第二部分　数 据 结 构

第3章　数 据 描 述

从本章开始我们进行数据结构的研究，一直到第12章为止。尽管本章的重点是介绍线性表，但一个主要目标是为了使大家明白，数据可以用不同的形式进行描述或存储在计算机存储器中。最常见的数据描述方法有：公式化描述、链接描述、间接寻址和模拟指针。

公式化描述借助数学公式来确定元素表中的每个元素分别存储在何处（如存储器地址）。最简单的情形就是把所有元素依次连续存储在一片连续的存储空间中，这就是通常所说的连续线性表。

在链接描述中，元素表中的每个元素可以存储在存储器的不同区域中，每个元素都包含一个指向下一个元素的指针。同样，在间接寻址方式中，元素表中的每个元素也可以存储在存储器的不同区域中，不同的是，此时必须保存一张表，该表的第 i项指向元素表中的第 i个元素，所以这张表是一个用来存储元素地址的表。

在公式化描述中，元素地址是由数学公式来确定的；在链接描述中，元素地址分布在每一个表元素中；而在间接寻址方式下，元素地址则被收集在一张表中。

模拟指针非常类似于链接描述，区别在于它用整数代替了 C++指针，整数所扮演的角色与指针所扮演的角色完全相同。

本章介绍了线性表的四种描述形式，通过考察常见表操作（如插入、删除）的复杂性，给出了每种描述方法的优缺点。在本章中还将学习如何用数组来模拟 C++指针。

本章所给出的有关数据结构的概念如下：

- 抽象数据类型。
- 公式化描述、链接描述、间接寻址和模拟指针。
- 单向链表、循环链表和双向链表。

本章的应用部分集中介绍了链表的应用，之所以这样做，是因为第 1章和第2章中所给出的所有程序都采用了公式化的数据表示形式，而现在希望采用链表形式来改写其中的部分程序。二叉排序、基数排序和等价类应用都使用了链表，而凸面体应用则采用了双向链表。二叉排序和基数排序可用来对 n 个元素进行排序，如果关键值介于一个“ 合适的范围 ”内，排序所需要的时间为 $\Theta(n)$。虽然在第2章中所给出的排序算法需要耗时 $O(n^2)$，但它不需要将关键值限制在一个“ 合适的范围 ”内。当关键值介于一个“ 合适的范围 ”内时，二叉排序和基数排序要比第2章所给出的排序算法快出许多。在二叉排序的应用中还可以看到如何把函数名作为一个参数传递给C++函数。

3.1　引言

数据对象（data object）即一组实例或值，例如：

- *Boolean* ={false, true }

- *Digit* = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
- *Letter* = {*A, B, C, ···, Z, a, b, ···, z*}
- *NaturalNumber* = {0, 1, 2, ···}
- *Integer* = {0, ±1, ±2, ±3, ··· }
- *String* = {*a, b, ···, aa, ab, ac, ···*}

Boolean, *Digit*, *Letter*, *NaturalNumber*, *Integer*和*String*都是数据对象，true和false是*Boolean*的实例，而0, 1, ···, 和9 都是*Digit* 的实例。数据对象的每个实例要么是一个原语（primitive）（或原子（atomic）），要么是由其他数据对象的实例复合而成。在后一种情形下，用术语“元素（element）”来表示对象实例的单个组件。

例如，数据对象*NaturalNumber* 的每个实例均可以视为原子，在这种情况下，不必考虑对这些实例做进一步的分解。另一种观点是把*NaturalNumber* 的每个实例看成是由*Digit*数据对象的若干实例复合而成。按照这种观点，整数675是由数字6，7和5按顺序组成。

数据对象*String*是所有可能的串实例的集合，每个实例均由字符组成。串实例的一些具体例子如：*good, a trip to Hawaii, going down hill*和*abcabcdabcde*。其中第一个串由四个元素 *g, o, o* 和*d* 按序构成，而每个元素又都是数据对象*Letter*的一个实例。数据对象的实例以及构成实例的元素通常都有某种相关性，例如，自然数 0是最小的自然数，1是仅比0大的自然数，而2是1之后的下一个自然数。在自然数 675中, 6是最高有效位，7在其次，而5是最低有效位。在串*good* 中，*g* 是第一字母，*o* 是第二和第三个字母，而*d* 是最后一个字母。

除了相互关联之外，对于任一个数据对象通常都有一组相关的函数。这些函数可以把对象的某个实例转换成该对象的另外一个实例，或转换成另一个数据对象的实例，或者同时进行上述两种转换。函数也可以简单地创建一个新的实例，而不必对一个新创建的实例进行转换。例如，进行自然数相加的函数将创建一个新的自然数，该自然数是两个相加数之和，两个参与加操作的自然数本身不会发生任何变化。

数据结构（data structure）包括数据对象和实例以及构成实例的每个元素之间所存在的各种关系。这些关系可由相关的函数来实现。

当我们研究数据结构时，关心的是数据对象（实际上是实例）的描述以及与数据对象相关函数的具体实现。数据对象的良好描述可以有效地促进函数的高效实现。

最常使用的数据对象以及函数都已经在 C++中被当作标准的数据类型加以实现，如整数对象(int)、实数对象(float)、布尔对象(bool)等。所有其他的数据对象均可以采用标准数据类型、枚举类型以及由C++的类、数组和指针特性所提供的组合功能来描述。例如，可以用一个字符数组s 来描述String 的实例：

```
char s[MaxSize];
```

有关数据结构的研究包括两个部分。本章按照数据的各种描述方法进行组织，即基于公式的描述、链表描述、简接寻址和模拟指针。我们利用数据对象线性表来演示这些方法。在后续章节中将研究其他流行的数据描述方法，如矩阵、栈、队列、字典、优先队列和图。

3.2 线性表

线性表（linear list）是这样的数据对象，其实例形式为：$(e_1, e_2, \ldots e_n)$，其中n 是有穷自然数。e_i 是表中的元素，n 是表的长度。元素可以被视为原子，因为它们本身的结构与线性表的结构无关。当$n = 0$ 时，表为空；当$n > 0$时，e_1 是第一个元素，e_n 是最后一个元素，可以认为e_1 优先于e_2，e_2 优先于e_3，如此等等。除了这种优先关系之外，在线性表中不再有其他的结构。

我们用s表示每个元素e_i所需要的字节数，因此，s是一个元素的大小。

以下是一些线性表的例子：1) 一个班级学生姓名按字母顺序排列的列表；2) 按递增次序排列的考试分数表；3) 按字母顺序排列的会议列表；4) 奥林匹克男子篮球比赛中金牌获得者按年代次序排列的列表。根据这些例子可知对于线性表有必要执行下列操作：

- 创建一个线性表。
- 确定线性表是否为空。
- 确定线性表的长度。
- 查找第k个元素。
- 查找指定的元素。
- 删除第k个元素。
- 在第k个元素之后插入一个新元素。

可以用一个抽象数据类型 (abstract data type, ADT) 来说明线性表，它给出了实例及相关操作的描述(见ADT 3-1)。

请注意，这种抽象数据类型说明与C++ 的类定义之间具有很多相似性。抽象数据类型说明独立于我们已提到的任何描述方法。抽象数据类型的描述方法必须满足抽象数据类型说明，而说明又反过来保证了描述方法的合法性。除此以外，所有满足说明的描述方法都可以在数据类型应用中替换使用。

ADT 3-1 线性表的抽象数据类型描述

抽象数据类型*LinearList* {
实例
0或多个元素的有序集合
操作
Create (): 创建一个空线性表
Destroy (): 删除表
IsEmpty(): 如果表为空则返回true，否则返回false
Length (): 返回表的大小 (即表中元素个数)
Find (*k*,*x*): 寻找表中第*k* 个元素，并把它保存到*x* 中；如果不存在，则返回false
Search (*x*): 返回元素x在表中的位置；如果*x* 不在表中，则返回0
Delete (*k*,*x*): 删除表中第*k* 个元素，并把它保存到*x* 中；函数返回修改后的线性表
Insert (*k*,*x*): 在第*k*个元素之后插入*x*；函数返回修改后的线性表
Output (*out*): 把线性表放入输出流*out* 之中
}

除了用ADT 3-1中非正式的自然语言来说明抽象数据类型外，还可以使用 C++的抽象类来说明抽象数据类型。在该方法中需使用派生类、抽象类和虚拟函数，我们将在第 5章和第12章中详细讨论这些话题，目前还只能使用非正式的自然语言。如果你已经很熟悉派生类和抽象类，可以查阅12.9.4节看看如何用C++抽象类来说明线性表抽象数据类型。

3.3 公式化描述

3.3.1 基本概念

公式化描述（formala-based）采用数组来表示一个对象的实例，数组中的每个位置被称之

为单元（cell）或节点（node），每个数组单元应该足够大，以便能够容纳数据对象实例中的任意一个元素。在某些情况下，每个实例可分别用一个独立的数组来描述，而在其他情况下，可能要使用一个数组来描述几个实例。实例中每个元素在数组中的位置可以用一个数学公式来指明。

假定使用一个数组来描述表，需要把表中的每个元素映射到数组的具体位置上。第一个元素在什么地方？第二个元素在什么地方？在公式化描述中，可用一个数学公式来确定每个元素的位置。一个简单的映射公式如下：

$$location(i) = i - 1 \qquad (3\text{-}1)$$

公式（3-1）指明表中第i个元素（如果存在的话）位于数组中i-1位置处。图3-1a给出了一个利用公式（3-1）作为映射公式，在数组element中描述一个5元素线性表的例子。（一个更简洁的公式是：$location(i)=i$，它不使用0位置，在练习8至练习13中将使用这个公式）。

为了完整地描述线性表，需要了解表的当前长度或大小，为此，使用变量 length作为表的长度。当表为空时， length为0。程序3-1给出了相应的C++类定义。由于表元素的数据类型随着应用的变化而变化，所以定义了一个模板类，在该模板类中，用户指定元素的数据类型为 T。数据成员length、MaxSize和element都是私有成员，其他成员均为共享成员。 Insert和Delete均返回一个线性表的引用。我们将要看到，具体实现时首先会修改表 *this，然后返回一个引用（指向修改后的表）。因此，同时组合多个表操作是可行的，如X.Insert(0,a).Delete(3,b)。

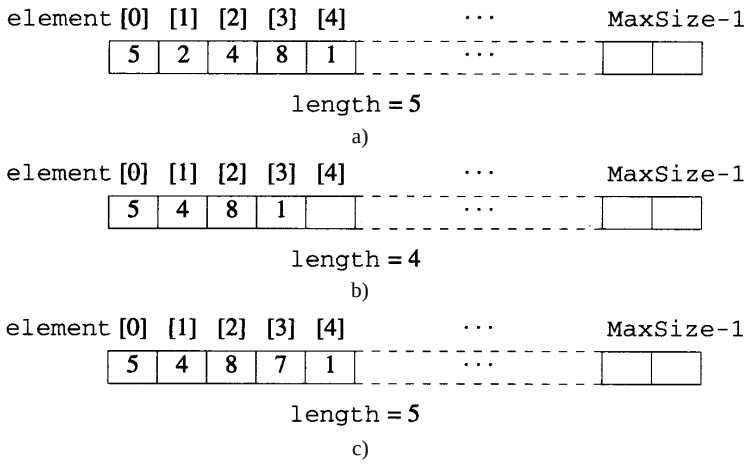

图3-1 线性表

程序3-1 基于公式的类 LinearList

```
template<class T>
class LinearList {
 public:
   LinearList(int MaxListSize = 10); //构造函数
   ~LinearList()  {delete [] element;} //析构函数
   bool IsEmpty() const {return length == 0;}
   int Length() const {return length;}
   bool Find(int k, T& x) const; //返回第k个元素至x中
   int Search(const T& x) const; // 返回x所在位置
```

```
    LinearList<T>& Delete(int k, T& x); // 删除第k个元素并将它返回至x中
    LinearList<T>& Insert(int k, const T& x); // 在第k个元素之后插入x
    void Output(ostream& out) const;
  private:
    int length;
    int MaxSize;
    T *element; // 一维动态数组
};
```

3.3.2 异常类NoMem

如果分配内存失败，本书中所给出的大多数代码都能引发一个异常。有时，异常可能由new引起，而有时则需要我们自己来引发。我们希望在所有情形下都能引发同一个异常，因此，定义了一个异常类NoMem，见程序3-2。函数my_new_handler简单地引发一个类型为NoMem的异常。程序3-2的最后一行调用了C++函数set_new_handler，每当分配内存失败时，该函数就让操作符new调用函数my_new_handler。所以，new引发的是异常NoMem而不是xalloc。每当分配内存失败时，set_new_handler将返回一个指针，指向由new此前所调用的那个函数，该指针保存在变量Old_Handle_中。为了恢复new的原始行为，可以作如下调用：

```
set_new_handler(Old_Handler_);
```

注意，程序3-2中用一个函数名作为实参，另外，不管new以前引发的是哪一种类型的异常，程序3-2都将把new引发的异常变成NoMem异常。

程序3-2 使new引发NoMem异常而不是xalloc异常

```
// 内存不足
class NoMem {
  public:
    NoMem () {}
} ;
// 使new引发NoMem异常而不是xalloc异常
void my_new_handler()
{
  throw NoMem();
}
new_handler Old_Handler_=set_new_handler(my_new_handler);
```

3.3.3 操作

操作Create和Destroy分别被作为类的构造函数和析构函数加以实现。构造函数（见程序 3-3）创建一个最大缺省长度为 10的表。当计算机没有足够的内存来创建一个所期望长度的数组时，操作符new引发一个类型为NoMem的异常，构造函数代码并没有捕获这个异常。如果所引发的异常没有在程序的任何位置被捕获，程序将非正常终止。在大多数代码中，我们都希望应用代码能够捕获所引发的异常。

程序3-3 基本的表操作

```
template<class T>
```

```
LinearList<T>::LinearList(int MaxListSize)
{// 基于公式的线性表的构造函数
  MaxSize = MaxListSize;
  element = new T[MaxSize];
  length = 0;
}
template<class T>
bool LinearList<T>::Find(int k, T& x) const
{//把第k个元素取至x中
 //如果不存在第k个元素则返回false，否则返回true
  if (k < 1 || k > length)  return false; // 不存在第k个元素
  x = element[k - 1];
  return true;
}
template<class T>
int LinearList<T>::Search(const T& x) const
{// 查找x，如果找到，则返回x所在的位置
 // 如果x不在表中，则返回0
  for (int i = 0; i < length; i++)
    if (element[i] == x) return ++i;
  return 0;
}
```

下面的语句创建一个整数线性表y，其最大长度为100：

```
LinearList<int> y(100);
```

析构函数（见程序3-1）调用delete以便释放由构造函数分配给数组element的空间。程序3-1中包含了IsEmpty和Length的代码，而程序3-3给出了Find和Search的代码。IsEmpty、Length和Find的复杂性为$\Theta(1)$，而Search的复杂性为O(length)。

为了从一个表中删除第k个元素，需要首先确认表中包含第k个元素，然后再删除这个元素。如果表中不存在第k个元素，则出现一个异常。ADT LinearList (见ADT 3-1)没有告诉我们这个时候该做什么。我们的代码将引发一个类型为 OutOfBounds的异常。每当正在执行的函数中任一参数超出所期望的范围时，就引发这种类型的异常。

如果存在第 k个元素，可以将元素 k+1，k+2，…，length依次向前移动一个位置，并将length的值减1，从而删除第k个元素。例如，为了从图3-1a 的表中删除第二个元素，需要把元素4，8和1分别移动到表的1，2和3位置处，这些位置分别对应数组element的1、2和3位置。图3-1b 给出了删除第二个元素之后的表，这时，表的长度为4。

当线性表按照公式（3-1）进行描述时，函数Delete(见程序3-4)给出了删除操作。如果不存在第k个元素，将引发一个异常， Delete所需要的时间为$\Theta(1)$；如果存在第k个元素，则移动length-k个元素，需要耗时$\Theta((length-k)s)$，其中s是每个元素的大小。此外，被删除的元素被移动至x，因此，总的时间复杂性为O((length-k)s)。

程序3-4 从线性表中删除一个元素

```
template<class T>
LinearList<T>& LinearList<T>::Delete(int k, T& x)
{// 把第k个元素放入x中，然后删除第k个元素
```

```
// 如果不存在第k个元素，则引发异常OutOfBounds
  if (Find(k, x)) {// 把元素 k+l, ...向前移动一个位置
     for (int i = k; i < length; i++)
      element[i-l] = element[i];
     length--;
     return *this;
     }
  else throw OutOfBounds();
}
```

为了在表中第k个元素之后插入一个新元素，首先需要把 k+1至length元素向后移动一个位置，然后把新元素插入到 k+1位置处。例如，在图 3-1b 的表中第三个元素之后插入 7，将得到图3-1c 的结果。程序3-5中给出了完整的、插入一个新元素的 C++代码。可以注意到，在插入操作期间，可能出现两类异常。第一类异常发生的情形是：没有正确指定插入点，如在插入新元素之前，表中元素个数少于 k-1个，或者k<0。在这种情形下，引发一个 OutOfBounds异常。当表已经满时，会发生第二类异常，此时，数组没有剩余的空间来容纳新元素，因此将引发一个NoMem异常。Insert的时间复杂性为O((length-k)s)。

程序3-5 向线性表中插入一个元素

```
template<class T>
LinearList<T>& LinearList<T>::Insert(int k, const T& x)
{// 在第k个元素之后插入x
 // 如果不存在第k个元素，则引发异常OutOfBounds
 // 如果表已经满，则引发异常NoMem
  if (k < 0 || k > length)  throw OutOfBounds();
  if (length == MaxSize) throw NoMem();
  //向后移动一个位置
  for (int i = length-i; i >= k; i--)
     element[i+l] = element[i];
  element[k] = x;
  length++;
  return *this;
}
```

程序3-6给出了Output的代码，其时间复杂性为 Θ(length)。这段代码简单地把表元素插入到输出流out之中。为了实际显示线性表，可以重载操作符 <<，见程序3-6。

程序3-6 把线性表输送至输出流

```
template<class T>
void LinearList<T>::Output(ostream& out) const
{//把表输送至输出流
 for (int i = 0; i < length; i++)
    out << element[i] << " ";
}
// 重载 <<
template <class T>
```

```
ostream&  operator<<(ostream& out, const  LinearList<T>&  x)
{x.Output(out);  return out;}
```

程序3-7是一个使用类LinearList的C++程序，它假定程序3-1至3-6均存储在文件llist.h之中，且异常类定义位于文件xcept.h之中。该示例完成如下工作：创建一个大小为5的整数线性表L；输出该表的长度（为0）；在第0个元素之后插入2；在第一个元素之后插入6（至此，线性表为2，6）；寻找并输出第一个元素（为2）；输出当前表的长度（为2）；删除并输出第一个元素。图3-2给出了由程序3-7产生的输出。

程序3-7　采用类LinearList的例子

```
#include <iostream.h>
#include "llist.h"
#include "xcept.h"
void main(void)
{
  try {
    LinearList<int> L(5);
    cout << "Length = " << L.Length()  << endl;
    cout << "IsEmpty = " << L.IsEmpty() <<endl;
    L.Insert(0,2).Insert(1,6);
    cout << "List is " << L <<endl;
    cout << "IsEmpty =  " << L.IsEmpty()  << endl;
    int z;
    L.Find(1,z);
    cout << "First element is " << z  << endl;
    cout << "Length =  " << L.Length()  << endl;
    L.Delete(1,z);
    cout << "Deleted element is " << z  << endl;
    cout << "List is " << L << endl;
   }
catch (...) {
     cerr << "An exception has occurred" << endl;
     }
}
```

```
Length = 0
IsEmpty = 1
List is 2  6
IsEmpty = 0
First element is 2
Length = 2
Deleted element is 2
List is 6
```

图3-2　程序3-7所产生的输出

3.3.4 评价

在接受一个线性表的公式化描述方法之前，先来考察一下这种描述方法的优缺点。的确，对于一个线性表的各种操作可以用非常简单的C++函数来实现。执行查找、删除和修改的函数都有一个最差的、与表的大小呈线性关系的时间复杂性。我们可能很满足于这种复杂性。（在第7章和第11章将看到能够更快地执行这些操作的描述方法。）

这种描述方法的一个缺点是空间的低效利用。考察如下的情形：我们需要维持三个表，而且已经知道在任何时候这三个表所拥有的元素总数都不会超过5000个。然而，很有可能在某个时刻一个表就需要5000个元素，而在另一时刻另一个表也需要5000个元素。若采用类LinearList，这三个表中的每一个表都需要有5000个元素的容量。因此，即使我们在任何时刻都不会使用5000以上的元素，也必须为此保留总共15 000个元素的空间。

为了避免这种情形，必须把所有的线性表都放在一个数组list中进行描述，并使用两个附加的数组first和last对这个数组进行索引。图3-3给出了在一个数组list中描述的三个线性表。我们采用大家很习惯的约定，即如果有m个表，则每个表从1到m进行编号，且first[i]为第i个表中的第一个元素。有关first[i]的约定使我们更容易地采用公式化描述方式。last[i]是表i的最后一个元素。注意，根据这些约定，每当第i个表不为空时，有last[i]>first[i]，而当第i个表为空时，有last[i]=first[i]。所以在图3-3的例子中，表2是空表。在数组中，各线性表从左至右按表的编号次序1，2，3，...，m进行排列。

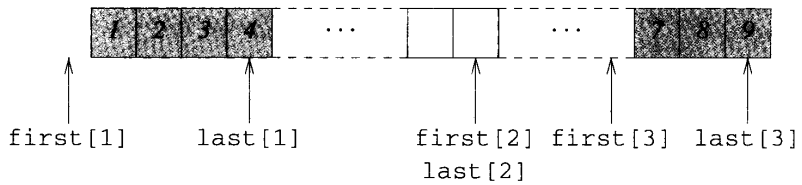

图3-3 一个数组中的所有线性表

为了避免第一个表和最后一个表的处理方法与其他的表不同，定义了两个边界表：表0和表m+1，其中first[0]=last[0]=-1，first[m+1]=last[m+1]=MaxSize-1。

为了在第i个表的第k个元素之后插入一个元素，首先需要为新元素创建空间。如果last[i]=first[i+1]，则在第i个表和第i+1个表之间没有空间，因此不能把第k+1至最后一个元素向后移动一个位置。在这种情况下，通过检查关系式last[i-1]<first[i]是否成立，可以确定是否有可能把第i个表的1至k-1元素向前移一个位置；如果这个关系式不成立，要么需要把表1至表i-1的元素向前移一个位置，要么把表i+1至表m向后移一个位置，然后为表i创建需要增长的空间。当表中所有的元素总数少于MaxSize时，这种移位的方法是可行的。

图3-4是一个伪C++函数，它向表i中插入一个新的元素，可以把该函数细化为兼容的C++代码。

尽管在一个数组中描述几个线性表比每个表用一个数组来描述空间的利用率更高，但在最坏的情况下，插入操作将耗费更多的时间。事实上，一次插入操作可能需要移动MaxSize-1个元素。

练习

1. 类LinearList的一个缺点是需要预测线性表最大可能的尺寸，解决的方法之一是：在创

```
int insert(int i, int k, int y)
{//在表i 的第k 个元素之后插入y
 // first和last是全局数组
 int j, m;
 m = last[i] - first[i]; //表i中的元素数
 if (k < 0 || k > m) return 0;
 //在右边有剩余空间吗？
 寻找最小的j (j≥i)，使得 last [j ] < first [j+l];
 如果存在这样的j, 则把表i+l至表j以及表i的k+l至最末元素均向后移动一个位置,
 然后将y插入i中；
 这种移动需要相应地修改last和first的值
 // 在左边有剩余空间吗？
 如果不存在上述的j，则寻找另一个最大的j (j<i)，使得last [j ] < first [j+l];
 如果找到这样的j，则把表j至表i-1以及表i的l至k-1元素均向前移动一个位置,
 然后将y插入i中；
 这种移动需要相应地修改last和first的值
 // 成功否?
 return ( (没有找到上述的j)?0: 1);
}
```

图3-4 向一个数组中（包含多个线性表）插入一个元素的伪代码

建线性表时，置MaxSize=1，之后在执行插入操作期间，如果在表中已经有了 MaxSize个元素，则将MaxSize加倍，按照这个新尺寸分配一个新数组，并将老数组中的数据复制到新数组中，最后将老数组删除。类似地，在执行删除操作期间，如果线性表的尺寸降至当前 MaxSize的四分之一，则分配一个更小的、尺寸为MaxSize/2的数组，并将老数组中的数据复制到新数组中，最后将老数组删除。

1) 采用上述思想，重新实现类LinearList。构造函数不应带参数，并将MaxSize置为1，并分配一个大小为1的数组，同时置length=0。

2) 考虑从一个空表开始，对其连续实施 n 次表操作。假定在使用以前的实现方法时用了 $f(n)$步，试证明对于上述新的实现方法，存在常数 c，使得执行步数最多为 $cf(n)$。

2. 假设一个线性表的描述满足公式（3-1）

1) 扩充LinearList类的定义，增加一个函数Reverse，该函数将表中元素的次序变反。反序操作是就地进行的（即在数组 element本身的空间内）。注意，在反序操作进行之前，表中第 k 个元素（如果存在）位于element[k-1]，完成反序之后，该元素位于element[length-k]。

2) 证明上述函数的复杂性与线性表的长度成线性关系。

3) 采用适当的测试数据来验证代码的正确性。

4) 请编写另外一个就地处理的反序函数，它能对 LinearList类型的对象进行反序操作。该函数不是LinearList类的成员函数，但它应利用成员函数来产生反序线性表。

5) 上述函数的时间复杂性是多少？

6) 分别采用大小为1000，5000和10 000的线性表来比较以上两种反序函数的执行效率。

3. 扩充LinearList类的定义，增加一个成员函数Half()。调用X.Half()将删除X中半数的元素。比如，如果X.length的初始值为7，且X.element[ ]=[2, 13, 4, 5, 17, 8, 29]，则执行X.Half()后，

X.length的值为4，且X.element[]=[2, 4, 17, 29]；如果X.length的初始值为4，且X.element[]=[2, 13, 4, 5]，则执行X.Half()后，X.length的值为2，且X.element[ ]=[2, 4]。如果X开始时是空表，则执行X.Half()后，X仍然为空表。

1) 试给出成员函数Half()的代码。不能利用任何其他的LinearList成员函数。代码的复杂性应为Θ(length)。

2) 证明代码的复杂性确实为Θ(length)。

3) 使用适当的测试数据来测试代码的正确性。

4. LinearList缺省的复制构造函数仅复制length,MaxSize和element的值。因此，当用线性表L作为函数F的实际参数（对应于值参X）来调用F时，L.length,L.MaxSize和L.element被复制到X的相应成员中。当函数F退出时，LinearList的析构函数被X唤醒，数组X.element（与数组L.element相同）被删除。避免L.element被删除的一种方法是定义复制构造函数如下：

```
LinearList<T>:: LinearList(const LinearList<T>& L)
```

该函数复制length和MaxSize的值，然后创建一个新数组element，并将L.element[0: MaxSize-1]复制到element 中。试编写这个复制构造函数，并估计其复杂性。

5. 在许多应用中，需要对一个线性表中的元素进行前移和后移操作。试扩充LinearList类的定义，增加一个私有成员变量current，它纪录线性表当前的位置。此外，需要增加的共享成员如下：

1) Reset—置current为1。

2) Current(x)—返回x中的当前元素。

3) End—当且仅当当前元素为表的最后一个元素时，返回true。

4) Front—当且仅当当前元素为表的第一个元素时，返回true。

5) Next—移动current至表中的下一个元素，如果操作失败则引发一个异常。

6) Previous—移动current至表中的前一个元素，如果操作失败则引发一个异常。

试编写上述代码，并使用适当的测试数据来测试代码的正确性。

6. 设A和B均为LinearList对象

1) 编写一个新的成员函数Alternate(A,B)以创建一个新的线性表，该表包含了A和B中的所有元素，其中A和B的元素轮流出现，表中的首元素为A中的第一个元素。在轮流排列元素时，如果某个表的元素用完了，则把另一个表的其余元素依次添加在新表的后部。代码的复杂性应与两个输入表的长度呈线性比例关系。

2) 证明代码具有线性复杂性。

3) 使用适当的测试数据来测试代码的正确性。

7. 设A和B均为LinearList对象。假定A和B中的元素都是按序排列的（如从左至右按递增次序排列）。

1) 试编写一个成员函数Merge(A, B)，用以创建一个新的有序线性表，该表中包含了A和B的所有元素。

2) 考察所编写的函数的时间复杂性。

3) 用适当的测试数据来测试代码的正确性。

8. 1) 试编写函数LinearList::Split (A, B)，该函数用来创建两个线性表A和B，A中包含*this的所有奇数元素（注意一个线性表的奇数元素是指element偶数位置上的元素），B中包含其余的元素。

2) 考察函数的时间复杂性。

3) 用适当的测试数据来测试代码的正确性。

9. 假定采用如下公式来描述一个线性表：

$$location(i) = i \tag{3-2}$$

1) 对于程序3-1中的类定义需要做相应的修改吗？如果是，请编写新的定义。

2) 能够描述的最长的表的长度是多少？

3) 修改程序3-1中的所有函数，以满足公式（3-2）

4) 用适当的测试数据来测试代码的正确性。

5) 每个函数的时间复杂性分别是多少？

10. 用公式（3-2）代替公式（3-1）来完成练习2。

11. 用公式（3-2）代替公式（3-1）来完成练习6。

12. 用公式（3-2）代替公式（3-1）来完成练习7。

13. 用公式（3-2）代替公式（3-1）来完成练习8。

14. 假定采用下述公式来描述一个线性表：

$$location(i) = (location(1) + i - 1) \% MaxSize \tag{3-3}$$

其中MaxSize是用来存储表元素的数组的大小。与专门保留一个表长的做法不同的是，用变量first 和last 来指出表的第一个元素和最后一个元素的位置。

1) 基于该公式，设计一个与LinearList相似的类。

2) first和last的初始值应该是多少？

3) 对于新的类，试编写出所有成员函数的代码。（通过适当的选择，将元素移动到欲插入/删除元素的左边或右边，可以编写出更高效的Delete和Insert代码。）

4) 考察每个函数的时间复杂性。

5) 用适当的测试数据来测试代码的正确性。

15. 用公式（3-3）代替公式（3-1）来完成练习2。

16. 用公式（3-3）代替公式（3-1）来完成练习6。

17. 用公式（3-3）代替公式（3-1）来完成练习7。

18. 用公式（3-3）代替公式（3-1）来完成练习8。

19. 将图3-4细化为C++ 函数，并测试其正确性。

20. 编写一个C++ 函数，该函数在表i 的第k 个元素之后插入一个元素。假定在一个数组中存放了n 个线性表。如果不得不移动多个表以便容纳新元素，新的函数应该首先确定可用空间的数量。移动表时应保证每个表所包含的可用空间数量应大体相同，以便适应未来增长的需要。通过编译和执行，测试代码的正确性。

21. 编写一个C++函数，该函数用来从表i 中删除第k 个元素。假定在一个数组中存放了 n 个线性表。通过编译和执行，测试代码的正确性。

3.4 链表描述

3.4.1 类ChainNode 和Chain

在链表描述中，数据对象实例的每个元素都放在单元或节点中进行描述。不过，节点不必是一个数组元素，因此没有什么公式可用来定位某个元素。取而代之的是，每个节点中都包含了与该节点相关的其他节点的位置信息。这种关于其他节点的位置信息被称之为链（ link）或

指针（pointer）。

令$L=(e_1, e_2, ..., e_n)$是一个线性表。在针对该表的一个可能的链表描述中，每个元素e_i都放在不同的节点中加以描述。每个节点都包含一个链接域，用以指向表中的下一个元素。所以节点e_i的指针将指向e_{i+1}，其中$1 \leq i < n$。节点e_n没有下一个节点，所以它的链接域为NULL(或0)。指针变量*first*指向描述中的第一个节点。图3-5给出了表$L=(e_1, e_2, ..., e_n)$的链表描述。

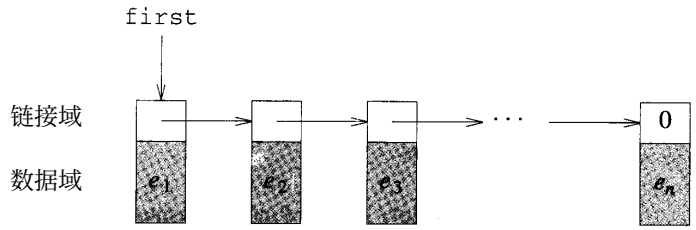

图3-5 线性表的链表描述

由于图3-5中的每个链表节点都正好有一个链接域，所以该图的链表结构被称之为单向链表（singly linked list）。并且，由于第一个节点e_1的指针指向第二个节点e_2，e_2的指针指向e_3，...，最后一个节点链接域为NULL(或0)，故这种结构也被称作链（chain）。为了把一个线性表表示成一个链，可以使用程序 3-8中的类定义 ChainNode和Chain。由于 Chain<T>是ChainNode<T>的一个友类，所以Chain<T>可以访问ChainNode<T>的所有成员（尤其是私有成员）。共享成员Length、Find、Delete和Insert的定义与程序3-1完全一致。

程序3-8 链表的类定义

```
template <class T>
class ChainNode {
  friend Chain<T>;
  private:
     T data;
     ChainNode<T> *link;
};
template<class T>
class Chain {
  public:
     Chain() {first = 0;}
     ~Chain();
     bool IsEmpty() const {return first == 0;}
     int Length() const;
     bool Find(int k, T& x) const;
     int Search(const T& x) const;
     Chain<T>& Delete(int k, T& x);
     Chain<T>& Insert(int k, const T& x);
     void Output(ostream& out) const;
  private:
     ChainNode<T> *first; // 指向第一个节点的指针
};
```

3.4.2 操作

可以采用如下的描述来创建一个空的整数型线性表：

```
Chain<int> L;
```

注意，线性表的链表描述不需要指定表的最大长度。

程序3-9给出了析构函数的代码，它的复杂性为$\Theta(n)$，其中n为链表的长度。程序3-10和程序3-11中的代码分别实现了Length操作和Find操作。Length的复杂性为$\Theta(n)$，Find的复杂性为$O(k)$。函数Search(见程序3-12)假定对于类型T定义了!= 操作，该函数的复杂性为$O(n)$。Output(见程序3-13)的复杂性为$\Theta(n)$，它要求对于类型T必须定义<<操作。

程序3-9 删除链表中的所有节点

```
template<class T>
Chain<T>::~Chain()
{// 链表的析构函数，用于删除链表中的所有节点
  ChainNode<T> *next; // 下一个节点
  while (first) {
     next = first->link;
     delete first;
     first = next;
    }
}
```

程序3-10 确定链表的长度

```
template<class T>
int Chain<T>::Length() const
{// 返回链表中的元素总数
  ChainNode<T> *current = first;
  int len = 0;
  while (current) {
     len++;
     current = current->link;
    }
  return len;
}
```

程序3-11 在链表中查找第k个元素

```
template<class T>
bool Chain<T>::Find(int k, T& x) const
{// 寻找链表中的第k个元素，并将其传送至x
 //如果不存在第k个元素，则返回false，否则返回true
   if (k < 1) return false;
   ChainNode<T> *current = first;
   int index = 1; // current的索引
   while (index < k && current) {
      current = current->link;
```

```
      index++;
     }
   if (current) {x = current->data; return true;}
   return false; // 不存在第k个元素
}
```

程序3-12 在链表中搜索

```
template<class T>
int Chain<T>::Search(const T& x) const
{// 寻找x，如果发现x，则返回x的地址
 //如果x不在链表中，则返回0
   ChainNode<T> *current = first;
   int index = 1; // current的索引
   while (current && current->data != x) {
     current = current->link;
     index++;
      }
   if (current) return index;
   return 0;
}
```

程序3-13 输出链表

```
template<class T>
void Chain<T>::Output(ostream& out) const
{// 将链表元素送至输出流
  ChainNode<T> *current;
   for (current = first; current; current = current->link)
     out << current->data << "   ";
}
//重载<<
template <class T>
ostream& operator<<(ostream& out, const Chain<T>& x)
  {x.Output(out); return out;}
```

为了从图3-6所示的链中删除第四个元素，需进行如下操作：

1) 找到第三和第四个节点。

2) 使第三个节点指向第五个节点。

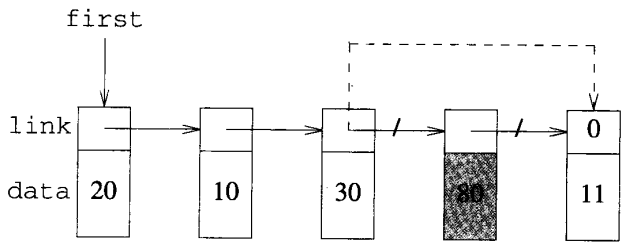

图3-6 删除第四个节点

3) 释放第四个节点所占空间，以便于重用。

程序3-14中给出了删除操作的代码。有三种情形需要考虑。第一种情形是： k小于1或链表为空；第二种情形是：第一个元素将被删除且链表不为空；最后一种情形是：从一个非空的链表中删除首元素之外的其他元素。

程序3-14 从链表中删除一个元素

```
template<class T>
Chain<T>& Chain<T>::Delete(int k, T& x)
{// 把第k个元素取至x，然后从链表中删除第k个元素
 //如果不存在第k个元素，则引发异常 OutOfBounds
   if (k < 1 || !first)
      throw OutOfBounds(); // 不存在第k个元素
// p最终将指向第k个节点
ChainNode<T> *p = first;
// 将p移动至第k个元素，并从链表中删除该元素
if (k == 1) // p已经指向第k个元素
   first = first->link; // 删除之
else ( // 用q指向第k-1个元素
  ChainNode<T> *q = first;
   for (int index = 1; index < k - 1 && q;  index++)
      q = q->link;
   if (!q || !q->link)
      throw OutOfBounds(); //不存在第k个元素
   p = q->link; // 存在第k个元素
   q->link = p->link;) // 从链表中删除该元素
//保存第k个元素并释放节点p
x = p->data;
delete p;
return *this;
}
```

程序3-14的代码首先处理第一种情形，即引发OutOfBounds 异常。对于其他两种情形，定义了一个指针变量p，并将它初始化为指向链表中的第一个元素。如果 k 是1，则p 将指向链表中的第k 个节点，语句first=first->link 则用来从链表中删除p 所指向的节点。如果k 大于1，指针变量q 用来定位第k-1个节点，定位过程可采用一个 for 循环，假定链表中有多个节点，则当从for 循环中退出时，q 会指向第k-1个节点。如果链表的节点数少于k-1，则q 为0。接下来的if 语句首先判断q 是否为0，如果q 不为0，则if 语句通过检查q->link 来判断链表中是否存在第k 个元素。如果if 语句的判断结果为真，则将修改p 指针以使其指向第k 个节点。通过将前一个节点（即q 节点）改为指向p 的下一个节点，即可将第k 个节点从链表中删除。

当从if-else结构中退出时，p 指向第k 个节点，并且该节点已经脱离链表。下面只需把p 的数据域传递给参数x，并释放节点p 所占用的空间。

为了检验程序3-14的正确性，分别用一个空表和一个至少包含一个节点的链表进行测试。此外，还可对不同的k值进行测试，如k≤0、k=n、k≥n、0<k<n 等，其中n 为链表长度。

插入和删除的过程很相似。为了在链表的第 k个元素之后插入一个新元素，需要首先找到第k 个元素，然后在该节点的后面插入新节点。图 3-7给出了k=0 和k≠0 两种情况下链表指针的变化。插入之前的实线指针，在插入之后被“ 打断 ”。程序3-15给出了相应的C++ 代码，它

的复杂性为O (k)。尽管插入操作的代码中引用了new，但它没有去捕获可能产生的异常，而是由Insert 的调用者捕获。

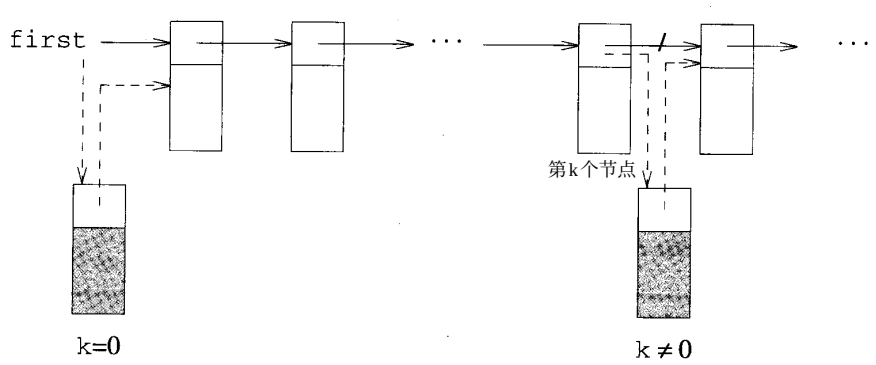

图3-7 向链表中插入元素

程序3-15 向链表中插入元素

```
template<class T>
Chain<T>& Chain<T>::Insert(int k, const T& x)
{// 在第k个元素之后插入x
 //如果不存在第k个元素，则引发异常OutOfBounds
 // 如果没有足够的空间，则传递NoMem异常
   if (k < 0) throw OutOfBounds();
   // p最终将指向第k个节点
   ChainNode<T> *p = first;
   //将p移动至第k个元素
   for (int index = 1; index < k && p;  index++)
     p = p->link;
   if (k > 0 && !p) throw OutOfBounds(); //不存在第k个元素
   // 插入
   ChainNode<T> *y=new ChainNode<T>;
   y->data = x;
   if (k) {// 在p之后插入
       y->link = p->link;
        p->link = y;}
   else {// 作为第一个元素插入
     y->link = first;
     first = y;}
  return *this;
}
```

3.4.3 扩充类Chain

在某些链表的应用中，希望执行一些未在抽象数据类型 LinearList(见ADT3-1)中出现的那些操作。因此有必要扩充Chain的类定义，以便包含一些附加的函数，如Erase(删除链表中的所有节点)、Zero(将first指针置为0，但并不删除任何节点)，Append(在链表的尾部添加一个元素)。函数Erase(见程序3-16)等价于类的析构函数。事实上，根据Erase的定义，可以把类的析构函数

简单地定义为对Erase的调用。

程序3-16 删除链表中的所有节点

```
template<class T>
void Chain<T>::Erase()
{//删除链表中的所有节点
  ChainNode<T> *next;
  while (first) {
    next = first->link;
    delete first;
    first = next;}
}
```

函数Zero可以定义为如下的内联函数：

```
void Zero() { first = 0;}
```

为了在O(1)的时间内添加一个元素，需要借助一个新的、类型为 ChainNode<T> *的私有成员last来跟踪链表的最后一个元素。程序 3-17给出了Append的代码，为了使该段代码能正确运行，必须在Delete函数（见程序3-14）的语句

```
p = q - > link
```

之后添加如下语句：

```
if ( p == last ) last = q ;
```

在Insert函数（见程序3-15）的语句return *this之前必须加入下面的语句：

```
if (!y -> link ) last = y;
```

程序3-17 在链表右端添加一个元素

```
template < class T >
Chain < T > & Chain < T > ::Append(const T& x)
{//在链表尾部添加x
  ChainNode< T > *y;
  y = new ChainNode< T >;
  y->data = x; y->link = 0;
  if (first) {//链表非空
       last->link = y;
       last = y;}
  else // 链表为空
      first = last = y;
  return *this;
}
```

3.4.4 链表遍历器类

暂且假定Output不是Chain类的成员函数，并且在该类中没有重载操作符 <<。为了输出链表X，将不得不执行如下代码：

```
int len = X.Length();
for (int i = 1; i <= len; i++) {
```

```
    X.Find(i,x);
    cout << X << ' ' ; }
```

这段代码的复杂性为$\Theta(n^2)$，而成员函数Output的复杂性为$\Theta(n)$，其中n为链表长度。类似于Output函数，许多使用链的应用代码都要求从链表的第一个元素开始，从左至右依次检查每一个元素。采用遍历器（Iterator）可以大大方便这种从左至右的检查，遍历器的功能是纪录当前位置并每次向前移动一个位置。

链表遍历器（见程序3-18）有两个共享成员Initialize和Next。Initialize返回一个指针，该指针指向第一个链表节点中所包含的数据，同时把私有变量 location设置为指向链表的第一个节点，该变量用来跟踪我们在链表中所处的位置。成员 Next用来调整location，使其指向链表中的下一个节点，并返回指向该节点数据域的指针。由于 ChainIterator类访问了Chain类的私有成员first，所以应把它定义为Chain的友类。

程序3-18 链表遍历器类

```
template<class T>
class ChainIterator {
public:
  T* Initialize(const Chain<T>& c)
     {location = c.first;
      if (location) return &location->data;
     return 0;}
  T* Next()
     {if (!location) return 0;
     location = location->link;
     if (location) return &location->data;
     return 0;}
private:
   ChainNode<T>  *location;
};
```

像前面一样，假定Output不是Chain类的成员函数，并且在该类中没有重载操作符 <<。采用链表遍历器，可以在线性时间内输出链表，见程序 3-19。

程序3-19 采用链表遍历器输出整数链表X

```
int *x;
ChainIterator<int> c;
x = c.Initialize(X);
while (x) {
   cout << *x << ' ' ;
   x = c.Next();
  }
cout << endl;
```

3.4.5 循环链表

采纳下面的一条或两条措施，使用链表的应用代码可以更简洁、更高效： 1)把线性表描述成一个单向循环链表（singly linked circular list），或简称循环链表（circular list），而不是一个

单向链表；2) 在链表的前部增加一个附加的节点，称之为头节点（head node）。通过把单向链表最后一个节点的链接指针改为指向第一个节点，就可以把一个单向链表改造成循环链表，如图3-8a 所示。图3-8b 给出了一个带有头指针的非空的循环链表，图 3-8c 给出了一个带有头指针的空的循环链表。

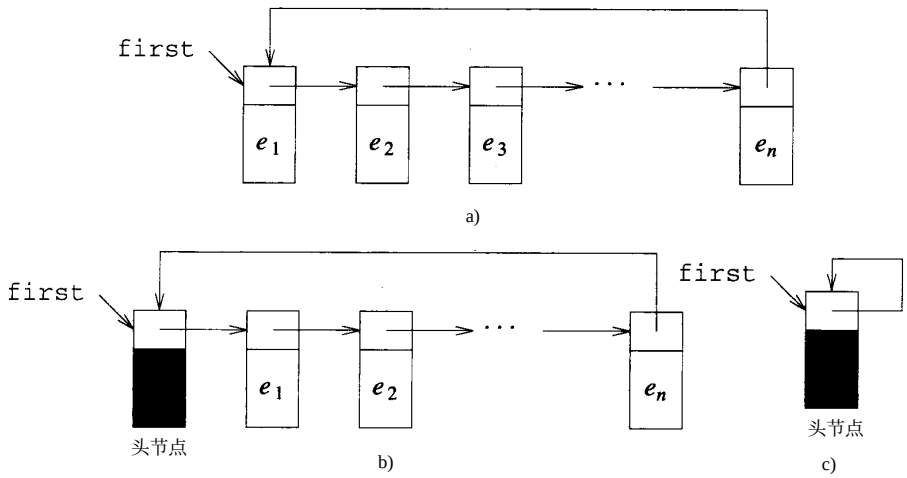

图3-8 循环链表

a) 循环链表 b) 带有头节点的循环链表 c) 空表

在使用链表时，头指针的使用非常普遍，因为利用头指针，通常可以使程序更简洁、运行速度更快。CircularList类的定义与Chain类的定义很类似。尽管链表搜索的复杂性仍然保持为O(*n*)，但代码本身要稍微简单一些。由于与程序3-12相比，程序3-20在for循环的每次循环中执行的比较次数较少，因此程序3-20将比程序3-12运行得更快一些，除非要查找的元素紧靠链表的左部。

程序3-20 在带有头节点的循环链表中进行查找

```
template<class T>
int CircularList<T>::Search(const T& x) const
{// 在带有头节点的循环链表中寻找x
  ChainNode<T> *current = first->link;
  int index = 1; // current的索引
  first->data = x; // 把x放入头节点
// 查找x
while (current->data != x) {
  current = current->link;
  index++;
  }
 // 是链表表头吗？
 return ((current == first) ? 0 : index);
}
```

3.4.6 与公式化描述方法的比较

采用公式化描述方法的线性表仅需要能够保存所有元素的空间以及保存表长所需要的空

间，而链表和循环链表描述还需要额外的空间，用来保存链接指针（线性表中的每个元素都需要一个相应的链接指针）。采用链表描述所实现的插入和删除操作要比采用公式化描述时执行得更快。当每个元素都很长时（字节数多），尤其如此。

还可以使用链接模式来描述很多表，这样做并不会降低空间利用率，也不会降低执行效率。对于公式化描述，为了提高空间利用率，不得不把所有的表都放在一个数组中加以描述，并使用了另外两个数组来对这个数组进行索引，更有甚者，与一个表对应一个数组的情形相比，插入和删除操作变得更为复杂，而且存在一个很显著的最坏运行时间。

采用公式化描述，可以在$\Theta(1)$的时间内访问第k 个元素。而在链表中，这种操作所需要的时间为O(k)。

3.4.7 双向链表

对于线性表的大多数应用来说，采用链表和 /或循环链表已经足够了。然而，对于有些应用，如果每个链表元素既有指向下一个元素的指针，又有指向前一个元素的指针，那么在设计应用代码时将更为方便。双向链表（ doubly linked list）即是这样一个有序的节点序列，其中每个节点都有两个指针：left和right。left指针指向左边节点（如果有），right指针指向右边节点（如果有）。图3-9给出了线性表(1,2,3,4)的双向链表表示。

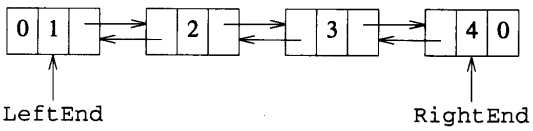

图3-9 一个双向链表

在C++中，可以采用程序 3-21中给出的类定义来描述双向链表。该定义所定义的函数都是基于链表左部的，如， Find(k, x) 从链表左部开始查找第k 个元素。当然，也可以定义基于链表右部的函数。

程序3-21 双向链表的类定义

```
template <class T>
class DoubleNode {
  friend Double<T>;
  private:
    T data;
    DoubleNode<T> *left, *right;
};
template<class T>
class Double {
    public:
    Double() {LeftEnd = RightEnd = 0;};
    ~Double();
    int Length() const;
    bool Find(int k, T& x) const;
    int Search(const T& x) const;
    Double<T>& Delete(int k, T& x);
    Double<T>& Insert(int k, const T& x);
    void Output(ostream& out) const;
    private:
    DoubleNode<T> *LeftEnd, *RightEnd;
};
```

通过在双向链表的左部和/或右部添加头节点，并且把链表变成循环的链表，可以提高双向链表的性能。在一个非空的双向链表中，LeftEnd->left是一个指向最右边节点的指针（即RightEnd），RightEnd->right是一个指向最左边节点的指针。所以，可以省去RightEnd，只需简单地使用变量LeftEnd来跟踪链表。

3.4.8 小结

本节引入了以下重要概念：

• 单向链表　令x是一个单向链表。当且仅当x.first=0时x为空。如果x不为空，则x.first指向链表的第一个节点。第一个节点指向第二个节点；第二个节点指向第三个节点，如此进行下去。最后一个节点的链指针为0。

• 单向循环链表　它与单向链表的唯一区别是最后一个节点又反过来指向了第一个节点。当循环链表x为空时，x.first=0。

• 头指针　这是在链表中引入的附加节点。利用该节点通常可以使程序设计更简洁，因为这样可以避免把空表作为一种特殊情况来对待。使用头指针时，每个链表（包括空表）都至少包含一个节点（即头指针）。

• 双向链表　双向链表由从左至右按序排列的节点构成。right指针用于把节点从左至右链接在一起，最右边节点的right指针为0。left指针用于把节点从右至左链接在一起，最左边节点的left指针为0。

• 双向循环链表　双向循环链表与双向链表的唯一区别在于，最左边节点的left指针指向最右边的节点，而最右边节点的right指针指向最左边的节点。

练习

22. 编写一个复制构造函数Chain<T>::Chain(const Chain<T>& C)，把链表C中的元素复制到新的节点中。这个构造函数的复杂性是多少？

23. 编写一个函数，把一个用数组表示的线性表转换成单向链表。要求利用LinearList的成员函数Find和Chain的成员函数Insert来实现。该函数的时间复杂性是多少？试测试代码的正确性。

24. 编写一个函数，把一个用单向链表表示的线性表转换成用数组表示的线性表。

1) 首先利用Chain的成员函数Find和LinearList的成员函数Insert来实现。该函数的时间复杂性是多少？试测试代码的正确性。

2) 利用链表遍历器来实现。函数的时间复杂性是多少？试用适当的测试数据来测试该函数的正确性。

25. 扩充Chain的类定义，把LinearList转换成Chain以及Chain转换成LinearList的函数作为成员函数添加到Chain的类定义之中。具体任务是编写函数FromList(L)和ToList(L)。FromList(L)把一个线性表L转换成单向链表，而ToList(L)把一个单向链表转换成线性表L。每个函数的时间复杂性分别是多少？试测试代码的正确性。

26. 试比较程序3-12和3-20中Search函数的运行性能。分别使用大小为100、1000、10 000和100 000的线性表比较最坏情况下的运行时间及平均运行时间。以表格和图的形式给出所得到的时间。

27. 1) 扩充Chain的类定义，增加函数Reverse，用于对x中的元素反序。要求反序操作就地进行，不需要分配任何新的节点。

2) 函数的时间复杂性是多少？

3) 通过编译和执行，测试函数的正确性。要求使用自己的测试数据。

28. 完成练习27，区别是Reverse 不作为Chain 的成员函数。要求利用 Chain的成员函数来完成反序操作。新的函数将拥有两个参数 A和B，A作为输入的链表，B是把A反序后得到的链表。在反序完成时，A变成一个空的链表。

29. 令A 和B 都是Chain 类型

1) 编写一个新的成员函数Alternate，用以创建一个新的线性表C，该表包含了A和B中的所有元素，其中A和B的元素轮流出现，表中的首元素为 A中的第一个元素。在轮流排列元素时，如果某个表的元素用完了，则把另一个表的其余元素依次添加在新表的后部。代码的复杂性应与两个输入表的长度呈线性比例关系。

2) 证明代码具有线性复杂性。

3) 通过编译和执行，测试函数的正确性。要求使用自己的测试数据。

30. 扩充Chain的类定义，增加一个与练习29中的Alternate函数相类似的函数Alternate。该函数应利用A和B中的物理节点来建立C。在执行完Alternate之后，A和B均变成空表。

1) 编写出Alternate的实现代码。代码的复杂性应与初始链表的长度呈线性关系。

2) 证明代码具有线性复杂性。

3) 通过编译和执行，测试函数的正确性。要求使用自己的测试数据。

31. 令A 和B 都是Chain 类型，假定A 和B 的元素都是按序排列的（即从左至右按递增次序排列）

1) 编写一个函数Merge，用以创建一个新的有序线性表C，该表中包含了A和B的所有元素。

2) 函数的时间复杂性如何？

3) 通过编译和执行，测试函数的正确性。要求使用自己的测试数据。

32. 重做练习31，要求函数是Chain的一个成员函数，并使用两个输入链表中的物理节点来建立新链表C，在Merge执行完之后，两个输入链表均变成空表。

33. 令C为Chain类型

1) 试编写函数Split，该函数用来创建两个单向链表A和B，A中包含C中所有奇数位置上的元素，B中包含其余的元素。函数不能修改线性表C。

2) 函数的时间复杂性如何？

3) 通过编译和执行，测试函数的正确性。要求使用自己的测试数据。

34. 为Chain类编写一个成员函数Split，该函数与练习33中的Split相类似，区别是本函数破坏输入链表，并采用输入链表中的节点来构造A和B。

35. 设计Circular类。该类的对象是如图3-8所示的循环链表，区别是没有头节点。必须实现为Chain类定义的所有函数（见程序 3-8）。每个函数的复杂性分别是多少？测试所编写代码的正确性。

36. 当每个表都有一个头节点时，完成练习35。

37. 用循环链表代替单向链表完成练习27。

38. 用循环链表代替单向链表完成练习29。

39. 用循环链表代替单向链表完成练习31。

40. 用循环链表代替单向链表完成练习33。

41. 令x 指向一个循环链表z 中的任意节点

1) 编写一个函数用来删除节点x 中的元素。提示：由于不知道哪个节点是 x 的左边相邻节

点，因此难以从链表中删除节点x；不过，为了删除x 中的元素，可以把x 的数据域用x 的下一个节点y 的数据域来替换，然后删除y 节点即可。

2) 所编写的函数的时间复杂性如何？

3) 通过编译和执行，测试函数的正确性。要求使用自己的测试数据。

42. 使用带有头节点的循环链表而不是普通的循环链表完成练习35。

43. 使用带有头节点的循环链表而不是普通的单向链表完成练习27。

44. 使用带有头节点的循环链表而不是普通的单向链表完成练习 29，要求释放额外的头节点。

45. 使用带有头节点的循环链表而不是普通的单向链表完成练习 31，要求释放额外的头节点。

46. 使用带有头节点的循环链表而不是普通的单向链表完成练习 33，要求分配一个新的节点，因为每个新链表都需要有一个头节点。

47. 使用双向链表代替循环链表完成练习35。

48. 使用双向链表代替普通的单向链表完成练习27。

49. 使用双向链表代替普通的单向链表完成练习29。

50. 使用双向链表代替普通的单向链表完成练习31。

51. 使用双向链表代替普通的单向链表完成练习33。

52. 使用带有一个头节点的双向循环链表完成练习35。

53. 使用带有一个头节点的双向循环链表完成练习27。

54. 使用带有一个头节点的双向循环链表完成练习29，要求释放额外的头节点。

55. 使用带有一个头节点的双向循环链表完成练习31，要求释放额外的头节点。

56. 使用带有一个头节点的双向循环链表完成练习 33，要求分配一个新的节点，因为每个新链表都需要有一个头节点。

57. 为了有效地支持在一个双向链表中进行前移和后移，需要扩充Double的类定义（见程序3-21），即增加一个私有成员current，用它来纪录链表的当前位置，为此，需要增加以下共享函数：

1) ResetLeft —— 将current置为LeftEnd。

2) ResetRight —— 将current置为RightEnd。

3) current(x) —— 取当前元素至x。如果current>length，函数返回false，否则返回true。

4) End —— 如果当前的位置恰好指向链表的最后一个元素（即最右边的元素），则返回true，否则返回false。

5) Front —— 如果当前的位置恰好指向链表的第一个元素（即最左边的元素），则返回true，否则返回false。

6) Next —— 移动current指向链表的下一个元素。如果没有下一个元素，函数返回 false，否则返回true。

7) Previous —— 移动current指向链表的前一个元素。如果没有前一个元素，函数返回 false，否则返回true。

编写以上扩充函数，使用适当的测试数据测试代码的正确性。

58. 为Chain增加成员函数InsertionSort，该函数可利用程序2-15给出的插入排序算法对链表中的元素按递增次序进行重新排列。不得创建新的节点或删除老的节点。

1) 程序在最坏情况下的时间复杂性是多少？如果链表中的元素已经按递增次序排列，程序需要消耗多长时间？

2) 通过编译和执行，测试程序的正确性。要求使用自己的测试数据。

59. 分别采用以下排序算法（详见第2章）完成练习58：

1) 冒泡排序。

2) 选择排序。

3) 计数排序。

3.5 间接寻址

3.5.1 基本概念

间接寻址（indirect addressing）是公式化描述和链表描述的组合。采用这种描述方法，可以保留公式化描述方法的许多优点——可以根据索引在 $\Theta(1)$的时间内访问每个元素、可采用二叉搜索方法在对数时间内对一个有序表进行搜索等等。与此同时，也可以获得链表描述方法的重要特色——在诸如插入和删除操作期间不必对元素进行实际的移动。因此，大多数间接寻址链表操作的时间复杂性都与元素的总数无关。

在间接寻址方式中，使用一个指针表来跟踪每个元素。可采用一个公式（如公式（ 3-1））来定位每个指针的位置，以便找到所需要的元素。元素本身可能存储在动态分配的节点或节点数组之中。图3-10给出了一个采用间接寻址表table 描述的5元素线性表。其中 table[i]是一个指针，它指向表中的第i+1个元素，length 是表的长度。

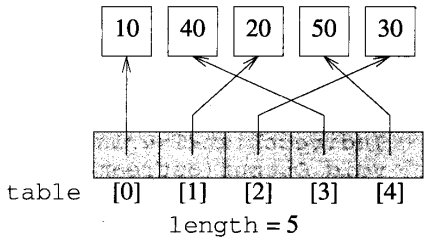

图3-10 间接寻址

尽管可以使用公式（ 3-1）来定位指向表中第 i 个元素的指针，但这个公式本身并不能直接定位第 i 个元素。在对表元素的寻址模式中 table提供了一级“ 间接 ”引用。

如果将图 3-10的间接寻址方式与图 3-5的链表描述方式进行比较，可以发现，二者都使用了指针域（或链接域）。在链表方式中，指针位于每个节点中，而在间接寻址方式中，指针全放在数组table之中，就像是一本书的目录索引一样。为了找到一本书中的某一项内容，首先需要查目录索引，索引会告诉我们该项内容在哪里。

当元素存储在动态分配的节点中时，相应的类定义见程序 3-22。私有成员有： table、length和MaxSize。table是一个指针数组，用来指向类型为 T的元素；MaxSize是指针数组的大小，其省缺值为10；length是表的当前长度。

程序3-22 间接寻址表的类定义

```
template<class T>
class IndirectList {
  public:
    IndirectList(int MaxListSize = 10);
    ~IndirectList();
    bool IsEmpty() const {return length == 0 ;}
    int Length() const {return length;}
    bool Find(int k, T& x) const;
    int Search(const T& x) const;
    IndirectList<T>& Delete(int k, T& x);
```

```
    IndirectList<T>& Insert(int k, const T& x);
    void Output(ostream& out) const;
  private:
    T **table; // 一维T类型指针数组
    int length, MaxSize;
}
```

3.5.2 操作

程序3-23给出了构造函数和析构函数的代码。为了创建一个大小不会超过 20的空的整数线性表x，可以采用如下的语句：

```
IndirectList<int> x(20);
```

程序3-23 间接寻址的构造函数和析构函数

```
template<class T>
IndirectList<T>::IndirectList(int MaxListSize)
{// 构造函数
  MaxSize = MaxListSize;
  table = new T *[MaxSize];
  length = 0;
}
template<class T>
IndirectList<T>::~IndirectList()
{// 删除表
  for (int i = 0; i < length; i++)
    delete table[i];
  delete [ ] table;
}
```

表x的长度可由length给出，在程序3-22中函数Length被定义为一个内联函数， IsEmpty也被定义为一个内联函数。假定1≤k≤length，第k个元素可由table[k-1]之后的下一个指针指出。对x进行搜索可通过依次检查指针 table[0]、table[1]、...所指向的元素。程序 3-24给出了函数Find的代码。函数 Length、IsEmpty和Find的时间复杂性均为 Θ(1)。注意观察这些代码与类LinearList相应代码之间的相似性。

程序3-24 间接寻址的Find函数

```
template<class T>
bool IndirectList<T>::Find(int k, T& x) const
{//取第k个元素至x
 //如果不存在第k个元素，函数返回false，否则返回 true
  if (k < 1 ||  k > length) return false; // 不存在第k个元素
  x = *table[k - 1];
  return true;
}
```

为了从图 3-10的表中删除第 3个元素，需要释放由第 3个元素所占用的空间，即把指针table[3:4]移动至table[2:3]，并将length减1。程序3-25是与程序3-4和程序3-14相对应的、进行间

接寻址删除操作的函数。请注意程序3-4和程序3-25之间的相似性。程序3-14和程序3-25的时间复杂性均与表的大小无关。不管每个表元素的大小是10个字节还是1000个字节，这些函数删除一个元素所需要的时间均相同。对于程序3-4来说，删除长度为1000字节的元素要比删除长度为10个字节的元素花费更多的时间，因为它需要移动表元素，每次移动将花费 $\Theta(s)$的时间，其中s为一个元素的大小。

程序3-25 从间接寻址表中删除元素

```
template<class T>
IndirectList<T>& IndirectList<T>::Delete(int k, T& x)
{ //把第k个元素传送至x，然后删除第k个元素
  //如果不存在第k个元素，则引发异常 OutOfBounds
  if (Find(k, x)) {//向前移动指针 k+l, ...
     for (int i = k; i < length; i++)
        table[i-l] = table[i];
     length--;
     return *this;
   }
 else throw OutOfBounds();
}
```

假定要在图3-10的表中第2和第3个元素之间插入一个元素x，需要建立如图3-11所示的结构。方法是首先将指针table[2:4]向右移动一个位置，然后在table[2]中填入一个指向y的指针。程序3-26可在线性表x的第k个元素之后插入一个元素，该程序在最坏情况下的时间复杂性与程序3-15（向链表中插入一个元素）完全相同，都是O(length)。而程序3-5中基于公式描述的插入函数所拥有的时间复杂性为O(s*length)，其中s是一个类型为T的元素的大小。

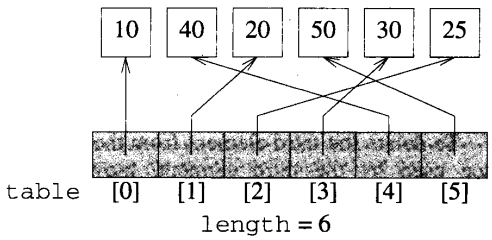

图3-11 向间接寻址表中插入元素

程序3-26 向间接寻址表中插入元素

```
template<class T>
IndirectList<T>& IndirectList<T>
            ::Insert(int k, const T& x)
{//在第k个元素之后插入x
 //如果不存在第k个元素，则引发异常 OutOfBounds
 //如果没有足够的空间，则传递NoMem异常
 if (k < 0 || k > length) throw OutOfBounds();
 if (length == MaxSize) throw NoMem();
 //向后移动一个位置
  for (int i = length-i; i >= k; i--)
     table[i+l] = table[i];
  table[k] = new T;
  *table[k] = x;
  length++;
```

```
    return *this;
}
```

练习

60. 编写IndirectList 的成员函数Output，并利用该函数重载操作符 <<。测试代码的正确性。

61. 编写IndirectList的成员函数Search，测试其正确性，并指出函数的时间复杂性。

62. 设计一个间接寻址的类遍历器，参照ChainIterator类（见程序3-18）的定义。通过用它从左至右输出一个线性表来测试其正确性。

63. 1) 编写一个折半搜索（见程序2-30）函数，用来对一个间接寻址表进行搜索。假定对于所有的i≤length-2有 *table[i]≤*table[i+1]。函数的时间复杂性应该是O(log(length))，试证明之。

2) 通过编译和执行，测试该程序的正确性。要求使用自己的测试数据。

3) 搜索一个有序的单向链表速度会有多快？试编写一个具有这种时间复杂性的，对单向链表进行搜索的函数。

64. 令x为IndirectList类型的对象

1) 编写一个排序函数，使 x按递增次序排列，即对于所有的 i≤length-2，有*table[i] ≤ table[i+l]。基于插入排序方法（见程序 2-15）实现该函数。函数的时间复杂性应该是O(length2)，并且与每个元素的大小无关。试证明之。

2) 通过编译和执行，测试该程序的正确性。要求使用自己的测试数据。

65. 分别采用以下排序算法（详见第2章）完成练习64：

1) 冒泡排序。

2) 选择排序。

3) 计数排序。

66. 给定一个类型为T的数组element [0:length-l]和一个整数数组table[0:length-l]。table[]是[0, 1, ..., length-1]的一种排列，使得对于 0≤i≤length-2有element [table[i]]≤element [table [i+l]]。

1) 编写一个函数对element[]进行排序，使得对于所有的i,有element [i]≤element [i+l]。函数的时间复杂性应该是O(*s*length)，其中*s* 为每个元素的大小，函数的空间复杂性应该是O(*s*)。试证明之。

2) 测试函数的正确性。

3.6 模拟指针

在大多数应用中，可以利用动态分配及 C++指针来实现链表和间接寻址表。不过，有时候采用一个节点数组以及对该数组进行索引的模拟指针（simulated pointer），可以使设计更方便、更高效。

假定采用一个数组node，该数组的每个元素中都包含两个域： data和link。数组中的节点分别是：node[0]、 node[l]、...、node[NumberOfNodes-l]。以下用节点i来代表node[i]。如果一个单向链表c由节点10，5和24按序构成，将得到c=10(指向链表c的第一个节点的指针是整数类型)，node[10].link=5(指向第二个节点的指针)，node[5].link=24(指向下一个节点的指针)，node[24].link=-1(表示节点24是链表中的最后一个节点)。在绘制链表时，可以把每个链接指针

画成一个箭头（如图3-12所示），与使用C++指针的时候一样。

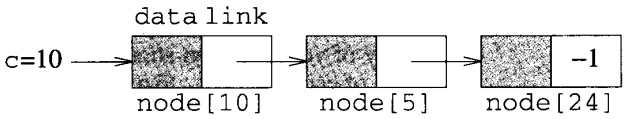

图3-12 采用模拟指针的链表

为了实现指针的模拟，需要设计一个过程来分配和释放一个节点。当前未被使用的节点将被放入一个存储池（storage pool）之中。开始时，存储池中包含了所有节点 node[0：NumberOfNodes-l]。Allocate从存储池中取出节点，每次取出一个。 Deallocate则将节点放入存储池中，每次放入一个。因此， Allocate 和Deallocate 分别对存储池执行插入和删除操作，等价于C++函数delete和new。如果存储池是一个节点链表（如图 3-13所示），这两个函数可以高效地执行。用作存储池的链表被称之为可用空间表（ available space list），其中包含了当前未使用的所有节点。first 是一个类型为int 的变量，它指向可用空间表中的第一个节点。添加和删除操作都是在可用空间表的前部进行的。

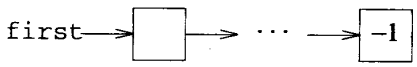

图3-13 可用空间表

为了实现一个模拟指针系统，定义了SimNode类和SimSpace类，见程序3-27。

程序3-27 模拟指针的类定义

```
template <class T>
class SimNode {
    friend SimSpace<T>;
    private:
    T data;
       int link;
};
template <class T>
class SimSpace {
    public:
       SimSpace (int MaxSpaceSize=100);
       ~SimSpace() {delete [] node;}
       int Allocate(); //分配一个节点
       void Deallocate (int& i) ; //释放节点i
    private:
       int NumberOfNodes, first;
       SimNode<T> *node;//节点数组
};
```

3.6.1 SimSpace的操作

由于所有节点初始时都是自由的，因此在刚被创建的时候，可用空间表中包含NumberOfNodes个节点。程序3-28用来对可用空间表进行初始化。程序 3-29和程序3-30分别实现Allocate和Deallocate操作。

程序3-28 初始化可用空间表

```
template<class T>
SimSpace<T>::SimSpace(int MaxSpaceSize)
{//构造函数
  NumberOfNodes = MaxSpaceSize;
  node = new SimNode<T> [NumberOfNodes];
  //初始化可用空间表
  //创建一个节点链表
  for (int i = 0; i < NumberOfNodes-l; i++)
    node[i].link = i+l;
  //链表的最后一个节点
  node[NumberOfNodes-l].link = -1;
  //链表的第一个节点
  first = 0;
}
```

程序3-29 使用模拟指针分配一个节点

```
template<class T>
int SimSpace<T>::Allocate()
{// 分配一个自由节点
  if (first == -1) throw NoMem();
  int i = first; //分配第一个节点
  first = node[i].link; //first指向下一个自由节点
  return i;
}
```

程序3-30 使用模拟指针释放一个节点

```
template<class T>
void SimSpace<T>::Deallocate(int& i)
{// 释放节点i.
  //使 i 成为可用空间表的第一个节点
  node[i].link = first;
  first = i;
  i = -1;
}
```

以上三个函数的时间复杂性分别为Θ(NumberOfNodes)，Θ(1)和Θ(1)。通过使用两个可用空间表，可以减少构造函数（见程序3-28）的运行时间，其中第一个表包含所有尚未被使用的自由节点，第二个表包含所有已被至少使用过一次的自由节点。每当一个节点被释放时，被放入第二个表中。当需要一个新节点时，如果第二个表非空，则从该表中取出一个节点，否则从第一个表中取出一个节点。令first1和first2分别指向第一个表和第二个表的首节点。基于上述分配节点的方式，第一个表中的元素为node[i]，其中first1$\leq$ i＜NumberOfNodes。释放一个节点的代码与程序3-30的唯一区别在于将所有first变量均替换成first2。新的构造函数和分配函数分别见程序3-31和程序3-32。为了使新函数能正常工作，需要把整型变量first1和first2设置为SimSpace的私有成员变量。

程序3-31 使用两个可用空间表的构造函数

```
template<class T>
SimSpace<T>::SimSpace(int MaxSpaceSize)
{ // 使用两个可用空间表的构造函数
  NumberOfNodes = MaxSpaceSize;
  node = new SimNode<T> [NumberOfNodes];
  //初始化可用空间表
  firstl = 0;
  first2 = -1;
}
```

程序3-32 使用两个可用空间表的Allocate函数

```
template<class T>
int SimSpace<T>::Allocate()
{//分配一个自由节点
  if (first2 == -1) {// 第2个表为空
     if (firstl == NumberOfNodes) throw NoMem();
     return firstl++;}
  //分配链表中的第一个节点
  int i = first2;
  first2 = node[i].link;
  return i;
}
```

我们希望在大多数应用中，由于使用了两个可用空间表，程序 3-31和程序3-32应该比只使用一个可用空间表的程序提供更好的性能。为此做以下的考察：

• 程序3-32所花费的时间与程序3-29所花费的时间相同，除非节点是从第一个表中取出的。这种例外至多发生 NumberOfNodes 次。在这种例外情况下额外花费的时间将抵销了初始化所节省出的时间。事实上，需要的节点数通常少于 NumberOfNodes 个（尤其是在调试程序以及解决实例特征变化较大的问题的时候），因此两个可用空间表模式将运行得更快。

• 在一个交互式的环境中减少初始化所需要的时间是很受欢迎的，因为程序的启动时间将会大大减少。

• 在只使用一个可用空间表时，对于所建立的单向链表，除了最后一个节点外，没有必要明确指定节点的链接指针，因为节点中已经体现了正确的链接值（如图 3-13所示）。这种优点也可以引入使用两个可用空间表的模式中，方法是编写一个 Get(n)函数，它产生一个有n个节点的单向链表。仅当从第一个表中取出一个节点时， Get(n)函数才明确地设置链接域的值。

• 采用可用空间表模式分解一个链表将比采用 C++指针更高效。例如，如果一个单向链表的首部和尾部分别为f 和e，可以采用如下语句来释放链表中的所有节点：

```
node[e].link = first; first = f;
```

• 如果c 是一个循环链表，则采用程序 3-33释放表中所有节点需要的时间为 $\Theta(1)$。图3-14给出了链接指针所发生的变化。

程序3-33 释放一个循环链表

```
template<class T>
void SimSpace<T>::DeallocateCircular(int& c)
{// 释放一个循环链表c
  if(c != -1 ) {
    int next = node[c].link;
    node[c].link = first;
    first = next;
    c = -1;
   }
}
```

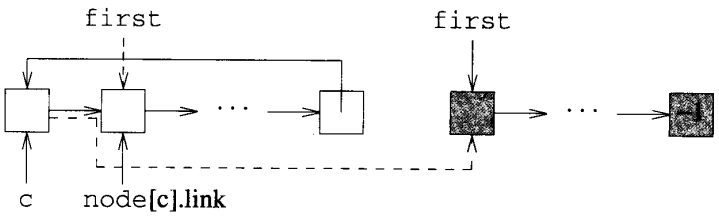

图3-14 释放一个循环链表

3.6.2 采用模拟指针的链表

可以使用模拟空间S来定义一个链表类（见程序3-34）。S被说明为一个static成员，目的是使所有类型为T的模拟链表共享相同的模拟空间。程序3-35至程序3-38给出了除Search和Output之外的各共享函数的代码。这些代码假定SimChain已经被说明为SimNode和SimSpace的友元。注意观察这些代码与Chain各相应成员函数之间的相似性。程序3-39给出了使用模拟链表的示例程序。在该程序中，simul.h和schain.h分别包含了SimSpace类和SimChain类的代码。

程序3-34 模拟链表的类定义

```
template<class T>
class SimChain {
  public:
    SimChain() {first = -1;}
    ~SimChain() {Destroy();}
    void Destroy(); // 使表为空
    int Length() const;
    bool Find(int k, T& x) const;
    int Search(const T& x) const;
    SimChain<T>& Delete(int k, T& x);
    SimChain<T>& Insert(int k, const T& x);
    void Output(ostream& out) const;
  private:
    int first; // 第一个节点的索引
    static SimSpace<T> S;
};
```

程序3-35 模拟指针的构造函数和Length函数

```
template<class T>
void SimChain<T>::Destroy()
{// 释放链表节点
  int next;
  while (first != -1) {
    next = S.node[first].link;
    S.Deallocate(first);
    first = next;}
}
template<class T>
int SimChain<T>::Length() const
{// 返回链表的长度
 int current = first；//链节点的当前位置
   len = 0;        //元素计数
 while (current != -1) {
    current = S.node[current].link;
    len++;}
 return len;
}
```

程序3-36 模拟指针的Find函数

```
template<class T>
bool SimChain<T>::Find(int k, T& x) const
{// 取第k个元素至x
  //如果不存在第k个元素，函数返回false，否则返回true
  if (k < 1) return false;
  int current = first, // 链节点的当前位置
    index = 1;      //当前节点的索引
  //移动current至第k个节点
  while (index < k && current != -1) {
    current = S.node[current].link;
    index++;}
  //验证是否到达了第k个节点
  if (current != -1) {x = S.node[current].data; return true;}
  return false; // 不存在第k个元素
}
```

程序3-37 模拟指针的Delete函数

```
template<class T>
SimChain<T>& SimChain<T>::Delete(int k, T& x)
{//把第k个元素取至x，然后删除第k个元素
 //如果不存在第k个元素，则引发异常 OutOfBounds
 if (k < 1 || first == -1)
    throw OutOfBounds(); // 不存在第k个元素
 // p最终将指向第k个节点
```

```
  int p = first;
  //将p移动至第k个节点，并从链表中删除该节点
  if (k == 1) // p已经指向第k个节点
     first = S.node[first].link; // 从链表中删除
  else { // 使用q指向第k-1个元素
     int q = first;
     for (int index = 1; index < k - 1 && q != -1; index++)
        q = S.node[q].link;
  // 验证第k个元素的存在性
  if (q == -1 || S.node[q].link == -1)
    throw OutOfBounds(); // 不存在第k个元素
  //使p指向第k个元素
  p = S.node[q].link;
  //从链表中删除第k个元素
  S.node[q].link = S.node[p].link;
  }
  //保存第k个元素并释放节点p
  x = S.node[p] .data;
  S.Deallocate(p);
  return *this;
}
```

程序3-38 模拟指针的Insert函数

```
template<class T>
SimChain<T>& SimChain<T>::Insert(int k, const T& x)
{//在第k个元素之后插入x
 //如果不存在第k个元素，则引发异常 OutOfBounds
 //如果没有足够的空间，则传递NoMem异常
  if (k < 0) throw OutOfBounds();
  //定义一个指针p，p最终将指向第k个节点
  int p = first;
  //将p移向第k个节点
  for (int index = 1; index < k && p != -1; index++)
     p = S.node[pl.link;
  // 验证第k个节点的存在性
  if (k > 0 && p == -1)
     throw OutOfBounds();
  // 为插入操作分配一个新节点
  int y = S.Allocate();
  S.node[y].data = x;
  //向链表中插入新节点
  // 首先检查新节点是否要插到链表的首部
  if (k) {//在p之后插入
      S.node[y].link = S.node[p].link; S.node[p].link = y;}
  else {// 作为链表首节点
      S.node[y].link = first; first = y;}
  return *this;
}
```

程序3-39 使用模拟链表

```
#include <iostream.h>
#include "schain.h"
SimSpace<int> SimChain<int>::S;
void main(void)
{
    int x;
    SimChain<int> c;
    cout << "Chain length is" << c.Length( ) << endl;
    c.Insert (0, 2).Insert (1, 6);
    cout << "Chain length is" << c.Length( ) << endl;
    c.Find (1,x);
    cout <<"First element is" << x << endl;
    c.Delete (l,x);
    cout << "Deleted" << x << endl;
    cout << "New length is" << c.Length ( ) << endl;
    cout << "Position of 2 is" << c.Search (2) << endl;
    cout << "Position of 6 is" << c.Search (6) << endl;
    c.Insert (0, 9).Insert (1, 8). Insert (2, 7);
    cout << "Current chain is" << c << endl;
    cout << "Its length is" << c.Length() << endl;
}
```

练习

67. 为SimChain类设计一个类遍历器SimIterator。请参考程序3-18中关于Chains类遍历器的定义。SimIterator中应包含与ChainIterator 相同的成员函数。编写并测试代码。

68. 1) 修改SimSpace类，使得 Allocate返回一个指向node[i]的指针，而不是返回索引值 i。类似地，修改Deallocate函数，使它的输入参数为欲释放节点的指针。

2) 采用1) 中的SimSpace代码重新编写SimChain的实现代码。请留意新代码与Chain的代码之间的相似性。

69. 1) 修改SimNode类的定义，为它添加一个类型为SimSpace<T>的静态成员S。这样，所有类型为SimSpace<T>的节点都可以共享同样的模拟空间。重载函数 new和delete，以便从模拟空间S中取得节点SimNodes或将SimNodes节点送回模拟空间S中。

2) 假定SimSpace是按照练习68的要求实现的， SimNode是按照1) 中要求实现的。修改Chain类的代码（见程序3-8），使得它能够用SimNodes代替ChainNodes进行工作。测试代码，并测量运行时间，以确定哪个版本的Chain更快。

70. 假定一个链表是采用模拟指针进行描述的，节点的类型为 SimNode

1) 编写一个程序，该程序使用插入排序算法对链表中的节点进行重新排序，要求按照 data域的递增次序进行排列。

2) 代码的时间复杂性是多少？如果不是 $O(n^2)$，请重写代码以使其具有这样的复杂性，其中n 为链表长度。

3) 测试代码的正确性。

71. 使用选择排序算法完成练习70。

72. 使用冒泡排序算法完成练习70。

73. 使用计数排序算法完成练习70。

74. 对new和delete的调用通常都要耗费很多时间，为此，可以使用自行编写的释放函数（该函数能将被删除的节点放入自由节点表中）来替换 delete函数，以便提高代码的运行效率。为了替换new，可以自行编写一个分配函数，每当自由节点链表为空时该函数才会去调用 new。修改Chain类（见程序3-8）以实现上述思想。要求编写如上所述的分配节点和释放节点函数，并对自由节点链表进行初始化。试比较两种Chain版本的执行时间，并评价新方法的优缺点。

75. 考察按如下方式定义的XOR操作（异或操作，也可以记为⊕）：

$$i \oplus j = \begin{cases} 0 & i = j \\ 1 & i = j \end{cases}$$

两个二进制串i 和j 的XOR操作结果由i 和j 各相应位的XOR结果组成。例如，如果i=10110且j=01100，则iXORj=i⊕j=11010。注意有：

$$a \oplus (a \oplus b) = (a \oplus a) \oplus b = b$$
$$(a \oplus b) \oplus b = a \oplus (b \oplus b) = a$$

以上规律提供了一种节省双向链表左、右链接指针所需存储空间的措施。假定可用的节点都放在数组node之中，节点的索引号分别为1，2，...。因此，node[0]尚未使用。现在可以用0而不是-1来表示NULL指针。每个节点都有两个域：data和link。如果l 是节点x 的左指针，r 是其右指针。对于最左边的节点，l=0，而对于最右边的节点，r=0。令（l, r）是按上述方法实现的双向链表，l 指向链表最左边的节点，r 指向最右边的节点。

1) 编写一个函数从左至右遍历双向链表(l,r)，并输出每个节点data域的内容。

2) 编写一个函数从右至左遍历双向链表(l,r)，并输出每个节点data域的内容。

3) 测试代码的正确性。

3.7 描述方法的比较

在图3-15中，分别给出了使用本章所介绍的四种数据描述方法执行各种链表操作所需要的时间复杂性。在表中，s 和n 分别表示sizeof(T)和链表长度。由于采用C++指针和采用模拟指针完成这些操作所需的时间复杂性完全相同，因此表中在把这两种情形的时间复杂性合并在同一行中进行描述。

描述方法	操作		
	查找第k 个元素	删除第k 个元素	在第k 个元素后插入
公式化描述 以公式（3-1）为例	$\Theta(1)$	$O((n-k)s)$	$O((n-k)s)$
链表描述 C++ 及(模拟指针)	$O(k)$	$O(k)$	$O(k+s)$
间接寻址	$\Theta(l)$	$O(n-k)$	$O(n-k)$

图3-15 四种描述方法的比较

使用间接寻址与使用链表描述所需要的空间大致相同，二者都比使用公式化描述所需要的

空间更多。不管是使用链表描述还是间接寻址，执行链表的插入和删除操作所需要的时间复杂性均与每个链表元素本身的大小无关。然而，在使用公式化描述时，插入和删除操作的复杂性与元素本身的大小成线性关系。所以，如果链表元素的大小 s 很大，那么使用链表描述和间接寻址将更适合于需要大量插入和删除操作的应用。

在使用公式化描述和间接寻址时，确定表的长度以及访问表中第 k 个元素所需要的时间复杂性均为$\Theta(1)$。而在使用链表描述时，这些操作的时间复杂性分别为Θ(length)和O(k)，所以链表描述不适合于这两种操作占优势的应用。

基于以上的讨论可以发现，间接寻址比较适合这样的应用：表元素本身很大，较频繁地进行插入、删除操作以及确定表的长度、访问第k个元素。同时，如果线性表本身已经按序排列，那么使用公式化描述或间接寻址进行搜索所需要时间均为 O(log$n$)，而使用链表描述时，所需要的时间为O(n)。

3.8 应用

3.8.1 箱子排序

假定一个链表中包含了一个班级内所有学生的信息，每个节点中含有这样的域：学生姓名、社会保险号码、每次作业和考试的分数以及所有作业和考试的加权总分。假定所有的分数均为0~100范围内的整数。如果采用第2章中所给出的任一种排序算法对表中的学生按分数进行排序，所需要花费的时间均为O(n^2)，其中n 为班级中的学生总数。一种更快的排序方法为箱子排序(bin sort)。在箱子排序过程中，节点首先被放入箱子之中，具有相同分数的节点都放在同一个箱子中，然后通过把箱子链接起来就可以创建一个有序的链表。

图3-16a 给出了一个箱子排序的例子，图中的链表含有 10个节点。该图仅列出了每个节点的姓名域和分数域。第一个域为姓名，第二个域为分数。为简便起见，假定每个姓名为一个字符，分数则介于0到5之间。需要六只箱子来分别存放具有 0~5之间某种分数的节点。图 3-16b给出了10个节点按分数分布于各个箱子中的情形。通过沿链表逐个检查每个节点，即可得到这

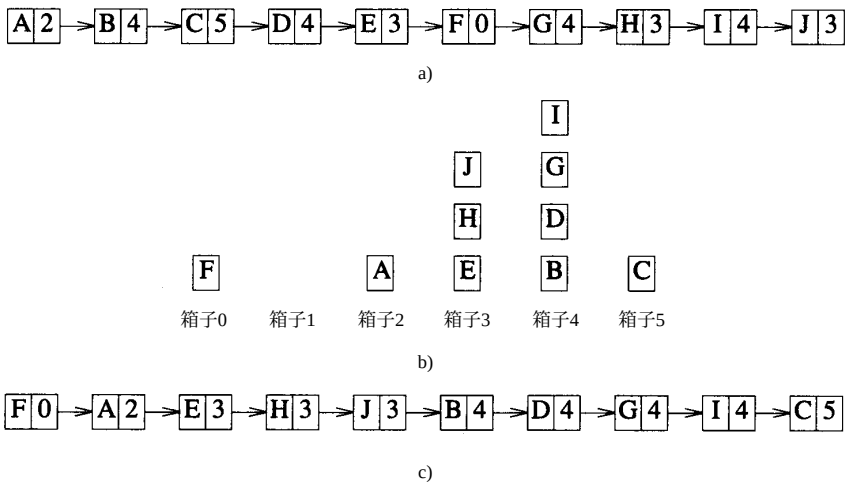

图3-16 箱子排序举例

a) 输入链表 b) 箱子中的节点 c) 排序后的链表

样的分布。当检查某个节点时，该节点被放入与它的分数相对应的那个箱子之中。所以第一个节点被放入2号箱子，第二个节点被放入4号箱子，依此类推。现在，如果从0号箱子开始收集节点，将得到一个如图3-16所示的有序链表。

怎样实现箱子呢？注意到每个箱子都是一个由节点组成的线性表。箱子中的节点数目介于0到n之间。一种简单的方法就是把每个箱子都描述成一个链表。在进行节点分配之前，所有的箱子都是空的。

对于箱子排序，需要能够：1)从欲排序链表的首部开始，逐个删除每个节点，并把所删除的节点放入适当的箱子中（即相应的链表中）；2) 收集并链接每个箱子中的节点，产生一个排序的链表。如果所输入的链表为Chain类型（见程序3-8），那么可以：1) 连续地删除链表首元素并将其插入到相应箱子链表的首部；2) 逐个删除每个箱子中的元素（从最后一个箱子开始）并将其插入到一个初始为空的链表的首部。

链表节点的数据域应该是Node类型（见程序3-40）。操作符!=和<<已经被重载，因为Chain类需要使用这两个操作符。

程序3-40 一种用于箱子排序的节点类

```
class Node {
  friend ostream& operator<<(ostream&, const Node &);
  public:
     int operator !=(Node x) const
       {return (score!= x.score);}
  private:
     int score;
     char *name;
};
ostream& operator<<(ostream& out, const Node& x)
   {out << x.score << '  '; return out;}
```

另一种可选的重载方式是提供一种从Node类型到数字类型的转换，因为数字类型可用于比较和输出。例如，可以重载类型转换操作符 int()，见程序3-41。那些在Node类型中未定义的算术和逻辑操作符（比如 +, /, <=, != 和输出操作符<<等）现在可以通过首先执行一个类型转换（从定义这些操作符的类型向int 类型）而成功地完成相应的操作。这种解决方法与前面介绍的重载操作符!= 和<< 相比可能会更通用，因为如果这样做，即使扩充了Chain类，为该类增添了对T -> data 进行其他操作的函数，这些类型转换程序仍可以正常工作。

程序3-41 另一种处理操作符重载的方法

```
class Node {
  public:
     //重载类型转换操作符
     operator int() const (return score;)
  private:
     int score;
     char *name;
};
```

上述两种重载方法可以联合使用，以便于仅当缺少类型转换操作符而导致失败时才去执行

int类型转换。因此，可以采用程序3-42中的定义。其中，向int类型的转换仅针对除!=和<<之外的操作符。

程序3-42 又一种处理操作符重载的方法

```
class Node {
  friend ostream& operator<<(ostream&, const Node &);
  public:
     int operator !=(Node x) const
     {return (score != x.score
        || name[0] != x.name[0];}
     operator int() const {return score;}
  private:
     int score;
     char *name;
};
ostream& operator<<(ostream& out, const Node& x)
{
  out << x.score << '  ' << x.name[0] << '  ';
  return out;
}
```

程序3-43给出了箱子排序函数的代码，该函数假定 BinSort是Node的一个友元。如果没有足够的空间来创建箱子，该函数允许传递 NoMem异常。在两个for循环中执行的每一次插入和删除操作所需要的时间均为 $\Theta(1)$，因此第一个for循环的的复杂性为 $\Theta(n)$，其中n 为输入链表的长度，第二个for循环的复杂性为(n+range)，因此函数BinSort总的复杂性为(n+range)(如果成功的话)。

程序3-43 箱子排序

```
void BinSort(Chain<Node>& X, int range)
{// 按分数排序
   int len = X,Length();
   Node x;
   Chain<Node> *bin;
   bin = new Chain<Node> [range + 1];
  //分配到每个箱子中
  for (int i = 1; i <= len; i++) {
     X,Delete(1,x);
     bin[x.score].Insert(0,x);
     }
  //从箱子中收集各元素
  for (int j = range; j >= 0; j--)
    while (!bin[j].IsEmpty()) {
      bin[j].Delete(1,x);
      X.Insert(0,x);
      }
  delete [ ] bin;
}
```

1. 把BinSort定义成Chain类的成员

细心的读者可能已经注意到，如果把 BinSort定义成Chain的一个成员函数，可以大大简化BinSort函数。当一个元素是链表中的成员并被放入箱子中时，这种方法能够让我们使用相同的物理节点，而且这种方法还可以消除所有对new和delete的调用（与bin相关的那些调用除外）。此外，通过跟踪每个箱子链表的首节点和尾节点，可以链接处于“收集状态”的箱子链表。见程序3-44。

程序3-44 Binsort作为Chain类的成员

```
template<class T>
void Chain<T>::BinSort(int range)
{// 按分数排序
  int b; // 箱子索引号
  ChainNode<T> **bottom, **top;
  //箱子初始化
  bottom = new ChainNode<T>* [range + 1];
  top = new ChainNode<T>* [range + 1];
  for (b = 0; b <= range; b++)
    bottom[b] = 0;
 //把节点分配到各箱子中
 for (; first; first = first->link) {// 添加到箱子中
   b = first->data;
   if (bottom[b]) {//箱子非空
     top[b]->link = first;
     top[b] = first;}
   else // 箱子为空
    bottom [ b ] = top[b] = first;
   }
  //收集各箱子中的元素，产生一个排序链表
 ChainNode<T> *y = 0;
 for (b = 0; b <= range; b++)
   if (bottom[b]) {//箱子非空
     if (y) // 不是第一个非空的箱子
        y->link = bottom[b];
     else // 第一个非空的箱子
        first = bottom[b];
     y = top[b];}
 if (y) y->link = 0;
 delete [] bottom;
 delete [] top;
}
```

对应于每个箱子的链表都是以箱子的底部节点作为首节点，其他节点依次排列直至箱子的顶部节点。每个箱子链表都有两个指针：bottom和top，它们均指向该链表。bottom[b]指向箱子b的底部节点，而 top[b]指向箱子 b 的顶部节点。所有空箱子的初始结构可以定义为bottom[b]=0（见程序3-44的第一个for循环）。在检查节点时，每个节点均被添加到相应箱子的顶部（见程序3-44的第二个for循环）。在第二个for循环的代码中，为了能返回score域，假定已经定义了从Node到int类型的转换。第三个for循环从0号箱子开始进行检查，依次把每个非空的

箱子链接在一起以产生一个有序的链表。

至于BinSort的时间复杂性，可以看到，第一和第三个 for循环所需要的时间为 Θ(range)，第二个for 循环所需要的时间为Θ(n)，因此总的时间复杂性为Θ(n+range)。

可以注意到BinSort函数并未改变具有同样分数的节点之间的相对次序。例如，假定在输入链中E、G和H的分数均为3，E出现在G之前，G出现在H之前，那么，在排序后的链表中，E仍出现在G之前，G也仍然出现在H之前。在有些排序应用中，要求排序算法不得改变同值元素之间的相对次序。如果一个排序算法能够保持同值元素之间的相对次序，则该算法被称之为稳定排序（stable sort）。

2. 概括

假定Node的每个元素都含有 exam1，exam2，exam3以及其他附加的域。在某些程序中，我们可能希望按exam1域进行排序，之后某个时刻可能又希望按 exam3域进行排序，再之后的某个阶段又可能需要按照 exam1+exam2+exam3进行排序。如果我们定义了数据类型 Node1，Node2和Node3，那么可以利用程序 3-44中的代码来执行这三种排序。在 Node1中，可将函数int()定义为返回exam1域的值，在Node2中则定义为返回exam2域的值，在Node3中定义为返回exam1+ exam2+exam3的值。在调用BinSort之前，必须把欲排序的数据复制到类型为 Node1或Node2或Node3（取决于所使用的排序值）的链表之中。

为了避免复制链表元素所带来的额外开销，可以为 BinSort增加一个附加的参数 value，并使该函数返回排序所使用的值。其语法如下：

```
void Chain<T>::BinSort(int range, int(*value)(T& x))
```

该语句表明函数Chain<T>::BinSort带有两个参数，而不返回任何值。第一个参数range的类型为int，第二个参数value是一个函数的名字，该函数带有一个类型为 T&的参数x并返回一个int值。

当BinSort按照上述形式定义时，程序3-44中的语句

```
j=first->data;
```

需要被改写为：

```
j=value(first->data);
```

至此，可以采用类似程序3-45的代码来进行排序。

程序3-45 按不同的域进行排序

```
inline int F1(Node& x) {return x.exam1;}
inline int F2(Node& x) {return x.exam2;}
inline int F3(Node& x)
    {return x.exam1 + x.exam2 + x.exam3;}
void main(void)
{
    Node x;
    Chain<Node>  L;
    randomize();
    for (int i = 1; i <= 20; i++) {
        x.exam1= i/2;
        x.exam2= 20 - i;
        x.exam3= random(100);
        x.name = i;
        L.Insert(0,x);}
```

```
    L. BinSort (10, F1);
    cout << "Sort on exam 1" << endl;
    cout << L << endl;
    L. BinSort(20, F2 );
    cout << "Sort on exam 2" << endl;
    cout << L << endl;
    L.BinSort(130, F3);
    cout << "Sort on sum of exams" << endl;
    cout << L << endl;
}
```

3.8.2 基数排序

可以扩充3.8.1节的箱子排序方法，使其在$\Theta(n)$时间内对范围在0~n^c-1之间的n个整数进行排序，其中c是一个常量。注意，如果用 range= n^c 来调用函数BinSort，则排序的复杂性将变成$\Theta(n+\text{range})=\Theta(nc)$。对于BinSort来说，一种替代的方法是，不直接对这些数进行排序，而是采用一些基数r 来分解这些数。例如，十进制数928可以按照基数10分解为数字9，2和8（即$928=9*10^2+2*10^1+8*10^0$）。最高位是9，最低位是8。用基数10来分解3725可得到3，7，2和5，而利用基数60来进行分解则可以得到1，2和5（即$(3725)_{10}=(125)_{60}$）。在基数排序（radix sort）中，把数按照某种基数分解为数字，然后对数字进行排序。

例3-1 假定对范围在0~999之间的10个整数进行排序。如果使用range=1000来调用BinSort，那么箱子的初始化将需要1000个执行步，节点分配需要10个执行步，从箱子中收集节点需要1000个执行步，总的执行步数为2010。另一种方法是：

1) 用BinSort根据数的最低位数字对 10个数进行排序。由于每个数字的范围为 0~9，因此range=10。图3-17a 给出了具有10个数的链表，图3-17b 给出了按最低位数字排序后的链表。

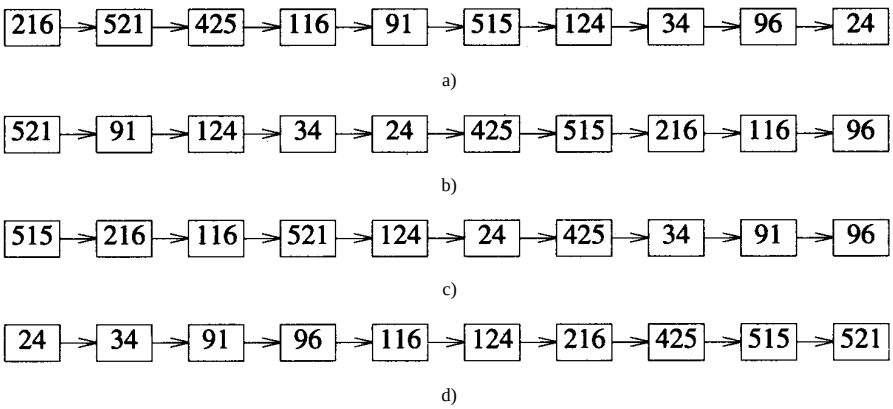

图3-17 用r=10和d=3进行基数排序

a) 输入链表 b) 按最后一位数字排序后的链表 c) 按倒数第二位数字排序后的链表 d) 按最高位数字排序后的链表

2) 用箱子排序算法对1) 中所得到的链表按次低位数字进行排序。同样，有range=10。由于箱子排序是稳定排序，次低位数字相同的节点，其相对次序保持不变（与按最低位数字排序时

所得到的次序相同)。因此，现在链表是按照最后两位数字进行排序的。图 3-17c 给出了相应的排序结果。

3) 用箱子排序算法对 2) 中所得到的链表按第三位 (最高位)数字进行排序。(如果一个数仅包含两位数字，则其第三位数字为 0)。由于按第三位数字排序是稳定排序，所以第三位数字相同的节点，其相对次序保持不变（与按最后两位数字排序时所得到的次序相同)。因此，现在链表是按照后三位数字进行排序的。图 3-17d 给出了相应的排序结果。

上述排序模式是基数为10的排序，被排序的数被分解为相应的十进制数字，排序是针对这些数字进行的。由于每个数都至少有三位数字，所以要进行三次排序，每次排序都使用range=10的箱子排序过程来完成。在每次的箱子排序过程中，需要 10个执行步来对箱子进行初始化，10个执行步用来把数分配至相应的箱子节点，10个执行步用来收集箱子节点。总的执行步数为90，比使用range=1000进行10个数的箱子排序要少得多。单个箱子排序模式实际上等价于r=1000的基数排序。

例3-2 假定对1000个范围在0~10^6-1之间的整数进行排序，使用基数$r=10^6$的排序方法（即直接使用BinSort排序函数）需要10^6执行步对箱子初始化，1000个执行步分配箱子节点，另外10^6执行步收集箱子节点，因此总的执行步数为 2 001 000。与此相应地，对于r=1000的排序，其过程如下：

1) 采用每个数的最低三位数字进行排序，令range=1000。

2) 对1中得到的结果再利用每个数的倒数次三位（即倒数第四到六位）数字进行排序。

上述每次排序都需要3000个执行步，所以排序完成时共需要6000个执行步。若使用r=100，则需使用三次箱子排序过程依次对每两位数字进行排序，每次箱子排序需要 1200个执行步，总的执行步数为3600。如果使用r=10，则要进行六次箱子排序，每次针对一位数字，总的执行步数为6(10+1000+10)=6120。因此，对于本例，采用基数r=100的排序效率最高。

为了实现例3-1和例3-2的基数排序，需要按给定的基数对数进行分解。可以采用除法和取模运算来完成这种分解。如果采用基数 10来进行分解，那么可以按照如下表达式来得到每位数字（从最低位到最高位）：

x%10; (x%100)/10; (x%1000)/100;...

若r=100，则相应的分解式为：

x%100; (x%10000)/100; (x%1000000)/10000;...

对于一般的基数r，相应的分解式为：

x%r; (x%r^2)/r; (x%r^3)/r^2; ...

当使用基数r=n对n个介于0~n^c-1范围内的整数进行分解时，每个数将可以分解出 c个数字。因此，可以采用c 次箱子排序，每次排序时取range=n。整个排序所需要的时间为 $\Theta(cn)=\Theta(n)$（因为c 是一个常量)。

3.8.3 等价类

1. 定义和动机

假定有一个具有n 个元素的集合U={1，2，...，n}，另有一个具有r 个关系的集合R={(i_1, j_1), (i_2, j_2), ..., (i_r, j_r)}。关系R是一个等价关系（equivalence relation），当且仅当如下条件为真时成立：

• 对于所有的a，有(a, a) R时（即关系是反身的）。
• 当且仅当(b, a) R时(a, b) R（即关系是对称的）。
• 若(a, b) R且(b, c) R，则有(a, c) R（即关系是传递的）。

在给出等价关系R时，我们通常会忽略其中的某些关系，这些关系可以利用等价关系的反身、对称和传递属性来获得。

例3-3 假定n=14，R={(1,11)，(7,11)，(2,12)，(12,8)，(11,12)，(3,13)，(4,13)，(13,14)，(14,9)，(5,14)，(6,10)}。我们忽略了所有形如(a,a)的关系，因为按照反身属性，这些关系是隐含的。同样也忽略了所有的对称关系。比如(1,11) R，按对称属性应有(11,1) R。其他被忽略的关系是由传递属性可以得到的属性。例如根据(7,11)和(11,12)，应有(7,12) R。

如果(a,b) R，则元素a 和b 是等价的。等价类（equivalence class）是指相互等价的元素的最大集合。“最大”意味着不存在类以外的元素，与类内部的元素等价。

例3-4 考察例3-3中的等价关系。由于元素1与11，11与12是等价的，因此，元素1,11,12是等价的，它们应属于同一个等价类。不过，这三个元素还不能构成一个等价类，因为还有其他的元素与它们等价（如7）。所以{1,11,12}不是等价元素的最大集合。集合{1,2,7,8,11,12}才是一个等价类。关系R还定义了另外两个等价类：{3,4,5,9,13,14}和{6,10}。

在离线等价类（offline equiralence class）问题中，已知n 和R，确定所有的等价类。注意每个元素只能属于某一个等价类。在在线等价类（online equiralence class）问题中，初始时有n 个元素，每个元素都属于一个独立的等价类。需要执行以下的操作： 1) Combine(a, b) 把包含a 和b 的等价类合并成一个等价类。 2) Find(e) 确定哪个类包含元素e，搜索的目的是为了确定给定的两个元素是否在同一个类之中。因此，对于同一类中的元素，Find将返回相同的结果，而对不同类的元素，则返回不同的结果。

可以利用两个Find操作和一个Union操作产生一个组合操作，该操作能把两个不同的类合并成一个类。因此Combine(a,b)等价于：

```
i = Find(a); j = Find(b);
if(i! = j) Union(i, j);
```

注意，利用Find和Union操作，可以向R中添加新关系。例如，为了添加关系(a, b)，可以首先判断a 和b 是否已经位于同一个等价类，如果是，则新关系是冗余的，如果不是，则对包含a 和b 的两个类执行Union操作。

本节主要关心在线等价类问题，这类问题通常又称之为 union-find问题。本节给出的解决方案很简单，但是效率不是最高的。8.10.2节给出了一个更快的解决方案。“离线等价类”问题的快速解决方案将在5.5.5节给出。

例3-5 **[根据最后期限进行调度]** 某工厂有一台机器能够执行n 个任务，任务i的释放时间为r_i（是一个整数），最后期限为d_i（也是整数）。在该机上完成每个任务都需要一个单元的时间。一种可行的调度方案是为每个任务分配相应的时间段，使得任务 i的时间段正好位于释放时间和最后期限之间。一个时间段不允许分配给多个任务。

考察下面的四个任务：

任务	1	2	3	4
释放时间	0	0	1	2
最后期限	4	4	2	3

任务1和任务2的释放时间为0，任务3的释放时间为1，任务4的释放时间为2。下面的任务-时间调度方案是可行的：在0~1期间执行任务1；1~2期间执行任务3；2~3期间执行任务4；3~4期间执行任务2。进行调度的一种直观方法如下：

1) 按释放时间的递增次序对任务进行排序。

2) 考察1) 中所得到的任务序列。对于每个任务，确定与最后期限最接近的空闲时间段（位于最后期限之前）。如果这个空闲时间段位于任务的释放时间之前，则失败，否则把这个时间段分配给任务。

练习83要求你证明，如果不存在一个可行的调度方案，则上述策略将失败。

在线等价类问题的方法可用来实现 2)。令d为所有任务中最后一个的完成期限，各可用时间段的形式为“从i-1至i”，其中$1 \le i \le d$，这些时间段被称为“时间段1，时间段2，...，时间段d”。对于任意时间段a，定义$near\ (a)$为最大i：$i \le a$且时间段i空闲。如果不存在这样的i，则定义$near\ (a)=near\ (0)=0$。当且仅当$near\ (a)=near\ (b)$时，两个时间段a和b属于同一个等价类。

在调度任务之前，对于所有时间段有$near\ (a)=a$，且每个时间段都是一个独立的等价类。当时间段a被分配给2) 中的某个任务时，对于所有$near\ (b)=a$的时间段b，其$near$值发生变化，对于这些时间段，其新的$near$值为$near\ (a\text{-}1)$。因此，当时间段a被分配给一个任务时，需要对当前包含时间段a和a-1的等价类进行合并（执行Union操作）。如果每个等价类E用$N[E]$表示，$near$的值是类中的成员，那么$near\ (a)$将由N[Find(a)]给出。（假定类名就是Find操作所返回的内容）。

例3-6 [布线] 一个电路由构件、针脚和电线构成。图3-18给出了一个由三个构件A，B和C组成的电路。每根电线连接了一对针脚。当且仅当要么有一根电线直接连接了a和b，要么存在一个针脚序列$a_1, a_2, ..., a_k$，使得a, a_1；a_1, a_2；a_2, a_3；...；a_{k-1}, a_k；和a_k, b均由电线直接相连时，两个针脚a和b是电子等价（electrically equivalent）的。网组(net)是指电子等价针脚的最大集合，“最大”是指不存在网组外的针脚与网组内的针脚电子等价。

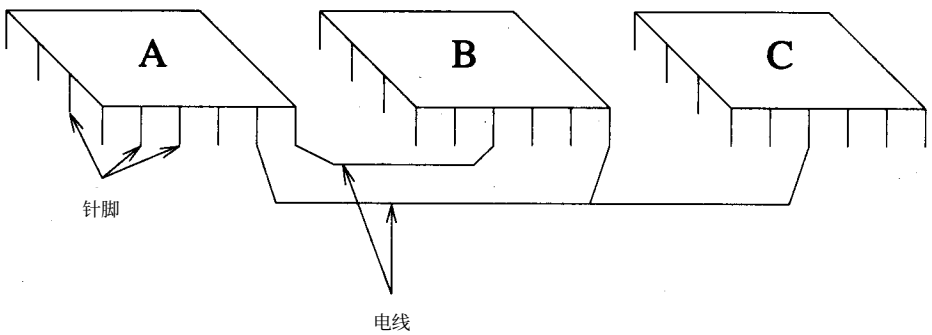

图3-18 一个印刷电路板上的3芯片电路

考察图3-19所示的电路。在该图中仅绘出了针脚和电线。14个针脚从1至14编号。每根电线可由它所连接的两个针脚来描述。例如，连接针脚 1和11的电线可以表示为 (1,11)，它与(11,1)等价。电线的集合为{(1,11)，(7,11)，(2,12)，(12,8)，(11,12)，(3,13)，(4,13)，(13,14)，(14,9)，(5,14)，(6,10)}。因此，该电路中所存在的网组如下：{1，2，3，8，11，12}，{3，4，5，9，13，14}和{6，10}。

在离线网组搜索（offline net finding）问题中，已知针脚和电线，需要确定相应的网组。如果把每个针脚看成U的一个成员，把每根电线看成R的成员，那么离线网组搜索问题与离线

等价类问题完全相同。

对于在线网组搜索（online net finding）问题，起始时有一组针脚的集合，没有电线，然后执行以下操作：1) 增加一根连接*a*和*b*的电线；2) 搜索包含针脚*a*的网组。搜索的目的是为了确定两个针脚是否位于同一个网组中。在线网组搜索问题实际上等同于在线等价类问题。初始时没有电线，相当于 $R=\phi$，网组搜索操作对应于等价类的 Find操作，添加电线 (a,b)对应于 Combine(a, b)。

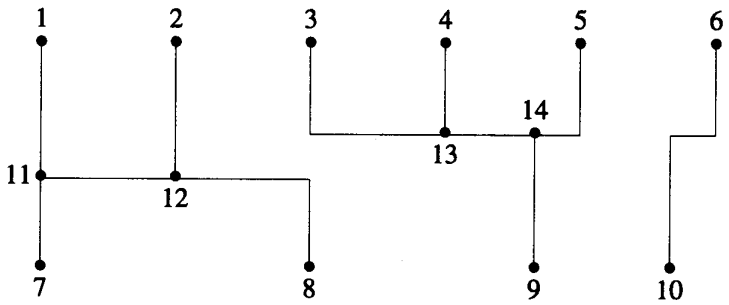

图3-19 仅给出针脚和电线的电路

2. 第一种解决方案

在线等价类问题的一种简单解决办法是使用一个数组 E并令E(e) 代表包含元素e 的等价类。完成初始化、合并及搜索操作的函数如程序 3-46所示。N 是元素的数目，n 和E 均被定义为全局变量。为了合并两个不同的类，可从类中任取一个元素，然后把该类中所有元素的 E 值修改成另一个类中元素的E 值。Initialize和Union函数的复杂性均为 $\Theta(n)$ (假定在Initialize中调用new时不产生异常)，Find的复杂性为 $\Theta(1)$。从例3-5和例3-6中可以看出，在应用这些函数时，通常执行一次初始化，u 次合并和f 次搜索，故所需要的总时间为 $\Theta(n+u*n+f)=\Theta(u*n+f)$。

程序3-46 使用数组的在线等价类函数

```
void Initialize(int n)
{// 初始化n个类，每个类仅有一个元素
   E = new int [n + 1];
   for (int e = 1; e <= n; e++)
     E[e] = e;
}
void Union(int i, int j)
{//合并类 i 和类 j
   for (int k = 1; k <= n; k++)
      if (E[k] == j) E[k] = i;
}
int Find(int e)
{//搜索包含元素 i 的类
   return E[e];
}
```

3. 第二种解决方案

针对每个等价类设立一个相应的链表，可以降低合并操作的时间复杂性，因为可以沿着类

j的链表找到所有E[k]=j 的元素，而不必去检查所有的E值。事实上，如果知道每个等价类的大小，可以改变较小类的E值，这样可以使合并操作更快。通过使用模拟指针，可以快速地访问代表元素e的节点。使用以下的约定：

- EquivNode是一个类，E，size和link是其私有数据成员，这些数据成员的类型均为int。
- 函数Initialize,Union和Find均为EquivNode的友元。
- node[1:n]用于描述n个元素（每个元素都有一个对应的等价类链表）。
- node[e].E既是Find(e)返回的值，也是一个指针，该指针指向类 node[e].E对应链表的首节点。
- 仅当e是链表的首节点时，才定义node[e].size，这时，它的值表示从node[e]开始，链表中的节点数目。
- node[e].link给出了包含节点e的链表的下一个节点。由于所使用的节点被编号为1至n，故可以用0而不是-1来表示空指针。

程序3-47给出了Initialize和Union的新代码。Find的代码与程序3-46相同。

程序3-47 使用链表的在线等价类函数

```
void Initialize(int n)
{// 初始化n个类，每个类仅有一个元素
  node = new EquivNode [n + 1];
  for (int e = 1; e <= n; e++) {
    node[e].E = e;
    node[e].link = 0;
    node[e].size = 1;
   }
}
void Union(int i, int j)
{ //合并类 i 和类 j
    // 使 i 代表较小的类
    if (node[i].size > node[j].size)
       swap(i,j);
    //改变较小类的 E值
    int k;
    for (k = i; node[k].link; k = node[k].link)
       node[k].E = j;
    node[k].E = j; // 链尾节点
    //在链表j的首节点之后插入链表 i
    // 并修改新链表的大小
    node[j].size += node[i].size;
    node[k].link = node[j].link;
    node[j].link = i;
}
int Find(int e)
{//搜索包含元素 i 的类
    return node[e].E;
}
```

在使用链表时，因为一个等价类的大小为O(n)，因此合并操作的复杂性为O(n)，而初始化和搜索操作的复杂性仍分别保持为 $\Theta(n)$和$\Theta(1)$。为了确定1次初始化操作、u 次合并操作和f

次搜索操作所需要的时间复杂性，需要使用如下的定理。

定理3-1 如果开始时有n个类，每个类有一个元素，则在执行u次合并操作以后，

a) 任何一个类的元素数都不会超过$u+1$。

b) 至少存在$n-2u$个单元素类。

c) $u<n$。

证明 见练习81。

1次初始化和f次搜索的复杂性为$\Theta(n+f)$。对于u次合并，每次合并操作的开销为Θ(较小类的大小)。令i表示合并操作中的较小类。在合并期间，i中的每个元素从类i移向类j，因此u次合并的复杂性由移动元素的总次数确定。移动类i之后，新类的大小至少是类i的两倍（因为在移动前有size[i]≤size[j]，而移动之后新类的大小为size[i]+size[j]）。因此，由于在操作结束时没有哪个类的元素数会超过$u+1$（定理3-1a），所以在u次合并期间，没有哪个元素的移动次数超过$\log_2(u+1)$。另外，根据定理3-1b，最多可以移动$2u$个元素，所以，元素移动的总次数不会超过$2u\log_2(u+1)$。至此可以知道执行u次合并操作所需要的时间为$O(u\log u)$。1次初始化、u次合并操作和f次搜索的复杂性为$O(n+u\log u+f)$。

3.8.4 凸包

多边形（polygon）是指至少有三条直线边的平面封闭图形。图3-20a所示的多边形有六条边，图3-20b中的多边形有八条边。多边形既包含其边线上的点，也包含由边所包围的所有点。凸多边形（convex polygon）的任意两个点（位于边线上或多边形内）之间的连线都包含在多边形内。图3-20a中的多边形是凸多边形，而图3-20b中的多边形为非凸多边形（noconvex polygon）。图3-20b中画出了两条虚线段，虽然这两条虚线段的端点都在边线上或多边形内，但它们都包含了多边形以外的点。

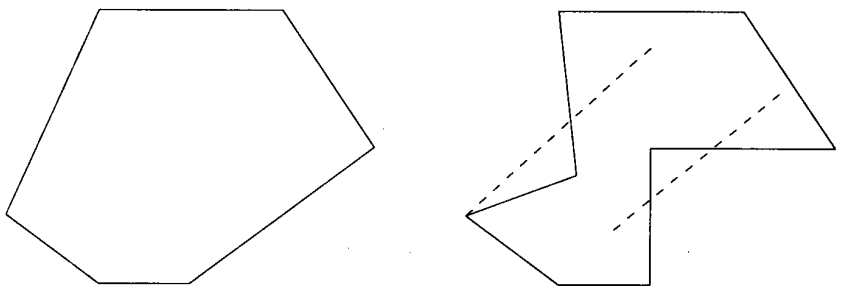

图3-20 凸多边形和非凸多边形

a) 凸多边形 b) 非凸多边形

平面上一个点集S的凸包（convex hull）是指包含所有这些点的最小凸多边形，该多边形的角是S的极点(extreme point)。图3-21给出了13个点（同一平面上），相应的凸包是由实线连成的多边形，所有的极点用圆圈来标记。如果S中的所有点都落在一条直线上，则是一种退化情形，此时凸包被定义为包含所有点的最短直线。

寻找平面点集对应凸包的问题是计算几何中的基本问题。计算几何中另外几个问题（如寻找包含指定点集的最小矩形）的求解都需要借助于对凸包的计算。此外，凸包在图象处理和统

计学中都有很多应用。

假定在S对应的凸包内取一个点X，然后从X向下画一条垂直线（如图3-22a所示）。练习85说明了如何选择点X。令a_i表示这条垂直线与X和S中第i个点连线间的夹角（又称极角）。对a_i的测量是按逆时针方向进行的（即从垂直线开始沿逆时针方向测量至X与第i个点的连线）。图3-22a中给出了夹角a_2。现在按照a_i的递增次序来排列S中的点，对于具有相同极角的点，再按照它们与X之间的距离来进行排列。在图3-22a中，所有点按照上述次序被依次编号为1至13。

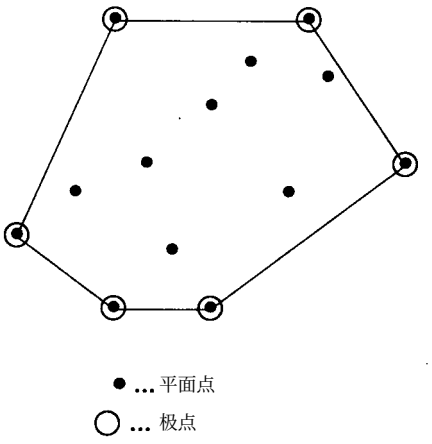

图3-21 平面点对应的凸包

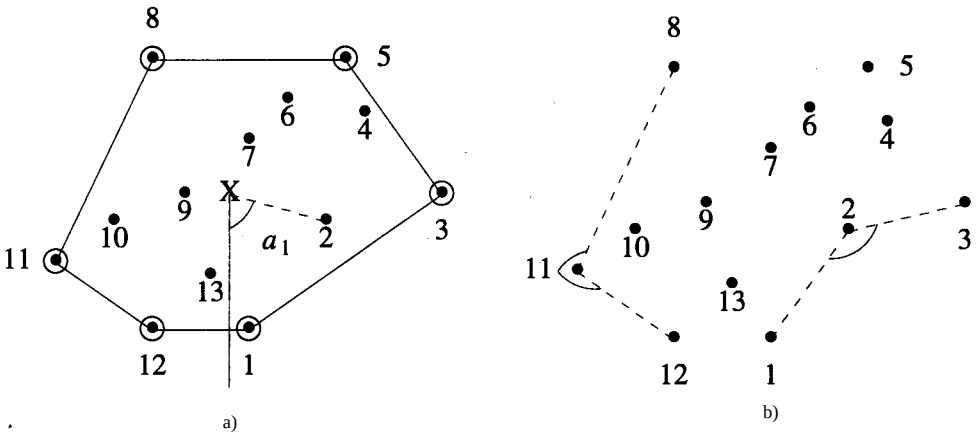

图3-22 识别极点

a) 顶点排序 b) 逆时针夹角

X向下的垂线沿逆时针扫描会依次（按照极角的a_i次序）遇到S中的极点。如果u, v和w是按照逆时针排列的三个连续的极点，那么u到v与w到v两条连线之间的逆时针夹角将大于180度。（图3-22b给出了点8，11，12之间的逆时针夹角）。当按照极角次序排列的3个连续点之间的逆时针夹角小于或等于180度时，那么其中的第二个点不是极点。注意当u, v, w间的夹角小于180度时，如果从u走到v再走到w，那么在v点将会向右转。当按逆时针方向在一个凸多边形上行走时，所有的方向变化都是向左转。根据这种观察结果得到了图3-23的算法，该算法用于寻找极点以及S对应的凸包。

1) 用于处理退化的情形，即S中的点数为0或1，或者S中的所有点共线。1) 可以在$\Theta(n)$时间内完成，其中n是S的点数。对于共线的判断，方法是，任取两个点，求出两点之间连线的方程式，然后检查余下的$n-2$个点，并判断它们是否在这条直线上。在这个过程中，如果所有点都是共线的，还可以同时确定包含这些点的最短直线的端点。

在2) 中，按照极角的次序排列所有的点，并把它们放入一个双向链表之中（之所以采用双

向链表是因为在3) 中，需要消除不是极点的点)，并在链表中反向移动，而这两种操作对于双向链表来说都是能直接实现的。练习85会要求你采用一个单向链表。因为需要排序，所以如果采用第2章中的排序算法，需要耗时$\Theta(n^2)$。在第9章和第14章中，可以在$O(n\log n)$时间内完成排序，因此2) 的时间复杂性可以计为$O(n\log n)$。

```
1) [处理退化情形]
   如果S的点数少于3个，则返回S
   如果所有点都在同一条直线上，则计算出包含所有点的最短直线的端点，并返回这两个端点
2) [按极角排序]
   在S对应的凸包内寻找点X
   按照极角对S中的点进行排序，若极角相同，则按与X的距离排序
   建立一个双向循环链表，把各个点按上述次序放入链表之中
   令right指向排序中的下一个点，left指向前一个点
3) [删除不是极点的点]
   令p为具有最小y坐标的点（也可以选择具有最大x坐标的点）
   for(x=p , rx=x的下一个点； x!= rx ；){
     rrx= rx的下一个点；
     if( x,rx,rrx间的夹角小于或等于180度) {
        从链表中删除rx;
        rx=x; x=rx的前一个节点；}
     else {x=rx; rx=rrx;}
     }
```

图3-23 寻找与S对应的凸包的伪代码

在3) 中，依次检查按逆时针次序排列的三个连续点，判断三个点构成的夹角是否小于或等于180度，如果是，则中间点rx 不是极点，从链表中删除之。如果夹角超过 180度，则第三点rrx 可能是也可能不是极点，可从点x 转向处理下一个点。当for循环终止时，双向循环链表中的每个点都满足属性：x, rx 和rrx 间的夹角超过180度。因此链表中的所有点都是极点。根据链表的right域遍历链表，可以得到逆时针排列的凸包的边界。从具有最小 y坐标的点开始的，因为这个点肯定位于凸包之中。

对于3) 的复杂性，注意到，在for循环中每次检查一个夹角之后会出现以下情形：1) 顶点rx被删除，x 在链表中后移一个位置，或2) x 前移一个位置。由于被删除的顶点数为 $O(n)$，x 最多向后移动$O(n)$个位置。因此情形2) 只会发生$O(n)$次，因而for循环将执行$O(n)$次。由于检查一个夹角需要耗时$\Theta(1)$，所以3) 的复杂性为$O(n)$。这样，为了找到n 个点的凸包，需要耗时$O(n\log n)$。

练习

76. 程序3-43是稳定排序吗？

77. 比较程序3-43和3-48所给出的箱子排序函数的运行时间，使用 n=10 000, 50 000和100 000进行测试。指出由于引入类Chain所产生的开销。

78. 给定一个n 节点的链表，节点类型为ChainNode<Node>。按照score域对链表进行排序。

1) 编写一个函数，采用基数排序方法进行排序。函数的输入为：欲排序的链表、基数 r、按基数r 分解的数字位数d。函数的复杂性应为$\Theta(d(r+n))$。试证明之。

2) 通过编译和执行（使用自己设计的测试数据）来测试函数的正确性。

3) 把所编写的函数与用链表完成插入排序的函数进行性能比较。可使用 n=100，1000，10 000；r=10和d=3来测量运行时间。

79. 1) 编写一个函数，使用r=n的基数排序算法对n个0~n^c-1范围内的整数进行排序。函数的复杂性应为$\Theta(cn)$。试证明之。假定函数的输入为待排序的链表。

2) 测试函数的正确性。

3) 对于n=10, 100, 1000, 10 000 和c=2，测量函数的运行时间。用表格和图的形式给出测量结果。

80. 给定n 堆卡片，每张卡片有三个域：卡片所在堆的编号、卡片样式及卡片面值。每堆卡片最多有52张卡片，因此卡片总数最多为52n。可以假定每一堆至少有一张卡片，所以卡片总数至少为n。

1) 解释如何按照堆的编号对卡片进行排序（对于堆号相同的卡片按卡片样式进行排序，对于样式也相同的卡片按其面值进行排序）。可以采用三次箱子排序过程来完成卡片排序。

2) 编写一段程序，其输入为 n和卡片堆，输出为排序后的卡片堆。把卡片堆表示成一个链表，链表中的每个节点包含如下域： deck, suit, face和link。程序的复杂性应为$\Theta(n)$，试证明之。

3) 测试程序的正确性。

81. 证明定理3-1。

82. 对于例3-6的在线网组搜索问题，编写一个C++函数。要求模仿在线等价类问题进行处理，并要求使用链表。测试程序的正确性。

83. 证明例3-5中所给出的策略：仅当不存在可行的调度方案时，搜索过程才会失败。

84. 对于例3-5的调度问题编写一个C++程序，按照在线等价类问题进行处理，要求使用链表。测试程序的正确性。

85. 1) 令u, v, w 是平面上的三个点，假定这三点不在同一直线上。编写一个函数，该函数从这三个点构成的三角形中取一个点。

2) 令S 是一个平面点集。编写一个函数来判断 S中的所有点是否共线。如果共线，计算出包含所有这些点的最短直线的端点。如果不共线，从点集中找出三个不共线的点。利用 1) 中的三个点以及函数来指定一个S 凸包内的点。函数的复杂性应为$\Theta(n)$，试证明之。

3) 使用1) 和2) 中的代码把图3-23中的算法细化成一个C++程序，程序的输入为点集S，输出为点集S 对应的凸包。在输入S期间，可同时把点放入双向链表之中（之后将按照极角对这些点进行排序）。对于排序，使用第2章所给出的排序算法，或者使用复杂性为O($n$log$n$)的排序算法。

4) 编写其他的凸包程序，采用a) 单向链表，b) 基于公式的线性表来替代双向链表。

5) 测试程序的正确性。

86. 令c 是一个链表。假定在链表中向右移动时，把链表指针的方向变反，因此，如果正处于节点p，链表将被分成两个子链表。一个子链表将从p 节点开始，一直延续到c 的最后一个节点。另一个子链表从链表c 中p 的前一个节点l 开始，一直向后延续到c 的首节点。初始时，p=c，l=0。

1) 绘出有6个节点的链表，给出p 为第三个节点，l 为第二个节点的情形。

2) 编写一个函数使l 和p 前进一个节点。

3) 编写一个函数使l 和p 后退一个节点。

4) 用适当的数据来测试代码。

87. 使用单向链表来完成练习85。用练习86的思想来确保图3-23中步骤3的for循环具有复杂性$\Theta(n)$。

88. 采用练习86的思想来扩充程序3-8中Chain类的定义，使得能够在单向链表中高效地前进和后退。为此，按照练习86应增加私有成员l 和p，同时增加以下共享函数：

1) Reset——把p 置为first，l 置为0。

2) Current(x)——返回由p 所指向的元素x；如果操作失败，则引发一个异常。

3) End——如果p 指向链表尾部，则返回true，否则返回false。

4) Front——如果p 指向链表首部，则返回true，否则返回false。

5) Next——使p 和l 右移一个位置；如果操作失败，则引发一个异常。

6) Previous——使p 和l 左移一个位置；如果操作失败，则引发一个异常。

对于扩充后的类定义，编写相应的 C++代码。为了高效地实现 Insert，Delete和Find函数，可以引入另外一个私有成员 current，它给出由p 所指向的元素的索引（即表中的第 1个元素、第2个元素等）。用适当的数据来测试代码。

注：带*号的练习题为提高题，较难。

*89. 给出一种整数的表示，它适合于对任意大的整数进行算术运算，而且算术运算的结果没有精度损失。编写一个 C++程序来输入和输出大的整数，同时实现以下算术操作：加、减、乘和除。除法函数应返回两个整数：商和余数。此外，应编写一个函数用以释放一个整数（即释放该整数所占用的空间）。

*90. 一个阶数为d 的一元多项式（univariate polynomial）形式如下：

$$c_d x^d + c_{d-1} x^{d-1} + c_{d-2} x^{d-2} \ldots + c_0$$

其中$c_0 \neq 0$，c_i 是系数，e_i 是指数。根据定义，指数是非负整数。每个 $c_i x^i$ 是多项式的一个项。我们希望设计一个模板类来支持涉及多项式的算术运算。为此，需要把每个多项式表示成一个由系数构成的线性表$(c_0, c_1, c_2, \ldots, c_d)$。

设计一个C++模板类Polynomial<T>，其中T 给出了系数的类型。类Polynomial 应该带有一个私有成员degree，它是多项式的阶数。当然，它还可能包含其他的私有成员。多项式类应支持以下操作：

1) Polynomial()——创建一个0阶多项式。这个多项式的阶数为 0，不包含任何项。它是类的构造函数。

2) Degree()——返回多项式的阶数。

3) Input()——读入一个多项式。可以假定输入是由多项式的阶数和一个系数表构成，系数表中的系数按指数递增的次序排列。

4) Output()——输出多项式。输出格式可以与输入格式相同。

5) Add(b)——把当前多项式加到多项式*b* 上，并返回所得结果。

6) Subtract(b)——减去多项式*b* 并返回所得结果。

7) Multiply(b)——乘以多项式*b* 并返回所得结果。

8) Divide(b)——除以多项式*b* 并返回所得结果。

9) Value(x)——返回按*x* 计算出的多项式的值。

对于3) 至9)，需要重载操作符<<、>>、+、 -、 *、 / 和( )。对于9)，语法P(x) 应返回多项式在x点的取值，其中P的类型为Polynomial。对程序进行测试。

*91. 设计一个链表类来表示和处理一元多项式（见练习 90）。使用带头节点的循环链表。

每个节点的data域包含一个系数域和一个指数域。除了头节点外，多项式的循环链表的，每个节点均对应于多项式的一个项（系数不为 0），系数为0的项不需考虑。所有的项按照指数的递减次序排列。头节点的指数域取值为 - 1。图3-24给出了一些示例。

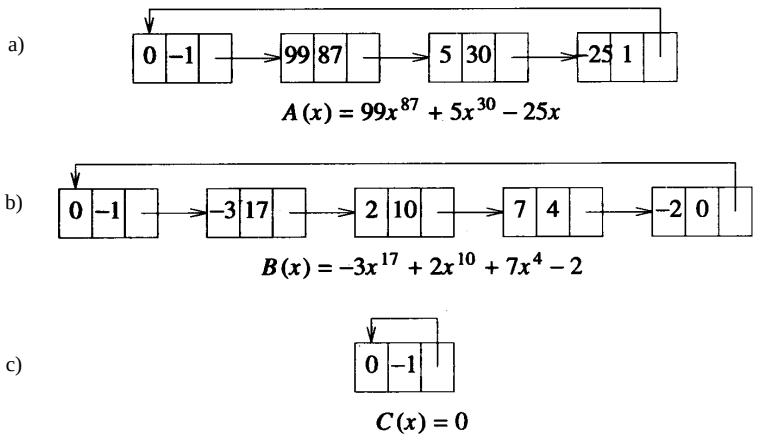

图3-24 多项式举例

一个一元多项式的外部表示（即针对输入和输出）将被假定为形如 n，e_1，c_1，e_2，c_2，e_3，c_3，...，e_n，c_n 的序列，其中e_i 表示指数，c_i 表示系数，n 给出了多项式的项数。指数按照递减的次序排列，即有：$e_1 > e_2 > ... > e_n$。

类应能支持练习90中的所有函数。采用适当的多项式测试代码。

3.9 参考及推荐读物

关于C++数据结构的其他参考书如下：

1) E.Horowitz，S.Sahni, D.Mehta. *Fundamentals of Data Structure in C++*. W.H.Freeman, 1994。

2) F.Carrano. *Data Abstraction and Problem Solving with C++*. Benjamin/Cummings, 1995。

3) A. Drozdek. *Data Structures and Algorithms in C++*. PWS, 1996。

4) M. Weiss. *Algoritlzms, Data Structures, and Problem Solving with C++*. Addison-Wesley, 1996。

5) T. Budd. *Classic Data Structures in C++*. Addison-Wesley, 1994。