



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#


### 第二章 线性表

王文伟 Wang Wenwei, Dr.-Ing.  
 Tel: 189-71562600  
 Email: [wwwang@aliyun.com](mailto: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	第1章 绪论	本章首先学习 <b>线性表逻辑结构</b> 的特性，然后讨论以 <b>顺序存储结构</b> 实现的线性表和以 <b>链式存储结构</b> 实现的线性表的结点结构和各种操作的实现，分析、比较这些不同实现的优缺点。
	第2章 线性表	
	第3章 栈与队列	
	第4章 串	
	第5章 数组和广义表	
	第6章 树和二叉树	
	第7章 图	
	第8章 查找	
	第9章 排序	
第二章 线性表		2

电子信息学院	Table of Contents	武汉大学 Wuhan University	
<hr/>			
2.0 简介			
2.1 线性表的概念及类型定义			
2.2 线性表的顺序存储结构			
2.3 线性链表			
<hr/>			
IPL	第二章 线性表		3

## 2.0 Introduction

- ◆ **线性表**(linear list)是最简单、最基本的一种数据结构。它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。
- ◆ 线性表（逻辑结构）可以用**顺序存储结构**和**链式存储结构**（物理结构）实现。
- ◆ 本章讨论线性表的逻辑结构，以顺序存储结构实现的线性表（顺序表）和以链式存储结构实现的线性表（链表）在结点结构和操作实现方面的特性，分析、比较顺序表和链表的优缺点。

---

 天津理工大学

第二章 线性表



4

## 2.1 线性表的概念及类型定义

- ◆ 线性表是一组具有某种共性的数据元素的有序排列，数据元素之间具有**顺序关系**。除第一个和最后一个元素外，每个元素只有一个**前驱**元素和一个**后继**元素，第一个元素没有前驱，最后一个元素没有后继。
- ◆ 线性表可在任意位置进行插入和删除元素的操作。
- ◆ 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**。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**。

2.1.1 线性表的抽象数据类型ADT

2.1.2 C#中的线性表类




第二章 线性表

5

### 2.1.1 线性表的抽象数据类型ADT

- ◆ **线性表的数据元素**: 线性表是由n个数据元素组成的有限序列, 记作:  
$$\text{LinearList} = \{a_0, a_1, a_2, \dots, a_{n-1}\}$$
- ◆ **线性表的长度**: 线性表的元素个数,  $n$
- ◆ 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 $a_{i+1}$
- ◆ 线性表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ADT**。
- ◆ 线性表有两种存储结构实现方式: **顺序存储结构**(顺序表)和**链式存储结构**(链表)。

 第二章 线性表 6

## 线性表的基本操作

- ◆ **初始化(Initialize)**: 创建一个线性表实例, 并对该实例进行初始化。
- ◆ **访问(Get/Set)**: 对线性表中指定位置的数据元素进行取值或置值等操作。
- ◆ **插入(Insert)**: 在指定位置插入新的元素, 插入后仍为一个线性表。
- ◆ **删除(Remove)**: 删除指定位置的元素, 更改后的线性表仍然具有线性表的连续性。
- ◆ **复制(Copy)**: 重新复制一个线性表。
- ◆ **求长度(Count)**: 求线性表的数据元素个数。

## 线性表的基本操作(II)

- ◆ **合并(Join)**: 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。
- ◆ **查找(Search)**: 在线性表中查找满足某种条件的数据元素。
- ◆ **排序(Sort)**: 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。
- ◆ **遍历(Traversal)**: 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

1. 非泛型线性表类ArrayList

- ◆ 在System.Collections命名空间中定义了一个线性表类**ArrayList**, 它提供了一种“元素个数可按需**动态增加**”的**数组**, 其数据元素的类型是object类。
- ◆ ArrayList类的属性和方法:

### 公共构造函数

- ◆ **ArrayList()**; //创建 ArrayList 类的新实例
- ◆ **ArrayList(ICollection c)**; 新实例包含从指定集合c复制的元素
- ◆ **ArrayList(int initCapacity)**;

## ArrayList的公共属性

- ◆ **virtual int Count {get;}**  
//返回线性表的长度
- ◆ **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

```
ArrayList al = new ArrayList();  
al.Add(100); al.Add(5);  
int v = (int)al[5];    al[10] = 100;  
int i = al.Count;
```

## ArrayList的公共方法

- ◆ **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)**;  
//确定表中是否有某个元素

## ArrayList的公共方法(II)

- ◆ **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

### 【例2.1】 创建并初始化 ArrayList 以及打印出其值

```
using System;
using System.Collections;
public class SamplesArrayList {
    public static void Main() {
        // Creates and initializes a new ArrayList.
        ArrayList al = new ArrayList();
        al.Add("Hello");
        al.Add("World");    al.Add("!");
        myAL.Insert(1, "C#");
    }
}
```



```
// Displays the properties and values of the ArrayList.
Console.WriteLine( "al" );
Console.WriteLine( "\tCount:  {0}", al.Count );
Console.WriteLine( "\tValues:" );
foreach(object o in al){
    Console.WriteLine( "\t{0}", o);
}
Console.WriteLine();
al.Sort();
Console.WriteLine( "\tSorted Values:" );
for(int i=0; i<al.Count;i++){
    Console.WriteLine( "\t{0}", al[i]);
}
Console.WriteLine();
} }
```

### 程序运行结果

My ArrayList:

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

该例本身很简单，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 (Josephus) 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  $1, \dots, n$ 。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， $\dots$ ，如此反复直到最后剩一个人在座位上为止。
- ◆ 例：  $n=5, s=1, d=2$  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。
- ◆ 算法：用有  $n$  个元素的线性表分别表示  $n$  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。



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

```
public static void JosephusStart(
    int n,int s,int d) {
    ArrayList aRing = new ArrayList();
    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("\nNo. {0} is the last
    person.", aRing[0]);

```

## 程序运行结果

```

1 2 3 4 5
out: 2
1 3 4 5
out: 4
1 3 5
out: 1
3 5
out: 5
No.3 is the last person.

```

## 2. 泛型线性表类List<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。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。

## List的公共构造函数

- ◆ List<T>(); //构造 List<T> 类的新实例
- ◆ List<T>( int initCapacity);
- ◆ List<T>(IEnumerable<T> c); // 新实例包含从指定集合c复制的元素

## List的公共属性

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

```

List<double> ml = new List<double>();
ml.Add(1.5); ml.Add(4.3);
double d = ml [5]; ml[10] = 10.5;

```

## List的公共方法

- ◆ 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 int IndexOf(T x); //返回给定数据首次出现位置
- ◆ virtual bool Contains(T x); //确定某个元素是否在表中
- ◆ virtual void Remove(T x); //从表中移除特定对象的第一个匹配项

## 声明并构造特定类型的列表举例

```

List<int> a = new List<int> ();
// 声明并构造int型数列表
a.Add(86); a.Add(100);
// 向列表中添加整型元素
List<int> nums = new List<int> {0,1,2,3};
List<string> s = new List<string> ();
// 声明并构造字符串列表
s.Add("Hello"); s.Add("C# 2.0");
var st = new List< Student>();
// 声明并构造学生列表
st.Add(new Student ("200518001", "王兵" ));

```

## 2.2 线性表的顺序存储结构Sequenced List

- ◆用顺序存储结构实现的线性表称为**顺序表**。
- ◆顺序表用一组**连续的存储单元顺序存放线性表的数据元素**，数据元素在内存空间的物理存储次序与它们在线性表中的逻辑次序是一致的，即元素 $a_i$ 与其前驱元素 $a_{i-1}$ 及后继数据元素 $a_{i+1}$ 的位置相邻。

### 2.2.1 顺序表的类定义

### 2.2.2 顺序表的操作

### 2.2.3 顺序表操作的算法分析

## 顺序表具有随机访问特性

- ◆第 $i$ 个数据元素的地址为：

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

下标	元素内容	元素地址	元素的地址是该元素在线性表中位置（下标）的线性函数，而且每次寻址所花费的时间都是相同的。
0	$a_0$	$\text{Loc}(a_0)$	
1	$a_1$	$\text{Loc}(a_0) + c$	
...	...		
$i$	$a_i$	$\text{Loc}(a_0) + i \times c$	
$i+1$	$a_{i+1}$		
...	...		
$n-1$	$a_{n-1}$	$\text{Loc}(a_0) + (n-1) \times c$	

### 2.2.1 顺序表的类型定义

```
public class SequencedList<T> {  
    private T[] items;  
    private int count = 0;  
    private int capacity = 0;  
    ....  
}
```

- ◆用SequencedList类来刻画线性表。当我们需要使用一个线性表时，就创建该类的一个对象(实例)，它表示具体的线性表对象，通过对这个对象调用类中定义的公有（public）的属性和方法来进行相应的操作。
- ◆类中数据成员一般设置为私有的（private），对外是不可见的。**类的封装性**

### 2.2.2 顺序表的操作

- ◆顺序表的**初始化**
- ◆返回顺序**表长度**
- ◆判断顺序表的**空与满状态**
- ◆获取或设置顺序表的**容量**
- ◆获取或设置指定位置的数据**元素值**
- ◆**查找**
- ◆在顺序表的指定位置**插入**数据元素
- ◆**删除**顺序表指定位置的数据元素

#### 1) 顺序表的初始化

- ◆使用构造方法创建并初始化顺序表对象：
  - 形式1：为顺序表实例分配存储空间，设置顺序表为空状态。
  - 形式2：以一个数组的多个元素构造一个线性表

```
public SequencedList(int c) {  
    capacity = c;  
    items = new T[capacity];  
    count = 0; // 此时顺序表长度为0  
}  
public SequencedList() : this(16) {} // 省略构造方法  
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];  
    }  
}
```

#### 2) 返回顺序表长度

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

$a_1.Count$  返回表 $a_1$ 的元素个数

- ◆将返回顺序表长度的成员定义为**属性**，功能上与定义为方法成员类似，但相对于后者，前者显得更简洁。
- ◆属性语法并不陌生：C#用数组对象的Length属性获取数组的存储单元个数，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与满状态

- ◆ 定义布尔类型的Empty属性来指示顺序表的空状态：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。
- ◆ 定义布尔类型的Full属性来指示顺序表的满状态：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。当count>=预分配空间大小时，顺序表为满。
- ◆ 顺序表预分配的存储空间可以而且应该根据需要而调整。

```
public bool Empty {
    get {
        return count==0;
    }
}

public bool Full {
    get {
        return count >= items.Length;
        // items.Length表示 数组长度
    }
}
```

IPL

第二章 线性表

31

### 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，依次进行：
  - 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，
  - 设置capacity的新值。

IPL

第二章 线性表

32

```
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

- ◆ 声明索引器以提供对类的实例进行类似于数组的访问。

```
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() );
        }
    }
}
```

myList[10] = 10.5;

IPL

第二章 线性表

34

### 6) 查找具有特定值的元素

```
// 查找线性表是否包含k
// 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;
}
```

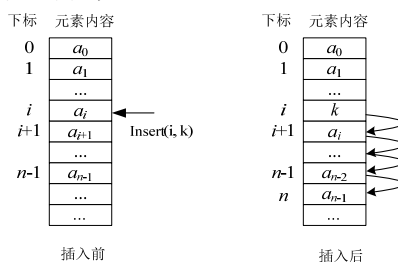
IPL

第二章 线性表

35

### 7) 在指定位置插入数据元素 Insert(int i,T k)

- ◆ 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，空出第i个内存单元位置，然后在第i个位置上放入给定值k。



IPL

第二章 线性表

36



## 在指定位置插入数据元素(II)

```
public void Insert(int i, T k) {
    if (count >= capacity) Capacity = capacity * 2;
    if (i < count) {
        for (int j = count - 1; j >= i; j--)
            items[j + 1] = items[j];
    }
    items[i] = k; count++; return; }

public void Add(T k) {
    if (count >= capacity) Capacity = capacity * 2;
    items[count] = k;
    count++;
}
```

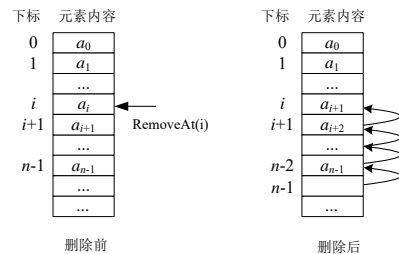
IPL

第二章 线性表

37

## 8) 删除指定位置的数据元素

- 删除第*i*个元素后要保持线性表的连续性，需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。将 $a_{i+1}$ 移动到位置*i*上，实际上就是删除了 $a_i$ 。



IPL

第二章 线性表

38

## 删除指定位置的数据元素(II)

```
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());
}
```

IPL

第二章 线性表

39

## 删除首个出现的k值元素

```
public void Remove(T k) {
    int i = IndexOf(k); //查找k值的位置
    if (i != -1) {
        for (int j = i + 1; j < count; j++)
            items[j - 1] = items[j];
        count--;
    } else
        Console.WriteLine(k + "值未找到，无法删除!");
}
```

IPL

第二章 线性表

40

## 9) 输出顺序表

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

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

IPL

第二章 线性表

41

## 2.2.3 顺序表操作的算法分析

- 时间复杂度为 $O(1)$ 的操作：判断空/满状态，求长度，获取或设置指定位置的元素值等操作实现所蕴涵的元操作的执行次数都与元素的个数无关。
- 时间复杂度为 $O(n)$ 的操作：查找给定值，插入和删除数据元素等操作实现所蕴涵的元操作的执行次数都与元素的个数有关。
- 插入和删除操作所花费的时间主要用于移动元素。设在第*i*个位置插入新的数据的概率为 $p_i$ ，则插入一个数据元素所做的平均移动次数为

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

IPL

第二章 线性表

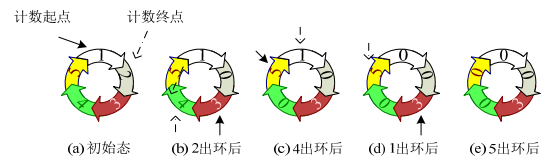
42

## 线性表顺序存储结构的特点

- **存储次序**直接反映逻辑次序，可以直接访问第*i*个数据元素；**随机访问特性**
- 所有的存储空间都可以用来存放数据元素；**存储密度高**
- 插入和删除操作很不方便，每插入或删除一个数据元素，都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；**时间复杂度为 $O(n)$** 。
- 预分配数组空间时，需要给出数组存储单元的个数，这个值只能根据不成系统的情况估算，可能出现因空间估算过大而在随后的某个操作中重新分配存储空间。

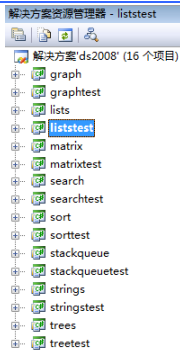
## 【例2.3】以顺序表求解约瑟夫环问题的改进算法

- ◆ 例2.2中的算法多次使用删除操作，即每当一个元素出环时，删除表中相应位置的数据元素，这时必须移动其他元素，操作的时间复杂度高。不使用删除操作，而是将应出环元素相应位置的值设为**空值标志**，并且在计时时跳过值为空的单元。



## 代码组织：类，项目（命名空间）

- ◆ SequencedList类的源代码编辑在SequencedList.cs文件中；该类及其他**自定义线性表类**都实现在名为lists的**类库型项目**中。
- ◆ 它们及后续章节将介绍的其他基础类，如栈、二叉树等，**统一**定义在**DSAGL命名空间**，而使用这些基础类的测试与应用类则各自独立定义和实现在相应的项目和命名空间中（**xxxtest**）。
- ◆ 一般需要：1）添加引用；2）加上using DSAGL指令。



## 2.3 线性表的链式存储结构

- ◆ 链式存储结构是指将元素分别存放在一个个结点（Node）中，结点由**数据域**和**链域**组成，**链**指向其后继结点。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。链表的结点个数称为链表的长度。
- ◆ 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

### 2.3.2 单向链表

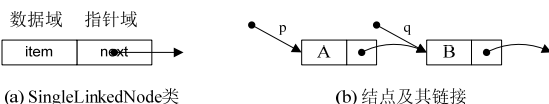
### 2.3.3 单向循环链表

### 2.3.4 双向链表

### 2.3.5 双向循环链表

## 2.3.1 线性链表的结点结构

- ◆ 在线性表的链式存储结构中，元素分别存放在一个个**结点**中。结点由**数据域**和**指针域**组成，指针指向其他结点。线性表数据元素之间的逻辑次序就由结点间的指针来实现。



(a) SingleLinkedList类

(b) 结点及其链接

- ◆ **C#**中用类类型（class type）定义的类型是一种引用类型（reference type），它保存的内容是某个对象的引用，即地址，因此可以定义“自引用的类”（self referential class）表示链表的结点结构。

## 声明结点类：一种自引用类

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

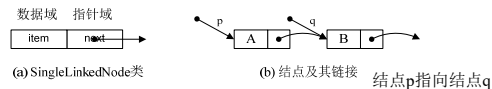
- ◆ 一个**结点**就是SingleLinkedListNode类型的一个**实例**。
- ◆ 该类成员next是SingleLinkedListNode类型的变量（实例变量），它保留的是某个对象的引用，实际起着地址的作用，称为**链（link）**。
- ◆ **C#**的类类型是引用类型，通过用**自引用类型的链域**将**结点（对象）**链接起来，实现结点间的链接以及多种动态的数据结构，如链表和二叉树等。



## 创建并使用结点对象

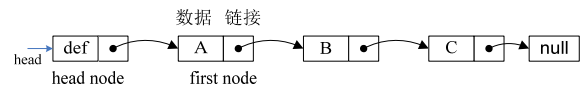
```
SingleLinkedListNode<string> p;
//声明p是SingleLinkedListNode<string>类型的变量
p = new SingleLinkedListNode<string>();
//创建SingleLinkedListNode<string>类的一个对象，由p引用
```

- ◆ 创建和维护动态数据结构需要**动态内存分配**（Dynamic Memory Allocation）。
- ◆ C#使用 **new** 操作符创建对象并为之分配内存。
- ◆ 由p引用对象中两个成员变量的格式为p.item和p.next。
- ◆ 通过下述语句可将p、q两个对象链接起来：**p.next = q;**



## 2.3.2 单向链表

- ◆ **单向链表**（single linked list）：每个结点只有一个链的线性链表。单向链表各结点的链指向其后继结点。
- ◆ 在单向链表中，从**头指针head**开始找到**头结点**，头结点的链指向第一个数据结点（**First Node**），沿链表的方向前进，就可以顺序访问链表中的每个结点。



## 单向链表的结点类代码

```
public class SingleLinkedListNode<T> {
    private T item;           //存放结点值
    SingleLinkedListNode<T> next; //后继结点的引用
    //构造值为k的结点
    public SingleLinkedListNode(T k) {
        item = k; next = null;
    }
    //无参数时构造值为缺省值的结点
    public SingleLinkedListNode() {next = null; }
    ..... }
}
```

## 单向链表的结点类(II)

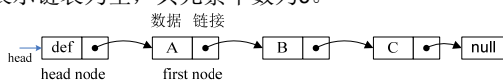
```
public T Item{
    get{ return item; }
    set{ item = value; }
}
public SingleLinkedListNode<T> Next{
    get{ return next; }
    set{ next = value; }
}
```

提供公有属性Next和Item允许外部模块来间接访问next和item

## 单向链表类

```
public class SingleLinkedList<T> {
    private SingleLinkedListNode<T> head; //头指针
    public SingleLinkedListNode<T> Head {
        get{ return head; } set{ head = value; } }
    ..... }
}
```

- ◆ SingleLinkedList类的一个实例（对象）表示一条具体的**单向链表**，类的（实例）成员head作为链表的**头指针**，指向链表的头结点。当头结点的Next域为null时，表示链表为空，其元素个数为0。



## 单向链表的操作

- ◆ 单向链表的**初始化**，建立单向链表
- ◆ 返回链表的**长度**
- ◆ 判断单向链表**是否为空**
- ◆ 获取或设置**指定位置的数据元素值**
- ◆ **输出**单向链表
- ◆ 在链表的**指定位置插入**数据元素
- ◆ **删除**链表指定位置的数据元素

各种操作作为SingleLinkedList类的属性和方法成员予以实现

### 1) 单向链表的初始化

- ◆ 用SingleLinkedList类的构造方法建立一条链表。

```
// 构造空的单向链表，头结点是个标志结点
public SingleLinkedList() {
    head = new SingleLinkedListNode<T>();
}
//构造由参数f作为第一个数据结点的单向链表
public SingleLinkedList(SingleLinkedListNode<T> f)
: this() {
    head.Next = f;
}
```

TPL

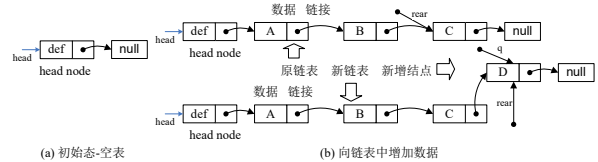
第二章 线性表

55

### 用一个数组的值构造单向链表

```
public SingleLinkedList(T[] a)
```

- ◆ 创建结点对象，依次链入表尾



```
q = new SingleLinkedListNode<T>(k); // 建立结点
rear.Next = q; // q结点链入表尾
rear = q; // 更新rear, 指向新链尾结点
```

TPL

第二章 线性表

56

### 以一个数组的值构造单向链表的代码

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

TPL

第二章 线性表

57

### 2) 返回链表的长度

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

TPL

第二章 线性表

58

### 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head.Next为null时，表示链表为空。

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

TPL

第二章 线性表

59

### 判断单向链表是否已满

- ◆ 动态分配结点内存空间：当有一个新数据元素需要加入链表时，向系统申请一个结点的存储空间，可以认为系统所提供的可用空间是足够的，因此不必判断链表是否已满。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。如果需要在链表类型中实现判断链表是否已满的功能，则可以按下列方式实现：

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

TPL

第二章 线性表

60

#### 4) 获取或设置指定位置的数据元素值

- ◆ 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获取或设置该结点的值。

```
public virtual T this[int i]{
    get{
        int n = 0; // count of elements
        SingleLinkedListNode<T> q = head.Next;
        while (q != null && n != i) {
            n++; q = q.Next; }
        if(q==null) throw new IndexOutOfRangeException();
        return q.Item; }
    set{ int n = 0; // count of elements
        SingleLinkedListNode<T> q = head.Next;
        while (q != null && n != i) {
            n++;q = q.Next; }
        if (q == null)throw new IndexOutOfRangeException();
        q.Item = value; } }
```

#### 5) 输出单向链表

```
public virtual void Show(bool showTypeName) {
    if(showTypeName)
        Console.WriteLine(this.GetType() + ": ");
    SingleLinkedListNode<T> q = head.Next;
    if (q != null) q.Show();
}
```

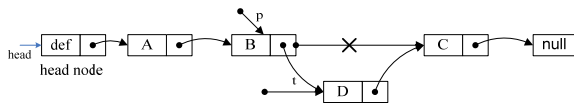
##### SingleLinkedListNode类

```
public void Show() {
    SingleLinkedListNode<T> p = this;
    while(p!=null) {
        Console.WriteLine(p.Item);
        p = p.Next;
        if(p!=null) Console.WriteLine(" -> ");
        else Console.WriteLine(".");
    } Console.WriteLine(); }
```

由结点p到达p的后继结点

#### 6) 插入结点

- ◆ 生成新的结点并插入单向链表中:  
**t = new SingleLinkedListNode<T>(k);**  
**t.Next = p.Next; p.Next = t;**



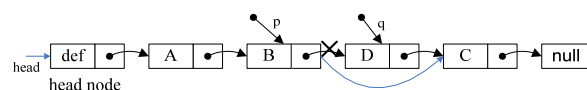
在单向链表中插入结点，  
只要修改相关的几条链，  
而不需移动数据元素。

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

#### 7) 删除结点

- ◆ 在单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。在C#中，该结点所占的存储单元由系统管理，适时自动回收。

- ◆ Remove(int k)和RemoveAt(int i)见讲义。

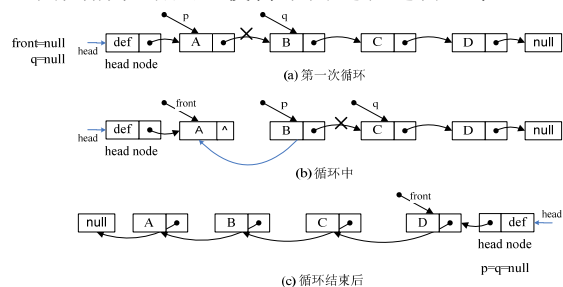


p.Next = q.Next;

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。

#### 8) 单向链表逆转Reverse

- ◆ 已建立单向链表，将各结点的next链改为指向其前驱结点，使得单向链表逆转过来。



```

public virtual void Reverse() {
    SingleLinkedListNode<T> p = head.Next;
    SingleLinkedListNode<T> q = null, front = null;
    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;
}

```

【例2.4】单向链表逆转实现与测试

## 两种存储结构性能的比较

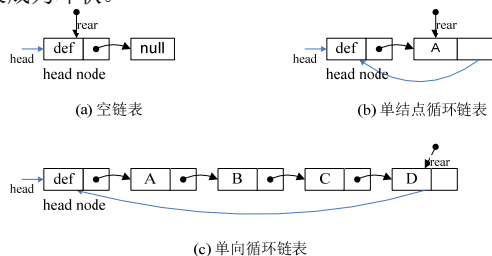
- ◆ **元素的随机访问特性**
  - 顺序表能够直接访问数据元素
  - 链式结构不能直接访问任意指定位置的数据元素
- ◆ **插入和删除操作**
  - 顺序表的插入和删除操作很不方便，有时需要移动大量元素
  - 链表则容易进行插入和删除操作，只要改动相关结点的链即可，不需移动数据元素。
- ◆ **存储密度**
  - 顺序表存储密度高，全部空间都用来存放数据
  - 链表存储密度较低，结点包含数据和指针

## 两种存储结构性能的比较(II)

- ◆ **存储空间的动态利用特性**
  - 顺序表存在使用空间的浪费和频繁的存储空间重分配问题
  - 链表向系统动态申请存储单元
- ◆ **查找和排序**
  - 顺序表和链表都可以采用查找和排序的一些简单算法，如顺序查找、插入排序、选择排序等
  - 顺序表还可以采用多种复杂的查找和排序算法，包括折半查找、快速排序、堆排序等

## 2.3.3 单向循环链表

- ◆ 单向循环链表（circular linked list）：单向链表中，将最后一个结点的链设置为指向链表的头结点，则该链表成为环状。



## 循环链表及其结点

- ◆ 循环链表的结点与普通的单向链表的结点类型相同。（SingleLinkedListNode）
- ◆ **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**一个新类作为循环链表类的实现。

```
public class CircularLinkedList<T>: SingleLinkedList<T>
```

## 单向循环链表的特征

- ◆ 循环链表的所有结点链接成一条环路。
- ◆ 仍用head指向头结点，设置变量rear指向循环链表的最后一个结点，则有 **rear.next==head**。Rear为尾指针。
- ◆ 当**head.next==null**或**head==rear**时，循环链表为空。
- ◆ 当**head.next.next==head**时，循环链表只有一个数据结点。

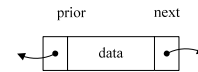
```

public class CircularLinkedList<T>:SingleLinkedList<T> {
    private SingleLinkedListNode<T> rear;
    public override SingleLinkedListNode<T> Rear {
        get { return this.rear; }
    }
    public CircularLinkedList():base() { rear = Head; }
    public override bool Empty{get {return Head==rear;}}
    public override int Count {
        get { int n = 0;
            SingleLinkedListNode<T> p = Head.Next;
            if (p == null) return 0;
            while (p != Head) {
                n++; p = p.Next;
            }
            return n; }
    }
    .....
}

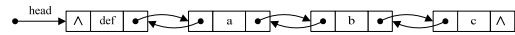
```

### 2.3.4 双向链表

- ◆ 双向链表(doubly linked list): 每个结点有两个(链)成员变量, 其中prior指向前驱结点, next指向后继结点。方便实现既向前又向后的操作。



(a) 双向链表中的结点结构



(b) 双向链表

### 双向链表的结点类

- ◆ DoubleLinkedListNode<T>类的一个实例表示双向链表中的一个结点对象。

```

public class DoubleLinkedListNode<T> {
    private T item; //存放结点值
    private DoubleLinkedListNode<T> prior, next;
    //前驱与后继结点的引用
    public DoubleLinkedListNode(T k) { //构造值为k的结点
        item = k; prior = next = null; }
    //无参数时构造缺省值的结点
    public DoubleLinkedListNode() {
        prior = next = null; } }

```

### 双向链表类

- ◆ DoubleLinkedList<T>类的一个对象表示一条双向链表

```

public class DoubleLinkedList<T> {
    private DoubleLinkedListNode<T> head;
    // 构造空的双向链表
    public DoubleLinkedList() {
        head = new DoubleLinkedListNode<T>();
        //头结点是个标志结点
    } }

```

### 双向链表对象特征

- ◆ 线性表的头结点没有前驱结点, 最后一个元素没有后继结点, 所以有:  
 $\text{head.prior} == \text{null}$   
 $\text{rear.next} == \text{null}$
- ◆ 设p指向双向链表中的某一结点(除尾结点), 则双向链表的本质特征如下:  
 $(\text{p.prior}).\text{next} == \text{p}$   
 $(\text{p.next}).\text{prior} == \text{p}$
- ◆ 双向链表能够沿着链向两个方向移动, 既可以找到后继结点, 也可以找到前驱结点。

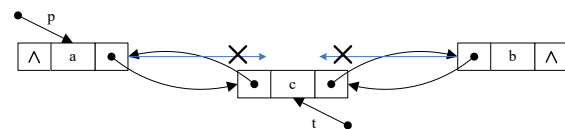
### 在双向链表中插入结点

- ◆ 在非空链表的p结点之后插入t结点, 形成新的链表:

```

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

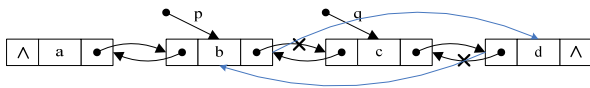
```



## 在双向链表中删除结点

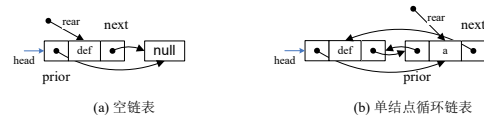
- ◆ 删除给定位置的结点，需要把该结点从链表中退出，并改变相邻结点的链接关系。
- ◆ 在双向链表中删除指定位置结点 $q$ ，设它的前驱结点为 $p$ ：

$p.next = q.next;$   
 $(p.next).prior = p;$



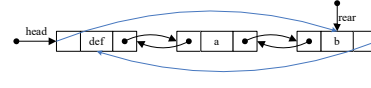
## 双向循环链表

- ◆ 双向循环链表最后一个结点的 $next$ 链指向头结点，而头结点的 $prior$ 链指向最后一个结点
- ◆ 即:  $rear.next == head$  且  $head.prior == rear$



(a) 空链表

(b) 单结点循环链表



(c) 双向循环链表

## 本章学习要点

1. 线性表的**逻辑结构特性**是数据元素之间存在着线性关系，在计算机中表示这种关系的两类不同的存储结构是**顺序存储结构**和**链式存储结构**。用前者表示的线性表简称为顺序表，用后者表示的线性表简称为链表。
2. 熟练掌握这两类存储结构的描述方法，以及线性表的各种基本操作的实现。
3. 能够从时间和空间复杂度的角度综合比较线性表两种存储结构的不同特点及其适用场合。

**强调：加强编程实验。** You haven't really learned something well until you've taught it to a computer. (Don Knuth)

## 作业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： `public SingleLinkedList(SingleLinkedList a);`
2. 返回第 $i$ 个结点的值。
3. 求各结点的数值之和。
4. 查找值为 $k$ 的节点，返回值为`bool`类型。
5. 删除值为 $x$ 的节点。
6. 将两条单向链表连接起来，形成一条单向链表。
- 2.3 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();`

## 实习2

- ◆ 实验目的  
理解线性表的基本概念，熟练运用C#自引用类的方式实现线性链表的基本操作。
- ◆ 题意：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，一些操作的实现需判断链表是否是单结点的情况。