Лабораторная работа №11
Тема: «Списки. Стек»

Цель работы: сформировать умения и навыки написания программ с использованием стека, списков.

Время выполнения: 4 часа.

Теоретические сведения

Стек — это коллекция, элементы которой получают по принципу «последний вошел, первый вышел» (Last-In-First-Out или LIFO). Это значит, что мы будем иметь доступ только к последнему добавленному элементу.

В отличие от списков, мы не можем получить доступ к произвольному элементу стека. Мы можем только добавлять или удалять элементы с помощью специальных методов. У стека нет также метода Contains, как у списков. Кроме того, у стека нет итератора. Для того, чтобы понимать, почему на стек накладываются такие ограничения, давайте посмотрим на то, как он работает и как используется.

Наиболее часто встречающаяся аналогия для объяснения стека — стопка тарелок. Вне зависимости от того, сколько тарелок в стопке, мы всегда можем снять верхнюю. Чистые тарелки точно так же кладутся на верх стопки, и мы всегда будем первой брать ту тарелку, которая была положена последней.

Класс Stack

Класс Stack определяет методы Push, Pop, Peek для доступа к элементам и поле Count. В реализации мы будем использовать LinkedList<T> для хранения элементов.

```
public class Stack
{
   LinkedList _items = new LinkedList();

   public void Push(T value)
   {
      throw new NotImplementedException();
   }

   public T Pop()
   {
      throw new NotImplementedException();
```

```
    }

    public T Peek()
    {
        throw new NotImplementedException();
    }

    public int Count
    {
        get;
    }
}
```

Метод Push

Поведение: Добавляет элемент на вершину стека.

Сложность: O(1).

Поскольку мы используем связный список для хранения элементов, можно просто добавить новый в конец списка.

```
public void Push(T value)
{
    _items.AddLast(value);
}
```

Метод Pop

Поведение: Удаляет элемент с вершины стека и возвращает его. Если стек пустой, кидает InvalidOperationException.

Сложность: O(1).

Push добавляет элементы в конец списка, поэтому забирать их будет также с конца. В случае, если список пуст, будет выбрасываться исключение.

```
public T Pop()
{
    if (_items.Count == 0)
    {
        throw new InvalidOperationException("The stack is empty");
    }

    T result = _items.Tail.Value;
```

```
    _items.RemoveLast();

    return result;
}
```

Метод Peek

Поведение: Возвращает верхний элемент стека, но не удаляет его. Если стек пустой, кидает InvalidOperationException.

Сложность: O(1).

```
public T Peek()
{
    if (_items.Count == 0)
    {
        throw new InvalidOperationException("The stack is empty");
    }

    return _items.Tail.Value;
}
```

Метод Count

Поведение: Возвращает количество элементов в стеке.

Сложность: O(1).

Зачем нам знать, сколько элементов находится в стеке, если мы все равно не имеем к ним доступа? С помощью этого поля мы можем проверить, есть ли элементы на стеке или он пуст. Это очень полезно, учитывая, что метод Pop кидает исключение.

```
public int Count
{
    get
    {
        return _items.Count;
    }
}
```

Связный список

Основное назначение связного списка — предоставление механизма для хранения и доступа к произвольному количеству данных. Как следует из названия, это достигается связыванием данных вместе в список.

Метод Add

Поведение: Добавляет элемент в конец списка.

Сложность: O(1)

Добавление элемента в связный список производится в три этапа:

Создать экземпляр класса LinkedListNode.

Найти последний узел списка.

Установить значение поля Next последнего узла списка так, чтобы оно указывало на созданный узел.

Основная сложность заключается в том, чтобы найти последний узел списка. Можно сделать это двумя способами. Первый — сохранять указатель на первый узел списка и перебирать узлы, пока не дойдем до последнего. В этом случае нам не требуется сохранять указатель на последний узел, что позволяет использовать меньше памяти (в зависимости от размера указателя на вашей платформе), но требует прохода по всему списку при каждом добавлении узла. Это значит, что метод Add займет O(n) времени.

Второй метод заключается в сохранении указателя на последний узел списка, и тогда при добавлении нового узла мы поменяем указатель так, чтобы он указывал на новый узел. Этот способ предпочтительней, поскольку выполняется за O(1) времени.

Первое, что нам необходимо сделать — добавить два приватных поля в класс LinkedList: ссылки на первый (head) и последний (tail) узлы.

```
private LinkedListNode _head;
private LinkedListNode _tail;
```

Теперь мы можем добавить метод, который выполняет три необходимых шага.

```
public void Add(T value)
{
   LinkedListNode node = new LinkedListNode(value);

   if (_head == null)
   {
      _head = node;
      _tail = node;
   }
```

```
        else
        {
            _tail.Next = node;
            _tail = node;
        }

        Count++;
    }
```

Сначала мы создаем экземпляр класса LinkedListNode. Затем проверяем, является ли список пустым. Если список пуст, мы просто устанавливаем значения полей _head и _tail так, чтобы они указывали на новый узел. Этот узел в данном случае будет являться одновременно и первым, и последним в списке. Если список не пуст, узел добавляется в конец списка, а поле _tail теперь указывает на новый конец списка.

Поле Count инкрементируется при добавлении узла для того, чтобы сохранялся контракт интерфейса ICollection<T>. Поле Count возвращает точное количество элементов списка.

Метод Remove

Поведение: Удаляет первый элемент списка со значением, равным переданному. Возвращает true, если элемент был удален и false в противном случае.

Сложность: O(n)

Прежде чем разбирать метод Remove, давайте посмотрим, чего мы хотим добиться. На следующем рисунке список с четырьмя элементами. Мы удаляем элемент со значением «3».

После удаления узла поле Next узла со значением «2» будет указывать на узел со значением «4».

Основной алгоритм удаления элемента такой:

Найти узел, который необходимо удалить.

Изменить значение поля Next предыдущего узла так, чтобы оно указывало на узел, следующий за удаляемым.

Как всегда, основная проблема кроется в мелочах. Вот некоторые из случаев, которые необходимо предусмотреть:

Список может быть пустым, или значение, которое мы передаем в метод может не присутствовать в списке. В этом случает список останется без изменений.

Удаляемый узел может быть единственным в списке. В этом случае мы установим значения полей _head и _tail равными null.

Удаляемый узел будет в начале списка. В этом случае мы записываем в _head ссылку на следующий узел.

Удаляемый узел будет в середине списка.

Удаляемый узел будет в конце списка. В этом случае мы записываем в _tail ссылку на предпоследний узел, а в его поле Next записываем null.

```
public bool Remove(T item)
{
   LinkedListNode previous = null;
   LinkedListNode current = _head;

   // 1: Пустой список: ничего не делать.
   // 2: Один элемент: установить Previous = null.
   // 3: Несколько элементов:
   //    a: Удаляемый элемент первый.
   //    b: Удаляемый элемент в середине или конце.

   while (current != null)
   {
      if (current.Value.Equals(item))
      {
         // Узел в середине или в конце.
         if (previous != null)
         {
            // Случай 3b.

            // До:    Head -> 3 -> 5 -> null
            // После: Head -> 3 ------> null
            previous.Next = current.Next;

            // Если в конце, то меняем _tail.
            if (current.Next == null)
            {
               _tail = previous;
            }
         }
         else
         {
            // Случай 2 или 3a.
```

```
                // До:    Head -> 3 -> 5
                // После: Head ------> 5

                // Head -> 3 -> null
                // Head ------> null
                _head = _head.Next;

                // Список теперь пустой?
                if (_head == null)
                {
                    _tail = null;
                }
            }

            Count--;

            return true;
        }

        previous = current;
        current = current.Next;
    }

    return false;
}
```

Поле Count декрементируется при удалении узла.

Метод Contains

Поведение: Возвращает true или false в зависимости от того, присутствует ли искомый элемент в списке.

Сложность: O(n)

Метод Contains достаточно простой. Он просматривает каждый элемент списка, от первого до последнего, и возвращает true как только найдет узел, чье значение равно переданному параметру. Если такой узел не найден, и метод дошел до конца списка, то возвращается false.

```
public bool Contains(T item)
{
```

```
    LinkedListNode current = _head;
    while (current != null)
    {
        if (current.Value.Equals(item))
        {
            return true;
        }

        current = current.Next;
    }

    return false;
}
```

Метод GetEnumerator

Поведение: Возвращает экземпляр IEnumerator, который позволяет итерироваться по элементам списка.

Сложность: Получение итератора — O(1). Проход по всем элементам — O(n).

Возвращаемый итератор проходит по всему списку от первого до последнего узла и возвращает значение каждого элемента с помощью ключевого слова yield.

```
IEnumerator IEnumerable.GetEnumerator()
{
    LinkedListNode current = _head;
    while (current != null)
    {
        yield return current.Value;
        current = current.Next;
    }
}

IEnumerator IEnumerable.GetEnumerator()
{
    return ((IEnumerable)this).GetEnumerator();
}
```

Метод Clear

Поведение: Удаляет все элементы из списка.

Сложность: O(1)

Метод Clear просто устанавливает значения полей _head и _tail равными null. Поскольку C# — язык с автоматическим управлением памятью, нет необходимости явно удалять неиспользуемые узлы. Клиент, вызывающий метод, должен убедиться в корректном удалении значений узлов, если это необходимо.

```
public void Clear()
{
   _head = null;
   _tail = null;
   Count = 0;
}
```

Метод CopyTo

Поведение: Копирует содержимое списка в указанный массив, начиная с указанного индекса.

Сложность: O(n)

Метод CopyTo проходит по списку и копирует элементы в массив с помощью присваивания. Клиент, вызывающий метод должен убедиться, что массив имеет достаточный размер для того, чтобы вместить все элементы списка.

```
public void CopyTo(T[] array, int arrayIndex)
{
   LinkedListNode current = _head;
   while (current != null)
   {
      array[arrayIndex++] = current.Value;
      current = current.Next;
   }
}
```

Метод Count

Поведение: Возвращает количество элементов списка. Возвращает 0, если список пустой.

Сложность: O(1)

Count — поле с публичным геттером и приватным сеттером. Изменение его значения осуществляется в методах Add, Remove и Clear.

```
public int Count
{
   get;
   private set;
}
```

Метод IsReadOnly

Поведение: Возвращает true, если список только для чтения.

Сложность: O(1)

```
public bool IsReadOnly
{
   get { return false; }
}
```

Двусвязный список

Связный список, который мы только что создали, называется также «односвязным». Это значит, что между узлами только одна связь в единственном направлении от первого узла к последнему. Есть также достаточно распространенный вариант списка, который предоставляет доступ к обоим концам — двусвязный список.

Для того, чтобы создать двусвязный список, мы должны добавить в класс LinkedListNode поле Previous, которое будет содержать ссылку на предыдущий элемент списка.

Далее мы рассмотрим только отличия в реализации односвязного и двусвязного списка.

Класс Node

Единственное изменение, которое надо внести в класс LinkedListNode — добавить поле со ссылкой на предыдущий узел.

```
public class LinkedListNode
{
   ///
   /// Конструктор нового узла со значением Value.
   ///
   ///
   public LinkedListNode(T value)
   {
      Value = value;
   }
```

```
    ///
    /// Поле Value.
    ///
    public T Value { get;  internal set; }

    ///
    /// Ссылка на следующий узел списка (если узел последний, то null).
    ///
    public LinkedListNode Next { get; internal set; }

    ///
    /// Ссылка на предыдущий узел списка (если узел первый, то null).
    ///
    public LinkedListNode Previous { get;  internal set; }
}
```

Метод AddFirst

В то время, как односвязный список позволяет добавлять элементы только в конец, используя двусвязный список мы можем добавлять элементы как в начало, так и в конец, с помощью методов AddFirst и AddLast соответственно. Метод ICollection<T>.Add будет вызывать AddLast для совместимости с односвязным списком.

Поведение: Добавляет переданный элемент в начало списка.

Сложность: O(1)

При добавлении элемента в начало списка последовательность действий примерно такая же, как и при добавлении элемента в односвязный список.

Установить значение поля Next в новом узле так, чтобы оно указывало на бывший первый узел.

Установить значение поля Previous в бывшем первом узле так, чтобы оно указывало на новый узел.

Обновить поле _tail при необходимости и инкрементировать поле Count

```
public void AddFirst(T value)
{
    LinkedListNode node = new LinkedListNode(value);

    // Сохраняем ссылку на первый элемент.
    LinkedListNode temp = _head;
```

```
        // _head указывает на новый узел.
        _head = node;

        // Вставляем список позади первого элемента.
        _head.Next = temp;

        if (Count == 0)
        {
            // Если список был пуст, то head and tail должны
            // указывать на новой узел.
            _tail = _head;
        }
        else
        {
            // До:    head -------> 5  7 -> null
            // После: head  -> 3  5  7 -> null
            temp.Previous = _head;
        }

        Count++;
    }
```

Метод AddLast

Поведение: Добавляет переданный элемент в конец списка.

Сложность: O(1)

Добавление узла в конец списка легче, чем в начало. Мы просто создаем новый узел и обновляем поля _head и _tail, а затем инкрементируем поле Count.

```
    public void AddLast(T value)
    {
        LinkedListNode node = new LinkedListNode(value);

        if (Count == 0)
        {
            _head = node;
        }
        else
        {
```

```
            _tail.Next = node;

            // До:    Head -> 3  5 -> null
            // После:Head -> 3  5  7 -> null
            // 7.Previous = 5
            node.Previous = _tail;
        }

        _tail = node;
        Count++;
    }
```

Как было сказано ранее, ICollection<T>.Add просто зовет AddLast.

```
    public void Add(T value)
    {
        AddLast(value);
    }
```

Метод RemoveFirst

Как и метод Add, Remove будет разделен на два метода, позволяющих удалять элементы из начала и из конца списка. Метод ICollection<T>.Remove будет также удалять элементы из начала, но теперь будет еще обновлять поля Previous в тех узлах, где это необходимо.

Поведение: Удаляет первый элемент списка. Если список пуст, не делает ничего. Возвращает true, если элемент был удален и false в противном случае.

Сложность: O(1)

RemoveFirst устанавливает ссылку head на второй узел списка и обнуляет поле Previous этого узла, удаляя таким образом все ссылки на предыдущий первый узел. Если список был пуст или содержал только один элемент, то поля _head и _tail становятся равны null.

```
    public void RemoveFirst()
    {
        if (Count != 0)
        {
            // До:    Head -> 3  5
            // После: Head -------> 5
```

```
            // Head -> 3 -> null
            // Head ------> null
            _head = _head.Next;

            Count--;

            if (Count == 0)
            {
                _tail = null;
            }
            else
            {
                // 5.Previous было 3; теперь null.
                _head.Previous = null;
            }
        }
    }
```

Метод RemoveLast

Поведение: Удаляет последний элемент списка. Если список пуст, не делает ничего. Возвращает true, если элемент был удален и false в противном случае.

Сложность: O(1)

RemoveLast устанавливает значение поля _tail так, чтобы оно указывало на предпоследний элемент списка и, таким образом, удаляет последний элемент. Если список был пустым, или содержал только один элемент, то поля _head и _tail становятся равны null.

```
    public void RemoveLast()
    {
        if (Count != 0)
        {
            if (Count == 1)
            {
                _head = null;
                _tail = null;
            }
            else
            {
                // До:    Head --> 3 --> 5 --> 7
```

```
            //       Tail = 7
            // После: Head --> 3 --> 5 --> null
            //       Tail = 5
            // Обнуляем 5.Next
            _tail.Previous.Next = null;
            _tail = _tail.Previous;
        }

        Count--;
    }
}
```

Метод Remove

Поведение: Удаляет первый элемент списка со значением, равным переданному. Возвращает true, если элемент был удален и false в противном случае.

Сложность: O(n)

Метод ICollection<T>.Remove() почти такой же, как и в односвязном списке. Единственное отличие — теперь нам необходимо поменять значение поля Previous при удалении узла. Для того, чтобы не повторять код, этот метод зовет RemoveFirst при удалении первого узла.

```
public bool Remove(T item)
{
    LinkedListNode previous = null;
    LinkedListNode current = _head;

    // 1: Пустой список: ничего не делать.
    // 2: Один элемент: установить Previous = null.
    // 3: Несколько элементов:
    //    a: Удаляемый элемент первый.
    //    b: Удаляемый элемент в середине или конце.

    while (current != null)
    {
        // Head -> 3 -> 5 -> 7 -> null
        // Head -> 3 ------> 7 -> null
        if (current.Value.Equals(item))
        {
            // Узел в середине или в конце.
```

```
            if (previous != null)
            {
                // Случай 3b.
                previous.Next = current.Next;

                // Если в конце, то меняем _tail.
                if (current.Next == null)
                {
                    _tail = previous;
                }
                else
                {
                    // До:    Head -> 3  5  7 -> null
                    // После: Head -> 3  7 -> null

                    // previous = 3
                    // current = 5
                    // current.Next = 7
                    // Значит... 7.Previous = 3
                    current.Next.Previous = previous;
                }

                Count--;
            }
            else
            {
                // Случай 2 или 3a.
                RemoveFirst();
            }

            return true;
        }

        previous = current;
        current = current.Next;
    }

    return false;
}
```

Зачем нужен двусвязный список?

Итак, мы можем добавлять элементы в начало списка и в его конец. Что нам это дает? В том виде, в котором он реализован сейчас, нет особых преимуществ перед обычным односвязным списком. Но если добавить геттеры для полей head и tail, пользователь нашего списка сможет реализовать множество различных алгоритмов.

```
public LinkedListNode Head
{
   get
   {
      return _head;
   }
}

public LinkedListNode Tail
{
   get
   {
      return _tail;
   }
}
```

Так мы сможем итерироваться по списку вручную, в том числе от последнего элемента к первому.

В этом примере мы используем поля Tail и Previous для того, чтобы обойти список задом наперед.

```
public void ProcessListBackwards()
{
   LinkedList list = new LinkedList();
   PopulateList(list);

   LinkedListNode current = list.Tail;
   while (current != null)
   {
      ProcessNode(current);
      current = current.Previous;
   }
```

}

Кроме того, двусвязный список позволяет легко реализовать двусвязную очередь, которая, в свою очередь, является строительным блоком для других структур данных.

Индивидуальные задания к лабораторной работе №11

1. Постройте с помощью массива стек из 6 строковых элементов. Разместите в стеке шесть элементов: ‘name’, ‘fio’, ‘ves’, ‘age’, ‘rost’, ‘dlina’. Удалите из стека два элемента ‘dlina’ и ‘fio’ и добавьте новый элемент ‘size’. Результаты как промежуточных, так и конечных результатов отобразить на экране.
2. Опишите и постройте с помощью двумерного массива Sps линейный однонаправленный список из семи целых чисел и сделайте этот список пустым. После этого добавьте в список шесть элементов 1,3,5,7,9,11, затем найдите указатель на элемент 9 и удалите этот элемент. В конце работы со списком вставьте после элемента со значением 11 элемент со значением 13, предварительно отыскав указатель на элемент со значением 11, а элемент со значением 15 вставьте после элемента со значением 3. Результаты как промежуточных, так и конечных результатов отобразить на экране.
3. Дан указатель head на вершину стека (если стек пуст, то head = NULL). Разработать программу, позволяющую добавлять, извлекать элементы стека, просматривать содержимое стека, подсчитывать количество элементов в стеке.
4. Даны указатели head1 и head2 на вершины стеков, хранящих некоторые отсортированные по убыванию наборы целых чисел. Разработать программу, позволяющую построить стек (head3), содержащий отсортированный по возрастанию набор чисел, извлеченных из исходных стеков.
5. Даны указатели head1 и head2 на вершины стеков, хранящих некоторые отсортированные по убыванию наборы целых чисел. Разработать программу, позволяющую построить стек (head3), содержащий набор упорядоченных по возрастанию четных чисел, извлеченных из исходных стеков.
6. Загрузить данные из исходного файла в двунаправленный список. Сохранить двунаправленный список в контрольный файл. Упорядочить исходные данные по возрастанию. Перевести данные из двунаправленного списка в два кольцевых списка (один для четных чисел, другой для нечетных). Сохранить два кольцевых списка в два файла.

7. Создать список из случайных целых чисел, лежащих в диапазоне от –50 до +50 и преобразовать его в два списка. Первый должен содержать только положительные числа, а второй – только отрицательные. Порядок следования чисел должен быть сохранен.

8. Создать список из случайных целых чисел и удалить из него записи с четными числами.

9. Создать список из случайных положительных и отрицательных целых чисел (от –10 до 10) и удалить из него отрицательные элементы.

10. Создать список из случайных целых чисел и поменять местами крайние элементы.

11. Создать список из случайных целых чисел и удалить элементы, заканчивающиеся на цифру 5.

12. Создать список из случайных целых чисел и поменять местами элементы, содержащие максимальное и минимальное значения.

13. Создать список из случайных целых чисел. Перенести в другой список все элементы, находящиеся между вершиной и элементом с максимальным значением.

14. Создать список из случайных целых чисел. Перенести в другой список все элементы, находящиеся между вершиной и элементом с минимальным значением.

15. Создать список из случайных чисел, определить количество элементов, находящихся между минимальным и максимальным элементами, и удалить их.

16. Создать список из случайных чисел и определить количество элементов, имеющих значения, меньше среднего значения от всех элементов, и удалить эти элементы.

17. Создать список из случайных чисел, вычислить среднее арифметическое и заменить им первый элемент.

18. Создать список из случайных целых чисел, разделить его на два: в первый поместить все четные, а во второй – нечетные числа.

19. Создать список из случайных целых чисел в диапазоне от 1 до 10, определить наиболее часто встречающееся число и удалить его.

20. Создать список из случайных целых чисел и удалить из него каждый второй элемент.

21. Создать список из случайных целых чисел и удалить из него каждый нечетный элемент.

ПОРЯДОК ВЫПОЛНЕНИЯ РАБОТЫ

1. Изучить теоретическую часть лабораторной работы.

2. Реализовать индивидуальное задание по вариантам, представленные в теоретических сведениях, сделать скриншоты работающих программ. Написать комментарии.

3. Написать отчет, содержащий:

1. Титульный лист, на котором указывается:

а) полное наименование министерства образование и название учебного заведения;

б) название дисциплины;

в) номер практического занятия;

г) фамилия преподавателя, ведущего занятие;

д) фамилия, имя и номер группы студента;

е) год выполнения лабораторной работы.

2. Индивидуальное задание из раздела «Теоретические сведения» с кодом, комментариями и скриншотами работающих программ.

3. Построение блок-схем.