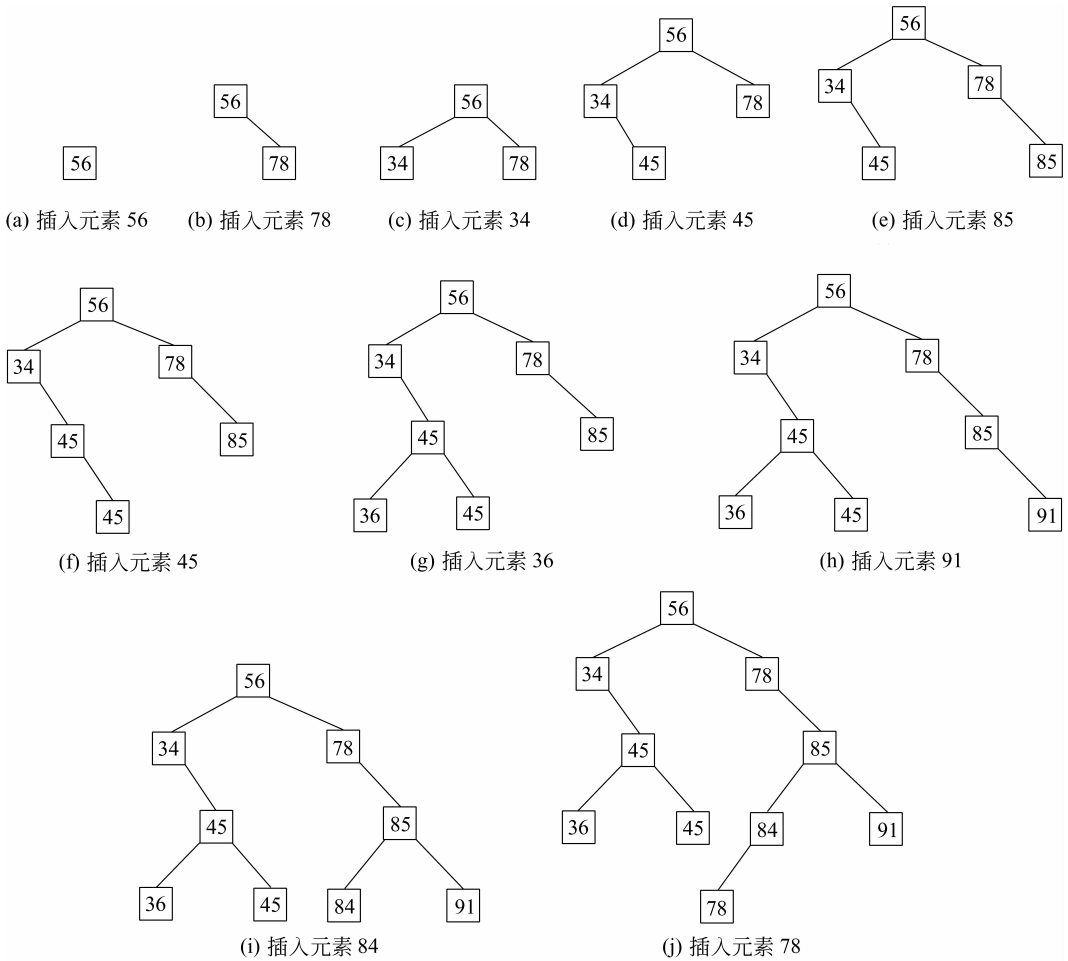
第 3 章　查找与排序技术

3.1　依次输入以下元素序列：

56,78,34,45,85,45,36,91,84,78

试构造一棵二叉排序树。要在这棵二叉排序树中查找 55,需要比较多少次？

解：构造二叉排序树的过程如图 3.1 所示。



(a) 插入元素 56　(b) 插入元素 78　(c) 插入元素 34　(d) 插入元素 45　(e) 插入元素 85

(f) 插入元素 45　(g) 插入元素 36　(h) 插入元素 91

(i) 插入元素 84　(j) 插入元素 78

图 3.1　构造 3.1 题中的二叉排序树

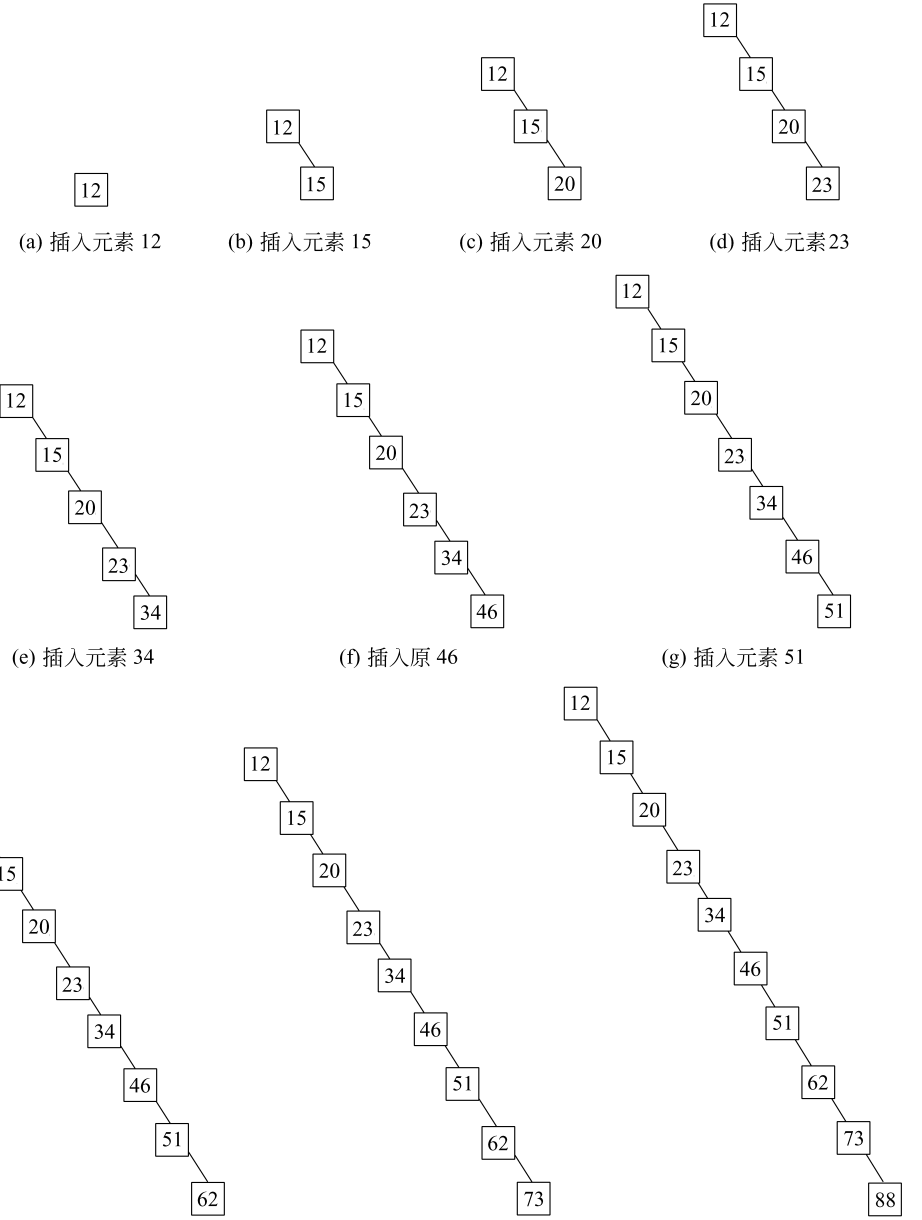
要在这棵二叉排序树中查找 55,需要比较 4 次以失败告终。

3.2　依次输入以下元素序列：

12,15,20,23,34,46,51,62,73,88

试构造一棵二叉排序树。

解：构造二叉排序树的过程如图 3.2 所示。



(a) 插入元素 12　(b) 插入元素 15　(c) 插入元素 20　(d) 插入元素23

(e) 插入元素 34　(f) 插入原 46　(g) 插入元素 51

(h) 插入元素 62　(i) 插入元素 73　(j) 插入元素 88

图 3.2　构造 3.2 题中的二叉排序树

3.3　依次输入以下元素序列：

04,18,13,76,34,45,06,23,35,12

试构造一棵 5 阶(即 $m=2$)B^- 树。

解：构造 5 阶 B^- 树的过程如图 3.3 所示。

3.4　设线性 Hash 表的长度 $n=12$，分别用下列 Hash 码将关键字元素序列(09,12,04,16,19,31,20,45,01,11,25,26)填入线性 Hash 表，并指出各关键字元素在填入过程中

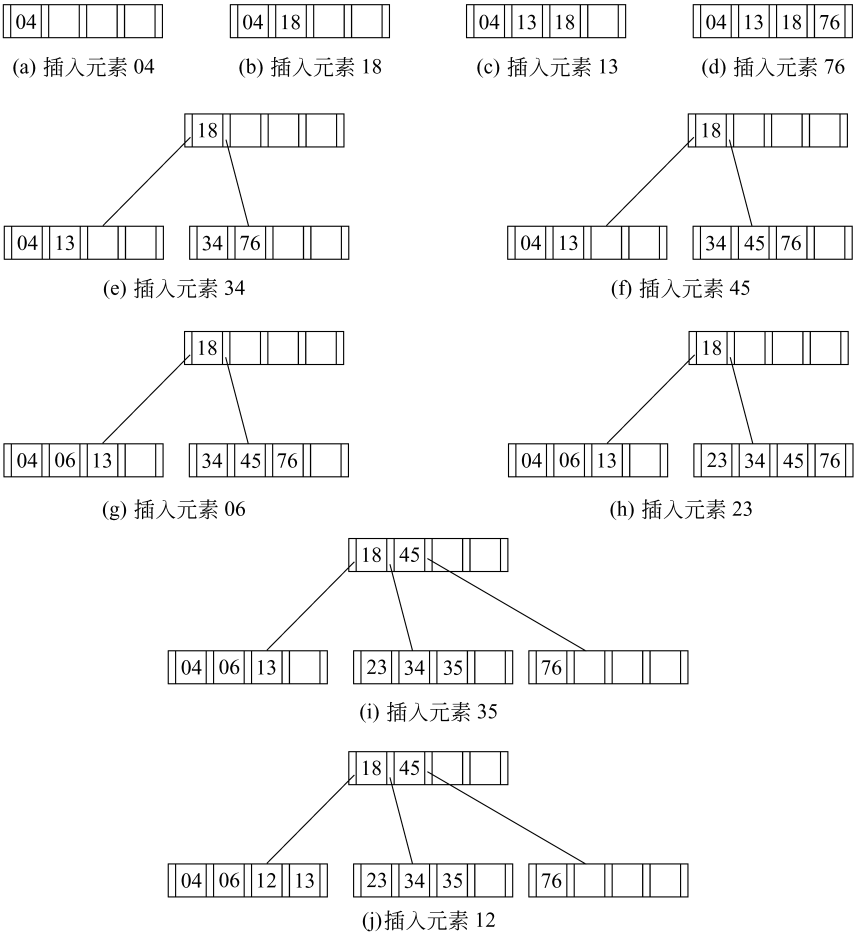


图 3.3　构造 3.3 题中的 5 阶 B^- 树

的冲突次数。

(1) $i=\mathrm{mod}(k,n)$

(2) $i=\mathrm{mod}(k*0.618,n)$

解：

(1) $i=\mathrm{mod}(k,n)$

表项序号 i	1	2	3	4	5	6	7	8	9	10	11	12
关键字 k	01	11	25	04	16	26	19	31	09	20	45	12
冲突次数	0	3	2	0	1	3	0	1	0	2	2	0

(2) $i=\mathrm{mod}(k*0.618,n)$

表项序号 i	1	2	3	4	5	6	7	8	9	10	11	12
关键字 k	01	04	45	25	09	11	12	31	16	26	19	20
冲突次数	1	0	0	1	0	0	0	1	0	6	0	0

3.5 设溢出 Hash 表中的关键字元素均为非负整数，其存储空间为数组 H(1：m)。其中 Hash 表的长度为 $n(n<m)$，并使用数组 H 的前 n 个元素；溢出表为栈结构，存储空间使用数组 H 的后 $m-n$ 个元素。Hash 码为 $i=\mathrm{mod}(k,n)$。

(1) 你准备如何表示 Hash 表中的空表项？

(2) 给出 Hash 表与溢出表的初始状态。

(3) 编写在溢出 Hash 表中填入关键字元素的算法。在此算法中应考虑关键字元素的合法性及表空间是否溢出。

(4) 编写在溢出 Hash 表中查找关键字元素的算法。在此算法中要求检查待查元素的合法性，并要求设置一个标志说明是否查到。

解：

(1) Hash 表与溢出 Hash 表中的空表项值均为 0。

(2) Hash 表与溢出表中各表项的初始值均为 0。溢出表虽然要求是栈结构，但由于在本题中对溢出表只进行填入与查找运算，因此可以将溢出表空间看成是顺序表。查找溢出表时从顺序空间的最后一项开始。

(3) 溢出 Hash 表的填入。作为溢出 Hash 表类的一个成员函数。

(4) 溢出 Hash 表的查找。作为溢出 Hash 表类的一个成员函数。

满足上述要求的溢出 Hash 表类的 C++ 描述如下：

```
//ch3_5.cpp
#include<iostream>
using namespace std;
class Over_hash                                    //溢出 Hash 表类
{ private:                                         //数据成员
    int N;                                         //Hash 表长度
    int M;                                         //Hash 表与溢出表的总长度
    int *H;                                        //Hash 表与溢出表的存储空间首地址
  public:                                          //成员函数
    Over_hash(int, int);                           //建立溢出 Hash 表存储空间
    void prt_O_hash();                             //顺序输出溢出 Hash 表中的元素
    int ins_O_hash(int, int (*f)(int,int));        //溢出 Hash 表的填入
    int sch_O_hash(int,int *,int (*f)(int,int));   //溢出 Hash 表的查找
};

//建立溢出 Hash 表存储空间
Over_hash::Over_hash(int n, int m)
{ int k;
  if (n>=m) { cout<<"n>=merr!"<<endl; return; }
  N=n;                                             //Hash 表长度
  M=m;                                             //Hash 表与溢出表的总长度
  H=new int[M];                                    //动态申请存储空间
  for (k=0; k<M; k++) H[k]=0;                      //Hash 表与溢出表各项均为空
  return;
}
```

```
//顺序输出溢出 Hash 表中的元素
void Over_hash::prt_O_hash()
{ int k;
  cout<<"Hash 表:"<<endl;
  for (k=0; k<N; k++) cout<<"<"<<H[k]<<">";
  cout<<endl;
  cout<<"溢出表:"<<endl;
  for (k=M-1; (k>=N)&&(H[k]!=0); k--) cout<<"<"<<H[k]<<">";
  cout<<endl;
  return;
}

//溢出 Hash 表的填入
//若填入的关键字不合法(x≤0),则返回函数值 0
//若溢出表溢出,则返回函数值-1
//若正常填入,则返回函数值 1
int Over_hash::ins_O_hash(int x, int (*f)(int,int))
{ int k;
  if (x<=0) return(0);                              //关键字不合法
  k=(*f)(x,N);                                      //计算 Hash 码
  if (H[k-1]==0) H[k-1]=x;                          //填入到 Hash 表
  else                                              //填入到溢出表
      { k=M-1;
        while ((k>=N)&&(H[k]!=0)) k=k-1;
        if (k<N) { cout<<"溢出表已满!"<<endl; return(-1); }
        else { H[k]=x; return(1); }
      }
  return(1);
}

//溢出 Hash 表的查找
//若查找的关键字不合法(x≤0),则返回函数值 0
//若找不到关键字 x,则返回函数值-1
//若找到关键字 x,则返回函数值 1,k 为关键字 x 所在的表项序号
int Over_hash::sch_O_hash(int x, int *k,int (*f)(int,int))
{ if (x<=0) return(0);                              //关键字不合法
  *k=(*f)(x,N);                                     //计算 Hash 码
  if (H[*k-1]==0) return(-1);                       //Hash 表表项为空,找不到
  if (H[*k-1]==x) return(1);                        //在 Hash 表中找到
  *k=M-1;                                           //到溢出表中去找
  while ((*k>=N)&&(H[*k]!=0)&&(H[*k]!=x)) *k=*k-1;
  if (H[*k]==x) { *k=*k+1; return(1); }             //在溢出表中找到
  return(-1);                                       //溢出表中也没有这个关键字,返回
}
```

主函数如下：

编制一个 C++ 程序，首先将关键字序列(09,31,26,19,01,13,02,11,27,16,05,21)依次填入长度为 $n=12$ 的溢出 Hash 表中；然后在该溢出 Hash 表中查找一个指定的元素(由键盘输入)。设存储空间为 H(1：20)(即 $m=20$)，Hash 码为 $i=\mathrm{mod}(k,n)$。

```
//主函数
int main()
{ int k,xx,i(int,int);
  Over_hash p(12,20);            //建立一个长度为 12,存储空间为 20 的溢出 Hash 表
  int x[12]={9,31,26,19,1,13,2,11,27,16,5,21};
  for (k=0; k<12; k++) cout<<p.ins_O_hash(x[k], i);
  cout<<endl;
  p.prt_O_hash();                //顺序输出溢出 Hash 表中的元素
  cout<<"input xx=";
  cin>>xx;
  cout<<"flag="<<p.sch_O_hash(xx,&k,i)<<endl;
  cout<<"k="<<k<<endl;
  return 0;
}
//Hash 码
int i(int x, int n)
{ int k;
  k=x% n;
  if (k==0) k=n;
  return(k);
}
```

运行结果如下(带下画线的为键盘输入)：

```
111111111111
Hash 表：
<1><26><27><16><5><0><31><0><9><0><11><0>
溢出表：
<19><13><2><21>
input k=31 [查找关键字 31]
flag=1
k=7
```

3.6 设随机 Hash 表的长度为 $n=8$。

(1) 利用本章给出的算法，写出伪随机数序列中的前 6 个随机数。

(2) 设 Hash 码为 $i=\mathrm{mod}(k*0.618,n)$。将关键字元素序列(19,31,20,45,01,11,25,26)填入随机 Hash 表，并注明冲突次数。

解：

(1) 伪随机数序列中前 6 个随机数为 1,6,7,4,5,2。

(2) 随机 Hash 表为

表项序号	1	2	3	4	5	6	7	8
关键字 k	45	26	19	31	20	01	11	25
冲突次数	2	6	0	1	1	2	1	1

3.7 Hash 表技术的目标是什么？如何提高 Hash 表的查找效率？

解：Hash 表技术的目标是提高查找效率。

为了提高 Hash 表查找效率，一方面要使 Hash 码的均匀性比较好，另一方面要使 Hash 码的计算比较简单。

3.8 扼要归纳本章所介绍的几种 Hash 表的适用对象及其主要优、缺点。

解：

(1) 线性 Hash 表。

优点：适用于 Hash 码冲突较少并且不应填满的情况。

其主要缺点如下：

① 容易产生“堆聚”现象。

② 在处理冲突的过程中会产生新的冲突。

③ 在 Hash 表填满时，其平均查找次数为无穷。

(2) 随机 Hash 表。

优点：适用于 Hash 码冲突较少并且不应填满的情况。

其主要缺点如下：

① 在处理冲突的过程中会产生新的冲突。

② 在 Hash 表填满时，其平均查找次数为无穷。

(3) 溢出 Hash 表。

优点：适用于 Hash 码冲突较少的情况。

其主要缺点如下：

① 除 Hash 表本身外，还要增加一个溢出表。当 Hash 码不能遍历 Hash 表本身时，额外的溢出表空间是一种浪费。

② 在 Hash 码不太均匀而冲突较多的情况下，溢出表中项数较多，顺序查找的效率会降低。

(4) 拉链 Hash 表。

优点：适用于 Hash 码冲突较多的情况。

其主要缺点：拉链需要额外开销。

(5) 指标 Hash 表。

优点：主要用于关键字元素内容长度不等的情况。

其主要缺点如下：指标表的存储空间是一种浪费。

3.9 分别用冒泡排序及希尔排序对下列线性表进行排序。要求给出中间每一步的结果。

(1) (81,52,57,22,95,04,83,96,42,32,48,78,14,87,67)

(2) (424,887,807,709,882,616,573,413,679,180,975,264)

解：

(1)

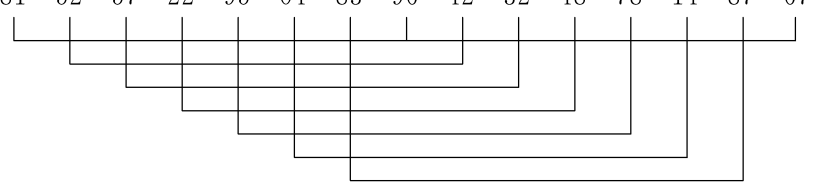
① 冒泡排序(其中符号“～”表示前后两个元素交换)：

原序列	81	52	57	22	95	04	83	96	42	32	48	78	14	87	67
第 1 遍(从前向后)	81～52～57～22				95～04～83			96～42～32～48～78～14～87～67							
结果	52	57	22	81	04	83	95	42	32	48	78	14	87	[67]	96
(从后向前)	52～57～22～81～04				83～95～42～32～48～78～14			87～67							
结果	04	[52]	57	22	81	14	83	95	42	32	48	78	67	[87]	96
第 2 遍(从前向后)	04	52	57～22		81～14		83	95～42～32～48～78～67～87							96
结果	04	[52]	22	57	14	81	83	42	32	48	78	67	[87]	95	96
(从后向前)	04	52～22～57～14			81～83～42～32			48	78～67			87	95		96
结果	04	14	[52]	22	57	32	81	83	42	48	67	78	[87]	95	96
第 3 遍(从前向后)	04	14	52～22		57～32		81	83～42～48～67～78					87	95	96
结果	04	14	[22]	52	32	57	81	42	48	67	[78]	83	87	95	96
(从后向前)	04	14	22	52～32		57～81～42		48	67	78	83	87	95		96
结果	04	14	22	32	[52]	42	57	81	48	67	[78]	83	87	95	96
第 4 遍(从前向后)	04	14	22	32	52～42		57	81～48～67～78				83	87	95	96
结果	04	14	22	32	[42]	52	57	48	67	[78]	81	83	87	95	96
(从后向前)	04	14	22	32	42	52～57～48		67	78	81	83	87	95		96
结果	04	14	22	32	42	48	[52]	57	67	[78]	81	83	87	95	96
第 5 遍(从前向后)	04	14	22	32	42	48	52	57	67	78	81	83	87	95	96
最后结果	04	14	22	32	42	48	52	57	67	78	81	83	87	95	96

② 希尔排序：

增量序列一般取 $h_t=n/2^k(k=1,2,\cdots,[\log_2 n])$，其中 n 为待排序序列的长度。

原序列	81	52	57	22	95	04	83	96	42	32	48	78	14	87	67
$h=7$	81	52	57	22	95	04	83	96	42	32	48	78	14	87	67



结果	67	42	32	22	78	04	83	81	52	57	48	95	14	87	96
$h=3$	67	42	32	22	78	04	83	81	52	57	48	95	14	87	96

结果	14	42	04	22	48	32	57	78	52	67	81	95	83	87	96
$h=1$	14	42	04	22	48	32	57	78	52	67	81	95	83	87	96
最后结果	04	14	22	32	42	48	52	57	67	78	81	83	87	95	96

(2)

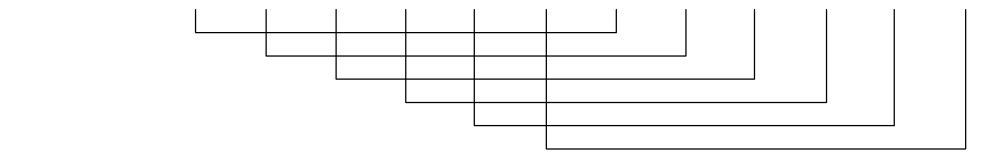
① 冒泡排序：

原序列	424	887	807	709	882	616	573	413	679	180	975	264
第 1 遍(从前向后)	424	887～807～709～882～616～573～413～679～180									975～264	
结果	424	807	709	882	616	573	413	679	180	887	[264]	975
(从后向前)	424～807～709～882～616～573～413～679～180									887～264		975
结果	180	[424]	807	709	882	616	573	413	679	264	[887]	975
第 2 遍(从前向后)	180	424	807～709		882～616～573～413～679～264						887	975
结果	180	[424]	709	807	616	573	413	679	[264]	882	887	975
(从后向前)	180	424～709～807～616～573～413～679～264								882	887	975
结果	180	264	[424]	709	807	616	573	413	[679]	882	887	975
第 3 遍(从前向后)	180	264	424	709	807～616～573～413～679					882	887	975
结果	180	264	[424]	709	616	573	413	[679]	807	882	887	975
(从后向前)	180	264	424～709～616～573～413					679	807	882	887	975
结果	180	264	413	[424]	709	616	573	[679]	807	882	887	975
第 4 遍(从前向后)	180	264	413	424	709～616～573～679				807	882	887	975
结果	180	264	413	[424]	616	573	[679]	709	807	882	887	975
(从后向前)	180	264	413	424	616～573		679	709	807	882	887	975
结果	180	264	413	[424]	573	[616]	679	709	807	882	887	975
第 5 遍(从前向后)	180	264	413	424	573	616	679	709	807	882	887	975
最后结果	180	264	413	424	573	616	679	709	807	882	887	975

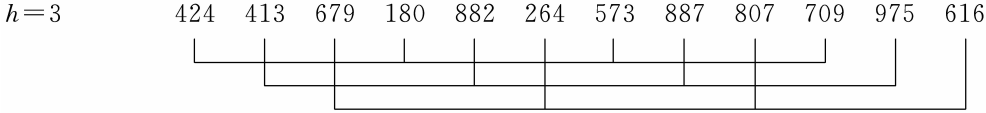
② 希尔排序：

增量序列一般取 $h_t=n/2^k(k=1,2,\cdots,[\log_2 n])$，其中 n 为待排序序列的长度。

原序列	424	887	807	709	882	616	573	413	679	180	975	264
$h=6$	424	887	807	709	882	616	573	413	679	180	975	264



结果	424	413	679	180	882	264	573	887	807	709	975	616
$h=3$	424	413	679	180	882	264	573	887	807	709	975	616



结果	180	413	264	424	882	616	573	887	679	709	975	807
$h=1$	180	413	264	424	882	616	573	887	679	709	975	807
最后结果	180	264	413	424	573	616	679	709	807	882	887	975

3.10 试编写归并排序的递归算法。

解：归并排序递归算法的 C++ 描述如下(其中两两归并算法与主教材上的算法完全相同)：

```
//ch3_010.cpp
#include<iostream>
using namespace std;
template<class T>
void merg_sort(T p[], int m, int n, T a[])
{ if (m==n) a[m-1]=p[m-1];
  else
  { merg_sort(p, m, (m+n)/2, a);          //对前半部分进行归并排序
    merg_sort(p, (m+n)/2+1, n, a);        //对后半部分进行归并排序
    merge(a, m, (m+n)/2, n, p);           //两两归并
  }
  return;
}

//两两归并
template<class T>
static void merge(T p[], int low, int mid, int high, T a[])
{ int i,j,k;
  i=low; j=mid+1; k=low;
  while ((i<=mid)&&(j<=high))
      { if(p[i-1]<=p[j-1])
          { a[k-1]=p[i-1]; i=i+1; }
        else
          { a[k-1]=p[j-1]; j=j+1; }
        k=k+1;
      }
  if (i<=mid)
      for (j=i; j<=mid; j++)
        { a[k-1]=p[j-1]; k=k+1; }
  else
      if (j<=high)
        for (i=j; i<=high; i++)
          { a[k-1]=p[i-1]; k=k+1; }
  for (i=low; i<=high; i++)
          p[i-1]=a[i-1];
  return;
}
```

主函数如下：

```
//主函数
#include<iomanip>
int main()
{ int i,j;
```

```
    double a[50],p[50],r=1.0;
    for (i=0; i<50; i++)              //产生 50 个 0~1 的随机数
        { r=2053.0*r+13849.0; j=r/65536.0;
          r=r-j*65536.0; p[i]=r/65536.0;
        }
    for (i=0; i<50; i++)              //产生 50 个 100~300 的随机数
          p[i]=100.0+200.0*p[i];
    cout<<"排序前的序列:"<<endl;
    for (i=0; i<10; i++)              //每行输出 5 个数据
        { for (j=0; j<5; j++) cout<<setw(10)<<p[5*i+j];
          cout<<endl;
        }
    merg_sort(p, 1, 50, a);           //对原序列进行归并排序
    cout<<"排序后的序列:"<<endl;
    for (i=0; i<10; i++)              //每行输出 5 个数据
        { for (j=0; j<5; j++) cout<<setw(10)<<p[5*i+j];
          cout<<endl;
        }
    return 0;
}
```

运行结果如图 3.4 所示。

图 3.4　运行结果

3.11　分别依次读入 3.9 题中的两个无序序列中的元素,构造两个堆。

解:

(1) 序列(81,52,57,22,95,04,83,96,42,32,48,78,14,87,67)的堆如图 3.5 所示。

(2) 序列(424,887,807,709,882,616,573,413,679,180,975,264)的堆如图 3.6 所示。

3.12　用快速排序法对 3.9 题中的两个无序序列进行排序。

解:快速排序主要是对序列进行分割。下面给出对序列进行分割以及对分割出的各子序列再进行分割后的结果。各子序列用下画线表示,各子序列的分界线位置上的元素用方

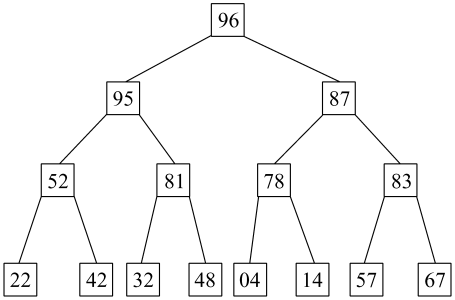


图 3.5　堆(1)

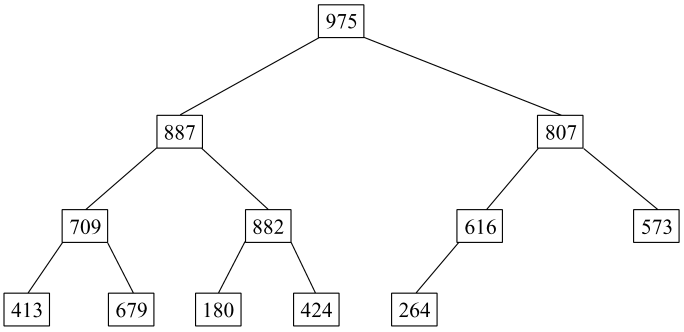


图 3.6　堆(2)

框表示。

(1) 序列(81,52,57,22,95,04,83,96,42,32,48,78,14,87,67)

67	52	57	22	14	04	78	48	42	32	81	96	83	87	95
04	14	22	32	57	52	78	48	42	67	81	87	83	95	96
04	14	22	32	57	52	42	48	67	78	81	83	87	95	96
04	14	22	32	48	42	52	57	67	78	81	83	87	95	96
04	14	22	32	42	48	52	57	67	78	81	83	87	95	96

最后结果为　04　14　22　32　42　48　52　57　67　78　81　83　87　95　96。

(2) 序列(424,887,807,709,882,616,573,413,679,180,975,264)

264	180	413	573	424	616	882	709	679	807	975	887
264	180	413	573	424	616	807	709	679	882	975	887
180	264	413	424	573	616	679	709	807	882	887	975
180	264	413	424	573	616	679	709	807	882	887	975

最后结果为　180　264　413　424　573　616　679　709　807　882　887　975。

3.13　编写下列程序：

(1) 产生 1000 个伪随机数，并依次存入一个数据文件中。

(2) 对此 1000 个伪随机数序列分别用冒泡排序、快速排序、希尔排序和堆排序方法进行排序，并比较它们的运行时间。

解：略。

3.14 设自然数集合为 $D=\{1,2,3,4,5,6,7\}$,反映元素间前后件关系的二元组集合为

$$R=\{(1,2),(1,3),(2,4),(2,5),(2,6),(3,5),(3,7),(5,7),(6,7)\}$$

试确定一个相对于 R 的 D 的拓扑分类序列。

解:拓扑分类序列不是唯一的。运行主教材上的算法程序所得到的拓扑分类序列为

$$1,2,4,6,3,5,7$$

3.15 编写线性表在链式存储结构下的冒泡排序算法。

解:面向对象的 C++ 描述如下:

```
//ch3_15.cpp
#include<iostream>
using namespace std;
//定义结点类型
template<class T>                          //T 为虚拟类型
struct node
{ T d;
  node * next;
};
template<class T>                          //模板声明,数据元素虚拟类型为 T
class linked_List                          //单链表类
{ private:                                 //数据成员
    node<T> * head;                        //链表头指针
  public:                                  //成员函数
    linked_List() { head=NULL; return; }   //构造函数,建立空链表
    void prt_linked_List();                //从头指针开始扫描输出链表中的元素
    void ins_linked_List(T);               //将新元素插入到链头
    void bub_sort();                       //冒泡排序
};
//从头指针开始扫描输出链表中的元素
template<class T>
void linked_List<T>::prt_linked_List()
{ node<T> * p;
  p=head;
  if (p==NULL) { cout<<"空链表!"<<endl; return; }
  do { cout<<p->d<<",";
       p=p->next;
    } while (p!=NULL);
  cout<<endl;
  return;
}
//将新元素插入到链头
template<class T>
void linked_List<T>::ins_linked_List(T x)
{ node<T> * p;
  p=new node<T>;
  p->d=x;
```

```
  p->next=head; head=p;
  return;
}
//冒泡排序
template<class T>
void linked_List<T>::bub_sort()
{ T t;
  node<T> * p, * q, * s=NULL;
  p=head;
  while (s!=head)
    { p=head; q=p->next;
      while (q!=s)
        { if (p->d>q->d)
              { t=p->d; p->d=q->d; q->d=t; }
          p=q; q=q->next;
        }
      s=p;
    }
  return;
}
```

主函数如下：

```
//主函数
int main()
{ linked_List<int>s;
  int k;
  int d[19]={81,52,57,22,95,4,83,52,96,42,52,22,32,78,14,87,67,45,64};
  for (k=18; k>=0; k--)
      s.ins_linked_List(d[k]);
  cout<<"排序前:"<<endl;
  s.prt_linked_List();
  s.bub_sort();
  cout<<"排序后:"<<endl;
  s.prt_linked_List();
  return 0;
}
```

运行结果如下：

```
排序前:
81,52,57,22,95,4,83,52,96,42,52,22,32,78,14,87,67,45,64
排序后:
4,14,22,22,32,42,45,52,52,52,57,64,67,78,81,83,87,95,96
```

第4章　资源管理技术

4.1　什么是操作系统？它的主要功能是什么？

解：操作系统实际上是由一些程序模块组成的，是系统软件中最基本的部分，其主要作用有以下几个方面：

（1）管理系统资源。包括对CPU、内存储器、输入输出设备、数据文件和其他软件资源的管理。

（2）为用户提供资源共享的条件和环境，并对资源的使用进行合理调度。

（3）提供输入输出的方便环境，简化用户的输入输出工作，提供良好的用户界面。

（4）规定用户的接口，发现、处理或报告计算机操作过程中所发生的各种错误。

由此可以看出，操作系统既是计算机系统资源的控制和管理者，又是用户和计算机系统之间的接口，当然它本身也是计算机系统的一部分。因此，概略地说，操作系统是用以控制和管理系统资源、方便用户使用计算机的程序的集合。

如果把操作系统看成是计算机系统资源的管理者，则操作系统的功能和任务主要有以下五个方面：

（1）处理机管理。

处理机管理的主要任务是：充分发挥处理机的作用，提高它的使用效率。

（2）存储器管理。

存储器管理的主要任务是：对有限的内存储器进行合理的分配，以满足多个用户程序运行的需要。

（3）设备管理。

设备管理的主要任务是：有效地管理各种外部设备，使这些设备充分发挥效率；并且还要给用户提供简单而易于使用的接口，以便在用户不了解设备性能的情况下，也能很方便地使用它们。

（4）文件管理。

文件管理的主要任务是：唯一地标识计算机系统中的每一组信息，以便能够对它们进行合理的访问和控制；以及有条理地组织这些信息，使用户能够方便且安全地使用它们。

（5）作业管理。

作业管理的主要任务是：对所有的用户作业进行分类，并且根据某种原则，源源不断地选取一些作业交给计算机去处理。

4.2　分时系统与实时系统的主要区别是什么？

解：分时系统是指多个用户分享使用同一台计算机，即在一台计算机上连接若干台终端，每个用户可以独占一台终端。分时系统具有以下几方面的特点：

（1）同时性。若干远、近程终端上的用户，在各自的终端上同时使用一台计算机。

（2）独立性。同一台计算机上的用户在各自的终端上独立工作，互不干扰。

(3) 及时性。用户可以在很短的时间内得到计算机的响应。

(4) 交互性。分时系统提供了人机对话的条件,用户可以根据系统对自己请求的响应情况,继续向系统提出新的要求,便于程序的检查和调试。

具有实时要求的系统称为实时系统。在实时系统中,要求对随机发生的外部事件及时响应并对其进行处理。实时系统的特点是严格的时间限制,它要求计算机对输入的信息快速响应,并在规定的时间内完成规定的操作。

4.3 多窗口系统的主要特点是什么?

解:

(1) 灵活、方便的窗口操作。用户可以随时决定窗口的位置、大小、有无,还可以随时启动命令的执行,而在命令执行过程中还可以与系统"对话"。这样,用户就可以得心应手地对计算机进行各种操作。

(2) 在多窗口系统下,各个程序都有自己的菜单,系统还有自己的主菜单,它们不可能在有限的屏幕上同时显示。因此,在多窗口系统中,一般采用"弹出式菜单"方式,即每个应用程序的命令按其性质分成若干组,在窗口上只列出菜单的名字,需要时选择适当的菜单名将其菜单"弹出"。

(3) 在多窗口系统中,各种对话一般是通过对话框来实现的。若某命令执行时需要和用户对话,就会在屏幕上显示一个对话框,用户可以在对话框内选择对象、输入参数或与程序对话。对话完成后,对话框就消失。

多窗口系统能提供将多个作业同时展现在用户面前的操作环境,每个作业占据一个窗口,用户可以交替地与各个窗口进行对话,各窗口之间也可以互相通信、交换信息。

4.4 并发执行的程序有什么特点?它与顺序执行的程序有什么本质的区别?

解:

(1) 并发程序没有封闭性。

顺序程序具有封闭性,程序执行后的输出结果是一个与时间无关的函数;并发程序的输出结果与各程序执行的相对速度有关,失去了程序的封闭性这个特点。

(2) 程序与其执行过程不是一一对应的关系。

在顺序程序设计中,由程序的封闭性决定了程序与其执行过程是完全对应的,程序的执行路径、执行时间和所执行的操作都可以从程序中反映出来。并发执行程序的情况下,程序的执行过程由当时的系统环境与条件所决定,程序与其执行过程不再有一一对应的关系。当多个执行过程共享某个程序时,它们都可以调用这个程序,调用一次即对应一个执行过程,也就是说,这个共享的程序对应多个执行过程。

(3) 程序并发执行可以互相制约。

并发程序的执行过程是复杂的,它们之间不但可能有互为因果的直接制约关系,而且还可能由于共享某些资源或过程而具有间接的互相制约关系。

4.5 什么是进程?它与程序有什么关系?

解: 所谓进程,是指一个具有一定独立功能的程序关于某个数据集合的一次运行活动。简单地说,进程是可以并发执行的程序的执行过程,它是控制程序管理下的基本的多道程序单位。

进程与程序有关,但它与程序又有本质的区别。主要反映在以下几个方面。

(1) 进程是程序在处理机上的一次执行过程,它是动态的概念。程序只是一组指令的有序集合,其本身没有任何运行的含义,它是一个静态的概念。

(2) 进程是程序的执行过程,是一次运行活动。因此,进程具有一定的生命期,它能够动态地产生和消亡。即进程可以由创建而产生,由调度而执行,因得不到资源而暂停,以致最后由撤销而消亡。也就是说,进程的存在是暂时的;而程序是可以作为一种软件资源长期保存的,它的存在是永久的。

(3) 进程是程序的执行过程,因此,进程的组成应包括程序和数据。除此之外,进程还包括由记录进程状态信息的"进程控制块"。

(4) 一个程序可能对应多个进程。

(5) 一个进程可以包含多个程序。

4.6 进程的三种状态之间是如何转换的?

解:

(1) 处于就绪状态的进程,一旦分配到 CPU,就转为运行状态。

(2) 处于运行状态的进程,当需要等待某个事件发生才能继续运行时,则转为等待状态;或者由于分配给它的时间片用完,就让出 CPU 而转为就绪状态。

(3) 处于等待状态的进程,如果它等待的事件已经发生,即条件得到满足,就转为就绪状态。

进程只能在运行状态下结束。

4.7 什么是死锁?为什么会发生死锁?

解:在多个进程并发执行时,共享系统的软硬件资源。各进程互相独立地动态获得、不断申请和释放系统中的软硬件资源,这就有可能使系统出现这样一种状态:其中若干个进程均因互相"无知地"等待对方所占有的资源而无限等待。这种状态称为死锁。

发生死锁的四个必要条件如下:

(1) 资源的独占使用。

(2) 资源的非抢占分配。

(3) 资源的部分分配。

(4) 对资源的循环等待。

4.8 进程的互斥与同步有什么区别?它们之间有什么共同之处?

解:当多个进程共享数据块或其他排他性使用的资源时,不能同时进入存取或使用,但进入的次序可以任意,这些进程之间的这种制约关系称为互斥。

进程之间为了合作完成一个任务,而需要互相等待和互相交换信息的相互制约关系称为同步。

互斥也是一种特殊的同步关系。进程之间为了实现互斥或同步,都需要有信息传递,也就是说需要进行通信。

4.9 考虑一个主程序用如下方式调用子程序:$y=f(x)$。如果把子程序 f 作为进程实现,则要考虑哪些同步问题?

解:在本问题中,有两个进程合作完成 $y=f(x)$的计算工作,其中与主程序对应的进程(假设为 MAIN)为子程序对应的计算进程(假设为 F)提供形参 x 的值,而计算进程 F 为进程 MAIN 提供计算结果 y 的值。其中形参 x 与计算结果 y 分别用各自的缓冲单元存放。

为了正确完成计算并输出结果，它们之间必须是同步工作的。

当计算进程 F 尚未完成 $y=f(x)$的计算，即还没有把结果 y 送到缓冲单元之前，进程 MAIN 在需要使用计算值 y 时应该等待；一旦计算进程 F 把计算结果送入缓冲单元中，则应给进程 MAIN 发出一个通知信号；进程 MAIN 收到通知信号后，便可以从缓冲单元中取出计算结果。反之，在进程 MAIN 还未将 x 值送入缓冲单元之前，计算进程 F 也不能进行计算；一旦进程 MAIN 将 x 值送入缓冲单元中，则应给计算进程 F 发出一个通知信号；计算进程 F 收到通知信号后，便可以从缓冲单元中取出 x 值进行 $y=f(x)$的计算。

4.10 P/V 操作与消息缓冲通信有什么共同之处？又有什么区别？

解：进程之间为了实现互斥或同步，需要有信息传递，也就是说需要进行通信。为此，需要一种实现进程之间通信的机构，这种机构通常称为通信原语。P/V 操作属于低级通信原语，消息缓冲通信属于高级通信原语，它们都用于实现进程之间的通信。

P/V 操作是采用信号同步，发送者只给对方发出一个简单的信号，接收者在收到该信号后，就能够知道其中的含义，从而采取相应的操作。

消息缓冲通信采用信件同步，发送者向对方发出的不是简单的信号，而是一个复杂的信件，接收者收到信件后，要对信件进行分析，然后再采取相应的操作。

4.11 存储管理系统有哪些主要功能？

解：在多道程序系统中，存储管理一般应包括以下一些功能：

(1) 地址变换。要把用户程序中的相对地址转换成实际内存空间的绝对地址。

(2) 内存分配。根据各用户程序的需要以及内存空间的实际大小，按照一定的策略划分内存，以便分配给各个程序使用。

(3) 存储共享与保护。由于各用户程序与操作系统同在内存，因此，一方面允许各用户程序能够共享系统或用户的程序和数据，另一方面又要求各程序之间互不干扰或破坏对方。

(4) 存储器扩充。由于多道程序共享内存，使内存资源尤为紧张，这就要求操作系统根据各时刻用户程序允许的情况合理地利用内存，以便确保当前需要的程序和数据在内存，而其余部分可以暂时放在外存中，等确实需要时再调入内存。

4.12 什么是重定位？为什么要对程序进行重定位？

解：在进行地址变换时，必须修改程序中所有与地址有关的项，也就是说，要对程序中的指令地址以及指令中有关地址的部分(称为有效地址)进行调整，这个调整过程称为地址重定位。

用户在编写程序时并不知道自己的程序在执行时放在内存空间的什么区域，因此不可能用内存中的实际地址(称为绝对地址)来编写程序，只能相对于某个基准地址(通常为 0 地址)来编写程序、安排指令和数据的位置，这种在用户程序中所用的地址通常称为相对地址。当用户程序进入内存执行时，又必须要把用户程序中的所有相对地址转换成内存中的实际地址(称为地址变换)，否则用户程序就无法执行。

4.13 什么是虚拟存储器？它的大小受什么限制？

解：一般来说，一个作业的大小不能超过实际内存空间的大小，实际内存空间是用户进行程序设计时可以利用的最大空间。实际上，根据程序的时间局部性和空间局部性，在作业运行过程中可以只让当前用到的信息进入内存，其他当前未用的信息留在外存；而当作业进

一步运行需要用到外存中的信息时，再把已经用过但暂时还不会用到的信息换到外存，把当前要用的信息换到已空出的内存区中，从而给用户提供了一个比实际内存空间大得多的地址空间。对于用户来说，这个特别大的地址空间就好像是可以自由使用的内存空间一样。这种大容量的地址空间并不是真实的存储空间，而是虚拟的，因此，称这样的存储器为虚拟存储器，用于支持虚拟存储器的外存称为后备存储器。

虚拟存储器的大小受外存容量的限制。

4.14 分页系统与分段系统各有什么优缺点？

解：分页系统的优点为：

(1) 由于提供了大容量的虚拟存储器，用户的地址空间不再受内存大小的限制，大大方便了用户的程序设计。

(2) 由于作业地址空间中的各页面都是按照需要调入内存的，不用的信息不会调入内存，很少用的信息也只是短时间驻留在内存，因此更有效地利用了内存。

(3) 由于动态分页管理提供了虚拟存储器，每个作业一般只有一部分信息占用内存，从而可以容纳更多的作业进入系统，这就更有利于多道程序的运行。

其缺点是不利于程序的动态连接装配，也不利于程序与数据的共享。

分段系统的优点为：有利于程序的动态连接装配，也有利于程序与数据的共享，从而更有利于用户的程序设计。

其缺点是不利于内存的有效利用。

4.15 文件系统的主要任务是什么？它为用户提供哪些主要功能？

解：所谓文件系统，是指负责存取和管理文件信息的软件机构。文件系统的主要任务是：唯一地标识计算机系统中的每一组信息，以便能够对它们进行合理的访问和控制；以及有条理地组织这些信息，使用户能够方便且安全地使用它们。

其主要功能为：

(1) 自动处理文件的存储和访问。借助于文件系统，用户可以简单、方便地使用文件，而不必考虑文件存储空间的分配，也无须知道文件的具体存放位置。

(2) 通过文件的存取权限，对文件提供保护措施，并提供转储功能，为文件复制后备副本等。

总之，文件系统一方面要方便用户，实现对文件的“按名存取”；另一方面要实现对文件存储空间的组织、分配和文件信息的存储，并且要对文件提供保护和有效的检索等功能。

4.16 文件的物理组织形式主要有哪几种？文件系统对磁盘空闲空间是如何管理的？

解：根据文件在存储空间中的存放形式，文件可以分为连续文件、链接文件和索引文件。

文件系统对磁盘空闲空间通常有以下几种管理方案：

(1) 空闲文件项和空闲区表。

空闲文件项是一种最简单的空闲区管理方法。在这种方法中，空闲区与其他文件目录放在一张表中。在分配时，系统依次扫描这个目录表，从标记为空闲的项中寻找长度满足要求的项，然后把相应项的空闲标记去掉，填上文件名。在删除文件时，只要把文件名栏标记

为空闲即可。

在这种空闲区的管理方案中，由于空闲区与真正的目录表混在一起，因此，无论是分配空间还是查找目录，效率都不高。另外，如果空闲区比所申请的区大，则多余的部分有可能被浪费。为解决这些问题，可以采用空闲区表的方法，即将空闲区项抽出来单独构成一张表，这样可以减少目录管理的复杂性，提高文件查找和空闲区查找的速度。

(2) 空闲块链。

空闲块链是指将所有空闲块链接在一起。当需要空闲块时，从链头依次摘取一(些)块，且将链头指针依次指向后面的空闲块。当文件被删除而释放空闲块时，只需将被释放的空闲块挂到空闲块链的链头即可。

(3) 位示图。

位示图的方法是用若干字节构成一张表，表中的每一个二进制位对应一个物理块，并依次顺序编号。如果位标记为“1”，则表示对应的物理块已分配；位标记为“0”，则表示对应物理块为空闲。在存储分配时，只要把找到的空闲块位标记改为“1”；释放时，只要把相应的位标记改为“0”即可。

(4) 空闲块成组链接法。

在 UNIX 操作系统中，采用改进空闲块链方法来组织管理文件存储空间。其方法是，将所有的空闲块进行分组，再通过指针将组与组之间链接起来。这种空闲块的管理方法称为成组链接法。

4.17 设一个文件有 326 个逻辑块，另一个文件有 2127 个逻辑块，按文件的多级索引结构画出这两个文件的索引结构图。

解：具有 326 个逻辑块的文件多级索引结构图如图 4.1 所示。具有 2127 个逻辑块的文件多级索引结构图如图 4.2 所示。

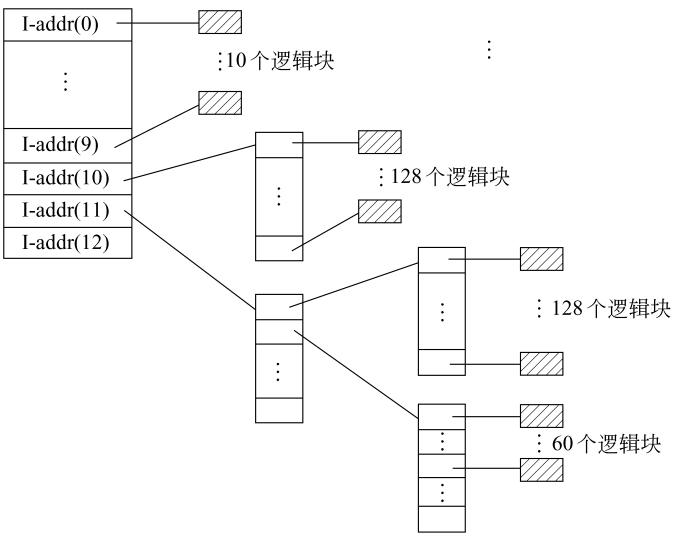


图 4.1 具有 326 个逻辑块的文件多级索引结构图