

**ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ  
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ  
«САМАРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ  
УНИВЕРСИТЕТ имени академика С. П. КОРОЛЕВА»  
(САМАРСКИЙ УНИВЕРСИТЕТ)**

**ЛИНЕЙНЫЕ ДИНАМИЧЕСКИЕ  
СТРУКТУРЫ ДАННЫХ В C#. СПИСКИ**

**Самара 2017**

МИНИСТЕРСТВО ОБРАЗОВАНИЯ И НАУКИ РОССИЙСКОЙ ФЕДЕРАЦИИ

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ ОБРАЗОВАТЕЛЬНОЕ  
УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ  
«САМАРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ  
УНИВЕРСИТЕТ имени академика С. П. КОРОЛЕВА»  
(САМАРСКИЙ УНИВЕРСИТЕТ)

# ЛИНЕЙНЫЕ ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ В C#. СПИСКИ

Составитель *Е.В. Симонова*

САМАРА  
Издательство Самарского университета  
2017

УДК 519.876.5

ББК 22.18я73

*Составитель Е.В. Симонова*

Рецензент: канд. техн. наук, доц. Л.С. Зеленко

**Линейные динамические структуры данных в С#. Списки:**  
[Электронный ресурс]: метод. указания / *Е.В. Симонова*. – Самара: Изд-во Самарского университета, 2017. – 48 с. : ил. Электрон. текстовые и граф. дан. ( Кбайт).- 1 эл. опт. диск (CD-ROM)

Методические указания содержат теоретические сведения по организации списковых структур, большое количество программных фрагментов, реализующих алгоритмы обработки списков, а также варианты заданий для выполнения лабораторных и самостоятельных работ.

Предназначены для студентов направления 09.03.01 – «Информатика и вычислительная техника» в качестве методических указаний по курсу «Программирование».

Подготовлены на кафедре информационных систем и технологий.

УДК 519.876.5

ББК 22.18я73

© Самарский университет, 2017

## ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ.....	5
ВВЕДЕНИЕ.....	5
1. ОРГАНИЗАЦИЯ УПРАВЛЯЕМОЙ КУЧИ.....	6
2. ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ.....	13
3. ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ.....	39
4. КОНТРОЛЬНЫЕ ВОПРОСЫ.....	39
5. ИНДИВИДУАЛЬНЫЕ ЗАДАНИЯ.....	41
ЗАКЛЮЧЕНИЕ .....	44
БИБЛИОГРАФИЧЕСКИЙ СПИСОК.....	44

## **ПРЕДИСЛОВИЕ**

В методических указаниях описаны структуры данных и алгоритмы, которые широко используются при решении разнообразных задач в широком спектре предметных областей.

Подробно рассматриваются особенности организации линейных динамических структур данных (односвязных, двусвязных списков, мультисписков). Особое внимание уделено вопросам управления динамической памятью и эффективного использования ее возможностей при работе с динамическими структурами данных в программах пользователей.

Примеры программ, описывающих алгоритмы обработки структур данных различных типов, реализованы на языке C# версии 3.0 с использованием Visual Studio 2008 .NET Framework 3.5.

Методические указания предназначены для студентов, обучающихся по направлению 09.03.01 – Информатика и вычислительная техника.

Содержание методических указаний соответствует разделам рабочей программы по дисциплине «Программирование» федерального компонента ГОС подготовки бакалавров по направлению 09.03.01 – Информатика и вычислительная техника.

## **ВВЕДЕНИЕ**

Цель лабораторной работы – изучение теоретических основ построения и методов программной реализации линейных динамических структур данных – списков. Деревья представляют собой иерархическую структуру некоторого множества объектов. Древовидные структуры широко используются для организации хранения информации в системах управления базами данных и для представления синтаксических структур в компиляторах программ.

# 1 ОРГАНИЗАЦИЯ УПРАВЛЯЕМОЙ КУЧИ

## 1.1 Создание объекта ссылочного типа

Хотя стек обеспечивает очень высокую производительность, правило вложенности времени жизни переменных во многих случаях слишком ограничено. Часто возникает необходимость использовать метод, чтобы выделить память для хранения некоторых данных и сохранить доступ к ним после завершения работы метода. Это возможно, если память для переменных ссылочных типов явным образом запрашивается операцией *new*. Для размещения переменных при этом используется управляемая куча (*managed heap*).

**Операция *new*.** Служит для создания нового объекта. Формат операции:

*new* тип ( [аргументы] )

С помощью этой операции можно создавать объекты как ссылочных, так и значимых типов, например:

```
int i = new int();           // создание объекта значимого типа, аналогично int i = 0
```

```
using System;
namespace MyClass
{
    class MyClass
    {
        public int a;
        public MyClass()           // конструктор по умолчанию
        {
            a = 100;
        }
        public MyClass( int x)     // конструктор с параметрами
        {
            a = x;
        }
        public int A
        { get
          { return a; }
        }
    }
}

class Demo
```

```

{
    static void Main()
    {
        MyClass ob = new MyClass(); // создание объекта ссылочного типа, a = 100
        Console.WriteLine(ob.A); // a = 100
        MyClass ob1; // объявление ссылки
        ob1 = ob; // установка ссылки на ранее созданный объект
        Console.WriteLine(ob1.A); // a = 100, (рис. 8)
        MyClass oc = new MyClass(50); // создание второго объекта ссылочного
                                     // типа, a = 50
        ob1 = oc; // переустановка ссылки на второй объект
        Console.WriteLine(ob1.A); // a = 50, (рис. 9)
    }
}
}

```

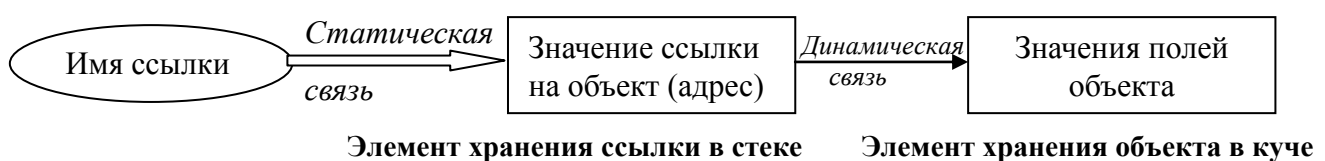
Объекты ссылочного типа формируются только с помощью операции *new*, а переменные значимого типа чаще всего формируются с помощью объявления.

Для **создания объекта ссылочного типа** необходимо, прежде всего, объявить ссылку. Память для элемента хранения ссылки (4 байта) выделяется в стеке. По умолчанию ссылка инициализируется значением *null*.

При выполнении операции *new* производятся следующие действия:

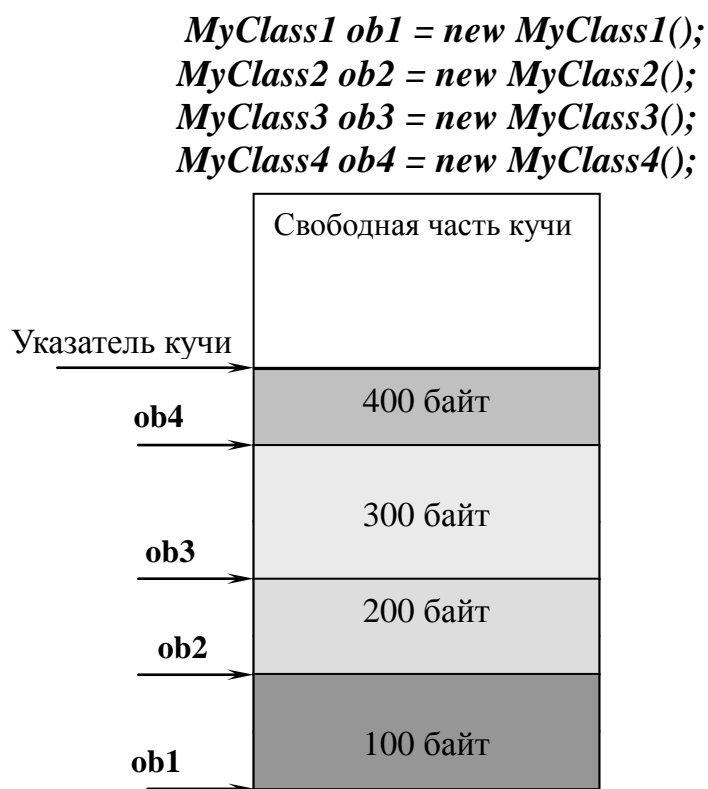
- ◆ выделяется необходимый объем памяти (для ссылочных типов в куче, для значимых в стеке);
- ◆ значение ссылки устанавливается равным адресу выделенной области памяти;
- ◆ вызывается конструктор по умолчанию, т.е. метод, с помощью которого инициализируется объект. Переменной значимого типа присваивается значение по умолчанию, которое равно нулю соответствующего типа. Для ссылочных типов конструктор по умолчанию инициализирует значениями по умолчанию все поля объекта.

При идентификации с помощью ссылки существует статическая связь между именем ссылки и элементом хранения ссылки. Между элементом хранения ссылки и тем объектом, на который она указывает, устанавливается динамическая связь (рис. 1).



**Рис. 1. Идентификация объектов с использованием ссылки**

Чтобы найти местоположение в куче для нового объекта, исполняющая система .NET реализует метод “первого подходящего”: просматривает кучу, находит первый встретившийся непрерывный блок, размер которого не меньше требуемого, и занимает необходимое количество байтов. Указатель кучи установлен на начало следующего свободного блока памяти и корректируется по мере добавления в кучу новых объектов. В отличие от стека, память в куче выделяется по направлению от младших адресов к старшим. Распределение области памяти кучи после выполнения запросов на выделение памяти показано на рис. 2.



**Рис. 2. Распределение области памяти кучи после выполнения запросов на выделение памяти**

Таким образом, процесс создания ссылочной переменной более сложен, чем переменной значимого типа, а потому требует некоторых накладных расходов, отражающихся на производительности. Исполняющая система должна поддерживать информацию о состоянии кучи, и эта информация должна обновляться всякий раз, когда в кучу добавляются новые данные.

## 1.2 Действия над ссылками

### *Присваивание ссылок.*



Одной переменной ссылочного типа можно **присвоить** значение другой переменной того же типа. При этом две различные переменные будут ссылаться на один и тот же объект в куче и, соответственно, открывать доступ к полям одного и того же объекта. Результат выполнения этой операции иллюстрируется рис. 3.

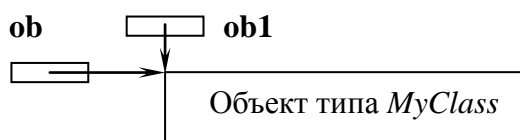


Рис. 3. Присваивание ссылок

### **Сравнение ссылок.**

Ссылки одного и того же типа можно **сравнивать** между собой на равенство и неравенство. Две ссылки считаются равными, если они указывают на один и тот же объект или обе никуда не указывают (обе равны *null*). Неравные ссылки указывают на разные объекты или одна из них никуда не указывает. Ссылку можно сравнивать с константой *null*, чтобы узнать, ссылается ли она на конкретный объект. Если ссылка установлена на некоторый объект, можно обращаться к этому объекту и его полям.

### **Доступ к объекту и его полям через ссылку.**

Доступ к объекту и его полям через ссылку осуществляется в несколько шагов:

- ◆ **имя ссылки** – ссылка используется для получения адреса того объекта, с которым она связана,
- ◆ **имя ссылки.имя поля** – ссылка открывает доступ к конкретному полю объекта.

### **Переустановка ссылки.**

Т.к. элемент хранения ссылки содержит адрес объекта, в процессе выполнения программы одна и та же ссылка может открывать **доступ к различным объектам одного и того же типа** (и полям этих объектов).

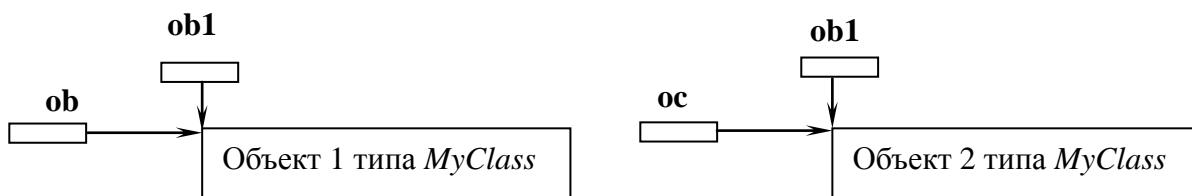


Рис. 4. Переустановка ссылки ob1

### 1.3 Сборка мусора

Когда ссылочная переменная выходит из своей области видимости, она удаляется из стека, но данные, на которые она ссылается, остаются в куче до тех пор, пока либо не завершится программа, либо не останется ни одной ссылочной переменной, указывающей на эти данные. Данные, на которые в программе нет ни одной ссылки, называются **мусором**. Таким образом, время жизни объектов, размещенных в куче, не связано с областями видимости находящихся в стеке переменных-ссылок, которые указывают на них.

В .NET отсутствует операция освобождения памяти. Функции по освобождению памяти выполняет специальная системная программа – **сборщик мусора**. Сборщик мусора уничтожает объекты, располагающиеся в стеке, каждый раз, когда соответствующая переменная выходит из области видимости или завершается работа метода.

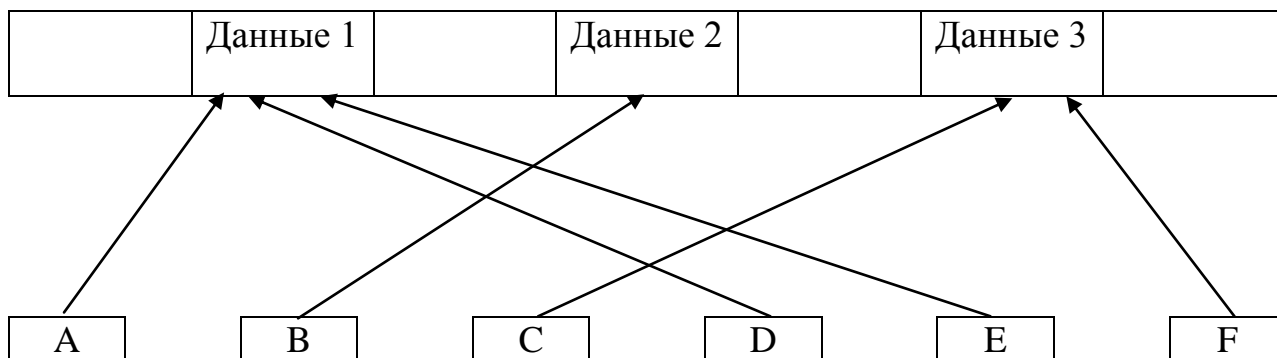
Объекты в куче, на которые нет ссылок, также удаляются сборщиком мусора. Сразу после этого получается, что оставшиеся объекты разбросаны по куче вперемишку со свободными участками памяти. Разбиение всей доступной памяти области кучи на большое количество свободных блоков относительно небольшого размера называется **фрагментацией** (рис. 5).



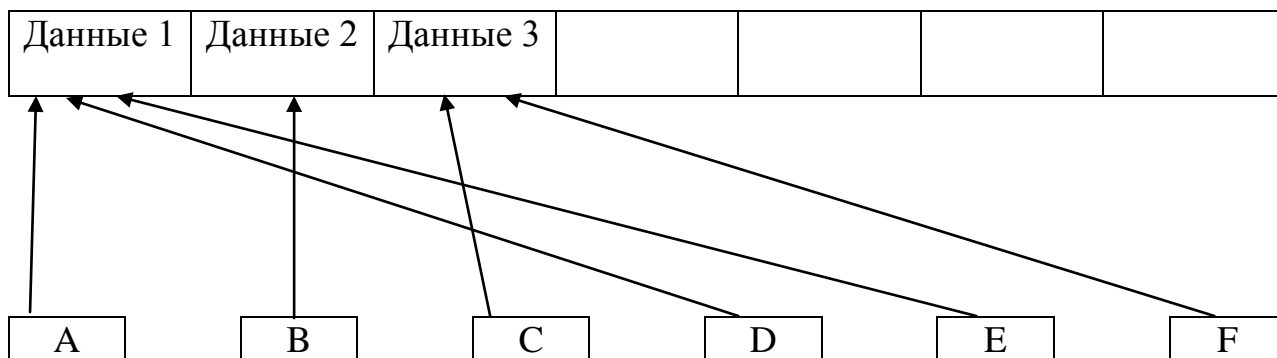
Рис. 5. Фрагментация кучи

Если бы управляемая куча оставалась в таком состоянии, то выделение памяти для новых объектов было бы затруднено, т.к. исполняющей системе пришлось бы каждый раз искать в куче свободный участок достаточного размера для размещения нового объекта. Но, как только сборщик мусора освобождает все объекты, которые возможно, он тут же сжимает кучу,

перемещая оставшиеся объекты в один непрерывный блок памяти, расположенный в начале кучи. Когда объект перемещается в куче, все ссылки, указывающие на него, должны быть обновлены в соответствии с их новыми адресами. Физическое передвижение блоков данных из одних областей памяти в другие с целью сбора всех свободных блоков в один большой блок, называется *уплотнением*. Процесс уплотнения памяти кучи показан на рис. 6.



**а. Перед уплотнением**



**б. После уплотнения**

**Рис. 6. Процесс уплотнения кучи**

Простая схема уплотнения заключается в том, что сначала нужно просмотреть все блоки (как заполненные, так и пустые) и вычислить так называемый «адрес передачи» (*forwarding address*) для каждого заполненного блока. Адрес передачи блока – это его текущая позиция минус сумма всего пустого пространства ниже его, т.е. позиция, в которую в итоге необходимо переместить этот блок. Адрес передачи блока вычисляется следующим образом. Когда просматриваются все блоки в направлении от нижних адресов пространства памяти к верхним адресам, нужно суммировать размер встречающихся свободных блоков и вычитать этот размер из адреса каждого встречающегося блока.

После вычисления адресов передачи сборщик мусора анализирует все ссылки. Каждая ссылка на некоторый блок заменяется адресом передачи, найденным для данного блока. Наконец, все заполненные блоки перемещаются по их адресам передачи. Перемещение заполненных блоков занимает время, пропорциональное объему используемой памяти кучи.

## **1.4. Контрольные вопросы к разделу 1**

1. Для чего предназначена область оперативной памяти «управляемая куча»? Как выполняется создание объекта ссылочного типа с помощью операции *new*?

2. Какие действия над ссылками Вы знаете?

3. Как выполняется связывание идентификатора объекта и его элемента хранения при идентификации с использованием ссылки?

4. Что такое «мусор»? Как выполняется сборка мусора?

5. Для чего используется уплотнение памяти? Как функционирует механизм уплотнения памяти?

## **1.5. Упражнения к разделу 1**

1. Задача о простых дробях

Реализовать операции сложения и умножения двух простых дробей.

Выполнить реализацию в виде класса.

2. Задача о записной книжке.

На каждой странице записной книжки указаны фамилии людей, начинающиеся с одной и той же буквы (английского алфавита), и телефоны этих людей.

Каждая буква занимает одну страницу. На каждой странице содержится одинаковое количество записей.

По заданной фамилии распечатать номер телефона. Выполнить реализацию в виде классов.

### 3. Задача об итогах сессии.

Результаты сессии на студенческом курсе представлены в виде сводных ведомостей по группам. Ведомость по студенческой группе содержит перечень фамилий студентов группы, названий сданных предметов и оценок по этим предметам.

В каждой группе содержится одинаковое количество студентов, студенты всех групп сдают одинаковое количество предметов, но предметы в различных группах могут быть различными.

Для курса распечатать фамилии студентов, имеющих задолженности, с указанием названий предметов, по которым задолженность. Выполнить реализацию в виде классов.

## 2. ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ

### 2.1. Метод вычисляемого и хранимого адреса.

#### Последовательная и связанная организация памяти

Существует два способа организации памяти: *последовательная* и *связанная организация* и, соответственно, два метода доступа к объектам: *метод вычисляемого адреса* и *метод хранимого адреса*. Методу *вычисляемого адреса* соответствует *последовательная организация памяти*. При такой организации объекты размещаются в смежных последовательно расположенных ячейках памяти. Адреса объектов вычисляются от начального адреса стека или кучи с учетом размеров элементов хранения объектов.

Последовательную организацию памяти можно реализовать как в стеке, так и в куче. Примером структуры данных с последовательной организацией является массив.

Массивы в любом языке программирования имеют несколько общих свойств:

- ◆ содержимое массива хранится в непрерывном блоке памяти,
- ◆ все элементы массива должны быть одного типа или одного производного типа, поэтому массивы называют однородными структурами данных,
- ◆ к элементам массива возможен непосредственный доступ.

Наиболее общие операции, выполняемые с массивом:

- ◆ размещение массива,
- ◆ доступ к элементам массива.

В C#, когда объявляется массив или любая другая переменная ссылочного типа, она имеет значение *null*. Следующая строка кода создает переменную с именем *ArrayName*, равную *null*:

```
ArrayType[] ArrayName;
```

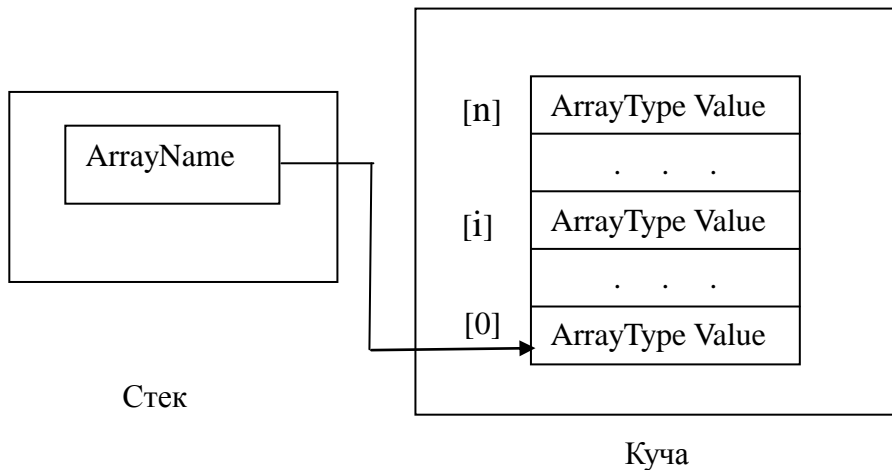
Перед началом использования массива необходимо создать экземпляр массива, который может хранить заданное число элементов:

```
ArrayName = new ArrayType[10];
```

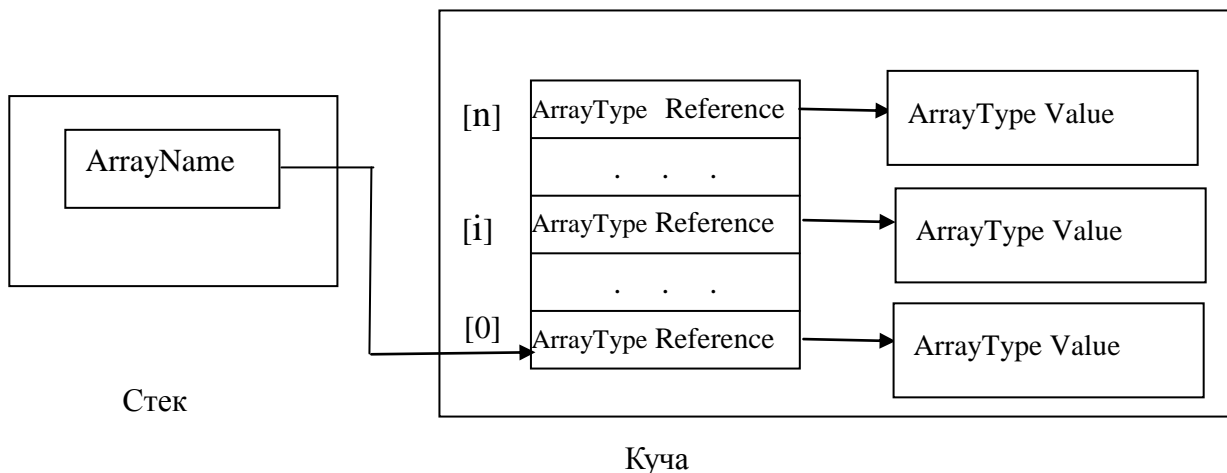
В более общем виде объявление массива и выделение памяти для размещения его элемента хранения можно объединить:

```
ArrayType[] ArrayName = new ArrayType [AllocationSize];
```

Эта операция распределяет непрерывный блок памяти в куче, достаточный для размещения *AllocationSize* элементов типа *ArrayType*. Если *ArrayType* является типом значений, будет создано *AllocationSize* значений типа *ArrayType* (рис. 7). Если *ArrayType* является ссылочным типом, то создается *AllocationSize* ссылок на элементы типа *ArrayType* (рис. 8).



**Рис. 7. Размещение массива: *ArrayType* – значимый тип**



**Рис. 8. Размещение массива: *ArrayType* –ссылочный тип**

Адрес  $A_i$   $i$ -го элемента массива  $ArrayName[i]$  компилятор вычисляет по формуле  $A_i = A_0 + sizeof( ArrayType ) * i, i \geq 0$ , где *sizeof* – размер элемента хранения (в байтах) данных типа *ArrayType*.

В C# реализованы следующие виды массивов:

- ◆ *System.Array* – массив, не допускающий автоматического изменения размера и позволяющий хранить данные одного типа.
- ◆ *System.Collection.ArrayList* – динамический массив, не поддерживающий безопасность типов. В одном и том же массиве

могут храниться элементы различных типов. *ArrayList* поддерживает внутренний массив элементов типа *object* и обеспечивает автоматическое изменение размера массива при увеличении количества элементов, добавляемых к *ArrayList*. При чтении значения из *ArrayList* необходимо явно приводить тип элемента к типу переменной, в которой будет храниться это значение.

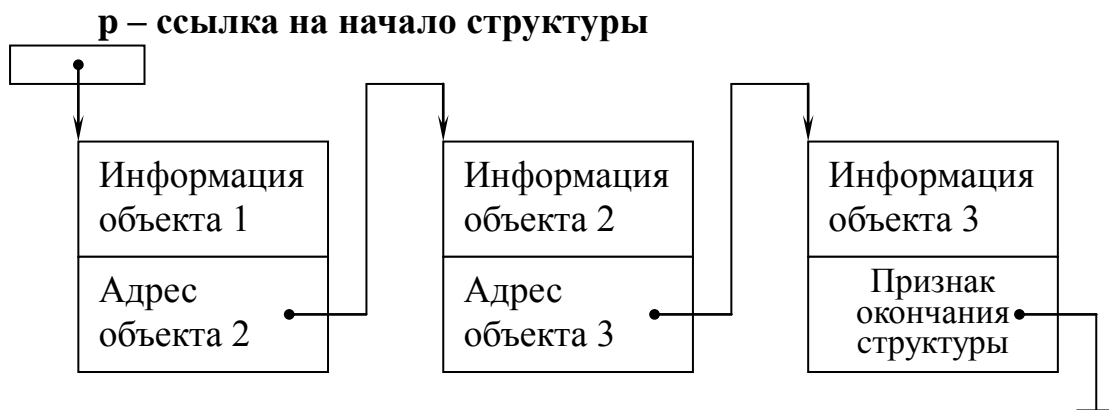
- ◆ *System.Collection.Generic.List* – динамический массив, поддерживающий безопасность типов. В классе *List* реализована оболочка для массива, обеспечивающая доступ к элементам для чтения и записи, а также автоматическое изменение размера массива при необходимости.

```
List<int> IntList = new List<int>();  
IntList.Add( 1 );  
...  
IntList[2] = 10;  
Console.WriteLine(IntList[5]);
```

Достоинством последовательной организации является простота доступа к объектам по имени, а недостатком – накладные расходы при модификации структур объектов.

Согласно *методу хранимого адреса*, адреса объектов не вычисляются, а хранятся в ссылках на эти объекты. Для доступа к объекту сначала необходимо получить ссылку на него, а затем обратиться к полям объекта. Каждый объект имеет возможность хранить в виде ссылок связи с другими объектами, с которыми он взаимодействует в программе. Для реализации этой возможности необходимо ввести в структуру объекта специальные поля, называемые полями связи или ссылочными полями для хранения связей со смежными объектами, что соответствует ***связанной организации памяти***. Графическая иллюстрация структур связанной организации памяти использует прямоугольники для изображения элементов хранения объектов и стрелки для изображения связей (ссылок) между объектами. Необходима специальная ссылка, которая бы определяла местонахождение начального объекта связанной структуры. На рис. 9 приведен пример графической иллюстрации связанной организации структуры из трех объектов.



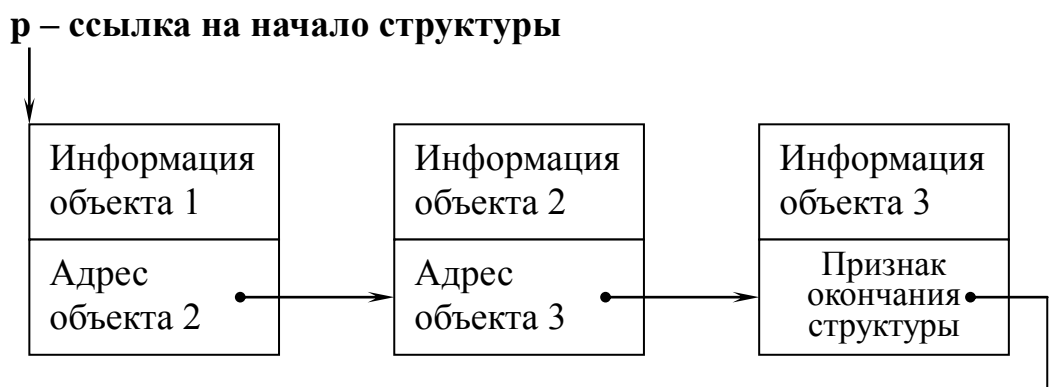


**Рис. 9. Графическое представление связанной организации памяти**

В полях “информация объекта” находятся поля данных объекта, а в полях “адрес объекта” – ссылочные поля для хранения связей. Доступ к объекту 1 открывает ссылка *p*. Доступ к объекту 2 возможен только через объект 1, а к объекту 3 – через объект 2 с использованием соответствующих полей связи.

На рис. 10 приведено упрощенное графическое представление связанной организации памяти.

Достоинством связанной организации памяти является удобство модификации структур, т.к. в них соседние объекты могут располагаться в физически несмежных областях памяти. Необязательно сразу создавать структуру максимального размера. Включение / исключение объектов можно выполнять в процессе работы программы, что не потребует “раздвигать” или “сжимать” структуру за счет копирования информации. Однако “платой” за использование гибкой и эффективной связанной организации памяти являются дополнительные затраты памяти для хранения адресов соседних объектов и более сложный доступ к полям объектов.



**Рис. 10. Упрощенное графическое представление связанной организации памяти**

Метод хранимого адреса и связанная организация памяти эффективно используются для конструирования динамических структур данных.

## 2.2. Понятие динамической структуры данных

*Динамической структурой* называется множество объектов, состав и взаимное расположение которых в процессе выполнения программы может динамически изменяться.

Операции по модификации динамических структур:

- ◆ создание / разрушение структуры,
- ◆ включение объектов в структуру / исключение объектов из структуры,
- ◆ выделение подмножества объектов структуры по определенным признакам,
- ◆ объединение нескольких подмножеств объектов в определенном порядке в единую структуру.

Если на множестве объектов определено отношение порядка, различают линейные и нелинейные структуры данных.

## 2.3. Линейные динамические структуры данных (связанные списки)

*Линейной динамической структурой (связанным списком)* называется множество объектов (элементов, узлов)  $S = \{s_i\}$ ,  $i=1, \dots, n$ , на котором определены отношения предшествования / следования, причем для любого объекта  $s_i$ ,  $i=2, \dots, n-1$  существует единственный “предшественник”  $s_{i-1}$  и единственный “последователь”  $s_{i+1}$ . Объект  $s_1$  имеет последователя, но не имеет предшественника и является первым элементом списка, объект  $s_n$  имеет предшественника, но не имеет последователя и является “хвостом” списка. Ситуация  $n=0$  определяет особое состояние: “список пуст”.

Реализация динамической структуры линейного списка на связанной памяти требует включения в структуру каждого его элемента полей для связи с соседними элементами. В зависимости от того, с каким количеством соседних объектов связан каждый объект в списке, различаются односвязные, двусвязные и многосвязные списки.

### 2.3.1. Основные виды связанных списков

СТЕК (*stack*) – структура, у которой включение / исключение элементов и доступ к элементам производятся на одном конце структуры, называемом верхушкой стека (рис. 11). Для стека характерна дисциплина обслуживания “последним пришел – первым вышел” (*LIFO – Last Input First Output*).

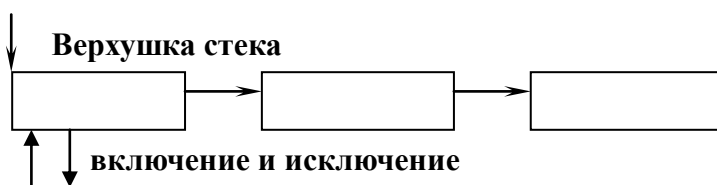


Рис. 11. Структура стека

**ОЧЕРЕДЬ** (*queue*) – структура, у которой включение элемента производится в хвост, а исключение элемента и доступ к элементам выполняются в начале списка (рис. 12). Для очереди характерна дисциплина обслуживания “первым пришел – первым вышел” (*FIFO – First Input First Output*).

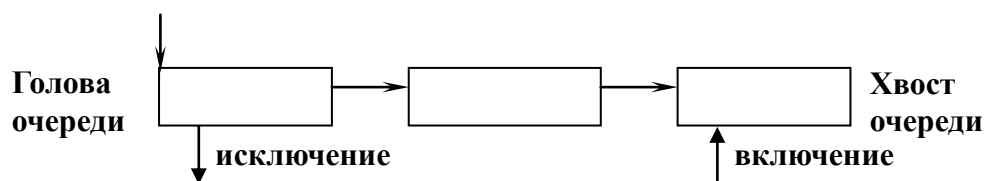


Рис. 12. Структура очереди

**ДЕК** (*deque* – двусторонняя очередь) – структура, у которой операции включения / исключения элементов и доступ к элементам выполняются как в начале, так и в хвосте списка.

**СПИСКИ ПРОИЗВОЛЬНОГО ВИДА** – операции включения / исключения элементов выполняются в любой точке структуры, возможен доступ к произвольному элементу списка.

## 2.4. Односвязные (однонаправленные) списки

Каждый элемент односвязного списка содержит одно или несколько информационных полей и единственное поле связи. Следовательно, одно из полей элемента хранения односвязного списка должно иметь тот же тип, что и сам элемент. Элементом связанного списка не может быть структура, т.к. структура непосредственно содержит свои данные. Например, невозможно объявить следующую структуру:

```
Struct MyType
{
    int info;
    MyType t;
}
```

Внутри объекта типа *MyType* будет содержаться поле *t*, которое, в свою очередь, будет содержать поле *t* и т.д. до бесконечности. Чтобы получить возможность использовать объект в объекте того же типа, необходимо использовать ссылки и объекты классов.

Элемент хранения узла односвязного списка описывается следующим образом:

```
public class Node                                     // Класс «Узел односвязного списка»
{
    private int info;                                // информационное поле узла
    private Node link;                               // поле связи узла

    public int Info {...}                            // свойства
    public Node Link {...}

    public Node (int info)                           // конструкторы
    {
        Info = info;
    }

    public Node (int info, Node link)
    {
        Info = info; Link = link;
    }
}

public class SingleLinkedList                         // Класс «Односвязные списки»
{
    private Node first;                              // ссылка на первый узел списка

    public SingleLinkedList()                        // конструктор: создание пустого списка
    {
        first = null;
    }
    ...
}
```

На рис. 13 представлен пример структуры односвязного списка.

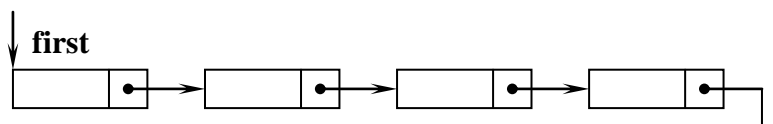


Рис. 13. Структура односвязного списка

На первый элемент списка указывает ссылка *first*. Если список пуст, то *first == null*. Если список не пуст, то к полю любого его элемента (например, первого) можно получить доступ через ссылку.

```
int x = first.Info;           // значение информационного поля первого элемента
Node p = first.Link;        // значение поля связи первого элемента – адрес второго
                             //элемента
```

К полям любого элемента списка, кроме первого, возможен дистанционный доступ.

```
int x = first.Link.Info;     // значение информационного поля второго элемента
```

Дистанционный доступ ко второму узлу списка эквивалентен следующей последовательности операторов:

```
Node p = first.Link;        // установка вспомогательной ссылки на второй узел списка
int x = p.Info;             // значение информационного поля второго узла списка
```

### 2.4.1. Включение узла в начало односвязного списка

Одна из самых простых операций по модификации списка – включение нового узла в его начало в предположении, что список существует (рис. 14). Элемент хранения типа *Node* размещается в куче и ссылка на него присваивается некоторой вспомогательной переменной *p*, затем устанавливается связь между вставленным узлом и списком, после чего ссылка на первый элемент списка получает новое значение.

```
public void InsertBeforeFirst (int data)           // first – ссылка на первый узел списка
{
    Node p = new Node(data);                       // data – значение информационного поля
                                                    // узла списка
                                                    // создание узла списка со значением data

    p.Link = first;                                // установка связи между вставленным узлом и списком
    first = p;                                     // новое значение ссылки на первый узел
}
```

За счет использования конструктора с двумя параметрами решение можно упростить.

```
public void InsertBeforeFirst (int data)
{
    Node p = new Node ( data, first );              // создание узла списка со значением data,
                                                    // указывающего на первый узел

    first = p;                                     // новое значение ссылки на первый узел
}
```

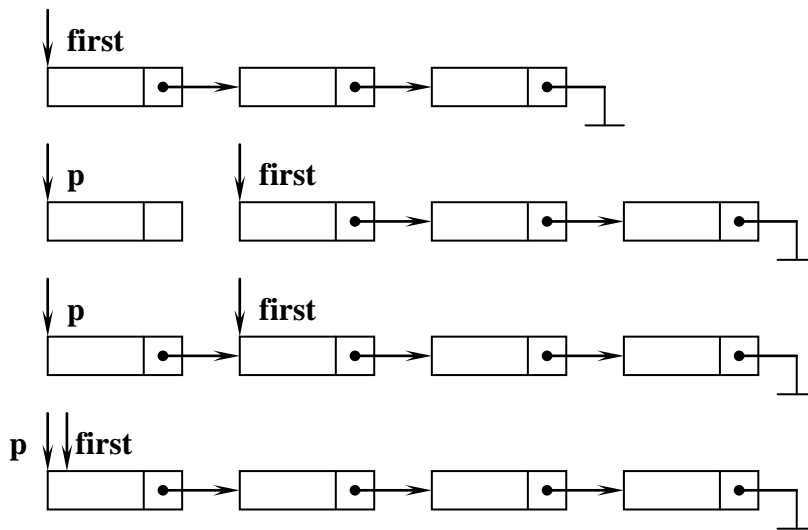


Рис. 14. Включение узла в начало списка

#### 2.4.2. Создание односвязного списка из N узлов: добавление узлов в начало списка

Используя операцию включения элемента в начало списка, можно сформировать список из  $n$  элементов: начиная с пустого списка, следует  $n$  раз выделить память для узлов списка и последовательно добавить узлы в начало списка. Эти операции можно реализовать с помощью любого итерационного цикла. Порядок следования узлов получается обратным, т.е. первым в списке оказывается элемент, который был добавлен последним.

```
public SingleLinkedList(int[ ] dates)           // first – ссылка на первый узел списка
                                                // dates – массив значений информационных полей
{
    first = null;                               // создание пустого списка
    for (int i = 0; i < dates.Length; i++)
    {
        Node p = new Node( dates[i], first );   // вставка узла в начало списка
        first = p;
    }
    { создание списка из трех узлов }
    SingleLinkedList list = new SingleLinkedList( new int [] { 1, 2, 3 } );
}
```

Создание первого узла списка и включение его в пустой список (“перед” несуществующим узлом) выполняется точно так же, как включение в непустой список любого другого узла (см. рис. 15).

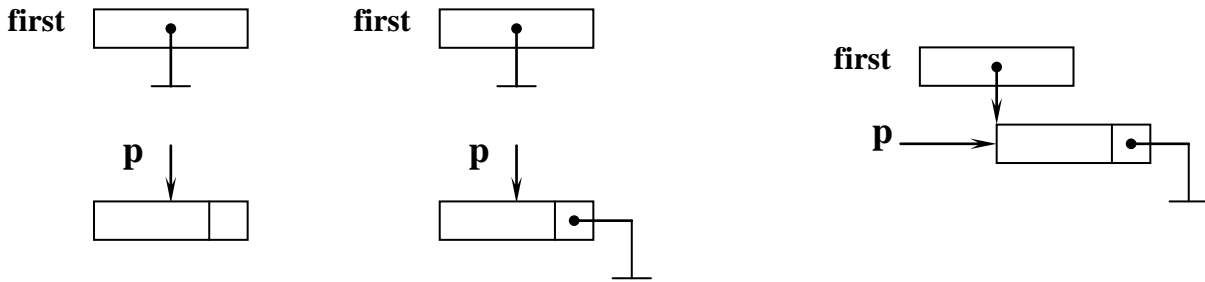


Рис. 15. Включение узла в пустой список

### 2.4.3. Создание односвязного списка из N узлов: добавление узлов в конец списка

Первый узел создается отдельно (т.к. включить узел «за» несуществующим узлом невозможно), а остальные (n-1) узлов создаются и включаются в хвост списка одинаковым образом. При этом удобно использовать вспомогательную ссылку на последний добавленный узел. Значение этой ссылки изменяется в процессе создания списка, значение ссылки на первый узел списка не изменяется после создания первого узла. Порядок следования узлов в списке получается прямым, т.к. в начале списка находится тот узел, который был включен в список первым (рис. 16).

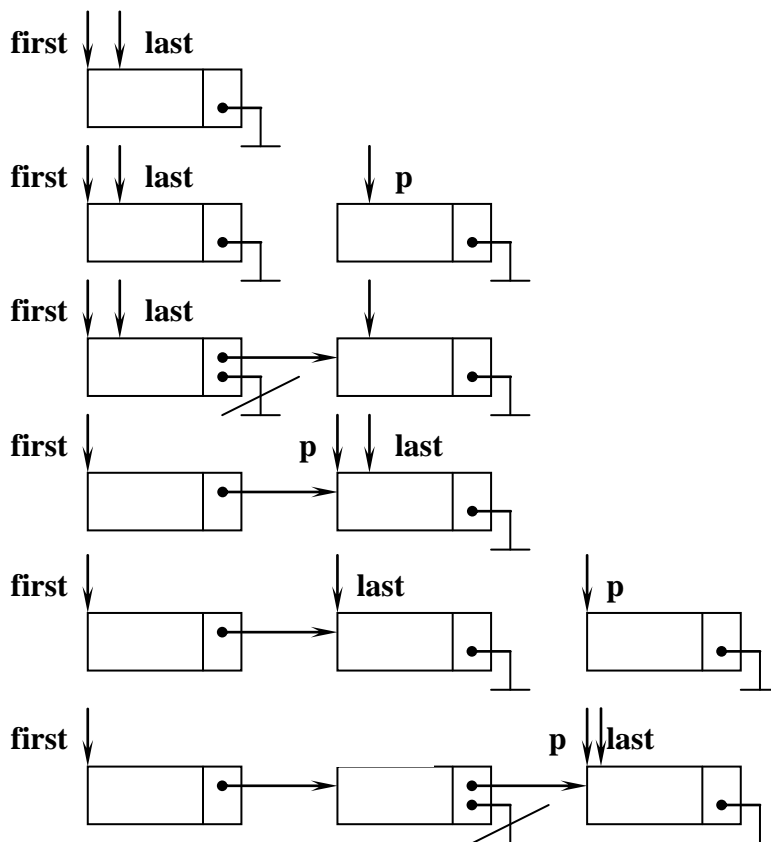


Рис. 16. Включение узла в конец списка

```

public SingleLinkedList(int[ ] dates)           // first – ссылка на первый узел списка
{
    Node p, last;                               // p – ссылка на вставляемый узел,
                                                // last – ссылка на последний узел списка
    if (dates.Length == 0) first = null;        // создание пустого списка
    else
    {
        first = new Node (dates[0], null);      // создание первого узла списка
        last = first;
                                                // цикл включения в список остальных элементов
        for (int i = 1; i < dates.Length; i++)
        {
            Node p = new Node ( dates[i], null ); // создание узла списка
            last.Link = p;                       // вставка нового узла за текущим последним узлом
            last = p;                            // новое значение ссылки на последний узел
        }
    }
}

```

#### 2.4.4. Исключение узла из начала односвязного списка

Для того чтобы исключить из списка первый элемент, необходимо переустановить ссылку на следующий элемент списка. Память, занятую бывшим первым элементом, вернет в кучу сборщик мусора (рис. 17).

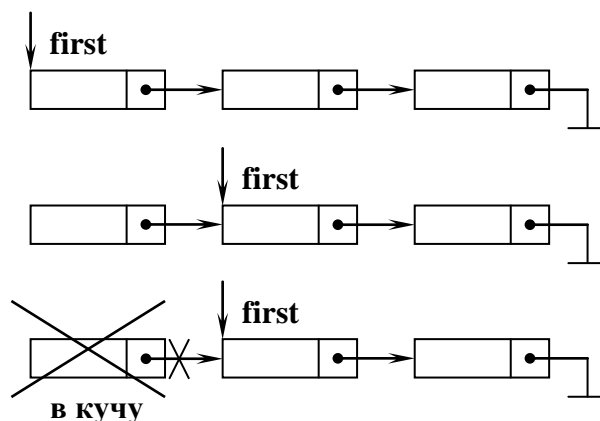


Рис. 17. Исключение узла из начала списка

На исключаемый элемент следует предварительно установить ссылку, если требуется дальнейшее использование этого элемента с целью включения его в другую точку данного списка или в другой список.



## 2.4.5. Переустановка ссылки

Доступ к объектам динамической структуры может быть получен с помощью единственной вспомогательной ссылки, которая будет последовательно изменяться, всякий раз принимая значение адреса соседнего объекта, в направлении стрелки, изображающей связь. Адрес соседнего объекта извлекается из поля связи того элемента списка, на который в текущий момент указывает ссылка, затем полученный адрес присваивается этой ссылке, которая теперь открывает доступ к соседнему элементу списка. Такая операция называется переустановкой ссылки (рис. 18). Операция переустановки ссылки используется, если необходимо единообразно обработать все или несколько следующих подряд элементов списка (для этого следует организовать цикл, включающий операции обработки элемента и переустановки ссылки). При этом последовательность операций переустановки ссылки обеспечивает проход по списку.

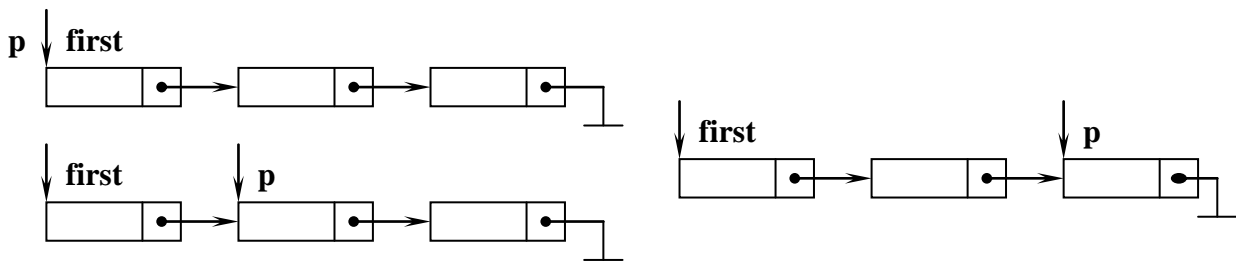


Рис. 18. Переустановка ссылки

```
...  
if first != null // список не пуст?  
{  
    p = first; // установка вспомогательной ссылки на первый узел списка  
    p = p.Link; // переустановка вспомогательной ссылки на второй узел списка  
    Console.WriteLine( p.Info ); // обработка информац. поля второго узла списка  
}
```

## 2.4.6. Поиск узла в односвязном списке по заданному условию

Условием поиска элемента в списке может быть:

- ◆ значение информационного поля элемента,
- ◆ порядковый номер элемента в списке, начиная от первого узла,
- ◆ адрес элемента списка, который хранится в некоторой переменной-ссылке.

Поиск элемента в списке по заданному условию обычно организуется в цикле, включающем операции проверки выполнения условия для текущего элемента, на который указывает ссылка, и переустановки ссылки на соседний элемент списка (т.е. поиск осуществляется в процессе прохода по

списку). Проверка условия связана с вычислением булевских выражений в условных операторах или операторах цикла, использующих доступ к полям объектов через ссылки. Например,

```
if ( p != null && p.Info == значение ) < тело условного оператора >
или
while ( p != null && p.Info != значение ) < тело цикла >.
```

Поиск заканчивается либо при обнаружении элемента списка, соответствующего заданному условию (результатом поиска в этом случае является значение ссылки, установленной на искомый узел), либо при достижении конца списка, если элемент, соответствующий условию поиска, не обнаружен (результатом поиска в этом случае является *null*).

#### 2.4.7. Включение в односвязный список нового узла за тем узлом, на который предварительно установлена ссылка

Для того чтобы включить в список новый элемент за тем элементом, на который предварительно установлена ссылка, необходимо разместить элемент хранения в куче и выполнить последовательность операций, которая иллюстрируется рис. 19. После выполнения операции вставки значение ссылки на первый элемент списка не изменяется. Значение ссылки на тот элемент, за которым выполнена вставка, также не изменяется.

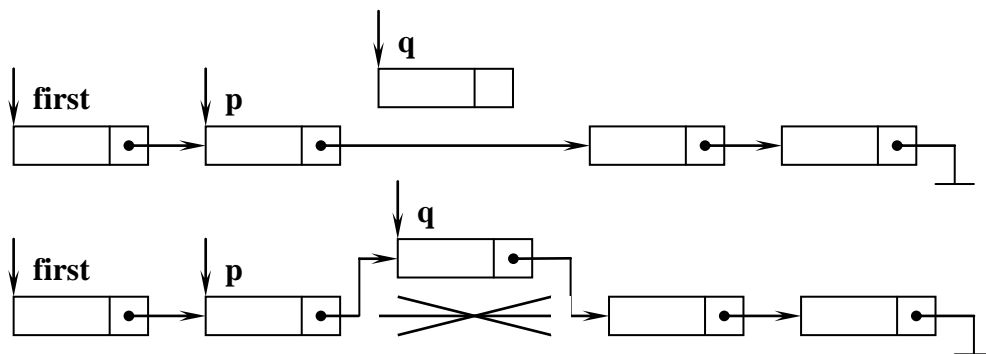


Рис. 19. Включение узла в список за тем узлом, на который предварительно установлена ссылка

```
public void InsertAfter( int data, Node p )
{
    // p – предварительно установленная ссылка,
    // data – значение информационного поля узла
    Node q;
    // q – ссылка на вставляемый узел
    if ( p != null )
        // ссылка p действительно установлена?
    {
        Node q = new Node ( data );
        // создание нового узла списка для вставки
```

```

q.Link = p.Link; // заполнение поля связи нового узла
p.Link = q; // изменение поля связи того узла, за которым вставлен новый узел
}
}

```

### 2.4.8. Исключение из односвязного списка узла за тем узлом, на который предварительно установлена ссылка

Последовательность операций исключения иллюстрируется рис. 20. Память, занятую исключаемым элементом, вернет в кучу сборщик мусора. На исключаемый элемент следует предварительно установить ссылку, если требуется дальнейшее использование этого элемента с целью включения его в другую точку данного списка или в другой список. После выполнения операции исключения значение ссылки на первый элемент списка не изменяется. Значение ссылки на тот элемент, за которым выполнено исключение, также не изменяется.

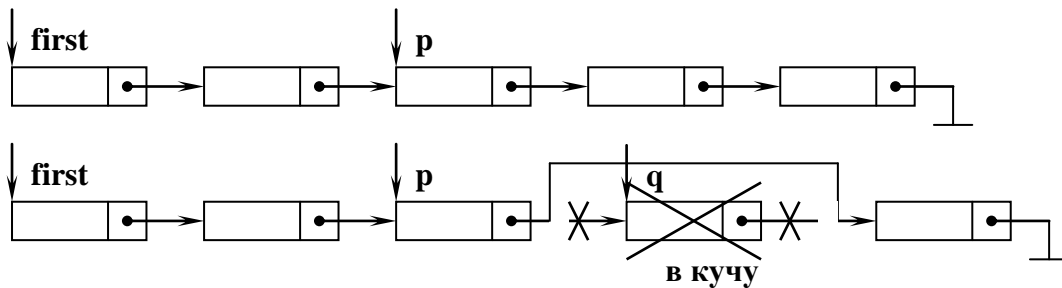


Рис. 20. Исключение узла за тем узлом, на который предварительно установлена ссылка

```

public void DeleteRightElement( Node p)
{
    // p – предварительно установленная ссылка
    Node q; // q – ссылка на исключаемый узел
    if ( p != null && p.Link != null ) // ссылка p действительно установлена?
        // ссылка p не указывает на последний узел в списке?
    {
        q = p.Link; // установить ссылку q на узел, следующий за элементом p
        p.Link = q.Link; // изменить поле связи узла, за которым выполняется исключение
    }
}

```

### 2.4.9. Исключение из односвязного списка узла, на который предварительно установлена ссылка

Для исключения самого указанного элемента необходима предварительная установка ссылки на элемент, предшествующий заданному,

т.к. в этом случае с целью сохранения структуры списка у узла-предшественника должна быть изменена связь (см. рис. 21). Для установки ссылки на узел, предшествующий данному узлу, необходим проход от начала списка до узла-предшественника, т.к. в элементе односвязного линейного списка нет ссылки на предыдущий узел. Если не предполагается дальнейшее использование исключенного узла, ссылку на него следует сделать равной *null*, чтобы сборщик мусора мог вернуть в кучу область памяти, занятую элементом хранения исключенного узла.

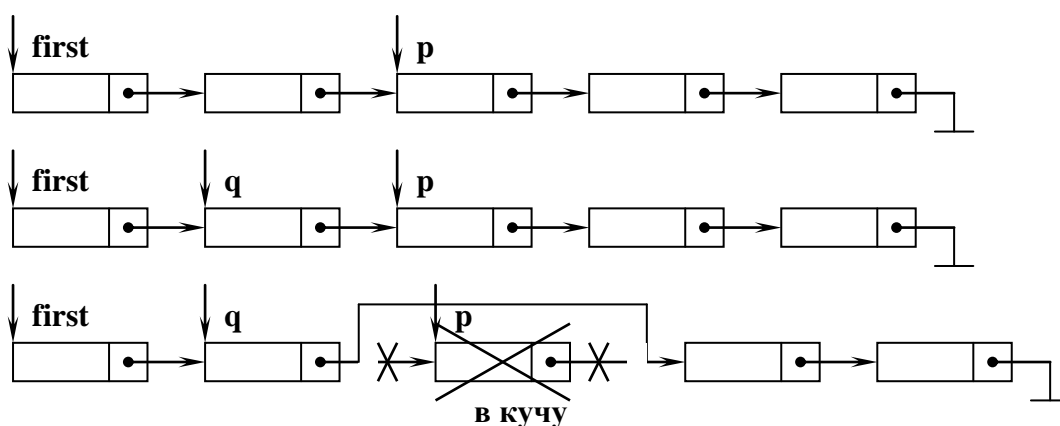


Рис. 21. Исключение узла, на который предварительно установлена ссылка

??? Реализуйте алгоритм исключения узла перед узлом, на который предварительно установлена ссылка, самостоятельно.

#### 2.4.10. Разрушение односвязного списка

Для разрушения списка необходимо просто присвоить *null* ссылке на его первый элемент, т.к. память, занятую элементами хранения узлов списка, возвращает в кучу сборщик мусора.

#### 2.4.11. Демонстрационная программа, реализующая операции создания, обработки, просмотра содержимого списка произвольного вида

```

Using System;
public class Node // описание узла списка
{
    private int info; // информационное поле узла
    private Node link; // поле связи узла

    public int Info {...} // свойства
    public Node Link {...}

    public Node (int info) // конструкторы

```

```

{
    Info = info;
}

public Node (int info, Node link)
{
    Info = info; Link = link;
}
}

public class SingleLinkedList                                // класс «односвязные списки»
{
    private Node first;                                     // ссылка на первый узел списка
    public Node First                                     // свойства
    {
        get{ return first; }
        set{ first = value; }
    }

    public SingleLinkedList()                               // конструктор по умолчанию
    {
        first = null;                                     // создание пустого списка
    }

    public SingleLinkedList(int[ ] dates)                   // конструктор с параметрами
                                                            // first – ссылка на первый узел списка,
                                                            // dates – массив значений информационных полей
    {
        first = null;                                     // создание пустого списка
        for (int i = 0; i < dates.Length; i++)
        {
            Node p = new Node( dates[i], first );         // вставка узла в начало списка
            first = p;
        }
    }

    public void Print()                                     // просмотр информационных полей узлов списка
    {
        Node p = first;
        while (p != null)
        {
            Console.WriteLine (p.Info);
            p = p.Link;
        }
    }
}

```

```

public int Work( )           // суммирование значений информ. полей узлов списка
{
    Node p = first; int s = 0;
    while (p != null)
    {
        s = s + p.Info; p = p.Link;
    }
}

public void Destroy( )      // разрушение списка
{
    first = null;
}
}

public class Program
{
    static void Main()
    {
        SingleLinkedList list;
        list = new SingleLinkedList(new Int32[] { 1, 2, 3, 4, 5, 6 }); // конструктор
        Console.ReadLine();
        list.Print(); // просмотр списка
        Console.Write ( "Сумма значений инф. полей = " );
        Console.WriteLine ( list.Work() ); // обработка списка
        Console.ReadLine();
        list.Destroy(); // разрушение списка
    }
}

```

## 2.5. Односвязные циклические списки

**Циклически связанный список** (сокращенно – циклический список) обладает той особенностью, что поле связи его последнего элемента не содержит значения *null*, а указывает на первый узел списка. В целях удобства обработки в структуру циклического списка включают специальный узел с особым содержанием информационного поля (на рис. 26 это поле заштриховано), называемый **головой списка** или **“сторожем”**. Голова списка является дополнительным элементом. Назначение этого элемента состоит в том, чтобы отметить точку входа в циклический список, а также упростить включение узлов в начало списка и исключение узлов из

начала списка. На рис. 22 показана структура односвязного циклического списка с головным элементом.

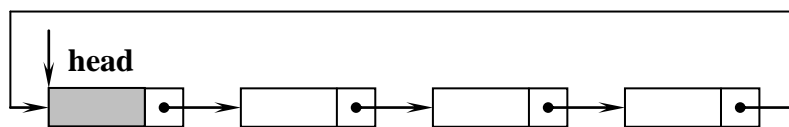


Рис. 22. Структура односвязного циклического списка с головным узлом

Элемент хранения узла односвязного циклического списка описывается так же, как узел односвязного нециклического списка.

Выполнение условия  $head == null$  означает, что односвязный циклический список не существует. Выполнение условия  $head.Link == head$  означает, что односвязный циклический список существует, но является пустым, т.е. содержит только головной элемент. Пустой циклический список с головным элементом представляется структурой элементарного кольца (рис. 23).

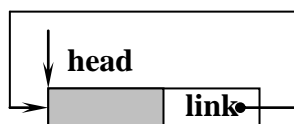


Рис. 23. Структура элементарного односвязного кольца

```
public class CycleSingleLinkedList // класс "Односвязные циклические списки"
{
    private Node head; // ссылка на головной узел списка

    public CycleSingleLinkedList() // создание элементарного односвязного кольца
    {
        head = new Node();
        head.Link = head;
    }
    ...
}
```

Циклические списки, в том числе, односвязные можно использовать для реализации линейных структур, таких как очереди, стеки, списки произвольного вида. При создании очереди новый узел включается в "хвост" списка, т.е. "перед" головным элементом (рис. 24). При создании стека новый узел включается в начало списка, т.е. непосредственно "за" головным элементом (рис. 25).

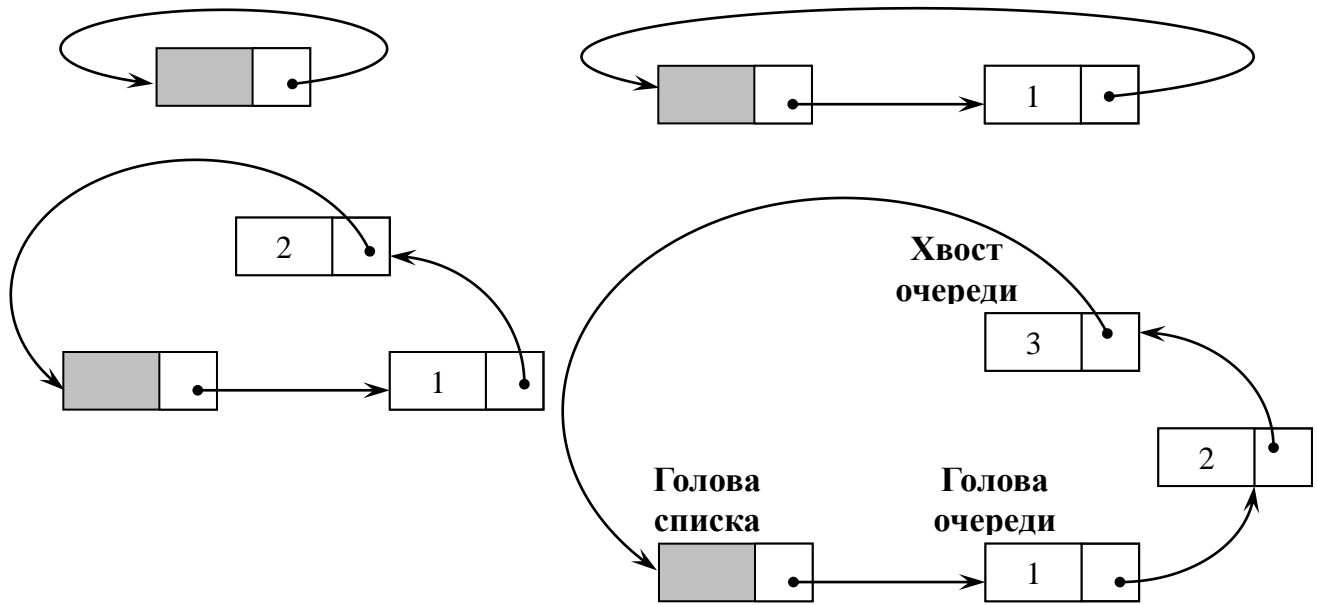


Рис. 24. Создание очереди на структуре односвязного циклического списка

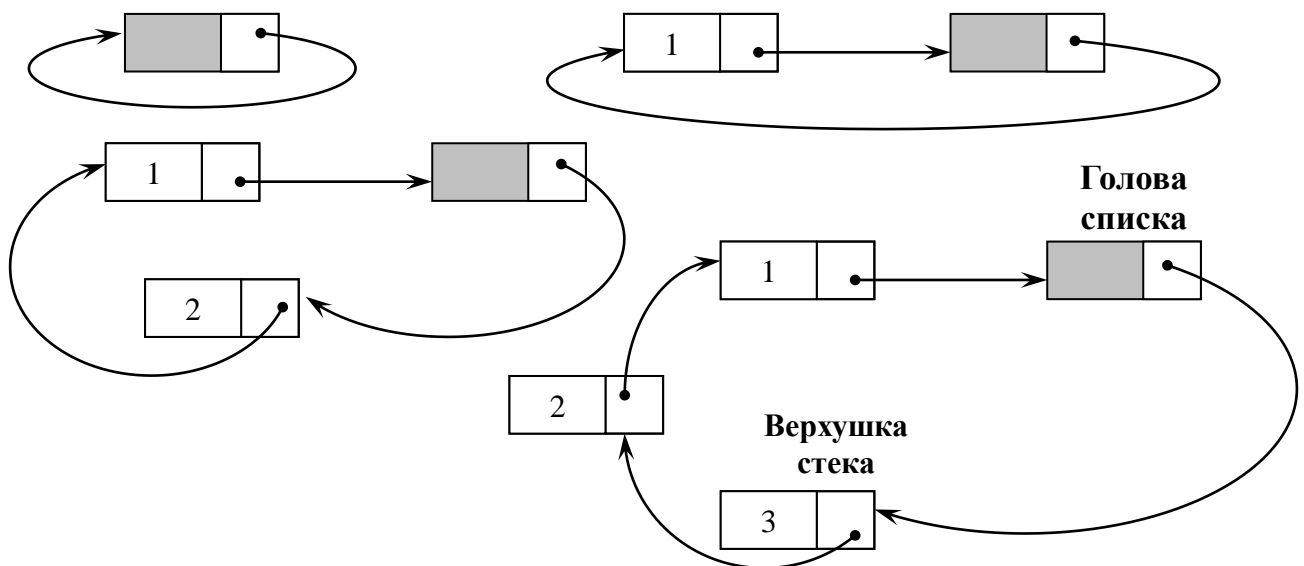


Рис. 25. Создание стека на структуре односвязного циклического списка

Ниже приведен конструктор, выполняющий создание односвязного циклического списка из заданного количества узлов.

```
public CycleSingleLinkedList(int[ ] dates)
    // dates – массив значений информационных полей
{
    head = new Node(); // создание пустого односвязного циклического списка
```



```

head.Link = head;
for (int i = 0; i < dates.Length; i++)
{
    Node p = new Node( dates[i] );           // вставка узла в начало списка
    p.Link = head.Link;                     // установка связи между вставленным узлом и списком
    head.Link = p;                          // обновление поля связи головного узла
}
}

```

// создание списка из трех узлов

```
CycleSingleLinkedList list = new CycleSingleLinkedList(new int [ ] { 1, 2, 3 });
```

Т.к. в циклическом списке каждый элемент, в том числе первый и последний, имеют предшественника и последователя (“перед” первым элементом и “за” последним элементом находится голова), все элементы списка создаются и включаются в список одинаково (см. конструктор *CycleSingleLinkedList*).

**???** *Каким образом включаются узлы в список при выполнении конструктора CycleSingleLinkedList: “за” головным узлом или “перед” ним?*

Исключение первого или последнего узла циклического списка также не имеет никаких особенностей за счет использования головного элемента. Операции включения / исключения узлов в список произвольного вида, реализованный в виде циклического списка, выполняются так же, как в нециклическом списке.

В циклическом списке можно получить доступ к любому элементу списка, продвигаясь от произвольного элемента по кольцу. Поиск элемента по заданному условию в односвязном циклическом списке организуется в цикле, включающем операции проверки выполнения условия для текущего элемента, на который указывает ссылка, и переустановки ссылки на соседний элемент. В процессе поиска используется вспомогательная ссылка, которую первоначально следует установить на узел, следующий за головным. Например,

```

if ( head != null )                               // список существует?
{
    Node p = head.Link;                           // установить вспомогательную ссылку
    while ( p!= head && p.Info != значение ) < тело цикла >           // поиск
}
...

```

Поиск заканчивается либо при обнаружении элемента списка, соответствующего заданному условию (результатом поиска в этом случае является значение ссылки, установленной на искомый узел), либо при возвращении к головному узлу после прохождения всего кольца, если элемент, соответствующий условию поиска, не обнаружен (результатом поиска в этом случае является адрес головного узла). Заметим, что в случае пустого списка цикл не выполнится ни разу.

Ссылка на головной элемент циклического списка не изменяет своего значения при выполнении любых операций над элементами списка, за исключением разрушения списка. При разрушении списка ссылка *head* устанавливается равной *null*.

## 2.6. Двусвязные (двунаправленные) списки

Для достижения большей гибкости в работе с линейными списками можно включить в каждый узел два поля связи – ссылки на следующий узел (т.е. на “правого соседа”) и на предыдущий узел (т.е. на “левого соседа”). Списки с двумя связями занимают больше памяти, чем односвязные, однако в процессе прохода по списку они дают возможность продвигаться как “вперед”, так и “назад”, что повышает эффективность реализации алгоритмов обработки списков.

Элемент хранения узла двусвязного списка содержит информационное поле, поле связи со следующим узлом и поле связи с предыдущим узлом.

В *двусвязном нециклическом списке* первый узел не имеет предшественника (поле связи *prev* первого узла равно *null*), а последний узел не имеет последователя (поле связи *next* последнего узла равно *null*). На первый узел двусвязного списка, имеющего нециклическую структуру, указывает ссылка *first*. Выполнение условия *first == null* означает, что двусвязный нециклический список пуст. Структура двусвязного нециклического списка представлена на рис. 26.

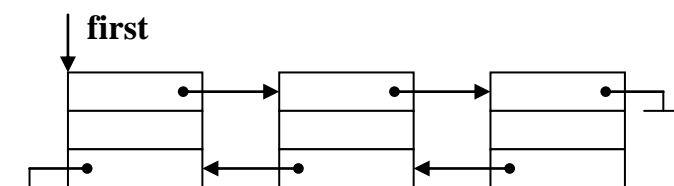


Рис. 26. Структура двусвязного нециклического списка

В .NET Framework Base Class Library двусвязный нециклический список реализован в виде коллекции *LinkedList(T)*, где *T* – тип элемента, хранящегося в узле списка. Каждый элемент этой коллекции (узел двусвязного нециклического списка) является экземпляром класса *LinkedListNode(T)*.

В классе *LinkedListNode(T)* определены следующие свойства и методы (таблица 1).

Таблица 1 – Методы и свойства класса *LinkedListNode(T)*

Сигнатура	Описание
public LinkedListNode(T value);	конструктор инициализирует новый экземпляр класса <i>LinkedListNode(T)</i> , содержащего указанное значение
public LinkedList<T> List { get; }	получить коллекцию <i>LinkedList(T)</i> , к которой принадлежит узел <i>LinkedListNode(T)</i>
public LinkedListNode<T> Next { get; }	получить следующий узел в коллекции <i>LinkedList(T)</i>
public LinkedListNode<T> Previous { get; }	получить предыдущий узел в коллекции <i>LinkedList(T)</i>
public T Value { get; set; }	получить значение, содержащееся в узле

В классе *LinkedList(T)* реализованы стандартные операции по работе с двусвязным списком (Таблица 2).

Таблица 2 – Методы и свойства класса *LinkedList(T)*

Сигнатура	Описание
public LinkedList();	конструктор инициализирует новый экземпляр класса <i>LinkedList(T)</i> ,
public int Count { get; }	получить количество узлов в списке
public LinkedListNode<T> First { get; }	получить первый узел списка
public LinkedListNode<T> Last { get; }	получить последний узел списка
public void AddAfter(LinkedListNode<T> node, LinkedListNode<T> newNode);	добавить в список новый узел после заданного узла
public LinkedListNode<T> AddAfter(LinkedListNode<T> node, T value);	добавить в список новый узел, содержащий заданное значение, после заданного узла
public void AddBefore(LinkedListNode<T> node, LinkedListNode<T> newNode);	добавить в список новый узел перед заданным узлом
public LinkedListNode<T> AddBefore(LinkedListNode<T> node, T value);	добавить в список новый узел, содержащий заданное значение, перед заданным узлом
public void AddFirst(LinkedListNode<T> node);	добавить новый узел в начало списка
public LinkedListNode<T> AddFirst(T value);	добавить новый узел, содержащий

	заданное значение, в начало списка
<code>public void AddLast(LinkedListNode&lt;T&gt; node);</code>	добавить новый узел в конец списка
<code>public LinkedListNode&lt;T&gt; AddLast(T value);</code>	добавить новый узел, содержащий заданное значение, в конец списка
<code>public void Clear();</code>	удалить все узлы из списка
<code>public bool Contains(T value);</code>	проверить, содержится ли в списке заданное значение
<code>public void CopyTo(T[] array, int index);</code>	скопировать все узлы списка в массив, разместить узлы в массиве, начиная с заданного индекса
<code>public LinkedListNode&lt;T&gt; Find(T value);</code>	найти первый встретившийся узел списка, содержащий заданное значение
<code>public LinkedListNode&lt;T&gt; FindLast(T value);</code>	найти последний встретившийся узел списка, содержащий заданное значение
<code>public void Remove(LinkedListNode&lt;T&gt; node);</code>	удалить из списка заданный узел
<code>public bool Remove(T value);</code>	удалить из списка узел, содержащий заданное значение
<code>public void RemoveFirst();</code>	удалить из списка первый узел
<code>public void RemoveLast();</code>	удалить из списка последний узел

Нециклическая структура двусвязного списка порождает те же проблемы при включении / исключении первого и последнего узлов, что и структура односвязного нециклического списка.

На практике более широко применяются двусвязные циклические списки. В *двусвязном циклическом списке* поле связи *next* его последнего элемента не содержит значения *null*, а указывает на первый узел списка и поле связи *prev* его первого элемента также не содержит значения *null*, а указывает на последний узел списка. В целях удобства обработки в структуру двусвязного циклического списка включают специальный дополнительный узел с особым содержанием информационного поля (на рис. 31 это поле заштриховано), называемый **“головой” списка или “сторожем”**. Заметим, что “левым соседом” первого узла двусвязного циклического списка является его последний узел, а “правым соседом” его последнего узла является первый узел, т.к. информационное поле головного узла имеет особое содержание (а нередко и вовсе не используется). Так что функция “сторожа” в двусвязном циклическом списке оказывается чисто технологической и полностью аналогичной функции “сторожа” в односвязном циклическом списке. Структура двусвязного циклического списка приведена на рис. 27.

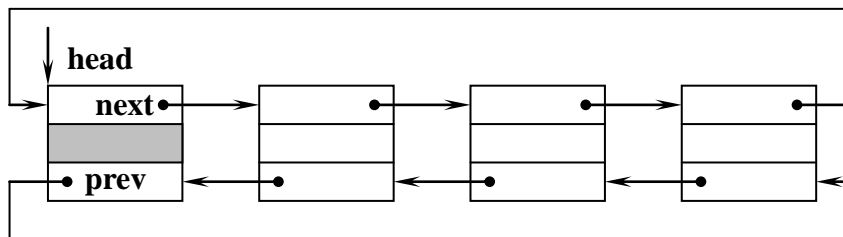


Рис. 27. Структура двусвязного циклического списка

```

public class DoubleNode // Класс «Узел двусвязного циклического списка»
{
    private int info; // информационное поле узла }
    private DoubleNode next; // поле связи со следующим узлом }
    private DoubleNode prev; // поле связи с предыдущим узлом }

    public int Info {...} // свойства }
    public DoubleNode Next {...}
    public DoubleNode Prev {...}

    public DoubleNode () {} // конструкторы }
    public DoubleNode (int info)
    {
        Info = info;
    }
    public DoubleNode (int info, DoubleNode next, DoubleNode prev )
    {
        Info = info; Next = next; Prev = prev;
    }
}

public class CycleDoubleLinkedList // Класс «Двусвязные циклические списки»
{
    private DoubleNode head; // ссылка на головной узел списка
    public CycleDoubleLinkedList() // создание элементарного двусвязного кольца
    {
        head = new DoubleNode ();
        head.Next = head;
        head.Prev = head;
    }
    ...
}

```

Выполнение условия *head == null* означает, что двусвязный циклический список не существует. Выполнение условия *head.Next ==*

$head.Prev == head$  означает, что двусвязный циклический список существует, но является пустым, т.е. содержит только головной элемент. Пустой циклический список с головным элементом представляется структурой элементарного кольца (рис. 28).

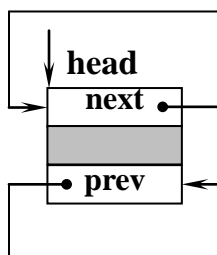


Рис. 28. Структура элементарного двусвязного кольца

Для любого узла двусвязного циклического списка, на который установлена ссылка *DoubleNode p* (в том числе, и для головного), справедливо выражение:  $p.Next.Prev == p.Prev.Next == p$ ;

### 2.6.1. Включение в список нового узла справа или слева от узла, на который предварительно установлена ссылка

Чтобы включить в список новый узел, необходимо выделить память для размещения элемента хранения этого узла и выполнить четыре операции установки связей (рис. 29). Ниже приведен метод включения нового узла “справа” от узла, на который предварительно установлена ссылка.

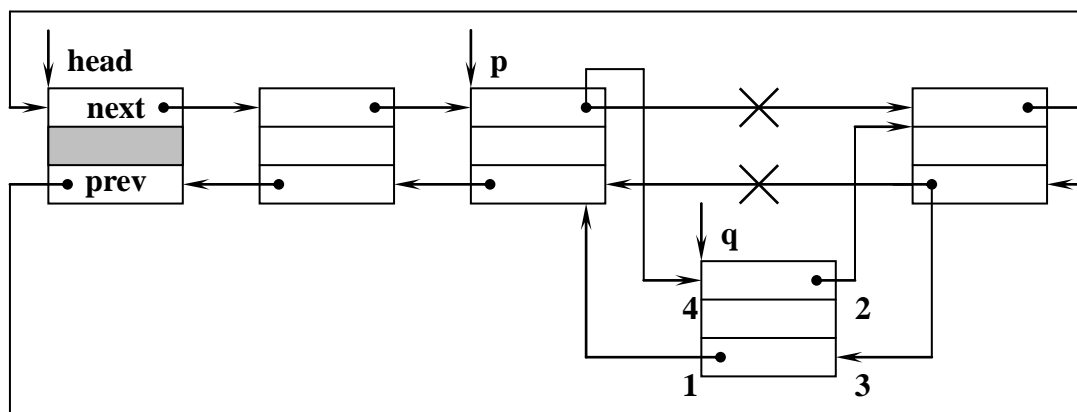


Рис. 29. Включение узла “справа” от узла, на который предварительно установлена ссылка

```
public void InsertAfter(DoubleNode p, int data)
    // p – предварительно установленная ссылка
    // data – значение информационного поля узла списка
{
    DoubleNode q; // q – ссылка на новый узел
    if ( p != null ) // ссылка p действительно установлена?
    {
```

```

q = new DoubleNode( data );           // создание и инициализация нового узла
q. Prev = p;                          // 1 – установка связи нового узла с предыдущим
q. Next = p.Next;                     // 2 – установка связи нового узла со следующим
p.Next.Prev = q;                      // 3 – установка связи следующего узла с новым
p.Next = q;                           // 4 - установка связи предыдущего узла с новым
}
}

```

Связи (1) и (2) можно установить с использованием конструктора, инициализирующего не только информационное поле, но и поля связи узла.

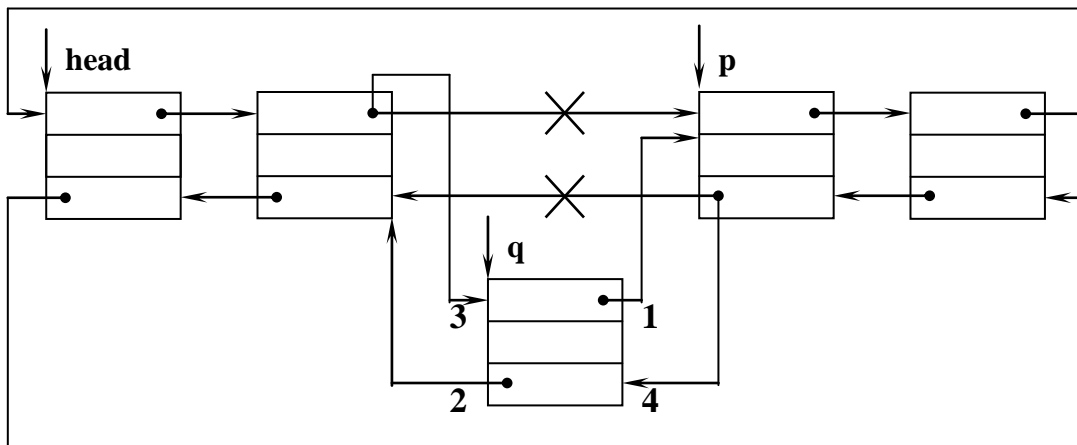
```

q = new DoubleNode( data, p.Next, p );

```

Процедура включения нового узла “слева” от узла, на который предварительно установлена ссылка, выполняется аналогично (рис. 30).

В отличие от односвязного списка, никакого прохода до узла, предшествующего узлу, идентифицируемому ссылкой *p*, не требуется, достаточно обратиться к соответствующему полю связи узла.



**Рис. 30. Включение узла “слева” от узла, на который предварительно установлена ссылка**

Ниже приведен метод создания двусвязного циклического списка, информационные поля которого инициализируются значениями элементов массива.

```

public void Create( int[] dates)           // создание циклического списка
{
    DoubleNode p;
    for ( int i=0; i<dates.Length; i++ )
    {
        p = new DoubleNode( dates[i], head, head.Prev ); // создание узла списка
        head.Prev.Next = p;
    }
}

```

```

    head.Prev = p;
  }
}
...
// создание элементарного кольца
CycleDoubleLinkedList L = new CycleDoubleLinkedList();
L.Create( new int[] {1,2,3,4,5} ); // создание списка из заданного кол-ва узлов

```

??? Каким образом включаются узлы в список при выполнении метода *Create*: “за” головным узлом или “перед” ним?

### 2.6.2. Исключение из списка узла, на который предварительно установлена ссылка

Исключение узла, на который предварительно установлена ссылка, не требует поиска предыдущего узла (рис. 31). Исключенный узел может быть доступен для дальнейшего использования через ссылку *p*. Если дальнейшее использование данного узла не предполагается, ссылку *p* следует установить равной *null*, чтобы сборщик мусора очистил участок памяти, занятый элементом хранения узла.

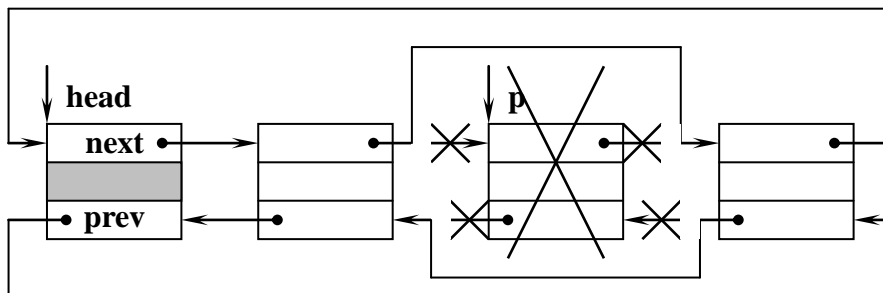


Рис. 31. Исключение узла, на который предварительно установлена ссылка

```

public void Delete (DoubleNode p) // p – ссылка на исключаемый узел
{
  if ( head != null && head.Next != head // список не пуст и ссылка p
      && head.Prev != head && p != null ) // установлена?
  {
    p.Prev.Next = p.Next; // изменить поле связи предыдущего узла
    p.Next.Prev = p.Prev; // изменить поле связи следующего узла
  }
}

```

Операция поиска узла в двусвязном циклическом списке и операция разрушения выполняются так же, как в односвязном циклическом списке, только проход возможен в любом из двух направлений: с использованием



поля связи *next* (т.е. “вперед”) или с использованием поля связи *prev* (т.е. “назад”).

Двусвязные циклические списки, как и односвязные, можно использовать для реализации разнообразных линейных структур.

### 3 ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

Лабораторная работа выполняется в четыре этапа.

1. *Ознакомительный этап*, на котором студент изучает методы представления и обработки динамических списковых структур.
2. *Подготовительный этап*, на котором составляются алгоритм и программа работы с динамическими списковыми структурами.
3. *Лабораторный этап*, на котором производится отладка программы.
4. *Составление отчета*. Отчет должен содержать:
  - название лабораторной работы и текст варианта задания;
  - листинг программы.

Для сдачи отчета необходимо продемонстрировать работу программы.

Требования к программе:

- корректность исходных данных, вводимых с клавиатуры, должна контролироваться программой;
- работа программы должна иллюстрироваться средствами графического и текстового вывода;
- должен быть разработан “дружественный” пользовательский интерфейс, обеспечивающий многократное выполнение программы с переопределением исходных данных, без выхода из оболочки программы.

### 4 КОНТРОЛЬНЫЕ ВОПРОСЫ

1. Что такое ссылочный тип? Какие действия над ссылками Вам известны?

2. Какова структура переменной ссылочного типа?

3. В чем отличия принципа вычисляемого адреса от принципа хранимого адреса?

4. Определите понятие линейной динамической структуры данных.

5. Сравните последовательное и связанное представление структур данных.
6. С какой дополнительной проверкой связан доступ к объекту через ссылку?
7. Какие преимущества дает циклическая организация линейных списков?
8. Каковы особенности организации “проходов” по спискам?
9. Чем отличается организация и обработка двусвязных линейных списков от односвязных?
10. В чем преимущества двусвязной реализации списка перед односвязной? В чем недостатки?
11. Какова функция “сторожа” в циклических списках?
12. Какие связи необходимо установить при включении элемента в односвязный список? В двусвязный?
13. Какие связи необходимо изменить при исключении элемента из односвязного списка? Из двусвязного?
14. В чем заключается особенность удаления элемента, на который предварительно установлена ссылка, из односвязного списка?
15. Как представляется стек на структуре односвязного списка? Двусвязного?
16. Как представляется очередь на структуре односвязного списка? Двусвязного?

## 5 ИНДИВИДУАЛЬНЫЕ ЗАДАНИЯ

1. Найдите среднее арифметическое значений информационных полей элементов списка.

2. Подсчитайте количество положительных и отрицательных элементов в списке.

3. Постройте копию списка.

4. Постройте копию списка, изменив порядок составляющих его элементов на обратный.

5. Переставьте элементы списка в обратном порядке, начиная с номера  $N$  до номера  $K$ , не меняя их размещения в памяти компьютера.

6. Исключите из списка элементы, начиная с номера  $N$  до номера  $K$ .

7. Исключите из списка все элементы между двумя элементами с заданными значениями информационных полей.

8. Элементы списка хранят слова, состоящие из 10 символов, включающих только русские и английские прописные и строчные буквы. Исключите из списка слова, начинающиеся с заданной комбинации символов.

9. Элементы списка хранят слова, состоящие из 10 символов, включающих только русские и английские прописные и строчные буквы. Исключите из списка слова, содержащие заданную комбинацию символов.

10. Из исходного списка получите два новых списка, не меняя размещения элементов в памяти: 1-й список должен содержать элементы, у которых значение информационного поля больше заданного значения  $N$ ; 2-й – все остальные элементы.

11. Из исходного списка получите два новых списка путем копирования: 1-й список должен содержать слова, начинающиеся с заданной комбинации символов; 2-й список – слова, заканчивающиеся заданной комбинацией символов.

12. Создайте список, вставляя положительные числа непосредственно в начало списка, а отрицательные числа – в хвост списка. Разделите полученный список на два списка, содержащие положительные и

отрицательные элементы соответственно, не меняя расположение элементов в памяти.

13. Вставьте в список новый элемент перед каждым элементом с заданным значением информационного поля (информационные поля могут повторяться).

14. Объедините два списка в один путем копирования в следующем порядке: 1-й элемент 1-го списка, 1-й элемент 2-го списка; 2-й элемент 1-го списка, 2-й элемент 2-го списка и т.д.

15. Объедините два списка в один без использования копирования в следующем порядке: 1-й элемент 1-го списка, 1-й элемент 2-го списка; 2-й элемент 1-го списка, 2-й элемент 2-го списка и т.д.

16. Объедините два списка, упорядоченных отношением " $<$ ", в один список за счет изменения связей между элементами. В списке не может быть элементов с одинаковыми значениями информационных полей.

17. Реализуйте представление многочлена

$$P(x) = a_n x_n + a_{n-1} x_{n-1} + \dots + a_1 x + a_0$$

с произвольными целыми коэффициентами в виде списка. При этом, если  $a_i=0$ , то соответствующий элемент-слагаемое должен отсутствовать в списке.

Напишите процедуру, проверяющую два многочлена на равенство, и процедуру вычисления значения многочлена в заданной целочисленной точке.

18. Реализуйте представление многочлена

$$P(x) = a_n x_n + a_{n-1} x_{n-1} + \dots + a_1 x + a_0$$

с произвольными целыми коэффициентами в виде списка. При этом, если  $a_i=0$ , то соответствующий элемент-слагаемое должен отсутствовать в списке.

Напишите процедуру сложения двух многочленов. Результатом такого сложения должен быть новый многочлен-сумма.

19. Реализуйте представление многочлена

$$P(x) = a_n x_n + a_{n-1} x_{n-1} + \dots + a_1 x + a_0$$

с произвольными целыми коэффициентами в виде списка. При этом, если  $a_i=0$ , то соответствующий элемент-слагаемое должен отсутствовать в списке.

Напишите процедуру умножения двух многочленов. Результатом такого умножения должен быть новый многочлен-произведение.

20. Задан многочлен (от переменной  $x$ ) произвольной степени с целыми коэффициентами, причем его одночлены могут быть и не упорядочены по степеням  $x$ , а одночлены одной и той же степени могут повторяться. Приведите подобные члены в этом многочлене, после чего расположите его одночлены по убыванию степеней  $x$ . Представьте многочлен в виде списка.

21. Напишите процедуру копирования значения информационного поля каждого  $k$ -го по счету элемента списка, если его значение не превышает  $N$ , в новый список.

22. Напишите процедуру, выполняющую следующие действия: удалить из первого списка элементы, информационные поля которых совпадают с информационными полями элементов второго списка. Информационные поля у элементов каждого из списков могут повторяться.

23. Дан неупорядоченный список (информационные поля элементов могут повторяться). Не изменяя расположения элементов списка в памяти, постройте список, упорядоченный по неубыванию значений информационных полей его элементов.

24. Два списка заданы ссылками  $F1$  и  $F2$  на их первые элементы. Определите, содержится ли список  $F1$  в  $F2$ ,  $F2$  в  $F1$ , и подсчитайте число полных вхождений элементов первого списка во второй и элементов второго в первый. Например, первый список состоит из элементов 1, 2, 3, 4, 5, 1, 2, 3, 6; второй список состоит из элементов 1, 2, 3; второй список входит в первый 2 раза.

25. Три списка заданы ссылками  $F1$ ,  $F2$  и  $F3$  на их первые элементы. Выполните замену в списке  $F1$  каждого полного вхождения списка  $F2$  (если такое есть) на копию списка  $F3$ . Например, первый список состоит из элементов 1, 2, 3, 4, 5, 1, 2, 3, 6; второй список состоит из элементов 1, 2, 3; третий список состоит из элементов 10, 20. В результате замены первый список будет содержать элементы 10, 20, 4, 5, 10, 20, 6. Количество элементов во втором и третьем списках может не совпадать.

26. Реализуйте моделирование очереди элементов ограниченной длины с дисциплиной "первым пришел – первым вышел" на структуре циклического списка. Моделирование связано с включением новых элементов в очередь, выводом из очереди и идентификацией ситуаций "Очередь полна" и "Очередь пуста".

## ЗАКЛЮЧЕНИЕ

Методические указания “Линейные динамические структуры данных в С#. Списки” посвящены рассмотрению структур данных и алгоритмов, которые являются фундаментом современной методологии разработки программ.

Методические указания подробно описывают линейные динамические структуры данных (односвязные, двусвязные списки, мультисписки).

Теоретический материал иллюстрируется большим количеством примеров программ, реализующих алгоритмы обработки линейных динамических структур данных на языке С#.

Логическим завершением методических указаний являются контрольные вопросы и индивидуальные задания для самостоятельной работы студентов.

Вопросы, имеющие практическое значение для студентов при выполнении домашних заданий и лабораторных работ, освещены с необходимой для использования полнотой.

## БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Н. Вирт. Алгоритмы и структуры данных: пер. с англ. / Н.Вирт. Изд. 2-е, испр. – СПб.: Невский диалект, 2005. – 352 с.
2. Ахо А. Структуры данных и алгоритмы: пер. с англ. / А.Ахо, Д. Хопкрофт, Д.Ульман. – М.: Вильямс, 2016. – 400 с.
3. Уильям Топп, Уильям Форд. Структуры данных в С++: Пер. с англ. – М. ЗАО «Издательство БИНОМ», 1999. – 816 с.
4. Павловская Т.А. С#. Программирование на языке высокого уровня: учебник для вузов / Т.А.Павловская. – СПб.: Питер, 2014. – 432 с.
5. Фаронов В.В. Создание приложений с помощью С#. Руководство программиста / В.В. Фаронов. – М.: Эксмо, 2008. – 576 с.
6. Биллиг В.А. Основы программирования на С#: учебное пособие / В.А.Биллиг – М.: Интернет-Университет Информационных Технологий; БИНОМ. Лаборатория знаний, 2009. – 483 с.
7. Шилдт, Герберт. С# 3.0: руководство для начинающих: Пер. с англ. – М.: ООО «И.Д. Вильямс», 2009. – 688 с.
8. Шилдт, Герберт. С# 4.0: Полное руководство: Пер. с англ. – М.: Издательский дом «Вильямс», 2011. – 1056 с.
9. Гросс, Кристиан. С# 2008 и платформа NET 3.5 Framework: базовое руководство: Пер. с англ. – М.: Издательский дом «Вильямс», 2009. – 480 с.
10. Петцольд, Чарльз. Программирование для Microsoft Windows 8. Пер. с англ. – СПб.: Издательский дом «Питер», 2014. – 1008 с.

11. Стиллмен, Эндрю, Грин, Дженнифер. Изучаем С#. Пер. с англ. – СПб.: Издательский дом «Питер», 2017. – 816 с.
12. Кнут Д. Искусство программирования для ЭВМ: в 3 т. Т 1: пер. с англ. / Д.Кнут. – М.: Вильямс, 2000. – 720 с.

Методические материалы

**ЛИНЕЙНЫЕ ДИНАМИЧЕСКИЕ  
СТРУКТУРЫ ДАННЫХ В C#. СПИСКИ**

*Методические указания*

Составитель *Симонова Елена Витальевна*

ФЕДЕРАЛЬНОЕ ГОСУДАРСТВЕННОЕ АВТОНОМНОЕ  
ОБРАЗОВАТЕЛЬНОЕ УЧРЕЖДЕНИЕ ВЫСШЕГО ОБРАЗОВАНИЯ  
«САМАРСКИЙ НАЦИОНАЛЬНЫЙ ИССЛЕДОВАТЕЛЬСКИЙ  
УНИВЕРСИТЕТ имени академика С. П. КОРОЛЕВА»  
(Самарский университет)  
443086, САМАРА, МОСКОВСКОЕ ШОССЕ, 34.

---

Изд-во Самарского университета.  
443086 Самара, Московское шоссе, 34.