МИНИСТЕРСТВО НАУКИ И ВЫСШЕГО ОБРАЗОВАНИЯ РОССИЙСКОЙ ФЕДЕРАЦИИ

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

Е.В. СИМОНОВА

СТРУКТУРЫ ДАННЫХ В C#

Часть I. ЛИНЕЙНЫЕ ДИНАМИЧЕСКИЕ СТРУКТУРЫ

Рекомендовано редакционно-издательским советом федерального государственного автономного образовательного учреждения высшего образования «Самарский национальный исследовательский университет имени академика С.П. Королева» в качестве учебного пособия для студентов, обучающихся по основной образовательной программе высшего образования по направлению подготовки 09.03.01 Информатика и вычислительная техника

© Самарский университет, 2018

ISBN 978-5-7883-1290-3 (Ч.I)
ISBN 978-5-7883-1294-1

С А М А Р А

Издательство Самарского университета

2018

УДК 004.9(075)
ББК 32.97я7
С 375

Рецензенты: канд. техн. наук, доц. Л. С. З е л е н к о;
д-р техн. наук, проф. С. В. С м и р н о в

Симонова, Елена Витальевна

С 375 **Структуры данных в C#. Часть I. Линейные динамические структуры**: учеб. пособие / *Е.В. Симонова.* – Электрон. текст. и граф. дан. (1,9 Мб). – Самара: Издательство Самарского университета, 2018. – 1 опт. компакт-диск (CD-ROM). – Систем. требования: ПК Pentium, Adobe Acrobat Reader. – Загл. с титул. экрана.

ISBN 978-5-7883-1290-3 (Ч.I)
ISBN 978-5-7883-1294-1

Учебное пособие включает разделы, которые подробно описывают абстрагирование типов, идентификацию объектов, классы памяти, динамические структуры данных (односвязные, двусвязные списки, мультисписки). Теоретический материал иллюстрируется большим количеством программных фрагментов, реализующих алгоритмы обработки различных структур данных. Содержит контрольные вопросы и упражнения по всем разделам.

Предназначено для студентов направления подготовки 09.03.01 Информатика и вычислительная техника.

Подготовлено на кафедре информационных систем и технологий.

УДК 004.9 (075)
ББК 32.97я7

№ госрегистрации 0321900457

© Самарский университет, 2018

Редактор М.С. Сараева

Компьютерная вёрстка А.В. Ярославцевой

Подписано для тиражирования 16.11.2018.

Объем издания 1,9 Мб.

Количество носителей 1 диск.

Тираж 10 экз.

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.

ОГЛАВЛЕНИЕ

ПРЕДИСЛОВИЕ 6

ВВЕДЕНИЕ 7

1. ИДЕНТИФИКАЦИЯ ОБЪЕКТОВ. КЛАССЫ ПАМЯТИ 8

1.1. Понятие типа данных 8

1.2. Идентификация объектов 9

1.2.1. Именование 9

1.2.2. Организация адресного пространства оперативной памяти Windows 9

1.2.3. Понятие ссылки 10

1.3. Классы памяти 10

1.3.1. Распределение адресного пространства оперативной памяти 10

1.3.2. Стек 11

1.3.3. Понятие фрейма активации 13

1.3.4. Управляемая куча 15

1.3.5. Действия над ссылками 18

1.3.6. Сборка мусора 19

1.4. Контрольные вопросы к разделу 1 21

1.5. Упражнения к разделу 1 22

2. ДИНАМИЧЕСКИЕ СТРУКТУРЫ ДАННЫХ 23

2.1. Метод вычисляемого и хранимого адреса. Последовательная и связанная организация памяти 23

2.2. Понятие динамической структуры данных 26

2.3. Линейные динамические структуры данных (связанные списки) 27

2.3.1. Основные виды связанных списков 27

2.4. Односвязные (однонаправленные) списки 28

2.4.1. Включение узла в начало односвязного списка 29

2.4.2. Создание односвязного списка из N узлов: добавление узлов в начало списка 30
2.4.3. Создание односвязного списка из N узлов: 31
добавление узлов в конец списка 31
2.4.4. Исключение узла из начала односвязного списка 32
2.4.5. Переустановка ссылки 33
2.4.6. Поиск узла в односвязном списке по заданному условию 34
2.4.7. Включение в односвязный список нового узла за тем 34
узлом, на который предварительно установлена ссылка 34
2.4.8. Исключение из односвязного списка узла за тем 35
узлом, на который предварительно установлена ссылка 35
2.4.9. Исключение из односвязного списка узла, на 36
который предварительно установлена ссылка 36
2.4.10. Разрушение односязного списка 36
2.4.11. Демонстрационная программа, реализующая операции создания, обработки, просмотра содержимого списка произвольного вида 37
2.4.12. Демонстрационная программа, реализующая пример 39
использования стека 39
2.4.13. Демонстрационная программа, реализующая пример использования очереди 43
2.5. ОДНОСВЯЗНЫЕ ЦИКЛИЧЕСКИЕ СПИСКИ 47
2.6. ДВУСВЯЗНЫЕ (ДВУНАПРАВЛЕННЫЕ) СПИСКИ 50
2.6.1. Включение в список нового узла справа или слева от узла, на который предварительно установлена ссылка 54
2.6.2. Исключение из списка узла, на который предварительно установлена ссылка 56
2.7. ОРТОГОНАЛЬНЫЕ СПИСКИ (МУЛЬТИСПИСКИ) 57
2.8. КОНТРОЛЬНЫЕ ВОПРОСЫ К РАЗДЕЛУ **2** 62
2.9. УПРАЖНЕНИЯ К РАЗДЕЛУ **2** 63
ЗАКЛЮЧЕНИЕ 67
БИБЛИОГРАФИЧЕСКИЙ СПИСОК 68

ПРЕДИСЛОВИЕ

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

Учебное пособие предназначено для студентов, обучающихся по направлению 09.03.01 – Информатика и вычислительная техника.

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

В главе 1 представлены рассматриваются два вида идентификации объектов в программе – именование и указание, понятия адреса объекта и ссылки как универсального идентификатора объектов. Приводятся сведения об организации адресного пространства оперативной памяти. Описывается распределение рабочего пространства оперативной памяти компьютера во время исполнения программы, функциональность различных классов оперативной памяти.

В главе 2 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.

Структуры данных во внешней памяти в предлагаемом пособии не рассматриваются.

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.

Программы, реализующие алгоритмы обработки структур данных различных видов, соответствуют современному уровню информационных технологий. Разделы, рассмотренные в пособии, имеют большое учебно-методическое значение и необходимы при самостоятельной работе студентов во время выполнения ими домашних заданий и лабораторного практикума.

В конце каждой главы приведены упражнения и контрольные вопросы, с помощью которых можно проверить усвоение изложенного материала.

В учебном пособии содержатся сведения, недостаточно освещенные в учебной литературе, а также заложены новые методики преподавания, активизирующие самостоятельную работу студентов.

Учебное пособие может быть полезно широкому кругу читателей, практикующихся в области компьютерных наук.

ВВЕДЕНИЕ

Виды структур данных

Данные, относящиеся к какой-либо проблеме, являются абстрактным, т.е. упрощенным представлением объектов реального мира. Алгоритмы и строение данных неразрывно связаны между собой: представление данных невозможно выбрать, не зная, какие алгоритмы к ним будут применяться, и наоборот, выбор алгоритма часто очень сильно зависит от строения данных. По определению Н. Вирта, программы представляют собой, в конечном счете, конкретные формулировки абстрактных алгоритмов, основанные на конкретных представлениях и структурах данных.

Понятие структуры всегда соответствует сложному объекту, обладающему свойством целостности, и вместе с тем сконструированному из простых компонентов путем использования определенной системы правил. Можно выделить следующие основные виды структур данных в соответствии с возможностью изменения их строения в процессе выполнения программы:

- ***фундаментальные структуры***, включающие стандартные (встроенные в язык) структуры и сложные структуры-агрегаты, которые конструируются на основе стандартных типов и ранее определенных агрегатов. Данные, представленные в виде фундаментальных структур, во время выполнения программы могут изменять значение, но изменить их строение нельзя;
- ***составные*** (***динамические***) ***структуры***, конструируемые на основе фундаментальных структур. Данные, представленные в виде динамических структур, в процессе выполнения программы могут изменять как значение, так и строение.

1. ИДЕНТИФИКАЦИЯ ОБЪЕКТОВ. КЛАССЫ ПАМЯТИ

1.1. Понятие типа данных

Тип данных рассматривается как множество данных, т.е. допустимых значений, которые некоторый объект может принимать в программе в соответствии с внутренним представлением данных, совместно с множеством действий по обработке этих данных. Данные задаются с помощью описания их полей. Совокупность действий, производимых над данными, определяется с помощью операций и функций, реализующих эти действия.

Типы данных в C# классифицируются по различным признакам:

1. В соответствии со строением данных – на простые типы (не имеющие внутренней структуры) и структурированные (состоящие из элементов других типов).
2. В соответствии с тем, кто создает данные – на встроенные в язык (стандартные) типы и конструируемые программистом.
3. В соответствии с продолжительностью существования в программе – на статические, существующие в течение всего времени выполнения программы, и динамические.
4. По способу хранения данных – на значимые типы и ссылочные.

Классификация типов данных по способу хранения в языке C# приведена на рис. 1.

Рис. 1. Классификация типов данных в языке C# по способу хранения

Значения, которые может принимать объект данного типа, называются ***значениями типа***. Во время выполнения программы для каждого объекта

выделяется область оперативной памяти, называемая ***элементом хранения***, в которой размещаются значения типа. Максимальное количество значений в множестве, включающем все значения некоторого типа, называется ***мощностью типа***. ***Размер элемента хранения*** объекта данного типа должен выбираться так, чтобы этот размер был достаточен для размещения любого значения типа. Размер элемента хранения определяется архитектурой компьютера и транслятором. Формат внутреннего представления данных в элементе хранения определяется типом объекта.

Например, множество натуральных чисел 0,1,2,…,255 образует тип ***byte***. Соответственно, мощность типа *byte* составляет 256 = 2^8 значений. Для того чтобы их закодировать, требуется 8 двоичных разрядов (бит), поэтому размер элемента хранения типа *byte* составляет 8 бит. Внутреннее представление данных типа *byte* соответствует беззнаковому целому числу, когда старший разряд рассматривается как часть кода числа.

1.2. Идентификация объектов

Идентификация – это получение доступа к значению программного объекта (далее – объекта), а также определение местонахождения элемента хранения объекта в оперативной памяти. В C# существует два способа идентификации объектов:

- именование,
- обращение по ссылке.

1.2.1. Именование

Именование заключается в назначении объекту определенного имени (идентификатора). Имена однозначно связываются с объектами на этапе компиляции программы, эту связь в процессе выполнения программы изменить нельзя.

```
int x; float y;
```

Именоваться могут также отдельные поля объектов структурированных типов.

С помощью именования обращаются к объектам значимых типов.

1.2.2. Организация адресного пространства оперативной памяти Windows

Для того чтобы определить понятие ссылки, необходимо, прежде всего, кратко рассмотреть, каким образом организовано адресное пространство оперативной памяти.

Оперативная память представляет собой совокупность элементарных ячеек для хранения информации – байтов, каждый из которых имеет свой собственный номер, называемый его ***физическим адресом***. ***Адрес*** позволяет обращаться к любому байту памяти.

Операционная система *Windows* использует метод ***виртуальной адресации***, при которой отображение видимых программе адресов памяти на реальное местоположение в аппаратной памяти полностью управляется операционной системой. В результате, каждому процессу, исполняемому на 32-разрядном процессоре, доступно 4 Гбайта памяти, независимо от того, сколько физической оперативной памяти имеется в компьютере. Эти 4 Гбайта памяти включают все, что относится к программе пользователя: исполняемый код, любые загружаемые им динамические библиотеки, а также элементы хранения всех переменных, используемых работающей программой. Эти 4 Гбайта памяти называют ***виртуальным адресным пространством*** или ***виртуальной памятью*** (далее – память).

Каждый байт в пределах доступных 4 Гбайт памяти нумеруется, начиная с нуля. Чтобы обратиться к элементу хранения, расположенному в определенном месте памяти, необходимо указать номер байта (адрес), начиная с которого в памяти располагается элемент хранения. Чтобы представить адрес памяти в пределах 4 Гбайт, необходимо 32 двоичных разряда или 4 байта (4 Гбайт = 2^2 Гбайт = $2^2 * 2^{10}$ Мбайт = $2^2 * 2^{10} * 2^{20}$ байт = 2^{32} байт).

1.2.3. Понятие ссылки

Ссылка – это особый объект, в элементе хранения которого могут содержаться адреса любых объектов. Таким образом, значениями типа ссылка являются адреса ячеек оперативной памяти и особое значение – ***null***, которое не указывает ни на один из существующих в программе объектов. Мощность ссылочного типа определяется виртуальным адресным пространством. Размер элемента хранения ссылочного типа равен размеру элемента хранения адреса и составляет 4 байта.

1.3. Классы памяти

1.3.1. Распределение адресного пространства оперативной памяти

Распределение рабочего пространства оперативной памяти не является жестким, а происходит во время выполнения программы (см. рис. 2). В нижних адресах располагаются системные программы. Выше располагается исполняемый код программы. Далее располагается область системного стека, необходимая для работы методов. Стек заполняется от своей верхней границы по направлению к началу. В верхних адресах оперативной памяти размещается куча, необходимая для работы с динамическими объектами программы.

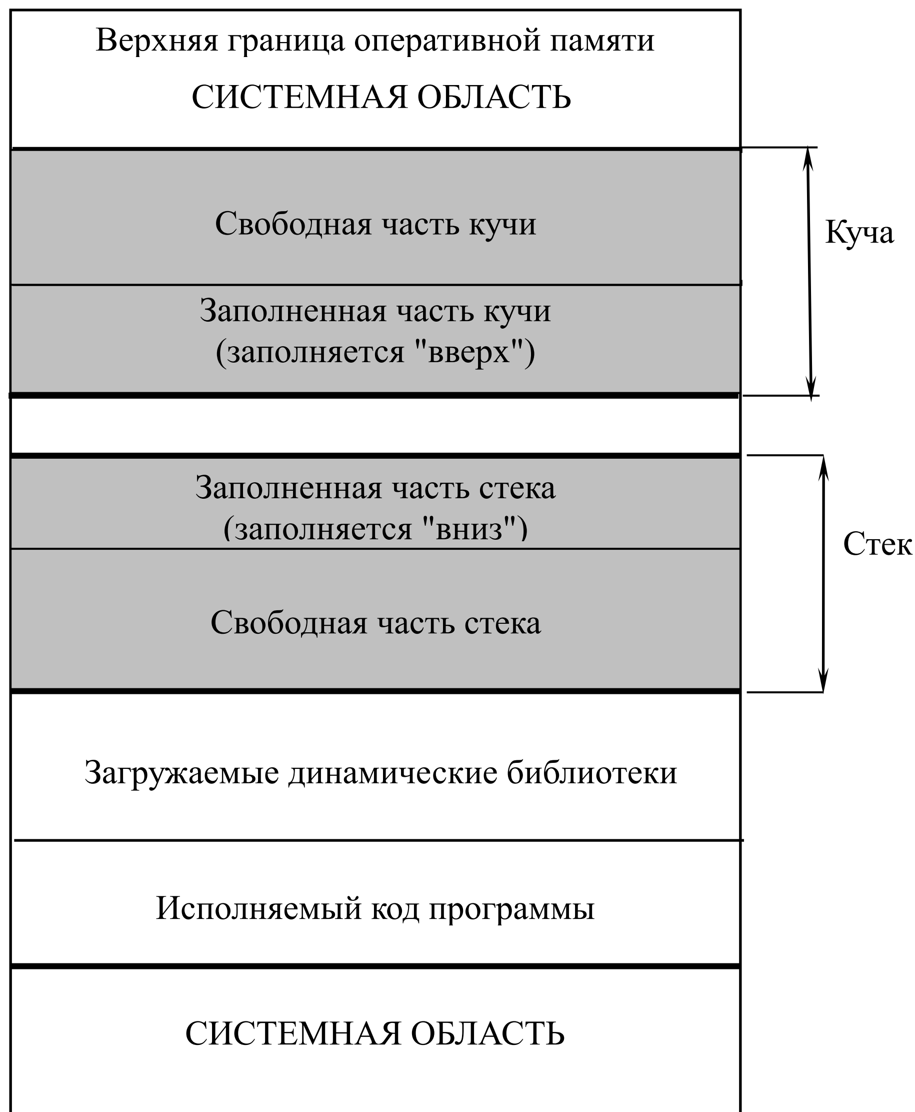


Рис. 2. Распределение рабочего пространства оперативной памяти

Создание объекта следует рассматривать как выделение памяти под его элемент хранения. Согласно такому подходу, рабочее пространство оперативной памяти подразделяется на два класса: стек и куча.

1.3.2. Стек

Стек – это область памяти, которая сохраняет данные по принципу: “последним пришел – первым вышел” (*LIFO – Last Input First Output*). Стек предназначен для хранения данных значимых типов, а также для передачи параметров при выполнении методов. Доступ к элементам хранения переменных, находящихся в стеке, осуществляется по именам переменных. При идентификации именованием существует статическая связь между именем объекта и его элементом хранения (рис. 3).

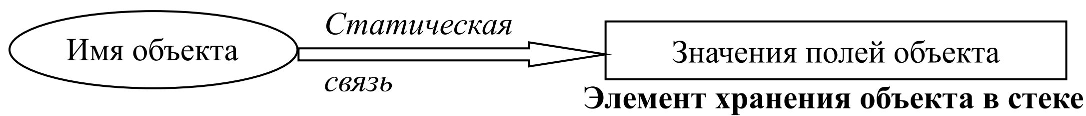


Рис. 3. Идентификация именованием объектов значимых типов

Стек поддерживает область видимости (время жизни) переменных. Если переменная *b* объявлена внутри области видимости переменной *a*, то вначале завершается вложенный блок кода и переменная *b* покидает область видимости, после чего из области видимости выходит переменная *a*. Таким образом, времена жизни переменных вложены друг в друга.

```
{
   int a = 10;
   // действия
   {
     double b = 20.5;
     // действия
   }
}
```

В любом компилируемом языке программирования высокого уровня компилятор преобразует имена переменных, указанные в программе, в адреса, используемые процессором. Для работы стека используется *указатель стека* – специальная переменная, поддерживаемая операционной системой и определяющая адрес следующего свободного байта в стеке.

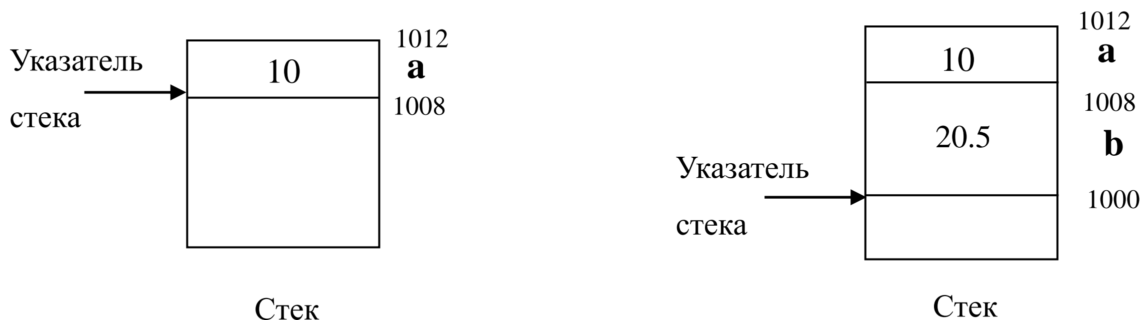


Рис. 4. Размещение переменных в стеке

Когда программа впервые запускается на выполнение, указатель стека установлен на верхний адрес блока памяти, зарезервированного для стека, т.к. стек заполняется от старших адресов памяти к младшим. Когда данные помещаются в стек, указатель стека уменьшается в соответствии с размером элемента хранения размещенных данных (рис. 4).

Например, пусть указатель стека первоначально установлен на адрес 1012. Переменная *a* размещается по адресу 1012, после чего указатель стека уменьшается на величину размера элемента хранения типа *int*, т.е. на 4 байта. Переменная *b* размещается по адресу 1008, после чего указатель стека

уменьшается на величину размера элемента хранения типа *double*, т.е. на 8 байт и указывает на байт, расположенный в памяти по адресу 1000.

Удаление переменных из стека производится в порядке, обратном их размещению, по мере выхода переменных из областей их видимости. Указатель стека при этом увеличивается в соответствии с размером элемента хранения удаленных данных.

1.3.3. Понятие фрейма активации

Стек управляется также директивами программы, связанными с вызовами методов и их окончанием. Каждому методу для работы требуется индивидуальная локальная среда, которая называется фреймом активации метода. ***Фрейм активации*** включает значения фактических параметров, подставляемых на место формальных параметров, указанных в заголовке метода, значения локальных переменных, описываемых внутри метода, а также элемент хранения адреса возврата из метода. Фрейм активации однозначно характеризует метод, т.к. содержит набор объектов, необходимых для его выполнения. Размещение локальной среды связано с активацией метода и происходит автоматически в момент его вызова, а удаление локальной среды связано с пассивацией метода при завершении его выполнения. В программе может одновременно существовать несколько активных методов. Последовательность активации и пассивации методов связана с вложенностью их вызовов (первым должен завершиться метод, который был позже всех вызван). Для обеспечения корректного выполнения вызовов методов в соответствии с дисциплиной *LIFO* используется структура стека. При активации каждого нового метода указатель стека "опускается вниз" на величину, определяемую размером локальной среды данного метода. При пассивации метода указатель стека "поднимается вверх" на эту же величину.

Ниже для фрагмента программы приведена иллюстрация распределения стека (рис. 5). В области *Main* программы описаны две переменные значимых типов: *x* и *y*. В стеке для переменной с именем *x* выделен элемент хранения размером *4 байта*, в который в результате выполнения операции присваивания занесено значение 10, для переменной с именем *y* выделен элемент хранения размером *8 байтов*, который инициализирован значением 0. Значение переменной *y* должно измениться после завершения выполнения метода *W2*, поэтому оно будет передаваться по ссылке.

```
using System;
namespace Storage_Stack
{
  class Storage_Stack
  {
     public static void W1()
       {
```

```
      int b;
      Console.WriteLine("W1");
    }

    public static void W2( int x1, ref float y1)
    {
       char a;
       y1 = 2 * x1; W1();
     }

     static void Main( string[] args )
     {
        int x = 10; float y = 0;
        W2( x, ref y );
        Console.WriteLine( y );
      }
  }
}
```

Рис. 5. Распределение стека

Активация метода *W2* приведет к созданию локальной среды, в которой будут размещены элементы хранения следующих объектов:

- значение фактического параметра *x* (4 байта), равное 10, подставляемого на место формального параметра *int x1*, т.к. данный параметр передается по значению,
- адрес (4 байта) в стеке переменной *y* – фактического параметра, подставляемого вместо формального параметра *ref float y1,* т.к. данный параметр передается по ссылке,
- значение локальной переменной *char a* (2 байта),
- адрес возврата из метода (4 байта).

В процессе выполнения метода *W2* происходит вызов метода *W1*. Активация метода *W1* приведет к созданию локальной среды, в которой будут размещены элементы хранения следующих объектов:

- значение локальной переменной *int b* (4 байта),
- адрес возврата из метода (4 байта).

Пассивация методов *W1*, *W2* и, соответственно, освобождение локальной среды каждого из них происходит в обратном порядке.

Если бы параметр *y1* был объявлен выходным – *out float y1*, распределение стека и передача параметров выполнялась бы аналогично, только не потребовалась бы инициализация фактического параметра *y*.

1.3.4. Управляемая куча

Хотя стек обеспечивает очень высокую производительность, правило вложенности времени жизни переменных во многих случаях слишком ограниченно. 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. 6).

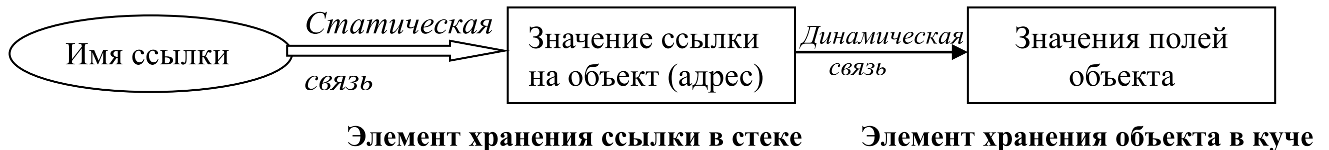


Рис. 6. Идентификация объектов с использованием ссылки

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. 7.

MyClass1 ob1 = new MyClass1();
MyClass2 ob2 = new MyClass2();
MyClass3 ob3 = new MyClass3();
MyClass4 ob4 = new MyClass4();

Рис. 7. 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.

1.3.5. 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. 8.

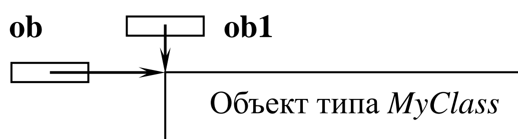


Рис. 8. Присваивание ссылок

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).

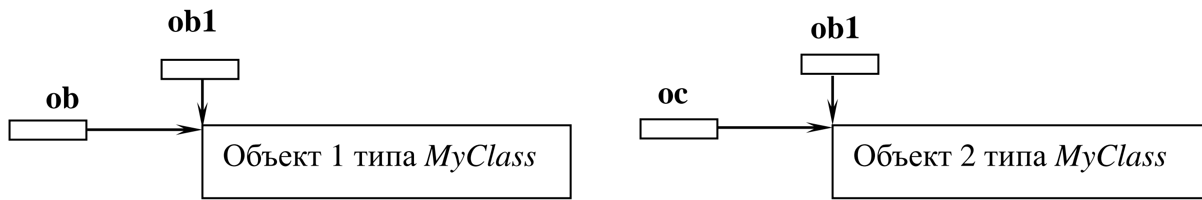


Рис. 9. Переустановка ссылки ob1

1.3.6. 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 вперемешку со свободными участками памяти. Разбиение всей доступной памяти области кучи на большое количество свободных блоков относительно небольшого размера называется ***фрагментацией*** (рис. 10).

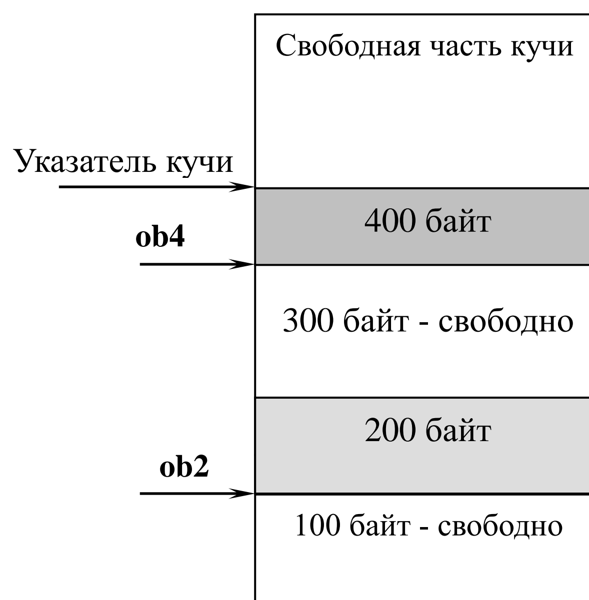
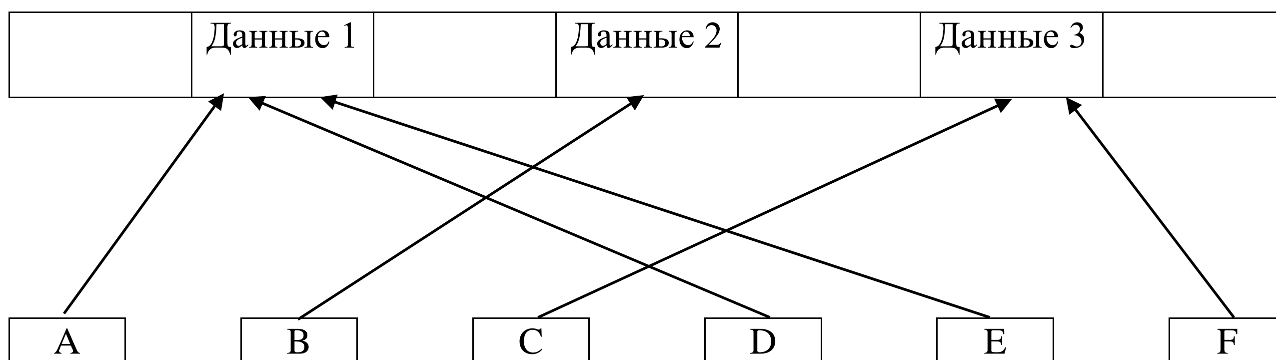
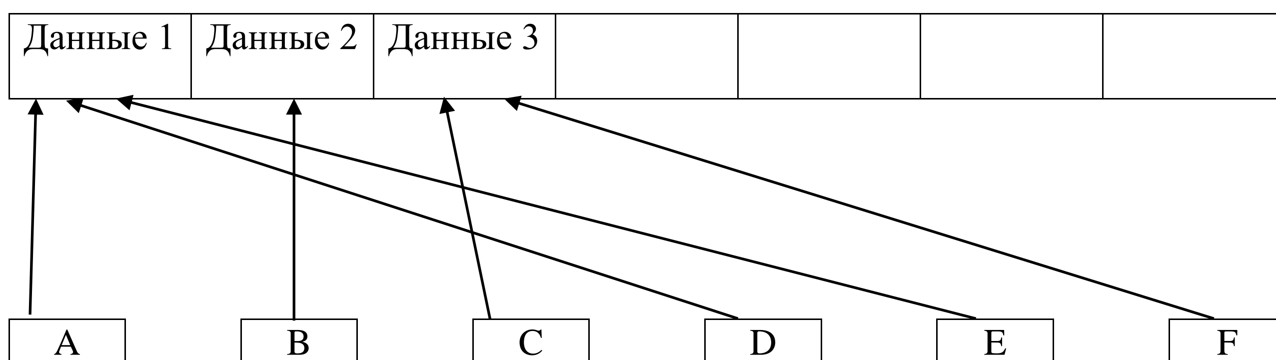


Рис. 10. Фрагментация кучи

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 осовобождает 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. 11.



а. Перед уплотнением



б. После уплотнения

Рис. 11. 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. Определите понятие типа данных. Дайте определение константы типа, элемента хранения типа, мощности типа.

2. Как выполняется классификация типов данных в C# по способу хранения данных?

3. Какие виды идентификации объектов существуют? Чем они отличаются?

4. Дайте определение адреса. Для чего используется адрес? Каким образом организовано адресное пространство оперативной памяти Windows?

5. Какие классы памяти Вам известны? Каким образом распределяется рабочее пространство оперативной памяти во время исполнения программы?

6. Для чего предназначена область оперативной памяти «стек»? Каким образом она функционирует?

7. Что такое фрейм активации метода? Для чего он используется?

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

9. Какие действия над ссылками Вы знаете?

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

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

12. 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, соответветственно, 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. 12). Если *ArrayType* является ссылочным типом, то создается *AllocationSize* ссылок на элементы типа *ArrayType* (рис. 13).

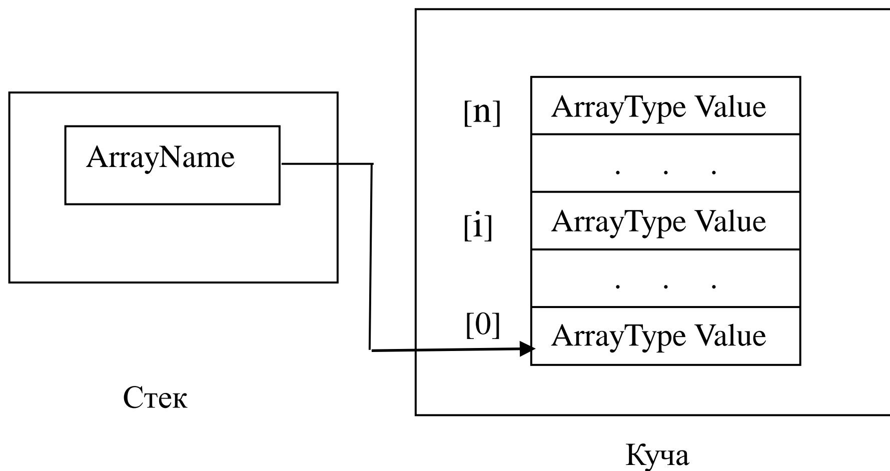


Рис. 12. Размещение массива: ArrayType – значимый тип

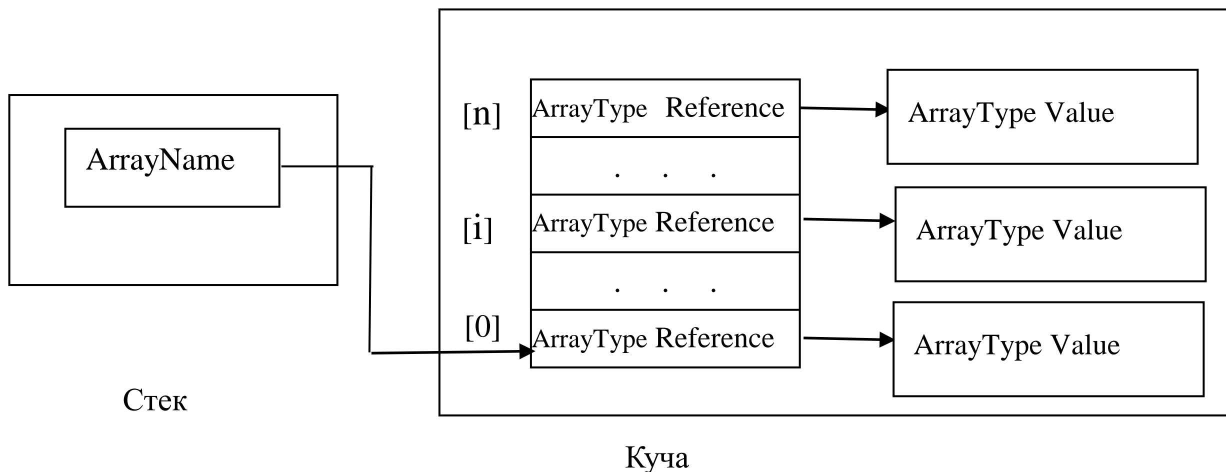


Рис. 13. Размещение массива: ArrayType –ссылочный тип

Адрес A_i i–го элемента массива *ArrayName[i]* компилятор вычисляет по формуле $A_i = A_0 + sizeof(\ ArrayType\ ) * i,\ i >= 0$, 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. 14 приведен пример графической иллюстрации связанной организации структуры из трех объектов.

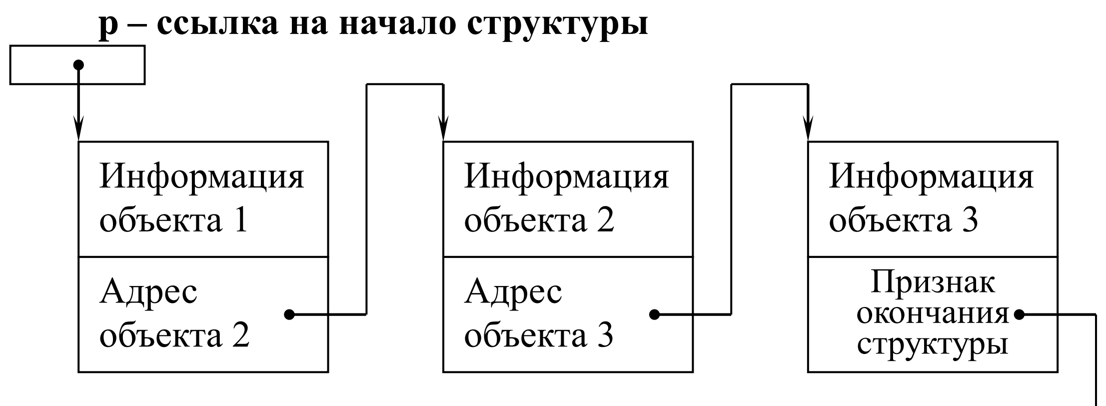


Рис. 14. Графическое представление связанной организации памяти

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

На рис. 15 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.

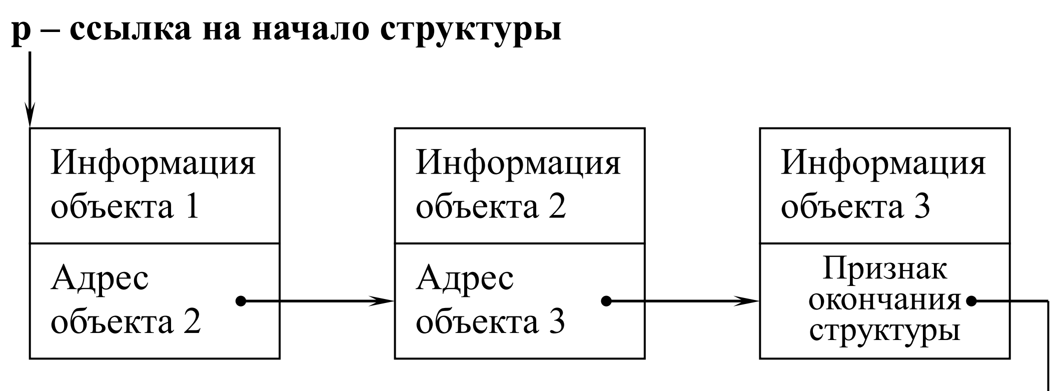


Рис. 15. Упрощенное графическое представление связанной организации памяти

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) $S=\{s_i\}$, $i=1,...,n$, на котором определены отношения предшествования / следования, причем для любого объекта s_i, $i=2,...,n-1$ 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. 16). Для стека характерна дисциплина обслуживания "последним пришел – первым вышел" (*LIFO – Last Input First Output*).

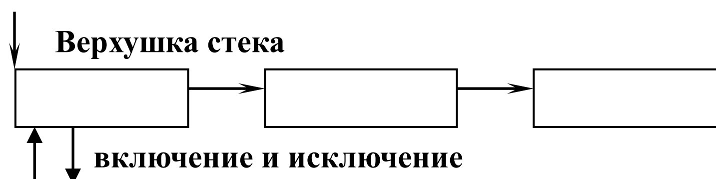


Рис. 16. Структура стека

ОЧЕРЕДЬ (*queue*) – структура, у которой включение элемента производится в хвост, а исключение элемента и доступ к элементам выполняются в начале списка (рис. 17). Для очереди характерна дисциплина обслуживания "первым пришел – первым вышел" (*FIFO – First Input First Output*).

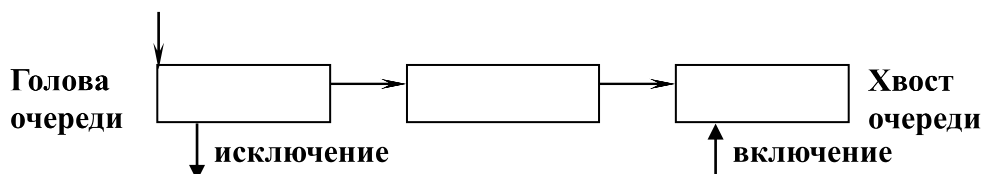


Рис. 17. Структура очереди

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 () { }                       // конструкторы
  public Node (int info)
  {
    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. 17 представлен пример структуры односвязного списка.

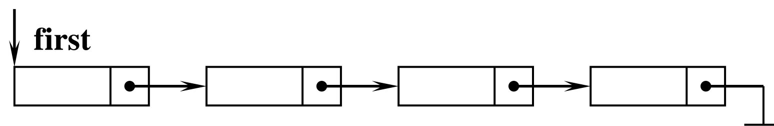


Рис. 17. Структура односвязного списка

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. 18). 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
{                                               // data – значение информационного поля
                                                // узла списка
    Node p = new Node(data);                    // 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)
{                                               // создание узла списка со значением data,
    Node p = new Node ( data, first );          указывающего на первый узел

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

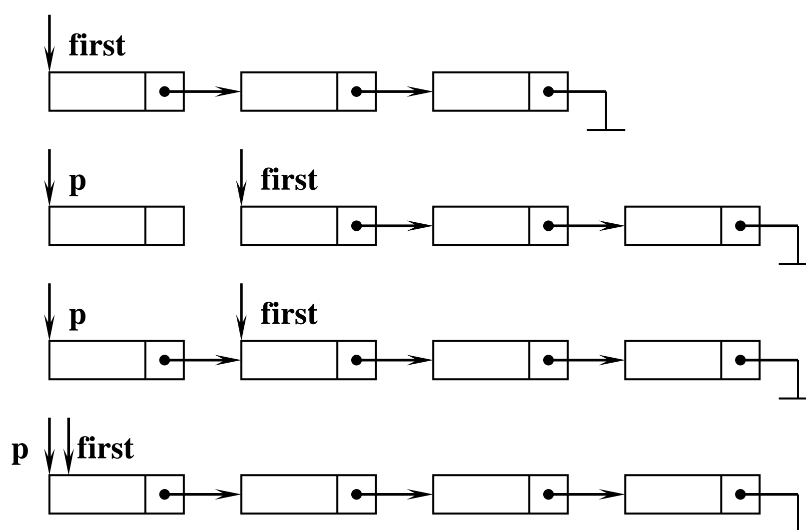


Рис. 18. Включение узла в начало списка

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 void Create (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 L = new SingleLinkedList( new int [] { 1, 2, 3 } );
L.Create( 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. 19).

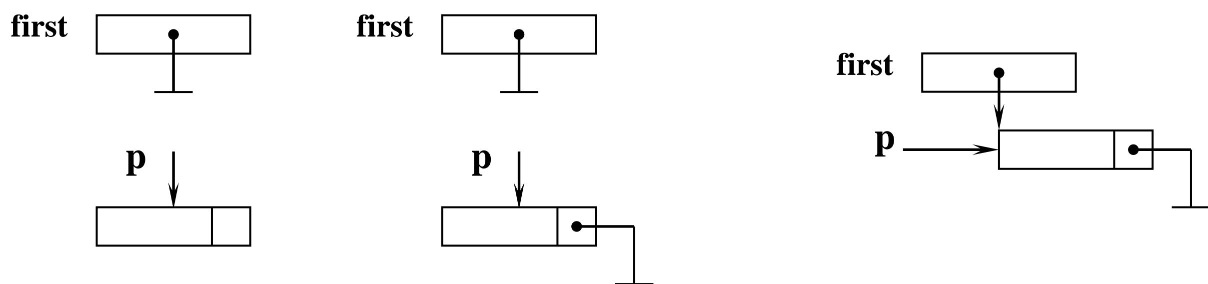


Рис. 19. Включение узла в пустой список

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. 20).

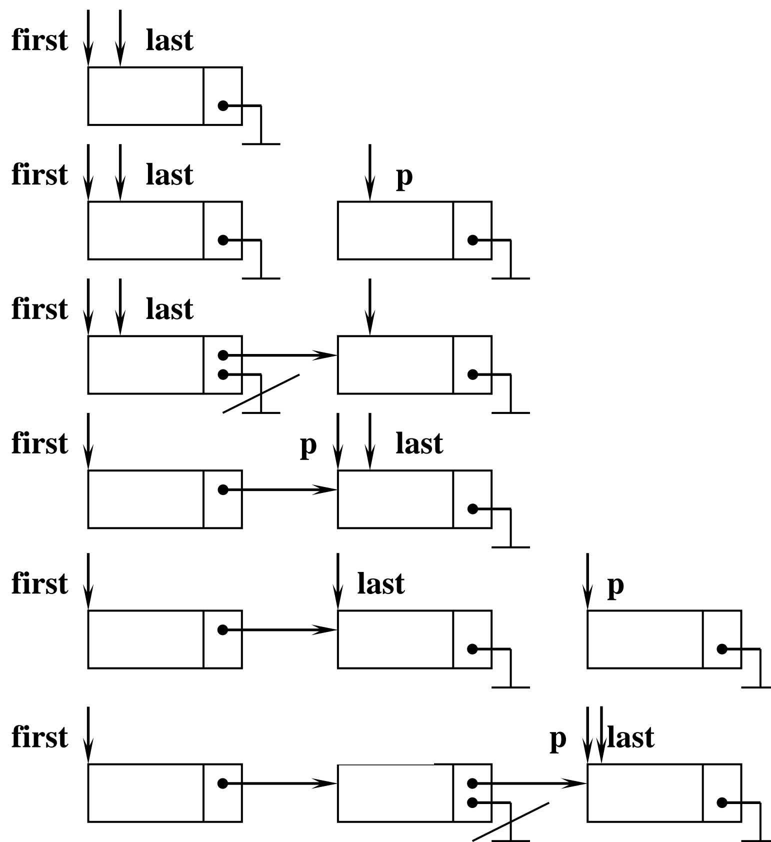


Рис. 20. Включение узла в конец списка

```
public SingleLinkedList(int[ ] dates)            // first – ссылка на первый узел списка
{                                      // dates – массив значений информационных полей
   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. 21), если дальнейшее использование элемента не требуется.

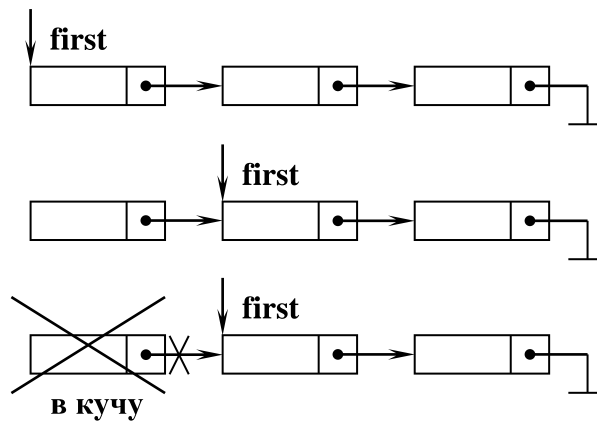


Рис. 21. Исключение узла из начала списка

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. 22). 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). В этом случае последовательность операций переустановки ссылки обеспечивает проход по списку.

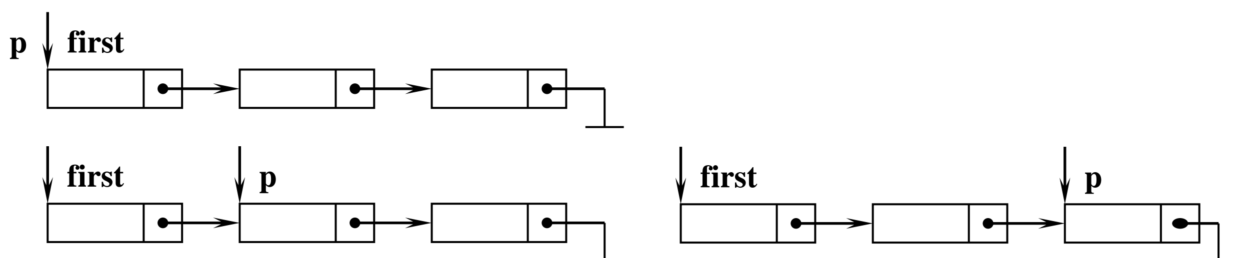


Рис. 22. Переустановка ссылки

```
. . .
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. 23. После выполнения операции вставки значение ссылки на первый элемент списка не изменяется. Значение ссылки на тот элемент, за которым выполнена вставка, также не изменяется.

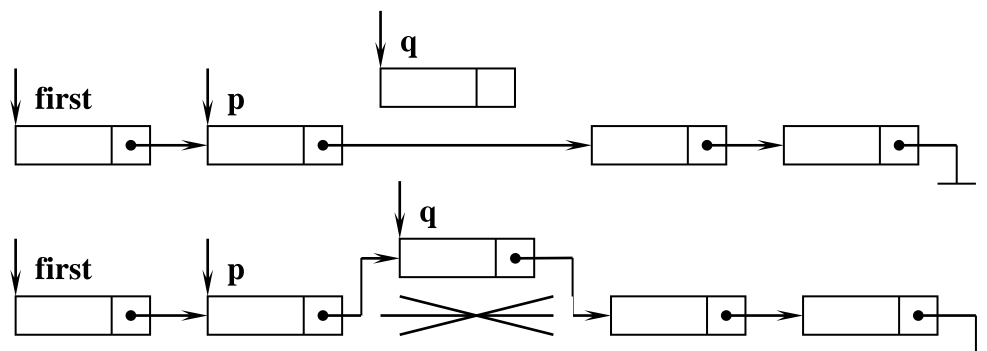


Рис. 23. Включение узла в список за тем узлом, на который предварительно установлена ссылка

```
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. 24. 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.

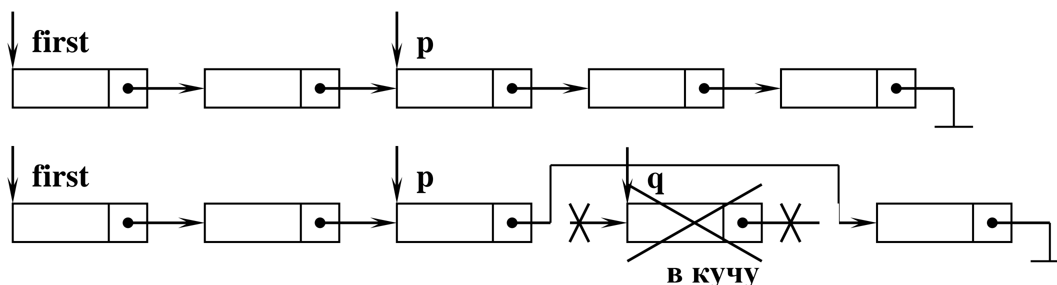


Рис. 24. Исключение узла за тем узлом, на который предварительно установлена ссылка

```
public void DeleteRightElement( Node p)
{                                            // p – предварительно установленная ссылка
  Node q;                                    // q – ссылка на исключаемый узел
  if  ( p != null  &&  p.Link != null )      // сылка 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. 25). 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 возвратить в кучу область памяти, занятую элементом хранения исключенного узла.

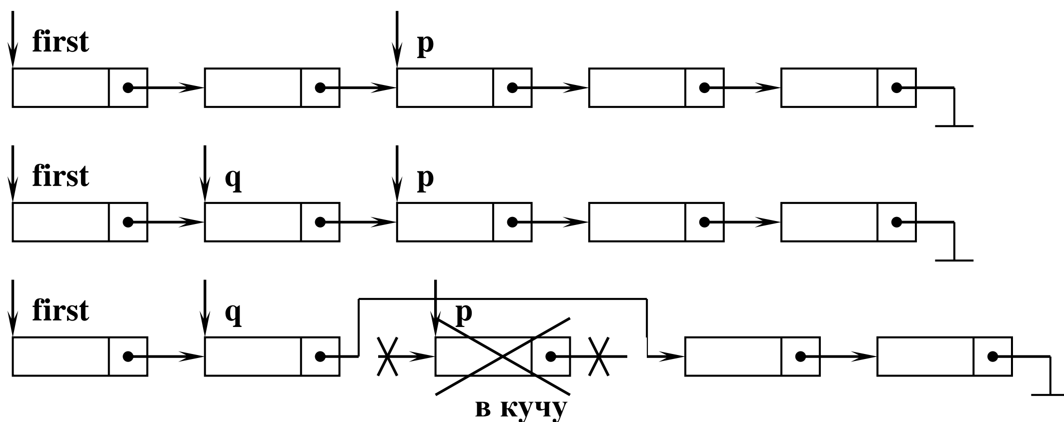


Рис. 25. Исключение узла, на который предварительно установлена ссылка

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

2.4.10. Разрушение односязного 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
    }
}
```

2.4.12. Демонстрационная программа, реализующая пример использования стека

Рассмотрим пример использования списковой структуры стека для решения задачи проверки баланса скобок различного вида в тексте [14].

Будем рассматривать последовательность открывающихся и закрывающихся круглых и квадратных скобок ( ) [ ]. Среди всех таких последовательностей выделим правильные, т.е. те, которые могут быть получены по следующим правилам:

- пустая последовательность правильна;
- если A и B правильны, то и AB правильна;
- если A правильна, то [A] и (A) правильны.

Например, последовательности (), [[]], [()[]()][] правильны, а последовательности],), ([, ([)] – нет. Закодируем члены последовательности числами: (– 1, [– 2,) – -1,] – -2.

Вначале стек пуст. Члены последовательности просматриваем слева направо. Встретив открывающуюся скобку (круглую или квадратную), заносим ее в стек. Встретив закрывающуюся скобку, забираем открывающуюся скобку из вершины стека и проверяем, что в вершине стека находится скобка парная к закрывающейся. Если это не так, можно утверждать, что последовательность неправильна. Последовательность правильна, если в конце стек оказывается пуст.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace BracketControl_Stack
{

   public class Node   // Класс "Элемент стека"
   {
      private int info;
      private Node next;

      public int Info
      {
         get
         {
            return info;
         }
         set
         {
            info = value;
         }
      }
```

```
    public Node Next
    {
        get
        {
            return next;
        }
        set
        {
            next = value;
        }
    }

    public Node(int info, Node next)   // Конструктор
    {
        Next = next;
        Info = info;
    }
}

public class Stack    // Класс "Стек"
{
    private Node top; // Ссылка на верхушку стека

    public Node Top
    {
        get
        {
            return top;
        }
        set
        {
            top = value;
        }
    }

    // Конструктор: создание пустого стека
    public Stack()
    {
        top = null;
    }

    // Занесение элемента в стек
    // info - значение,заносимое в верхушку стека
    public void Push(int info)
    {
        Node p = new Node(info, top);
        top = p;
    }

    // Выталкивание элемента из стека
    // x - значение, извлекаемое из верхушки стека
    public int Pop()
    {  int x = 0;
```

```
        if (top != null)
        {
            x = top.Info;
            top = top.Next;
        }
        return x;
    }

    // Вывод содержимого стека
    public void Print()
    {
        Node p = top;
        while (p != null)
        {
            Console.Write(p.Info + " ");
            p = p.Next;
        }
        Console.WriteLine();
    }

}

public class Bracket        // Класс "Контроль баланса скобок"
{
    private String input;   // Исходная последовательность символов
    private int[] coded;   // Закодированная последовательность символов


    public String Input
    {
        get
        {
            return input;
        }
        set
        {
            input = value;
        }
    }

    public int[] Coded
    {
        get
        {
            return coded;
        }
        set
        {
            coded = value;
        }
    }
```

```
    public Bracket(String s)    // Конструктор
    {
       Input = s; coded = new int[Input.Length];
    }

    // Кодирование последовательности открывающихся и закрывающихся скобок
    public void Coder()
    {
       Console.WriteLine(Input); Console.ReadLine();
       for (int j = 0; j < Input.Length; j++ )
          switch (Input[j])
          {
             case '(': { Coded[j] = 1; break; }
             case ')': { Coded[j] = -1; break; }
             case '[': { Coded[j] = 2; break; }
             case ']': { Coded[j] = -2; break; }
             default: break;
          };
    }

    // Контроль баланса скобок
    public bool Bracket_Control()
    {
       Stack St = new Stack(); // Стек для хранения закодированной послед-ти
       int i = 0; int t;
       Boolean er = false; // Признак соблюдения баланса скобок
       Coder(); // Кодирование исходной последовательности символов
       // Пока последовательность не окончена и ошибка не обнаружена
       while ( (i < Input.Length) && (er == false) )
       {
          // Если обнаружена открывающаяся скобка, занести ее в стек
          if ((Coded[i] == 1) || (Coded[i] == 2)) St.Push(Coded[i]);
          // Если обнаружена закрывающаяся скобка, проверить состояние стека
          else
             // Если стек пуст, зафиксировать ошибку
             if (St.Top == null) er = true;
             else
             {
             // Иначе - вытолкнуть из стека открывающуюся скобку
                t = St.Pop();
             // Если открывающаяся скобка не соответствует закрывающейся скобке,
            //  зафиксировать ошибку
                er = ( t != (-Coded[i]) );
             }
          i++;
       }
       // Ошибка не обнаружена, если все закрывающиеся скобки соответствуют
       // открывающимся скобкам и стек пуст
       er = ( (er == false) && (St.Top == null) );
       return er;
    }

 }
```

```
    public class Program
    {
        static void Main(string[] args)
        {
            String str = "[()[]()][]";  // Тестовая последовательность символов
            Console.WriteLine(str);
            Bracket B = new Bracket(str); // Объект для контроля последовательности символов
            if (B.Bracket_Control()) Console.WriteLine("Баланс скобок соблюдается");
            else Console.WriteLine("Баланс скобок не соблюдается");
            Console.ReadLine();
        }
    }

}
```

2.4.13. Демонстрационная программа, реализующая пример использования очереди

На основании заданной строки текста требуется сформировать новую строку так, чтобы все цифры были перенесены в конец строки и порядок следования символов был сохранен.

Т.к. символы в модифицированной строке следует расположить в прямом порядке, для решения задачи можно использовать списковую структуру очереди. Используем две вспомогательные очереди. В процессе просмотра исходной строки в первую очередь занесем все символы, кроме цифр, а во вторую очередь – цифры. По окончании просмотра исходной строки в модифицированную строку вначале перепишем все символы из первой очереди, а затем – все цифры из второй очереди.

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;

namespace ModifyString_Queue
{

    public class Node   // Класс "Элемент очереди"
    {
        private char info;
        private Node next;

        public char Info
        {
            get
            {
                return info;
            }
            set
            {
```

```
            info = value;
        }
    }

    public Node Next
    {
        get
        {
            return next;
        }
        set
        {
            next = value;
        }
    }

    public Node(char info, Node next)   // Конструктор
    {
        Next = next;
        Info = info;
    }
}

public class Queue    // Класс "Очередь"
{
    private Node first; // Указатель на голову очереди
    private Node last;  // Указатель на хвост очереди

    public Node First
    {
        get
        {
            return first;
        }
        set
        {
            first = value;
        }
    }

    public Node Last
    {
        get
        {
            return last;
        }
        set
        {
            last = value;
        }
    }
```

```
    // Конструктор: создание пустой очереди
    public Queue()
    {
        first = null; last = null;
    }

    // Занесение элемента в очередь
    // info - значение,заносимое в голову очереди
    public void In_Queue(char info)
    {
        Node p = new Node(info, null);
        if (first == null) first = p;
        else last.Next = p;
        last = p;
    }

    // Выталкивание элемента из очереди
    // x - значение, извлекаемое из головы очереди
    public char Out_Queue()
    {
        char x = '0';
        if (first != null)
        {
            x = first.Info;
            first = first.Next;
        }
        if (first == null) last = null;
        return x;
    }

    // Вывод содержимого очереди
    public void Print()
    {
        Node p = first;
        while (p != null)
        {
            Console.Write(p.Info + " ");
            p = p.Next;
        }
        Console.WriteLine();
    }

}

public class ModifyString    // Класс "Модификация строки"
{
    private String input;        // Исходная последовательность символов
    private String coded;        // Модифицированная последоват-ть символов

    public String Input
    {
        get
        {
```

```
        return input;
    }
    set
    {
        input = value;
    }
}

public String Coded
{
    get
    {
        return coded;
    }
    set
    {
        coded = value;
    }
}

public ModifyString(String s)    // Конструктор
{
    Input = s;
}
```

// Модификация строки

```
public void Coder()
{
    Queue Q1 = new Queue();     // очередь1 для букв
    Queue Q2 = new Queue();     // очередь2 для цифр
    char ch;
    for (int j = 0; j < Input.Length; j++)  // строка не окончена?
    {
        if ( char.IsDigit(Input[j]) )            // если обнаружена цифра,
          Q2.In_Queue(Input[j]);                 // занести ее в очередь2
        else Q1.In_Queue(Input[j]);              // иначе - занести букву в очередь1
    }
    // переписать буквы из очереди1 в модифицированную строку
    while (Q1.First != null)
    {
       ch = Q1.Out_Queue();
       Coded = Coded + ch;
    }
    // переписать буквы из очереди2 в модифицированную строку
    while (Q2.First != null)
    {
       ch = Q2.Out_Queue();
       Coded = Coded + ch;
    }
}

// Распечатать модифицированную строку
public void Print_ModifyedString()
```

```
        {
            Console.WriteLine(Coded); Console.ReadLine();
        }

    }

    public class Program
    {
        static void Main(string[] args)
        {
            String S1;
            Console.WriteLine("Введите входную строку");
            S1 = Console.ReadLine(); Console.WriteLine();
            Console.WriteLine("Входная строка");
            Console.WriteLine(S1); Console.WriteLine();
            Console.WriteLine("Модифицированная строка");
            // Объект для модификации строки
            ModifyString M = new ModifyString(S1);
            M.Coder();
            M.Print_ModifyedString();
        }
    }

}
```

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. 26 показана структура односвязного циклического списка с головным элементом.

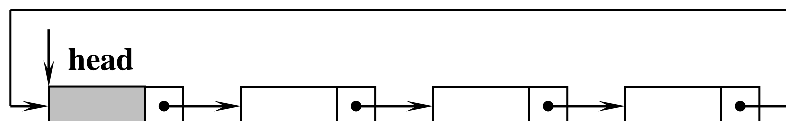


Рис. 26. Структура односвязного циклического списка с головным узлом

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. 27).

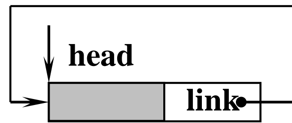


Рис. 27. Структура элементарного односвязного кольца

```
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. 28). При создании стека новый узел включается в начало списка, т.е. непосредственно “за” головным элементом (рис. 29).

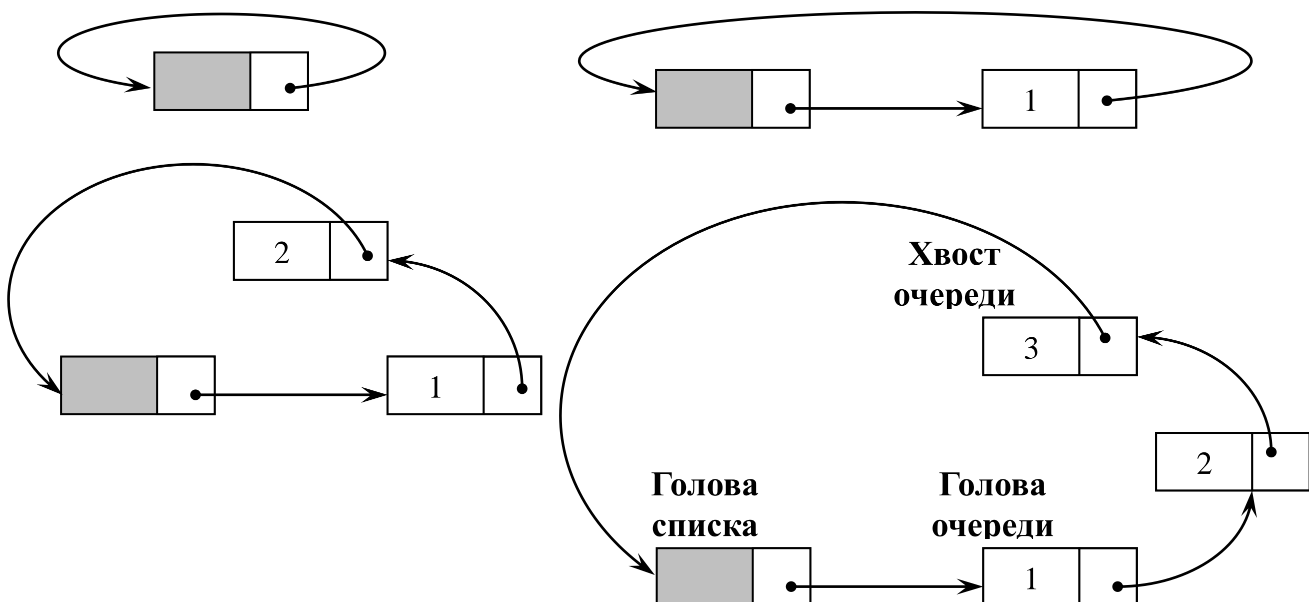


Рис. 28. Создание очереди на структуре односвязного циклического списка

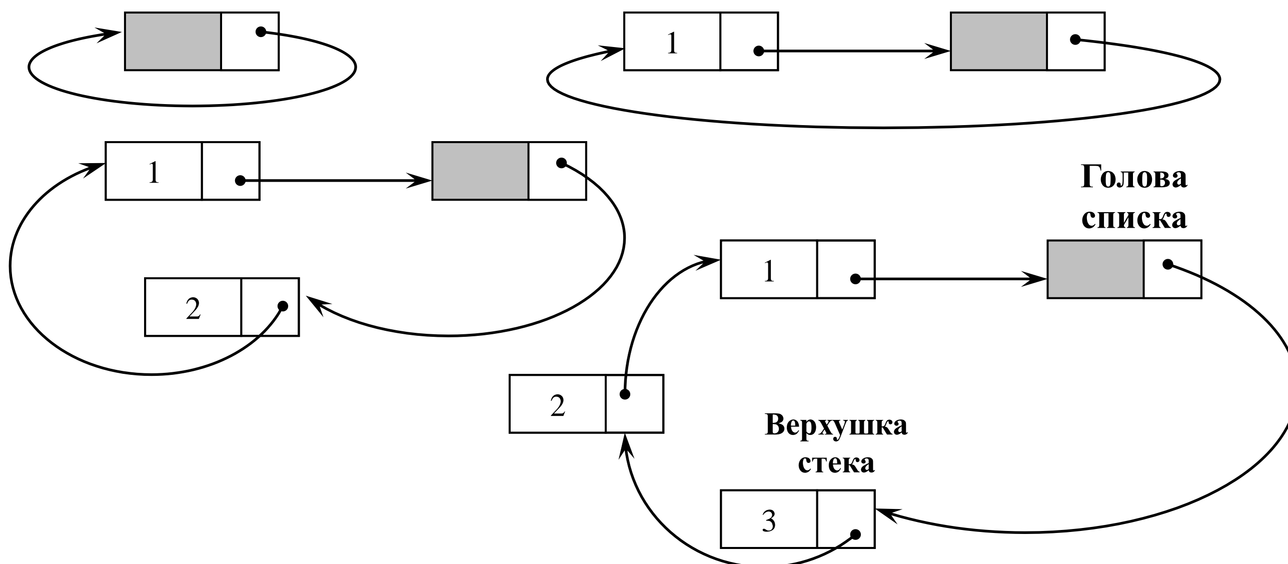


Рис. 29. Создание стека на структуре односвязного циклического списка

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. 30.

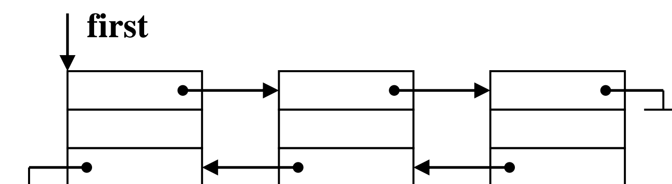


Рис. 30. Структура двусвязного нециклического списка

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 *LinkedNodeList(T)*

Сигнатура	**Описание**
public LinkedListNode(T value);	конструктор инициализирует новый экземпляр класса *LinkedListNode(T)*, содержащего указанное значение
public LinkedList<T> List { get; }	получить коллекцию *LinkedList(T)*, к которой принадлежит узел *LinkedListNode(T)*
public LinkedListNode<T> Next { get; }	получить следующий узел в коллекции *LinkedList(T)*
public LinkedListNode<T> Previous { get; }	получить предыдущий узел в коллекции *LinkedList(T)*
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 *LinkedList(T)*,
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>	добавить в список новый узел после

node, LinkedListNode<T> newNode);	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
public void AddLast(LinkedListNode<T> node);	добавить новый узел в конец списка
public LinkedListNode<T> AddLast(T value);	добавить новый узел, содержащий заданное значение, в конец списка
public void Clear();	удалить все узлы из списка
public bool Contains(T value);	проверить, содержится ли в списке заданное значение
public void CopyTo(T[] array, int index);	скопировать все узлы списка в массив, разместить узлы в массиве, начиная с заданного индекса
public LinkedListNode<T> Find(T value);	найти первый встретившийся узел списка, содержащий заданное значение
ublic LinkedListNode<T> FindLast(T value);	найти последний встретившийся узел списка, содержащий заданное значение
public void Remove(LinkedListNode<T> node);	удалить из списка заданный узел
public bool Remove(T value);	удалить из списка узел, содержащий заданное значение
public void RemoveFirst();	удалить из списка первый узел
public void RemoveLast();	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. 31.

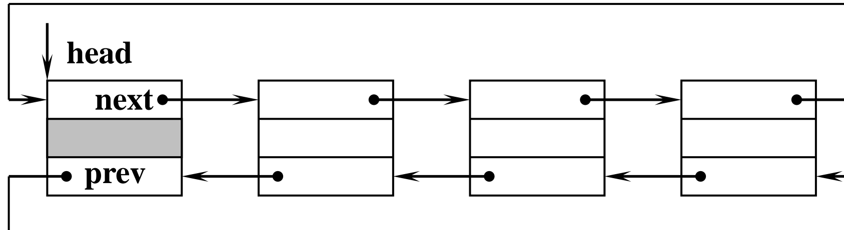


Рис. 31. Структура двусвязного циклического списка

```
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. 32).

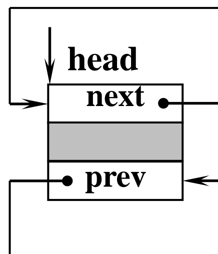


Рис. 32. Структура элементарного двусвязного кольца

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. 33). Ниже приведен метод включения нового узла “справа” от узла, на который предварительно установлена ссылка.

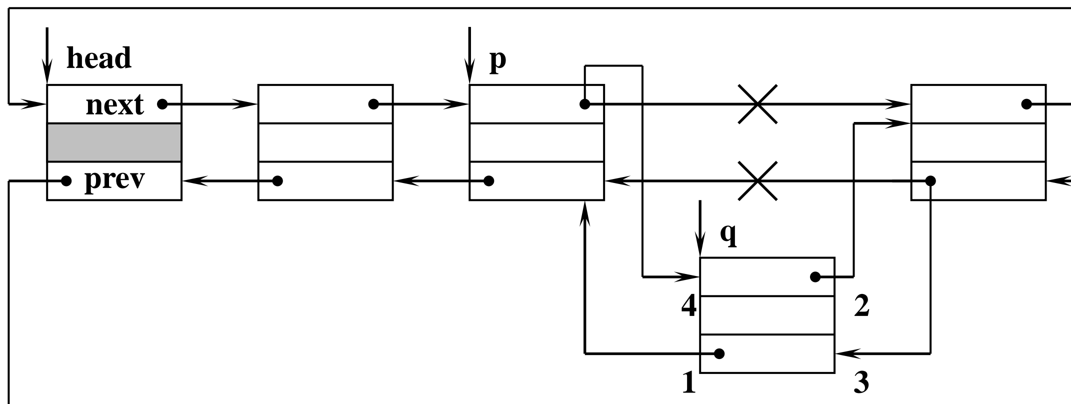


Рис. 33. Включение узла “справа” от узла, на который предварительно установлена ссылка

```
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 следуюшим
      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. 34).

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

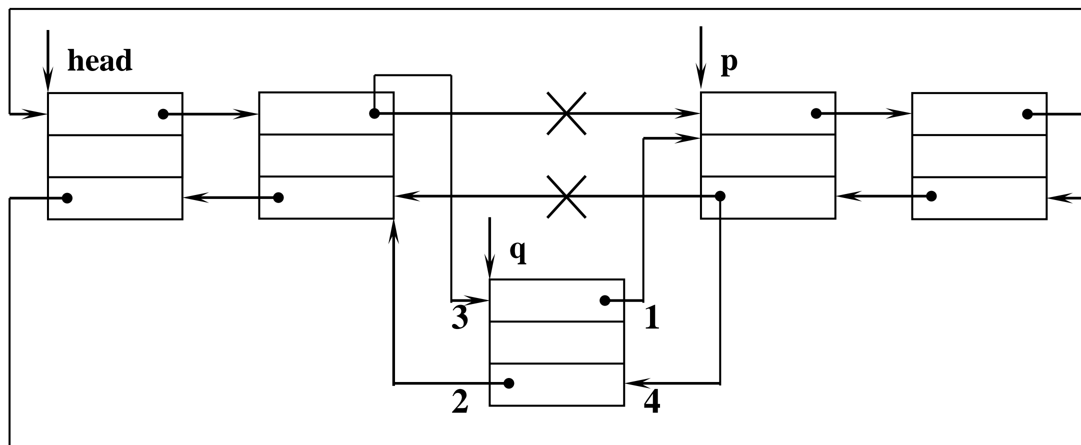


Рис. 34. Включение узла “слева” от узла, на который предварительно установлена ссылка

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. 35). 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.

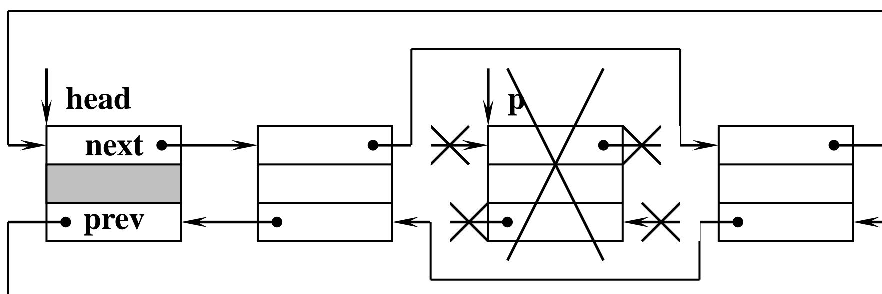


Рис. 35. Исключение узла, на который предварительно установлена ссылка

```
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 любых линейных структур.

2.7. Ортогональные списки (мультисписки)

Ортогональный список (или мультисписок) – это структура, каждый элемент которой входит более чем в один список одновременно и имеет соответствующее числу списков количество полей связи. Реализация каждого из списков может быть выполнена в виде одно- или двусвязного нециклического или циклического списка. Технология обработки мультисписков ничем не отличается от обработки обычных списков, но, так как мультисписок одновременно содержит несколько списков, каждую операцию следует выполнить отдельно для каждого списка.

Мультисписок позволяет на множестве одних и тех же полей, содержащих информацию, организовать различные списки, упорядоченные по различным признакам. Рассмотрим список студентов, каждый узел которого содержит следующие информационные поля: фамилия_ имя_отчество, средний балл, дата рождения, адрес, номер зачетки и т.п. Пусть необходимо упорядочить список студентов по двум признакам: в алфавитном порядке по фамилии и в соответствии со средним баллом. Этого можно достичь, если построить два отдельных списка, но элементы хранения информационных полей в этом случае продублируются, что приведет к неэффективному использованию оперативной памяти, особенно, если количество информационных полей велико. Более рациональным решением является использование мультисписка, в состав которого входят два списка, каждый из которых организован, например, в виде двусвязного циклического списка. По алфавиту элементы списка упорядочены с помощью полей связи *fnext* и *fprev*, а по среднему баллу те же самые элементы упорядочены с помощью полей связи *mnext* и *mprev*. Для удобства обработки мультисписок содержит головной элемент, на который установлена ссылка. Структура мультисписка студентов представлена на рис. 36.

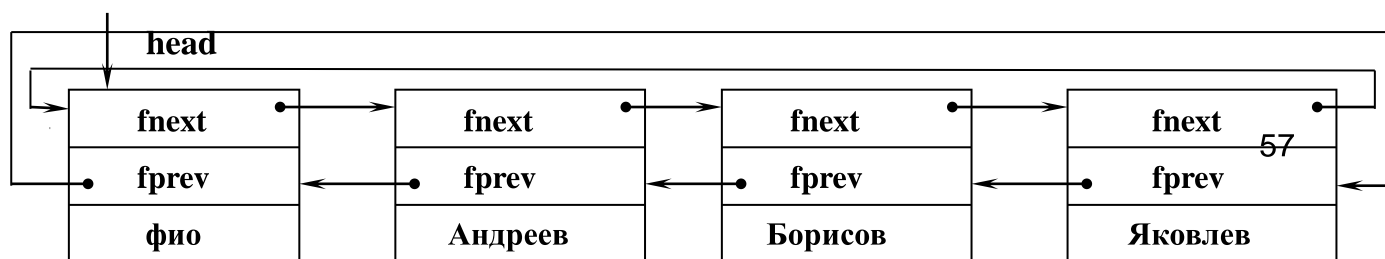


Рис. 36. Структура мультисписка студентов

Описание элемента хранения мультисписка студентов и класса “мультисписок студентов”:

```
public class MultiNode                          // Класс «Узел мультисписка»
{
   private string fam;                          // фамилия_имя_отчество
   private double mark;                         // средний балл
   private MultiNode fnext;                     // поля связи в списке по фамилии
   private MultiNode fprev;
   private MultiNode mnext;                     // поля связи в списке по среднему баллу
   private MultiNode mprev;

   public string Fam {…}                        // свойства
   public double Mark {…}
   public MultiNode Fnext {…}
   public MultiNode Fprev {…}
   public MultiNode Mnext {…}
   public MultiNode Mprev {…}

  public MultiNode () {}                        // конструкторы
  public MultiNode (string fam, double mark)
  {

     Fam = fam; Mark = mark;
  }
  public MultiNode (string fam, double mark, MultiNode fnext,
          MultiNode fprev, MultiNode mnext, MultiNode mprev)
```

```
  {
    Fam = fam; Mark = mark;  Fnext = fnext; Fprev = fprev;
    Mnext = mnext; Mprev = mprev;
  }
}

public class MultiLinkedList                              // Класс “Мультиспиcки”
{
   private MultiNode head;                 // ссылка на головной узел мультисписка

  public MultiLinkedList()                 // создание элементарного мультисписка
   {
      head = new MultiNode ();
      head.Fnext = head;
      head.Fprev = head;
      head.Mnext = head;
      head.Mprev = head;
    }
   . . .                                                              // методы
}
```

Пример обработки мультисписка – методы, распечатывающие содержимое узлов в виде списка, упорядоченного по алфавиту, а затем – в виде списка, упорядоченного по среднему баллу.

```
public void Print_Fam( );                         //  распечатка мультисписка,
{                                                 // упорядоченного по алфавиту
   if  ( head != null )                                  // список существует?
   {
      MultiNode p = head.Fnext;          // установить вспомогательную ссылку
      while ( p <> head )                              // весь список пройден?
      {                                  // распечатать информационные поля}
         Console.WriteLine (p.Fam, p.Mark );
          p:=p.Fnext                           // перейти к следующему узлу
       }
   }
}

public void Print_Mark( );                        // распечатка мультисписка,}
{                                          // упорядоченного по среднему баллу
   if  ( head != null )                                  // список существует?
   {
      MultiNode p = head.Mnext;          // установить вспомогательную ссылку
      while ( p <> head )                              // весь список пройден?
      {                                    // распечатать информационные поля
         Console.WriteLine (p.Fam, p.Mark );
```

```
        p:=p.Mnext                                   // перейти к следующему узлу
      }
   }
}
```

В виде ортогональных списков представляются матрицы очень большой размерности, в которых большинство элементов равны нулю (такие матрицы называются разреженными). Пример разреженной матрицы

$$\begin{bmatrix} 3 & 0 & 5 \\ 0 & 0 & 30 \\ 2 & 10 & 0 \end{bmatrix}$$

Мультисписки обеспечивают эффективное хранение таких структур в памяти, т.к. хранятся только те элементы, которые отличны от нуля (рис. 37). Каждый элемент входит в два списка – в список-строку и в список-столбец. Вся матрица представляется (m + n) списками, где m и n соответственно число строк и число столбцов. Каждый элемент мультисписка хранит значение элемента матрицы, номер строки, номер столбца и две ссылки – на следующий элемент в строке и на следующий элемент в столбце (если используются односвязные списки). Ссылки на первые элементы этих списков хранятся в двух массивах (или в списках).

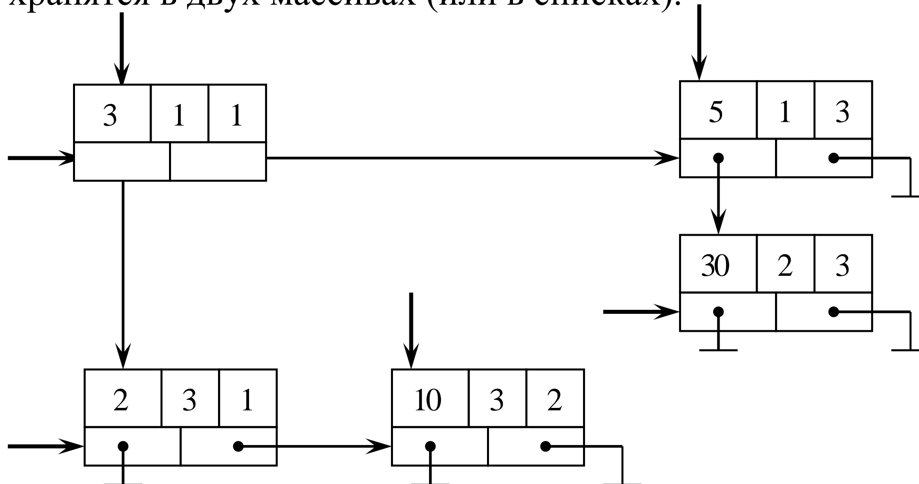


Рис. 37. Разреженная матрица, представленная в виде структуры мультисписка

Описание элемента хранения разреженной матрицы и метод, распечатывающий значения элементов матрицы по строкам, приведен ниже.

```
public class MatrixNode                    // Класс «Узел разреженной матрицы»
{
   private int val;                                      // значение элемента
   private int row, col;                         // номер строки, номер столбца
   private MatrixNode lrow, lcol;   // поля связи в списках по строке и по столбцу
```

```
  public int Val {…}                                          // свойства
  public int Row {…}
  Public int Col {…}
  public MatrixNode Lrow {…}
  public MatrixNode Lcol {…}

 public MatrixNode () {}                                  // конструкторы
 public MatrixNode (int val, int row, int col)
 {
   Val = val; Row = row; Col = col;
 }
}

public class MultiMatrix                    // Класс “Разреженные матрицы”
{
 private int nrow;                                   // количество строк
 private int ncol;                                 // количество столбцов
 private MatrixNode[] prow;                     // массив ссылок на строки
 private MatrixNode[] pcol;                    // массив ссылок на столбцы

 public int Nrow {…}                                          // свойства
 public int Ncol {…}
 public MatrixNode[] Prow {…}
 public MatrixNode[] Pcol {…}

 public MultiMatrix(int nrow, int ncol)                     // конструктор
 {
   Nrow = nrow; Ncol = ncol;
   Prow = new MatrixNode[Nrow];     // инициализация массива ссылок на строки
   for (int i=0; i<=Nrow; i++) Prow[i] = null;
   Pcol = new MatrixNode[Ncol];    // инициализация массива ссылок на столбцы
   for (int i=0; i<=Ncol; i++) Pcol[i] = null;
 }

 . . .                                                          // методы

 public void Print_Matrix()       // распечатка значений матрицы по строкам
 {
   MatrixNode p;           // p – вспомогательная ссылка для прохода по строке
   for ( int i = 0; i < Nrow; i++ )                      // просмотр строк
   {
     p =  Prow[i];      // установка вспогательной ссылки на 1-й элемент списка строки
     while ( p!=null )                              // список строки не пуст?
     {                                                   // вывод  значения
       Console.WriteLine("Matrix"+'['+p.Row+ ','+p.Col+']'+'='+p.Val);
```

```
        p = p.Lrow;                    // переход к следующему элементу в строке
      }
    }
}

public class Program
{
  static void Main()
  {
    MultiMatrix M = new MultiMatrix(2,3);          //  создание матрицы
    . . .                                   // заполнение матрицы значениями
    M.Print_Matrix();     // распечатка содержимого элементов матрицы по строкам
  }
}
}
```

2.8. Контрольные вопросы к разделу 2

1. В чем заключаются отличия принципа вычисляемого адреса от принципа хранимого адреса?

2. Сравните последовательное и связанное представление структур данных.

3. Массив – структура данных с последовательной организацией. Виды массивов в C#.

4. Определите понятие линейной динамической структуры данных.

5. С какой дополнительной проверкой связан доступ к объекту с использованием ссылки?

6. Какие преимущества дает циклическая организация линейных списков?

7. Каковы особенности организации "проходов" по спискам?

8. Чем отличается организация и обработка двусвязных линейных списков от односвязных?

9. В чем преимущества двусвязной реализации списка перед односвязной? В чем недостатки?

10. Какова функция "сторожа" в циклических списках?

11. Какие связи необходимо установить при включении элемента в односвязный список? В двусвязный?

12. Какие связи необходимо изменить при исключении элемента из односвязного списка? Из двусвязного?

13. В чем заключается особенность удаления элемента, на который предварительно установлена ссылка, из односвязного списка?

14. Как представляется стек на структуре односвязного списка? Двусвязного?

15. Как представляется очередь на структуре односвязного списка? Двусвязного?

16. Каково назначение мультисписков? В чем особенности их представления и обработки?

2.9. Упражнения к разделу 2

1. Подсчитайте количество положительных и отрицательных элементов в списке.

2. Постройте копию списка.

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

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

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

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

7. 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.

8. 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.

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

10. 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.

11. 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.

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

13. 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.

14. 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.

15. Реализуйте представление многочлена

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + ... + a_1 x + a_0$$

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.

16. Реализуйте представление многочлена

$$P(x) = a_n x^n + a_{n-1} x^{n-1} + ... + a_1 x + a_0$$

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.

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

18. 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.

19. Два списка заданы ссылками F1 и F2 на их первые элементы. Определите, содержатся ли элементы списка 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; элементы второго списка входят в первый список 2 раза.

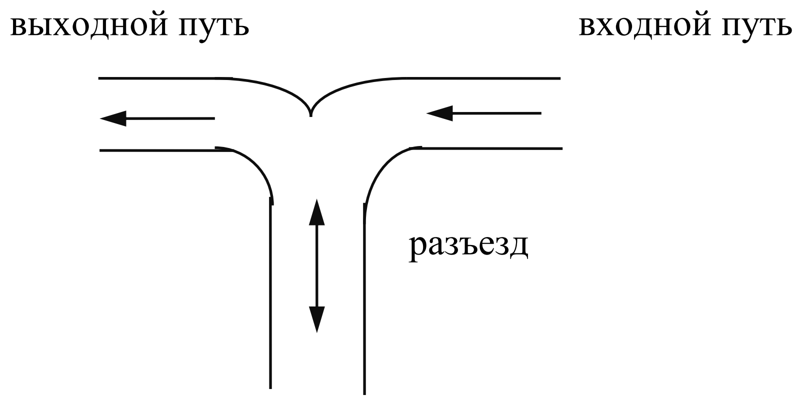
20. Три списка заданы ссылками F1, F2 и F3 на их первые элементы. Напишите процедуру, которая заменяет в списке F1 каждое полное вхождение списка F2 (если такое есть) на копию списка F3. Например, первый список состоит из элементов 1, 2, 3 , 4, 5, 1, 2, 3, 6; второй список состоит из элементов 1, 2, 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.

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

22. Реализуйте моделирование дека ограниченной длины на структуре циклического списка. Дек (двусторонняя очередь) – линейный список, в котором включения и исключения элементов выполняются с любого конца структуры. Моделирование связано с постановкой новых элементов в дек, выводом из дека и идентификацией ситуаций "Дек полон" и "Дек пуст".

23. Смоделируйте работу железнодорожного разъезда. Имеется N железнодорожных вагонов, стоящих в определенном порядке, например, a, b, c, d, ... , x. Необходимо сформировать из них состав в соответствии с заданным порядком следования вагонов, например, c, b, x, ... , a, d.

Возможные направления перемещения вагонов указаны стрелками.



ЗАКЛЮЧЕНИЕ

Учебное пособие “Структуры данных в C#. Часть I. Линейные динамические структуры” посвящено рассмотрению структур данных и алгоритмов, которые являются фундаментом современной методологии разработки программ.

Учебное пособие включает разделы, которые подробно описывают методы идентификации объектов, классы памяти, линейные динамические структуры данных (односвязные, двусвязные списки, мультисписки), алгоритмы управления динамической памятью. Учебное пособие отличается методикой изложения всех разделов с точки зрения особенностей внутреннего представления структур данных различных видов. Теоретический материал иллюстрируется большим количеством примеров программ, реализующих алгоритмы обработки различных структур данных.

Каждый раздел содержит большое количество примеров программ обработки данных различной структуры на языке C#. Логическим завершением каждого раздела являются контрольные вопросы и упражнения для самостоятельной работы студентов.

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

БИБЛИОГРАФИЧЕСКИЙ СПИСОК

1. Вирт, Н. 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 / А.Ахо, Д. Хопкрофт, Д.Ульман; пер. с англ. – М.: Вильямс, 2016. – 400 с.
3. Кнут, Д. Искусство программирования для ЭВМ. В 3 т. Т 1 / Д. Кнут; пер. с англ. – М.: Вильямс, 2000. – 720 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 C#. Руководство программиста / В.В. Фаронов. – М.: Эксмо, 2008. – 576 с.
6. Биллиг, В.А.Основы программирования на C#: учеб. пособие / В.А. Биллиг. – М.: Интернет-университет информационных технологий; БИНОМ. Лаборатория знаний, 2009. – 483 с.
7. Шилдт, Герберт. C# 3.0: руководство для начинающих / Герберт Шилдт; пер. с англ. – М.: ООО «И.Д. Вильямс», 2009. – 688 с.
8. Шилдт, Герберт. C# 4.0: Полное руководство / Герберт Шилдт; пер. с англ. – М.: Издательский дом «Вильямс», 2011. – 1056 с.
9. Гросс, Кристиан. C# 2008 и платформа NET 3.5 Framework: базовое руководство / Кристиан Гросс; пер. с англ. – М.: Издательский дом «Вильямс», 2009. – 480 с.
10. Петцольд, Чарльз. Программирование для Microsoft Windows 8 / Чарльз Петцольд; пер. с англ. – СПб.: Издательский дом «Питер», 2014. – 1008 с.
11. Стиллмен, Эндрю. Изучаем C# / Эндрю Стиллмен, Грин Дженнифер; пер. с англ. – СПб.: Издательский дом «Питер», 2017. – 816 с.
12. Симонова, Е.В. Структуры данных. Часть I. Линейные динамические структуры: учеб. пособие для вузов / Е.В. Симонова. – Самара: Изд-во СГАУ, 2006. – 82 с.
13. Симонова, Е.В. Структуры данных. Часть II. Нелинейные динамические структуры: учеб. пособие для вузов / Е.В. Симонова. – Самара: Изд-во СГАУ, 2007. – 80 с.
14. Шень, А. Программирование. Теоремы и задачи: учеб. пособие для вузов/ А.Шень. – М.: МЦНМО, 2007. – 296 с.
15. Кормен, Т. Алгоритмы: построение и анализ / Т. Кормен, Ч. Лейзерсон, Р. Ривест; пер. с англ. – М.: МЦНМО, 2000. – 960 с.