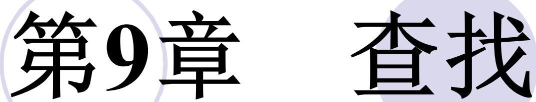


第9章　查找

数据的组织和查找是大多数应用程序的核心，而查找是所有数据处理中最基本、最常用的操作。特别当查找的对象是一个庞大数量的数据集合中的元素时，查找的方法和效率就显得格外重要。

本章主要讨论顺序表、有序表、树表和哈希表查找的各种实现方法，以及相应查找方法在等概率情况下的平均查找长度。

9.1 查找的概念

查找表(Search Table)：相同类型的数据元素(对象)组成的集合，每个元素通常由若干数据项构成。

关键字(Key，码)：数据元素中某个(或几个)数据项的值，它可以标识一个数据元素。若关键字能唯一标识一个数据元素，则关键字称为主关键字；将能标识若干个数据元素的关键字称为次关键字。

查找/检索(Searching)：根据给定的K值，在查找表中确定一个关键字等于给定值的记录或数据元素。

◆ 查找表中存在满足条件的记录：查找成功；结果：所查到的记录信息或记录在查找表中的位置。

◆ 查找表中不存在满足条件的记录：查找失败。

查找有两种基本形式：静态查找和动态查找。

静态查找(Static Search)：在查找时只对数据元素进行查询或检索，查找表称为静态查找表。

动态查找(Dynamic Search)：在实施查找的同时，插入查找表中不存在的记录，或从查找表中删除已存在的某个记录，查找表称为动态查找表。

查找的对象是查找表，采用何种查找方法，首先取决于查找表的组织。查找表是记录的集合，而集合中的元素之间是一种完全松散的关系，因此，查找表是一种非常灵活的数据结构，可以用多种方式来存储。

根据存储结构的不同，查找方法可分为三大类：

① 顺序表和链表的查找：将给定的K值与查找表中记录的关键字逐个进行比较， 找到要查找的记录；

② 散列表的查找：根据给定的K值直接访问查找表，从而找到要查找的记录；

③ 索引查找表的查找：首先根据索引确定待查找记录所在的块，然后再从块中找到要查找的记录。

查找方法评价指标

查找过程中主要操作是关键字的比较，查找过程中关键字的平均比较次数(平均查找长度ASL：Average Search Length)作为衡量一个查找算法效率高低的标准。ASL定义为：

$$ASL=\sum_{i=1}^{n} P_i \times C_i \quad n\text{为查找表中记录个数} \qquad \sum_{i=1}^{n} P_i=1$$

其中：

P_i：查找第i个记录的概率，不失一般性，认为查找每个记录的概率相等，即$P_1=P_2=\ldots=P_n=1/n$；

C_i：查找第i个记录需要进行比较的次数。

一般地，认为记录的关键字是一些可以进行比较运算的类型，如整型、字符型、实型等，本章以后各节中讨论所涉及的关键字、数据元素等的类型描述如下：

典型的关键字类型说明是：

```
typedef  float   KeyType ;      /*  实型  */
typedef  int   KeyType ;      /*  整型  */
typedef  char  KeyType ;      /*  字符串型  */
```

数据元素类型的定义是：

```
typedef  struct  RecType
  {  KeyType   key ;          /* 关键字码  */
     ┇   /* 其他域  */
  }RecType ;
```

对两个关键字的比较约定为如下带参数的宏定义：

```
/*  对数值型关键字  */
#define  EQ(a, b)   ((a)==(b))
#define  LT(a, b)   ((a)<(b))
#define  LQ(a, b)   ((a)<=(b))
/*  对字符串型关键字  */
#define  EQ(a, b)   (!strcmp((a), (b)) )
#define  LT(a, b)   (strcmp((a), (b))<0 )
#define  LQ(a, b)   (strcmp((a), (b))<=0 )
```

9. 2 静态查找

静态查找表的抽象数据类型定义如下：

ADT Static_SearchTable{

数据对象D：D是具有相同特性的数据元素的集合，
各个数据元素有唯一标识的关键字。

数据关系R：数据元素同属于一个集合。

基本操作P：

⋮

} ADT Static_SearchTable 详见p_{216}。

线性表是查找表最简单的一种组织方式，本节介绍几种主要的关于顺序存储结构的查找方法。

9.2.1 顺序查找(Sequential Search)

1 查找思想

从表的一端开始逐个将记录的关键字和给定K值进行比较，若某个记录的关键字和给定K值相等，查找成功；否则，若扫描完整个表，仍然没有找到相应的记录，则查找失败。顺序表的类型定义如下：

```
#define MAX_SIZE  100
typedef  struct  SSTable
  {  RecType  elem[MAX_SIZE] ;    /*  顺序表  */
     int  length ;     /*  实际元素个数  */
  }SSTable ;
```

```
int  Seq_Search(SSTable  ST , KeyType key)
  {   int p ;
      ST. elem[0].key=key ;    /* 设置监视哨兵,失败返回0  */
      for (p=ST.length; !EQ(ST. elem[p].key, key); p--)
      return(p) ;
  }
```

比较次数：

查找第n个元素：　1

……….

查找第i个元素：　n-i+1

查找第1个元素：　n

查找失败：　n+1

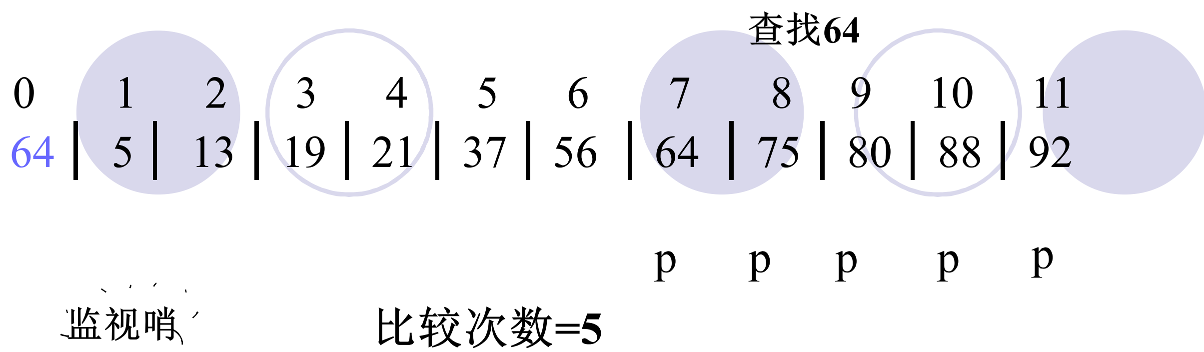


图9-1 顺序查找示例

2 算法分析

不失一般性，设查找每个记录成功的概率相等，即 $P_i=1/n$；查找第i个元素成功的比较次数 $C_i=n-i+1$；

- 查找成功时的平均查找长度ASL：

$$ASL=\sum_{i=1}^{n} P_i \times C_i = \frac{1}{n}\sum_{i=1}^{n}(n-i+1) = \frac{n+1}{2}$$

◆ 包含查找不成功时：查找失败的比较次数为**n+1**，若成功与不成功的概率相等，对每个记录的查找概率为$P_i=1/(2n)$，则平均查找长度**ASL**：

$$ASL=\sum_{i=1}^{n} P_i \times C_i = \frac{1}{2n}\sum_{i=1}^{n}(n-i+1)+\frac{n+1}{2}=3(n+1)/4$$

9.2.2 折半查找(Binary Search)

折半查找又称为二分查找，是一种效率较高的查找方法。

前提条件：查找表中的所有记录是按关键字有序(升序或降序) 。

查找过程中，先确定待查找记录在表中的范围，然后逐步缩小范围(每次将待查记录所在区间缩小一半)，直到找到或找不到记录为止。

1 查找思想

用**Low**、**High**和**Mid**表示待查找区间的下界、上界和中间位置指针，初值为**Low=1**，**High=n**。

(1) 取中间位置**Mid**：**Mid=⌊(Low+High)/2⌋**；

(2) 比较中间位置记录的关键字与给定的**K**值：

① 相等： 查找成功；

② 大于：待查记录在区间的前半段，修改上界指针： **High=Mid-1**，转(1)；

③ 小于：待查记录在区间的后半段，修改下界指针：**Low=Mid+1**，转(1)；

直到越界**(Low>High)**，查找失败。

2 算法实现

```
int  Bin_Search(SSTable  ST , KeyType  key)
  {   int  Low=1，High=ST.length, Mid ;
     while (Low<High)
       {    Mid=(Low+High)/2 ;
          if  (EQ(ST. elem[Mid].key, key))
             return(Mid) ;
          else if (LT(ST. elem[Mid].key, key))
                  Low=Mid+1 ;
               else   High=Mid-1 ;
       }
     return(0) ;      /*  查找失败  */
  }
```

3 算法示例

如图**9-2(a)，(b)**所示。

查找**21**

1	2	3	4	5	6	7	8	9	10	11
5	13	19	21	37	56	64	75	80	88	92

Low Mid High

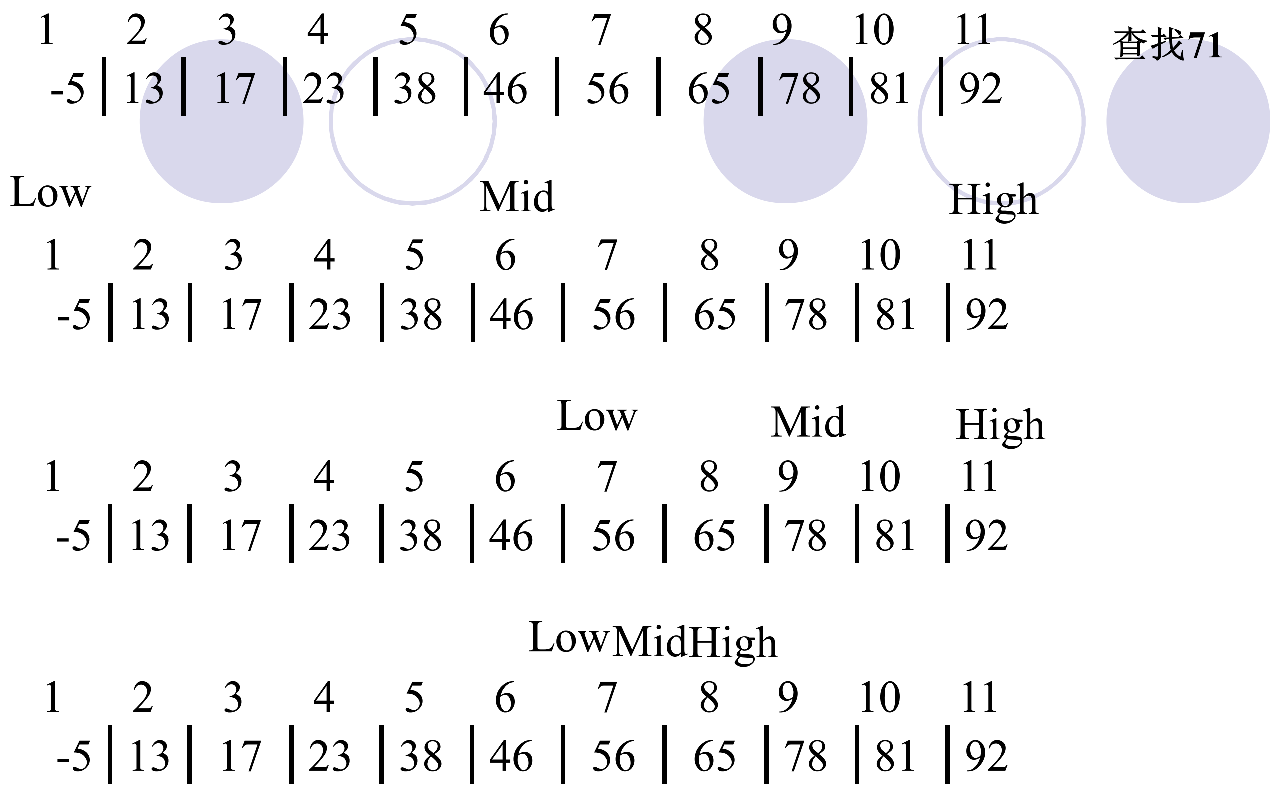
1	2	3	4	5	6	7	8	9	10	11
5	13	19	21	37	56	64	75	80	88	92

Low Mid High

1	2	3	4	5	6	7	8	9	10	11
5	13	19	21	37	56	64	75	80	88	92

LowMidHigh

(a) 查找成功示例



(b) 查找不成功示例

图**9-2** 折半查找示例

4 算法分析

① 查找时每经过一次比较，查找范围就缩小一半，该过程可用一棵二叉树表示：

- 根结点就是第一次进行比较的中间位置的记录；
- 排在中间位置前面的作为左子树的结点；
- 排在中间位置后面的作为右子树的结点；

对各子树来说都是相同的。这样所得到的二叉树称为判定树(**Decision Tree**)。

② 将二叉判定树的第$\lfloor \log_2 n \rfloor+1$层上的结点补齐就成为一棵满二叉树，深度不变，$h=\lfloor \log_2(n+1) \rfloor$。

③ 由满二叉树性质知，第i 层上的结点数为2^{i-1}($i \le h$)，设表中每个记录的查找概率相等，即P_i=1/n，查找成功时的平均查找长度ASL：

$$ASL=\sum_{i=1}^{n} P_i \times C_i = \frac{1}{n}\sum_{j=1}^{h} j \times 2^{j-1} = \frac{n+1}{n}\log_2(n+1)-1$$

当n很大 (n>50)时， $ASL \approx \log_2(n+1)-1$。

9.2.3 分块查找

分块查找(**Blocking Search**)又称索引顺序查找，是前面两种查找方法的综合。

1 查找表的组织

① 将查找表分成几块。块间有序，即第**i+1**块的所有记录关键字均大于(或小于)第**i**块记录关键字；块内无序。

② 在查找表的基础上附加一个索引表，索引表是按关键字有序的，索引表中记录的构成是：

最大关键字
起始指针

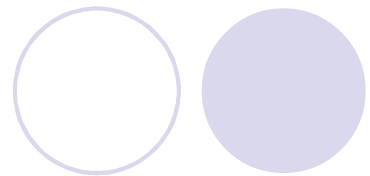
2 查找思想

先确定待查记录所在块，再在块内查找(顺序查找)。

3 算法实现

```
typedef struct IndexType
  { keyType  maxkey ;     /* 块中最大的关键字 */
     int startpos ;     /* 块的起始位置指针 */
  }Index;
```

```
int Block_search(RecType ST[] , Index ind[] , KeyType
key , int n , int b)
    /*  在分块索引表中查找关键字为key的记录 */
    /*表长为n ， 块数为b */
   {   int i=0 , j , k ;
      while ((i<b)&&LT(ind[i].maxkey, key) )  i++  ;
         if (i>b) {  printf("\nNot found");   return(0);  }
      j=ind[i].startpos ;
      while ((j<n)&&LQ(ST[j].key, ind[i].maxkey) )
         {   if ( EQ(ST[j].key, key) )  break ;
            j++ ;
         }    /*  在块内查找  */
```



```
    if (j>n||!EQ(ST[j].key, key) )
      {  j=0; printf("\nNot found");  }
    return(j);
}
```

4 算法示例

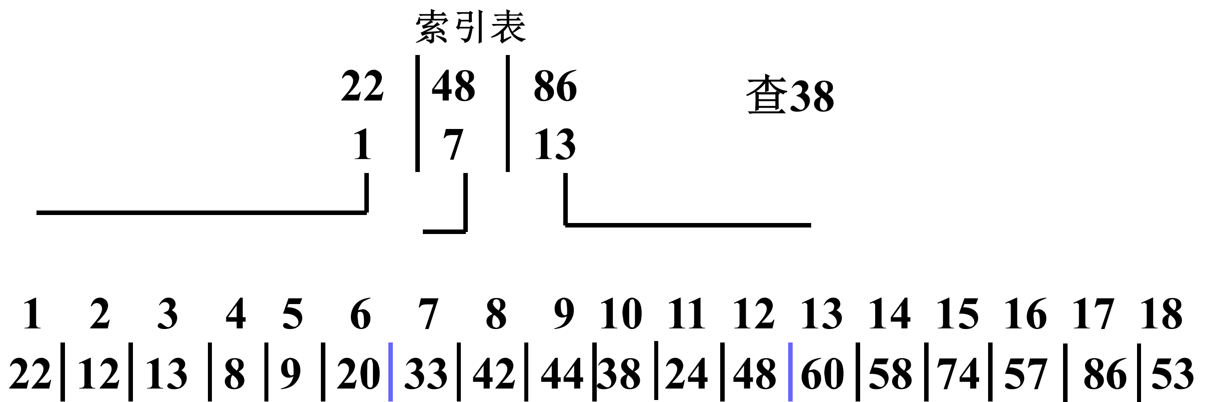


图9-3 分块查找示例

5 算法分析

设表长为n个记录，均分为b块，每块记录数为s，则 $b=\lceil n/s \rceil$。设记录的查找概率相等，每块的查找概率为1/b，块中记录的查找概率为1/s，则平均查找长度ASL：

$$ASL=L_b+L_w=\sum_{j=1}^{b} j+\frac{1}{s}\sum_{i=1}^{s} i=\frac{b+1}{2}+\frac{s+1}{2}$$

9.2.4 Fibonacci查找

Fibonacci查找方法是根据**Fibonacci**数列的特点对查找表进行分割。**Fibonacci**数列的定义是：

F(0)=0，**F(1)=1**，**F(j)=F(j-1)+F(j-2)** 。

1 查找思想

设查找表中的记录数比某个**Fibonacci**数小**1**，即设**n=F(j)-1**。用**Low**、**High**和**Mid**表示待查找区间的下界、上界和分割位置，初值为**Low=1**，**High=n**。

⑴ 取分割位置**Mid**：**Mid=F(j-1)** ；

⑵ 比较分割位置记录的关键字与给定的**K**值：

① 相等： 查找成功；

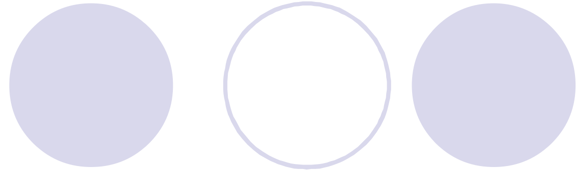
② 大于：待查记录在区间的前半段(区间长度为F(j-1)-1)，修改上界指针： High=Mid-1，转(1)；

③ 小于：待查记录在区间的后半段(区间长度为F(j-2)-1)，修改下界指针：Low=Mid+1，转(1)；

直到越界(Low>High)，查找失败。

2 算法实现

在算法实现时，为了避免频繁计算Fibonacci数，可用两个变量f1和f2保存当前相邻的两个Fibonacci数，这样在以后的计算中可以依次递推计算出。



```
int fib(int n)
  {   int i, f , f0=0 , f1=1 ;
      if (n==0)  return(0) ;
      if (n==1)  return(1) ;
      for (i=2 ; i<=n ; i++ )
         {   f=f0+f1 ; f0=f1 ; f1=f ;   }
      return(f) ;
  }
int Fib_search(RecType ST[] , KeyType key , int n)
  /*  在有序表ST中用Fibonacci方法查找关键字为key的记录  */
  {   int Low=1, High, Mid, f1, f2 ;
      High=n ; f1=fib(n-1) ; f2=fib(n-2) ;
      while (Low<=High)
         {   Mid=Low+f1-1;
```

```
        if ( EQ(ST.[Mid].key, key) )   return(Mid) ;
        else if ( LT(key, ST.[Mid].key) )
            {   High=Mid-1 ; f2=f1-f2 ; f1=f1-f2 ;    }
            else
              {   Low=Mid+1 ;f1=f1-f2 ; f2=f2-f1 ; }
      }
    return(0) ;
  }
```

由算法知，**Fibonacci**查找在最坏情况下性能比折半查找差，但折半查找要求记录按关键字有序；**Fibonacci**查找的优点是分割时只需进行加、减运算。

查找方法比较

	顺序查找	折半查找	分块查找
ASL	最大	最小	两者之间
表结构	有序表、无序表	有序表	分块有序表
存储结构	顺序存储结构 线性链表	顺序存储结构	顺序存储结构 线性链表

9.3 动态查找

当查找表以线性表的形式组织时，若对查找表进行插入、删除或排序操作，就必须移动大量的记录，当记录数很多时，这种移动的代价很大。

利用树的形式组织查找表，可以对查找表进行动态高效的查找。

9.3.1 二叉排序树(BST)的定义

二叉排序树(**Binary Sort Tree**或**Binary Search Tree**) 的定义为：二叉排序树或者是空树，或者是满足下列性质的二叉树。

(1)：若左子树不为空，则左子树上所有结点的值(关键字)都小于根结点的值；

(2)：若右子树不为空，则右子树上所有结点的值(关键字)都大于根结点的值；

(3)：左、右子树都分别是二叉排序树。

结论：若按中序遍历一棵二叉排序树，所得到的结点序列是一个递增序列。

BST仍然可以用二叉链表来存储，如图**9-4**所示。

结点类型定义如下：

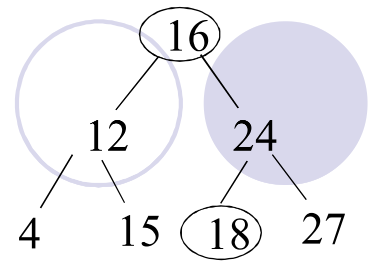


图9-4 二叉排序树

```
typedef struct Node
  { KeyType key ;   /* 关键字域 */
     …      /* 其它数据域 */
     struct Node *Lchild , *Rchild ;
  }BSTNode ;
```

9.3.2 BST树的查找

1 查找思想

首先将给定的K值与二叉排序树的根结点的关键字进行比较：若相等：则查找成功；

① 给定的K值小于BST的根结点的关键字：继续在该结点的左子树上进行查找；

② 给定的K值大于BST的根结点的关键字：继续在该结点的右子树上进行查找。

2 算法实现

(1) 递归算法

```
BSTNode *BST_Serach(BSTNode *T , KeyType
key)
  { if (T==NULL) return(NULL) ;
     else
        { if  (EQ(T->key, key) )  return(T) ;
           else if ( LT(key, T->key) )
                     return(BST_Serach(T->Lchild, key))
           ;
                 else  return(BST_Serach(T->Rchild,
           key)) ;
        }
  }
```

(2) 非递归算法

```
BSTNode *BST_Serach(BSTNode *T , KeyType
key)
  { BSTNode p=T ;
     while (p!=NULL&& !EQ(p->key, key) )
       { if ( LT(key, p->key) ) p=p->Lchild ;
          else p=p->Rchild ;
       }
     if (EQ(p->key, key) ) return(p) ;
     else return(NULL) ;
  }
```

在随机情况下，二叉排序树的平均查找长度ASL和log(n)(树的深度)是等数量级的。

9.3.3 BST树的插入

在BST树中插入一个新结点，要保证插入后仍满足BST的性质。

1 插入思想

在BST树中插入一个新结点x时，若BST树为空，则令新结点x为插入后BST树的根结点；否则，将结点x的关键字与根结点T的关键字进行比较：

① 若相等： 不需要插入；

② 若x.key<T->key：结点x插入到T的左子树中；

③ 若x.key>T->key：结点x插入到T的右子树中。

2 算法实现

(1) 递归算法

```
void  Insert_BST (BSTNode *T , KeyType  key)
  {  BSTNode *x ;
     x=(BSTNode *)malloc(sizeof(BSTNode)) ;
     X->key=key; x->Lchild=x->Rchild=NULL ;
     if (T==NULL)  T=x ;
     else
       {  if (EQ(T->key, x->key) ) return  ;/*  已有结点  */
          else if (LT(x->key, T->key) )
                  Insert_BST(T->Lchild, key) ;
              else   Insert_BST(T->Rchild, key) ;   }
  }
```

(2) 非递归算法

```
void Insert_BST (BSTNode *T , KeyType key)
  {  BSTNode *x, *p , *q ;
     x=(BSTNode *)malloc(sizeof(BSTNode)) ;
     X->key=key; x->Lchild=x->Rchild=NULL ;
     if (T==NULL)  T=x ;
     else
        {  p=T ;
           while (p!=NULL)
              {  if  (EQ(p->key, x->key) )  return  ;
                 q=p ;    /*q作为p的父结点  */
```

```
            if (LT(x->key, p->key) )  p=p->Lchild ;
             else p=p->Rchild ;
           }
        if (LT(x->key, q->key) )  q->Lchild=x ;
        else q->Rchild=x ;
    }
}
```

由结论知，对于一个无序序列可以通过构造一棵**BST**树而变成一个有序序列。

由算法知，每次插入的新结点都是**BST**树的叶子结点，即在插入时不必移动其它结点，仅需修改某个结点的指针。

利用**BST**树的插入操作，可以从空树开始逐个插入每个结点，从而建立一棵**BST**树，算法如下：

```
#define ENDKEY  65535
BSTNode *create_BST()
  {   KeyType  key ;
      BSTNode *T=NULL ;
      scanf("%d", &key) ;
      while (key!=ENDKEY)
         {   Insert_BST(T, key) ;
             scanf("%d", &key) ;
         }
      return(T) ;
  }
```

9.3.4 BST树的删除

1 删除操作过程分析

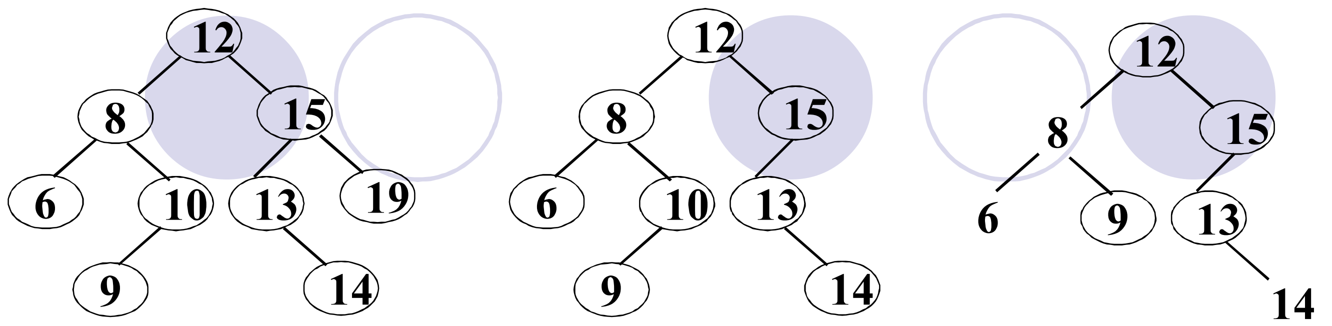
从BST树上删除一个结点，仍然要保证删除后满足BST的性质。设被删除结点为p，其父结点为f，删除情况如下：

① 若p是叶子结点： 直接删除p，如图9-5(b)所示。

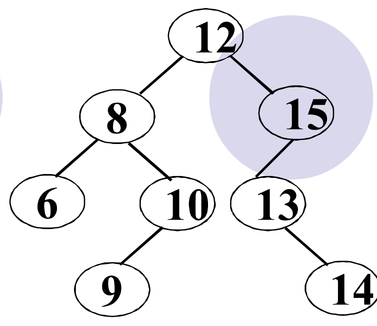
② 若p只有一棵子树(左子树或右子树)：直接用p的左子树(或右子树)取代p的位置而成为f的一棵子树。即原来p是f的左子树，则p的子树成为f的左子树；原来p是f的右子树，则p的子树成为f的右子树，如图9-5(c)、 (d)所示。

③ 若p既有左子树又有右子树：处理方法有以下两种，可以任选其中一种。

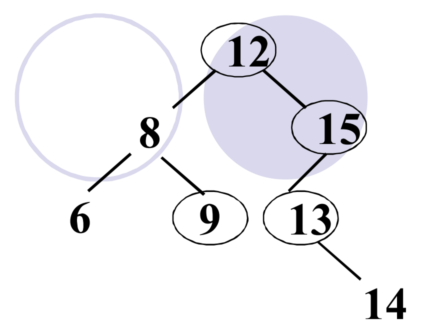
- ◆ 用p的直接前驱结点代替p。即从p的左子树中选择值最大的结点s放在p的位置(用结点s的内容替换结点p内容)，然后删除结点s。s是p的左子树中的最右边的结点且没有右子树，对s的删除同②，如图9-5(e)所示。
- ◆ 用p的直接后继结点代替p。即从p的右子树中选择值最小的结点s放在p的位置(用结点s的内容替换结点p内容)，然后删除结点s。s是p的右子树中的最左边的结点且没有左子树，对s的删除同②，如图9-5(f)所示。



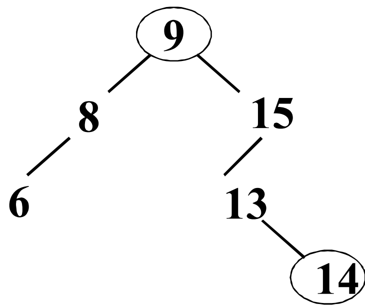
(a) BST树



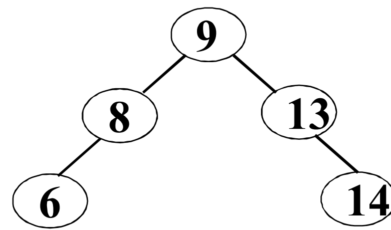
(b) 删除结点19



(c) 删除结点10



(e) 删除结点12



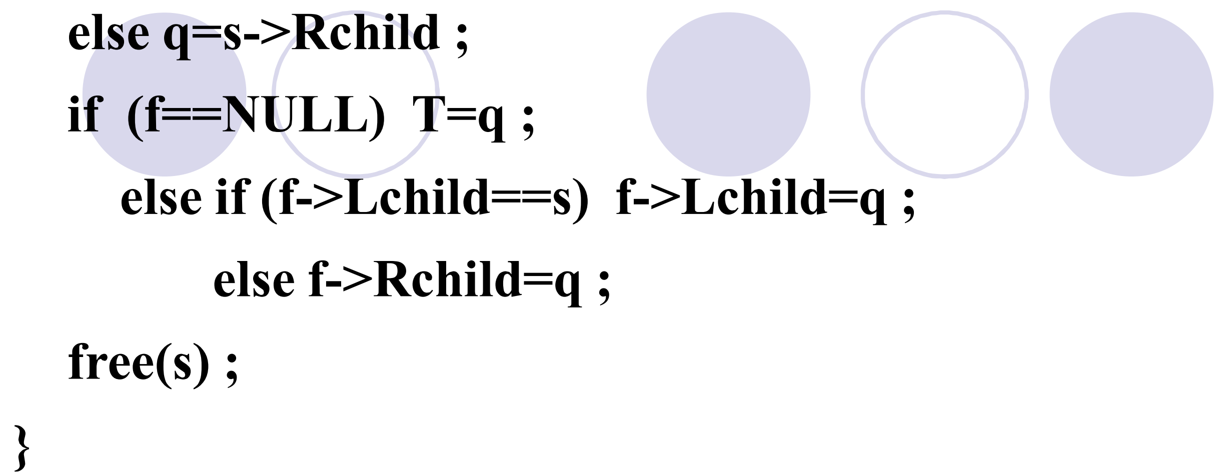
(d) 删除结点15

图9-5 BST树的结点删除情况

2 算法实现

```
void Delete_BST (BSTNode *T , KeyType  key )
   /*  在以T为根结点的BST树中删除关键字为key的结点   */
   {  BSTNode *p=T , *f=NULL , *q , *s ;
      while ( p!=NULL&&!EQ(p->key, key) )
         {  f=p ;
            if (LT(key, p->key) ) p=p->Lchild ;  /*  搜索左子树 */
            else p=p->Rchild ;  /*  搜索右子树 */
         }
      if  (p==NULL)  return ;    /*  没有要删除的结点   */
```

```
s=p ;    /* 找到了要删除的结点为p  */
if (p->Lchild!=NULL&& p->Rchild!=NULL)
  { f=p ; s=p->Lchild ;       /* 从左子树开始找  */
    while (s->Rchild!=NULL)
      {   f=s ; s=s->Rchild ;  }
    /*  左、右子树都不空，找左子树中最右边的结点 */
    p->key=s->key ; p->otherinfo=s->otherinfo ;
         /* 用结点s的内容替换结点p内容 */
  }  /* 将第3种情况转换为第2种情况*/
if  (s->Lchild!=NULL)  /* 若s有左子树，右子树为空 */
  q=s->Lchild ;
```



```
    else q=s->Rchild ;
    if  (f==NULL)  T=q ;
      else if (f->Lchild==s)  f->Lchild=q ;
           else f->Rchild=q ;
    free(s) ;
}
```

9.4 平衡二叉树(AVL)

BST是一种查找效率比较高的组织形式，但其平均查找长度受树的形态影响较大，形态比较均匀时查找效率很好，形态明显偏向某一方向时其效率就大大降低。因此，希望有更好的二叉排序树，其形态总是均衡的，查找时能得到最好的效率，这就是平衡二叉排序树。

平衡二叉排序树(**Balanced Binary Tree**或**Height-Balanced Tree**)是在**1962**年由**Adelson-Velskii**和**Landis**提出的，又称**AVL**树。

9.4.1 平衡二叉树的定义

平衡二叉树或者是空树，或者是满足下列性质的二叉树。

⑴：左子树和右子树深度之差的绝对值不大于1；

⑵：左子树和右子树也都是平衡二叉树。

平衡因子(Balance Factor)：二叉树上结点的左子树的深度减去其右子树深度称为该结点的平衡因子。

因此，平衡二叉树上每个结点的平衡因子只可能是-1、0和1，否则，只要有一个结点的平衡因子的绝对值大于1， 该二叉树就不是平衡二叉树。

如果一棵二叉树既是二叉排序树又是平衡二叉树，称为平衡二叉排序树(Balanced Binary Sort Tree)。

结点类型定义如下：

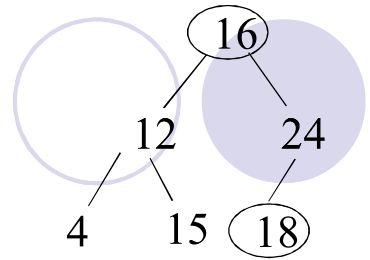


图9-6 平衡二叉树

```
typedef struct BNode
  { KeyType key ;    /* 关键字域 */
    int Bfactor ;    /* 平衡因子域 */
    …     /* 其它数据域 */
    struct BNode *Lchild , *Rchild ;
  }BSTNode ;
```

在平衡二叉排序树上执行查找的过程与二叉排序树上的查找过程完全一样，则在**AVL**树上执行查找时，和给定的**K**值比较的次数不超过树的深度。

设深度为**h**的平衡二叉排序树所具有的最少结点数为N_h，则由平衡二叉排序树的性质知：

$N_0=0$，$N_1=1$，$N_2=2$，…，$N_h= N_{h-1}+N_{h-2}$

该关系和**Fibonacci**数列相似。根据归纳法可证明，当**h≥0**时，$N_h=F_{h+2}-1$，…而

$$F_h \approx \frac{\phi^h}{\sqrt{5}} \quad 其中\ \phi=\frac{1+\sqrt{5}}{2} \qquad 则N_h \approx \frac{\phi^h}{\sqrt{5}}-1$$

这样，含有**n**个结点的平衡二叉排序树的最大深度为

$$h \approx \log_{\phi}(\sqrt{5}\times(n+1))-2$$

则在平衡二叉排序树上进行查找的平均查找长度和$\log_2 n$是一个数量级的，平均时间复杂度为$O(\log_2 n)$。

9.4.2 平衡化旋转

一般的二叉排序树是不平衡的，若能通过某种方法使其既保持有序性，又具有平衡性，就找到了构造平衡二叉排序树的方法，该方法称为平衡化旋转。

在对**AVL**树进行插入或删除一个结点后，通常会影响到从根结点到插入(或删除)结点的路径上的某些结点，这些结点的子树可能发生变化。以插入结点为例，影响有以下几种可能性

- 以某些结点为根的子树的深度发生了变化；
- 某些结点的平衡因子发生了变化；
- 某些结点失去平衡。

沿着插入结点上行到根结点就能找到某些结点，这些结点的平衡因子和子树深度都会发生变化，这样的结点称为失衡结点。

1 LL型平衡化旋转

(1) 失衡原因

在结点**a**的左孩子的左子树上进行插入，插入使结点**a**失去平衡。**a**插入前的平衡因子是**1**，插入后的平衡因子是**2**。设**b**是**a**的左孩子，**b**在插入前的平衡因子只能是**0**，插入后的平衡因子是**1(**否则**b**就是失衡结点**)**。

(2) 平衡化旋转方法

通过顺时针旋转操作实现，如图**9-7**所示。

用b取代a的位置，a成为b的右子树的根结点，b原来的右子树作为a的左子树。

(3) 插入后各结点的平衡因子分析

① 旋转前的平衡因子

设插入后b的左子树的深度为H_{bL}，则其右子树的深度为$H_{bL}-1$； a的左子树的深度为$H_{bL}+1$。

a的平衡因子为2，则a的右子树的深度为：

$H_{aR}=H_{bL}+1-2=H_{bL}-1$。

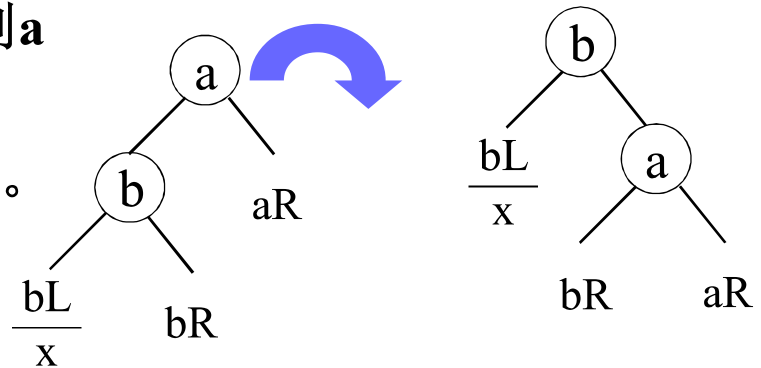


图9-7 LL型平衡化旋转示意图

② 旋转后的平衡因子

a的右子树没有变，而左子树是**b**的右子树，则平衡因子是：$H_{aL}-H_{aR}=(H_{bL}-1)-(H_{bL}-1)=0$

即**a**是平衡的，以**a**为根的子树的深度是H_{bL}。

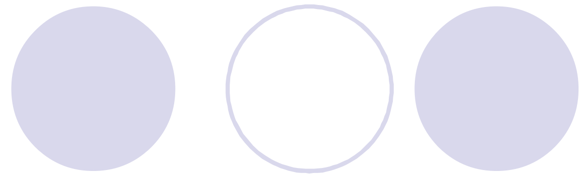
b的左子树没有变化，右子树是以**a**为根的子树，则平衡因子是： $H_{bL}-H_{bL}=0$

即**b**也是平衡的，以**b**为根的子树的深度是$H_{bL}+1$，与插入前**a**的子树的深度相同，则该子树的上层各结点的平衡因子没有变化，即整棵树旋转后是平衡的。

(4) 旋转算法

```
void  LL_rotate(BBSTNode *a)
  {  BBSTNode *b ;
     b=a->Lchild ; a->Lchild=b->Rchild ;
     b->Rchild=a ;
     a->Bfactor=b->Bfactor=0 ; a=b ;
  }
```

2 LR型平衡化旋转



(1) 失衡原因

在结点a的左孩子的右子树上进行插入，插入使结点a失去平衡。a插入前的平衡因子是1，插入后a的平衡因子是2。设b是a的左孩子，c为b的右孩子， b在插入前的平衡因子只能是0，插入后的平衡因子是-1；c在插入前的平衡因子只能是0，否则，c就是失衡结点。

(2) 插入后结点c的平衡因子的变化分析

① 插入后c的平衡因子是1：即在c的左子树上插入。设c的左子树的深度为H_{cL}，则右子树的深度为$H_{cL}-1$；b插入后的平衡因子是-1，则b的左子树的深度为H_{cL}，以b为根的子树的深度是$H_{cL}+2$。

因插入后a的平衡因子是2，则a的右子树的深度是H_{cL}。

②插入后c的平衡因子是0：c本身是插入结点。设c的左子树的深度为H_{cL}，则右子树的深度也是H_{cL}；因b插入后的平衡因子是-1，则b的左子树的深度为H_{cL}，以b为根的子树的深度是$H_{cL}+2$；插入后a的平衡因子是2，则a的右子树的深度是H_{cL}。

③插入后c的平衡因子是-1：即在c的右子树上插入。设c的左子树的深度为H_{cL}，则右子树的深度为$H_{cL}+1$，以c为根的子树的深度是$H_{cL}+2$；因b插入后的平衡因子是-1，则b的左子树的深度为$H_{cL}+1$，以b为根的子树的深度是$H_{cL}+3$；则a的右子树的深度是$H_{cL}+1$。

(3) 平衡化旋转方法

先以b进行一次逆时针旋转(将以b为根的子树旋转为以c为根)，再以a进行一次顺时针旋转，如图9-8所示。将整棵子树旋转为以c为根，b是c的左子树，a是c的右子树；c的右子树移到a的左子树位置， c的左子树移到b的右子树位置。

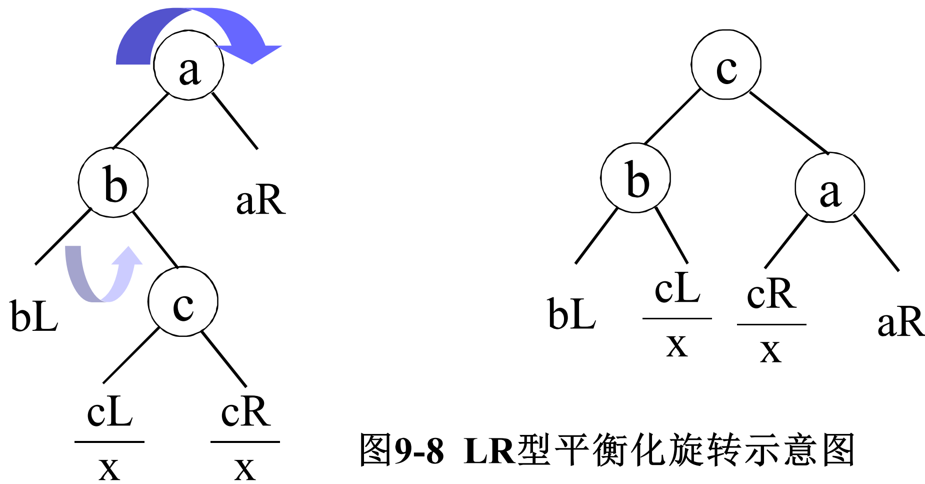


图9-8 LR型平衡化旋转示意图

(4) 旋转后各结点(a,b,c)平衡因子分析

① 旋转前 (插入后)c的平衡因子是1：

a的左子树深度为H_{cL}-1，其右子树没有变化，深度是H_{cL}，则a的平衡因子是-1；b的左子树没有变化，深度为H_{cL}，右子树是c旋转前的左子树，深度为H_{cL}，则b的平衡因子是0； c的左、右子树分别是以b和a为根的子树，则c的平衡因子是0 。

② 旋转前 (插入后)c的平衡因子是0：

旋转后a，b，c的平衡因子都是0 。

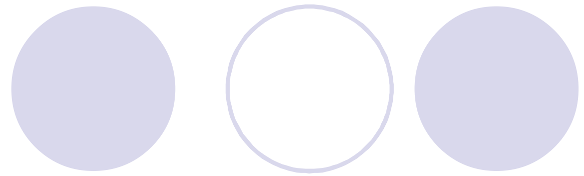
③ 旋转前 (插入后)c的平衡因子是-1：

旋转后a，b，c的平衡因子分别是0，-1，0 。

综上所述，即整棵树旋转后是平衡的。

(5) 旋转算法

```
void  LR_rotate(BBSTNode *a)
  {  BBSTNode *b,*c  ;
     b=a->Lchild ; c=b->Rchild ;      /*  初始化  */
     a->Lchild=c->Rchild ;  b->Rchild=c->Lchild ;
     c->Lchild=b ;  c->Rchild=a ;
     if (c->Bfactor==1)
       {   a->Bfactor=-1 ;b->Bfactor=0 ;   }
     else if (c->Bfactor==0)  a->Bfactor=b->Bfactor=0 ;
          else {   a->Bfactor=0 ;b->Bfactor=1 ;   }
  }
```



3 RL型平衡化旋转

(1) 失衡原因

在结点a的右孩子的左子树上进行插入，插入使结点a失去平衡，与LR型正好对称。对于结点a，插入前的平衡因子是-1，插入后a的平衡因子是-2。设b是a的右孩子，c为b的左孩子， b在插入前的平衡因子只能是0，插入后的平衡因子是1；同样，c在插入前的平衡因子只能是0，否则，c就是失衡结点。

(2) 插入后结点c的平衡因子的变化分析

① 插入后c的平衡因子是1：在c的左子树上插入。设c的左子树的深度为H_{cL}，则右子树的深度为$H_{cL}-1$。

因b插入后的平衡因子是1，则其右子树的深度为H_{cL}，以b为根的子树的深度是$H_{cL}+2$；因插入后a的平衡因子是-2，则a的左子树的深度是H_{cL}。

② 插入后c的平衡因子是0：c本身是插入结点。设c的左子树的深度为H_{cL}，则右子树的深度也是H_{cL}；因b插入后的平衡因子是1，则b的右子树的深度为H_{cL}，以b为根的子树的深度是$H_{cL}+2$；因插入后a的平衡因子是-2，则a的左子树的深度是H_{cL}。

③ 插入后c的平衡因子是-1：在c的右子树上插入。设c的左子树的深度为H_{cL}，则右子树的深度为$H_{cL}+1$，以c为根的子树的深度是$H_{cL}+2$；因b插入后的平衡因子是1，则b的右子树的深度为$H_{cL}+1$，以b为根的子树的深度是$H_{cL}+3$；则a的右子树的深度是$H_{cL}+1$。

(3) 平衡化旋转方法

先以b进行一次顺时针旋转，再以a进行一次逆时针旋转，如图9-9所示。即将整棵子树(以a为根)旋转为以c为根，a是c的左子树，b是c的右子树；c的右子树移到b的左子树位置，c的左子树移到a的右子树位置。

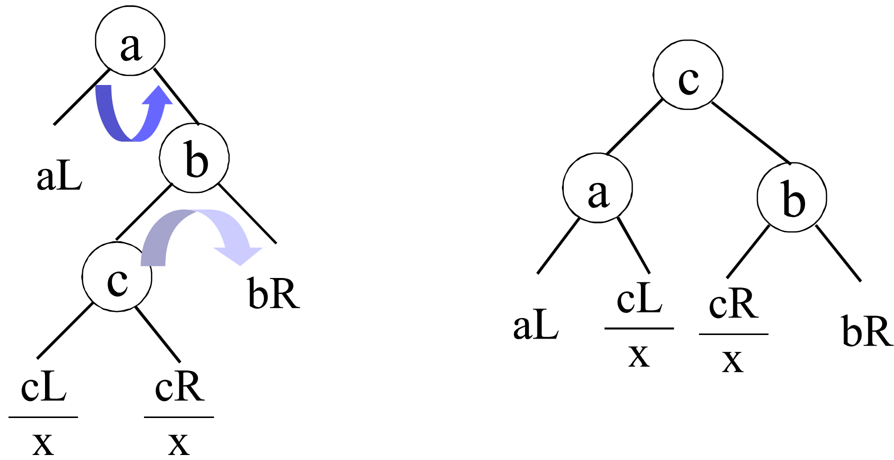


图9-9 RL型平衡化旋转示意图

(4) 旋转后各结点(a, b, c)的平衡因子分析

① 旋转前 (插入后)c的平衡因子是1：

a的左子树没有变化，深度是H_{cL}，右子树是c旋转前的左子树，深度为H_{cL}，则a的平衡因子是0；b的右子树没有变化，深度为H_{cL}，左子树是c旋转前的右子树，深度为$H_{cL}-1$，则b的平衡因子是-1； c的左、右子树分别是以a 和b为根的子树，则c的平衡因子是0 。

② 旋转前 (插入后)c的平衡因子是0：

旋转后a，b，c的平衡因子都是0 。

③ 旋转前 (插入后)c的平衡因子是-1：

旋转后a，b，c的平衡因子分别是1，0，0 。

综上所述，即整棵树旋转后是平衡的。

(5) 旋转算法

```
Void  LR_rotate(BBSTNode *a)
  {  BBSTNode *b,*c  ;
     b=a->Rchild ; c=b->Lchild ;      /*  初始化  */
     a->Rchild=c->Lchild ;  b->Lchild=c->Rchild ;
     c->Lchild=a ;  c->Rchild=b ;
     if (c->Bfactor==1)
       {   a->Bfactor=0 ; b->Bfactor=-1 ;   }
     else if (c->Bfactor==0)  a->Bfactor=b->Bfactor=0 ;
          else {   a->Bfactor=1 ;b->Bfactor=0 ;   }
  }
```

4 RR型平衡化旋转

(1) 失衡原因

在结点a的右孩子的右子树上进行插入，插入使结点a失去平衡。要进行一次逆时针旋转，和LL型平衡化旋转正好对称。

(2) 平衡化旋转方法

设b是a的右孩子，通过逆时针旋转实现，如图9-10所示。用b取代a的位置，a作为b的左子树的根结点，b原来的左子树作为a的右子树。

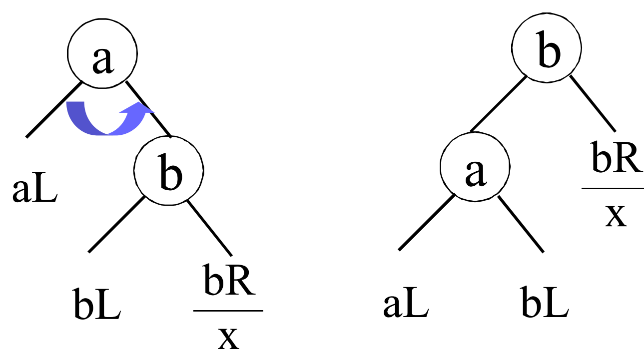


图9-10 RR型平衡化旋转示意图

(3) 旋转算法

```
BBSTNode  *RR_rotate(BBSTNode *a)
  { BBSTNode *b ;
      b=a->Rchild ; a->Rchild=b->Lchild ; b->Lchild=a ;
      a->Bfactor=b->Bfactor=0 ; a=b ;
  }
```

对于上述四种平衡化旋转，其正确性容易由“遍历所得中序序列不变”来证明。并且，无论是哪种情况，平衡化旋转处理完成后，形成的新子树仍然是平衡二叉排序树，且其深度和插入前以**a**为根结点的平衡二叉排序树的深度相同。所以，在平衡二叉排序树上因插入结点而失衡，仅需对失衡子树做平衡化旋转处理。

9.4.3 平衡二叉排序树的插入

平衡二叉排序树的插入操作实际上是在二叉排序插入的基础上完成以下工作：

⑴：判别插入结点后的二叉排序树是否产生不平衡?

⑵：找出失去平衡的最小子树；

⑶：判断旋转类型，然后做相应调整。

失衡的最小子树的根结点**a**在插入前的平衡因子不为**0**，且是离插入结点最近的平衡因子不为**0**的结点的。

若**a**失衡，从**a**到插入点的路径上的所有结点的平衡因子都会发生变化，在该路径上还有一个结点的平衡因子不为**0**且该结点插入后没有失衡，其平衡因子只能是由**1**到**0**或由**-1**到**0**，以该结点为根的子树深度不变。该结点的所有祖先结点的平衡因子也不变，更不会失衡。

1 算法思想(插入结点的步骤)

①：按照二叉排序树的定义，将结点s插入；

②：在查找结点s的插入位置的过程中，记录离结点s最近且平衡因子不为0的结点a，若该结点不存在，则结点a指向根结点；

③： 修改结点a到结点s路径上所有结点的；

④：判断是否产生不平衡，若不平衡，则确定旋转类型并做相应调整。

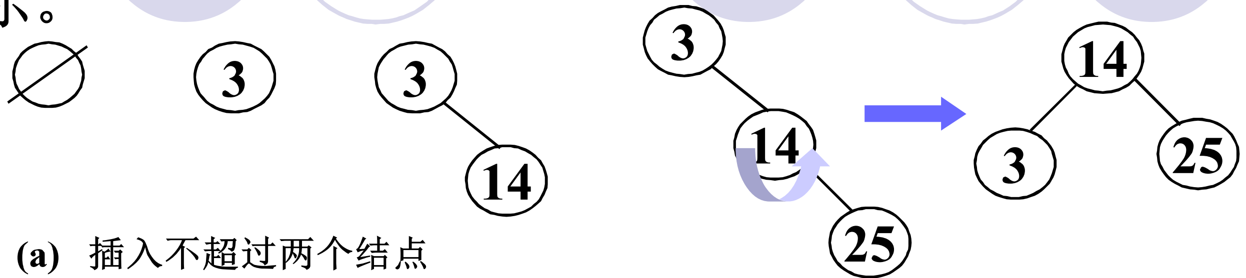
2 算法实现

```
void  Insert_BBST(BBSTNode *T, BBSTNode *S)
  {   BBSTNode *f,*a,*b,*p,*q;
     if  (T==NULL)  {   T=S ; T->Bfactor=1 ; return ;
     }
     a=p=T ;      /* a指向离s最近且平衡因子不为0的结点 */
     f=q=NULL ;        /*   f指向a的父结点,q指向p父结点   */
     while (p!=NULL)
       {  if (EQ(S->key, p->key) )  return ;     /*  结点已
       存在  */
          if (p->Bfactor!=0)  {  a=p ; f=q ;  }
          q=p ;
          if (LT(S->key, p->key) )  p=p->Lchild ;
          else p=p->Rchild ;      /*  在右子树中搜索  */
       }     /* 找插入位置 */
```

```
if (LT(S->key,p->key))  q->Lchild=S ;/*  s为左孩子
*/
else   q->Rchild=S ;      /*  s插入为q的右孩子  */
p=a ;
while (p!=S)
  {   if (LT(S->key, p->key) )
        {   p->Bfactor++ ;  p=p->Lchild ;   }
      else {   p->Bfactor-- ; p=p->Rchild ;  }
  }     /*  插入到左子树,平衡因子加1,插入到左子树,减1  */
if (a->Bfactor>-2&& a->Bfactor<2)
  return ;   /*   未失去平衡,不做调整  */
if (a->Bfactor==2)
  {   b=a->Lchild ;
```

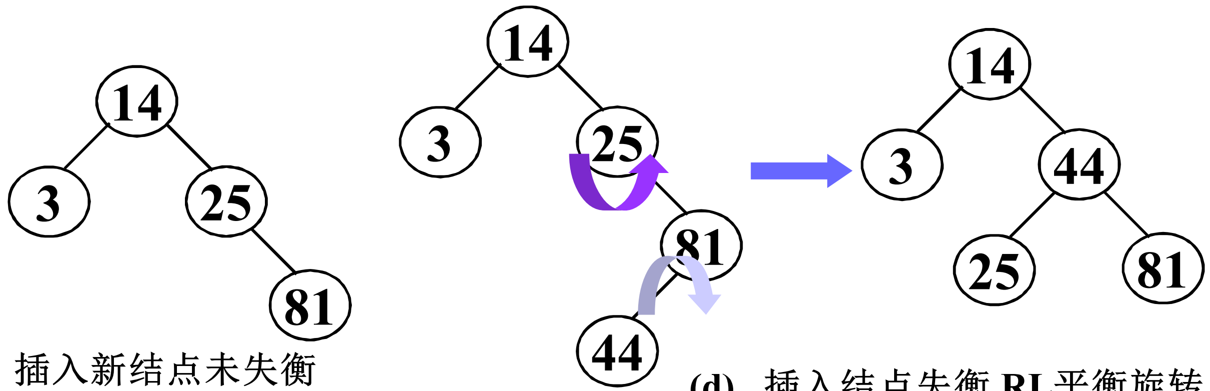
```
        if (b->Bfactor==1)    p=LL_rotate(a) ;
        else p=LR_rotate(a) ;
    }
  else
    {  b=a->Rchild ;
        if (b->Bfactor==1)    p=RL_rotate(a) ;
        else  p=RR_rotate(a) ;
    }   /*   修改双亲结点指针  */
  if (f==NULL)  T=p ;       /*   p为根结点   */
  else    if (f->Lchild==a) f->Lchild=p ;
        else  f->Lchild=p ;
}
```

例： 设要构造的平衡二叉树中各结点的值分别是(3, 14, 25, 81, 44)，平衡二叉树的构造过程如图9-11所示。



(a) 插入不超过两个结点

(b) 插入新结点失衡,RR平衡旋转



(c) 插入新结点未失衡

(d) 插入结点失衡,RL平衡旋转

图9-11 平衡二叉树的构造过程

9.5 索引查找

索引技术是组织大型数据库的重要技术，索引结构的基本组成是索引表和数据表两部分，如图9-12所示。

- 数据表：存储实际的数据记录；
- 索引表：存储记录的关键字和记录(存储)地址之间的对照表，每个元素称为一个索引项。

通过索引表可实现对数据表中记录的快速查找。索引表的组织有线性结构和树形结构两种。

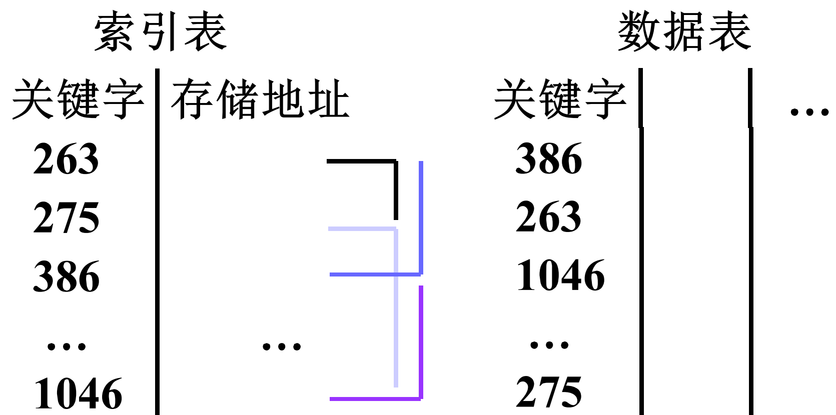


图9-12 索引结构的基本形式

9.5.1 顺序索引表

是将索引项按顺序结构组织的线性索引表，而表中索引项一般是按关键字排序的，其特点是：

优点：

- 可以用折半查找方法快速找到关键字，进而找到数据记录的物理地址，实现数据记录的快速查找；
- 提供对变长数据记录的便捷访问；
- 插入或删除数据记录时不需要移动记录，但需要对索引表进行维护。

缺点：

◆ 索引表中索引项的数目与数据表中记录数相同，当索引表很大时，检索记录需多次访问外存；

◆ 对索引表的维护代价较高，涉及到大量索引项的移动，不适合于插入和删除操作。

9.5.2 树形索引表

平衡二叉排序树便于动态查找，因此用平衡二叉排序树来组织索引表是一种可行的选择。当用于大型数据库时，所有数据及索引都存储在外存，因此，涉及到内、外存之间频繁的数据交换，这种交换速度的快慢成为制约动态查找的瓶颈。若以二叉树的结点作为内、外存之间数据交换单位，则查找给定关键字时对磁盘平均进行$\log_2 n$次访问是不能容忍的，因此，必须选择一种能尽可能降低磁盘I/O次数的索引组织方式。树结点的大小尽可能地接近页的大小。

R.Bayer和E.Mc Creight在1972年提出了一种多路平衡查找树，称为B_树(其变型体是B^+树)。

1 B_树

B_树主要用于文件系统中，在B_树中，每个结点的大小为一个磁盘页，结点中所包含的关键字及其孩子的数目取决于页的大小。一棵度为m的B_树称为m阶B_树，其定义是：

一棵m阶B_树，或者是空树，或者是满足以下性质的m叉树：

⑴ 根结点或者是叶子，或者至少有两棵子树，至多有m棵子树；

⑵ 除根结点外，所有非终端结点至少有$\lceil m/2 \rceil$棵子树，至多有m棵子树；

⑶ 所有叶子结点都在树的同一层上；

⑷ 每个结点应包含如下信息：

$$(n，A_0，K_1，A_1，K_2，A_2，…，K_n，A_n)$$

其中$K_i(1≤i≤n)$是关键字，且$K_i<K_{i+1}$ $(1≤i≤n-1)$；$A_i(i=0，1，…，n)$为指向孩子结点的指针，且A_{i-1}所指向的子树中所有结点的关键字都小于K_i，A_i所指向的子树中所有结点的关键字都大于K_i；n是结点中关键字的个数，且$\lfloor m/2 \rfloor -1≤n≤m-1$，n+1为子树的棵数。

当然，在实际应用中每个结点中还应包含n个指向每个关键字的记录指针，如图9-13是一棵包含13个关键字的4阶B_树。

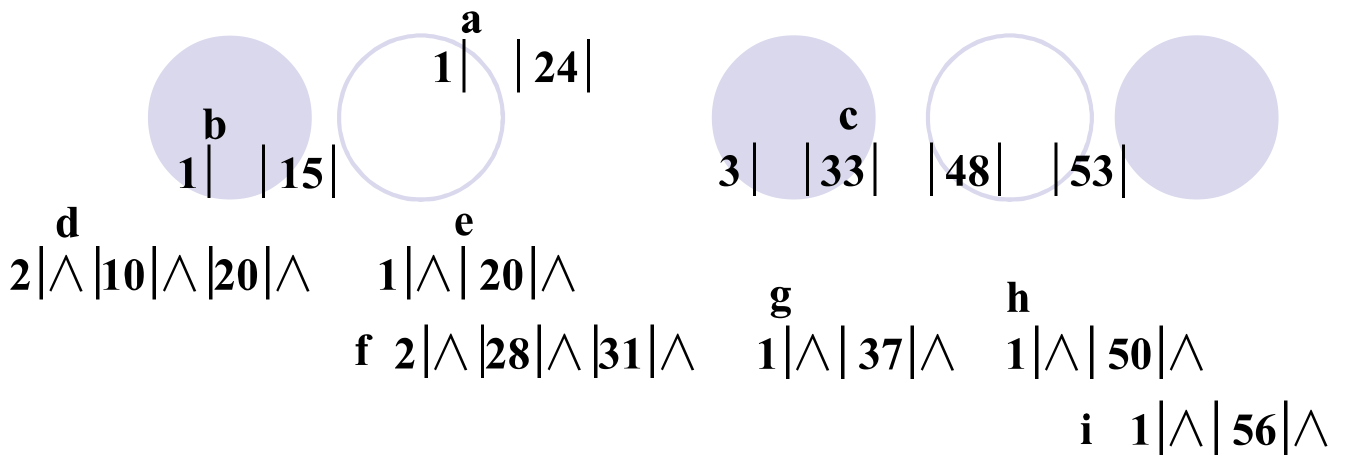


图9-13 一棵包含13个关键字的4阶B_树

根据m阶B_树的定义，结点的类型定义如下：

```
#define M   5     /*  根据实际需要定义B_树的阶数 */
typedef  struct BTNode
  {  int   keynum ;   /*   结点中关键字的个数   */
     struct BTNode  *parent ;    /*   指向父结点的指针   */
     KeyType  key[M+1] ;     /*   关键字向量,key[0]未用   */
     struct BTNode  *ptr[M+1] ; /*   子树指针向量   */
     RecType   *recptr[M+1] ;
       /*   记录指针向量,recptr[0]未用   */
  }BTNode ;
```

2 B_树的查找

由B_树的定义可知，在其上的查找过程和二叉排序树的查找相似。

(1) 算法思想

① 从树的根结点T开始，在T所指向的结点的关键字向量key[1…keynum]中查找给定值K(用折半查找)：

若key[i]=K(1≤i≤keynum)，则查找成功，返回结点及关键字位置；否则，转(2)；

② 将K与向量key[1…keynum]中的各个分量的值进行比较，以选定查找的子树：

◆ 若K<key[1]：T=T->ptr[0]；

◆ 若key[i]<K<key[i+1](i=1, 2, …keynum-1)：
T=T->ptr[i]；

◆ 若K>key[keynum]：T=T->ptr[keynum]；

转①，直到T是叶子结点且未找到相等的关键字，则查找失败。

(2) 算法实现

```
int  BT_search(BTNode *T, KeyType K, BTNode *p)
  /*  在B_树中查找关键字K, 查找成功返回在结点中的位置  */
  /*  及结点指针p; 否则返回0及最后一个结点指针  */
  {  BTNode *q ; int n ;
     p=q=T ;
```

```
    while  (q!=NULL)
      {  p=q ; q->key[0]=K ;      /*  设置查找哨兵  */
         for (n=q->keynum ; K<q->key[n] ; n--)
            if (n>0&&EQ(q->key[n], K) )    return n ;
         q=q->ptr[n] ;
      }
    return 0 ;
  }
```

(3) 算法分析

在B_树上的查找有两中基本操作：

- 在B_树上查找结点(查找算法中没有体现)；

◆ 在结点中查找关键字：在磁盘上找到指针ptr所指向的结点后，将结点信息读入内存后再查找。因此，磁盘上的查找次数(待查找的记录关键字在B_树上的层次数)是决定B_树查找效率的首要因素。

根据m阶B_树的定义，第一层上至少有1个结点，第二层上至少有2个结点；除根结点外，所有非终端结点至少有$\lceil m/2 \rceil$棵子树，…，第h层上至少有$\lceil m/2 \rceil^{h-2}$个结点。在这些结点中：根结点至少包含1个关键字，其它结点至少包含$\lceil m/2 \rceil$-1个关键字，设$s=\lceil m/2 \rceil$，则总的关键字数目n满足：

$$n \geqq 1+(s-1)\sum_{i=2}^{h} 2s^{i} = 2(s-1)\frac{s^{h-1}-1}{s-1} = 2s^{h-1}-1$$

因此有： $h \leqq 1+\log_s((n+1)/2)=1+\log_{\lceil m/2 \rceil}((n+1)/2)$

即在含有n个关键字的B_树上进行查找时，从根结点到待查找记录关键字的结点的路径上所涉及的结点数不超过$1+\log_{\lceil m/2 \rceil}((n+1)/2)$。

3　B_树的插入

B_树的生成也是从空树起，逐个插入关键字。插入时不是每插入一个关键字就添加一个叶子结点，而是首先在最低层的某个叶子结点中添加一个关键字，然后有可能“分裂”。

(1) 插入思想

① 在B_树的中查找关键字K，若找到，表明关键字已存在，返回；否则，K的查找操作失败于某个叶子结点，转 ②；

② 将K插入到该叶子结点中，插入时，若：

- 叶子结点的关键字数<m-1：直接插入；
- 叶子结点的关键字数=m-1：将结点“分裂”。

(2) 结点“分裂”方法

设待“分裂”结点包含信息为：

$(m，A_0，K_1，A_1，K_2，A_2，…，K_m，A_m)$，从其中间位置分为两个结点：

$(\lceil m/2 \rceil-1，A_0，K_1，A_1，…，K_{\lceil m/2 \rceil-1}，A_{\lceil m/2 \rceil-1})$

$(m-\lceil m/2 \rceil，A_{\lceil m/2 \rceil}，K_{\lceil m/2 \rceil+1}，A_{\lceil m/2 \rceil+1}，…，K_m，A_m)$

并将中间关键字$K_{\lceil m/2 \rceil}$插入到p的父结点中，以分裂后的两个结点作为中间关键字$K_{\lceil m/2 \rceil}$的两个子结点。

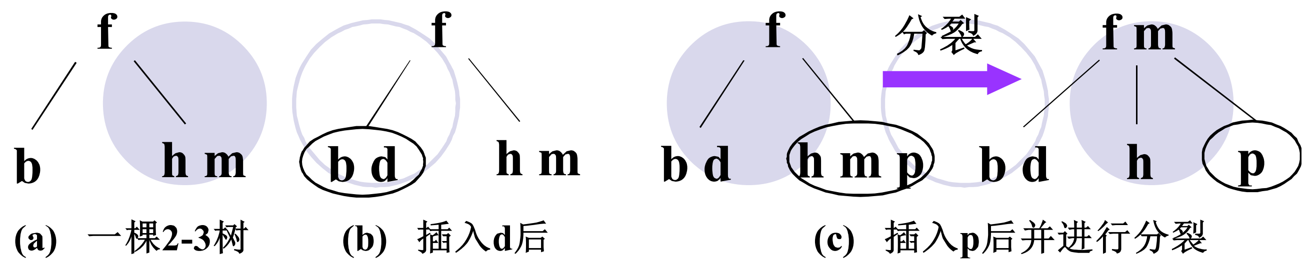
当将中间关键字$K_{\lceil m/2 \rceil}$插入到p的父结点后，父结点也可能不满足m阶B_树的要求(分枝数大于m)，则必须对父结点进行“分裂”，一直进行下去，直到没有父结点或分裂后的父结点满足m阶B_树的要求。

当根结点分裂时，因没有父结点，则建立一个新的根，B_树增高一层。

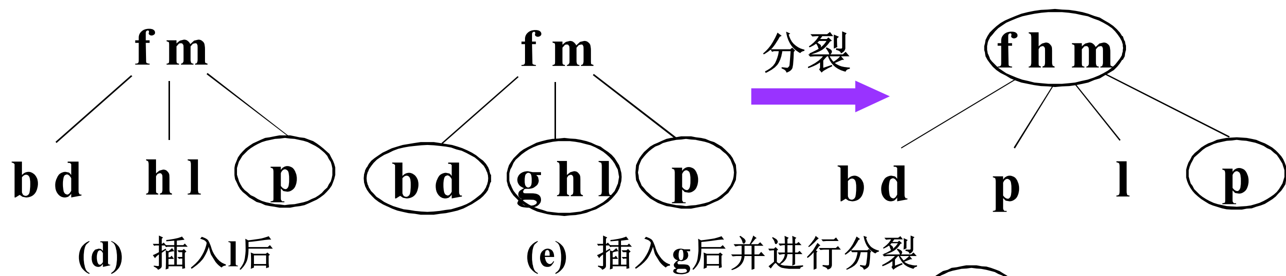
例：在一个3阶B_树(2-3树)上插入结点，其过程如图9-14所示。

(3) 算法实现

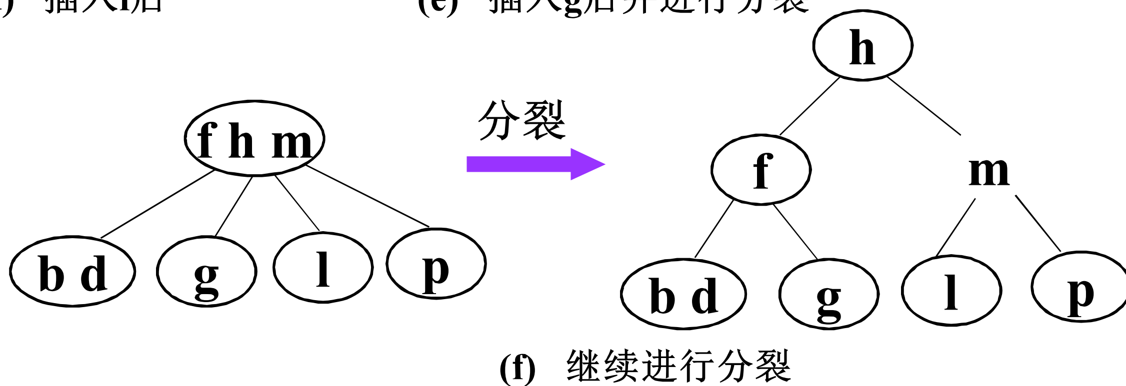
要实现插入，首先必须考虑结点的分裂。设待分裂的结点是p，分裂时先开辟一个新结点，依此将结点p中后半部分的关键字和指针移到新开辟的结点中。分裂之后，而需要插入到父结点中的关键字在p的关键字向量的p->keynum+1位置上。



(a) 一棵2-3树 (b) 插入d后 (c) 插入p后并进行分裂



(d) 插入l后 (e) 插入g后并进行分裂



(f) 继续进行分裂

图9-14 在B_树中进行插入的过程

```
BTNode  *split(BTNode *p)
  /*  结点p中包含m个关键字，从中分裂出一个新的结点  */
  {  BTNode *q ;  int k, mid, j ;
     q=(BTNode *)malloc(sizeof( BTNode)) ;
     mid=(m+1)/2 ;   q->ptr[0]=p->ptr[mid] ;
     for (j=1,k=mid+1; k<=m; k++)
        {  q->key[j]=p->key[k] ;
           q->ptr[j++]=p->ptr[k] ;
        }  /*  将p的后半部分移到新结点q中  */
     q->keynum=m-mid ;  p->keynum=mid-1 ;
     return(q) ;
  }
```

```
void  insert_BTree(BTNode *T, KeyType  K)
   /*  在B_树T中插入关键字K，*/
  {  BTNode *q, *s1=NULL, *s2=NULL ;
     int n ;
     if  (!BT_search(T, K, p))     /*  树中不存在关键字K  */
       {  while (p!=NULL)
          {  p->key[0]=K ;     /*  设置哨兵  */
             for (n=p->keynum ; K<p->key[n] ; n--)
               {  p->key[n+1]=p->key[n] ;
                  p->ptr[n+1]=p->ptr[n] ;
               }   /*  后移关键字和指针  */
             p->key[n]=K ; p->ptr[n-1]=s1 ;
```

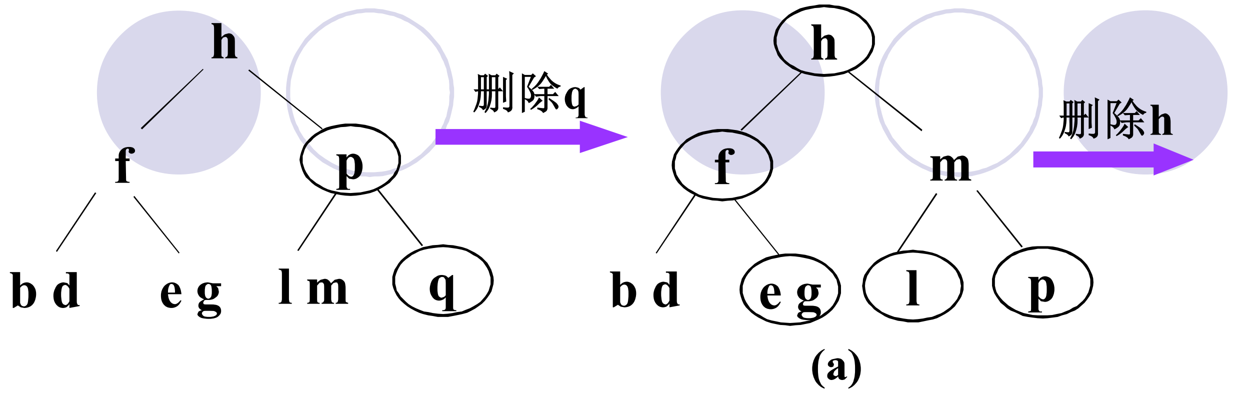
```
        p->ptr[n+1]=s2 ;  /*  置关键字K的左右指针  */
    if (++(p->keynum ))<m  break ;
    else {  s2=split(p) ; s1=p ;  /*  分裂结点p  */
            K=p->key[p->keynum+1] ;
            p=p->parent ;   /*  取出父结点*/
          }
    if (p==NULL)       /*  需要产生新的根结点  */
        {  p=(BTNode *)malloc(sizeof(
BTNode)) ;
            p->keynum=1 ; p->key[1]=K ;
            p->ptr[0]=s1 ; p->ptr[1] =s2 ;
        }
```

利用m阶B_树的插入操作，可从空树起，将一组关键字依次插入到m阶B_树中，从而生成一个m阶B_树。

4 B_树的删除

在B_树上删除一个关键字K，首先找到关键字所在的结点N，然后在N中进行关键字K的删除操作。

若N不是叶子结点，设K是N中的第i个关键字，则将指针A_{i-1}所指子树中的最大关键字(或最小关键字)K'放在(K)的位置，然后删除K'，而K'一定在叶子结点上。如图9-15(b)，删除关键字h，用关键字g代替h的位置，然后再从叶子结点中删除关键字g。



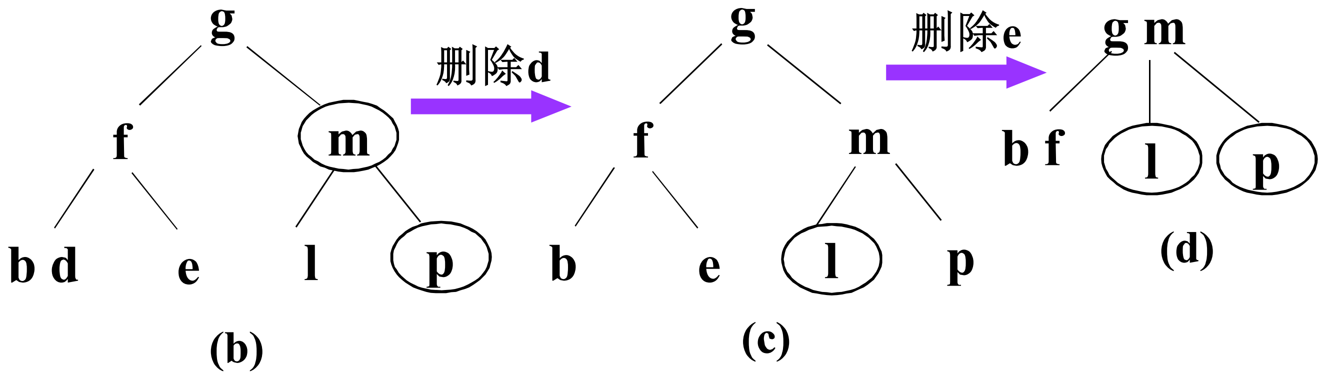


图9-15　在B_树中进行删除的过程

从叶子结点中删除一个关键字的情况是：

⑴ 若结点N中的关键字个数>⌈m/2⌉-1：在结点中直接删除关键字K，如图9-15(b)∽©所示。

⑵ 若结点N中的关键字个数=⌈m/2⌉-1：若结点N的左(右)兄弟结点中的关键字个数>⌈m/2⌉-1，则将结点N的左(或右)兄弟结点中的最大(或最小)关键字上移到其父结点中，而父结点中大于(或小于)且紧靠上移关键字的关键字下移到结点N，如图9-15(a)。

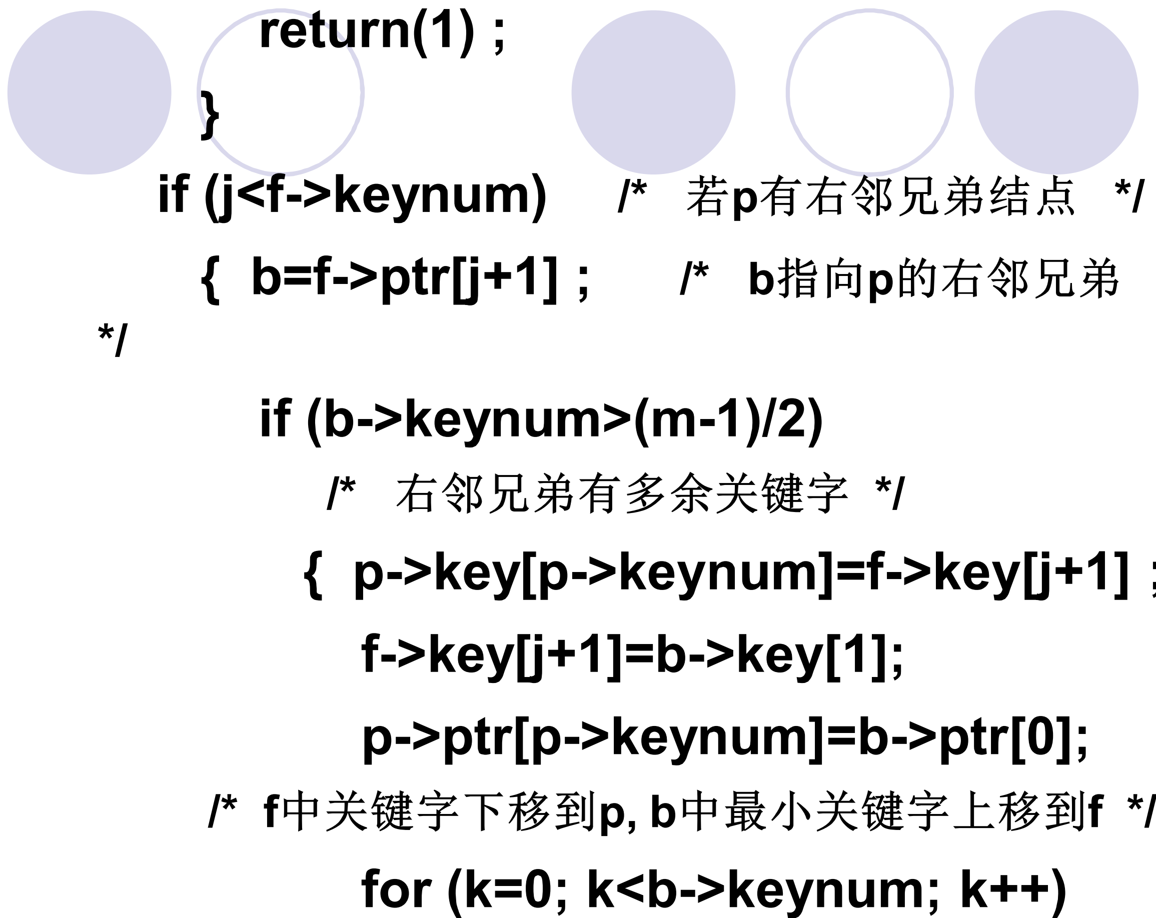
⑶ 若结点N和其兄弟结点中的关键字数=⌈m/2⌉-1：删除结点N中的关键字，再将结点 中的关键字、指针与其兄弟结点以及分割二者的父结点中的某个关键字K_i，合并为一个结点，若因此使父结点中的关键字个数<⌈m/2⌉-1，则依此类推，如图9-15(d)。

算法实现

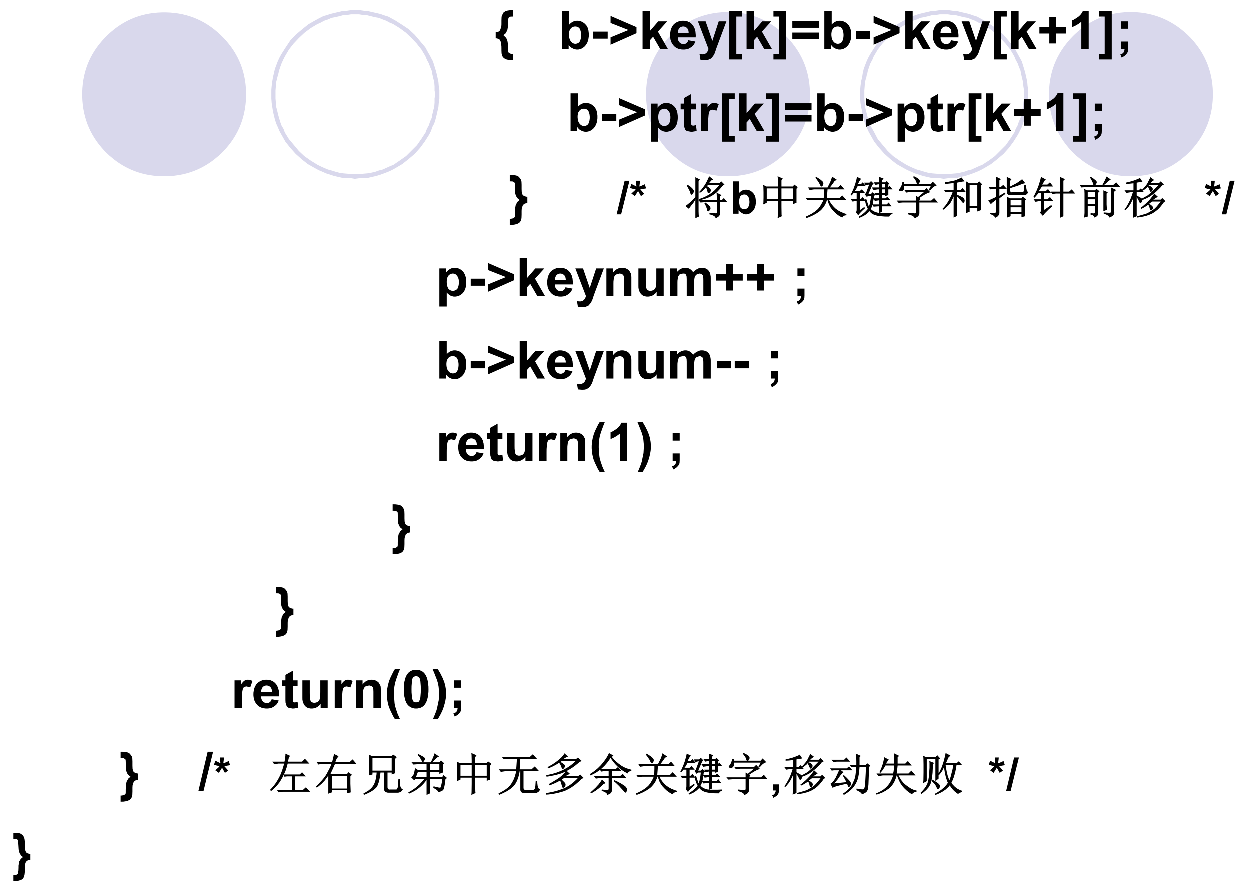
在B_树上删除一个关键字的操作，针对上述的(2)和(3)的情况，相应的算法如下：

```
int  BTNode  MoveKey(BTNode *p)
  /*  将p的左(或右)兄弟结点中的最大(或最小)关键字上移  */
  /*  到其父结点中,父结点中的关键字下移到p中  */
  {  BTNode *b , *f=p->parent ;    /*  f指向p的父结点  */
     int k, j ;
     for (j=0; f->ptr[j]!=p; j++)    /*  在f中找p的位置  */
        if (j>0)        /*  若p有左邻兄弟结点  */
           {  b=f->ptr[j-1] ;     /*  b指向p的左邻兄弟  */
```

```
if (b->keynum>(m-1)/2)
        /*  左邻兄弟有多余关键字  */
    {  for (k=p->keynum; k>=0; k--)
          {   p->key[k+1]=p->key[k];
              p->ptr[k+1]=p->ptr[k];
          }      /*  将p中关键字和指针后移  */
        p->key[1]=f->key[j];
        f->key[j]=b->key[keynum] ;
    /*  f中关键字下移到p, b中最大关键字上移到f  */
        p->ptr[0]= b->ptr[keynum] ;
        p->keynum++ ;
        b->keynum-- ;
```



```
        return(1) ;
    }
  if (j<f->keynum)     /*  若p有右邻兄弟结点  */
    {  b=f->ptr[j+1] ;      /*  b指向p的右邻兄弟
*/
        if (b->keynum>(m-1)/2)
            /*  右邻兄弟有多余关键字  */
          {  p->key[p->keynum]=f->key[j+1] ;
             f->key[j+1]=b->key[1];
             p->ptr[p->keynum]=b->ptr[0];
    /*  f中关键字下移到p, b中最小关键字上移到f  */
             for (k=0; k<b->keynum; k++)
```



```
                    {   b->key[k]=b->key[k+1];
                        b->ptr[k]=b->ptr[k+1];
                    }       /*  将b中关键字和指针前移  */
                p->keynum++ ;
                b->keynum-- ;
                return(1) ;
              }
          }
        return(0);
    }   /*  左右兄弟中无多余关键字,移动失败  */
}
```

```
BTNode  *MergeNode(BTNode *p)
   /*  将p与其左(右)邻兄弟合并,返回合并后的结点指针  */
  {   BTNode *b, f=p->parent ;
      int j, k ;
      for (j=0; f->ptr[j]!=p; j++)   /*  在f中找出p的位置   */
      if (j>0) b=f->ptr[j-1];    /*  b指向p的左邻兄弟   */
      else {  b=p; p=p->ptr[j+1];  }    /*  p指向p的右邻   */
      b->key[++b->keynum]=f->key[j] ;
      b->ptr[p->keynum]=p->ptr[0] ;
      for (k=1; k<=b->keynum ; k++)
        {   b->key[++b->keynum]=p->key[k] ;
            b->ptr[b->keynum]=p->ptr[k] ;
        }     /*  将p中关键字和指针移到b中   */
```

```
    free(p);
    for (k=j+1; k<=f->keynum ; k++)
      {   f->key[k-1]=f->key[k] ;
          f->ptr[k-1]=f->ptr[k] ;
      }     /*   将f中第j个关键字和指针前移   */
    f->keynum-- ;
    return(b) ;
}
```

```
void  DeleteBTNode(BTNode *T, KeyType K)
  {   BTNode  *p, *S ;
      int j,n ;
      m=BT_search(T, K, p) ; /* 在T中查找K的结点  */
      if (j==0)  return(T) ;
      if (p->ptr[j-1])
        {   S=p->ptr[j-1] ;
            while (S->ptr[S->keynum])
                S=S->ptr[S->keynum] ;
             /*  在子树中找包含最大关键字的结点   */
            p->key[j]=S->key[S ->keynum] ;
            p=S ; j=S->keynum ;
        }
```

```
for (n=j+1; n<p->keynum; n++)
    p->key[n-1]=p->key[n] ;
        /*  从p中删除第m个关键字   */
p->keynum-- ;
while (p->keynum<(m-1)/2&&p->parent)
    {   if (!MoveKey(p) )  p=MergeNode(p);
         p=p->parent ;
    }    /*  若p中关键字树目不够,按⑵处理    */
if (p==T&&T->keynum==0)
    {   T=T->ptr[0] ;   free(p) ;    }
}
```

5 B^+树

在实际的文件系统中，基本上不使用B_树，而是使用B_树的一种变体，称为m阶B^+树。 它与B_树的主要不同是叶子结点中存储记录。在B^+树中，所有的非叶子结点可以看成是索引，而其中的关键字是作为“分界关键字”，用来界定某一关键字的记录所在的子树。一棵m阶B^+树与m阶B_树的主要差异是：

⑴ 若一个结点有n棵子树，则必含有n个关键字；

⑵ 所有叶子结点中包含了全部记录的关键字信息以及这些关键字记录的指针，而且叶子结点按关键字的大小从小到大顺序链接；

⑶ 所有的非叶子结点可以看成是索引的部分，结点中只含有其子树的根结点中的最大(或最小)关键字。

如图9-16是一棵3阶B+树。

由于B^+树的叶子结点和非叶子结点结构上的显著区别，因此需要一个标志域加以区分，结点结构定义如下：

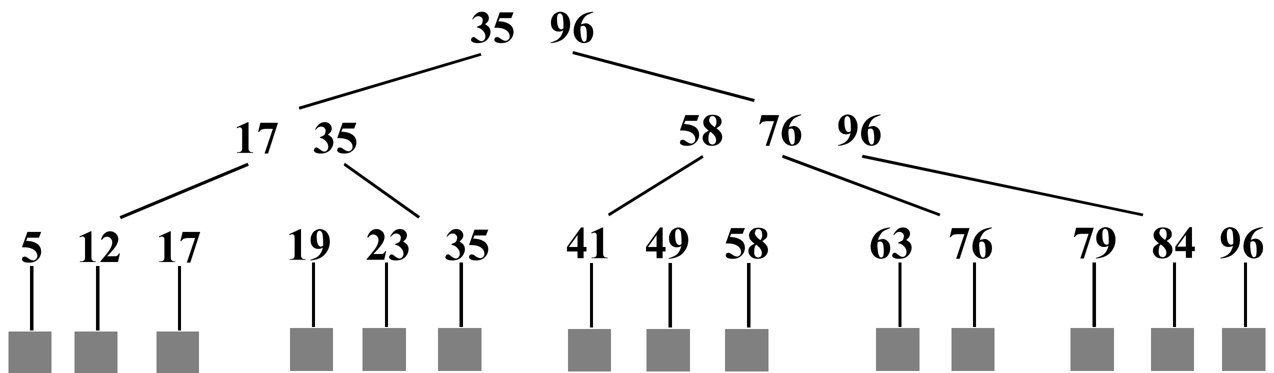


图9-16　一棵3阶B^+树

```
typedef  enum{branch, left}  NodeType；
typedef  struct BPNode
  {   NodeTag   tag；     /*   结点标志   */
      int   keynum；   /*   结点中关键字的个数   */
      struct BTNode  *parent；    /*   指向父结点的指针   */
      KeyType  key[M+1]；      /*   组关键字向量,key[0]未用
      */
      union pointer
        {   struct BTNode  *ptr[M+1]；/*   子树指针向量
        */
            RecType   *recptr[M+1]；  /*   recptr[0]未用   */
        }ptrType；    /*   用联合体定义子树指针和记录指针  */
  }BPNode；
```

与B_树相比，对B^+树不仅可以从根结点开始按关键字随机查找，而且可以从最小关键字起，按叶子结点的链接顺序进行顺序查找。在B^+树上进行随机查找、插入、删除的过程基本上和B_树类似。

在B^+树上进行随机查找时，若非叶子结点的关键字等于给定的K值，并不终止，而是继续向下直到叶子结点(只有叶子结点才存储记录)， 即无论查找成功与否，都走了一条从根结点到叶子结点的路径。

B^+树的插入仅仅在叶子结点上进行。当叶子结点中的关键字个数大于m时，“分裂”为两个结点，两个结点中所含有的关键字个数分别是$\lfloor (m+1)/2 \rfloor$和$\lceil (m+1)/2 \rceil$，且将这两个结点中的最大关键字提升到父结点中，用来替代原结点在父结点中所对应的关键字。提升后父结点又可能会分裂，依次类推。

9.6 哈希(散列)查找

基本思想：在记录的存储地址和它的关键字之间建立一个确定的对应关系；这样，不经过比较，一次存取就能得到所查元素的查找方法。

例 30个地区的各民族人口统计表

编号	省、市(区)	总人口	汉族	回族	……
1	北京				
2	上海				
……	……				

以编号作关键字，
构造哈希函数：H(key)=key
H(1)=1， H(2)=2

以地区别作关键字，取地区名称第一个拼音字母的序号作哈希函数：H(Beijing)=2
H(Shanghai)=19 H(Shenyang)=19

9.6.1 基本概念

哈希函数：在记录的关键字与记录的存储地址之间建立的一种对应关系叫哈希函数。

哈希函数是一种映象，是从关键字空间到存储地址空间的一种映象。可写成：$addr(a_i)=H(k_i)$，其中i是表中一个元素，$addr(a_i)$是a_i的地址，k_i是a_i的关键字。

哈希表：应用哈希函数，由记录的关键字确定记录在表中的地址，并将记录放入此地址，这样构成的表叫哈希表。

哈希查找(又叫散列查找)：利用哈希函数进行查找的过程叫哈希查找。

冲突：对于不同的关键字k_i、k_j，若$k_i \neq k_j$，但$H(k_i)=H(k_j)$的现象叫冲突(**collision**) 。

同义词：具有相同函数值的两个不同的关键字，称为该哈希函数的同义词。

哈希函数通常是一种压缩映象，所以冲突不可避免，只能尽量减少；当冲突发生时，应该有处理冲突的方法。设计一个散列表应包括：

① 散列表的空间范围，即确定散列函数的值域；

② 构造合适的散列函数，使得对于所有可能的元素(记录的关键字)，函数值均在散列表的地址空间范围内，且出现冲突的可能尽量小；

③ 处理冲突的方法。即当冲突出现时如何解决。

9.6.2 哈希函数的构造

哈希函数是一种映象，其设定很灵活，只要使任何关键字的哈希函数值都落在表长允许的范围之内即可。哈希函数“好坏”的主要评价因素有：

- 散列函数的构造简单；
- 能“均匀”地将散列表中的关键字映射到地址空间。所谓“均匀”(uniform)是指发生冲突的可能性尽可能最少。

1 直接定址法

取关键字或关键字的某个线性函数作哈希地址，即 H(key)=key 或 H(key)=a·key+b(a,b为常数)

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

2 数字分析法

对关键字进行分析，取关键字的若干位或组合作为哈希地址。

适用于关键字位数比哈希地址位数大，且可能出现的关键字事先知道的情况。

例： 设有**80**个记录，关键字为**8**位十进制数，哈希地址为**2**位十进制数。

①	②	③	④	⑤	⑥	⑦	⑧
8	1	3	4	6	5	3	2
8	1	3	7	2	2	4	2
8	1	3	8	7	4	2	2
8	1	3	0	1	3	6	7
8	1	3	2	2	8	1	7
8	1	3	3	8	9	6	7
8	1	3	6	8	5	3	7
8	1	4	1	9	3	5	5

分析： ① 只取**8**
② 只取**1**
③ 只取**3**、**4**
⑧ 只取**2**、**7**、**5**
④⑤⑥⑦数字分布近乎随机

所以：取④⑤⑥⑦任意两位或两位与另两位的叠加作哈希地址

3 平方取中法

将关键字平方后取中间几位作为哈希地址。

一个数平方后中间几位和数的每一位都有关，则由随机分布的关键字得到的散列地址也是随机的。散列函数所取的位数由散列表的长度决定。这种方法适于不知道全部关键字情况，是一种较为常用的方法。

4 折叠法

将关键字分割成位数相同的几部分(最后一部分可以不同)，然后取这几部分的叠加和作为哈希地址。

数位叠加有移位叠加和间界叠加两种。

- 移位叠加：将分割后的几部分低位对齐相加。
- 间界叠加：从一端到另一端沿分割界来回折迭，然后对齐相加。

适于关键字位数很多，且每一位上数字分布大致均匀情况。

例：设关键字为**0442205864**，哈希地址位数为**4**。两种不同的地址计算方法如下：

```
  5 8 6 4                      5 8 6 4
  4 2 2 0                      0 2 2 4
      0 4                          0 4
---------    移位叠加        ---------    间界叠加
1 0 0 8 8                      6 0 9 2
H(key)=0088                  H(key)=6092
```

5 除留余数法

取关键字被某个不大于哈希表表长m的数p除后所得余数作哈希地址，即H(key)=key MOD p (p≤m)

是一种简单、常用的哈希函数构造方法。

利用这种方法的关键是p的选取，p选的不好，容易产生同义词。p的选取的分析：

- 选取$p=2^i$(p≤m)：运算便于用移位来实现，但等于将关键字的高位忽略而仅留下低位二进制数。高位不同而低位相同的关键字是同义词。
- 选取p=q×f(q、f都是质因数，p≤m)：则所有含有q或f因子的关键字的散列地址均是q或f的倍数。

- 选取p为素数或p=q×f(q、f是质数且均大于20，p≤m)：常用的选取方法，能减少冲突出现的可能性。

6 随机数法

取关键字的随机函数值作哈希地址，即

H(key)=random(key)

当散列表中关键字长度不等时，该方法比较合适。

选取哈希函数，考虑以下因素

- 计算哈希函数所需时间；
- 关键字的长度；
- 哈希表长度（哈希地址范围）；
- 关键字分布情况；
- 记录的查找频率。

9.6.3 冲突处理的方法

冲突处理：当出现冲突时，为冲突元素找到另一个存储位置。

1 开放定址法

基本方法：当冲突发生时，形成某个探测序列；按此序列逐个探测散列表中的其他地址，直到找到给定的关键字或一个空地址(开放的地址)为止，将发生冲突的记录放到该地址中。散列地址的计算公式是：

$H_i(key)=(H(key)+d_i)$ MOD m，$i=1, 2, \ldots, k(k\leq m-1)$

其中：H(key)：哈希函数；m：散列表长度；

d_i：第i次探测时的增量序列；

$H_i(key)$ ：经第i次探测后得到的散列地址。

(1) 线性探测法

将散列表**T[0 …m-1]**看成循环向量。当发生冲突时，从初次发生冲突的位置依次向后探测其他的地址。

增量序列为：d_i**=1, 2, 3, …, m-1**

设初次发生冲突的地址是**h**，则依次探测**T[h+1]**，**T[h+2]…**，直到**T[m-1]**时又循环到表头，再次探测**T[0]**，**T[1]…**，直到**T[h-1]**。探测过程终止的情况是：

- ◆ 探测到的地址为空：表中没有记录。若是查找则失败；若是插入则将记录写入到该地址；
- ◆ 探测到的地址有给定的关键字：若是查找则成功；若是插入则失败；

◆ 直到T[h]：仍未探测到空地址或给定的关键字，散列表满。

例1： 设散列表长为7，记录关键字组为：15, 14, 28, 26, 56, 23，散列函数：H(key)=key MOD 7，冲突处理采用线性探测法。

解：H(15)=15 MOD 7=1　　H(14)=14 MOD 7=0

H(28)=28 MOD 7=0 冲突　H_1(28)=1 又冲突

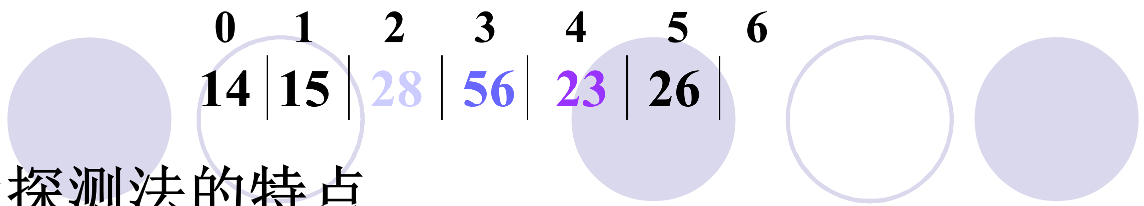
H_2(28)=2　　H(26)=26 MOD 7=5

H(56)=56 MOD 7=0　冲突　H_1(56)=1　又冲突

H_2(56)=2　又冲突　H_3(56)=3

H(23)=23 MOD 7=2　冲突　H_1(23)=3　又冲突

H_3(23)=4



线性探测法的特点

- 优点：只要散列表未满，总能找到一个不冲突的散列地址；
- 缺点：每个产生冲突的记录被散列到离冲突最近的空地址上，从而又增加了更多的冲突机会(这种现象称为冲突的“聚集”)。

(2) 二次探测法

增量序列为：$d_i=1^2,-1^2,2^2,-2^2,3^2,\ldots\ldots\pm k^2\ \ (k\le\lfloor m/2\rfloor)$

上述例题若采用二次探测法进行冲突处理，则：

H(15)=15 MOD 7=1　　　H(14)=14 MOD 7=0

$H(28)=28\ MOD\ 7=0$ 冲突 $H_1(28)=1$ 又冲突

$H_2(28)=4$

$H(26)=26\ MOD\ 7=5$

$H(56)=56\ MOD\ 7=0$ 冲突 $H_1(56)=1$ 又冲突

$H_2(56)=0$ 又冲突 $H_3(56)=4$ 又冲突 $H_4(56)=2$

$H(23)=23\ MOD\ 7=2$ 冲突 $H_1(23)=3$

二次探测法的特点

- 优点：探测序列跳跃式地散列到整个表中，不易产生冲突的“聚集”现象；
- 缺点：不能保证探测到散列表的所有地址。

0	1	2	3	4	5	6
14	15	56	23	28	26	

(3) 伪随机探测法

增量序列使用一个伪随机函数来产生一个落在闭区间[1，m-1]的随机序列。

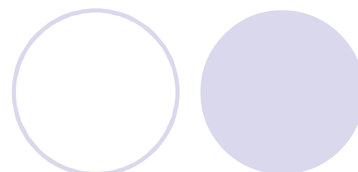
例2： 表长为11的哈希表中已填有关键字为17，60，29的记录，散列函数为H(key)=key MOD 11 。 现有第4个记录，其关键字为38，按三种处理冲突的方法，将它填入表中。

(1) H(38)=38 MOD 11=5 冲突

H1=(5+1) MOD 11=6 冲突

H2=(5+2) MOD 11=7 冲突

H3=(5+3) MOD 11=8 不冲突



(2) $H(38)=38 \text{ MOD } 11=5$　冲突

$H_1=(5+1^2) \text{ MOD } 11=6$　冲突

$H_2=(5-1^2) \text{ MOD } 11=4$　不冲突

(3) $H(38)=38 \text{ MOD } 11=5$　冲突

设伪随机数序列为9，则$H_1=(5+9) \text{ MOD } 11=3$ 不冲突

0	1	2	3	4	5	6	7	8	9	10
			38	38	60	17	29	38		

2 再哈希法

构造若干个哈希函数，当发生冲突时，利用不同的哈希函数再计算下一个新哈希地址，直到不发生冲突为止。即：$H_i=RH_i(key)$　$i=1, 2, \ldots, k$

RH_i：一组不同的哈希函数。第一次发生冲突时，用RH_1计算，第二次发生冲突时，用RH_2计算…依此类推知道得到某个H_i不再冲突为止。

- 优点：不易产生冲突的“聚集”现象；
- 缺点：计算时间增加。

3 链地址法

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

设散列表长为**m**，定义一个一维指针数组：

RecNode *linkhash[m]，其中**RecNode**是结点类型，每个分量的初值为空。凡散列地址为**k**的记录都插入到以**linkhash[k]**为头指针的链表中，插入位置可以在表头或表尾或按关键字排序插入。

例： 已知一组关键字**(19, 14, 23, 1, 68, 20, 84, 27, 55, 11, 10, 79)**，哈希函数为：**H(key)=key MOD 13**，用链地址法处理冲突，如右图图**9-17**所示 。

优点：不易产生冲突的“聚集”；删除记录也很简单。

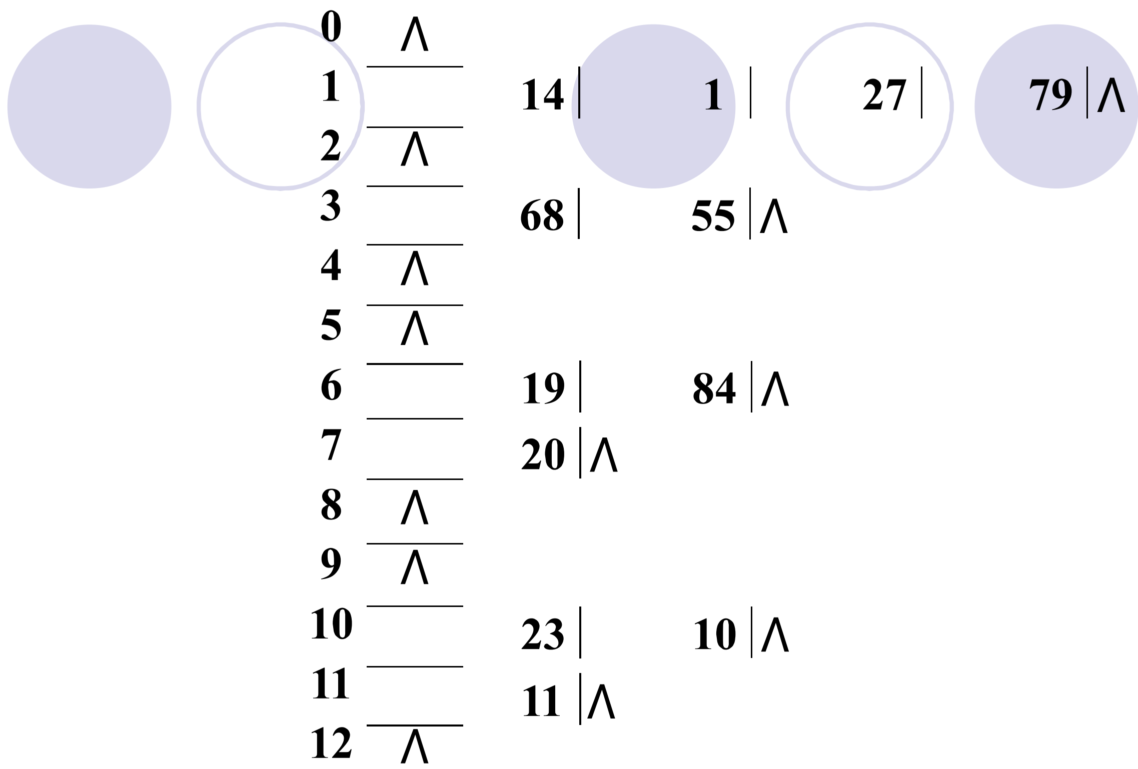


图9-17 用链地址法处理冲突的散列表

4 建立公共溢出区

方法：在基本散列表之外，另外设立一个溢出表保存与基本表中记录冲突的所有记录。

设散列表长为**m**，设立基本散列表**hashtable[m]**，每个分量保存一个记录；溢出表**overtable[m]**，一旦某个记录的散列地址发生冲突，都填入溢出表中。

例： 已知一组关键字**(15, 4, 18, 7, 37, 47)**，散列表长度为**7**，哈希函数为：**H(key)=key MOD 7**，用建立公共溢出区法处理冲突。得到的基本表和溢出表如下：

Hashtable表：

散列地址	0	1	2	3	4	5	6
关键字	7	15	37		4	47	

overtable表：

溢出地址	0	1	2	3	4	5	6
关键字	18						

9.6.4 哈希查找过程及分析

1 哈希查找过程

哈希表的主要目的是用于快速查找，且插入和删除操作都要用到查找。由于散列表的特殊组织形式，其查找有特殊的方法。

设散列为**HT[0…m-1]**，散列函数为**H(key)**，解决冲突的方法为**R(x, i)**，则在散列表上查找定值为**K**的记录的过程如图**9-18**所示。

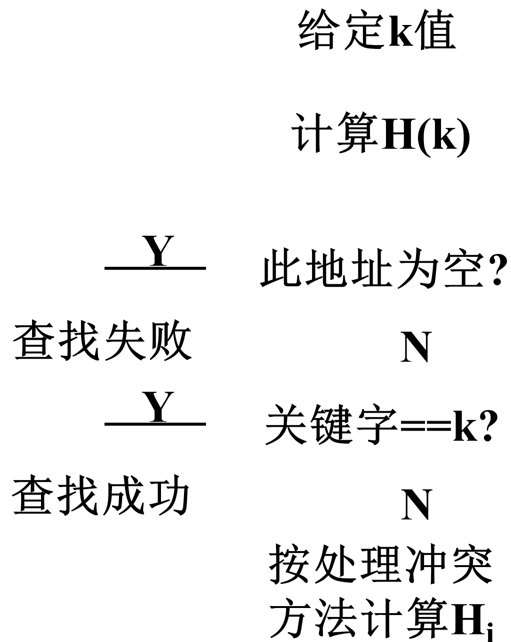


图9-18 散列表的查找过程

2 查找算法

```
#define NULLKEY   -1    /*  根据关键字类型定义空标识   */
typedef  struct
   {  KeyType   key ;     /*  关键字域  */
      otherType  otherinfo ;  /*  记录的其它域  */
   }RecType ;
int  Hash_search(RecType HT[], KeyType k, int m)
   /*   查找散列表HT中的关键字K,用开放定址法解决冲突   */
   {  int h, j ;
      h=h(k) ;
      while (j<m && !EQ(HT[h].key, NULLKEY) )
```

```
      {   if (EQ(HT[h].key, k) )   return(h) ;
          else h=R(k, ++j) ;
      }
    return(-1) ;
  }

#define M 15
typedef struct node
  {  KeyType key;
     struct node *link;
  }HNode;
```

```
HNode *hash_search(HNode *t[], KeyType k)
  {  HNode *p;    int i;
     i=h(k);
     if (t[i]==NULL)     return(NULL);
     p=t[i];
     while(p!=NULL)
        if (EQ(p->key, k))  return(p);
        else  p=p->link;
     return(NULL);
  }      /*   查找散列表HT中的关键字K,用链地址法解决冲突  */
```

3 哈希查找分析

从哈希查找过程可见：尽管散列表在关键字与记录的存储地址之间建立了直接映象，但由于“冲突”，查找过程仍是一个给定值与关键字进行比较的过程，评价哈希查找效率仍要用**ASL**。

哈希查找时关键字与给定值比较的次数取决于：

- 哈希函数；
- 处理冲突的方法；
- 哈希表的填满因子α 。填满因子α的定义是：

$$\alpha=\frac{\text{表中填入的记录数}}{\text{哈希表长度}}$$

各种散列函数所构造的散列表的ASL如下：

(1) 线性探测法的平均查找长度是：

$$S_{nl成功} \approx \frac{1}{2} \times (1+\frac{1}{1-\alpha})$$

$$S_{nl失败} \approx \frac{1}{2} \times (1+\frac{1}{(1-\alpha)^2})$$

(2) 二次探测、伪随机探测、再哈希法的平均查找长度是：

$$S_{nl成功} \approx -\frac{1}{\alpha} \times \ln(1-\alpha)$$

$$S_{nl失败} \approx \frac{1}{1-\alpha}$$

(3) 用链地址法解决冲突的平均查找长度是：

$$S_{nl成功} \approx 1+\frac{\alpha}{2}$$

$$S_{nl失败} \approx \alpha+e^{-\alpha}$$