第 2 章　线性表

教学要点

线性表是一种基本的线性数据结构，这种数据集合的数据元素间具有线性逻辑关系，可以在线性表的任意位置进行插入和删除数据元素的操作。一个线性表数据结构可以用顺序存储结构和链式存储结构两种方式实现，前者称为顺序表，后者称为链表。

本章首先学习线性表在逻辑结构层次方面的特性，然后讨论以顺序存储结构实现的线性表和以链式存储结构实现的线性表在结点结构和各种操作的实现方面的特性，分析、比较这些不同实现的优缺点。

本章在 Visual Studio 中用名为 lists 的类库型项目实现有关数据结构的类型定义，用名为 liststest 的应用程序型项目实现对这些数据结构的测试和演示程序。

建议本章授课 8 学时，实验 6 学时。

2.1　线性表的概念及类型定义

线性数据结构是一组具有某种共性的数据元素按照某种逻辑上的顺序关系组成的一个数据集合。线性结构的数据元素之间具有顺序关系，除第一个和最后一个数据元素外，每个元素只有一个前驱元素和一个后继元素，第一个数据元素没有前驱元素，最后一个元素没有后继元素。

线性表（Linear List）是一种典型的线性数据结构，其数据元素之间具有顺序关系,并且可以在表中任意位置进行插入和删除数据元素的操作。

线性表中元素的类型可以是数值型或字符串型，也可以是其他更复杂的自定义数据类型。例如：

- 数字表： 个位数字表{0，1，2，…，9}可以看成是一个线性表，数据元素是单个数字，数据元素间是按顺序排列的。
- 学生成绩表：一个班级学生的成绩列表可以看成是一个线性表，数据元素是每个学生的学号、姓名、成绩等信息。
- 科研设备信息表：一个实验室的科研设备信息表可以看成是一个线性表，数据元素是每台设备的编号、类型、名称和保管人等信息。

2.1.1　线性表的抽象数据类型

1. 线性表的数据元素

当我们讨论线性表时，我们使用抽象数据元素 a_i 表示线性表中的某个数据元素。线性表是由 n（$n \geqslant 0$）个数据元素 a_0，a_1，a_2，…，a_{n-1} 组成的有限序列，记作：

$$\text{LinearList} = \{a_0, a_1, a_2, \cdots, a_{n-1}\}$$

其中，n 表示线性表的元素个数，称为线性表的长度。若 n=0，则表示线性表中没有元素，我们称之为空表。若 n>0，对于线性表中第 i 个数据元素 a_i，有且仅有一个直接前驱数据元素 a_{i-1} 和一个直接后继数据元素 a_{i+1}，而 a_0 没有前驱数据元素，a_{n-1} 没有后继数据元素。

线性表中的数据元素至少具有一种相同的属性，我们称这些数据元素属于同一种抽象数据类型。具体设计线性表的物理实现时，a_i 的数据类型将具体化，各元素的具体类型可以相同，也可以不同，但是至少具有一种相同的属性。例如，在 C#中的线性表（ArrayList 类），元素的类型可以是相同的整数 int 类型、字符 char 类型或字符串 string 类型；如果数据元素的类型不相同，则可以用 object 类来描述它们，以表明它们至少具有某种相同的性质（都是某种类型的对象），因为 object 类是 C#中类层次的根类，所有其它的类都是由 object 类派生出来的。在这个意义上，可以简单地说，线性表中的数据元素具有相同的类型。

线性表逻辑结构有两种存储结构实现方式：顺序存储结构和链式存储结构。用顺序存储结构实现的线性表称为顺序表（Sequenced List），用链式存储结构实现的线性表称为链表（Linked List）。

2. 线性表的基本操作

线性表的数据元素之间具有顺序关系，可以在表中任意位置进行插入和删除数据元素的操作。线性表的典型操作有：

- Initialize: 初始化。创建一个线性表实例，并对该实例进行初始化，例如设置表状态为空。
- Get/Set: 访问。对线性表中指定位置的数据元素进行取值或置值操作。
- Count: 求长度。求线性表的数据元素的个数。
- Insert: 插入。在线性表指定位置上，插入一个新的数据元素，插入后，其所有元素仍构成一个线性表。一种常见的插入操作是在表尾添加一个新元素（Add）。
- Remove: 删除。删除线性表指定位置的数据元素，同时保证更改后的线性表仍然具有线性表的连续性。
- Copy: 复制。重新复制一个线性表。
- Join: 合并。将两个或两个以上的线性表合并起来，形成一个新的线性表。
- Search: 查找。在线性表中查找满足某种条件的数据元素。
- Sort: 排序。对线性表中的数据元素按关键字的值，以递增或递减的次序进行排列。
- Traversal: 遍历。按次序访问线性表中的所有数据元素，并且每个数据元素恰好访问一次。

2.1.2 C#中的线性表类

在.NET Framework 的类库中定义了一个非泛型线性表 ArrayList 类和一个泛型线性表 List<T>类。

1. 非泛型线性表类 ArrayList

.NET Framework 的类库在 System . Collections 命名空间中定义了一个线性表类 ArrayList。ArrayList 类提供了一种元素数目可按需动态增加的数组，其数据元素的类型是 object 类。插入线性表的数据元素要求是 object 对象，因为 object 类是 C#中类层次的根类，所有其它的类都是由 object 类派生出来的，所以实际上可以将任意类型的对象加入线性表；获取线性表某元素的数据可用 object 类型的变量引用，一般需根据要求转化为 string 或其他类型的对象。

ArrayList 类具有如下成员（属性和方法）实现线性表的各种操作：

公共构造函数

ArrayList();	初始化 ArrayList 类的新实例
ArrayList(ICollection c);	初始化 ArrayList 类的新实例，它包含从指定集合复制的元素并且具有与所复制的元素数相同的初始容量。
ArrayList(int initCapacity);	初始化 ArrayList 类的新实例，它具有指定的初始容量。

公共属性

virtual int Count {get;}	返回线性表的长度，即包含的元素数
virtual int Capacity {get; set; }	获取或设置线性表可包含的元素数
virtual object this[int index] {get; set;}	获取或设置指定索引处的元素

公共方法

virtual void Insert(int i, object x);	将数据元素插入指定位置
virtual int Add(object x);	将对象添加到表的结尾处
virtual void AddRange(ICollection c);	将集合对象添加到表的结尾处
virtual int IndexOf(object x);	返回给定数据首次出现位置
virtual bool Contains(object x);	确定某个元素是否在表中
virtual void Remove(object x);	从表中移除特定对象的第一个匹配项
virtual void RemoveAt(int i);	删除指定位置的数据元素
virtual void Reverse();	将表中元素的顺序反转
virtual void Sort ();	对表中元素进行排序

2. 泛型线性表类 List<T>

较新的 C#语言（2.0 版起）和公共语言运行库 (CLR) 中增加了泛型（Generics），包括泛型类和泛型方法，泛型通常与集合一起使用。C#语言基础类库提供一个新的命名空间 System.Collections.Generic，它包含多个定义泛型集合的类（class）和接口（interface），泛型集合允许用户创建强类型的数据集合，并且能提供比非泛型集合更好的类型安全性和性能。建议面向 2.0 及更新版的所有应用程序都使用新的泛型集合类，如 List<T>，而不要使用旧的非泛型集合类，如 ArrayList。

List<T>类是与 ArrayList 类逻辑上等效的泛型线性表类，表示可通过索引访问的对象的强类型列表，它定义在 System.Collections.Generic 命名空间中。泛型类 List<T>在大多数情况下比非泛型类 ArrayList 执行得更好并且是类型安全的。

List<T>类所具有的属性和方法非常类似于 ArrayList 类对应的属性和方法，差别在于前者是强类型列表，在列表（实例）上进行操作时，元素的类型要与列表（实例）定义时声明的类型保持一致，即具有所谓的型安全性。

公共构造函数

List<T>();	初始化 List<T>类的新实例
List<T>(IEnumerable<T> c);	初始化 List<T>类的新实例，它包含从指定集合复制的元素并且具有与所复制的元素数相同的初始容量。

公共属性

virtual T this[int index] {get; set;}	获取或设置指定索引处的元素

公共方法

virtual void Insert(int i, T x);	将数据元素插入指定位置
virtual void Add(T x);	将对象添加到表的结尾处
virtual void AddRange(IEnumerable<T> c);	将集合对象添加到表的结尾处
virtual int IndexOf(T x);	返回给定数据首次出现位置
virtual bool Contains(T item);	确定某个元素是否在表中
virtual void Remove(T item);	从表中移除特定对象的第一个匹配项

下面的几个例子分别声明并构造特定类型的列表：

```
List<int>  a = new List<int> ();      // 声明并构造整型数列表
a.Add(86); a.Add(100);                // 向列表中添加整型元素
List<string>  s = new List<string> ();// 声明并构造字符串列表
s.Add("Hello"); s.Add("C# 2.0");      // 向列表中添加字符串型元素
List< Student>  st = new List< Student>(); // 声明并构造学生列表
st.Add(new Student ("200518001", "王兵")); // 向列表中添加学生类型元素
```

3.0 版 C#语言增加了“集合初始化器”来方便集合对象的初始化，例如：

```
List<int> nums = new List<int> { 0, 1, 2, 6, 7, 8, 9 };
```

集合初始化器其实是利用编译时技术对初始化器中的元素进行按序调用 Add(T)方法 。

【例2.1】 创建并初始化 ArrayList 以及打印出其值。

```
public class SamplesArrayList {
    public static void Main() {
        // Creates and initializes a new ArrayList.
        ArrayList myAL = new ArrayList();
        myAL.Add("Hello"); myAL.Add("World");myAL.Add("!");
        myAL.Insert(1, "C#");
        // Displays the properties and values of the ArrayList.
        Console.WriteLine("myAL");
        Console.WriteLine("\tCount:    {0}", myAL.Count);
        Console.Write("\tValues:");
        foreach (object o in myAL) {
            Console.Write("\t{0}", o);
        }
        Console.WriteLine();
        myAL.Sort();
        Console.Write("\tSorted Values:");
        for (int i = 0; i < myAL.Count; i++) {
            Console.Write("\t{0}", myAL[i]);
        }
        Console.WriteLine();
    }
}
```

程序运行结果如下：

```
myAL
        Count:      4
        Values: Hello   C#   World    !
        Sorted Values:   !    C#      Hello     World
```

这个例子本身很简单，但演示了 ArrayList 类的实例 myAL 线性表的元素数目可按需动态增加，可在表中任意位置进行插入和删除数据元素的操作，一般的数组不具备这种方便的特性。

【例2.2】　以顺序表求解约瑟夫环问题。

约瑟夫（Joseph）环问题：有 n 个人围坐在一个圆桌周围，把这 n 个人依次编号为 1，…，n。从编号是 s 的人开始报数，数到第 d 个人离席，然后从离席的下一个人重新开始报数，数到第 d 个人又离席，…，如此反复直到最后剩一个人在座位上为止。比如当 n=5，s=1，d=2 的时候，离席的顺序依次是 2，4，1，5，最后留在座位上的是 3 号。

解决这个问题的直接思路是：建立一个有 n 个元素的线性表，每个元素分别表示 n 个人，利用取模运算实现环形位置记录，当某人该出环时，删除表中相应位置的数据元素。实现这种直接思路的程序为：

```
using System;
using System.Collections;
    public class Joseph {
        public static void Main(string[] args) {
            JosephStart(5,1,2);
        }
        public static void Show(ArrayList alist){
            foreach(object o in alist){
                Console.Write(o + " ");
            }
            Console.WriteLine();
        }

        public static void JosephStart(int n,int s,int d) {
        // n为总人数，从第s个人开始数起，每次数到d。
            ArrayList aRing = new ArrayList();// List<int> aRing = new List<int>()
            int i,j,k;
            for(i=1;i<=n;i++)
                aRing.Add(i); //n个人依次插入线性表
            Show(aRing);
            i = s-2;          //第s个人的下标为s-1，i初始指向第s个人的前一位置
            k = n;                          //每轮的当前人数
            while(k>1) {                    //n-1个人依次出环
                j = 0;
                while(j<d) {
                    j++;                    //计数
                    i = (i+1)%k;            //取模运算实现环形位置记录
                }
                Console.WriteLine("out:  " + aRing[i]);
                aRing.RemoveAt(i);          //第i个人出环，删除第i个位置的元素
                k--; i = (i-1)%k;
                Show(aRing);
            }
            Console.WriteLine("\n{0} is the last person", aRing[0]);
        }
```

```
        }
    }
```

程序运行结果如下：

```
1 2 3 4 5
out:  2
1 3 4 5
out:  4
1 3 5
out:  1
3 5
out:  5
3
3 is the last person
```

上面这个解决方案的思路是简单直接的，但是算法的实际运行效率非常低，本章后面将进行分析和并实现改进的算法。

2.2 线性表的顺序存储结构

线性表的顺序存储结构是用一组连续的内存空间来顺序存放线性表的数据元素，数据元素在内存空间中的物理存储次序与它们的逻辑次序是一致的，即数据元素 a_i 与其前驱数据元素 a_{i-1} 及后继数据元素 a_{i+1} 无论在逻辑上还是在物理存储上，它们的位置都是相邻的。用顺序存储结构实现的线性表称为顺序表（Sequenced List）。

如前所述，线性表中的数据元素属于同一种抽象数据类型，因此每个数据元素在内存中占用的存储空间的大小是相同的。假设每个数据元素占据 c 个存储单元，第一个数据元素的地址为 $\mathrm{Loc}(a_0)$，它即是顺序表的起始地址，则第 i 个数据元素的地址为：

$$\mathrm{Loc}(a_i) = \mathrm{Loc}(a_0) + i \times c,\ i = 0, 1, 2, ..., n\text{-}1$$

可见，每个数据元素的地址是该元素在线性表中位置（或称下标）的线性函数，该地址可以直接由下标通过公式计算出来，而且每次寻址所花费的时间都是相同的。因此，顺序表中的每个数据元素是可以随机访问的，访问一个数据元素所需的时间与该元素的位置以及顺序表中元素的个数没有关系。顺序表的存储结构如图 2.1 所示。

下标	元素内容	元素地址
0	a_0	$\mathrm{Loc}(a_0)$
1	a_1	$\mathrm{Loc}(a_0)+c$
	...	
i	a_i	$\mathrm{Loc}(a_0)+i\times c$
i+1	a_{i+1}	
	...	
n-1	a_{n-1}	$\mathrm{Loc}(a_0)+(n-1)\times c$

图 2.1 线性表的顺序存储结构

2.2.1　顺序表的类型定义

C#程序设计语言中的数组可以得到连续的存储空间，因此数组可以作为实现顺序表的基础。本节将顺序表用 C#语言的自定义类刻画和实现出来，不妨将该类命名为 SequencedList<T>。在该类的定义中，字段 items 是一维数组变量，将记录数据元素所占用的存储空间，数组的元素类型标记为 T，即与泛型顺序表的类型参数 T 相同的类型；字段 count 表示顺序表的长度，即顺序表中元素的个数；字段 capacity 表示顺序表的当前容量，也就是数组 items 的当前容量大小。SequencedList<T>类声明如下：

```
public class SequencedList<T> {
    private T[] items;
    private int count = 0;
    private int capacity = 0;
    其他成员……
}
```

线性表的操作将作为 SequencedList 类的属性和方法成员予以实现。完整定义好该类后，就可用它来声明和构造实例，实例用来表示一个具体的线性表对象。当我们需要使用一个线性表时，就用 new 操作符创建 SequencedList 类的一个实例，而通过在这个对象上调用类中定义的公有（public）的属性和方法来操作线性表对象。

对于这个类的使用者来说，关心的是该类的功能，而不必关心类中的具体设计；另一方面，对这个类的设计者而言，不需要也不应该向类之外的对象提供直接操作数据成员的通道，所以设计者将类中的数据成员都声明为私有的（private），即设置其对外不可见。这就是面向对象程序设计所要求的类的封装性。

SequencedList 类的源代码编辑在 SequencedList.cs 文件中，该类及本章介绍的其他自定义线性表类都实现在名为 lists 的类库型项目中，它们与后续章节将介绍的实现其他相关数据结构的基础类，如栈类、二叉树类等，都统一定义在 DSAGL 命名空间，而将使用这些基础类的测试与应用类都各自独立地定义和实现在相应的项目和命名空间中，例如本章的测试与应用程序隶属于名为 liststest 的项目，命名空间的名称则采用了 Visual Studio 为项目中的新增代码自动选择的名称（通常与项目名称相同，对于本章即 liststest）。

2.2.2　顺序表的操作

1) 顺序表的初始化

使用类的构造方法创建并初始化顺序表对象，为顺序表实例预分配存储空间，并设置顺序表为空状态。该操作的实现编码如下：

```
public SequencedList(int c) {          //构造空的顺序表，分配c个存储单元
    capacity = c;
    items = new T[capacity];           // 申请capacity个存储单元
    count = 0;                         // 此时顺序表中元素个数为0，即长度为0
}
```

构造方法可以有多个重载形式，方便调用者以不同方式初始化顺序表对象。下面是一个重载的构造方法，不带参数，又称为缺省构造方法。自动构造具有 16 个存储单元的空表，编码如下：

```
public SequencedList() : this(16) { }
```

下面是另一个重载的构造方法，它以一个数组的多个元素作为初值来构造顺序表实例，该操作的实现编码如下：

```
//以一个数组的多个元素构造顺序表实例
public SequencedList(T[] itemArray) {
    count = itemArray.Length;          // 计算元素个数，即求表长度
    capacity = count + 16;
    items = new T[capacity];
    for (int i = 0; i < count; i++) {
        items[i] = itemArray[i];
    } // for循环结束
} // 方法体结束
```

2) 返回顺序表长度

该操作告知线性表实例中的数据元素的个数，将这个操作通过定义成类的公有整型属性 Count 来实现，编码如下：

```
public int Count{ get{ return count; } }
```

在 SequencedList 类中将返回顺序表长度的功能定义为类的属性（property）成员，功能上与将其定义为方法成员类似，但相对于后者，前者在使用时显得更简洁。方法的调用必须加上括号，而属性的调用无须括号。在前面的例子中我们也看到，C#用数组对象所具有的 Length 属性来获取数组的元素个数，如 a.Length 返回数组 a 的元素个数，此时 Length 后没有括号。

3) 判断顺序表的空状态和满状态

通过定义布尔类型的属性 Empty 来实现判断顺序表为空的功能，如果 Empty 返回值为 true，则表明顺序表为空；如果 Empty 返回值为 false，则表明顺序表为非空。

当 count 等于 0 时，顺序表为空状态，此时应设置 Empty 返回 true，否则返回 false。功能实现如下：

```
// 判断顺序表是否空
public bool Empty{ get{ return count==0;} }
```

通过定义布尔类型的属性 Full 来实现判断顺序表当前预分配的空间已满的功能，如果 Full 返回值为 true，则表明顺序表当前预分配空间已满；如果 Full 返回值为 false，则表明顺序表非满。

```
// 判断顺序表是否满
public bool Full{ get{ return count == items.Length;} }   // items.Length表示数组长度
```

后面将看到，顺序表预分配的存储空间可以而且应该根据需要而动态地调整，在使用顺序表进行应用编程时，一般可以认为系统所能提供的可用空间是足够大的，在绝大多数情况下不必判断顺序表是否已满。如果原分配的空间已用完，而系统无法分配新的存储空间，则产生运行时异常。

4) 获取或设置顺序表的容量

私有数据成员 capacity 表示顺序表的当前容量，定义公有属性 Capacity 供外部获取或设置顺序表的容量。获取顺序表的容量，仅是简单地返回数据成员 capacity 的当前值。设置顺序表的新容量，则要依次进行以下操作：重新分配指定大小的存储空间作为顺序表的“数据仓库”，将原数组中的数据元素逐个拷贝到新数组，并设置 capacity 的新值。后面将看到，在执行插入等操作时，当

count 等于 capacity 时，表明顺序表当前预分配的存储空间已装满数据元素，在进行后续的操作前，需要调用 Capacity 属性重新分配存储空间。该操作的实现编码如下：

```
public int Capacity {
    get { return capacity; }
    set {
        capacity = value;
        T[] copy = new T[capacity];          // 重新分配指定大小的存储空间
        if (count > capacity) count = capacity;
        Array.Copy(items, copy, count);      //将原数组中的元素拷贝到新数组
        items = copy;                        // assign items to the new array
    }
}
```

5) 获取或设置指定位置的数据元素值

通过定义索引器（indexer）来提供获得或设置顺序表的第 i 个数据元素值的功能，并实现对顺序表实例进行类似于数组的访问。就像 C#的数组下标从 0 开始一样，我们用从 0 开始的索引参数 *i* 来指示顺序表的第 *i* 个元素。该操作的实现编码如下：

```
// 声明索引器以提供对顺序表实例进行类似于数组的访问；获得或设置顺序表的第i个数据元素值
public T this[int i]{
    get{
        if(i>=0 && i<count)
            return items[i];
        else
            throw new IndexOutOfRangeException(
                "Index Out Of Range Exception in " + this.GetType() );
    }
    set{
        if(i>=0 && i<count) {
            items[i] = value;
        } else{
            throw new IndexOutOfRangeException(
                "Index Out Of Range Exception in " + this.GetType() );
        }
    }
}
```

6) 查找具有特定值的元素

在线性表中顺序查找具有特定值 *k* 的元素的过程为：从线性表的第一个数据元素开始，依次检查线性表中的数据元素是否等于 *k*，若当前数据元素与 *k* 相等，则查找成功；否则继续与下一个数据元素进行比较，当完成与线性表的全部数据元素的比较后仍未找到，则返回查找不成功的信息。上述操作分别用 Contains 方法和 IndexOf 方法实现，Contains 方法查找成功时返回 true，不成功则返回 false。IndexOf 方法查找成功时返回 *k* 值首次出现位置，否则返回-1。该操作的实现编码如下：

```
// 查找线性表是否包含k值,查找成功时返回true，否则返回false
```

```
public bool Contains(T k){
    int j = IndexOf(k);
    if(j!=-1)
        return true;
    else
        return false;
}
// 查找k值在线性表中的位置，查找成功时返回k值首次出现位置，否则返回-1
public int IndexOf(T k) {
    int j = 0;
    while( j<count && !k.Equals(items[j]) )
        j++;
    if(j>=0 && j<count)
        return j;
    else
        return -1;
}
```

7) 在顺序表的指定位置插入数据元素

在线性表指定位置上，插入一个新的数据元素，插入后，其所有元素仍构成一个线性表。要想在顺序表的第 i 个位置上插入给定值 k，使得插入后的线性表仍然保持连续性，首先必须将第 n-1 到第 i 个位置上的数据元素依次向后移动一个位置，即依次向后移动从 a_{n-1} 到 a_i 的数据元素，以空出第 i 个位置的内存单元，然后在第 i 个位置上放入给定值 k，其过程如图 2.2 所示，图中 n 表示数组数据元素的个数，它等于顺序表的 Count 属性。

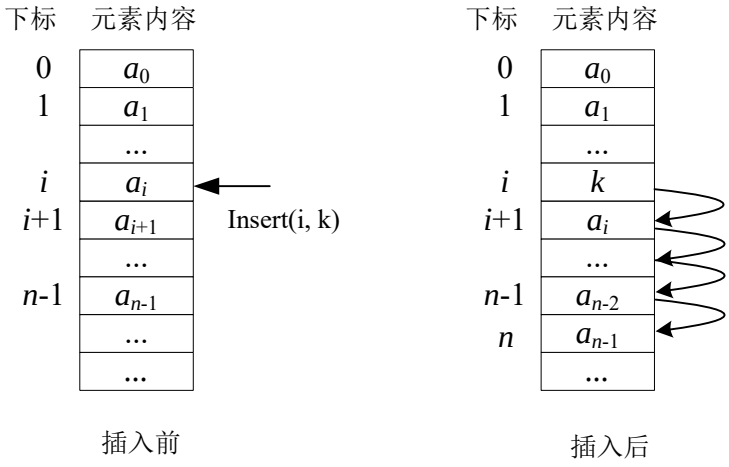


图 2.2 顺序表中插入数据元素

该操作的实现编码如下：

```
// 在顺序表的第i个位置上插入数据元素k
public void Insert(int i,T k) {
    if (count >= capacity) { // 若顺序表的当前空间已满，需要调用Capacity属性重新分配存储空间
        Capacity = capacity * 2;        // double capacity
```

```
        }
        if (i < 0) i = 0;
        if (i > count) i = count;
        if (i < count) {
            for (int j = count - 1; j >= i; j--)
                items[j + 1] = items[j];
        }
        items[i] = k;
        count++;
        return;
    }
```

一种常见的插入操作是在线性表的表尾添加（Add）一个新元素，算法实现如下：

```
    // 将k 添加到顺序表的结尾处
    public void Add(T k) {
        // see if array needs to be resized
        if (count >= capacity) {           // resize array
            Capacity = capacity * 2;      // double capacity
        }
        items[count] = k;
        count++;
    }
```

在执行插入或添加操作时，当 count 等于 capacity 时，表明顺序表当前预分配的存储空间已装满数据元素，在进行后续的操作前，需要调用 Capacity 属性重新分配存储空间。

8) 删除顺序表指定位置的数据元素

删除线性表指定位置的数据元素，同时保证更改后的线性表仍然具有线性表的连续性。若想删除顺序表的第 i 个数据元素，使得删除后线性表仍然保持连续性，则必须将顺序表中原来的第 i+1 到第 n-1 位置上的数据元素依次向前移动。数据元素移动次序是从 a_{i+1} 到 a_{n-1}，将 a_{i+1} 移动到位置 i 上，实际上就是删除了 a_i，如图 2.3 所示，图中 n 表示数组中数据元素的个数，它等于顺序表的 Count 属性。

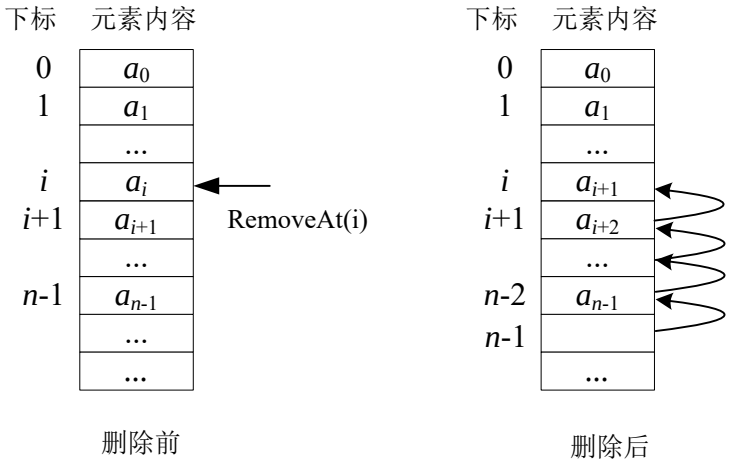


图 2.3 删除顺序表中的数据元素

算法实现如下：

```
// 删除顺序表的第i个数据元素
public void RemoveAt(int i) {
    if(i>=0 && i<count){
        for(int j=i+1;j<count;j++)
            items[j-1] = items[j];
        count --;
    } else
        throw new IndexOutOfRangeException(
            "Index Out Of Range Exception in" + this.GetType() );
}
```

一种常见的删除操作是从表中移除特定对象的第一个匹配项，算法实现如下：

```
// 删除顺序表中首个出现的k值数据元素
public void Remove(T k) {
    int i = IndexOf(k);                  //查找k值的位置i
    if(i!=-1){
        for(int j=i+1;j<count;j++)       //删除第i个值
            items[j-1] = items[j];
        count --;
    } else
        Console.WriteLine(k + "值未找到，无法删除!" );
}
```

9) 输出顺序表

```
public void Show(bool showTypeName){
    if(showTypeName)
        Console.Write("SequencedList: ");
    for (int i = 0; i < this.count; i++) {
        Console.Write(items[i] + "  ");
```

```
        }
        Console.WriteLine();
    }
```

这是本类设计的一个实用工具方法，它在控制台上显示顺序表对象的内容。

比在控制台上显示信息更为一般的操作，是以字符串的形式返回对顺序表对象有意义的值，这可以通过在顺序表 SequencedList 类中重写（override）从 Object 类继承的 ToString 方法来实现。按照 C#语言的惯例，在实现一个自定义类时，一般需重写 ToString 方法，使得它以字符串的形式返回对该类型对象有意义的值。

```
    public override string ToString() {
        StringBuilder s = new StringBuilder();
        for (int i = 0; i < this.count; i++) {
            s.Append(items[i]);
            s.Append("; ");
        }
        return s.ToString();
    }
```

2.2.3　顺序表操作的算法效率分析

上一节给出了顺序表基本操作的实现代码，从中可以看出，不同的操作可能会有不同的效率特性。有些操作实现所蕴涵的元操作的执行次数与表中数据元素的个数无关，例如判断顺序表的空状态、返回顺序表的长度、获取或设置指定位置的数据元素的值等操作，所以这些操作的时间复杂度为 $O(1)$；而有些操作实现所蕴涵的元操作的执行次数与表中数据元素的个数有关，例如在线性表中查找给定值、插入和删除数据元素等操作，所以这些操作的时间复杂度为 $O(n)$。

如果暂不考虑有时可能会发生的内存空间的重分配问题，在顺序表中进行插入和删除数据元素的操作时，算法所花费的时间主要用在移动数据元素。插入的位置不同，则移动数据元素的次数也不同；若在表头（即表的第 0 个位置）插入新的数据，则移动操作的次数为 n（n 为顺序表中数据元素的个数）；若在顺序表的最后一个位置插入新的数据，则移动操作次数为 0。设在第 i 个位置插入新的数据的概率为 p_i，则在顺序表中插入一个数据元素所做的平均移动次数为

$$\sum_{i=0}^{n}(n-i)\times p_i$$

如果在各位置插入数据元素的概率相同，即

$$p_0 = p_1 = \cdots = p_n = \frac{1}{n+1}$$

则插入操作的平均移动次数为

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

上述分析表明，插入一个数据元素的操作，在等概率条件下，平均需要移动线性表全部数据元素的一半，所以插入操作的时间复杂度为 $O(n)$。

同理，在等概率条件下，删除一个数据元素的操作平均需要移动线性表全部数据元素的一半，

所以删除操作的时间复杂度也为 $O(n)$。

综上所述，顺序表具有以下特点：

1) 随机访问：顺序表的存储次序直接反映了其逻辑次序，可以直接访问第 i 个数据元素；
2) 存储密度高：所有的存储空间都可以用来存放数据元素；
3) 插入和删除操作的效率不高：每插入或删除一个数据元素，都可能需要移动大量的数据元素，其平均移动次数是线性表长度的一半；
4) 预分配数组空间时，需要给出数组存储单元的个数，这个数值只能根据不同的情况估算。可能出现因空间估算过大而造成系统内存资源的浪费，或因空间估算过小而在随后的某个操作中重新分配存储空间。

【例2.3】 以顺序表求解约瑟夫环问题的改进算法。

在【例 2.2】中实现的算法多次使用删除操作，即每当一个数据元素出环时，删除表中相应位置的元素，这时必须移动其他元素（ArrayList 类完成该操作），操作的时间复杂度高。为了避免这个问题，下面的程序没有使用删除操作，而是采用了一种设置特殊标志的变通方法，将应出环元素相应位置的值置为零（将 0 作为空单元的标志），以后在计数时跳过值为零的单元。当 n=5，s=1，d=2 时，基于这种思路的约瑟夫环问题执行过程如图 2.4 所示。程序实现代码如下：

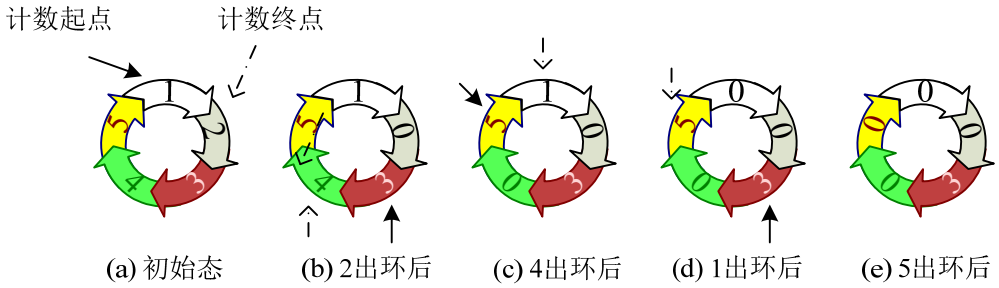


图 2.4 设置空标志 0 的约瑟夫环执行过程

```
using System;
using DSAGL;
namespace liststest {
    public class Joseph1 {
        public static void Main(string[] args) {
            (new Josephus1()).JosephStart(5,1,2);
        }

        public void JosephStart(int n,int s,int d) {
            const int KilledValue = 0;
            int i,j,k;
            int[] a = new int[n];
            for (i = 0; i < n; i++)                    //n个人依次插入线性表
                a[i] = i + 1;
            SequencedList<int> ring1 = new SequencedList<int>(a);
            ring1.Show(true);
            i = s-2;                    //第s个人的下标为s-1，i初始指向第s个人的前一位置
            k = n;                      //每轮的当前人数
```

```
            while(k>1) {            //n-1个人依次出环
                j = 0;
                while(j<d) {
                    i = (i+1)%n;                    //将线性表看成环形
                    if(ring1[i]!=KilledValue)
                        j++;                        //计数，但跳过值为空的单元
                }
                Console.WriteLine("out:  " + ring1[i]);
                ring1[i] = KilledValue;             //第i个人出环,设置第i个位置为空
                k--;
                ring1.Show(true);
            }
            i = 0;
            while(i<n && ring1[i]==KilledValue)     //寻找最后一个人
                i++;
            Console.WriteLine("The last person is " + ring1[i]);
        }
    }
}
```

粗略地测算一下前后两种算法的运行速度：如果去掉算法中显示中间结果的语句，对于 n=50000 的情况，【例 2.2】中的算法耗时 1.828 秒，而本例的算法耗时仅 0.016 秒，相差 100 多倍。

2.3　线性表的链式存储结构

线性表的链式存储结构是指将线性表的数据元素分别存放在一个个结点（Node）中，每个结点由数据元素域和一个或若干个指针域组成，指针用来指向其他结点。这样，线性表数据元素之间的逻辑次序就由结点间的链接关系来实现。用链式存储结构实现的线性表称为线性链表（Linear Linked List），简称链表（Linked List）。

指向线性链表第一个结点的指针称为线性链表的头指针，一个线性链表由头指针指向链表的头结点（Head Node），头结点的链指向第一个数据结点（First Node），每个数据结点的链指向其后继结点，最后一个结点的链为空（null）。链表的数据结点个数称为链表的长度，长度为 0 时称为空表。

线性链表根据链的个数分为单向链表和双向链表两种。

2.3.1　线性链表的结点结构

线性表的数据元素分别存放在链表的结点中，结点由值域和指针域组成，值域保存数据元素的值，指针域则包含指向其他结点的引用（即指针），指针域又称链域。这样，线性表数据元素之间的逻辑次序就由结点间的链接来实现。

在 C#程序中不能直接使用类似于 C 语言中的指针，但 C#中用类类型（class type）定义的变量都属于引用类型（Reference Type）变量，用来指向具体的对象，或称实例。引用类型的变量保存的内容，不是某个对象（实例）自身的内容，而是到对象的引用，它在功能上起着记录实例地址的作用。因此，在 C#程序设计中可以定义“自引用的类”（Self Referential Class）表示链表的结点结构。

1. 声明自引用的结点类

自引用的类包含一个属于同一类型对象的成员，因为对象类型的变量为引用类型，该成员存储的内容是某个对象（实例）的引用，实际起着记录对象地址的作用。例如，

```
public class SingleLinkedNode<T> {
    T item;                          //数据域，存放结点值
    SingleLinkedNode<T> next;        //指针域，后继结点的引用
    ……
}
```

SingleLinkedNode<T>类的定义中声明了两个成员变量：item 和 next。成员 item，构成结点的值域，用于记载（结点）数据；成员 next 构成结点的链域，用于引用同类的某个对象，例如数据集合中的其他结点。所以 SingleLinkedNode<T>类构成自引用的类。实例变量 next 称为链（link），它是一种引用类型，在功能上类似于 C/C++语言中的指针，将一个 SingleLinkedNode<T>类的对象与另一个同类型的对象“连接”起来，实现结点间的链接。

我们在本章将链表中的结点设计成独立的 SingleLinkedNode 类，一个结点就是用 SingleLinkedNode 类定义和创建的一个实例。C#中定义的类都是引用类型，通过用自引用类型的链域将多个实例（结点对象）链接起来，就可以实现多种动态的数据结构，如链表、二叉树、图等结构，在后面的章节中我们还会多次用到这种方法。

2. 创建并使用结点对象

创建和维护动态数据结构需要动态内存分配（Dynamic Memory Allocation），即一个程序在运行时申请所需的内存空间，系统分配内存后程序才可使用，使用完后释放不再需要的空间。

C#使用 new 操作符创建对象并为之分配内存。例如，

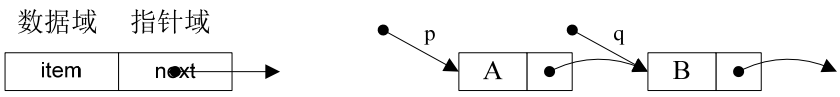
```
SingleLinkedNode<string>   p, q;          //声明 p 和 q 是 SingleLinkedNode<string>类的变量
p = new SingleLinkedNode<string>();       //创建 SingleLinkedNode 类的一个对象，由 p 引用
q = new SingleLinkedNode<string>();       //创建 SingleLinkedNode 类的一个对象，由 q 引用
```

如果没有可用内存，new 操作产生一个 OutOfMemoryException 类型的异常。

结点对象的两个成员变量 item 和 next 记录该实例的状态，称为实例变量，由 p 引用对象的这两个实例变量的语法格式为 p.item 和 p.next。通过下述语句可将 p、q 两个结点对象链接起来：

```
p.next = q;
```

这时，称结点对象 p 的 next 成员变量指向结点对象 q，简称结点 p 指向结点 q。链表的结点结构和链接起来的两个结点如图 2.5 所示。



(a) SingleLinkedNode类　　(b) 结点及其链接

图 2.5 结点结构及其链接

2.3.2 单向链表

在线性链表中，如果每个结点只有一个链，则只能表达一种链接关系，这种结构称为单向链表（Single Linked List），单向链表各结点的链指向其后继结点。在单向链表的实现中，一种常用的方式

是在链表的第一个数据结点之前附设一个结点，称之为头结点（Head Node）。头结点的数据域可以存储任意信息，而该结点的链域则存储第一个数据结点（First Node）的引用。

图 2.6 是一个单向链表的结构示意图，图中头指针 head 指向头结点，若线性表为空，则头结点的链域为 null，否则指向第一个数据结点。从头指针 head 开始，沿链表的方向前进，就可以顺序访问链表中的每个结点。

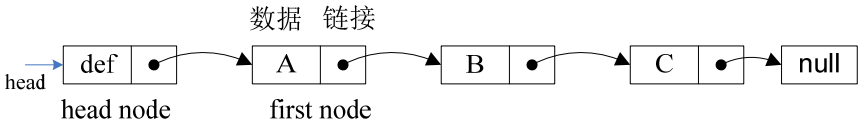


图 2.6　单向链表结构示意图

1. 单向链表的结点类

用 C#语言描述单向链表的结点结构，声明泛型类 SingleLinkedNode 如下：

```
public class SingleLinkedNode<T> {
    private T item;                          //存放结点值
    private SingleLinkedNode<T> next;        //后继结点的引用

    //构造值为k的结点
    public SingleLinkedNode(T k) { item = k; next = null;   }
    //无参数时构造缺省值的结点
    public SingleLinkedNode() { next = null;}

    //获取和设置结点值
    public T Item {
        get { return item; }
        set { item = value; }
    }

    //获取和设置链值
    public SingleLinkedNode<T> Next {
        get { return next; }
        set { next = value; }
    }

    //输出以本结点为第一结点的单向链表
    public void Show() {
        SingleLinkedNode<T> p = this;
        while (p != null) {
            Console.Write(p.Item);
            p = p.Next;
            if (p != null)
                Console.Write(" -> ");
```

```
                Else
                    Console.Write(".");
            }
            Console.WriteLine();
        }

        //重写ToString方法
        public override string ToString() {
            StringBuilder s = new StringBuilder();
            SingleLinkedNode<T> p = this;
            while (p != null) {
                s.Append(p.Item);
                p = p.Next;
                if (p != null) s.Append(" -> ");
                else s.Append(".");
            }
            return s.ToString();
        }
    }
```

SingleLinkedNode 类声明了单向链表的结点类型，用这个类型定义和创建的实例即表示一个具体的链结点对象，通过它的成员变量 next 的引用方式，指向其他结点，以实现线性表中各数据元素的逻辑关系。SingleLinkedNode 类中成员变量 next 和 item 设计为私有的（private），不允许其他类直接访问。但 SingleLinkedNode 类提供公有属性 Next 和 Item 允许外部模块来间接访问 next 和 item。这种设计满足面向对象编程封装性的要求。

2. 单向链表类

用 C#语言描述单向链表，声明泛型类 SingleLinkedList<T>如下：

```
public class SingleLinkedList<T> {
    private SingleLinkedNode<T> head;          //指向链表的头结点
    public SingleLinkedNode<T> Head {
        get { return head; }
        set { head = value; }
    }
    ... ...
}
```

线性表的各种操作将在 SingleLinkedList 类的多个属性和方法成员中予以实现。一个 SingleLinkedList 类型的实例（对象）表示一条具体的单向链表，它的成员变量 head 作为该链表的头指针，指向链表中仅作为标志的头结点，头结点的链域（head.Next）指向第一个数据结点。当头结点的链域为 null 时，表示链表为空，其元素个数为 0。

3. 单向链表的操作

单向链表的常用操作通过下述 SingleLinkedList 类的若干属性或方法成员实现。

1) 建立单向链表，单向链表的初始化

用 SingleLinkedList 类的构造方法建立一条空链表，操作实现如下：

```
// 构造空的单向链表
public SingleLinkedList() {
    head = new SingleLinkedNode<T>();          //构造头结点，它仅是个标志结点
}
```

构造一条单向链表，并使其第一个数据结点为指定的结点：通过重载 SingleLinkedList 类的构造方法来实现。代码实现如下：

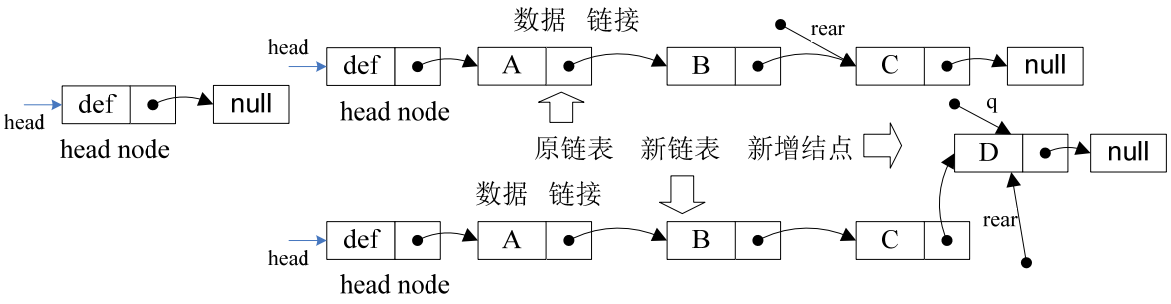
```
//构造由参数f所指向结点为第一个数据结点的单向链表
public SingleLinkedList(SingleLinkedNode<T> f): this() { head.Next = f;}
```

用某一数组中的一组数值初始化一个单向链表：通过重载 SingleLinkedList 类的构造方法来实现。在建立单向链表的过程中，使用 new 操作符创建 SingleLinkedNode 类型的对象作为一个新的结点，并依次链入链表的末尾，如图 2.7 所示。设变量 rear 指向原链表的最后一个结点，q 指向新创建的结点，则下列语句将 q 结点链在 rear 结点之后，并更新 rear，使其指向新链尾结点：

```
rear.Next = q;              // q结点链入原链表尾
rear = q;                   // 更新rear，指向新链尾结点
```

这样就将 q 作为最后一个结点链入到表中，重复上述操作可以建立一条单向链表，该初始化操作实现如下：

```
//以一个数组的多个元素构造单向链表
public SingleLinkedList(T[] itemArray): this() {
    SingleLinkedNode<T> rear, q;
    rear = head;                                    // rear指向链表尾结点
    for (int i = 0; i < itemArray.Length; i++) {
        q = new SingleLinkedNode<T>(itemArray[i]);  //建立结点q
        rear.Next = q;
        rear = q;
    }
}
```



(a) 初始态-空表　　(b) 向链表中增加数据

图 2.7　建立单向链表

2) 返回链表的长度

该操作告知线性链表的数据元素个数，假设没有设立专门的成员变量记录表中的元素个数，当

需要知道元素的数目时，必须从第一个数据结点计数到最后一个结点，编码如下：

```
public virtual int Count{
    get {
        int n = 0;
        SingleLinkedNode<T> p = head.Next;
        while (p != null) {
            n++;
            p = p.Next;
        }
        return n;
    }
}
```

3) 判断单向链表是否为空

用属性 Empty 实现该操作，如果 Empty 返回值为 true，则表明表为空；如果 Empty 返回值为 false，则表明表为非空。当头结点的链域（即 head.Next）为 null 时，表示链表为空，Empty 应指示 true；否则，指示 false。编码如下：

```
public virtual bool Empty{
    get{ return head.Next == null;}
}
```

在链表的实现中，采用动态分配方式为每个结点分配内存空间，当有一个数据元素需要加入链表时，就向系统申请一个结点的存储空间，在编程时可认为系统所提供的可用空间是足够大的，一般不必判断链表是否已满。如果内存空间已用完，系统无法分配新的存储单元，则产生运行时异常。

如果也想在链表类中定义一个 Full 属性，可以让它总是返回 false，编码如下：

```
public virtual bool Full{
    get{ return false;}
}
```

4) 获取或设置指定位置的数据元素值

在链表的实现中，不能像顺序结构一样根据数据结点的序号直接找到该结点。在单向链表的每个结点中都有一个指向后继结点的链域，如果以索引参数 *i* 来指定结点的位置，则必须从表头顺着链找到相应的结点，以达到进一步获取或设置该结点的值的目的。

我们仍然用从 0 开始的索引参数 *i* 来指示线性链表的第 i 个数据元素。操作实现如下：

```
public virtual T this[int i] {
    get {
        if (i < 0)
            throw new IndexOutOfRangeException(
                "Index is negative in " + this.GetType());
        int n = 0;                              // count of elements
        SingleLinkedNode<T> q = head.Next;
        while (q != null && n != i) {
            n++;
            q = q.Next;
```

```
            }
            if (q == null)
                throw new IndexOutOfRangeException(
                    "Index Out Of Range Exception in " + this.GetType());
            return q.Item;
        }
        set {
            if (i < 0)
                throw new IndexOutOfRangeException(
                    "Index is negative in " + this.GetType());
            int n = 0;                      // count of elements
            SingleLinkedNode<T> q = head.Next;
            while (q != null && n != i) {
                n++;
                q = q.Next;
            }
            if (q == null)
                throw new IndexOutOfRangeException(
                    "Index Out Of Range Exception in " + this.GetType());
            q.Item = value;
        }
    }
```

5) 输出单向链表

将已建立的单向链表按顺序在控制台输出其每个结点的值。从 head.Next 所指向的结点（这是线性表的第一个数据结点）开始，首先访问结点，再沿着其链方向到达后继结点，访问该结点，直至达到链表的最后一个结点。

设 p 指向链表中的某结点，由结点 p 到达 p 的后继结点的语句是：p = p.Next;

输出整个链表的操作实现编码如下：

```
    // 输出head指向的单向链表
    public virtual void Show(bool showTypeName) {
        if (showTypeName) {
            Console.Write("SingleLinkedList:  ");
        }
        SingleLinkedNode<T> q = head.Next;
        if (q != null) q.Show();
    }
```

比在控制台上显示更为一般的操作，是以字符串的形式返回对链表对象有意义的值，这可以通过在链表 SingleLinkedList 类中重写从 Object 类继承的 ToString 方法来实现。

```
    public override string ToString() {
        SingleLinkedNode<T> q = head.Next;
        if (q != null)
            return q.ToString();
```

```
        else
            return null;
    }
```

上述代码分别调用 SingleLinkedNode 类中定义的同名方法。

6) 插入结点

在单向链表中插入新的结点，如果以索引参数 *i* 来指定结点的位置，则必须先从表头顺着链找到相应的结点，再插入新的结点，过程如图 2.8 所示。

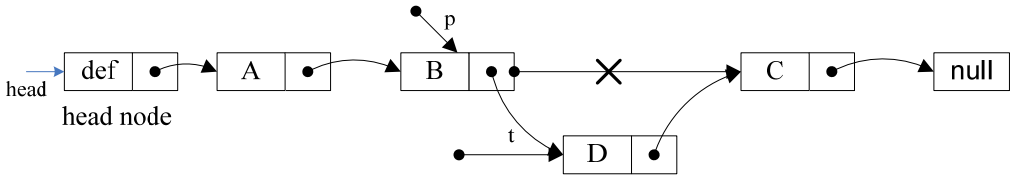


图 2.8 单向链表插入结点

生成值为 k 的新结点并做相应准备工作如下：

```
SingleLinkNode<T>  p, q;
SingleLinkNode<T>  t = new SingleLinkedNode<T>(k);
```

找到正确的插入位置后，设 p 指向链表中的某结点，在结点 p 之后插入结点 t，形成新的链表。语句如下：

```
t.Next = p.Next;
p.Next = t;
```

由此可见，在单向链表中插入结点，只要修改相关的几条链，而不需移动数据元素。完整的操作实现编码如下：

```
    public virtual void Insert(int i, T k) {
        int j = 0;
        SingleLinkedNode<T> p = head;
        SingleLinkedNode<T> q = p.Next;
        if (i < 0) i = 0;
        SingleLinkedNode<T> t = new SingleLinkedNode<T>(k);
        while (q != null) {
            if (j == i) break;
            p = q;
            q = q.Next;
            j++;
        }
        t.Next = p.Next;
        p.Next = t;
    }
```

由于在单向链表中无法直接访问结点的前驱结点，所以算法中设置 p 作为 q 的前驱结点，q 每前进一步，p 也跟随前进。

7) 删除结点

要在单向链表中删除指定位置的结点，需要把该结点从链表中退出，并改变相邻结点的链接关系。该结点所占用的存储单元，在 C/C++语言实现中必须归还给系统，而在 C#语言实现中，该结点所占的存储单元由系统管理，适时自动回收。

删除结点的操作过程如图 2.9 所示，设 p 指向单向链表中的某一结点，删除 p 的后继结点 q 的语句是：

```
p.Next = q.Next;
```

执行该操作前要根据不同的要求定位将被删除的结点 q 和它的前驱结点 p。

执行上述语句之后，则建立了新的链接关系，替代了原链接关系。因此，在单向链表中删除结点，只要修改相关的几条链，而不需移动数据。

如果需要删除结点 p 自己，则必须修改 p 前驱结点的 next 链。由于单向链表中的结点没有指向前驱结点的链，无法直接修改 p 前驱结点的 next 链。所以，在单向链表中，要删除 p 的后继结点，操作简单；而要删除结点 p 自己，则操作比较麻烦。

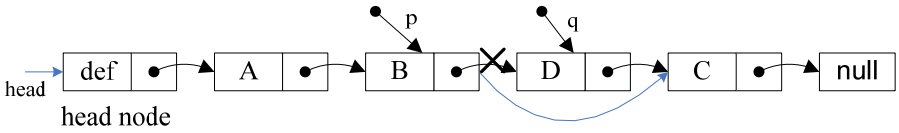


图 2.9　单向链表删除结点

删除结点操作的实现程序如下：

```
// 删除链表中首个出现的k值数据元素
public void Remove(T k) {
    SingleLinkedNode<T> p = head;
    SingleLinkedNode<T> q = p.Next;
    while (q != null) {
        if (k.Equals(q.Item)) {
            p.Next = q.Next;
            return;
        }
        p = q;
        q = q.Next;
    }
    Console.WriteLine(k + "值未找到，无法删除!");
}

// 删除链表的第i个数据元素
public void RemoveAt(int i) {
    int j = 0;
    SingleLinkedNode<T> p = head;
    SingleLinkedNode<T> q = p.Next;
    while (q != null) {
        if (j == i) {
            p.Next = q.Next;
            return;
```

```
            }
            p = q;
            q = q.Next;
            j++;
        }
        throw new IndexOutOfRangeException(
            "Index Out Of Range Exception in" + this.GetType());
    }
```

8) 单向链表逆转

设已建立一条单向链表，现欲将各结点的链域 next 改为指向其前驱结点，使得单向链表逆转过来。算法描述如图 2.10 所示。

设 p 指向链表的某一结点，front 和 q 分别指向 p 的前驱和后继结点，则使 p.Next 指向其前驱结点的语句是：

```
p.Next = front;
```

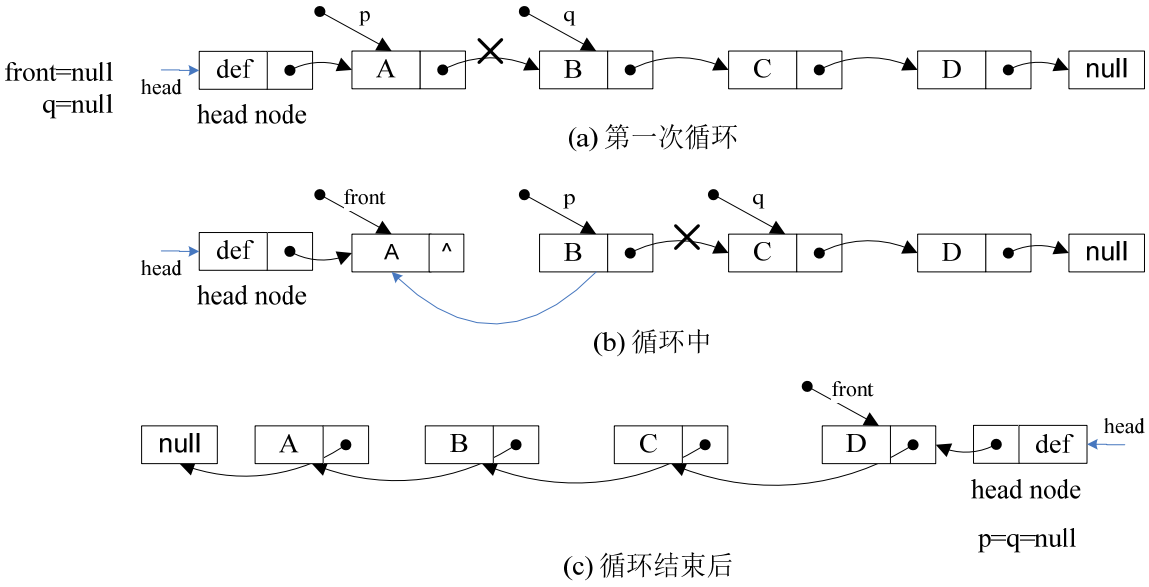


图 2.10 单向链表逆转

单向链表逆转算法描述如下：

- 第 1 次循环时，front = null，p 指向链表的第一个数据结点，执行 p.Next = front 语句。
- 以 p!=null 为循环条件，front、p 和 q 沿链表方向依次前进，对于 p 指向的每一个结点，执行语句 p.Next = front。
- 循环结束后，front 指向原链表的最后一个结点，该结点应成为新链表的第 1 个数据结点，需由 head.Next 指向，语句为：head.Next = front;

【例2.4】 单向链表逆转算法实现与测试。

下面的程序代码，Reverse 方法是在 SingleLinkedList 类中定义的，而 SingleLinkedList 类是在 DSAGL 命名空间中声明的，测试类 SingleLinkedListTest 则声明于 liststest 命名空间中，故在测试类代码中，需加上语句“using DSAGL;”，编译该源程序时则需指明引用 lists 类库。

```
public virtual void Reverse() {
```

```
        SingleLinkedNode<T> q = null, front = null;
        SingleLinkedNode<T> p = head.Next;
        while (p != null) {
            q = p.Next;
            p.Next = front;                    //p.next指向p结点的前驱结点
            front = p;
            p = q;
        }
        head.Next = front;
    }
```

测试类代码如下：

```
using System;
using DSAGL;
namespace liststest {
    class SingleLinkedListTest {
        public static void Main(string[] args) {
            int[] ia = new int[8];
            RandomizeIntArray(ia);
            SingleLinkedList<int> a = new SingleLinkedList<int>(ia); // 以8个随机值建立单向链表
            a.Show(true);
            Console.WriteLine("Reverse!");
            a.Reverse(); a.Show(true);
        }

        //产生一个随机数数组
        static void RandomizeIntArray(int[] ia)  {
            Random random = new Random();
            for (int i = 0; i < ia.Length; i++) {
                ia[i] = random.Next(100);            //产生随机数
            }
            return;
        }
    }
}
```

程序运行结果如下：

```
    DSAGL.SingleLinkedList:  55 -> 47 -> 5 -> 56 -> 8 -> 59 -> 86 -> 4.
    Reverse!
    DSAGL.SingleLinkedList:  4 -> 86 -> 59 -> 8 -> 56 -> 5 -> 47 -> 55.
```

4. 两种存储结构性能的比较

从以下几个方面的对比，让我们来看一看线性表的两种存储结构各自的优缺点：

（1）元素的随机访问特性

顺序表能够如同访问数组元素一样，直接访问数据元素，即顺序表可以根据元素的下标直接引用任意一个数据元素；而链表不能直接访问任意指定位置的数据元素，只能从链表的第一个结点开始，沿着链的方向，依次查找后继结点，直至到达指定的位置，才可以访问该结点的数据元素。

（2）插入和删除操作

顺序表的插入和删除操作很不方便，插入和删除操作有时需要移动大量元素；而链表则容易进行插入和删除操作，只要简单地改动相关结点的链即可，不需移动数据元素。

（3）存储密度

顺序表每个单元（即数据结点）的存储密度高，数据结点的全部空间都用来存放数据元素。而链表的结点存储密度较低，每个结点不仅要包含数据的值，还要包含其后继结点的引用。

（4）存储空间的动态利用特性

顺序表中不易动态利用存储空间，例如进行插入操作时，要判断顺序表预分配的存储空间是否已满，当原空间已满时，则需重新分配存储空间，并将原空间中的数据拷贝到新的空间，然后才进行插入操作。预分配的存储空间如果过多，会造成空间的浪费；过小，则会造成频繁的存储空间重分配的问题。而在链表中插入一个结点，程序会动态地向系统申请一个存储单元，只要系统资源够用，就会分配到需要的存储空间，所以链表进行插入操作时无需判断是否已满，也没有数据移动的问题。

（5）查找和排序

顺序表具有元素的随机访问特性，查找和排序可以较方便地实施多种算法，如折半查找和快速排序算法等。在链表中实施一些查找和排序算法相对复杂。本课程后面的相关章节将具体介绍查找和排序算法的不同实现。

2.3.3 单向循环链表

如果在单向链表中，将最后一个结点的链域设置为指向链表的头结点，则这样的链表呈现为环状，称为单向循环链表（Circular Linked List），如图 2.11 所示。

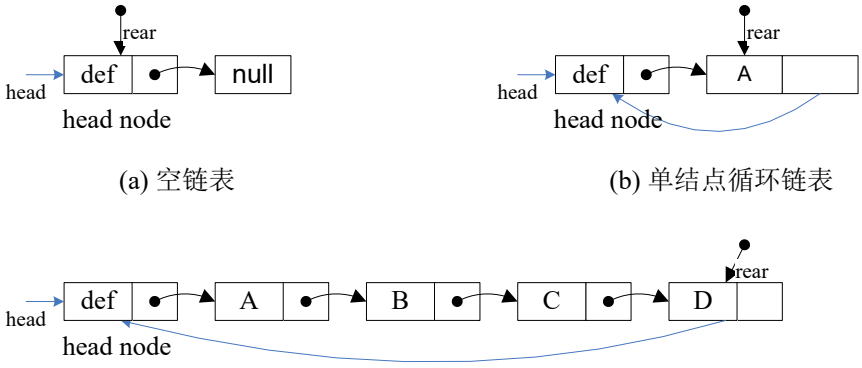


图 2.11 单向循环链表

在循环链表中设置了一个仅作为开始标志的头结点（Head Node），链表的 Head 属性成员指向头结点，头结点的链域（Head.Next）指向第一个数据结点。设置成员变量 rear 指向循环链表的最后一个数据结点（相对第一个数据结点而言），所以有 rear.Next 等于 Head，rear 起着尾指针的作用。

当 Head.Next = = null 或 Head = = rear 时，循环链表为空，如图 2.11（a）所示。当 Head.Next.Next = = Head 时，循环链表只有一个数据结点，如图 2.11（b）所示的单结点循环链表。一般情况则如图 2.11（c）所示，单向循环链表的所有结点链接成一条环路，即从链表中任意一结点出发，沿着链的方向，访问链表中所有结点之后，又回到出发点。

循环链表的结点与普通的单向链表的结点类型相同，而且循环链表类的实现也不必从头设计，我们可以利用面向对象技术，从单向链表类 SingleLinkedList 中导出（派生）一个新类作为循环链表类的实现。

用 C#语言描述单向循环链表，声明泛型类 CircularLinkedList<T>如下：

```
public class CircularLinkedList<T> : SingleLinkedList<T> {
    private SingleLinkedNode<T> rear;
    public override SingleLinkedNode<T> Rear {get { return this.rear; }
    ... ...
}
```

一个 CircularLinkedList 类型的对象表示一条单向循环链表，该类继承自 SingleLinkedList 类，继承的公有属性 Head 作为链表的头指针，指向链表中仅作为标志的头结点，头结点的链域则指向第一个数据结点。派生类也继承了基类的其他公有属性和方法；对基类中声明的虚方法或虚属性，派生类可以重写（override），即为声明的方法或属性提供新的实现。这些方法/属性在基类中用 virtual 修饰符声明，而在派生类中，将被重写的方法/属性用 override 修饰符声明。

循环链表的操作将作为 CircularLinkedList 类的属性和方法成员予以实现。部分操作的实现代码如下：

1) 单向循环链表的初始化

用 CircularLinkedList 类的构造方法建立一条循环链表，算法如下：

```
public CircularLinkedList(): base() {
    this.rear = this.Head;
}
public CircularLinkedList(T[] itemArray): this() {
    SingleLinkedNode<T> q = null;
    for (int i = 0; i < itemArray.Length; i++) {
        q = new SingleLinkedNode<T>(itemArray[i]);
        rear.Next = q;
        rear = q;
    }
    q.Next = Head;
}
```

循环链表的初始化操作与普通链表的初始化操作类似，主要差别在于合理地设置导出类的新成员 rear。

2) 返回链表的长度

```
public override int Count {
    get {
        int n = 0;
        SingleLinkedNode<T> p = Head.Next;
        if (p == null) return 0;
```

```
            while (p != Head) {
                n++;
                p = p.Next;
            }
            return n;
        }
    }
```

3) 判断单向链表是否为空

```
public override bool Empty {
    get {return Head == rear;}
}
```

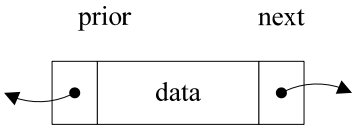
当尾结点指针 rear 等于头结点指针 Head 时，说明循环链表仅包含一个头结点，而没有数据结点。

2.3.4 双向链表

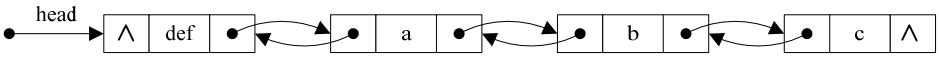
前面介绍的单链线性链表，每个结点只有一个链，也就只能表达一种链接关系，一般情况下，链指向后继的结点，结点中并没有记载前驱结点的信息。所以，单向链表的这种结构对于向后的操作较方便，而对向前的操作则很不方便。例如要查找某结点的前驱结点，每次都必须从链表的头指针开始沿着链表方向逐个结点进行检测。

如果在结点结构中再增加一个链用于指向前驱结点，则会产生一种双向链表，它会极大地方便实现既向前又向后的操作。

双向链表（Doubly Linked List）的每个结点除了保存数据的成员变量 item 之外，还有两个作为链的成员变量：prior 指向前驱结点，next 指向后继结点。图 2 12 给出了双向链表的结构示意图。



(a) 双向链表中的结点结构



(b) 双向链表

图 2 12 双向链表结构示意图

1. 双向链表的结点类

为描述具有双链的结点结构，用 C#语言声明 DoubleLinkedNode<T>类如下：

```
public class DoubleLinkedNode<T> {
    private T item;                              //存放结点值
    private DoubleLinkedNode<T> prior, next;     //前驱与后继结点的引用
```

```
        //构造值为k的结点
        public DoubleLinkedNode(T k) {item = k; prior = next = null;}
        //无参数时构造缺省值的结点
        public DoubleLinkedNode() { prior = next = null; }
        public T Item {
            get { return item; }
            set { item = value; }
        }
        public DoubleLinkedNode<T> Next{ get { return next; } set { next = value;} }
        public DoubleLinkedNode<T> Prior{ get { return prior;} set { prior = value;} }
    }
```

用 DoubleLinkedNode<T>类定义和构造的实例即可表示双向链表中的一个结点对象。

2. 双向链表类

用 C#语言描述双向链表结构，声明 DoubleLinkedList<T>类如下：

```
    public class DoubleLinkedList<T> {
        private DoubleLinkedNode<T> head;          //指向链表作为标志的头结点
        // 构造空的双向链表
        public DoubleLinkedList() {
            head = new DoubleLinkedNode<T>();      //头结点是个标志结点
        }
        protected DoubleLinkedNode<T> Head {get { return head; } set {head = value; } }
    }
```

用 DoubleLinkedList<T>类构造的一个实例即可用来表示一条双向链表对象，它的缺省构造方法建立一条仅有头结点的空链表。双向链表比单向链表在结点结构上增加了一个链，但给链表的操作带来很大的便利，能够沿着不同的链向两个方向移动，从而既可以找到后继结点，也可以找到前驱结点。

设 p 指向双向链表中的某一数据结点（尾结点除外），则双向链表具有下列本质特征：

(p.Prior).Next 等于 p

(p.Next).Prior 等于 p

而当 p 指向双向链表的最后一个结点时，由于线性表的最后一个数据元素没有后继数据元素，所以有 p.Next 等于 null。

双向链表的头结点的前向链总为空，即有 head.prior 等于 null。

3. 双向链表的操作

双向链表的操作分别用 DoubleLinkedList 类的不同方法成员来实现，部分操作代码如下。

1）判断双向链表是否为空。算法如下：

```
    public virtual bool Empty{  get{ return head.Next==null; } }
```

2）在双向链表中插入结点

在双向链表中插入新的结点，如果以索引参数 *i* 来指定结点的位置，则必须从表头顺着链找到相应的结点，再插入新的结点，过程如图 2.13 所示。

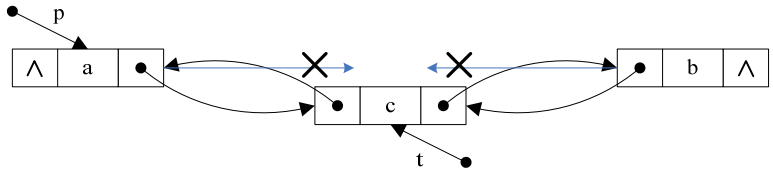


图 2.13 双向链表插入结点

生成值为 k 的新结点并做相应准备工作如下：

```
DoubleLinkedNode<T>  p, q, t = new DoubleLinkedNode<T> (k);
```

找到正确的插入位置后，设 p 指向链表中的某结点，在结点 p 之后插入结点 t，形成新的链表。语句如下：

```
t.prior = p;
t.next = p.next;
(p.next).prior = t;
p.next = t;
```

由此可见，在双向链表中插入结点，不需移动数据元素，只要修改相关的几条链，但比单向链表需要维护的工作多一些。完整的插入操作的实现代码如下：

```
public virtual void Insert(int i, T k) {
    int j = 0;
    DoubleLinkedNode<T> p = head;
    DoubleLinkedNode<T> q = head.Next;
    if (i < 0) i = 0;
    DoubleLinkedNode<T> t = new DoubleLinkedNode<T>(k);
    while (q != null) {
        if (j == i)break;
        p = q; q = q.Next;
        j++;
    }
    t.Next = p.Next; t.Prior = p;
    p.Next = t;
    if (q != null) q.Prior = t;
}
```

3）双向链表删除结点

在双向链表中删除给定位置的结点，需要把该结点从链表中退出，并改变相邻结点的链接关系。

删除结点的操作过程如图 2.14 所示，首先根据不同的要求定位将被删除的结点 q 和它的前驱结点 p，执行下列语句将结点 q 从链表中退出：

```
p.Next = q.Next;
(p.Next).Prior = p;
```

执行上述语句之后，则建立了新的链接关系，替代了原链接关系。因此，在双向链表中删除结点，只要修改相关的几条链，而不需移动数据。完整算法如下：

```
public void RemoveAt(int i) {
    int j = 0;
```

```
        DoubleLinkedNode<T> p = head;
        DoubleLinkedNode<T> q = head.Next;
        while (q != null) {
            if (j == i) {
                p.Next = q.Next;
                (p.Next).Prior = p;
                return;
            }
            p = q; q = q.Next;
            j++;
        }
        throw new IndexOutOfRangeException(
            "Index Out Of Range Exception in" + this.GetType());
    }
```

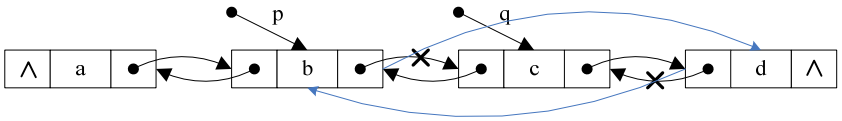


图 2.14　双向链表删除结点

读者不难参考前面的内容实现双向链表的其他操作。

4. 双向循环链表

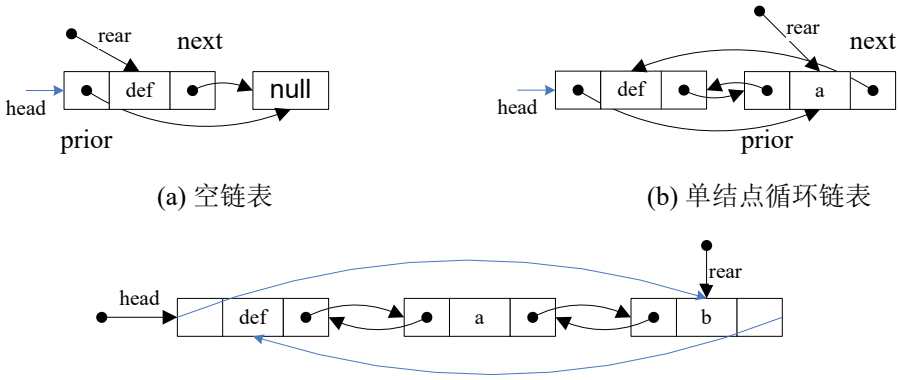
双向链表中，如果最后一个结点 rear 的 next 链指向链表的头结点（Head Node），而链表的头结点的 prior 链指向最后一个结点 rear，便形成双向循环链表（Circular Double Linked List），即在双向循环链表中有下列关系成立：

rear.next 等于 head

head.prior 等于 rear

当 Head.Next 等于 null 或 Head 等于 rear 时，循环链表为空。当 head.next 不等于 null 且 head.prior 等于 head.next 时，链表只有单个数据结点，如图 2.15 所示。

双向循环链表类的定义及其查找、插入和删除等操作的实现可以参照前面介绍的单向循环链表类和一般双向链表类的相关方法，此处从略。



(a) 空链表　　(b) 单结点循环链表

(c) 双向循环链表

图 2.15 双向循环链表

习题 2

2.1 编程实现下列操作。在单向链表中：

1) 构造单向链表，它复制另一个链表，构造方法声明为：
public SingleLinkedList(SingleLinkedList<T> a);
2) 返回第 i 个结点的值，方法声明为：public T GetNodeValue(int i);
3) 求各结点的数值之和，（对 int 型元素）方法声明为：public int Sum( );
4) 查找表中是否有值为 k 的节点，返回值为 bool 类型，方法声明为：public bool Contains(T k);
5) 删除链表中首个出现的值为 k 的节点，方法声明为：public void Remove(T k);。
6) 将两条单向链表连接起来，形成一条单向链表。方法声明为：
public void AddRange(SingleLinkedList<T> ll);

2.2 分别在 SequencedList 和 SingleLinkedList 类中编程实现（重写）基类 Object 中定义的虚方法“ToString()”的操作：public override string ToString();

2.3 编程实现下列操作。在双向线性链表中：

1) 构造双向链表，它复制另一个链表，构造方法声明为：
public DoubleLinkedList(DoubleLinkedList<T> a);
2) 删除值为 k 的结点，方法声明为：public void Remove(T k);。
3) 查找值为 k 的结点，方法声明为：public int IndexOf(T k);
4) 编程实现（重写）基类 Object 中定义的虚方法“ToString()”的操作：public override string ToString();
5) 实现插入排序。

2.4 编程实现一个不包含起标志作用的头结点的单向链表类。它的头结点是链表的第一个数据结点。提示：一些操作的实现需判断链表是否是单结点的情况。