浙江师范大学数理与信息工程学院

省级精品课程《数据结构与算法分析》

# 例题选编

2006 年 10 月

# 目　　录

# 第一章 绪论

1-1 什么是数据？它与信息是什么关系?

【解答】

什么是信息？广义地讲，信息就是消息。宇宙三要素（物质、能量、信息）之一。它是现实世界各种事物在人们头脑中的反映。此外，人们通过科学仪器能够认识到的也是信息。信息的特征为：可识别、可存储、可变换、可处理、可传递、可再生、可压缩、可利用、可共享。

什么是数据？因为信息的表现形式十分广泛，许多信息在计算机中不方便存储和处理，例如，一个大楼中 4 部电梯在软件控制下调度和运行的状态、一个商店中商品的在库明细表等，必须将它们转换成数据才能很方便地在计算机中存储、处理、变换。因此，数据(data)是信息的载体，是描述客观事物的数、字符、以及所有能输入到计算机中并被计算机程序识别和处理的符号的集合。在计算机中，信息必须以数据的形式出现。

1-2 什么是数据结构? 有关数据结构的讨论涉及哪三个方面?

【解答】

数据结构是指数据以及相互之间的关系。记为：数据结构 = { D, R }。其中，D 是某一数据对象，R 是该对象中所有数据成员之间的关系的有限集合。

有关数据结构的讨论一般涉及以下三方面的内容：

① 数据成员以及它们相互之间的逻辑关系，也称为数据的逻辑结构，简称为数据结构；

② 数据成员极其关系在计算机存储器内的存储表示，也称为数据的物理结构，简称为存储结构；

③ 施加于该数据结构上的操作。

数据的逻辑结构是从逻辑关系上描述数据，它与数据的存储不是一码事，是与计算机存储无关的。因此，数据的逻辑结构可以看作是从具体问题中抽象出来的数据模型，是数据的应用视图。数据的存储结构是逻辑数据结构在计算机存储器中的实现（亦称为映像），它是依赖于计算机的，是数据的物理视图。数据的操作是定义于数据逻辑结构上的一组运算，每种数据结构都有一个运算的集合。例如搜索、插入、删除、更新、排序等。

1-3 数据的逻辑结构分为线性结构和非线性结构两大类。线性结构包括数组、链表、 栈、队列、优先级队列等; 非线性结构包括树、图等、这两类结构各自的特点是什么？

【解答】

线性结构的特点是：在结构中所有数据成员都处于一个序列中，有且仅有一个开始成员和一个终端成员，并且所有数据成员都最多有一个直接前驱和一个直接后继。例如，一维数组、线性表等就是典型的线性结构

非线性结构的特点是：一个数据成员可能有零个、一个或多个直接前驱和直接后继。例如，树、图或网络等都是典型的非线性结构。

1-4．什么是抽象数据类型？试用 C++的类声明定义“复数”的抽象数据类型。要求

(1) 在复数内部用浮点数定义它的实部和虚部。

(2) 实现 3 个构造函数：缺省的构造函数没有参数；第二个构造函数将双精度浮点数赋给复数的实部，虚部置为 0；第三个构造函数将两个双精度浮点数分别赋给复数的实部和虚部。

(3) 定义获取和修改复数的实部和虚部，以及+、−、*、/等运算的成员函数。

(4) 定义重载的流函数来输出一个复数。

【解答】

抽象数据类型通常是指由用户定义，用以表示应用问题的数据模型。抽象数据类型由基本的数据类型构成，并包括一组相关的服务。

```
//在头文件 complex.h 中定义的复数类
#ifndef _complex_h_
#define _complex_h_
#include <iostream.h>

class comlex {
public:
   complex ( ){ Re = Im = 0; }                                  //不带参数的构造函数
   complex ( double r ) { Re = r;    Im = 0; }                  //只置实部的构造函数
   complex ( double r, double i ) { Re = r;    Im = i; }        //分别置实部、虚部的构造函数
   double getReal ( ) { return Re; }                            //取复数实部
   double getImag ( ) { return Im; }                            //取复数虚部
   void setReal ( double r ) { Re = r; }                        //修改复数实部
   void setImag ( double i ) { Im = i; }                        //修改复数虚部
   complex& operator = ( complex& ob) { Re = ob.Re;    Im = ob.Im; }    //复数赋值
   complex& operator + ( complex& ob );                         //重载函数：复数四则运算
   complex& operator – ( complex& ob );
   complex& operator * ( complex& ob );
   complex& operator / ( complex& ob );
   friend ostream& operator << ( ostream& os, complex& c );      //友元函数：重载<<
private:
   double Re, Im;                                               //复数的实部与虚部
};
#endif

//复数类 complex 的相关服务的实现放在 C++源文件 complex.cpp 中
#include <iostream.h>
#include <math.h>
#include “complex.h”
complex& complex :: operator + ( complex & ob ) {
//重载函数：复数加法运算。
      complex * result = new complex ( Re + ob.Re,    Im + ob.Im );
      return *result;
}
complex& complex :: operator – ( complex& ob ) {
//重载函数：复数减法运算
      complex * result = new complex ( Re – ob.Re,    Im – ob.Im );
      return * result;
}
complex& complex :: operator * ( complex& ob ) {
```

```
//重载函数：复数乘法运算
    complex * result =
        new complex ( Re * ob.Re – Im * ob.Im,    Im * ob.Re + Re * ob.Im );
    return *result;
}
complex& complex :: operator / ( complex& ) {
//重载函数：复数除法运算
    double d = ob.Re * ob.Re + ob.Im * ob.Im;
    complex * result = new complex ( ( Re * ob.Re + Im * ob.Im ) / d,
        ( Im * ob. Re – Re * ob.Im ) / d );
    return * result;
}
friend ostream& operator << ( ostream& os, complex & ob ) {
//友元函数：重载<<，将复数 ob 输出到输出流对象 os 中。
    return os << ob.Re << ( ob.Im >= 0.0 ) ? "+" : "–" << fabs ( ob.Im ) << "i";
}
```

1-5 用归纳法证明：

(1) $\sum_{i=1}^{n} i = \frac{n(n+1)}{2}, \quad n \geq 1$

(2) $\sum_{i=1}^{n} i^2 = \frac{n(n+1)(2n+1)}{6}, \quad n \geq 1$

(3) $\sum_{i=0}^{n} x^i = \frac{x^{n+1}-1}{x-1}, \quad x \neq 1, \quad n \geq 0$

【证明】略

1-6 什么是算法？算法的 5 个特性是什么？试根据这些特性解释算法与程序的区别。

【解答】

通常，定义算法为“为解决某一特定任务而规定的一个指令序列。”一个算法应当具有以下特性：

① **有输入**。一个算法必须有 0 个或多个输入。它们是算法开始运算前给予算法的量。这些输入取自于特定的对象的集合。它们可以使用输入语句由外部提供，也可以使用赋值语句在算法内给定。

② **有输出**。一个算法应有一个或多个输出，输出的量是算法计算的结果。

③ **确定性**。算法的每一步都应确切地、无歧义地定义。对于每一种情况，需要执行的动作都应严格地、清晰地规定。

④ **有穷性**。一个算法无论在什么情况下都应在执行有穷步后结束。

⑤ **有效性**。算法中每一条运算都必须是足够基本的。就是说，它们原则上都能精确地执行，甚至人们仅用笔和纸做有限次运算就能完成。

**算法和程序不同，程序可以不满足上述的特性(4)**。例如，一个操作系统在用户未使用前一直处于“等待”的循环中，直到出现新的用户事件为止。这样的系统可以无休止地运行，直到系统停工。

此外，算法是面向功能的，通常用面向过程的方式描述；程序可以用面向对象方式搭建它的框架。

1-7 设 n 为正整数，分析下列各程序段中加下划线的语句的程序步数。

(1)
```
for (int i = 1; i <= n; i++)
  for (int j = 1; j <= n; j++) {
    c[i][j] = 0.0;
    for (int k = 1; k <= n; k++)
      c[i][j] = c[i][j] + a[i][k] * b[k][j];
  }
```

(2)
```
x = 0;  y = 0;
for (int i = 1; i <= n; i++)
  for (int j = 1; j <= i; j++)
      for (int k = 1; k <= j; k++)
        x = x + y;
```

(3)
```
int i = 1,  j = 1;
while (i<=n && j<=n) {
  i = i + 1;  j = j + i;
}
```

(4)
```
int i =1;
do {
  for (int j = 1; j <= n; j++)
    i = i + j;
} while ( i < 100 + n );
```

【解答】

(1) $\sum_{i=1}^{n}\sum_{j=1}^{n}\sum_{k=1}^{n}1 = n^3$

(2) $$\sum_{i=1}^{n}\sum_{j=1}^{i}\sum_{k=1}^{j}1 = \sum_{i=1}^{n}\sum_{j=1}^{i} j = \sum_{i=1}^{n}\left(\frac{i(i+1)}{2}\right) = \frac{1}{2}\sum_{i=1}^{n} i^2 + \frac{1}{2}\sum_{i=1}^{n} i =$$
$$= \frac{1}{2}\frac{n(n+1)(2n+1)}{6} + \frac{1}{2}\frac{n(n+1)}{2} = \frac{n(n+1)(n+2)}{6}$$

(3) i = 1 时，i = 2，j = j + i = 1 + 2 = 2 + 1，
i = 2 时，i = 3，j = j + i = ( 2 + 1 ) + 3 = 3 + 1 + 2，
i = 3 时，i = 4，j = j + i = ( 3 + 1 + 2 ) + 4 = 4 + 1 + 2 + 3，
i = 4 时，i = 5，j = j + i = ( 4 + 1 + 2 + 3 ) + 5 = 5 + 1 + 2 + 3 + 4，
……
i = k 时，i = k + 1，j = j + i = ( k + 1 ) + ( 1 + 2 + 3 + 4 + … + k )，

$\because\ j = (k+1) + \sum_{i=1}^{k} i \le n$

$\therefore (k+1) + \frac{k(k+1)}{2} = \frac{k^2+3k+3}{2} \le n$

解出满足上述不等式的 k 值，即为语句 i = i + 1 的程序步数。

(4) $i = 1$ 时，$i = 1 + \sum_{j=1}^{n} j = 1 + \frac{n(n+1)}{2}$

$i = 2$ 时，$i = \left(1 + \frac{n(n+1)}{2}\right) + \sum_{j=1}^{n} j = \left(1 + \frac{n(n+1)}{2}\right) + \frac{n(n+1)}{2} = 1 + 2\left(\frac{n(n+1)}{2}\right)$

$i = 3$ 时，$i = \left(1 + 2\left(\frac{n(n+1)}{2}\right)\right) + \sum_{j=1}^{n} j = 1 + 3\left(\frac{n(n+1)}{2}\right)$

一般地，

$i = k$ 时，$i = 1 + k\left(\frac{n(n+1)}{2}\right) < 100 + n$

求出满足此不等式的 k 值，即为语句 i = i + j 的程序步数。

1-8 试编写一个函数计算 n!*$2^n$ 的值，结果存放于数组 A[arraySize]的第 n 个数组元素中，0 ≤

n ≤ arraySize。若设计算机中允许的整数的最大值为 maxInt，则当 n > arraySize 或者对于某一个 k (0 ≤ k ≤ n)，使得 $k!*2^k$ > maxInt 时，应按出错处理。可有如下三种不同的出错处理方式：

(1) 用 **cerr<<**及 **exit** (1)语句来终止执行并报告错误；

(2) 用返回整数函数值 0, 1 来实现算法，以区别是正常返回还是错误返回；

(3) 在函数的参数表设置一个引用型的整型变量来区别是正常返回还是某种错误返回。

试讨论这三种方法各自的优缺点，并以你认为是最好的方式实现它。

【解答】

```
#include "iostream.h"
#define arraySize 100
#define MaxInt 0x7fffffff

int calc ( int T[ ], int n ) {
    int i, value = 1;
    if ( n != 0 ) {
        int edge = MaxInt / n / 2;
        for ( i = 1; i < n; i++ ) {
            value *= i*2;
            if ( value > edge ) return 0;
        }
        value *= n * 2;
    }
    T[n] = value;
    cout << "A[" << n << "]=" << T[n] << endl;
    return 1;
}

void main ( ) {
    int A[arraySize];
    int i;
    for ( i = 0; i < arraySize; i++ )
        if ( !calc ( A, i ) ) {
            cout << "failed at " << i << " ." << endl;
            break;
        }
}
```

1–9 (1) 在下面所给函数的适当地方插入计算 count 的语句：

```
void d (ArrayElement x[ ], int n ) {
    int i = 1;
    do {
        x[i] += 2;    i +=2;
    } while (i <= n );
    i = 1;
    while ( i <= (n/2) ) {
```

```
            x[i] += x[i+1];   i++;
        }
    }
```

(2) 将由(1)所得到的程序化简。使得化简后的程序与化简前的程序具有相同的 count 值。

(3) 程序执行结束时的 count 值是多少？

(4) 使用执行频度的方法计算这个程序的程序步数，画出程序步数统计表。

【解答】

(1) 在适当的地方插入计算 count 语句

```
void d ( ArrayElement x [ ],  int n ) {
    int i = 1;
    count ++;
    do {
        x[i] += 2;   count ++;
        i += 2;   count ++;
        count ++;                          //针对 while 语句
    } while ( i <= n );
    i = 1;
    count ++;
    while ( i <= ( n / 2 ) ) {
        count ++;                          //针对 while 语句
        x[i] += x[i+1];
        count ++;
        i ++;
        count ++;
    }
    count ++;                              //针对最后一次 while 语句
}
```

(2) 将由(1)所得到的程序化简。化简后的程序与原来的程序有相同的 count 值：

```
    void d ( ArrayElement x [ ],  int n ) {
    int i = 1;
    do {
        count += 3;   i += 2;
    } while ( i <= n );
    i = 1;
    while ( i <= ( n / 2 ) ) {
        count += 3;   i ++;
    }
    count += 3;
}
```

(3) 程序执行结束后的 count 值为 3n + 3。

当 n 为偶数时，count = 3 * ( n / 2 ) + 3 * ( n / 2 ) + 3 = 3 * n + 3

当 n 为奇数时，count = 3 * ( ( n + 1 ) / 2 ) + 3 * ( ( n – 1 ) / 2 ) + 3 = 3 * n + 3

(4) 使用执行频度的方法计算程序的执行步数，画出程序步数统计表：

| 行 号 | 程 序 语 句 | 一次执行步数 | 执行频度 | 程序步数 |
|---|---|---|---|---|
| 1 | **void** d ( ArrayElement x [ ], **int** n ) **{** | 0 | 1 | 0 |
| 2 | **int** i = 1**;** | 1 | 1 | 1 |
| 3 | **do {** | 0 | ⌊(n+1)/2⌋ | 0 |
| 4 | x[i] += 2**;** | 1 | ⌊(n+1)/2⌋ | ⌊(n+1)/2⌋ |
| 5 | i += 2**;** | 1 | ⌊(n+1)/2⌋ | ⌊(n+1)/2⌋ |
| 6 | **} while** ( i <= n )**;** | 1 | ⌊(n+1)/2⌋ | ⌊(n+1)/2⌋ |
| 7 | i = 1**;** | 1 | 1 | 1 |
| 8 | **while** ( i <= ( n / 2 ) ) **{** | 1 | ⌊n/2+1⌋ | ⌊n/2+1⌋ |
| 9 | x[i] += x[i+1]**;** | 1 | ⌊n/2⌋ | ⌊n/2⌋ |
| 10 | i ++**;** | 1 | ⌊n/2⌋ | ⌊n/2⌋ |
| 11 | **}** | 0 | ⌊n/2⌋ | 0 |
| 12 | **}** | 0 | 1 | 0 |
| | | | ( n ≠ 0 ) | 3n + 3 |

1-10 设有 3 个值大小不同的整数 a、b 和 c，试求

(1) 其中值最大的整数；

(2) 其中值最小的整数；

(3) 其中位于中间值的整数。

【解答】

(1) 求 3 个整数中的最大整数的函数

【方案 1】

```
int max ( int a, int b, int c ) {
   int m = a;
   if ( b > m ) m = b;
   if ( c > m ) m = c;
   return m;
}
```

【方案 2】(此程序可修改循环终止变量扩大到 n 个整数)

```
int max ( int a, int b, int c ) {
   int data[3] = { a, b, c };
   int m = 0;                          //开始时假定 data[0]最大
   for ( int i = 1; i < 3; i++ )       //与其他整数逐个比较
      if ( data[i] > data[m] ) m = i;  //m 记录新的最大者
   return data[m];
}
```

(2) 求 3 个整数中的最小整数的函数

可将上面求最大整数的函数稍做修改，“>”改为“<”，可得求最小整数函数。

【方案 1】

```
int min ( int a, int b, int c ) {
   int m = a;
   if ( b < m ) m = b;
   if ( c < m ) m = c;
   return m;
}
```

【方案 2】(此程序可修改循环终止变量扩大到 n 个整数)

```
int max ( int a, int b, int c ) {
   int data[3] = { a, b, c };
   int m = 0;                                  //开始时假定 data[0]最小
   for ( int i = 1; i < 3; i++ )               //与其他整数逐个比较
      if ( data[i] < data[m] ) m = i;          //m 记录新的最小者
   return data[m];
}
```

(3) 求 3 个整数中具有中间值的整数

可将上面求最大整数的函数稍做修改，“>”改为“<”，可得求最小整数函数。

【方案 1】

```
int mid ( int a, int b, int c ) {
   int m1 = a, m2;
   if ( b < m1 ) { m2 = m1;    m1 = b; }
   else m2 = b;
   if ( c < m1 ) { m2 = m1;    m1 = c; }
   else if ( c < m2 ) { m2 = c; }
   return m2;
}
```

【方案 2】(此程序可修改循环终止变量扩大到 n 个整数寻求次小元素)

```
int mid ( int a, int b, int c ) {
   int data[3] = { a, b, c };
   int m1 = 0, m2 = −1;                                  //m1 指示最小整数, m2 指示次小整数
   for ( int i = 1; i < 3; i++ )                         //与其他整数逐个比较
      if ( data[i] < data[m1] ) { m2 = m1;    m1 = i; }      //原来最小变为次小, m1 指示新的最小
      else if ( m2 == −1 || data[i] < data[m2] ) m2 = i;  //m2 记录新的次小者
   return data[m2];
}
```

# 第二章 数组

作为抽象数据类型数组的类声明。

```
#include <iostream.h>                    //在头文件“array.h”中
#include <stdlib.h>
const int DefaultSize = 30;

 template <class Type> class Array {
//数组是数据类型相同的 n（size）个元素的一个集合，下标范围从 0 到 n−1。对数组中元素
//可按下标所指示位置直接访问。
private:
   Type *elements;                                    //数组
   int ArraySize;                                   //元素个数
public:
   Array ( int Size = DefaultSize );                  //构造函数
   Array ( const Array<Type> & x );                   //复制构造函数
   ~Array ( ) { delete [ ] elements; }                //析构函数
   Array<Type> & operator = ( const Array<Type> & A );  //数组整体赋值 (复制)
   Type& operator [ ] ( int i );                      //按下标访问数组元素
   int Length ( ) const { return ArraySize; }         //取数组长度
   void ReSize ( int sz );                            //修改数组长度
}
```

顺序表的类定义

```
#include < iostream.h>                        //定义在头文件“seqlist.h”中
 #include <stdlib.h>
 template <class Type> class SeqList {
private:
    Type *data;                              //顺序表的存放数组
    int MaxSize;                           //顺序表的最大可容纳项数
    int last;                              //顺序表当前已存表项的最后位置
    int current;                             //顺序表的当前指针（最近处理的表项）
 public:
    SeqList ( int MaxSize );               //构造函数
    ~SeqList ( ) { delete [ ] data; }      //析构函数
    int Length ( ) const { return last+1; }          //计算表长度
    int Find ( Type& x ) const;                      //定位函数：找 x 在表中位置，置为当前表项
    int IsIn ( Type& x );                            //判断 x 是否在表中，不置为当前表项
    Type *GetData ( ) { return current == −1？ NULL : data[current]; }    //取当前表项的值
    int Insert ( Type& x );                          //插入 x 在表中当前表项之后，置为当前表项
    int Append ( Type& x );                //追加 x 到表尾，置为当前表项
    Type * Remove ( Type& x );                       //删除 x，置下一表项为当前表项
    Type * First ( );                                //取表中第一个表项的值，置为当前表项
    Type * Next ( ) { return current < last ? &data[++current] : NULL; }
```

```
                                        //取当前表项的后继表项的值，置为当前表项
    Type * Prior ( ) { return current > 0 ? &data[--current] : NULL; }
                                        //取当前表项的前驱表项的值，置为当前表项
    int IsEmpty ( ) { return last == -1; }          //判断顺序表空否，空则返回 1；否则返回 0
    int IsFull ( ) { return last == MaxSize-1; }    //判断顺序表满否，满则返回 1；否则返回 0
}
```

2-1 设 n 个人围坐在一个圆桌周围，现在从第 s 个人开始报数，数到第 m 个人，让他出局；然后从出局的下一个人重新开始报数，数到第 m 个人，再让他出局，……，如此反复直到所有的人全部出局为止。下面要解决的 Josephus 问题是：对于任意给定的 n，s 和 m，求出这 n 个人的出局序列。请以 n = 9，s = 1，m = 5 为例，人工模拟 Josephus 的求解过程以求得问题的解。

【解答】

出局人的顺序为 5, 1, 7, 4, 3, 6, 9, 2, 8。

2-2 试编写一个求解 Josephus 问题的函数。用整数序列 1，2，3，……，n 表示顺序围坐在圆桌周围的人，并采用数组表示作为求解过程中使用的数据结构。然后使用 n = 9，s = 1，m = 5，以及 n = 9，s = 1，m = 0，或者 n = 9，s = 1，m = 10 作为输入数据，检查你的程序的正确性和健壮性。最后分析所完成算法的时间复杂度。

【解答】函数源程序清单如下：

```
void Josephus( int A[ ], int n, s, m ) {
    int i, j, k, tmp;
    if ( m == 0 ) {
        cout << "m = 0 是无效的参数！" << endl;
        return;
    }
    for ( i = 0; i < n; i++ ) A[i] = i + 1;          /*初始化，执行 n 次*/
    i = s - 1;                                       /*报名起始位置*/
    for ( k = n; k > 1; i-- ) {                      /*逐个出局，执行 n-1 次*/
        if ( i == k ) i = 0;
        i = ( i + m - 1 ) % k;                       /*寻找出局位置*/
        if ( i != k-1 ) {
            tmp = A[i];                              /*出局者交换到第 k-1 位置*/
            for ( j = i; j < k-1; j++ ) A[j] = A[j+1];
            A[k-1] = tmp;
        }
    }
    for ( k = 0; k < n / 2; k++ ) {                  /*全部逆置，得到出局序列*/
        tmp = A[k]; A[k] = A[n-k+1]; A[n-k+1] = tmp;
    }
}
```

例：n = 9, s = 1, m = 5

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | |
|---|---|---|---|---|---|---|---|---|---|---|
| k = 9 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 第 5 人出局, i = 4 |
| k = 8 | 1 | 2 | 3 | 4 | 6 | 7 | 8 | 9 | 5 | 第 1 人出局, i = 0 |

| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| k = 7 | 2 | 3 | 4 | 6 | 7 | 8 | 9 | 1 | 5 | 第 7 人出局, i = 4 |
| k = 6 | 2 | 3 | 4 | 6 | 8 | 9 | 7 | 1 | 5 | 第 4 人出局, i = 2 |
| k = 5 | 2 | 3 | 6 | 8 | 9 | 4 | 7 | 1 | 5 | 第 3 人出局, i = 1 |
| k = 4 | 2 | 6 | 8 | 9 | 3 | 4 | 7 | 1 | 5 | 第 6 人出局, i = 1 |
| k = 3 | 2 | 8 | 9 | 6 | 3 | 4 | 7 | 1 | 5 | 第 9 人出局, i = 2 |
| k = 2 | 2 | 8 | 9 | 6 | 3 | 4 | 7 | 1 | 5 | 第 2 人出局, i = 0 |
| | 8 | 2 | 9 | 6 | 3 | 4 | 7 | 1 | 5 | 第 8 人出局, i = 0 |
| 逆置 | 5 | 1 | 7 | 4 | 3 | 6 | 9 | 2 | 8 | 最终出局顺序 |

例：n = 9, s = 1, m = 0

报错信息　m = 0 是无效的参数！

例：n = 9, s = 1, m = 10

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | |
|---|---|---|---|---|---|---|---|---|---|---|
| k = 9 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 第 1 人出局, i = 0 |
| k = 8 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 1 | 第 3 人出局, i = 1 |
| k = 7 | 2 | 4 | 5 | 6 | 7 | 8 | 9 | 3 | 1 | 第 6 人出局, i = 3 |
| k = 6 | 2 | 4 | 5 | 7 | 8 | 9 | 6 | 3 | 1 | 第 2 人出局, i = 0 |
| k = 5 | 4 | 5 | 7 | 8 | 9 | 2 | 6 | 3 | 1 | 第 9 人出局, i = 4 |
| k = 4 | 4 | 5 | 7 | 8 | 9 | 2 | 6 | 3 | 1 | 第 5 人出局, i = 1 |
| k = 3 | 4 | 7 | 8 | 5 | 9 | 2 | 6 | 3 | 1 | 第 7 人出局, i = 1 |
| k = 2 | 4 | 8 | 7 | 5 | 9 | 2 | 6 | 3 | 1 | 第 4 人出局, i = 0 |
| | 8 | 4 | 7 | 5 | 9 | 2 | 6 | 3 | 1 | 第 8 人出局, i = 0 |
| 逆置 | 1 | 3 | 6 | 2 | 9 | 5 | 7 | 4 | 8 | 最终出局顺序 |

当 m = 1 时，时间代价最大。达到( n−1 ) + ( n−2 ) + ····· + 1 = n(n−1)/2 ≈ $O(n^2)$。

2−3 设有一个线性表 $(e_0, e_1, \cdots, e_{n-2}, e_{n-1})$ 存放在一个一维数组 A[arraySize]中的前 n 个数组元素位置。请编写一个函数将这个线性表原地逆置，即将数组的前 n 个原址内容置换为 $(e_{n-1}, e_{n-2}, \cdots, e_1, e_0)$。

【解答】

```
template<class Type> void inverse ( Type A[ ], int n ) {
    Type tmp;
    for ( int i = 0; i <= ( n−1 ) / 2; i++ ) {
        tmp = A[i];   A[i] = A[n−i−1];   A[n−i−1] = tmp;
    }
}
```

2−4 假定数组 A[arraySize]中有多个零元素，试写出一个函数，将 A 中所有的非零元素依次移到数组 A 的前端 A[i]（0≤ i ≤ arraySize）。

【解答】

因为数组是一种直接存取的数据结构，在数组中元素不是像顺序表那样集中存放于表的前端，而是根据元素下标直接存放于数组某个位置，所以将非零元素前移时必须检测整个数组空间，并将后面变成零元素的空间清零。函数中设置一个辅助指针 free，指示当前可存放的位置，初值为 0。

```
template<class Type> void Array<Type> :: compact( ) {
  int free = 0;
```

```
    for ( int i = 0; i < ArraySize; i++ )                    //检测整个数组
        if ( elements[I] != 0 )                              //发现非零元素
            { elements[free] = elements[i];   free++;   elements[i] = 0; } //前移
}
```

2-5 顺序表的插入和删除要求仍然保持各个元素原来的次序。设在等概率情形下，对有 127 个元素的顺序表进行插入，平均需要移动多少个元素？删除一个元素，又平均需要移动多少个元素?

【解答】

若设顺序表中已有 n = last+1 个元素，last 是顺序表的数据成员，表明最后表项的位置。又设插入或删除表中各个元素的概率相等，则在插入时因有 n+1 个插入位置(可以在表中最后一个表项后面追加)，每个元素位置插入的概率为 1/(n+1)，但在删除时只能在已有 n 个表项范围内删除，所以每个元素位置删除的概率为 1/n。

插入时平均移动元素个数 AMN(Averagy Moving Number )为

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

删除时平均移动元素个数 AMN 为

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

2-6 若矩阵 $A_{m\times n}$ 中的某一元素 A[i][j]是第 i 行中的最小值，同时又是第 j 列中的最大值，则称此元素为该矩阵的一个鞍点。假设以二维数组存放矩阵，试编写一个函数，确定鞍点在数组中的位置（若鞍点存在时），并分析该函数的时间复杂度。

【解答】

```
int minmax ( int A[ ][ ], const int m, const int n ) {
//在二维数组 A[m][n]中求所有鞍点，它们满足在行中最小同时在列中最大
    int *row = new int[m];   int * col = new int[n];
    int i, j;
    for ( i = 0; i < m; i++ ) {                              //在各行中选最小数组元素，存于 row[i]
        row[i] = A[i][0];
        for ( j = 1; j < n; j++ )
            if ( A[i][j] < row[i] ) row[i] = A[i][j];
    }
    for ( j = 0; j < n; j++ ) {                              //在各列中选最大数组元素，存于 col[j]
        col[i] = A[0][j];
        for ( i = 1; i < m; i++ )
            if ( A[i][j] > col[j] ) col[j] = A[i][j];
    }
    for ( i = 0; i < m; i++ ) {                              //检测矩阵，寻找鞍点并输出其位置
        for ( j = 0; j < n; j++ )
            if ( row[i] == col[j] ) cout << "The saddle point is : (" << i << ", " << j << ")" << endl;
    delete [ ] row;   delete [ ] col;
}
```

此算法有 3 个并列二重循环，其时间复杂度为 O(m×n)。

2-7　设有一个二维数组 A[m][n]，假设 A[0][0]存放位置在 $644_{(10)}$，A[2][2]存放位置在 $676_{(10)}$，每个元素占一个空间，问 $A[3][3]_{(10)}$存放在什么位置？脚注$_{(10)}$表示用 10 进制表示。

【解答】

设数组元素 A[i][j]存放在起始地址为 Loc ( i, j ) 的存储单元中。

∵ Loc ( 2, 2 ) = Loc ( 0, 0 ) + 2 * n + 2 = 644 + 2 * n + 2 = 676.

∴ n = ( 676 − 2 − 644 ) / 2 = 15

∴ Loc ( 3, 3 ) = Loc ( 0, 0 ) + 3 * 15 + 3 = 644 + 45 + 3 = 692.

2-8　利用顺序表的操作，实现以下的函数。

(1) 从顺序表中删除具有最小值的元素并由函数返回被删元素的值。空出的位置由最后一个元素填补，若顺序表为空则显示出错信息并退出运行。

(2) 从顺序表中删除第 i 个元素并由函数返回被删元素的值。如果 i 不合理或顺序表为空则显示出错信息并退出运行。

(3) 向顺序表中第 i 个位置插入一个新的元素 x。如果 i 不合理则显示出错信息并退出运行。

(4) 从顺序表中删除具有给定值 x 的所有元素。

(5) 从顺序表中删除其值在给定值 s 与 t 之间（要求 s 小于 t）的所有元素，如果 s 或 t 不合理或顺序表为空则显示出错信息并退出运行。

(6) 从有序顺序表中删除其值在给定值 s 与 t 之间（要求 s 小于 t）的所有元素，如果 s 或 t 不合理或顺序表为空则显示出错信息并退出运行。

(7) 将两个有序顺序表合并成一个新的有序顺序表并由函数返回结果顺序表。

(8) 从顺序表中删除所有其值重复的元素，使表中所有元素的值均不相同。

【解答】

(1) 实现删除具有最小值元素的函数如下：

```
template<Type> Type SeqList<Type> :: DelMin ( ) {
    if ( last == -1 )                                    //表空, 中止操作返回
            { cerr << " List is Empty! " << endl;   exit(1); }
    int m = 0;                                           //假定 0 号元素的值最小
        for ( int i = 1; i <= last; i++ ) {              //循环, 寻找具有最小值的元素
            if ( data[i] < data[m] ) m = i;              //让 m 指向当前具最小值的元素
        Type temp = data[m];
    data[m] = data[last];   last--;                      //空出位置由最后元素填补, 表最后元素位置减 1
    return temp;
}
```

(2) 实现删除第 i 个元素的函数如下(设第 i 个元素在 data[i], i=0,1,…,last)：

```
template<Type> Type SeqList<Type> :: DelNo#i ( int i ) {
    if ( last == -1 ||  i < 0 ||  i > last )             //表空, 或 i 不合理, 中止操作返回
            { cerr << " List is Empty or Parameter is out range! " << endl;   exit(1); }
        Type temp = data[i];                                 //暂存第 i 个元素的值
    for ( int j = i; j < last; j++ )                     //空出位置由后续元素顺次填补
        data[j] = data[j+1];
    last--;                                              //表最后元素位置减 1
    return temp;
}
```

(3) 实现向第 i 个位置插入一个新的元素 x 的函数如下(设第 i 个元素在 data[i],

i=0,1,…,last)：

```
template<Type> void SeqList<Type> :: InsNo#i ( int i, Type& x ) {
    if ( last == MaxSize−1 || i < 0 || i > last+1 )        //表满或参数 i 不合理，中止操作返回
            { cerr << " List is Full or Parameter is out range! " << endl;   exit(1); }
    for ( int j = last; j >= i; j−− )                      //空出位置以便插入，若 i=last+1，此循环不做
        data[j+1] = data[j];
    data[i] = x;                                           //插入
    last++;                                                //表最后元素位置加 1
}
```

(4) 从顺序表中删除具有给定值 x 的所有元素。

```
template<Type> void SeqList<Type> :: DelValue ( Type& x ) {
    int i = 0, j;
        while ( i <= last )                                //循环，寻找具有值 x 的元素并删除它
            if ( data[i] == x ) {                          //删除具有值 x 的元素，后续元素前移
            for ( j = i; j < last; j++ ) data[j] = data[j+1];
            last−−;                                        //表最后元素位置减 1
        }
        else i++;
}
```

(5) 实现删除其值在给定值 s 与 t 之间（要求 s 小于 t）的所有元素的函数如下：

```
template<Type> void SeqList<Type> :: DelNo#sto#t ( Type& s, Type& t ) {
    if ( last == −1 || s >= t )
        { cerr << "List is empty or parameters are illegal!" << endl;   exit(1); }
    int i = 0, j;
        while ( i <= last )                                //循环，寻找具有值 x 的元素并删除它
            if ( data[i] >= s && data[i] <= t ) {          //删除满足条件的元素，后续元素前移
            for ( j = i; j < last; j++ ) data[j] = data[j+1];
            last−−;                                        //表最后元素位置减 1
        }
        else i++;
}
```

(6) 实现从有序顺序表中删除其值在给定值 s 与 t 之间的所有元素的函数如下：

```
template<Type> void SeqList<Type> :: DelNo#sto#t1 ( Type& s, Type& t ) {
    if ( last == −1 || s >= t )
        { cerr << "List is empty or parameters are illegal!" << endl;   exit(1); }
        for ( int i = 0; i <= last; i++ )                  //循环，寻找值 ≥s 的第一个元素
            if ( data[i] >= s ) break;                     //退出循环时, i 指向该元素
    if ( i <= last ) {
        for ( int j = 1; i + j <= last; j++ )              //循环，寻找值 > t 的第一个元素
            if ( data[i+j] > t ) break;                    //退出循环时, i+j 指向该元素
        for ( int k = i+j; k <= last; k++ )                //删除满足条件的元素，后续元素前移
            data[k−j] = data[k];
        last−= j;                                          //表最后元素位置减 j
    }
```

```
}
```

(7) 实现将两个有序顺序表合并成一个新的有序顺序表的函数如下：

```
template<Type> SeqList<Type>& SeqList<Type> :: Merge ( SeqList<Type>& A, SeqList<Type>& B )
{
//合并有序顺序表 A 与 B 成为一个新的有序顺序表并由函数返回
    if ( A.Length() + B.Length() > MaxSize )
       { cerr << "The summary of The length of Lists is out MaxSize!" << endl;    exit(1); }
    Type value1 = A.Fist ( ), value2 = B.Fisrt ( );
    int i = 0, j = 0, k = 0;
    while ( i < A.length ( ) && j < B.length ( ) ) {          //循环，两两比较，小者存入结果表
              if ( value1 <= value2 )
           { data[k] = value1;    value1 = A.Next ( );    i++; }
        else { data[k] = value2;    value2 = B.Next ( );    j++; }
        k++;
    }
    while ( i < A.Length ( ) )                                //当 A 表未检测完，继续向结果表传送
        { data[k] = value1;    value1 = A.Next ( );    i++;    k++; }
    while ( j < B.Length ( ) )                                //当 B 表未检测完，继续向结果表传送
        { data[k] = value2;    value2 = B.Next ( );    j++;    k++; }
    last = k – 1;
    return *this;
}
```

(8) 实现从表中删除所有其值重复的元素的函数如下：

```
template<Type> void SeqList<Type> :: DelDouble ( ) {
    if ( last == −1 )
       { cerr << "List is empty!" << endl;    exit(1); }
         int i = 0, j, k;    Type temp;
    while ( i <= last ) {                              //循环检测
              j = i + 1;    temp = data[i];
        while ( j <= last ) {                          //对于每一个 i，重复检测一遍后续元素
            if ( temp == data[j] ) {                   //如果相等，后续元素前移
                for ( k = j+1; k <= last; k++ ) data[k−1] = data[k];
                last−−;                                    //表最后元素位置减 1
            }
            else j++;
        }
        i++;                                           //检测完 data[i]，检测下一个
    }
}
```

2−9 设有一个 n×n 的对称矩阵 A，如图(a)所示。为了节约存储，可以只存对角线及对角线以上的元素，或者只存对角线或对角线以下的元素。前者称为上三角矩阵，后者称为下三角矩阵。我们把它们按行存放于一个一维数组 B 中，如图(b)和图(c)所示。并称之为对称矩阵 A 的压缩存储方式。试问：

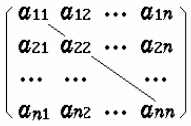
(1) 存放对称矩阵 A 上三角部分或下三角部分的一维数组 B 有多少元素？

(2) 若在一维数组 B 中从 0 号位置开始存放，则如图(a)所示的对称矩阵中的任一元素 $a_{ij}$ 在只存上三角部分的情形下(图(b))应存于一维数组的什么下标位置？给出计算公式。

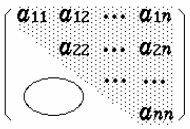
(3) 若在一维数组 B 中从 0 号位置开始存放，则如图(a)所示的对称矩阵中的任一元素 $a_{ij}$ 在只存下三角部分的情形下(图(c))应存于一维数组的什么下标位置？给出计算公式。

【解答】

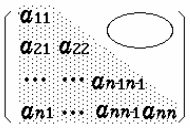
(1) 数组 B 共有 n + ( n−1 ) +…… + 1= n * ( n+1 ) / 2 个元素。



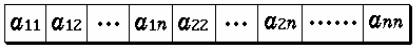
（a）对称矩阵

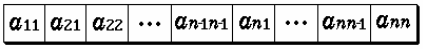


（b）只存上三角部分



（c）只存下三角部分





(2) 只存上三角部分时，若 i ≤ j，则数组元素 A[i][j]前面有 i−1 行（1~i−1，第 0 行第 0 列不算)，第 1 行有 n 个元素，第 2 行有 n−1 个元素，……，第 i−1 行有 n−i+2 个元素。在第 i 行中，从对角线算起，第 j 号元素排在第 j−i+1 个元素位置（从 1 开始），因此，数组元素 A[i][j]在数组 B 中的存放位置为

n + (n−1) + (n−2) + …… + (n−i+2) + j−i+1

= (2n−i+2) * (i−1) / 2 + j−i+1

= (2n−i) * (i−1) / 2 + j

若 i > j，数组元素 A[i][j]在数组 B 中没有存放，可以找它的对称元素 A[j][i]。在数组 B 的第 (2n−j) * (j−1) / 2 + i 位置中找到。

如果第 0 行第 0 列也计入，数组 B 从 0 号位置开始存放，则数组元素 A[i][j]在数组 B 中的存放位置可以改为

当 i ≤ j 时，= (2n−i+1) * i / 2 + j − i = ( 2n − i − 1 ) * i / 2 + j

当 i > j 时，= (2n − j − 1) * j / 2 + i

(3) 只存下三角部分时，若 i ≥ j，则数组元素 A[i][j]前面有 i−1 行（1~i−1，第 0 行第 0 列不算)，第 1 行有 1 个元素，第 2 行有 2 个元素，……，第 i−1 行有 i−1 个元素。在第 i 行中，第 j 号元素排在第 j 个元素位置，因此，数组元素 A[i][j]在数组 B 中的存放位置为

1 + 2 + …… + (i−1) + j = ( i−1)*i / 2 + j

若 i < j，数组元素 A[i][j]在数组 B 中没有存放，可以找它的对称元素 A[j][i]。在数组 B 的第 (j−1)*j / 2 + i 位置中找到。

如果第 0 行第 0 列也计入，数组 B 从 0 号位置开始存放，则数组元素 A[i][j]在数组 B 中的存放位置可以改为

当 i ≥ j 时，= i*(i+1) / 2 + j

当 i < j 时，= j*(j+1) / 2 + i

2−10 设 A 和 B 均为下三角矩阵，每一个都有 n 行。因此在下三角区域中各有 n(n+1)/2 个元素。另设有一个二维数组 C，它有 n 行 n+1 列。试设计一个方案，将两个矩阵 A 和 B 中的下三角区域元素存放于同一个 C 中。要求将 A 的下三角区域中的元素存放于 C 的下三角区域中，B 的下三角区域中的元素转置后存放于 C 的上三角区域中。并给出计算 A 的矩阵元素 $a_{ij}$ 和 B 的矩阵元素 $b_{ij}$ 在 C 中的存放位置下标的公式。

【解答】

$$A=\begin{pmatrix} a_{00} & & & \\ a_{10} & a_{11} & & \\ \cdots & \cdots & \ddots & \\ a_{n-10} & a_{n-11} & \cdots & a_{n-1n-1} \end{pmatrix} \qquad B=\begin{pmatrix} b_{00} & & & \\ b_{10} & b_{11} & & \\ \cdots & \cdots & \ddots & \\ b_{n-10} & b_{n-11} & \cdots & b_{n-1n-1} \end{pmatrix}$$

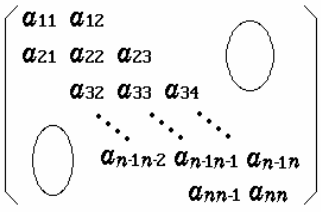
$$C=\begin{pmatrix} a_{00} & b_{00} & b_{10} & \cdots & b_{n-20} & b_{n-10} \\ a_{10} & a_{11} & b_{11} & \cdots & b_{n-21} & b_{n-11} \\ a_{20} & a_{21} & a_{22} & & b_{n-22} & b_{n-12} \\ \cdots & \cdots & \cdots & & \cdots & \cdots \\ a_{n-10} & a_{n-11} & a_{n-12} & \cdots & a_{n-1n-1} & b_{n-1n-1} \end{pmatrix}$$

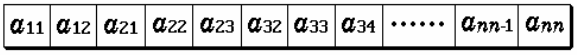
计算公式

$$A[i][\ j\ ]=\begin{cases} C[\ i\ ][\ j\ ], & \text{当 } i \geq j \text{ 时} \\ C[\ j\ ][\ i\ ], & \text{当 } i < j \text{ 时} \end{cases}$$

$$B[i][\ j\ ]=\begin{cases} C[\ j\ ][i+1], & \text{当 } i \geq j \text{ 时} \\ C[i][\ j+1], & \text{当 } i < j \text{ 时} \end{cases}$$

2−11 在实际应用中经常遇到的稀疏矩阵是三对角矩阵，如右图所示。在该矩阵中除主对角线及在主对角线上下最临近的两条对角线上的元素外，所有其它元素均为 0。现在要将三对角矩阵 A 中三条对角线上的元素按行存放在一维数组 B 中，且 $a_{11}$ 存放于 B[0]。试给出计算 A 在三条对角线上的元素 $a_{ij}$（$1 \le i \le n$, $i-1 \le j \le i+1$）在一维数组 B 中的存放位置的计算公式。





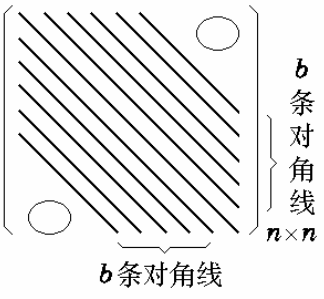

【解答】

在 B 中的存放顺序为 [ $a_{11}, a_{12}, a_{21}, a_{22}, a_{23}, a_{32}, a_{33}, a_{34}, \cdots, a_{n\,n-1}, a_{nn}$ ]，总共有 3n−2 个非零元素。元素 $a_{ij}$ 在第 i 行，它前面有 3(i−1)−1 个非零元素，而在本行中第 j 列前面有 j−i+1 个，所以元素 $a_{ij}$ 在 B 中位置为 2*i+j−3。

2−12 设带状矩阵是 n×n 阶的方阵，其中所有的非零元素都在由主对角线及主对角线上下各 b 条对角线构成的带状区域内，其它都为零元素。试问：

(1) 该带状矩阵中有多少个非零元素？

(2) 若用一个一维数组 B 按行顺序存放各行的非零元素，且设 $a_{11}$ 存放在 B[0]中，请给出一个公式，计算任一非零元素 $a_{ij}$ 在一维数组 B 中的存放位置。




【解答】

(1) 主对角线包含 n 个非零元素，其上下各有一条包含 n−1 个非零元素的次对角线，再向外，由各有一条包含 n−2 个非零元素的次对角线，……，最外层上下各有一条包含 n−b 个非零元素的次对角线。则总共的非零元素个数有

$$n + 2(n-1) + 2(n-2) + \cdots + 2(n-b) = n + 2( (n-1) + (n-2) + \cdots + (n-b) )$$

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

(2) 在用一个一维数组 B 按行顺序存放各行的非零元素时，若设 b≤n/2，则可按各行非零元素个数变化情况，分 3 种情况讨论。

① 当 1≤i≤b+1 时，矩阵第 1 行有 b+1 个元素，第 2 行有 b+2 个元素，第 3 行有 b+3 个元素，……，第 i 行存有 b+i 个元素，因此，数组元素 A[i][j]在 B[ ]中位置分析如下：

第 i 行(i≥1)前面有 i−1 行，元素个数为 (b+1)+(b+2)+…+(b+i−1) = (i−1)*b+i*(i−1)/2，在第 i 行第 j 列(j≥1)前面有 j−1 个元素，则数组元素 A[i][j]在 B[ ]中位置为

$$(i-1)*b+\frac{i*(i-1)}{2}+j-1$$

② 当 b+1<i≤n−b+1 时，各行都有 2b+1 个元素。因为数组 A[ ][ ]前 b 行共有 b*b+(b+1)*b/2 = b*(3*b+1)/2 个元素，所以数组元素 A[i][j]在 B[ ]中位置为

$$\frac{b*(3*b+1)}{2}+(i-b-1)*(2*b+1)+j-i+b$$

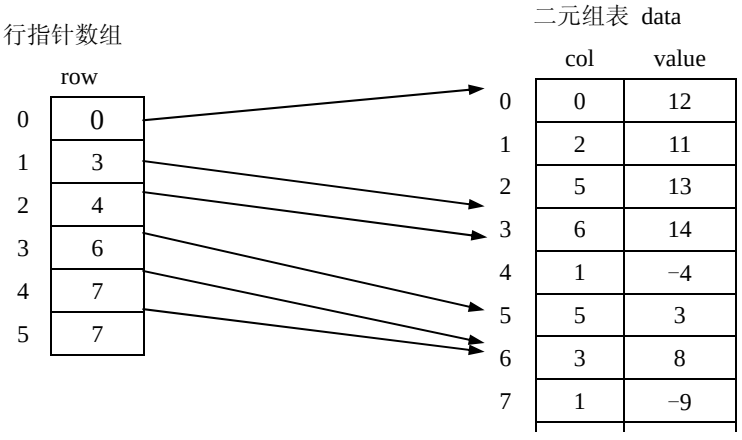
③ 当 n−b+1<i≤n 时，各行元素个数逐步减少。当 i=n−b+1 时有 2b 个非零元素，当 i=n−b+2 时有 2b−1 个非零元素，当 i=n−b+3 时有 2b−2 个非零元素，…，当 i=n 时有 b+1 个非零元素。因为前面 n−b 行总共有 b*(3*b+1)/2+(n−2*b)*(2*b+1)个非零元素，所以在最后各行数组元素 A[i][j]在 B[ ]中位置为

$$\frac{b*(3*b+1)}{2}+(n-2*b)*(2*b+1)+\frac{(i-n+b-1)*(4*b-i+n-b+2)}{2}+j-i+b$$

2−13 稀疏矩阵的三元组表可以用带行指针数组的二元组表代替。稀疏矩阵有多少行，在行指针数组中就有多少个元素：第 i 个元素的数组下标 i 代表矩阵的第 i 行，元素的内容即为稀疏矩阵第 i 行的第一个非零元素在二元组表中的存放位置。二元组表中每个二元组只记录非零元素的列号和元素值，且各二元组按行号递增的顺序排列。试对右图所示的稀疏矩阵，分别建立它的三元组表和带行指针数组的二元组表。

$$\begin{pmatrix} 12 & 0 & 11 & 0 & 0 & 13 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 14 \\ 0 & -4 & 0 & 0 & 0 & 3 & 0 \\ 0 & 0 & 0 & 8 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -9 & 0 & 0 & 2 & 0 & 0 \end{pmatrix}$$

【解答】




2−14 字符串的替换操作 replace (String **&**s, String **&**t, String **&**v)是指：若 t 是 s 的子串，则用串 v 替换串 t 在串 s 中的所有出现；若 t 不是 s 的子串，则串 s 不变。例如，若串 s 为“aabbabcbaabaaacbab”，串 t 为“bab”，串 v 为“abdc”，则执行 replace 操作后，串 s 中的结果为“aababdccbaabaaacabdc”。试利用字符串的基本运算实现这个替换操作。

【解答】

```
String & String :: Replace ( String & t, String &v) {
  if ( ( int id = Find ( t ) ) == −1 )                    //没有找到，当前字符串不改，返回
     { cout << "The (replace) operation failed." << endl;   return *this; }
  String temp( ch );                                      //用当前串建立一个空的临时字符串
  ch[0] = '\0';   curLen = 0;                             //当前串作为结果串，初始为空
  int j, k = 0, l;                                        //存放结果串的指针
  while ( id != −1 ) {
      for ( j = 0; j < id; j++) ch[k++] = temp.ch[j]; //摘取 temp.ch 中匹配位置
      if ( curLen+ id + v.curLen <= maxLen )
          l = v.curLen;                                   //确定替换串 v 传送字符数 l
      else l = maxLen− curLen− id;
      for ( j = 0; j < l; j++ ) ch[k++] = v.ch[j];        //连接替换串 v 到结果串 ch 后面
      curLen += id + l;                                   //修改结果串连接后的长度
      if ( curLen == maxLen ) break;                      //字符串超出范围
      for ( j = id + t.curLen; j < temp.curLen; j++ )
          temp.ch[j− id − t.curLen] = temp.ch[j];         //删改原来的字符串
      temp.curLen −= id + t.curLen;
      id = temp.Find ( t );
  }
  return *this;

}
```

2-15 编写一个算法 frequency，统计在一个输入字符串中各个不同字符出现的频度。用适当的测试数据来验证这个算法。

【解答】

```
include <iostream.h>
include "string.h"
void frequency( String& s, char& A[ ], int& C[ ], int &k ) {
// s 是输入字符串，数组 A[ ]中记录字符串中有多少种不同的字符，C[ ]中记录每
//一种字符的出现次数。这两个数组都应在调用程序中定义。k 返回不同字符数。
    int i, j, len = s.length( );
    if ( !len ) { cout << "The string is empty. " << endl;   k = 0;   return; }
    else {   A[0] = s[0];   C[0] = 1;   k = 1;                  /*语句 s[i]是串的重载操作*/
             for ( i = 1; i < len; i++ ) C[i] = 0;              /*初始化*/
             for ( i = 1; i < len; i++ ) {                      /*检测串中所有字符*/
                                  j = 0;
            while ( j < k && A[j] != s[i] ) j++;    /*检查 s[i]是否已在 A[ ]中*/
                 if ( j == k ) { A[k] = s[i]; C[k]++; k++ }     /*s[i]从未检测过*/
                 else C[j]++;                                   /*s[i]已经检测过*/
             }
         }
}
```

测试数据 s = "cast cast sat at a tasa\0"

测试结果

| A | c | a | s | t | b |
|---|---|---|---|---|---|
| C | 2 | 7 | 4 | 5 | 5 |

【另一解答】

```
include <iostream.h>
include "string.h"
const int charnumber = 128;                                  /*ASCII 码字符集的大小*/
void frequency( String& s, int& C[ ] ) {
// s 是输入字符串，数组 C[ ]中记录每一种字符的出现次数。
    for ( int i = 0; i < charnumber; i++ ) C[i] = 0;         /*初始化*/
    for ( i = 0; i < s.length ( ); i++ )                     /*检测串中所有字符*/
        C[ atoi (s[i]) ]++;                                  /*出现次数累加*/
    for ( i = 0; i < charnumber; i++ )                       /*输出出现字符的出现次数*/
        if ( C[i] > 0 ) cout << "( " << i << " ) : \t" << C[i] << "\t";
}
```

2-16 设串 s 为“aaab”，串 t 为“abcabaa”，串 r 为“abc□aabbabcabaacbacba”，试分别计算它们的失效函数 f (j)的值。

【解答】

| j | 0 | 1 | 2 | 3 | j | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| s | a | a | a | b | t | a | b | c | a | b | a | a |
| f (j) | −1 | 0 | 1 | −1 | f (j) | −1 | −1 | −1 | 0 | 1 | 0 | 0 |

2−17 设定整数数组 B[m+1][n+1]的数据在行、列方向上都按从小到大的顺序排序，且整型变量 x 中的数据在 B 中存在。试设计一个算法，找出一对满足 B[i][j] == x 的 i, j 值。要求比较次数不超过 m+n。

【解答】

算法的思想是逐次二维数组右上角的元素进行比较。每次比较有三种可能的结果：若相等，则比较结束；若右上角的元素小于 x，则可断定二维数组的最上面一行肯定没有与 x 相等的数据，下次比较时搜索范围可减少一行；若右上角的元素大于 x，则可断定二维数组的最右面一列肯定不包含与 x 相等的数据，下次比较时可把最右一列剔除出搜索范围。这样，每次比较可使搜索范围减少一行或一列，最多经过 m+n 次比较就可找到要求的与 x 相等的数据。

```
void find ( int B[ ][ ], int m, int n, int x, int& i, int& j ) {
//在二维数组 B[m][n]中寻找与 x 相等的元素，找到后，由 i 与 j 返回该数组元素的位置
   i = 0;   j = n;
   while ( B[i][j] != x )
      if ( B[i][j] < x ) i++;
      else j−−;
}
```

# 第三章 链表

单链表的结点类(ListNode **class**)和链表类(List **class**)的类定义。

```
template <class Type> class List;                    //前视的类定义

template <class Type> class ListNode {               //链表结点类的定义
friend class List<Type>;                                  //List 类作为友元类定义
private:
    Type data;                                            //数据域
    ListNode<Type> *link;                                 //链指针域
public:
    ListNode ( ) : link (NULL) { }                        //仅初始化指针成员的构造函数
    ListNode ( const Type& item ) : data (item), link (NULL) { }
                                                          //初始化数据与指针成员的构造函数
    ListNode<Type> * getNode ( const Type& item, ListNode<Type> *next = NULL )
                                                          //以 item 和 next 建立一个新结点
    ListNode<Type> * getLink ( ) { return link; }         //取得结点的下一结点地址
    Type getData ( ) { return data; }                     //取得结点中的数据
    void setLink ( ListNode<Type> * next ) { link = next; }          //修改结点的 link 指针
    void setData ( Type value ) { data = value; }         //修改结点的 data 值
};

template <class Type> class List {                    //单链表类定义
private:
    ListNode<Type> *first, *current;                      //链表的表头指针和当前元素指针
public:
    List ( const Type& value ) { first = current = new ListNode<Type> ( value ); }
                                                           //构造函数
    ~List ( ) { MakeEmpty ( );    delete first; }         //析构函数
    void MakeEmpty ( );                                   //将链表置为空表
    int Length ( ) const;                                 //计算链表的长度
    ListNode<Type> * Find ( Type value );                 //搜索含数据 value 的元素并成为当前元素
    ListNode<Type> * Locate( int i );                     //搜索第 i 个元素的地址并置为当前元素
    Type * GetData ( );                                   //取出表中当前元素的值
    int Insert ( Type value );                            //将 value 插在表当前位置之后并成为当前元素
    Type *Remove ( );                                     //将链表中的当前元素删去，填补者为当前元素
    ListNode<Type> * Firster ( ) { current = first; return first; }          //当前指针定位于表头结点
    Type *First ( );                                      //当前指针定位于表中第一个元素并返回其值
    Type *Next ( );                                       //将当前指针进到表中下一个元素并返回其值
    int NotNull ( ) { return current != NULL; }           //表中当前元素空否？空返回 1，不空返回 0
    int NextNotNull ( ) { return current != NULL && current->link != NULL; }
                                                          //当前元素下一元素空否？空返回 1，不空返回 0
};
```

3-1 线性表可用顺序表或链表存储。试问：

(1) 两种存储表示各有哪些主要优缺点?

(2) 如果有 n 个表同时并存，并且在处理过程中各表的长度会动态发生变化，表的总数也可能自动改变、在此情况下，应选用哪种存储表示？为什么？

(3) 若表的总数基本稳定，且很少进行插入和删除，但要求以最快的速度存取表中的元素，这时，应采用哪种存储表示？为什么？

【解答】

(1) 顺序存储表示是将数据元素存放于一个连续的存储空间中，实现顺序存取或(按下标)直接存取。它的存储效率高，存取速度快。但它的空间大小一经定义，在程序整个运行期间不会发生改变，因此，不易扩充。同时，由于在插入或删除时，为保持原有次序，平均需要移动一半(或近一半)元素，修改效率不高。

链接存储表示的存储空间一般在程序的运行过程中动态分配和释放，且只要存储器中还有空间，就不会产生存储溢出的问题。同时在插入和删除时不需要保持数据元素原来的物理顺序，只需要保持原来的逻辑顺序，因此不必移动数据，只需修改它们的链接指针，修改效率较高。但存取表中的数据元素时，只能循链顺序访问，因此存取效率不高。

(2) 如果有 n 个表同时并存，并且在处理过程中各表的长度会动态发生变化，表的总数也可能自动改变、在此情况下，应选用链接存储表示。

如果采用顺序存储表示，必须在一个连续的可用空间中为这 n 个表分配空间。初始时因不知道哪个表增长得快，必须平均分配空间。在程序运行过程中，有的表占用的空间增长得快，有的表占用的空间增长得慢；有的表很快就用完了分配给它的空间，有的表才用了少量的空间，在进行元素的插入时就必须成片地移动其他的表的空间，以空出位置进行插入；在元素删除时，为填补空白，也可能移动许多元素。这个处理过程极其繁琐和低效。

如果采用链接存储表示，一个表的存储空间可以连续，可以不连续。表的增长通过动态存储分配解决，只要存储器未满，就不会有表溢出的问题；表的收缩可以通过动态存储释放实现，释放的空间还可以在以后动态分配给其他的存储申请要求，非常灵活方便。对于 n 个表(包括表的总数可能变化)共存的情形，处理十分简便和快捷。所以选用链接存储表示较好。

(3) 应采用顺序存储表示。因为顺序存储表示的存取速度快，但修改效率低。若表的总数基本稳定，且很少进行插入和删除，但要求以最快的速度存取表中的元素，这时采用顺序存储表示较好。

3-2 针对带表头结点的单链表，试编写下列函数。

(1) 定位函数 Locate：在单链表中寻找第 i 个结点。若找到，则函数返回第 i 个结点的地址；若找不到，则函数返回 NULL。

(2) 求最大值函数 max：通过一趟遍历在单链表中确定值最大的结点。

(3) 统计函数 number：统计单链表中具有给定值 x 的所有元素。

(4) 建立函数 create：根据一维数组 a[n]建立一个单链表，使单链表中各元素的次序与 a[n]中各元素的次序相同，要求该程序的时间复杂性为 O(n)。

(5) 整理函数 tidyup：在非递减有序的单链表中删除值相同的多余结点。

【解答】

(1) 实现定位函数的算法如下：

```
template <class Type> ListNode <Type> * List <Type> :: Locate ( int i ) {
//取得单链表中第 i 个结点地址, i 从 1 开始计数, i <= 0 时返回指针 NULL
    if ( i <= 0 ) return NULL;                          //位置 i 在表中不存在
    ListNode <Type> * p = first;   int k = 0;           //从表头结点开始检测
    while ( p != NULL && k < i ) { p = p->link;   k++; }    //循环, p == NULL 表示链短, 无第 i 个结点
```

```
    return p;                                          //若 p != NULL, 则 k == i, 返回第 i 个结点地址
}
```

(2) 实现求最大值的函数如下：

```
template <class Type> ListNode <Type> * List <Type> :: Max ( ) {
//在单链表中进行一趟检测，找出具有最大值的结点地址，如果表空，返回指针 NULL
    if ( first->link == NULL ) return NULL;                //空表，返回指针 NULL
    ListNode <Type> * pmax = first->link, p = first->link->link;    //假定第一个结点中数据具有最大值
    while ( p != NULL ) {                                  //循环，下一个结点存在
        if ( p->data > pmax->data ) pmax = p;              //指针 pmax 记忆当前找到的具最大值结点
        p = p->link;                                       //检测下一个结点
    }
    return pmax;
}
```

(3) 实现统计单链表中具有给定值 x 的所有元素的函数如下：

```
template <class Type> int List <Type> :: Count ( Type& x ) {
//在单链表中进行一趟检测，找出具有最大值的结点地址，如果表空，返回指针 NULL
    int n = 0;
    ListNode <Type> * p = first->link;                     //从第一个结点开始检测
    while ( p != NULL ) {                                  //循环，下一个结点存在
        if ( p->data == x ) n++;                           //找到一个，计数器加 1
        p = p->link;                                       //检测下一个结点
    }
    return n;
}
```

(4) 实现从一维数组 A[n]建立单链表的函数如下：

```
template <class Type> void List <Type> :: Create ( Type A[ ], int n ) {
//根据一维数组 A[n]建立一个单链表，使单链表中各元素的次序与 A[n]中各元素的次序相同
    ListNode<Type> * p;
    first = p = new ListNode<Type>;                        //创建表头结点
    for ( int i = 0; i < n; i++ ) {
        p->link = new ListNode<Type> ( A[i] );             //链入一个新结点，值为 A[i]
        p = p->link;                                       //指针 p 总指向链中最后一个结点
    }
    p->link = NULL;
}
```

采用递归方法实现时，需要通过引用参数将已建立的单链表各个结点链接起来。为此，在递归地扫描数组 A[n]的过程中，先建立单链表的各个结点，在退出递归时将结点地址 p（被调用层的形参）带回上一层（调用层）的实参 p->link。

```
template<Type> void List<Type> :: create ( Type A[ ], int n, int i, ListNode<Type> *& p ) {
//私有函数：递归调用建立单链表
    if ( i == n ) p = NULL;
    else {   p = new ListNode<Type>( A[i] );               //建立链表的新结点
             create ( A, n, i+1, p->link );                //递归返回时 p->link 中放入下层 p 的内容
         }
```

```
}
template<Type> void List<Type> :: create ( Type A[ ], int n ) {
//外部调用递归过程的共用函数
    first = current = new ListNode<Type>;                    //建立表头结点
    create ( A, n, 0, first->link );                         //递归建立单链表
}
```

(5) 实现在非递减有序的单链表中删除值相同的多余结点的函数如下：

```
template <class Type> void List <Type> :: tidyup ( ) {
    ListNode<Type> * p = first->link, temp;                  //检测指针，初始时指向链表第一个结点
    while ( p != NULL && p->link != NULL )                   //循环检测链表
        if ( p->data == p->link->data ) {                    //若相邻结点所包含数据的值相等
            temp = p->first;    p->link = temp->link;        //为删除后一个值相同的结点重新拉链
            delete temp;                                     //删除后一个值相同的结点
        }
        else p = p->link;                                    //指针 p 进到链表下一个结点
}
```

3-3 设 ha 和 hb 分别是两个带表头结点的非递减有序单链表的表头指针，试设计一个算法，将这两个有序链表合并成一个非递增有序的单链表。要求结果链表仍使用原来两个链表的存储空间，不另外占用其它的存储空间。表中允许有重复的数据。

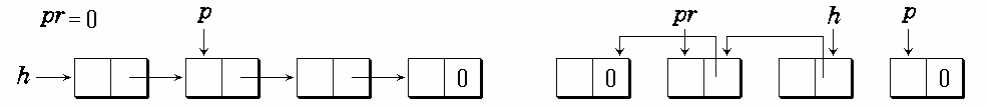
【解答】

```
#include <iostream.h>
template <class Type> class List;
template <class Type> class ListNode {
friend class List<Type>;
public:
    ListNode ( );                                    //构造函数
    ListNode ( const Type& item );                   //构造函数
private:
    Type data;
    ListNode<Type> *link;
};

template <class Type> class List {
public:
    List ( const Type finishied );                   //建立链表
    void Browse ( );                                 //打印链表
    void Merge ( List<Type> &hb );                   //连接链表
private:
    ListNode<Type> *first, *last;
};

//各成员函数的实现
template <class Type>
ListNode<Type> :: ListNode ( ) : link ( NULL ) { }
```

```
//构造函数，仅初始化指针成员。

template <class Type> ListNode<Type> :: ListNode ( const Type & item ) : data ( item ), link ( NULL ) { }
//构造函数，初始化数据与指针成员。

template <class Type> List<Type> :: List ( const Type finishied ) {
//创建一个带表头结点的有序单链表，finished 是停止建表输入标志，是所有输入值中不可能出现的数值。
    first = last = new ListNode<Type>( );              //创建表头结点
    Type value;      ListNode<Type> *p, *q, *s;
    cin >> value;
    while ( value != finished ) {                      //循环建立各个结点
        s = new ListNode<Type>( value );
        q = first;   p = first->link;
        while ( p != NULL && p->data <= value )
            { q = p;   p = p->link; }                  //寻找新结点插入位置
        q->link = s;   s->link = p;                    //在 q, p 间插入新结点
        if ( p == NULL ) last = s;
        cin >> value;
    }
}

template <class Type> void List<Type> :: Browse ( ) {
//浏览并输出链表的内容
    cout<<"\nThe List is : \n";
    ListNode<Type> *p = first->link;
    while ( p != NULL ) {
        cout << p->data;
        if ( p != last ) cout << "->";
        else cout << endl;
        p = p->link;
    }
}

template <class Type> void List <Type> :: Merge ( List<Type>& hb) {
//将当前链表 this 与链表 hb 按逆序合并，结果放在当前链表 this 中。
    ListNode<Type> *pa, *pb, *q, *p;
    pa = first->link;   pb = hb.first->link;           //检测指针跳过表头结点
    first->link = NULL;                                //结果链表初始化
    while ( pa != NULL && pb != NULL ) {               //当两链表都未结束时
        if ( pa->data <= pb->data )
            { q = pa;   pa = pa->link; }               //从 pa 链中摘下
        else
            { q = pb;   pb = pb->link; }               //从 pb 链中摘下
```

```
            q→link = first->link;    first->link = q;          //链入结果链的链头
        }
        p = ( pa != NULL ) ? pa : pb;                          //处理未完链的剩余部分
        while ( p != NULL ) {
            q = p;    p = p->link;
            q->link = first->link;    first->link = q;
        }
    }
```

3-4 设有一个表头指针为 h 的单链表。试设计一个算法，通过遍历一趟链表，将链表中所有结点的链接方向逆转，如下图所示。要求逆转结果链表的表头指针 h 指向原链表的最后一个结点。




【解答 1】

```
template<class Type> void List<Type> :: Inverse ( ) {
    if ( first == NULL ) return;
    ListNode<Type> *p = first->link, *pr = NULL;
    while ( p != NULL ) {
        first->link = pr;                                      //逆转 first 指针
        pr = first;    first = p;    p = p->link;              //指针前移
    }
    first->link = pr;
}
```

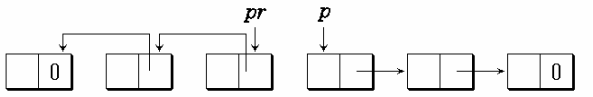
【解答 2】

```
template<class Type> void List<Type> :: Inverse ( ) {
    ListNode<Type> *p, *head = new ListNode<Type> ( );     //创建表头结点, 其 link 域默认为 NULL
    while ( first != NULL ) {
        p = first;    first = first->link;                 //摘下 first 链头结点
        p->link = head->link;    head->link = p;           //插入 head 链前端
    }
    first = head->link;    delete head;                    //重置 first, 删去表头结点
}
```

3-5 从左到右及从右到左遍历一个单链表是可能的，其方法是在从左向右遍历的过程中将连接方向逆转，如右图所示。在图中的指针 p 指向当前正在访问的结点，指针 pr 指向指针 p 所指结点的左侧的结点。此时，指针 p 所指结点左侧的所有结点的链接方向都已逆转。

(1) 编写一个算法，从任一给定的位置(pr, p)开始，将指针 p 右移 k 个结点。如果 p 移出链表，则将 p 置为 0，并让 pr 停留在链表最右边的结点上。

(2) 编写一个算法，从任一给定的位置(pr, p)开始，将指针 p 左移 k 个结点。如果 p 移出链表，则将 p 置为 0，并让 pr 停留在链表最左边的结点上。



【解答】

(1) 指针 p 右移 k 个结点

```
template<class Type> void List<Type> ::
siftToRight ( ListNode<Type> *& p, ListNode<Type> *& pr, int k ) {
    if ( p == NULL && pr != first ) {                    //已经在链的最右端
        cout << "已经在链的最右端，不能再右移。" << endl;
        return;
    }
    int i;   ListNode<Type> *q;
    if ( p == NULL )                                     //从链头开始
        { i = 1;   pr = NULL;   p = first; }             //重置 p 到链头也算一次右移
    else i = 0;
    while ( p != NULL && i < k ) {                       //右移 k 个结点
        q = p->link;   p->link = pr;                     //链指针 p→link 逆转指向 pr
        pr = p;   p = q;   i++;                          //指针 pr, p 右移
    }
    cout << "右移了" << i << "个结点。" << endl;
}
```

(2) 指针 p 左移 k 个结点

```
template<class Type> void List<Type> ::
siftToLeft ( ListNode<Type> *& p, ListNode<Type> *& pr, int k ) {
    if ( p == NULL && pr == first ) {                    //已经在链的最左端
        cout << "已经在链的最左端，不能再左移。" << endl;
        return;
    }
    int i = 0;   ListNode<Type> *q;
    while ( pr != NULL && i < k ) {                      //左移 k 个结点
        q = pr->link;   pr->link = p;                    //链指针 pr->link 逆转指向 p
        p = pr;   pr = q;   i++;                         //指针 pr, p 左移
    }
    cout << "左移了" << i << "个结点。" << endl;
    if ( i < k ) { pr = p;   p = NULL; }                 //指针 p 移出表外，重置 p, pr
}
```

3-6 试写出用单链表表示的字符串类及字符串结点类的定义，并依次实现它的构造函数、以及计算串长度、串赋值、判断两串相等、求子串、两串连接、求子串在串中位置等 7 个成员函数。要求每个字符串结点中只存放一个字符。

【解答】

```
//用单链表表示的字符串类 string1 的头文件 string1.h
#include <iostream.h>
const int maxLen = 300;                                  //字符串最大长度为 300（理论上可以无限长）
class string1 {
public:
    string1 ( );                                         //构造空字符串
    string1 ( char * obstr );                            //从字符数组建立字符串
```

```
    ~string1 ( );                                   //析构函数
    int Length ( ) const { return curLen; }         //求字符串长度
    string1& operator = ( string1& ob );            //串赋值
    int operator == ( string1& ob );                //判两串相等
    char& operator [ ] ( int i );                   //取串中字符
    string1& operator ( ) ( int pos, int len );     //取子串
    string1& operator += ( string1& ob );           //串连接
    int Find ( string1& ob );                       //求子串在串中位置(模式匹配)
    friend ostream& operator << ( ostream& os, string1& ob );
    friend istream& operator >> ( istream& is, string1& ob );
private:
    ListNode<char>*chList;                          //用单链表存储的字符串
    int curLen;                                     //当前字符串长度
}

//单链表表示的字符串类 string1 成员函数的实现，在文件 string1.cpp 中
#include <iostream.h>
#include "string1.h"
string1 :: string1( ) {                             //构造函数
    chList = new ListNode<char> ( '\0' );
    curLen = 0;
}

string1 :: string1( char *obstr ) {                 //复制构造函数
    curLen = 0;
    ListNode<char> *p = chList = new ListNode<char> ( *obstr );
    while ( *obstr != '\0' ) {
        obstr++;
        p = p->link = new ListNode<char> ( *obstr );
        curLen++;
    }
}

string1& string1 :: operator = ( string1& ob ) {    //串赋值
    ListNode<char> *p = ob.chList;
    ListNode<char> *q = chList = new ListNode<char> ( p->data );
    curLen = ob.curLen;
    while ( p->data != '\0' ) {
        p = p->link;
        q = q->link = new ListNode<char> ( p->data );
    }
    return *this;
}
```

```
int string1 :: operator == ( string1& ob ) {                    //判两串相等
    if ( curLen != ob.curLen ) return 0;
    ListNode <char> *p = chList, *q = ob.chList;
    for ( int i = 0; i < curLen; i++ )
        if ( p->data != q->data ) return 0;
        else { p = p->link;    q = q->link; }
    return 1;
}

char& string1 :: operator [ ] ( int i ) {                       //取串中字符
    if ( i >= 0 && i < curLen ) {
       ListNode <char> *p = chList;    int k = 0;
       while ( p != NULL && k < i ) { p = p->link; k++; }
       if ( p != NULL ) return p->data;
    }
    return '\0';
}

string1& string1 :: operator ( ) ( int pos, int len ) {         //取子串
    string1 temp;
    if ( pos >= 0 && len >= 0 && pos < curLen && pos + len - 1 < curLen ) {
        ListNode<char> *q, *p = chList;
        for ( int k = 0; k < pos; k++; ) p = p->link;           //定位于第 pos 结点
        q = temp.chList = new ListNode<char> ( p->data );
        for ( int i = 1; i < len; i++ ) {                       //取长度为 len 的子串
          p = p->link;
          q = q->link = new ListNode<char> ( p->data );
        }
        q->link = new ListNode<char> ( '\0' );                  //建立串结束符
        temp.curLen = len;
    }
    else { temp.curLen = 0;    temp.chList = new ListNode<char> ( '\0' ); }
    return *temp;
}
string1& string1 :: operator += ( string1& ob ) {               //串连接
    if ( curLen + ob.curLen > maxLen ) len = maxLen - curLen;
    else len = ob.curLen;                                       //传送字符数
    ListNode<char> *q = ob.chList, *p = chList;
    for ( int k = 0; k < curLen - 1; k++; ) p = p->link;        //this 串的串尾
    k = 0;
    for ( k = 0; k < len; k++ ) {                               //连接
        p = p->link = new ListNode<char> ( q->data );
        q = q->link;
    }
```

```
    p→link = new ListNode<char> ( '\0' );
}

int string1 :: Find ( string1& ob ) {                    //求子串在串中位置(模式匹配)
    int slen = curLen,   oblen = ob.curLen,   i = slen − oblen;
    string1 temp = this;
    while ( i > −1 )
        if ( temp( i, oblen ) == ob ) break;
        else i−− ;
    return i;
}
```

3-7 如果用循环链表表示一元多项式，试编写一个函数 Polynomial :: Calc(x)，计算多项式在 x 处的值。

【解答】

下面给出表示多项式的循环链表的类定义。作为私有数据成员，在链表的类定义中封装了 3 个链接指针：first、last 和 current，分别指示链表的表头结点、链尾结点和最后处理到的结点。

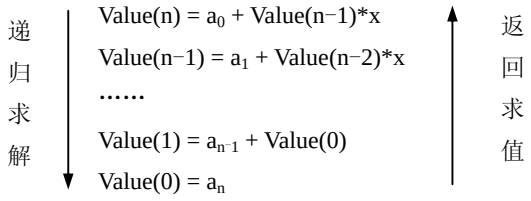
```
enum Boolean { False, True }
class Polynomal;                                         //多项式前视类定义

class Term {                                             //项类定义
friend class Polynomal;
private:
    double coef, expn;                                   //系数与指数
    Term *link;                                          //项链接指针
public:
    Term ( double c = 0, double e = 0, Term * next = NULL ) : coef (c), expn(e), link (next) { }
}

class Polynomal {                                        //多项式类定义
private:
    Term *first, *current;                               //头指针，当前指针
    int n;                                               //多项式阶数
public:
    Polynomal ( );                                       //构造函数
    ~Polynomal ( );                                      //析构函数
    int Length ( ) const;                                //计算多项式项数
    Boolean IsEmpty ( ) { return first->link == first; }  //判是否零多项式
    Boolean Find ( const double& value );                //在多项式中寻找其指数值等于 value 的项
    double getExpn ( ) ( ) const;                        //返回当前项中存放的指数值
    double getCoef ( ) ( ) const;                        //返回当前项中存放的系数值
    void Firster ( ) { current = first; }                //将当前指针置于头结点
    Boolean First ( );                                   //将当前指针指向链表的第一个结点
    Boolean Next ( );                                    //将当前指针指到当前结点的后继结点
```

```
    Boolean Prior ( );                                  //将当前指针指到当前结点的前驱结点
    void Insert ( const double coef, double expn );     //插入新结点
    void Remove ( );                                    //删除当前结点
    double Calc ( double x );                           //求多项式的值
    friend Polynomial operator + ( Polynomial &, Polynomial & );
    friend Polynomial operator * ( Polynomial &, Polynomial & );
};
```

对于多项式 $P_n(x) = a_0 + a_1x + a_2x^2 + a_3x^3 + \cdots + a_{n-1}x^{n-1} + a_nx^n$，可用 Horner 规则将它改写求值：

$$P_n(x) = a_0 + (a_1x + ( a_2 + ( a_3 + \cdots + ( a_{n-1} + a_n*x )*x \cdots )*x )*x )*x$$

因为不是顺序表，必须采用递归算法实现：




```
double Polynomal :: Value ( Term *p, double x ) {
//私有函数：递归求子多项式的值
    if ( p->link == first ) return p->coef;
    else return p->coef + x * Value ( p->link, x );
}

double Polynomal :: Calc ( double x ) {
//共有函数：递归求多项式的值
    Term * pc = first->link;
    if ( pc == first ) cout << 0 << endl;
    else cout << Value ( pc, x ) << endl;
}
```

但是，当多项式中许多项的系数为 0 时，变成稀疏多项式，如 $P_{50}(x) = a_0 + a_{13}x^{13} + a_{35}x^{35} + a_{50}x^{50}$，为节省存储起见，链表中不可能保存有零系数的结点。此时，求值函数要稍加改变：

```
#include <math.h>
double Polynomal :: Value ( Term *p, double e, double x ) {
//私有函数：递归求子多项式的值。pow(x, y)是求 x 的 y 次幂的函数，它的原型在“math.h”中
    if ( p->link == first ) return p->coef;
    else return p->coef + pow( x, p->expn – e ) * Value ( p->link, p->expn, x );
}

double Polynomal :: Calc ( double x ) {
//共有函数：递归求多项式的值
    Term * pc = first->link;
    if ( pc == first ) cout << 0 << endl;
    else cout << Value ( pc, 0, x ) << endl;
```

**}**

3-8 设 a 和 b 是两个用带有表头结点的循环链表表示的多项式。试编写一个算法，计算这两个多项式的乘积 c = a*b，要求计算后多项式 a 与 b 保持原状。如果这两个多项式的项数分别为 n 与 m, 试说明该算法的执行时间为 O(nm²)或 O(n²m)。但若 a 和 b 是稠密的, 即其很少有系数为零的项, 那么试说明该乘积算法的时间代价为 O(nm)。

【解答】

假设 $a = a_0 + a_1x + a_2x^2 + \cdots + a_{n-1}x^{n-1} + a_nx^n = \sum_{i=0}^{n} a_i x^i$

$b = b_0 + b_1x + b_2x^2 + \cdots + b_{m-1}x^{m-1} + b_mx^m = \sum_{j=0}^{m} b_j x^j$

则它们的乘积为 $c = a \times b = \left(\sum_{i=0}^{n} a_i x^i\right) \times \left(\sum_{j=0}^{m} b_j x^j\right) = \sum_{i=0}^{n} a_i x^i \sum_{j=0}^{m} b_j x^j = \sum_{i=0}^{n} a_i \sum_{j=0}^{m} b_j x^{i+j}$

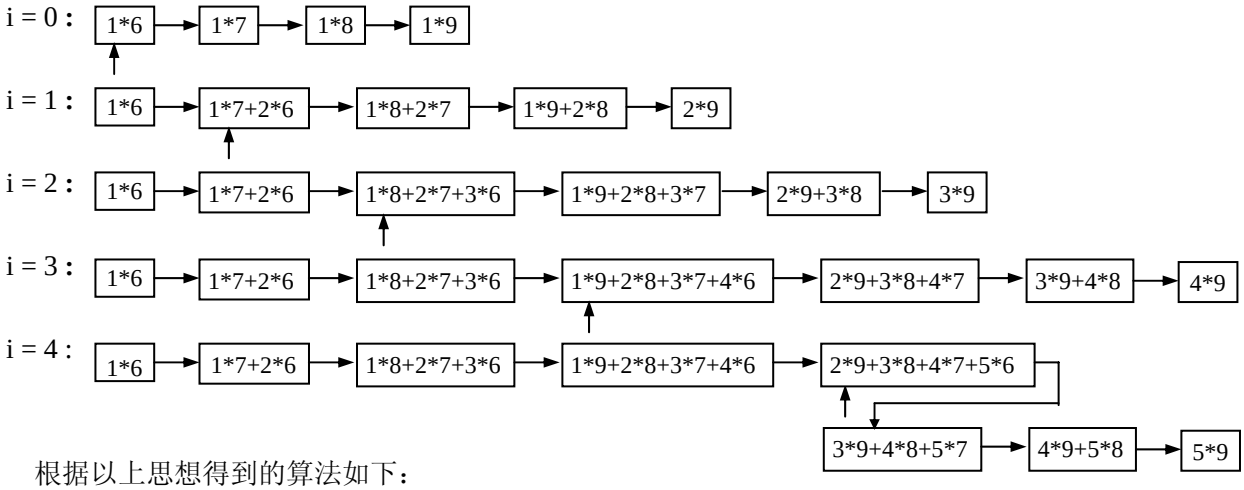
例如，$a = 1 + 2x + 3x^2 + 4x^3 + 5x^4$，　$b = 6 + 7x + 8x^2 + 9x^3$，它们的乘积

$c = (1+2x+3x^2+4x^3+5x^4)*(6+7x+8x^2+9x^3)$

$=1*6 + (1*7+2*6)x + (1*8+2*7+3*6)x^2+(1*9+2*8+3*7+4*6)x^3+(2*9+3*8+4*7+5*6)x^4$

$+(3*9+4*8+5*7)x^5+(4*9+5*8)x^6+5*9x^7$

在求解过程中，固定一个 $a_i$，用它乘所有 $b_j$，得到 $x^{i+j}$ 的系数的一部分。这是一个二重循环。




根据以上思想得到的算法如下：

```
Polynomal& Polynomal :: operator * ( Polynomal& a, Polynomal& b ) {
    Term * pa = a.first->link, pb, pc, fc;          //pa 与 pb 是两个多项式链表的检测指针
    first = fc = pc = new Term;                     //fc 是每固定一个 ai 时 ai 结点指针, pc 是存放指针
    while ( pa != NULL ) {                          //每一个 ai 与 b 中所有项分别相乘
        pb = b.first->link;
        while ( pb != NULL ) {                          //扫描多项式 b 所有项
            temp = pa->data * pb->data;                 //计算 ai * bj
            if ( pc->link != NULL ) pc->link->data = pc->link->data + temp      //累加
            else pc->link = new Term (temp);            //增加项,事实上,每次 pa 变化,链结点要随之增加
            pc = pc->link;   pb = pb->link;
        }
        pc = fc = fc->link;   pa = pa->link;            //处理多项式 a 的下一 ai
    }
    pc->link = NULL;
```

```
    return *this;
}
```

这个算法有一个二重循环，内层循环中语句的重复执行次数是 O(n*m)。其中，n 是第一个多项式的阶数，m 是第二个多项式的阶数。这是稠密多项式的情形。

对于稀疏多项式的情形请自行考虑。

3-9 计算多项式 $P_n(x) = a_0 x^n + a_1 x^{n-1} + a_2 x^{n-2} + \cdots\cdots + a_{n-1} x + a_n$ 的值，通常使用的方法是一种嵌套的方法。它可以描述为如下的迭代形式：$b_0 = a_0$， $b_{i+1} = x * b_i + a_{i+1}$， $i = 0, 1, \cdots, n-1$。若设 $b_n = p_n(x)$．则问题可以写为如下形式：$P_n(x) = x * P_{n-1}(x) + a_n$ ，此处， $P_{n-1}(x) = a_0 x^{n-1} + a_1 x^{n-2} + \cdots\cdots + a_{n-2} x + a_{n-1}$，这是问题的递归形式。试编写一个递归函数，计算这样的多项式的值。

【解答】

如果用循环链表方式存储多项式，求解方法与 3-7 题相同。如果用数组方式存储多项式，当零系数不多时，可用顺序存放各项系数的一维数组存储多项式的信息，指数用数组元素的下标表示

| | 0 | 1 | 2 | 3 | | i | | n-2 | n-1 |
|---|---|---|---|---|---|---|---|---|---|
| coef | $a_0$ | $a_1$ | $a_2$ | $a_3$ | … | $a_i$ | … | $a_{n-2}$ | $a_{n-1}$ |

多项式的类定义如下：

```
struct Polynomal {
    double * coef;
    int n;
}
```

这样可得多项式的解法：

```
double Polynomal :: Value ( int i, double x ) {
//私有函数：递归求子多项式的值
    if ( i == n-1 ) return coef[n-1];
    else return coef[i] + x * Value ( i+1, x );
}

double Polynomal :: Calc ( double x ) {
//共有函数：递归求多项式的值
    if ( n == 0 ) cout << 0 << endl;
    else cout << Value ( 0, x ) << endl;
}
```

3-10 试设计一个实现下述要求的 Locate 运算的函数。设有一个带表头结点的双向链表 L，每个结点有 4 个数据成员：指向前驱结点的指针 prior、指向后继结点的指针 next、存放数据的成员 data 和访问频度 freq。所有结点的 freq 初始时都为 0。每当在链表上进行一次 Locate (L, x)操作时，令元素值为 x 的结点的访问频度 freq 加 1，并将该结点前移，链接到与它的访问频度相等的结点后面，使得链表中所有结点保持按访问频度递减的顺序排列，以使频繁访问的结点总是靠近表头。

【解答】

```
#include <iostream.h>
//双向循环链表结点的构造函数
DblNode (Type value, DblNode<Type> *left, DblNode<Type> *right ) :
```

```
        data ( value ), freq ( 0 ), lLink ( left ), rLink ( right ) { }
DblNode (Type value ) :
        data ( value ), freq ( 0 ), lLink ( NULL ), rLink ( NULL ) { }

template <class Type>
DblList<Type> :: DblList ( Type uniqueVal ) {
    first = new DblNode<Type>( uniqueVal );
    first->rLink = first->lLink = first;                    //创建表头结点
    current = NULL;
    cout << "开始建立双向循环链表：\n";
    Type value;    cin >> value;
    while ( value != uniqueVal ) {                          //每次新结点插入在表头结点后面
        first->rLink = new DblNode<Type>( value, first, first->rLink );
        cin >> value;
    }
}

template <class Type>
void DblList<Type> :: Locate ( Type & x ) {
//定位
    DblNode<Type> *p = first->rLink;
    while ( p != first && p->data != x ) p = p->rLink;
    if ( p != first ) {                                     //链表中存在 x
        p→freq++;                                           //该结点的访问频度加 1
        current = p;                                        //从链表中摘下这个结点
        current->lLink->rLink = current->rLink;
        current->rLink->lLink = current->lLink;
        p = current->lLink;                                 //寻找从新插入的位置
        while ( p != first && current->freq > p->freq )
            p = p->lLink;
        current->rLink = p->rLink;                          //插入在 p 之后
        current->lLink = p;
        p->rLink->lLink = current;
        p->rLink = current;
    }
    else cout<<"Sorry. Not find!\n";                        //没找到
}
```

3-11 利用双向循环链表的操作改写 2-2 题，解决约瑟夫(Josephus)问题。

【解答】

```
#include <iostream.h>
#include "DblList.h"
Template <class Type> void DblList <Type> :: Josephus ( int n, int m ) {
    DblNode<Type> p = first, temp;
    for ( int i = 0; i < n-1; i++ ) {                       //循环 n-1 趟，让 n-1 个人出列
```

```
        for ( int j = 0; j < m−1; j++ ) p = p−>rLink;          //让 p 向后移动 m−1 次
        cout << "Delete person " << p−>data << endl;
        p−>lLink−>rLink = p−>rLink;                            //从链中摘下 p
        p−>rLink−>lLink = p−>lLink;
        temp = p−>rlink;    delete p;    p = temp;             //删除 p 所指结点后, p 改指下一个出发点
    }
    cout << "The winner is " << p−>data << endl;
}

void main ( ) {
    DblList<int> dlist;                                        //定义循环链表 dlist 并初始化
    int n, m;                                                  //n 是总人数，m 是报数值
    cout << "Enter the Number of Contestants?";
    cin >> n >> m;
    for ( int i = 1; i <= n; i++ ) dlist.insert (i);           //建立数据域为 1, 2, … 的循环链表
    dlist.Josephus (n, m);                                     //解决约瑟夫问题，打印胜利者编号
}
```

3-12 试设计一个算法，改造一个带表头结点的双向链表，所有结点的原有次序保持在各个结点的 rLink 域中，并利用 lLink 域把所有结点按照其值从小到大的顺序连接起来。

【解答】

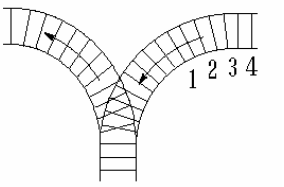
```
template<Type> void DblList<Type> :: sort ( ) {
    DblNode<Type> *
    s = first−>link;                                          //指针 s 指向待插入结点, 初始时指向第一个结点
    while ( s != NULL ) {                                      //处理所有结点
        pre = first;    p = first−>lLink;                      //指针 p 指向待比较的结点, pre 是 p 的前驱指针
        while ( p != NULL && s−>data < p−>data )               //循 lLink 链寻找结点 *s 的插入位置
          { pre = p;    p = p−>lLink; }
        pre−>lLink = s;    s−>lLink = p;                       //结点 *s 在 lLink 方向插入到 *pre 与 *p 之间
    }
}
```

# 第四章 栈与队列

4-1 改写顺序栈的进栈成员函数 Push (x )，要求当栈满时执行一个 stackFull ( )操作进行栈满处理。其功能是：动态创建一个比原来的栈数组大二倍的新数组，代替原来的栈数组，原来栈数组中的元素占据新数组的前 MaxSize 位置。

【解答】
```
template<class Type>void stack<Type> :: push ( const Type & item ) {
    if ( isFull ( ) ) stackFull ( );                    //栈满，做溢出处理
    elements [ ++top ] = item;                          //进栈
}

template<class Type> void stack<Type> :: stackFull ( ) {
    Type * temp = new Type [ 3 * maxSize ];             //创建体积大二倍的数组
    for ( int i = 0; i <= top; i++ )                    //传送原数组的数据
        temp[i] = elements[i];
    delete [ ] elements;                                //删去原数组
    maxSize *= 3;                                       //数组最大体积增长二倍
    elements = temp;                                    //新数组成为栈的数组空间
}
```

4-2 铁路进行列车调度时，常把站台设计成栈式结构的站台，如右图所示。试问：



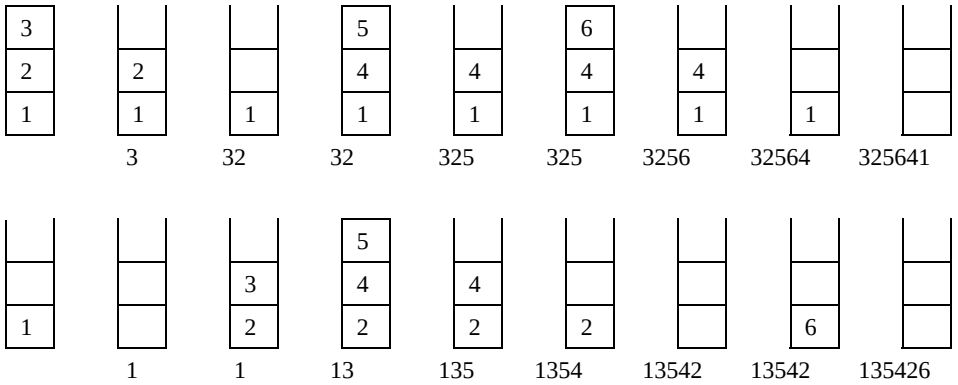

(1) 设有编号为 1,2,3,4,5,6 的六辆列车，顺序开入栈式结构的站台，则可能的出栈序列有多少种?

(2) 若进站的六辆列车顺序如上所述，那么是否能够得到 435612, 325641, 154623 和 135426 的出站序列，如果不能，说明为什么不能；如果能，说明如何得到(即写出"进栈"或"出栈"的序列)。

【解答】

(1) 可能的不同出栈序列有 $(1/(6+1))*C_{12}^{6}=132$ 种。

(2) 不能得到 435612 和 154623 这样的出栈序列。因为若在 4, 3, 5, 6 之后再将 1, 2 出栈，则 1, 2 必须一直在栈中，此时 1 先进栈，2 后进栈，2 应压在 1 上面，不可能 1 先于 2 出栈。154623 也是这种情况。出栈序列 325641 和 135426 可以得到。




4-3 试证明：若借助栈可由输入序列 1, 2, 3, …, n 得到一个输出序列 $p_1, p_2, p_3, \cdots, p_n$ (它是

输入序列的某一种排列)，则在输出序列中不可能出现以下情况，即存在 i < j < k，使得 $p_j < p_k < p_i$。(提示：用反证法)

【解答】

因为借助栈由输入序列 1, 2, 3, …, n，可得到输出序列 $p_1, p_2, p_3, \cdots, p_n$ ，如果存在下标 i, j, k，满足 i < j < k，那么在输出序列中，可能出现如下 5 种情况：

① i 进栈，i 出栈，j 进栈，j 出栈，k 进栈，k 出栈。此时具有最小值的排在最前面 $p_i$ 位置，具有中间值的排在其后 $p_j$ 位置，具有最大值的排在 $p_k$ 位置，有 $p_i < p_j < p_k$，不可能出现 $p_j < p_k < p_i$ 的情形；

② i 进栈，i 出栈，j 进栈，k 进栈，k 出栈，j 出栈。此时具有最小值的排在最前面 $p_i$ 位置，具有最大值的排在 $p_j$ 位置，具有中间值的排在最后 $p_k$ 位置，有 $p_i < p_k < p_j$ ，不可能出现 $p_j < p_k < p_i$ 的情形；

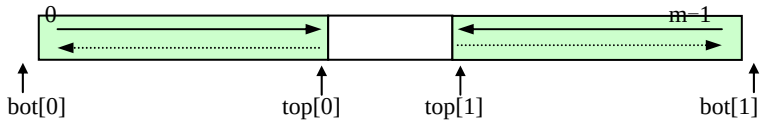
③ i 进栈，j 进栈，j 出栈，i 出栈，k 进栈，k 出栈。此时具有中间值的排在最前面 $p_i$ 位置，具有最小值的排在其后 $p_j$ 位置，有 $p_j < p_i < p_k$，不可能出现 $p_j < p_k < p_i$ 的情形；

④ i 进栈，j 进栈，j 出栈，k 进栈，k 出栈，i 出栈。此时具有中间值的排在最前面 $p_i$ 位置，具有最大值的排在其后 $p_j$ 位置，具有最小值的排在 $p_k$ 位置，有 $p_k < p_i < p_j$， 也不可能出现 $p_j < p_k < p_i$ 的情形；

⑤ i 进栈，j 进栈，k 进栈，k 出栈，j 出栈，i 出栈。此时具有最大值的排在最前面 $p_i$ 位置，具有中间值的排在其后 $p_j$ 位置，具有最小值的排在 $p_k$ 位置，有 $p_k < p_j < p_i$， 也不可能出现 $p_j < p_k < p_i$ 的情形；

4-4 将编号为 0 和 1 的两个栈存放于一个数组空间 V[m]中，栈底分别处于数组的两端。当第 0 号栈的栈顶指针 top[0]等于-1 时该栈为空，当第 1 号栈的栈顶指针 top[1]等于 m 时该栈为空。两个栈均从两端向中间增长。当向第 0 号栈插入一个新元素时，使 top[0]增 1 得到新的栈顶位置，当向第 1 号栈插入一个新元素时，使 top[1]减 1 得到新的栈顶位置。当 top[0]+1 == top[1]时或 top[0] == top[1]-1 时，栈空间满，此时不能再向任一栈加入新的元素。试定义这种双栈(Double Stack)结构的类定义，并实现判栈空、判栈满、插入、删除算法。

【解答】




双栈的类定义如下：

```
#include <assert.h>
template <class Type> class DblStack {                  //双栈的类定义
    private:
        int top[2], bot[2];                             //双栈的栈顶指针和栈底指针
        Type *elements;                                 //栈数组
        int m;                                          //栈最大可容纳元素个数
    public:
        DblStack ( int sz =10 );                        //初始化双栈，总体积 m 的默认值为 10
        ~DblStack ( ) { delete [ ] elements; }          //析构函数
        void DblPush ( const Type& x, int i );          //把 x 插入到栈 i 的栈顶
        int DblPop ( int i );                           //退掉位于栈 i 栈顶的元素
        Type * DblGetTop ( int i );                     //返回栈 i 栈顶元素的值
        int IsEmpty ( int i ) const { return top[i] == bot[i]; }  //判栈 i 空否，空则返回 1，否则返回 0
```

```
        int IsFull ( ) const { return top[0]+1 == top[1]; }        //判栈满否，满则返回 1，否则返回 0
        void MakeEmpty ( int i );                                  //清空栈 i 的内容
    }

    template <class Type> DblStack<Type> :: DblStack ( int sz ) : m(sz), top[0] (−1), bot[0](−1), top[1](sz),
bot[1](sz) {
    //建立一个最大尺寸为 sz 的空栈，若分配不成功则错误处理。
        elements = new Type[m];                                    //创建栈的数组空间
        assert ( elements != NULL );                               //断言：动态存储分配成功与否
    }

    template <class Type> void DblStack<Type> :: DblPush ( const Type& x, int i ) {
    //如果 IsFull ( )，则报错；否则把 x 插入到栈 i 的栈顶
        assert ( !IsFull ( ) );                                    //断言：栈满则出错处理，停止执行
        if ( i == 0 ) elements[ ++top[0] ] = x;                    //栈 0 情形:栈顶指针先加 1，然后按此地址进栈
        else elements[−−top[1] ] = x;                              //栈 1 情形:栈顶指针先减 1，然后按此地址进栈
    }

    template <class Type> int DblStack<Type> :: DblPop ( int i ) {
    //如果 IsEmpty ( i )，则不执行退栈，返回 0；否则退掉位于栈 i 栈顶的元素，返回 1
        if ( IsEmpty ( i ) ) return 0;                             //判栈空否，若栈空则函数返回 0
        if ( i == 0 ) top[0]−−;                                    //栈 0 情形：栈顶指针减 1
        else top[1]++;                                             //栈 1 情形：栈顶指针加 1
        return 1;
    }

    template <class Type> Type * DblStack<Type> :: DblGetTop ( int i ) {
    //若栈不空则函数返回该栈栈顶元素的地址。
        if ( IsEmpty ( int i ) ) return NULL;                      //判栈 i 空否，若栈空则函数返回空指针
        return& elements[ top[i] ];                                //返回栈顶元素的值
    }

    template <class Type> void MakeEmpty ( int i ) {
       if ( i == 0 ) top[0] = bot[0] = −1;
       else top[1] = bot[1] = m;
    }
```

4−5 写出下列中缀表达式的后缀形式：

(1) A * B * C

(2) − A + B − C + D

(3) A* − B + C

(4) (A + B) * D + E / (F + A * D) + C

(5) A && B||!(E > F)　/*注：按 C++的优先级*/

(6) !(A && !( (B < C)||(C > D) ) )||(C < E)

【解答】

(1) A B * C *

(2) A – B + C – D +

(3) A B – * C +

(4) A B + D * E F A D * + / + C +

(5) A B **&&** E F > ! | |

(6) A B C < C D > | | ! **&&** ! C E < | |

4-6 根据课文中给出的优先级，回答以下问题：

(1) 在函数 postfix 中，如果表达式 e 含有 n 个运算符和分界符，问栈中最多可存入多少个元素？

(2) 如果表达式 e 含有 n 个运算符，且括号嵌套的最大深度为 6 层，问栈中最多可存入多少个元素？

【解答】

(1) 在函数 postfix 中，如果表达式 e 含有 n 个运算符和分界符，则可能的运算对象有 n+1 个。因此在利用后缀表达式求值时所用到的运算对象栈中最多可存入 n+1 个元素。

(2) 同上。

4-7 设表达式的中缀表示为 a * x – b / x ↑ 2，试利用栈将它改为后缀表示 ax * bx2 ↑ / –。写出转换过程中栈的变化。

【解答】

若设当前扫描到的运算符 ch 的优先级为 icp(ch)，该运算符进栈后的优先级为 isp(ch)，则可规定各个算术运算符的优先级如下表所示。

| 运算符 | ; | ( | ^ | *, /, % | +, – | ) |
|---|---|---|---|---|---|---|
| isp | 0 | 1 | 7 | 5 | 3 | 8 |
| icp | 0 | 8 | 6 | 4 | 2 | 1 |

isp 也叫做栈内(**in stack priority**)优先数，icp 也叫做栈外(**in coming priority**)优先数。当刚扫描到的运算符 ch 的 icp(ch)大于 isp(stack)时，则 ch 进栈；当刚扫描到的运算符 ch 的 icp(ch)小于 isp(stack)时，则位于栈顶的运算符退栈并输出。从表中可知，icp(“(”)最高，但当“(”进栈后，isp(“(”)变得极低。其它运算符进入栈中后优先数都升 1，这样可体现在中缀表达式中相同优先级的运算符自左向右计算的要求。运算符优先数相等的情况只出现在括号配对“)”或栈底的“**;**”号与输入流最后的“**;**”号配对时。前者将连续退出位于栈顶的运算符，直到遇到“(”为止。然后将“(”退栈以对消括号，后者将结束算法。

| 步序 | 扫描项 | 项类型 | 动　　作 | 栈的变化 | 输　出 |
|---|---|---|---|---|---|
| 0 | | | ☞ ';' 进栈, 读下一符号 | ; | |
| 1 | a | 操作数 | ☞ 直接输出, 读下一符号 | ; | A |
| 2 | * | 操作符 | ☞ isp ( ';' ) < icp ( '*' ), 进栈, 读下一符号 | ; * | A |
| 3 | x | 操作数 | ☞ 直接输出, 读下一符号 | ; * | a x |
| 4 | – | 操作符 | ☞ isp ( '*' ) > icp ( '–' ), 退栈输出 | ; | a x * |
| | | | ☞ isp ( ';' ) < icp ( '–' ), 进栈, 读下一符号 | ; – | a x * |
| 5 | b | 操作数 | ☞ 直接输出, 读下一符号 | ; – | a x * b |
| 6 | / | 操作符 | ☞ isp ( '–' ) < icp ( '/' ), 进栈, 读下一符号 | ; –/ | a x * b |
| 7 | x | 操作数 | ☞ 直接输出, 读下一符号 | ; –/ | a x * b x |
| 8 | ↑ | 操作符 | ☞ isp ( '/' ) < icp ( '↑' ), 进栈, 读下一符号 | ; –/ ↑ | a x * b x |

| | | | | | |
|---|---|---|---|---|---|
| 9 | 2 | 操作数 | ☞ 直接输出，读下一符号 | ; −/ ↑ | a x * b x 2 |
| 10 | ; | 操作符 | ☞ isp ( ' ↑ ' ) > icp ( ' ; ' ), 退栈输出 | ; −/ | a x * b x 2 ↑ |
| | | | ☞ isp ( ' / ' ) > icp ( ' ; ' ), 退栈输出 | ; − | a x * b x 2 ↑ / |
| | | | ☞ isp ( ' − ' ) > icp ( ' ; ' ), 退栈输出 | ; | a x * b x 2 ↑ / − |
| | | | ☞ 结束 | | |

4-8 试利用运算符优先数法，画出对如下中缀算术表达式求值时运算符栈和运算对象栈的变化。

a + b * (c − d) − e ↑ f / g

【解答】

设在表达式计算时各运算符的优先规则如上一题所示。因为直接对中缀算术表达式求值时必须使用两个栈，分别对运算符和运算对象进行处理，假设命名运算符栈为 OPTR (operator 的缩写)，运算对象栈为 OPND(operand 的缩写)，下面给出对中缀表达式求值的一般规则：

(1) 建立并初始化 OPTR 栈和 OPND 栈，然后在 OPTR 栈中压入一个“;”

(2) 从头扫描中缀表达式，取一字符送入 ch。

(3) 当 ch 不等于“;”时，执行以下工作，否则结束算法。此时在 OPND 栈的栈顶得到运算结果。

① 如果 ch 是运算对象，进 OPND 栈，从中缀表达式取下一字符送入 ch；

② 如果 ch 是运算符，比较 ch 的优先级 icp(ch)和 OPTR 栈顶运算符 isp(OPTR)的优先级：

☞ 若 icp(ch) > isp(OPTR)，则 ch 进 OPTR 栈，从中缀表达式取下一字符送入 ch；

☞ 若 icp(ch) < isp(OPTR)，则从 OPND 栈退出一个运算符作为第 2 操作数 a2，再退出一个运算符作为第 1 操作数 a1，从 OPTR 栈退出一个运算符 θ 形成运算指令 (a1) θ (a2)，执行结果进 OPND 栈；

☞ 若 icp(ch) == isp(OPTR) 且 ch == “)”，则从 OPTR 栈退出栈顶的“(”，对消括号，然后从中缀表达式取下一字符送入 ch；

根据以上规则，给出计算 a + b * (c − d) − e ↑ f / g 时两个栈的变化。

| 步序 | 扫描项 | 项类型 | 动作 | OPND 栈变化 | OPTR 栈变化 |
|---|---|---|---|---|---|
| 0 | | | ☞ OPTR 栈与 OPND 栈初始化，‘;’ 进 OPTR 栈，取第一个符号 | | ; |
| 1 | a | 操作数 | ☞ a 进 OPND 栈，取下一符号 | a | ; |
| 2 | + | 操作符 | ☞ icp (‘ + ’) > isp (‘ ; ’), 进 OPTR 栈，取下一符号 | a | ; + |
| 3 | b | 操作数 | ☞ b 进 OPND 栈，取下一符号 | a b | ; + |
| 4 | * | 操作符 | ☞ icp (‘ * ’) > isp (‘ + ’), 进 OPTR 栈，取下一符号 | a b | ; + * |
| 5 | ( | 操作符 | ☞ icp (‘ ( ’) > isp (‘ * ’), 进 OPTR 栈，取下一符号 | a b | ; + * ( |
| 6 | c | 操作数 | ☞ c 进 OPND 栈，取下一符号 | a b c | ; + * ( |
| 7 | − | 操作符 | ☞ icp (‘ − ’) > isp (‘ ( ’), 进 OPTR 栈，取下一符号 | a b | ; + * ( − |
| 8 | d | 操作数 | ☞ d 进 OPND 栈，取下一符号 | a b c d | ; + * ( − |
| 9 | ) | 操作符 | ☞ icp (‘ ) ’) < isp (‘ − ’), 退 OPND 栈 ‘d’，退 OPND 栈 ‘c’，退 OPTR 栈 ‘−’，计算 c – d → $s_1$，结果进 OPND 栈 | a b $s_1$ | ; + * ( |
| 10 | 同上 | 同上 | ☞ icp (‘ ) ’) == isp (‘ ( ’), 退 OPTR 栈‘(’，对消括号，取下一符号 | a b $s_1$ | ; + * |

| | | | | | |
|---|---|---|---|---|---|
| 11 | – | 操作符 | ☞ icp (‘ – ’) < isp (‘ * ’), 退 OPND 栈 ‘$s_1$’, 退 OPND 栈 ‘b’, 退 OPTR 栈 ‘*’, 计算 b * $s_1$ → $s_2$, 结果进 OPND 栈 | a $s_2$ | ; + |
| 12 | 同上 | 同上 | ☞ icp (‘ – ’) < isp (‘ + ’), 退 OPND 栈 ‘$s_2$’, 退 OPND 栈 ‘a’, 退 OPTR 栈 ‘+’, 计算 a * $s_2$ → $s_3$, 结果进 OPND 栈 | $s_3$ | ; |
| 13 | 同上 | 同上 | ☞ icp (‘ – ’) > isp (‘ ; ’), 进 OPTR 栈, 取下一符号 | $s_3$ | ; – |
| 14 | e | 操作数 | ☞ e 进 OPND 栈, 取下一符号 | $s_3$ e | ; – |
| 15 | ↑ | 操作符 | ☞ icp (‘ ↑ ’) > isp (‘ – ’), 进 OPTR 栈, 取下一符号 | $s_3$ e | ; – ↑ |
| 16 | f | 操作数 | ☞ f 进 OPND 栈, 取下一符号 | $s_3$ e f | ; – ↑ |
| 17 | / | 操作符 | ☞ icp (‘ / ’) < isp (‘ ↑ ’), 退 OPND 栈 ‘f’, 退 OPND 栈 ‘e’, 退 OPTR 栈 ‘ ↑ ’, 计算 e ↑ f → $s_4$, 结果进 OPND 栈 | $s_3$ $s_4$ | ; – |
| 18 | 同上 | 同上 | ☞ icp (‘ / ’) > isp (‘ – ’), 进 OPTR 栈, 取下一符号 | $s_3$ $s_4$ | ; – / |
| 19 | g | 操作数 | ☞ g 进 OPND 栈, 取下一符号 | $s_3$ $s_4$ g | ; – / |
| 20 | ; | 操作符 | ☞ icp (‘ ; ’) < isp (‘ / ’), 退 OPND 栈 ‘g’, 退 OPND 栈 ‘$s_4$’, 退 OPTR 栈 ‘/’, 计算 $s_4$ / g → $s_5$, 结果进 OPND 栈 | $s_3$ $s_5$ | ; – |
| 21 | 同上 | 同上 | ☞ icp (‘ ; ’) < isp (‘ – ’), 退 OPND 栈 ‘$s_5$’, 退 OPND 栈 ‘$s_3$’, 退 OPTR 栈 ‘ – ’, 计算 $s_3$ – $s_5$ → $s_6$, 结果进 OPND 栈 | $s_6$ | ; |
| 22 | 同上 | 同上 | ☞ icp (‘ ; ’) == isp (‘ ; ’), 退 OPND 栈 ‘$s_6$’, 结束 | | ; |

4–9 假设以数组 Q[m]存放循环队列中的元素, 同时以 rear 和 length 分别指示环形队列中的队尾位置和队列中所含元素的个数。试给出该循环队列的队空条件和队满条件, 并写出相应的插入(enqueue)和删除(dlqueue)元素的操作。

【解答】

循环队列类定义

```
#include <assert.h>
template <class Type> class Queue {                    //循环队列的类定义
public:
    Queue ( int=10 );
    ~Queue ( ) { delete [ ] elements; }
    void EnQueue ( Type & item );
    Type DeQueue ( );
    Type GetFront ( );
    void MakeEmpty ( ) { length = 0; }                 //置空队列
    int IsEmpty ( ) const { return length == 0; }      //判队列空否
    int IsFull ( ) const { return length == maxSize; } //判队列满否
private:
    int rear, length;                                  //队尾指针和队列长度
    Type *elements;                                    //存放队列元素的数组
    int maxSize;                                       //队列最大可容纳元素个数
}
```

构造函数

```
template <class Type>
Queue<Type>:: Queue ( int sz ) : rear (maxSize−1), length (0), maxSize (sz) {
                                        //建立一个最大具有 maxSize 个元素的空队列。
    elements = new Type[maxSize];       //创建队列空间
    assert ( elements != 0 );           //断言：动态存储分配成功与否
}
```

插入函数

```
template<class Type>
void Queue<Type> :: EnQueue ( Type &item ) {
    assert ( ! IsFull ( ) );            //判队列是否不满，满则出错处理
    length++;                           //长度加 1
    rear = ( rear +1) % maxSize;        //队尾位置进 1
    elements[rear] = item;              //进队列
}
```

删除函数

```
template<class Type>
Type Queue<Type> :: DeQueue ( ) {
    assert ( ! IsEmpty ( ) );           //判断队列是否不空，空则出错处理
    length−−;                           //队列长度减 1
    return elements[(rear−length+maxSize) % maxSize];        //返回原队头元素值
}
```

读取队头元素值函数

```
template<class Type>
Type Queue<Type> :: GetFront ( ) {
    assert ( ! IsEmpty ( ) );
    return elements[(rear−length+1+maxSize) % maxSize];      //返回队头元素值
}
```

4−10 假设以数组 Q[m]存放循环队列中的元素，同时设置一个标志 tag，以 tag == 0 和 tag == 1 来区别在队头指针(front)和队尾指针(rear)相等时，队列状态为“空”还是“满”。试编写与此结构相应的插入(enqueue)和删除(dlqueue)算法。

【解答】

循环队列类定义

```
#include <assert.h>
template <class Type> class Queue {                  //循环队列的类定义
public:
    Queue ( int=10 );
    ~Queue ( ) { delete [ ] Q; }
    void EnQueue ( Type & item );
    Type DeQueue ( );
    Type GetFront ( );
    void MakeEmpty ( ) { front = rear = tag = 0; }   //置空队列
    int IsEmpty ( ) const { return front == rear && tag == 0; } //判队列空否
```

```
    int IsFull ( ) const { return front == rear && tag == 1; }    //判队列满否
  private:
    int rear, front, tag;                          //队尾指针、队头指针和队满标志
    Type *Q;                                       //存放队列元素的数组
    int m;                                         //队列最大可容纳元素个数
  }
```

构造函数

```
  template <class Type>
  Queue<Type>:: Queue ( int sz ) : rear (0), front (0), tag(0), m (sz) {
                                                   //建立一个最大具有 m 个元素的空队列。
    Q = new Type[m];                               //创建队列空间
    assert ( Q != 0 );                             //断言：动态存储分配成功与否
  }
```

插入函数

```
  template<class Type>
  void Queue<Type> :: EnQueue ( Type &item ) {
      assert ( ! IsFull ( ) );                     //判队列是否不满，满则出错处理
      rear = ( rear + 1 ) % m;                     //队尾位置进 1，队尾指针指示实际队尾位置
      Q[rear] = item;                              //进队列
      tag = 1;                                     //标志改 1，表示队列不空
  }
```

删除函数

```
  template<class Type>
  Type Queue<Type> :: DeQueue ( ) {
      assert ( ! IsEmpty ( ) );                    //判断队列是否不空，空则出错处理
      front = ( front + 1 ) % m;                   //队头位置进 1，队头指针指示实际队头的前一位置
      tag = 0;                                     //标志改 0，表示栈不满
      return Q[front];                             //返回原队头元素的值
  }
```

读取队头元素函数

```
  template<class Type>
  Type Queue<Type> :: GetFront ( ) {
      assert ( ! IsEmpty ( ) );                    //判断队列是否不空，空则出错处理
      return Q[(front + 1) % m];                   //返回队头元素的值
  }
```

4-11 若使用循环链表来表示队列，p 是链表中的一个指针。试基于此结构给出队列的插入(enqueue)和删除(dequeue)算法，并给出 p 为何值时队列空。

【解答】

链式队列的类定义

```
  template <class Type> class Queue;                    //链式队列类的前视定义

  template <class Type> class QueueNode {               //链式队列结点类定义
  friend class Queue<Type>;
```

```
private:
    Type data;                                          //数据域
    QueueNode<Type> *link;                              //链域
public:
    QueueNode ( Type d = 0, QueueNode *l = NULL ) : data (d), link (l) { }      //构造函数
};

template <class Type> class Queue {                     //链式队列类定义
public:
    Queue ( ) : p ( NULL ) { }                          //构造函数
    ~Queue ( );                                         //析构函数
    void EnQueue ( const Type & item );                 //将 item 加入到队列中
    Type DeQueue ( );                                   //删除并返回队头元素
    Type GetFront ( );                                  //查看队头元素的值
    void MakeEmpty ( );                                 //置空队列，实现与~Queue ( ) 相同
    int IsEmpty ( ) const { return p == NULL; }         //判队列空否
private:
    QueueNode<Type> *p;                                 //队尾指针（在循环链表中）
};
```

队列的析构函数

```
template <class Type> Queue<Type>::~Queue ( ) {         //队列的析构函数
    QueueNode<Type> *s;
    while ( p != NULL ) { s = p;   p = p->link;   delete s; }   //逐个删除队列中的结点
}
```

队列的插入函数

```
template <class Type> void Queue<Type>::EnQueue ( const Type & item ) {
    if ( p == NULL ) {                                  //队列空，新结点成为第一个结点
        p = new QueueNode<Type> ( item, NULL );   p->link = p;
    }
    else  {                                             //队列不空，新结点链入 p 之后
        QueueNode<Type> *s = new QueueNode<Type> ( item, NULL );
        s->link = p->link;   p = p->link = s;           //结点 p 指向新的队尾
    }
}
```

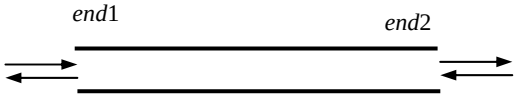
队列的删除函数

```
template <class Type> Type Queue<Type>::DeQueue ( ) {
    if ( p == NULL ) { cout << "队列空，不能删除！" << endl;   return 0; }
    QueueNode<Type> *s = p;                 //队头结点为 p 后一个结点
    p->link = s->link;                      //重新链接，将结点 s 从链中摘下
    Type retvalue = s->data;    delete s;   //保存原队头结点中的值，释放原队头结点
    return retvalue;                        //返回数据存放地址
}
```

队空条件 p == NULL。

4-12 若将一个双端队列顺序表示在一维数组 V[m]中，两个端点设为 end1 和 end2，并组织成一个循环队列。试写出双端队列所用指针 end1 和 end2 的初始化条件及队空与队满条件，并编写基于此结构的相应的插入(enqueue)新元素和删除(dlqueue)算法。

【解答】




初始化条件 end1 = end2 = 0;

队空条件 end1 = end2;

队满条件 ( end1 + 1 ) % m = end2;　　//设 end1 端顺时针进栈，end2 端逆时针进栈

循环队列类定义

```
#include <assert.h>
template <class Type> class DoubleQueue {                  //循环队列的类定义
public:
    DoubleQueue ( int=10 );
    ~DoubleQueue ( ) { delete [ ] V; }
    void EnQueue ( Type & item, const int end );
    Type DeQueue (const int end );
    Type GetFront (const int end );
    void MakeEmpty ( ) { end1 = end2 = 0; }                //置空队列
    int IsEmpty ( ) const { return end1 == end2; }         //判两队列空否
    int IsFull ( ) const { return (end1+1) % m == end2; }//判两队列满否
private:
    int end1, end2;                                        //队列两端的指针
    Type *V;                                               //存放队列元素的数组
    int m;                                                 //队列最大可容纳元素个数
}
```

构造函数

```
template <class Type>
DoubleQueue<Type>:: DoubleQueue ( int sz ) : end1 (0), end2 (0), m (sz) {
//建立一个最大具有 m 个元素的空队列。
    V = new Type[m];                                       //创建队列空间
    assert ( V != 0 );                                     //断言：动态存储分配成功与否
}
```

插入函数

```
template<class Type>
void DoubleQueue<Type> :: EnQueue ( Type &item,    const int end ) {
    assert ( !IsFull ( ) );
    if ( end == 1 ) {
        end1 = ( end1 + 1 ) % m;                           //end1 端指针先进 1, 再按指针进栈
        V[end1] = item;                                    //end1 指向实际队头位置
    }
    else {
        V[end2] = item;                                    //end2 端先进队列, 指针再进 1
        end2 = ( end2 - 1 + m ) % m;                       //end2 指向实际队头的下一位置
    }
}
```

删除函数

```
template<class Type>
Type DoubleQueue<Type> :: DeQueue ( const int end ) {
    assert ( !IsEmpty ( ) );
    Type& temp;
    if ( end == 1 ) {
        temp = V[end1];                          //先保存原队头元素的值, end1 端指针退 1
        end1 = ( end1 + m – 1 ) % m;
    }
    else {
        end2 = ( end2 + 1 ) % m;
        temp = V[end2];                          //end2 端指针先退 1。再保存原队头元素的值
    }
    return temp;
}
```

读取队头元素的值

```
template<class Type>
Type DoubleQueue<Type> :: GetFront ( const int end ) {
    assert ( !IsEmpty ( ) );
    Type& temp;
    if ( end == 1 ) return V[end1];              //返回队头元素的值
    else return V[(end2+1) % m];
}
```

4–13 设用链表表示一个双端队列，要求可在表的两端插入，但限制只能在表的一端删除。试编写基于此结构的队列的插入(enqueue)和删除(dequeue)算法，并给出队列空和队列满的条件。

【解答】

链式双端队列的类定义

```
template <class Type> class DoubleQueue;                    //链式双端队列类的前视定义

template <class Type> class DoubleQueueNode {               //链式双端队列结点类定义
friend class DoubleQueue<Type>;
private:
    Type data;                                              //数据域
    DoubleQueueNode<Type> *link;                            //链域
public:
    DoubleQueueNode (Type d = 0, DoubleQueueNode *l = NULL) : data (d), link (l) { }  // 构造函数
};

template <class Type> class DoubleQueue {                   //链式双端队列类定义
public:
    DoubleQueue ( );                                        //构造函数
```

```
        ~DoubleQueue ( );                                   //析构函数
        void EnDoubleQueue1 ( const Type& item );           //从队列 end1 端插入
        void EnDoubleQueue2 ( const Type& item );           //从队列 end2 端插入
        Type DeDoubleQueue ( );                             //删除并返回队头 end1 元素
        Type GetFront ( );                                  //查看队头 end1 元素的值
        void MakeEmpty ( );                                 //置空队列
        int IsEmpty ( ) const { return end1 == end1->link; }  //判队列空否
    private:
        QueueNode<Type> *end1, *end2;//end1 在链头，可插可删; end2 在链尾，可插不可删
    };
```

队列的构造函数

```
    template<class Type> doubleQueue<Type> :: doubleQueue ( ) {     //构造函数
        end1 = end2 = new DoubleQueueNode<Type>( );                 //创建循环链表的表头结点
        assert ( !end1 || !end2 );
        end1->link = end1;
    }
```

队列的析构函数

```
    template <class Type> Queue<Type> :: ~Queue ( ) {               //队列的析构函数
        QueueNode<Type> *p;                                         //逐个删除队列中的结点，包括
表头结点
        while ( end1 != NULL ) { p = end1;   end1 = end1->link;   delete p; }
    }
```

队列的插入函数

```
    template<class Type>                                            //从队列 end1 端插入
    void DoubleQueue<Type> :: EnDoubleQueue1 ( const Type& item ) {
        if ( end1 == end1->link )                                   //队列空，新结点成为第一个结点
            end2 = end1->link = new DoubleQueueNode<Type> ( item, end1 );
        else                                                        //队列不空，新结点链入 end1 之后
            end1->link = new DoubleQueueNode<Type> ( item, end1->link );
    }

    template <class Type>                                           //从队列 end2 端插入
    void DoubleQueue<Type> :: EnDoubleQueue2 ( const Type& item ) {
        end2 = end2->link = new DoubleQueueNode<Type> ( item, end1 );
    }
```

队列的删除函数

```
    template <class Type>
    Type DoubleQueue<Type> :: DeDoubleQueue ( ) {
        if ( IsEmpty ( ) ) return { cout << "队列空，不能删除！" << endl;   return 0; }
        DoubleQueueNode<Type> *p = end1->link ;                 //被删除结点
        end1->link = p->link;                                   //重新链接
        Type retvalue = p->data;   delete p;                    //删除 end1 后的结点 p
        if ( IsEmpty ( ) ) end2 = end1;
        return retvalue;
```

```
}
```

读取队列 end1 端元素的内容

```
template <class Type> Type DoubleQueue<Type> :: GetFront ( ) {
    assert ( !IsEmpty ( ) );
    return end1->link->data;
}
```

置空队列

```
template <class Type> void Queue<Type>:: MakeEmpty ( ) {
    QueueNode<Type> *p;                          //逐个删除队列中的结点，包括表头结点
    while ( end1 != end1->link ) { p = end1;   end1 = end1->link;   delete p; }
}
```

4-14 试建立一个继承结构，以栈、队列和优先级队列为派生类，建立它们的抽象基类——Bag 类。写出各个类的声明。统一命名各派生类的插入操作为 Add，删除操作为 Remove，存取操作为 Get 和 Put，初始化操作为 MakeEmpty，判空操作为 Empty，判满操作为 Full，计数操作为 Length。

【解答】

Bag 类的定义

```
template<class Type> class Bag {
public:
    Bag ( int sz = DefaultSize );                    //构造函数
    virtual ~Bag ( );                                //析构函数
    virtual void Add ( const Type& item );           //插入函数
    virtual Type *Remove ( );                        //删除函数
    virtual int IsEmpty ( ) { return top == -1; }    //判空函数
    virtual int IsFull ( ) { return top == maxSize - 1; }   //判满函数
private:
    virtual void Empty ( ) { cout << "Data Structure is empty." << endl; }
    virtual void Full ( ) { cerr << "DataStructure is full." << endl; }
    Type *elements;                                  //存储数组
    int maxSize;                                     //数组的大小
    int top;                                         //数组当前元素个数
};
```

Bag 类的构造函数

```
template<class Type> Bag<Type> :: Bag ( int MaxBagSize ) : MaxSize ( MaxBagSize ) {
    elements = new Type [ MaxSize ];
    top = -1;
}
```

Bag 类的析构函数

```
template<class Type> Bag<Type> :: ~Bag ( ) {
    delete [ ] elements;
}
```

Bag 类的插入函数

```
template<class Type> void Bag<Type> :: Add ( const Type & item ) {
```

```
        if ( IsFull ( ) ) Full ( );
        else elements [ ++top ] = item;
    }
```

Bag 类的删除函数

```
    template <class Type> Type *Bag<Type> :: Remove ( ) {
        if ( IsEmpty ( ) ) { Empty ( );   return NULL; }
        Type & x = elements [0];                          //保存被删除元素的值
        for ( int i = 0; i < top; i++ )                   //后面元素填补上来
            elements [i] = elements [ i+1];
        top--;
        return &x;
    }
```

栈的类定义（继承 Bag 类）

```
    template<class Type> class Stack : public Bag {
    public:
        Stack ( int sz = DefaultSize );                   //构造函数
        ~Stack ( );                                       //析构函数
        Type *Remove ( );                                 //删除函数
    };
```

栈的构造函数

```
    template<class Type> Stack<Type> :: Stack ( int sz ) : Bag ( sz ) { }
    //栈的构造函数 Stack 将调用 Bag 的构造函数
```

栈的析构函数

```
    template<class Type> Stack<Type> :: ~Stack ( ) { }
    //栈的析构函数将自动调用 Bag 的析构函数, 以确保数组 elements 的释放
```

栈的删除函数

```
    template<class Type> Type * Stack<Type> :: Remove ( ) {
        if ( IsEmpty ( ) ) { Empty ( ); return NULL; }
        Type& x = elements [ top-- ];
        return &x;
    }
```

队列的类定义（继承 Bag 类）

```
    template<class Type> class Queue : public Bag {
    public:
        Queue ( int sz = DefaultSize );                   //构造函数
        ~Queue ( );                                       //析构函数
    };
```

队列的构造函数

```
    template<class Type> Queue<Type> :: Queue ( int sz ) : Bag ( sz ) { }
    //队列的构造函数 Queue 将调用 Bag 的构造函数
```

优先级队列的类定义（继承 Bag 类）

```
    template <class Type> class PQueue : public Bag {
    public:
        PQueue ( int sz = DefaultSize );                  //构造函数
```

```
    ~PQueue ( ) { }                                    //析构函数
    Type *PQRemove ( );                                //删除函数
}
```

优先级队列的构造函数

```
template <class Type> PQueue<Type> :: PQueue ( int sz ) : Bag ( sz ) { }
//建立一个最大具有 sz 个元素的空优先级队列。top = −1。
```

优先级队列的删除函数

```
template <class Type> Type *PQueue<Type> :: Remove ( ) {
//若优先级队列不空则函数返回该队列具最大优先权(值最小)元素的值，同时将该元素删除。
    if ( IsEmpty ( ) ) { Empty ( ); return NULL; }
    Type& min = elements[0];                  //假设 elements[0]是最小值，继续找最小值
    int minindex = 0;
    for ( int i = 1; i <= top; i++ )
        if ( elements[i] < min ) { min = elements[i];     minindex = i; }
    elements[minindex] = elements[top];       //用最后一个元素填补要取走的最小值元素
    top−−;
    return& min;                              //返回最小元素的值
}
```

4−15 试利用优先级队列实现栈和队列。

【解答】

```
template <class Type> class PQueue;                    //前视的类定义

template <class Type> class PQueueNode {               //优先级队列结点类的定义
friend class PQueue<Type>;                             //PQueue 类作为友元类定义
public:
    PQueueNode ( Type& value, int newpriority, PQueue<Type> * next )
            : data ( value ), priority ( newpriority ), link ( next ) { }      //构造函数
    virtual Type GetData ( ) { return data; }                                  //取得结点数据
    virtual int GetPriority ( ) { return priority; }                           //取得结点优先级
    virtual PQueueNode<Type> * GetLink ( ) { return link; }                    //取得下一结点地址
    virtual void SetData ( Type& value ) { data = value; }                     //修改结点数据
    virtual void SetPriority ( int newpriority ) { priority = newpriority; }  //修改结点优先级
    virtual void SetLink ( PQueueNode<Type> * next ) { link = next; }          //修改指向下一结点的指
针
private:
    Type data;                                         //数据
    int priority;                                      //优先级
    ListNode<Type> *link;                              //链指针
};

template <class Type> class PQueue {                   //优先级队列的类定义
public:
    PQueue ( ) : front ( NULL ), rear ( NULL ) { }     //构造函数
```

```
    virtual ~PQueue ( ) { MakeEmpty ( ); }                    //析构函数
    virtual void Insert ( Type& value, int newpriority );     //插入新元素 value 到队尾
    virtual Type Remove ( );                                  //删除队头元素并返回
    virtual Type Get ( );                                     //读取队头元素的值
    virtual void MakeEmpty ( );                               //置空队列
    virtual int IsEmpty ( ) { return front == NULL; }         //判队列空否
private:
    PQueueNode<Type> *front, *rear;                           //队头指针，队尾指针
};

template<class Type>
void PQueue<Type> :: MakeEmpty ( ) {                          //将优先级队列置空
    PQueueNode<Type> *q;
    while ( front != NULL )                                   //链不空时，删去链中所有结点
        { q = front;    front = front->link;    delete q; }   //循链逐个删除
    rear = NULL;                                              //队尾指针置空
}

template<class Type>
void PQueue<Type> :: Insert ( Type& value, int newpriority ) {      //插入函数
    PQueueNode<Type> *q = new PQueueNode ( value, newpriority, NULL );
    if ( IsEmpty ( ) ) front = rear = q;                      //队列空时新结点为第一个结点
    else {
        PQueueNode<Type> *p = front, *pr = NULL;              //寻找 q 的插入位置
        while ( p != NULL && p->priority >= newpriority )     //队列中按优先级从大到小链接
            { pr = p;    p = p->link; }
        if ( pr == NULL ) { q->link = front;    front = q; }  //插入在队头位置
        else {    q->link = p;    pr->link = q;               //插入在队列中部或尾部
                  if ( pr == rear ) rear = q;
             }
}

template<class Type> Type PQueue<Type> :: Remove ( ) {        //删除队头元素并返回
    if ( IsEmpty ( ) ) return NULL;
    PQueueNode<Type> *q = front;    front = front->link;      //将队头结点从链中摘下
    Type &retvalue = q->data;    delete q;
    if ( front == NULL ) rear = NULL;
    return& retvalue;
}

template<class Type> Type PQueue<Type> :: Get ( ) {           //读取队头元素的值
    if ( IsEmpty ( ) ) return NULL;
    else return front->data;
}
```

(1) 栈的定义与实现

```
template <class Type> class Stack : public PQueue {                    //栈类定义
public:
    Stack ( ) : front ( NULL ), rear ( NULL ) { }                      //构造函数
    void Insert ( Type & value );                                      //插入新元素 value 到队尾
}
template<class Type>
void Stack<Type> :: Insert ( Type& value ) {                           //插入函数
    PQueueNode<Type> * q = new PQueueNode ( value, 0, NULL );
    if ( IsEmpty ( ) ) front = rear = q;                               //栈空时新结点为第一个结点
    else { q->link = front;    front = q; }                            //插入在前端
}
```

(2) 队列的定义与实现

```
template <class Type> class Queue : public PQueue {                    //队列类定义
public:
    Queue ( ) : front ( NULL ), rear ( NULL ) { }                      //构造函数
    void Insert ( Type& value );                                       //插入新元素 value 到队尾
}
template<class Type>
void Queue<Type> :: Insert ( Type & value ) {                          //插入函数
    PQueueNode<Type>* q = new PQueueNode ( value, 0, NULL );
    if ( IsEmpty ( ) ) front = rear = q;                               //队列空时新结点为第一个结点
    else rear = rear->link = q;                                        //插入在队尾位置
}
```

4-15 所谓回文，是指从前向后顺读和从后向前倒读都一样的不含空白字符的串。例如 did，madamimadam，pop 即是回文。试编写一个算法，以判断一个串是否是回文。

【解答 1】

将字符串中全部字符进栈，然后将栈中的字符逐个与原字符串中的字符进行比较。算法如下：

```
int palindrome ( char A[ ], int n ) {
    stack<char> st (n+1);
    int yes = 1, i = 0;
    while ( A[i] != "\0" ) { st.Push ( A[i] );   i++; }                //扫描字符串，所有字符进栈
    i = 0;
    while ( A[i] != "\0" )                                             //比较字符串
      if ( A[i] == st.GetTop ( ) ) { st.Pop ( );   i++; }
      else { yes = 0;   break; }
    return yes;
}
```

【解答 2】

采用递归算法，判断从 s 到 e 中的字符串是否回文，通过函数返回是或不是。

```
int palindrome ( char A[ ], int s, int e ) {
    if ( A[s] != A[e] ) return 0;
```

```
    else if ( s > e ) return 1;
    else palindrome ( A, s+1, e−1 );
}
```

4−16 设有一个双端队列，元素进入该队列的顺序是 1, 2, 3, 4。试分别求出满足下列条件的输出序列。

(1) 能由输入受限的双端队列得到，但不能由输出受限的双端队列得到的输出序列；

(2) 能由输出受限的双端队列得到，但不能由输入受限的双端队列得到的输出序列；

(3) 既不能由输入受限的双端队列得到，又不能由输出受限的双端队列得到的输出序列。

【解答】

允许在一端进行插入和删除，但在另一端只允许插入的双端队列叫做输出受限的双端队列，允许在一端进行插入和删除，但在另一端只允许删除的双端队列叫做输入受限的双端队列。

输出受限双端队列不能得到的输出序列有：

4 1 3 2　　4 2 3 1

输人受限双端队列不能得到的输出序列有：

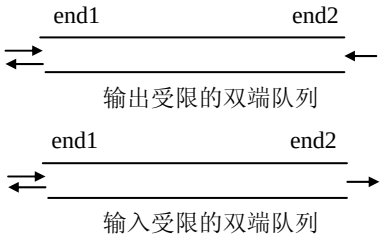
4 2 1 3　　4 2 3 1

所以有：

(1) 4 1 3 2

(2) 4 2 1 3

(3) 4 2 3 1



# 第五章 递归与广义表

5-1 已知 A[n]为整数数组，试写出实现下列运算的递归算法：

(1) 求数组 A 中的最大整数。

(2) 求 n 个整数的和。

(3) 求 n 个整数的平均值。

【解答】

```
#include <iostream.h>
class RecurveArray {                        //数组类声明
private:
    int *Elements;                          //数组指针
    int ArraySize;                          //数组尺寸
    int CurrentSize;                        //当前已有数组元素个数
public :
    RecurveArray ( int MaxSize =10 ) :
    ArraySize ( MaxSize ), Elements ( new int[MaxSize] ){ }
    ~RecurveArray ( ) { delete [ ] Elements; }
    void InputArray();                      //输入数组的内容
    int MaxKey ( int n );                   //求最大值
    int Sum ( int n );                      //求数组元素之和
    float Average ( int n );                //求数组元素的平均值
};

void RecurveArray :: InputArray ( ){                //输入数组的内容
    cout << "Input the number of Array: \n";
    for ( int i = 0; i < ArraySize; i++ ) cin >> Elements[i];
}

int RecurveArray :: MaxKey ( int n ) {              //递归求最大值
    if ( n == 1 ) return Elements[0];
    int temp = MaxKey ( n - 1 );
    if ( Elements[n-1] > temp ) return Elements[n-1];
    else return temp;
}

int RecurveArray :: Sum ( int n ) {                 //递归求数组之和
    if ( n == 1) return Elements[0];
    else return Elements[n-1] + Sum (n-1);
}

float RecurveArray :: Average ( int n ) {           //递归求数组的平均值
    if ( n == 1) return (float) Elements[0];
    else return ( (float) Elements[n-1] + ( n - 1) * Average ( n - 1 ) ) / n;
}
int main ( int argc,    char* argv [ ] ) {
int size = -1;
```

```
    cout << "No. of the Elements : ";
    while ( size < 1 ) cin >> size;
    RecurveArray ra ( size );
    ra.InputArray();
    cout<< "\nThe max is:    " << ra.MaxKey ( ra.MaxSize ) << endl;
    cout<< "\nThe sum is:    " << ra.Sum ( ra.MaxSize ) << endl;
    cout<< "\nthe avr is:    " << ra.Average ( ra.MaxSize ) << endl;
    return 0;
}
```

5-2 已知 Ackerman 函数定义如下：

$$akm(m,n)=\begin{cases} n+1 & \text{当} m=0\text{时} \\ akm(m-1,\ 1) & \text{当} m\neq 0, n=0\text{时} \\ akm(m-1,\ akm(m,n-1)) & \text{当} m\neq 0, n\neq 0\text{时} \end{cases}$$

(1) 根据定义，写出它的递归求解算法；

(2) 利用栈，写出它的非递归求解算法。

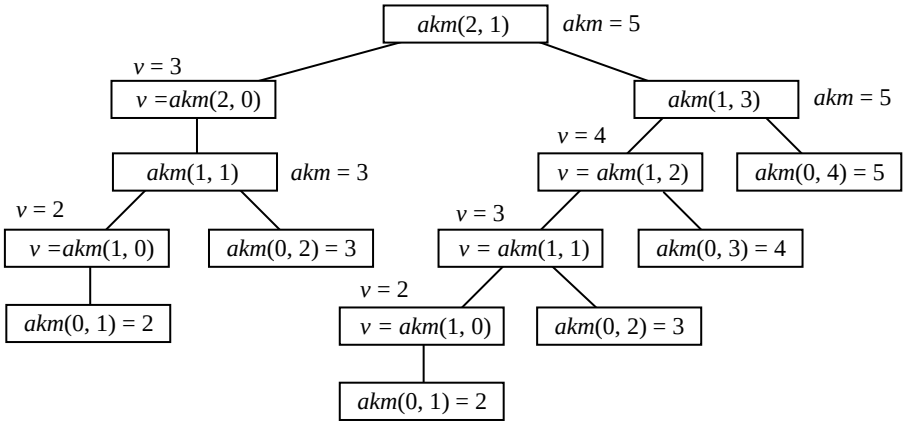
【解答】(1) 已知函数本身是递归定义的，所以可以用递归算法来解决：

```
unsigned akm ( unsigned m, unsigned n ) {
    if ( m == 0 ) return n+1;                          // m == 0
        else if ( n == 0 ) return akm ( m−1, 1 );      // m > 0, n == 0
            else return akm ( m−1, akm ( m, n−1 ) );   // m > 0, n > 0
}
```

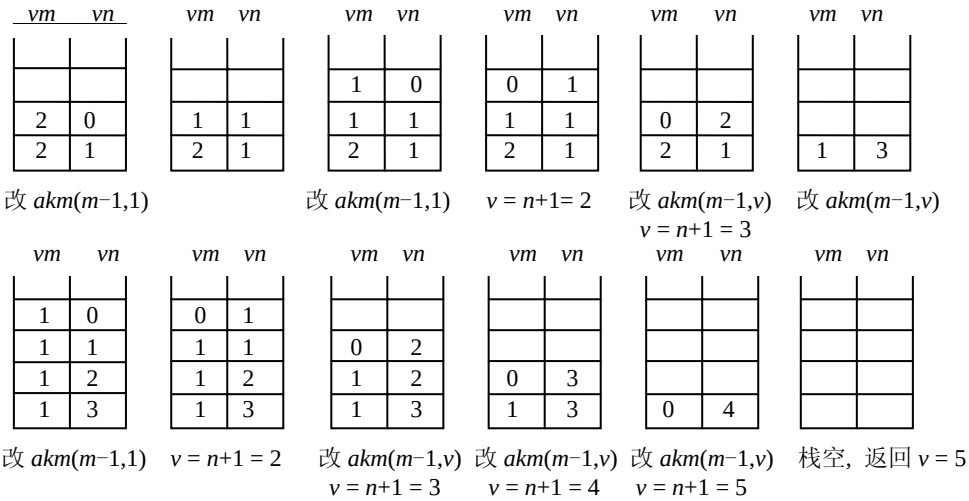
为了将递归算法改成非递归算法，首先改写原来的递归算法，将递归语句从结构中独立出来：

```
unsigned akm ( unsigned m, unsigned n ) {
    unsigned v;
    if ( m == 0 ) return n+1;                          // m == 0
        if ( n == 0 ) return akm ( m−1, 1 );           // m > 0, n ==0
            v = akm ( m, n−1 );                        // m > 0, n > 0
        return akm ( m−1, v );
}
```

计算 akm(2, 1)的递归调用树如图所示：



用到一个栈记忆每次递归调用时的实参值，每个结点两个域**{**vm, vn**}**。对以上实例，栈的变化如下：




相应算法如下

```
#include <iostream.h>
#include "stack.h"
#define maxSize 3500;
unsigned akm ( unsigned m, unsigned n ) {
    struct node { unsigned vm, vn; }
    stack<node> st ( maxSize );   node w;   unsigned v;
    w.vm = m;   w.vn = n;   st.Push (w);
    do {
        while ( st.GetTop( ).vm > 0 ) {              //计算 akm(m−1, akm(m, n−1) )
            while ( st.GetTop( ).vn > 0 )                 //计算 akm(m, n−1), 直到 akm(m, 0)
                { w.vn−−;   st.Push( w ); }
            w = st.GetTop( );   st.Pop( );           //计算 akm(m−1, 1)
            w.vm−−;   w.vn = 1;   st.Push( w );
        }                                            //直到 akm( 0, akm( 1, * ) )
        w = st.GetTop();   st.Pop( );   v = w.vn++;  //计算 v = akm( 1, * )+1
        if ( st.IsEmpty( ) == 0 )                         //如果栈不空, 改栈顶为( m−1, v )
            { w = st.GetTop();   st.Pop( );   w.vm−−;   w.vn = v;   st.Push( w ); }
    } while ( st.IsEmpty( ) == 0 );
    return v;
}
```

5−3 【背包问题】设有一个背包可以放入的物品的重量为 s，现有 n 件物品，重量分别为 w[1], w[2], …, w[n]。问能否从这 n 件物品中选择若干件放入此背包中，使得放入的重量之和正好为 s。如果存在一种符合上述要求的选择，则称此背包问题有解(或称其解为真)；否则称此背包问题无解(或称其解为假)。试用递归方法设计求解背包问题的算法。（提示：此背包问题的递归定义如下：）

$$KNAP(s,n)=\begin{cases} \text{True} & s=0 & \text{此时背包问题一定有解} \\ \text{False} & s<0 & \text{总重量不能为负数} \\ \text{False} & s>0\text{且}n<1 & \text{物品件数不能为负数} \\ KNAP(s,n-1)\text{或} & s>0\text{且}n\geq 1 & \text{所选物品中不包括}w[n]\text{时} \\ KNAP(s-w[n],n-1) & & \text{所选物品中包括}w[n]\text{时} \end{cases}$$

【解答】　根据递归定义，可以写出递归的算法。

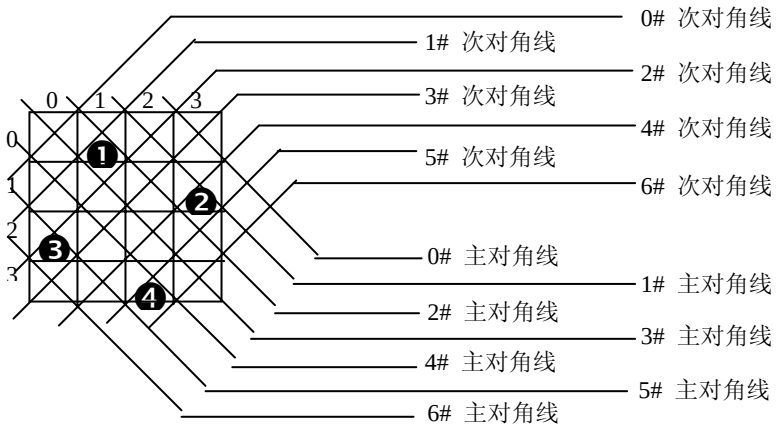
```
enum boolean { False, True }
boolean Knap( int s, int n ) {
    if ( s == 0 ) return True;
    if ( s < 0 || s > 0 && n < 1 ) return False;
    if ( Knap ( s – W[n], n−1 ) == True )
        { cout << W[n] << ' , ';   return True; }
    return Knap( s, n−1 );
}
```

若设 w = { 0, 1, 2, 4, 8, 16, 32 }，s = 51, n = 6。则递归执行过程如下

| | | | | | | |
|---|---|---|---|---|---|---|
| 递归 ↓ | Knap(51, 6) | | | | | **return** True, 完成 |
| | Knap(51−32, 5) | | | | | **return** True, 打印 32 |
| | Knap(19−16, 4) | | | | | **return** True, 打印 16 |
| | Knap(3−8, 3) | **return** False | Knap(3, 3) | | | **return** True, 无动作 |
| | s = −5 < 0 | **return** False | Knap(3−4, 4) | **return** False | Knap(3, 2) | **return** True, 无动作 |
| | | | s = −1 < 0 | **return** False | Knap(3−2, 1) | **return** True, 打印 2 |
| | | | | | Knap(1−1, 0) | **return** True, 打印 1 |
| | | | | | s = 0 | **return** True |

5-4 【八皇后问题】设在初始状态下在国际象棋棋盘上没有任何棋子(皇后)。然后顺序在第 1 行，第 2 行，…。第 8 行上布放棋子。在每一行中有 8 个可选择位置，但在任一时刻，棋盘的合法布局都必须满足 3 个限制条件，即任何两个棋子不得放在棋盘上的同一行、或者同一列、或者同一斜线上。试编写一个递归算法，求解并输出此问题的所有合法布局。(提示：用回溯法。在第 n 行第 j 列安放一个棋子时，需要记录在行方向、列方向、正斜线方向、反斜线方向的安放状态，若当前布局合法，可向下一行递归求解，否则可移走这个棋子，恢复安放该棋子前的状态，试探本行的第 j+1 列。)

【解答】此为典型的回溯法问题。



在解决 8 皇后时，采用回溯法。在安放第 i 行皇后时，需要在列的方向从 1 到 n 试探( j = 1, …, n )：首先在第 j 列安放一个皇后，如果在列、主对角线、次对角线方向有其它皇后，则出现攻击，撤消在第 j 列安放的皇后。如果没有出现攻击，在第 j 列安放的皇后不动，递归安放第 i+1 行皇后。

解题时设置 4 个数组：

col [n] ： col[i] 标识第 i 列是否安放了皇后

md[2n-1] ： md[k] 标识第 k 条主对角线是否安放了皇后

sd[2n-1] ： sd[k] 标识第 k 条次对角线是否安放了皇后

q[n] ： q[i] 记录第 i 行皇后在第几列

利用行号 i 和列号 j 计算主对角线编号 k 的方法是 k = n+i-j-1；计算次对角线编号 k 的方法是 k = i+j。n 皇后问题解法如下：

```
void Queen( int i ) {
    for ( int j = 0;   j < n;   j++ ) {
        if ( col[j] == 0 && md[n+i-j-1] == 0 && sd[i+j] == 0 ) {      //第 i 行第 j 列没有攻击
            col[j] = md[n+i-j-1] = sd[i+j] = 1;   q[i] = j;           //在第 i 行第 j 列安放皇后
            if ( i == n ) {                                           //输出一个布局
        for ( j = 0; j < n; j++ ) cout << q[j] << ‘,’;
        cout << endl;
    }
    else { Queen ( i+1 );                                             //在第 i+1 行安放皇后
            col[j] = md[n+i-j-1] = sd[i+j] = 0;   q[i] = 0;           //撤消第 i 行第 j 列的皇后
                }
        }
}
```

5-5 已知 f 为单链表的表头指针，链表中存储的都是整型数据，试写出实现下列运算的递归算法：

(1) 求链表中的最大整数。

(2) 求链表的结点个数。

(3) 求所有整数的平均值。

【解答】

```
#include <iostream.h>                          //定义在头文件"RecurveList.h"中
class List;
class ListNode {                               //链表结点类
friend class List;
private:
    int data;                                  //结点数据
    ListNode *link;                            //结点指针
    ListNode ( const int item ) : data(item), link(NULL) { }   //构造函数
};
class List {                                   //链表类
private:
    ListNode *first, current;
    int Max ( ListNode *f );
```

```
    int Num ( ListNode *f );
    float Avg ( ListNode *f,   int& n );
public:
    List ( ) : first(NULL), current (NULL) { }      //构造函数
    ~List ( ){ }                                    //析构函数
    ListNode* NewNode ( const int item );           //创建链表结点，其值为 item
    void NewList ( const int retvalue );            //建立链表，以输入 retvalue 结束
    void PrintList ( );                             //输出链表所有结点数据
    int GetMax ( ) { return Max ( first ); }        //求链表所有数据的最大值
    int GetNum ( ) { return Num ( first ); }        //求链表中数据个数
    float GetAvg ( ) { return Avg ( first ); }      //求链表所有数据的平均值
};

ListNode* List :: NewNode ( const int item ) {          //创建新链表结点
    ListNode *newnode = new ListNode (item);
    return newnode;
}

void List :: NewList ( const int retvalue ) {           //建立链表，以输入 retvalue 结束
    first = NULL;   int value;   ListNode *q;
    cout << "Input your data:\n";                       //提示
    cin >> value;                                       //输入
    while ( value != retvalue ) {                       //输入有效
        q = NewNode ( value );                          //建立包含 value 的新结点
        if ( first == NULL ) first = current = q;       //空表时，新结点成为链表第一个结点
        else { current->link = q;   current = q; }      //非空表时，新结点链入链尾
        cin >> value;                                   //再输入
    }
    current->link = NULL;                               //链尾封闭
}

void List :: PrintList ( ) {                            //输出链表
    cout << "\nThe List is : \n";
    ListNode *p = first;
    while ( p != NULL ) {       cout << p->data << '   ';   p = p->link;       }
    cout << '\n';
}

int List :: Max ( ListNode *f ) {                       //递归算法 ：求链表中的最大值
    if ( f ->link == NULL ) return f ->data;            //递归结束条件
    int temp = Max ( f ->link );                        //在当前结点的后继链表中求最大值
    if ( f ->data > temp ) return f ->data;             //如果当前结点的值还要大，返回当前检点值
    else return temp;                                   //否则返回后继链表中的最大值
}
```

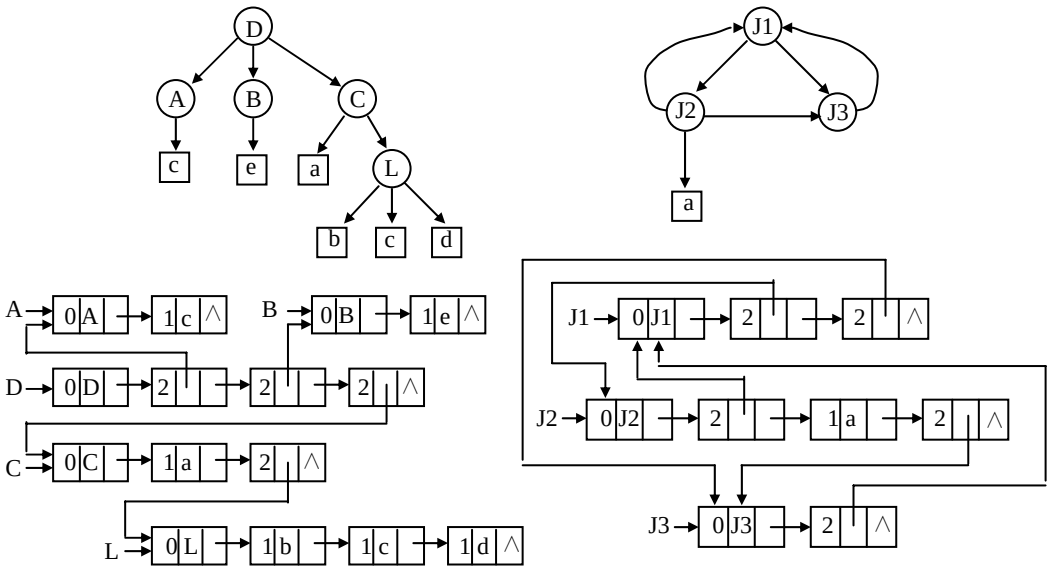
```
int List :: Num ( ListNode *f ) {                    //递归算法 ：求链表中结点个数
    if ( f == NULL ) return 0;                       //空表，返回 0
    return 1+ Num ( f ->link );                      //否则，返回后继链表结点个数加 1
}

float List :: Avg ( ListNode *f , int& n ) {         //递归算法 ：求链表中所有元素的平均值
    if ( f ->link == NULL )                          //链表中只有一个结点，递归结束条件
        { n = 1;   return ( float ) (f ->data ); }
    else { float Sum = Avg ( f ->link, n ) * n;   n++;   return ( f  ->data + Sum ) / n; }
}

#include "RecurveList.h"                             //定义在主文件中
int main ( int argc, char* argv[ ] ) {
    List test;     int finished;
    cout << "输入建表结束标志数据 ： ";
    cin >> finished;                                 //输入建表结束标志数据
    test.NewList ( finished );                       //建立链表
    test.PrintList ( );                              //打印链表
    cout << "\nThe Max is : " << test.GetMax ( );
    cout << "\nThe Num is : " << test.GetNum ( );
    cout << "\nThe Ave is : " << test.GetAve () << '\n';
    printf ( "Hello World!\n" );
    return 0;
}
```

5-6 画出下列广义表的图形表示和它们的存储表示：

(1) D(A(c), B(e), C(a, L(b, c, d)))

(2) J1(J2(J1, a, J3(J1)), J3(J1))

【解答】(1) D(A(c), B(e), C(a, L(b, c, d)))　　(2) J1(J2(J1, a, J3(J1)), J3(J1))



5-7 利用广义表的 head 和 tail 操作写出函数表达式，把以下各题中的单元素 banana 从广义表中分离出来：

(1) L1(apple, pear, banana, orange)

(2) L2((apple, pear), (banana, orange))

(3) L3(((apple), (pear), (banana), (orange)))

(4) L4((((apple))), ((pear)), (banana), orange)

(5) L5((((apple), pear), banana), orange)

(6) L6(apple, (pear, (banana), orange))

【解答】

(1) Head (Tail (Tail (L1) ) )

(2) Head (Head (Tail (L2) ) )

(3) Head (Head (Tail (Tail (Head (L3) ) ) ) ) )

(4) Head (Head (Tail (Tail (L4) ) ) )

(5) Head (Tail (Head(L5) ) )

(6) Head (Head (Tail (Head (Tail (L6) ) ) ) ) )

5-8 广义表具有可共享性，因此在遍历一个广义表时必需为每一个结点增加一个标志域 mark，以记录该结点是否访问过。一旦某一个共享的子表结点被作了访问标志，以后就不再访问它。

(1) 试定义该广义表的类结构；

(2) 采用递归的算法对一个非递归的广义表进行遍历。

(3) 试使用一个栈，实现一个非递归算法，对一个非递归广义表进行遍历。

【解答】(1) 定义广义表的类结构

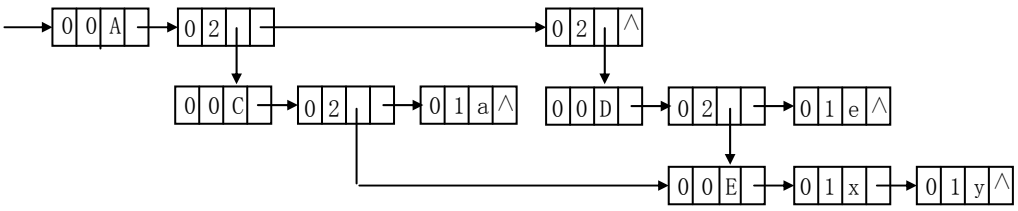
为了简化广义表的操作，在广义表中只包含字符型原子结点，并用除大写字母外的字符表示数据，表头结点中存放用大写字母表示的表名。这样，广义表中结点类型三种：表头结点、原子结点和子表结点。

```
class GenList;                                  //GenList 类的前视声明
class GenListNode {                             //广义表结点类定义
friend class Genlist;
private:
    int mark, utype;                            // utype  =0 / 1 / 2, mark 是访问标记, 未访问为 0
    GenListNode* tlink;                         //指向同一层下一结点的指针
    union {                                     //联合
        char listname;                          // utype = 0, 表头结点情形：存放表名
        char atom;                              // utype = 1, 存放原子结点的数据
        GenListNode* hlink;                     // utype = 2, 存放指向子表的指针
    } value;
public:
    GenListNode ( int tp, char info ) : mark (0), utype (tp), tlink (NULL)     //表头或原子结点构造函数
        { if ( utype == 0 ) value.listname = info; else value.atom = info; }
    GenListNode (GenListNode* hp )                                  //子表构造函数
        : mark (0), utype (2), value.hlink (hp) { }
    char Info ( GenListNode* elem )                                 //返回表元素 elem 的值
        { return ( utype == 0 ) ? elem->value.listname : elem->value.atom; }
```

```
};
class GenList {                                    //广义表类定义
private:
    GenListNode *first;                            //广义表头指针
    void traverse ( GenListNode * ls );            //广义表遍历
    void Remove ( GenListNode* ls );               //将以 ls 为表头结点的广义表结构释放
public:
    Genlist ( char& value );                       //构造函数, value 是指定的停止建表标志数据
    ~GenList ( );                                  //析构函数
    void traverse ( );                             //遍历广义表
}
```

(2) 广义表遍历的递归算法

```
void GenList :: traverse ( ) {                     //共有函数
    traverse ( first );
}

#include <iostream.h>
void GenList :: traverse ( GenListNode * ls ) {    //私有函数, 广义表的遍历算法
    if ( ls != NULL ) {
        ls->mark = 1;
        if ( ls->utype == 0 ) cout << ls->value.listname << '(';          //表头结点
        else if ( ls->utype == 1 ) {                                      //原子结点
            cout << ls->value.atom;
            if ( ls->tlink != NULL ) cout << ',';
        }
        else if ( ls->utype == 2 ) {                                      //子表结点
            if ( ls->value.hlink->mark == 0 ) traverse( ls->value.hlink );  //向表头搜索
            else cout << ls->value.hlink->value.listname;
            if ( ls->tlink != NULL ) cout << ',';
        }
        traverse ( ls->tlink );                                           //向表尾搜索
    }
    else cout << ')';
}
```




对上图所示的广义表进行遍历，得到的遍历结果为 A(C(E(x, y), a), D(E, e) )。

(2) 利用栈可实现上述算法的非递归解法。栈中存放回退时下一将访问的结点地址。

```
#include <iostream.h>
```

```
#include "stack.h"
void GenList :: traverse ( GenListNode *ls ) {
    Stack <GenListNode<Type> *> st;
    while ( ls != NULL ) {
        ls->mark = 1;
        if ( ls->utype == 2 ) {                                  //子表结点
            if ( ls->value.hlink->mark == 0 )                    //该子表未访问过
              { st.Push ( ls->tlink );        ls = ls->value.hlink; }          //暂存下一结点地址，访
      问子表
            else {
                cout << ls->value.hlink->value.listname;         //该子表已访问过，仅输出表名
                if ( ls->tlink != NULL ) { cout << ',' ; ls = ls->tlink; }
            }
        }
        else {
            if ( ls->utype == 0 ) cout << ls->value.listname << '(';    //表头结点
            else if ( ls->utype == 1 ) {                         //原子结点
                    cout << ls->value.atom;
                    if ( ls->tlink != NULL ) cout << ',';
                }
            if ( ls->tlink == NULL ) {                           //子表访问完，子表结束处理
                cout >> ')';
                if ( st.IsEmpty( ) == 0 ) {                      //栈不空
                    ls = st.GetTop ( );   st.Pop ( );            //退栈
                    if ( ls != NULL ) cout << ',';
                    else cout << ')';
                }
            }
            else ls = ls->tlink;                                 //向表尾搜索
        }
    }
}
```

(4) 广义表建立操作的实现

```
#include <iostream.h>
#include <ctype.h>
#include "stack.h"
const int maxSubListNum = 20;                                    //最大子表个数
GenList :: GenList ( char& value ) {
    Stack <GenListNode* > st (maxSubListNum);                    //用于建表时记忆回退地址
    SeqList <char> Name (maxSubListNum);                         //记忆建立过的表名
    SeqList <GenListNode * > Pointr (maxSubListNum);             //记忆对应表头指针
    GenListNode * p, q, r;    Type ch;     int m = 0, ad, br;    //m 为已建表计数, br 用于对消括号
    cout << "广义表停止输入标志数据 value : ";     cin >> value;
    cout << "开始输入广义表数据，如 A(C(E( x, y ), a ), D(E(x, y), e) )"
```

```
    cin >> ch;    first = q = new GenListNode ( 0, ch );                    //建立整个表的表头结点
    if ( ch != value ) { Name.Insert (ch, m);    Pointr.Insert (q, m);    m++; }  //记录刚建立的表头结点
    else return 1;                                                          //否则建立空表，返回 1
    cin >> ch;    if ( ch == '(' ) st.Push ( q );                           //接着应是'('，进栈
    cin >> ch;
    while ( ch != value ) {                                                 //逐个结点加入
        switch ( ch ) {
          case '(' :   p = new GenListNode <Type> ( q );                    //建立子表结点, p->hlink = q
                       r = st.GetTop( );    st.Pop( );    r->tlink = p;     //子表结点插在前一结点 r 之后
                       st.Push( p );    st.Push( q );                       //子表结点及下一表头结点进栈
                       break;
          case ')' :   q->tlink = NULL;    st.pop( );                       //子表建成，封闭链，退到上层
                       if ( st.IsEmpty ( ) == 0 ) q = st.GetTop( );         //栈不空，取上层链子表结点
                       else return 0;                                       //栈空，无上层链，算法结束
                       break;
          case ',' :   break;
          default :    ad = Name.Find (ch);                                 //查找是否已建立，返回找到位置
                       if ( ad == -1 ) {                                    //查不到，建新结点
                           p = q;
                           if ( isupper (ch) ) {                            //大写字母，建表头结点
                               q = new GenListNode ( 0, ch );
                               Name.Insert (ch, m);    Pointr.Insert (q, m);    m++;
                           }
                           else q = new GenListNode ( 1, ch );              //非大写字母，建原子结点
                           p->tlink = q;                                    //链接
                       }
                       else {                                               //查到，已加入此表
                           q = Pointr.Get (ad);    p = new GenListNode (q);    //建立子表结点, p->hlink = q
                           r = st.GetTop ( );    st.Pop ( );    r->tlink = p;    st.Push (p);    q = p;
                           br = 0;                                          //准备对消括号
                           cin >> ch;    if ( ch == '(' ) br++;             //若有左括号, br 加 1
                           while ( br == 0 ) {                              //br 为 0 表示括号对消完，出循环
                               cin >> ch;
                               if ( ch == '(' ) br++;    else if ( ch == ')' ) br--;
                           }
                       }
          }
          cin >> ch;
    }
}
```

# 第六章 树与森林

6-1 写出用广义表表示法表示的树的类声明，并给出如下成员函数的实现：

(1) **operator** >> ( ) 接收用广义表表示的树作为输入，建立广义表的存储表示；

(2) 复制构造函数 用另一棵表示为广义表的树初始化一棵树；

(3) **operator** == ( ) 测试用广义表表示的两棵树是否相等；

(4) **operator** << ( ) 用广义表的形式输出一棵树；

(5) 析构函数 清除一棵用广义表表示的树。

【解答】

```
#include <iostream.h>
#define maxSubTreeNum 20;                          //最大子树(子表)个数
class GenTree;                                     //GenTree 类的前视声明

class GenTreeNode {                                //广义树结点类的声明
friend class GenTree;
private:
    int utype;                                     //结点标志：=0, 数据; =1, 子女
    GenTreeNode * nextSibling;                     //utype=0, 指向第一个子女;
                                                    //utype=1 或 2, 指向同一层下一兄弟
    union {                                        //联合
        char RootData;                             //utype=0, 根结点数据
        char Childdata;                            //utype=1, 子女结点数据
        GenTreeNode *firstChild;                   //utype=2, 指向第一个子女的指针
    }
public:
    GenTreeNode ( int tp, char item ) : utype (tp), nextSibling (NULL)
      { if ( tp == 0 ) RootData = item;    else ChildData = item; }
      //构造函数：构造广义表表示的树的数据结点
    GenTreeNode ( GenTreeNode *son = NULL ) : utype (2), nextSibling (NULL), firstChild ( son ) { }
      //构造函数：构造广义表表示的树的子树结点
    int nodetype ( ) { return utype; }                          //返回结点的数据类型
    char GetData ( ) { return data; }                           //返回数据结点的值
    GenTreeNode * GetFchild ( ) { return firstChild; }          //返回子树结点的地址
    GenTreeNode * GetNsibling ( ) { return nextSibling; }       //返回下一个兄弟结点的地址
    void setInfo ( char item ) { data = item; }                 //将结点中的值修改为 item
    void setFchild ( GenTreeNode * ptr ) { firstChild = ptr; }  //将结点中的子树指针修改为 ptr
    void setNsinbilg ( GenTreeNode * ptr ) { nextSibling = ptr; }
};

class GenTree {                                    //广义树类定义
private:
    GenTreeNode *first;                            //根指针
    char retValue;                                 //建树时的停止输入标志
    GenTreeNode *Copy ( GenTreeNode * ptr );       //复制一个 ptr 指示的子树
```

```
    void Traverse ( GenListNode * ptr );                    //对 ptr 为根的子树遍历并输出
    void Remove ( GenTreeNode *ptr );                       //将以 ptr 为根的广义树结构释放
    friend int Equal ( GenTreeNode *s, GenTreeNode *t );    //比较以 s 和 t 为根的树是否相等
public:
    GenTree ( );                                            //构造函数
    GenTree ( GenTree& t );                                 //复制构造函数
    ~GenTree ( );                                           //析构函数
    friend int operator == ( GenTree& t1, GenTree& t2 );    //判两棵树 t1 与 t2 是否相等
    friend istream& operator >> ( istream& in, GenTree& t );  //输入
    friend ostream& operator << ( ostream& out, GenTree& t );  //输出
}
```

(1) **operator** >> ( ) 接收用广义表表示的树作为输入，建立广义表的存储表示

```
istream& operator >> ( istream& in, GenTree& t ) {
//友元函数，从输入流对象 in 接受用广义表表示的树，建立广义表的存储表示 t。
    t.ConstructTree ( in, retValue );
    return in;
}

void GenTree :: ConstructTree ( istream& in, char& value ) {
//从输入流对象 in 接受用广义表表示的非空树，建立广义表的存储表示 t。
    Stack <GenTreeNode* > st (maxSubTreeNum);                  //用于建表时记忆回退地址
    GenTreeNode * p, q, r;   Type ch;
    cin >> value;                                              //广义树停止输入标志数据
    cin >> ch;    first = q = new GenTreeNode ( 0, ch );       //建立整个树的根结点
    cin >> ch;    if ( ch == '(' ) st.Push ( q );              //接着应是'(', 进栈
    cin >> ch;
    while ( ch != value ) {                                    //逐个结点加入
        switch ( ch ) {
          case '(' :   p = new GenTreeNode <Type> ( q );       //建立子树, p->firstChild = q
                       r = st.GetTop( );    st.Pop( );         //从栈中退出前一结点
                       r->nextSibling = p;                     //插在前一结点 r 之后
                       st.Push( p );    st.Push( q );          //子树结点及子树根结点进栈
                       break;
          case ')' :   q->nextSibling = NULL;  st.pop( );      //子树建成, 封闭链, 退到上层
                       if ( st.IsEmpty ( ) == 0 ) q = st.GetTop( );  //栈不空, 取上层链子树结点
                       else return 0;                          //栈空, 无上层链, 算法结束
                       break;
          case ',' :   break;
          default :    p = q;
                       if ( isupper (ch) ) q = new GenTreeNode ( 0, ch );  //大写字母, 建根结点
                       else q = new GenTreeNode ( 1, ch );     //非大写字母, 建数据结点
                       p->nextSibling = q;                     //链接
        }
```

```
        cin >> ch;
    }
}
```

(2) 复制构造函数 用另一棵表示为广义表的树初始化一棵树；

```
GenTree :: GenTree ( const GenTree& t ) {                    //共有函数
    first = Copy ( t.first );
}

GenTreeNode* GenTree :: Copy ( GenTreeNode *ptr ) {
//私有函数，复制一个 ptr 指示的用广义表表示的子树
    GenTreeNode *q = NULL;
    if ( ptr != NULL ) {
        q = new GenTreeNode ( ptr->utype, NULL );
        switch ( ptr->utype ) {                              //根据结点类型 utype 传送值域
          case 0 : q->RootData = ptr->RootData;   break;     //传送根结点数据
          case 1 : q->ChildData = ptr->ChildData;   break;   //传送子女结点数据
          case 2 : q->firstChild = Copy ( ptr->firstChild );   break;   //递归传送子树信息
        }
        q->nextSibling = Copy ( ptr->nextSibling );          //复制同一层下一结点为头的表
    }
    return q;
}
```

(3) **operator** == ( )　测试用广义表表示的两棵树是否相等

```
int operator == ( GenTree& t1, GenTree& t2 ) {
    //友元函数 ： 判两棵树 t1 与 t2 是否相等，若两表完全相等，函数返回 1，否则返回 0。
    return Equal ( t1.first, t2.first );
}

int Equal ( GenTreeNode *t1, GenTreeNode *t2 ) {
//是 GenTreeNode 的友元函数
    int x;
    if ( t1 == NULL && t2 == NULL ) return 1;                //表 t1 与表 t2 都是空树，相等
    if ( t1 != NULL && t2 != NULL && t1->utype == t2->utype ) {   //两子树都非空且结点类型相同
        switch ( t1->utype ) {                               //比较对应数据
          case 0 :  x = ( t1->RootData == t2->RootData ) ? 1 : 0;   //根数据结点
                    break;
          case 1 :  x = ( t1->ChildData == t2->ChildData ) ? 1 : 0;   //子女数据结点
                    break;
          case 2 : x = Equal ( t1->firstChild,   t2->firstChild );   //递归比较其子树
        }
        if ( x ) return Equal ( t1->nextSibling,   t2->nextSibling );
                    //相等，递归比较同一层的下一个结点；不等，不再递归比较
```

```
    }
    return 0;
}
```

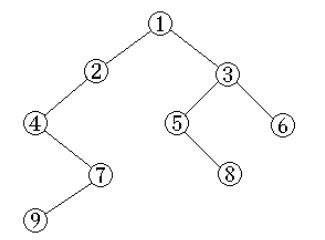
(4) **operator** << ( ) 用广义表的形式输出一棵树

```
ostream& operator << ( ostream& out, GenTree& t ) {
//友元函数，将树 t 输出到输出流对象 out。
    t.traverse ( out, t.first );
    return out;
}

void GenTree :: traverse ( ostream& out, GenTreeNode * ptr ) {
//私有函数，广义树的遍历算法
    if ( ptr != NULL ) {
        if ( ptr->utype == 0 ) out << ptr->RootData << '(';          //根数据结点
        else if ( ptr->utype == 1 ) {                                //子女数据结点
                out << ptr->ChildData;
                if ( ptr->nextSibling != NULL ) out << ',';
            }
        else {                                                       //子树结点
                traverse ( ptr->firstChild );                        //向子树方向搜索
                if ( ptr->nextSibling != NULL ) out << ',';
        }
        traverse ( ptr->nextSibling );                               //向同一层下一兄弟搜索
    }
    else out << ')';
}
```

(5) 析构函数 清除一棵用广义表表示的树

```
GenTree :: ~ GenTree ( ) {
//用广义表表示的树的析构函数，假定 first ≠ NULL
    Remove ( first );
}

void GenTree :: Remove ( GenTreeNode *ptr ) {
    GenTreeNode * p;
    while ( ptr != NULL ) {
        p = ptr->nextSibling;
        if ( p->utype == 2 ) Remove ( p->firstChild );               //在子树中删除
        ptr->nextSibling = p->nextSibling;    delete ( p );          //释放结点 p
    }
}
```

6-2 列出右图所示二叉树的叶结点、分支结点和每个结点的层次。

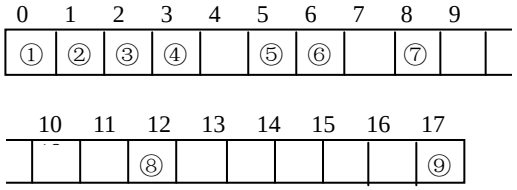



【解答】

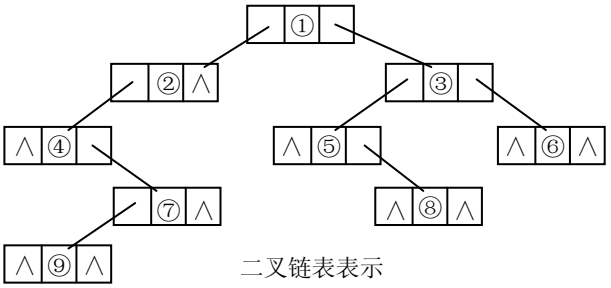
二叉树的叶结点有⑥、⑧、⑨。分支结点有①、②、③、④、⑤、⑦。

结点①的层次为 0；结点②、③的层次为 1；结点④、⑤、⑥的层次为 2；结点⑦、⑧的层次为 3；结点⑨的层次为 4。

6-3 使用 (1) 顺序表示和 (2) 二叉链表表示法，分别画出右图所示二叉树的存储表示。




顺序表示




二叉链表表示

6-4 用嵌套类写出用链表表示的二叉树的类声明。

【解答】

```
#include <iostream.h>
template <class Type> class BinaryTree {
private:
    template <class Type> class BinTreeNode {
    public:
        BinTreeNode<Type> *leftChild, *rightChild;
        Type data;
    }
    Type RefValue;
    BinTreeNode<Type> * root;
    BinTreeNode<Type> *Parent ( BinTreeNode<Type> *start, BinTreeNode<Type> *current );
    int Insert ( BinTreeNode<Type> *current, const Type& item );
    int Find ( BinTreeNode<Type> *current, const Type& item ) const;
    void destroy ( BinTreeNode<Type> *current );
    void Traverse ( BinTreeNode<Type> *current, ostream& out ) const;
public:
    BinaryTree ( ) : root ( NULL ) { }
    BinaryTree ( Type value ) : RefValue ( value ), root ( NULL ){ }
    ~BinaryTree ( ) { destroy (root); }
    BinTreeNode ( ) : leftChild ( NULL ), rightChild ( NULL ) { }
    BinTreeNode ( Type item ) : data ( item ), leftChild ( NULL ), rightChild ( NULL ) { }
    Type& GetData ( ) const { return data; }
    BinTreeNode<Type>* GetLeft ( ) const { return leftChild; }
    BinTreeNode<Type>* GetRight ( ) const { return rightChild; }
    void SetData ( const Type& item ){ data = item; }
    void SetLeft ( BinTreeNode<Type> *L ) { leftChild = L; }
    void SetRight ( BinTreeNode<Type> *R ){ RightChild =R; }
```

```
    int IsEmpty ( ) { return root == NULL ? 1 : 0; }
    BinTreeNode<Type> *Parent ( BinTreeNode<Type> *current )
      { return root == NULL || root == current ? NULL : Parent ( root, current ); }
    BinTreeNode<Type> * LeftChild ( BinTreeNode<Type> *current )
      { return current != NULL ? current->leftChild : NULL; }
    BinTreeNode<Type> * RighttChild ( BinTreeNode<Type> *current )
      { return current != NULL ? current->rightChild : NULL; }
    int Insert ( const Type& item );
    BinTreeNode<Type> * Find ( const Type& item );
    BinTreeNode<Type> * GetRoot ( ) const { return root; }
    friend istream& operator >> ( istream& in, BinaryTree<Type>& Tree );        //输入二叉树
    friend ostream& operator << ( ostream& out, BinaryTree<Type>& Tree );       //输出二叉树
}
```

6-5 在结点个数为 n (n>1)的各棵树中，高度最小的树的高度是多少？它有多少个叶结点？多少个分支结点？高度最大的树的高度是多少？它有多少个叶结点？多少个分支结点？

【解答】结点个数为 n 时，高度最小的树的高度为 1，有 2 层；它有 n−1 个叶结点，1 个分支结点；高度最大的树的高度为 n−1，有 n 层；它有 1 个叶结点，n−1 个分支结点。

6-6 试分别画出具有 3 个结点的树和 3 个结点的二叉树的所有不同形态。

6-7 如果一棵树有 $n_1$ 个度为 1 的结点，有 $n_2$ 个度为 2 的结点，…，$n_m$ 个度为 m 的结点，试问有多少个度为 0 的结点？试推导之。

【解答】　总结点数 $n = n_0 + n_1 + n_2 + \cdots + n_m$

总分支数 $e = n-1 = n_0 + n_1 + n_2 + \cdots + n_m - 1$

$= m*n_m + (m-1)*n_{m-1} + \cdots + 2*n_2 + n_1$

则有 $n_0 = \left( \sum_{i=2}^{m} (i-1) n_i \right) + 1$

6-8 试分别找出满足以下条件的所有二叉树:

(1) 二叉树的前序序列与中序序列相同;

(2) 二叉树的中序序列与后序序列相同;

(3) 二叉树的前序序列与后序序列相同。

【解答】

(1) 二叉树的前序序列与中序序列相同：空树或缺左子树的单支树；

(2) 二叉树的中序序列与后序序列相同：空树或缺右子树的单支树；

(3) 二叉树的前序序列与后序序列相同：空树或只有根结点的二叉树。

6-9 若用二叉链表作为二叉树的存储表示，试针对以下问题编写递归算法：

(1) 统计二叉树中叶结点的个数。

(2) 以二叉树为参数，交换每个结点的左子女和右子女。

【解答】

(1) 统计二叉树中叶结点个数

```
int BinaryTree<Type> :: leaf ( BinTreeNode<Type> * ptr ) {
```

```
    if ( ptr == NULL ) return 0;
    else if ( ptr->leftChild == NULL && ptr->rightChild == NULL ) return 1;
    else return leaf ( ptr->leftChild ) + leaf ( ptr->rightChild );
}
```

(2) 交换每个结点的左子女和右子女

```
void BinaryTree<Type> :: exchange ( BinTreeNode<Type> * ptr ) {
    BinTreeNode<Type> * temp;
    if ( ptr->leftChild != NULL || ptr->rightChild != NULL ) {
        temp = ptr->leftChild;
        ptr->leftChild = ptr->rightChild;
        ptr->rightChild = temp;
        exchange ( ptr->leftChild );
        exchange ( ptr->rightChild );
    }
}
```

6-10 一棵高度为 h 的满 k 叉树有如下性质：第 h 层上的结点都是叶结点，其余各层上每个结点都有 k 棵非空子树，如果按层次自顶向下，同一层自左向右，顺序从 1 开始对全部结点进行编号，试问：

(1) 各层的结点个数是多少?

(2) 编号为 i 的结点的父结点(若存在)的编号是多少?

(3) 编号为 i 的结点的第 m 个孩子结点(若存在)的编号是多少?

(4) 编号为 i 的结点有右兄弟的条件是什么？其右兄弟结点的编号是多少?

(5) 若结点个数为 n，则高度 h 是 n 的什么函数关系?

【解答】

(1) $k^i$ ( i = 0, 1, ……, h )

(2) $\left\lfloor \frac{i+k-2}{k} \right\rfloor$

(3) ( i-1)*k + m + 1

(4) ( i-1 ) % k ≠ 0 或 $i \le \left\lfloor \frac{i+k-2}{k} \right\rfloor * k$ 时有右兄弟，右兄弟为 i + 1。

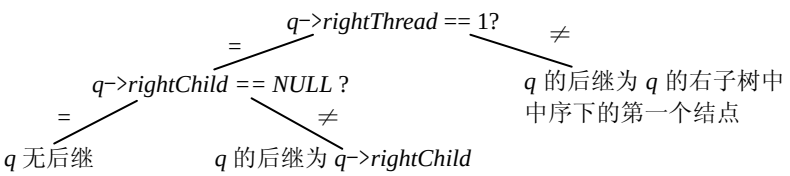
(5) $h = \log_k (n*(k-1)+1)-1$ (n = 0 时 h = -1 )

6-11 试用判定树的方法给出在中序线索化二叉树上：

【解答】

(1) 搜索指定结点的在中序下的后继。

设指针 q 指示中序线索化二叉树中的指定结点，指针 p 指示其后继结点。

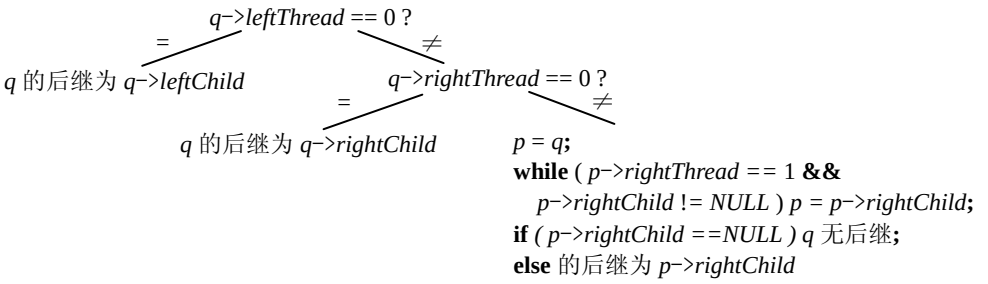



找 q 的右子树中在中序下的第一个结点的程序为：

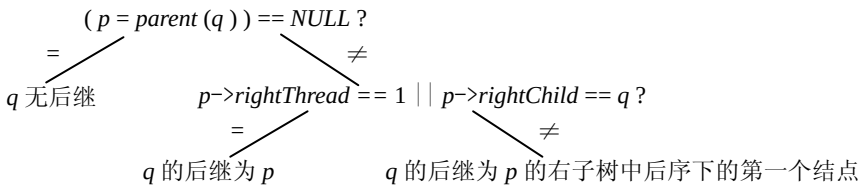
```
p = q->rightChild;
```

```
while ( p->leftThread == 1 ) p = p->leftChild;   // p 即为 q 的后继
```

(2) 搜索指定结点的在前序下的后继。




(3) 搜索指定结点的在后序下的后继。




可用一段遍历程序来实现寻找 p 的右子树中后序下的第一个结点：即该子树第一个找到的叶结点。找到后立即返回它的地址。

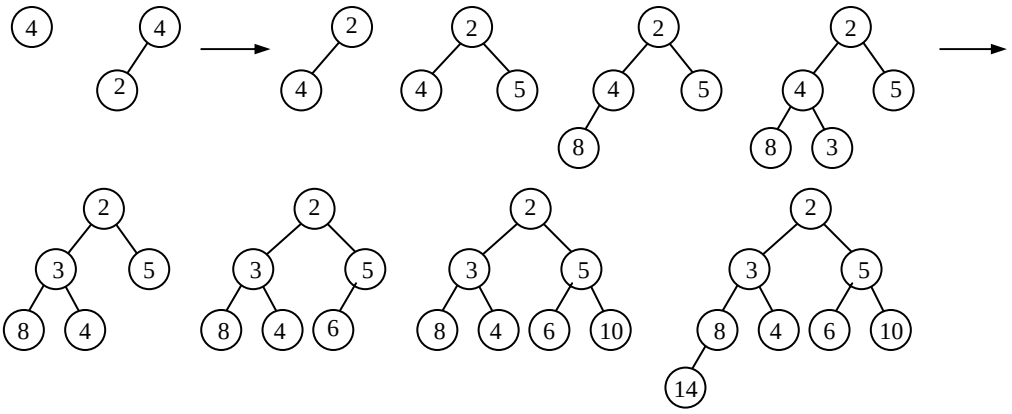
6-12 已知一棵完全二叉树存放于一个一维数组 T[n]中，T[n]中存放的是各结点的值。试设计一个算法，从 T[0]开始顺序读出各结点的值，建立该二叉树的二叉链表表示。

【解答】

```
template <class Type>                                          //建立二叉树
istream& operator >> ( istream& in, BinaryTree<Type>& t ) {
    int n;
    cout << "Please enter the number of node : ";   in >> n;
    Type *A = new Type[n+1];
    for ( int i = 0; i < n; i++ ) in >> A[i];
    t. ConstructTree( T, n, 0, ptr );                          //以数组建立一棵二叉树
    delete [ ] A;
    return in;
}

template <class Type>
void BinaryTree<Type> :: ConstructTree ( Type T[ ], int n, int i, BinTreeNode<Type> *& ptr ) {
//私有函数 ： 将用 T[n]顺序存储的完全二叉树，以 i 为根的子树转换成为用二叉链表表示的
//以 ptr 为根的完全二叉树。利用引用型参数 ptr 将形参的值带回实参。
    if ( i >= n ) ptr = NULL;
    else {
        ptr = new BinTreeNode<Type> ( T[i] );                  //建立根结点
        ConstructTree ( T, n, 2*i+1, ptr->leftChild );
        ConstructTree ( T, n, 2*i+2, ptr->rightChild );
    }
}
```

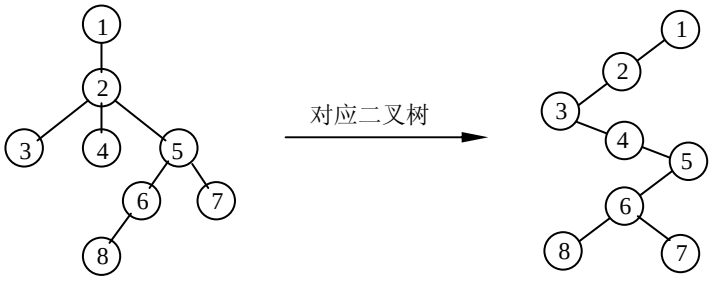
6-14　写出向堆中加入数据 4, 2, 5, 8, 3, 6, 10, 14 时，每加入一个数据后堆的变化。

【解答】以最小堆为例：



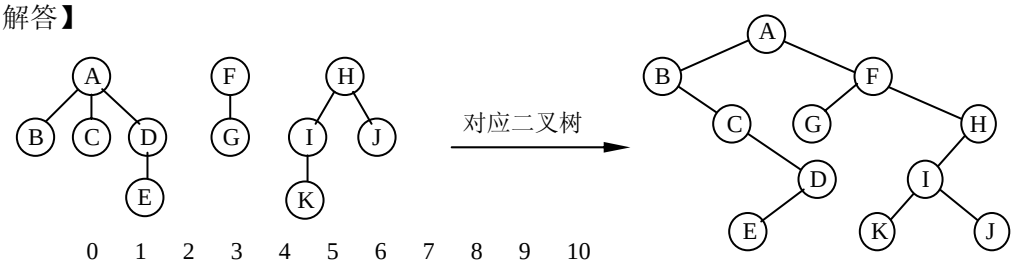
6-16　请画出右图所示的树所对应的二叉树。

【解答】




6-17　在森林的二叉树表示中，用 llink 存储指向结点第一个子女的指针，用 rlink 存储指向结点下一个兄弟的指针，用 data 存储结点的值。如果我们采用静态二叉链表作为森林的存储表示，同时按森林的先根次序依次安放森林的所有结点，则可以在它们的结点中用只有一个二进位的标志 ltag 代替 llink，用 rtag 代替 rlink。并设定若 ltag = 0，则该结点没有子女，若 ltag ≠ 0，则该结点有子女；若 rtag = 0，则该结点没有下一个兄弟，若 rtag ≠ 0，则该结点有下一个兄弟。试给出这种表示的结构定义，并设计一个算法，将用这种表示存储的森林转换成用 llink－rlink 表示的森林。

【解答】




| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| *llink* | 1 | -1 | -1 | 4 | -1 | 6 | -1 | 8 | 9 | -1 | -1 |
| *data* | A | B | C | D | E | F | G | H | I | K | J |
| *rlink* | 5 | 2 | 3 | -1 | -1 | 7 | -1 | -1 | 10 | -1 | -1 |

森林的左子女-右兄弟表示的静态二叉链表

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| *ltag* | 1 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 1 | 0 | 0 |
| *data* | A | B | C | D | E | F | G | H | I | K | J |
| *rtag* | 1 | 1 | 1 | 0 | 0 | 1 | 0 | 0 | 1 | 0 | 0 |

森林的双标记表示

(1) 结构定义

```
template <class Type> class LchRsibNode {          //左子女-右兄弟链表结点类的定义
protected:
   Type data;                                      //结点数据
   int llink, rlink;                               //结点的左子女、右兄弟指针
public:
   LchRsibNode ( ) : llink(NULL), rlink(NULL) { }
   LchRsibNode ( Type x ) : llink(NULL), rlink(NULL), data(x) { }
}

template <class Type> class DoublyTagNode {        //双标记表结点类的定义
protected:
   Type data;                                      //结点数据
   int ltag, rtag;                                 //结点的左子女、右兄弟标记
public:
   DoublyTagNode ( ) : ltag(0), rtag(0) { }
   DoublyTagNode ( Type x ) : ltag(0), rtag(0), data(x) { }
}

template <class Type> class staticlinkList         //静态链表类定义
      : public LchRsibNode<Type>, public DoublyTagNode <Type>{
private:
   LchRsibNode<Type> *V;                           //存储左子女-右兄弟链表的向量
   DoublyTagNode <Type> *U;                        //存储双标记表的向量
   int MaxSize, CurrentSize;                       //向量中最大元素个数和当前元素个数
public:
   dstaticlinkList ( int Maxsz ) : MaxSize ( Maxsz ), CurrentSize (0) {
      V = new LchRsibNode <Type> [Maxsz];
      U = new DoublyTagNode <Type> [Maxsz];
   }
}
```

(2) 森林的双标记先根次序表示向左子女-右兄弟链表先根次序表示的转换

```
void staticlinkList<Type> :: DtagF-LchRsibF ( ) {
    Stack<int> st;    int k;
    for ( int i = 0;    i < CurrentSize;    i++ ) {
            switch ( U[i].ltag ) {
               case 0 : switch ( U[i].rtag ) {
                        case 0 :    V[i].llink = V[i].rlink = -1;
                                    if ( st.IsEmpty ( ) == 0 )
                                  { k = st.GetTop ( );    st.Pop ( );    V[k].rlink = i + 1; }
                                 break;
                        case 1 :    V[i].llink = -1;    V[i].rlink = i + 1;    break;
                      }
                      break;
```

```
                case 1 : switch ( U[i].rtag ) {
                            case 0 :   V[i].llink = i + 1;   V[i].rlink = −1;   break;
                            case 1 :   V[i].llink = i + 1;   st.Push ( i );
                         }
            }
        }
    }
```

6−18 二叉树的双序遍历(Double−order traversal)是指：对于二叉树的每一个结点来说，先访问这个结点，再按双序遍历它的左子树，然后再一次访问这个结点，接下来按双序遍历它的右子树。试写出执行这种双序遍历的算法。

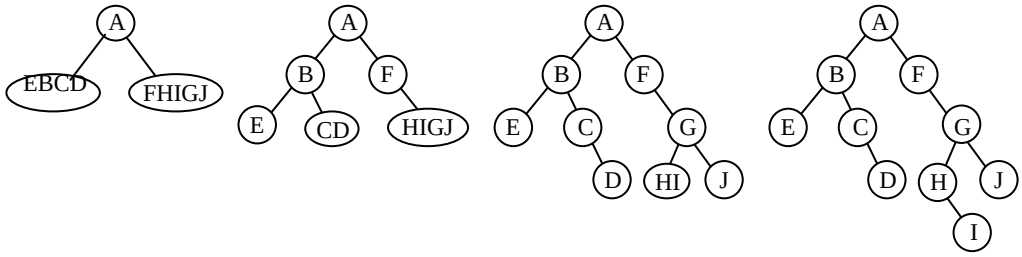
【解答】

```
template <class Type>
void BinaryTree<Type> :: Double_order ( BinTreeNode<Type> *current ){
    if ( current != NULL ) {
        cout << current−>data << ' ';
        Double_order ( current−>leftChild );
        cout << current−>data << ' ';
        Double_order ( current−>rightChild );
    }
}
```

6−19 已知一棵二叉树的前序遍历的结果是 ABECDFGHIJ, 中序遍历的结果是 EBCDAFHIGJ, 试画出这棵二叉树。

【解答】

当前序序列为 ABECDFGHIJ，中序序列为 EBCDAFHIGJ 时，逐步形成二叉树的过程如下图所示：



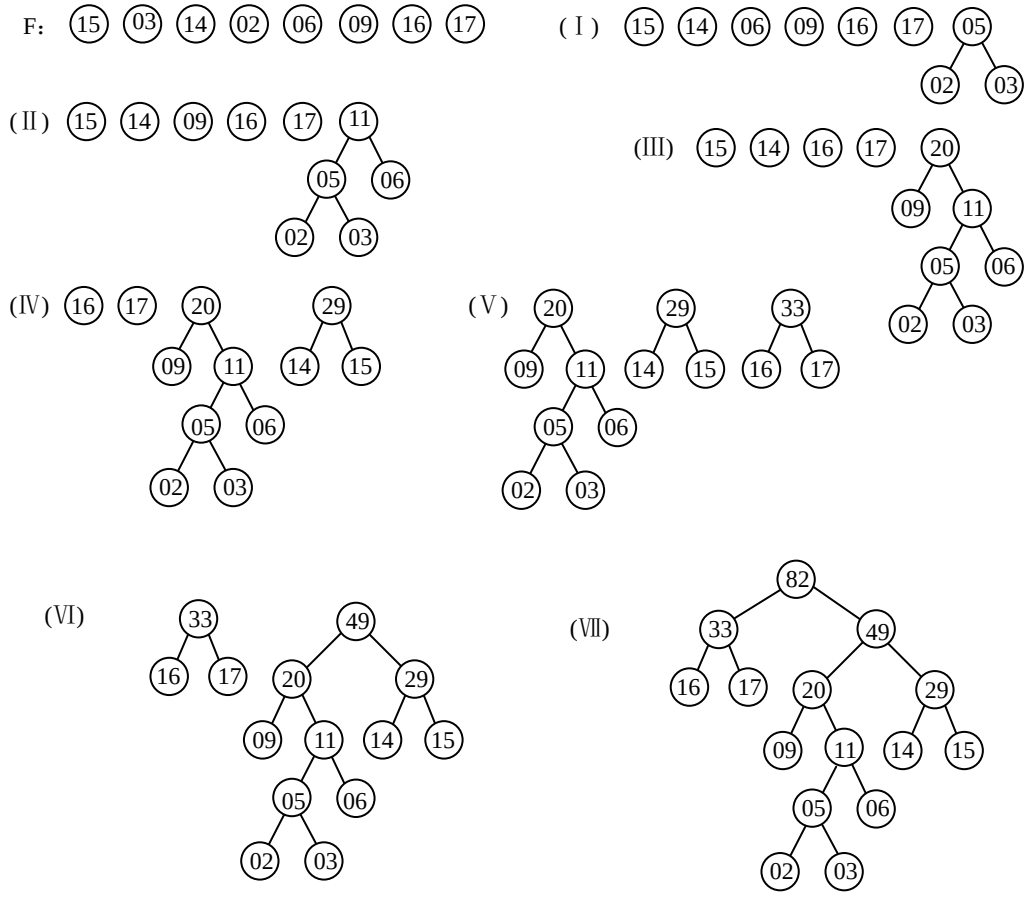

6−20 已知一棵树的先根次序遍历的结果与其对应二叉树表示(长子−兄弟表示)的前序遍历结果相同, 树的后根次序遍历结果与其对应二叉树表示的中序遍历结果相同。试问利用树的先根次序遍历结果和后根次序遍历结果能否唯一确定一棵树？举例说明。

【解答】

因为给出二叉树的前序遍历序列和中序遍历序列能够唯一地确定这棵二叉树， 能够唯一地确定一棵树。

6−21 给定权值集合{15, 03, 14, 02, 06, 09, 16, 17}, 构造相应的霍夫曼树, 并计算它的带权外部路径长度。

【解答】







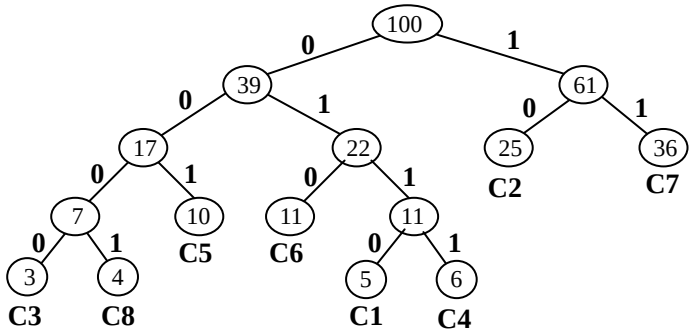
此树的带权路径长度 WPL = 229。

6-22 假定用于通信的电文仅由 8 个字母 c1, c2, c3, c4, c5, c6, c7, c8 组成，各字母在电文中出现的频率分别为 5, 25, 3, 6, 10, 11, 36, 4。试为这 8 个字母设计不等长 Huffman 编码，并给出该电文的总码数。

【解答】已知字母集 { c1, c2, c3, c4, c5, c6, c7, c8 }，频率 {5, 25, 3, 6, 10, 11, 36, 4 }，则 Huffman 编码为

| c1 | c2 | c3 | c4 | c5 | c6 | c7 | c8 |
|---|---|---|---|---|---|---|---|
| 0110 | 10 | 0000 | 0111 | 001 | 010 | 11 | 0001 |

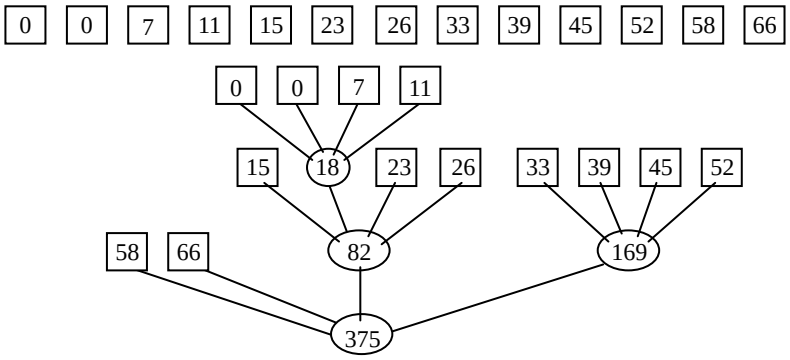
电文总码数为 4 * 5 + 2 * 25 + 4 * 3 + 4 * 6 + 3 * 10 + 3 * 11 + 2 * 36 + 4 * 4= 257




6-23 给定一组权值: 23, 15, 66, 07, 11, 45, 33, 52, 39, 26, 58, 试构造一棵具有最小带权外部路径长度的扩充 4 叉树，要求该 4 叉树中所有内部结点的度都是 4，所有外部结点的度都是 0。这棵扩充 4 叉树的带权外部路径长度是多少?

【解答】权值个数 n = 11，扩充 4 叉树的内结点的度都为 4，而外结点的度都为 0。设内结点个数为 $n_4$，外结点个数为 $n_0$，则可证明有关系 $n_0 = 3 * n_4 + 1$。由于在本题中 $n_0 = 11 \neq 3 * n_4 + 1$，需要补 2 个权值为 0 的外结点。此时内结点个数 $n_4 = 4$。

仿照霍夫曼树的构造方法来构造扩充 4 叉树，每次合并 4 个结点。




此树的带权路径长度 WPL = 375 + 82 + 169 + 18 = 644。

# 第七章 集合与搜索

7-1 设 A = { 1, 2, 3 }, B = { 3, 4, 5 }，求下列结果：

(1) A + B　　(2) A * B　　(3) A - B

(4) A.Contains (1)　　(5) A.AddMember (1)　　(6) A.DelMember (1)　　(7) A.Min ( )

【解答】

(1) 集合的并 A + B = { 1, 2, 3, 4, 5 }

(2) 集合的交 A * B = { 3 }

(3) 集合的差 A - B = { 1, 2 }

(4) 包含 A.Contains (1) = 1，表示运算结果为"True"

(5) 增加 A.AddMember (1)，集合中仍为{ 1, 2, 3 }，因为增加的是重复元素，所以不加入

(6) 删除 A.DelMember (1)，集合中为{ 2, 3 }

(7) 求最小元素 A.Min ( )，结果为 1

7-2 试编写一个算法，打印一个有穷集合中的所有成员。要求使用集合抽象数据类型中的基本操作。如果集合中包含有子集合，各个子集合之间没有重复的元素，采用什么结构比较合适。

【解答】

集合抽象数据类型的部分内容

```
template <class Type> class Set {
//对象: 零个或多个成员的聚集。其中所有成员的类型是一致的, 但没有一个成员是相同的。
    int Contains ( const Type x );                  //判元素 x 是否集合 this 的成员
    int SubSet ( Set <Type>& right );               //判集合 this 是否集合 right 的子集
    int operator == ( Set <Type>& right );          //判集合 this 与集合 right 是否相等
    int Elemtype ( );                               //返回集合元素的类型
    Type GetData ( );                               //返回集合原子元素的值
    char GetName ( );                                   //返回集合 this 的集合名
    Set <Type>* GetSubSet ( );                      //返回集合 this 的子集合地址
    Set <Type>* GetNext ( );                            //返回集合 this 的直接后继集合元素
    int IsEmpty ( );                              //判断集合 this 是否空。空则返回 1, 否则返回 0
};

ostream& operator << ( ostream& out, Set <Type> t ) {
//友元函数, 将集合 t 输出到输出流对象 out。
    t.traverse ( out, t );    return out;
}

void traverse ( ostream& out, Set <Type> s ) {
//友元函数, 集合的遍历算法
    if ( s.IsEmpty ( ) == 0 ) {                             //集合元素不空
        if ( s.Elemtype ( ) == 0 ) out << s.GetName ( ) << '{';     //输出集合名及花括号
```

```
        else if ( s.Elemtype ( ) == 1 ) {                    //集合原子元素
                out << s.GetData ( );                        //输出原子元素的值
                if ( s.GetNext ( ) != NULL ) out << ',';
            }
        else {                                               //子集合
                traverse ( s. GetSubSet ( ) );               //输出子集合
                if ( s.GetNext ( ) != NULL ) out << ',';
        }
        traverse ( s.GetNext ( ) );                          //向同一集合下一元素搜索
    }
    else out << '}';
}
```

如果集合中包含有子集合，各个子集合之间没有重复的元素，采用广义表结构比较合适。也可以使用并查集结构。

7-3 当全集合可以映射成 1 到 N 之间的整数时，可以用位数组来表示它的任一子集合。当全集合是下列集合时，应当建立什么样的映射？用映射对照表表示。

(1) 整数 0, 1, …, 99

(2) 从 n 到 m 的所有整数，$n \le m$

(3) 整数 n, n+2, n+4, …, n+2k

(4) 字母 'a', 'b', 'c', …, 'z'

(5) 两个字母组成的字符串，其中，每个字母取自 'a', 'b', 'c', …, 'z'。

【解答】

(1) i → i 的映射关系，i = 0, 1, 2, …, 99

(2) i → n−i 的映射关系，i = n, n+1, n+2, …, m

| 0 | 1 | 2 | | m−n |
|---|---|---|---|---|
| n | n+1 | n+2 | … | m |

(3) i → (i−n)/2 的映射关系，i = n, n+2, n+4, …, n+2k

| 0 | 1 | 2 | | k |
|---|---|---|---|---|
| n | n+2 | n+4 | … | n+2k |

(4) ord (c) → ord (c) − ord ('a') 的映射关系，c = 'a', 'b', 'c', …, 'z'

| 0 | 1 | 2 | | 25 |
|---|---|---|---|---|
| 'a' | 'b' | 'c' | … | 'z' |

(5) (ord (c1) − ord ('a') )*26 + ord (c2) − ord ('a')的映射关系，c1 = c2 = 'a', 'b', 'c', …, 'z'

| 0 | 1 | 2 | | 675 |
|---|---|---|---|---|
| 'aa' | 'ab' | 'ba' | … | 'zz' |

7-4 试证明：集合 A 是集合 B 的子集的充分必要条件是集合 A 和集合 B 的交集是 A。

【证明】

必要条件：因为集合 A 是集合 B 的子集，有 $A \subseteq B$，此时，对于任一 $x \in A$，必有 $x \in B$，因此可以推得 $x \in A \cap B$，就是说，如果 A 是 B 的子集，一定有 $A \cap B = A$。

充分条件：如果集合 A 和集合 B 的交集 A∩B 是 A，则对于任一 x ∈ A，一定有 x ∈ A ∩B，因此可以推得 x ∈ B，由此可得 A⊆B，即集合 A 是集合 B 的子集。

7-5 试证明：集合 A 是集合 B 的子集的充分必要条件是集合 A 和集合 B 的并集是 B。

【证明】

必要条件：因为集合 A 是集合 B 的子集，有 A⊆B，此时，对于任一 x ∈ A，必有 x ∈ B，它一定在 A∪B 中。另一方面，对于那些 x' ∉ A, 但 x' ∈ B 的元素，它也必在 A∪B 中，因此可以得出结论：凡是属于集合 B 的元素一定在 A∪B 中，A∪B = B。

充分条件：如果存在元素 x ∈ A 且 x ∉ B，有 x ∈ A∪B，但这不符合集合 A 和集合 B 的并集 A∪B 是 B 的要求。集合的并 A∪B 是集合 B 的要求表明，对于任一 x ∈ A∪B，同时应有 x ∈ B。对于那些满足 x' ∈ A 的 x'，既然 x' ∈ A∪B，也应当 x' ∈ B，因此，在此种情况下集合 A 应是集合 B 的子集。

7-6 设+、*、-是集合的或、与、差运算，试举一个例子，验证

A + B = (A - B) + (B - A) + A * B

【解答】

若设集合 A = { 1, 3, 4, 7, 9, 15 }，集合 B = { 2, 3, 5, 6, 7, 12, 15, 17 }。则

A + B = { 1, 2, 3, 4, 5, 6, 7, 9, 12, 15, 17 }

又 A * B = { 3, 7, 15 }，　A – B = { 1, 4, 9 }，　B – A = { 2, 5, 6, 12, 17 }

则　　(A – B) + (B – A) + A * B = { 1, 2, 3, 4, 5, 6, 7, 9, 12, 15, 17 }

有　A + B = (A – B) + ( B – A ) + A * B。

7-7 给定一个用无序链表表示的集合，需要在其上执行 operator+( ), operator*( ), operator- ( ), Contains(x), AddMember (x), DelMember(x), Min( )，试写出它的类声明，并给出所有这些成员函数的实现。

【解答】

下面给出用无序链表表示集合时的类的声明。

```
template <class Type> class Set;                          //用以表示集合的无序链表的类的前视定义
template <class Type> class SetNode {                     //集合的结点类定义
friend class SetList<Type>;
private:
    Type data;                                            //每个成员的数据
    SetNode <Type> *link;                                 //链接指针
public:
    SetNode (const Type& item ) : data (item), link (NULL); //构造函数
};

template <class Type> class Set {                         //集合的类定义
private:
    SetNode <Type> *first, *last;                         //无序链表的表头指针，表尾指针
public:
    SetList ( ) { first = last = new SetNode <Type>(0); }  //构造函数
    ~SetList ( ) { MakeEmpty ( );   delete first; }       //析构函数
```

```
    void MakeEmpty ( );                                  //置空集合
    int AddMember ( const Type& x );                     //把新元素 x 加入到集合之中
    int DelMember ( const Type& x );                     //把集合中成员 x 删去
    Set <Type>& operator = ( Set <Type>& right );        //复制集合 right 到 this。
    Set <Type>& operator + ( Set <Type>& right );        //求集合 this 与集合 right 的并
    Set <Type>& operator * ( Set <Type>& right );        //求集合 this 与集合 right 的交
    Set <Type>& operator − ( Set <Type>& right );        //求集合 this 与集合 right 的差
    int Contains ( const Type& x );                      //判 x 是否集合的成员
    int operator == ( Set <Type>& right );               //判集合 this 与集合 right 相等
    Type& Min ( );                                       //返回集合中的最小元素的值
}
```

(1) **operator** + ( )

```
template <class Type> Set <Type>& Set <Type> :: operator + ( Set <Type>& right ) {
//求集合 this 与集合 right 的并，计算结果通过临时集合 temp 返回，this 集合与 right 集合不变。
    SetNode <Type> *pb = right.first−>link;              //right 集合的链扫描指针
    SetNode <Type> *pa, *pc;                             //this 集合的链扫描指针和结果链的存放指针
    Set <Type> temp;
    pa = first−>link;    pc = temp.first;
    while ( pa != NULL ) {                               //首先把集合 this 的所有元素复制到结果链
        pc−>link = new SetNode<Type> ( pa−>data ) ;
        pa = pa−>link;    pc = pc−>link;
    }
    while ( pb != NULL ) {                               //将集合 right 中元素一个个拿出到 this 中查重
        pa = first−>link;
        while ( pa != NULL && pa−>data != pb−>data ) pa = pa−>link;
        if ( pa == NULL )                                //在集合 this 中未出现，链入到结果链
            { pc−>link = new SetNode<Type> ( pa−>data );    pc = pc−>link; }
        pb = pb−>link;
    }
    pc−>link = NULL;    last = pc;                       //链表收尾
    return temp;
}
```

(2) **operator** * ( )

```
template <class Type> Set <Type>& Set <Type> :: operator * ( Set <Type>& right ) {
//求集合 this 与集合 right 的交，计算结果通过临时集合 temp 返回，this 集合与 right 集合不变。
    SetNode<Type> *pb = right.first−>link;               //right 集合的链扫描指针
    Set <Type> temp;
    SetNode <Type> *pc = temp. first;                    //结果链的存放指针
    while ( pb != NULL ) {                               //将集合 right 中元素一个个拿出到 this 中查重
        SetNode<Type> *pa = first−>link;                 //this 集合的链扫描指针
        while ( pa != NULL ) {
            if ( pa−>data == pb−>data )                  //两集合公有的元素，插入到结果链
```

```
                { pc->link = new SetNode<Type> ( pa->data );    pc = pc->link; }
            pa = pa->link;
        }
        pb = pb->link;
    }
    pc->link = NULL;    last = pc;                          //置链尾指针
    return temp;
}
```

(3) **operator** − ( ),

```
template <class Type> Set <Type>& Set <Type> :: operator − ( Set <Type>& right ) {
//求集合 this 与集合 right 的差, 计算结果通过临时集合 temp 返回，this 集合与 right 集合不变。
    SetNode<Type> *pa = first->link;                        //this 集合的链扫描指针
    Set <Type> temp;
    SetNode<Type> *pc = temp->first;                        //结果链的存放指针
    while ( pa != NULL ) {                                  //将集合 this 中元素一个个拿出到 right 中查重
        SetNode <Type> *pb = right.first->link;             //right 集合的链扫描指针
        while ( pb != NULL && pa->data != pb->data )
            pb = pb->link;
        if ( pb == NULL )                                   //此 this 中的元素在 right 中未找到, 插入
            { pc->link = new SetNode <Type> ( pa->data );    pc = pc->link; }
        pa = pa->link;
    }
    pc->link = NULL;    last = pc;                          //链表收尾
    return temp;
}
```

(4) Contains(x)

```
template <class Type> int Set <Type> :: Contains ( const Type& x ) {
//测试函数: 如果 x 是集合的成员, 则函数返回 1, 否则返回 0。
    SetNode<Type> * temp = first->link;                     //链的扫描指针
    while ( temp != NULL && temp->data != x ) temp = temp->link;    //循链搜索
    if ( temp != NULL ) return 1;                           //找到, 返回 1
    else return 0;                                          //未找到, 返回 0
}
```

(5) AddMember (x)

```
template <class Type> int Set <Type> :: AddMember ( const Type& x ) {
//把新元素 x 加入到集合之中。若集合中已有此元素, 则函数返回 0, 否则函数返回 1。
    SetNode<Type> * temp = first->link;                     // temp 是扫描指针
    while ( temp != NULL && temp->data != x ) temp = temp->link;    /循链扫描
    if ( temp != NULL ) return 0;                           //集合中已有此元素, 不加
    last = last->link = new SetNode (x);                    //否则, 创建数据值为 x 的新结点, 链入
    return 1;
```

```
}
```

(6) DelMember (x)

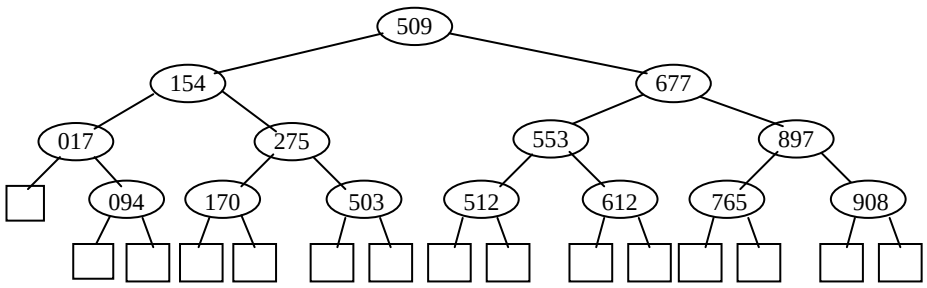
```
template <class Type> int Set <Type> :: DelMember ( const Type& x ) {
//把集合中成员 x 删去。若集合不空且元素 x 在集合中，则函数返回 1，否则返回 0。
    SetNode<Type> * p = first->link,   *q = first;
    while ( p != NULL ) {
        if ( p->data == x ) {                          //找到
            q->link = p->link;                         //重新链接
            if ( p == last ) last = q;                 //删去链尾结点时改链尾指针
            delete p;   return 1;                      //删除含 x 结点
        }
        else { q = p;   p = p->link; }                 //循链扫描
    return 0;                                          //集合中无此元素
}
```

(7) Min ( )

```
template <class Type> SetNode<Type> * Set <Type> :: Min ( ) {
//在集合中寻找值最小的成员并返回它的位置。
    SetNode<Type> * p = first->link,   *q = first->link;        //p 是检测指针, q 是记忆最小指针
    while ( p != NULL ) {
        if ( p->data < q->data ) q = p;                //找到更小的，让 q 记忆它
        p = p->link;                                   //继续检测
    }
    return q;
}
```

7-8 设有序顺序表中的元素依次为 017, 094, 154, 170, 275, 503, 509, 512, 553, 612, 677, 765, 897, 908。试画出对其进行折半搜索时的二叉搜索树，并计算搜索成功的平均搜索长度和搜索不成功的平均搜索长度。

【解答】




$$ASL_{succ} = \frac{1}{14}\sum_{i=1}^{14} C_i = \frac{1}{14}(1+2*2+3*4+4*7) = \frac{45}{14}$$

$$ASL_{unsucc} = \frac{1}{15}\sum_{i=0}^{15} C_i' = \frac{1}{15}(3*1+4*14) = \frac{59}{15}$$

7-9 若对有 n 个元素的有序顺序表和无序顺序表进行顺序搜索，试就下列三种情况分别讨论两者在等搜索概率时的平均搜索长度是否相同?

(1) 搜索失败;

(2) 搜索成功，且表中只有一个关键码等于给定值 k 的对象;

(3) 搜索成功，且表中有若干个关键码等于给定值 k 的对象，要求一次搜索找出所有对象。

【解答】

(1) 不同。因为有序顺序表搜索到其关键码比要查找值大的对象时就停止搜索，报告失败信息，不必搜索到表尾；而无序顺序表必须搜索到表尾才能断定搜索失败。

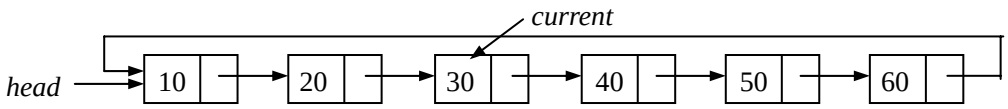
(2) 相同。搜索到表中对象的关键码等于给定值时就停止搜索，报告成功信息。

(3) 不同。有序顺序表中关键码相等的对象相继排列在一起，只要搜索到第一个就可以连续搜索到其它关键码相同的对象。而无序顺序表必须搜索全部表中对象才能确定相同关键码的对象都找了出来，所需时间就不相同了。

前两问可做定量分析。第三问推导出的公式比较复杂，不再进一步讨论。

7-10 假定用一个循环链表来实现一个有序表，并让指针 head 指向具有最小关键码的结点。指针 current 初始时等于 head，每次搜索后指向当前检索的结点，但如果搜索不成功则 current 重置为 head。试编写一个函数 search(head, current, key)实现这种搜索。当搜索成功时函数返回被检索的结点地址，若搜索不成功则函数返回空指针 0。请说明如何保持指针 current 以减少搜索时的平均搜索长度。

【解答】

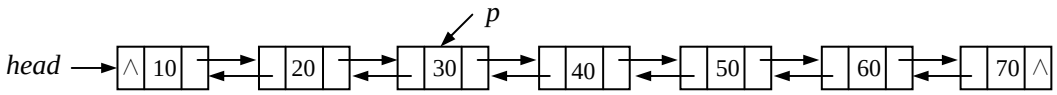



相应的搜索函数可以定义为链表及链表结点类的友元函数，直接使用链表及链表结点类的私有数据成员。

```
template<class Type>
ListNode <Type> * Search ( ListNode<Type> * head, ListNode<Type> *& current, Type key ) {
    ListNode <Type> * p, * q;
    if ( key < current ) { p = head;   q = current; }          //确定检测范围, 用 p, q 指示
    else { p = current;   q = head; }
    while ( p != q && p->data < key ) p = p->link;             //循链搜索其值等于 key 的结点
    if ( p->data == key ) { current = p;   return p; }         //找到, 返回结点地址
    else { current = head;   return NULL; }                    //未找到, 返回空指针
}
```

7-11 考虑用双向链表来实现一个有序表，使得能在这个表中进行正向和反向搜索。若指针 p 总是指向最后成功搜索到的结点，搜索可以从 p 指示的结点出发沿任一方向进行。试根据这种情况编写一个函数 search(head, p, key)，检索具有关键码 key 的结点，并相应地修改 p。最后请给出搜索成功和搜索不成功时的平均搜索长度。

【解答】



```
template <class Type>
DblListNode<Type> * Search ( DblListNode<Type> * head,  DblListNode<Type> *& p, Type key ) {
//在以 head 为表头的双向有序链表中搜索具有值 key 的结点。算法可视为双向链表类和双向链表结点类的友元
//函数。若给定值 key 大于结点 p 中的数据，从 p 向右正向搜索，否则，从 p 向左反向搜索。
   DblListNode<Type> * q = p;
   if ( key < p->data ){ while ( q != NULL && q->data > key ) q = q-> lLink; }          //反向搜索
   else { while ( q != NULL && q->data < key ) q = q-> rLink; }                        //正向搜索
   if ( q != NULL && q->data == key ) { p = q;   return p; }                           //搜索成功
   else return NULL;
}
```

如果指针 p 处于第 i 个结点（i = 1, 2, …, n），它左边有 i−1 个结点，右边有 n−i 个结点。找到左边第 i−1 号结点比较 2 次，找到第 i−2 号结点比较 3 次，…，找到第 1 号结点比较 i 次，一般地，找到左边第 k 个结点比较 i−k+1 次（k = 1, 2, …, i−1）。找到右边第 i+1 号结点比较 2 次，找到第 i+2 号结点比较 3 次，…，找到第 n 号结点比较 n−i+1 次，一般地，找到右边第 k 个结点比较 k−i+1 次（k = i+1, i+2, …, n）。因此，当指针处于第 i 个结点时的搜索成功的平均数据比较次数为

$$\left(1+\sum_{k=1}^{i-1}(i-k+1)+\sum_{k=i+1}^{n}(k-i+1)\right)\Big/n=\left(\frac{n*(n+3)}{2}+i^2-i-i*n\right)\Big/n=\frac{n+3}{2}+\frac{i^2-i-i*n}{n}$$

一般地，搜索成功的平均数据比较次数为

$$ASL_{succ}=\frac{1}{n}\sum_{i=1}^{n}\left(\frac{n+3}{2}+\frac{i^2-i-i*n}{n}\right)=\frac{n^2+3n-1}{3n}$$

如果指针 p 处于第 i 个结点（i = 1, 2, …, n），它左边有 i 个不成功的位置，右边有 n−i+1 个不成功的位置。

$$\left(\sum_{k=0}^{i-1}(i-k)+\sum_{k=i}^{n}(k-i+1)\right)\Big/(n+1)=\left(\frac{n*(n+3)}{2}+i^2-i-i*n+1\right)\Big/(n+1)$$

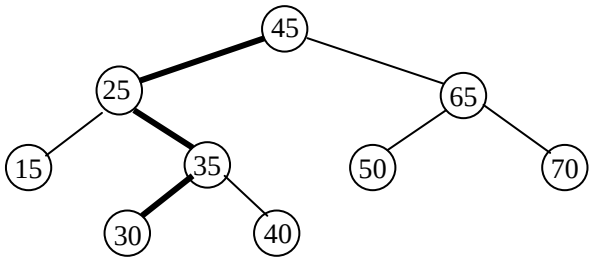
一般地，搜索不成功的平均数据比较次数为

$$ASL_{unsucc}=\frac{1}{(n+1)^2}\sum_{i=0}^{n}\left(\frac{n*(n+3)}{2}+i^2-i-i*n+1\right)=\frac{2n^2+7n+6}{n+1}$$

7-12 在一棵表示有序集 S 的二叉搜索树中，任意一条从根到叶结点的路径将 S 分为 3 部分：在该路径左边结点中的元素组成的集合 $S_1$；在该路径上的结点中的元素组成的集合 $S_2$；在该路径右边结点中的元素组成的集合 $S_3$。$S = S_1 \cup S_2 \cup S_3$。若对于任意的 $a \in S_1, b \in S_2, c \in S_3$， 是否总有 $a \le b \le c$？为什么？

【解答】

答案是否定的。举个反例：看下图粗线所示的路径




S1 = { 15 }，　S2 = { 25, 30, 35, 45 }，　S3 = { 40, 50, 65, 70 }

$c = 40 \in S3$，$b = 45 \in S2$，$b \le c$ 不成立。

7-13 请给出下列操作序列运算的结果：Union(1, 2), Union(3, 4), Union(3, 5), Union(1, 7), Union(3, 6), Union(8, 9), Union(1, 8), Union(3, 10), Union(3, 11), Union(3, 12), Union(3, 13), Union(14, 15), Union(16, 17), Union(14, 16), Union(1, 3), Union(1, 14)，要求

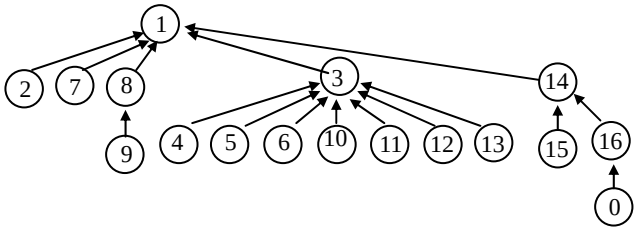
(1) 以任意方式执行 Union；

(2) 根据树的高度执行 Union；

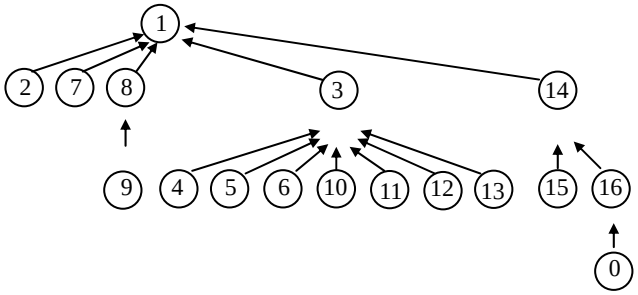
(3) 根据树中结点个数执行 Union。

【解答】

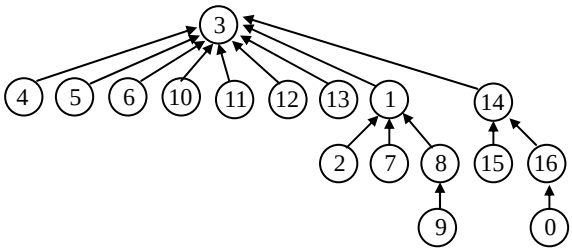
(1) 对于 union(i, j)，以 i 作为 j 的双亲




(2) 按 i 和 j 为根的树的高度实现 union(i, j)，高度大者为高度小者的双亲；




(3) 按 i 和 j 为根的树的结点个数实现 union(i, j)，结点个数大者为结点个数小者的双亲。



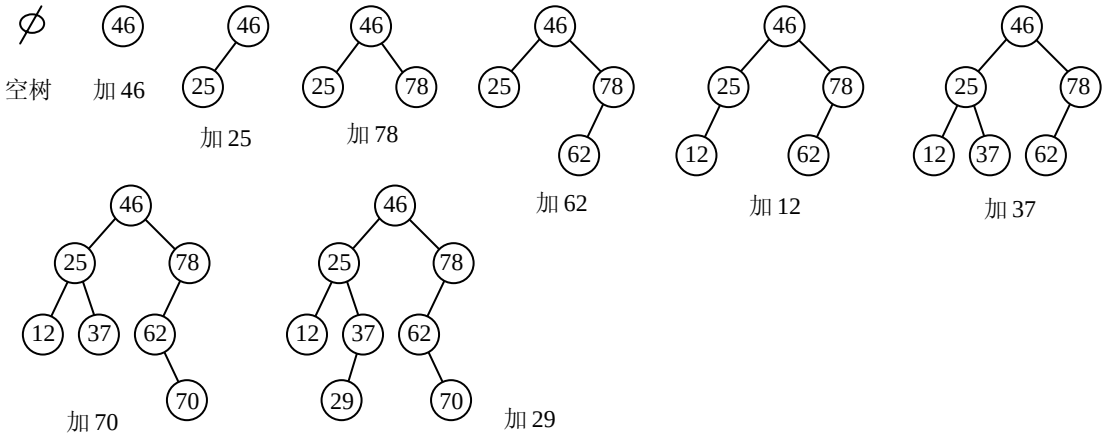
7-14 有 n 个结点的二叉搜索树具有多少种不同形态?

【解答】

$$\frac{1}{n+1}C_{2n}^{n}$$

7-15 设有一个输入数据的序列是 **{** 46, 25, 78, 62, 12, 37, 70, 29 **}**, 试画出从空树起，逐个输入各个数据而生成的二叉搜索树。

【解答】



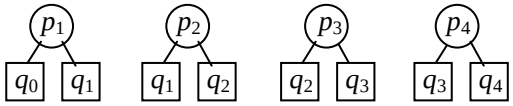

7-16 设有一个标识符序列{else, public, return, template}，$p_1$=0.05, $p_2$=0.2, $p_3$=0.1, $p_4$=0.05, $q_0$=0.2, $q_1$=0.1, $q_2$=0.2, $q_3$=0.05, $q_4$=0.05，计算 W[i][j]、C[i][j]和 R[i][j]，构造最优二叉搜索树。

【解答】

将标识符序列简化为 { e, p, r, t }，并将各个搜索概率值化整，有

| e | p | r | t |
|---|---|---|---|
| $p_1$ = 1 | $p_2$ = 4 | $p_3$ = 2 | $p_4$ = 1 |
| $q_0$ = 4 | $q_1$ = 2 | $q_2$ = 4 | $q_3$ = 1 | $q_4$ = 1 |

(1) 首先构造只有一个内结点的最优二叉搜索树:




三个矩阵的内容如下:

| | 0 | 1 | 2 | 3 | 4 | |
|---|---|---|---|---|---|---|
| 0 | 4 | 7 | 15 | 18 | 20 | 0 |
| 1 | | 2 | 10 | 13 | 15 | 1 |
| 2 | | | 4 | 7 | 9 | 2 |
| 3 | | | | 1 | 3 | 3 |
| 4 | | | | | 1 | 4 |

W[i][j]

| | 0 | 1 | 2 | 3 | 4 | |
|---|---|---|---|---|---|---|
| 0 | 0 | 7 | 22 | 32 | 39 | 0 |
| 1 | | 0 | 10 | 20 | 27 | 1 |
| 2 | | | 0 | 7 | 12 | 2 |
| 3 | | | | 0 | 3 | 3 |
| 4 | | | | | 0 | 4 |

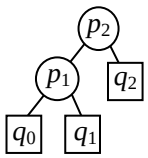
C[i][j]

| | 0 | 1 | 2 | 3 | 4 |
|---|---|---|---|---|---|
| 0 | 0 | 1 | 2 | 2 | 2 |
| 1 | | 0 | 2 | 2 | 2 |
| 2 | | | 0 | 3 | 3 |
| 3 | | | | 0 | 4 |
| 4 | | | | | 0 |

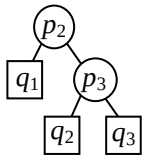
R[i][j]

(2) 构造具有两个内结点的最优二叉搜索树

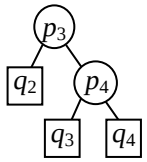
C[0][2] = 22
T[0][2] = 2




C[1][3] = 20
T[1][3] = 2

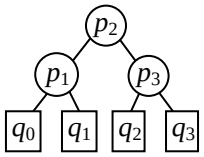



C[2][4] = 12
T[2][4] = 2

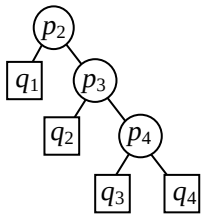



(3) 构造具有三个内结点的最优二叉搜索树

C[0][3] = 32
T[0][3] = 2

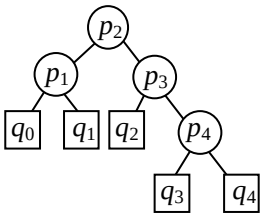



C[1][4] = 27
T[1][4] = 2




(4) 构造具有四个内结点的最优二叉搜索树

C[0][4] = 39
T[0][4] = 2




左子树 T[0][1], 其 C[0][1] = 7
右子树 T[2][4], 其 C[2][4] = 12

7-17 在二叉搜索树上删除一个有两个子女的结点时，可以采用以下三种方法：

(1) 用左子树 $T_L$ 上具有最大关键码的结点 X 顶替，再递归地删除 X。

(2) 交替地用左子树 $T_L$ 上具有最大关键码的结点和右子树 $T_R$ 上具有最小关键码的结点顶替，再递归地删除适当的结点。

(3) 用左子树 $T_L$ 上具有最大关键码的结点或者用右子树 $T_R$ 上具有最小关键码的结点顶替，再递归地删除适当的结点。可随机选择其中一个方案。

试编写程序实现这三个删除方法，并用实例说明哪一个方法最易于达到平衡化。

【解答】

① 在被删结点有两个子女时用左子树 $T_L$ 中具最大关键码的结点顶替的算法：

```
template<class Type> BstNode<Type> * BST<Type> :: leftReplace ( BstNode<Type> * ptr ) {
    BstNode<Type> * temp = ptr->leftChild;                  //进到 ptr 的左子树
    while ( temp->rightChild != NULL ) temp = temp->rightChild;  //搜寻中序下最后一个结点
    ptr->data = temp->data;                                 //用该结点数据代替根结点数据
    return temp;
}
```

② 在被删结点有两个子女时用右子树 $T_R$ 中具最小关键码的结点顶替的算法：

```
    template<class Type> BstNode<Type> * BST<Type> :: rightReplace ( BstNode<Type> * ptr ) {
    BstNode<Type> * temp = ptr->rightChild;                 //进到 ptr 的右子树
    while ( temp->leftChild != NULL ) temp = temp->leftChild;    //搜寻中序下最后一个结点
    ptr->data = temp->data;                                 //用该结点数据代替根结点数据
    return temp;
}
```

(1) 用左子树 $T_L$ 上具有最大关键码的结点 X 顶替，再递归地删除 X。

```
template <class Type> void BST<Type> :: Remove ( Type& x,   BstNode<Type> *& ptr ) {
//私有函数：在以 ptr 为根的二叉搜索树中删除含 x 的结点。若删除成功则新根通过 ptr 返回。
```

```
    BstNode<Type> * temp;
    if ( ptr != NULL )
      if ( x < ptr->data ) Remove ( x, ptr->leftChild );              //在左子树中执行删除
      else if ( x > ptr->data ) Remove ( x, ptr->rightChild );        //在右子树中执行删除
        else if ( ptr->leftChild != NULL && ptr->rightChild != NULL ) {
                                                  // ptr 指示关键码为 x 的结点，它有两个子女
            temp = leftReplace ( ptr );           //在 ptr 的左子树中搜寻中序下最后一个结点顶替 x
            Remove ( ptr->data, temp );           //在 temp 为根的子树中删除该结点
          }
          else {                                  // ptr 指示关键码为 x 的结点，它只有一个或零个子女
              temp = ptr;
              if ( ptr->leftChild == NULL ) ptr = ptr->rightChild;        //只有右子女
              else if ( ptr->rightChild == NULL ) ptr = ptr->leftChild;   //只有左子女
              delete temp;
          }
}
```

(2) 交替地用左子树 $T_L$ 上具有最大关键码的结点和右子树 $T_R$ 上具有最小关键码的结点顶替，再递归地删除适当的结点。

```
template <class Type> void BST<Type> :: Remove ( Type& x,  BstNode<Type> *& ptr,  int& dir ) {
//私有函数：在以 ptr 为根的二叉搜索树中删除含 x 的结点。若删除成功则新根通过 ptr 返回。在参数
//表中有一个引用变量 dir, 作为调整方向的标记。若 dir = 0, 用左子树上具有最大关键码的结点顶替被
//删关键码; 若 dir = 1,用右子树上具有最小关键码的结点顶替被删关键码结点, 在调用它的程序中设
//定它的初始值为 0。
    BstNode<Type> * temp;
    if ( ptr != NULL )
      if ( x < ptr->data ) Remove ( x, ptr->leftChild, dir );          //在左子树中执行删除
      else if ( x > ptr->data ) Remove ( x, ptr->rightChild, dir );    //在右子树中执行删除
        else if ( ptr->leftChild != NULL && ptr->rightChild != NULL ) {
                                                  // ptr 指示关键码为 x 的结点，它有两个子女
          if ( dir == 0 ) {
            temp = leftReplace ( ptr );  dir = 1; //在 ptr 的左子树中搜寻中序下最后一个结点顶替 x
            } else {
             temp = rightReplace ( ptr );  dir = 0; //在 ptr 的右子树中搜寻中序下第一个结点顶替 x
            }
            Remove ( ptr->data, temp, dir );          //在 temp 为根的子树中删除该结点
          }
          else {                                  // ptr 指示关键码为 x 的结点，它只有一个或零个子女
              temp = ptr;
              if ( ptr->leftChild == NULL ) ptr = ptr->rightChild;        //只有右子女
              else if ( ptr->rightChild == NULL ) ptr = ptr->leftChild;   //只有左子女
              delete temp;
          }
}
```

(3) 用左子树 $T_L$ 上具有最大关键码的结点或者用右子树 $T_R$ 上具有最小关键码的结点顶替，再递归地删除适当的结点。可随机选择其中一个方案。

```
#include <stdlib.h>
template <class Type> void BST<Type> :: Remove ( Type& x,   BstNode<Type> *& ptr ) {
//私有函数：在以 ptr 为根的二叉搜索树中删除含 x 的结点。若删除成功则新根通过 ptr 返回。在程序中用到一个
//随机数发生器 rand( ), 产生 0~32767 之间的随机数, 将它除以 16384, 得到 0~2 之间的浮点数。若其大于 1, 用左
//子树上具有最大关键码的结点顶替被删关键码; 若其小于或等于 1, 用右子树上具有最小关键码的结点顶替被删
//关键码结点, 在调用它的程序中设定它的初始值为 0。
    BstNode<Type> * temp;
    if ( ptr != NULL )
      if ( x < ptr->data ) Remove ( x, ptr->leftChild );                  //在左子树中执行删除
      else if ( x > ptr->data ) Remove ( x, ptr->rightChild );            //在右子树中执行删除
          else if ( ptr->leftChild != NULL && ptr->rightChild != NULL ) {
                                                          // ptr 指示关键码为 x 的结点，它有两个子女
              if ( (float) ( rand ( ) / 16384 ) > 1 ) {
                  temp = leftReplace ( ptr );   dir = 1;   //在 ptr 的左子树中搜寻中序下最后一个结点顶替 x
              } else {
                  temp = rightReplace ( ptr );   dir = 0;      //在 ptr 的右子树中搜寻中序下第一个结点顶替 x
              }
              Remove ( ptr->data, temp );                     //在 temp 为根的子树中删除该结点
          }
          else {                                          // ptr 指示关键码为 x 的结点，它只有一个或零个子女
              temp = ptr;
              if ( ptr->leftChild == NULL ) ptr = ptr->rightChild;            //只有右子女
              else if ( ptr->rightChild == NULL ) ptr = ptr->leftChild;       //只有左子女
              delete temp;
          }
}
```

7-18 (1) 设 T 是具有 n 个内结点的扩充二叉搜索树, I 是它的内路径长度, E 是它的外路径长度。试利用归纳法证明 $E = I + 2n,\quad n \geq 1$。

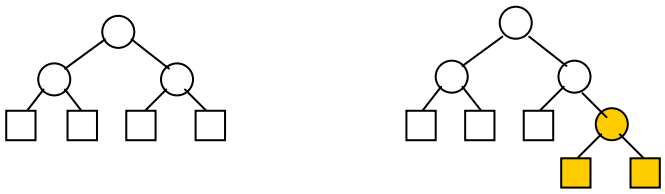
(2) 利用(1)的结果, 试说明: 成功搜索的平均搜索长度 $S_n$ 与不成功搜索的平均搜索长度 $U_n$ 之间的关系可用公式

$$S_n = ( 1 + 1/n ) U_n - 1,\ n \geq 1$$

表示。

【解答】

(1) 用数学归纳法证明。当 n = 1 时，有 1 个内结点(I = 0)，2 个外结点(E = 2)，满足 E = I+2n。设 n = k 时结论成立，$E_k = I_k + 2k$。则当 n = k + 1 时，将增加一个层次为 l 的内结点，代替一个层次为 l 的外结点，同时在第 l+1 层增加 2 个外结点，则 $E_{k+1} = E_k - l + 2*(l+1) = E_k + l + 2$，$I_{k+1} = I_k + l$，将 $E_k = I_k + 2k$ 代入，有 $E_{k+1} = E_k + l + 2 = I_k + 2k + l + 2 = I_{k+1} + 2(k+1)$，结论得证。



(2) 因为搜索成功的平均搜索长度 $S_n$ 与搜索不成功的平均搜索长度 $U_n$ 分别为

$$S_n = \frac{1}{n}\sum_{i=1}^{n}(c_i + 1) = \frac{1}{n}\sum_{i=1}^{n} c_i + 1 = \frac{1}{n} I_n + 1$$

$$U_n = \frac{1}{n+1}\sum_{j=0}^{n} c_j = \frac{1}{n+1} E_k$$

其中，$c_i$ 是各内结点所处层次，$c_j$ 是各外结点所处层次。因此有

$(n+1)U_n = E_n = I_n + 2n = nS_n - n + 2n = nS_n + n$

$$\frac{n+1}{n}U_n = S_n + 1 \quad \Longrightarrow \quad S_n = \left(1 + \frac{1}{n}\right)U_n - 1$$

7-19 求最优二叉搜索树的算法的计算时间为 $O(n^3)$，下面给出一个求拟最优二叉搜索树的试探算法，可将计算时间降低到 $O(n\log_2 n)$。算法的思想是对于关键码序列{ $key_l$, $key_{l+1}$, …, $key_h$ }， 轮流以 $key_k$ 为根，k = l, l+1, …, h，求使得 |W[l−1][k−1] − W[k][h]| 达到最小的 k，用 $key_k$ 作为由该序列构成的拟最优二叉搜索树的根。然后对以 $key_k$ 为界的左子序列和右子序列，分别施行同样的操作，建立根 $key_k$ 的左子树和右子树。要求：

(1) 使用 7.17 题的数据，执行这个试探算法建立拟最优二叉搜索树，该树建立的时间代价是多少？

(2) 编写一个函数，实现上述试探算法。要求该函数的时间复杂度应为 $O(n\log_2 n)$。

【解答】

(1) 各个关键码的权值为 { $p_1$, $p_2$, $p_3$, $p_4$ }，利用题 7-17 中的 W 矩阵，轮流让 k = 1, 2, 3, 4 为根，此时，下界 l = 1, 上界 h = 4。有

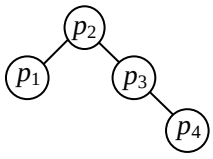
$$\min\left|W[l-1][k-1] - W[k][h]\right| = \left|W[0][1] - W[2][4]\right| = 2$$

求得 k = 2。则根结点为 2，左子树的权值为 { $p_1$ }，右子树的权值为 { $p_3$, $p_4$ }。

因为左子树只有一个结点，所以，权值为 p1 的关键码为左子树的根即可。　对于右子树 { $p_3$, $p_4$ }，采用上面的方法，轮流让 k = 3, 4 为根，此时，下界 l = 3，上界 h = 4。有

$$\min\left|W[l-1][k-1] - W[k][h]\right| = \left|W[2][2] - W[3][4]\right| = 1$$

求得 k = 3。于是以权值为 $p_3$ 的关键码为根，其左子树为空，右子树为 { $p_4$ }。这样，得到拟最优二叉搜索树的结构如下：




建立该拟最优二叉搜索树的时间代价为 O(4+1+2+1) = O(8)

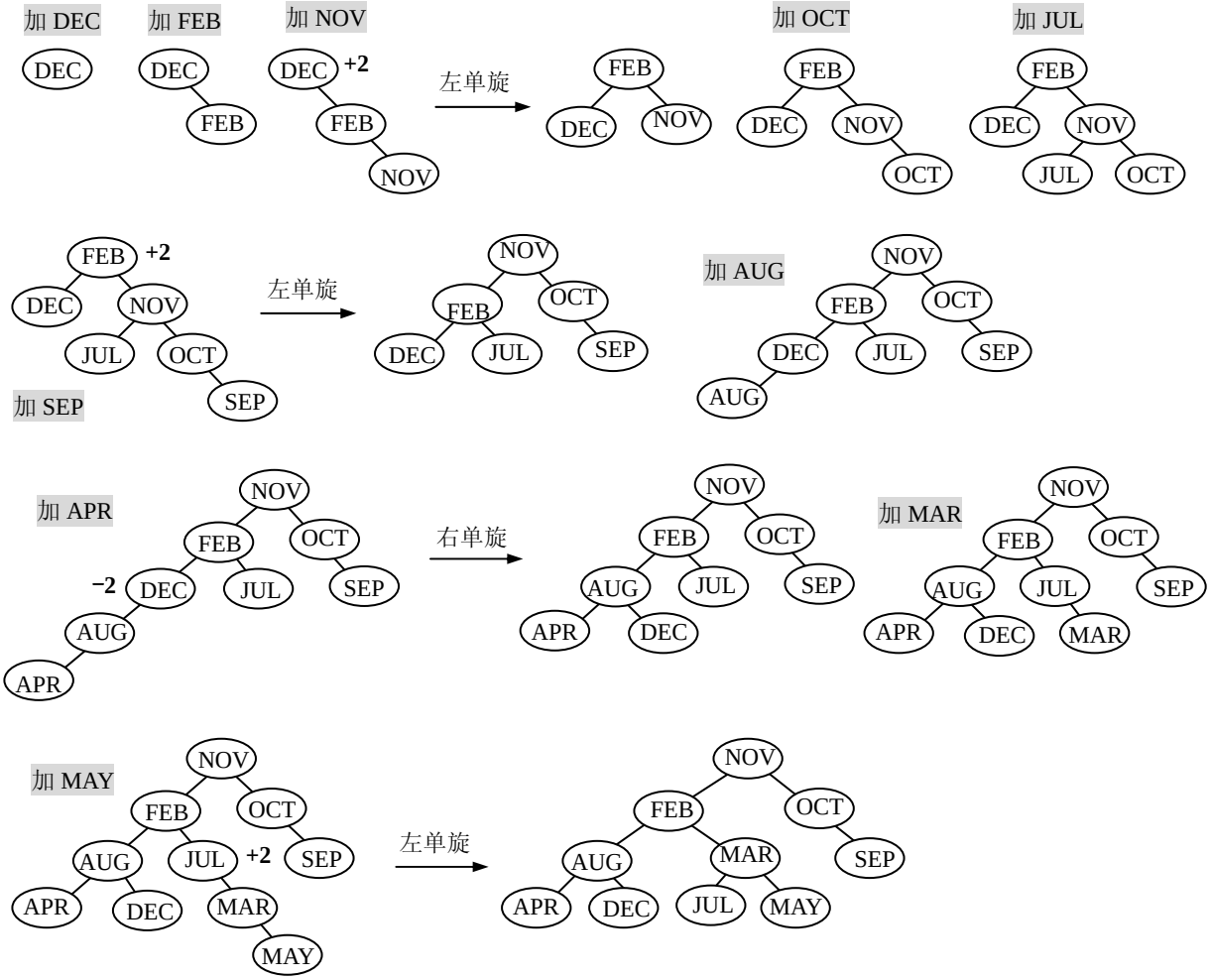
(2) 建立该拟最优二叉搜索树的算法

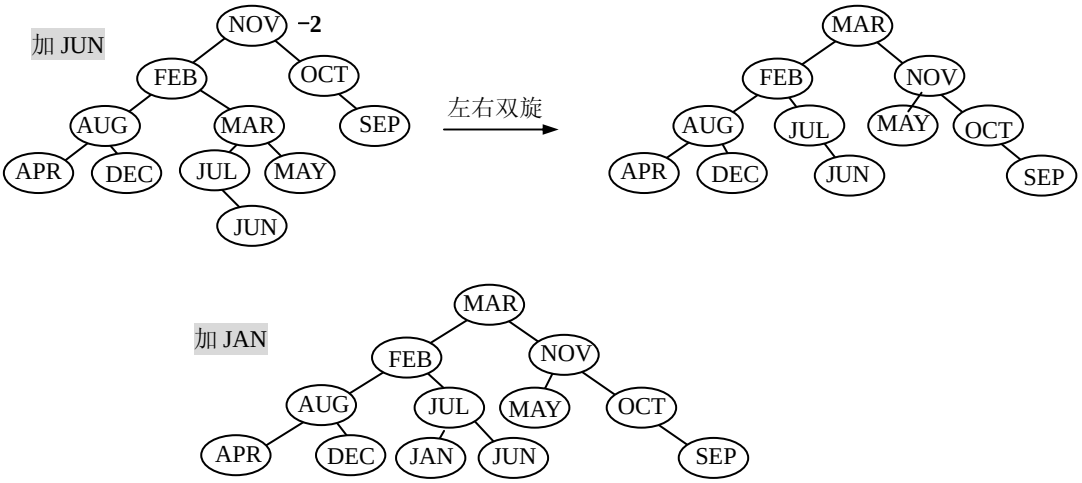
```
void nearOptiSrchTree ( int W[n+1][n+1], int n, int left, int right ) {
    if ( left > right ) { cout << "Empty Sequence! " << endl;   return; }
```

```
        if ( left == right ) { cout << left;    return; }
        int p = 0;    int k;
        for ( int j = left; j <= right; j++ )
              if ( p > abs ( W[left−1][j−1] – W[j][right] )
                    { p = abs ( W[left−1][j−1] – W[j][right] );    k = j; }
        cout << k;
        if ( k == left ) nearOptiSrchTree ( W[n+1][n+1], n, k+1, right );
        else    if ( k == right ) nearOptiSrchTree ( W[n+1][n+1], n, left, k−1 );
                 else { nearOptiSrchTree ( W[n+1][n+1], n, left, k−1 );
                        nearOptiSrchTree ( W[n+1][n+1], n, k+1, right );
                      }
}
```

7−20 将关键码 DEC, FEB, NOV, OCT, JUL, SEP, AUG, APR, MAR, MAY, JUN, JAN 依次插入到一棵初始为空的 AVL 树中，画出每插入一个关键码后的 AVL 树，并标明平衡旋转的类型。

【解答】

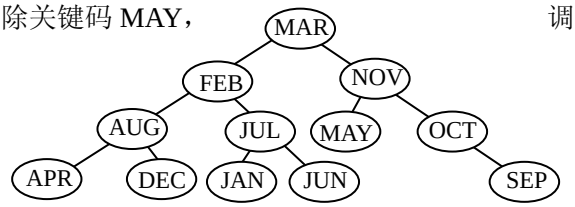




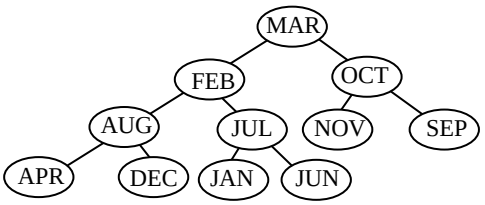

7−21 从第 7−20 题所建立的 AVL 树中删除关键码 MAY，为保持 AVL 树的特性，应如何进行删除和调整？若接着删除关键码 FEB，又应如何删除与调整?

【解答】

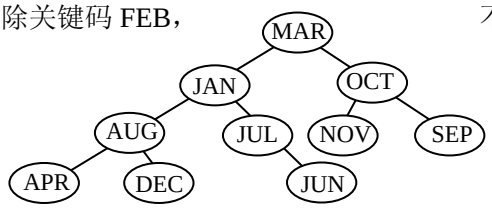
删除关键码 MAY，




调整




删除关键码 FEB，




不用调整

7−22 将关键码 1, 2, 3, …, $2^k-1$ 依次插入到一棵初始为空的 AVL 树中。试证明结果树是完全平衡的。

【解答】

所谓“完全平衡”是指所有叶结点处于树的同一层次上，并在该层是满的。此题可用数学归纳法证明。

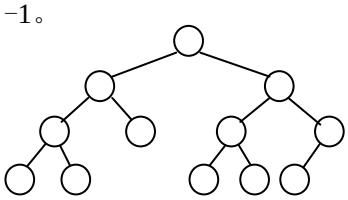
当 k = 1 时，$2^1-1 = 1$，AVL 树只有一个结点，它既是根又是叶并处在第 0 层，根据二叉树性质，应具有 $2^0 = 1$ 个结点。因此，满足完全平衡的要求。

设 k = n 时，插入关键码 1, 2, 3, …, $2^n-1$ 到 AVL 树中，恰好每一层(层次号码 i = 0, 1, …, n−1)有 $2^i$ 个结点，根据二叉树性质，每一层达到最多结点个数，满足完全平衡要求。则当 k = n+1 时，插入关键码为 1, 2, 3, …, $2^n-1$, $2^n$, …, $2^{n+1}-1$，总共增加了从 $2^n$ 到 $2^{n+1}-1$ 的 $2^{n+1}-1-2^n+1 = 2^n$ 个关键码，使得 AVL 树在新增的第 n 层具有 $2^n$ 个结点，达到该层最多结点个数，因此，满足完全平衡要求。

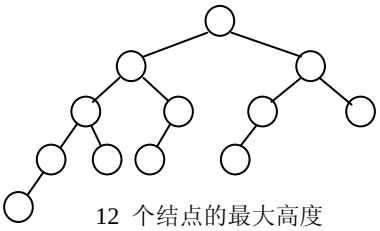
7−23 对于一个高度为 h 的 AVL 树，其最少结点数是多少？反之，对于一个有 n 个结点的 AVL 树，其最大高度是多少？最小高度是多少?

【解答】

设高度为 h（空树的高度为 −1）的 AVL 树的最少结点数为 $N_h$，则 $N_h = F_{h+3} - 1$。$F_h$ 是斐波那契数。又设 AVL 树有 n 个结点，则其最大高度不超过 $3 / 2 * \log_2(n + 1)$，最小高度为 $\lceil \log_2(n+1) \rceil - 1$。



12 个结点的最小高度

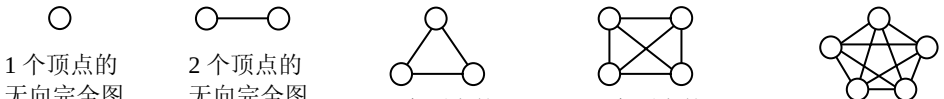


12 个结点的最大高度

# 第八章 图

8-1 画出 1 个顶点、2 个顶点、3 个顶点、4 个顶点和 5 个顶点的无向完全图。试证明在 n 个顶点的无向完全图中，边的条数为 n(n−1)/2。

【解答】

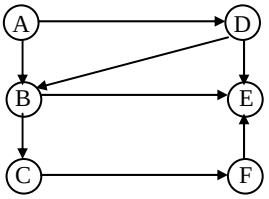


1 个顶点的无向完全图　2 个顶点的无向完全图　3 个顶点的无向完全图　4 个顶点的无向完全图　5 个顶点的无向完全图

【证明】

在有 n 个顶点的无向完全图中，每一个顶点都有一条边与其它某一顶点相连，所以每一个顶点有 n−1 条边与其他 n−1 个顶点相连，总计 n 个顶点有 n(n−1)条边。但在无向图中，顶点 i 到顶点 j 与顶点 j 到顶点 i 是同一条边，所以总共有 n(n−1)/2 条边。

8-2 右边的有向图是强连通的吗？请列出所有的简单路径。




【解答】

判断一个有向图是否强连通，要看从任一顶点出发是否能够回到该顶点。右面的有向图做不到这一点，它不是强连通的有向图。各个顶点自成强连通分量。

所谓简单路径是指该路径上没有重复的顶点。

从顶点 A 出发，到其他的各个顶点的简单路径有 A→B，A→D→B，A→B→C，A→D→B→C，A→D，A→B→E，A→D→E，A→D→B→E，A→B→C→F→E，A→D→B→C→F→E，A→B→C→F，A→D→B→C→F。

从顶点 B 出发，到其他各个顶点的简单路径有 B→C，B→C→F，B→E，B→C→F→E。

从顶点 C 出发，到其他各个顶点的简单路径有 C→F，C→F→E。

从顶点 D 出发，到其他各个顶点的简单路径有 D→B，D→B→C，D→B→C→F，D→E，D→B→E，D→B→C→F→E。

从顶点 E 出发，到其他各个顶点的简单路径无。

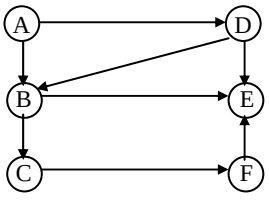
从顶点 F 出发，到其他各个顶点的简单路径有 F→E。

8-3 给出右图的邻接矩阵、邻接表和邻接多重表表示。

【解答】

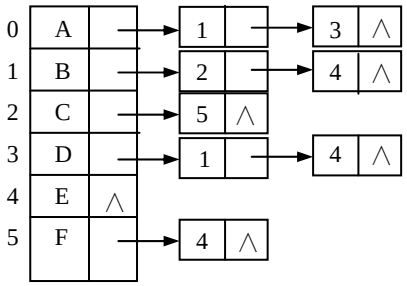
(1) 邻接矩阵

$$Edge = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

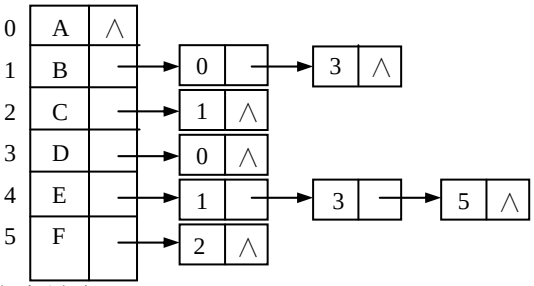


(2) 邻接表

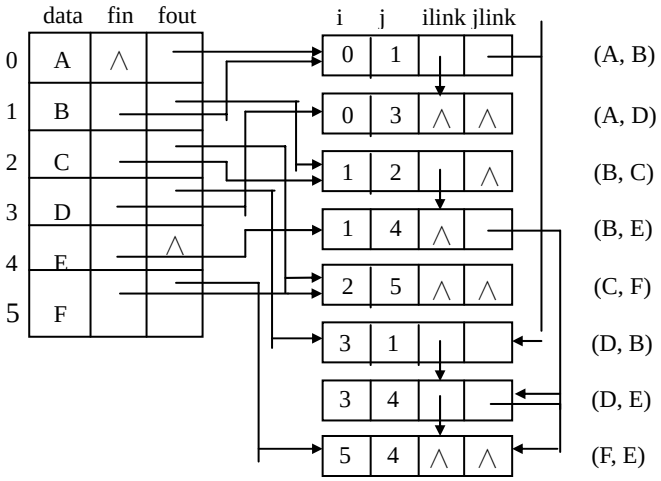
（出边表）




（入边表）




(3) 邻接多重表（十字链表）




8-4 用邻接矩阵表示图时，若图中有 1000 个顶点，1000 条边，则形成的邻接矩阵有多少矩阵元素？有多少非零元素？是否稀疏矩阵？

【解答】

一个图中有 1000 个顶点，其邻接矩阵中的矩阵元素有 $1000^2$ = 1000000 个。它有 1000 个非零元素（对于有向图）或 2000 个非零元素（对于无向图），因此是稀疏矩阵。

8-5 用邻接矩阵表示图时，矩阵元素的个数与顶点个数是否相关？与边的条数是否相关？

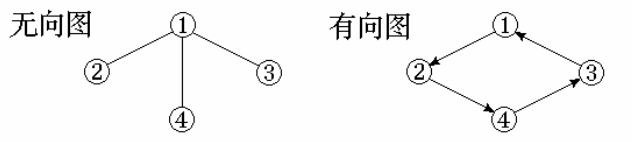
【解答】

用邻接矩阵表示图，矩阵元素的个数是顶点个数的平方，与边的条数无关。矩阵中非零元素的个数与边的条数有关。

8-6 有 n 个顶点的无向连通图至少有多少条边？有 n 个顶点的有向强连通图至少有多少条边？试举例说明。

【解答】

n 个顶点的无向连通图至少有 n-1 条边，n 个顶点的有向强连通图至少有 n 条边。例如：




特例情况是当 n = 1 时，此时至少有 0 条边。

8-7 对于有 n 个顶点的无向图，采用邻接矩阵表示，如何判断以下问题： 图中有多少条边？任意两个顶点 i 和 j 之间是否有边相连？任意一个顶点的度是多少？

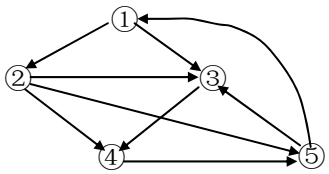
【解答】

用邻接矩阵表示无向图时，因为是对称矩阵，对矩阵的上三角部分或下三角部分检测一遍，统计其中的非零元素个数，就是图中的边数。如果邻接矩阵中 A[i][j] 不为零，说明顶点 i 与顶点 j 之间有边相连。此外统计矩阵第 i 行或第 i 列的非零元素个数，就可得到顶点 i 的度数。

8-8 对于如右图所示的有向图，试写出：

(1) 从顶点①出发进行深度优先搜索所得到的深度优先生成树；

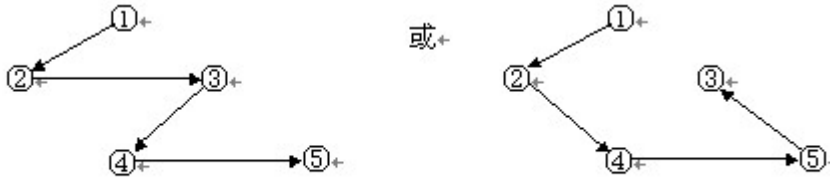
(2) 从顶点②出发进行广度优先搜索所得到的广度优先生成树；



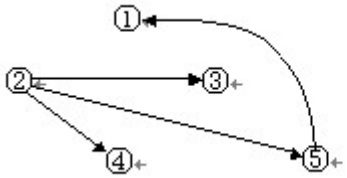

【解答】

(1) 以顶点①为根的深度优先生成树（不唯一）：② ③ ④ ⑤ ⑥

(1) 以顶点①为根的深度优先生成树（不唯一）：② ③ ④ ⑤ ⑥




(2) 以顶点②为根的广度优先生成树：




8-9 试扩充深度优先搜索算法，在遍历图的过程中建立生成森林的左子女-右兄弟链表。算法的首部为 **void** Graph**::**DFS ( **const int** v, **int** visited [ ], TreeNode<**int**> * t ) 其中，指针 t 指向生成森林上具有图顶点 v 信息的根结点。（提示：在继续按深度方向从根 v 的某一未访问过的邻接顶点 w 向下遍历之前，建立子女结点。但需要判断是作为根的第一个子女还是作为其子女的右兄弟链入生成树。）

【解答】

为建立生成森林，需要先给出建立生成树的算法，然后再在遍历图的过程中，通过一次次地调用这个算法，以建立生成森林。

```
te mplate<Type> void Graph<Type> :: DFS_Tree ( const int v,  int visited [ ],  TreeNode<Type> *t ) {
//从图的顶点 v 出发，深度优先遍历图，建立以 t (已在上层算法中建立)为根的生成树。
```

```
    Visited[v] = 1;   int first = 1;   TreeNode<Type> * p, * q;
    int w = GetFirstNeighbor ( v );                    //取第一个邻接顶点
    while ( w != −1 ) {                                //若邻接顶点存在
       if ( vosited[w] == 0 ) {                            //且该邻接结点未访问过
          p = new TreeNode<Type> ( GetValue (w) );     //建立新的生成树结点
          if ( first == 1 )                            //若根*t 还未链入任一子女
             { t−>setFirstChild ( p );   first = 0; }  //新结点*p 成为根*t 的第一个子女
          else q−>setNextSibling ( p );                //否则新结点*p 成为*q 的下一个兄弟
          q = p;                                       //指针 q 总指示兄弟链最后一个结点
          DFS_Tree ( w, visited, q );                  //从*q 向下建立子树
       }
       w = GetNextNeighbor ( v, w );                 //取顶点 v 排在邻接顶点 w 的下一个邻接顶点
    }
}
```

下一个算法用于建立以左子女-右兄弟链表为存储表示的生成森林。

```
template<Type> void Graph<Type> :: DFS_Forest ( Tree<Type> & T ) {
//从图的顶点 v 出发, 深度优先遍历图, 建立以左子女-右兄弟链表表示的生成森林 T。
    T.root = NULL;   int n = NumberOfVertices ( );          //顶点个数
    TreeNode<Type> * p, * q;
    int * visited = new int [ n ];                          //建立访问标记数组
    for ( int v = 0; v < n; v++ ) visited[v] = 0;
    for ( v = 0; v < n; v++ )                               //逐个顶点检测
       if ( visited[v] == 0 ) {                             //若尚未访问过
          p = new TreeNode<Type> ( GetValue ( v ) );        //建立新结点*p
          if ( T.root == NULL ) T.root = p;                 //原来是空的生成森林, 新结点成为根
          else q−> setNextSibling ( p );                    //否则新结点*p 成为*q 的下一个兄弟
          q = p;
          DFS_Tree ( v, visited, p );                       //建立以*p 为根的生成树
       }
}
```

8−10 用邻接表表示图时，顶点个数设为 n，边的条数设为 e，在邻接表上执行有关图的遍历操作时，时间代价是 O(n*e)？还是 O(n+e)？或者是 O(max(n,e))？

【解答】

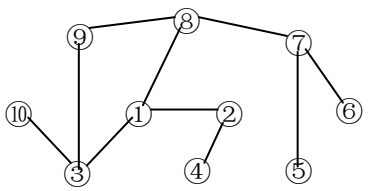
在邻接表上执行图的遍历操作时，需要对邻接表中所有的边链表中的结点访问一次，还需要对所有的顶点访问一次，所以时间代价是 O(n+e)。

8−11 右图是一个连通图，请画出

(1) 以顶点①为根的深度优先生成树；

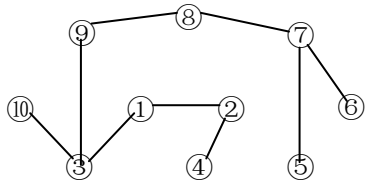
(2) 如果有关节点，请找出所有的关节点。

(3) 如果想把该连通图变成重连通图，至少在图中加几条边？如何加？



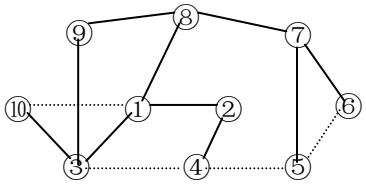

【解答】

(1) 以顶点①为根的深度优先生成树：




(2) 关节点为 ①, ②, ③, ⑦, ⑧

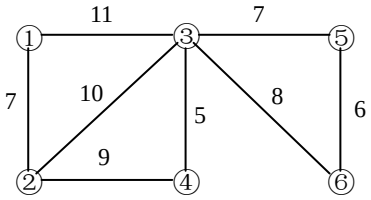
(3) 至少加四条边 (1, 10), (3, 4), (4, 5), (5, 6)。从③的子孙结点⑩到③的祖先结点①引一条边，从②的子孙结点④到根①的另一分支③引一条边，并将⑦的子孙结点⑤、⑥与结点④连结起来，可使其变为重连通图。




8–12 试证明在一个有 n 个顶点的完全图中，生成树的数目至少有 $2^{n-1}-1$。

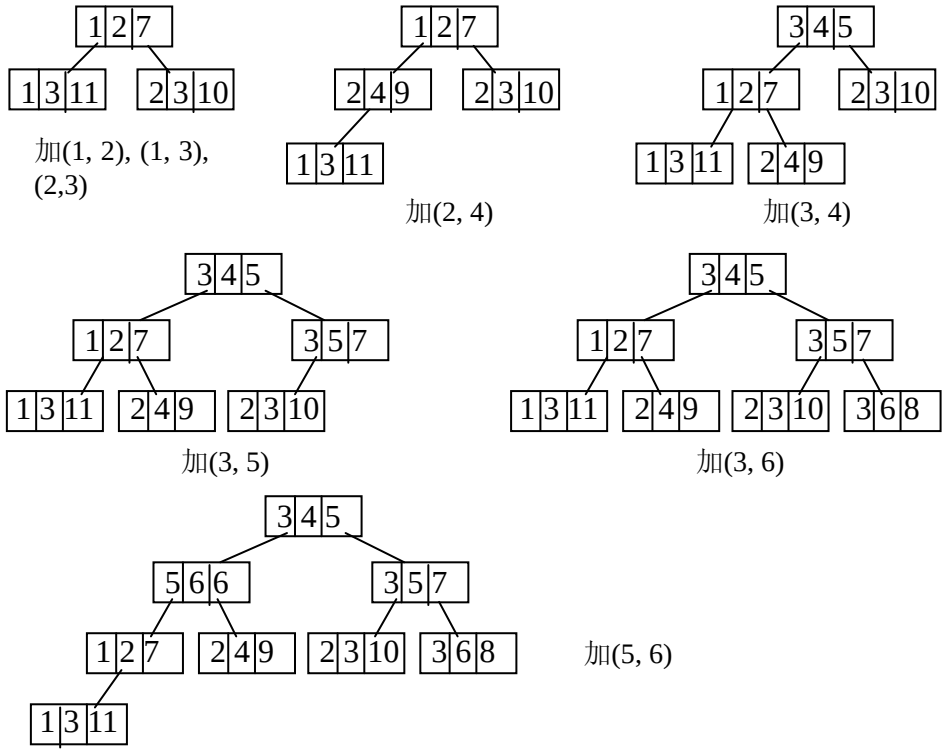
【证明】略

8–13 编写一个完整的程序，首先定义堆和并查集的结构类型和相关操作，再定义 Kruskal 求连通网络的最小生成树算法的实现。并以右图为例，写出求解过程中堆、并查集和最小生成树的变化。

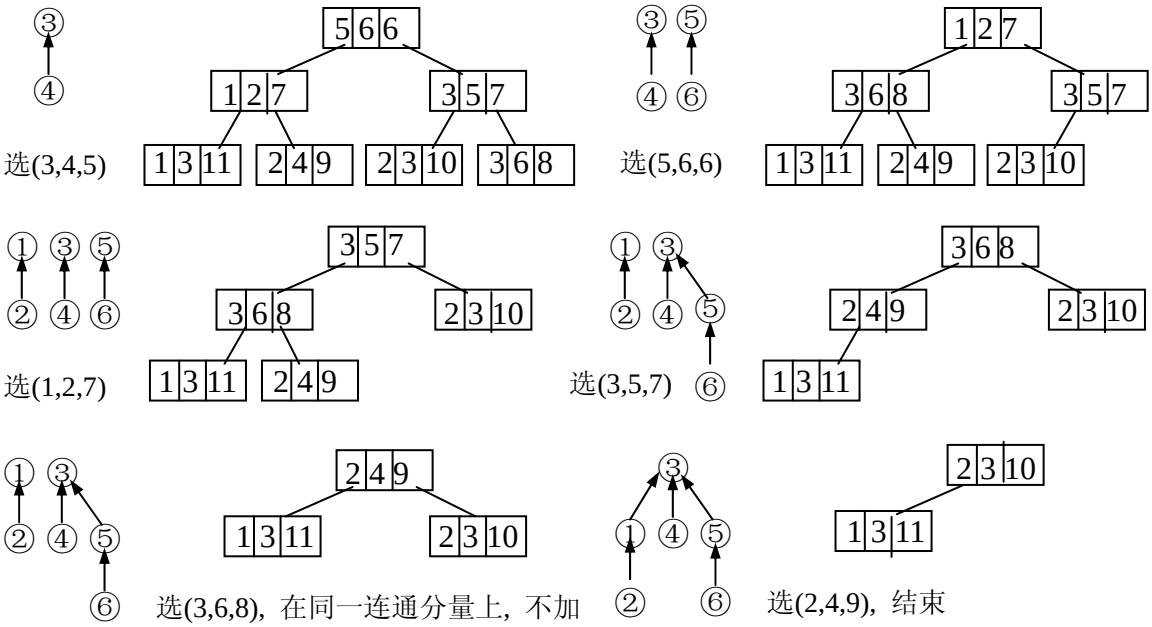



【解答】

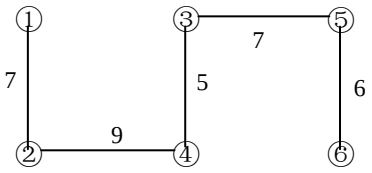
求解过程的第一步是对所有的边，按其权值大小建堆：




求解过程中并查集与堆的变化：




最后得到的生成树如下




| 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| | 3 | 1 | −6 | 3 | 3 | 5 |

并查集的存储表示

完整的程序如下:

```
#include <iostream.h>
template <class Type> class MinHeap {
public:
    enum { MaxHeapSize = 50 };
    MinHeap ( int Maxsize = MaxHeapSize );
    MinHeap ( Type Array[ ], int n );
    void Insert ( const Type &ele );
    void RemoveMin ( Type &Min );
    void Output ();
private:
    void FilterDown ( int start, int end );
    void FilterUp ( int end );
    Type    *pHeap;
    int HMaxSize;
    int CurrentSize;
};

class UFSets {
public:
    enum { MaxUnionSize = 50 };
    UFSets ( int MaxSize = MaxUnionSize );
```

```
    ~UFSets () { delete [ ] m_pParent; }
    void Union ( int Root1, int Root2 );
    int Find ( int x );
private:
    int m_iSize;
    int *m_pParent;
};

class Graph {
public:
    enum { MaxVerticesNum = 50 };
    Graph( int Vertices = 0) { CurrentVertices = Vertices; InitGraph(); }
    void InitGraph ();
    void Kruskal ();
    int GetVerticesNum () { return CurrentVertices; }
private:
    int Edge[MaxVerticesNum][MaxVerticesNum];
    int CurrentVertices;
};

class GraphEdge {
public:
    int head, tail;
    int cost;
    int operator <= ( GraphEdge &ed );
};

GraphEdge :: operator <= ( GraphEdge &ed ) {
    return this->cost <= ed.cost;
}

UFSets :: UFSets ( int MaxSize ) {
    m_iSize = MaxSize;
    m_pParent = new int[m_iSize];
    for ( int i = 0; i < m_iSize; i++ ) m_pParent[i] = -1;
}

void UFSets :: Union ( int Root1, int Root2 ) {
    m_pParent[Root2] = Root1;
}

int UFSets :: Find ( int x ) {
    while ( m_pParent[x] >= 0 ) x = m_pParent[x];
    return x;
```

```
}

template <class Type> MinHeap<Type> :: MinHeap ( int Maxsize ) {
    HMaxSize = Maxsize;
    pHeap = new Type[HMaxSize];
    CurrentSize = -1;
}

template <class Type> MinHeap<Type> :: MinHeap ( Type Array[], int n ) {
    HMaxSize = ( n < MaxHeapSize ) ? MaxHeapSize : n;
    pHeap = new Type[HMaxSize];
    for ( int i = 0; i < n; i++ ) pHeap[i] = Array[i];
    CurrentSize = n-1;
    int iPos = ( CurrentSize - 1 ) / 2;
    while ( iPos >= 0 ) {
        FilterDown ( iPos, CurrentSize );
        iPos--;
    }
}

template <class Type> void MinHeap<Type> :: FilterDown ( int start, int end ) {
    int i = start, j = 2 * start + 1;
    Type Temp = pHeap[i];
    while ( j <= end ) {
        if ( j < end && pHeap[j+1] <= pHeap[j] ) j++;
        if ( Temp <= pHeap[j] ) break;
        pHeap[i] = pHeap[j];
        i = j; j = 2 * j + 1;
    }
    pHeap[i] = Temp;
}

template <class Type> void MinHeap<Type> :: FilterUp ( int end ) {
    int i = end, j = ( end - 1 ) / 2;
    Type Temp = pHeap[i];
    while ( i > 0 ) {
        if ( pHeap[j] <= Temp ) break;
        pHeap[i] = pHeap[j];
        i = j; j = ( j - 1 ) / 2;
    }
    pHeap[i] = Temp;
}

template <class Type> void MinHeap<Type> :: Insert ( const Type &ele ) {
```

```
        CurrentSize++;
        if ( CurrentSize == HMaxSize ) return;
        pHeap[CurrentSize] = ele;
        FilterUp ( CurrentSize );
    }

    template <class Type> void MinHeap<Type> :: RemoveMin ( Type &Min ) {
        if ( CurrentSize < 0 )      return;
        Min = pHeap[0];
        pHeap[0] = pHeap[CurrentSize--];
        FilterDown ( 0, CurrentSize );
    }

    template <class Type> void MinHeap<Type> :: Output ( ) {
        for ( int i = 0; i <= CurrentSize; i++ ) cout << pHeap[i] << " ";
        cout << endl;
    }

    void Graph :: InitGraph( ) {
        Edge[0][0] = −1;  Edge[0][1] = 28;  Edge[0][2] = −1;  Edge[0][3] = −1;  Edge[0][4] = −1;
Edge[0][5] = 10;
        Edge[0][6] = −1;
        Edge[1][1] = −1;  Edge[1][2] = 16;  Edge[1][3] = −1;  Edge[1][4] = −1;  Edge[1][5] = −1;
Edge[1][6] = 14;
        Edge[2][2] = −1;  Edge[2][3] = 12;  Edge[2][4] = −1;  Edge[2][5] = −1;  Edge[2][6] = −1;
        Edge[3][3] = −1;  Edge[3][4] = 22;  Edge[3][5] = −1;  Edge[3][6] = 18;
        Edge[4][4] = −1;  Edge[4][5] = 25;  Edge[4][6] = 24;
        Edge[5][5] = −1;  Edge[5][6] = −1;
        Edge[6][6] = −1;
        for ( int i = 1; i < 6; i++ )
            for ( int j = 0; j < i; j ++ ) Edge[i][j] = Edge[j][i];
    }

    void Graph :: Kruskal( ) {
        GraphEdge e;
        int VerticesNum = GetVerticesNum ( );
        int i, j, count;
        MinHeap<GraphEdge> heap ( VerticesNum *VerticesNum );
        UFSets set ( VerticesNum );
        for ( i = 0; i < VerticesNum; i++ )
            for ( j = i + 1; j < VerticesNum; j++ )
                if ( Edge[i][j] > 0 )  {
                    e.head = i; e.tail = j; e.cost = Edge[i][j];
                    heap.Insert ( e );
```

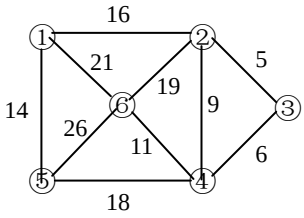
```
            }
        count = 1;
        while ( count < VerticesNum ) {
            heap.RemoveMin ( e );
            i = set.Find ( e.head );
            j = set.Find ( e.tail );
            if ( i != j ) {
                set.Union ( i, j );
                count++;
                cout << "( " << e.head << ", " << e.tail << ", " <<    e.cost << " )" << endl;
            }
        }
    }
```

8-14 利用 Dijkstra 算法的思想，设计一个求最小生成树的算法。

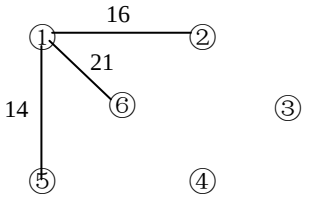
【解答】

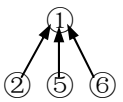
计算连通网络的最小生成树的 Dijkstra 算法可描述如下：将连通网络中所有的边以方便的次序逐步加入到初始为空的生成树的边集合 T 中。每次选择并加入一条边时，需要判断它是否会与先前加入 T 的边构成回路。如果构成了回路，则从这个回路中将权值最大的边退选。

下面以邻接矩阵作为连通网络的存储表示，并以并查集作为判断是否出现回路的工具，分析算法的执行过程。

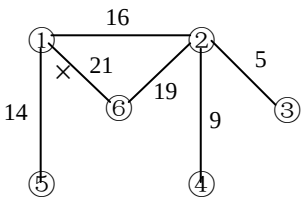


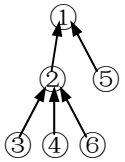

$$\text{Edge} = \begin{array}{c} \begin{matrix} ① & ② & ③ & ④ & ⑤ & ⑥ \end{matrix} \\ \begin{pmatrix} 0 & 16 & \infty & \infty & 14 & 21 \\ - & 0 & 5 & 9 & \infty & 19 \\ - & - & 0 & 6 & \infty & \infty \\ - & - & - & 0 & 18 & 11 \\ - & - & - & - & 0 & 26 \\ - & - & - & - & - & 0 \end{pmatrix} \end{array} \begin{matrix} ① \\ ② \\ ③ \\ ④ \\ ⑤ \\ ⑥ \end{matrix}$$

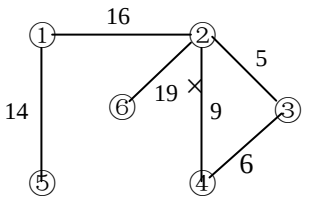


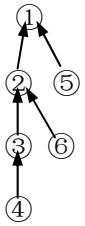



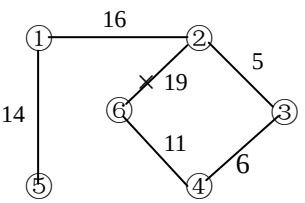
并查集，表明 4 个结点在同一连通分量上

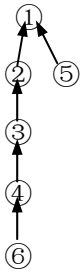


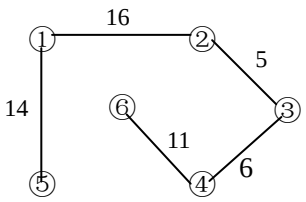




最终得到的最小生成树为




实现算法的过程为

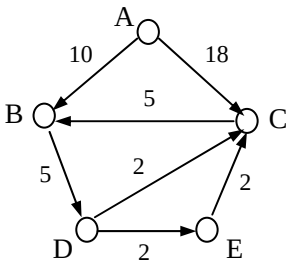
```
const int MaxNum = 10000;
void Graph :: Dijkstra ( ) {
    GraphEdge e;
    int VerticesNum = GetVerticesNum ( );
    int i, j, p, q, k;
    int disJoint[VerticesNum];                          //并查集
    for ( i = 0; i < VerticesNum; i++ ) disJoint[i] = -1;     //并查集初始化
    for ( i = 0; i < VerticesNum-1; i++ )                     //检查所有的边
        for ( j = i + 1; j < VerticesNum; j++ )
            if ( Edge[i][j] < MaxNum )      {           //边存在
                p = i;   q = j;                         //判结点 i 与 j 是否在同一连通分量上
                while ( disJoint[p] >= 0 ) p = disJoint[p];
                while ( disJoint[q] >= 0 ) p = disJoint[q];
                if ( p != q ) disJoint[j] = i;          // i 与 j 不在同一连通分量上, 连通之
            } else {                                    // i 与 j 在同一连通分量上
                p = i;                                  //寻找离结点 i 与 j 最近的祖先结点
                while ( disJoint[p] >= 0 ) {            //每变动一个 p, 就对 q 到根的路径检测一遍
                    q = j;
                    while ( disJoint[q] >= 0 && disJoint[q] == disJoint[p] )
                        q = disJoint[q];
                    if ( disJoint[q] == disJoint[p] ) break;
                    else p = disJoint[p];
                }
                k = disJoint[p];                                //结点 k 是 i 和 j 的最近的共同祖先
                p = i;   q = disJoint[p];   max = -MaxNum;   //从 i 到 k 找权值最大的边(s1, s2)
                while ( q <= k ) {
                    if ( Edge[q][p] > max ) { max = Edge[q][p];   s1 = p;   s2 = q; }
                    p =q;   q = disJoint[p];
                }
                p = j;   q = disJoint[p];   max = -MaxNum;   //从 j 到 k 找权值最大的边(t1, t2)
                while ( q <= k ) {
                    if ( Edge[q][p] > max ) { max = Edge[q][p];   t1 = p;   t2 = q; }
                    p =q;   q = disJoint[p];
                }
                max = Edge[i][j];   k1 = i;   k2 = j;
                if ( max < Edge[s1][s2] ) { max = Edge[s1][s2];   k1 = s1;   k2 = s2; }
                if ( max < Edge[t1][t2] ) { max = Edge[t1][t2];   k1 = t1;   k2 = t2; }
```

```
            if ( max != Edge[i][j] ) {                    //当 Edge[i][j] == max 时边不改
               if ( disJoint[k1] == k2 ) disJoint[k1] = -1;
               else disJoint[k2] = -1;                    //删除权值最大的边
                disJoint[j] = i;                          //加入新的边
               Edge[j][i] = - Edge[j][i];
            }
         }
   }
```

8-15 以右图为例，按 Dijkstra 算法计算得到的从顶点①(A)到其它各个顶点的最短路径和最短路径长度。

【解答】

| 源点 | 终点 | 最短路径 | | | | 最短路径长度 | | | |
|---|---|---|---|---|---|---|---|---|---|
| A | B | (A,B) | (A,B) | (A,B) | (A,B) | 10 | 10 | 10 | 10 |
| | C | (A,C) | (A,C) | (A,C) | (A,C) | 18 | 18 | 18 | 18 |
| | D | — | (A,B,D) | (A,B,D) | (A,B,D) | ∞ | 15 | 15 | 15 |
| | E | — | — | (A,B,D,E) | (A,B,D,E) | ∞ | ∞ | 17 | 17 |




8-16 在以下假设下，重写 Dijkstra 算法：

(1) 用邻接表表示带权有向图 G，其中每个边结点有 3 个域：邻接顶点 vertex，边上的权值 length 和边链表的链接指针 link。

(2) 用集合 T = V(G) - S 代替 S (已找到最短路径的顶点集合)，利用链表来表示集合 T。

试比较新算法与原来的算法，计算时间是快了还是慢了，给出定量的比较。

【解答】

(1) 用邻接表表示的带权有向图的类定义：

```
const int DefaultSize = 10;                   //缺省顶点个数
class Graph;                                  //图的前视类定义
struct Edge {                                 //边的定义
friend class Graph;
   int vertex;                                //边的另一顶点位置
   float length;                              //边上的权值
   Edge *link;                                //下一条边链指针
   Edge ( ) { }                               //构造函数
   Edge ( int num, float wh ) : vertex (num), length (wh), link (NULL) { }      //构造函数
   int operator < ( const Edge & E ) const { return length != E.length; }      //判边上权值小否
}

struct Vertex {                               //顶点的定义
friend class Graph;
   char data;                                 //顶点的名字
   Edge *adj;                                 //边链表的头指针
}

class Graph {                                 //图的类定义
```

```
private:
   Vertex *NodeTable;                    //顶点表 (各边链表的头结点)
   int NumVertices;                      //当前顶点个数
   int NumEdges;                         //当前边数
   int GetVertexPos ( const Type vertex );   //给出顶点 vertex 在图中的位置
public:
   Graph ( int sz );                     //构造函数
   ~Graph ( );                           //析构函数
   int NumberOfVertices ( ) { return NumVertices; }          //返回图的顶点数
   int NumberOfEdges ( ) { return NumEdges; }          //返回图的边数
   char GetValue ( int i )               //取位置为 i 的顶点中的值
      { return i >= 0 && i < NumVertices ? NodeTable[i].data : ' '; }
   float GetWeight ( int v1, int v2 );   //返回边(v1, v2)上的权值
   int GetFirstNeighbor ( int v );       //取顶点 v 的第一个邻接顶点
   int GetNextNeighbor ( int v, int w );    //取顶点 v 的邻接顶点 w 的下一个邻接顶点
}
```

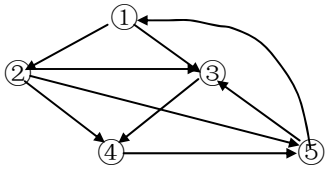
(2) 用集合 T = V(G) − S 代替 S (已找到最短路径的顶点集合)，利用链表来表示集合 T。

8-17 试证明：对于一个无向图 G = (V, E)，若 G 中各顶点的度均大于或等于 2，则 G 中必有回路。

【解答】

反证法：对于一个无向图 G=（V，E），若 G 中各顶点的度均大于或等于 2，则 G 中没有回路。此时从某一个顶点出发，应能按拓扑有序的顺序遍历图中所有顶点。但当遍历到该顶点的另一邻接顶点时，又可能回到该顶点，没有回路的假设不成立。

8-18 设有一个有向图存储在邻接表中。试设计一个算法，按深度优先搜索策略对其进行拓扑排序。并以右图为例检验你的算法的正确性。




【解答】

(1) 利用题 8-16 定义的邻接表结构。

增加两个辅助数组和一个工作变量：

- 记录各顶点入度 **int** indegree[NumVertices]。
- 记录各顶点访问顺序 **int** visited[NumVertices]，初始时让 visited[i] = 0, i = 1, 2, …, NumVertices。
- 访问计数 **int** count，初始时为 0。

(2) 拓扑排序算法

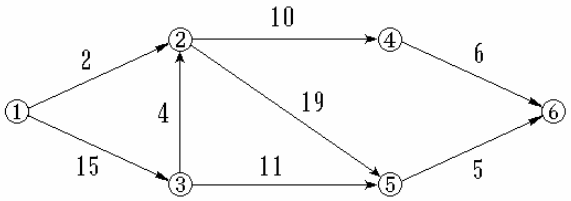
```
void Graph :: dfs ( int visited[ ], int indegree[ ], int v, int & count ) {
   count++;   visited[v] = count;
   cout << NodeTable[v].data << endl;
   Edge *p = NodeTable[v].adj;
   while ( p != NULL ) {
      int w = p->vertex;
      indegree[w]--;
      if ( visited[w] == 0 && indegree[w] == 0 ) dfs ( visited, indegree, w, count );
      p = p->link;
```

```
        }
    }
```

主程序

```
    int i, j;   Edge *p;   float w;
    cin >> NumVertices;
    int * visited = new int[NumVertices+1];
    int * indegree = new int[NumVertices+1];
    for ( i = 1; i <= NumVertices; i++ ) {
        NodeTable[i].adj = NULL;   cin >> NodeTable[i].data;   cout << endl;
        visited[i] = 0;   indegree[i] = 0;
    }
    int count = 0;
    cin >> i >> j >> w;   cout << endl;
    while ( i != 0 && j != 0 ) {
        p = new Edge ( j, w );
        if ( p == NULL ) { cout << "存储分配失败！" << endl;   exit(1); }
        indegree[j]++;
        p->link = NodeTable[i].adj;   NodeTable[i].adj = p;
        NumEdges++;
        cin >> i >> j >> w;   cout << endl;
    }
    for ( i = 1; i <= NumVertices; i++ )
      if ( visited[i] == 0 && indegree[i] == 0 ) dfs ( visited, indegree, i, count );
    if ( count < NumVertices ) cout << "排序失败!" << endl;
    else cout << "排序成功!" << endl;
    delete [ ] visited;   delete [ ] indegree;
```

8-19 试对右图所示的 AOE 网络，解答下列问题。




(1) 这个工程最早可能在什么时间结束。

(2) 求每个事件的最早开始时间 Ve[i]和最迟开始时间 Vl[I]。

(3) 求每个活动的最早开始时间 e( )和最迟开始时间 l( )。

(4) 确定哪些活动是关键活动。画出由所有关键活动构成的图，指出哪些活动加速可使整个工程提前完成。

【解答】

按拓扑有序的顺序计算各个顶点的最早可能开始时间 Ve 和最迟允许开始时间 Vl。然后再计算各个活动的最早可能开始时间 e 和最迟允许开始时间 l，根据 l - e = 0? 来确定关键活动，从而确定关键路径。

| | 1□ | 2□ | 3□ | 4□ | 5□ | 6□ |
|---|---|---|---|---|---|---|
| Ve | 0 | 19 | 15 | 29 | 38 | 43 |
| Vl | 0 | 19 | 15 | 37 | 38 | 43 |

| | <1, 2> | <1, 3> | <3, 2> | <2, 4> | <2, 5> | <3, 5> | <4, 6> | <5, 6> |
|---|---|---|---|---|---|---|---|---|
| e | 0 | 0 | 15 | 19 | 19 | 15 | 29 | 38 |
| l | 17 | 0 | 15 | 27 | 19 | 27 | 37 | 38 |
| l−e | 17 | 0 | 0 | 8 | 0 | 12 | 8 | 0 |

此工程最早完成时间为 43。关键路径为<1, 3><3, 2><2, 5><5, 6>

8−20 若 AOE 网络的每一项活动都是关键活动。令 G 是将该网络的边去掉方向和权后得到的无向图。

(1) 如果图中有一条边处于从开始顶点到完成顶点的每一条路径上，则仅加速该边表示的活动就能减少整个工程的工期。这样的边称为桥(bridge)。证明若从连通图中删去桥，将把图分割成两个连通分量。

(2) 编写一个时间复杂度为 O(n+e)的使用邻接表表示的算法，判断连通图 G 中是否有桥，若有。输出这样的桥。

# 第九章 排序

静态数据表类定义

```
#include <iostream.h>
const int    DefaultSize = 100;
template <class Type> class dataList                          //数据表的前视声明

template <class Type> class Element {                         //数据表元素类的定义
friend class dataList <Type>;
private:
    Type key;                                                 //排序码
    field otherdata;                                          //其它数据成员
public:
    Type getKey ( ) { return key; }                           //取当前结点的排序码
    void setKey ( const Type x ) { key = x; }                 //将当前结点的排序码修改为 x
    Element<Type>& operator = ( Element<Type>& x )            //结点 x 的值赋给 this
      { key = x->key;    otherdata = x->otherdata; }
    int operator == ( Type& x ) { return key == x->key; }     //判 this 与 x 相等
    int operator <= ( Type& x ) { return key <= x->key; }     //判 this 小于或等于 x
    int operator > ( Type& x ) { return key > x->key; }       //判 this 大于 x
    int operator < ( Type& x ) { return key > x->key; }       //判 this 小于 x
}

template <class Type> class dataList {
//用顺序表来存储待排序的元素，这些元素的类型是 Type
private:
    Element <Type> * Vector;                                  //存储待排序元素的向量
    int MaxSize, CurrentSize;                                 //最大元素个数与当前元素个数
    int Partition ( const int low, const int high )           //用于快速排序的一次划分算法
public:
    datalist ( int MaxSz = DefaultSize ) : MaxSize ( Maxsz ), CurrentSize (0)
      { Vector = new Element <Type> [MaxSize]; }              //构造函数
    int length ( ) { return CurrentSize; }
    Element<Type>& operator [ ] ( int i ) { return Vector[i]; }
    void swap ( Element <Type>& x, Element <Type>& y )        //交换 x, y
      { Element <Type> temp = x;    x = y;    y = temp; }
    void Sort ( );                                            //排序
}
```

静态链表类定义

```
template <class Type> class staticlinkList;          //静态链表类的前视声明

template <class Type> class Element {                //静态链表元素类的定义
```

```
friend class staticlinkList<Type>;
private:
   Type key;                                  //排序码，其它信息略
   int link;                                  //结点的链接指针
public:
   Type getKey ( ) { return key; }            //取当前结点的排序码
   void setKey ( const Type x ) { key = x; }  //将当前结点的排序码修改为 x
   int getLink ( ) { return link; }           //取当前结点的链接指针
   void setLink ( const int ptr ) { link = ptr; }  //将当前结点的链接指针置为 ptr
}

template <class Type> class staticlinkList {       //静态链表的类定义
private:
   Element <Type> *Vector;                    //存储待排序元素的向量
   int MaxSize, CurrentSize;                  //向量中最大元素个数和当前元素个数
public:
   dstaticlinkList ( int Maxsz = DefaultSize ) : MaxSize ( Maxsz ), CurrentSize (0)
      { Vector = new Element <Type> [Maxsz]; }
}
```

9-1 什么是内排序？什么是外排序？什么排序方法是稳定的？什么排序方法是不稳定的？

【解答】内排序是排序过程中参与排序的数据全部在内存中所做的排序，排序过程中无需进行内外存数据传送，决定排序方法时间性能的主要是数据排序码的比较次数和数据对象的移动次数。外排序是在排序的过程中参与排序的数据太多，在内存中容纳不下，因此在排序过程中需要不断进行内外存的信息传送的排序。决定外排序时间性能的主要是读写磁盘次数和在内存中总的记录对象的归并次数。

不稳定的排序方法主要有希尔排序、直接选择排序、堆排序、快速排序。不稳定的排序方法往往是按一定的间隔移动或交换记录对象的位置，从而可能导致具有相等排序码的不同对象的前后相对位置在排序前后颠倒过来。其他排序方法中如果有数据交换，只是在相邻的数据对象间比较排序码，如果发生逆序（与最终排序的顺序相反的次序）才交换，因此具有相等排序码的不同对象的前后相对位置在排序前后不会颠倒，是稳定的排序方法。但如果把算法中判断逆序的比较“>（或<）”改写成“≥（或≤）”，也可能造成不稳定。

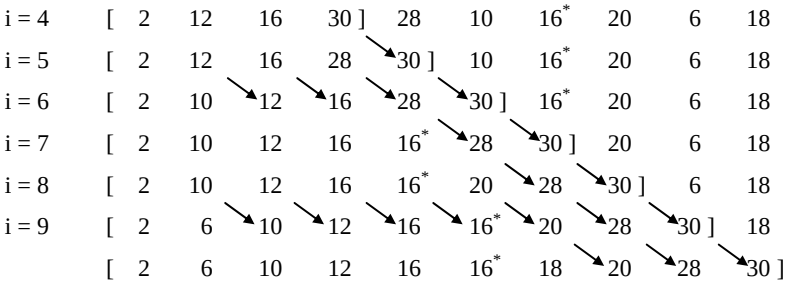
9-2 设待排序的排序码序列为{12, 2, 16, 30, 28, 10, 16*, 20, 6, 18}，试分别写出使用以下排序方法每趟排序后的结果。并说明做了多少次排序码比较。

(1) 直接插入排序　　(2) 希尔排序(增量为 5,2,1)　　(3) 起泡排序
(4) 快速排序　　(5) 直接选择排序　　(6) 锦标赛排序
(7) 堆排序　　(8) 二路归并排序　　(9) 基数排序

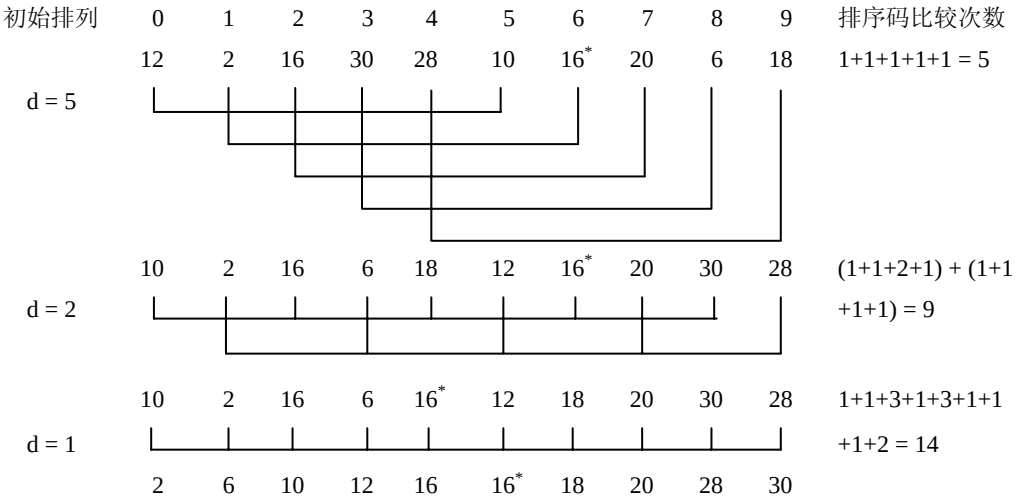
【解答】

(1) 直接插入排序

| 初始排列 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 排序码比较次数 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| i = 1 | [ 12 ] | 2 | 16 | 30 | 28 | 10 | 16* | 20 | 6 | 18 | 1 |
| i = 2 | [ 2 | 12 ] | 16 | 30 | 28 | 10 | 16* | 20 | 6 | 18 | 1 |
| i = 3 | [ 2 | 12 | 16 ] | 30 | 28 | 10 | 16* | 20 | 6 | 18 | 1 |

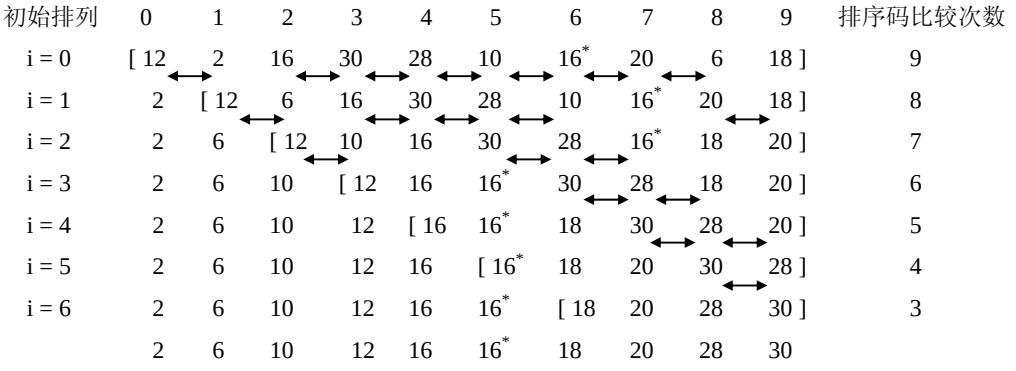



(2) 希尔排序(增量为 5,2,1)

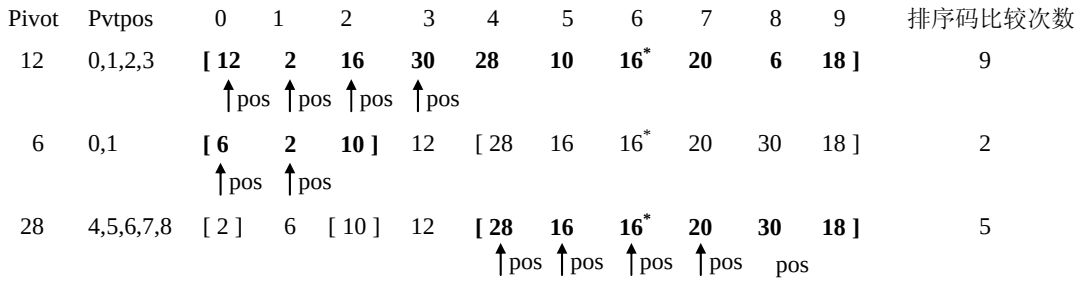



希尔(shell)本人采取的增量序列为 ⌊n/2⌋, ⌊⌊n/2⌋/2⌋, ⌊⌊n/2⌋/2⌋/2⌋, …,1。一般地，增量序列可采用⌊n α ⌋,⌊⌊n α ⌋α ⌋,⌊⌊n α ⌋α ⌋α ⌋, …, 1。大量实验表明，取 α =0.45454 的增量序列比取其他的增量序列的优越性更显著。计算 ⌊0.45454n⌋ 的一个简单方法是用整数算术计算 (5*n−1)/11。需要注意，当α< 1/2 时，增量序列可能不以 1 结束，需要加以判断和调整。

(3) 起泡排序




(4) 快速排序



18　4,5,6　2　6　10　12　**[ 18　16　16*　20 ]**　28　[ 30 ]　3

pos pos pos

16*　4　2　6　10　12　**[ 16　16 ]**　18　[ 20 ]　28　30　1

* pos

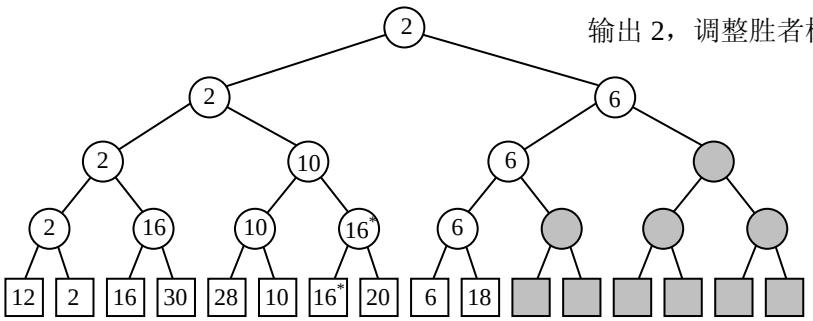
2　6　10　12　16*　[ 16 ]　18　20　28　30

左子序列递归深度为 1，右子序列递归深度为 3。

(5) 直接选择排序

| 初始排列 | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 排序码比较次数 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| i = 0 | [ 12 | 2 | 16 | 30 | 28 | 10 | 16* | 20 | 6 | 18 ] | 9 |
| i = 1 | 2 | [ 12 | 16 | 30 | 28 | 10 | 16* | 20 | 6 | 18 ] | 8 |
| i = 2 | 2 | 6 | [ 16 | 30 | 28 | 10 | 16* | 20 | 12 | 18 ] | 7 |
| i = 3 | 2 | 6 | 10 | [ 30 | 28 | 16 | 16* | 20 | 12 | 18 ] | 6 |
| i = 4 | 2 | 6 | 10 | 12 | [ 28 | 16 | 16* | 20 | 30 | 18 ] | 5 |
| i = 5 | 2 | 6 | 10 | 12 | 16 | [ 28 | 16* | 20 | 30 | 18 ] | 4 |
| i = 6 | 2 | 6 | 10 | 12 | 16 | 16* | [ 28 | 20 | 30 | 18 ] | 3 |
| i = 7 | 2 | 6 | 10 | 12 | 16 | 16* | 18 | [ 20 | 30 | 28 ] | 2 |
| i = 8 | 2 | 6 | 10 | 12 | 16 | 16* | 16 | 20 | [ 30 | 28 ] | 1 |
| | 2 | 6 | 10 | 12 | 16 | 16* | 16 | 20 | 28 | [ 30 ] | |

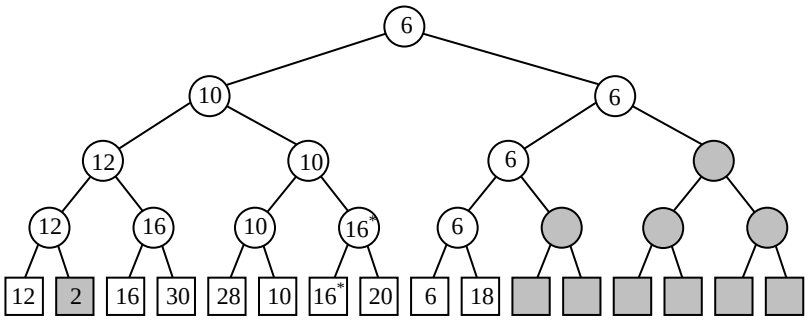
(6) 锦标赛排序

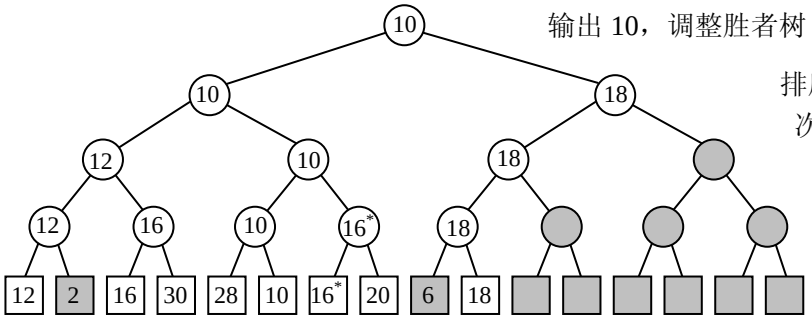



输出 2，调整胜者树

输出 6，调整胜者树

当参加排序的数据对象个数 n 不足 2 的某次幂时，将其补足到 2 的某次幂。本题的 n = 10，将叶结点个数补足到 $2^4 = 16$ 个。排序码比较次数 = 9。

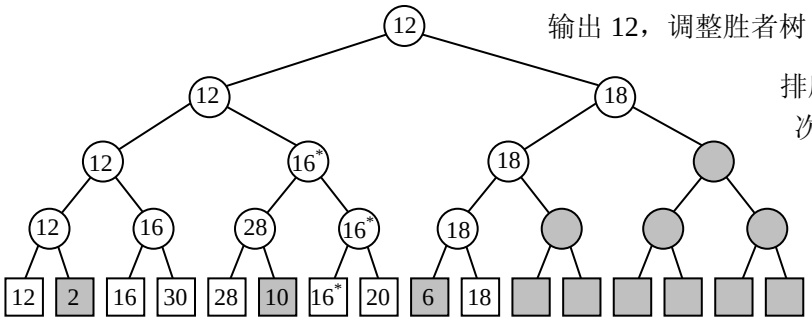



输出 10，调整胜者树

排序码比较次数 = 1。

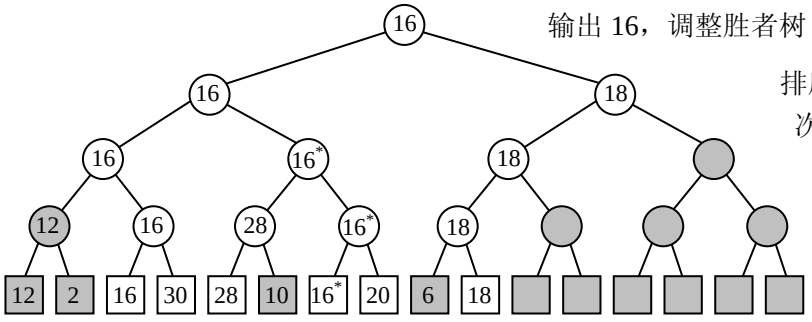
当某结点的比较对手的参选标志为“不再参选”，该结点自动升入双亲结点，此动作不计入排序码比较次数。




输出 12，调整胜者树

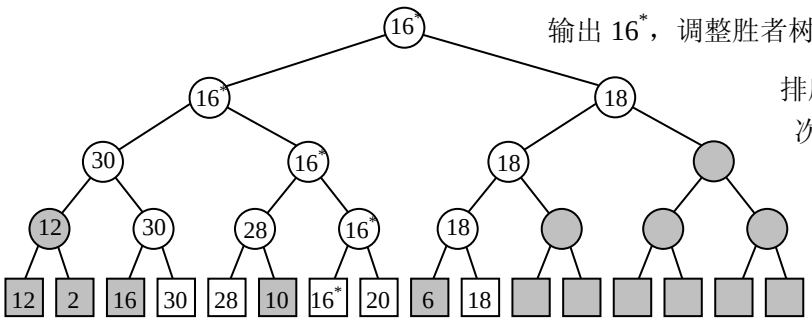
排序码比较次数 = 3。

排序码比较次数=3。某对象输出后，对手自动升到双亲，不计入排序码比较次数。



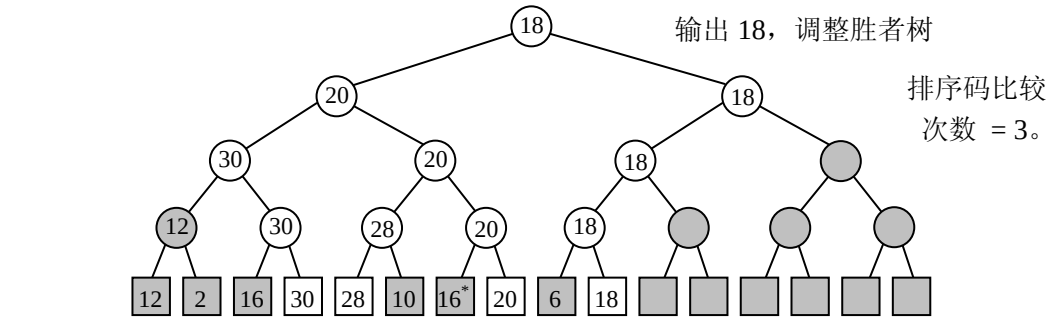

输出 16，调整胜者树

排序码比较次数 = 2。



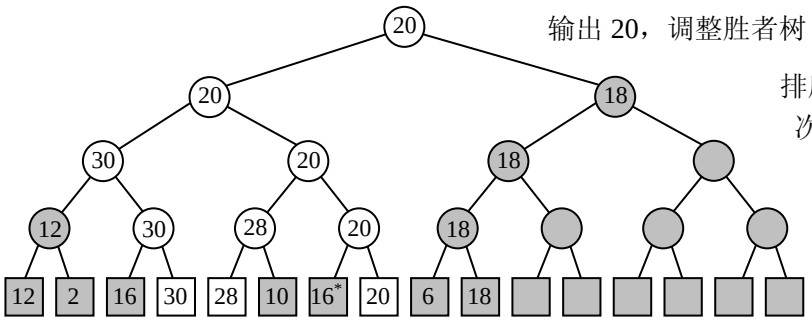

输出 16*，调整胜者树

排序码比较次数 = 2。



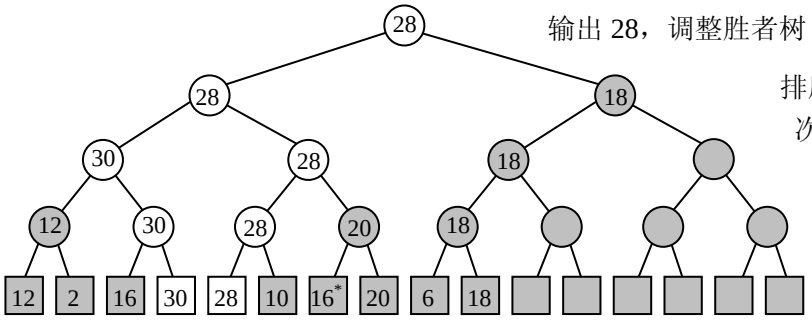

输出 18，调整胜者树

排序码比较
次数 = 3。



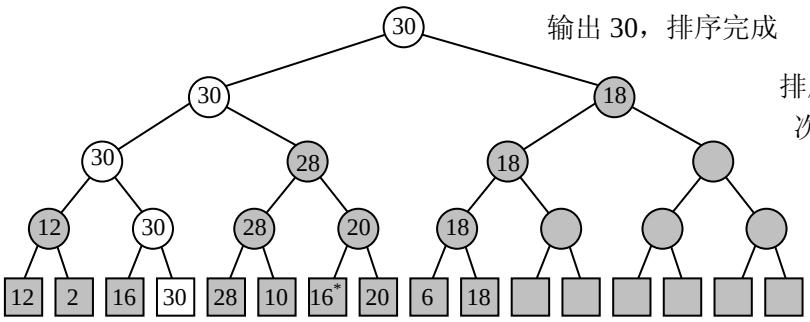

输出 20，调整胜者树

排序码比较
次数 = 0。




输出 28，调整胜者树

排序码比较
次数 = 2。

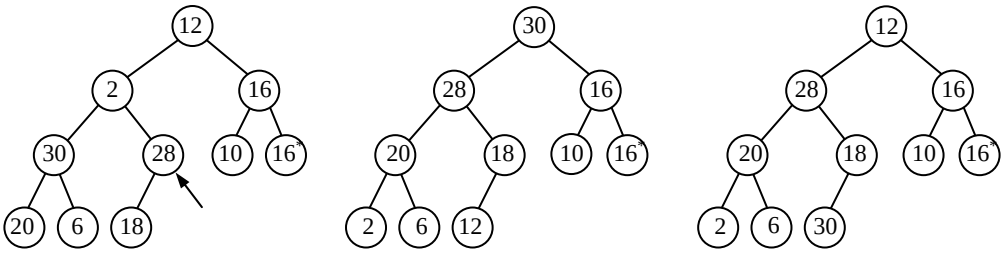



输出 30，排序完成

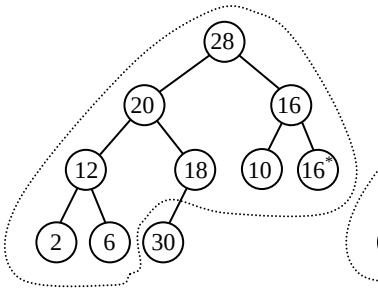
排序码比较
次数 = 0。

(7) 堆排序

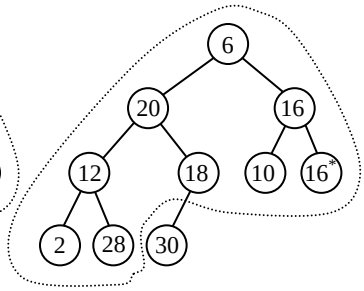
第一步，形成初始的最大堆 (略)，第二步，做堆排序。



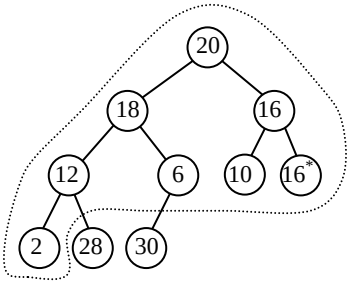
初始排列，不是最大堆



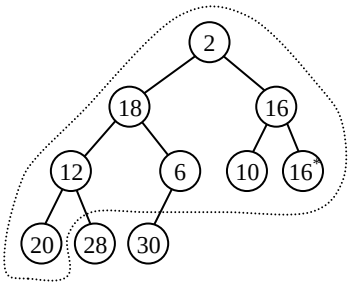

形成初始最大堆



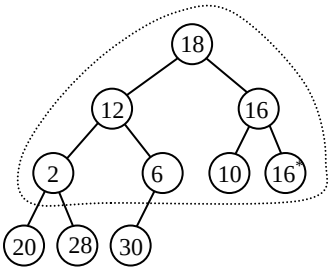

交换 $0^{\#}$ 与 $9^{\#}$ 对象



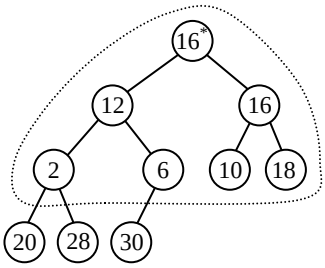

从 0# 到 8# 重新形成堆



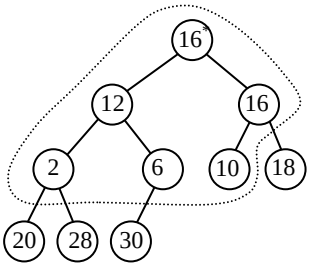

交换 0# 与 8# 对象



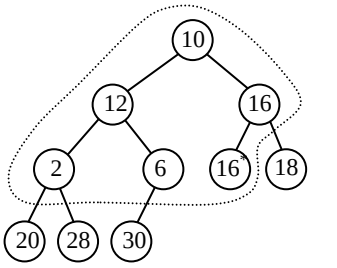

从 0# 到 7# 重新形成堆



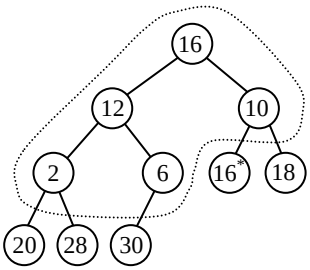

交换 0# 与 7# 对象



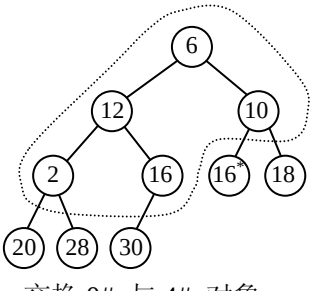

从 0# 到 6# 重新形成堆



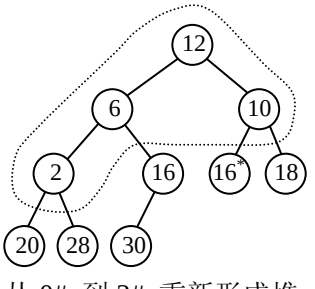

交换 0# 与 6# 对象



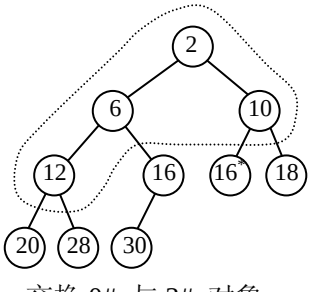

从 0# 到 5# 重新形成堆



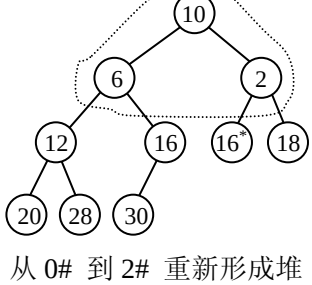

交换 0# 与 5# 对象



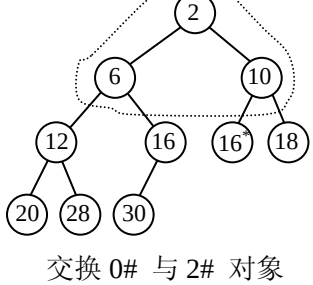

从 0# 到 4# 重新形成堆



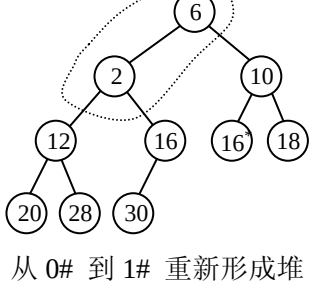

交换 0# 与 4# 对象




从 0# 到 3# 重新形成堆




交换 0# 与 3# 对象



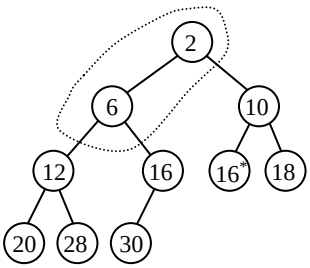

从 0# 到 2# 重新形成堆



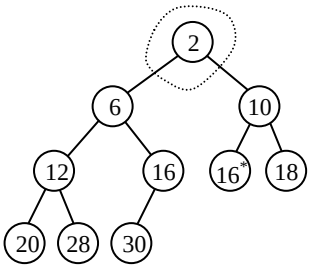

交换 0# 与 2# 对象




从 0# 到 1# 重新形成堆



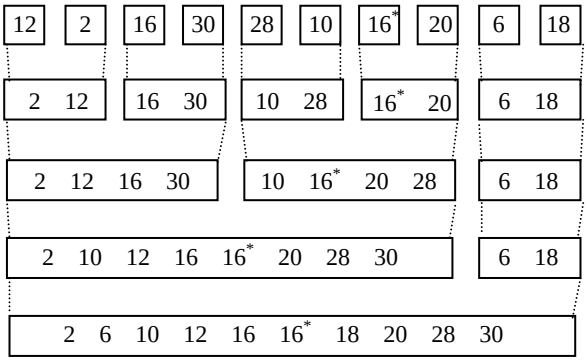

交换 0# 与 1# 对象



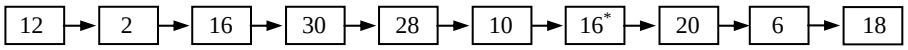

从 0# 到 1# 重新形成堆，得到结果

(8) 二路归并排序

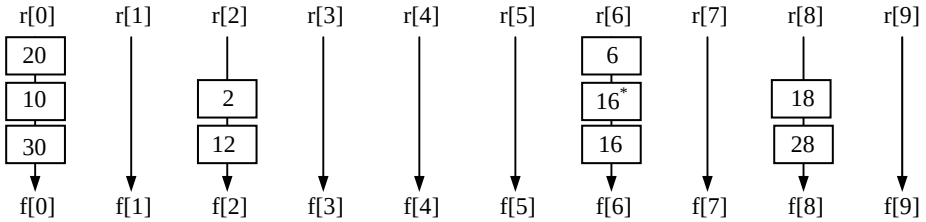
采用迭代的方法进行归并排序。设待排序的数据对象有 n 个。首先把每一个待排序的数据对象看作是长度为的初始归并项，然后进行两两归并，形成长度为 2 的归并项，再对它们两两归并，形成长度为 4 的归并项，如此一趟一趟做下去，最后得到长度为 n 的归并结果。



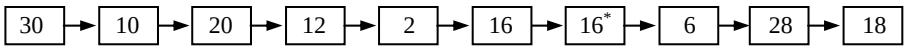

(9) 基数排序



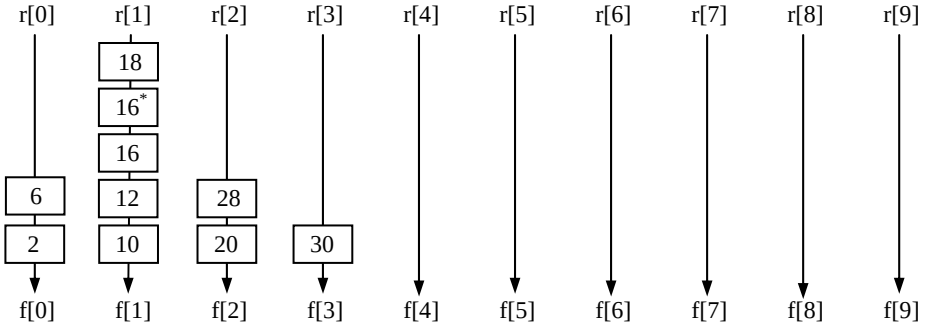

按最低位分配

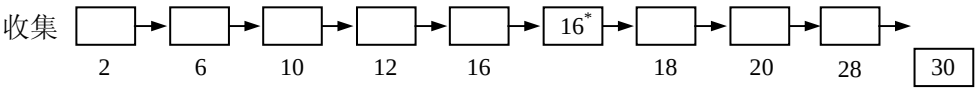



收集




按最高位分配






9-3 在起泡排序过程中，什么情况下排序码会朝向与排序相反的方向移动，试举例说明。在快速排序过程中有这种现象吗？

【解答】

如果在待排序序列的后面的若干排序码比前面的排序码小，则在起泡排序的过程中，排序码可能向与最终它应移向的位置相反的方向移动。例如，

| | |
|---|---|
| 57 40 38 11 13 34 48 75 6 19 9 7 | 如 9 向相反方向移动 |
| 6 57 40 38 11 13 34 48 75 7 19 9 | 如 19 向相反方向移动 |
| 6 7 57 40 38 11 13 34 48 75 9 19 | 如 9 向最终方向移动 |
| 6 7 9 57 40 38 11 13 34 48 75 19 | 如 13 向相反方向移动 |
| 6 7 9 11 57 40 38 13 19 34 48 75 | 如 13 向最终方向移动 |
| 6 7 9 11 13 57 40 38 19 34 48 75 | 如 34 向相反方向移动 |
| 6 7 9 11 13 19 57 40 38 34 48 75 | |
| 6 7 9 11 13 19 34 57 40 38 48 75 | |

9-4 试修改起泡排序算法，在正反两个方向交替进行扫描，即第一趟把排序码最大的对象放到序列的最后，第二趟把排序码最小的对象放到序列的最前面。如此反复进行。

【解答 1】

```
template <class Type> void dataList<Type> :: shaker_Sort ( ) {
//奇数趟对表 Vector 从前向后，比较相邻的排序码，遇到逆序即交换，直到把参加比较排序码序列
//中最大的排序码移到序列尾部。偶数趟从后向前，比较相邻的排序码，遇到逆序即交换，直到把
//参加比较排序码序列中最小的排序码移到序列前端。
    int i = 1, j;    int exchange;
    while ( i < CurrentSize ) {                    //起泡排序趟数不超过 n-1
        exchange = 0;                              //假定元素未交换
        for ( j = CurrentSize-i; j >= i; j-- )     //逆向起泡
            if ( Vector[j-1] > Vector[j] ) {       //发生逆序
                Swap ( Vector[j-1], Vector[j] );       //交换，最小排序码放在 Vector[i-1]处
                exchange = 1;                      //做“发生了交换”标志
            }
        if ( exchange == 0 ) break;                ////当 exchange 为 0 则停止排序
        for ( j = i; j <= CurrentSize-i-1; j++ )   //正向起泡
            if ( Vector[j] > Vector[j+1] ) {       //发生逆序
                Swap ( Vector[j], Vector[j+1] );       //交换，最大排序码放在 Vector[n-i]处
                exchange = 1;                      //做“发生了交换”标志
            }
        if ( exchange == 0 ) break;                //当 exchange 为 0 则停止排序
        i++;
    }
}
```

【解答 2】

```
template <class Type> void dataList<Type> :: shaker_Sort ( ) {
```

```
    int low = 1, high = CurrentSize−1, i, j;   int exchange;
    while ( low < high ) {                          //当比较范围多于一个对象时排序
        j = low;                                    //记忆元素交换位置
        for ( i = low; i < high; i++ )                  //正向起泡
            if ( Vector[i] > Vector[i+1] ) {        //发生逆序
                Swap ( Vector[i], Vector[i+1] );        //交换
                j = i;                              //记忆右边最后发生交换的位置 j
            }
        high = j;                                   //比较范围上界缩小到 j
        for ( i = high; i > low; i−−)               //反向起泡
            if ( Vector[i−1] > Vector[i] ) {        //发生逆序
                Swap ( Vector[i−1], Vector[i] );        //交换
                j = i;                              //记忆左边最后发生交换的位置 j
            }
        low = j;                                    //比较范围下界缩小到 j
    }
}
```

9−5 如果待排序的排序码序列已经按非递减次序有序排列，试证明函数 QuickSort( )的计算时间将下降到 $O(n^2)$。

【证明】

若待排序的 n 个对象的序列已经按排序码非递减次序有序排列，且设排序的时间代价为 T(n)。当以第一个对象作为基准对象时，应用一次划分算法 Partition，通过 n−1 次排序码比较，把只能把整个序列划分为：基准对象的左子序列为空序列，右子序列为有 n−1 个对象的非递减有序序列。对于这样的递归 QuickSort( )算法，其时间代价为

$$
\begin{aligned}
T(n) &= (n-1) + T(n-1) \\
&= (n-1) + \{( n-2) + T(n-2) \} \\
&= (n-1) + (n-2) + \{(n-3) + T(n-3) \} \\
&= \cdots\cdots \\
&= (n-1) + (n-2) + (n-3) + \cdots + \{ 2 + T(1) \} \\
&= (n-1) + (n-2) + (n-3) + \cdots + 2 + 1 \\
&= n(n-1)/2 = O(n^2)
\end{aligned}
$$

9−6 在实现快速排序的非递归算法时，可根据基准对象，将待排序排序码序列划分为两个子序列。若下一趟首先对较短的子序列进行排序，试证明在此做法下，快速排序所需要的栈的深度为 $O(\log_2 n)$。

【解答】

由快速排序的算法可知, 所需递归工作栈的深度取决于所需划分的最大次数。如果在排序过程中每次划分都能把整个待排序序列根据基准对象划分为左、右两个子序列。假定这两个子序列的长度相等，则所需栈的深度为

$$
\begin{aligned}
S(n) &= 1 + S(n/2) = \\
&= 1 + \{ 1 + S(n/4) \} = 2 + S(n/4) \\
&= 2 + \{ 1 + S(n/8) \} = 3 + S(n/8) \\
&= \cdots\cdots
\end{aligned}
$$

$= \log_2 n + S(1) = \log_2 n$　　(假设 1 个对象的序列所用递归栈的深度为 0)

如果每次递归左、右子序列的长度不等，并且先将较长的子序列的左、右端点保存在递归栈中，再对较短的子序列进行排序，可用表示最坏情况的大 O 表示法表示。此时其递归栈的深度不一定正好是 $\log_2 n$，其最坏情况为 $O(\log_2 n)$。

9-7 在实现快速排序算法时，可先检查位于两端及中点的排序码，取三者之中的数值不是最大也不是最小的排序码作为基准对象。试编写基于这种思想的快速排序算法，并证明对于已排序的排序码序列，该算法的计算时间为 $O(n\log_2 n)$。

【解答】参看教科书

9-8 在使用非递归方法实现快速排序时，通常要利用一个栈记忆待排序区间的两个端点。那么能否用队列来代替这个栈？为什么？

【解答】

可以用队列来代替栈。在快速排序的过程中，通过一趟划分，可以把一个待排序区间分为两个子区间，然后分别对这两个子区间施行同样的划分。栈的作用是在处理一个子区间时，保存另一个子区间的上界和下界，待该区间处理完成后再从栈中取出另一子区间的边界，对其进行处理。这个功能利用队列也可以实现，只不过是处理子区间的顺序有所变动而已。

9-9 试设计一个算法，使得在 O(n)的时间内重排数组，将所有取负值的排序码排在所有取正值(非负值)的排序码之前。

【解答】

```
template<class Type> void reArrange ( dataList<Type>& L ) {
//数组元素类型 Type 只可能取 int 或 float
    int i = 0, j = L.length () – 1;
    while ( i != j ) {
        while ( L[i].getKey( ) < 0 ) i++;
        while ( L[j].getKey( ) >= 0 ) j--;
        swap ( L[i], L[j] );
        i++;   j--;
    }
}
```

9-10 奇偶交换排序是另一种交换排序。它的第一趟对序列中的所有奇数项 i 扫描，第二趟对序列中的所有偶数项 i 扫描。若 A[i] > A[i+1]，则交换它们。第三趟有对所有的奇数项，第四趟对所有的偶数项，…，如此反复，直到整个序列全部排好序为止。

(1) 这种排序方法结束的条件是什么？

(2) 写出奇偶交换排序的算法。

(3) 当待排序排序码序列的初始排列是从小到大有序，或从大到小有序时，在奇偶交换排序过程中的排序码比较次数是多少？

【解答】

(1) 设有一个布尔变量 exchange，判断在每一次做过一趟奇数项扫描和一趟偶数项扫描后是否有过交换。若 exchange = 1，表示刚才有过交换，还需继续做下一趟奇数项扫描和一趟偶数项扫描；若 exchange = 0，表示刚才没有交换，可以结束排序。

(2) 奇偶排序的算法

```
template<Type> void dataList<Type> :: odd-evenSort ( ) {
    int i, exchange;
    do {
        exchange = 0;
        for ( i = 1; i < CurrentSize; i += 2 )          //扫描所有奇数项
          if ( Vector[i] > Vector[i+1] ) {              //相邻两项比较，发生逆序
              exchange = 1;                             //作交换标记
              swap ( Vector[i], Vector[i+1] );          //交换
          }
        for ( i = 0; i < CurrentSize; i += 2 )          //扫描所有偶数项
          if ( Vector[i] > Vector[i+1] ) {              //相邻两项比较，发生逆序
              exchange = 1;                             //作交换标记
              swap ( Vector[i], Vector[i+1] );          //交换
          }
    } while ( exchange != 0 );
}
```

(3) 设待排序对象序列中总共有 n 个对象。序列中各个对象的序号从 0 开始。则当所有待排序对象序列中的对象按排序码从大到小初始排列时，执行 m = ⌊(n+1)/2⌋ 趟奇偶排序。当所有待排序对象序列中的对象按排序码从小到大初始排列时，执行 1 趟奇偶排序。

在一趟奇偶排序过程中，对所有奇数项扫描一遍，排序码比较 ⌊(n−1)/2⌋ 次；对所有偶数项扫描一遍，排序码比较 ⌊n/2⌋ 次。所以每趟奇偶排序两遍扫描的结果，排序码总比较次数为 ⌊(n−1)/2⌋+⌊n/2⌋ = n−1。

9−11 请编写一个算法，在基于单链表表示的待排序排序码序列上进行简单选择排序。

【解答】

采用静态单链表作为存储表示。用 Vector[0]作为表头结点，各待排序数据对象从 Vector[1]开始存放。算法的思想是每趟在原始链表中摘下排序码最大的结点(几个排序码相等时为最前面的结点)，把它插入到结果链表的最前端。由于在原始链表中摘下的排序码越来越小，在结果链表前端插入的排序码也越来越小，最后形成的结果链表中的结点将按排序码非递减的顺序有序链接。

```
Template<class Type> void staticlinkList<Type> :: selectSort ( ) {
    int h = Vector[0].link, p, q, r, s;
    Vector[0].link = 0;
    while ( h != 0 ) {                              //原始链表未扫描完
        p = s = h;    q = r = 0;
        while ( p != 0 ) {                              //扫描原始链表，寻找排序码最大的结点 s
          if ( Vector[p].data > Vector[s].data )         //记忆当前找到的排序码最大结点
              { s = p;    r = q; }
          q = p;    p = Vector[p].link;
        }
        if ( s == h ) h = Vector[h];                    //排序码最大的结点是原始链表前端结点，摘下
        else Vector[r].link = Vector[s].link;           //排序码最大的结点是原始链表表中结点，摘下
        Vector[s].link = Vector[0].link;                //结点 s 插入到结果链表的前端
        Vector[0].link = s;
```

```
    }
}
```

9-12 若参加锦标赛排序的排序码有 11 个，为了完成排序，至少需要多少次排序码比较？

【解答】

对于有 n(n>0)个数据的序列，锦标赛排序选最小数据需进行 n-1 次数据比较，以后每选一个数据，进行数据比较的次数，均需 $\lfloor \log_2 n \rfloor - 1$ 次（在外结点层无比较）。对于有 11 个排序码的序列，第一次选具最小排序码的数据，需进行 10 次排序码比较，以后在剩下的序列中每选一个具最小排序码的数据，都需进行 $\lfloor \log_2 11 \rfloor - 1 = 2$ 次排序码比较，因此，为了完成排序，需要 10 + 2*10 = 30 次排序码比较。

9-13 试给出适用于锦标赛排序的胜者树的类型声明。并写一个函数，对 n 个参加排序的对象，构造胜者树。设 n 是 2 的幂。

【解答】

适用于锦标赛排序的胜者树的类型声明.

```
template <class Type> class DataNode {          //胜者树结点的类定义
public:
    Type data;                  //数据值
    int index;                  //树中的结点号, 即在完全二叉树顺序存储中的下标
    int active;                 //是否参选的标志， =1, 参选;  =0, 不再参选
}

template <class Type> void TournamentSort ( Type a[ ], int n ) {
//建立树的顺序存储数组 tree, 将数组 a[ ]中的元素复制到胜者树中, 对它们进行排序, 并把结
//果返送回数组中, n 是待排序元素个数。
    DataNode<Type> *tree;           //胜者树结点数组
    DataNode<Type> item;
    int bottomRowSize = PowerOfTwo ( n );
                    //计算满足>=n 的 2 的最小次幂的数：树的底行大小 n=7 时它为 8
    int TreeSize = 2 * bottomRowSize - 1;       //计算胜者树的大小: 内结点+外结点数
    int loadindex = bottomRowSize - 1;          //外结点开始位置
    tree = new DataNode<Type>[TreeSize];        //动态分配胜者树结点数组空间
    int j = 0;                                  //在数组 a 中取数据指针
    for ( int i = loadindex; i < TreeSize; i++ ) {  //复制数组数据到树的外结点中
        tree[i].index = i;                      //下标
        if ( j < n ) { tree[i].active = 1;   tree[i].data = a[j++]; }   //复制数据
        else tree[i].active = 0;                //后面的结点为空的外结点
    }
    i = loadindex;                              //进行初始比较寻找最小的项
    while ( i ) {
        j = i;
        while ( j < 2*i ) {                     //处理各对比赛者
            if ( !tree[j+1].active || tree[j].data <= tree[j+1].data )
                tree[(j-1)/2] = tree[j];                //胜者送入双亲
```

```
            else tree[(j-1)/2] = tree[j+1];
            j += 2;                                  //下一对参加比较的项
          }
          i = (i-1)/2;                               //i 退到双亲, 直到 i=0 为止
        }
        for ( i=0; i<n-1; i++) {                     //处理其它 n-1 元素
          a[i] = tree[0].data;                       //当前最小元素送数组 a
          tree[tree[0].index].active = 0;            //该元素相应外结点不再比赛
          UpdateTree ( tree, tree[0].index );        //从该处向上修改
        }
        a[n-1] = tree[0].data;
    }

    template <class Type> void UpdateTree ( DataNode<Type> *tree, int i ) {
    //锦标赛排序中的调整算法：i 是表中当前最小元素的下标, 即胜者。从它开始向上调整。
        if ( i %2 == 0 ) tree[(i-1)/2] = tree[i-1];  // i 为偶数, 对手为左结点
        else tree[(i-1)/2] = tree[i+1];              // i 为奇数, 对手为右结点
        //最小元素输出之后, 它的对手上升到父结点位置
        i = (i - 1) / 2;                             // i 上升到双亲结点位置
        while ( i ) {
          if ( i %2 == 0) j = i - 1;                 //确定 i 的对手是左结点还是右结点
          else j = i + 1;
          if ( !tree[i].active || !tree[j].active )  //比赛对手中间有一个为空
            if ( tree[i].active ) tree[(i-1)/2] = tree[i];
            else tree[(i-1)/2] = tree[j];            //非空者上升到双亲结点
          else                                       //比赛对手都不为空
            if ( tree[i].data < tree[j].data ) tree[(i-1)/2] = tree[i];
            else tree[(i-1)/2] = tree[j];            //胜者上升到双亲结点
          i = (i - 1) / 2;                           // i 上升到双亲结点
        }
    }
```

9-14 手工跟踪对以下各序列进行堆排序的过程。给出形成初始堆及每选出一个排序码后堆的变化。

(1) 按字母顺序排序：Tim, Dot, Eva, Rom, Kim, Guy, Ann, Jim, Kay, Ron, Jan

(2) 按数值递增顺序排序：26, 33, 35, 29, 19, 12, 22

(3) 同样 7 个数字，换一个初始排列，再按数值的递增顺序排序：12, 19, 33, 26, 29, 35, 22

【解答】　为节省篇幅，将用数组方式给出形成初始堆和进行堆排序的变化结果。阴影部分表示参与比较的排序码。请读者按照完全二叉树的顺序存储表示画出堆的树形表示。

(1) 按字母顺序排序

形成初始堆（按最大堆）

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 |
|---|---|---|---|---|---|---|---|---|---|---|---|
| | Tim | Dot | Eva | Rom | Kim | Guy | Ann | Jim | Kay | Ron | Jan |
| i=4 | Tim | Dot | Eva | Rom | [ Ron | Guy | Ann | Jim | Kay | Kim | Jan ] |

| | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| i=3 | Tim | Dot | Eva | [ Rom | Ron | Guy | Ann | Jim | Kay | Kim | Jan ] |
| i=2 | Tim | Dot | [ Guy | Rom | Ron | Eva | Ann | Jim | Kay | Kim | Jan ] |
| i=1 | Tim | [ Ron | Guy | Rom | Kim | Eva | Ann | Jim | Kay | Dot | Jan ] |
| i=0 | [ Tim | Ron | Guy | Rom | Kim | Eva | Ann | Jim | Kay | Dot | Jan ] |

堆排序

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| j=10 | [ Jan | Ron | Guy | Rom | Kim | Eva | Ann | Jim | Kay | Dot | Tim ] | 交换 |
| | [ Ron | Rom | Guy | Kay | Kim | Eva | Ann | Jim | Jan | Dot ] | Tim | 调整 |
| j=9 | [ Dot | Rom | Guy | Kay | Kim | Eva | Ann | Jim | Jan | Ron ] | Tim | 交换 |
| | [ Rom | Kim | Guy | Kay | Dot | Eva | Ann | Jim | Jan ] | Ron | Tim | 调整 |
| j=8 | [ Jan | Kim | Guy | Kay | Dot | Eva | Ann | Jim | Rom ] | Ron | Tim | 交换 |
| | [ Kim | Kay | Guy | Jim | Dot | Eva | Ann | Jan ] | Rom | Ron | Tim | 调整 |
| j=7 | [ Jan | Kay | Guy | Jim | Dot | Eva | Ann | Kim ] | Rom | Ron | Tim | 交换 |
| | [ Kay | Jim | Guy | Jan | Dot | Eva | Ann ] | Kim | Rom | Ron | Tim | 调整 |
| j=6 | [ Ann | Jim | Guy | Jan | Dot | Eva | Kay ] | Kim | Rom | Ron | Tim | 交换 |
| | [ Jim | Jan | Guy | Ann | Dot | Eva ] | Kay | Kim | Rom | Ron | Tim | 调整 |
| j=5 | [ Eva | Jan | Guy | Ann | Dot | Jim ] | Kay | Kim | Rom | Ron | Tim | 交换 |
| | [ Jan | Eva | Guy | Ann | Dot ] | Jim | Kay | Kim | Rom | Ron | Tim | 调整 |
| j=4 | [ Dot | Eva | Guy | Ann | Jan ] | Jim | Kay | Kim | Rom | Ron | Tim | 交换 |
| | [ Guy | Eva | Dot | Ann ] | Jan | Jim | Kay | Kim | Rom | Ron | Tim | 调整 |
| j=3 | [ Ann | Eva | Dot | Guy ] | Jan | Jim | Kay | Kim | Rom | Ron | Tim | 交换 |
| | [ Eva | Ann | Dot ] | Guy | Jan | Jim | Kay | Kim | Rom | Ron | Tim | 调整 |
| j=2 | [ Dot | Ann | Eva ] | Guy | Jan | Jim | Kay | Kim | Rom | Ron | Tim | 交换 |
| | [ Dot | Ann ] | Eva | Guy | Jan | Jim | Kay | Kim | Rom | Ron | Tim | 调整 |
| j=1 | [ Dot | Ann ] | Eva | Guy | Jan | Jim | Kay | Kim | Rom | Ron | Tim | 交换 |
| | [ Ann ] | Dot | Eva | Guy | Jan | Jim | Kay | Kim | Rom | Ron | Tim | 调整 |

(2) 按数值递增顺序排序

形成初始堆 (按最大堆)

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| | 26 | 33 | 35 | 29 | 19 | 12 | 22 |
| i=2 | 26 | 33 | [ 35 | 29 | 19 | 12 | 22 ] |
| i=0 | 26 | [ 33 | 35 | 29 | 19 | 12 | 22 ] |
| i=1 | [ 35 | 33 | 26 | 29 | 19 | 12 | 22 ] |

堆排序

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | |
|---|---|---|---|---|---|---|---|---|
| j=6 | [ 22 | 33 | 26 | 29 | 19 | 12 | 35 ] | 交换 |
| | [ 33 | 29 | 26 | 22 | 19 | 12 ] | 35 | 调整为堆 |
| j=5 | [ 12 | 29 | 26 | 22 | 19 | 33 ] | 35 | 交换 |
| | [ 29 | 22 | 26 | 12 | 19 ] | 33 | 35 | 调整为堆 |
| j=4 | [ 19 | 22 | 26 | 12 | 29 ] | 33 | 35 | 交换 |

| | | | | | | | | |
|---|---|---|---|---|---|---|---|---|
| | [ 26 | 22 | 19 | 12 ] | 29 | 33 | 35 | 调整为堆 |
| j=3 | [ 12 | 22 | 19 | 26 ] | 29 | 33 | 35 | 交换 |
| | [ 22 | 12 | 19 ] | 26 | 29 | 33 | 35 | 调整为堆 |
| j=2 | [ 19 | 12 | 22 ] | 26 | 29 | 33 | 35 | 交换 |
| | [ 19 | 12 ] | 22 | 26 | 29 | 33 | 35 | 调整为堆 |
| j=1 | [ 12 | 19 ] | 22 | 26 | 29 | 33 | 35 | 交换 |
| | [ 12 ] | 19 | 22 | 26 | 29 | 33 | 35 | 调整为堆 |

(3) 同样 7 个数字，换一个初始排列，再按数值的递增顺序排序

形成初始堆 (按最大堆)

| | 0 | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|---|
| | 12 | 19 | 33 | 26 | 29 | 35 | 22 |
| i=2 | 12 | 19 | [ 35 | 26 | 29 | 33 | 22 ] |
| i=0 | 12 | [ 29 | 35 | 26 | 19 | 33 | 22 ] |
| i=1 | [ 35 | 29 | 33 | 26 | 19 | 12 | 22 ] |

堆排序

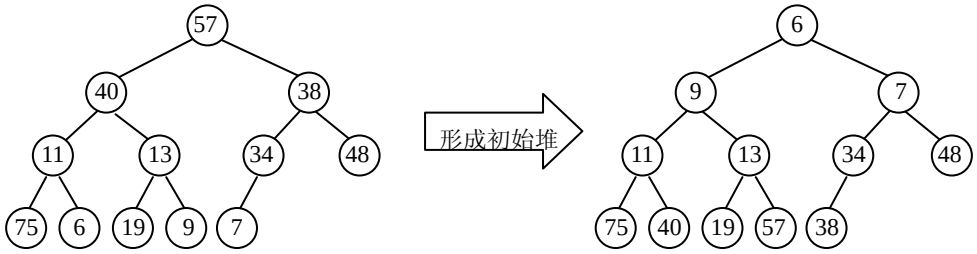
| | 0 | 1 | 2 | 3 | 4 | 5 | 6 | |
|---|---|---|---|---|---|---|---|---|
| j=6 | [ 22 | 29 | 33 | 26 | 19 | 12 | 35 ] | 交换 |
| | [ 33 | 29 | 22 | 26 | 19 | 12 ] | 35 | 调整为堆 |
| j=5 | [ 12 | 29 | 22 | 26 | 19 | 33 ] | 35 | 交换 |
| | [ 29 | 26 | 22 | 12 | 19 ] | 33 | 35 | 调整为堆 |
| j=4 | [ 19 | 26 | 22 | 12 | 29 ] | 33 | 35 | 交换 |
| | [ 26 | 19 | 22 | 12 ] | 29 | 33 | 35 | 调整为堆 |
| j=3 | [ 12 | 19 | 22 | 26 ] | 29 | 33 | 35 | 交换 |
| | [ 22 | 19 | 12 ] | 26 | 29 | 33 | 35 | 调整为堆 |
| j=2 | [ 12 | 19 | 22 ] | 26 | 29 | 33 | 35 | 交换 |
| | [ 19 | 12 ] | 22 | 26 | 29 | 33 | 35 | 调整为堆 |
| j=1 | [ 12 | 19 ] | 22 | 26 | 29 | 33 | 35 | 交换 |
| | [ 12 ] | 19 | 22 | 26 | 29 | 33 | 35 | 调整为堆 |

9-15 如果只想在一个有 n 个元素的任意序列中得到其中最小的第 k (k<<n) 个元素之前的部分排序序列，那么最好采用什么排序方法？为什么？例如有这样一个序列: **{**503, 017, 512, 908, 170, 897, 275, 653, 612, 154, 509, 612*, 677, 765, 094**}**，要得到其第 4 个元素之前的部分有序序列: **{**017, 094, 154, 170**}**，用所选择的算法实现时，要执行多少次比较?

【解答】

一般来讲，当 n 比较大且要选的数据 k << n 时，采用堆排序方法中的调整算法 FilterDown( )最好。但当 n 比较小时，采用锦标赛排序方法更好。

例如，对于序列{ 57, 40, 38, 11, 13, 34, 48, 75, 6, 19, 9, 7 }，选最小的数据 6，需形成初始堆，进行 18 次数据比较；选次小数据 7 时，需进行 4 次数据比较；再选数据 9 时，需进行 6 次数据比较；选数据 11 时，需进行 4 次数据比较。




但如果选用锦标赛排序，对于有 n(n>0)个数据的序列，选最小数据需进行 n−1 次数据比较，以后每选一个数据，进行数据比较的次数，均需 $\lfloor \log_2 n \rfloor - 1$ 次。例如，同样 12 个数据，第一次选最小的数据 6，需进行 11 次数据比较，以后选 7、9、11 时，都是 $\lfloor \log_2 12 \rfloor - 1 = 2$ 次数据比较。

9−16 希尔排序、简单选择排序、快速排序和堆排序是不稳定的排序方法，试举例说明。

【解答】

(1) 希尔排序　{ 512　275　275*　061 }　增量为 2

{ 275*　061　512　275 }　增量为 1

{ 061　275*　275　512 }

(2) 直接选择排序 { 275　275*　512　061 }　i = 1

{ **061**　275*　512　275 }　i = 2

{ **061**　**275***　512　275 }　i = 3

{ **061**　**275***　**275**　512 }

(3) 快速排序　{ 512　275　275* }

{ 275*　275　**512** }

(4) 堆排序　{ 275　275*　061　170 }　已经是最大堆，交换 275 与 170

{ 170　275*　061　**275** }　对前 3 个调整

{ 275*　170　061　**275** }　前 3 个最大堆，交换 275*与 061

{ 061　170　**275***　**275** }　对前 2 个调整

{ 170　061　**275***　**275** }　前 2 个最大堆，交换 170 与 061

{ 061　**170**　**275***　**275** }

9−17 设有 n 个待排序元素存放在一个不带表头结点的单链表中，每个链表结点只存放一个元素，头指针为 r。试设计一个算法，对其进行二路归并排序，要求不移动结点中的元素，只改各链结点中的指针，排序后 r 仍指示结果链表的第一个结点。（提示：先对待排序的单链表进行一次扫描，将它划分为若干有序的子链表， 其表头指针存放在一个指针队列中。当队列不空时重复执行，从队列中退出两个有序子链表，对它们进行二路归并， 结果链表的表头指针存放到队列中。如果队列中退出一个有序子链表后变成空队列，则算法结束。这个有序子链表即为所求。)

【解答】

(1) 两路归并算法

```
template<Type> void staticlinkList<Type> :: merge ( int ha;   int hb;   int& hc ) {
//合并两个以 ha 和 hb 为表头指针的有序链表，结果链表的表头由 hc 返回
```

```
    int pa, pb, pc;
    if ( Vector[ha].data <= Vector[hb].data )          //确定结果链的表头
       { hc = ha;   pa = Vector[ha].link;   pb = hb; }
    else { hc = hb;   pb = Vector[hb].link;   pa = ha; }
    pc = hc;                                           //结果链的链尾指针
    while ( pa != 0 ) and ( pb != 0 )                  //两两比较，小者进结果链
       if ( Vector[pa].data <= Vector[pb].data )
          { Vector[pc].link = pa;   pc = pa;   pa = Vector[pa].link; }
       else { Vector[pc].link = pb;   pc = pb;   pb = Vector[pb].link; }
    if ( pa != 0 ) Vector[pc].link = pa;               // pb 链处理完, pa 链链入结果链
    else Vector[pc].link = pb;                                 // pa 链处理完, pb 链链入结果链
}
```

(2) 归并排序主程序

```
template<class type> void staticlinkList<Type> :: merge_sort ( ) {
    int r, s, t;   Queue <int> Q;
    if ( Vector[0].link == 0 ) return;
    s = Vector[0].link;   Q.Enqueue( s );              //链表第一个结点进队列
    while ( 1 ) {
        t = Vector[s].link;                            //结点 t 是结点 s 的下一个链中结点
        while ( t != 0 && Vector[s].data <= Vector[t].data )
            { s = t;   t = Vector[t].link; }           //在链表中寻找一段有序链表
        Vector[s].link = 0;   s = t;
        if ( t != 0 ) Q.EnQueue( s );                  //存在一段有序链表，截取下来进队列
        else break;                                    //到链尾
    }
    while ( ! Q.IsEmpty( ) ) {
        r = Q.getFront( );   Q.DlQueue( );             //从队列退出一个有序链表的表头 r
        if ( Q.IsEmpty( ) ) break;                     //队列空，表示排序处理完成，退出
        s = Q.getFront( );   Q.DlQueue( );             //从队列再退出一个有序链表的表头 s
        merge( r, s, t );   Q.EnQueue( t );            //归并两个有序链表后结果链表进队列
    }
    Vector[0].link = r;
}
```

9-18 若设待排序排序码序列有 n 个排序码，n 是一个完全平方数。将它们划分为$\sqrt{n}$块，每块有$\sqrt{n}$个排序码。这些块分属于两个有序表。下面给出一种 O(1)空间的非递归归并算法：

**step1:** 在两个待归并的有序表中从右向左总共选出$\sqrt{n}$个具有最大值的排序码；

**step2:** 若设在 **step1** 选出的第 2 个有序表中的排序码有 s 个，则从第 1 个有序表选出的排序码有$\sqrt{n}$- s 个。将第 2 个有序表选出的 s 个排序码与第 1 个有序表选出的排序码左边的同样数目的排序码对调；

**step3:** 交换具有最大$\sqrt{n}$个排序码的块与最左块(除非最左块就是具有最大$\sqrt{n}$个排序码的块)。对最右块进行排序；

**step4:** 除去具有最大$\sqrt{n}$个排序码的块以外，对其它的块根据其最后的排序码按非递减顺序排序；

**step5:** 设置 3 个指针，分别位于第 1 块、第 2 块和第 3 块的起始位置，执行多次 **substep**，直到 3 个指针都走到第 $\sqrt{n}$ 块为止。此时前 $\sqrt{n}-1$ 块已经排好序。

☞ **subStep** 所做的工作是比较第 2 个指针与第 3 个指针所指排序码，将值小的与第 1 个指针所指排序码对调，相应指针前进 1 个排序码位置。

**step6:** 对最后第 $\sqrt{n}$ 块中最大的 $\sqrt{n}$ 个排序码进行排序。

(1) 设有 16 个排序码，分别存放于两个有序表**{**10, 12, 14, 16, 18, 20, 22, 25**}**和**{**11, 13, 15, 17, 19, 21, 23, 24**}**中，试根据上面的描述，写出排序的全过程，并说明它具有时间复杂度 O(n)和空间复杂度 O(1)。

(2) 编写相应的算法。要求两个待排序有序表的长度可以不同，但每一个表的长度都是 $\sqrt{n}$ 的倍数。

(3) 假设两个有序表分别为($x_1$, …, $x_m$)和($x_{m+1}$, …, $x_n$)，编写一个算法归并这两个有序表，得到($x_1$, …, $x_n$)。设 s = $\sqrt{n}$ 。

9-19 试编写一个算法，将对象序列($x_1$, $x_2$, …, $x_n$)循环右移 p 个位置，$0 \le p \le n$。要求该算法的时间复杂度为 O(n)而空间复杂度为 O(1)。

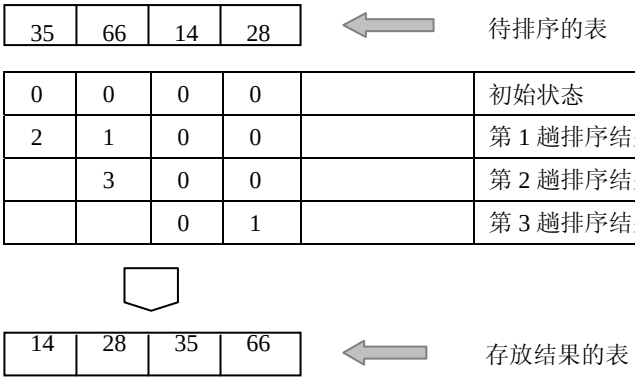
【解答】略

9-20 在什么条件下，MSD 基数排序比 LSD 基数排序效率更高？

【解答】由于高位优先的 MSD 方法是递归的方法，就一般情况来说，不像低位优先的 LSD 方法那样直观自然，而且实现的效率较低。但如果待排序的排序码的大小只取决于高位的少数几位而与大多数低位无关时，采用 MSD 方法比 LSD 方法的效率要高。

9-21 在已排好序的序列中，一个对象所处的位置取决于具有更小排序码的对象的个数。基于这个思想，可得计数排序方法。该方法在声明对象时为每个对象增加一个计数域 count，用于存放在已排好序的序列中该对象前面的对象数目，最后依 count 域的值，将序列重新排列，就可完成排序。试编写一个算法，实现计数排序。并说明对于一个有 n 个对象的序列，为确定所有对象的 count 值，最多需要做 n(n−1)/2 次排序码比较。

【解答】




```
template <class Type> void datalist<Type> :: count_sort ( ) {
//initList是待排序表，resultList是结果表
    int i, j;
    int *c = new datalist <Type>;                    // c是存放计数排序结果的临时表
    for ( i = 0; i < CurrentSize; i++ ) Vector[i].count = 0;     //初始化，计数值都为0
    for ( i = 0; i < CurrentSize−1; i++ )
        for ( j = i+1; j < CurrentSize; j++ )
```

```
            if ( Vector[j].key < Vector[i].key ) Vector[i].count++;
            else Vector[j].count++;                          //统计
        for ( i = 0; i < CurrentSize; i++ )                  //在c->Vector[ ]中各就各位
            c->Vector[ Vector[i].count ] = Vector[i];
        for ( i = 0; i < CurrentSize; i++ ) Vector[i] = c->Vector[i];  //结果复制回当前表对象中
        delete c;
    }
```

9-22 试证明对一个有 n 个对象的序列进行基于比较的排序，最少需要执行 $n\log_2 n$ 次排序码比较。

【解答】

基于比较的排序方法中，采用分治法进行排序是平均性能最好的方法。方法描述如下：

```
Sort ( List ) {
  if ( List 的长度大于 1) {
    将序列 List 划分为两个子序列 LeftList 和 Right List;
    Sort ( LeftList );    Sort ( RightList );      //分别对两个子序列施行排序
    将两个子序列 LeftList 和 RightList 合并为一个序列 List;
  }
}
```

典型的例子就是快速排序和归并排序。若设 T(n) 是对 n 个对象的序列进行排序所需的时间，而且把序列划分为长度相等的两个子序列后，对每个子序列进行排序所需的时间为 T(n/2)，最后合并两个已排好序的子序列所需时间为 cn（c 是一个常数）。此时，总的计算时间为：

$T(n) \leqslant cn + 2\,T(n/2)$　　// c 是一个常数

$\leqslant cn + 2\,(cn/2 + 2T(n/4)) = 2cn + 4T(n/4)$

$\leqslant 2cn + 4\,(cn/4 + 2T(n/8)) = 3cn + 8T(n/8)$

……

$\leqslant cn\log_2 n + nT(1) = O(n\log_2 n)$

9-23 如果某个文件经内排序得到 80 个初始归并段，试问

(1) 若使用多路归并执行 3 趟完成排序，那么应取的归并路数至少应为多少？

(2) 如果操作系统要求一个程序同时可用的输入/输出文件的总数不超过 15 个，则按多路归并至少需要几趟可以完成排序？如果限定这个趟数，可取的最低路数是多少？

【解答】

(1) 设归并路数为 k，初始归并段个数 m = 80，根据归并趟数计算公式 $S = \lceil \log_k m \rceil = \lceil \log_k 80 \rceil = 3$ 得：$k^3 \geqslant 80$。由此解得 $k \geqslant 3$，即应取的归并路数至少为 3。

(2) 设多路归并的归并路数为 k，需要 k 个输入缓冲区和 1 个输出缓冲区。1 个缓冲区对应 1 个文件，有 k +1 = 15，因此 k = 14，可做 14 路归并。由 $S = \lceil \log_k m \rceil = \lceil \log_{14} 80 \rceil = 2$。即至少需 2 趟归并可完成排序。

若限定这个趟数，由 $S = \lceil \log_k 80 \rceil = 2$，有 $80 \leqslant k^2$，可取的最低路数为 9。即要在 2 趟内完成排序，进行 9 路排序即可。

9-24 假设文件有 4500 个记录，在磁盘上每个页块可放 75 个记录。计算机中用于排序的内存区可容纳 450 个记录。试问：

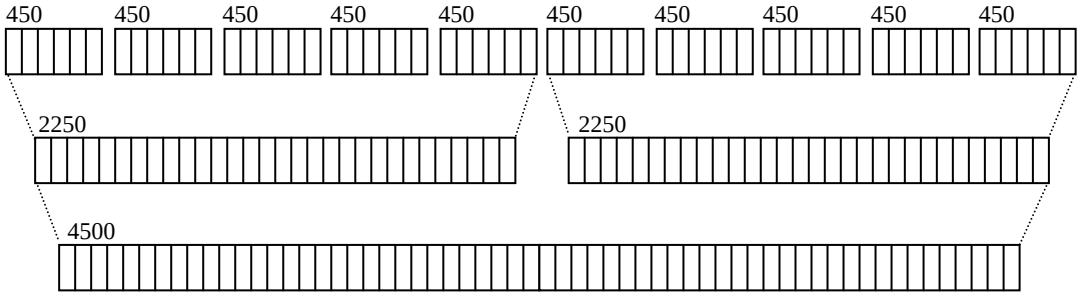
(1) 可建立多少个初始归并段？每个初始归并段有多少记录？存放于多少个页块中？

(2) 应采用几路归并？请写出归并过程及每趟需要读写磁盘的页块数。

【解答】

(1) 文件有 4500 个记录，计算机中用于排序的内存区可容纳 450 个记录，可建立的初始归并段有 4500 / 450 = 10 个。每个初始归并段中有 450 个记录，存于 450 / 75 = 6 个页块中。

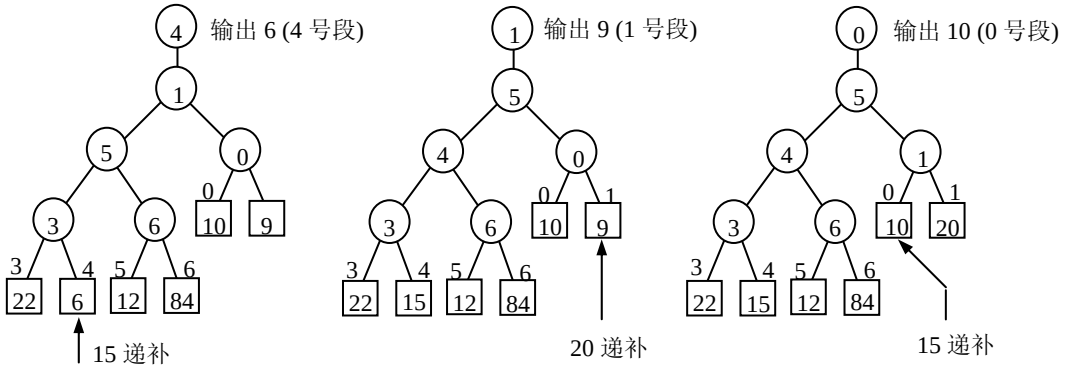
(2) 内存区可容纳 6 个页块，可建立 6 个缓冲区，其中 5 个缓冲区用于输入，1 个缓冲区用于输出，因此，可采用 5 路归并。归并过程如下：

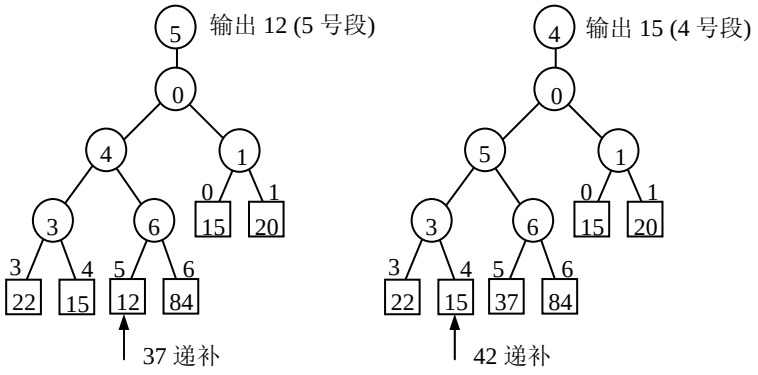



共做了 2 趟归并，每趟需要读 60 个磁盘页块，写出 60 个磁盘页块。

9-25 设初始归并段为(10, 15, 31, ∞), (9, 20, ∞), (22, 34, 37, ∞), (6, 15, 42, ∞), (12, 37, ∞), (84, 95, ∞)，试利用败者树进行 k 路归并，手工执行选择最小的 5 个排序码的过程。

【解答】做 6 路归并排序，选择最小的 5 个排序码的败者树如下图所示。




9-26 设输入文件包含以下记录：14, 22, 7, 24, 15, 16, 11, 100, 10, 9, 20, 12, 90, 17, 13, 19, 26, 38, 30, 25, 50, 28, 110, 21, 40。现采用置换-选择方法生成初始归并段，并假设内存工作区可同时容纳 5 个记录，请画出选择的过程。

【解答】

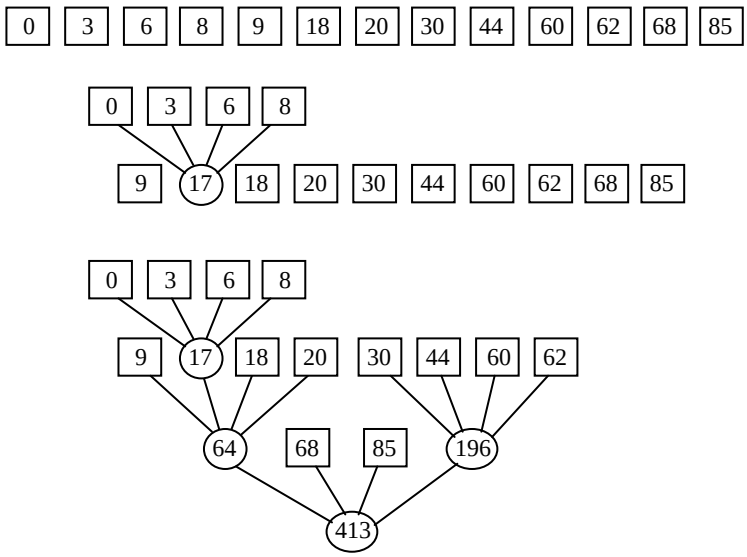
设内存工作区在某一时刻可以处理 6 个记录，利用败者树生成初始归并段的过程如下。

| 输入文件 InFile | 内存工作区 | 输出文件 OutFile | 动作 |
|---|---|---|---|
| 14, 22, 07, 24, 15, 16, 11, 100, 10, 09, 20, 12, 90, 17, 13, 19, 26, 38, 30, 25, 50, 28, 110, 21, 40 | | | 输入 6 个记录 |
| 11, 100, 10, 09, 20, 12, 90, 17, 13, 19, 26, 38, 30, 25, 50, 28, 110, 21, 40 | 14, 22, **07**, 24, 15, 16 | | 选择 07, 输出 07, 门槛 07, 置换 11 |
| 100, 10, 09, 20, 12, 90, 17, 13, 19, 26, 38, 30, 25, 50, 28, 110, 21, 40 | 14, 22, **11**, 24, 15, 16 | 07 | 选择 11, 输出 11, 门槛 11, 置换 100 |
| 10, 09, 20, 12, 90, 17, 13, 19, 26, 38, 30, 25, 50, 28, 110, 21, 40 | **14**, 22, 100, 24, 15, 16 | 07, 100 | 选择 14, 输出 14, 门槛 14, 置换 10 |
| 09, 20, 12, 90, 17, 13, 19, 26, 38, 30, 25, 50, 28, 110, 21, 40 | 10, 22, 100, 24, **15**, 16 | 07, 100, 14 | 选择 15, 输出 15, 门槛 15, 置换 09 |
| 20, 12, 90, 17, 13, 19, 26, 38, 30, 25, 50, 28, 110, 21, 40 | 10, 22, 100, 24, 09, **16** | 07, 100, 14, 15 | 选择 16, 输出 16, 门槛 16, 置换 20 |
| 12, 90, 17, 13, 19, 26, 38, 30, 25, 50, 28, 110, 21, 40 | 10, 22, 100, 24, 09, **20** | 07, 100, 14, 15, 16 | 选择 20, 输出 20, 门槛 20, 置换 12 |
| 90, 17, 13, 19, 26, 38, 30, 25, 50, 28, 110, 21, 40 | 10, **22**, 100, 24, 09, 12 | 07, 100, 14, 15, 16, 20 | 选择 22, 输出 22, 门槛 22, 置换 90 |
| 17, 13, 19, 26, 38, 30, 25, 50, 28, 110, 21, 40 | 10, 90, 100, **24**, 09, 12 | 07, 100, 14, 15, 16, 20, 22 | 选择 24, 输出 24, 门槛 24, 置换 17 |
| 13, 19, 26, 38, 30, 25, 50, 28, 110, 21, 40 | 10, **90**, 100, 17, 09, 12 | 07, 100, 14, 15, 16, 20, 22, 24 | 选择 90, 输出 90, 门槛 90, 置换 13 |
| 19, 26, 38, 30, 25, 50, 28, 110, 21, 40 | 10, 13, **100**, 17, 09, 12 | 07, 100, 14, 15, 16, 20, 22, 24, 90 | 选择 100, 输出 100, 门槛 100, 置换 19 |
| 26, 38, 30, 25, 50, 28, 110, 21, 40 | 10, 13, 19, 17, 09, 12 | 07, 100, 14, 15, 16, 20, 22, 24, 90, 100, ∞ | 无大于门槛的的记录, 输出段结束符 |
| 26, 38, 30, 25, 50, 28, 110, 21, 40 | 10, 13, 19, 17, **09**, 12 | | 选择 09, 输出 09, 门槛 09, 置换 26 |
| 38, 30, 25, 50, 28, 110, 21, 40 | **10**, 13, 19, 17, 26, 12 | 09 | 选择 10, 输出 10, 门槛 10, 置换 38 |
| 30, 25, 50, 28, 110, 21, 40 | 38, 13, 19, 17, 26, **12** | 09, 10 | 选择 12, 输出 12, 门槛 12, 置换 30 |
| 25, 50, 28, 110, 21, 40 | 38, **13**, 19, 17, 26, 30 | 09, 10, 12 | 选择 13, 输出 13, 门槛 13, 置换 25 |
| 50, 28, 110, 21, 40 | 38, 25, 19, **17**, 26, 30 | 09, 10, 12, 13 | 选择 17, 输出 17, 门槛 17, 置换 50 |
| 28, 110, 21, 40 | 38, 25, **19**, 50, 26, 30 | 09, 10, 12, 13, 17 | 选择 19, 输出 19, 门槛 19, 置换 28 |
| 110, 21, 40 | 38, **25**, 28, 50, 26, 30 | 09, 10, 12, 13, 17, 19 | 选择 25, 输出 25, 门槛 25, 置换 110 |
| 21, 40 | 38, 110, 28, | 09, 10, 12, 13, 17, 19, 25 | 选择 26, 输出 26, |

| | | | |
|---|---|---|---|
| | 50, **26**, 30 | | 门槛 26, 置换 21 |
| 40 | 38, 110, **28**, 50, 21, 30 | 09, 10, 12, 13, 17, 19, 25, 26 | 选择 28, 输出 28, 门槛 28, 置换 40 |
| | 38, 110, 40, 50, 21, **30** | 09, 10, 12, 13, 17, 19, 25, 26, 28 | 选择 30, 输出 30, 门槛 30, 无输入 |
| | **38**, 110, 40, 50, 21, ∞ | 09, 10, 12, 13, 17, 19, 25, 26, 28, 30 | 选择 38, 输出 38, 门槛 38, 无输入 |
| | —, 110, **40**, 50, 21, — | 09, 10, 12, 13, 17, 19, 25, 26, 28, 30, 38 | 选择 40, 输出 40, 门槛 40, 无输入 |
| | —, 110, —, **50**, 21, — | 09, 10, 12, 13, 17, 19, 25, 26, 28, 30, 38, 40 | 选择 50, 输出 50, 门槛 50, 无输入 |
| | —, **110**, —, —, 21, — | 09, 10, 12, 13, 17, 19, 25, 26, 28, 30, 38, 40, 50 | 选择 110, 输出 110, 门槛 110, 无输入 |
| | —, —, —, —, 21, — | 09, 10, 12, 13, 17, 19, 25, 26, 28, 30, 38, 40, 50, 110, ∞ | 无大于门槛的的记录, 输出段结束符 |
| | —, —, —, —, **21**, — | | 选择 21, 输出 21, 门槛 21, 无输入 |
| | —, —, —, —, —, — | 21, ∞ | 无大于门槛的的记录, 输出段结束符 |

9–27 给出 12 个初始归并段，其长度分别为 30, 44, 8, 6, 3, 20, 60, 18, 9, 62, 68, 85。现要做 4 路外归并排序，试画出表示归并过程的最佳归并树，并计算该归并树的带权路径长度 WPL。

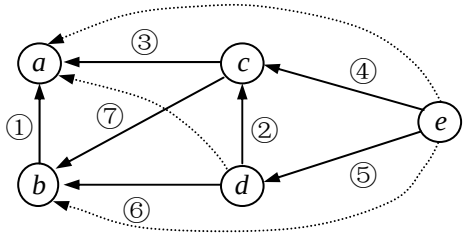
【解答】设初始归并段个数 n = 12，外归并路数 k = 4，计算(n−1) % (k−1) = 11 % 3 = 2 ≠ 0，必须补 k−2−1 = 1 个长度为 0 的空归并段，才能构造 k 路归并树。此时，归并树的内结点应有(n−1+1)/(k−1) = 12/3 = 4 个。




WPL = (3+6+8)*3 + (9+18+20+30+44+60+62)*2 + (68+85)*1 = 51 + 486 + 153 = 690

9–28 试构造排序 5 个整数最多用 7 次比较的算法。

【解答】算法的思想可以用如下的有向图来描述：




| 顶点 | *a* | *b* | *c* | *d* | *e* |
|---|---|---|---|---|---|
| 入度 | 4 | 3 | 2 | 1 | 0 |

在图中有 5 个顶点，代表 5 个可比较的整数 *a, b, c, d, e*。有向边的箭头从较大的整数指向较小的整数，虚线表示的有向边表示不用比较，而是通过传递性得到的。图中各有向边的编号给出 7 次比较的先后次序。

首先比较 *a* 与 *b* 和 *c* 与 *d*，得 $a < b, c < d$，这需要 2 次比较。然后比较 *a* 与 *c*，得 $a < c$，从而可得 $a < c < d$，这需要 3 次比较。

再比较 *c* 与 *e* 和 *d* 与 *e*，得 $c < e, d < e$，从而可得 $a < c < d < e$。最后 2 次比较，将 *b* 插入到 a 与 c 之间，得 $a < b < c < d < e$。

9–29 下面的程序是一个的两路归并算法 merge，只需要一个附加存储。设算法中参加归并的两个归并段是 A[left]~A[mid] 和 A[mid]~A[right]，归并后结果归并段放在原地。

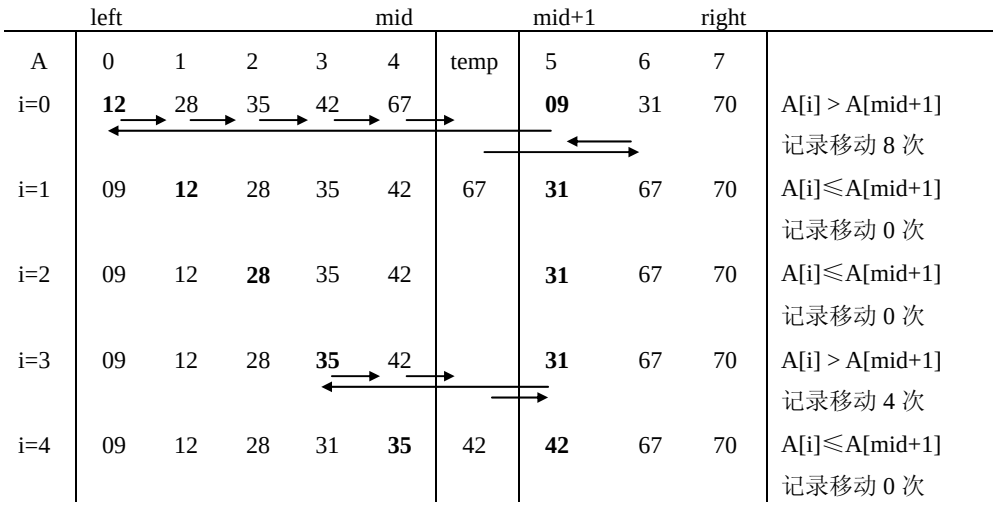
```
template<Type> void dataList<Type> :: merge( const int left, const int mid, const int right ) {
  int i, j;   Type temp;
  for ( i = left; i <= mid; i++ ) {
    if ( A[i] > A[mid+1] ) {
      temp = A[mid];
      for ( j = mod−1; j >= i; j−− ) A[j+1] = A[j];
      A[i] = A[mid+1];
      if ( temp <= A[mid+2] ) A[mid+1] = temp;
      else {
        for ( j = mid+2; j <= right; j++ )
          if ( temp > A[j] ) A[j−1] = A[j];
          else { A[j−1] = temp;   break; }
      }
    }
  }
}
```

(1) 若 A = { 12, 28, 35, 42, 67, 9, 31, 70 }, left = 0, mid = 4, right = 7。写出每次执行算法最外层循环后数组的变化。

(2) 试就一般情况 A[n]、left、mid 和 right，分析此算法的性能。

【解答】

(1) 数组 A 每次执行最外层循环后数组的变化如下：




| | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|
| A | 0 | 1 | 2 | 3 | 4 | temp | 5 | 6 | 7 | |
| i=0 | **12** | 28 | 35 | 42 | 67 | | **09** | 31 | 70 | A[i] > A[mid+1] |
| | | | | | | | | | | 记录移动 8 次 |
| i=1 | 09 | **12** | 28 | 35 | 42 | 67 | **31** | 67 | 70 | A[i]≤A[mid+1] |
| | | | | | | | | | | 记录移动 0 次 |
| i=2 | 09 | 12 | **28** | 35 | 42 | | **31** | 67 | 70 | A[i]≤A[mid+1] |
| | | | | | | | | | | 记录移动 0 次 |
| i=3 | 09 | 12 | 28 | **35** | 42 | | **31** | 67 | 70 | A[i] > A[mid+1] |
| | | | | | | | | | | 记录移动 4 次 |
| i=4 | 09 | 12 | 28 | 31 | **35** | 42 | **42** | 67 | 70 | A[i]≤A[mid+1] |
| | | | | | | | | | | 记录移动 0 次 |

(2) 本算法的记录比较次数和移动次数与待排序记录序列的初始排列有关。因此，性能分析需要讨论最好情况、最坏情况。

最好情况，例如参加排序的后一个有序表中所有记录(从 mid+1 到 right)的排序码均大于前一个有序表(从 left 到 mid)的排序码。此时，记录排序码的比较次数为 mid−left+1，与前一个有序表的长度相同，记录的移动次数为 0。

最坏情况，例如参加排序的后一个有序表中所有记录(从 mid+1 到 right)的排序码均小于前一个有序表(从 left 到 mid)的排序码，并且前一个表中有 n = mid−left+1 个元素，后一个表中有 m = right−mid 个元素，那么，估计记录排序码比较次数约为 n+m+m(m+4)/8，记录移动次数约为 (n(n+1)+m(m+1))/2。

9−30 当记录对象存放在数据表中时，进行排序时会移动许多记录对象，降低了排序的效率。为避免记录的移动，使用静态链表进行排序。在排序过程中，不移动记录对象，只修改链接指针。例如对如下的静态链表(图中只显示了排序码)进行排序：（V[0].link 是表头指针）

初始配置

| | V[0] | V[1] | V[2] | V[3] | V[4] | V[5] | V[6] | V[7] | V[8] |
|---|---|---|---|---|---|---|---|---|---|
| data | | 49 | 65 | 38 | 27 | 97 | 13 | 76 | 49* |
| link | 1 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 |

在排序结束后，各记录对象的排序顺序由各记录对象的 link 指针指示。V[0].link 指示排序码最小的记录对象，而排序码最大的记录对象的 link 指针为 0。

排序结果

| | V[0] | V[1] | V[2] | V[3] | V[4] | V[5] | V[6] | V[7] | V[8] |
|---|---|---|---|---|---|---|---|---|---|
| data | | 49 | 65 | 38 | 27 | 97 | 13 | 76 | 49* |
| link | 6 | 8 | 7 | 1 | 3 | 0 | 4 | 5 | 2 |

最后可以根据需要按排序码大小从小到大重排记录的物理位置。试设计一个算法，实现这种记录的重排。

【解答】

重排记录的基本思想是：从 i＝1 起，检查在排序之后应该是第 i 个记录的记录是否正好在第 i 个记录位置。

| i=1 | V[0] | V[1] | V[2] | V[3] | V[4] | V[5] | V[6] | V[7] | V[8] | head | temp |
|---|---|---|---|---|---|---|---|---|---|---|---|
| data | | **49** | 65 | 38 | 27 | 97 | **13** | 76 | 49* | | 13 |
| link | 6 | **8** | 7 | 1 | 3 | 0 | **4** | 5 | 2 | **6** | 4 |

当 i = 1 时，第 1 个记录不是具有最小排序码的记录，具有最小排序码的记录地址在 head = Vector[0].link = 6。交换 Vector[head]与 Vector[i]并将原位置 head 记入 Vector[i].link。此时，

在 temp.link 中存有下一个具次小排序码记录的地址 4，记入 head。再处理 i = 2 的情形。

| i=2 | V[0] | V[1] | V[2] | V[3] | V[4] | V[5] | V[6] | V[7] | V[8] | head | Temp |
|---|---|---|---|---|---|---|---|---|---|---|---|
| data | | 13 | **65** | 38 | **27** | 97 | 49 | 76 | 49* | | 13 |
| link | 6 | 6 | **7** | 1 | **3** | 0 | 8 | 5 | 2 | 4 | 4 |

当 i = 2 时，第 2 个记录不是具有次小排序码的记录，具有次最小排序码的记录地址在 head = 4。交换 Vector[head]与 Vector[i] 并将原位置 head = 4 记入 Vector[i].link。此时，在 temp.link 中存有下一个具次小排序码记录的地址 3，记入 head。再处理 i = 3 的情形。

| i=3 | V[0] | V[1] | V[2] | V[3] | V[4] | V[5] | V[6] | V[7] | V[8] | head | temp |
|---|---|---|---|---|---|---|---|---|---|---|---|
| data | | 13 | 27 | **38** | 65 | 97 | 49 | 76 | 49* | | 27 |
| link | 6 | 6 | 4 | **1** | 7 | 0 | 8 | 5 | 2 | 3 | 3 |

当 i = 3 时，第 3 个记录正应排在此位置(head == i)，原地交换并将位置 head = 3 记入 Vector[i]. link。此时，在 temp.link 中存有下一个具次小排序码记录的地址 head = 1，当下一次处理 i = 4 的情形时，head < i，表明它已处理过，但在 Vector[head].link 中记有原来此位置的记录交换到的新位置 6，令 head = Vector[head].link = 6。再处理 i = 4 的情形。

| i=4 | V[0] | V[1] | V[2] | V[3] | V[4] | V[5] | V[6] | V[7] | V[8] | head | Temp |
|---|---|---|---|---|---|---|---|---|---|---|---|
| data | | 13 | 27 | 38 | **65** | 97 | **49** | 76 | 49* | | 38 |
| link | 6 | 6 | 4 | 3 | **7** | 0 | **8** | 5 | 2 | 1,6 | 1 |

当 i = 4 时，交换 Vector[head]与 Vector[i]，并将位置 head = 6 记入 Vector[i].link。此时，在 temp.link 中存有下一个具次小排序码记录的地址 head = 8，再处理 i = 5 的情形

| i=4 | V[0] | V[1] | V[2] | V[3] | V[4] | V[5] | V[6] | V[7] | V[8] | head | temp |
|---|---|---|---|---|---|---|---|---|---|---|---|
| data | | 13 | 27 | 38 | 49 | **97** | 65 | 76 | **49*** | | 49 |
| link | 6 | 6 | 4 | 3 | 6 | **0** | 7 | 5 | **2** | 8 | 8 |

当 i = 5 时，交换 Vector[head]与 Vector[i]，并将位置 head = 8 记入 Vector[i].link。此时，在 temp.link 中存有下一个具次小排序码记录的地址 head = 8，再处理 i = 6 的情形

| i=4 | V[0] | V[1] | V[2] | V[3] | V[4] | V[5] | V[6] | V[7] | V[8] | head | temp |
|---|---|---|---|---|---|---|---|---|---|---|---|
| data | | 13 | 27 | 38 | 49 | 49* | **65** | 76 | 97 | | 49* |
| link | 6 | 6 | 4 | 3 | 6 | 8 | **7** | 5 | 0 | 2 | 2 |

当 i = 6 时，应存放于此的记录不是 Vector[head]，因为 head = 2 时的 Vector[2]已处理过，通过 head = Vector[head].link = 4，此位置记录也已处理过，再求 head = Vector[head].link = 6 ≮ i, 原在此位置的记录还应在此位置(head == i)，原地交换并将位置 head = 6 记入 Vector[i]. link。此时，在 temp.link 中存有下一个具次小排序码记录的地址 head = 7。再处理 i = 7 的情形。

| i=4 | V[0] | V[1] | V[2] | V[3] | V[4] | V[5] | V[6] | V[7] | V[8] | head | temp |
|---|---|---|---|---|---|---|---|---|---|---|---|
| data | | 13 | 27 | 38 | 49 | 49* | 65 | **76** | 97 | | 65 |
| link | 6 | 6 | 4 | 3 | 6 | 8 | 6 | **5** | 0 | 7 | 7 |

当 i = 7 时，第 7 个记录正应排在此位置(head == i)，原地交换并将位置 head = 7 记入 Vector[i]. link。此时，在 temp.link 中存有下一个具次小排序码记录的地址 head = 5，当下一次处理 i = 8 的情形时，head < i，表明它已处理过，但在 Vector[head].link 中记有原来此位置的记录交换到的新位置 8，令 head = Vector[head].link = 8。再处理 i = 8 的情形。

| i=4 | V[0] | V[1] | V[2] | V[3] | V[4] | V[5] | V[6] | V[7] | V[8] | Head | temp |
|---|---|---|---|---|---|---|---|---|---|---|---|
| data | | 13 | 27 | 38 | 49 | 49* | 65 | 76 | **97** | | 76 |
| link | 6 | 6 | 4 | 3 | 6 | 8 | 6 | 7 | **0** | 5 | 5 |

当 i = 8 时，第 8 个记录正应排在此位置(head == i)，原地交换并将位置 head = 8 记入 Vector[i]. link。此时，在 temp.link 中存有下一个具次小排序码记录的地址 head = 0，它符合退出循环的条件，因此退出循环，算法结束。

| i=4 | V[0] | V[1] | V[2] | V[3] | V[4] | V[5] | V[6] | V[7] | V[8] | Head | temp |
|---|---|---|---|---|---|---|---|---|---|---|---|
| data | | 13 | 27 | 38 | 49 | 49* | 65 | 76 | 97 | | 97 |
| link | 6 | 6 | 4 | 3 | 6 | 8 | 6 | 7 | 8 | 0 | 0 |

下面给出重新安排物理位置的算法：

```
template <class Type> void StaticdataList<Type> :: ReArrange ( ) {
//按照已排好序的静态链表中的链接顺序，重新排列所有记录对象，使得所有对象按链接顺序物理
//地重新排列。
    int i = 1, head = Vector[0];   Element<Type> temp;
    while ( head != 0 ) {
        temp = Vector[head];   Vector[head] = Vector[i];   Vector[i] = temp;
        Vector[i].link = head;
        head = temp.link;
        i++;
        while ( head < i && head > 0 ) head = Vector[head].link;
    }
}
```

# 第十章 索引与散列

10-1 什么是静态索引结构？什么是动态索引结构？它们各有哪些优缺点？

【解答】

静态索引结构指这种索引结构在初始创建，数据装入时就已经定型，而且在整个系统运行期间，树的结构不发生变化，只是数据在更新。动态索引结构是指在整个系统运行期间，树的结构随数据的增删及时调整，以保持最佳的搜索效率。静态索引结构的优点是结构定型，建立方法简单，存取方便；缺点是不利于更新，插入或删除时效率低。动态索引结构的优点是在插入或删除时能够自动调整索引树结构，以保持最佳的搜索效率；缺点是实现算法复杂。

10-2 设有 10000 个记录对象，通过分块划分为若干子表并建立索引，那么为了提高搜索效率，每一个子表的大小应设计为多大?

【解答】

每个子表的大小 $s = \lceil n \rceil = \lceil 10000 \rceil = 100$ 个记录对象。

10-3 如果一个磁盘页块大小为 1024 (=1K) 字节，存储的每个记录对象需要占用 16 字节，其中关键码占 4 字节，其它数据占 12 字节。所有记录均已按关键码有序地存储在磁盘文件中，每个页块的第 1 个记录用于存放线性索引。另外在内存中开辟了 256K 字节的空间可用于存放线性索引。试问：

(1) 若将线性索引常驻内存，文件中最多可以存放多少个记录？(每个索引项 8 字节，其中关键码 4 字节，地址 4 字节)

(2) 如果使用二级索引，第二级索引占用 1024 字节（有 128 个索引项），这时文件中最多可以存放多少个记录？

【解答】

(1) 因为一个磁盘页块大小为 1024 字节，每个记录对象需要占用 16 字节，则每个页块可存放 1024 / 16 = 64 个记录，除第一个记录存储线性索引外，每个页块可存储 63 个记录对象。又因为在磁盘文件中所有记录对象按关键码有序存储，所以线性索引可以是稀疏索引，每一个索引项存放一个页块的最大关键码及该页块的地址。若线性索引常驻内存，那么它最多可存放 256 * (1024 / 8 ) = 256 * 128 = 32768 个索引项，文件中可存放 32768 * 63 = 2064384 个记录对象。

(2) 由于第二级索引占用 1024 个字节，内存中还剩 255K 字节用于第一级索引。第一级索引有 255 * 128 = 32640 个索引项，作为稀疏索引，每个索引项索引一个页块，则索引文件中可存放 32640 * 63 = 2056320。

10-4 假设在数据库文件中的每一个记录是由占 2 个字节的整型数关键码和一个变长的数据字段组成。数据字段都是字符串。为了存放右面的那些记录，应如何组织线性索引？

| | |
|---|---|
| 397 | Hello World! |
| 82 | XYZ |
| 1038 | This string is rather long |
| 1037 | This is Shorter |
| 42 | ABC |
| 2222 | Hello new World! |

【解答】

将所有字符串依加入的先后次序存放于一个连续的存储空间 store 中，这个空间也叫做“堆”，它是存放所有字符串的顺序文件。它有一个指针 free，指示在堆 store 中当前可存放数据的开始地址。初始时 free 置为 0，表示可从文件的 0 号位置开始存放。线性索引中每个索引项给出记录关键码，字符串在 store 中的起始地址和字符串的长度：

索引表 ID

| 关键码 | 串长度 | 串起始地址 |
|---|---|---|
| 42 | 3 | 56 |
| 82 | 3 | 12 |
| 397 | 12 | 0 |
| 1037 | 15 | 41 |
| 1038 | 26 | 15 |
| 2222 | 16 | 59 |

堆 store

| | |
|---|---|
| 0 | Hello World! XYZ This string is rather long This |
| | is Shorter ABC Hello new World! |
| | free |

10-5 设有一个职工文件：

| 记录地址 | 职工号 | 姓　名 | 性别 | 职　业 | 年龄 | 籍贯 | 月工资(元) |
|---|---|---|---|---|---|---|---|
| 10032 | 034 | 刘激扬 | 男 | 教　师 | 29 | 山东 | 720.00 |
| 10068 | 064 | 蔡晓莉 | 女 | 教　师 | 32 | 辽宁 | 1200.00 |
| 10104 | 073 | 朱　力 | 男 | 实验员 | 26 | 广东 | 480.00 |
| 10140 | 081 | 洪　伟 | 男 | 教　师 | 36 | 北京 | 1400.00 |
| 10176 | 092 | 卢声凯 | 男 | 教　师 | 28 | 湖北 | 720.00 |
| 10212 | 123 | 林德康 | 男 | 行政秘书 | 33 | 江西 | 480.00 |
| 10248 | 140 | 熊南燕 | 女 | 教　师 | 27 | 上海 | 780.00 |
| 10284 | 175 | 吕　颖 | 女 | 实验员 | 28 | 江苏 | 480.00 |
| 10320 | 209 | 袁秋慧 | 女 | 教　师 | 24 | 广东 | 720.00 |

其中，关键码为职工号。试根据此文件，对下列查询组织主索引和倒排索引，再写出搜索结果关键码。(1) 男性职工；(2) 月工资超过 800 元的职工；(3) 月工资超过平均工资的职工；(4) 职业为实验员和行政秘书的男性职工；(5) 男性教师或者年龄超过 25 岁且职业为实验员和教师的女性职工。

【解答】

主索引

| | 职工号 | 记录地址 |
|---|---|---|
| 0 | 034 | 10032 |
| 1 | 064 | 10068 |
| 2 | 073 | 10104 |
| 3 | 081 | 10140 |
| 4 | 092 | 10176 |
| 5 | 123 | 10212 |
| 6 | 140 | 10248 |
| 7 | 175 | 10284 |
| 8 | 209 | 10320 |

月工资 倒排索引

| 月工资 | 长度 | 指针 |
|---|---|---|
| 480. | 3 | 073 |
| | | 123 |
| | | 175 |
| 720. | 3 | 034 |
| | | 092 |
| | | 209 |
| 780. | 1 | 140 |
| 1200. | 1 | 064 |
| 1400. | 1 | 081 |

职务 倒排索引

| 职务 | 长度 | 指针 |
|---|---|---|
| 教师 | 6 | 034 |
| | | 064 |
| | | 081 |
| | | 092 |
| | | 140 |
| | | 209 |
| 实验员 | 2 | 073 |
| | | 175 |
| 行政秘书 | 1 | 123 |

性别 倒排索引

| 性别 | 长度 | 指针 |
|---|---|---|
| 男 | 5 | 034 |
| | | 073 |
| | | 081 |
| | | 092 |

年龄 倒排索引

| 年龄 | 长度 | 指针 |
|---|---|---|
| 24 | 1 | 209 |
| 26 | 1 | 073 |
| 27 | 1 | 140 |
| 28 | 2 | 092 |

| | | |
|---|---|---|
| | | 123 |
| 女 | 4 | 064 |
| | | 140 |
| | | 175 |
| | | 209 |

| | | |
|---|---|---|
| | | 175 |
| 29 | 1 | 034 |
| 32 | 1 | 064 |
| 33 | 1 | 123 |
| 36 | 1 | 081 |

搜索结果：

(1) 男性职工 (搜索性别倒排索引)：**{**034, 073, 081, 092, 123**}**

(2) 月工资超过 800 元的职工 (搜索月工资倒排索引)：**{**064, 081**}**

(3) 月工资超过平均工资的职工(搜索月工资倒排索引) **{**月平均工资 776 元**}**：

**{**064, 081, 140**}**

(4) 职业为实验员和行政秘书的男性职工(搜索职务和性别倒排索引)：

**{**073, 123, 175**} && {**034, 073, 081, 092, 123**}** = **{**073, 123**}**

(5) 男性教师 (搜索性别与职务倒排索引)：

**{**034, 073, 081, 092, 123**} && {** 034, 064, 081, 092, 140, 209**} = {**034, 081, 092**}**

年龄超过 25 岁且职业为实验员和教师的女性职工 (搜索性别、职务和年龄倒排索引)：

**{**064, 140, 175, 209**} && {**034, 064, 073, 081, 092, 140, 175, 209**} && {**034, 064, 073, 081,092, 123, 140, 175**} = {**064, 140, 175**}**

10-6 倒排索引中的记录地址可以是记录的实际存放地址，也可以是记录的关键码。试比较这两种方式的优缺点。

【解答】

在倒排索引中的记录地址用记录的实际存放地址，搜索的速度快；但以后在文件中插入或删除记录对象时需要移动文件中的记录对象，从而改变记录的实际存放地址，这将对所有的索引产生影响：修改所有倒排索引的指针，不但工作量大而且容易引入新的错误或遗漏，使得系统不易维护。

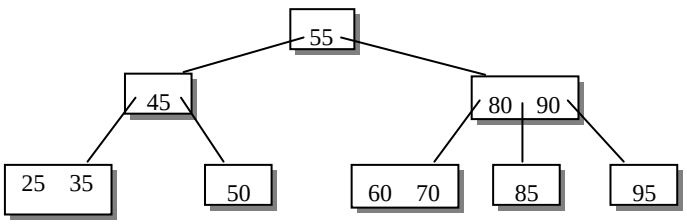
记录地址采用记录的关键码，缺点是寻找实际记录对象需要再经过主索引，降低了搜索速度；但以后在文件中插入或删除记录对象时，如果移动文件中的记录对象，导致许多记录对象的实际存放地址发生变化，只需改变主索引中的相应记录地址，其他倒排索引中的指针一律不变，使得系统容易维护，且不易产生新的错误和遗漏。

10-7 m = 2 的平衡 m 路搜索树是 AVL 树，m = 3 的平衡 m 路搜索树是 2-3 树。它们的叶结点必须在同一层吗？m 阶 B 树是平衡 m 路搜索树，反过来，平衡 m 路搜索树一定是 B 树吗？为什么？

【解答】

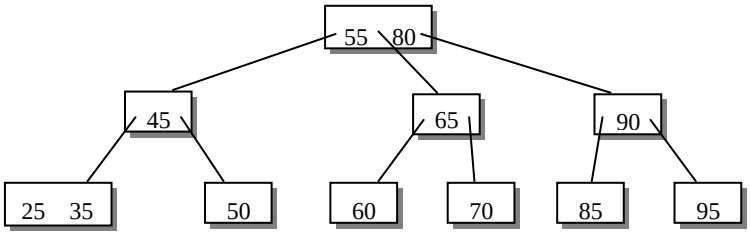
m = 3 的平衡 m 路搜索树的叶结点不一定在同一层，而 m 阶 B_树的叶结点必须在同一层，所以 m 阶 B_树是平衡 m 路搜索树，反过来，平衡 m 路搜索树不一定是 B_树。

10-8 下图是一个 3 阶 B 树。试分别画出在插入 65、15、40、30 之后 B 树的变化。

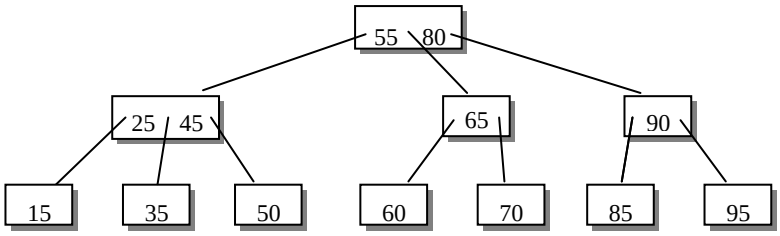



【解答】

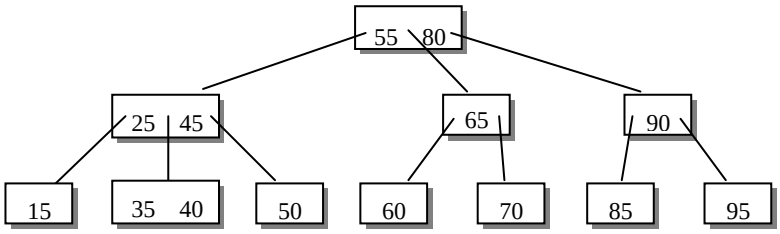
插入 65 后：



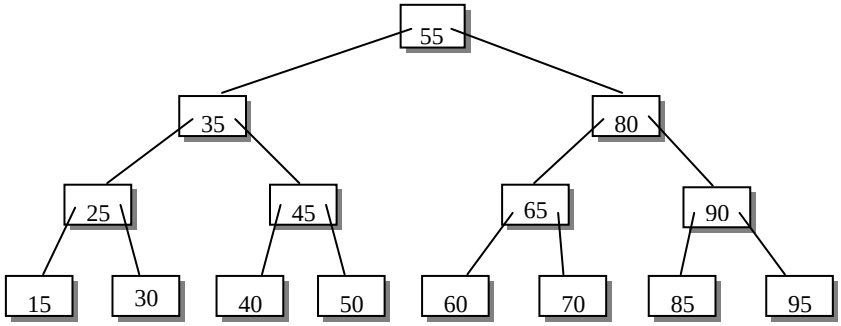

插入 15 后：



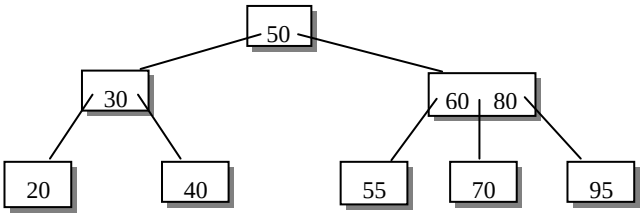

插入 40 后：




插入 30 后：

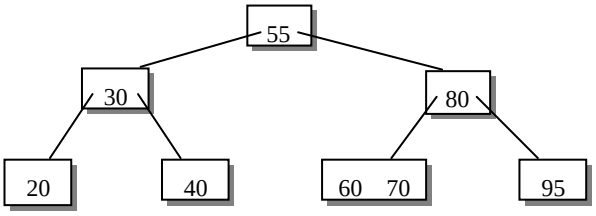



10–9 下图是一个 3 阶 B 树。试分别画出在删除 50、40 之后 B 树的变化。

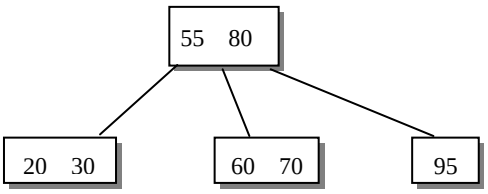



【解答】

删除 50 后：



删除 40 后：




10-10 对于一棵有 1999999 个关键码的 199 阶 B 树，试估计其最大层数(不包括失败结点)及最小层数(不包括失败结点)。

【解答】

设 B 树的阶数 m = 199，则⌈m/2⌉ = 100。若不包括失败结点层，则其最大层数为

$\lfloor \log_{\lceil m/2 \rceil}((N+1)/2) \rfloor + 1 = \lfloor \log_{100} 1000000 \rfloor + 1 = 4$

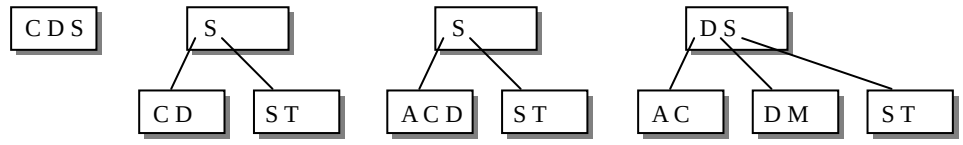
若使得每一层关键码数达到最大，可使其层数达到最小。第 0 层最多有(m-1)个关键码，第 1 层最多有$(m-1)^2$个关键码，…，第 h-1 层最多有$(m-1)^h$个关键码。层数为 h 的 B 树最多有$(m-1) + (m-1)^2 + \cdots + (m-1)^{h-1} = (m-1)\,((m-1)^h - 1)\,/\,(m-2)$个关键码。反之，若有 n 个关键码，$n \leqslant (m-1)\,((m-1)^h - 1)\,/\,(m-2)$，则 $h \geqslant \log_{(m-1)}(n(m-2)/(m-1)+1)$，所以，有 1999999 个关键码的 199 阶 B 树的最小层数为

$\lceil \log_{(m-1)}(n*(m-2)/(m-1)+1) \rceil = \lceil \log_{198}(1999999*197/198+1) \rceil = \lceil \log_{198} 1989899 \rceil$

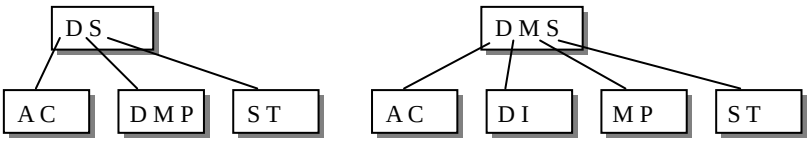
10-11 给定一组记录，其关键码为字符。记录的插入顺序为 { C, S, D, T, A, M, P, I, B, W, N, G, U, R, K, E, H, O, L, J }，给出插入这些记录后的 4 阶 B+树。假定叶结点最多可存放 3 个记录。

【解答】

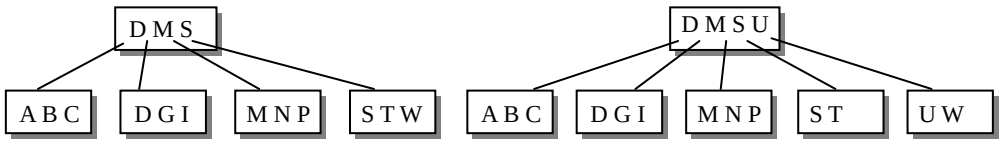
插入 C, S, D　插入 T　插入 A　插入 M



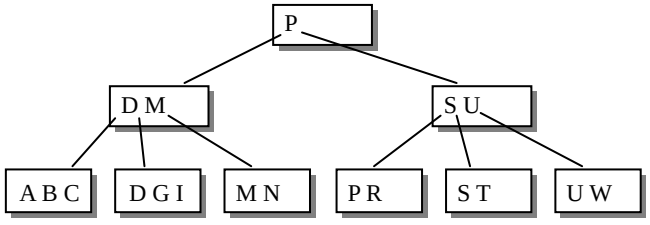

插入 P　插入 I



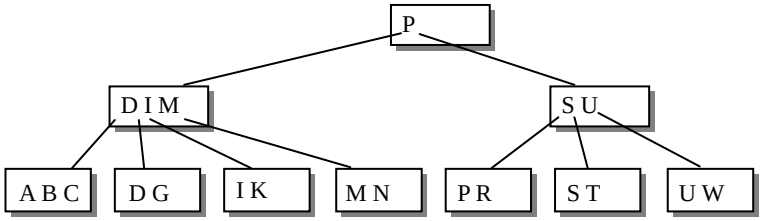

插入 B, W, N, G　插入 U



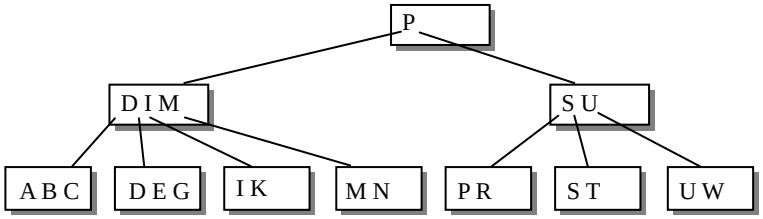

插入 R



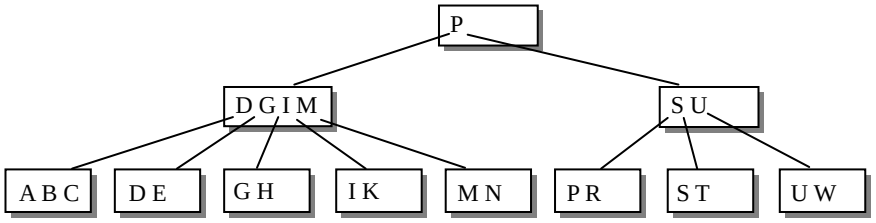

插入 K



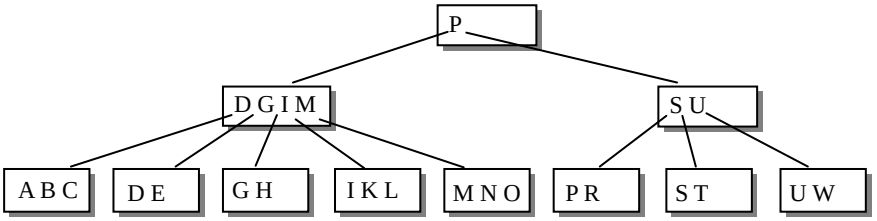

插入 E



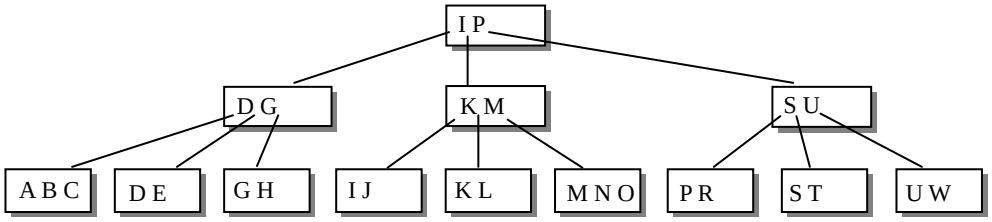

插入 H




插入 O, L




插入 J




10−12 设有一棵 B+树，其内部结点最多可存放 100 个子女，叶结点最多可存储 15 个记录。对于 1, 2, 3, 4, 5 层的 B+树，最多能存储多少记录，最少能存储多少记录。

【解答】

一层 B+树：根据 B+树定义，一层 B+树的结点只有一个，它既是根结点又是叶结点，最多可存储 m1 = 15 个记录，最少可存储 ⌈m1/2⌉= 8 个记录。

二层 B+树：第 0 层是根结点，它最多有 m = 100 棵子树，最少有 2 个结点；第 1 层是叶结点，它最多有 m 个结点，最多可存储 m*m1 = 100*15 = 1500 个记录，最少有 2 个结点，最少可存储 2*⌈m1/2⌉= 16 个记录。

三层 B+树：第 2 层是叶结点。它最多有 $m^2$ 个结点，最多可存储 $m^2$ * m1 = 150000 个记

录。最少有 2*⌈m/2⌉= 100 个结点，最少可存储 2*⌈m/2⌉*⌈m1/2⌉= 800 个记录。

四层 B+树：第 3 层是叶结点。它最多有 $m^3$ 个结点，可存储 $m^3$ * m1 = 15000000 个记录。最少有 2*⌈m/2⌉$^2$ = 2 * $50^2$ = 5000 个结点，存储 2*⌈m/2⌉$^2$ *⌈m1/2⌉= 40000 个记录。

五层 B+树：第 4 层是叶结点。它最多有 $m^4$ 个结点，可存储 $m^4$ * m1 = 1500000000 个记录。最少有 2*⌈m/2⌉$^3$ = 2 * $50^3$ = 250000 个结点，存储 2*⌈m/2⌉$^3$ *⌈m1/2⌉= 2000000 个记录。

10-13 设散列表为 HT[13]，散列函数为 H (*key*) = *key* %13。用闭散列法解决冲突，对下列关键码序列 12, 23, 45, 57, 20, 03, 78, 31, 15, 36 造表。

(1) 采用线性探查法寻找下一个空位，画出相应的散列表，并计算等概率下搜索成功的平均搜索长度和搜索不成功的平均搜索长度。

(2) 采用双散列法寻找下一个空位，再散列函数为 RH (*key*) = (7**key*) % 10 + 1，寻找下一个空位的公式为 $H_i = (H_{i-1} + RH(key)) \% 13$, $H_1 = H(key)$。画出相应的散列表，并计算等概率下搜索成功的平均搜索长度。

【解答】

使用散列函数 H(*key*) = *key* **mod** 13，有

H(12) = 12,　H(23) = 10,　H(45) = 6,　H(57) = 5,
H(20) = 7,　H(03) = 3,　H(78) = 0,　H(31) = 5,
H(15) = 2,　H(36) = 10.

(1) 利用线性探查法造表：

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 78 | | 15 | 03 | | 57 | 45 | 20 | 31 | | 23 | 36 | 12 |
| (1) | | (1) | (1) | | (1) | (1) | (1) | (4) | | (1) | (2) | (1) |

搜索成功的平均搜索长度为

$$ASL_{succ} = \frac{1}{10}(1 + 1 + 1 + 1 + 1 + 1 + 4 + 1 + 2 + 1) = \frac{14}{10}$$

搜索不成功的平均搜索长度为

$$ASL_{unsucc} = \frac{1}{13}(2 + 1 + 3 + 2 + 1 + 5 + 4 + 3 + 2 + 1 + 5 + 4 + 3) = \frac{36}{13}$$

(2) 利用双散列法造表：

$H_i = (H_{i-1} + RH(key)) \% 13$,　$H_1 = H(key)$

| 0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | 11 | 12 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 78 | | 15 | 03 | | 57 | 45 | 20 | 31 | 36 | 23 | | 12 |
| (1) | | (1) | (1) | | (1) | (1) | (1) | (3) | (5) | (1) | | (1) |

搜索成功的平均搜索长度为

$$ASL_{succ} = \frac{1}{10}(1 + 1 + 1 + 1 + 1 + 1 + 3 + 5 + 1 + 1) = \frac{16}{10}$$

10-14 设有 150 个记录要存储到散列表中，要求利用线性探查法解决冲突，同时要求找到所需记录的平均比较次数不超过 2 次。试问散列表需要设计多大？ 设$\alpha$是散列表的装载因子，则有

$$ASL_{succ} = \frac{1}{2}(1 + \frac{1}{1-\alpha})$$

【解答】

已知要存储的记录数为 n = 150，查找成功的平均查找长度为 $ASL_{succ} \le 2$，则有 $ASL_{succ} = \frac{1}{2}\left(1+\frac{1}{1-\alpha}\right) \le 2$，解得 $\alpha \le \frac{2}{3}$。又有$\alpha = \frac{n}{m} = \frac{150}{m} \le \frac{2}{3}$，则 $m \ge 225$。

10−15 若设散列表的大小为 m，利用散列函数计算出的散列地址为 h = hash(x)。

(1) 试证明：如果二次探查的顺序为$(h + q^2), (h + (q-1)^2), \cdots, (h+1), h, (h-1), \cdots, (h-q^2)$，其中， q = (m−1)/2。因此在相继被探查的两个桶之间地址相减所得的差取模(%m)的结果为

m−2, m−4, m−6, …, 5, 3, 1, 1, 3, 5, …, m−6, m−4, m−2

(2) 编写一个算法，使用课文中讨论的散列函数 h(x)和二次探查解决冲突的方法，按给定值 x 来搜索一个大小为 m 的散列表。如果 x 不在表中，则将它插入到表中。

【解答】

(1) 将探查序列分两部分讨论：

$(h + q^2), (h + (q-1)^2), \cdots, (h+1), h$ 和 $(h-1), (h-2^2), \cdots, (h-q^2)$。

对于前一部分，设其通项为 $h + ( q - d )^2$, d = 0, 1, …, q，则相邻两个桶之间地址相减所得的差取模：

$( h + (q - (d-1) )^2 - ( h + (q - d )^2 ) ) \% m = ( (q - (d-1) )^2 - (q - d )^2 ) \% m$

$= (2*q - 2*d + 1) \% m = ( m - 2*d ) \% m.$　　( 代换 q = (m−1)/2 )

代入 d = 1, 2, …, q，则可得到探查序列如下：

m−2, m−4, m−6, …, 5, 3, 1。 ( m – 2*q = m – 2* (m−1)/2 = 1 )

对于后一部分，其通项为 $h - ( q - d )^2$, d = q, q+1, …, 2q，则相邻两个桶之间地址相减所得的差取模：

$( h - ( q - d )^2 - ( h - ( q - (d+1) )^2 ) ) \% m = ( ( q - (d+1)^2 - (q - d )^2 ) \% m$

$= ( 2*d - 2*q + 1) \% m = ( 2*d - m + 2) \% m$　　( 代换 q = (m−1)/2 )

代入 d = q, q+1, …, 2q−1，则可得到

2*d–m+2 = 2*q – m +2 = m – 1 – m +2 = 1,

2*d–m+2 = 2*q + 2 – m +2 = m – 1 + 2 – m +2 = 3, ……,

2*d–m+2 = 2*(2*q−1) – m +2 = 2*(m–1–1) – m + 2 = 2*m – 4 – m +2 = m – 2。〖证毕〗

(2) 编写算法

下面是使用二次探查法处理溢出时的散列表类的声明。

```
template <class Type> class HashTable {                    //散列表类的定义
public:
    enum KindOfEntry { Active, Empty, Deleted };           //表项分类 (活动 / 空 / 删)
    HashTable ( ) : TableSize ( DefaultSize ) { AllocateHt ( ); CurrentSize = 0; }    //构造函数
    ~HashTable ( ) { delete [ ] ht; }                      //析构函数
    const HashTable & operator = ( const HashTable & ht2 ); //重载函数：表赋值
    int Find ( const Type & x );                           //在散列表中搜索 x
    int IsEmpty ( ) { return !CurrentSize ? 1 : 0; }       //判散列表空否，空则返回 1
private:
    struct HashEntry {                                     //散列表的表项定义
        Type Element;                                      //表项的数据, 即表项的关键码
        KindOfEntry info;                                  //三种状态: Active, Empty, Deleted
        HashEntry ( ) : info (Empty ) { }                  //表项构造函数
        HashEntry ( const Type &E, KindOfEntry i = Empty ) : Element (E), info (i) { }
    };
```

```
    enum { DefualtSize = 31; }
    HashEntry *ht;                          //散列表存储数组
    int TableSize;                          //数组长度，要求是满足 4k+3 的质数，k 是整数
    int CurrentSize;                        //已占据散列地址数目
    void AllocateHt ( ) { ht = new HashEntry[TableSize ]; }     //为散列表分配存储空间;
    int FindPos ( const Type & x );         //散列函数
};

template <class Type> const HashTable<Type> & HashTable<Type> ::
operator = ( const HashTable<Type> &ht2 ) {
//重载函数：复制一个散列表 ht2
    if ( this != &ht2 ) {
        delete [ ] ht;   TableSize = ht2.TableSize;   AllocateHt ( );   //重新分配目标散列表存储空间
        for ( int i = 0; i < TableSize; i++ ) ht[i] = ht2.ht[i];        //从源散列表向目标散列表传送
        CurrentSize = ht2.CurrentSize;                                  //传送当前表项个数
    }
    return *this;                                                       //返回目标散列表结构指针
}

template <class Type> int HashTable<Type> :: Find ( const Type& x ) {
//共有函数：  找下一散列位置的函数
    int i = 0, q = ( TableSize −1 ) / 2, h0;            // i 为探查次数
    int CurrentPos = h0 = HashPos ( x );                //利用散列函数计算 x 的散列地址
    while ( ht[CurrentPos].info != Empty && ht[CurrentPos].Element != x ) {
                                                        /搜索是否要求表项
        if ( i <= q ) {                                 //求“下一个”桶
            CurrentPos = h0 + (q − i ) * ( q − i );
            while ( CurrentPos >= TableSize ) CurrentPos −= TableSize;          //实现取模
        }
        else {
            CurrentPos = h0 − ( i −q ) * ( i − q );
            while ( CurrentPos < 0 ) CurrentPos += TableSize;                   //实现取模
        }
        i++;
    }
    if ( ht[CurrentPos].info == Active && ht[CurrentPos].Element == x )
        return CurrentPos;                                                      //返回桶号
    else {
        ht[CurrentPos].info = Active;   ht[CurrentPos].Element = x;             //插入 x
        if ( ++CurrentSize < TableSize / 2 ) return CurrentPos;
                                        //当前已有项数加 1, 不超过表长的一半返回
        HashEntry *Oldht = ht;          //分裂空间处理：保存原来的散列表
        int OldTableSize = TableSize;
        CurrentSize = 0;
```

```
        TableSize = NextPrime ( 2 * OldTableSize );  //原表大小的 2 倍，取质数
        Allocateht ( );                              //建立新的二倍大小的空表
        for ( i = 0; i < OldTableSize; i++)          //原来的元素重新散列到新表中
            if ( Oldht[i].info == Active ) {
                Find ( Oldht[i].Element );           //递归调用
                if ( Oldht[i].Element == x ) CurrentPos = i;
            }
        delete [ ] Oldht;
        return CurrentPos;
    }
}
```

求下一个大于参数表中所给正整数 *N* 的质数的算法。

```
int NextPrime ( int N ) {                //求下一个>N 的质数，设 N >= 5
    if ( N % 2 == 0 ) N++;               //偶数不是质数
    for ( ; !IsPrime (N); N += 2 );      //寻找质数
    return N;
}

int IsPrime ( int N ) {                  //测试 N 是否质数
    for ( int i = 3; i*i <= N; i += 2 )  //若 N 能分解为两个整数的乘积, 其中一个一定 ≤√N
        if ( N % i == 0 ) return 0;      //N 能整除 i, N 不是质数
    return 1;                            //N 是质数
}
```

10-16 编写一个算法，以字典顺序输出散列表中的所有标识符。设散列函数为 hash(x) = x 中的第一个字符，采用线性探查法来解决冲突。试估计该算法所需的时间。

【解答】

用线性探查法处理溢出时散列表的类的声明。

```
#define DefaultSize 1000
#include <iostream.h>
#include <string.h>
#include <stdlib.h>
class HashTable {                                  //散列表类定义
public:
    enum KindOfEntry { Active, Empty, Deleted };   //表项分类 (活动 / 空 / 删)
    HashTable ( ) : TableSize ( DefaultSize ) { ht = new HashEntry[TableSize]; } //构造函数
    ~HashTable ( ) { delete [ ] ht; }              //析构函数
    int Find-Ins ( const char * id );              //在散列表中搜索标识符 id
    void HashSort ( );
private:
    struct HashEntry {                             //表项定义
        Type Element;                              //表项的数据, 即表项的关键码
        KindOfEntry info;                          //三种状态: Active, Empty, Deleted
```

```
        HashEntry ( ) : info (Empty ) { }                    //表项构造函数，置空
    };
    HashEntry *ht;                                           //散列表存储数组
    int TableSize;                                           //最大桶数
    int FindPos ( string s ) const { return atoi (*s) - 32; }    //散列函数
}

int HashTable :: Find-Ins ( const char * id ) {
    int i = FindPos ( id ),   j = i;                         //i 是计算出来的散列地址
    while ( ht[j].info != Empty && strcmp ( ht[j].Element, id ) != 0 ) {     //冲突
        j = ( j + 1 ) % TableSize;                           //当做循环表处理，找下一个空桶
        if ( j == i ) return -TableSize;                     //转一圈回到开始点，表已满，失败
    }
    if ( ht[j].info != Active ) {                            //插入
      if ( j > i ) {
          while ( int k = j; k > i; k--)
              { ht[k].Element = ht[k-1].Element;   ht[k].info = ht[k-1].info; }
          ht[i].Element = id;   ht[i].info = Active;         //插入
      } else {
          HashEntry temp;
          temp.Element = ht[TableSize-1].Element;   temp.info = ht[TableSize-1].info;
          while ( int k = TableSize-1; k > i; k--)
              { ht[k].Element = ht[k-1].Element;   ht[k].info = ht[k-1].info; }
          ht[i].Element = id;   ht[i].info = Active;         //插入
          while ( int k = j; k > 0; k--)
              { ht[k].Element = ht[k-1].Element;   ht[k].info = ht[k-1].info; }
          ht[0].Element = temp.Element;   ht[0].info = temp.info;
      }
      return j;
}

void HashTable :: HashSort ( ) {
    int n, i;   char * str;
    cin >> n >> str;
    for ( i = 0; i < n; i++ ) {
      if ( Find-Ins ( str ) == - Tablesize ) { cout << "表已满" << endl;   break; }
      cin >> str;
    }
    for ( i = 0; i < TableSize; i++ )
      if ( ht[i].info == Active ) cout << ht[i].Element << endl;
}
```

10-17 设有 1000 个值在 1 到 10000 的整数，试设计一个利用散列方法的算法，以最少的数据比较次数和移动次数对它们进行排序。

【解答 1】

建立 TableSize = 10000 的散列表，散列函数定义为

```
int HashTable :: FindPos ( const int x ) { return x−1; }
```

相应排序算法基于散列表类

```
#define DefaultSize 10000
#define n 1000
class HashTable {                                          //散列表类的定义
public:
    enum KindOfEntry { Active, Empty, Deleted };           //表项分类 (活动 / 空 / 删)
    HashTable ( ) : TableSize ( DefaultSize ) { ht = new HashEntry[TableSize ]; }   //构造函数
    ~HashTable ( ) { delete [ ] ht; }                          //析构函数
    void HashSort ( int A[ ], int n );                     //散列法排序
private:
    struct HashEntry {                                     //散列表的表项定义
        int Element;                                       //表项的数据, 整数
        KindOfEntry info;                                  //三种状态: Active, Empty, Deleted
        HashEntry ( ) : info (Empty ) { }                  //表项构造函数
    };
    HashEntry *ht;                                         //散列表存储数组
    int TableSize;                                         //数组长度
    int FindPos ( int x );                                 //散列函数
};

void HashTable<Type> :: HashSort ( int A[ ], int n ) {     //散列法排序
    for ( int i = 0; i < n; i++ ) {
        int position = FindPos( A[i] );
        ht[position].info = Active;    ht[position].Element = A[i];
    }
    int pos = 0;
    for ( int i = 0; i < TableSize; i++ )
        if ( ht[i].info == Active )
          { cout << ht[i].Element << endl;    A[pos] = ht[i].Element;    pos++; }
}
```

【解答 2】

利用开散列的方法进行排序。其散列表类及散列表链结点类的定义如下：

```
#define DefaultSize 3334
#define n 1000
class HashTable;                                //散列表类的前视声明

class ListNode {                                //各桶中同义词子表的链结点(表项)定义
friend class HashTable;
private:
    int key;                                    //整数数据
```

```
    ListNode *link;                          //链指针
public:
    ListNode ( int x ) : key(x), link(NULL) { }      //构造函数
};
typedef ListNode *ListPtr;                   //链表指针

class HashTable {                            //散列表(表头指针向量)定义
public:
    HashTable( int size = DefaultSize )      //散列表的构造函数
       { TableSize = size; ht = new ListPtr[size]; }  //确定容量及创建指针数组
    void HashSort ( int A[ ]; int n )
private:
    int TableSize;                           //容量(桶的个数)
    ListPtr *ht;                             //散列表定义
    int FindPos ( int x ) { return x / 3; }
}

void HashTable<Type> :: HashSort ( int A[ ]; int n ) {
    ListPtr * p , *q;    int i, bucket, k = 0;
    for ( i = 0; i < n; i++ ) {              //对所有数据散列, 同义词子表是有序链表
      bucket = FindPos ( A[i] );             //通过一个散列函数计算桶号
      p = ht[bucket];    q = new ListNode(A[i]);
      if ( p == NULL || p->key > A[i] )      //空同义词子表或*q 的数据最小
        { q->link = ht[bucked];    ht[bucked] = q; }
      else if ( p->link == NULL || p->link->key > A[i] )
            { q->link = p->link;    p->link = q; }
          else p->link->link = q;            //同义词子表最多 3 个结点
    }
    for ( i = 0; i < TableSize; i++ ) {      //按有序次序输出
      p = ht[i];
      while ( p != NULL ) {
          cout << p->key << endl;    A[k] = p->key;    k++;
          p = p->link;
      }
    }
}
```

10-18 设有 15000 个记录需放在散列文件中，文件中每个桶内各页块采用链接方式连结，每个页块可存放 30 个记录。若采用按桶散列，且要求搜索到一个已有记录的平均读盘时间不超过 1.5 次，则该文件应设置多少个桶？

【解答】

已知用链地址法（开散列）解决冲突，搜索成功的平均搜索长度为 $1+\alpha/2 \leqslant 1.5$，解出 $\alpha \leqslant 1$，又 $\alpha = n / m = 15000 / 30 / m = 500 / m \leqslant 1$，$m \geqslant 500$。由此可知，该文件至少应设置 500 个桶。

10-19 用可扩充散列法组织文件时，若目录深度为 d，指向某个页块的指针有 n 个，则该页块的局部深度有多大？

【解答】

设页块的局部深度为 d'，根据题意有 $n = 2^{d-d'}$，因此，$d' = d - \log_2 n$。

10-20 设一组对象的关键码为 { 69, 115, 110, 255, 185, 143, 208, 96, 63, 175, 160, 99, 171, 137, 149, 229, 167, 121, 204, 52, 127, 57, 1040 }。要求用散列函数将这些对象的关键码转换成二进制地址，存入用可扩充散列法组织的文件里。定义散列函数为 hash(key) = key % 64, 二进制地址取 6 位。设每个页块可容纳 4 个对象。要求按 10 .4 节介绍的方法设置目录表的初始状态，使目录表的深度为 3。然后按题中所给的顺序，将各个对象插入的可扩充散列文件中。试画出每次页块分裂或目录扩充时的状态和文件的最后状态。

【解答】

$hash(69) = 5_{(10)} = 000101_{(2)}$　　$hash(115) = 51_{(10)} = 110011_{(2)}$

$hash(110) = 46_{(10)} = 101110_{(2)}$　　$hash(255) = 63_{(10)} = 111111_{(2)}$

$hash(185) = 57_{(10)} = 111001_{(2)}$　　$hash(143) = 15_{(10)} = 001111_{(2)}$

$hash(208) = 16_{(10)} = 010000_{(2)}$　　$hash(96) = 32_{(10)} = 100000_{(2)}$

$hash(63) = 63_{(10)} = 111111_{(2)}$　　$hash(175) = 47_{(10)} = 101111_{(2)}$

$hash(160) = 32_{(10)} = 100000_{(2)}$　　$hash(99) = 35_{(10)} = 100011_{(2)}$

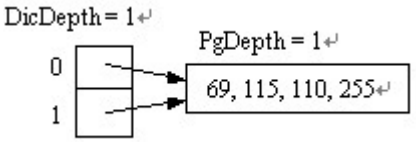
$hash(171) = 43_{(10)} = 101011_{(2)}$　　$hash(137) = 9_{(10)} = 001001_{(2)}$

$hash(149) = 21_{(10)} = 010101_{(2)}$　　$hash(229) = 37_{(10)} = 100101_{(2)}$

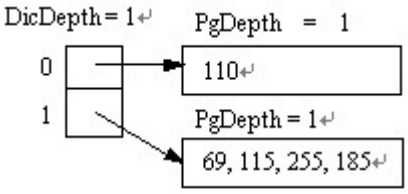
$hash(167) = 39_{(10)} = 101001_{(2)}$　　$hash(121) = 57_{(10)} = 111001_{(2)}$

$hash(204) = 12_{(10)} = 001100_{(2)}$　　$hash(52) = 52_{(10)} = 110100_{(2)}$

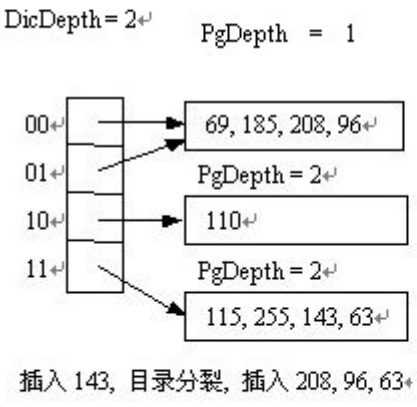
根据题意，每个页块可容纳 4 个对象，为画图清晰起见仅给出前 20 个关键码插入后的结果。目录表的深度 d = 3。



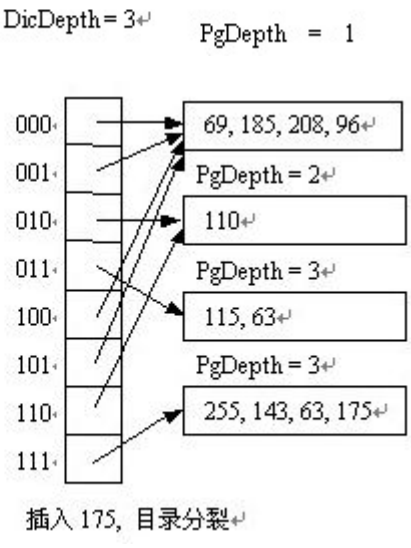

初态，插入 69, 115, 110, 255



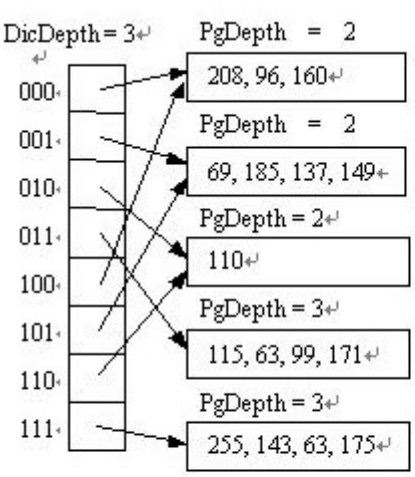

插入 185，页块分裂



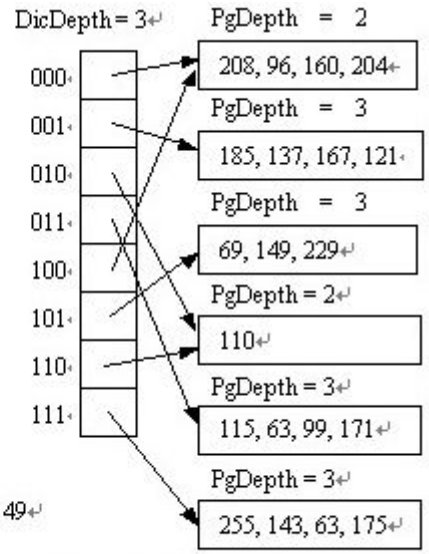

插入 143，目录分裂，插入 208, 96, 63



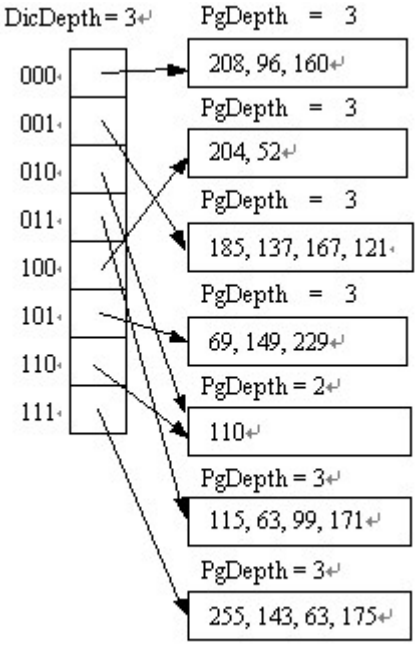

插入 175，目录分裂




插入 160，页块分裂，插入 99, 171, 137, 149




插入 229，页块分裂，插入 167, 121, 204




插入 52，页块分裂

【后记】

编写此资料的目的有两个：第一期望师大学子能通过课程学习，切实打好基础，更希望非计算机专业的同学，特别是那些以自学形式在学习该课程的同学，该学习材料肯定会对你们有所帮助；第二期望有意参与 ACM/ICPC 竞赛的同学将该资料作为你竞赛入门的学习材料，丰富自己的知识面、夯实自己的基础，为更有效的参与该项赛事做好准备。