# Практическая работа № 12. Классы-коллекции, создаваемые пользователем

Цель работы: Получить практические навыки создания классов, реализующих коллекции.

### Динамические структуры данных

**Абстрактный тип данных** — это тип данных, который предоставляет для работы с элементами этого типа определённый набор функций, а также возможность создавать элементы этого типа при помощи специальных функций. Вся внутренняя структура такого типа спрятана от разработчика программного обеспечения — в этом и заключается суть абстракции. Абстрактный тип данных определяет набор функций, независимых от конкретной реализации типа, для оперирования его значениями. Конкретные реализации АТД называются **структурами данных.**

Во многих задачах требуется использовать данные, у которых конфигурация, размеры и состав могут меняться в процессе выполнения программы. Для их представления используют **динамические информационные структуры**. К таким структурам относят:

* однонаправленные списки;
* двунаправленные списки;
* бинарные деревья.

Они отличаются способом связи отдельных элементов и/или допустимыми операциями. Динамическая структура может занимать несмежные участки динамической памяти.

### Однонаправленные списки

Наиболее простой динамической структурой является однонаправленный список, элементами которого служат объекты структурного типа (рис.1).

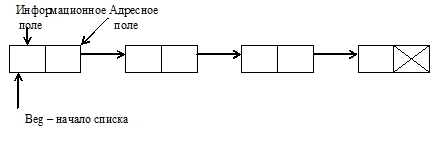


Рис.1. Линейный однонаправленный список

Описание простейшего элемента такого списка выглядит следующим образом:

class имя\_типа

{

информационное поле;

адресное поле;

};

* информационное поле – это поле любого, ранее объявленного или стандартного, типа;
* адресное поле – это указатель на объект того же типа, что и определяемая структура, в него записывается адрес следующего элемента списка.

// простейшее определение элемента списка

class Point

{

public int data; //информационное поле

public Point next;//адресное поле

}

Информационных полей может быть несколько.

Для удобства работы с элементом списка добавим в класс Point конструктор без параметров, конструктор с параметрами и перегрузим метод ToString() для вывода информационного поля.

class Point

{

public int data;//информационное поле

public Point next;//адресное поле

public Point()//конструктор без параметров

{

data=0;

next=null;

}

public Point(int d)//конструктор с параметрами

{

data=d;

next=null;

}

public override string ToString()

{

return data+" ";

}

}

Для того, чтобы создать список, нужно создать сначала первый элемент списка, а затем в цикле добавить к нему остальные элементы. Добавление может выполняться как в начало, так и в конец списка.

//создание элемента списка

static Point MakePoint(int d)

{

Point p = new Point(d);

return p;

}

//добавление в начало однонаправленного списка

static Point MakeList(int size)

{

Random rnd = new Random();

int info = rnd.Next(0, 11);

Console.WriteLine("The element {0} is adding...", info);

Point beg = MakePoint(info);//создаем первый элемент

for (int i = 1; i < size; i++)

{

info = rnd.Next(0, 11);

Console.WriteLine("The element {0} is adding...", info);

//создаем элемент и добавляем в начало списка

Point p = MakePoint(info);

p.next = beg;

beg = p;

}

return beg;

}

//добавление в конец списка

static Point MakeListToEnd(int size)

{

Random rnd = new Random();

int info = rnd.Next(0, 11);

Console.WriteLine("The element {0} is adding...", info);

Point beg = MakePoint(info);//первый элемент

Point r = beg;//переменная хранит адрес конца списка

for (int i = 1; i < size; i++)

{

info = rnd.Next(0, 11);

Console.WriteLine("The element {0} is adding...",info);

//создаем элемент и добавляем в конец списка

Point p = MakePoint(info);

r.next = p;

r = p;

}

return beg;

}

Для обработки списка организуется цикл, в котором нужно переставлять указатель p с помощью оператора p=p.next на следующий элемент списка до тех пор, пока указатель p не станет равен 0, т. е. будет достигнут конец списка.

static void ShowList(Point beg)

{

//проверка наличия элементов в списке

if (beg == null) {

Console.WriteLine("The List is empty");

return;

}

Point p = beg;

while (p!=null)

{

Console.Write(p);

p = p.next;//переход к следующему элементу

}

Console.WriteLine();

}

В динамические структуры легко добавлять элементы и из них легко удалять элементы, т. к. для этого достаточно изменить значения адресных полей.

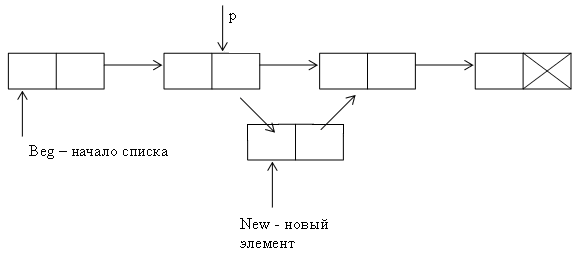


Рис.2. Добавление элемента в список

static Point AddPoint(Point beg, int number)

{

Random rnd = new Random();

int info = rnd.Next(10, 100);

Console.WriteLine("The element {0} is adding...", info);

//создаем новый элемент

Point NewPoint = MakePoint(info);

if (beg == null)//список пустой

{

beg = MakePoint(rnd.Next(10, 100));

return beg;

}

if (number == 1) //добавление в начало списка

{

NewPoint.next = beg;

beg = NewPoint;

return beg;

}

//вспом. переменная для прохода по списку

Point p = beg;

//идем по списку до нужного элемента

for (int i = 1; i < number-1 && p != null; i++)

p = p.next;

if (p == null)//элемент не найден

{

Console.WriteLine("Error! The size of List less than Number");

return beg;

}

//добавляем новый элемент

NewPoint.next = p.next;

p.next = NewPoint;

return beg;

}

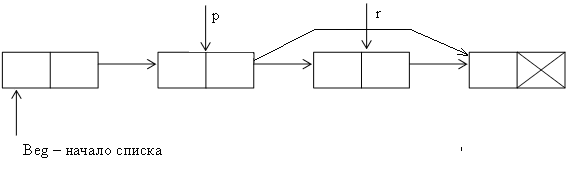


Рис.3. Удаление элемента из списка

static Point DelElement(Point beg, int number)

{

if (beg == null)//пустой список

{

Console.WriteLine("Error! The List is empty");

return null;

}

if (number == 1)//удаляем первый элемент

{

beg = beg.next;

return beg;

}

Point p = beg;

//ищем элемент для удаления и встаем на предыдущий

for (int i = 1; i < number - 1 && p != null; i++)

p = p.next;

if (p == null)//если элемент не найден

{

Console.WriteLine("Error! The size of List less than Number");

return beg;

}

//исключаем элемент из списка

p.next = p.next.next;

return beg;

}

### Двунаправленные списки

Двунаправленный список имеет два адресных поля, которые указывают на следующий элемент списка и на предыдущий. Поэтому двигаться по такому списку можно как слева направо, так и справа налево.

//определение двунаправленного списка

class Point

{

public int data;

public Point next, //адрес следующего элемента

pred;//адрес предыдущего элемента

public Point()

{

data = 0;

next = null;

pred = null;

}

public Point(int d)

{

data = d;

next = null;

pred = null;

}

public override string ToString()

{

return data + " ";

}

}

//формирование двунаправленного списка

static Point MakeList(int size) //добавление в начало

{

Random rnd = new Random();

int info = rnd.Next(0, 11);

Console.WriteLine("The element {0} is adding...", info);

Point beg = MakePoint(info);

for (int i = 1; i < size; i++)

{

info = rnd.Next(0, 11);

Console.WriteLine("The element {0} is adding...", info);

Point p = MakePoint(info);

p.next = beg;

beg.pred = p;

beg = p;

}

return beg;

}

### Бинарные деревья

Бинарное дерево – это динамическая структура данных, состоящая из узлов, ка­ждый из которых содержит, кроме данных, не более двух ссылок на различные бинарные деревья. На каждый узел имеется ровно одна ссылка.

Описать такую структуру можно следующим образом:

class Point

{

public int data;

public Point left,//адрес левого поддерева

right;//адрес правого поддерева

public Point()

{

data = 0;

left = null;

right = null;

}

public Point(int d)

{

data = d;

left = null;

right = null;

}

public override string ToString()

{

return data + " ";

}

}

Начальный узел называется корнем дерева. Узел, не имеющий поддеревьев, называется листом. Исходящие узлы называются предками, входящие — потом­ками. Высота дерева определяется количеством уровней, на которых располага­ются его узлы.

Если дерево организовано таким образом, что для каждого узла все ключи его ле­вого поддерева меньше ключа этого узла, а все ключи его правого поддерева — больше, оно называется деревом поиска. Одинаковые ключи не допускаются. В дереве поиска можно найти элемент по ключу, двигаясь от корня и переходя на левое или правое поддерево в зависимости от значения ключа в каждом узле. Такой поиск гораздо эффективнее поиска по списку, поскольку время поиска определяется высотой дерева, а она пропорциональна двоичному логарифму ко­личества узлов.

В идеально сбалансированном дереве количество узлов справа и сле­ва отличается не более чем на единицу.

Линейный список можно представить как вырожденное бинарное дерево, в котором каждый узел имеет не более одной ссылки. Для списка среднее время поиска равно половине длины списка.

Деревья и списки являются рекурсивными структурами, т. к. каждое поддерево также является деревом. Таким образом, дерево можно определить как рекурсивную структуру, в которой каждый элемент является:

* либо пустой структурой;
* либо элементом, с которым связано конечное число поддеревьев.

Действия с рекурсивными структурами удобнее всего описываются с помощью рекурсивных алгоритмов.

### Обход дерева

Для того, чтобы выполнить определенную операцию над всеми узлами дерева, все узлы надо обойти. Такая задача называется обходом дерева. При обходе узлы должны посещаться в определенном порядке. Существуют три принципа упорядочивания. Рассмотрим дерево, представленное на рис.

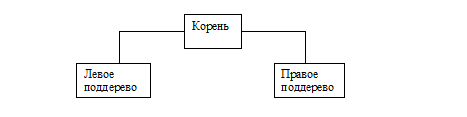


Рис.4. Бинарное дерево

На этом дереве можно определить три метода упорядочивания:

* Слева направо: Левое поддерево – Корень – Правое поддерево;
* Сверху вниз: Корень – Левое поддерево – Правое поддерево;
* Снизу вверх: Левое поддерево – Правое поддерево – Корень.

Эти три метода можно сформулировать в виде рекурсивных алгоритмов.

void Run(Point p)

//обход сверху вниз

{

if(p!=null)

{

<обработка p.data>

Run(p.left);//переход к левому поддереву

Run(p.right);//переход к правому поддереву

}

}

Если в качестве операции обработки узла поставить операцию вывода информационного поля, то мы получим функцию для печати дерева.

/\*рекурсивная функция для печати дерева по уровням с обходом слева направо\*/

static void ShowTree(Point p, int l)

{

if(p!=null)

{

ShowTree(p.left,l+3);//переход к левому поддереву

//формирование отступа

for (int i = 0; i < l; i++) Console.Write(" ");

Console.WriteLine(p.data);//печать узла

ShowTree(p.right,l+3);//переход к правому поддереву

}

}

### Формирование дерева

//построение дерева поиска

//формирование первого элемента дерева

static Point first(int d)

{

Point p=new Point(d);

return p;

}

//добавление элемента d в дерево поиска

static Point Add(Point root, int d)

{

Point p = root; //корень дерева

Point r = null;

//флаг для проверки существования элемента d в дереве

bool ok = false;

while (p!=null && !ok)

{

r = p;

//элемент уже существует

if (d == p.data) ok = true;

else

if (d < p.data) p = p.left; //пойти в левое поддерево

else p = p.right; //пойти в правое поддерево

}

if (ok) return p;//найдено, не добавляем

//создаем узел

Point NewPoint = new Point(d);//выделили память

// если d<r.key, то добавляем его в левое поддерево

if (d < r.data) r.left = NewPoint;

// если d>r.key, то добавляем его в правое поддерево

else r.right = NewPoint;

return NewPoint;

}

//построение идеально сбалансированного дерева

static Point IdealTree(int size, Point p)

{

Point r;

int nl, nr;

if(size==0){p=null;return p;}

nl=size/2;

nr=size-nl-1;

int number=Convert.ToInt32(Console.ReadLine());

r = new Point(d);

i++;

r.left=IdealTree(nl,r.left);

r.right=IdealTree(nr,r.right);

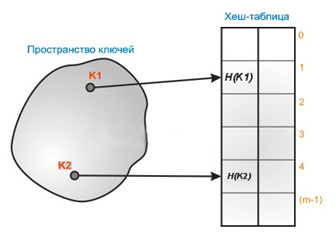
return r;

}

### Хеш-таблицы

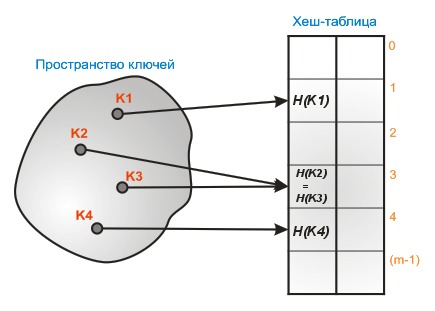
*Хеширование* – процесс превращения некоторых данных на входе в некоторую битовую последовательность фиксированной длины на выходе. Эта строка позволяет отличить ее от большинства других строк, так же полученных путем хеширования. Другое название таких преобразований – хеш-функции или функции свертки, а результат работы – хеш или хеш-код.

Результат работы хеш-функции помогает различать одни данные от других, приводя в сравнение только значения функций хеширования, обработавших соответствующие данные, что очень удобно, так как конечные данные весят гораздо меньше информации, нежели данные, которым они соответствуют.

  
Рисунок 5– Пример работы ключей

Хеширование используется для создания ассоциативных массивов, в которых индексами являются не числа, а произвольные значения. Оно применяется в поиске дубликатов в текстах, то есть поиска фрагментов, у которых имеется одинаковое значения хеш-функций. Кроме того, хеширование часто используется для хранения паролей; для создания уникальных идентификаторов, например, если для файла требуется уникальное название, можно посчитать результат обработки хешированием этого файла и сделать его названием. Оно также очень важно для подсчета контрольной суммы текста. Допустим, пользователю необходимо передать по сети какой-либо текст. Вместе с текстом передается контрольная сумма, которая, когда будет принята, сверяется с исходной. И, если суммы не совпадут, это будет означать, что при передаче текста возникли некоторые ошибки.

Однако, нередко нескольким различным данным, соответственно, различной длины на входе могут соответствовать одинаковые данные на выходе. Ситуации, когда разные данные имеют одинаковое значение хеш-функций, называются *коллизиями* (рисунок 6). При этом алгоритм хеширования должен стремиться к тому, чтобы разные данные имели разные значения. В редких случаях удается избежать коллизий вообще.

  
Рисунок 6 – Пример коллизии: ключи К2 и K3 в результате хеширования приобрели одинаковые значения хеш-функций

*Хеш-таблица* – структура данных, реализующая интерфейс ассоциативного массива. В ней хранятся пары, состоящие из ключа и хешированного ключа. Она позволяет добавлять новые пары, искать и удалять пары по ключу. Хеш-таблица формируется в определенном порядке хеш-функцией.

Хеш-таблицы часто применяются в базах данных, и, особенно, в языковых процессорах типа компиляторов и ассемблеров, где они повышают скорость обработки таблицы идентификаторов.

Хеш-таблицы должны соответствовать следующим свойствам.

* Прежде чем выполнять операции в хеш-таблице, вычисляется хеш-функция от ключа, результат которой является индексом в исходном массиве.
* Коэффициент заполнения хеш-таблицы - количество хранимых элементов массива, деленное на число возможных значений хеш-функции. Является важным параметром, от которого зависит среднее время выполнения операций.
* Среднее время, которое нужно затратить на поиск, вставку или удаление должно занимать О(1). Стоит отметить, что возможные аппаратные затраты на перестройку индекса хеш-таблицы, связанную с увеличением размера хеш-таблицы и добавлением в нее новой пары, учитываться не будут.

Принято считать, что хорошей является такая хеш-функция, которая удовлетворяет следующим условиям. Функция должна:

* быть простой с вычислительной точки зрения (зависит от характеристик компьютера).
* распределять ключи в хеш-таблице наиболее равномерно (зависит от значений данных).
* стремиться уменьшить число коллизий.

Функция не должна отображать какую-либо связь между значениями ключей в связь между значениями адресов.

Пример создания хеш-таблицы с цепочками

//элемент хеш-таблицы

class LPoint

{

public int key;//ключ

public string value;//значение

public LPoint next;//ссылка на следующий элемент

static Random rnd = new Random();

public LPoint(string s)

{

value = s;

key = GetHashCode();

next = null;

}

public override string ToString()

{

return key + ":" + value.ToString();

}

public override int GetHashCode()

{

int code = 0;

foreach (char c in value)

code += (int)c;

return code;

}

}//конец класса LPoint

//хеш-таблица

class **HTable**

{

public LPoint[] table;

public int Size;

public HTable(int size=10)

{

Size = size;

table = new LPoint[Size];

}

public bool **Add** (string s)

{

LPoint point = new LPoint(s);

if (s == null) return false;

int index = Math.Abs(point.GetHashCode()) % Size;

if (table[index] == null)

table[index] = point;

else

{

Point cur = table[index];

if (string.Compare (cur.ToString(), point.ToString()) == 0)return false;

while (cur.next != null)

{

if (string.Compare (cur.ToString(), point.ToString()) == 0) return false;

cur=cur.next;

}

cur.next = point;

}

return true;

}

public void **Print()**

{

if (table == null) { Console.WriteLine("Таблица пустая!"); return; }

for (int i = 0; i < Size; i++)

{

if (table[i] == null) Console.WriteLine(i + " : ");

else{

Console.Write(i + " : ");

LPoint p = table[i];

while (p != null)

{

Console.Write(p.ToString() + "\t");

p = p.next;

}

Console.WriteLine();

}

public bool **FindPoin**t(string str)

{

LPoint lp = new LPoint(str);

int code = Math.Abs(lp.GetHashCode()) % Size;

if (String.Compare (table[code].value, str) == 0)

return true;

lp = table[code];

while (lp != null)

{

if (string.Compare (lp.value, str) == 0)return true;

lp = lp.next;

}

return false;

}

public string DelPoint(string str)

{

LPoint lp = new LPoint(str);

int code = Math.Abs(lp.GetHashCode()) % Size;

lp = table[code];

if (table[code] == null) return null;

if (table[code]!=null && String.Compare (table[code].value, str) == 0)

{

lp = table[code];

table[code] = table[code].next;

return lp.value;

}

while (lp.next != null && (string.Compare(lp.next.value, str) != 0))

lp = lp.next;

if (lp.next!=null)

{

str=lp.next.value;

lp.next = lp.next.next;

return str;

}

return null;

}

}//конец класса **HTable**

static void Main(string[] args)

{

string[] arr = new string[22];

arr[0] = "aaa";

arr[1] = "aab"; arr[2] = "aba";

arr[3] = "baa"; arr[4] = "abb";

arr[5] = "bba"; arr[6] = "aac";

arr[7] = "caa"; arr[8] = "aca";

arr[9] = "abc"; arr[10] = "bca";

arr[11] = "acc"; arr[12] = "cca";

arr[13] = "bcb"; arr[14] = "bbc";

arr[15] = "ccb"; arr[16] = "bbb";

arr[17] = "ccc"; arr[18] = "acb";

arr[19] = "cba"; arr[20] = "ccb";

arr[21] = "acc";

HTable ht = new HTable();

foreach (string s in arr)

ht.Add(s);

ht.Print();

string findStr;

do

{

Console.WriteLine("Введите строку для поиска");

findStr = Console.ReadLine();

if (findStr == "end") continue;

if (ht.FindPoint(findStr)) Console.WriteLine("Строка найдена");

else Console.WriteLine("Строка не найдена");

} while (findStr != "end");

}

### Обобщенные коллекции. Пространство имен System.Collections.Generic.

Многие алгоритмы не зависят от типов данных, с которыми они работают. Простейшими примерами могут служить сортировка и поиск. Возможность отделить алгоритмы от типов данных предоставляют *классы-прототипы* (generics) — классы, имеющие в качестве параметров типы данных. Чаще всего эти классы применяются для хранения данных, то есть в качестве контейнерных классов, или коллекций.

Начиная с версии .NET 2.0, язык программирования С# был расширен поддержкой средства, которое называется ***обобщением (generic)***. Вместе с ним библиотеки базовых классов пополнились совершенно новым пространством имен, связанным с коллекциями — System.Collections.Generic.

Общая форма объявления обобщенного класса:

*class имя\_класса<список\_параметров\_типа> {...}*

Ссылка на обобщенный класс:

*имя\_класса<список\_аргументов\_типа> имя\_переменной =*

*new имя\_класса<список\_параметров\_типа> (список\_аргументов\_конструктора);*

Во вторую версию библиотеки .NET добавлены параметризованные коллекции для представления основных структур данных, применяющихся при создании программ, — стека, очереди, списка, словаря и т. д. Эти коллекции, расположенные в пространстве имен **System.Collections.Generic**, дублируют аналогичные коллекции пространства имен **System.Collections.**

**Параметризованные коллекции библиотеки .NET**

|  |  |
| --- | --- |
| Класс- прототип | Обычный класс |
| Comparer <T> | Comparer |
| Dictionary <K,T> | HashTable |
| LinkedList <T> | - |
| List <T> | ArrayList |
| Queue<T> | Queue |
| SortedDictionary <K,T> | SortedList |
| Stack <T> | Stack |

**Параметризованные интерфейсы библиотеки .NET.**

|  |  |
| --- | --- |
| Параметризованный интерфейс | Обычный интерфейс |
| ICollection<T> | ICollection |
| IComparable<T> | IComparable |
| IDictionary<Т> | IDictionary |
| IEnumerable<T> | I Enumerable |
| IEnumerator<T> | I Enumerator |
| IList<T> | IList |

Параметром класса-прототипа является тип данных, с которым он работает (T – это тип, который является параметром коллекции, т. е. вместо него можно подставить любой другой тип данных).

### Доступ к элементам коллекции с помощью нумератора

**Нумератор –** это объект, который используется для перебора коллекции циклом foreach. Нумератор представляет собой «курсор», который перемещается по коллекции только в одну сторону.

**Нумератор** реализует интерфейс IEnumerator. В интерфейсе IEnumerator определены:

* object Current { get; }- свойство, которое позволяет получить элемент, соответствующий текущему значению нумератора. Поскольку свойство Current предназначено только для чтения, нумератор можно использовать только для считывания значения объекта в коллекции, а не для его модификации.
* bool MoveNext() – метод, при каждом обращении к которому текущая позиция нумератора перемещается к следующему элементу коллекции. Метод возвращает значение true , если к следующему элементу можно получить доступ, или значение false, если достигнут конец коллекции. До выполнения первого обращения к методу MoveNext () значение свойства Current неопределено.
* void Reset() – метод, который устанавливает нумератор в начало коллекции. После вызова метода Reset() нумерация элементов начнется с начала коллекции, и для доступа к первому ее элементу необходимо вызвать метод MoveNext ().

Класс коллекции, который реализует интерфейс IDictionary, предназначен для хранения пар ключ/значение. Для опроса элементов в такой коллекции используется интерфейс IDictionaryEnumerator, а не IEnumerator. Класс DictionaryEnumerator является производным от класса IEnumerator и дополнительно определяет три свойства.

* DictionaryEntry Entry { get; }- позволяет получить следующую пару ключ/значение в форме структуры типа DictionaryEntry.
* object Key { get; } - позволяет получить прямой доступ к ключу.
* object Value { get; }- позволяет получить прямой доступ к значению.

Интерфейс IDictionaryEnumerator используется подобно обычному нумератору за исключением того, что текущие значения элементов здесь можно получить с помощью свойств Entry, Key или Value, а не свойства Current. Таким образом, реализовав IDictionaryEnumerator-нумератор, вы должны вызвать метод MoveNext(), чтобы получить первый элемент. Остальные элементы коллекции опрашиваются путем последующих вызовов метода MoveNext (). Когда все элементы коллекции будут исчерпаны, метод MoveNext () возвратит значение false.

**Перечислимый объект**– это объект, по которому движется нумератор. Перечислимый объект (коллекция) либо реализует интерфейс IEnumerable, либо содержит метод GetEnumerator, который возвращает нумератор.

Интерфейсы IEnumerator и IEnumerable почти всегда реализуются в сочетании со своими обобщенными версиями:

public interface IEnumerator<T>:IEnumerator, IDisposable

{

T Current {get;}

}

public interface IEnumerable<T>:IEnumerable

{

IEnumerator<T> GetEnumerator();

}

Например, можно реализовать локальный класс MyNumerator, объект которого будет выполнять перебор коллекции. Данный класс должен реализовывать обобщенный интерфейс IEnumerator<T> для обобщенных коллекций (интерфейс IEnumerator для необобщенных коллекций).

class MyEnumerator : IEnumerator<T>

{

Point<T> beg;//начало коллекции

Point<T> current;//текущий элемент коллекции

public MyEnumerator(MyGenericCollection<T> c)//конструктор

{

beg = c.beg;

current = c.beg;

}

/\*свойство, которое возвращает информационное поле текущего элемента, реализует свойство интерфейса IEnumerator<T>\*/

public T Current

{

get { return current.Data; }

}

/\*свойство, которое реализует интерфейс IEnumerator и преобразует Т в object\*/

object IEnumerator.Current

{

get { return Current; }

}

/\*метод для перехода к следующему элементу списка, реализует интерфейс IEnumerator \*/

public bool MoveNext()

{

if (current.next == null)//конец списка

{

Reset();//переход на начало коллекции

return false;

}

else

{

//переход к следующему элементу коллекции

current = current.next;

return true;

}

}

/\*метод, который ставит текущий элемент на начало коллекции, реализует интерфейс IEnumerator \*/

public void Reset()

{

current = this.beg;

}

/\*метод для удаления ресурсов нумератора, реализует интерфейс IEnumerator<T> \*/

public void Dispose() { }

}

В самой коллекции необходимо реализовать интерфейс IEnumerable<T>:

/\*метод необобщенного интерфейса IEnumerable, который возвращает объект-нумератор\*/

IEnumerator IEnumerable.GetEnumerator()

{

return GetEnumerator();

}

/\*метод обобщенного интерфейса IEnumerable<T>, который возвращает обобщенный объект-нумератор \*/

public IEnumerator<T> GetEnumerator()

{

return new MyEnumerator(this);

}

Метод foreach получит объект-нумератор с помощью метода IEnumerator<T> GetEnumerator() и с его помощью будет перебирать элементы коллекции.

foreach (Person p2 in c)//Person подставляется вместо Т

Console.WriteLine(p2.ToString());

В версию C#2.0 были введены средства, облегчающие выполнение перебора в объекте — итераторы. Итератор представляет собой блок кода, задающий последовательность перебора элементов объекта. На каждом проходе цикла foreach выполняется один шаг итератора, заканчивающийся выдачей очередного значения.

Выдача значения выполняется с помощью ключевого слова yield.

Итератор использует две специальных инструкции:

* yield return: определяет возвращаемый элемент;
* yield break: указывает, что последовательность больше не имеет элементов.

Таким образом, при получении каждого элемента в цикле foreach будет срабатывать оператор yield return, который будет возвращать один элемент и запоминать текущую позицию.

С помощью итераторов реализовать интерфейс IEnumerable<T> для коллекции можно следующим образом:

//реализовать необобщенный нумератор

IEnumerator IEnumerable.GetEnumerator()

{

return GetEnumerator();

}

//реализовать обобщенный нумератор

public IEnumerator<T> GetEnumerator()

{

Point<T> current = beg;

while (curr != null)

{

yield return current.Data;

current = current.next;

}

}

## Постановка задачи

### 2.1. Задание 1.

1. Сформировать двунаправленный список, в информационное поле записать объекты из иерархии классов лабораторной работы №10.
2. Распечатать полученный список.
3. Выполнить обработку списка в соответствии с заданием.
4. Распечатать полученный список.
5. Выполнить клонирование списка, показать что под объекты, хранящиеся в информационном поле выделена разная память
6. Удалить список из памяти.

### 2.2. Задание 2.

1. Сформировать идеально сбалансированное бинарное дерево, в информационное поле записать объекты из иерархии классов лабораторной работы №10.
2. Распечатать полученное дерево.
3. Выполнить обработку дерева в соответствии с заданием, вывести полученный результат.
4. Преобразовать идеально сбалансированное дерево в дерево поиска[[1]](#footnote-1). Информационное поле склонировать.
5. Распечатать полученное дерево.
6. Удалить дерево из памяти.

### 2.3. Задание 3

1. Создать хеш-таблицу и заполнить ее элементами.
2. Выполнить поиск элемента в хеш-таблице
3. Удалить найденный элемент из хеш-таблицы.
4. Выполнить поиск элемента в хеш-таблице
5. Показать, что будет при добавлении элемента в хеш-таблицу, если в таблице уже находится максимальное число элементов (для метода открытой адресации, для метода цепочек просто показать добавление в таблицу).

### 2.4. Задание 4

Реализовать **обобщенную** коллекцию, указанную в варианте. Для этого:

1. Реализовать конструкторы:

* public MyCollection() - предназначен для создания пустой коллекции.
* public MyCollection (int capacity*)* - создает пустую коллекцию с начальной емкостью, заданной параметром capacity. **(?)**
* public MyCollection (MyCollection c) - служит для создания коллекции, которая инициализируется элементами и емкостью коллекции, заданной параметром с.

1. Для всех коллекций реализовать:
   * методы для добавления одного или нескольких **(?)** элементов в коллекцию;
   * методы для удаления одного или нескольких **(?)** элементов из коллекции;
   * метод для поиска элемента по ключу и/или по значению;
   * метод для глубокого клонирования коллекции (вместе с элементами);
   * метод для поверхностного копирования;
   * индексатор для получения/изменения элемента по индексу/ключу.
2. Для всех коллекций реализовать интерфейсы IEnumerable, ICollection.
3. Написать демонстрационную программу, в которой создаются коллекции, и демонстрируется работа всех реализованных методов, в том числе, **перебор коллекции циклом foreach.**

При работе с коллекцией использовать объекты из иерархии классов, разработанной в работе №10.

## Варианты

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| №  варианта | Двунаправленный список | Бинарное дерево | Хеш-таблица **(?)** | Коллекция (можно выбрать) |
| 1 | Добавить в список элемент с заданным номером. | Найти количество элементов с заданным ключом (информационное поле ссылается на объект, который является ключом). | Открытая адресация, поиск и удаление по ключу | Однонаправленный список (6 баллов)  Однонаправленный кольцевой список (7 баллов)  Двунаправленный список (6 баллов)  Двунаправленный кольцевой список (7 баллов)  Стек на базе однонаправленного списка (7 баллов)  Стек на базе двунаправленного списка (7 баллов)  Очередь на базе однонаправленного списка (7 баллов)  Хеш-таблица (открытая адресация) (8 баллов)  Хеш-таблица (метод цепочек) (8 баллов)  Дерево поиска (9 баллов)  Дерево поиска + идеально сбалансированное дерево (10 баллов) |
| 2 | Добавить в список элементы с номерами 1, 3, 5 и т. д. | Найти максимальный элемент в дереве (например элемент с максимальным возрастом). | Метод цепочек, поиск и удаление по ключу |
| 3 | Добавить в список элемент после элемента с заданным информационным полем (например, с заданным именем). | Найти количество листьев в дереве. | Открытая адресация, поиск и удаление по ключу |
| 4 | Добавить в список элемент с заданным номером. | Найти минимальный элемент в дереве (например элемент с минимальным возрастом). | Метод цепочек, поиск и удаление по ключу |
| 5 | Удалить из списка все элементы с заданными информационными полями (например, с заданным именем). | Найти высоту дерева. | Открытая адресация, поиск и удаление по ключу |
| 6 | Удалить из списка все элементы с четными номерами (2, 4, 6 и. т. д.). | Найти среднее арифметическое элементов дерева (например, средний возраст). | Метод цепочек, поиск и удаление по ключу |
| 7 | Удалить из списка первый элемент с заданным информационным полем (например, с заданным именем). | Найти количество элементов дерева, у которых поле (например, имя) начинается с заданного символа. | Открытая адресация, поиск и удаление по ключу |
| 8 | Удалить из списка последний элемент с заданным информационным полем (например, с заданным именем). | Найти количество элементов с заданным ключом (информационное поле ссылается на объект, который является ключом). | Метод цепочек, поиск и удаление по ключу |
| 9 | Добавить в список элементы с номерами 1, 3, 5 и т. д. | Найти максимальный элемент в дереве (например элемент с максимальным возрастом). | Открытая адресация, поиск и удаление по ключу |
| 10 | Добавить в список элемент после элемента с заданным информационным полем (например, с заданным именем). | Найти количество листьев в дереве. | Метод цепочек, поиск и удаление по ключу |
| 11 | Добавить в список элемент с заданным номером. | Найти минимальный элемент в дереве (например элемент с минимальным возрастом). | Открытая адресация, поиск и удаление по ключу |
| 12 | Добавить в список элемент с заданным номером. | Найти высоту дерева. | Метод цепочек, поиск и удаление по ключу |
| 13 | Удалить из списка все элементы с заданными информационными полями (например, с заданным именем). | Найти среднее арифметическое элементов дерева (например, средний возраст). | Открытая адресация, поиск и удаление по ключу |
| 14 | Удалить из списка последний элемент с заданным информационным полем (например, с заданным именем). | Найти количество элементов с заданным ключом (информационное поле ссылается на объект, который является ключом). | Метод цепочек, поиск и удаление по ключу |
| 15 | Удалить из списка все элементы с четными номерами (2, 4, 6 и. т. д.). | Найти количество элементов дерева, у которых поле (например, имя) начинается с заданного символа. | Открытая адресация, поиск и удаление по ключу |
| 16 | Удалить из списка первый элемент с заданным информационным полем (например, с заданным именем). | Найти минимальный элемент в дереве (например элемент с минимальным возрастом). | Метод цепочек, поиск и удаление по ключу |
| 17 | Добавить в список элемент с заданным номером. | Найти количество листьев в дереве. | Открытая адресация, поиск и удаление по ключу |
| 18 | Добавить в список элемент с заданным номером. | Найти минимальный элемент в дереве (например элемент с минимальным возрастом). | Метод цепочек, поиск и удаление по ключу |
| 19 | Добавить в список элементы с номерами 1, 3, 5 и т. д. | Найти высоту дерева. | Открытая адресация, поиск и удаление по ключу |
| 20 | Добавить в список элемент после элемента с заданным информационным полем (например, с заданным именем). | Найти среднее арифметическое элементов дерева (например, средний возраст). | Метод цепочек, поиск и удаление по ключу |
| 21 | Удалить из списка все элементы с заданными информационными полями (например, с заданным именем). | Найти количество элементов дерева, у которых поле (например, имя) начинается с заданного символа. | Открытая адресация, поиск и удаление по ключу |
| 22 | Удалить из списка все элементы с четными номерами (2, 4, 6 и. т. д.). | Найти количество элементов с заданным ключом (информационное поле ссылается на объект, который является ключом). | Метод цепочек, поиск и удаление по ключу |
| 23 | Удалить из списка первый элемент с заданным информационным полем (например, с заданным именем). | Найти количество листьев в дереве. | Открытая адресация, поиск и удаление по ключу |
| 24 | Удалить из списка последний элемент с заданным информационным полем (например, с заданным именем). | Найти минимальный элемент в дереве (например элемент с минимальным возрастом). | Метод цепочек, поиск и удаление по ключу |
| 25 | Добавить в список элементы с номерами 1, 3, 5 и т. д. | Найти максимальный элемент в дереве (например элемент с максимальным возрастом). | Открытая адресация, поиск и удаление по ключу |

## 6. Методические указания

1. Интерфейс реализовать в виде текстового меню.
2. Структуры данных (список, дерево, хеш-таблицу) в 1, 2, 3 задании реализовать как обобщенный класс.
3. Метод GetEnumerator() в 4 задании можно реализовать одним из двух способов: либо с помощью итератора yield, либо с помощью нумератора на основе интерфейса IEnumerator.
4. Для обобщенных классов использовать обобщенные интерфейсы.
5. При копировании структур данных должно выполняться глубокое клонирование (вместе с элементами).

## 7. Содержание отчета

1. Постановка задачи (общая и конкретного варианта)
2. Диаграмма классов для каждой части работы (из VS или Visio)
3. Алгоритмы построения динамических структур данных (для 1, 2 и 3 задачи).
4. Код программы
5. Код unit-тестов (или тесты в Excel)
6. Анализ покрытия кода тестами (из VS или в Excel)

1. За реализацию дерева поиска, которое будет одновременно идеально-сбалансированным ставятся дополнительные баллы [↑](#footnote-ref-1)