IPL 电子信息学院 武汉大学 Wuhan University

数据结构与算法 (C#语言版)

DATA STRUCTURE & ALGORITHM IN C#

第三章 栈与队列

王文伟 Wang Wenwei, Dr.-Ing.
Tel: 189-71562600
Email: wwwang@aliyun.com
Web: http://ipl.whu.edu.cn/sites/ced/st/

电子信息学院 Table of Contents 武汉大学 Wuhan University

本章位置

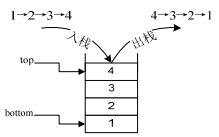
本章首先学习栈与队列的概念和抽象数据类型定义，然后分析它们的不同存储结构的实现和应用举例；最后介绍递归的定义、递归的算法和数据结构设计举例。

电子信息学院 Table of Contents 武汉大学 Wuhan University

3.0 Introduction

- 栈和队列是两种特殊的线性结构。
  - 它们的数据元素之间也具有顺序的逻辑关系，都可以采用顺序存储结构和链式存储结构实现。
  - 线性表的插入和删除操作不受限制，可以在任意位置进行。
  - 栈的插入和删除操作只允许在结构的一端进行。
  - 队列的插入和删除操作则分别在结构的两端进行。
- 栈的特点是后进先出(LIFO)，队列的特点是先进先出(FIFO)。

3.1 栈的概念及类型定义



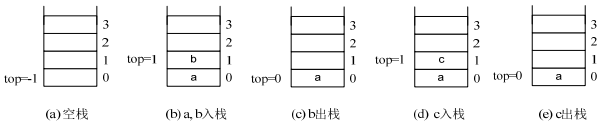
- 栈是一种“后进先出”（Last In First Out，LIFO）的线性结构。栈就像某种只有单个出入口的仓库，每次只允许一件件地往里面堆货物（入栈），然后一件件地往外取货物（出栈），不允许从中间放入或抽出货物。

3.1.1 栈的基本概念

- 栈（stack）是一种特殊的线性数据结构，元素之间具有顺序的逻辑关系，但插入和删除操作只允许在结构的一端进行。LIFO。
- 栈中插入数据元素的过程称为入栈(push)，删除数据元素的过程称为出栈(pop)。
- 允许插入和删除操作的一端称为栈顶（stack top），不允许操作的一端称为栈底（stack bottom）。
- 栈顶指针：标识栈顶的当前位置（动态）。

栈状态随插入和删除操作而进行的变化

- 栈顶指针：标识栈顶的当前位置（动态）。



对于数据序列{a，b，c }依次进行
{ Push(a)，Push(b)，Pop( )，Push(c)，Pop( )}

3.1.2 栈的抽象数据类型

- 栈是由若干数据元素组成的有限数据序列。
- 对于由$n(n\geq 0)$个数据元素a_0，a_1，a_2，…，a_{n-1}组成的栈，记作：Stack = { a_0，a_1，a_2，…，a_{n-1}}
- 栈中的数据元素至少具有一种相同的属性，属于同一种抽象数据类型。
- n表示栈中数据元素的个数，称为栈的长度。若n=0，则称为空栈。

栈的基本操作

- Initialize: 栈的初始化。创建一个栈实例，并进行初始化操作，例如设置栈的状态为空。
- Count: 返回栈中元素个数。
- Empty/Full: 判断栈的状态是否为空或满。
- Push: 入栈。该操作将数据元素插入栈中作为新的栈顶元素。核心操作
- Pop: 出栈。该操作取出当前栈顶数据元素，下一个数据元素成为新的栈顶元素。核心操作
- Peek:探测栈顶。获得但不移除栈顶数据元素。栈顶指针不变。

3.1.3 C#中的栈类

1. 非泛型栈类Stack

- 在System . Collections名字空间中定义了一个栈类Stack，刻画一种具有后进先出性质的数据集合，其数据元素的类型定义为object。
- Stack类的属性和方法：
  公共构造函数
- Stack(); //初始化Stack类的新实例
- Stack(ICollection c);
- Stack (int capacity);

```
Stack  s1 = new Stack();
Stack  s2 = new Stack(30);
……
i = s1.Count
```

公共属性

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

Stack的公共方法

- virtual void Push(object obj);
  //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();
  //返回栈顶的对象，但不将其移除
- virtual bool Contains(object obj);
  //确定某个元素是否在栈中

2. 泛型栈类Stack<T>

- 2.0版C#语言增加了泛型（Generics）。泛型通常与集合一起使用。 新的命名空间System.Collections.Generic，它包含定义泛型集合的接口和类，泛型集合允许用户创建强类型集合，它能提供比非泛型集合更好的类型安全性和性能。

```
Stack<int>  s1 = new Stack<int> ();
s1.Push(12);
Stack<string>  s2 = new Stack<string> ();
s2.Push("Wuhan University");
```

【例3.1】 创建Stack对象，向其添加元素以及打印出其值

```
using System; using System.Collections;
  Stack s = new Stack();
  myStack.Push("Hello");
  myStack.Push("World");s.Push("!");
  Console.WriteLine( "My Stack:" );
  Console.WriteLine("\tCount: {0}",
     s.Count );
  Console.Write( "\tValues:\n" );
  foreach(object o in  s)
      Console.WriteLine( "\t{0}", o);
```


程序运行结果

```
My Stack:
        Count:    3
        Values:
        !
        World
        Hello
```

输出序列的顺序与入栈的顺序相反，这是栈的先进后出（LIFO）特性造成的。

【例3.2】利用栈进行数制转换

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

数制转换就是要确定序列 $\{a_0, a_1, a_2, \ldots, a_n\}$

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

例如： $(2468)_{10}$ 转换成 $(4644)_8$ 的运算过程如下：

	N	N / 8	N % 8
计算顺序 ↓	2468	308	4
	308	38	4
	38	4	6
	4	0	4

输出顺序 ↑


```
using System; using System.Collections.Generic;
namespace stackqueuetest {
  public class DecOctConversion {
    public static void Main(string[] args){
          int n = 2468;
         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();
    } } }
```

3.2 栈的存储结构及实现

3.2.1 栈的顺序存储结构及操作实现
3.2.2 栈的链式存储结构及操作实现
3.2.3 栈的应用举例

- 栈作为一种特殊的线性结构，可以如同一般线性表一样采用顺序存储结构和链式存储结构实现。顺序存储结构的栈称为顺序栈（Sequenced Stack），链式存储结构的栈称为链式栈（Linked Stack）。

3.2.1 栈的顺序存储结构及操作实现

- 栈的顺序存储结构：用一组连续的存储空间存放栈的数据元素。定义SequencedStack类刻画之。
- 成员变量items定义为数组类型，即准备用数组存储栈的元素。成员变量top指示当前栈顶元素的下标，起着栈顶指针的作用。
- 用SequencedStack类构造的对象就是栈实例。通过对这个对象调用（公有的）属性和方法进行相应的操作。

```
public class SequencedStack<T> {
    private T[] items;
    private const int empty = -1;
    private int top = empty;
    ……
}
```

顺序栈的操作

- 栈的初始化: 构造方法
- 返回栈的元素个数：Count
- 判断栈的空与满状态：Empty/Full
- 入栈Push(k)：当栈不满时，栈顶元素下标top加1，将k放入top位置，作为新的栈顶数据元素。
- 出栈Pop：当栈不空时，取走top处的元素，top减1，下一位置的数据元素作为新的栈顶元素。
- 获得栈顶数据元素的值Peek。当栈非空时，获得top位置处的数据元素，此时该数据元素未出栈，top值不变。

1）栈的初始化

- 构造方法初始化一个栈对象，它为items数组申请指定大小的存储空间来存放栈的数据元素，设置栈初始状态为空。

```
public SequencedStack(int n) {
    items = new T[n];
    top = empty;
}
public SequencedStack():this(16){ }
```

使用

```
var s = new SequencedStack<int>();
var s = new SequencedStack<int>(128);
```


2）返回栈的元素个数

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

- 将返回栈的元素个数的成员定义为属性，相对于成员方法显得更简洁。
- 属性可以是计算出来的，比变量更具动态性。

3）判断栈的空与满状态

- Empty:当top==empty时，表明栈为空状态。
- Full :当top ≥ items.Length-1时，表明栈为满状态。

```
public bool Empty{
  get{
    return top==empty;
  }
}
```

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


4）入栈Push(k)

- 当栈不满时，栈顶元素下标top自加1，将k放入top位置，作为新的栈顶元素。
- 入栈的数据是T类型，在调用该操作时，实参的类型要与栈定义时声明的元素类型保持一致。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。
- 当栈当前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。

Push(k)的代码

```
public void Push(T k) {
    if (Full) DoubleCapacity();
    top++;
    items[top] = k;   }
```

重新分配存储空间

```
private void DoubleCapacity() {
    int count = Count;
    int capacity = 2 * items.Length;
    T[] copy = new T[capacity];
    for (int i = 0; i < count; i++)
        copy[i] = items[i];
    items = copy;  }
```

Push操作的时间复杂度

- 当栈不满时，Push操作的时间复杂度为O(1)。
- 如果经常需要增加容量以容纳新元素，则Push操作的时间复杂度 成为O(n)。

5）出栈Pop

- 当栈不空时，取走栈顶top位置处的元素，top自减1，下一位置上的元素成为新的栈顶元素。出栈的数据元素具有类型T，在调用该操作时，将与栈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);
    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();
}
```


6）获得栈顶数据元素的值Peek

- 当栈非空时，获得top位置处的元素，此时该数据未出栈，变量top保持不变。

```
public T Peek(){
  if(!Empty)
    return items[top];
  else
    new InvalidOperationException();
}
```


7）显示栈中每个数据元素的值

- 当栈非空时，从栈顶结点开始，直至栈底结点，依次输出结点值。

```
public void Show(bool showTypeName) {
  if(showTypeName)
    Console.Write(“Stack: ”);
  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

- 编译并运行SequencedStackTest .cs，运行时从命令行输入参数：

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



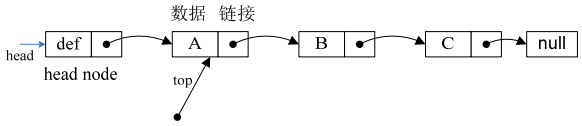
```
int i = 0;
SequencedStack<string> s1 = new SequencedStack<string>(20);
Console.Write("Push: ");
while (i < args.Length) {
  s1.Push(args[i]); Console.Write(args[i] + "  "); i++; }
s1.Show(true);                    //输出栈中各元素值
Console.Write("Pop : ");
while (!s1.Empty) {               //全部出栈
    Console.Write(s1.Pop() + "  ");   }
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();
```

3.2.2 栈的链式存储结构及操作实现

链式栈可以看成一种特殊的单向链表

本项目还要添加对lists项目的引用

```
public class LinkedStack<T>:
        SingleLinkedList<T> {
    private SingleLinkedNode<T> top;
    ……
}
```



单向链表及其结点类（以前的设计成果）

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

```
public class SingleLinkedList<T> {
  private SingleLinkedNode<T> head;
  public SingleLinkedNode<T> Head {
    get{return head;} set{head = value;}   }
  ……
}
```


链式栈的操作

- 栈的初始化：构造方法
- 判断栈的空与满状态：Empty/Full
- 返回栈的元素个数：Count
- 入栈：Push
- 出栈：Pop
- 获得栈顶数据元素的值，该数据元素未出栈Peek。

- 成员变量top指向栈顶数据元素结点，结点类型为单向链结点类SingleLinkedNode，结点数据域的类型为泛型T。
- LinkedStack类的一个对象就是一个栈。

1）栈的初始化

- 用构造方法创建一个栈，它用基类（SingleLinkedList类）的构造方法创建一条单向链表，栈的状态初始为空。

```
public LinkedStack(): base() {
    top = base.Head.Next;
}
```


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，LinkedStack类继承该功能，无需再写实现该功能的代码。

3）判断栈的空与满状态

- 当top == null时，栈为空；
- 动态向系统申请存储空间，不必判断栈是否已满。

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

与前面比较，可以体会到面向对象程序设计带来的便利。导出类既可以直接继承基类的属性和方法（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Push

- 在栈顶结点top之前插入一个结点来存放数据k，并使top指向新的栈顶结点。

```
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;
}
```

运算复杂度是 O(1)

5）出栈Pop

- 当栈不为空时，取走top指向的栈顶结点的值，删除该结点，使top指向新的栈顶结点。

```
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();
}
```

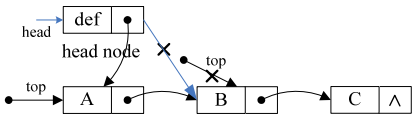
运算复杂度是 O(1)

6）获得栈顶数据元素的值Peek

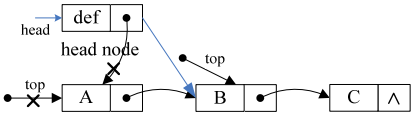
- 当栈非空时，获得栈顶top的数据，此时该数据元素未出栈，top值不变。

```
public T Peek(){
  if(!Empty)
    return top.Item;
  else
    throw new InvalidOperationException();
}
```


链式栈的基本操作示意



(a) 元素入栈



(b) 元素出栈

应用中首先关注数据结构的抽象功能

- 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。

- 基于栈结构的函数嵌套调用
  - 判断表达式中括号是否匹配
  - 使用栈计算表达式的值

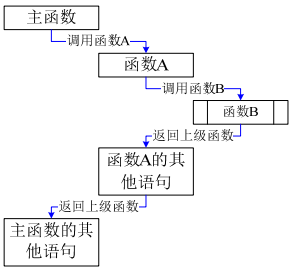
1. 基于栈结构的函数嵌套调用

- 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中存在对另一个函数的调用，每个函数在执行完后应返回到调用它的函数中，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。系统建立一个栈结构可以实现这种函数嵌套调用机制。
- 执行函数A时，A中的某语句调用函数B，系统要做一系列的入栈操作：
  - 将函数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；
  - 将A调用函数B的实参保存在栈中，该过程称为实参压栈；
  - 在栈中分配函数B的局部变量。

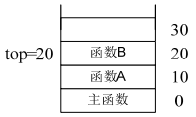
子函数返回的过程

- B函数执行完成时，系统则要做一系列的出栈操作才能保证将系统控制返回调用B的函数A中
  - 退回栈中为函数B的局部变量分配的空间；
  - 退回栈中为函数B的参数变量分配的空间；
  - 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A函数。
- 函数返回的次序与函数调用的次序正好相反。可见，系统栈结构是实现函数嵌套调用或递归调用的基础。

函数嵌套调用时的系统栈



(a) 函数的嵌套调用与返回



(b) 函数调用时的系统栈

2）判断表达式中括号是否匹配

如表达式中

（[] ()）或[（[] []）]

等为正确的括号匹配，

[（]）或（[（）]或[()]）

均为括号不匹配。

检验括弧匹配：按“期待的急迫程度”进行处理 。

括弧匹配算法的设计思想

1）凡出现左括弧(，则进栈；

2）凡出现右括弧)，首先检查栈是否空？

若栈空，则表明该“右括弧)”多余

否则和栈顶元素比较，

若相匹配，则“左括弧(”出栈

否则表明不匹配

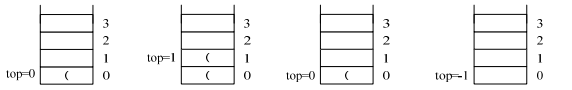
3）表达式检验结束时，

若栈空，则表明表达式中匹配正确

否则表明“左括弧(”有余。

表达式括号匹配过程中栈状态的变化

((9-1)*(3+4))



(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

```
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(!";
}
```

3）表达式求值

设 Exp = S1 + OP + S2

则称 OP + S1 + S2 为前缀表示法

S1 + OP + S2 为中缀表示法

S1 + S2 + OP 为后缀表示法

例如: **Exp** = a × b + (c – d / e) × f

前缀式: + × a b × – c / d e f

中缀式: a × b + c – d / e × f

后缀式: a b × c d e / – f × +

结论:

5) 后缀式的运算规则为:

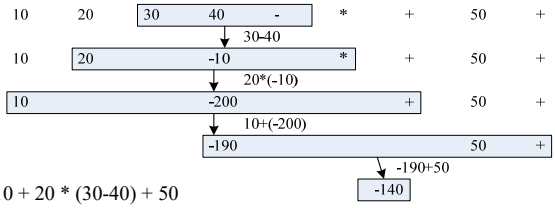
1）操作数之间的相对次序不变;

运算符出现的顺序恰为表达式的运算顺序;每个运算符和在它之前出现且紧靠它的两个操作数构成一个最小表达式;

2）运算符的相对次序不同;

后缀表达式的计算过程

- 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

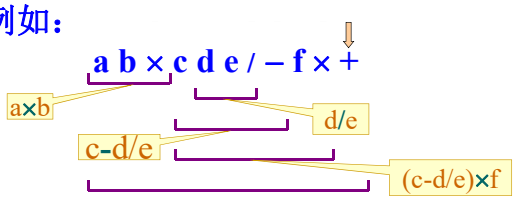


10 + 20 * (30-40) + 50

如何从后缀式求值？

- 先找运算符，
- 再找操作数

例如：



如何从原表达式求得后缀式？

分析“原表达式”和“后缀式”中的运算符:

原表达式: a + b × c – d / e × f

后缀式: a b c × + d e / f × –

在后缀式中，优先权高的运算符领先于优先权低的运算符出现。

每个运算符的运算次序要由它之后的一个运算符来定。

从原表达式求得后缀式的规律

1) 设立暂存运算符的**栈**OPTR和暂存操作数的**栈**OPND

2) 设表达式的结束符为“#”，
予设运算符栈的栈底为“#”

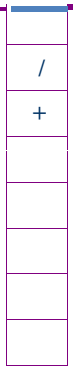
3) 若当前字符是操作数，则进入**栈**OPND;

4) 若当前运算符的优先数高于栈顶运算符，则进栈;

5) 否则，退出栈顶运算符发送给后缀式;

6) “(”对它之前后的运算符起隔离作用，“)”可视为自相应左括弧开始的表达式的结束符。

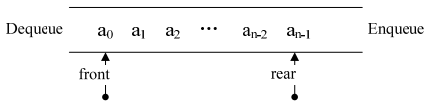
a b c d e / + × f - × #



3.3 队列的概念及类型定义

3.3.1 队列的基本概念

- 队列(queue)是一种特殊的线性数据结构，其插入和删除操作分别在表的两端进行，是一种“先进先出”（First In First Out，FIFO）的线性结构。
- 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。此类问题的求解，需要用到队列数据结构。
- 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）。



3.3.2 队列的抽象数据类型

- 队列是由若干数据元素组成的有限数据序列
- 对于由$n(n\geq 0)$个数据元素a_0，a_1，a_2，…，a_{n-1}组成的队列，记作：
  Queue= $\{a_0, a_1, a_2, \ldots, a_{n-1}\}$
- n表示队列的元素个数，称为队列的长度，若n=0，则称为空队列。
- 队列中的数据元素至少具有一种相同的属性，属于同一种抽象数据类型。

队列的基本操作

- Initialize: 队列的初始化。创建一个队列，并进行初始化操作。
- Count: 返回队列中元素个数。
- Empty:　判断队列的状态是否为空。
- Full: 判断队列的状态是否已满。
- Enqueue: 入队。该操作将数据加入队列队尾处。在入队前须判断队列的状态是否已满。
- Dequeue: 出队。该操作取出队头元素。在出队前，须判断队列的状态是否为空。
- Peek:探测队首。获得但不移除队首数据元素。

3.3.3 C#中的队列类

1. 非泛型队列类Queue

- 在System . Collections命名空间中定义了一个队列类Queue，提供了一种数据先进先出的集合，其数据元素的类型是object类。
- Queue类的属性和方法：
  公共构造函数
- Queue();　　//初始化Queue类的新实例
- Queue(ICollection c);
- Queue(int capacity);

公共属性

- virtual int Count {get;}
  //获取包含在队列中的元素数

```
Queue q = new Queue();
int  i = q.Count;
```


Queue的公共方法

- virtual void Enqueue(object d);
  //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 x);
  //确定某个元素是否在队列中

【例3.5】创建Queue对象，向其添加值并打印出其值

```
using System; using System.Collections;
Queue myQ = new Queue();
myQ.Enqueue("Hello"); myQ.Enqueue("World");
myQ.Enqueue("!");
Console.WriteLine( "myQ" );
Console.WriteLine( "\tCount: {0}", myQ.Count );
Console.Write( "\tValues: " );
foreach(object o in  myQ)
    Console.Write( "{0}\t", o);
Console.WriteLine();
```

程序运行结果：

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


2. 泛型队列类Queue<T>

- 2.0版C#语言增加了泛型（Generics）。泛型通常与集合一起使用。 新的命名空间System.Collections.Generic，它包含定义泛型集合的接口和类，泛型集合允许用户创建强类型集合，它能提供比非泛型集合更好的类型安全性和性能。

```
Queue<int> q = new Queue<int> ();
q.Enqueue(14);
```


3.4 队列的存储结构及实现

- 队列作为一种特殊的线性结构，可以如同栈和线性表一样，采用顺序存储结构和链式存储结构实现。顺序存储结构的队列称为顺序队列（Sequenced Queue），链式存储结构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

3.4.2 队列的链式存储结构及操作实现

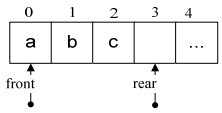
3.4.3 队列的应用举例

3.4.1 队列的顺序存储结构及操作实现

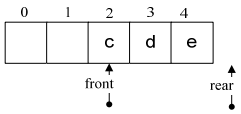
```
public class SequencedQueue<T> {
    private T[] items;
    private int front, rear;
    ……
}
```

- 顺序队列用一组连续的存储空间存放队列的元素。成员变量items用数组存储队列的元素，成员变量front和rear分别作为队头元素下标和下一个入队数据元素位置的下标，构成队头指针和队尾指针。
- 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。

顺序队列的“假溢出”问题



(a) 队列的顺序存储结构



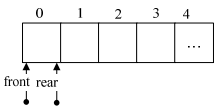
(b) 顺序队列的“假溢出”

- 设先有(a，b，c)入队，那么front=0，rear=3。接着a和b出队，d和e入队，front=2，rear=5。如果再有新数据入队，应存放于rear=5位置处，数组下标溢出。
- 顺序队列因多次入队和出队操作后出现的有存储空间但不能进行入队操作的溢出现象称为**假溢出**。
- 假溢出缺陷是因为顺序队列没有重复使用存储单元的机制。解决办法是将顺序队列设计成“环形”结构。

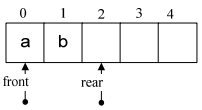
顺序循环队列

取模运算

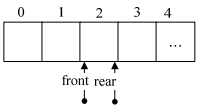
- front = (front + 1) % items.Length;
- rear = (rear + 1) % items.Length;



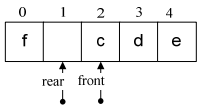
(a) 队列的初始状态



(b) a和b入队



(c) a和b出队，队列呈空状态



(d) c、d、e和f入队，队列满

顺序“循环”队列的操作

- 队列的初始化：SequencedQueue构造方法
- 返回队列的元素个数：Count
- 判断队列的空与满状态：Empty/Full
- 入队：Enqueue
- 出队：Dequeue
- 获得队首对象，但不将其移除：Peek

1）队列的初始化

- 构造方法初始化一个队列对象，它为items数组变量申请指定大小的存储空间，以存放队列的数据元素，设置队列初始状态为空。

```
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

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

3）判断队列的空与满状态Empty/Full

- 当front == rear时，表明队列中没有数据元素，队列为空；
- 当front == (rear+1) % items.Length时，表明队列已满，此时items数组中仍有一个空位置。

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

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


4）入队

- 当队列不满时，将参数*k*表示的元素存放在rear位置，作为新的队尾数据元素，rear循环加1。
- 当队列当前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。
- 参数*k*（即入队的数据元素）声明为T类型，在调用该操作时，实参的类型要与队列定义时声明的类型保持一致。

```
public void Enqueue(T k) {
    if (Full) DoubleCapacity();
    items[rear] = k;
    rear = (rear + 1) % items.Length;
}
```

```
var q = new SequencedQueue<int> ();
q.Enqueue(14);
```


重新分配存储空间

```
private void DoubleCapacity() {
  int i, j , count = Count;
  int capacity = 2 * items.Length - 1;
  T[] copy = new T[capacity];
  for (i = 0; i < count; i++) {
    j = (i + front) % items.Length;
    copy[i] = items[j];
  }
  front = 0; rear = count; items = copy;
}
```


入队操作的时间复杂度

- 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

- 当队列不空时，取走front位置上的队首元素，front循环加1，指向新的队首元素。
- 运算复杂度是O(1)。

```
public T Dequeue() {
  T k = default(T);
  if (!Empty) {//队列不空
    k = items[front];
    front = (front + 1) % items.Length;
    return k;
  } else//栈空时产生异常
   throw new InvalidOperationException();
}
```


6）获得队首对象

- 获得但不移除队首对象。当队列不为空时，取走front位置上的队首数据，front不变。

```
public T Peek() {
  if (!Empty)
    return items[front];
  else
    throw new InvalidOperationException();
}
```

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。

```
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();
}
```

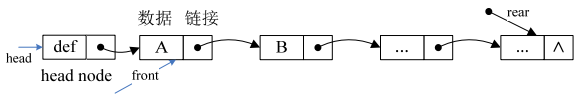

顺序循环队列小结

- 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 ~ items.Length - 1，这使得存储单元可以重复使用，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，则队列满的条件也是front == rear，那么就无法与队列空的条件（front == rear）相区别。而设立一个空位置，则队列满的条件是(rear - front) % items.Length == 1

3.4.2 队列的链式存储结构及操作实现

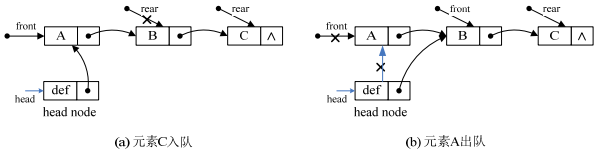
```
public class LinkedQueue<T> {
  private SingleLinkedList<T> items;
  private SingleLinkedNode<T> front, rear;
}
```

- 成员变量items记录存储队列的单向链表，成员变量front和rear分别指向队头和队尾结点，结点类型为单链表节点类SingleLinkedNode<T>。
- LinkedQueue类的实例即为具体的队列。



链式队列队列的操作

- 队列的初始化：构造方法
- 返回队列的元素个数：Count
- 判断队列的空与满状态：Empty/Full
- 入队：Enqueue
- 出队：Dequeue
- 获得队首对象：Peek



1）队列的初始化

- 构造方法创建一条单向链表准备用以存储队列数据，设置初始状态为空 。

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


2）返回队列中元素的个数

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

3）判断队列的空与满状态

- 当front == null且rear == items.Head 时，队列为空：Empty应指示true。
- 不需要判断队列是否已满。

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


4）入队

- 在rear指向的队尾结点之后插入一个结点存放k，并更新rear指向新的队尾结点。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;
    rear = q;
    front = items.Head.Next;
}
```


5）出队

- 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。此方法的运算复杂度是 O(1)。

```
public T Dequeue() {
  T k = default(T);
  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();
}
```


6）获得队首对象

- 获得但不移除队首对象。当队列不为空时，取走front位置上的队首数据元素，front不变。

```
public T Peek() {
  if (!Empty)
    return front.Item;
  else
    throw new InvalidOperationException();
}
```


应用中首先关注数据结构的抽象功能

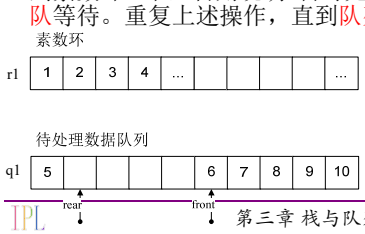
- 由以上多个操作的算法实现分析可知，顺序队列SequencedQueue和链式队列LinkedQueue，都实现了栈Queue这个抽象数据结构的基本操作。无论是SequencedQueue类还是LinkedQueue类，都可以用来建立具体的队列实例，通过队列实例调用入队或出队方法进行相应的操作。
- 一般情况下，解决某个问题关注的是队列的抽象功能，而不必关注队列的存储结构及其实现细节。

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，某些过程需要按一定次序等待特定资源就绪，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：处理“等待服务”问题时常使用队列。
- 实现广度遍历算法时使用队列。
- 解素数环问题。将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。

[例3.8] 解素数环问题　　试探法

- 创建一个线性表对象r1存放素数环的数据元素，创建一个对象q1作为队列，存放待检测的数据元素。方法IsPrime(k)判断k是否为素数。
- 首先将2~n的数全部入队q1，将出队数据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r1中，否则说明k暂时无法处理，必须再次入队等待。重复上述操作，直到队列q1为空。




```
ring1.Add(1);      //“1”添加到素数环中
for(i=2;i<=n;i++) //  2—n 全部入队q1
  q1.Enqueue(i);
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); }
```

3.5 递归

- 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，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。
- 递归可以出现在算法和数据结构的定义中。存在自调用的算法称为递归算法。若一个对象用它自己来定义自己，则称这个对象是递归的。

1. 递归算法（recursive algorithm）
2. 递归数据结构（recursive data structure）

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。
- 为了得到问题的解，将问题分解成几个相对简单且解法相同或类似的子问题时，只要解决了子问题，那么原问题就迎刃而解。
- 当分解后的子问题可以直接解决时，就停止分解。这些可以直接求解的问题称为递归结束条件。

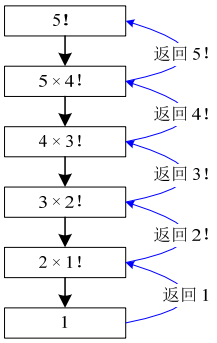
$$n!=\begin{cases}1 & n=0,1\\ n\times(n-1)! & n\geq 2\end{cases}$$

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

阶乘函数$n!$的递归定义式

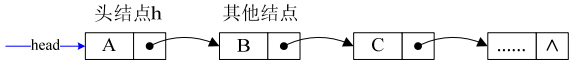
$$n!=\begin{cases}1 & n=0,1\\ n\times(n-1)! & n\geq 2\end{cases}$$

```
int f(int n){
  if(n==0)
    return 1;
  else {
   return n * f(n-1);
  }
}
```



2) 递归数据结构

- 有些数据结构是递归定义的。例：树（tree）结构：树T是由n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。
- 单向链表结点也可以递归定义为：
  Node＝(data, next_Node)
- 链表Z＝(头结点h,子链表Z_h)
  h代表头结点，Z_h代表头结点的链域指向的子链表



使用递归的方式，定义链表就只需要一种数据结构类型。

本章学习要点

1. 掌握栈和队列类型的特点，并能在相应的应用问题中正确选用它们。
2. 熟练掌握栈类型的两种实现方法，特别应注意栈满和栈空的条件以及它们的描述方法。
3. 熟练掌握顺序循环队列和链队列的基本操作实现算法，特别注意队满和队空的描述方法。

作业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表，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__出型线性表。队列是只能在_____插入和_____删除元素的特殊线性表，所以队列又称为____进出型结构。

3.2 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。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？

作业3（II）

3.3 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。

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

```
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);
   }  }
```


实习3

- 实验目的
  理解栈的基本概念及其基本操作。
- 题意
  将中缀表达式转换为后缀表达式，再求后缀表达式的值。