可用的地址空间不一致，所以，散列函数的设计尤为重要，要尽可能少冲突，取值均匀，还要计算简单。

习 题

1. 简要回答下列问题：
   - 什么是检索结构？为什么要引入检索结构？

- 检索结构一般可分为哪几类？
- 检索算法分为哪几类？
- 检索算法的时间性能与哪些因素相关？
- 为什么要引入平均检索长度？它与时间复杂度有何异同？

2. 拆半检索算法是否适合链式存贮结构？为什么？
3. 编写一个实现插值检索的程序。
4. 扩充折半检索算法， 使它能在检索失败时指出待查关键字值在数据集中的排序位置（即指出待查关键字按大小次序应位于数据集中第几个位置）。
5. 按上题要求扩充 Fibonacii 检索算法。
6. 编写实现分块索引的索引创建子程序与索引顺序检索子程序。假定， 数据集（已有序）与索引表均为一维数组。
7. 编写一个非递归的二叉排序树检索子程序和一个非递归的二叉排序树插入位置检索子程序。
8. 编写递归程序，实现检索指定关键字对应的结点的父亲。
9. 设某关键字序列为（100,50,80,70,160,95,88,180,120,99），请画出按此序列次序建立的二叉排序树，然后画出按删除算法删除结点 80 后的树。
10. 编写一个二叉排序树删除子程序， 要求用被删结点的右子树（若存在）替代被删结点的位置。
11. 编写一个 B 树的定位（查找）程序（递归和非递归），找到时，给出所找到的结点的父亲；找不到时，给出待找关键字所属的范围（即在终端结点中的叶子指针）
12. 编写一个 B 树的插入程序（递归和非递归）
13. 编写一个 B 树的删除程序（递归和非递归）
14. 给出完整的 B 树（结点和树两个方面）类定义，包括逻辑（抽象）类、存储实现类。
15. 给出完整的 B+树（结点和树两个方面）类定义，包括逻辑（抽象）类、存储实现类。