

LinksPlatform's Platform.Collections Class Library

1.1 ./csharp/Platform.Collections/Arrays/ArrayFiller[TElement, TReturnConstant].cs

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  namespace Platform.Collections.Arrays
5  {
6      /// <summary>
7      /// <para>Represents <see cref="T:TElement[]"> array filler with additional methods that
8      ///     → return a given constant of type <typeparamref cref="TReturnConstant"/>.</para>
9      /// <para>Представляет заполнитель массива <see cref="T:TElement[]"> с дополнительными
10     → методами, возвращающими заданную константу типа <typeparamref
11     → cref="TReturnConstant"/>.</para>
12     </summary>
13     <typeparam name="TElement"><para>The elements' type.</para><para>Тип элементов
14     → массива.</para></typeparam>
15     <typeparam name="TReturnConstant"><para>The return constant's type.</para><para>Тип
16     → возвращаемой константы.</para></typeparam>
17     public class ArrayFiller<TElement, TReturnConstant> : ArrayFiller<TElement>
18     {
19         protected readonly TReturnConstant _returnConstant;
20
21         /// <summary>
22         /// <para>Initializes a new instance of the <see cref="ArrayFiller"/> class using the
23         → specified array, the offset from which filling will start and the constant returned
24         → when elements are being filled.</para>
25         /// <para>Инициализирует новый экземпляр класса <see cref="ArrayFiller"/>, используя
26         → указанный массив, смещение с которого начнётся заполнение и константу возвращаемую
27         → при заполнении элементов.</para>
28         </summary>
29         <param name="array"><para>The array to fill.</para><para>Массив для
30         → заполнения.</para></param>
31         <param name="offset"><para>The offset from which to start the array
32         → filling.</para><para>Смещение с которого начнётся заполнение массива.</para></param>
33         <param name="returnConstant"><para>The constant's value.</para><para>Значение
34         → константы.</para></param>
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         public ArrayFiller(TElement[] array, long offset, TReturnConstant returnConstant) :
37         → base(array, offset) => _returnConstant = returnConstant;
38
39         /// <summary>
40         /// <para>Initializes a new instance of the <see cref="ArrayFiller"/> class using the
41         → specified array and the constant returned when elements are being filled. Filling
42         → will start from the beginning of the array.</para>
43         /// <para>Инициализирует новый экземпляр класса <see cref="ArrayFiller"/>, используя
44         → указанный массив и константу возвращаемую при заполнении элементов. Заполнение
45         → начнётся с начала массива.</para>
46         </summary>
47         <param name="array"><para>The array to fill.</para><para>Массив для
48         → заполнения.</para></param>
49         <param name="returnConstant"><para>The constant's value.</para><para>Значение
50         → константы.</para></param>
51         [MethodImpl(MethodImplOptions.AggressiveInlining)]
52         public ArrayFiller(TElement[] array, TReturnConstant returnConstant) : this(array, 0,
53         → returnConstant) { }
54
55         /// <summary>
56         /// <para>Adds an item into the array and returns the constant.</para>
57         /// <para>Добавляет элемент в массив и возвращает константу.</para>
58         </summary>
59         <param name="element"><para>The element to add.</para><para>Добавляемый
60         → элемент.</para></param>
61         </returns>
62         <para>The constant's value.</para>
63         <para>Значение константы.</para>
64         </returns>
65         [MethodImpl(MethodImplOptions.AggressiveInlining)]
66         public TReturnConstant AddAndReturnConstant(TElement element) =>
67         → _array.AddAndReturnConstant(ref _position, element, _returnConstant);
68
69         /// <summary>
70         /// <para>Adds the first element from the specified list to the filled array and returns
71         → the constant.</para>
72         /// <para>Добавляет первый элемент из указанного списка в заполняемый массив и
73         → возвращает константу.</para>
74         </summary>

```

```

51     /// <param name="element"><para>The list from which the first item will be
    → added.</para><para>Список из которого будет добавлен первый элемент.</para></param>
52     /// <returns>
53     /// <para>The constant's value.</para>
54     /// <para>Значение константы.</para>
55     /// <returns>
56     [MethodImpl(MethodImplOptions.AggressiveInlining)]
57     public TReturnConstant AddFirstAndReturnConstant(ICollection<TElement> elements) =>
    → _array.AddFirstAndReturnConstant(ref _position, elements, _returnConstant);
58
59     /// <summary>
60     /// <para>Adds all elements from the specified list to the filled array and returns the
    → constant.</para>
61     /// <para>Добавляет все элементы из указанного списка в заполняемый массив и возвращает
    → константу.</para>
62     /// </summary>
63     /// <param name="elements"><para>The list of values to add.</para><para>Список значений
    → для добавления.</para></param>
64     /// <returns>
65     /// <para>The constant's value.</para>
66     /// <para>Значение константы.</para>
67     /// <returns>
68     [MethodImpl(MethodImplOptions.AggressiveInlining)]
69     public TReturnConstant AddAllAndReturnConstant(ICollection<TElement> elements) =>
    → _array.AddAllAndReturnConstant(ref _position, elements, _returnConstant);
70
71     /// <summary>
72     /// <para>Adds the elements of the list to the array, skipping the first element and
    → returns the constant.</para>
73     /// <para>Добавляет элементы списка в массив пропуская первый элемент и возвращает
    → константу.</para>
74     /// </summary>
75     /// <param name="elements"><para>The list of values to add.</para><para>Список значений
    → для добавления.</para></param>
76     /// <returns>
77     /// <para>The constant's value.</para>
78     /// <para>Значение константы.</para>
79     /// </returns>
80     [MethodImpl(MethodImplOptions.AggressiveInlining)]
81     public TReturnConstant AddSkipFirstAndReturnConstant(ICollection<TElement> elements) =>
    → _array.AddSkipFirstAndReturnConstant(ref _position, elements, _returnConstant);
82 }
83 }

```

1.2 ./csharp/Platform.Collections/Arrays/ArrayFiller[TElement].cs

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  namespace Platform.Collections.Arrays
5  {
6      /// <summary>
7      /// <para>Represents an <see cref="T:TElement[]"> array filler.</para>
8      /// <para>Представляет заполнитель массива <see cref="T:TElement[]">.</para>
9      /// </summary>
10     /// <typeparam name="TElement"><para>The elements' type.</para><para>Тип элементов
    → массива.</para></typeparam>
11     public class ArrayFiller<TElement>
12     {
13         protected readonly TElement[] _array;
14         protected long _position;
15
16         /// <summary>
17         /// <para>Initializes a new instance of the <see cref="ArrayFiller"/> class using the
    → specified array as the array to fill and the offset from which to start
    → filling.</para></summary>
18         /// <para>Инициализирует новый экземпляр класса <see cref="ArrayFiller"/>, используя
    → указанный массив в качестве заполняемого и смещение с которого начнётся
    → заполнение.</para>
19         /// </summary>
20         /// <param name="array"><para>The array to fill.</para><para>Массив для
    → заполнения.</para></param>
21         /// <param name="offset"><para>The offset from which to start filling the
    → array.</para><para>Смещение с которого начнётся заполнение массива.</para></param>
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         public ArrayFiller(TElement[] array, long offset)
24         {
25             _array = array;

```

```

26     _position = offset;
27 }
28
29 /// <summary>
30 /// <para>Initializes a new instance of the <see cref="ArrayFiller"/> class using the
    → specified array. Filling will start from the beginning of the array.</para>
31 /// <para>Инициализирует новый экземпляр класса <see cref="ArrayFiller"/>, используя
    → указанный массив. Заполнение начнётся с начала массива.</para>
32 /// </summary>
33 /// <param name="array"><para>The array to fill.</para><para>Массив для
    → заполнения.</para></param>
34 [MethodImpl(MethodImplOptions.AggressiveInlining)]
35 public ArrayFiller(TElement[] array) : this(array, 0) { }
36
37 /// <summary>
38 /// <para>Adds an item into the array.</para>
39 /// <para>Добавляет элемент в массив.</para>
40 /// </summary>
41 /// <param name="element"><para>The element to add.</para><para>Добавляемый
    → элемент.</para></param>
42 [MethodImpl(MethodImplOptions.AggressiveInlining)]
43 public void Add(TElement element) => _array[_position++] = element;
44
45 /// <summary>
46 /// <para>Adds an item into the array and returns <see langword="true"/>.</para>
47 /// <para>Добавляет элемент в массив и возвращает <see langword="true"/>.</para>
48 /// </summary>
49 /// <param name="element"><para>The element to add.</para><para>Добавляемый
    → элемент.</para></param>
50 /// <returns>
51 /// <para>The <see langword="true"/> value.</para>
52 /// <para>Значение <see langword="true"/>.</para>
53 /// </returns>
54 [MethodImpl(MethodImplOptions.AggressiveInlining)]
55 public bool AddAndReturnTrue(TElement element) => _array.AddAndReturnConstant(ref
    → _position, element, true);
56
57 /// <summary>
58 /// <para>Adds the first element from the specified list to the array to fill and
    → returns <see langword="true"/>.</para>
59 /// <para>Добавляет первый элемент из указанного списка в заполняемый массив и
    → возвращает <see langword="true"/>.</para>
60 /// </summary>
61 /// <param name="element"><para>The list from which the first item will be
    → added.</para><para>Список из которого будет добавлен первый элемент.</para></param>
62 /// <returns>
63 /// <para>The <see langword="true"/> value.</para>
64 /// <para>Значение <see langword="true"/>.</para>
65 /// </returns>
66 [MethodImpl(MethodImplOptions.AggressiveInlining)]
67 public bool AddFirstAndReturnTrue(IList<TElement> elements) =>
    → _array.AddFirstAndReturnConstant(ref _position, elements, true);
68
69 /// <summary>
70 /// <para>Adds all elements from the specified list to the array to fill and returns
    → <see langword="true"/>.</para>
71 /// <para>Добавляет все элементы из указанного списка в заполняемый массив и возвращает
    → <see langword="true"/>.</para>
72 /// </summary>
73 /// <param name="elements"><para>The list of values to add.</para><para>Список значений
    → которые необходимо добавить.</para></param>
74 /// <returns>
75 /// <para>The <see langword="true"/> value.</para>
76 /// <para>Значение <see langword="true"/>.</para>
77 /// </returns>
78 [MethodImpl(MethodImplOptions.AggressiveInlining)]
79 public bool AddAllAndReturnTrue(IList<TElement> elements) =>
    → _array.AddAllAndReturnConstant(ref _position, elements, true);
80
81 /// <summary>
82 /// <para>Adds values to the array skipping the first element and returns <see
    → langword="true"/>.</para>
83 /// <para>Добавляет значения в массив пропуская первый элемент и возвращает <see
    → langword="true"/>.</para>
84 /// </summary>

```

```

85     /// <param name="elements"><para>A list from which elements will be added except the
    ↪ first.</para><para>Список из которого будут добавлены элементы кроме
    ↪ первого.</para></param>
86     /// <returns>
87     /// <para>The <see langword="true"/> value.</para>
88     /// <para>Значение <see langword="true"/>.</para>
89     /// </returns>
90     [MethodImpl(MethodImplOptions.AggressiveInlining)]
91     public bool AddSkipFirstAndReturnTrue(IList<TElement> elements) =>
    ↪     _array.AddSkipFirstAndReturnConstant(ref _position, elements, true);
92 }
93 }

```

1.3 ./csharp/Platform.Collections/Arrays/ArrayPool.cs

```

1  using System.Runtime.CompilerServices;
2
3  namespace Platform.Collections.Arrays
4  {
5      /// <summary>
6      /// <para>Represents a set of wrapper methods over <see cref="ArrayPool{T}"/> class methods
    ↪ to simplify access to them.</para>
7      /// <para>Представляет набор методов обёрток над методами класса <see cref="ArrayPool{T}">
    ↪ для упрощения доступа к ним.</para>
8      /// </summary>
9      public static class ArrayPool
10     {
11         public static readonly int DefaultSizesAmount = 512;
12         public static readonly int DefaultMaxArraysPerSize = 32;
13
14         /// <summary>
15         /// <para>Allocation of an array of a specified size from the array pool.</para>
16         /// <para>Выделение массива указанного размера из пула массивов.</para>
17         /// </summary>
18         /// <typeparam name="T"><para>The array elements type.</para><para>Тип элементов
    ↪ массива.</para></typeparam>
19         /// <param name="size"><para>The allocated array size.</para><para>Размер выделяемого
    ↪ массива.</para></param>
20         /// <returns>
21         /// <para>The array from a pool of arrays.</para>
22         /// <para>Массив из пулла массивов.</para>
23         /// </returns>
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         public static T[] Allocate<T>(long size) => ArrayPool<T>.ThreadInstance.Allocate(size);
26
27         /// <summary>
28         /// <para>Freeing an array into an array pool.</para>
29         /// <para>Освобождение массива в пул массивов.</para>
30         /// </summary>
31         /// <typeparam name="T"><para>The array elements type.</para><para>Тип элементов
    ↪ массива.</para></typeparam>
32         /// <param name="array"><para>The array to be freed into the pull.</para><para>Массив
    ↪ который нужно освободить в пулл.</para></param>
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         public static void Free<T>(T[] array) => ArrayPool<T>.ThreadInstance.Free(array);
35     }
36 }

```

1.4 ./csharp/Platform.Collections/Arrays/ArrayPool[T].cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Disposables;
5  using Platform.Collections.Stacks;
6
7  namespace Platform.Collections.Arrays
8  {
9      /// <summary>
10     /// <para>Represents a set of arrays ready for reuse.</para>
11     /// <para>Представляет собой набор массивов готовых к повторному использованию.</para>
12     /// </summary>
13     /// <typeparam name="T"><para>The array elements type.</para><para>Тип элементов
    ↪ массива.</para></typeparam>
14     /// <remarks>
15     /// Original idea from
    ↪ http://geekswithblogs.net/blackrob/archive/2014/12/18/array-pooling-in-csharp.aspx
16     /// </remarks>
17     public class ArrayPool<T>
18     {

```

```

19 // May be use Default class for that later.
20 [ThreadStatic]
21 private static ArrayPool<T> _threadInstance;
22 internal static ArrayPool<T> ThreadInstance => _threadInstance ?? (_threadInstance = new
    ↳ ArrayPool<T>());
23
24 private readonly int _maxArraysPerSize;
25 private readonly Dictionary<long, Stack<T[]>> _pool = new Dictionary<long,
    ↳ Stack<T[]>>(ArrayPool.DefaultSizesAmount);
26
27 /// <summary>
28 /// <para>Initializes a new instance of the ArrayPool class using the specified maximum
    ↳ number of arrays per size.</para>
29 /// <para>Инициализирует новый экземпляр класса ArrayPool, используя указанное
    ↳ максимальное количество массивов на каждый размер.</para>
30 /// </summary>
31 /// <param name="maxArraysPerSize"><para>The maximum number of arrays in the pool per
    ↳ size.</para><para>Максимальное количество массивов в пуле на каждый
    ↳ размер.</para></param>
32 [MethodImpl(MethodImplOptions.AggressiveInlining)]
33 public ArrayPool(int maxArraysPerSize) => _maxArraysPerSize = maxArraysPerSize;
34
35 /// <summary>
36 /// <para>Initializes a new instance of the ArrayPool class using the default maximum
    ↳ number of arrays per size.</para>
37 /// <para>Инициализирует новый экземпляр класса ArrayPool, используя максимальное
    ↳ количество массивов на каждый размер по умолчанию.</para>
38 /// </summary>
39 [MethodImpl(MethodImplOptions.AggressiveInlining)]
40 public ArrayPool() : this(ArrayPool.DefaultMaxArraysPerSize) { }
41
42 /// <summary>
43 /// <para>Retrieves an array from the pool, which will automatically return to the pool
    ↳ when the container is disposed.</para>
44 /// <para>Извлекает из пула массив, который автоматически вернётся в пул при
    ↳ высвобождении контейнера.</para>
45 /// </summary>
46 /// <param name="size"><para>The allocated array size.</para><para>Размер выделяемого
    ↳ массива.</para></param>
47 /// <returns>
48 /// <para>The disposable container containing either a new array or an array from the
    ↳ pool.</para>
49 /// <para>Высвобождаемый контейнер содержащий либо новый массив, либо массив из
    ↳ пула.</para>
50 /// </returns>
51 [MethodImpl(MethodImplOptions.AggressiveInlining)]
52 public Disposable<T[]> AllocateDisposable(long size) => (Allocate(size), Free);
53
54 /// <summary>
55 /// <para>Replaces the array with another array from the pool with the specified
    ↳ size.</para>
56 /// <para>Заменяет массив на другой массив из пула с указанным размером.</para>
57 /// </summary>
58 /// <param name="source"><para>The source array.</para><para>Исходный
    ↳ массив.</para></param>
59 /// <param name="size"><para>A new array size.</para><para>Новый размер
    ↳ массива.</para></param>
60 /// <returns>
61 /// <para>An array with a new size.</para>
62 /// <para>Массив с новым размером.</para>
63 /// </returns>
64 [MethodImpl(MethodImplOptions.AggressiveInlining)]
65 public Disposable<T[]> Resize(Disposable<T[]> source, long size)
66 {
67     var destination = AllocateDisposable(size);
68     T[] sourceArray = source;
69     if (!sourceArray.IsNullOrEmpty())
70     {
71         T[] destinationArray = destination;
72         Array.Copy(sourceArray, destinationArray, size < sourceArray.LongLength ? size :
            ↳ sourceArray.LongLength);
73         source.Dispose();
74     }
75     return destination;
76 }
77
78 /// <summary>

```

```

79     /// <para>Clears the pool.</para>
80     /// <para>Очищает пул.</para>
81     /// </summary>
82     [MethodImpl(MethodImplOptions.AggressiveInlining)]
83     public virtual void Clear() => _pool.Clear();
84
85     /// <summary>
86     /// <para>Retrieves an array with the specified size from the pool.</para>
87     /// <para>Извлекает из пула массив с указанным размером.</para>
88     /// </summary>
89     /// <param name="size"><para>The allocated array size.</para><para>Размер выделяемого
    ↳ массива.</para></param>
90     /// <returns>
91     /// <para>An array from the pool or a new array.</para>
92     /// <para>Массив из пула или новый массив.</para>
93     /// </returns>
94     [MethodImpl(MethodImplOptions.AggressiveInlining)]
95     public virtual T[] Allocate(long size) => size <= 0L ? Array.Empty<T>() :
    ↳ _pool.GetOrDefault(size)?.PopOrDefault() ?? new T[size];
96
97     /// <summary>
98     /// <para>Frees the array to the pool for later reuse.</para>
99     /// <para>Освобождает массив в пул для последующего повторного использования.</para>
100    /// </summary>
101    /// <param name="array"><para>The array to be freed into the pool.</para><para>Массив
    ↳ который нужно освободить в пул.</para></param>
102    [MethodImpl(MethodImplOptions.AggressiveInlining)]
103    public virtual void Free(T[] array)
104    {
105        if (array.IsNullOrEmpty())
106        {
107            return;
108        }
109        var stack = _pool.GetOrAdd(array.LongLength, size => new
    ↳ Stack<T[]>(_maxArraysPerSize));
110        if (stack.Count == _maxArraysPerSize) // Stack is full
111        {
112            return;
113        }
114        stack.Push(array);
115    }
116 }
117 }

```

1.5 ./csharp/Platform.Collections/Arrays/ArrayString.cs

```

1  using System.Runtime.CompilerServices;
2  using Platform.Collections.Segments;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Collections.Arrays
7  {
8      public class ArrayString<T> : Segment<T>
9      {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public ArrayString(int length) : base(new T[length], 0, length) { }
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public ArrayString(T[] array) : base(array, 0, array.Length) { }
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public ArrayString(T[] array, int length) : base(array, 0, length) { }
18     }
19 }

```

1.6 ./csharp/Platform.Collections/Arrays/CharArrayExtensions.cs

```

1  using System.Runtime.CompilerServices;
2
3  namespace Platform.Collections.Arrays
4  {
5      public static unsafe class CharArrayExtensions
6      {
7          /// <summary>
8          /// <para>Generates a hash code for an array segment with the specified offset and
    ↳ length. The hash code is generated based on the values of the array elements
    ↳ included in the specified segment.</para>
9          /// <para>Генерирует хэш-код сегмента массива с указанным смещением и длиной. Хэш-код
    ↳ генерируется на основе значений элементов массива входящих в указанный
    ↳ сегмент.</para>

```

```

10  /// </summary>
11  /// <param name="array"><para>The array to hash.</para><para>Массив для
    ↳ хеширования.</para></param>
12  /// <param name="offset"><para>The offset from which reading of the specified number of
    ↳ elements in the array starts.</para><para>Смещение, с которого начинается чтение
    ↳ указанного количества элементов в массиве.</para></param>
13  /// <param name="length"><para>The number of array elements used to calculate the
    ↳ hash.</para><para>Количество элементов массива, на основе которых будет вычислен
    ↳ хэш.</para></param>
14  /// <returns>
15  /// <para>The hash code of the segment in the array.</para>
16  /// <para>Хэш-код сегмента в массиве.</para>
17  /// </returns>
18  /// <remarks>
19  /// Based on https://github.com/Microsoft/referencesource/blob/3b1eaf5203992df69de44c783\_
    ↳ a3eda37d3d4cd10/mscorlib/system/string.cs#L833
20  /// </remarks>
21  [MethodImpl(MethodImplOptions.AggressiveInlining)]
22  public static int GenerateHashCode(this char[] array, int offset, int length)
23  {
24      var hashSeed = 5381;
25      var hashAccumulator = hashSeed;
26      fixed (char* arrayPointer = &array[offset])
27      {
28          for (char* charPointer = arrayPointer, last = charPointer + length; charPointer
    ↳ < last; charPointer++)
29          {
30              hashAccumulator = (hashAccumulator << 5) + hashAccumulator ^ *charPointer;
31          }
32      }
33      return hashAccumulator + (hashSeed * 1566083941);
34  }
35
36  /// <summary>
37  /// <para>Checks if all elements of two lists are equal.</para>
38  /// <para>Проверяет равны ли все элементы двух списков.</para>
39  /// </summary>
40  /// <param name="left"><para>The first compared array.</para><para>Первый массив для
    ↳ сравнения.</para></param>
41  /// <param name="leftOffset"><para>The offset from which reading of the specified number
    ↳ of elements in the first array starts.</para><para>Смещение, с которого начинается
    ↳ чтение элементов в первом массиве.</para></param>
42  /// <param name="length"><para>The number of checked elements.</para><para>Количество
    ↳ проверяемых элементов.</para></param>
43  /// <param name="right"><para>The second compared array.</para><para>Второй массив для
    ↳ сравнения.</para></param>
44  /// <param name="rightOffset"><para>The offset from which reading of the specified
    ↳ number of elements in the second array starts.</para><para>Смещение, с которого
    ↳ начинается чтение элементов в втором массиве.</para></param>
45  /// <returns>
46  /// <para><see langword="true"/> if the segments of the passed arrays are equal to each
    ↳ other otherwise <see langword="false"/>.</para>
47  /// <para><see langword="true"/>, если сегменты переданных массивов равны друг другу,
    ↳ иначе же <see langword="false"/>.</para>
48  /// </returns>
49  /// <remarks>
50  /// Based on https://github.com/Microsoft/referencesource/blob/3b1eaf5203992df69de44c783\_
    ↳ a3eda37d3d4cd10/mscorlib/system/string.cs#L364
51  /// </remarks>
52  [MethodImpl(MethodImplOptions.AggressiveInlining)]
53  public static bool ContentEqualTo(this char[] left, int leftOffset, int length, char[]
    ↳ right, int rightOffset)
54  {
55      fixed (char* leftPointer = &left[leftOffset])
56      {
57          fixed (char* rightPointer = &right[rightOffset])
58          {
59              char* leftPointerCopy = leftPointer, rightPointerCopy = rightPointer;
60              if (!CheckArraysMainPartForEquality(ref leftPointerCopy, ref
    ↳ rightPointerCopy, ref length))
61              {
62                  return false;
63              }
64              CheckArraysRemainderForEquality(ref leftPointerCopy, ref rightPointerCopy,
    ↳ ref length);
65              return length <= 0;

```

```

66     }
67 }
68 }
69
70 [MethodImpl(MethodImplOptions.AggressiveInlining)]
71 private static bool CheckArraysMainPartForEquality(ref char* left, ref char* right, ref
    ↪ int length)
72 {
73     while (length >= 10)
74     {
75         if ((* (int*)left != *(int*)right)
76             || (*(int*)(left + 2) != *(int*)(right + 2))
77             || (*(int*)(left + 4) != *(int*)(right + 4))
78             || (*(int*)(left + 6) != *(int*)(right + 6))
79             || (*(int*)(left + 8) != *(int*)(right + 8)))
80         {
81             return false;
82         }
83         left += 10;
84         right += 10;
85         length -= 10;
86     }
87     return true;
88 }
89
90 [MethodImpl(MethodImplOptions.AggressiveInlining)]
91 private static void CheckArraysRemainderForEquality(ref char* left, ref char* right, ref
    ↪ int length)
92 {
93     // This depends on the fact that the String objects are
94     // always zero terminated and that the terminating zero is not included
95     // in the length. For odd string sizes, the last compare will include
96     // the zero terminator.
97     while (length > 0)
98     {
99         if ((* (int*)left != *(int*)right)
100            {
101                break;
102            }
103            left += 2;
104            right += 2;
105            length -= 2;
106        }
107    }
108 }
109 }

```

1.7 ./csharp/Platform.Collections/Arrays/GenericArrayExtensions.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4
5  namespace Platform.Collections.Arrays
6  {
7      /// <summary>
8      /// <para>Represents a set of extension methods for a <see cref="T:T[]" /> array.</para>
9      /// <para>Представляет набор методов расширения для массива <see cref="T:T[]" />.</para>
10     /// </summary>
11     public static class GenericArrayExtensions
12     {
13         /// <summary>
14         /// <para>Checks if an array exists, if so, checks the array length using the index
15         ↪ variable type int, and if the array length is greater than the index - return
16         ↪ array[index], otherwise - default value.</para>
17         /// <para>Проверяет, существует ли массив, если да - идет проверка длины массива с
18         ↪ помощью переменной index, и если длина массива больше индекса - возвращает
19         ↪ array[index], иначе - значение по умолчанию.</para>
20         /// </summary>
21         /// <typeparam name="T"><para>Array elements type.</para><para>Тип элементов
22         ↪ массива.</para></typeparam>
23         /// <param name="array"><para>Array that will participate in
24         ↪ verification.</para><para>Массив который будет участвовать в
25         ↪ проверке.</para></param>
26         /// <param name="index"><para>Number type int to compare.</para><para>Число типа int для
27         ↪ сравнения.</para></param>
28         /// <returns><para>Array element or default value.</para><para>Элемент массива или же
29         ↪ значение по умолчанию.</para></returns>
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```



```

22 public static T GetElementOrDefault<T>(this T[] array, int index) => array != null &&
23     → array.Length > index ? array[index] : default;
24
25 /// <summary>
26 /// <para>Checks whether the array exists, if so, checks the array length using the
27     → index variable type long, and if the array length is greater than the index - return
28     → array[index], otherwise - default value.</para>
29 /// <para>Проверяет, существует ли массив, если да - идет проверка длины массива с
30     → помощью переменной index, и если длина массива больше индекса - возвращает
31     → array[index], иначе - значение по умолчанию.</para>
32 /// </summary>
33 /// <typeparam name="T"><para>Array elements type.</para><para>Тип элементов
34     → массива.</para></typeparam>
35 /// <param name="array"><para>Array that will participate in
36     → verification.</para><para>Массив который будет участвовать в
37     → проверке.</para></param>
38 /// <param name="index"><para>Number type long to compare.</para><para>Число типа long
39     → для сравнения.</para></param>
40 /// <returns><para>Array element or default value.</para><para>Элемент массива или же
41     → значение по умолчанию.</para></returns>
42 [MethodImpl(MethodImplOptions.AggressiveInlining)]
43 public static T GetElementOrDefault<T>(this T[] array, long index) => array != null &&
44     → array.LongLength > index ? array[index] : default;
45
46 /// <summary>
47 /// <para>Checks whether the array exist, if so, checks the array length using the index
48     → variable type int, and if the array length is greater than the index, set the element
49     → variable to array[index] and return <see langword="true"/>.</para>
50 /// <para>Проверяет, существует ли массив, если да, то идет проверка длины массива с
51     → помощью переменной index типа int, и если длина массива больше значения index,
52     → устанавливает значение переменной element - array[index] и возвращает <see
53     → langword="true"/>.</para>
54 /// </summary>
55 /// <typeparam name="T"><para>Array elements type.</para><para>Тип элементов
56     → массива.</para></typeparam>
57 /// <param name="array"><para>Array that will participate in
58     → verification.</para><para>Массив который будет участвовать в
59     → проверке.</para></param>
60 /// <param name="index"><para>Number type int to compare.</para><para>Число типа int для
61     → сравнения.</para></param>
62 /// <param name="element"><para>Passing the argument by reference, if successful, it
63     → will take the value array[index] otherwise default value.</para><para>Передаёт
64     → аргумент по ссылке, в случае успеха он примет значение array[index] в противном
65     → случае значение по умолчанию.</para></param>
66 /// <returns><para><see langword="true"/> if successful otherwise <see
67     → langword="false"/>.</para><para><see langword="true"/> в случае успеха, в противном
68     → случае <see langword="false"/>.</para></returns>
69 [MethodImpl(MethodImplOptions.AggressiveInlining)]
70 public static bool TryGetElement<T>(this T[] array, int index, out T element)
71 {
72     if (array != null && array.Length > index)
73     {
74         element = array[index];
75         return true;
76     }
77     else
78     {
79         element = default;
80         return false;
81     }
82 }
83
84 /// <summary>
85 /// <para>Checks whether the array exist, if so, checks the array length using the
86     → index variable type long, and if the array length is greater than the index, set the
87     → element variable to array[index] and return <see langword="true"/>.</para>
88 /// <para>Проверяет, существует ли массив, если да, то идет проверка длины массива с
89     → помощью переменной index типа long, и если длина массива больше значения index,
90     → устанавливает значение переменной element - array[index] и возвращает <see
91     → langword="true"/>.</para>
92 /// </summary>
93 /// <typeparam name="T"><para>Array elements type.</para><para>Тип элементов
94     → массива.</para></typeparam>
95 /// <param name="array"><para>Array that will participate in
96     → verification.</para><para>Массив который будет участвовать в
97     → проверке.</para></param>

```

```

65  /// <param name="index"><para>Number type long to compare.</para><para>Число типа long
    ↳ для сравнения.</para></param>
66  /// <param name="element"><para>Passing the argument by reference, if successful, it
    ↳ will take the value array[index] otherwise default value.</para><para>Передаёт
    ↳ аргумент по ссылке, в случае успеха он примет значение array[index] в противном
    ↳ случае значение по умолчанию.</para></param>
67  /// <returns><para><see langword="true"/> if successful otherwise <see
    ↳ langword="false"/>.</para><para><see langword="true"/> в случае успеха, в противном
    ↳ случае <see langword="false"/>.</para></returns>
68  [MethodImpl(MethodImplOptions.AggressiveInlining)]
69  public static bool TryGetElement<T>(this T[] array, long index, out T element)
70  {
71      if (array != null && array.LongLength > index)
72      {
73          element = array[index];
74          return true;
75      }
76      else
77      {
78          element = default;
79          return false;
80      }
81  }
82
83  /// <summary>
84  /// <para>Copying of elements from one array to another array.</para>
85  /// <para>Копирует элементы из одного массива в другой массив.</para>
86  /// </summary>
87  /// <typeparam name="T"><para>Array elements type.</para><para>Тип элементов
    ↳ массива.</para></typeparam>
88  /// <param name="array"><para>The array to copy.</para><para>Массив который необходимо
    ↳ скопировать.</para></param>
89  /// <returns><para>Copy of the array.</para><para>Копию массива.</para></returns>
90  [MethodImpl(MethodImplOptions.AggressiveInlining)]
91  public static T[] Clone<T>(this T[] array)
92  {
93      var copy = new T[array.LongLength];
94      Array.Copy(array, 0L, copy, 0L, array.LongLength);
95      return copy;
96  }
97
98  /// <summary>
99  /// <para>Shifts all the elements of the array by one position to the right.</para>
100  /// <para>Сдвигает вправо все элементы массива на одну позицию.</para>
101  /// </summary>
102  /// <typeparam name="T"><para>The array item type.</para><para>Тип элементов
    ↳ массива.</para></typeparam>
103  /// <param name="array"><para>The array to copy from.</para><para>Массив для
    ↳ копирования.</para></param>
104  /// <returns>
105  /// <para>Array with a shift of elements by one position.</para>
106  /// <para>Массив со сдвигом элементов на одну позицию.</para>
107  /// </returns>
108  [MethodImpl(MethodImplOptions.AggressiveInlining)]
109  public static IList<T> ShiftRight<T>(this T[] array) => array.ShiftRight(1L);
110
111  /// <summary>
112  /// <para>Shifts all elements of the array to the right by the specified number of
    ↳ elements.</para>
113  /// <para>Сдвигает вправо все элементы массива на указанное количество элементов.</para>
114  /// </summary>
115  /// <typeparam name="T"><para>The array item type.</para><para>Тип элементов
    ↳ массива.</para></typeparam>
116  /// <param name="array"><para>The array to copy from.</para><para>Массив для
    ↳ копирования.</para></param>
117  /// <param name="skip"><para>The number of items to shift.</para><para>Количество
    ↳ сдвигаемых элементов.</para></param>
118  /// <returns>
119  /// <para>If the value of the shift variable is less than zero - an <see
    ↳ cref="NotImplementedException"/> exception is thrown, but if the value of the shift
    ↳ variable is 0 - an exact copy of the array is returned. Otherwise, an array is
    ↳ returned with the shift of the elements.</para>
120  /// <para>Если значение переменной shift меньше нуля - выбрасывается исключение <see
    ↳ cref="NotImplementedException"/>, если же значение переменной shift равно 0 -
    ↳ возвращается точная копия массива. Иначе возвращается массив со сдвигом
    ↳ элементов.</para>
121  /// </returns>

```

```

122 [MethodImpl(MethodImplOptions.AggressiveInlining)]
123 public static IList<T> ShiftRight<T>(this T[] array, long shift)
124 {
125     if (shift < 0)
126     {
127         throw new NotImplementedException();
128     }
129     if (shift == 0)
130     {
131         return array.Clone<T>();
132     }
133     else
134     {
135         var restrictions = new T[array.LongLength + shift];
136         Array.Copy(array, 0L, restrictions, shift, array.LongLength);
137         return restrictions;
138     }
139 }
140
141 /// <summary>
142 /// <para>Adding in array the passed element at the specified position and increments
143   → position value by one.</para>
144 /// <para>Добавляет в массив переданный элемент на указанную позицию и увеличивает
145   → значение position на единицу.</para>
146 /// </summary>
147 /// <typeparam name="T"><para>Array elements type.</para>Тип элементов
148   → массива.</para></typeparam>
149 /// <param name="array"><para>The array to add the element to.</para><para>Массив в
150   → который необходимо добавить элемент.</para></param>
151 /// <param name="position"><para>A reference to the position of type int where the
152   → element will be added.</para><para>Ссылка на позицию типа int, в которую будет
153   → добавлен элемент.</para></param>
154 /// <param name="element"><para>The element to add to the array.</para><para>Элемент,
155   → который нужно добавить в массив.</para></param>
156 [MethodImpl(MethodImplOptions.AggressiveInlining)]
157 public static void Add<T>(this T[] array, ref int position, T element) =>
158   → array[position++] = element;
159
160 /// <summary>
161 /// <para>Adding in array the passed element at the specified position and increments
162   → position value by one.</para>
163 /// <para>Добавляет в массив переданный элемент на указанную позицию и увеличивает
164   → значение position на единицу.</para>
165 /// </summary>
166 /// <typeparam name="T"><para>Array elements type.</para>Тип элементов
167   → массива.</para></typeparam>
168 /// <param name="array"><para>The array to add the element to.</para><para>Массив в
169   → который необходимо добавить элемент.</para></param>
170 /// <param name="position"><para>A reference to the position of type long where the
171   → element will be added.</para><para>Ссылка на позицию типа long, в которую будет
172   → добавлен элемент.</para></param>
173 /// <param name="element"><para>The element to add to the array.</para><para>Элемент
174   → который необходимо добавить в массив.</para></param>
175 [MethodImpl(MethodImplOptions.AggressiveInlining)]
176 public static void Add<T>(this T[] array, ref long position, T element) =>
177   → array[position++] = element;
178
179 /// <summary>
180 /// <para>Adding in array the passed element, at the specified position, increments
181   → position value by one and returns the value of the passed constant.</para>
182 /// <para>Добавляет в массив переданный элемент на указанную позицию, увеличивает
183   → значение position на единицу и возвращает значение переданной константы.</para>
184 /// </summary>
185 /// <typeparam name="TElement"><para>The array element type.</para><para>Тип элемента
186   → массива.</para></typeparam>
187 /// <typeparam name="TReturnConstant"><para>Type of return constant.</para><para>Тип
188   → возвращаемой константы.</para></typeparam>
189 /// <param name="array"><para>The array to add the element to.</para><para>Массив в
190   → который необходимо добавить элемент.</para></param>
191 /// <param name="position"><para>Reference to the position to which the element will be
192   → added.</para><para>Ссылка на позицию, в которую будет добавлен
193   → элемент.</para></param>
194 /// <param name="element"><para>The element to add to the array.</para><para>Элемент
195   → который необходимо добавить в массив.</para></param>
196 /// <param name="returnConstant"><para>The constant value that will be
197   → returned.</para><para>Значение константы, которое будет возвращено.</para></param>

```

```

173 /// <returns>
174 /// <para>The constant value passed as an argument.</para>
175 /// <para>Значение константы, переданное в качестве аргумента.</para>
176 /// </returns>
177 [MethodImpl(MethodImplOptions.AggressiveInlining)]
178 public static TReturnConstant AddAndReturnConstant<TElement, TReturnConstant>(this
    → TElement[] array, ref long position, TElement element, TReturnConstant
    → returnConstant)
179 {
180     array.Add(ref position, element);
181     return returnConstant;
182 }
183
184 /// <summary>
185 /// <para>Adds the first element from the passed collection to the array, at the
    → specified position and increments position value by one.</para>
186 /// <para>Добавляет в массив первый элемент из переданной коллекции, на указанную
    → позицию и увеличивает значение position на единицу.</para>
187 /// </summary>
188 /// <typeparam name="T"><para>Array element type.</para><para>Тип элементов
    → массива.</para></typeparam>
189 /// <param name="array"><para>The array to add the element to.</para><para>Массив в
    → который необходимо добавить элемент.</para></param>
190 /// <param name="position"><para>Reference to the position to which the element will be
    → added.</para><para>Ссылка на позицию, в которую будет добавлен
    → элемент.</para></param>
191 /// <param name="elements"><para>List, the first element of which will be added to the
    → array.</para><para>Список, первый элемент которого будет добавлен в
    → массив.</para></param>
192 [MethodImpl(MethodImplOptions.AggressiveInlining)]
193 public static void AddFirst<T>(this T[] array, ref long position, IList<T> elements) =>
    → array[position++] = elements[0];
194
195 /// <summary>
196 /// <para>Adds the first element from the passed collection to the array, at the
    → specified position, increments position value by one and returns the value of the
    → passed constant.</para>
197 /// <para>Добавляет в массив первый элемент из переданной коллекции, на указанную
    → позицию, увеличивает значение position на единицу и возвращает значение переданной
    → константы.</para>
198 /// </summary>
199 /// <typeparam name="TElement"><para>The array element type.</para><para>Тип элемента
    → массива.</para></typeparam>
200 /// <typeparam name="TReturnConstant"><para>Type of return constant.</para><para>Тип
    → возвращаемой константы.</para></typeparam>
201 /// <param name="array"><para>The array to add the element to.</para><para>Массив в
    → который необходимо добавить элемент.</para></param>
202 /// <param name="position"><para>Reference to the position to which the element will be
    → added.</para><para>Ссылка на позицию, в которую будет добавлен
    → элемент.</para></param>
203 /// <param name="elements"><para>List, the first element of which will be added to the
    → array.</para><para>Список, первый элемент которого будет добавлен в
    → массив.</para></param>
204 /// <param name="returnConstant"><para>The constant value that will be
    → returned.</para><para>Значение константы, которое будет возвращено.</para></param>
205 /// <returns>
206 /// <para>The constant value passed as an argument.</para>
207 /// <para>Значение константы, переданное в качестве аргумента.</para>
208 /// </returns>
209 [MethodImpl(MethodImplOptions.AggressiveInlining)]
210 public static TReturnConstant AddFirstAndReturnConstant<TElement, TReturnConstant>(this
    → TElement[] array, ref long position, IList<TElement> elements, TReturnConstant
    → returnConstant)
211 {
212     array.AddFirst(ref position, elements);
213     return returnConstant;
214 }
215
216 /// <summary>
217 /// <para>Adding in array all elements from the passed collection, at the specified
    → position, increases the position value by the number of elements added and returns
    → the value of the passed constant.</para>
218 /// <para>Добавляет в массив все элементы из переданной коллекции, на указанную позицию,
    → увеличивает значение position на количество добавленных элементов и возвращает
    → значение переданной константы.</para>
219 /// </summary>

```

```

220 /// <typeparam name="TElement"><para>The array element type.</para><para>Тип элемента
221   ↳ массива.</para></typeparam>
222 /// <typeparam name="TReturnConstant"><para>Type of return constant.</para><para>Тип
223   ↳ возвращаемой константы.</para></typeparam>
224 /// <param name="array"><para>The array to add the element to.</para><para>Массив в
225   ↳ который необходимо добавить элементы.</para></param>
226 /// <param name="position"><para>Reference to the position from which elements will be
227   ↳ added to the array.</para><para>Ссылка на позицию, начиная с которой будут
228   ↳ добавляться элементы в массив.</para></param>
229 /// <param name="elements"><para>List, whose elements will be added to the
230   ↳ array.</para><para>Список, элементы которого будут добавлены в
231   ↳ массив.</para></param>
232 /// <param name="returnConstant"><para>The constant value that will be
233   ↳ returned.</para><para>Значение константы, которое будет возвращено.</para></param>
234 /// <returns>
235 /// <para>The constant value passed as an argument.</para>
236 /// <para>Значение константы, переданное в качестве аргумента.</para>
237 /// </returns>
238 [MethodImpl(MethodImplOptions.AggressiveInlining)]
239 public static TReturnConstant AddAllAndReturnConstant<TElement, TReturnConstant>(this
240   ↳ TElement[] array, ref long position, IList<TElement> elements, TReturnConstant
241   ↳ returnConstant)
242 {
243     array.AddAll(ref position, elements);
244     return returnConstant;
245 }
246
247 /// <summary>
248 /// <para>Adding in array a collection of elements, starting from a specific position
249   ↳ and increases the position value by the number of elements added.</para>
250 /// <para>Добавляет в массив все элементы коллекции, начиная с определенной позиции и
251   ↳ увеличивает значение position на количество добавленных элементов.</para>
252 /// </summary>
253 /// <typeparam name="T"><para>Array elements type.</para><para>Тип элементов
254   ↳ массива.</para></typeparam>
255 /// <param name="array"><para>The array to add the element to.</para><para>Массив в
256   ↳ который необходимо добавить элементы.</para></param>
257 /// <param name="position"><para>Reference to the position from which elements will be
258   ↳ added to the array.</para><para>Ссылка на позицию, начиная с которой будут
259   ↳ добавляться элементы в массив.</para></param>
260 /// <param name="elements"><para>List, whose elements will be added to the
261   ↳ array.</para><para>Список, элементы которого будут добавлены в
262   ↳ массив.</para></param>
263 [MethodImpl(MethodImplOptions.AggressiveInlining)]
264 public static void AddAll<T>(this T[] array, ref long position, IList<T> elements)
265 {
266     for (var i = 0; i < elements.Count; i++)
267     {
268         array.Add(ref position, elements[i]);
269     }
270 }
271
272 /// <summary>
273 /// <para>Adding in array all elements of the collection, skipping the first position,
274   ↳ increments position value by one and returns the value of the passed constant.</para>
275 /// <para>Добавляет в массив все элементы коллекции, пропуская первую позицию,
276   ↳ увеличивает значение position на единицу и возвращает значение переданной
277   ↳ константы.</para>
278 /// </summary>
279 /// <typeparam name="TElement"><para>The array element type.</para><para>Тип элемента
280   ↳ массива.</para></typeparam>
281 /// <typeparam name="TReturnConstant"><para>Type of return constant.</para><para>Тип
282   ↳ возвращаемой константы.</para></typeparam>
283 /// <param name="array"><para>The array to add items to.</para><para>Массив в который
284   ↳ необходимо добавить элементы.</para></param>
285 /// <param name="position"><para>Reference to the position from which to start adding
286   ↳ elements.</para><para>Ссылка на позицию, с которой начинается добавление
287   ↳ элементов.</para></param>
288 /// <param name="elements"><para>List, whose elements will be added to the
289   ↳ array.</para><para>Список, элементы которого будут добавлены в
290   ↳ массив.</para></param>
291 /// <param name="returnConstant"><para>The constant value that will be
292   ↳ returned.</para><para>Значение константы, которое будет возвращено.</para></param>
293 /// <returns>
294 /// <para>The constant value passed as an argument.</para>

```

```

266 /// <para>Значение константы, переданное в качестве аргумента.</para>
267 /// </returns>
268 [MethodImpl(MethodImplOptions.AggressiveInlining)]
269 public static TReturnConstant AddSkipFirstAndReturnConstant<TElement,
    → TReturnConstant>(this TElement[] array, ref long position, IList<TElement> elements,
    → TReturnConstant returnConstant)
270 {
271     array.AddSkipFirst(ref position, elements);
272     return returnConstant;
273 }
274
275 /// <summary>
276 /// <para>Adding in array all elements of the collection, skipping the first position
    → and increments position value by one.</para>
277 /// <para>Добавляет в массив все элементы коллекции, пропуская первую позицию и
    → увеличивает значение position на единицу.</para>
278 /// </summary>
279 /// <typeparam name="T"><para>Array elements type.</para><para>Тип элементов
    → массива.</para></typeparam>
280 /// <param name="array"><para>The array to add items to.</para><para>Массив в который
    → необходимо добавить элементы.</para></param>
281 /// <param name="position"><para>Reference to the position from which to start adding
    → elements.</para><para>Ссылка на позицию, с которой начинается добавление
    → элементов.</para></param>
282 /// <param name="elements"><para>List, whose elements will be added to the
    → array.</para><para>Список, элементы которого будут добавлены в
    → массив.</para></param>
283 [MethodImpl(MethodImplOptions.AggressiveInlining)]
284 public static void AddSkipFirst<T>(this T[] array, ref long position, IList<T> elements)
    → => array.AddSkipFirst(ref position, elements, 1);
285
286 /// <summary>
287 /// <para>Adding in array all but the first element, skipping a specified number of
    → positions and increments position value by one.</para>
288 /// <para>Добавляет в массив все элементы коллекции, кроме первого, пропуская
    → определенное количество позиций и увеличивает значение position на единицу.</para>
289 /// </summary>
290 /// <typeparam name="T"><para>Array elements type.</para><para>Тип элементов
    → массива.</para></typeparam>
291 /// <param name="array"><para>The array to add items to.</para><para>Массив в который
    → необходимо добавить элементы.</para></param>
292 /// <param name="position"><para>Reference to the position from which to start adding
    → elements.</para><para>Ссылка на позицию, с которой начинается добавление
    → элементов.</para></param>
293 /// <param name="elements"><para>List, whose elements will be added to the
    → array.</para><para>Список, элементы которого будут добавлены в
    → массив.</para></param>
294 /// <param name="skip"><para>Number of elements to skip.</para><para>Количество
    → пропускаемых элементов.</para></param>
295 [MethodImpl(MethodImplOptions.AggressiveInlining)]
296 public static void AddSkipFirst<T>(this T[] array, ref long position, IList<T> elements,
    → int skip)
297 {
298     for (var i = skip; i < elements.Count; i++)
299     {
300         array.Add(ref position, elements[i]);
301     }
302 }
303 }
304 }

```

1.8 ./csharp/Platform.Collections/BitString.cs

```

1 using System;
2 using System.Collections.Concurrent;
3 using System.Collections.Generic;
4 using System.Numerics;
5 using System.Runtime.CompilerServices;
6 using System.Threading.Tasks;
7 using Platform.Exceptions;
8 using Platform.Ranges;
9
10 // ReSharper disable ForCanBeConvertedToForeach
11 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
12
13 namespace Platform.Collections
14 {
15     /// <remarks>

```

```

16  /// А что если хранить карту значений, где каждый бит будет означать присутствует ли блок в
17  ↪ 64 бит в массиве значений.
18  /// 64 бита по 0 бит, будут означать отсутствие 64-х блоков по 64 бита. Т.е. упаковка 512
19  ↪ байт в 8 байт.
20  /// Подобный принцип можно применять и к 64-ём блокам и т.п. По сути это карта значений. С
21  ↪ помощью которой можно быстро
22  /// проверять есть ли значения непосредственно далее (ниже по уровню).
23  /// Или как таблица виртуальной памяти где номер блока означает его присутствие и адрес.
24  </remarks>
25  public class BitString : IEquatable<BitString>
26  {
27      private static readonly byte[] [] _bitsSetIn16Bits;
28      private long[] _array;
29      private long _length;
30      private long _minPositiveWord;
31      private long _maxPositiveWord;
32
33      public bool this[long index]
34      {
35          [MethodImpl(MethodImplOptions.AggressiveInlining)]
36          get => Get(index);
37          [MethodImpl(MethodImplOptions.AggressiveInlining)]
38          set => Set(index, value);
39      }
40
41      public long Length
42      {
43          [MethodImpl(MethodImplOptions.AggressiveInlining)]
44          get => _length;
45          [MethodImpl(MethodImplOptions.AggressiveInlining)]
46          set
47          {
48              if (_length == value)
49              {
50                  return;
51              }
52              Ensure.Always.ArgumentInRange(value, GetValidLengthRange(), nameof(Length));
53              // Currently we never shrink the array
54              if (value > _length)
55              {
56                  var words = GetWordsCountFromIndex(value);
57                  var oldWords = GetWordsCountFromIndex(_length);
58                  if (words > _array.LongLength)
59                  {
60                      var copy = new long[words];
61                      Array.Copy(_array, copy, _array.LongLength);
62                      _array = copy;
63                  }
64                  else
65                  {
66                      // What is going on here?
67                      Array.Clear(_array, (int)oldWords, (int)(words - oldWords));
68                  }
69                  // What is going on here?
70                  var mask = (int)(_length % 64);
71                  if (mask > 0)
72                  {
73                      _array[oldWords - 1] &= (1L << mask) - 1;
74                  }
75              }
76              else
77              {
78                  // Looks like minimum and maximum positive words are not updated
79                  throw new NotImplementedException();
80              }
81              _length = value;
82          }
83      }
84
85      #region Constructors
86
87      [MethodImpl(MethodImplOptions.AggressiveInlining)]
88      static BitString()
89      {
90          _bitsSetIn16Bits = new byte[65536] [];
91          int i, c, k;
92          byte bitIndex;
93          for (i = 0; i < 65536; i++)
94          {

```

```

92         // Calculating size of array (number of positive bits)
93         for (c = 0, k = 1; k <= 65536; k <= 1)
94         {
95             if ((i & k) == k)
96             {
97                 c++;
98             }
99         }
100         var array = new byte[c];
101         // Adding positive bits indices into array
102         for (bitIndex = 0, c = 0, k = 1; k <= 65536; k <= 1)
103         {
104             if ((i & k) == k)
105             {
106                 array[c++] = bitIndex;
107             }
108             bitIndex++;
109         }
110         _bitsSetIn16Bits[i] = array;
111     }
112 }
113
114 [MethodImpl(MethodImplOptions.AggressiveInlining)]
115 public BitString(BitString other)
116 {
117     Ensure.Always.ArgumentNotNull(other, nameof(other));
118     _length = other._length;
119     _array = new long[GetWordsCountFromIndex(_length)];
120     _minPositiveWord = other._minPositiveWord;
121     _maxPositiveWord = other._maxPositiveWord;
122     Array.Copy(other._array, _array, _array.LongLength);
123 }
124
125 [MethodImpl(MethodImplOptions.AggressiveInlining)]
126 public BitString(long length)
127 {
128     Ensure.Always.ArgumentInRange(length, GetValidLengthRange(), nameof(length));
129     _length = length;
130     _array = new long[GetWordsCountFromIndex(_length)];
131     MarkBordersAsAllBitsReset();
132 }
133
134 [MethodImpl(MethodImplOptions.AggressiveInlining)]
135 public BitString(long length, bool defaultValue)
136     : this(length)
137 {
138     if (defaultValue)
139     {
140         SetAll();
141     }
142 }
143
144 #endregion
145
146 [MethodImpl(MethodImplOptions.AggressiveInlining)]
147 public BitString Not()
148 {
149     for (var i = 0L; i < _array.LongLength; i++)
150     {
151         _array[i] = ~_array[i];
152         RefreshBordersByWord(i);
153     }
154     return this;
155 }
156
157 [MethodImpl(MethodImplOptions.AggressiveInlining)]
158 public BitString ParallelNot()
159 {
160     var threads = Environment.ProcessorCount / 2;
161     if (threads <= 1)
162     {
163         return Not();
164     }
165     var partitioner = Partitioner.Create(0L, _array.LongLength, _array.LongLength /
166     ↪ threads);
167     Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
168     ↪ MaxDegreeOfParallelism = threads }, range =>
169     {
170         var maximum = range.Item2;

```



```

169         for (var i = range.Item1; i < maximum; i++)
170         {
171             _array[i] = ~_array[i];
172         }
173     });
174     MarkBordersAsAllBitsSet();
175     TryShrinkBorders();
176     return this;
177 }
178
179 [MethodImpl(MethodImplOptions.AggressiveInlining)]
180 public BitString VectorNot()
181 {
182     if (!Vector.IsHardwareAccelerated || _array.LongLength >= int.MaxValue)
183     {
184         return Not();
185     }
186     var step = Vector<long>.Count;
187     if (_array.Length < step)
188     {
189         return Not();
190     }
191     VectorNotLoop(_array, step, 0, _array.Length);
192     MarkBordersAsAllBitsSet();
193     TryShrinkBorders();
194     return this;
195 }
196
197 [MethodImpl(MethodImplOptions.AggressiveInlining)]
198 public BitString ParallelVectorNot()
199 {
200     var threads = Environment.ProcessorCount / 2;
201     if (threads <= 1)
202     {
203         return VectorNot();
204     }
205     if (!Vector.IsHardwareAccelerated)
206     {
207         return ParallelNot();
208     }
209     var step = Vector<long>.Count;
210     if (_array.Length < (step * threads))
211     {
212         return VectorNot();
213     }
214     var partitioner = Partitioner.Create(0, _array.Length, _array.Length / threads);
215     Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
216         ↪ MaxDegreeOfParallelism = threads }, range => VectorNotLoop(_array, step,
217         ↪ range.Item1, range.Item2));
218     MarkBordersAsAllBitsSet();
219     TryShrinkBorders();
220     return this;
221 }
222
223 [MethodImpl(MethodImplOptions.AggressiveInlining)]
224 static private void VectorNotLoop(long[] array, int step, int start, int maximum)
225 {
226     var i = start;
227     var range = maximum - start - 1;
228     var stop = range - (range % step);
229     for (; i < stop; i += step)
230     {
231         (~new Vector<long>(array, i)).CopyTo(array, i);
232     }
233     for (; i < maximum; i++)
234     {
235         array[i] = ~array[i];
236     }
237 }
238
239 [MethodImpl(MethodImplOptions.AggressiveInlining)]
240 public BitString And(BitString other)
241 {
242     EnsureBitStringHasTheSameSize(other, nameof(other));
243     GetCommonOuterBorders(this, other, out long from, out long to);
244     var otherArray = other._array;
245     for (var i = from; i <= to; i++)
246     {

```

```

245         _array[i] &= otherArray[i];
246         RefreshBordersByWord(i);
247     }
248     return this;
249 }
250
251 [MethodImpl(MethodImplOptions.AggressiveInlining)]
252 public BitString ParallelAnd(BitString other)
253 {
254     var threads = Environment.ProcessorCount / 2;
255     if (threads <= 1)
256     {
257         return And(other);
258     }
259     EnsureBitStringHasTheSameSize(other, nameof(other));
260     GetCommonOuterBorders(this, other, out long from, out long to);
261     var partitioner = Partitioner.Create(from, to + 1, (to - from) / threads);
262     Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
263         ↪ MaxDegreeOfParallelism = threads }, range =>
264     {
265         var maximum = range.Item2;
266         for (var i = range.Item1; i < maximum; i++)
267         {
268             _array[i] &= other._array[i];
269         }
270     });
271     MarkBordersAsAllBitsSet();
272     TryShrinkBorders();
273     return this;
274 }
275
276 [MethodImpl(MethodImplOptions.AggressiveInlining)]
277 public BitString VectorAnd(BitString other)
278 {
279     if (!Vector.IsHardwareAccelerated || _array.LongLength >= int.MaxValue)
280     {
281         return And(other);
282     }
283     var step = Vector<long>.Count;
284     if (_array.Length < step)
285     {
286         return And(other);
287     }
288     EnsureBitStringHasTheSameSize(other, nameof(other));
289     GetCommonOuterBorders(this, other, out int from, out int to);
290     VectorAndLoop(_array, other._array, step, from, to + 1);
291     MarkBordersAsAllBitsSet();
292     TryShrinkBorders();
293     return this;
294 }
295
296 [MethodImpl(MethodImplOptions.AggressiveInlining)]
297 public BitString ParallelVectorAnd(BitString other)
298 {
299     var threads = Environment.ProcessorCount / 2;
300     if (threads <= 1)
301     {
302         return VectorAnd(other);
303     }
304     if (!Vector.IsHardwareAccelerated)
305     {
306         return ParallelAnd(other);
307     }
308     var step = Vector<long>.Count;
309     if (_array.Length < (step * threads))
310     {
311         return VectorAnd(other);
312     }
313     EnsureBitStringHasTheSameSize(other, nameof(other));
314     GetCommonOuterBorders(this, other, out int from, out int to);
315     var partitioner = Partitioner.Create(from, to + 1, (to - from) / threads);
316     Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
317         ↪ MaxDegreeOfParallelism = threads }, range => VectorAndLoop(_array, other._array,
318         ↪ step, range.Item1, range.Item2));
319     MarkBordersAsAllBitsSet();
320     TryShrinkBorders();
321     return this;
322 }

```

```

320 [MethodImpl(MethodImplOptions.AggressiveInlining)]
321 static private void VectorAndLoop(long[] array, long[] otherArray, int step, int start,
322 ↪ int maximum)
323 {
324     var i = start;
325     var range = maximum - start - 1;
326     var stop = range - (range % step);
327     for (; i < stop; i += step)
328     {
329         (new Vector<long>(array, i) & new Vector<long>(otherArray, i)).CopyTo(array, i);
330     }
331     for (; i < maximum; i++)
332     {
333         array[i] &= otherArray[i];
334     }
335 }
336
337 [MethodImpl(MethodImplOptions.AggressiveInlining)]
338 public BitString Or(BitString other)
339 {
340     EnsureBitStringHasTheSameSize(other, nameof(other));
341     GetCommonOuterBorders(this, other, out long from, out long to);
342     for (var i = from; i <= to; i++)
343     {
344         _array[i] |= other._array[i];
345         RefreshBordersByWord(i);
346     }
347     return this;
348 }
349
350 [MethodImpl(MethodImplOptions.AggressiveInlining)]
351 public BitString ParallelOr(BitString other)
352 {
353     var threads = Environment.ProcessorCount / 2;
354     if (threads <= 1)
355     {
356         return Or(other);
357     }
358     EnsureBitStringHasTheSameSize(other, nameof(other));
359     GetCommonOuterBorders(this, other, out long from, out long to);
360     var partitioner = Partitioner.Create(from, to + 1, (to - from) / threads);
361     Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
362 ↪ MaxDegreeOfParallelism = threads }, range =>
363     {
364         var maximum = range.Item2;
365         for (var i = range.Item1; i < maximum; i++)
366         {
367             _array[i] |= other._array[i];
368         }
369     });
370     MarkBordersAsAllBitsSet();
371     TryShrinkBorders();
372     return this;
373 }
374
375 [MethodImpl(MethodImplOptions.AggressiveInlining)]
376 public BitString VectorOr(BitString other)
377 {
378     if (!Vector.IsHardwareAccelerated || _array.LongLength >= int.MaxValue)
379     {
380         return Or(other);
381     }
382     var step = Vector<long>.Count;
383     if (_array.Length < step)
384     {
385         return Or(other);
386     }
387     EnsureBitStringHasTheSameSize(other, nameof(other));
388     GetCommonOuterBorders(this, other, out int from, out int to);
389     VectorOrLoop(_array, other._array, step, from, to + 1);
390     MarkBordersAsAllBitsSet();
391     TryShrinkBorders();
392     return this;
393 }
394
395 [MethodImpl(MethodImplOptions.AggressiveInlining)]
396 public BitString ParallelVectorOr(BitString other)

```

```

396 {
397     var threads = Environment.ProcessorCount / 2;
398     if (threads <= 1)
399     {
400         return VectorOr(other);
401     }
402     if (!Vector.IsHardwareAccelerated)
403     {
404         return ParallelOr(other);
405     }
406     var step = Vector<long>.Count;
407     if (_array.Length < (step * threads))
408     {
409         return VectorOr(other);
410     }
411     EnsureBitStringHasTheSameSize(other, nameof(other));
412     GetCommonOuterBorders(this, other, out int from, out int to);
413     var partitioner = Partitioner.Create(from, to + 1, (to - from) / threads);
414     Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
415         ↪ MaxDegreeOfParallelism = threads }, range => VectorOrLoop(_array, other._array,
416         ↪ step, range.Item1, range.Item2));
417     MarkBordersAsAllBitsSet();
418     TryShrinkBorders();
419     return this;
420 }
421
422 [MethodImpl(MethodImplOptions.AggressiveInlining)]
423 static private void VectorOrLoop(long[] array, long[] otherArray, int step, int start,
424 ↪ int maximum)
425 {
426     var i = start;
427     var range = maximum - start - 1;
428     var stop = range - (range % step);
429     for (; i < stop; i += step)
430     {
431         (new Vector<long>(array, i) | new Vector<long>(otherArray, i)).CopyTo(array, i);
432     }
433     for (; i < maximum; i++)
434     {
435         array[i] |= otherArray[i];
436     }
437 }
438
439 [MethodImpl(MethodImplOptions.AggressiveInlining)]
440 public BitString Xor(BitString other)
441 {
442     EnsureBitStringHasTheSameSize(other, nameof(other));
443     GetCommonOuterBorders(this, other, out long from, out long to);
444     for (var i = from; i <= to; i++)
445     {
446         _array[i] ^= other._array[i];
447         RefreshBordersByWord(i);
448     }
449     return this;
450 }
451
452 [MethodImpl(MethodImplOptions.AggressiveInlining)]
453 public BitString ParallelXor(BitString other)
454 {
455     var threads = Environment.ProcessorCount / 2;
456     if (threads <= 1)
457     {
458         return Xor(other);
459     }
460     EnsureBitStringHasTheSameSize(other, nameof(other));
461     GetCommonOuterBorders(this, other, out long from, out long to);
462     var partitioner = Partitioner.Create(from, to + 1, (to - from) / threads);
463     Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
464         ↪ MaxDegreeOfParallelism = threads }, range =>
465     {
466         var maximum = range.Item2;
467         for (var i = range.Item1; i < maximum; i++)
468         {
469             _array[i] ^= other._array[i];
470         }
471     });
472     MarkBordersAsAllBitsSet();
473     TryShrinkBorders();
474 }

```

```

470     return this;
471 }
472
473 [MethodImpl(MethodImplOptions.AggressiveInlining)]
474 public BitString VectorXor(BitString other)
475 {
476     if (!Vector.IsHardwareAccelerated || _array.LongLength >= int.MaxValue)
477     {
478         return Xor(other);
479     }
480     var step = Vector<long>.Count;
481     if (_array.Length < step)
482     {
483         return Xor(other);
484     }
485     EnsureBitStringHasTheSameSize(other, nameof(other));
486     GetCommonOuterBorders(this, other, out int from, out int to);
487     VectorXorLoop(_array, other._array, step, from, to + 1);
488     MarkBordersAsAllBitsSet();
489     TryShrinkBorders();
490     return this;
491 }
492
493 [MethodImpl(MethodImplOptions.AggressiveInlining)]
494 public BitString ParallelVectorXor(BitString other)
495 {
496     var threads = Environment.ProcessorCount / 2;
497     if (threads <= 1)
498     {
499         return VectorXor(other);
500     }
501     if (!Vector.IsHardwareAccelerated)
502     {
503         return ParallelXor(other);
504     }
505     var step = Vector<long>.Count;
506     if (_array.Length < (step * threads))
507     {
508         return VectorXor(other);
509     }
510     EnsureBitStringHasTheSameSize(other, nameof(other));
511     GetCommonOuterBorders(this, other, out int from, out int to);
512     var partitioner = Partitioner.Create(from, to + 1, (to - from) / threads);
513     Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
514         ↪ MaxDegreeOfParallelism = threads }, range => VectorXorLoop(_array, other._array,
515         ↪ step, range.Item1, range.Item2));
516     MarkBordersAsAllBitsSet();
517     TryShrinkBorders();
518     return this;
519 }
520
521 [MethodImpl(MethodImplOptions.AggressiveInlining)]
522 static private void VectorXorLoop(long[] array, long[] otherArray, int step, int start,
523     ↪ int maximum)
524 {
525     var i = start;
526     var range = maximum - start - 1;
527     var stop = range - (range % step);
528     for (; i < stop; i += step)
529     {
530         (new Vector<long>(array, i) ^ new Vector<long>(otherArray, i)).CopyTo(array, i);
531     }
532     for (; i < maximum; i++)
533     {
534         array[i] ^= otherArray[i];
535     }
536 }
537
538 [MethodImpl(MethodImplOptions.AggressiveInlining)]
539 private void RefreshBordersByWord(long wordIndex)
540 {
541     if (_array[wordIndex] == 0)
542     {
543         if (wordIndex == _minPositiveWord && wordIndex != _array.LongLength - 1)
544         {
545             _minPositiveWord++;
546         }
547         if (wordIndex == _maxPositiveWord && wordIndex != 0)

```

```

545         {
546             _maxPositiveWord--;
547         }
548     }
549     else
550     {
551         if (wordIndex < _minPositiveWord)
552         {
553             _minPositiveWord = wordIndex;
554         }
555         if (wordIndex > _maxPositiveWord)
556         {
557             _maxPositiveWord = wordIndex;
558         }
559     }
560 }
561
562 [MethodImpl(MethodImplOptions.AggressiveInlining)]
563 public bool TryShrinkBorders()
564 {
565     GetBorders(out long from, out long to);
566     while (from <= to && _array[from] == 0)
567     {
568         from++;
569     }
570     if (from > to)
571     {
572         MarkBordersAsAllBitsReset();
573         return true;
574     }
575     while (to >= from && _array[to] == 0)
576     {
577         to--;
578     }
579     if (to < from)
580     {
581         MarkBordersAsAllBitsReset();
582         return true;
583     }
584     var bordersUpdated = from != _minPositiveWord || to != _maxPositiveWord;
585     if (bordersUpdated)
586     {
587         SetBorders(from, to);
588     }
589     return bordersUpdated;
590 }
591
592 [MethodImpl(MethodImplOptions.AggressiveInlining)]
593 public bool Get(long index)
594 {
595     Ensure.Always.ArgumentInRange(index, GetValidIndexRange(), nameof(index));
596     return (_array[GetWordIndexFromIndex(index)] & GetBitMaskFromIndex(index)) != 0;
597 }
598
599 [MethodImpl(MethodImplOptions.AggressiveInlining)]
600 public void Set(long index, bool value)
601 {
602     if (value)
603     {
604         Set(index);
605     }
606     else
607     {
608         Reset(index);
609     }
610 }
611
612 [MethodImpl(MethodImplOptions.AggressiveInlining)]
613 public void Set(long index)
614 {
615     Ensure.Always.ArgumentInRange(index, GetValidIndexRange(), nameof(index));
616     var wordIndex = GetWordIndexFromIndex(index);
617     var mask = GetBitMaskFromIndex(index);
618     _array[wordIndex] |= mask;
619     RefreshBordersByWord(wordIndex);
620 }
621
622 [MethodImpl(MethodImplOptions.AggressiveInlining)]
623 public void Reset(long index)

```

```

624 {
625     Ensure.Always.ArgumentInRange(index, GetValidIndexRange(), nameof(index));
626     var wordIndex = GetWordIndexFromIndex(index);
627     var mask = GetBitMaskFromIndex(index);
628     _array[wordIndex] &= ~mask;
629     RefreshBordersByWord(wordIndex);
630 }
631
632 [MethodImpl(MethodImplOptions.AggressiveInlining)]
633 public bool Add(long index)
634 {
635     var wordIndex = GetWordIndexFromIndex(index);
636     var mask = GetBitMaskFromIndex(index);
637     if ((_array[wordIndex] & mask) == 0)
638     {
639         _array[wordIndex] |= mask;
640         RefreshBordersByWord(wordIndex);
641         return true;
642     }
643     else
644     {
645         return false;
646     }
647 }
648
649 [MethodImpl(MethodImplOptions.AggressiveInlining)]
650 public void SetAll(bool value)
651 {
652     if (value)
653     {
654         SetAll();
655     }
656     else
657     {
658         ResetAll();
659     }
660 }
661
662 [MethodImpl(MethodImplOptions.AggressiveInlining)]
663 public void SetAll()
664 {
665     const long fillValue = unchecked((long)0xffffffffffffffff);
666     var words = GetWordsCountFromIndex(_length);
667     for (var i = 0; i < words; i++)
668     {
669         _array[i] = fillValue;
670     }
671     MarkBordersAsAllBitsSet();
672 }
673
674 [MethodImpl(MethodImplOptions.AggressiveInlining)]
675 public void ResetAll()
676 {
677     const long fillValue = 0;
678     GetBorders(out long from, out long to);
679     for (var i = from; i <= to; i++)
680     {
681         _array[i] = fillValue;
682     }
683     MarkBordersAsAllBitsReset();
684 }
685
686 [MethodImpl(MethodImplOptions.AggressiveInlining)]
687 public List<long> GetSetIndices()
688 {
689     var result = new List<long>();
690     GetBorders(out long from, out long to);
691     for (var i = from; i <= to; i++)
692     {
693         var word = _array[i];
694         if (word != 0)
695         {
696             AppendAllSetBitIndices(result, i, word);
697         }
698     }
699     return result;
700 }
701
702 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

703 public List<ulong> GetSetUInt64Indices()
704 {
705     var result = new List<ulong>();
706     GetBorders(out ulong from, out ulong to);
707     for (var i = from; i <= to; i++)
708     {
709         var word = _array[i];
710         if (word != 0)
711         {
712             AppendAllSetBitIndices(result, i, word);
713         }
714     }
715     return result;
716 }
717
718 [MethodImpl(MethodImplOptions.AggressiveInlining)]
719 public long GetFirstSetBitIndex()
720 {
721     var i = _minPositiveWord;
722     var word = _array[i];
723     if (word != 0)
724     {
725         return GetFirstSetBitForWord(i, word);
726     }
727     return -1;
728 }
729
730 [MethodImpl(MethodImplOptions.AggressiveInlining)]
731 public long GetLastSetBitIndex()
732 {
733     var i = _maxPositiveWord;
734     var word = _array[i];
735     if (word != 0)
736     {
737         return GetLastSetBitForWord(i, word);
738     }
739     return -1;
740 }
741
742 [MethodImpl(MethodImplOptions.AggressiveInlining)]
743 public long CountSetBits()
744 {
745     var total = 0L;
746     GetBorders(out long from, out long to);
747     for (var i = from; i <= to; i++)
748     {
749         var word = _array[i];
750         if (word != 0)
751         {
752             total += CountSetBitsForWord(word);
753         }
754     }
755     return total;
756 }
757
758 [MethodImpl(MethodImplOptions.AggressiveInlining)]
759 public bool HaveCommonBits(BitString other)
760 {
761     EnsureBitStringHasTheSameSize(other, nameof(other));
762     GetCommonInnerBorders(this, other, out long from, out long to);
763     var otherArray = other._array;
764     for (var i = from; i <= to; i++)
765     {
766         var left = _array[i];
767         var right = otherArray[i];
768         if (left != 0 && right != 0 && (left & right) != 0)
769         {
770             return true;
771         }
772     }
773     return false;
774 }
775
776 [MethodImpl(MethodImplOptions.AggressiveInlining)]
777 public long CountCommonBits(BitString other)
778 {
779     EnsureBitStringHasTheSameSize(other, nameof(other));
780     GetCommonInnerBorders(this, other, out long from, out long to);
781     var total = 0L;

```



```

782     var otherArray = other._array;
783     for (var i = from; i <= to; i++)
784     {
785         var left = _array[i];
786         var right = otherArray[i];
787         var combined = left & right;
788         if (combined != 0)
789         {
790             total += CountSetBitsForWord(combined);
791         }
792     }
793     return total;
794 }
795
796 [MethodImpl(MethodImplOptions.AggressiveInlining)]
797 public List<long> GetCommonIndices(BitString other)
798 {
799     EnsureBitStringHasTheSameSize(other, nameof(other));
800     GetCommonInnerBorders(this, other, out long from, out long to);
801     var result = new List<long>();
802     var otherArray = other._array;
803     for (var i = from; i <= to; i++)
804     {
805         var left = _array[i];
806         var right = otherArray[i];
807         var combined = left & right;
808         if (combined != 0)
809         {
810             AppendAllSetBitIndices(result, i, combined);
811         }
812     }
813     return result;
814 }
815
816 [MethodImpl(MethodImplOptions.AggressiveInlining)]
817 public List<ulong> GetCommonUInt64Indices(BitString other)
818 {
819     EnsureBitStringHasTheSameSize(other, nameof(other));
820     GetCommonBorders(this, other, out ulong from, out ulong to);
821     var result = new List<ulong>();
822     var otherArray = other._array;
823     for (var i = from; i <= to; i++)
824     {
825         var left = _array[i];
826         var right = otherArray[i];
827         var combined = left & right;
828         if (combined != 0)
829         {
830             AppendAllSetBitIndices(result, i, combined);
831         }
832     }
833     return result;
834 }
835
836 [MethodImpl(MethodImplOptions.AggressiveInlining)]
837 public long GetFirstCommonBitIndex(BitString other)
838 {
839     EnsureBitStringHasTheSameSize(other, nameof(other));
840     GetCommonInnerBorders(this, other, out long from, out long to);
841     var otherArray = other._array;
842     for (var i = from; i <= to; i++)
843     {
844         var left = _array[i];
845         var right = otherArray[i];
846         var combined = left & right;
847         if (combined != 0)
848         {
849             return GetFirstSetBitForWord(i, combined);
850         }
851     }
852     return -1;
853 }
854
855 [MethodImpl(MethodImplOptions.AggressiveInlining)]
856 public long GetLastCommonBitIndex(BitString other)
857 {
858     EnsureBitStringHasTheSameSize(other, nameof(other));
859     GetCommonInnerBorders(this, other, out long from, out long to);
860     var otherArray = other._array;

```

```

861     for (var i = to; i >= from; i--)
862     {
863         var left = _array[i];
864         var right = otherArray[i];
865         var combined = left & right;
866         if (combined != 0)
867         {
868             return GetLastSetBitForWord(i, combined);
869         }
870     }
871     return -1;
872 }
873
874 [MethodImpl(MethodImplOptions.AggressiveInlining)]
875 public override bool Equals(object obj) => obj is BitString @string ? Equals(@string) :
    ↪ false;
876
877 [MethodImpl(MethodImplOptions.AggressiveInlining)]
878 public bool Equals(BitString other)
879 {
880     if (_length != other._length)
881     {
882         return false;
883     }
884     var otherArray = other._array;
885     if (_array.Length != otherArray.Length)
886     {
887         return false;
888     }
889     if (_minPositiveWord != other._minPositiveWord)
890     {
891         return false;
892     }
893     if (_maxPositiveWord != other._maxPositiveWord)
894     {
895         return false;
896     }
897     GetCommonBorders(this, other, out ulong from, out ulong to);
898     for (var i = from; i <= to; i++)
899     {
900         if (_array[i] != otherArray[i])
901         {
902             return false;
903         }
904     }
905     return true;
906 }
907
908 [MethodImpl(MethodImplOptions.AggressiveInlining)]
909 private void EnsureBitStringHasTheSameSize(BitString other, string argumentName)
910 {
911     Ensure.Always.ArgumentNotNull(other, argumentName);
912     if (_length != other._length)
913     {
914         throw new ArgumentException("Bit string must be the same size.", argumentName);
915     }
916 }
917
918 [MethodImpl(MethodImplOptions.AggressiveInlining)]
919 private void MarkBordersAsAllBitsReset() => SetBorders(_array.LongLength - 1, 0);
920
921 [MethodImpl(MethodImplOptions.AggressiveInlining)]
922 private void MarkBordersAsAllBitsSet() => SetBorders(0, _array.LongLength - 1);
923
924 [MethodImpl(MethodImplOptions.AggressiveInlining)]
925 private void GetBorders(out long from, out long to)
926 {
927     from = _minPositiveWord;
928     to = _maxPositiveWord;
929 }
930
931 [MethodImpl(MethodImplOptions.AggressiveInlining)]
932 private void GetBorders(out ulong from, out ulong to)
933 {
934     from = (ulong)_minPositiveWord;
935     to = (ulong)_maxPositiveWord;
936 }
937
938 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

939 private void SetBorders(long from, long to)
940 {
941     _minPositiveWord = from;
942     _maxPositiveWord = to;
943 }
944
945 [MethodImpl(MethodImplOptions.AggressiveInlining)]
946 private Range<long> GetValidIndexRange() => (0, _length - 1);
947
948 [MethodImpl(MethodImplOptions.AggressiveInlining)]
949 private static Range<long> GetValidLengthRange() => (0, long.MaxValue);
950
951 [MethodImpl(MethodImplOptions.AggressiveInlining)]
952 private static void AppendAllSetBitIndices(List<ulong> result, ulong wordIndex, long
    ↳ wordValue)
953 {
954     GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
    ↳ bits32to47, out byte[] bits48to63);
955     AppendAllSetIndices(result, wordIndex, bits00to15, bits16to31, bits32to47,
    ↳ bits48to63);
956 }
957
958 [MethodImpl(MethodImplOptions.AggressiveInlining)]
959 private static void AppendAllSetBitIndices(List<long> result, long wordIndex, long
    ↳ wordValue)
960 {
961     GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
    ↳ bits32to47, out byte[] bits48to63);
962     AppendAllSetBitIndices(result, wordIndex, bits00to15, bits16to31, bits32to47,
    ↳ bits48to63);
963 }
964
965 [MethodImpl(MethodImplOptions.AggressiveInlining)]
966 private static long CountSetBitsForWord(long word)
967 {
968     GetBits(word, out byte[] bits00to15, out byte[] bits16to31, out byte[] bits32to47,
    ↳ out byte[] bits48to63);
969     return bits00to15.LongLength + bits16to31.LongLength + bits32to47.LongLength +
    ↳ bits48to63.LongLength;
970 }
971
972 [MethodImpl(MethodImplOptions.AggressiveInlining)]
973 private static long GetFirstSetBitForWord(long wordIndex, long wordValue)
974 {
975     GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
    ↳ bits32to47, out byte[] bits48to63);
976     return GetFirstSetBit(wordIndex, bits00to15, bits16to31, bits32to47, bits48to63);
977 }
978
979 [MethodImpl(MethodImplOptions.AggressiveInlining)]
980 private static long GetLastSetBitForWord(long wordIndex, long wordValue)
981 {
982     GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
    ↳ bits32to47, out byte[] bits48to63);
983     return GetLastSetBit(wordIndex, bits00to15, bits16to31, bits32to47, bits48to63);
984 }
985
986 [MethodImpl(MethodImplOptions.AggressiveInlining)]
987 private static void AppendAllSetBitIndices(List<long> result, long i, byte[] bits00to15,
    ↳ byte[] bits16to31, byte[] bits32to47, byte[] bits48to63)
988 {
989     for (var j = 0; j < bits00to15.Length; j++)
990     {
991         result.Add(bits00to15[j] + (i * 64));
992     }
993     for (var j = 0; j < bits16to31.Length; j++)
994     {
995         result.Add(bits16to31[j] + 16 + (i * 64));
996     }
997     for (var j = 0; j < bits32to47.Length; j++)
998     {
999         result.Add(bits32to47[j] + 32 + (i * 64));
1000     }
1001     for (var j = 0; j < bits48to63.Length; j++)
1002     {
1003         result.Add(bits48to63[j] + 48 + (i * 64));
1004     }
1005 }

```

```

1006 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1007 private static void AppendAllSetIndices(List<ulong> result, ulong i, byte[] bits00to15,
1008 ↪ byte[] bits16to31, byte[] bits32to47, byte[] bits48to63)
1009 {
1010     for (var j = 0; j < bits00to15.Length; j++)
1011     {
1012         result.Add(bits00to15[j] + (i * 64));
1013     }
1014     for (var j = 0; j < bits16to31.Length; j++)
1015     {
1016         result.Add(bits16to31[j] + 16UL + (i * 64));
1017     }
1018     for (var j = 0; j < bits32to47.Length; j++)
1019     {
1020         result.Add(bits32to47[j] + 32UL + (i * 64));
1021     }
1022     for (var j = 0; j < bits48to63.Length; j++)
1023     {
1024         result.Add(bits48to63[j] + 48UL + (i * 64));
1025     }
1026 }
1027
1028 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1029 private static long GetFirstSetBit(long i, byte[] bits00to15, byte[] bits16to31, byte[]
1030 ↪ bits32to47, byte[] bits48to63)
1031 {
1032     if (bits00to15.Length > 0)
1033     {
1034         return bits00to15[0] + (i * 64);
1035     }
1036     if (bits16to31.Length > 0)
1037     {
1038         return bits16to31[0] + 16 + (i * 64);
1039     }
1040     if (bits32to47.Length > 0)
1041     {
1042         return bits32to47[0] + 32 + (i * 64);
1043     }
1044     return bits48to63[0] + 48 + (i * 64);
1045 }
1046
1047 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1048 private static long GetLastSetBit(long i, byte[] bits00to15, byte[] bits16to31, byte[]
1049 ↪ bits32to47, byte[] bits48to63)
1050 {
1051     if (bits48to63.Length > 0)
1052     {
1053         return bits48to63[bits48to63.Length - 1] + 48 + (i * 64);
1054     }
1055     if (bits32to47.Length > 0)
1056     {
1057         return bits32to47[bits32to47.Length - 1] + 32 + (i * 64);
1058     }
1059     if (bits16to31.Length > 0)
1060     {
1061         return bits16to31[bits16to31.Length - 1] + 16 + (i * 64);
1062     }
1063     return bits00to15[bits00to15.Length - 1] + (i * 64);
1064 }
1065
1066 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1067 private static void GetBits(long word, out byte[] bits00to15, out byte[] bits16to31, out
1068 ↪ byte[] bits32to47, out byte[] bits48to63)
1069 {
1070     bits00to15 = _bitsSetIn16Bits[word & 0xfffffu];
1071     bits16to31 = _bitsSetIn16Bits[(word >> 16) & 0xfffffu];
1072     bits32to47 = _bitsSetIn16Bits[(word >> 32) & 0xfffffu];
1073     bits48to63 = _bitsSetIn16Bits[(word >> 48) & 0xfffffu];
1074 }
1075
1076 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1077 public static void GetCommonInnerBorders(BitString left, BitString right, out long from,
1078 ↪ out long to)
1079 {
1080     from = Math.Max(left._minPositiveWord, right._minPositiveWord);
1081     to = Math.Min(left._maxPositiveWord, right._maxPositiveWord);

```

```

1078     }
1079
1080     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1081     public static void GetCommonOuterBorders(BitString left, BitString right, out long from,
1082     ↪ out long to)
1083     {
1084         from = Math.Min(left._minPositiveWord, right._minPositiveWord);
1085         to = Math.Max(left._maxPositiveWord, right._maxPositiveWord);
1086     }
1087
1088     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1089     public static void GetCommonOuterBorders(BitString left, BitString right, out int from,
1090     ↪ out int to)
1091     {
1092         from = (int)Math.Min(left._minPositiveWord, right._minPositiveWord);
1093         to = (int)Math.Max(left._maxPositiveWord, right._maxPositiveWord);
1094     }
1095
1096     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1097     public static void GetCommonBorders(BitString left, BitString right, out ulong from, out
1098     ↪ ulong to)
1099     {
1100         from = (ulong)Math.Max(left._minPositiveWord, right._minPositiveWord);
1101         to = (ulong)Math.Min(left._maxPositiveWord, right._maxPositiveWord);
1102     }
1103
1104     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1105     public static long GetWordsCountFromIndex(long index) => (index + 63) / 64;
1106
1107     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1108     public static long GetWordIndexFromIndex(long index) => index >> 6;
1109
1110     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1111     public static long GetBitMaskFromIndex(long index) => 1L << (int)(index & 63);
1112
1113     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1114     public override int GetHashCode() => base.GetHashCode();
1115
1116     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1117     public override string ToString() => base.ToString();
1118 }
1119 }

```

1.9 ./csharp/Platform.Collections/BitStringExtensions.cs

```

1 using System.Runtime.CompilerServices;
2 using Platform.Random;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Collections
7 {
8     public static class BitStringExtensions
9     {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public static void SetRandomBits(this BitString @string)
12         {
13             for (var i = 0; i < @string.Length; i++)
14             {
15                 var value = RandomHelpers.Default.NextBoolean();
16                 @string.Set(i, value);
17             }
18         }
19     }
20 }

```

1.10 ./csharp/Platform.Collections/Concurrent/ConcurrentQueueExtensions.cs

```

1 using System.Collections.Concurrent;
2 using System.Collections.Generic;
3 using System.Runtime.CompilerServices;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Collections.Concurrent
8 {
9     public static class ConcurrentQueueExtensions
10     {
11         [MethodImpl(MethodImplOptions.AggressiveInlining)]
12         public static IEnumerable<T> DequeueAll<T>(this ConcurrentQueue<T> queue)
13         {

```

```

14         while (queue.TryDequeue(out T item))
15         {
16             yield return item;
17         }
18     }
19 }
20 }

```

1.11 ./csharp/Platform.Collections/Concurrent/ConcurrentStackExtensions.cs

```

1 using System.Collections.Concurrent;
2 using System.Runtime.CompilerServices;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Collections.Concurrent
7 {
8     public static class ConcurrentStackExtensions
9     {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public static T PopOrDefault<T>(this ConcurrentStack<T> stack) => stack.TryPop(out T
12             ↪ value) ? value : default;
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         public static T PeekOrDefault<T>(this ConcurrentStack<T> stack) => stack.TryPeek(out T
16             ↪ value) ? value : default;
17     }
18 }

```

1.12 ./csharp/Platform.Collections/EnsureExtensions.cs

```

1 using System;
2 using System.Collections.Generic;
3 using System.Diagnostics;
4 using System.Runtime.CompilerServices;
5 using Platform.Exceptions;
6 using Platform.Exceptions.ExtensionRoots;
7
8 #pragma warning disable IDE0060 // Remove unused parameter
9 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11 namespace Platform.Collections
12 {
13     public static class EnsureExtensions
14     {
15         #region Always
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public static void ArgumentNotEmpty<T>(this EnsureAlwaysExtensionRoot root,
19             ↪ ICollection<T> argument, string argumentName, string message)
20         {
21             if (argument.IsNullOrEmpty())
22             {
23                 throw new ArgumentException(message, argumentName);
24             }
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public static void ArgumentNotEmpty<T>(this EnsureAlwaysExtensionRoot root,
28             ↪ ICollection<T> argument, string argumentName) => ArgumentNotEmpty(root, argument,
29             ↪ argumentName, null);
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         public static void ArgumentNotEmpty<T>(this EnsureAlwaysExtensionRoot root,
33             ↪ ICollection<T> argument) => ArgumentNotEmpty(root, argument, null, null);
34
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureAlwaysExtensionRoot root,
37             ↪ string argument, string argumentName, string message)
38         {
39             if (string.IsNullOrEmpty(argument))
40             {
41                 throw new ArgumentException(message, argumentName);
42             }
43
44         [MethodImpl(MethodImplOptions.AggressiveInlining)]
45         public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureAlwaysExtensionRoot root,
46             ↪ string argument, string argumentName) => ArgumentNotEmptyAndNotWhiteSpace(root,
47             ↪ argument, argumentName, null);
48     }
49 }

```

```

44     [MethodImpl(MethodImplOptions.AggressiveInlining)]
45     public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureAlwaysExtensionRoot root,
46         ↪ string argument) => ArgumentNotEmptyAndNotWhiteSpace(root, argument, null, null);
47
48     #endregion
49
50     #region OnDebug
51
52     [Conditional("DEBUG")]
53     public static void ArgumentNotEmpty<T>(this EnsureOnDebugExtensionRoot root,
54         ↪ ICollection<T> argument, string argumentName, string message) =>
55         ↪ Ensure.Always.ArgumentNotEmpty(argument, argumentName, message);
56
57     [Conditional("DEBUG")]
58     public static void ArgumentNotEmpty<T>(this EnsureOnDebugExtensionRoot root,
59         ↪ ICollection<T> argument, string argumentName) =>
60         ↪ Ensure.Always.ArgumentNotEmpty(argument, argumentName, null);
61
62     [Conditional("DEBUG")]
63     public static void ArgumentNotEmpty<T>(this EnsureOnDebugExtensionRoot root,
64         ↪ ICollection<T> argument) => Ensure.Always.ArgumentNotEmpty(argument, null, null);
65
66     [Conditional("DEBUG")]
67     public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureOnDebugExtensionRoot
68         ↪ root, string argument, string argumentName, string message) =>
69         ↪ Ensure.Always.ArgumentNotEmptyAndNotWhiteSpace(argument, argumentName, message);
70
71     [Conditional("DEBUG")]
72     public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureOnDebugExtensionRoot
73         ↪ root, string argument, string argumentName) =>
74         ↪ Ensure.Always.ArgumentNotEmptyAndNotWhiteSpace(argument, argumentName, null);
75
76     [Conditional("DEBUG")]
77     public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureOnDebugExtensionRoot
78         ↪ root, string argument) => Ensure.Always.ArgumentNotEmptyAndNotWhiteSpace(argument,
79         ↪ null, null);
80
81     #endregion
82 }
83
84 }
```

1.13 ./csharp/Platform.Collections/ICollectionExtensions.cs

```

1  using System.Collections.Generic;
2  using System.Linq;
3  using System.Runtime.CompilerServices;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Collections
8  {
9      public static class ICollectionExtensions
10     {
11         [MethodImpl(MethodImplOptions.AggressiveInlining)]
12         public static bool IsNullOrEmpty<T>(this ICollection<T> collection) => collection ==
13             ↪ null || collection.Count == 0;
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         public static bool AllEqualToDefault<T>(this ICollection<T> collection)
17         {
18             var equalityComparer = EqualityComparer<T>.Default;
19             return collection.All(item => equalityComparer.Equals(item, default));
20         }
21     }
22 }
```

1.14 ./csharp/Platform.Collections/IDictionaryExtensions.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Collections
8  {
9      public static class IDictionaryExtensions
10     {
11         [MethodImpl(MethodImplOptions.AggressiveInlining)]
12         public static TValue GetOrDefault<TKey, TValue>(this IDictionary<TKey, TValue>
13             ↪ dictionary, TKey key)
```

```

13     {
14         dictionary.TryGetValue(key, out TValue value);
15         return value;
16     }
17
18     [MethodImpl(MethodImplOptions.AggressiveInlining)]
19     public static TValue GetOrAdd<TKey, TValue>(this IDictionary<TKey, TValue> dictionary,
20     ↪ TKey key, Func<TKey, TValue> valueFactory)
21     {
22         if (!dictionary.TryGetValue(key, out TValue value))
23         {
24             value = valueFactory(key);
25             dictionary.Add(key, value);
26             return value;
27         }
28         return value;
29     }
30 }

```

1.15 ./csharp/Platform.Collections/Lists/CharListExtensions.cs

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  namespace Platform.Collections.Lists
5  {
6      public static class CharListExtensions
7      {
8          /// <summary>
9          /// <para>Generates a hash code for the entire list based on the values of its
10         ↪ elements.</para>
11         /// <para>Генерирует хэш-код всего списка, на основе значений его элементов.</para>
12         /// </summary>
13         /// <param name="list"><para>The list to be hashed.</para><para>Список для
14         ↪ хеширования.</para></param>
15         /// <returns>
16         /// <para>The hash code of the list.</para>
17         /// <para>Хэш-код списка.</para>
18         /// </returns>
19         /// <remarks>
20         /// Based on https://github.com/Microsoft/referencesource/blob/3b1eaf5203992df69de44c783\_
21         ↪ a3eda37d3d4cd10/mscorlib/system/string.cs#L833
22         /// </remarks>
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         public static int GenerateHashCode(this IList<char> list)
25         {
26             var hashSeed = 5381;
27             var hashAccumulator = hashSeed;
28             for (var i = 0; i < list.Count; i++)
29             {
30                 hashAccumulator = (hashAccumulator << 5) + hashAccumulator ^ list[i];
31             }
32             return hashAccumulator + (hashSeed * 1566083941);
33         }
34
35         /// <summary>
36         /// <para>Compares two lists for equality.</para>
37         /// <para>Сравнивает два списка на равенство.</para>
38         /// </summary>
39         /// <param name="left"><para>The first compared list.</para><para>Первый список для
40         ↪ сравнения.</para></param>
41         /// <param name="right"><para>The second compared list.</para><para>Второй список для
42         ↪ сравнения.</para></param>
43         /// <returns>
44         /// <para>True, if the passed lists are equal to each other otherwise false.</para>
45         /// <para>True, если переданные списки равны друг другу, иначе false.</para>
46         /// </returns>
47         [MethodImpl(MethodImplOptions.AggressiveInlining)]
48         public static bool EqualTo(this IList<char> left, IList<char> right) =>
49         ↪ left.EqualTo(right, ContentEqualTo);
50
51         /// <summary>
52         /// <para>Compares each element in the list for equality.</para>
53         /// <para>Сравнивает на равенство каждый элемент списка.</para>
54         /// </summary>
55         /// <param name="left"><para>The first compared list.</para><para>Первый список для
56         ↪ сравнения.</para></param>

```



```

50    /// <param name="right"><para>The second compared list.</para><para>Второй список для
    → сравнения.</para></param>
51    /// <returns>
52    /// <para>If at least one element of one list is not equal to the corresponding element
    → from another list returns false, otherwise - true.</para>
53    /// <para>Если как минимум один элемент одного списка не равен соответствующему элементу
    → из другого списка возвращает false, иначе - true.</para>
54    /// </returns>
55    [MethodImpl(MethodImplOptions.AggressiveInlining)]
56    public static bool ContentEqualTo(this IList<char> left, IList<char> right)
57    {
58        for (var i = left.Count - 1; i >= 0; --i)
59        {
60            if (left[i] != right[i])
61            {
62                return false;
63            }
64        }
65        return true;
66    }
67 }
68 }

```

1.16 ./csharp/Platform.Collections/Lists/IListComparer.cs

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  namespace Platform.Collections.Lists
5  {
6      public class IListComparer<T> : IComparer<IList<T>>
7      {
8          /// <summary>
9          /// <para>Compares two lists.</para>
10         /// <para>Сравнивает два списка.</para>
11         /// </summary>
12         /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
            → списка.</para></typeparam>
13         /// <param name="left"><para>The first compared list.</para><para>Первый список для
            → сравнения.</para></param>
14         /// <param name="right"><para>The second compared list.</para><para>Второй список для
            → сравнения.</para></param>
15         /// <returns>
16         /// <para>
17         ///     A signed integer that indicates the relative values of <paramref name="left" />
            → and <paramref name="right" /> lists' elements, as shown in the following table.
18         ///     <list type="table">
19         ///         <listheader>
20         ///             <term>Value</term>
21         ///             <description>Meaning</description>
22         ///         </listheader>
23         ///         <item>
24         ///             <term>Is less than zero</term>
25         ///             <description>First non equal element of <paramref name="left" /> list is
            → less than first not equal element of <paramref name="right" /> list.</description>
26         ///         </item>
27         ///         <item>
28         ///             <term>Zero</term>
29         ///             <description>All elements of <paramref name="left" /> list equals to all
            → elements of <paramref name="right" /> list.</description>
30         ///         </item>
31         ///         <item>
32         ///             <term>Is greater than zero</term>
33         ///             <description>First non equal element of <paramref name="left" /> list is
            → greater than first not equal element of <paramref name="right" /> list.</description>
34         ///         </item>
35         ///     </list>
36         /// <para>
37         /// <para>
38         ///     Целое число со знаком, которое указывает относительные значения элементов
            → списков <paramref name="left" /> и <paramref name="right" /> как показано в
            → следующей таблице.
39         ///     <list type="table">
40         ///         <listheader>
41         ///             <term>Значение</term>
42         ///             <description>Смысл</description>
43         ///         </listheader>
44         ///         <item>

```

```

45     /// <term>Меньше нуля</term>
46     /// <description>Первый не равный элемент <paramref name="left" /> списка
    → меньше первого неравного элемента <paramref name="right" /> списка.</description>
47     /// </item>
48     /// <item>
49     /// <term>Ноль</term>
50     /// <description>Все элементы <paramref name="left" /> списка равны всем
    → элементам <paramref name="right" /> списка.</description>
51     /// </item>
52     /// <item>
53     /// <term>Больше нуля</term>
54     /// <description>Первый не равный элемент <paramref name="left" /> списка
    → больше первого неравного элемента <paramref name="right" /> списка.</description>
55     /// </item>
56     /// </list>
57     /// </para>
58     /// </returns>
59     [MethodImpl(MethodImplOptions.AggressiveInlining)]
60     public int Compare(IList<T> left, IList<T> right) => left.CompareTo(right);
61 }
62 }

```

1.17 ./csharp/Platform.Collections/Lists/IListEqualityComparer.cs

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  namespace Platform.Collections.Lists
5  {
6      public class IListEqualityComparer<T> : IEqualityComparer<IList<T>>
7      {
8          /// <summary>
9          /// <para>Compares two lists for equality.</para>
10         /// <para>Сравнивает два списка на равенство.</para>
11         /// </summary>
12         /// <param name="left"><para>The first compared list.</para><para>Первый список для
            → сравнения.</para></param>
13         /// <param name="right"><para>The second compared list.</para><para>Второй список для
            → сравнения.</para></param>
14         /// <returns>
15         /// <para>If the passed lists are equal to each other, true is returned, otherwise
            → false.</para>
16         /// <para>Если переданные списки равны друг другу, возвращается true, иначе же
            → false.</para>
17         /// </returns>
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         public bool Equals(IList<T> left, IList<T> right) => left.EqualTo(right);
20
21         /// <summary>
22         /// <para>Generates a hash code for the entire list based on the values of its
            → elements.</para>
23         /// <para>Генерирует хэш-код всего списка, на основе значений его элементов.</para>
24         /// </summary>
25         /// <param name="list"><para>Hash list.</para><para>Список для
            → хеширования.</para></param>
26         /// <returns>
27         /// <para>The hash code of the list.</para>
28         /// <para>Хэш-код списка.</para>
29         /// </returns>
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         public int GetHashCode(IList<T> list) => list.GenerateHashCode();
32     }
33 }

```

1.18 ./csharp/Platform.Collections/Lists/IListExtensions.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4
5  namespace Platform.Collections.Lists
6  {
7      public static class IListExtensions
8      {
9          /// <summary>
10         /// <para>Gets the element from specified index if the list is not null and the index is
            → within the list's boundaries, otherwise it returns default value of type T.</para>
11         /// <para>Получает элемент из указанного индекса, если список не является null и индекс
            → находится в границах списка, в противном случае он возвращает значение по умолчанию
            → типа T.</para>

```

```

12  /// </summary>
13  /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
    ↳ списка.</typeparam>
14  /// <param name="list"><para>The checked list.</para><para>Проверяемый
    ↳ список.</param>
15  /// <param name="index"><para>The index of element.</para><para>Индекс
    ↳ элемента.</param>
16  /// <returns>
17  /// <para>If the specified index is within list's boundaries, then - list[index],
    ↳ otherwise the default value.</para>
18  /// <para>Если указанный индекс находится в пределах границ списка, тогда - list[index],
    ↳ иначе же значение по умолчанию.</para>
19  /// </returns>
20  [MethodImpl(MethodImplOptions.AggressiveInlining)]
21  public static T GetElementOrDefault<T>(this IList<T> list, int index) => list != null &&
    ↳ list.Count > index ? list[index] : default;
22
23  /// <summary>
24  /// <para>Checks if a list is passed, checks its length, and if successful, copies the
    ↳ value of list [index] into the element variable. Otherwise, the element variable has
    ↳ a default value.</para>
25  /// <para>Проверяет, передан ли список, сверяет его длину и в случае успеха копирует
    ↳ значение list[index] в переменную element. Иначе переменная element имеет значение
    ↳ по умолчанию.</para>
26  /// </summary>
27  /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
    ↳ списка.</typeparam>
28  /// <param name="list"><para>The checked list.</para><para>Список для
    ↳ проверки.</param>
29  /// <param name="index"><para>The index of element.</para><para>Индекс
    ↳ элемента.</param>
30  /// <param name="element"><para>Variable for passing the index
    ↳ value.</para><para>Переменная для передачи значения индекса.</param>
31  /// <returns>
32  /// <para>True on success, false otherwise.</para>
33  /// <para>True в случае успеха, иначе false.</para>
34  /// </returns>
35  [MethodImpl(MethodImplOptions.AggressiveInlining)]
36  public static bool TryGetElement<T>(this IList<T> list, int index, out T element)
37  {
38      if (list != null && list.Count > index)
39      {
40          element = list[index];
41          return true;
42      }
43      else
44      {
45          element = default;
46          return false;
47      }
48  }
49
50  /// <summary>
51  /// <para>Adds a value to the list.</para>
52  /// <para>Добавляет значение в список.</para>
53  /// </summary>
54  /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
    ↳ списка.</typeparam>
55  /// <param name="list"><para>The list to add the value to.</para><para>Список в который
    ↳ нужно добавить значение.</param>
56  /// <param name="element"><para>The item to add to the list.</para><para>Элемент который
    ↳ нужно добавить в список.</param>
57  /// <returns>
58  /// <para>True value in any case.</para>
59  /// <para>Значение true в любом случае.</para>
60  /// </returns>
61  [MethodImpl(MethodImplOptions.AggressiveInlining)]
62  public static bool AddAndReturnTrue<T>(this IList<T> list, T element)
63  {
64      list.Add(element);
65      return true;
66  }
67
68  /// <summary>
69  /// <para>Adds the value with first index from other list to this list.</para>
70  /// <para>Добавляет в этот список значение с первым индексом из другого списка.</para>
71  /// </summary>

```

```

72  /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
    → списка.</para></typeparam>
73  /// <param name="list"><para>The list to add the value to.</para><para>Список в который
    → нужно добавить значение.</para></param>
74  /// <param name="elements"><para>The item to add to the list.</para><para>Элемент
    → который нужно добавить в список</para></param>
75  /// <returns>
76  /// <para>True value in any case.</para>
77  /// <para>Значение true в любом случае.</para>
78  /// </returns>
79  [MethodImpl(MethodImplOptions.AggressiveInlining)]
80  public static bool AddFirstAndReturnTrue<T>(this IList<T> list, IList<T> elements)
81  {
82      list.AddFirst(elements);
83      return true;
84  }
85
86  /// <summary>
87  /// <para>Adds a value to the list at the first index.</para>
88  /// <para>Добавляет значение в список по первому индексу.</para>
89  /// </summary>
90  /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
    → списка.</para></typeparam>
91  /// <param name="list"><para>The list to add the value to.</para><para>Список в который
    → нужно добавить значение.</para></param>
92  /// <param name="elements"><para>The item to add to the list.</para><para>Элемент
    → который нужно добавить в список</para></param>
93  [MethodImpl(MethodImplOptions.AggressiveInlining)]
94  public static void AddFirst<T>(this IList<T> list, IList<T> elements) =>
    → list.Add(elements[0]);
95
96  /// <summary>
97  /// <para>Adds all elements from other list to this list and returns true.</para>
98  /// <para>Добавляет все элементы из другого списка в этот список и возвращает
    → true.</para>
99  /// </summary>
100  /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
    → списка.</para></typeparam>
101  /// <param name="list"><para>The list to add the values to.</para><para>Список в который
    → нужно добавить значения.</para></param>
102  /// <param name="elements"><para>List of values to add.</para><para>Список значений
    → которые необходимо добавить.</para></param>
103  /// <returns>
104  /// <para>True value in any case.</para>
105  /// <para>Значение true в любом случае.</para>
106  /// </returns>
107  [MethodImpl(MethodImplOptions.AggressiveInlining)]
108  public static bool AddAllAndReturnTrue<T>(this IList<T> list, IList<T> elements)
109  {
110      list.AddAll(elements);
111      return true;
112  }
113
114  /// <summary>
115  /// <para>Adds all elements from other list to this list.</para>
116  /// <para>Добавляет все элементы из другого списка в этот список.</para>
117  /// </summary>
118  /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
    → списка.</para></typeparam>
119  /// <param name="list"><para>The list to add the values to.</para><para>Список в который
    → нужно добавить значения.</para></param>
120  /// <param name="elements"><para>The list of values to add.</para><para>Список значений
    → которые необходимо добавить.</para></param>
121  [MethodImpl(MethodImplOptions.AggressiveInlining)]
122  public static void AddAll<T>(this IList<T> list, IList<T> elements)
123  {
124      for (var i = 0; i < elements.Count; i++)
125      {
126          list.Add(elements[i]);
127      }
128  }
129
130  /// <summary>
131  /// <para>Adds values to the list skipping the first element.</para>
132  /// <para>Добавляет значения в список пропуская первый элемент.</para>
133  /// </summary>

```

```

134  /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
    → списка.</para></typeparam>
135  /// <param name="list"><para>The list to add the values to.</para><para>Список в который
    → нужно добавить значения.</para></param>
136  /// <param name="elements"><para>The list of values to add.</para><para>Список значений
    → которые необходимо добавить.</para></param>
137  /// <returns>
138  /// <para>True value in any case.</para>
139  /// <para>Значение true в любом случае.</para>
140  /// </returns>
141  [MethodImpl(MethodImplOptions.AggressiveInlining)]
142  public static bool AddSkipFirstAndReturnTrue<T>(this IList<T> list, IList<T> elements)
143  {
144      list.AddSkipFirst(elements);
145      return true;
146  }
147
148  /// <summary>
149  /// <para>Adds values to the list skipping the first element.</para>
150  /// <para>Добавляет значения в список пропуская первый элемент.</para>
151  /// </summary>
152  /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
    → списка.</para></typeparam>
153  /// <param name="list"><para>The list to add the values to.</para><para>Список в который
    → нужно добавить значения.</para></param>
154  /// <param name="elements"><para>List of values to add.</para><para>Список значений
    → которые необходимо добавить.</para></param>
155  [MethodImpl(MethodImplOptions.AggressiveInlining)]
156  public static void AddSkipFirst<T>(this IList<T> list, IList<T> elements) =>
    → list.AddSkipFirst(elements, 1);
157
158  /// <summary>
159  /// <para>Adds values to the list skipping a specified number of first elements.</para>
160  /// <para>Добавляет в список значения пропуская определенное количество первых
    → элементов.</para>
161  /// </summary>
162  /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
    → списка.</para></typeparam>
163  /// <param name="list"><para>The list to add the values to.</para><para>Список в который
    → нужно добавить значения.</para></param>
164  /// <param name="elements"><para>List of values to add.</para><para>Список значений
    → которые необходимо добавить.</para></param>
165  /// <param name="skip"><para>Number of elements to skip.</para><para>Количество
    → пропускаемых элементов.</para></param>
166  [MethodImpl(MethodImplOptions.AggressiveInlining)]
167  public static void AddSkipFirst<T>(this IList<T> list, IList<T> elements, int skip)
168  {
169      for (var i = skip; i < elements.Count; i++)
170      {
171          list.Add(elements[i]);
172      }
173  }
174
175  /// <summary>
176  /// <para>Reads the number of elements in the list.</para>
177  /// <para>Считывает число элементов списка.</para>
178  /// </summary>
179  /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
    → списка.</para></typeparam>
180  /// <param name="list"><para>The checked list.</para><para>Список для
    → проверки.</para></param>
181  /// <returns>
182  /// <para>The number of items contained in the list or 0.</para>
183  /// <para>Число элементов содержащихся в списке или же 0.</para>
184  /// </returns>
185  [MethodImpl(MethodImplOptions.AggressiveInlining)]
186  public static int GetCountOrZero<T>(this IList<T> list) => list?.Count ?? 0;
187
188  /// <summary>
189  /// <para>Compares two lists for equality.</para>
190  /// <para>Сравнивает два списка на равенство.</para>
191  /// </summary>
192  /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
    → списка.</para></typeparam>
193  /// <param name="left"><para>The first compared list.</para><para>Первый список для
    → сравнения.</para></param>

```

```

194  /// <param name="right"><para>The second compared list.</para><para>Второй список для
    → сравнения.</para></param>
195  /// <returns>
196  /// <para>If the passed lists are equal to each other, true is returned, otherwise
    → false.</para>
197  /// <para>Если переданные списки равны друг другу, возвращается true, иначе же
    → false.</para>
198  /// </returns>
199  [MethodImpl(MethodImplOptions.AggressiveInlining)]
200  public static bool EqualTo<T>(this IList<T> left, IList<T> right) => EqualTo(left,
    → right, ContentEqualTo);
201
202  /// <summary>
203  /// <para>Compares two lists for equality.</para>
204  /// <para>Сравнивает два списка на равенство.</para>
205  /// </summary>
206  /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
    → списка.</para></typeparam>
207  /// <param name="left"><para>The first compared list.</para><para>Первый список для
    → проверки.</para></param>
208  /// <param name="right"><para>The second compared list.</para><para>Второй список для
    → сравнения.</para></param>
209  /// <param name="contentEqualityComparer"><para>Function to test two lists for their
    → content equality.</para><para>Функция для проверки двух списков на равенство их
    → содержимого.</para></param>
210  /// <returns>
211  /// <para>If the passed lists are equal to each other, true is returned, otherwise
    → false.</para>
212  /// <para>Если переданные списки равны друг другу, возвращается true, иначе же
    → false.</para>
213  /// </returns>
214  [MethodImpl(MethodImplOptions.AggressiveInlining)]
215  public static bool EqualTo<T>(this IList<T> left, IList<T> right, Func<IList<T>,
    → IList<T>, bool> contentEqualityComparer)
216  {
217      if (ReferenceEquals(left, right))
218      {
219          return true;
220      }
221      var leftCount = left.GetCountOrZero();
222      var rightCount = right.GetCountOrZero();
223      if (leftCount == 0 && rightCount == 0)
224      {
225          return true;
226      }
227      if (leftCount == 0 || rightCount == 0 || leftCount != rightCount)
228      {
229          return false;
230      }
231      return contentEqualityComparer(left, right);
232  }
233
234  /// <summary>
235  /// <para>Compares each element in the list for identity.</para>
236  /// <para>Сравнивает на равенство каждый элемент списка.</para>
237  /// </summary>
238  /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
    → списка.</para></typeparam>
239  /// <param name="left"><para>The first compared list.</para><para>Первый список для
    → сравнения.</para></param>
240  /// <param name="right"><para>The second compared list.</para><para>Второй список для
    → сравнения.</para></param>
241  /// <returns>
242  /// <para>If at least one element of one list is not equal to the corresponding element
    → from another list returns false, otherwise - true.</para>
243  /// <para>Если как минимум один элемент одного списка не равен соответствующему элементу
    → из другого списка возвращает false, иначе - true.</para>
244  /// </returns>
245  [MethodImpl(MethodImplOptions.AggressiveInlining)]
246  public static bool ContentEqualTo<T>(this IList<T> left, IList<T> right)
247  {
248      var equalityComparer = EqualityComparer<T>.Default;
249      for (var i = left.Count - 1; i >= 0; --i)
250      {
251          if (!equalityComparer.Equals(left[i], right[i]))
252          {
253              return false;

```

```

254     }
255 }
256 return true;
257 }
258
259 /// <summary>
260 /// <para>Creates an array by copying all elements from the list that satisfy the
    → predicate. If no list is passed, null is returned.</para>
261 /// <para>Создаёт массив, копируя из списка все элементы которые удовлетворяют
    → предикату. Если список не передан, возвращается null.</para>
262 /// </summary>
263 /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
    → списка.</para></typeparam>
264 /// <param name="list"><para>The list to copy from.</para><para>Список для копирования.</para></param>
265 /// <param name="predicate"><para>A function that determines whether an element should
    → be copied.</para><para>Функция определяющая должен ли копироваться
    → элемент.</para></param>
266 /// <returns>
267 /// <para>An array with copied elements from the list.</para>
268 /// <para>Массив с скопированными элементами из списка.</para>
269 /// </returns>
270 [MethodImpl(MethodImplOptions.AggressiveInlining)]
271 public static T[] ToArray<T>(this IList<T> list, Func<T, bool> predicate)
272 {
273     if (list == null)
274     {
275         return null;
276     }
277     var result = new List<T>(list.Count);
278     for (var i = 0; i < list.Count; i++)
279     {
280         if (predicate(list[i]))
281         {
282             result.Add(list[i]);
283         }
284     }
285     return result.ToArray();
286 }
287
288 /// <summary>
289 /// <para>Copies all the elements of the list into an array and returns it.</para>
290 /// <para>Копирует все элементы списка в массив и возвращает его.</para>
291 /// </summary>
292 /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
    → списка.</para></typeparam>
293 /// <param name="list"><para>The list to copy from.</para><para>Список для
    → копирования.</para></param>
294 /// <returns>
295 /// <para>An array with all the elements of the passed list.</para>
296 /// <para>Массив со всеми элементами переданного списка.</para>
297 /// </returns>
298 [MethodImpl(MethodImplOptions.AggressiveInlining)]
299 public static T[] ToArray<T>(this IList<T> list)
300 {
301     var array = new T[list.Count];
302     list.CopyTo(array, 0);
303     return array;
304 }
305
306 /// <summary>
307 /// <para>Executes the passed action for each item in the list.</para>
308 /// <para>Выполняет переданное действие для каждого элемента в списке.</para>
309 /// </summary>
310 /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
    → списка.</para></typeparam>
311 /// <param name="list"><para>The list of elements for which the action will be
    → executed.</para><para>Список элементов для которых будет выполняться
    → действие.</para></param>
312 /// <param name="action"><para>A function that will be called for each element of the
    → list.</para><para>Функция которая будет вызываться для каждого элемента
    → списка.</para></param>
313 [MethodImpl(MethodImplOptions.AggressiveInlining)]
314 public static void ForEach<T>(this IList<T> list, Action<T> action)
315 {
316     for (var i = 0; i < list.Count; i++)
317     {
318         action(list[i]);
319     }
320 }

```

```

319     }
320 }
321
322 /// <summary>
323 /// <para>Generates a hash code for the entire list based on the values of its
    → elements.</para>
324 /// <para>Генерирует хэш-код всего списка, на основе значений его элементов.</para>
325 /// </summary>
326 /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
    → списка.</para></typeparam>
327 /// <param name="list"><para>Hash list.</para><para>Список для
    → хеширования.</para></param>
328 /// <returns>
329 /// <para>The hash code of the list.</para>
330 /// <para>Хэш-код списка.</para>
331 /// </returns>
332 /// <remarks>
333 /// Based on http://stackoverflow.com/questions/263400/what-is-the-best-algorithm-for-an
    → -overridden-system-object-gethashcode
334 /// </remarks>
335 [MethodImpl(MethodImplOptions.AggressiveInlining)]
336 public static int GenerateHashCode<T>(this IList<T> list)
337 {
338     var hashAccumulator = 17;
339     for (var i = 0; i < list.Count; i++)
340     {
341         hashAccumulator = unchecked((hashAccumulator * 23) + list[i].GetHashCode());
342     }
343     return hashAccumulator;
344 }
345
346 /// <summary>
347 /// <para>Compares two lists.</para>
348 /// <para>Сравнивает два списка.</para>
349 /// </summary>
350 /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
    → списка.</para></typeparam>
351 /// <param name="left"><para>The first compared list.</para><para>Первый список для
    → сравнения.</para></param>
352 /// <param name="right"><para>The second compared list.</para><para>Второй список для
    → сравнения.</para></param>
353 /// <returns>
354 /// <para>
355 ///     A signed integer that indicates the relative values of <paramref name="left" />
    → and <paramref name="right" /> lists' elements, as shown in the following table.
356 ///     <list type="table">
357 ///         <listheader>
358 ///             <term>Value</term>
359 ///             <description>Meaning</description>
360 ///         </listheader>
361 ///         <item>
362 ///             <term>Is less than zero</term>
363 ///             <description>First non equal element of <paramref name="left" /> list is
    → less than first not equal element of <paramref name="right" /> list.</description>
364 ///         </item>
365 ///         <item>
366 ///             <term>Zero</term>
367 ///             <description>All elements of <paramref name="left" /> list equals to all
    → elements of <paramref name="right" /> list.</description>
368 ///         </item>
369 ///         <item>
370 ///             <term>Is greater than zero</term>
371 ///             <description>First non equal element of <paramref name="left" /> list is
    → greater than first not equal element of <paramref name="right" /> list.</description>
372 ///         </item>
373 ///     </list>
374 /// </para>
375 /// <para>
376 ///     Целое число со знаком, которое указывает относительные значения элементов
    → списков <paramref name="left" /> и <paramref name="right" /> как показано в
    → следующей таблице.
377 ///     <list type="table">
378 ///         <listheader>
379 ///             <term>Значение</term>
380 ///             <description>Смысл</description>
381 ///         </listheader>
382 ///         <item>

```



```

383     /// <term>Меньше нуля</term>
384     /// <description>Первый не равный элемент <paramref name="left" /> списка
    → меньше первого неравного элемента <paramref name="right" /> списка.</description>
385     /// </item>
386     /// <item>
387     /// <term>Ноль</term>
388     /// <description>Все элементы <paramref name="left" /> списка равны всем
    → элементам <paramref name="right" /> списка.</description>
389     /// </item>
390     /// <item>
391     /// <term>Больше нуля</term>
392     /// <description>Первый не равный элемент <paramref name="left" /> списка
    → больше первого неравного элемента <paramref name="right" /> списка.</description>
393     /// </item>
394     /// </list>
395     /// </para>
396     /// </returns>
397     [MethodImpl(MethodImplOptions.AggressiveInlining)]
398     public static int CompareTo<T>(this IList<T> left, IList<T> right)
399     {
400         var comparer = Comparer<T>.Default;
401         var leftCount = left.GetCountOrZero();
402         var rightCount = right.GetCountOrZero();
403         var intermediateResult = leftCount.CompareTo(rightCount);
404         for (var i = 0; intermediateResult == 0 && i < leftCount; i++)
405         {
406             intermediateResult = comparer.Compare(left[i], right[i]);
407         }
408         return intermediateResult;
409     }
410
411     /// <summary>
412     /// <para>Skips one element in the list and builds an array from the remaining
    → elements.</para>
413     /// <para>Пропускает один элемент списка и составляет из оставшихся элементов
    → массив.</para>
414     /// </summary>
415     /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
    → списка.</para></typeparam>
416     /// <param name="list"><para>The list to copy from.</para><para>Список для
    → копирования.</para></param>
417     /// <returns>
418     /// <para>If the list is empty, returns an empty array, otherwise - an array with a
    → missing first element.</para>
419     /// <para>Если список пуст, возвращает пустой массив, иначе - массив с пропущенным
    → первым элементом.</para>
420     /// </returns>
421     [MethodImpl(MethodImplOptions.AggressiveInlining)]
422     public static T[] SkipFirst<T>(this IList<T> list) => list.SkipFirst(1);
423
424     /// <summary>
425     /// <para>Skips the specified number of elements in the list and builds an array from
    → the remaining elements.</para>
426     /// <para>Пропускает указанное количество элементов списка и составляет из оставшихся
    → элементов массив.</para>
427     /// </summary>
428     /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
    → списка.</para></typeparam>
429     /// <param name="list"><para>The list to copy from.</para><para>Список для
    → копирования.</para></param>
430     /// <param name="skip"><para>The number of items to skip.</para><para>Количество
    → пропускаемых элементов.</para></param>
431     /// <returns>
432     /// <para>If the list is empty, or the number of skipped elements is greater than the
    → list, returns an empty array, otherwise - an array with the specified number of
    → missing elements.</para>
433     /// <para>Если список пуст, или количество пропускаемых элементов больше списка -
    → возвращает пустой массив, иначе - массив с указанным количеством пропущенных
    → элементов.</para>
434     /// </returns>
435     [MethodImpl(MethodImplOptions.AggressiveInlining)]
436     public static T[] SkipFirst<T>(this IList<T> list, int skip)
437     {
438         if (list.IsNullOrEmpty() || list.Count <= skip)
439         {
440             return Array.Empty<T>();

```

```

441     }
442     var result = new T[list.Count - skip];
443     for (int r = skip, w = 0; r < list.Count; r++, w++)
444     {
445         result[w] = list[r];
446     }
447     return result;
448 }
449
450 /// <summary>
451 /// <para>Shifts all the elements of the list by one position to the right.</para>
452 /// <para>Сдвигает вправо все элементы списка на одну позицию.</para>
453 /// </summary>
454 /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
455   → списка.</para></typeparam>
456 /// <param name="list"><para>The list to copy from.</para><para>Список для
457   → копирования.</para></param>
458 /// <returns>
459 /// <para>Array with a shift of elements by one position.</para>
460 /// <para>Массив со сдвигом элементов на одну позицию.</para>
461 /// </returns>
462 [MethodImpl(MethodImplOptions.AggressiveInlining)]
463 public static IList<T> ShiftRight<T>(this IList<T> list) => list.ShiftRight(1);
464
465 /// <summary>
466 /// <para>Shifts all elements of the list to the right by the specified number of
467   → elements.</para>
468 /// <para>Сдвигает вправо все элементы списка на указанное количество элементов.</para>
469 /// </summary>
470 /// <typeparam name="T"><para>The list's item type.</para><para>Тип элементов
471   → списка.</para></typeparam>
472 /// <param name="list"><para>The list to copy from.</para><para>Список для
473   → копирования.</para></param>
474 /// <param name="skip"><para>The number of items to shift.</para><para>Количество
475   → сдвигаемых элементов.</para></param>
476 /// <returns>
477 /// <para>If the value of the shift variable is less than zero - an <see
478   → cref="NotImplementedException"/> exception is thrown, but if the value of the shift
479   → variable is 0 - an exact copy of the array is returned. Otherwise, an array is
480   → returned with the shift of the elements.</para>
481 /// <para>Если значение переменной shift меньше нуля - выбрасывается исключение <see
482   → cref="NotImplementedException"/>, если же значение переменной shift равно 0 -
483   → возвращается точная копия массива. Иначе возвращается массив со сдвигом
484   → элементов.</para>
485 /// </returns>
486 [MethodImpl(MethodImplOptions.AggressiveInlining)]
487 public static IList<T> ShiftRight<T>(this IList<T> list, int shift)
488 {
489     if (shift < 0)
490     {
491         throw new NotImplementedException();
492     }
493     if (shift == 0)
494     {
495         return list.ToArray();
496     }
497     else
498     {
499         var result = new T[list.Count + shift];
500         for (int r = 0, w = shift; r < list.Count; r++, w++)
501         {
502             result[w] = list[r];
503         }
504         return result;
505     }
506 }
507 }

```

1.19 ./csharp/Platform.Collections/Lists/ListFiller.cs

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3
4 namespace Platform.Collections.Lists
5 {
6     public class ListFiller<TElement, TReturnConstant>
7     {
8         protected readonly List<TElement> _list;

```

```

9         protected readonly TReturnConstant _returnConstant;
10
11         /// <summary>
12         /// <para>Initializes a new instance of the ListFiller class.</para>
13         /// <para>Инициализирует новый экземпляр класса ListFiller.</para>
14         /// </summary>
15         /// <param name="list"><para>The list to be filled.</para><para>Список который будет
16         → заполняться.</para></param>
17         /// <param name="returnConstant"><para>The value for the constant returned by
18         → corresponding methods.</para><para>Значение для константы возвращаемой
19         → соответствующими методами.</para></param>
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         public ListFiller(List<TElement> list, TReturnConstant returnConstant)
22         {
23             _list = list;
24             _returnConstant = returnConstant;
25         }
26
27         [MethodImpl(MethodImplOptions.AggressiveInlining)]
28         public ListFiller(List<TElement> list) : this(list, default) { }
29
30         /// <summary>
31         /// <para>Adds an item to the end of the list.</para>
32         /// <para>Добавляет элемент в конец списка.</para>
33         /// </summary>
34         /// <param name="element"><para>Element to add.</para><para>Добавляемый
35         → элемент.</para></param>
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         public void Add(TElement element) => _list.Add(element);
38
39         /// <summary>
40         /// <para>Adds an item to the end of the list and return true.</para>
41         /// <para>Добавляет элемент в конец списка и возвращает true.</para>
42         /// </summary>
43         /// <param name="element"><para>Element to add.</para><para>Добавляемый
44         → элемент.</para></param>
45         /// <returns>
46         /// <para>True value in any case.</para>
47         /// <para>Значение true в любом случае.</para>
48         /// </returns>
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         public bool AddAndReturnTrue(TElement element) => _list.AddAndReturnTrue(element);
51
52         /// <summary>
53         /// <para>Adds a value to the list at the first index and return true.</para>
54         /// <para>Добавляет значение в список по первому индексу и возвращает true.</para>
55         /// </summary>
56         /// <param name="element"><para>Element to add.</para><para>Добавляемый
57         → элемент.</para></param>
58         /// <returns>
59         /// <para>True value in any case.</para>
60         /// <para>Значение true в любом случае.</para>
61         /// </returns>
62         [MethodImpl(MethodImplOptions.AggressiveInlining)]
63         public bool AddFirstAndReturnTrue(ICollection<TElement> elements) =>
64         → _list.AddFirstAndReturnTrue(elements);
65
66         /// <summary>
67         /// <para>Adds all elements from other list to this list and returns true.</para>
68         /// <para>Добавляет все элементы из другого списка в этот список и возвращает
69         → true.</para>
70         /// </summary>
71         /// <param name="elements"><para>List of values to add.</para><para>Список значений
72         → которые необходимо добавить.</para></param>
73         /// <returns>
74         /// <para>True value in any case.</para>
75         /// <para>Значение true в любом случае.</para>
76         /// </returns>
77         [MethodImpl(MethodImplOptions.AggressiveInlining)]
78         public bool AddAllAndReturnTrue(ICollection<TElement> elements) =>
79         → _list.AddAllAndReturnTrue(elements);
80
81         /// <summary>
82         /// <para>Adds values to the list skipping the first element.</para>
83         /// <para>Добавляет значения в список пропуская первый элемент.</para>
84         /// </summary>
85         /// <param name="elements"><para>The list of values to add.</para><para>Список значений
86         → которые необходимо добавить.</para></param>

```

```

76     /// <returns>
77     /// <para>True value in any case.</para>
78     /// <para>Значение true в любом случае.</para>
79     /// </returns>
80     [MethodImpl(MethodImplOptions.AggressiveInlining)]
81     public bool AddSkipFirstAndReturnTrue(IList<TElement> elements) =>
82         _list.AddSkipFirstAndReturnTrue(elements);
83
84     /// <summary>
85     /// <para>Adds an item to the end of the list and return constant.</para>
86     /// <para>Добавляет элемент в конец списка и возвращает константу.</para>
87     /// </summary>
88     /// <param name="element"><para>Element to add.</para><para>Добавляемый
89     <para>→ элемент.</para></param>
90     /// <returns>
91     /// <para>Constant value in any case.</para>
92     /// <para>Значение константы в любом случае.</para>
93     /// </returns>
94     [MethodImpl(MethodImplOptions.AggressiveInlining)]
95     public TReturnConstant AddAndReturnConstant(TElement element)
96     {
97         _list.Add(element);
98         return _returnConstant;
99     }
100
101     /// <summary>
102     /// <para>Adds a value to the list at the first index and return constant.</para>
103     /// <para>Добавляет значение в список по первому индексу и возвращает константу.</para>
104     /// </summary>
105     /// <param name="element"><para>Element to add.</para><para>Добавляемый
106     <para>→ элемент.</para></param>
107     /// <returns>
108     /// <para>Constant value in any case.</para>
109     /// <para>Значение константы в любом случае.</para>
110     /// </returns>
111     [MethodImpl(MethodImplOptions.AggressiveInlining)]
112     public TReturnConstant AddFirstAndReturnConstant(IList<TElement> elements)
113     {
114         _list.AddFirst(elements);
115         return _returnConstant;
116     }
117
118     /// <summary>
119     /// <para>Adds all elements from other list to this list and returns constant.</para>
120     /// <para>Добавляет все элементы из другого списка в этот список и возвращает
121     <para>→ константу.</para>
122     /// </summary>
123     /// <param name="elements"><para>List of values to add.</para><para>Список значений
124     <para>→ которые необходимо добавить.</para></param>
125     /// <returns>
126     /// <para>Constant value in any case.</para>
127     /// <para>Значение константы в любом случае.</para>
128     /// </returns>
129     [MethodImpl(MethodImplOptions.AggressiveInlining)]
130     public TReturnConstant AddAllAndReturnConstant(IList<TElement> elements)
131     {
132         _list.AddAll(elements);
133         return _returnConstant;
134     }
135
136     /// <summary>
137     /// <para>Adds values to the list skipping the first element and return constant
138     <para>→ value.</para>
139     /// <para>Добавляет значения в список пропуская первый элемент и возвращает значение
140     <para>→ константы.</para>
141     /// </summary>
142     /// <param name="elements"><para>The list of values to add.</para><para>Список значений
143     <para>→ которые необходимо добавить.</para></param>
144     /// <returns>
145     /// <para>constant value in any case.</para>
146     /// <para>Значение константы в любом случае.</para>
147     /// </returns>
148     [MethodImpl(MethodImplOptions.AggressiveInlining)]
149     public TReturnConstant AddSkipFirstAndReturnConstant(IList<TElement> elements)
150     {
151         _list.AddSkipFirst(elements);
152         return _returnConstant;
153     }

```

```

145     }
146 }
147 }

```

1.20 ./csharp/Platform.Collections/Segments/CharSegment.cs

```

1  using System.Linq;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Collections.Arrays;
5  using Platform.Collections.Lists;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Collections.Segments
10 {
11     public class CharSegment : Segment<char>
12     {
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public CharSegment(IList<char> @base, int offset, int length) : base(@base, offset,
15             ↪ length) { }
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public override int GetHashCode()
19         {
20             // Base can be not an array, but still IList<char>
21             if (Base is char[] baseArray)
22             {
23                 return baseArray.GenerateHashCode(Offset, Length);
24             }
25             else
26             {
27                 return this.GenerateHashCode();
28             }
29         }
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         public override bool Equals(Segment<char> other)
33         {
34             bool contentEqualityComparer(IList<char> left, IList<char> right)
35             {
36                 // Base can be not an array, but still IList<char>
37                 if (Base is char[] baseArray && other.Base is char[] otherArray)
38                 {
39                     return baseArray.ContentEqualTo(Offset, Length, otherArray, other.Offset);
40                 }
41                 else
42                 {
43                     return left.ContentEqualTo(right);
44                 }
45             }
46             return this.EqualTo(other, contentEqualityComparer);
47         }
48
49         public override bool Equals(object obj) => obj is Segment<char> charSegment ?
50             ↪ Equals(charSegment) : false;
51
52         [MethodImpl(MethodImplOptions.AggressiveInlining)]
53         public static implicit operator string(CharSegment segment)
54         {
55             if (!(segment.Base is char[] array))
56             {
57                 array = segment.Base.ToArray();
58             }
59             return new string(array, segment.Offset, segment.Length);
60         }
61
62         [MethodImpl(MethodImplOptions.AggressiveInlining)]
63         public override string ToString() => this;
64     }
65 }

```

1.21 ./csharp/Platform.Collections/Segments/Segment.cs

```

1  using System;
2  using System.Collections;
3  using System.Collections.Generic;
4  using System.Runtime.CompilerServices;
5  using Platform.Collections.Arrays;
6  using Platform.Collections.Lists;
7

```

```

8 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Collections.Segments
11 {
12     public class Segment<T> : IEquatable<Segment<T>>, IList<T>
13     {
14         public IList<T> Base
15         {
16             [MethodImpl(MethodImplOptions.AggressiveInlining)]
17             get;
18         }
19         public int Offset
20         {
21             [MethodImpl(MethodImplOptions.AggressiveInlining)]
22             get;
23         }
24         public int Length
25         {
26             [MethodImpl(MethodImplOptions.AggressiveInlining)]
27             get;
28         }
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         public Segment(IList<T> @base, int offset, int length)
32         {
33             Base = @base;
34             Offset = offset;
35             Length = length;
36         }
37
38         [MethodImpl(MethodImplOptions.AggressiveInlining)]
39         public override int GetHashCode() => this.GenerateHashCode();
40
41         [MethodImpl(MethodImplOptions.AggressiveInlining)]
42         public virtual bool Equals(Segment<T> other) => this.EqualTo(other);
43
44         [MethodImpl(MethodImplOptions.AggressiveInlining)]
45         public override bool Equals(object obj) => obj is Segment<T> other ? Equals(other) :
46             ↪ false;
47
48         #region IList
49         public T this[int i]
50         {
51             [MethodImpl(MethodImplOptions.AggressiveInlining)]
52             get => Base[Offset + i];
53             [MethodImpl(MethodImplOptions.AggressiveInlining)]
54             set => Base[Offset + i] = value;
55         }
56
57         public int Count
58         {
59             [MethodImpl(MethodImplOptions.AggressiveInlining)]
60             get => Length;
61         }
62
63         public bool IsReadOnly
64         {
65             [MethodImpl(MethodImplOptions.AggressiveInlining)]
66             get => true;
67         }
68
69         [MethodImpl(MethodImplOptions.AggressiveInlining)]
70         public int IndexOf(T item)
71         {
72             var index = Base.IndexOf(item);
73             if (index >= Offset)
74             {
75                 var actualIndex = index - Offset;
76                 if (actualIndex < Length)
77                 {
78                     return actualIndex;
79                 }
80             }
81             return -1;
82         }
83
84         [MethodImpl(MethodImplOptions.AggressiveInlining)]
85         public void Insert(int index, T item) => throw new NotSupportedException();
86

```

```

87     [MethodImpl(MethodImplOptions.AggressiveInlining)]
88     public void RemoveAt(int index) => throw new NotSupportedException();
89
90     [MethodImpl(MethodImplOptions.AggressiveInlining)]
91     public void Add(T item) => throw new NotSupportedException();
92
93     [MethodImpl(MethodImplOptions.AggressiveInlining)]
94     public void Clear() => throw new NotSupportedException();
95
96     [MethodImpl(MethodImplOptions.AggressiveInlining)]
97     public bool Contains(T item) => IndexOf(item) >= 0;
98
99     [MethodImpl(MethodImplOptions.AggressiveInlining)]
100    public void CopyTo(T[] array, int arrayIndex)
101    {
102        for (var i = 0; i < Length; i++)
103        {
104            array.Add(ref arrayIndex, this[i]);
105        }
106    }
107
108    [MethodImpl(MethodImplOptions.AggressiveInlining)]
109    public bool Remove(T item) => throw new NotSupportedException();
110
111    [MethodImpl(MethodImplOptions.AggressiveInlining)]
112    public IEnumerator<T> GetEnumerator()
113    {
114        for (var i = 0; i < Length; i++)
115        {
116            yield return this[i];
117        }
118    }
119
120    [MethodImpl(MethodImplOptions.AggressiveInlining)]
121    IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
122
123    #endregion
124 }
125 }

```

1.22 ./csharp/Platform.Collections.Segments.Walkers.AllSegmentsWalkerBase.cs

```

1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  namespace Platform.Collections.Segments.Walkers
4  {
5      public abstract class AllSegmentsWalkerBase
6      {
7          public static readonly int DefaultMinimumStringSegmentLength = 2;
8      }
9  }

```

1.23 ./csharp/Platform.Collections.Segments.Walkers.AllSegmentsWalkerBase[T, TSegment].cs

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Collections.Segments.Walkers
7  {
8      public abstract class AllSegmentsWalkerBase<T, TSegment> : AllSegmentsWalkerBase
9          where TSegment : Segment<T>
10     {
11         private readonly int _minimumStringSegmentLength;
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         protected AllSegmentsWalkerBase(int minimumStringSegmentLength) =>
15             ↪ _minimumStringSegmentLength = minimumStringSegmentLength;
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         protected AllSegmentsWalkerBase() : this(DefaultMinimumStringSegmentLength) { }
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         public virtual void WalkAll(ICollection<T> elements)
22         {
23             for (int offset = 0, maxOffset = elements.Count - _minimumStringSegmentLength;
24                 ↪ offset <= maxOffset; offset++)
25             {
26                 for (int length = _minimumStringSegmentLength, maxLength = elements.Count -
27                     ↪ offset; length <= maxLength; length++)
28                 {
29                     // ...
30                 }
31             }
32         }
33     }
34 }

```

```

25         {
26             Iteration(CreateSegment(elements, offset, length));
27         }
28     }
29 }
30
31 [MethodImpl(MethodImplOptions.AggressiveInlining)]
32 protected abstract TSegment CreateSegment(ICollection<T> elements, int offset, int length);
33
34 [MethodImpl(MethodImplOptions.AggressiveInlining)]
35 protected abstract void Iteration(TSegment segment);
36 }
37 }

```

1.24 ./csharp/Platform.Collections.Segments/Walkers/AllSegmentsWalkerBase[T].cs

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Collections.Segments.Walkers
7 {
8     public abstract class AllSegmentsWalkerBase<T> : AllSegmentsWalkerBase<T, Segment<T>>
9     {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         protected override Segment<T> CreateSegment(ICollection<T> elements, int offset, int length)
12             => new Segment<T>(elements, offset, length);
13     }
14 }

```

1.25 ./csharp/Platform.Collections.Segments/Walkers/AllSegmentsWalkerExtensions.cs

```

1 using System.Runtime.CompilerServices;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Collections.Segments.Walkers
6 {
7     public static class AllSegmentsWalkerExtensions
8     {
9         [MethodImpl(MethodImplOptions.AggressiveInlining)]
10         public static void WalkAll(this AllSegmentsWalkerBase<char> walker, string @string) =>
11             walker.WalkAll(@string.ToCharArray());
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public static void WalkAll<TSegment>(this AllSegmentsWalkerBase<char, TSegment> walker,
15             string @string) where TSegment : Segment<char> =>
16             walker.WalkAll(@string.ToCharArray());
17     }
18 }

```

1.26 ./csharp/Platform.Collections.Segments/Walkers/DictionaryBasedDuplicateSegmentsWalkerBase[T, TSegment]

```

1 using System;
2 using System.Collections.Generic;
3 using System.Runtime.CompilerServices;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Collections.Segments.Walkers
8 {
9     public abstract class DictionaryBasedDuplicateSegmentsWalkerBase<T, TSegment> :
10         DuplicateSegmentsWalkerBase<T, TSegment>
11         where TSegment : Segment<T>
12     {
13         public static readonly bool DefaultResetDictionaryOnEachWalk;
14
15         private readonly bool _resetDictionaryOnEachWalk;
16         protected IDictionary<TSegment, long> Dictionary;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<TSegment, long>
20             dictionary, int minimumStringSegmentLength, bool resetDictionaryOnEachWalk)
21             : base(minimumStringSegmentLength)
22         {
23             Dictionary = dictionary;
24             _resetDictionaryOnEachWalk = resetDictionaryOnEachWalk;
25         }
26
27         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```



```

26     protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<TSegment, long>
    ↪ dictionary, int minimumStringSegmentLength) : this(dictionary,
    ↪ minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
27
28     [MethodImpl(MethodImplOptions.AggressiveInlining)]
29     protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<TSegment, long>
    ↪ dictionary) : this(dictionary, DefaultMinimumStringSegmentLength,
    ↪ DefaultResetDictionaryOnEachWalk) { }
30
31     [MethodImpl(MethodImplOptions.AggressiveInlining)]
32     protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength,
    ↪ bool resetDictionaryOnEachWalk) : this(resetDictionaryOnEachWalk ? null : new
    ↪ Dictionary<TSegment, long>(), minimumStringSegmentLength, resetDictionaryOnEachWalk)
    ↪ { }
33
34     [MethodImpl(MethodImplOptions.AggressiveInlining)]
35     protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength) :
    ↪ this(minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
36
37     [MethodImpl(MethodImplOptions.AggressiveInlining)]
38     protected DictionaryBasedDuplicateSegmentsWalkerBase() :
    ↪ this(DefaultMinimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
39
40     [MethodImpl(MethodImplOptions.AggressiveInlining)]
41     public override void WalkAll(ICollection<T> elements)
42     {
43         if (_resetDictionaryOnEachWalk)
44         {
45             var capacity = Math.Ceiling(Math.Pow(elements.Count, 2) / 2);
46             Dictionary = new Dictionary<TSegment, long>((int)capacity);
47         }
48         base.WalkAll(elements);
49     }
50
51     [MethodImpl(MethodImplOptions.AggressiveInlining)]
52     protected override long GetSegmentFrequency(TSegment segment) =>
    ↪ Dictionary.GetOrDefault(segment);
53
54     [MethodImpl(MethodImplOptions.AggressiveInlining)]
55     protected override void SetSegmentFrequency(TSegment segment, long frequency) =>
    ↪ Dictionary[segment] = frequency;
56 }
57 }

```

1.27 ./csharp/Platform.Collections.Segments.Walkers/DictionaryBasedDuplicateSegmentsWalkerBase[T].cs

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Collections.Segments.Walkers
7  {
8      public abstract class DictionaryBasedDuplicateSegmentsWalkerBase<T> :
    ↪ DictionaryBasedDuplicateSegmentsWalkerBase<T, Segment<T>>
9      {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<Segment<T>, long>
    ↪ dictionary, int minimumStringSegmentLength, bool resetDictionaryOnEachWalk) :
    ↪ base(dictionary, minimumStringSegmentLength, resetDictionaryOnEachWalk) { }
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<Segment<T>, long>
    ↪ dictionary, int minimumStringSegmentLength) : base(dictionary,
    ↪ minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<Segment<T>, long>
    ↪ dictionary) : base(dictionary, DefaultMinimumStringSegmentLength,
    ↪ DefaultResetDictionaryOnEachWalk) { }
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength,
    ↪ bool resetDictionaryOnEachWalk) : base(minimumStringSegmentLength,
    ↪ resetDictionaryOnEachWalk) { }
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength) :
    ↪ base(minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }

```

```

24     [MethodImpl(MethodImplOptions.AggressiveInlining)]
25     protected DictionaryBasedDuplicateSegmentsWalkerBase() :
26         ↪ base(DefaultMinimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
27 }
28 }

```

1.28 ./csharp/Platform.Collections.Segments.Walkers/DuplicateSegmentsWalkerBase[T, TSegment].cs

```

1  using System.Runtime.CompilerServices;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Collections.Segments.Walkers
6  {
7      public abstract class DuplicateSegmentsWalkerBase<T, TSegment> : AllSegmentsWalkerBase<T,
8          ↪ TSegment>
9          where TSegment : Segment<T>
10     {
11         [MethodImpl(MethodImplOptions.AggressiveInlining)]
12         protected DuplicateSegmentsWalkerBase(int minimumStringSegmentLength) :
13             ↪ base(minimumStringSegmentLength) { }
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         protected DuplicateSegmentsWalkerBase() : base(DefaultMinimumStringSegmentLength) { }
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected override void Iteration(TSegment segment)
20         {
21             var frequency = GetSegmentFrequency(segment);
22             if (frequency == 1)
23             {
24                 OnDuplicateFound(segment);
25             }
26             SetSegmentFrequency(segment, frequency + 1);
27         }
28
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         protected abstract void OnDuplicateFound(TSegment segment);
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         protected abstract long GetSegmentFrequency(TSegment segment);
34
35         [MethodImpl(MethodImplOptions.AggressiveInlining)]
36         protected abstract void SetSegmentFrequency(TSegment segment, long frequency);
37     }
38 }

```

1.29 ./csharp/Platform.Collections.Segments.Walkers/DuplicateSegmentsWalkerBase[T].cs

```

1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  namespace Platform.Collections.Segments.Walkers
4  {
5      public abstract class DuplicateSegmentsWalkerBase<T> : DuplicateSegmentsWalkerBase<T,
6          ↪ Segment<T>>
7      {
8      }
9  }

```

1.30 ./csharp/Platform.Collections.Sets/ISetExtensions.cs

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Collections.Sets
7  {
8      public static class ISetExtensions
9      {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public static void AddAndReturnVoid<T>(this ISet<T> set, T element) => set.Add(element);
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public static void RemoveAndReturnVoid<T>(this ISet<T> set, T element) =>
15             ↪ set.Remove(element);
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public static bool AddAndReturnTrue<T>(this ISet<T> set, T element)
19         {

```

```

19         set.Add(element);
20         return true;
21     }
22
23     [MethodImpl(MethodImplOptions.AggressiveInlining)]
24     public static bool AddFirstAndReturnTrue<T>(this ISet<T> set, IList<T> elements)
25     {
26         AddFirst(set, elements);
27         return true;
28     }
29
30     [MethodImpl(MethodImplOptions.AggressiveInlining)]
31     public static void AddFirst<T>(this ISet<T> set, IList<T> elements) =>
32         ↪ set.Add(elements[0]);
33
34     [MethodImpl(MethodImplOptions.AggressiveInlining)]
35     public static bool AddAllAndReturnTrue<T>(this ISet<T> set, IList<T> elements)
36     {
37         set.AddAll(elements);
38         return true;
39     }
40
41     [MethodImpl(MethodImplOptions.AggressiveInlining)]
42     public static void AddAll<T>(this ISet<T> set, IList<T> elements)
43     {
44         for (var i = 0; i < elements.Count; i++)
45         {
46             set.Add(elements[i]);
47         }
48     }
49
50     [MethodImpl(MethodImplOptions.AggressiveInlining)]
51     public static bool AddSkipFirstAndReturnTrue<T>(this ISet<T> set, IList<T> elements)
52     {
53         set.AddSkipFirst(elements);
54         return true;
55     }
56
57     [MethodImpl(MethodImplOptions.AggressiveInlining)]
58     public static void AddSkipFirst<T>(this ISet<T> set, IList<T> elements) =>
59         ↪ set.AddSkipFirst(elements, 1);
60
61     [MethodImpl(MethodImplOptions.AggressiveInlining)]
62     public static void AddSkipFirst<T>(this ISet<T> set, IList<T> elements, int skip)
63     {
64         for (var i = skip; i < elements.Count; i++)
65         {
66             set.Add(elements[i]);
67         }
68     }
69
70     [MethodImpl(MethodImplOptions.AggressiveInlining)]
71     public static bool DoNotContains<T>(this ISet<T> set, T element) =>
72         ↪ !set.Contains(element);
73 }

```

1.31 ./csharp/Platform.Collections/Sets/SetFiller.cs

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Collections.Sets
7 {
8     public class SetFiller<TElement, TReturnConstant>
9     {
10         protected readonly ISet<TElement> _set;
11         protected readonly TReturnConstant _returnConstant;
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public SetFiller(ISet<TElement> set, TReturnConstant returnConstant)
15         {
16             _set = set;
17             _returnConstant = returnConstant;
18         }
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         public SetFiller(ISet<TElement> set) : this(set, default) { }
22     }

```

```

23     [MethodImpl(MethodImplOptions.AggressiveInlining)]
24     public void Add(TElement element) => _set.Add(element);
25
26     [MethodImpl(MethodImplOptions.AggressiveInlining)]
27     public bool AddAndReturnTrue(TElement element) => _set.AddAndReturnTrue(element);
28
29     [MethodImpl(MethodImplOptions.AggressiveInlining)]
30     public bool AddFirstAndReturnTrue(IList<TElement> elements) =>
31         => _set.AddFirstAndReturnTrue(elements);
32
33     [MethodImpl(MethodImplOptions.AggressiveInlining)]
34     public bool AddAllAndReturnTrue(IList<TElement> elements) =>
35         => _set.AddAllAndReturnTrue(elements);
36
37     [MethodImpl(MethodImplOptions.AggressiveInlining)]
38     public bool AddSkipFirstAndReturnTrue(IList<TElement> elements) =>
39         => _set.AddSkipFirstAndReturnTrue(elements);
40
41     [MethodImpl(MethodImplOptions.AggressiveInlining)]
42     public TReturnConstant AddAndReturnConstant(TElement element)
43     {
44         _set.Add(element);
45         return _returnConstant;
46     }
47
48     [MethodImpl(MethodImplOptions.AggressiveInlining)]
49     public TReturnConstant AddFirstAndReturnConstant(IList<TElement> elements)
50     {
51         _set.AddFirst(elements);
52         return _returnConstant;
53     }
54
55     [MethodImpl(MethodImplOptions.AggressiveInlining)]
56     public TReturnConstant AddAllAndReturnConstant(IList<TElement> elements)
57     {
58         _set.AddAll(elements);
59         return _returnConstant;
60     }
61
62     [MethodImpl(MethodImplOptions.AggressiveInlining)]
63     public TReturnConstant AddSkipFirstAndReturnConstant(IList<TElement> elements)
64     {
65         _set.AddSkipFirst(elements);
66         return _returnConstant;
67     }
68 }

```

1.32 ./csharp/Platform.Collections/Stacks/DefaultStack.cs

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Collections.Stacks
7  {
8      public class DefaultStack<TElement> : Stack<TElement>, IStack<TElement>
9      {
10         public bool IsEmpty
11         {
12             [MethodImpl(MethodImplOptions.AggressiveInlining)]
13             get => Count <= 0;
14         }
15     }
16 }

```

1.33 ./csharp/Platform.Collections/Stacks/IStack.cs

```

1  using System.Runtime.CompilerServices;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Collections.Stacks
6  {
7      public interface IStack<TElement>
8      {
9         bool IsEmpty
10         {
11             [MethodImpl(MethodImplOptions.AggressiveInlining)]
12             get;
13         }
14     }
15 }

```

```

13     }
14
15     [MethodImpl(MethodImplOptions.AggressiveInlining)]
16     void Push(TElement element);
17
18     [MethodImpl(MethodImplOptions.AggressiveInlining)]
19     TElement Pop();
20
21     [MethodImpl(MethodImplOptions.AggressiveInlining)]
22     TElement Peek();
23 }
24 }

```

1.34 ./csharp/Platform.Collections/Stacks/IStackExtensions.cs

```

1 using System.Runtime.CompilerServices;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Collections.Stacks
6 {
7     public static class IStackExtensions
8     {
9         [MethodImpl(MethodImplOptions.AggressiveInlining)]
10        public static void Clear<T>(this IStack<T> stack)
11        {
12            while (!stack.IsEmpty)
13            {
14                _ = stack.Pop();
15            }
16        }
17
18        [MethodImpl(MethodImplOptions.AggressiveInlining)]
19        public static T PopOrDefault<T>(this IStack<T> stack) => stack.IsEmpty ? default :
20            ↪ stack.Pop();
21
22        [MethodImpl(MethodImplOptions.AggressiveInlining)]
23        public static T PeekOrDefault<T>(this IStack<T> stack) => stack.IsEmpty ? default :
24            ↪ stack.Peek();
25    }
26 }

```

1.35 ./csharp/Platform.Collections/Stacks/IStackFactory.cs

```

1 using Platform.Interfaces;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Collections.Stacks
6 {
7     public interface IStackFactory<TElement> : IFactory<IStack<TElement>>
8     {
9     }
10 }

```

1.36 ./csharp/Platform.Collections/Stacks/StackExtensions.cs

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Collections.Stacks
7 {
8     public static class StackExtensions
9     {
10        [MethodImpl(MethodImplOptions.AggressiveInlining)]
11        public static T PopOrDefault<T>(this Stack<T> stack) => stack.Count > 0 ? stack.Pop() :
12            ↪ default;
13
14        [MethodImpl(MethodImplOptions.AggressiveInlining)]
15        public static T PeekOrDefault<T>(this Stack<T> stack) => stack.Count > 0 ? stack.Peek()
16            ↪ : default;
17    }
18 }

```

1.37 ./csharp/Platform.Collections/StringExtensions.cs

```

1 using System;
2 using System.Globalization;
3 using System.Runtime.CompilerServices;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

```

```

6
7 namespace Platform.Collections
8 {
9     public static class StringExtensions
10    {
11        [MethodImpl(MethodImplOptions.AggressiveInlining)]
12        public static string CapitalizeFirstLetter(this string @string)
13        {
14            if (@string.IsNullOrEmpty(@string))
15            {
16                return @string;
17            }
18            var chars = @string.ToCharArray();
19            for (var i = 0; i < chars.Length; i++)
20            {
21                var category = char.GetUnicodeCategory(chars[i]);
22                if (category == UnicodeCategory.UppercaseLetter)
23                {
24                    return @string;
25                }
26                if (category == UnicodeCategory.LowercaseLetter)
27                {
28                    chars[i] = char.ToUpper(chars[i]);
29                    return new string(chars);
30                }
31            }
32            return @string;
33        }
34
35        [MethodImpl(MethodImplOptions.AggressiveInlining)]
36        public static string Truncate(this string @string, int maxLength) =>
37            ↪ @string.IsNullOrEmpty(@string) ? @string : @string.Substring(0,
38            ↪ Math.Min(@string.Length, maxLength));
39
40        [MethodImpl(MethodImplOptions.AggressiveInlining)]
41        public static string TrimSingle(this string @string, char charToTrim)
42        {
43            if (!@string.IsNullOrEmpty(@string))
44            {
45                if (@string.Length == 1)
46                {
47                    if (@string[0] == charToTrim)
48                    {
49                        return "";
50                    }
51                    else
52                    {
53                        return @string;
54                    }
55                }
56                else
57                {
58                    var left = 0;
59                    var right = @string.Length - 1;
60                    if (@string[left] == charToTrim)
61                    {
62                        left++;
63                    }
64                    if (@string[right] == charToTrim)
65                    {
66                        right--;
67                    }
68                    return @string.Substring(left, right - left + 1);
69                }
70            }
71            else
72            {
73                return @string;
74            }
75        }
76    }
77 }

```

1.38 ./csharp/Platform.Collections/Trees/Node.cs

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3
4 // ReSharper disable ForCanBeConvertedToForeach
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

```

```

6
7 namespace Platform.Collections.Trees
8 {
9     public class Node
10    {
11        private Dictionary<object, Node> _childNodes;
12
13        public object Value
14        {
15            [MethodImpl(MethodImplOptions.AggressiveInlining)]
16            get;
17            [MethodImpl(MethodImplOptions.AggressiveInlining)]
18            set;
19        }
20
21        public Dictionary<object, Node> ChildNodes
22        {
23            [MethodImpl(MethodImplOptions.AggressiveInlining)]
24            get => _childNodes ?? (_childNodes = new Dictionary<object, Node>());
25        }
26
27        public Node this[object key]
28        {
29            [MethodImpl(MethodImplOptions.AggressiveInlining)]
30            get => GetChild(key) ?? AddChild(key);
31            [MethodImpl(MethodImplOptions.AggressiveInlining)]
32            set => SetChildValue(value, key);
33        }
34
35        [MethodImpl(MethodImplOptions.AggressiveInlining)]
36        public Node(object value) => Value = value;
37
38        [MethodImpl(MethodImplOptions.AggressiveInlining)]
39        public Node() : this(null) { }
40
41        [MethodImpl(MethodImplOptions.AggressiveInlining)]
42        public bool ContainsChild(params object[] keys) => GetChild(keys) != null;
43
44        [MethodImpl(MethodImplOptions.AggressiveInlining)]
45        public Node GetChild(params object[] keys)
46        {
47            var node = this;
48            for (var i = 0; i < keys.Length; i++)
49            {
50                node.ChildNodes.TryGetValue(keys[i], out node);
51                if (node == null)
52                {
53                    return null;
54                }
55            }
56            return node;
57        }
58
59        [MethodImpl(MethodImplOptions.AggressiveInlining)]
60        public object GetChildValue(params object[] keys) => GetChild(keys)?.Value;
61
62        [MethodImpl(MethodImplOptions.AggressiveInlining)]
63        public Node AddChild(object key) => AddChild(key, new Node(null));
64
65        [MethodImpl(MethodImplOptions.AggressiveInlining)]
66        public Node AddChild(object key, object value) => AddChild(key, new Node(value));
67
68        [MethodImpl(MethodImplOptions.AggressiveInlining)]
69        public Node AddChild(object key, Node child)
70        {
71            ChildNodes.Add(key, child);
72            return child;
73        }
74
75        [MethodImpl(MethodImplOptions.AggressiveInlining)]
76        public Node SetChild(params object[] keys) => SetChildValue(null, keys);
77
78        [MethodImpl(MethodImplOptions.AggressiveInlining)]
79        public Node SetChild(object key) => SetChildValue(null, key);
80
81        [MethodImpl(MethodImplOptions.AggressiveInlining)]
82        public Node SetChildValue(object value, params object[] keys)
83        {
84            var node = this;
85            for (var i = 0; i < keys.Length; i++)

```

```

86         {
87             node = SetChildValue(value, keys[i]);
88         }
89         node.Value = value;
90         return node;
91     }
92
93     [MethodImpl(MethodImplOptions.AggressiveInlining)]
94     public Node SetChildValue(object value, object key)
95     {
96         if (!ChildNodes.TryGetValue(key, out Node child))
97         {
98             child = AddChild(key, value);
99         }
100         child.Value = value;
101         return child;
102     }
103 }
104 }

```

1.39 ./csharp/Platform.Collections.Tests/ArrayTests.cs

```

1  using Xunit;
2  using Platform.Collections.Arrays;
3
4  namespace Platform.Collections.Tests
5  {
6      public class ArrayTests
7      {
8          [Fact]
9          public void GetElementTest()
10         {
11             var nullArray = (int[])null;
12             Assert.Equal(0, nullArray.GetElementOrDefault(1));
13             Assert.False(nullArray.TryGetElement(1, out int element));
14             Assert.Equal(0, element);
15             var array = new int[] { 1, 2, 3 };
16             Assert.Equal(3, array.GetElementOrDefault(2));
17             Assert.True(array.TryGetElement(2, out element));
18             Assert.Equal(3, element);
19             Assert.Equal(0, array.GetElementOrDefault(10));
20             Assert.False(array.TryGetElement(10, out element));
21             Assert.Equal(0, element);
22         }
23     }
24 }

```

1.40 ./csharp/Platform.Collections.Tests/BitStringTests.cs

```

1  using System;
2  using System.Collections;
3  using Xunit;
4  using Platform.Random;
5
6  namespace Platform.Collections.Tests
7  {
8      public static class BitStringTests
9      {
10         [Fact]
11         public static void BitGetSetTest()
12         {
13             const int n = 250;
14             var bitArray = new BitArray(n);
15             var bitString = new BitString(n);
16             for (var i = 0; i < n; i++)
17             {
18                 var value = RandomHelpers.Default.NextBoolean();
19                 bitArray.Set(i, value);
20                 bitString.Set(i, value);
21                 Assert.Equal(value, bitArray.Get(i));
22                 Assert.Equal(value, bitString.Get(i));
23             }
24         }
25
26         [Fact]
27         public static void BitVectorNotTest()
28         {
29             TestToOperationsWithSameMeaning((x, y, w, v) =>
30             {
31                 x.VectorNot();
32                 w.Not();

```



```

33     });
34 }
35
36 [Fact]
37 public static void BitParallelNotTest()
38 {
39     TestToOperationsWithSameMeaning((x, y, w, v) =>
40     {
41         x.ParallelNot();
42         w.Not();
43     });
44 }
45
46 [Fact]
47 public static void BitParallelVectorNotTest()
48 {
49     TestToOperationsWithSameMeaning((x, y, w, v) =>
50     {
51         x.ParallelVectorNot();
52         w.Not();
53     });
54 }
55
56 [Fact]
57 public static void BitVectorAndTest()
58 {
59     TestToOperationsWithSameMeaning((x, y, w, v) =>
60     {
61         x.VectorAnd(y);
62         w.And(v);
63     });
64 }
65
66 [Fact]
67 public static void BitParallelAndTest()
68 {
69     TestToOperationsWithSameMeaning((x, y, w, v) =>
70     {
71         x.ParallelAnd(y);
72         w.And(v);
73     });
74 }
75
76 [Fact]
77 public static void BitParallelVectorAndTest()
78 {
79     TestToOperationsWithSameMeaning((x, y, w, v) =>
80     {
81         x.ParallelVectorAnd(y);
82         w.And(v);
83     });
84 }
85
86 [Fact]
87 public static void BitVectorOrTest()
88 {
89     TestToOperationsWithSameMeaning((x, y, w, v) =>
90     {
91         x.VectorOr(y);
92         w.Or(v);
93     });
94 }
95
96 [Fact]
97 public static void BitParallelOrTest()
98 {
99     TestToOperationsWithSameMeaning((x, y, w, v) =>
100    {
101        x.ParallelOr(y);
102        w.Or(v);
103    });
104 }
105
106 [Fact]
107 public static void BitParallelVectorOrTest()
108 {
109     TestToOperationsWithSameMeaning((x, y, w, v) =>
110    {

```

```

111         x.ParallelVectorOr(y);
112         w.Or(v);
113     });
114 }
115
116 [Fact]
117 public static void BitVectorXorTest()
118 {
119     TestToOperationsWithSameMeaning((x, y, w, v) =>
120     {
121         x.VectorXor(y);
122         w.Xor(v);
123     });
124 }
125
126 [Fact]
127 public static void BitParallelXorTest()
128 {
129     TestToOperationsWithSameMeaning((x, y, w, v) =>
130     {
131         x.ParallelXor(y);
132         w.Xor(v);
133     });
134 }
135
136 [Fact]
137 public static void BitParallelVectorXorTest()
138 {
139     TestToOperationsWithSameMeaning((x, y, w, v) =>
140     {
141         x.ParallelVectorXor(y);
142         w.Xor(v);
143     });
144 }
145
146 private static void TestToOperationsWithSameMeaning(Action<BitString, BitString,
147     ↪ BitString, BitString> test)
148 {
149     const int n = 5654;
150     var x = new BitString(n);
151     var y = new BitString(n);
152     while (x.Equals(y))
153     {
154         x.SetRandomBits();
155         y.SetRandomBits();
156     }
157     var w = new BitString(x);
158     var v = new BitString(y);
159     Assert.False(x.Equals(y));
160     Assert.False(w.Equals(v));
161     Assert.True(x.Equals(w));
162     Assert.True(y.Equals(v));
163     test(x, y, w, v);
164     Assert.True(x.Equals(w));
165 }
166 }

```

1.41 ./csharp/Platform.Collections.Tests/CharsSegmentTests.cs

```

1 using Xunit;
2 using Platform.Collections.Segments;
3
4 namespace Platform.Collections.Tests
5 {
6     public static class CharsSegmentTests
7     {
8         [Fact]
9         public static void GetHashCodeEqualsTest()
10        {
11            const string testString = "test test";
12            var testArray = testString.ToCharArray();
13            var firstHashCode = new CharSegment(testArray, 0, 4).GetHashCode();
14            var secondHashCode = new CharSegment(testArray, 5, 4).GetHashCode();
15            Assert.Equal(firstHashCode, secondHashCode);
16        }
17
18        [Fact]
19        public static void EqualsTest()
20        {

```

```

21         const string testString = "test test";
22         var testArray = testString.ToCharArray();
23         var first = new CharSegment(testArray, 0, 4);
24         var second = new CharSegment(testArray, 5, 4);
25         Assert.True(first.Equals(second));
26     }
27 }
28 }

```

1.42 ./csharp/Platform.Collections.Tests/ListTests.cs

```

1  using System.Collections.Generic;
2  using Xunit;
3  using Platform.Collections.Lists;
4
5
6  namespace Platform.Collections.Tests
7  {
8      public class ListTests
9      {
10         [Fact]
11         public void GetElementTest()
12         {
13             var nullList = (IList<int>)null;
14             Assert.Equal(0, nullList.GetElementOrDefault(1));
15             Assert.False(nullList.TryGetElement(1, out int element));
16             Assert.Equal(0, element);
17             var list = new List<int>() { 1, 2, 3 };
18             Assert.Equal(3, list.GetElementOrDefault(2));
19             Assert.True(list.TryGetElement(2, out element));
20             Assert.Equal(3, element);
21             Assert.Equal(0, list.GetElementOrDefault(10));
22             Assert.False(list.TryGetElement(10, out element));
23             Assert.Equal(0, element);
24         }
25     }
26 }

```

1.43 ./csharp/Platform.Collections.Tests/StringTests.cs

```

1  using Xunit;
2
3  namespace Platform.Collections.Tests
4  {
5      public static class StringTests
6      {
7         [Fact]
8         public static void CapitalizeFirstLetterTest()
9         {
10             Assert.Equal("Hello", "hello".CapitalizeFirstLetter());
11             Assert.Equal("Hello", "Hello".CapitalizeFirstLetter());
12             Assert.Equal(" Hello", " hello".CapitalizeFirstLetter());
13         }
14
15         [Fact]
16         public static void TrimSingleTest()
17         {
18             Assert.Equal("", "".TrimSingle('\'));
19             Assert.Equal("", "''.TrimSingle('\'));
20             Assert.Equal("hello", "'hello'".TrimSingle('\'));
21             Assert.Equal("hello", "hello'".TrimSingle('\'));
22             Assert.Equal("hello", "'hello".TrimSingle('\'));
23         }
24     }
25 }

```

Index

- ./csharp/Platform.Collections.Tests/ArrayTests.cs, 56
- ./csharp/Platform.Collections.Tests/BitStringTests.cs, 56
- ./csharp/Platform.Collections.Tests/CharsSegmentTests.cs, 58
- ./csharp/Platform.Collections.Tests/ListTests.cs, 59
- ./csharp/Platform.Collections.Tests/StringTests.cs, 59
- ./csharp/Platform.Collections/Arrays/ArrayFiller[TElement, TReturnConstant].cs, 1
- ./csharp/Platform.Collections/Arrays/ArrayFiller[TElement].cs, 2
- ./csharp/Platform.Collections/Arrays/ArrayPool.cs, 4
- ./csharp/Platform.Collections/Arrays/ArrayPool[T].cs, 4
- ./csharp/Platform.Collections/Arrays/ArrayString.cs, 6
- ./csharp/Platform.Collections/Arrays/CharArrayExtensions.cs, 6
- ./csharp/Platform.Collections/Arrays/GenericArrayExtensions.cs, 8
- ./csharp/Platform.Collections/BitString.cs, 14
- ./csharp/Platform.Collections/BitStringExtensions.cs, 29
- ./csharp/Platform.Collections/Concurrent/ConcurrentQueueExtensions.cs, 29
- ./csharp/Platform.Collections/Concurrent/ConcurrentStackExtensions.cs, 30
- ./csharp/Platform.Collections/EnsureExtensions.cs, 30
- ./csharp/Platform.Collections/ICollectionExtensions.cs, 31
- ./csharp/Platform.Collections/IDictionaryExtensions.cs, 31
- ./csharp/Platform.Collections/Lists/CharIListExtensions.cs, 32
- ./csharp/Platform.Collections/Lists/IListComparer.cs, 33
- ./csharp/Platform.Collections/Lists/IListEqualityComparer.cs, 34
- ./csharp/Platform.Collections/Lists/IListExtensions.cs, 34
- ./csharp/Platform.Collections/Lists/ListFiller.cs, 42
- ./csharp/Platform.Collections/Segments/CharSegment.cs, 45
- ./csharp/Platform.Collections/Segments/Segment.cs, 45
- ./csharp/Platform.Collections/Segments/Walkers/AllSegmentsWalkerBase.cs, 47
- ./csharp/Platform.Collections/Segments/Walkers/AllSegmentsWalkerBase[T, TSegment].cs, 47
- ./csharp/Platform.Collections/Segments/Walkers/AllSegmentsWalkerBase[T].cs, 48
- ./csharp/Platform.Collections/Segments/Walkers/AllSegmentsWalkerExtensions.cs, 48
- ./csharp/Platform.Collections/Segments/Walkers/DictionaryBasedDuplicateSegmentsWalkerBase[T, Segment].cs, 48
- ./csharp/Platform.Collections/Segments/Walkers/DictionaryBasedDuplicateSegmentsWalkerBase[T].cs, 49
- ./csharp/Platform.Collections/Segments/Walkers/DuplicateSegmentsWalkerBase[T, TSegment].cs, 50
- ./csharp/Platform.Collections/Segments/Walkers/DuplicateSegmentsWalkerBase[T].cs, 50
- ./csharp/Platform.Collections/Sets/ISetExtensions.cs, 50
- ./csharp/Platform.Collections/Sets/SetFiller.cs, 51
- ./csharp/Platform.Collections/Stacks/DefaultStack.cs, 52
- ./csharp/Platform.Collections/Stacks/IStack.cs, 52
- ./csharp/Platform.Collections/Stacks/IStackExtensions.cs, 53
- ./csharp/Platform.Collections/Stacks/IStackFactory.cs, 53
- ./csharp/Platform.Collections/Stacks/StackExtensions.cs, 53
- ./csharp/Platform.Collections/StringExtensions.cs, 53
- ./csharp/Platform.Collections/Trees/Node.cs, 54