**Лабораторная работа**

**Динамические структуры и классы коллекций**

**Задание.** Составить программу на языке C#, содержащую библиотеку классов согласно индивидуальному варианту (стек, очередь, дерево) и используя встроенные классы коллекций (ArrayList, Stack, Queue). Программа выполняет следующие действия над списком:

1. добавление элемента по ключу;
2. удаление элемента по ключу;
3. выполнение запроса;
4. просмотр списка;

 На примере продемонстрировать состояние списка после выполнения операций

Вариант имеет вид ns, где

n - вид структуры и класса коллекций (1-стек и Stack, 2 –очередь и Queue, 3- дерево и ArrayList)). Можете выбрать сами

s – вид запроса;

к полям структуры добавляется ключ – числовое поле.

**Индивидуальные варианты:**

8. Структура **“Книги”** (номер книги, наименование книги, год издания, цена одного экземпляра, фамилия автора):

S= определение цены книги с определенным номером

**Стек, алгоритмы основных операций**

**Стек** представляет собой объект, работа с которым — добавление и удаление — осуществляется по принципу LIFO (last-in first-out — последним пришел — первым ушел). Добавление объектов в стек может осуществляться в любой момент, удаляется же лишь объект, добавленный последним. Поведение стека напоминает работу со стопкой тарелок в столовой самообслуживания: чистые тарелки помещают сверху (добавление в вершину), а берут тарелки для использования тоже сверху (удаление из вершины). Стеки относятся к базовым структурам данных, используемым во многих приложениях.

Стек является абстрактным типом данных, который поддерживает следующие основные функции:

push (о): помещает объект о на вершину стека.

pop (): удаляет объект из стека и возвращает новый верхний объект стека; если стек пуст, выдается сообщение об ошибке.

Кроме того, стек выполняет следующие дополнительные функции:

size (): возвращает число объектов в стеке.

isEmpty (): возвращает логическое значение, подтверждающее, что стек пуст.

top (): возвращает верхний объект в стеке, не удаляя его; если стек пуст, выдается сообщение об ошибке.

**Реализация стека на основе однонаправленного связного списка**

Используем переменную ссылочного типа top для обращения к началу списка (имеет нулевое значение при пустом списке). При добавлении в стек нового элемента е создается новый узел v, из v устанавливается ссылка на е, и v добавляется в начало списка. Аналогично при извлечении элемента из стека узел удаляется из начала списка и возвращается его элемент. Таким образом выполняются все добавления и удаления элементов в начале списка.

*public class LinkedStack {*

*private Node top; // обращается к головному узлу*

*private int cnt ;// число элементов стека*

 *public Node Top*

 *{get { return top; }*

 *set { top = value; } }*

 *public int Cnt*

 *{ get { return cnt; }*

 *set { cnt= value; } }*

*public LinkedStack() { // инициализирует пустой стек*

*Top = null; Cnt = 0; }*

*public int size() { return Cnt; }*

*public bool isEmpty() { if (Top == null) return true;*

*return false; }*

*public void push(int elem) { Node v = new Node(elem); // создает новый узел*

*v.Next=Top; // связывает новый узел*

*Top = v;*

*Cnt++; }*

*public int topp() { return Top.Data; }*

*public int pop() { int temp = Top.Data;*

*Top = Top.Next;*

*Cnt--; return temp; }}*

Узлами списка являются объекты класса Node.

**Двоичное дерево, алгоритмы основных операций**

Дерево — это абстрактный тип данных для иерархического хранения элементов. За исключением элемента во главе дерева, каждый элемент структуры имеет родителя (parent element) и ноль или более дочерних элементов (children element). Для придания древовидной структуре большей наглядности ее элементы изображаются в виде овалов или прямоугольников, а связи между ними — в виде прямых линий.

Обычно головной элемент дерева называется корнем (root), хотя он является самым верхним элементом структуры, а остальные расположены под ним.

Если узел u является родителем узла v, то можно говорить, что v является дочерним узлом u. Два узла, выступающие дочерними элементами по отношению к одному и тому же родителю, будут называться сестрами/братьями. Предком узла может быть либо родительский узел, либо предок родителя этого узла. И наоборот, можно сказать, что v является потомком узла u, если u является предком v.

Бинарным деревом (binary tree) называется упорядоченное дерево, в котором каждый из узлов имеет максимум два дочерних элемента. дочерних элемента. Дочерние элементы в таких узлах имеют маркировку «левый» и «правый» (left child и right child). Эти элементы упорядочены таким образом, что левый меньше правого.

Естественным способом реализации бинарного дерева является использование связной структуры. При таком подходе каждый узел v дерева Т представляется объектом со ссылками на элемент, хранящийся в v, и на объекты, связанные с дочерними и родительскими элементами узла v (рис. 44).



Рис. 44. Представление двоичного дерева

Определим два класса: узел дерева TreeNode и дерево SearchTree

 *public class TreeNode*

 *{ private TreeNode left, right, parent;*

 *private int dat;*

 *public TreeNode Left*

 *{ get { return left; }*

 *set { left = value; } }*

 *public TreeNode Right*

 *{ get { return right; }*

 *set { right = value; } }*

 *public TreeNode Parent*

 *{ get { return parent; }*

 *set { parent = value; } }*

 *public int Dat*

 *{ get { return dat; }*

 *set { dat = value; } }*

 *public TreeNode(int v, TreeNode p)*

 *{ Dat = v; Left = null; Right = null; Parent = p; } }*

Конструктор класса формирует узел с определенными данными, заданным предком и пустыми потомками.

 *public class SearchTree*

 *{*

 *private data[] rez;*

 *private TreeNode root; private int num;*

*public TreeNode Root*

*{ get { return root; }*

*set { root = value; } }*

*public int Num*

*{*

 *get { return num; }*

 *set { num = value; }*

*}*

*public data[] Rez*

*{*

 *get { return rez; }*

 *set { rez = value; }*

*}*

*public SearchTree() { Root = null; Num = 0; Rez = new data[10]; }*

*public bool isEmpty() { return (Root == null); }*

*public TreeNode find(int el)*

*{*

 *TreeNode current = Root;*

*while (current != null)*

*if (el == current.Dat.shifr)*

*return (current);*

*else*

*current = el <current.Dat.shifr ? current.Left :current.Right;*

*return (null); }*

*public data prisv(data r)*

*{ data r1; r1 = r; return r1; }*

*public data[] inorder(TreeNode n)*

*{ data b;*

*b.shifr = 0;7*

*b.name = "";*

 *if (n != null)*

*{inorder(n.Left);*

*b = prisv(n.Dat);*

 *Rez[Num]=b; Num++;*

*inorder(n.Right);*

*}*

 *else { }return Rez;*

*}*

*public TreeNode insert(data el)*

*{ TreeNode x, current, parent;*

*Num=0;*

*current = Root;*

*parent = null;*

*while (current != null)*

 *{ if (el.shifr == current.Dat.shifr) return (current);*

*parent = current;*

*current = el.shifr <current.Dat.shifr ? current.Left :current.Right; }*

 *x = new TreeNode(el, parent);*

*if (parent != null)*

*if (x.Dat.shifr<parent.Dat.shifr)*

*parent.Left=x;*

*else*

*parent.Right=x;*

*else*

 *Root = x;*

*return (x); }*

*public void remove(TreeNode z)*

 *{ TreeNode x, y, t;*

*if (z == null) return;*

*if (z.Left == null || z.Right == null)*

 *y = z;*

*else*

 *{ y = z.Right;*

*while (y.Left != null) y = y.Left; }*

*if (y.Left != null)*

 *x = y.Left;*

*else*

 *x = y.Right;*

*if (x != null) x.Parent=y.Parent;*

*if (y.Parent != null)*

*{ t = y.Parent;*

*if (y == t.Left)*

*t.Left=x;*

*else*

*t.Right=x; }*

*else*

 *Root = x;*

*if (y != z)*

 *{ y.Left=z.Left;*

*if (y.Left != null)*

*{ t = y.Left; t.Parent=y; }*

*y.Right=z.Right;*

*if (y.Right != null)*

*{ t = y.Right; t.Parent=y; }*

*y.Parent=z.Parent;*

*if (z.Parent != null)*

*{ t = z.Parent; if (z == t.Left)*

*t.Left=y;*

*else*

*t.Right=y; }*

*else*

 *Root = y;*

 *z = null; }*

 *else y = null; } }}*

Класс содержит поле root, которое указывает на корень двоичного дерева поиска (объект класса TreeNode). Конструктор класса формирует двоичное дерево с пустым корнем. Дерево пусто, если поле root пустое. Чтобы найти заданный элемент dat, поиск начинается с корня и затем продолжается вниз до узла, содержащего dat. В каждом узле n сравнивается el с элементом dat узла n. Если el меньше, чем dat узла n, то поиск продолжается, начиная с левого потомка узла n, если el больше, чем dat узла n, то поиск продолжается, начиная с правого потомка n, в противном случае возвращается адрес узла n (и задача решена). Путь от корневого узла вниз до el называется *путем поиска* для el. Обход двоичного дерева — это процесс, при котором каждый узел посещается только один раз. Метод inorder выполняет специальную форму обхода, известную как *симметричный обход*.

Стратегия заключается сначала в симметричном обходе левого поддерева, затем посещения корня и потом в симметричном обходе правого поддерева. Узел посещается путем добавления его данных в поле rez – массив данных. Размер этого массива –свойство num.

Для включения нового элемента в двоичное дерево вначале нужно определить его точное положение — а именно предок, который должен быть заменен путем отслеживания пути поиска элемента, начиная с корня. Кроме сохранения указателя n на текущий узел мы будем хранить указатель р на предка узла n. Таким образом, когда n достигнет некоторого узла, р будет указывать на узел, который должен стать предком нового узла.

Удаление узла, у которого есть не более чем один непустой потомок, является сравнительно простой задачей — устанавливается ссылка от предка узла на потомка. Однако ситуация становится гораздо сложнее, если у подлежащего удалению узла есть два непустых потомка: предок узла может быть связан с одним из потомков, а что делать с другим? Решение может заключаться не в удалении узла из дерева, а скорее в замене элемента, содержащегося в нем, на потомка этого элемента, а затем в удалении узла-потомка.

**Очередь, алгоритмы основных операций**

 Еще одной базовой структурой данных является **очередь**. Эта структура данных аналогична стеку, так как очередь объединяет объекты, работа с которыми — добавление и удаление — осуществляется по принципу FIFO (first-in first-out) (первым пришел — первым ушел). Добавление объектов в очередь может осуществляться в любое время, однако удаленным может быть лишь объект, который был добавлен первым. Говорят, что элементы добавляются в очередь с конца, а удаляются с начала. Можно сравнить очередь данных с очередью людей в любом магазине: они встают в очередь с конца, а тот, кто был первым в очереди, обслуживается.

Очередь S является абстрактным типом данных, который поддерживает две основные функции:

enqueue (о): помещает объект о в конец очереди.

Вход: объект; Выход: нет.

dequeue (): производит удаление и возвращает объект из начала очереди; если очередь пуста, выдается сообщение об ошибке.

Вход: нет; Выход: объект.

Кроме того, очередь выполняет следующие дополнительные функции:

size (): возвращает число объектов в очереди.

Вход: нет; Выход: целое число.

isEmpty (): возвращает логическое значение, подтверждающее, что очередь пуста.

Вход: нет; Выход: логическое значение.

front (): возвращает первый объект в очереди, не удаляя его; если очередь пуста, выдается сообщение об ошибке.

Вход: нет; Выход: объект.

Рассмотрим реализацию очереди с помощью массива Q для хранения элементов очереди, длина которого N, например, N = 1000. Так как основной характеристикой очереди является добавление и удаление объектов по принципу FIFO, необходимо определить способ наблюдения за началом и концом очереди.

Для того чтобы избежать перемещения объектов в Q, применим две переменные f и r

• f является индексом ячейки Q, в которой хранится первый элемент очереди (кандидат на удаление при выполнении метода dequeue) при условии, что очередь не является пустой (в этом случае f= r).

• r является индексом следующей доступной ячейки массива Q.

Изначально присваиваем f=г=0, что означает пустую очередь. Теперь, после удаления элемента из начала очереди просто увеличиваем значение f на 1 для получения индекса следующей ячейки массива. Если добавляем новый элемент, то увеличиваем значение r на 1 для получения индекса следующей доступной ячейки массива Q. Для обеспечения возможности использования всего массива Q предположим, что индексы f и г «охватывают» конечные точки массива Q. В этом случае Q рассматривается как «круговой» массив, который продолжается от Q[0] до Q[N- 1], а затем снова возвращается к Q[0].

Реализация подобного представления Q довольно проста. При каждом увеличении значения f и r на 1 необходимо записать эту операцию в виде (f + 1) mod N или (r + 1) mod N соответственно, где mod N обозначает операцию деления по модулю, которая вычисляется путем получения остатка от деления. Размер очереди вычисляется с помощью выражения (N-f+r) mod N, которое позволяет получить правильный результат как в случае нормальной конфигурации, где f<г, так и при «круговой» конфигурации, где r<f.

Алгоритмы функций:

**size():**

1. Возврат (N - f+ r) mod N

**isEmpty():**

1. Возврат (f = r)

**front():**

1. Возврат Q[f]

**dequeue():**

1. temp =Q[f]

2. Q[f]=0

3.f= (f+ 1) mod N

4. Возврат temp

**enqueue(o):**

1. О[t]= о

r =(r + 1) mod N

**Реализация очереди на основе однонаправленного связного списка**

Очередь также можно реализовать на основе однонаправленного связного списка. Для наибольшей эффективности поместим начало очереди, где можно только удалять элементы, в начало списка, а конец очереди, где можно добавлять элементы, — в хвост списка. Не следует забывать о необходимости сохранения ссылок на начальный и конечный узлы списка.

*class Que*

 *{*

 *private Node first, last;*

 *private int size;*

 *public int Size*

 *{*

 *get { return size; }*

 *set { size = value; }*

 *}*

 *public Node First*

 *{ get { return first; }*

 *set { first = value; }*

 *}*

 *public Node Last*

 *{*

 *get { return last; }*

 *set { last = value; }*

 *}*

 *public Que() { First = null; Last = null; }*

 *public void enqueue(int obj)*

 *{*

 *Node node = new Node(obj);*

 *node.Next=null; // узел будет новым концевым узлом*

 *if (Size == 0)*

 *First = node; //особый случай, если до этого очередь была пуста*

 *else*

 *Last.Next=node; // добавляет узел в конец списка*

 *Last = node; // перенаправляет ссылку к концевому узлу*

 *Size++;*

 *}*

 *public int dequeue()*

 *{*

 *int obj;*

 *obj = First.Data;*

 *First = First.Next;*

 *Size--;*

 *if (Size == 0)*

 *Last = null; // сейчас очередь пуста*

 *return obj;*

 *}*

 *public int siz() { return Size; }*

 *public bool isEmpty() { if (Size == 0) return true; else return false; }*

 *public Node front() { return First; }*

 *}*

# Классы коллекций

#  Класс ArrayList и его методы

Очень часто случается так, что размер массива неизвестен заранее. Именно в таких случаях удобно использовать класс ArrayList – динамический одномерный массив, то есть массив, позволяющий динамически менять свой размер. ArrayList реализует абстрактный тип «список».

Конструктор ArrayList() инициализирует новый пустой экземпляр класса ArrayList.

Реальное количество элементов, хранящихся в данный момент в ArrayList, можно узнать из свойства Count.

Метод Add(объект) добавляет объект в конец списка ArrayList.

# Метод BinarySearch (объект) выполняет поиск элемента по всему отсортированному списку [ArrayList](http://msdn.microsoft.com/ru-ru/library/system.collections.arraylist.aspx) и возвращает индекс (с нуля) этого элемента или -1, если элемента нет.

Метод Clear() удаляет все объекты из коллекции ArrayList

Метод Contains(объект) определяет, входит ли элемент в состав коллекции [ArrayList](http://msdn.microsoft.com/ru-ru/library/system.collections.arraylist.aspx).

Метод Insert (индекс, объект) вставляет элемент в список [ArrayList](http://msdn.microsoft.com/ru-ru/library/system.collections.arraylist.aspx) по указанному нндексу.

Метод Remove(объект) удаляет первое вхождение указанного объекта из списка [ArrayList](http://msdn.microsoft.com/ru-ru/library/system.collections.arraylist.aspx).

Метод RemoveAt(индекс) удаляет элемент с указанным индексом из списка [ArrayList](http://msdn.microsoft.com/ru-ru/library/system.collections.arraylist.aspx).

 В условиях примера с двумя классами вначале создается пустая коллекция:

*Global.A1 = new ArrayList();*

Добавление элемента в коллекцию:

*if (radioButton1.Checked){*

 *Global.A1.Add(new stud(Global.st1.name,Global.st1.age, Global.st1.ball, Global.st1.group)); }*

 *else*

 *{Global.A1.Add(newgrad\_stud(Global.st1.name,Global.st1.age, Global.st1.ball,Global.st1.group, Global.st1.place)); }*

Вывод элементов коллекции:

*listBox1.Items.Clear();*

 *foreach (stud A0 in Global.A1){*

 *listBox1.Items.Add(A0.info()); }*

# Класс Stack и его методы

Этот класс представляет собой коллекцию объектов типа "последним пришел — первым вышел" (LIFO).

Конструктор Stack() инициализирует новый пустой экземпляр класса Stack.

Реальное количество элементов, хранящихся в данный момент в Stack, можно узнать из свойства Count.

Метод Contains(объект) определяет, входит ли элемент в состав коллекции Stack.

Метод Clear() удаляет все объекты из коллекции Stack.

Метод Peek()возвращает верхний объект стека [Stack](http://msdn.microsoft.com/ru-ru/library/system.collections.stack.aspx), но не удаляет его.

Метод Pop() удаляет и возвращает верхний объект стека [Stack](http://msdn.microsoft.com/ru-ru/library/system.collections.stack.aspx).

Метод Push(объект) вставляет объект как верхний элемент стека [Stack](http://msdn.microsoft.com/ru-ru/library/system.collections.stack.aspx).

Пример применения методов:

*using System.Collections;*

*public class SamplesStack {*

 *public static void Main() {*

*// Создание и инициализация нового стека*

*Stack myStack = new Stack();*

*myStack.Push( "The" );*

*myStack.Push( "quick" );*

*myStack.Push( "brown" );*

*myStack.Push( "fox" );*

*// Вывод содержимого стека с конца*

 *Console.Write( "Содержимое стека:" );*

*PrintValues( myStack, '\t' );*

*// Удаление элемента из стека*

*Console.WriteLine( "(Удаленный)\t\t{0}", myStack.Pop() );*

*// Вывод содержимого стека*

*Console.Write( " Содержимое стека:" );*

*PrintValues( myStack, '\t' );*

*// Просмотр элемента в вершине стека.*

*Console.WriteLine( "(Верхний элемент)\t\t{0}", myStack.Peek() );}*

*public static void PrintValues( Stack myCollection,char mySeparator )*

*{ foreach ( Object obj in myCollection )*

 *Console.Write( "{0}{1}", mySeparator, obj ); Console.WriteLine();} }*

# Класс Queue и его методы

Этот класс представляет коллекцию объектов, основанную на принципе "первым поступил — первым обслужен".

Конструктор Queue() инициализирует новый пустой экземпляр класса Queue.

Реальное количество элементов, хранящихся в данный момент в Queue, можно узнать из свойства Count.

Метод Contains(объект) определяет, входит ли элемент в состав коллекции Queue.

Метод Clear() удаляет все объекты из коллекции Queue.

Метод Dequeue() удаляет и возвращает объект, находящийся в начале коллекции [Queue](http://msdn.microsoft.com/ru-ru/library/system.collections.queue.aspx).

Метод Enqueue(объект) добавляет объект в конец коллекции [Queue](http://msdn.microsoft.com/ru-ru/library/system.collections.queue.aspx).

Метод Peek() возвращает объект, находящийся в начале коллекции [Queue](http://msdn.microsoft.com/ru-ru/library/system.collections.queue.aspx), но не удаляет его.

В следующем примере показано, как добавить элементы в коллекцию [Queue](http://msdn.microsoft.com/ru-ru/library/system.collections.queue.aspx), удалить элементы из коллекции [Queue](http://msdn.microsoft.com/ru-ru/library/system.collections.queue.aspx) или просмотреть элемент, находящийся в начале коллекции [Queue](http://msdn.microsoft.com/ru-ru/library/system.collections.queue.aspx).

*using System.Collections;*

*public class SamplesQueue {*

*public static void Main() {*

*// Создание и инициализация новой очереди*

 *Queue myQ = new Queue();*

 *myQ.Enqueue( "The" );*

 *myQ.Enqueue( "quick" );*

 *myQ.Enqueue( "brown" );*

 *myQ.Enqueue( "fox" );*

*// Вывод очереди в порядке поступления*

 *Console.Write( "Очередь:" );*

 *PrintValues( myQ );*

 *// Удаление первого элемента очереди*

*Console.WriteLine( "(Удален)\t{0}", myQ.Dequeue() );*

*// Вывод очереди в порядке поступления*

 *Console.Write( " Очередь:" );*

 *PrintValues( myQ );*

*// Вывод первого элемента без его удаления*

 *Console.WriteLine( "(Первый элемент) \t{0}", myQ.Peek() ); }*

 *public static void PrintValues( Queue myCollection ) {*

 *foreach ( Object obj in myCollection )*

 *Console.Write( " {0}", obj );Console.WriteLine(); }}*

**Класс List и его методы**

Класс List – это коллекция, элементами которой являются экземпляры класса. Поддерживает методы для поиска по списку, выполнения сортировки и других операций со списками.

Объявление списка имеет вид:

*List<тип входящего объекта> имя = new List< тип входящего объекта>();*

У класса имеются свойство

Count - Число элементов, которое фактически содержится в коллекции.

Полезные методы класса List перечислены в табл. 12.

Таблица 12

Некоторые методы класса List

|  |  |
| --- | --- |
| Метод | Описание |
| Add(объект) | Добавляет объект в конец коллекции  |
| Clear() | Удаляет все элементы из коллекции |
| bool Contains(элемент) | Определяет, входит ли элемент в состав коллекции |
| bool Exists(метод)  | Определяет, содержит ли коллекция элементы, удовлетворяющие условиям указанного метода, который в качестве параметра должен иметь объект того класса, который помещен в коллекцию |
| Тип объекта Find(метод)  | Выполняет поиск элемента, удовлетворяющего условиям метода, и возвращает первое найденное вхождение в пределах всего списка |
| Insert(индекс с 0,элемент) | Добавляет элемент в список в позиции с указанным индексом |
| bool Remove(элемент) | Удаляет первое вхождение указанного объекта из коллекции |
| Sort(метод сравнения)  | Сортирует элементы во всем списке с использованием указанного метода |

Один из примеров иллюстриует применение метода Exists

*List<string> dinosaurs = new List<string>();*

*dinosaurs.Add("Compsognathus");*

*dinosaurs.Add("Amargasaurus");*

*dinosaurs.Add("Oviraptor");*

*dinosaurs.Add("Velociraptor");*

*dinosaurs.Add("Deinonychus");*

*dinosaurs.Add("Dilophosaurus");*

*dinosaurs.Add("Gallimimus");*

*dinosaurs.Add("Triceratops");*

 *Console.WriteLine("\nExists(EndsWithSaurus): {0}",*

 *dinosaurs.Exists(EndsWithSaurus));*

 *// Метод поиска возвращает true, если строка заканчивается "saurus"*

 *private static bool EndsWithSaurus(String s)*

 *{ if ((s.Length > 5) && (s.Substring(s.Length - 6).ToLower() == "saurus"))*

 *{ return true; }*

 *else*

 *{return false; } }*

Второй пример иллюстрирует применение метода Sort:

*List<string> dinosaurs = new List<string>();*

*dinosaurs.Add("Pachycephalosaurus");*

*dinosaurs.Add("Amargasaurus");*

*dinosaurs.Add("");*

*dinosaurs.Add(null);*

*dinosaurs.Add("Mamenchisaurus");*

*dinosaurs.Add("Deinonychus");*

*Display(dinosaurs);*

*dinosaurs.Sort(CompareDinosByLength);*

*Display(dinosaurs);*

*private static int CompareDinosByLength(string x, string y)*

*{// сранение длин двух строк*

 *int retval = x.Length.CompareTo(y.Length);*

 *if (retval != 0)*

 *// Если строки неравной длины, то более длинная строка больше*

 *return retval;*

 *else*

 *{// Если строки равной длины, они сортируются обычным образом*

 *return x.CompareTo(y); }*

 *}*

 *private static void Display(List<string> list)*

 *{ Console.WriteLine();*

 *foreach( string s in list )*

 *{if (s == null)*

 *Console.WriteLine("(null)");*

 *else*

 *Console.WriteLine("\"{0}\"", s); } }*

*m(n => (long)n)));*