第 3 章　栈与队列

教学要点

栈和队列是两种特殊的线性结构，栈具有后进先出的特性，而队列则具有先进先出的特性。这两种数据结构中的数据元素之间也都具有顺序的逻辑关系，它们与线性表的不同之处在于，线性表可以在表中任意的位置进行插入和删除操作，而栈和队列的插入和删除操作则限制在特殊的位置，栈的插入和删除操作只允许在数据结构的一端进行，队列的插入和删除操作则分别在队列结构的两端进行。在实现方式上，栈和队列都可以采用顺序存储结构和链式存储结构。

存在自调用的算法称为递归算法，递归是一种有效的算法设计方法，是解决许多复杂问题的重要方法。栈和队列数据结构、递归算法在实际问题中有着广泛的应用。

本章首先学习栈与队列的相关概念和抽象数据类型的定义，然后分析它们的不同存储结构的实现和应用举例；最后介绍递归的定义、递归的算法和数据结构设计举例。

本章在 Visual Studio 中用名为 stackqueue 的类库型项目实现有关数据结构的类型定义，用名为 stackqueuetest 的应用程序型项目实现相应类型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。

3.1　栈的概念及类型定义

3.1.1　栈的基本概念

栈（stack）是一种特殊的线性数据结构，栈结构中的数据元素之间具有顺序的逻辑关系，但栈只允许在数据集合的一端进行插入和删除数据元素的操作。向栈中插入数据元素的操作称为入栈（push），从栈中删除数据元素的操作称为出栈（pop）。每次删除的数据元素总是最后插入的那个数据元素，因此栈是一种“后进先出”（Last In First Out，LIFO）的线性结构。栈就像某种只有单个出入口的仓库，每次只允许一件件地往里面堆货物（入栈），然后一件件地往外取货物（出栈），不允许从中间放入或抽出货物。

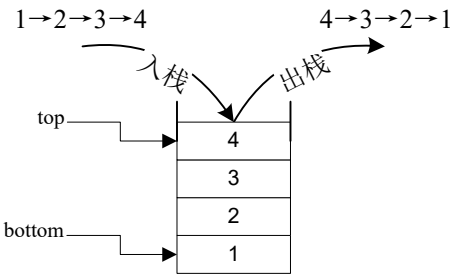


图 3.1　栈结构

栈结构中允许进行插入和删除操作的那一端称为栈顶（stack top），另一端则称为栈底（stack bottom）。栈顶的当前位置随着插入和删除操作的进行而动态地变化，标识栈顶当前位置的变量称为栈顶指针。栈结构及其操作如图 3.1 所示，图中，数据元素的入栈次序为 1→2→3→4，出栈次序为 4→3→2→1。对于一个的数据元素序列，通过控制其元素入栈和出栈时机，可以得到多种不同的出栈排列。

3.1.2 栈的抽象数据类型

1. 栈的数据元素

和线性表一样，栈也是由若干数据元素组成的有限数据序列。我们用抽象数据元素 a_i 表示栈的数据元素，对于由 n（$n \geqslant 0$）个数据元素 a_0，a_1，a_2，…，a_{n-1} 组成的栈结构可以记为：

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

其中，n 表示栈中数据元素的个数，称为栈的长度。若 n=0，则栈中没有元素，我们称之为空栈。栈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。

栈作为一种特殊的线性结构，可以如同线性表一样采用顺序存储结构和链式存储结构实现。顺序存储结构实现的栈称为顺序栈（Sequenced Stack），链式存储结构实现的栈称为链式栈（Linked Stack）。

2. 栈的基本操作

在一个栈数据结构上可以进行下列基本操作：

- Initialize: 栈的初始化。创建一个栈实例，并进行初始化操作，例如设置栈实例的状态为空。
- Count: 数据元素计数。返回栈中数据元素的个数。
- Empty: 判断栈的状态是否为空，即判断栈中是否已加入数据元素。
- Full: 判断栈的状态是否已满，即判断为栈预分配的空间是否已占满。
- Push: 入栈。该操作将一个数据元素插入栈中作为新的栈顶元素。在入栈操作之前必须判断栈的状态是否已满，如果栈不满，则直接接收新元素入栈；否则产生栈上溢错误（Stack Overflow Exception）；或者，为栈先重新分配更大的空间，然后接收新元素入栈。
- Pop: 出栈。该操作取出当前栈顶的数据元素，下一个数据元素成为新的栈顶元素。在出栈操作之前，必须判断栈的状态是否为空。如果栈的状态为空，则产生栈下溢错误（Stack Underflow Exception）。
- Peek: 探测栈顶。该操作获得栈顶数据元素，但不移除该数据元素，栈顶指针保持不变。

例如，对于数据序列{a，b，c }依次进行

$$\{ \text{Push(a)}, \text{Push(b)}, \text{Pop( )}, \text{Push(c)}, \text{Pop( )} \}$$

的操作，被实施该操作序列的栈实例的状态随着相应操作而进行的变化如图 3.2 所示。

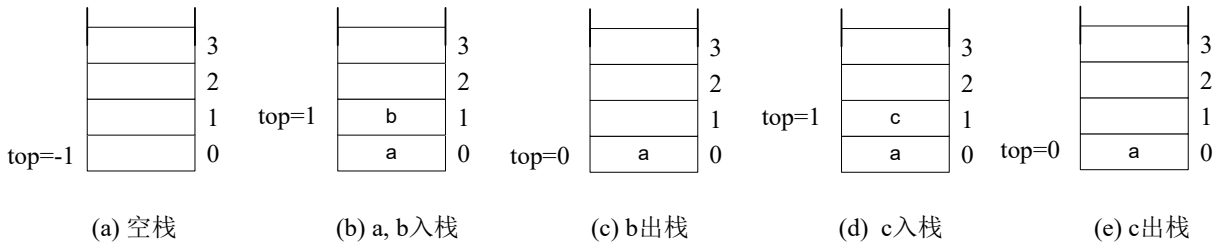


图 3.2 栈状态随插入和删除操作而进行的变化

3.1.3　C#中的栈类

在.NET Framework 的类库中定义了一个非泛型栈 Stack 类和一个泛型栈 Stack<T>类，是编程中常用的数据集合类型。

非泛型栈类 Stack 定义在 System . Collections 命名空间中，它刻画了一种数据后进先出的集合，其数据元素的类型是 object 类。入栈的数据元素类型定义为 object 类型，所以，在实际的入栈操作时可以是 int、string 等任意类型的对象，而出栈的数据元素类型也定义为 object 类型，一般需要用 object 对象保存，再转化为合适类型（如 string 或其他类型）的对象。

Stack<T>类是 Stack 类的泛型等效类，它定义在 System.Collections.Generic 命名空间中。Stack <T>类所具有的属性和方法非常类似于非泛型 Stack 类对应的属性和方法，差别在于前者是强类型栈，即元素的类型要与栈实例定义时声明的类型保持一致。泛型类在大多数情况下比非泛型类执行得更好并且是类型安全的。

Stack 类具有如下成员（属性和方法）实现栈的各种操作：

公共构造函数

```
Stack();                          //初始化 Stack 类的新实例
Stack( ICollection c);
Stack ( int capacity);
```

公共属性

```
virtual int Count {get;}          //获取包含在栈中的元素数
```

公共方法

```
virtual void Push( object x);     //将对象插入栈的顶部
virtual object Pop();             //移除并返回位于栈顶部的对象
virtual object Peek();            //返回位于栈顶部的对象，但不将其移除
virtual bool Contains(object x);  //确定某个元素是否在栈中
```

【例3.1】　创建 Stack 对象并向其添加元素，以及打印出其值。

```
using System;
using System.Collections;
namespace stackqueuetest{
    public class SamplesStack{
        public static void Main(){
            // Creates and initializes a new Stack.
            Stack myStack = new Stack();
            myStack.Push("Hello"); myStack.Push("World"); myStack.Push("!");
            // Displays the properties and values of the Stack.
            Console.Write( "myStack " );
            Console.WriteLine( "\tCount:    {0}", myStack.Count );
            Console.Write( "\tValues:\n" );
            foreach(object o in  myStack)
                Console.Write( "\t{0}", o);
        }
    }
```

```
}
```

程序运行结果如下：

```
myStack  Count:    3
        Values:    !    World    Hello
```

输出序列的顺序与元素入栈的顺序相反，这是由栈的先进后出（LIFO）特性形成的。

【例3.2】　利用栈进行数制转换。

数制转换是计算机实现计算的一个基本问题。十进制数 N 和其他 d 进制数的转换具有下列关系：

$$N = a_n \times d^n + a_{n-1} \times d^{n-1} + \ldots + a_1 \times d^1 + a_0$$

数制转换就是要确定序列{ a_0，a_1，a_2，…，a_n}，其解决方法很多，其中一个简单算法基于下列原理：

$$N = (N / d) \times d + N \% d$$

上式中“/”为整数的整除运算，“%”为求余运算。例如：$(2468)_{10} = (4644)_8$，其运算过程如下：

N	N / d	$N \% d$
2468	308	4
308	38	4
38	4	6
4	0	4

现要编写一个满足下列要求的程序：用户输入任意一个非负十进制整数，程序打印输出与其等值的八进制数。上述计算过程是从低位到高位顺序产生八进制数的各个数位，而打印输出一般来说要求符合人的读数习惯，应从高位到低位进行，这恰好与上述计算过程相反。因此若将计算过程中得到的八进制数的各位顺序进栈，则等完成计算后再依次出栈，并按出栈顺序打印输出的结果即为对应的八进制数，栈能很好地匹配这一过程。程序如下：

```
using System;
using System.Collections.Generic;
namespace stackqueuetest {
    public class DecOctConversion {
        public static void Main(string[] args){
            int n = 2468;
            if(args.Length>0){
                n = int.Parse(args[0]);
            }
            Stack<int> s = new Stack<int>(20);
            Console.Write("十进制数: {0} -> 八进制:",n);
            while (n!=0) {
                s.Push(n%8);
                n = n/8;
            }
            int i = s.Count;
            while (i>0){
                Console.Write(s.Pop());
                i--;
```

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

在上述程序中，我们使用泛型栈类 Stack<T>定义了一个整型数组成的栈 Stack<int>对象 s，Push 操作要求的参数为整型，Pop 操作返回的类型也是整型。如果换用非泛型类 Stack，它的 Push 和 Pop 操作要求的是 object 类型参数，而实参是整型变量，所以入栈时要先将整型实参装箱为 object 类型，而出栈时要将 object 类型拆箱为整型。如果频繁地进行装箱和拆箱操作，执行效率会受到很大的影响。这体现了泛型类在大多数情况下比非泛型类执行得更好并且是类型安全的。

3.2 栈的存储结构及实现

栈既可以采用顺序存储结构实现，也可以用链式存储结构实现。顺序存储结构实现的栈称为顺序栈（Sequenced Stack），链式存储结构实现的栈称为链式栈（Linked Stack）。

3.2.1 栈的顺序存储结构及操作实现

顺序栈用一组连续的存储空间存放栈的数据元素。可以用下面声明的 SequencedStack 类来实现顺序栈。

```
public class SequencedStack<T> {
    private T[] items;
    private const int empty = -1;
    private int top = empty;                    //top为栈顶元素下标
    ......
}
```

SequencedStack 类中的成员变量 items 定义为数组类型，即准备用数组存储栈的数据元素。成员变量 top 指示当前栈顶数据元素在数组 items 中的下标，起着栈顶指针的作用。定义好完整的 SequencedStack 类后，用该类构造的对象就是一个个具体的栈实例。

SequencedStack 类的源代码定义在 SequencedStack.cs 文件中，本章将 SequencedStack 类与其他章节介绍的实现相关数据结构的基础类一样，都定义在 DSAGL 命名空间。

顺序栈的操作作为 SequencedStack 类的属性和方法成员予以实现，下面分别描述实现这些操作的算法。

1) 栈的初始化

用类的构造方法初始化一个栈对象，在构造方法中为 items 数组变量申请指定大小的存储空间，以备用于存放栈的数据元素，通过使成员变量 top 值为 empty 来设置栈初始状态为空。多种形式的构造方法编码如下：

```
    //带参数时，构造具有n个存储单元的空栈
    public SequencedStack(int n) {
        items = new T[n];
        top = empty;                                //设置栈初始状态为空
```

```
}
//缺省构造方法。不带参数时，构造具有16个存储单元的空栈
public SequencedStack() : this(16) { }
```

2) 返回栈中元素的个数

该操作告知栈中已有的数据元素的个数，将这个操作通过定义属性 Count 来实现，编码如下：

```
public int Count {
    get { return top + 1; }
}
```

3) 判断栈是否为空和是否为满

将这两个测试操作分别通过定义相应的属性（Empty/Full）来实现。当栈顶指针 top 等于 empty 时，表明栈为空状态，Empty 属性应该指示 true。当栈顶指针 top 已指向数组当前预分配存储空间中的最后一个单元时，表明栈为满状态，Full 属性应该指示 true。

定义公有属性 Empty 和 Full 来实现栈是否为空和是否为满的判断，实现编码如下：

```
public bool Empty{  get{ return top==empty;}    }
public bool Full{  get{ return top>=items.Length-1;} }
```

4) 入栈

定义 Push 方法实现入栈操作。该操作将数据元素插入栈中作为新的栈顶元素。当栈的当前预分配存储空间尚未满时，移动栈顶指针，这里是将栈顶数据元素下标变量 top 加 1，将新数据 k 放入 top 位置，作为新的栈顶数据元素。

Push 方法的形参 k 声明为 T 类型，即入栈的数据元素声明为 T 类型。在调用该操作时，实参的类型要与栈实例定义时声明的类型保持一致。例如：定义 s 为 SequencedStack<string>类型，则以后入栈语句 s.Push(k)中的实参 k 必须为 string 类型。

当栈实例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 DoubleCapacity 方法重新分配存储空间，并将原数组中的数据元素逐个拷贝到新数组。

```
public void Push(T k) {
    if (Full) DoubleCapacity();
    top++;
    items[top] = k;
}
//扩充顺序栈的容量
private void DoubleCapacity() {
    int count = Count;
    int capacity = 2 * items.Length;
    T[] copy = new T[capacity];          // 按照新容量构造一个数组
    for (int i = 0; i < count; i++)
        copy[i] = items[i];
    items = copy;                        // items 指向新分配的空间
}
```

可见，如果为栈预分配的空间合理，栈处于非满状态，Push 操作的时间复杂度为 O(1)。如果经常需要增加存储容量以容纳新元素，则 Push 操作的时间复杂度成为 O(n)。

5) 出栈

定义 Pop 方法实现出栈操作。该操作取出当前栈顶数据元素，并将下一个数据元素设为新的栈顶元素。需要先判断栈是否为空，当栈不空时，取走变量 top 指示的位置处的数据元素，变量 top 自减 1，下一位置上的数据元素成为新的栈顶数据元素。Pop 方法的返回值声明为类型 T，即出栈的数据元素具有类型 T，在调用该操作时，将与栈实例定义时声明的类型保持一致。例如：定义 s 为 SequencedStack<string>类型，则以后出栈语句 s.Pop()得到的结果是 string 类型。此方法的运算复杂度是 O(1)。编码如下：

```
public T Pop() {
    T k = default(T);
    if (!Empty) {                    //栈不空
        k = items[top];              //取得栈顶元素
        top--;
        return k;
    }
    else                             //栈空时产生异常
        throw new InvalidOperationException("Stack is Empty: " + this.GetType());
}
```

6) 获得栈顶数据元素的值

定义 Peek 方法实现探测栈顶元素值的操作。该操作获得栈顶数据元素，但不移除该数据元素，栈顶指针 top 不变。实现上，当栈非空时，获得变量 top 指示的位置处的数据元素，此时该数据元素不出栈，top 的值保持不变。此方法的运算复杂度是 O(1)。

```
public T Peek() {
    if (!Empty)
        return items[top];
    else
        throw new InvalidOperationException("Stack is Empty: " + this.GetType());
}
```

7) 显示栈中所有数据元素的值

当栈非空时，从栈顶结点开始，直至栈底结点，依次显示各结点的值。

```
public void Show(bool showTypeName) {
    if (showTypeName)
        Console.Write("SequencedStack: ");
    if (!Empty) {
        for (int i = this.top; i >= 0; i--) {
            Console.Write(items[i] + "  ");
        }
        Console.WriteLine();
    }
}
```

【例3.3】　使用顺序栈的基本操作，测试 SequencedStack 类。

源程序 SequencedStackTest.cs 自身处于 stackqueuetest 命名空间，它使用 DSAGL 命名空间中定义的 SequencedStack 类，程序如下：

```
using System;
using DSAGL;
namespace stackqueuetest {
    class SequencedStackTest {
        public static void Main(string[] args) {
            int i = 0;
            SequencedStack<string> s1 = new SequencedStack<string>(20);
            Console.Write("Push: ");
            while (i < args.Length) {
                s1.Push(args[i]);                    //将命令行参数依次入栈
                Console.Write(args[i] + "  ");
                i++;
            }
            s1.Show(true);                           //输出栈中各元素值
            Console.Write("Pop : ");
            string str;
            while (!s1.Empty) {                      //全部出栈
                str = s1.Pop();
                Console.Write(str + "  ");
            }
            Console.WriteLine("栈中元素个数={0}", s1.Count);

             int m = 1357;
             SequencedStack<int>  s = new SequencedStack<int>(20);
             Console.Write("十进制数: {0} -> 八进制:", m);
             while (m != 0) {
                s.Push(m % 8);
                m = m / 8;
             }
             int j = s.Count;
             while (j > 0) {
                 Console.Write(s.Pop());
                 j--;
             }
             Console.WriteLine();
        }
    }
}
```

编译源文件 SequencedStackTest.cs，注意要引用（用命令行参数/r 指示）stackqueue 类库模块：

```
csc SequencedStackTest.cs /r: ..\stackqueue\bin\Debug\stackqueue.dll
```

从命令行输入参数运行 SequencedStackTest 程序：

```
SequencedStackTest a b c
```

运行结果如下：

```
Push: a  b  c  Stack: c  b  a
Pop : c  b  a  栈中元素个数=0
十进制数: 1357 -> 八进制:2515
```

3.2.2 栈的链式存储结构及操作实现

作为一种特殊的线性结构，栈结构如同线性表结构一样也可以采用链式存储结构来实现，用链式存储结构实现的栈称为链式栈（Linked Stack）。链式栈可以看成一种特殊的单向链表，因此可以从单向链表类导出链式栈类，这样就可以复用前一章的部分设计成果。链式栈具有一个仅作为标志的头结点，它的链域指向第 1 个数据结点，这个结点就是栈顶数据结点，设置变量 top 指向该结点，入栈和出栈操作都是针对栈顶指针 top 所指向的结点进行的。栈的链式存储结构如图 3.3 所示。

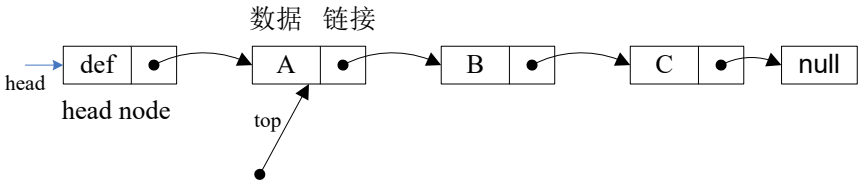


图 3.3 栈的链式存储结构

以下声明的LinkedStack类实现栈的链式存储。

```
//将要用上第二章定义的链表类SingleLinkedList和结点类SingleLinkedNode
namespace stackqueue {
    public class LinkedStack<T> : SingleLinkedList<T> {
        private SingleLinkedNode<T> top;
        ......
    }
}
```

在 Visual Studio 开发环境中对当前项目（stackqueue）要加上对 lists 项目（第二章中开发的类库项目）的引用。

成员变量 top 指向栈顶数据元素结点，结点类型为具有单链的结点类 SingleLinkedNode，结点数据域的类型为泛型 T（参见第二章线性表）。定义了 LinkedStack 类后，就可以用它来定义一个个具体的栈对象。

链式栈的操作作为 LinkedStack 类的属性和方法成员予以实现，下面分别描述实现这些操作的算法。

1) 栈的初始化

用构造方法创建栈对象并对它进行初始化，它首先用基类（SingleLinkedList 类）的构造方法创建一条单向链表，接着将成员变量 top 指向第一个数据结点，设置栈的状态初始为空。

```
public LinkedStack(): base() {
    top = base.Head.Next;      //Head是仅作为标志的头结点，top指向第一个数据结点
}
```

2) 返回栈的元素个数

由基类 SingleLinkedList 继承的公共属性 Count 即可完成返回栈的元素个数的功能，导出类 LinkedStack 继承了这个功能，无需再写实现这个功能的代码。

3) 判断栈状态是否为空或是否已满

用布尔类型的属性 Empty 来实现判断栈是否为空的功能；当变量 top 等于 null 时，栈为空状态，属性 Empty 此时应该返回 true，否则栈为非空状态，Empty 此时应该返回 false。实现代码如下：

```
public override bool Empty{ get{ return  top==null; } }
```

链式栈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入栈时，向系统申请一个结点的存储空间，一般可在编程时认为系统所提供的可用空间是足够大的，因此不必判断栈是否已满。如果空间已用完，系统无法分配新的存储单元，则产生运行时异常。

比较功能 2 和功能 3，可以体会到面向对象程序设计带来的便利。派生类（如类 LinkedStack）既可以直接继承基类（如类 SingleLinkedList）的属性和方法（子类与父类有相同的行为），也可以重写基类的属性和方法（子类有与父类不同的行为，但行为的命名是相同的）。

4) 入栈

定义 Push 方法实现入栈操作。该操作将数据元素插入栈中作为新的栈顶元素。

为将插入的数据元素值 k 构造一个新结点 q，在 top 指向的栈顶结点之前插入结点 q 作为新的栈顶结点，并使 top 指向它。入栈的数据元素是 T 类型，在调用该操作时，实参的类型要与栈实例定义时声明的元素类型保持一致。采用动态分配方式为每个新结点分配内存空间，此方法的运算复杂度是 O(1)。

```
public void Push(T k) {
    SingleLinkedNode<T> q = new SingleLinkedNode<T>(k);
    q.Next = top;                          //q结点作为新的栈顶结点
    top = q;
    base.Head.Next = top;
}
```

5) 出栈

定义 Pop 方法实现出栈操作。该操作取出当前栈顶数据元素，并将下一个数据元素设为新的栈顶元素。

当栈不为空时，取走 top 指向的栈顶结点的值，并删除该结点，使 top 指向新的栈顶结点。出栈的数据元素具有类型 T，在调用该操作时，将与栈实例定义时声明的类型保持一致。此方法的运算复杂度是 O(1)。

```
public T Pop() {
    T k = default(T);                      //置变量k为T类型的缺省值
    if (!Empty) {                          //栈不空
        k = top.Item;                      //取得栈顶数据元素值
        top = top.Next;                    //删除栈顶结点
        base.Head.Next = top;
        return k;
```

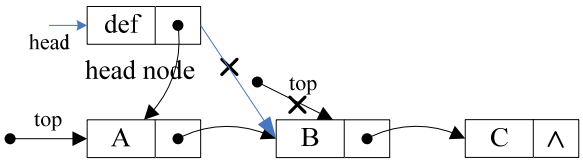
```
        }
        else                                    //栈空时产生异常
            throw new InvalidOperationException("Stack is Empty: " + this.GetType());
    }
```

6) 获得栈顶数据元素值

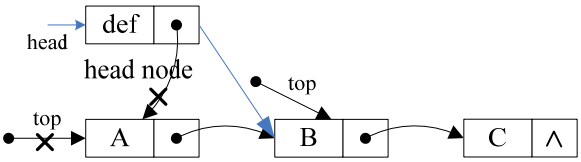
该操作获得栈顶数据元素，但不移除该数据元素，栈顶指针不变。当栈非空时，获得 top 位置处的数据元素，此时该数据元素不出栈，top 变量保持不变。此方法的运算复杂度是 O(1)。

```
    public T Peek() {
        if (!Empty)
            return top.Item;
        else                                    //栈空时产生异常
            throw new InvalidOperationException("Stack is Empty: " + this.GetType());
    }
```

链式栈的基本操作如图 3.4 所示。



(a) 元素入栈



(b) 元素出栈

图 3.4　链式栈的基本操作

由以上多个操作的算法实现分析可知，顺序栈 SequencedStack 和链式栈 LinkedStack，都实现了栈 Stack 这个抽象数据结构的基本操作。无论是 SequencedStack 类还是 LinkedStack 类，都可以用来建立具体的栈实例，通过栈实例调用入栈或出栈方法进行相应的操作。一般情况下，解决某个问题关注的是栈的抽象功能，而不必关注栈的存储结构及其实现细节。

3.2.3　栈的应用举例

栈是一种具有“后进先出”特性的特殊线性结构，适合作为求解具有后进先出特性问题的数学模型，因此栈成为解决相应问题算法设计的有力工具。

1. 基于栈结构的函数嵌套调用

程序中函数的嵌套调用是指在程序运行时，一个函数的执行语句序列存在对另一个函数的调用，每个函数在执行完后再返回到调用它的函数中继续执行，对于多层嵌套调用来说，函数返回的次序与

函数调用的次序正好相反，整个过程具有后进先出的特性，系统通过建立一个栈结构用以协助实现这种函数嵌套调用机制。

例如，执行函数 A 时，A 中的某语句又调用函数 B，系统要做一系列的入栈操作：

1） 将调用语句后的下一条语句作为返回地址信息保存在栈中，该过程称为保护现场；

2） 将 A 调用函数 B 的实参保存在栈中，该过程称为实参压栈；

3） 控制交给函数 B，在栈中分配函数 B 的局部变量，然后开始执行函数 B 内的其他语句。

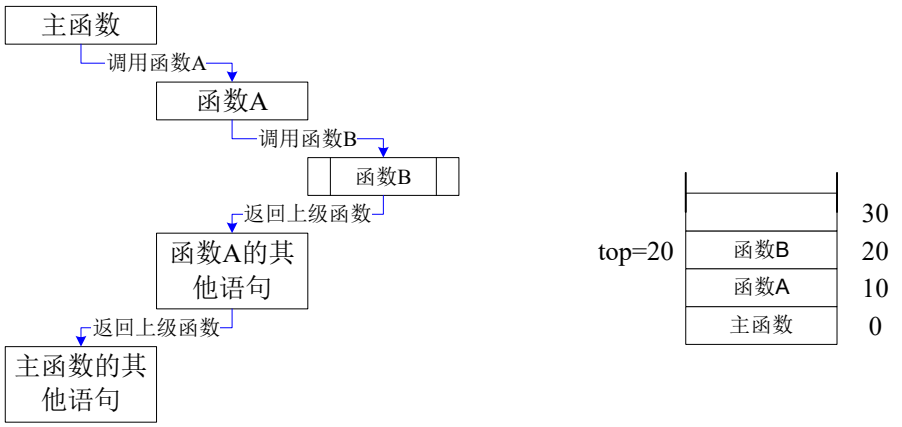
函数 B 执行完成时，系统则要做一系列的出栈操作才能保证将系统控制返回到调用 B 的函数 A 中：

1） 退回栈中为函数 B 的局部变量分配的空间；

2） 退回栈中为函数 B 的参数分配的空间；

3） 取出保存在栈中的返回地址信息，该过程称为恢复现场，程序继续运行函数 A 的其他语句。

函数嵌套调用时系统栈的变化如图 3.5 所示，函数调用的次序与返回的次序正好相反。可见，系统栈结构是实现函数嵌套调用或递归调用的基础。



(a) 函数的嵌套调用与返回　　(b) 函数调用时的系统栈

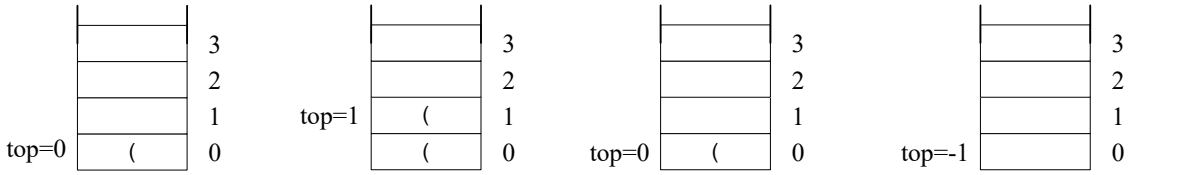
图 3.5 函数嵌套调用时的系统栈

2. 几个应用栈结构的典型例子

【例3.4】　判断 C#表达式中括号是否匹配。

在高级编程语言的表达式中，括号一般都是要求左右匹配的，对于一个给定的表达式，可使用栈来辅助判断其中括号是否匹配。

假设在 C#语言的表达式中，只能出现圆括号用以改变运算次序，而且圆括号是左右匹配的。本例声明 MatchExpBracket 类，对于一个字符串 expstr，方法 MatchingBracket(expstr)判断字符串 expstr 中的括号是否匹配。例如，当字符串 expstr 保存表达式“((9-1)*(3+4))”时，MatchingBracket()方法的算法描述如图 3.6 所示。



(a) 第一个“(”入栈　(b) 第二个“(”入栈　(c) 遇到“)”时，出栈一个“(”：一对括号匹配　(d) 表达式检测完，栈为空：括号全部匹配

图 3.6 表达式括号匹配过程中栈状态的变化

方法 MatchingBracket( )的实现算法描述如下：

(1) 设 NextToken 是待检测字符串 expstr 的当前字符，s1 是算法设置的一个栈：

- 若 NextToken 是左括号，则 NextToken 入栈
- 若 NextToken 是右括号，则从 s1 中出栈一个符号。若该符号为左括号，表示括号匹配。若出栈值为空或不为左括号时，则表示缺少左括号，期望左括号。

(2) 重复上一步。当对表达式串 expstr 检测结束后，若栈为空，表示括号匹配，否则表示缺少右括号，期望右括号。

程序中使用本章已声明的顺序栈 SequencedStack 类，数据元素的类型在声明栈对象时确定为 char 类型，因此入栈和出栈的数据元素都是 char 类型。程序如下：

```
using System;
using DSAGL;
namespace stackqueuetest {
   public class MatchExpBracket {
        public static void Main(string[] args){
             string expstr1 = "((9-1)*(3+4)";
             Console.WriteLine(expstr1);
            Console.WriteLine("Matching Bracket: " + MatchingBracket(expstr1));
        }
        public static string MatchingBracket(string expstr){
            SequencedStack<char> s1 = new SequencedStack<char>(30);  //创建空栈
             char NextToken, OutToken;
            int i=0; bool LlrR=true;
             while(LlrR && i<expstr.Length){
                  NextToken = expstr[i];
                  i++;
                  switch(NextToken){
                       case '(':                          //遇见左括号时,入栈
                            s1.Push(NextToken);
                            break;
                       case ')':                          //遇见右括号时,出栈
                         if (s1.Empty) {
                             LlrR = false;
                         }
                         else {
```

```
                    OutToken = s1.Pop();
                    if (!OutToken.Equals('('))
                        LlrR = false;           //判断出栈的是否为左括号
                }
                break;
            }
        }
        if(LlrR)
            if(s1.Empty)
                return "OK!";
            else
                return "期望)!";
        else
            return "期望(!";
    }
}
```

程序运行结果如下：

```
((9-1)*(3+4)
 Matching Bracket: 期望)!
```

【例3.5】　使用栈计算表达式的值。

程序在运行时，经常要计算算术表达式的值，例如，

$$10 + 20 * (30-40) + 50 \qquad (3\text{-}1)$$

我们在源程序中所写的表达式一般将运算符写在两个操作数中间，这种形式的表达式称为中缀表达式。表达式中的运算符具有不同的优先级，当前扫描到的运算符不能立即参与运算，这使得运算规律较复杂，求值过程不能从左到右顺序进行。

还可以有其他形式的表达式，例如，后缀表达式，它将运算符写在两个操作数之后。例如，式（3-1）可以转化为如下的后缀表达式：

$$10 \quad 20 \quad 30 \quad 40 \quad - \quad * \quad + \quad 50 \quad + \qquad (3\text{-}2)$$

后缀表达式中的运算符没有优先级，而且后缀表达式不需括号。后缀表达式的求值过程能够严格地从左到右顺序进行，符合运算器的求值规律。从左到右按顺序进行运算，遇到某个运算符时，则对它前面的两个操作数求值，过程如图 3.7 所示。

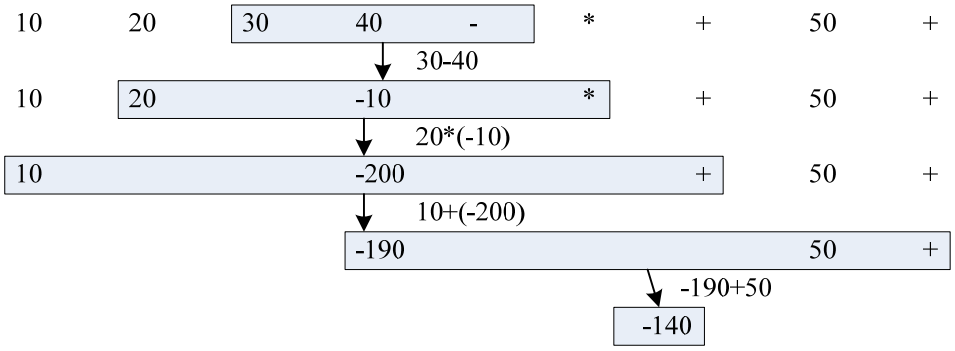


图 3.7 后缀表达式求值过程

为简化问题，本例对整型表达式求值，输入字符串类型的合法的中缀表达式，表达式由双目运算符+、-、*和圆括号“(”、“)”组成。表达式求值的算法分为两步进行：首先将中缀表达式转换为后缀表达式，再求后缀表达式的值。

1) 将中缀表达式转换为后缀表达式

对于字符串形式的合法的中缀表达式，“(”的运算优先级最高，“*”次之，“+”、“-”最低，同级运算符从左到右按顺序运算。

中缀表达式中，当前看到的运算符不能立即参与运算。例如式（3-1）中，第 1 个出现的运算符是“+”，此时另一个操作数没有出现，而后出现的“*”运算符的优先级较高，应该先运算，所以不能进行“+”运算，必须将“+”运算符保存起来。式（3-1）中“+”、“*”的出现次序与实际运算次序正好相反，因此将中缀表达式转换为后缀表达式时，运算符的次序可能改变，必须设立一个栈来存放运算符。转化过程的算法描述如下：

- 从左到右对中缀表达式进行扫描，每次处理一个字符。
- 若遇到左括号“(”，入栈。
- 若遇到数字，原样输出。
- 若遇到运算符，如果它的优先级比栈顶数据元素的优先级高，则入栈，否则栈顶数据元素出栈，直到新栈顶数据元素的优先级比它低，然后将它入栈。
- 若遇到右括号“)”，则运算符出栈，直到出栈的数据元素为左括号，表示左右括号相互抵销。
- 重复以上步骤，直至表达式结束。
- 若表达式已全部结束，将栈中数据元素全部出栈。

将中缀表达式（3-1）转换为后缀表达式（3-2）时，运算符栈状态的变化情况如图 3.8 所示。

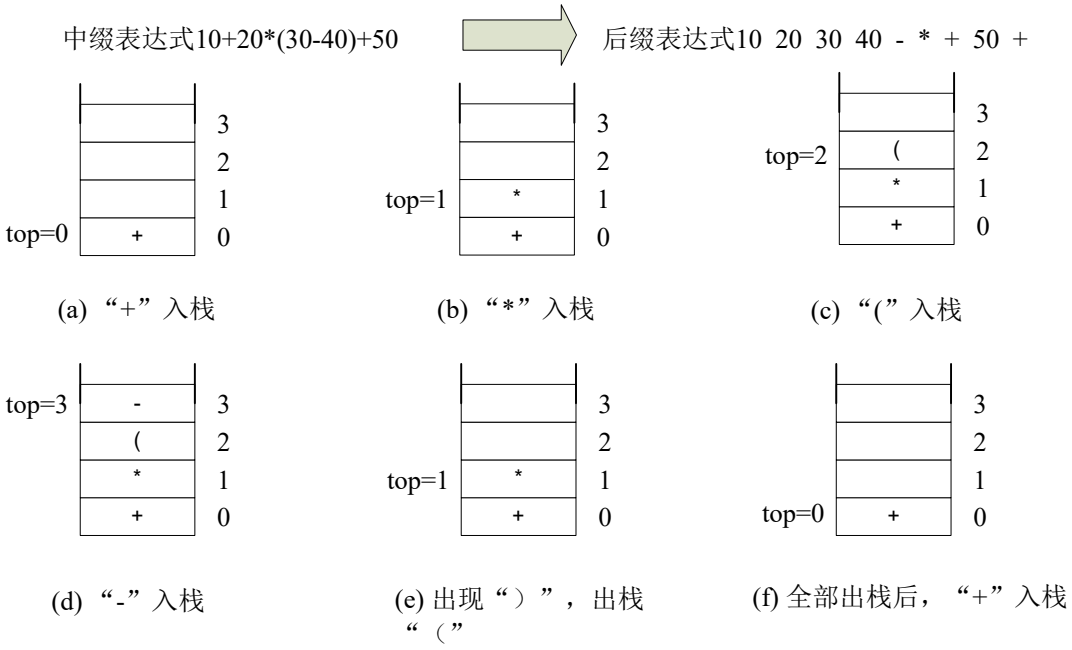


图 3.8 将中缀表达式变为后缀表达式时运算符栈状态的变化情况

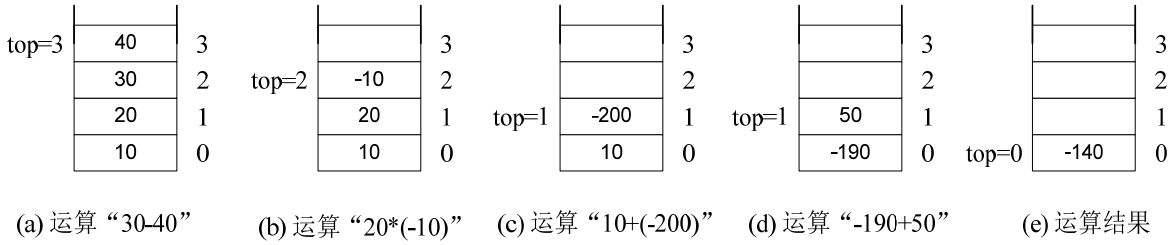
2）后缀表达式求值

由于后缀表达式没有括号，且运算符没有优先级，因此求值过程中，当运算符出现时，只要取得前两个操作数就可以立即进行运算。当两个操作数出现时，却不能立即求值，必须先保存等待运算符。所以，后缀表达式的求值过程中也必须设立一个栈，用于存放操作数。

后缀表达式求值算法描述如下：

- 从左到右对后缀表达式字符串进行扫描，每次处理一个字符。
- 若遇到数字，入栈。
- 若遇到运算符，出栈两个值进行运算，运算结果再入栈。
- 重复以上步骤，直至表达式结束，栈中最后一个数据元素是所求表达式的结果。

在后缀表达式（3-2）的求值过程中，操作数栈状态的变化情况如图 3.9 所示。



(a) 运算“30-40” (b) 运算“20*(-10)” (c) 运算“10+(-200)” (d) 运算“-190+50” (e) 运算结果

图 3.9 后缀表达式求值过程中数据栈状态的变化情况

本例声明 EvalExp 类对算术表达式求值。它的构造方法以字符串 str 构造表达式对象，expstr 和 pstr 分别表示表达式的中缀和后缀形式。

Transform( )方法将 expstr 中的中缀表达式转换为后缀表达式，保存在 pstr 中，转换时设立运算符栈 s1。s1 是类 ArrayStack 的对象，数据元素类型为 object。

Evaluate ( )方法对 pstr 中的后缀表达式求值，设立操作数栈 s2。s2 是类 LinkedStack 的对象，数据元素类型为 int。程序如下：

```
using System; using DSAGL;
namespace stackqueuetest {
   public class EvalExp {
        string expstr = "";                              //中缀表达式
        string pstr = "";                                //后缀表达式
        public EvalExp(string str){
             expstr = str;
        }
        public static void Main(string[] args){
             string str = "((1+2)*(4-3)-5+6)*8/2";
             EvalExp exp1 = new EvalExp(str);
             Console.WriteLine("Expression string: " + exp1.expstr);
             Console.WriteLine("Transformed string: "+ exp1.Transform());
             Console.WriteLine("Value: " + exp1.Evaluate());
        }
        public string Transform(){
            SequencedStack<string> s1 = new SequencedStack<string>(30);   //创建空栈
            char ch; string outstr; int i = 0;
             while(i<expstr.Length){
                  ch = expstr[i];
                  switch(ch){
                       case '+':                                 //遇到+、-时
                       case '-':
```

```
                while(!s1.Empty && !(s1.Peek()).Equals("(")){
                    outstr = s1.Pop();
                    pstr += outstr;
                }
                s1.Push(ch.ToString());
                i++;
                break;
            case '*':                              //遇到*、/时
            case '/':
                while( !s1.Empty && ((s1.Peek()).Equals("*") ||
                     (s1.Peek()).Equals("/")) ){
                    outstr = s1.Pop();
                    pstr += outstr;
                }
                s1.Push(ch.ToString());
                i++;
                break;
            case '(':
                s1.Push(ch.ToString());          //遇到左括号时，入栈
                i++;
                break;
            case ')':
                outstr = s1.Pop();               //遇到右括号时，出栈
                while (!s1.Empty && (outstr == null || !outstr.Equals("("))) {
                    pstr += outstr;
                    outstr = s1.Pop();
                }
                i++;
                break;
            default:
                while(ch>='0' && ch<='9'){  //遇到数字时
                    pstr += ch;
                    i++;
                    if(i<expstr.Length)
                        ch=expstr[i];
                    else
                        ch = '=';
                }
                pstr += " ";
                break;
        }
    }
    while (!s1.Empty) {
```

```
                outstr = s1.Pop();
                pstr = pstr + outstr;
            }
            return pstr;
        }
        public int Evaluate() {
            LinkedStack<int> s2 = new LinkedStack<int>();     //创建空栈
            char ch;
            int i=0,x,y,z=0;
            while(i<pstr.Length) {
                ch=pstr[i];
                if(ch>='0' && ch<='9') {
                    z=0;
                    while(ch!=' ') {
                        z = z*10+Int32.Parse(ch+"");
                        i++;
                        ch = pstr[i];
                    }
                    i++;
                    s2.Push(z);
                } else {
                    y=s2.Pop();
                    x=s2.Pop();
                    switch(ch) {
                        case '+': z=x+y; break;
                        case '-': z=x-y; break;
                        case '*': z=x*y; break;
                        case '/': z=x/y; break;
                    }
                    s2.Push(z);
                    i++;
                }
            }
            return  s2.Pop();
        }
    }
}
```

程序运行结果如下：

```
Expression string: ((1+2)*(4-3)-5+6)*8/2
Transformed string: 1 2 +4 3 -*5 -6 +8 *2 /
Value: 16
```

3.3　队列的概念及类型定义

3.3.1　队列的基本概念

与栈一样，队列（Queue）也是一种常用的线性数据结构，它的数据元素之间具有顺序的逻辑关系。与线性表可在任意位置进行插入和删除数据元素的操作不同，队列上插入和删除数据元素的操作分别限定在队列结构的两端进行。新的元素只能在队尾插入，而当前能从队列中删除的元素一定是最先插入队列的数据元素，因此队列是一种具有“先进先出”（First In First Out，FIFO）特性的线性数据结构，就像日常生活中常见的排队等待某种服务一样，先到先服务，后到排队尾。在算法设计中，当求解具有先进先出特性的问题时，需要用到队列这种数据结构。例如在计算机系统中，如果多个进程需要使用某个资源，它们就要排队等待该资源的就绪。

向队列中插入元素的操作称为入队（Enqueue），删除元素的操作称为出队（Dequeue）。允许入队的一端为队尾（Rear），允许出队的一端为队头（Front）。标识队头和队尾当前位置的变量分别称为队头指针和队尾指针。没有数据元素的队列称作空队列。

队列结构如图 3.10 所示。设有数据元素 a_0，a_1，a_2，…，a_{n-1} 依次入队，则出队次序为：$a_0 \rightarrow a_1 \rightarrow \cdots \rightarrow a_{n-1}$。

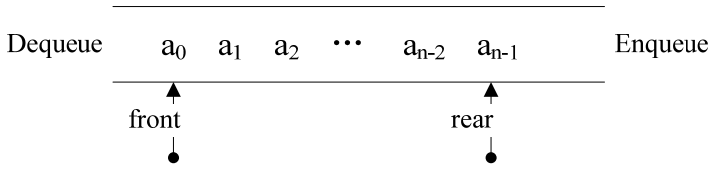


图 3.10 具有 n 个元素的队列

3.3.2　队列的抽象数据类型

1. 队列的数据元素

和线性表一样，队列也是由若干数据元素组成的有限数据序列。我们用抽象数据元素 a_i 表示队列的数据元素，对于由 n（$n \geqslant 0$）个数据元素 a_0，a_1，a_2，…，a_{n-1} 组成的队列结构可以记为：

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

其中，n 表示队列中的数据元素个数，称为队列的长度。若 n 等于 0，则队列中没有元素，称之为空队列。队列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。

队列作为一种特殊的线性结构，可以如同线性表以及栈一样采用顺序存储结构和链式存储结构实现。顺序存储结构实现的队列称为顺序队列（Sequenced Queue），链式存储结构实现的队列称为链式队列（Linked Queue）。

2. 队列的基本操作

在一个队列数据结构上可以进行下列基本操作：

- Initialize: 队列的初始化。创建一个队列实例，并进行初始化操作，例如设置队列状态为空。

- Count: 队列元素计数。返回队列中数据元素的个数。
- Empty: 判断队列的状态是否为空，即判断队列中是否已加入数据元素。
- Full: 判断队列的状态是否已满，即判断为队列预分配的空间是否已占满。
- Enqueue: 入队。该操作将新的数据元素从队尾处加入队列，该元素成为新的队尾元素。在入队之前必须判断队列的状态是否已满，如果队列不满，则接收新数据元素入队；否则产生队列上溢错误（Queue Overflow Exception），或者为队列先分配更大的空间，然后接收新元素入队。
- Dequeue: 出队。该操作取出队头处的数据元素，下一个数据元素成为新的队头元素。在出队之前，必须判断队列的状态是否为空。队列为空则产生下溢错误（Queue Underflow Exception）。
- Peek: 探测队首。获得队首数据元素，但不移除该元素，队头指针保持不变。

3.3.3 C#中的队列类

.NET Framework 的类库中定义了一个非泛型队列 Queue 类和一个泛型队列 Queue<T>类，是编程中常用的数据集合类型。

非泛型队列类 Queue 定义在 System.Collections 命名空间中。它刻画了一种具有先进先出特性的数据集合，其数据元素的类型是 object 类。入队的数据元素类型定义为 object 对象，所以，在执行实际的入队操作时可以是 int、string 等任意类型的对象，而出队的数据元素类型也定义为 object 类型，一般需要用 object 对象保存，再转化为合适的类型（如 string 或其他类型）的对象。

Queue<T>类是 Queue 类的泛型等效类，它定义在 System.Collections.Generic 命名空间中。Queue<T>类所具有的属性和方法非常类似于非泛型 Queue 类对应的属性和方法，差别在于前者是强类型队列，元素的类型要与队列实例定义时声明的类型保持一致。泛型类在大多数情况下比非泛型类执行得更好并且是类型安全的。

Queue 类具有如下成员（属性和方法）实现队列的各种操作：

公共构造函数

```
Queue ();                                  //初始化 Queue 类的新实例
Queue ( ICollection col);
Queue ( int capacity);
```

公共属性

```
virtual int Count {get;}                   //获取包含在 Queue 中的元素数
```

公共方法

```
virtual void Enqueue(object obj);          //将对象添加到 Queue 的结尾
virtual object Dequeue ();                 //移除并返回位于 Queue 开始处的对象
virtual object Peek();                     //返回队头处的对象但不将其移除。
virtual bool Contains(object obj);         //确定某个元素是否在队列中
```

【例3.6】 创建 Queue 对象并向其添加值，以及打印出其值。

```
using System;
using System.Collections;
namespace stackqueuetest {
public class SamplesQueue  {
    public static void Main()  {
        // Creates and initializes a new Queue.
```

```
            Queue myQ = new Queue();
            myQ.Enqueue("Hello"); myQ.Enqueue("World"); myQ.Enqueue("!");
            // Displays the properties and values of the Queue.
            Console.WriteLine( "myQ" );
            Console.WriteLine( "\tCount:    {0}", myQ.Count );
            Console.Write( "\tValues: " );
            foreach(object o in  myQ){
                Console.Write( "{0}\t", o);
            }
            Console.WriteLine();
        }
    } // end of class SamplesQueue
    } // end of namespace stackqueuetest
```

程序运行结果如下：

```
myQ
        Count:    3
        Values: Hello   World   !
```

输出序列的顺序与元素入队的顺序一致，这是队列的先进先出（FIFO）特性造成的。

3.4 队列的存储结构及实现

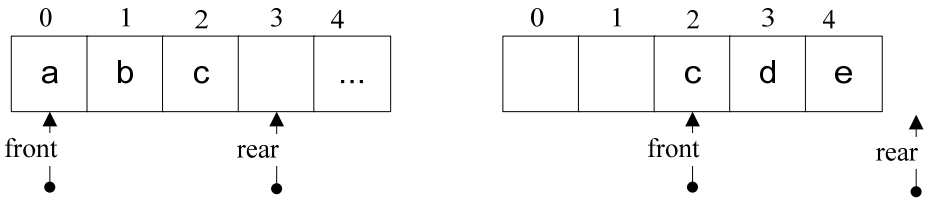
队列既可以采用顺序存储结构实现，也可以用链式存储结构实现。用顺序存储结构实现的队列称为顺序队列（Sequenced Queue），用链式存储结构实现的队列称为链式队列（Linked Queue）。

3.4.1 队列的顺序存储结构及操作实现

1. 队列的顺序存储结构

顺序队列用一组连续的存储空间存放队列的数据元素，如图 3.11 所示。可以用下面声明的 SequencedQueue 类来实现顺序队列。SequencedQueue 类中的成员变量 items 定义为数组，用以存储队列的数据元素；成员变量 front 和 rear 分别作为队头数据元素在数组 items 中的位置下标和下一个将入队的数据元素将占据的存储单元的位置下标，构成队头指针和队尾指针。SequencedQueue 类设计完整后，用该类定义和构造的对象就是一个个具体的队列实例。

```
public class SequencedQueue<T> {
    private T[] items;
    private int front, rear;                    //front和rear为队列头尾下标
    ......
}
```



(a) 队列的顺序存储结构　　(b) 顺序队列的“假溢出”

图 3.11 队列的顺序存储结构

front 是处于队头的数据元素的下标，简称队头下标；rear 不是当前队尾元素的下标，而是下一个入队的数据元素将占据的存储单元的位置下标，但我们仍将 rear 简称为队尾下标。元素入队或出队时，需要相应修改 front 或 rear 变量的值：一个元素入队时 rear 加 1，而一个元素出队时 front 加 1。

假设先有 3 个数据元素（a，b，c）已入队，那么 front=0，rear=3，如图 3.11（a）所示。接着有 2 个数据元素 a 和 b 出队，再接着又有 2 个数据元素（d 和 e）入队，那么 front=2，rear=5，如图 3.11（b）所示。设数组 items 的长度 Length 等于 5，此时如果有新的数据元素 f 要入队，应存放于 rear 指示的地方，注意此时 rear=5，数组下标越界而引起溢出。但此时并非所有预分配的存储空间被占满，数组的头部已空出一些存储单元，因此，这是一种假溢出。

顺序队列中出现的有剩余存储空间但不能进行新的入队操作的溢出现象称为假溢出。可以看出，上面描述的顺序队列在多次入队和出队操作后，虽然可能会仍有剩余的存储空间，但没有实现重复使用存储单元的机制，因而产生假溢出这样的缺陷。解决假溢出问题的办法是将顺序队列设计成逻辑上的“环形”结构，看似一种顺序循环队列。

2. 顺序循环队列的定义及操作实现

所谓顺序循环队列，是通过“取模”操作，将为顺序队列所分配的连续存储空间，变成一个逻辑上首尾相连的“环形”队列。为实现顺序队列循环利用存储空间，进行入队和出队操作时，front 和 rear 不是简单加 1，而应该是加 1 后再作取模运算，即入队时队尾指针按照如下规律变化：

```
rear = (rear + 1) % items.Length;
```

而出队时队头指针按照以下规律变化：

```
front = (front + 1) % items.Length;
```

顺序循环队列中，front 定义为当前头数据元素的下标，rear 定义为当前队尾数据元素下一位置的下标，items.Length 表示数组 items 的长度，即为队列预分配存储空间的大小。当 rear 和 front 逐步移动达到 items.Length – 1 位置后，再前进一个位置就又回到 0 位置，因此 rear 和 front 通过取模操作将在数组占据的存储空间中循环移动，使得数组的剩余存储单元可以重复使用，因而不会出现假溢出问题。顺序循环队列如图 3.12 所示。

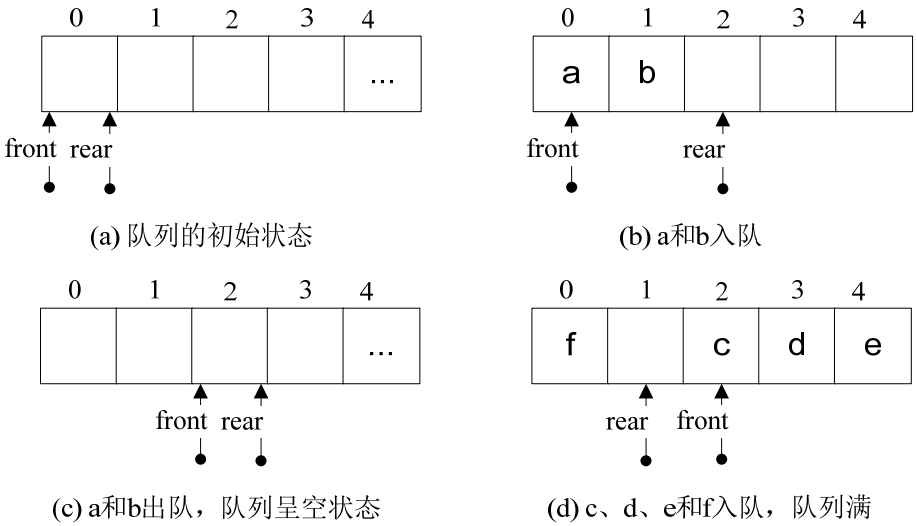


图 3.12　顺序循环队列

顺序循环队列的操作作为 SequencedQueue 类的方法和属性成员予以实现，下面分别描述实现这些操作的算法。

1) 队列的初始化

用类的构造方法初始化一个队列对象，在构造方法中首先为 items 数组变量申请指定大小的存储空间，以备用来存放队列的数据元素；接着，设置队列初始状态为空，即置 front = 0，rear = 0。

```
public SequencedQueue(int n) {
    items = new T[n + 1];
    front = rear = 0;
}
public SequencedQueue() : this(16) { }
```

2) 返回队列中元素的个数

该操作告知当前队列中已有的数据元素的个数，将这个操作用属性 Count 来实现，编码如下：

```
public int Count {
    get { return (rear - front + items.Length) % items.Length; }
}
```

3) 判断队列的状态是否为空和是否为满

将这两个测试操作分别用相应的属性（Empty/Full）来实现。

判断队列是否为空的操作即判断队列中是否有数据元素。当队列的队头指针与队尾指针相等时，即 front == rear 时，表明队列中没有数据元素，队列为空，Empty 属性应该指示 true。编码如下：

```
public bool Empty { get { return front == rear; } }
```

判断队列是否已满的操作即判断为队列预分配的空间是否已占满。当 front == (rear + 1) % items.Length，或者表示为(rear - front) % items.Length == 1 时，items 数组中虽然仍有一个空位置，但队列已不能新加入元素了，表明队列已满，此时 Full 属性应该指示 true。编码如下：

```
public bool Full {
    get {return front == (rear + 1) % items.Length; }
}
```

4) 入队

定义 Enqueue 方法实现入队操作。该操作将新的数据元素 k 从队尾处加入队列，该元素成为新的队尾元素。先测试队列是否已满，当队列不满时，将数据元素 k 存放在 rear 指示的位置，作为新的队尾数据元素；rear 循环加 1。

Enqueue 方法的形参 k 的类型声明为 T，即此时入队的数据元素声明为 T 类型，在调用入队操作时，实参的类型要与队列实例定义时声明的类型保持一致。例如：定义 q 为 SequencedQueue<string>类型，则以后入队语句 q. Enqueue (k)中的实参 k 必须为 string 类型。

如果队列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 DoubleCapacity 方法重新分配存储空间，将原数组中的数据元素逐个拷贝到新数组，并相应调整队首与队尾指针。编码如下：

```
public void Enqueue(T k) {
    if (Full) DoubleCapacity();
    items[rear] = k;
    rear = (rear + 1) % items.Length;
}
private void DoubleCapacity() {
    int i, j;
    int capacity = 2 * items.Length - 1;
    int count = Count;
    T[] copy = new T[capacity];  // 按照新容量构造一个数组
    for (i = 0; i < count; i++) {
        j = (i + front) % items.Length;
        copy[i] = items[j];
    }
    front = 0;
    rear = count;
    items = copy;                // items 指向新分配的空间
}
```

如果为队列预分配的空间合理，队列处于非满状态，入队操作的时间复杂度为 O(1)。如果经常需要重新分配内部数组以容纳新元素，则此操作成为时间复杂度 O(n) 级的操作。

5) 出队

定义 Dequeue 方法实现出队操作。该操作取出队头处的数据元素，下一个数据元素成为新的队头元素。当队列不为空时，取走 front 位置上的队首数据元素，front 循环加 1，新 front 位置上的数据元素成为新的队首数据元素。Dequeue 方法的返回值声明为类型 T，即此时出队的数据元素具有类型 T，在调用该操作时，将与队列实例定义时声明的类型保持一致。此方法的时间复杂度是 O(1)。编码如下：

```
public T Dequeue() {
    T k = default(T);
    if (!Empty) {                          //队列不空
        k = items[front];                  //取得队头结点数据元素
        front = (front + 1) % items.Length;
        return k;
```

```
        }
        else                                    //栈空时产生异常
            throw new InvalidOperationException("Queue is Empty: " + this.GetType());
    }
```

6) 获得队首对象，但不将其移除

该操作获得队首数据元素，但不移除该元素。当队列不为空时，取走 front 位置上的队首数据元素，front 变量保持不变。此方法的运算复杂度是 O(1)。编码如下：

```
    public T Peek() {
        if (!Empty)
            return items[front];
        else
            throw new InvalidOperationException("Queue is Empty: " + this.GetType());
    }
```

7) 输出队列中所有数据元素的值

当队列非空时，从队首结点开始，直至队尾结点，依次输出结点值。编码如下：

```
    public void Show(bool showTypeName) {
        if (showTypeName)
            Console.Write("Queue: ");
        int i = this.front;
        int n = i;
        if (!Empty) {
            if (i < this.rear) {
                n = this.rear - 1;
            }
            else {
                n = this.rear + this.items.Length - 1;
            }
            for (; i <= n; i++) {
                Console.Write(items[i % items.Length] + "  ");
            }
        }
        Console.WriteLine();
    }
```

由此可见，相对于原始顺序队列的设计，顺序循环队列 SequencedQueue 类在设计上有以下两个方面的改进：

- 入队时只改变下标 rear，出队时只改变下标 front， 它们都做“循环”移动，取值范围是 0 到 items.Length - 1，这样可以重复使用队列内部的存储空间，因而避免“假溢出”现象。
- 在队列中设立一个空位置。如果不设立一个空位置，则队列空和队列满两种状态的条件都是队头指针与队尾指针相等，即 front == rear，那么就无法区分这两种状态。通过保留一个空位置，则队列满的条件变为(rear - front) % items.Length == 1。

【例3.7】　测试顺序循环队列的操作实现。

源程序 SequencedQueueTest.cs 使用声明在 DSAGL 命名空间中的 SequencedQueue 类，程序如下：

```
using System;
using DSAGL;
namespace stackqueuetest {
    class SequencedQueueTest {
        public static void Main(string[] args) {
            int i = 0, n = 2;
            SequencedQueue<string> q1 = new SequencedQueue<string>(20);
            while (i < args.Length) {
                Console.Write("Enqueue: " + (i + 1) + "\t");
                q1.Enqueue( (i+1).ToString() );            //入队
                q1.Show(true);
                q1.Enqueue(args[i]);                       //将命令行参数入队
                Console.Write("Enqueue: " + args[i] + "\t");
                q1.Show(true);
                i++;
            }
            while (!q1.Empty) {                            //数据依次出队
                Console.Write("Dequeue: ");
                string str = q1.Dequeue();
                Console.Write(str + "  ");
                Console.Write("\t");
                q1.Show(true);
            }
            Console.WriteLine();
        }
    }
}
```

用如下命令进行编译：

```
csc SequencedQueueTest.cs  /r: ..\stackqueue\bin\Debug\stackqueue.dll
```

从命令行输入参数运行 SequencedQueueTest 程序：

```
SequencedQueueTest Hello World
```

程序结果如下：

```
Enqueue: 1          Queue: 1
Enqueue: Hello      Queue: 1  Hello
Enqueue: 2          Queue: 1  Hello  2
Enqueue: World      Queue: 1  Hello  2  World
Dequeue: 1          Queue: Hello  2  World
Dequeue: Hello      Queue: 2  World
Dequeue: 2          Queue: World
Dequeue: World      Queue:
```

3.4.2 队列的链式存储结构及操作实现

采用链式存储结构实现的队列称为链式队列。队列作为一种特殊的线性数据结构，可以用单向链表实现队列的链式存储结构。设置变量 front 和 rear 分别指向队头和队尾数据结点，链式队列如图 3.13 所示。

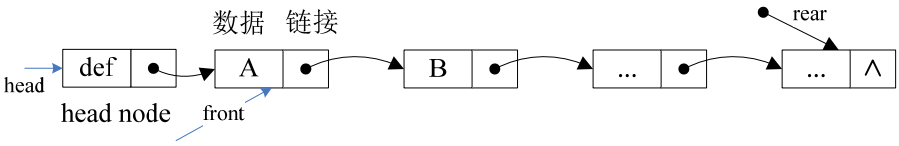


图 3.13 队列的链式存储结构

下面声明 LinkedQueue 类实现链式队列。

```
public class LinkedQueue<T> {
    private SingleLinkedList<T> items;
    private SingleLinkedNode<T> front, rear;
}
```

类中定义的成员变量 items 记录存储队列数据的单向链表，成员变量 front 和 rear 分别指向队头和队尾数据结点，结点类型为第二章中定义的单向链表的节点类 SingleLinkedNode，结点数据域的类型为泛型 T。用 LinkedQueue 类型定义和构造的对象就是一个具体的队列实例。

链式队列的基本操作作为 LinkedQueue 类的方法和属性成员予以实现，下面分别描述实现这些操作的算法。

1) 队列的初始化

用构造方法创建一条准备用以存储队列数据的单向链表，设置队列的初始状态为空。

```
public LinkedQueue() {
    items = new SingleLinkedList<T>();
    front = items.Head.Next;
    rear = items.Head;
}
```

2) 返回队列的元素个数

该操作告知当前队列中已有数据元素的个数，将这个操作用属性 Count 来实现，编码如下：

内嵌成员 items（单向链表对象）的结点个数（items.Count）也就是队列的元素个数。

```
public int Count {
    get { return items.Count; }
}
```

3) 判断队列的状态是否为空或是否已满

判断队列是否为空的操作即判断队列中是否有数据元素，该操作通过定义属性 Empty 来实现。当 front == null 且 rear == items.Head 时，队列为空，属性 Empty 此时应该返回 true。

```
public bool Empty {
    get { return (front == null) && (rear == items.Head); }
```

```
}
```

与链式栈一样，链式队列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入队时，向系统申请一个结点的存储空间，一般可在编程时认为系统所提供的可用空间是足够大的，所以不需要判断队列是否已满。

4) 入队

定义 Enqueue 方法实现入队操作。该操作将新的数据元素从队尾处加入队列，该元素成为新的队尾元素。在 rear 指向的队尾结点之后插入一个结点存放新数据 k，并更新 rear 指向新的队尾结点。Enqueue 方法的参数 k 的类型声明为 T 类型（形参类型为 T），即此时入队的数据元素类型是 T；而在调用入队操作时，实参的类型要与队列实例定义时声明的类型保持一致。此方法的运算复杂度是 O(1)。

```
public void Enqueue(T k) {
    SingleLinkedNode<T> q = new SingleLinkedNode<T>(k);
    rear.Next = q;
    front = items.Head.Next;
    rear = q;
}
```

5) 出队

定义 Dequeue 方法实现出队操作。该操作取出队头处的数据元素，下一个数据元素成为新的队头元素。当队列不为空时，取走 front 指向的队首结点的数据元素，并删除该结点，更新 front 指向新的队首结点。Dequeue 方法的返回值声明为类型 T，在调用出队操作时，返回值的类型将与队列实例定义时声明的类型保持一致。此方法的运算复杂度是 O(1)。

```
public T Dequeue() {
    T k = default(T);                    //置变量k为T类型的缺省值
    if (!Empty) {                        //队列不空
        k = front.Item;                  //取得队头结点数据元素
        front = front.Next;              //删除队头结点
        items.Head.Next = front;
        if (front == null)
            rear = items.Head;
        return k;
    }
    else
        throw new InvalidOperationException("Queue is Empty: " + this.GetType());
}
```

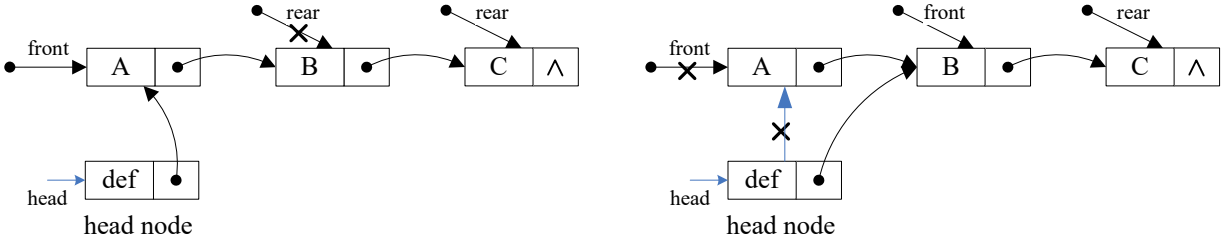
6) 获得队首对象，但不将其移除

当队列不为空时，取走 front 位置上的队首数据元素，front 不变。此方法的运算复杂度是 O(1)。

```
public T Peek() {
    if (!Empty)
        return front.Item;
    else
        throw new InvalidOperationException("Queue is Empty: " + this.GetType());
```

```
}
```

链式队列的基本操作如图 3.14 所示。



(a) 元素C入队　　(b) 元素A出队

图 3.14 链式队列的基本操作

由以上多个操作的算法实现分析可知，顺序队列 SequencedQueue 和链式队列 LinkedQueue，都实现了队列 Queue 这个抽象数据结构的基本操作。无论是 SequencedQueue 类还是 LinkedQueue 类，都可以用来建立具体的队列实例，通过队列实例调用入队或出队方法进行相应的操作。一般情况下，解决某个问题关注的是队列的抽象功能，而不必关注队列的存储结构及其实现细节。

3.4.3 队列的应用举例

队列是一种具有“先进先出”特性的特殊线性结构，可以作为求解具有“先进先出”特性问题的数学模型，因此队列结构成为解决相应问题算法设计的有力工具。在计算机系统中，当一些过程需要按一定次序等待特定资源就绪时，系统需设立一个具有“先进先出”特性的队列以解决这些过程的调度问题。在后面的章节中将介绍的非线性结构广度遍历算法，如按层次遍历二叉树、以广度优先算法遍历图，都具有“先进先出”的特性，这些算法的实现需要使用队列。下面的例题讨论一个应用队列结构的典型例子。

【例3.8】　解素数环问题。将 1, 2, …, n 共 n 个数排列成环形，使得每相邻两数之和为素数，构成一个素数环。

如图 3.15 所示，解素数环问题的算法思想是依次试探每个数：用一个线性表存放素数环的数据元素，用一个队列存放等待检查的数据元素，依次从队列取一个数据元素 k 与素数环最后一个数据元素相加，若两数之和是素数，则将 k 加入到素数环（线性表）中，否则 k 暂时无法进入素数环，此时须让它再次放入队列等待处理。重复上述操作，直到队列为空。

本例应用顺序表 SequencedList 类和顺序队列 SequencedQueue 类。创建 SequencedList 类的一个线性表实例 ring1，用以存放素数环的数据元素，创建 SequencedQueue 类的一个实例 q1 作为队列，存放待检测的数据元素。静态方法 IsPrime(k) 判断 k 是否为素数。

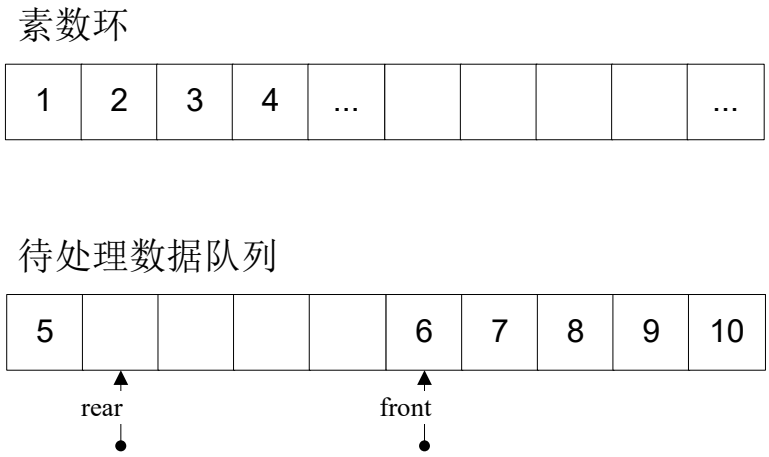


图 3.15 利用线性表和队列数据结构解素数环问题

```
using System; using DSAGL;
namespace stackqueuetest {
public class PrimeRing {
    public static bool IsPrime(int k){
        int j = 2;
        if(k==2)   return true;
        if(k<2 || k>2 && k%2==0) return false;
        else{
            j=3;
            while(j<k && k%j!=0)
                j = j+2;
            if(j>=k)    return true;
            else  return false;
        }
    }
    public static void Main(string[] args){
        int i,j,k,n=10;
        SequencedQueue<int> q1 = new SequencedQueue <int>(); //创建一个队列q1
        SequencedList<int> ring1 = new SequencedList<int>(n);//创建一个线性表ring1表示素数环
        ring1.Add(1);                                        //1添加到素数环中
        for(i=2;i<=n;i++)                                    //2--n全部入队
            q1.Enqueue(i);
        q1.Show(true);                                       //输出队列中全部数据元素
        i = 0;
        while(!q1.Empty){
            k = q1.Dequeue();                                //出队
            Console.Write("Dequeue: " + k + "\t");
            j = ring1[i] + k;
            if(IsPrime(j)){                                  //判断j是否为素数
                  ring1.Add(k);                              //k添加到素数环中
```

```
                    Console.Write("add into ring\t");
                    i++;
                } else{
                    q1.Enqueue(k);                    //k再次入队
                    Console.Write("wait again\t");
                }
                q1.Show(true);
            }
            Console.WriteLine();
            ring1.Show(true);
        } //end of Main
    }//end of class
}//end of namespace
```

程序运行结果如下：

```
Queue: 2  3  4  5  6  7  8  9  10
Dequeue: 2      add into ring   Queue: 3  4  5  6  7  8  9  10
Dequeue: 3      add into ring   Queue: 4  5  6  7  8  9  10
Dequeue: 4      add into ring   Queue: 5  6  7  8  9  10
Dequeue: 5      wait again      Queue: 6  7  8  9  10  5
Dequeue: 6      wait again      Queue: 7  8  9  10  5  6
Dequeue: 7      add into ring   Queue: 8  9  10  5  6
Dequeue: 8      wait again      Queue: 9  10  5  6  8
Dequeue: 9      wait again      Queue: 10  5  6  8  9
Dequeue: 10     add into ring   Queue: 5  6  8  9
Dequeue: 5      wait again      Queue: 6  8  9  5
Dequeue: 6      wait again      Queue: 8  9  5  6
Dequeue: 8      wait again      Queue: 9  5  6  8
Dequeue: 9      add into ring   Queue: 5  6  8
Dequeue: 5      wait again      Queue: 6  8  5
Dequeue: 6      wait again      Queue: 8  5  6
Dequeue: 8      add into ring   Queue: 5  6
Dequeue: 5      add into ring   Queue: 6
Dequeue: 6      add into ring   Queue:
SequencedList: 1  2  3  4  7  10  9  8  5  6
```

3.5　递归

在高级编程语言，如 C 语言中，若一个函数直接或间接地调用自己，则称这个函数是递归函数。递归（recursion）是数学定义和计算中的一种思维方式，它用对象自身来定义一个对象。在程序设计中常用递归方式来实现一些问题的求解。

在数学及程序设计方法中，递归可以出现在算法描述和数据结构的定义中。存在自调用的算法称为递归算法（recursive algorithm）。在数据结构的描述中，若一个对象用它自己来定义它的一部分，则

称这个对象是递归的。

1. 递归算法

递归算法将待求解的问题推到比原问题更简单且解法相同或类似的问题的求解，然后再得到原问题的解。例如，求阶乘函数 $f(n)=n!$，为计算 $f(n)$，将它推到 $f(n\text{-}1)$，即

$$f(n)=n \times f(n\text{-}1)$$

而计算 $f(n\text{-}1)$的算法与 $f(n)$是一样的。

由此可见，用递归算法求解较为复杂的问题的过程具有下列特点：

- 如果原问题能够分解成几个相对简单且解法相同或类似的子问题时，只要子问题能够解决，那么原问题就能用相同或类似的方法求解，即递归求解原问题。例如，9！=9×8！。
- 不断分解原问题，直至分解到某个可以直接解决的子问题时，就停止分解。这些可以直接求解的问题称为递归结束条件。例如，1！=1。

数学上常用的阶乘函数、幂函数、Fibonacci 数列等，它们的定义和计算都可以是递归的。在这类函数的递归定义中，一方面给出被定义函数在某些自变量处的值，另一方面则给出由已知的被定义函数值逐步计算未知的被定义函数值的规则。

例如，阶乘函数 $f(n)=n!$ 的递归定义式为

$$n!=\begin{cases}1 & n=0,1\\ n\times(n-1)! & n\geq 2\end{cases}$$

又如，Fibonacci 数列的首两项为 0 和 1，以后各项的值是其前两项值之和：

$$\{0, 1, 1, 2, 3, 5, 8, \ldots\}$$

Fibonacci 数列的递归定义为：

$$f(n)=\begin{cases}n & n=0,1\\ f(n-1)+f(n-2) & n\geq 2\end{cases}$$

【例3.9】　阶乘函数 n！的递归实现。

例如，求 5!所进行的分解及递归调用的情况如图 3.16 所示。

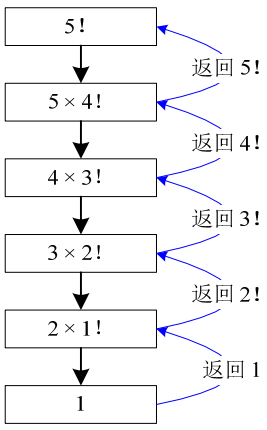


图 3.16 阶乘函数的分解和递归调用与返回

程序如下：

```
using System;
```

```
namespace stackqueuetest {
public class Factorial {
    //递归方法
    public static int f(int n){
        if(n==0)
            return 1;
        else {
            Console.WriteLine(n+"! = " + n + "*" + (n-1) + "! ");
            return n * f(n-1);
        }
    }
    public static void Main(string[] args){
        int i=5;
        Console.WriteLine(i + "! = " + f(i) );
    }
}
}
```

程序运行结果：

```
5! = 5*4!
4! = 4*3!
3! = 3*2!
2! = 2*1!
1! = 1*0!
5! = 120
```

2. 递归数据结构

有些数据结构是可以用递归方式定义的。例如，单向链表结点类可以递归定义为：

Node＝(Data, Next_Node)

再如，链表也可以看成是一种递归的数据结构，链表 Z＝(h, Z_h)，h 代表头结点，Z_h 代表头结点的链域指向的子链表，如图 3.17 所示。使用递归的方式，定义链表就只需要一种数据结构类型，而一般的方法（如本章前面介绍的方法），需要同时定义结点和链表两种数据类型（两个类）。

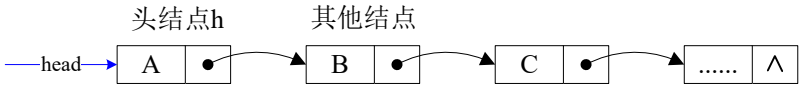


图 3.17　单向链表

我们也可以用递归形式来描述具有层次关系的树（tree）结构：树 T 是由 n（$n \geq 0$）个结点组成的有限集合，它或者是棵空树，或者包含一个根结点和零或若干棵互不相交的子树。

【例3.10】　单向链表结点递归定义的实现与测试。

定义递归式链表结点类 RLinkedNode 如下：

```
using System; using DSAGL;
public class RLinkedNode<T> : SingleLinkedNode<T> {
```

```
        //构造值为ch的结点
        public RLinkedNode(T ch) : base(ch) { }

         //构造值为ch，且是q的前驱结点
        public RLinkedNode(T ch, SingleLinkedNode<T> q) {
            this.Item = ch;
            this.Next = q;
        }

        //构造q的后继结点，且其值为ch
        public static RLinkedNode<T> AddNode(SingleLinkedNode<T> q, T ch) {
            SingleLinkedNode<T> p = q;
            RLinkedNode<T> t = new RLinkedNode<T>(ch);
            if (q != null) {
                while(p.Next!=null) p = p.Next;
                p.Next = t;
                return (RLinkedNode<T>) q;
            }
            else return t;
        }
    }
```

测试程序如下：

```
    using System;
    using DSAGL;
    namespace stackqueuetest {
        public class RLinkedNodeTest {
            public static void Main(string[] args) {
                Console.Write("Input: ");
                string str = Console.ReadLine();
                int i, count = str.Length;
                Console.Write("Show: ");
                for (i = 0; i < count; i++)                    //输出str元素值
                    Console.Write("  " + str[i]);
                Console.WriteLine();
                Console.WriteLine("count = " + count);      //str实际长度
                RLinkedNode<char> node = null;
                for (i = 0; i < count; i++) {                  //创建链表
                    node = new RLinkedNode<char>(str[i], node);
                    node.Show();                               //输出链表
                }
                Console.WriteLine();  node = null;
                for (i = 0; i < count; i++) {                  //重新创建链表
                    node = RLinkedNode<char>.AddNode(node, str[i]);
```

```
                node.Show();                                    //输出链表
            }
        }
    }
}
```

程序运行时，从键盘输入“CSharp”，结果如下：

```
Input: CSharp
Show:   C  S  h  a  r  p
count = 6
C.
S -> C.
h -> S -> C.
a -> h -> S -> C.
r -> a -> h -> S -> C.
p -> r -> a -> h -> S -> C.

C.
C -> S.
C -> S -> h.
C -> S -> h -> a.
C -> S -> h -> a -> r.
C -> S -> h -> a -> r -> p.
```

习题 3

3.1 填空：线性表、栈和队列都是_____结构，可以在线性表的_____位置插入和删除元素；栈是一种特殊的线性结构，允许插入和删除操作的一端称为_____。不允许插入和删除运算的一端称为_____，所以栈又称为____进____出型线性结构。队列是只能在_____插入和_____删除元素的特殊线性结构，所以队列又称为____进____出型结构。

3.2 填空：设栈 S 的初始状态为空，元素 a，b，c，d，e，f 依次入栈 S，出栈的序列为 b，d，c，f，e，a，则栈 S 的容量至少应该是_____。

3.3 说明顺序队列的“假溢出”是怎样产生的，并说明如何用循环队列解决假溢出。循环队列的基本操作，如初始化，判断队列满、判断队列空、返回队列元素个数、入队、出队等是如何实现的？

3.4 分别在 SequencedStack、SequencedQueue、LinkedStack 和 LinkedQueue 类中编程实现检测数据结构中是否包含某数据的操作：`public bool Contains(T k);`

3.5 分别在 SequencedStack、SequencedQueue、LinkedStack 和 LinkedQueue 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();`

3.6 构造顺序队列，它复制另一个队列，构造方法声明为：

```
public SequencedQueue(SequencedQueue<T> q);
```

3.7 说明以下算法的功能（栈 Stack 和队列 Queue 都是.NET Framework 在 System . Collections 命名空间中定义的类）。

```
void meth4(Queue q){
    Stack s = new Stack(); object d;
    while(q.Count!=0){
        d = q.Dequeue( ); s.Push(d);
    };
    while(s.Count!=0){
        d = s.Pop( ); q.Enqueue(d);
    }
}
```

3.8 分别用单向循环链表、双向循环链表结构实现队列，并讨论其差别。

3.9 实现用递归方式计算 Fibonacci 数列的算法。

3.10 写出表达式 a*(b+c)-d 的后缀表达式。

3.11 某个车站呈狭长形，宽度只能容下一台车，并且只有一个出入口。已知某时刻该车站状态为空，从这一时刻开始的出入记录为：“进，出，进，进，出，进，进，进，出，出，进，出”。假设车辆入站的顺序为 1，2，3，…，7，试写出车辆出站的顺序。