```
LinksPlatform's Platform Collections Class Library
     ./csharp/Platform. Collections/Arrays/ArrayFiller[TElement,\ TReturnConstant].cs
   using System.Collections.Generic;
   using System.Runtime.CompilerServices;
   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
   namespace Platform.Collections.Arrays
6
       public class ArrayFiller<TElement, TReturnConstant> : ArrayFiller<TElement>
9
           protected readonly TReturnConstant _returnConstant;
10
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
           public ArrayFiller(TElement[] array, long offset, TReturnConstant returnConstant) :
13
            → base(array, offset) => _returnConstant = returnConstant;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
15
           public ArrayFiller(TElement[] array, TReturnConstant returnConstant) : this(array, 0,
16
            → returnConstant) { }
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
18
           public TReturnConstant AddAndReturnConstant(TElement element) =>
19
                _array.AddAndReturnConstant(ref _position, element, _returnConstant);
20
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
21
           public TReturnConstant AddFirstAndReturnConstant(IList<TElement> elements) =>
                _array.AddFirstAndReturnConstant(ref _position, elements, _returnConstant);
23
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
24
           public TReturnConstant AddAllAndReturnConstant(IList<TElement> elements) =>
            _ array.AddAllAndReturnConstant(ref _position, elements, _returnConstant);
26
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
           public TReturnConstant AddSkipFirstAndReturnConstant(IList<TElement> elements) =>
            _ array.AddSkipFirstAndReturnConstant(ref _position, elements, _returnConstant);
       }
29
30
    ./csharp/Platform.Collections/Arrays/ArrayFiller[TElement].cs
1.2
   using System.Collections.Generic;
   using System.Runtime.CompilerServices;
   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
   namespace Platform.Collections.Arrays
6
       public class ArrayFiller<TElement>
9
           protected readonly TElement[] _array;
protected long _position;
10
12
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
           public ArrayFiller(TElement[] array, long offset)
14
15
                _array = array
16
                _position = offset;
17
            }
19
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
20
           public ArrayFiller(TElement[] array) : this(array, 0) { }
21
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
23
           public void Add(TElement element) => _array[_position++] = element;
24
25
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
26
           public bool AddAndReturnTrue(TElement element) => _array.AddAndReturnConstant(ref
               _position, element, true);
28
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
29
            public bool AddFirstAndReturnTrue(IList<TElement> elements) =>
30
               _array.AddFirstAndReturnConstant(ref _position, elements, true);
31
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
           public bool AddAllAndReturnTrue(IList<TElement> elements) =>
                _array.AddAllAndReturnConstant(ref _position, elements, true);
34
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
public bool AddSkipFirstAndReturnTrue(IList<TElement> elements) =>
36
                _array.AddSkipFirstAndReturnConstant(ref _position, elements, true);
       }
37
   }
38
     ./csharp/Platform.Collections/Arrays/ArrayPool.cs
1.3
   using System.Runtime.CompilerServices;
2
   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
   namespace Platform.Collections.Arrays
5
6
       public static class ArrayPool
            public static readonly int DefaultSizesAmount = 512;
            public static readonly int DefaultMaxArraysPerSize = 32;
10
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
12
            public static T[] Allocate<T>(long size) => ArrayPool<T>.ThreadInstance.Allocate(size);
13
14
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
15
            public static void Free<T>(T[] array) => ArrayPool<T>.ThreadInstance.Free(array);
16
       }
17
   }
18
     ./csharp/Platform.Collections/Arrays/ArrayPool[T].cs
1.4
   using System;
   using System.Collections.Generic;
   using System.Runtime.CompilerServices;
3
   using Platform.Disposables;
   using Platform.Collections.Stacks;
5
   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
   namespace Platform.Collections.Arrays
9
10
        /// <remarks>
11
       /// Original idea from
12
           http://geekswithblogs.net/blackrob/archive/2014/12/18/array-pooling-in-csharp.aspx
       /// </remarks>
13
       public class ArrayPool<T>
14
15
            // May be use Default class for that later.
16
            [ThreadStatic]
17
            private static ArrayPool<T> _threadInstance;
18
            internal static ArrayPool<T> ThreadInstance => _threadInstance ?? (_threadInstance = new
            → ArrayPool<T>());
20
            private readonly int _maxArraysPerSize;
21
            private readonly Dictionary<long, Stack<T[]>> _pool = new Dictionary<long,</pre>
22
               Stack<T[]>>(ArrayPool.DefaultSizesAmount);
23
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
24
            public ArrayPool(int maxArraysPerSize) => _maxArraysPerSize = maxArraysPerSize;
25
26
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public ArrayPool() : this(ArrayPool.DefaultMaxArraysPerSize) { }
29
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
30
            public Disposable<T[] > AllocateDisposable(long size) => (Allocate(size), Free);
31
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public Disposable<T[]> Resize(Disposable<T[]> source, long size)
34
35
                var destination = AllocateDisposable(size);
36
                T[] sourceArray = source;
37
                if (!sourceArray.IsNullOrEmpty())
38
39
                    T[] destinationArray = destination;
                    Array.Copy(sourceArray, destinationArray, size < sourceArray.LongLength ? size :
41

→ sourceArray.LongLength);
                    source.Dispose();
42
43
                return destination;
44
            }
45
46
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
47
            public virtual void Clear() => _pool.Clear();
49
```

```
[MethodImpl(MethodImplOptions.AggressiveInlining)]
50
            public virtual T[] Allocate(long size) => size <= OL ? Array.Empty<T>() :
                _pool.GetOrDefault(size)?.PopOrDefault() ?? new T[size];
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
5.3
            public virtual void Free(T[] array)
55
                if (array.IsNullOrEmpty())
56
                    return;
58
                }
59
                var stack = _pool.GetOrAdd(array.LongLength, size => new

    Stack<T[]>(_maxArraysPerSize));
                if (stack.Count == _maxArraysPerSize) // Stack is full
61
                {
62
63
                    return;
                }
64
                stack.Push(array);
65
            }
66
       }
67
68
     ./csharp/Platform.Collections/Arrays/ArrayString.cs
   using System.Runtime.CompilerServices;
   using Platform.Collections.Segments;
3
   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
   namespace Platform.Collections.Arrays
7
   {
       public class ArrayString<T> : Segment<T>
9
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
10
            public ArrayString(int length) : base(new T[length], 0, length) { }
11
12
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
13
            public ArrayString(T[] array) : base(array, 0, array.Length) { }
14
15
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
16
            public ArrayString(T[] array, int length) : base(array, 0, length) { }
       }
18
19
     ./csharp/Platform.Collections/Arrays/CharArrayExtensions.cs
1.6
   using System.Runtime.CompilerServices;
   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
   namespace Platform.Collections.Arrays
5
       public static unsafe class CharArrayExtensions
            /// <remarks>
            /// Based on https://github.com/Microsoft/referencesource/blob/3b1eaf5203992df69de44c783
10
                a3eda37d3d4cd10/mscorlib/system/string.cs#L833
            /// </remarks>
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
12
            public static int GenerateHashCode(this char[] array, int offset, int length)
13
                var hashSeed = 5381;
15
                var hashAccumulator = hashSeed;
16
                fixed (char* arrayPointer = &array[offset])
17
18
                    for (char* charPointer = arrayPointer, last = charPointer + length; charPointer
                        < last; charPointer++)
                    {
20
                        hashAccumulator = (hashAccumulator << 5) + hashAccumulator ^ *charPointer;
21
2.3
                return hashAccumulator + (hashSeed * 1566083941);
            }
26
27
            /// <remarks>
            /// Based on https://github.com/Microsoft/referencesource/blob/3b1eaf5203992df69de44c783
                a3eda37d3d4cd10/mscorlib/system/string.cs#L364
            /// </remarks>
29
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
30
            public static bool ContentEqualTo(this char[] left, int leftOffset, int length, char[]

→ right, int rightOffset)
```

```
32
                fixed (char* leftPointer = &left[leftOffset])
34
                    fixed (char* rightPointer = &right[rightOffset])
35
                         char* leftPointerCopy = leftPointer, rightPointerCopy = rightPointer;
37
                         if (!CheckArraysMainPartForEquality(ref leftPointerCopy, ref
38
                             rightPointerCopy, ref length))
                             return false;
40
41
                         CheckArraysRemainderForEquality(ref leftPointerCopy, ref rightPointerCopy,

→ ref length);
                        return length <= 0;</pre>
43
                    }
                }
45
            }
46
47
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
48
            private static bool CheckArraysMainPartForEquality(ref char* left, ref char* right, ref
49
                int length)
                while (length >= 10)
51
52
                    if ((*(int*)left != *(int*)right)
5.3
                      | | (*(int*)(left + 2) != *(int*)(right + 2))
                     || (*(int*)(left + 4) != *(int*)(right + 4))
55
                         (*(int*)(left + 6) != *(int*)(right + 6))
56
                     | | (*(int*)(left + 8) != *(int*)(right + 8)))
                    {
58
                        return false;
                    left += 10;
61
                    right += 10;
62
                    length -= 10;
64
65
                return true;
            }
66
67
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            private static void CheckArraysRemainderForEquality(ref char* left, ref char* right, ref
69
               int length)
70
                  / This depends on the fact that the String objects are
71
                // always zero terminated and that the terminating zero is not included
72
                // in the length. For odd string sizes, the last compare will include
73
                // the zero terminator.
                while (length > 0)
76
                    if (*(int*)left != *(int*)right)
77
                        break;
79
                    left += 2:
81
                    right += 2
82
                    length -= 2;
83
                }
            }
85
       }
86
87
     ./csharp/Platform.Collections/Arrays/GenericArrayExtensions.cs
1.7
   using System;
1
         System.Collections.Generic;
2
   using
   using System.Runtime.CompilerServices;
4
   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
   namespace Platform.Collections.Arrays
        public static class GenericArrayExtensions
9
10
            /// <summary>
11
            /// <para>Checks if an array exists, if so, checks the array length using the index
12
               variable type int, and if the array length is greater than the index - return
            \hookrightarrow
               array[index], otherwise - default value.</para>
            /// <para>Проверяется, существует ли массив, если да – идет проверка длины массива с
            🛶 помощью переменной index, и если длина массива больше индекса - возвращается
                array[index], иначе - значение по умолчанию.</para>
```

```
/// </summary>
14
           /// <typeparam name="T"><para>Array variable type.</para><para>Тип переменной
15
               массива.</para></typeparam>
           /// <param name="array"><para>Array that will participate in
               verification.</para><para>Массив который будет учавствовать в
            \hookrightarrow
               проверке.</para></param>
           /// <param name="index"><para>Number type int to compare.</para><para>Число типа int для
               сравнения.</para></param>
           /// <returns><para>Array element or default value.</para><para>Элемент массива или же
               значение по умолчанию.</para></returns>
           [MethodImpl(MethodImplOptions.AggressiveInlining)]
           public static T GetElementOrDefault<T>(this T[] array, int index) => array != null &&
20
               array.Length > index ? array[index] : default;
           /// <summary>
           /// <para>Checks whether the array exists, if so, checks the array length using the
              index variable type long, and if the array length is greater than the index - return
               array[index], otherwise - default value.</para>
           /// <para>Проверяется, существует ли массив, если да – идет проверка длины массива с
            🛶 помощью переменной index, и если длина массива больше индекса - возвращается
               array[index], иначе - значение по умолчанию.</para>
           /// </summary>
25
           /// <typeparam name="T"><para>Array variable type.</para><para>Тип переменной
26
               массива.</para></typeparam>
           /// <param name="array"><para>Array that will participate in
               verification.</para><para>Массив который будет учавствовать в
            \hookrightarrow
               проверке.</para></param>
           /// <param name="index"><para>Number type long to compare.</para><para>Число типа long
               для сравнения.</para></param>
           /// <returns><para>Array element or default value.</para><para>Элемент массива или же
29
               значение по умолчанию.</para></returns>
           [MethodImpl(MethodImplOptions.AggressiveInlining)]
30
           public static T GetElementOrDefault<T>(this T[] array, long index) => array != null &&
               array.LongLength > index ? array[index] : default;
32
           /// <summary>
33
           /// <para>Checks whether the array exist, if so, checks the array length using the index
               varible type int, and if the array length is greater than the index, set the element
               variable to array[index] and return true.
           /// <para>Проверяется, существует ли массив, если да, то идет проверка длины массива с
               помощью переменной index типа int, и если длина массива больше значения index,
               устанавливается значение переменной element - array[index] и возвращается
               true.</para>
           /// </summary>
36
           /// <typeparam name="T"><para>Array variable type.</para><para>Тип переменной
37
               массива.</para></typeparam>
           /// <param name="array"><para>Array that will participate in
               verification.</para><para>Массив который будет учавствовать в
               проверке.</para></param>
           /// <param name="index"><para>Number type int to compare.</para><para>Число типа int для
               сравнения.</para></param>
           /// <param name="element"><para>Passing the argument by reference, if successful, it
40
               will take the value array[index] otherwise default value.</para>Передается
               аргумент по ссылке, в случае успеха он примет значение array[index] в противном
               случае значение по умолчанию.</para></param>
           /// <returns><para>True if successful otherwise false.</para><para>True в случае успеха,
               в противном случае false</para></returns>
           [MethodImpl(MethodImplOptions.AggressiveInlining)]
           public static bool TryGetElement<T>(this T[] array, int index, out T element)
43
44
               if (array != null && array.Length > index)
45
46
                   element = array[index];
47
                   return true;
               }
49
               else
               {
51
                    element = default;
                   return false;
53
54
           }
56
           /// <summary>
           /// <para>Checks whether the array exist, if so, checks the array length using the
               index varible type long, and if the array length is greater than the index, set the
               element variable to array[index] and return true.</para>
```

```
/// <para>Проверяется, существует ли массив, если да, то идет проверка длины массива с
5.9
                помощью переменной index типа long, и если длина массива больше значения index,
                устанавливается значение переменной element - array[index] и возвращается
                true.</para>
            /// </summary>
            /// <typeparam name="T"><para>Array variable type.</para><para>Тип переменной
61
               массива.</para></typeparam>
            /// <param name="array"><para>Array that will participate in
62
                verification.</para><para>Массив который будет учавствовать в
                проверке.</para></param>
            /// <param name="index"><para>Number type long to compare.</para><para>Число типа long
                для сравнения.</para></param>
            /// <param name="element"><para>Passing the argument by reference, if successful, it
64
                will take the value array[index] otherwise default value.</para><para>Передаём
                аргумент по ссылке, в случае успеха он примет значение array[index] в противном
                случае значение по умолчанию.</para></param>
            /// <returns><para>True if successful otherwise false.</para><para>True в случае успеха,
6.5
             → в противном случае false</para></returns>
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
66
            public static bool TryGetElement<T>(this T[] array, long index, out T element)
67
68
                if (array != null && array.LongLength > index)
69
                    element = array[index];
7.1
72
                    return true;
                }
73
                else
74
75
                    element = default;
76
                    return false;
77
                }
78
            }
80
            /// <summary>
            /// <para>Copying of elements from one array to another array.</para>
82
            /// <para>Копируется элементы из одного массива в другой массив.</para>
83
            /// </summary>
84
            /// <typeparam name="T"><para>Array variable type.</para><para>Тип переменной

→ массива.</para></typeparam>

            /// <param name="array"><para>The array to copy.</para><para>Массив который необходимо
               скопировать.</para></param>
            /// <returns><para>Copy of the array.</para><para>Копию массива.</para></returns>
87
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
88
            public static T[] Clone<T>(this T[] array)
89
90
                var copy = new T[array.LongLength];
91
                Array.Copy(array, OL, copy, OL, array.LongLength);
                return copy;
93
94
95
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
96
            public static IList<T> ShiftRight<T>(this T[] array) => array.ShiftRight(1L);
98
            /// <para>Extending the array boundaries to shift elements and then copying it, but with
100
                the condition that shift > 0. If shift = = 0, the extension will not occur, but
               cloning will still be applied. If shift < 0, a NotImplementedException is
                thrown.</para>
            /// <para>Pacширение границ массива на shift элементов и последующее его копирование, но
101
               с условием что shift > 0. Если же shift == 0 - расширение не произойдет , но клонирование все равно применится. Если shift < 0, выбросится исключение
                NotImplementedException.</para>
            /// </summary>
102
            /// <typeparam name="T"><para>Array variable type.</para><para>Тип переменной
103
               массива.</para></typeparam>
            /// <param name="array"><para>Array to expand Elements.</para><para>Maccив для
            → расширения элементов.</para></param>
            /// <param name="shift"><para>The number to expand the array</para><para>Число на
105

→ которое необходимо рассширить массив.
            /// <returns>
106
            /// <para>If the value of the shift variable is < 0, it returns a
107
                NotImplementedException exception. If shift = = 0, the array is cloned, but the
            \hookrightarrow
                extension will not be applied. Otherwise, if the value shift > 0, the length of the
                array is increased by the shift elements and the array is cloned.</para>
```

```
/// <para>Если значение переменной shift < 0, возвращается исключение
108
                NotImplementedException. Если shift = = 0, то массив клонируется, но расширение не
                применяется. В противном случае, если значение shift > 0, длина массива
                увеличивается на shift элементов и массив клонируется.</para>
            /// </returns>
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
110
            public static IList<T> ShiftRight<T>(this T[] array, long shift)
111
112
                if (shift < 0)</pre>
                {
114
                    throw new NotImplementedException();
115
                }
                if (shift == 0)
117
118
                    return array.Clone<T>();
119
                }
120
                else
121
                    var restrictions = new T[array.LongLength + shift];
123
                    Array.Copy(array, OL, restrictions, shift, array.LongLength);
124
                    return restrictions;
125
126
            }
127
128
            /// <summary>
129
            /// <para>One of the array values with index on variable position++ type int is passed
                to the element variable.</para>
            /// <para>Одно из значений массива с индексом переменной position++ типа int назначается
131
                в переменную element.</para>
            /// </summary>
132
            /// <param name="array"><para>An array whose specific value will be assigned to the
133
                element variable. </para><para>Массив, определенное значений которого присваивается
                переменной element</para></param>
            /// <param name="position"><para>Reference to a position in an array of int
                type.</para><para>Ссылка на позицию в массиве типа int.</para></param>
            /// <param name="element"><para>The variable which needs to be assigned a specific value
135
               from the array.</para><para>Переменная, которой нужно присвоить определенное
                значение из массива.</para></param>
            /// <typeparam name="T"><para>Array variable type.</para>Тип переменной
136
                массива. <para > </typeparam>
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
137
            public static void Add<T>(this T[] array, ref int position, T element) =>
            → array[position++] = element;
139
            /// <summary>
140
            /// <para>One of the array values with index on variable position++ type long is passed
                to the element variable.</para>
            /// <para>Одно из значений массива с индексом переменной possition++ типа long
142
               назначается в переменную element.</para>
            /// </summary>
143
            /// <param name="array"><para>An array whose specific value will be assigned to the
144
               element variable.</para><para>Maccив, определенное значений которого присваивается
                переменной element</para></param>
            /// <param name="position"><para>Reference to a position in an array of long
                type.</para><para>Ссылка на позицию в массиве типа long.</para></param>
            /// <param name="element"><para>The variable which needs to be assigned a specific value
146
                from the array.</para><para>Переменная, которой нужно присвоить определенное
                значение из массива.</para></param>
            /// <typeparam name="T"><para>Array variable type.</para>Тип переменной
147
                массива. <para></typeparam>
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static void Add<T>(this T[] array, ref long position, T element) =>
                array[position++] = element;
150
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static TReturnConstant AddAndReturnConstant<TElement, TReturnConstant>(this
152
                TElement[] array, ref long position, TElement element, TReturnConstant
                returnConstant)
153
                array.Add(ref position, element);
                return returnConstant;
155
            }
156
157
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
158
            public static void AddFirst<T>(this T[] array, ref long position, IList<T> elements) =>
             → array[position++] = elements[0];
```

```
160
             [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static TReturnConstant AddFirstAndReturnConstant<TElement, TReturnConstant>(this
162
                 TElement[] array, ref long position, IList<TElement> elements, TReturnConstant
                 returnConstant)
             {
163
                 array.AddFirst(ref position, elements);
                 return returnConstant;
165
166
167
             [MethodImpl(MethodImplOptions.AggressiveInlining)]
168
            public static TReturnConstant AddAllAndReturnConstant<TElement, TReturnConstant>(this
                 TElement[] array, ref long position, IList<TElement> elements, TReturnConstant
                 returnConstant)
             {
170
                 array.AddAll(ref position, elements);
171
172
                 return returnConstant;
             }
173
              /// <summarv>
175
             /// <para>Adding a collection of elements starting from a specific position.</para>
176
             /// <para>Добавление коллекции элементов начиная с определенной позиции.</para>
             /// </summary>
178
             /// <param name="array"><para>An array to which the collection of elements will be
179
                 added.</para><para>Maccuв в который будет добавлена коллекция
                 элементов.</para></param>
             /// <param name="position"><para>The position from which to start adding
180
                 elements.</para><para>Позиция с которой начнется добавление элементов.</para></param>
             /// <param name="elements"><para>Added all collection of elements to
                 array.</para><para>Добавляется вся коллекция элементов в массив. </para></param>
             /// <typeparam name="T"><para>Array variables type.</para><para>Тип переменной
182
                массива.</para></typeparam>
             [MethodImpl(MethodImplOptions.AggressiveInlining)]
183
             public static void AddAll<T>(this T[] array, ref long position, IList<T> elements)
184
                 for (var i = 0; i < elements.Count; i++)</pre>
186
                 {
187
                     array.Add(ref position, elements[i]);
                 }
189
             }
190
191
             [MethodImpl(MethodImplOptions.AggressiveInlining)]
192
            public static TReturnConstant AddSkipFirstAndReturnConstant<TElement,</pre>
                 TReturnConstant>(this TElement[] array, ref long position, IList<TElement> elements,
                 TReturnConstant returnConstant)
             {
194
                 array.AddSkipFirst(ref position, elements);
195
196
                 return returnConstant;
197
198
             [MethodImpl(MethodImplOptions.AggressiveInlining)]
199
            public static void AddSkipFirst<T>(this T[] array, ref long position, IList<T> elements)
200
                => array.AddSkipFirst(ref position, elements, 1);
201
             [MethodImpl(MethodImplOptions.AggressiveInlining)]
202
            public static void AddSkipFirst<T>(this T[] array, ref long position, IList<T> elements,
203
                 int skip)
204
                 for (var i = skip; i < elements.Count; i++)</pre>
205
                 {
206
                     array.Add(ref position, elements[i]);
207
             }
209
        }
210
     ./csharp/Platform.Collections/BitString.cs
    using System;
    using System.Collections.Concurrent;
 2
    using System.Collections.Generic;
    using System. Numerics:
    using System.Runtime.CompilerServices;
    using System.Threading.Tasks;
using Platform.Exceptions;
    using Platform.Ranges;
    // ReSharper disable ForCanBeConvertedToForeach
10
    #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
```

```
12
   namespace Platform.Collections
13
14
        /// <remarks>
        /// А что если хранить карту значений, где каждый бит будет означать присутствует ли блок в
16
           64 бит в массиве значений.
        /// 64 бита по 0 бит, будут означать отсутствие 64-x блоков по 64 бита. Т.е. упаковка 512
17
            байт в 8 байт.
        /// Подобный принцип можно применять и к 64-ём блокам и т.п. По сути это карта значений. С
           помощью которой можно быстро
        /// проверять есть ли значения непосредственно далее (ниже по уровню).
19
        /// Или как таблица виртуальной памяти где номер блока означает его присутствие и адрес.
20
        /// </remarks>
        public class BitString : IEquatable<BitString>
22
23
            private static readonly byte[][] _bitsSetIn16Bits;
24
            private long[]
                            _array;
25
            private long _length;
private long _minPositiveWord;
private long _maxPositiveWord;
27
2.8
            public bool this[long index]
30
31
32
                 [MethodImpl(MethodImplOptions.AggressiveInlining)]
33
                 get => Get(index);
                 [MethodImpl(MethodImplOptions.AggressiveInlining)]
34
                 set => Set(index, value);
35
            }
37
            public long Length
38
39
                 [MethodImpl(MethodImplOptions.AggressiveInlining)]
40
                 get => _length;
41
                 [MethodImpl(MethodImplOptions.AggressiveInlining)]
42
43
                 set
                 {
44
                     if (_length == value)
45
                     {
46
                         return;
47
48
                     Ensure.Always.ArgumentInRange(value, GetValidLengthRange(), nameof(Length));
49
                     // Currently we never shrink the array
50
                     if (value > _length)
                         var words = GetWordsCountFromIndex(value);
53
                         var oldWords = GetWordsCountFromIndex(_length);
54
55
                         if (words > _array.LongLength)
                         {
56
                              var copy = new long[words];
57
                              Array.Copy(_array, copy, _array.LongLength);
                              _array = copy;
59
                         }
                         else
61
                         {
62
                              // What is going on here?
                              Array.Clear(_array, (int)oldWords, (int)(words - oldWords));
64
65
                         // What is going on here?
66
                         var mask = (int)(_length % 64);
67
                         if (mask > 0)
68
69
                              _array[oldWords - 1] &= (1L << mask) - 1;
70
7.1
72
                     else
73
74
                         // Looks like minimum and maximum positive words are not updated
                         throw new NotImplementedException();
76
77
                     _length = value;
78
                 }
79
            }
80
81
            #region Constructors
82
83
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
84
            static BitString()
86
                 _bitsSetIn16Bits = new byte[65536][];
87
```

```
int i, c, k;
    byte bitIndex;
    for (i = 0; i < 65536; i++)
        // Calculating size of array (number of positive bits)
        for (c = 0, k = 1; k \le 65536; k \le 1)
            if ((i & k) == k)
            {
                 c++;
            }
        var array = new byte[c];
        // Adding positive bits indices into array
        for (bitIndex = 0, c = 0, k = 1; k \leq 65536; k \leq 1)
            if ((i & k) == k)
                array[c++] = bitIndex;
            bitIndex++;
        _bitsSetIn16Bits[i] = array;
    }
}
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public BitString(BitString other)
    Ensure.Always.ArgumentNotNull(other, nameof(other));
    _length = other._length;
    _array = new long[GetWordsCountFromIndex(_length)];
    _minPositiveWord = other._minPositiveWord;
     _maxPositiveWord = other._maxPositiveWord;
    Array.Copy(other._array, _array, _array.LongLength);
}
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public BitString(long length)
    Ensure.Always.ArgumentInRange(length, GetValidLengthRange(), nameof(length));
    _length = length;
     _array = new long[GetWordsCountFromIndex(_length)];
    MarkBordersAsAllBitsReset();
}
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public BitString(long length, bool defaultValue)
    : this(length)
    if (defaultValue)
    {
        SetAll();
    }
}
#endregion
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public BitString Not()
    for (var i = OL; i < _array.LongLength; i++)</pre>
         _array[i] = ~_array[i]
        RefreshBordersByWord(i);
    return this;
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public BitString ParallelNot()
    var threads = Environment.ProcessorCount / 2;
    if (threads <= 1)</pre>
        return Not();
    var partitioner = Partitioner.Create(OL, _array.LongLength, _array.LongLength /

→ threads);
```

89

90

92

93 94

95

96

98 99

100

102 103

104 105

106 107

108 109

111

112 113

114

115

117

118

119

120

121

122

 $\frac{123}{124}$

126

127

128

129

130

132 133

134

135

136

138

139

140

141

142

144

146

147 148

149 150

151

152 153

154 155 156

157

158

160

161 162 163

164

```
Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
        MaxDegreeOfParallelism = threads }, range =>
        var maximum = range.Item2;
        for (var i = range.Item1; i < maximum; i++)</pre>
            _array[i] = ~_array[i];
    });
    MarkBordersAsAllBitsSet();
    TryShrinkBorders();
    return this;
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public BitString VectorNot()
    if (!Vector.IsHardwareAccelerated || _array.LongLength >= int.MaxValue)
        return Not();
    var step = Vector<long>.Count;
    if (_array.Length < step)</pre>
    {
        return Not();
    VectorNotLoop(_array, step, 0, _array.Length);
    MarkBordersAsAllBitsSet();
    TryShrinkBorders();
    return this;
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public BitString ParallelVectorNot()
    var threads = Environment.ProcessorCount / 2;
    if (threads <= 1)</pre>
        return VectorNot();
    if (!Vector.IsHardwareAccelerated)
        return ParallelNot();
    var step = Vector<long>.Count;
    if (_array.Length < (step * threads))</pre>
        return VectorNot();
    }
    var partitioner = Partitioner.Create(0, _array.Length, _array.Length / threads);
    Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
       MaxDegreeOfParallelism = threads }, range => VectorNotLoop(_array, step,
       range.Item1, range.Item2));
    MarkBordersAsAllBitsSet();
    TryShrinkBorders();
    return this;
[MethodImpl(MethodImplOptions.AggressiveInlining)]
static private void VectorNotLoop(long[] array, int step, int start, int maximum)
    var i = start;
    var range = maximum - start - 1;
    var stop = range - (range % step);
    for (; i < stop; i += step)</pre>
        (~new Vector<long>(array, i)).CopyTo(array, i);
    for (; i < maximum; i++)</pre>
        array[i] = ~array[i];
    }
}
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public BitString And(BitString other)
    EnsureBitStringHasTheSameSize(other, nameof(other));
```

167

168

169

171 172

174

175

177 178

179

180

182 183

184 185

186

188

189 190

192

193

194 195 196

197

198 199

200

201 202

 $\frac{203}{204}$

205

207 208

209

 $\frac{210}{211}$

212

213

 $\frac{214}{215}$

216

217

218 219 220

221

222

224

225

226 227

229 230

 $\frac{231}{232}$

233

234

 $\frac{235}{236}$

237

238 239

```
GetCommonOuterBorders(this, other, out long from, out long to);
    var otherArray = other._array;
    for (var i = from; i <= to; i++)</pre>
         _array[i] &= otherArray[i];
        RefreshBordersByWord(i);
    return this;
}
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public BitString ParallelAnd(BitString other)
    var threads = Environment.ProcessorCount / 2;
    if (threads <= 1)</pre>
        return And(other);
    EnsureBitStringHasTheSameSize(other, nameof(other));
    GetCommonOuterBorders(this, other, out long from, out long to);
    var partitioner = Partitioner.Create(from, to + 1, (to - from) / threads);
    Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
        MaxDegreeOfParallelism = threads }, range =>
    {
        var maximum = range.Item2;
        for (var i = range.Item1; i < maximum; i++)</pre>
            _array[i] &= other._array[i];
        }
    });
    MarkBordersAsAllBitsSet();
    TryShrinkBorders();
    return this;
}
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public BitString VectorAnd(BitString other)
    if (!Vector.IsHardwareAccelerated || _array.LongLength >= int.MaxValue)
    {
        return And(other);
    var step = Vector<long>.Count;
    if (_array.Length < step)</pre>
    {
        return And(other);
    EnsureBitStringHasTheSameSize(other, nameof(other));
    GetCommonOuterBorders(this, other, out int from, out int to);
    VectorAndLoop(_array, other._array, step, from, to + 1);
    MarkBordersAsAllBitsSet();
    TryShrinkBorders();
    return this;
}
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public BitString ParallelVectorAnd(BitString other)
    var threads = Environment.ProcessorCount / 2;
    if (threads <= 1)</pre>
    {
        return VectorAnd(other);
    }
    if (!Vector.IsHardwareAccelerated)
    {
        return ParallelAnd(other);
    var step = Vector<long>.Count;
    if (_array.Length < (step * threads))</pre>
    {
        return VectorAnd(other);
    EnsureBitStringHasTheSameSize(other, nameof(other));
    GetCommonOuterBorders(this, other, out int from, out int to);
    var partitioner = Partitioner.Create(from, to + 1, (to - from) / threads);
    Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
        MaxDegreeOfParallelism = threads }, range => VectorAndLoop(_array, other._array,
        step, range.Item1, range.Item2));
```

242

243

245

 $\frac{246}{247}$

248

249 250

251

 $\frac{252}{253}$

254

255

 $\frac{257}{258}$

260

261

262

264

265

267

268

270

 $\frac{271}{272}$

273 274

275

276

278

279

280 281

282

283

284

285 286

287

288

289 290

291

293 294

296 297

299

300

301

302

303

304

306

307

308

309

310

312

313

```
MarkBordersAsAllBitsSet();
316
                 TryShrinkBorders();
                 return this;
318
             }
320
             [MethodImpl(MethodImplOptions.AggressiveInlining)]
321
             static private void VectorAndLoop(long[] array, long[] otherArray, int step, int start,
322
                 int maximum)
             {
323
                 var i = start;
324
                 var range = maximum - start - 1;
325
                 var stop = range - (range % step);
326
                 for (; i < stop; i += step)</pre>
327
328
                      (new Vector<long>(array, i) & new Vector<long>(otherArray, i)).CopyTo(array, i);
329
                 }
                 for (; i < maximum; i++)</pre>
331
332
                      array[i] &= otherArray[i];
333
                 }
334
             }
335
             [MethodImpl(MethodImplOptions.AggressiveInlining)]
337
             public BitString Or(BitString other)
338
339
                 EnsureBitStringHasTheSameSize(other, nameof(other));
340
                 GetCommonOuterBorders(this, other, out long from, out long to);
341
                 for (var i = from; i <= to; i++)</pre>
342
                       .array[i] |= other._array[i];
344
                      RefreshBordersByWord(i);
345
346
                 return this;
347
             }
348
349
             [MethodImpl(MethodImplOptions.AggressiveInlining)]
350
351
             public BitString ParallelOr(BitString other)
352
                 var threads = Environment.ProcessorCount / 2;
353
                 if (threads <= 1)</pre>
354
                 {
355
                      return Or(other);
356
357
                 EnsureBitStringHasTheSameSize(other, nameof(other));
                 GetCommonOuterBorders(this, other, out long from, out long to);
359
                 var partitioner = Partitioner.Create(from, to + 1, (to - from) / threads);
360
                 Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
361
                      MaxDegreeOfParallelism = threads }, range =>
                 {
362
                      var maximum = range.Item2;
363
364
                      for (var i = range.Item1; i < maximum; i++)</pre>
365
                          _array[i] |= other._array[i];
366
                      }
367
                 });
368
                 MarkBordersAsAllBitsSet();
369
                 TryShrinkBorders();
370
371
                 return this;
372
373
             [MethodImpl(MethodImplOptions.AggressiveInlining)]
374
             public BitString VectorOr(BitString other)
375
376
                 if (!Vector.IsHardwareAccelerated || _array.LongLength >= int.MaxValue)
377
                 {
378
379
                      return Or(other);
380
                 var step = Vector<long>.Count;
381
                 if (_array.Length < step)</pre>
382
                 {
383
                      return Or(other);
384
                 EnsureBitStringHasTheSameSize(other, nameof(other));
386
                 GetCommonOuterBorders(this, other, out int from, out int to);
387
                 VectorOrLoop(_array, other._array, step, from, to + 1);
389
                 MarkBordersAsAllBitsSet();
                 TryShrinkBorders();
390
                 return this;
```

```
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public BitString ParallelVectorOr(BitString other)
    var threads = Environment.ProcessorCount / 2;
    if (threads <= 1)</pre>
        return VectorOr(other);
    if (!Vector.IsHardwareAccelerated)
    {
        return ParallelOr(other);
    var step = Vector<long>.Count;
    if (_array.Length < (step * threads))</pre>
        return VectorOr(other);
    EnsureBitStringHasTheSameSize(other, nameof(other));
    GetCommonOuterBorders(this, other, out int from, out int to);
    var partitioner = Partitioner.Create(from, to + 1, (to - from) / threads);
    Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
    __ MaxDegreeOfParallelism = threads }, range => VectorOrLoop(_array, other._array,
        step, range.Item1, range.Item2));
    MarkBordersAsAllBitsSet();
    TryShrinkBorders();
    return this;
}
[MethodImpl(MethodImplOptions.AggressiveInlining)]
static private void VectorOrLoop(long[] array, long[] otherArray, int step, int start,
   int maximum)
    var i = start;
    var range = maximum - start - 1;
    var stop = range - (range % step);
    for (; i < stop; i += step)</pre>
        (new Vector<long>(array, i) | new Vector<long>(otherArray, i)).CopyTo(array, i);
    }
    for (; i < maximum; i++)</pre>
    ₹
        array[i] |= otherArray[i];
    }
}
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public BitString Xor(BitString other)
    EnsureBitStringHasTheSameSize(other, nameof(other));
    GetCommonOuterBorders(this, other, out long from, out long to);
    for (var i = from; i <= to; i++)</pre>
         _array[i] ^= other._array[i];
        RefreshBordersByWord(i);
    return this;
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public BitString ParallelXor(BitString other)
    var threads = Environment.ProcessorCount / 2;
    if (threads <= 1)</pre>
    {
        return Xor(other);
    EnsureBitStringHasTheSameSize(other, nameof(other));
    GetCommonOuterBorders(this, other, out long from, out long to);
    var partitioner = Partitioner.Create(from, to + 1, (to - from) / threads);
    Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
        MaxDegreeOfParallelism = threads }, range =>
        var maximum = range.Item2;
        for (var i = range.Item1; i < maximum; i++)</pre>
        {
            _array[i] ^= other._array[i];
```

394

396

397

398 399

400 401

402

403

404 405

406

407 408

409

411

412

413

414

416

417

418 419

420

421

422

423 424

426 427

429

430

431

433

434 435

436

438

439 440

442

443

444 445

446 447 448

450 451

452

453

454

455

457

458

459

460

461

463

```
});
                MarkBordersAsAllBitsSet();
                TryShrinkBorders();
                return this;
470
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public BitString VectorXor(BitString other)
                if (!Vector.IsHardwareAccelerated || _array.LongLength >= int.MaxValue)
                {
                    return Xor(other);
                }
                var step = Vector<long>.Count;
                if (_array.Length < step)</pre>
                    return Xor(other);
                EnsureBitStringHasTheSameSize(other, nameof(other));
                GetCommonOuterBorders(this, other, out int from, out int to);
                VectorXorLoop(_array, other._array, step, from, to + 1);
                MarkBordersAsAllBitsSet();
                TryShrinkBorders();
                return this;
            }
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public BitString ParallelVectorXor(BitString other)
494
                var threads = Environment.ProcessorCount / 2;
                if (threads <= 1)</pre>
                {
                     return VectorXor(other);
                }
                if (!Vector.IsHardwareAccelerated)
                    return ParallelXor(other);
                }
                var step = Vector<long>.Count;
                if (_array.Length < (step * threads))</pre>
506
                     return VectorXor(other);
                EnsureBitStringHasTheSameSize(other, nameof(other));
                GetCommonOuterBorders(this, other, out int from, out int to);
                var partitioner = Partitioner.Create(from, to + 1, (to - from) / threads);
                Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions {
                 MaxDegreeOfParallelism = threads }, range => VectorXorLoop(_array, other._array,

    step, range.Item1, range.Item2));
                MarkBordersAsAllBitsSet();
                TryShrinkBorders();
                return this;
            }
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            static private void VectorXorLoop(long[] array, long[] otherArray, int step, int start,
                int maximum)
                var i = start;
                var range = maximum - start - 1;
                var stop = range - (range % step);
524
                for (; i < stop; i += step)</pre>
                     (new Vector<long>(array, i) ^ new Vector<long>(otherArray, i)).CopyTo(array, i);
                }
                for (; i < maximum; i++)</pre>
                {
530
                     array[i] ^= otherArray[i];
                }
            }
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            private void RefreshBordersByWord(long wordIndex)
536
                if (_array[wordIndex] == 0)
                {
                     if (wordIndex == _minPositiveWord && wordIndex != _array.LongLength - 1)
```

468

469

471 472

473

474

476

477

478

479

480

482

483

485

486

487

489

490

491 492

493

495

496

497

498

499

500

501 502 503

504

505

507

508

510

511

513

514 515

516

517 518

519

522

523

525 526

527

528 529

531

532

533

535

537

539

```
{
            _minPositiveWord++;
           (wordIndex == _maxPositiveWord && wordIndex != 0)
        {
            _maxPositiveWord--;
        }
    }
    else
    {
           (wordIndex < _minPositiveWord)</pre>
        {
            _minPositiveWord = wordIndex;
           (wordIndex > _maxPositiveWord)
            _maxPositiveWord = wordIndex;
        }
    }
}
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public bool TryShrinkBorders()
    GetBorders(out long from, out long to);
    while (from <= to && _array[from] == 0)</pre>
        from++;
    }
    if (from > to)
    {
        MarkBordersAsAllBitsReset();
        return true;
    while (to >= from && _array[to] == 0)
        to--;
    if (to < from)
        MarkBordersAsAllBitsReset();
        return true;
    var bordersUpdated = from != _minPositiveWord || to != _maxPositiveWord;
    if (bordersUpdated)
        SetBorders(from, to);
    return bordersUpdated;
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public bool Get(long index)
    Ensure.Always.ArgumentInRange(index, GetValidIndexRange(), nameof(index));
    return (_array[GetWordIndexFromIndex(index)] & GetBitMaskFromIndex(index)) != 0;
}
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public void Set(long index, bool value)
    if (value)
    {
        Set(index);
    }
    else
    {
        Reset(index);
    }
}
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public void Set(long index)
    Ensure.Always.ArgumentInRange(index, GetValidIndexRange(), nameof(index));
    var wordIndex = GetWordIndexFromIndex(index);
    var mask = GetBitMaskFromIndex(index);
    _array[wordIndex] |= mask;
    RefreshBordersByWord(wordIndex);
```

542 543

545

546

547

548

550

551

552

553 554

555 556

558

559

560 561

562 563

564

565

566 567

568

569

570

571

573 574

575 576

578

579 580

581 582

583

584

585

587 588 589

590

592

593 594

595

596

597 598

599

601

602

603

604

605

607

608

609

610 611 612

613 614 615

616

617

```
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public void Reset(long index)
    Ensure.Always.ArgumentInRange(index, GetValidIndexRange(), nameof(index));
    var wordIndex = GetWordIndexFromIndex(index);
    var mask = GetBitMaskFromIndex(index);
    _array[wordIndex] &= ~mask;
    RefreshBordersByWord(wordIndex);
}
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public bool Add(long index)
    var wordIndex = GetWordIndexFromIndex(index);
    var mask = GetBitMaskFromIndex(index);
    if ((_array[wordIndex] & mask) == 0)
        _array[wordIndex] |= mask;
        RefreshBordersByWord(wordIndex);
        return true;
    }
    else
    {
        return false;
    }
}
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public void SetAll(bool value)
    if (value)
    {
        SetAll();
    }
    else
    {
        ResetAll();
    }
}
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public void SetAll()
    var words = GetWordsCountFromIndex(_length);
    for (var i = 0; i < words; i++)</pre>
        _array[i] = fillValue;
    MarkBordersAsAllBitsSet();
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public void ResetAll()
    const long fillValue = 0;
    GetBorders(out long from, out long to);
    for (var i = from; i <= to; i++)</pre>
    {
        _array[i] = fillValue;
    MarkBordersAsAllBitsReset();
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public List<long> GetSetIndices()
    var result = new List<long>();
    GetBorders(out long from, out long to);
    for (var i = from; i <= to; i++)</pre>
        var word = _array[i];
        if (word != 0)
            AppendAllSetBitIndices(result, i, word);
    }
```

622

624

625

626

627

628

629

630 631 632

633

634

635

637 638

639

 $640 \\ 641$

642

643

645

646

647 648

649

650 651

653

654

655

656

657

659

 $660 \\ 661$

662

663 664

665

666 667

668

669 670

672

674

675 676

677

678

679

680

681 682 683

684 685

686

687 688

689

690

691 692

693

694 695

```
return result;
}
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public List<ulong> GetSetUInt64Indices()
    var result = new List<ulong>();
    GetBorders(out ulong from, out ulong to);
    for (var i = from; i <= to; i++)</pre>
    {
        var word = _array[i];
        if (word != 0)
             AppendAllSetBitIndices(result, i, word);
    return result;
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public long GetFirstSetBitIndex()
    var i = _minPositiveWord;
var word = _array[i];
if (word != 0)
    {
        return GetFirstSetBitForWord(i, word);
    return -1;
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public long GetLastSetBitIndex()
    var i = _maxPositiveWord;
var word = _array[i];
    if (word != 0)
        return GetLastSetBitForWord(i, word);
    return -1;
}
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public long CountSetBits()
    var total = 0L;
    GetBorders(out long from, out long to);
    for (var i = from; i <= to; i++)</pre>
        var word = _array[i];
        if (word != 0)
             total += CountSetBitsForWord(word);
    return total;
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public bool HaveCommonBits(BitString other)
    EnsureBitStringHasTheSameSize(other, nameof(other));
    GetCommonInnerBorders(this, other, out long from, out long to);
    var otherArray = other._array;
    for (var i = from; i <= to; i++)</pre>
    {
        var left = _array[i];
        var right = otherArray[i];
        if (left != 0 && right != 0 && (left & right) != 0)
             return true;
    return false;
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public long CountCommonBits(BitString other)
```

701

703 704

705

706

707

708

709

710 711 712

713

715 716 717

718

719 720

721 722 723

724

725 726 727

728 729

730

732

733 734

735 736

738

739

 $740 \\ 741$

743 744

745

746

747

749

750 751

752 753 754

755 756 757

758

759 760

761

762

763

764

766

767

768 769

770

772 773

774 775

776

```
EnsureBitStringHasTheSameSize(other, nameof(other));
    GetCommonInnerBorders(this, other, out long from, out long to);
    var total = 0L;
    var otherArray = other._array;
    for (var i = from; i <= to; i++)</pre>
        var left = _array[i];
        var right = otherArray[i];
        var combined = left & right;
        if (combined != 0)
            total += CountSetBitsForWord(combined);
    return total;
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public List<long> GetCommonIndices(BitString other)
    EnsureBitStringHasTheSameSize(other, nameof(other));
    GetCommonInnerBorders(this, other, out long from, out long to);
    var result = new List<long>();
    var otherArray = other._array;
    for (var i = from; i <= to; i++)
        var left = _array[i];
        var right = otherArray[i];
        var combined = left & right;
        if (combined != 0)
             AppendAllSetBitIndices(result, i, combined);
    return result;
}
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public List<ulong> GetCommonUInt64Indices(BitString other)
    EnsureBitStringHasTheSameSize(other, nameof(other));
    GetCommonBorders(this, other, out ulong from, out ulong to);
    var result = new List<ulong>();
    var otherArray = other._array;
    for (var i = from; i <= to; i++)</pre>
        var left = _array[i];
        var right = otherArray[i];
        var combined = left & right;
           (combined != 0)
             AppendAllSetBitIndices(result, i, combined);
    return result;
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public long GetFirstCommonBitIndex(BitString other)
    {\tt Ensure Bit String Has The Same Size (other, name of (other));}
    GetCommonInnerBorders(this, other, out long from, out long to);
    var otherArray = other._array;
    for (var i = from; i <= to; i++)</pre>
        var left = _array[i];
        var right = otherArray[i];
        var combined = left & right;
        if (combined != 0)
        {
            return GetFirstSetBitForWord(i, combined);
    return -1;
[{\tt MethodImpl}({\tt MethodImpl}{\tt Options}. {\tt AggressiveInlining}) \, \rfloor \,
public long GetLastCommonBitIndex(BitString other)
```

780

782

783

785

786

787

788 789 790

791 792

793 794 795

796

797

799

800

802

803 804

805

806

807

808 809

810 811 812

813

814 815

816

817

819

820

822 823

824

825

826

827

828 829

830 831 832

833 834 835

836

837 838

839

840

841

842

844

845

846

847

848

850 851 852

853

855

```
EnsureBitStringHasTheSameSize(other, nameof(other));
    GetCommonInnerBorders(this, other, out long from, out long to);
    var otherArray = other._array;
    for (var i = to; i >= from; i--)
        var left = _array[i];
var right = otherArray[i];
        var combined = left & right;
        if (combined != 0)
            return GetLastSetBitForWord(i, combined);
    }
    return -1;
}
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public override bool Equals(object obj) => obj is BitString @string ? Equals(@string) :

    false:

[MethodImpl(MethodImplOptions.AggressiveInlining)]
public bool Equals(BitString other)
    if (_length != other._length)
    {
        return false;
    var otherArray = other._array;
    if (_array.Length != otherArray.Length)
    {
        return false;
    }
      (_minPositiveWord != other._minPositiveWord)
    {
        return false;
    if (_maxPositiveWord != other._maxPositiveWord)
    {
        return false;
    GetCommonBorders(this, other, out ulong from, out ulong to);
    for (var i = from; i <= to; i++)</pre>
        if (_array[i] != otherArray[i])
            return false;
    return true;
}
[MethodImpl(MethodImplOptions.AggressiveInlining)]
private void EnsureBitStringHasTheSameSize(BitString other, string argumentName)
    Ensure.Always.ArgumentNotNull(other, argumentName);
    if (_length != other._length)
    {
        throw new ArgumentException("Bit string must be the same size.", argumentName);
    }
}
[MethodImpl(MethodImplOptions.AggressiveInlining)]
private void MarkBordersAsAllBitsReset() => SetBorders(_array.LongLength - 1, 0);
[MethodImpl(MethodImplOptions.AggressiveInlining)]
private void MarkBordersAsAllBitsSet() => SetBorders(0, _array.LongLength - 1);
[MethodImpl(MethodImplOptions.AggressiveInlining)]
private void GetBorders(out long from, out long to)
    from = _minPositiveWord;
    to = _maxPositiveWord;
[MethodImpl(MethodImplOptions.AggressiveInlining)]
private void GetBorders(out ulong from, out ulong to)
    from = (ulong)_minPositiveWord;
```

859

860

861 862

863

865

866 867

868 869

870

871

873

875

876

877

878 879

880

881

882 883

884

885

886

887

888

889

890

891

893

894

895 896

897

899

900 901

902 903

905

907

908

909 910

911

912

913

914

915

916 917

918

919

921

922 923

924

925

927

928 929 930

931

932 933

```
to = (ulong)_maxPositiveWord;
}
[MethodImpl(MethodImplOptions.AggressiveInlining)]
private void SetBorders(long from, long to)
    _minPositiveWord = from;
    _maxPositiveWord = to;
[MethodImpl(MethodImplOptions.AggressiveInlining)]
private Range<long> GetValidIndexRange() => (0, _length - 1);
[MethodImpl(MethodImplOptions.AggressiveInlining)]
private static Range<long> GetValidLengthRange() => (0, long.MaxValue);
[MethodImpl(MethodImplOptions.AggressiveInlining)]
private static void AppendAllSetBitIndices(List<ulong> result, ulong wordIndex, long
   wordValue)
{
    GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]

→ bits32to47, out byte[] bits48to63);
    AppendAllSetIndices(result, wordIndex, bits00to15, bits16to31, bits32to47,
    \rightarrow bits48to63);
}
[MethodImpl(MethodImplOptions.AggressiveInlining)]
private static void AppendAllSetBitIndices(List<long> result, long wordIndex, long
    wordValue)
{
    GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]

→ bits32to47, out byte[] bits48to63);
    AppendAllSetBitIndices(result, wordIndex, bits00to15, bits16to31, bits32to47,
    \rightarrow bits48to63);
[MethodImpl(MethodImplOptions.AggressiveInlining)]
private static long CountSetBitsForWord(long word)
    GetBits(word, out byte[] bits00to15, out byte[] bits16to31, out byte[] bits32to47,
    → out byte[] bits48to63)
    return bits00to15.LongLength + bits16to31.LongLength + bits32to47.LongLength +

→ bits48to63.LongLength;

[MethodImpl(MethodImplOptions.AggressiveInlining)]
private static long GetFirstSetBitForWord(long wordIndex, long wordValue)
    GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
    _{\rightarrow} bits32to47, out byte[] bits48to63);
    return GetFirstSetBit(wordIndex, bits00to15, bits16to31, bits32to47, bits48to63);
[MethodImpl(MethodImplOptions.AggressiveInlining)]
private static long GetLastSetBitForWord(long wordIndex, long wordValue)
    GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]

→ bits32to47, out byte[] bits48to63);
    return GetLastSetBit(wordIndex, bits00to15, bits16to31, bits32to47, bits48to63);
}
[MethodImpl(MethodImplOptions.AggressiveInlining)]
private static void AppendAllSetBitIndices(List<long> result, long i, byte[] bits00to15,
   byte[] bits16to31, byte[] bits32to47, byte[] bits48to63)
    for (var j = 0; j < bits00to15.Length; j++)</pre>
    {
        result.Add(bits00to15[j] + (i * 64));
    for (\text{var } j = 0; j < \text{bits} 16 \text{to} 31. \text{Length}; j++)
    {
        result.Add(bits16to31[j] + 16 + (i * 64));
    for (\text{var } j = 0; j < \text{bits} 32 \text{to} 47. \text{Length}; j++)
        result.Add(bits32to47[j] + 32 + (i * 64));
    for (\text{var } j = 0; j < \text{bits} 48 \text{to} 63. \text{Length}; j++)
```

937

939 940

941

942 943 944

945

946 947

948

949

951

952

953

955

956

958

959

960

961

962

963 964

965

966 967

968

970 971

972

973

975

976 977 978

979

980 981

982

983

985

987

988

989

991 992

994

995 996

997 998

```
1002
                       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)
                  for (\text{var } j = 0; j < \text{bits} 00 \text{to} 15. \text{Length}; j++)
1010
                  {
1011
                       result.Add(bits00to15[j] + (i * 64));
1012
1013
                  for (\text{var } j = 0; j < \text{bits16to31.Length}; j++)
1014
1015
                       result.Add(bits16to31[j] + 16UL + (i * 64));
1017
                  for (\text{var } j = 0; j < \text{bits} 32 \text{to} 47. \text{Length}; j++)
1018
1019
                       result.Add(bits32to47[j] + 32UL + (i * 64));
1020
1021
                  for (var j = 0; j < bits48to63.Length; j++)</pre>
1022
1023
                       result.Add(bits48to63[j] + 48UL + (i * 64));
1024
                  }
1025
              }
1026
1027
              [MethodImpl(MethodImplOptions.AggressiveInlining)]
1028
              private static long GetFirstSetBit(long i, byte[] bits00to15, byte[] bits16to31, byte[]
1029
                  bits32to47, byte[] bits48to63)
1030
                  if
                     (bits00to15.Length > 0)
1031
1032
                       return bits00to15[0] + (i * 64);
1033
1034
                      (bits16to31.Length > 0)
1035
                       return bits16to31[0] + 16 + (i * 64);
1037
1038
                      (bits32to47.Length > 0)
1039
                  {
1040
                       return bits32to47[0] + 32 + (i * 64);
1041
1042
1043
                  return bits48to63[0] + 48 + (i * 64);
              }
1044
1045
              [MethodImpl(MethodImplOptions.AggressiveInlining)]
1046
              private static long GetLastSetBit(long i, byte[] bits00to15, byte[] bits16to31, byte[]
1047
                  bits32to47, byte[] bits48to63)
1048
                  if (bits48to63.Length > 0)
                  {
1050
                       return bits48to63[bits48to63.Length - 1] + 48 + (i * 64);
1051
1052
                  if
                     (bits32to47.Length > 0)
1053
1054
                       return bits32to47[bits32to47.Length - 1] + 32 + (i * 64);
1055
                  }
                  i f
                     (bits16to31.Length > 0)
1057
                  {
1058
                       return bits16to31[bits16to31.Length - 1] + 16 + (i * 64);
1059
1060
                  return bits00to15[bits00to15.Length - 1] + (i * 64);
1061
              }
1062
1063
              [MethodImpl(MethodImplOptions.AggressiveInlining)]
1064
              private static void GetBits(long word, out byte[] bits00to15, out byte[] bits16to31, out
1065
                  byte[] bits32to47, out byte[] bits48to63)
              {
1066
                                 _bitsSetIn16Bits[word & Oxffffu]
                  bits00to15 =
1067
1068
                  bits16to31
                                  _bitsSetIn16Bits[(word >> 16) & 0xffffu];
                                  _bitsSetIn16Bits[(word >> 32) & 0xffffu]
                  bits32to47 =
                  bits48to63 = _bitsSetIn16Bits[(word >> 48) & Oxffffu];
1070
              }
1071
1072
              [MethodImpl(MethodImplOptions.AggressiveInlining)]
1073
              public static void GetCommonInnerBorders(BitString left, BitString right, out long from,
1074
                 out long to)
```

```
1075
                 from = Math.Max(left._minPositiveWord, right._minPositiveWord);
1076
1077
                 to = Math.Min(left._maxPositiveWord, right._maxPositiveWord);
1078
1079
             [MethodImpl(MethodImplOptions.AggressiveInlining)]
1080
             public static void GetCommonOuterBorders(BitString left, BitString right, out long from,
1081
                 out long to)
                 from = Math.Min(left._minPositiveWord, right._minPositiveWord);
1083
                 to = Math.Max(left._maxPositiveWord, right._maxPositiveWord);
1084
             }
1085
1086
             [MethodImpl(MethodImplOptions.AggressiveInlining)]
1087
             public static void GetCommonOuterBorders(BitString left, BitString right, out int from,
                 out int to)
             {
1089
                 from = (int)Math.Min(left._minPositiveWord, right._minPositiveWord);
1090
                 to = (int)Math.Max(left._maxPositiveWord, right._maxPositiveWord);
1091
1093
             [MethodImpl(MethodImplOptions.AggressiveInlining)]
             public static void GetCommonBorders(BitString left, BitString right, out ulong from, out
1095
                 ulong to)
1096
                 from = (ulong)Math.Max(left._minPositiveWord, right._minPositiveWord);
1097
                 to = (ulong)Math.Min(left._maxPositiveWord, right._maxPositiveWord);
1098
1099
1100
             [MethodImpl(MethodImplOptions.AggressiveInlining)]
1101
             public static long GetWordsCountFromIndex(long index) => (index + 63) / 64;
1102
1103
             [MethodImpl(MethodImplOptions.AggressiveInlining)]
1104
             public static long GetWordIndexFromIndex(long index) => index >> 6;
1105
1106
             [MethodImpl(MethodImplOptions.AggressiveInlining)]
1107
             public static long GetBitMaskFromIndex(long index) => 1L << (int)(index & 63);</pre>
1108
1109
1110
             [MethodImpl(MethodImplOptions.AggressiveInlining)]
             public override int GetHashCode() => base.GetHashCode();
1111
1112
             [MethodImpl(MethodImplOptions.AggressiveInlining)]
1113
             public override string ToString() => base.ToString();
1114
         }
1115
     }
1116
      ./csharp/Platform.Collections/BitStringExtensions.cs
    using System.Runtime.CompilerServices;
    using Platform.Random;
 3
     #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
 4
    namespace Platform.Collections
  6
         public static class BitStringExtensions
  9
             [MethodImpl(MethodImplOptions.AggressiveInlining)]
 10
             public static void SetRandomBits(this BitString @string)
 11
 12
                 for (var i = 0; i < @string.Length; i++)</pre>
 14
                      var value = RandomHelpers.Default.NextBoolean();
 15
                      @string.Set(i, value);
                 }
 17
             }
 18
         }
 19
    }
      ./csharp/Platform.Collections/Concurrent/ConcurrentQueueExtensions.cs
 1.10
    using System.Collections.Concurrent;
           System.Collections.Generic;
    using System.Runtime.CompilerServices;
 3
     #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
    namespace Platform.Collections.Concurrent
     {
         public static class ConcurrentQueueExtensions
 9
 10
```

```
[MethodImpl(MethodImplOptions.AggressiveInlining)]
11
            public static IEnumerable<T> DequeueAll<T>(this ConcurrentQueue<T> queue)
12
13
                while (queue.TryDequeue(out T item))
14
                    yield return item;
16
                }
17
            }
       }
19
20
      ./csharp/Platform.Collections/Concurrent/ConcurrentStackExtensions.cs
1.11
   using System.Collections.Concurrent;
   using System.Runtime.CompilerServices;
3
   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
   namespace Platform.Collections.Concurrent
7
       public static class ConcurrentStackExtensions
9
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
10
            public static T PopOrDefault<T>(this ConcurrentStack<T> stack) => stack.TryPop(out T
11
            → value) ? value : default;
12
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
13
            public static T PeekOrDefault<T>(this ConcurrentStack<T> stack) => stack.TryPeek(out T
               value) ? value : default;
       }
15
   }
16
      ./csharp/Platform.Collections/EnsureExtensions.cs
1.12
   using System;
   using System.Collections.Generic;
   using System.Diagnostics;
3
   using System.Runtime.CompilerServices;
   using Platform. Exceptions;
   using Platform.Exceptions.ExtensionRoots;
   #pragma warning disable IDE0060 // Remove unused parameter
   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
   namespace Platform.Collections
11
12
       public static class EnsureExtensions
13
14
            #region Always
15
16
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
17
            public static void ArgumentNotEmpty<T>(this EnsureAlwaysExtensionRoot root,
18
                ICollection<T> argument, string argumentName, string message)
19
                if (argument.IsNullOrEmpty())
2.0
21
                    throw new ArgumentException(message, argumentName);
22
                }
23
            }
24
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
26
            public static void ArgumentNotEmpty<T>(this EnsureAlwaysExtensionRoot root,
27
               ICollection<T> argument, string argumentName) => ArgumentNotEmpty(root, argument,
               argumentName, null);
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
29
            public static void ArgumentNotEmpty<T>(this EnsureAlwaysExtensionRoot root,
30
               ICollection<T> argument) => ArgumentNotEmpty(root, argument, null, null);
31
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
32
            public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureAlwaysExtensionRoot root,
33
               string argument, string argumentName, string message)
                if (string.IsNullOrWhiteSpace(argument))
3.5
                {
36
                    throw new ArgumentException(message, argumentName);
                }
38
            }
39
40
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
41
```

```
public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureAlwaysExtensionRoot root,
42
            string argument, string argumentName) => ArgumentNotEmptyAndNotWhiteSpace(root,
               argument, argumentName, null);
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
           public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureAlwaysExtensionRoot root,
            string argument) => ArgumentNotEmptyAndNotWhiteSpace(root, argument, null, null);
46
           #endregion
48
            #region OnDebug
49
50
            [Conditional("DEBUG")]
           public static void ArgumentNotEmpty<T>(this EnsureOnDebugExtensionRoot root,
52
               ICollection<T> argument, string argumentName, string message) =>
               Ensure.Always.ArgumentNotEmpty(argument, argumentName, message);
            [Conditional("DEBUG")]
54
           public static void ArgumentNotEmpty<T>(this EnsureOnDebugExtensionRoot root,
5.5
               ICollection<T> argument, string argumentName) =>
               Ensure.Always.ArgumentNotEmpty(argument, argumentName, null);
56
            [Conditional("DEBUG")]
           public static void ArgumentNotEmpty<T>(this EnsureOnDebugExtensionRoot root,

→ ICollection<T> argument) => Ensure.Always.ArgumentNotEmpty(argument, null, null);

59
            [Conditional("DEBUG")]
60
           public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureOnDebugExtensionRoot
               root, string argument, string argumentName, string message) =>
               Ensure.Always.ArgumentNotEmptyAndNotWhiteSpace(argument, argumentName, message);
            [Conditional("DEBUG")]
           public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureOnDebugExtensionRoot
64
               root, string argument, string argumentName) =>
               Ensure.Always.ArgumentNotEmptyAndNotWhiteSpace(argument, argumentName, null);
65
            [Conditional("DEBUG")]
           public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureOnDebugExtensionRoot
            root, string argument) => Ensure.Always.ArgumentNotEmptyAndNotWhiteSpace(argument,
            \rightarrow null, null);
           #endregion
69
       }
70
71
     ./csharp/Platform.Collections/ICollectionExtensions.cs
   using System.Collections.Generic;
   using System.Linq;
   using System.Runtime.CompilerServices;
   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
   namespace Platform.Collections
       public static class ICollectionExtensions
10
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
11
           public static bool IsNullOrEmpty<T>(this ICollection<T> collection) => collection ==
12
            → null | | collection.Count == 0;
13
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
           public static bool AllEqualToDefault<T>(this ICollection<T> collection)
16
                var equalityComparer = EqualityComparer<T>.Default;
17
                return collection.All(item => equalityComparer.Equals(item, default));
           }
19
       }
20
1.14 \quad ./csharp/Platform.Collections/IDictionaryExtensions.cs
   using System;
using System.Collections.Generic;
   using System.Runtime.CompilerServices;
   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
   namespace Platform.Collections
       public static class IDictionaryExtensions
```

```
10
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
11
            public static TValue GetOrDefault<TKey, TValue>(this IDictionary<TKey, TValue>
12
                dictionary, TKey key)
            {
13
                dictionary.TryGetValue(key, out TValue value);
14
                return value;
15
16
17
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
18
            public static TValue GetOrAdd<TKey, TValue>(this IDictionary<TKey, TValue> dictionary,
19
                TKey key, Func<TKey, TValue> valueFactory)
            {
20
                if (!dictionary.TryGetValue(key, out TValue value))
21
                {
22
23
                     value = valueFactory(key);
                    dictionary.Add(key, value);
24
                    return value;
                return value;
27
            }
       }
29
   }
30
      ./csharp/Platform.Collections/Lists/CharlListExtensions.cs
1.15
   using System.Collections.Generic;
   using System.Runtime.CompilerServices;
2
   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
   namespace Platform.Collections.Lists
6
7
        public static class CharIListExtensions
8
9
            /// <remarks>
10
            /// Based on https://github.com/Microsoft/referencesource/blob/3b1eaf5203992df69de44c783
11
                a3eda37d3d4cd10/mscorlib/system/string.cs#L833
            /// </remarks>
12
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
13
14
            public static int GenerateHashCode(this IList<char> list)
15
                var hashSeed = 5381;
                var hashAccumulator = hashSeed;
17
                for (var i = 0; i < list.Count; i++)</pre>
18
19
                    hashAccumulator = (hashAccumulator << 5) + hashAccumulator ^ list[i];</pre>
20
21
                return hashAccumulator + (hashSeed * 1566083941);
            }
23
24
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
25
            public static bool EqualTo(this IList<char> left, IList<char> right) =>
26
               left.EqualTo(right, ContentEqualTo);
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
28
            public static bool ContentEqualTo(this IList<char> left, IList<char> right)
29
30
                for (var i = left.Count - 1; i >= 0; --i)
31
32
                     if (left[i] != right[i])
33
                    {
                         return false;
35
36
37
                return true;
38
            }
39
        }
40
41
     ./csharp/Platform.Collections/Lists/IListComparer.cs
1.16
   using System.Collections.Generic;
   using System.Runtime.CompilerServices;
   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
   namespace Platform.Collections.Lists
6
   {
        public class IListComparer<T> : IComparer<IList<T>>
```

```
[MethodImpl(MethodImplOptions.AggressiveInlining)]
10
            public int Compare(IList<T> left, IList<T> right) => left.CompareTo(right);
11
       }
12
   }
13
      ./csharp/Platform. Collections/Lists/IL ist Equality Comparer.cs\\
1.17
   using System.Collections.Generic;
   using System.Runtime.CompilerServices;
3
   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
   namespace Platform.Collections.Lists
7
        public class IListEqualityComparer<T> : IEqualityComparer<IList<T>>
8
9
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
10
            public bool Equals(IList<T> left, IList<T> right) => left.EqualTo(right);
11
12
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
13
            public int GetHashCode(IList<T> list) => list.GenerateHashCode();
14
        }
15
   }
16
      ./csharp/Platform.Collections/Lists/IListExtensions.cs
1.18
   using System;
   using System. Collections. Generic;
   using System.Runtime.CompilerServices;
3
   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
   namespace Platform.Collections.Lists
7
        public static class IListExtensions
9
10
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static T GetElementOrDefault<T>(this IList<T> list, int index) => list != null &&
12
            → list.Count > index ? list[index] : default;
13
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static bool TryGetElement<T>(this IList<T> list, int index, out T element)
15
16
                if (list != null && list.Count > index)
18
                    element = list[index];
19
20
                    return true;
                }
21
                else
22
                {
23
                    element = default;
24
                    return false;
25
                }
            }
27
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
29
            public static bool AddAndReturnTrue<T>(this IList<T> list, T element)
30
31
                list.Add(element);
                return true;
33
            }
35
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
36
            public static bool AddFirstAndReturnTrue<T>(this IList<T> list, IList<T> elements)
37
38
                list.AddFirst(elements);
39
40
                return true;
41
42
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
43
            public static void AddFirst<T>(this IList<T> list, IList<T> elements) =>
44
               list.Add(elements[0]);
45
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
46
            public static bool AddAllAndReturnTrue<T>(this IList<T> list, IList<T> elements)
48
                list.AddAll(elements);
49
50
                return true;
51
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
53
```

```
public static void AddAll<T>(this IList<T> list, IList<T> elements)
    for (var i = 0; i < elements.Count; i++)</pre>
    {
        list.Add(elements[i]);
    }
}
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static bool AddSkipFirstAndReturnTrue<T>(this IList<T> list, IList<T> elements)
    list.AddSkipFirst(elements);
    return true;
}
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static void AddSkipFirst<T>(this IList<T> list, IList<T> elements) =>
   list.AddSkipFirst(elements, 1);
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static void AddSkipFirst<T>(this IList<T> list, IList<T> elements, int skip)
    for (var i = skip; i < elements.Count; i++)</pre>
        list.Add(elements[i]);
    }
}
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static int GetCountOrZero<T>(this IList<T> list) => list?.Count ?? 0;
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static bool EqualTo<T>(this IList<T> left, IList<T> right) => EqualTo(left,
→ right, ContentEqualTo);
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static bool EqualTo<T>(this IList<T> left, IList<T> right, Func<IList<T>,
   IList<T>, bool> contentEqualityComparer)
    if (ReferenceEquals(left, right))
    {
        return true;
    }
    var leftCount = left.GetCountOrZero();
    var rightCount = right.GetCountOrZero();
    if (leftCount == 0 && rightCount == 0)
    {
        return true;
    }
    if (leftCount == 0 || rightCount == 0 || leftCount != rightCount)
        return false;
    return contentEqualityComparer(left, right);
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static bool ContentEqualTo<T>(this IList<T> left, IList<T> right)
    var equalityComparer = EqualityComparer<T>.Default;
    for (var i = left.Count - 1; i >= 0; --i)
        if (!equalityComparer.Equals(left[i], right[i]))
            return false;
    return true;
}
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static T[] ToArray<T>(this IList<T> list, Func<T, bool> predicate)
    if (list == null)
    {
        return null;
    var result = new List<T>(list.Count);
    for (var i = 0; i < list.Count; i++)</pre>
```

56

57

59

60 61

62

63 64

66

68

7.0

72

73 74

76

77

78

79 80

82 83

84

85

86

87

88

90

91

93

94

96

97

99

100

102 103

104 105

107

108

110

111 112

113 114 115

116 117

118

120

121

122 123

124

125

126 127

128

```
if (predicate(list[i]))
            result.Add(list[i]);
    return result.ToArray();
}
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static T[] ToArray<T>(this IList<T> list)
    var array = new T[list.Count];
    list.CopyTo(array, 0);
    return array;
}
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static void ForEach<T>(this IList<T> list, Action<T> action)
    for (var i = 0; i < list.Count; i++)</pre>
    ₹
        action(list[i]);
    }
}
/// <remarks>
/// Based on http://stackoverflow.com/questions/263400/what-is-the-best-algorithm-for-an
    -overridden-system-object-gethashcode
/// </remarks>
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static int GenerateHashCode<T>(this IList<T> list)
    var hashAccumulator = 17;
    for (var i = 0; i < list.Count; i++)</pre>
        hashAccumulator = unchecked((hashAccumulator * 23) + list[i].GetHashCode());
    return hashAccumulator;
}
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static int CompareTo<T>(this IList<T> left, IList<T> right)
    var comparer = Comparer<T>.Default;
    var leftCount = left.GetCountOrZero();
    var rightCount = right.GetCountOrZero();
    var intermediateResult = leftCount.CompareTo(rightCount);
    for (var i = 0; intermediateResult == 0 && i < leftCount; i++)</pre>
    {
        intermediateResult = comparer.Compare(left[i], right[i]);
    return intermediateResult;
}
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static T[] SkipFirst<T>(this IList<T> list) => list.SkipFirst(1);
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static T[] SkipFirst<T>(this IList<T> list, int skip)
    if (list.IsNullOrEmpty() || list.Count <= skip)</pre>
    {
        return Array.Empty<T>();
    var result = new T[list.Count - skip];
    for (int r = skip, w = 0; r < list.Count; r++, w++)
    {
        result[w] = list[r];
    }
    return result;
}
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static IList<T> ShiftRight<T>(this IList<T> list) => list.ShiftRight(1);
[MethodImpl(MethodImplOptions.AggressiveInlining)]
public static IList<T> ShiftRight<T>(this IList<T> list, int shift)
```

132

133

135

136

137 138

139

140 141

142 143

144

145 146

147

148 149

150

151

153

154 155

156

157

158

159

160 161

162

163 164

165 166

167

168 169

170

172 173

174

175 176

177

178

179

181

182 183

184

186

187

188 189

190

192 193

194

195

196

198

199

200 201 202

 $\frac{203}{204}$

205

 $\frac{206}{207}$

```
if (shift < 0)</pre>
208
                      throw new NotImplementedException();
210
211
                 if (shift == 0)
                 {
213
                      return list.ToArray();
214
                 }
215
                 else
216
                 {
217
                      var result = new T[list.Count + shift];
218
                      for (int r = 0, w = shift; r < list.Count; r++, w++)
219
220
221
                          result[w] = list[r];
222
                      return result;
223
                 }
             }
225
         }
226
227
1.19
      ./csharp/Platform.Collections/Lists/ListFiller.cs
    using System.Collections.Generic;
using System.Runtime.CompilerServices;
    #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
 5
    namespace Platform.Collections.Lists
 6
         public class ListFiller<TElement, TReturnConstant>
 9
             protected readonly List<TElement> _list;
protected readonly TReturnConstant _returnConstant;
 10
11
12
             [MethodImpl(MethodImplOptions.AggressiveInlining)]
13
             public ListFiller(List<TElement> list, TReturnConstant returnConstant)
14
                 list = list;
16
                 _returnConstant = returnConstant;
17
18
19
             [MethodImpl(MethodImplOptions.AggressiveInlining)]
20
             public ListFiller(List<TElement> list) : this(list, default) { }
22
             [MethodImpl(MethodImplOptions.AggressiveInlining)]
23
             public void Add(TElement element) => _list.Add(element);
24
2.5
             [MethodImpl(MethodImplOptions.AggressiveInlining)]
             public bool AddAndReturnTrue(TElement element) => _list.AddAndReturnTrue(element);
27
             [MethodImpl(MethodImplOptions.AggressiveInlining)]
29
             public bool AddFirstAndReturnTrue(IList<TElement> elements) =>
30
             → _list.AddFirstAndReturnTrue(elements);
             [MethodImpl(MethodImplOptions.AggressiveInlining)]
32
             public bool AddAllAndReturnTrue(IList<TElement> elements) =>
33
                 _list.AddAllAndReturnTrue(elements);
             [MethodImpl(MethodImplOptions.AggressiveInlining)]
35
             public bool AddSkipFirstAndReturnTrue(IList<TElement> elements) =>
36
                 _list.AddSkipFirstAndReturnTrue(elements);
37
             [MethodImpl(MethodImplOptions.AggressiveInlining)]
38
             public TReturnConstant AddAndReturnConstant(TElement element)
39
40
                  list.Add(element);
41
                 return _returnConstant;
43
             [MethodImpl(MethodImplOptions.AggressiveInlining)]
45
             public TReturnConstant AddFirstAndReturnConstant(IList<TElement> elements)
46
47
                 _list.AddFirst(elements);
                 return _returnConstant;
49
5.1
             [MethodImpl(MethodImplOptions.AggressiveInlining)]
             public TReturnConstant AddAllAndReturnConstant(IList<TElement> elements)
53
54
```

```
_list.AddAll(elements);
5.5
                return _returnConstant;
56
            }
57
58
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
59
            public TReturnConstant AddSkipFirstAndReturnConstant(IList<TElement> elements)
60
61
                 _list.AddSkipFirst(elements);
62
                return _returnConstant;
            }
64
       }
65
66
      ./csharp/Platform.Collections/Segments/CharSegment.cs
1.20
   using System.Linq;
   using System.Collections.Generic;
   using System.Runtime.CompilerServices;
   using Platform.Collections.Arrays;
4
   using Platform.Collections.Lists;
   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
   namespace Platform.Collections.Segments
9
10
        public class CharSegment : Segment<char>
12
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
13
            public CharSegment(IList<char> @base, int offset, int length) : base(@base, offset,
14
               length) { }
15
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
16
            public override int GetHashCode()
17
18
                // Base can be not an array, but still IList<char>
19
                if (Base is char[] baseArray)
20
                {
21
                    return baseArray.GenerateHashCode(Offset, Length);
22
                }
23
                else
24
                {
25
                    return this.GenerateHashCode();
26
                }
27
            }
28
29
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
30
            public override bool Equals(Segment<char> other)
31
32
                bool contentEqualityComparer(IList<char> left, IList<char> right)
33
34
                    // Base can be not an array, but still IList<char>
                    if (Base is char[] baseArray && other.Base is char[] otherArray)
36
37
                         return baseArray.ContentEqualTo(Offset, Length, otherArray, other.Offset);
38
                    }
39
                    else
40
                    {
                         return left.ContentEqualTo(right);
42
43
44
                return this.EqualTo(other, contentEqualityComparer);
46
47
            public override bool Equals(object obj) => obj is Segment<char> charSegment ?
               Equals(charSegment) : false;
49
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
50
            public static implicit operator string(CharSegment segment)
5.1
52
                if (!(segment.Base is char[] array))
                {
54
                    array = segment.Base.ToArray();
55
56
                return new string(array, segment.Offset, segment.Length);
58
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
60
            public override string ToString() => this;
61
        }
62
   }
63
```

```
./csharp/Platform.Collections/Segments/Segment.cs
   using System;
   using System.Collections;
   using System.Collections.Generic;
   using System.Runtime.CompilerServices;
   using Platform.Collections.Arrays;
5
   using Platform.Collections.Lists;
   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
   namespace Platform.Collections.Segments
10
11
        public class Segment<T> : IEquatable<Segment<T>>, IList<T>
12
13
            public IList<T> Base
15
                 [MethodImpl(MethodImplOptions.AggressiveInlining)]
16
17
18
            public int Offset
20
                 [MethodImpl(MethodImplOptions.AggressiveInlining)]
21
22
23
            public int Length
24
25
                 [MethodImpl(MethodImplOptions.AggressiveInlining)]
26
27
                get;
            }
28
29
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
30
            public Segment(IList<T> @base, int offset, int length)
31
32
                Base = @base;
33
                Offset = offset;
34
35
                Length = length;
            }
36
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public override int GetHashCode() => this.GenerateHashCode();
39
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
41
            public virtual bool Equals(Segment<T> other) => this.EqualTo(other);
42
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
44
45
            public override bool Equals(object obj) => obj is Segment<T> other ? Equals(other) :
             \hookrightarrow false;
46
            #region IList
47
            public T this[int i]
49
50
                [MethodImpl(MethodImplOptions.AggressiveInlining)]
                get => Base[Offset + i];
                 [MethodImpl(MethodImplOptions.AggressiveInlining)]
53
                set => Base[Offset + i] = value;
54
            }
55
56
            public int Count
57
58
                 [MethodImpl(MethodImplOptions.AggressiveInlining)]
59
                get => Length;
60
            }
61
62
            public bool IsReadOnly
63
64
                 [MethodImpl(MethodImplOptions.AggressiveInlining)]
65
                get => true;
67
68
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
69
            public int IndexOf(T item)
70
71
                var index = Base.IndexOf(item);
72
                if (index >= Offset)
7.3
74
                     var actualIndex = index - Offset;
7.5
                     if (actualIndex < Length)</pre>
76
77
                     {
                         return actualIndex;
78
                     }
```

```
80
                 return -1;
81
82
83
             [MethodImpl(MethodImplOptions.AggressiveInlining)]
84
            public void Insert(int index, T item) => throw new NotSupportedException();
85
86
             [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public void RemoveAt(int index) => throw new NotSupportedException();
89
             [MethodImpl(MethodImplOptions.AggressiveInlining)]
90
            public void Add(T item) => throw new NotSupportedException();
91
             [MethodImpl(MethodImplOptions.AggressiveInlining)]
93
            public void Clear() => throw new NotSupportedException();
94
             [MethodImpl(MethodImplOptions.AggressiveInlining)]
96
            public bool Contains(T item) => IndexOf(item) >= 0;
97
98
             [MethodImpl(MethodImplOptions.AggressiveInlining)]
99
            public void CopyTo(T[] array, int arrayIndex)
100
                 for (var i = 0; i < Length; i++)</pre>
102
                 {
103
                     array.Add(ref arrayIndex, this[i]);
104
                 }
105
             }
106
107
             [MethodImpl(MethodImplOptions.AggressiveInlining)]
108
            public bool Remove(T item) => throw new NotSupportedException();
109
110
             [MethodImpl(MethodImplOptions.AggressiveInlining)]
111
            public IEnumerator<T> GetEnumerator()
112
113
                 for (var i = 0; i < Length; i++)
114
                 {
115
                     yield return this[i];
                 }
117
            }
118
119
             [MethodImpl(MethodImplOptions.AggressiveInlining)]
120
             IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
121
122
123
             #endregion
        }
124
    }
125
      ./csharp/Platform.Collections/Segments/Walkers/AllSegmentsWalkerBase.cs
    #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
 -1
 2
    namespace Platform.Collections.Segments.Walkers
 3
 4
        public abstract class AllSegmentsWalkerBase
            public static readonly int DefaultMinimumStringSegmentLength = 2;
        }
    }
      ./csharp/Platform.Collections/Segments/Walkers/AllSegmentsWalkerBase[T, TSegment].cs
    using System.Collections.Generic;
 2
    using System.Runtime.CompilerServices;
    #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
 4
    namespace Platform.Collections.Segments.Walkers
 6
 7
        public abstract class AllSegmentsWalkerBase<T, TSegment> : AllSegmentsWalkerBase
            where TSegment : Segment<T>
        {
10
            private readonly int _minimumStringSegmentLength;
11
12
             [MethodImpl(MethodImplOptions.AggressiveInlining)]
1.3
            protected AllSegmentsWalkerBase(int minimumStringSegmentLength) =>
             \  \   \   \   \  \, \_minimumStringSegmentLength = minimumStringSegmentLength;
             [MethodImpl(MethodImplOptions.AggressiveInlining)]
16
            protected AllSegmentsWalkerBase() : this(DefaultMinimumStringSegmentLength) { }
17
             [MethodImpl(MethodImplOptions.AggressiveInlining)]
19
            public virtual void WalkAll(IList<T> elements)
```

```
for (int offset = 0, maxOffset = elements.Count - _minimumStringSegmentLength;
                    offset <= maxOffset; offset++)
                    for (int length = _minimumStringSegmentLength, maxLength = elements.Count -
24
                        offset; length <= maxLength; length++)
25
                        Iteration(CreateSegment(elements, offset, length));
                    }
27
                }
28
            }
29
30
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
31
            protected abstract TSegment CreateSegment(IList<T> elements, int offset, int length);
33
34
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
           protected abstract void Iteration(TSegment segment);
35
       }
36
   }
37
     /csharp/Platform.Collections/Segments/Walkers/AllSegmentsWalkerBase|T|.cs
   using System.Collections.Generic;
   using System.Runtime.CompilerServices;
2
   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
   namespace Platform.Collections.Segments.Walkers
6
       public abstract class AllSegmentsWalkerBase<T> : AllSegmentsWalkerBase<T, Segment<T>>
9
10
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
           protected override Segment<T> CreateSegment(IList<T> elements, int offset, int length)
11
            → => new Segment<T>(elements, offset, length);
       }
12
   }
13
1.25
      ./csharp/Platform.Collections/Segments/Walkers/AllSegmentsWalkerExtensions.cs
   using System.Runtime.CompilerServices;
   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
   namespace Platform.Collections.Segments.Walkers
5
       public static class AllSegmentsWalkerExtensions
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
           public static void WalkAll(this AllSegmentsWalkerBase<char> walker, string @string) =>
10

→ walker.WalkAll(@string.ToCharArray());
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
           public static void WalkAll<TSegment>(this AllSegmentsWalkerBase<char, TSegment> walker,
13

→ string @string) where TSegment : Segment < char > = >
            → walker.WalkAll(@string.ToCharArray());
       }
14
   }
15
1.26
      ./ csharp/Platform. Collections/Segments/Walkers/DictionaryBasedDuplicateSegmentsWalkerBase [T, Segment Segments] \\
   using System;
   using System.Collections.Generic;
3
   using System.Runtime.CompilerServices;
   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
   namespace Platform.Collections.Segments.Walkers
8
       public abstract class DictionaryBasedDuplicateSegmentsWalkerBase<T, TSegment> :
9
           DuplicateSegmentsWalkerBase<T, TSegment>
10
           where TSegment : Segment<T>
11
           public static readonly bool DefaultResetDictionaryOnEachWalk;
12
13
            private readonly bool
                                   _resetDictionaryOnEachWalk;
14
            protected IDictionary<TSegment, long>Dictionary;
16
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
           protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<TSegment, long>
                dictionary, int minimumStringSegmentLength, bool resetDictionaryOnEachWalk)
                : base(minimumStringSegmentLength)
19
20
```

```
Dictionary = dictionary
                          _resetDictionaryOnEachWalk = resetDictionaryOnEachWalk;
22
2.3
                   [MethodImpl(MethodImplOptions.AggressiveInlining)]
25
                   protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<TSegment, long>
26
                         dictionary, int minimumStringSegmentLength) : this(dictionary,
                         minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
27
                   [MethodImpl(MethodImplOptions.AggressiveInlining)]
28
                   protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<TSegment, long>
                         dictionary) : this(dictionary, DefaultMinimumStringSegmentLength,
                         DefaultResetDictionaryOnEachWalk) { }
30
                   [MethodImpl(MethodImplOptions.AggressiveInlining)]
                   protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength,
                          bool resetDictionaryOnEachWalk) : this(resetDictionaryOnEachWalk ? null : new
                         Dictionary<TSegment, long>(), minimumStringSegmentLength, resetDictionaryOnEachWalk)
                         { }
                   [MethodImpl(MethodImplOptions.AggressiveInlining)]
                   \begin{picture}(c) \textbf{protected} & \textbf{DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength)} : \\ \textbf{DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength)} : \\ \textbf{DictionaryBasedDuplicateSegmentSegmentSegmentBase(int minimumStringSegmentLength)} : \\ \textbf{DictionaryBasedDuplicateSegmentSegmentBase(int minimumStringSegmentLength)} : \\ \textbf{DictionaryBasedDuplicateSegmentSegmentBase(int minimumStringSegmentLength)} : \\ \textbf{DictionaryBase(int minimumStringSegmentBase(int minimumStringS
35
                    this(minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
                   [MethodImpl(MethodImplOptions.AggressiveInlining)]
                   protected DictionaryBasedDuplicateSegmentsWalkerBase() :
38
                    this(DefaultMinimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
39
                   [MethodImpl(MethodImplOptions.AggressiveInlining)]
40
                   public override void WalkAll(IList<T> elements)
41
                          if (_resetDictionaryOnEachWalk)
                          {
44
                                var capacity = Math.Ceiling(Math.Pow(elements.Count, 2) / 2);
45
                                Dictionary = new Dictionary<TSegment, long>((int)capacity);
46
47
                          base.WalkAll(elements);
48
                   }
50
                   [MethodImpl(MethodImplOptions.AggressiveInlining)]
                   protected override long GetSegmentFrequency(TSegment segment) =>
                    → Dictionary.GetOrDefault(segment);
53
                   [MethodImpl(MethodImplOptions.AggressiveInlining)]
                   protected override void SetSegmentFrequency(TSegment segment, long frequency) =>
55
                    → Dictionary[segment] = frequency;
56
         ./csharp/Platform. Collections/Segments/Walkers/DictionaryBasedDuplicateSegmentsWalkerBase [T]. cs
1.27
     using System.Collections.Generic;
     using System.Runtime.CompilerServices;
      #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
 4
     namespace Platform.Collections.Segments.Walkers
 6
            public abstract class DictionaryBasedDuplicateSegmentsWalkerBase<T> :
                  DictionaryBasedDuplicateSegmentsWalkerBase<T, Segment<T>>
                   [MethodImpl(MethodImplOptions.AggressiveInlining)]
10
                   protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<Segment<T>, long>
11
                         dictionary, int minimumStringSegmentLength, bool resetDictionaryOnEachWalk) :
                         base(dictionary, minimumStringSegmentLength, resetDictionaryOnEachWalk) { }
                   [MethodImpl(MethodImplOptions.AggressiveInlining)]
                   protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<Segment<T>, long>
                         dictionary, int minimumStringSegmentLength) : base(dictionary
                         minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
                   [MethodImpl(MethodImplOptions.AggressiveInlining)]
16
                   protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<Segment<T>, long>
17
                         dictionary) : base(dictionary, DefaultMinimumStringSegmentLength,
                         DefaultResetDictionaryOnEachWalk) { }
                   [MethodImpl(MethodImplOptions.AggressiveInlining)]
19
```

```
protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength,
20
            bool resetDictionaryOnEachWalk) : base(minimumStringSegmentLength,
               resetDictionaryOnEachWalk) { }
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
           protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength) :
            → base(minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
24
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
25
            protected DictionaryBasedDuplicateSegmentsWalkerBase() :
            → base(DefaultMinimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
       }
27
28
1.28
      ./csharp/Platform.Collections/Segments/Walkers/DuplicateSegmentsWalkerBase[T, TSegment].cs
   using System.Runtime.CompilerServices;
   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
3
4
   namespace Platform.Collections.Segments.Walkers
5
   {
       public abstract class DuplicateSegmentsWalkerBase<T, TSegment> : AllSegmentsWalkerBase<T,
           TSegment>
        \hookrightarrow
           where TSegment : Segment<T>
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
10
           protected DuplicateSegmentsWalkerBase(int minimumStringSegmentLength) :
            → base(minimumStringSegmentLength) { }
12
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
13
            protected DuplicateSegmentsWalkerBase() : base(DefaultMinimumStringSegmentLength) { }
14
15
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
           protected override void Iteration(TSegment segment)
17
18
                var frequency = GetSegmentFrequency(segment);
19
                if (frequency == 1)
20
                {
21
                    OnDublicateFound(segment);
                SetSegmentFrequency(segment, frequency + 1);
24
            }
25
26
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
27
           protected abstract void OnDublicateFound(TSegment segment);
29
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
30
            protected abstract long GetSegmentFrequency(TSegment segment);
31
32
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
           protected abstract void SetSegmentFrequency(TSegment segment, long frequency);
34
       }
35
36
     ./csharp/Platform.Collections/Segments/Walkers/DuplicateSegmentsWalkerBase[T].cs
   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
1
2
   namespace Platform.Collections.Segments.Walkers
3
   ₹
4
       public abstract class DuplicateSegmentsWalkerBase<T> : DuplicateSegmentsWalkerBase<T,</pre>
5
           Segment<T>>
   }
      ./csharp/Platform.Collections/Sets/ISetExtensions.cs
   using System.Collections.Generic;
   using System.Runtime.CompilerServices;
3
   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
   namespace Platform.Collections.Sets
7
       public static class ISetExtensions
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
10
           public static void AddAndReturnVoid<T>(this ISet<T> set, T element) => set.Add(element);
11
```

```
[MethodImpl(MethodImplOptions.AggressiveInlining)]
13
            public static void RemoveAndReturnVoid<T>(this ISet<T> set, T element) =>
14
               set.Remove(element);
15
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
16
            public static bool AddAndReturnTrue<T>(this ISet<T> set, T element)
17
18
                set.Add(element);
19
20
                return true;
            }
21
22
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
23
            public static bool AddFirstAndReturnTrue<T>(this ISet<T> set, IList<T> elements)
24
26
                AddFirst(set, elements);
                return true;
            }
28
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
30
            public static void AddFirst<T>(this ISet<T> set, IList<T> elements) =>
31

    set.Add(elements[0]);
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
33
            public static bool AddAllAndReturnTrue<T>(this ISet<T> set, IList<T> elements)
34
35
                set.AddAll(elements);
36
                return true;
            }
39
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
40
            public static void AddAll<T>(this ISet<T> set, IList<T> elements)
41
42
                for (var i = 0; i < elements.Count; i++)</pre>
43
                {
44
                    set.Add(elements[i]);
45
                }
46
            }
48
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static bool AddSkipFirstAndReturnTrue<T>(this ISet<T> set, IList<T> elements)
50
51
                set.AddSkipFirst(elements);
52
53
                return true;
            }
54
55
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
56
            public static void AddSkipFirst<T>(this ISet<T> set, IList<T> elements) =>
57
               set.AddSkipFirst(elements, 1);
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
5.9
            public static void AddSkipFirst<T>(this ISet<T> set, IList<T> elements, int skip)
61
                for (var i = skip; i < elements.Count; i++)</pre>
62
63
                    set.Add(elements[i]);
64
                }
65
            }
66
67
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
68
            public static bool DoNotContains<T>(this ISet<T> set, T element) =>
69
               !set.Contains(element);
        }
70
   }
71
      ./csharp/Platform.Collections/Sets/SetFiller.cs
   using System.Collections.Generic;
   using System.Runtime.CompilerServices;
3
   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
   namespace Platform.Collections.Sets
        public class SetFiller<TElement, TReturnConstant>
8
            protected readonly ISet<TElement> _set;
10
            protected readonly TReturnConstant _returnConstant;
11
12
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
13
            public SetFiller(ISet<TElement> set, TReturnConstant returnConstant)
```

```
{
15
                _set = set;
16
                _returnConstant = returnConstant;
19
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
20
            public SetFiller(ISet<TElement> set) : this(set, default) { }
21
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
23
            public void Add(TElement element) => _set.Add(element);
24
25
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
26
            public bool AddAndReturnTrue(TElement element) => _set.AddAndReturnTrue(element);
2.8
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
29
            public bool AddFirstAndReturnTrue(IList<TElement> elements) =>
               _set.AddFirstAndReturnTrue(elements);
31
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
32
            public bool AddAllAndReturnTrue(IList<TElement> elements) =>
               _set.AddAllAndReturnTrue(elements);
34
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public bool AddSkipFirstAndReturnTrue(IList<TElement> elements) =>
36
                _set.AddSkipFirstAndReturnTrue(elements);
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public TReturnConstant AddAndReturnConstant(TElement element)
39
40
                _set.Add(element);
41
                return _returnConstant;
42
            }
44
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
45
            public TReturnConstant AddFirstAndReturnConstant(IList<TElement> elements)
47
                 set.AddFirst(elements);
48
                return _returnConstant;
49
50
51
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
52
            public TReturnConstant AddAllAndReturnConstant(IList<TElement> elements)
53
                _set.AddAll(elements);
55
                return _returnConstant;
56
            }
5.8
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
59
            public TReturnConstant AddSkipFirstAndReturnConstant(IList<TElement> elements)
60
61
                _set.AddSkipFirst(elements);
62
                return _returnConstant;
63
            }
64
        }
65
66
     ./csharp/Platform.Collections/Stacks/DefaultStack.cs
1.32
   using System.Collections.Generic;
   using System.Runtime.CompilerServices;
   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
   namespace Platform.Collections.Stacks
6
        public class DefaultStack<TElement> : Stack<TElement>, IStack<TElement>
            public bool IsEmpty
10
                [MethodImpl(MethodImplOptions.AggressiveInlining)]
12
                get => Count <= 0;</pre>
13
            }
        }
15
16
     ./csharp/Platform.Collections/Stacks/IStack.cs
1.33
   using System.Runtime.CompilerServices;
1
   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
3
```

```
namespace Platform.Collections.Stacks
        public interface IStack<TElement>
            bool IsEmpty
9
10
                [MethodImpl(MethodImplOptions.AggressiveInlining)]
                get;
12
14
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
15
            void Push(TElement element);
16
17
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            TElement Pop();
19
20
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
21
            TElement Peek();
22
        }
23
^{24}
      ./csharp/Platform.Collections/Stacks/IStackExtensions.cs
1.34
   using System.Runtime.CompilerServices;
   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
   namespace Platform.Collections.Stacks
5
6
        public static class IStackExtensions
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static void Clear<T>(this IStack<T> stack)
10
11
                while (!stack.IsEmpty)
12
                {
                    _ = stack.Pop();
14
                }
15
            }
16
17
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
18
            public static T PopOrDefault<T>(this IStack<T> stack) => stack.IsEmpty ? default :

    stack.Pop();

20
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static T PeekOrDefault<T>(this IStack<T> stack) => stack.IsEmpty ? default :
22
               stack.Peek();
        }
23
   }
^{24}
      ./csharp/Platform.Collections/Stacks/IStackFactory.cs
   using Platform.Interfaces;
   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
   namespace Platform.Collections.Stacks
6
        public interface IStackFactory<TElement> : IFactory<IStack<TElement>>
9
10
      ./csharp/Platform.Collections/Stacks/StackExtensions.cs
1.36
   using System.Collections.Generic;
   using System.Runtime.CompilerServices;
   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
   namespace Platform.Collections.Stacks
6
        public static class StackExtensions
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
10
            public static T PopOrDefault<T>(this Stack<T> stack) => stack.Count > 0 ? stack.Pop() :
1.1

    default;

12
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
13
            public static T PeekOrDefault<T>(this Stack<T> stack) => stack.Count > 0 ? stack.Peek()
14
               : default;
        }
15
   }
16
```

```
./csharp/Platform.Collections/StringExtensions.cs
   using System;
   using System.Globalization;
2
   using System.Runtime.CompilerServices;
3
   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
   namespace Platform.Collections
8
9
        public static class StringExtensions
10
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
11
            public static string CapitalizeFirstLetter(this string @string)
12
13
                 if (string.IsNullOrWhiteSpace(@string))
14
                     return @string;
16
                 }
17
                var chars = @string.ToCharArray();
                for (var i = 0; i < chars.Length; i++)</pre>
19
20
21
                     var category = char.GetUnicodeCategory(chars[i]);
                     if (category == UnicodeCategory.UppercaseLetter)
22
23
                         return @string;
24
25
                        (category == UnicodeCategory.LowercaseLetter)
26
27
                         chars[i] = char.ToUpper(chars[i]);
2.8
                         return new string(chars);
29
31
                 return @string;
32
            }
33
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public static string Truncate(this string @string, int maxLength) =>
36
                string.IsNullOrEmpty(@string) ? @string : @string.Substring(0,
                Math.Min(@string.Length, maxLength));
37
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
38
            public static string TrimSingle(this string @string, char charToTrim)
39
40
41
                 if (!string.IsNullOrEmpty(@string))
42
                       (@string.Length == 1)
43
44
                            (@string[0] == charToTrim)
45
                         {
46
                              return "";
47
48
                         else
49
                         {
50
                              return @string;
51
                         }
52
                     }
53
                     else
54
55
                         var left = 0;
                         var right = @string.Length - 1;
57
                         if (@string[left] == charToTrim)
58
                         {
59
                              left++;
60
                         }
                         if (@string[right] == charToTrim)
62
63
                              right--;
64
65
66
                         return @string.Substring(left, right - left + 1);
67
68
                 else
69
                 {
70
                     return @string;
71
                 }
72
            }
73
        }
74
   }
75
```

```
./csharp/Platform.Collections/Trees/Node.cs
   using System.Collections.Generic;
   using System.Runtime.CompilerServices;
   // ReSharper disable ForCanBeConvertedToForeach
4
   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
   namespace Platform.Collections.Trees
        public class Node
10
            private Dictionary<object, Node> _childNodes;
11
12
            public object Value
13
14
                [MethodImpl(MethodImplOptions.AggressiveInlining)]
15
                get;
16
                [MethodImpl(MethodImplOptions.AggressiveInlining)]
17
18
            }
19
            public Dictionary<object, Node> ChildNodes
21
                [MethodImpl(MethodImplOptions.AggressiveInlining)]
23
                get => _childNodes ?? (_childNodes = new Dictionary<object, Node>());
24
            }
25
26
            public Node this[object key]
27
                [MethodImpl(MethodImplOptions.AggressiveInlining)]
29
                get => GetChild(key) ?? AddChild(key);
30
                [MethodImpl(MethodImplOptions.AggressiveInlining)]
31
                set => SetChildValue(value, key);
32
33
34
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
35
            public Node(object value) => Value = value;
36
37
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
38
39
            public Node() : this(null) { }
40
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
41
            public bool ContainsChild(params object[] keys) => GetChild(keys) != null;
43
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public Node GetChild(params object[] keys)
45
46
                var node = this;
47
                for (var i = 0; i < keys.Length; i++)</pre>
48
49
50
                    node.ChildNodes.TryGetValue(keys[i], out node);
                    if (node == null)
51
52
                         return null;
54
55
                return node;
56
            }
57
58
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
59
            public object GetChildValue(params object[] keys) => GetChild(keys)?.Value;
60
61
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
62
            public Node AddChild(object key) => AddChild(key, new Node(null));
64
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
65
            public Node AddChild(object key, object value) => AddChild(key, new Node(value));
66
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
68
            public Node AddChild(object key, Node child)
69
70
                ChildNodes.Add(key, child);
                return child;
72
            }
73
74
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
75
76
            public Node SetChild(params object[] keys) => SetChildValue(null, keys);
77
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public Node SetChild(object key) => SetChildValue(null, key);
```

```
80
             [MethodImpl(MethodImplOptions.AggressiveInlining)]
             public Node SetChildValue(object value, params object[] keys)
82
83
                 var node = this;
84
                 for (var i = 0; i < keys.Length; i++)</pre>
85
86
                 {
                     node = SetChildValue(value, keys[i]);
                 }
                 node. Value = value;
                 return node;
90
             }
91
92
             [MethodImpl(MethodImplOptions.AggressiveInlining)]
93
             public Node SetChildValue(object value, object key)
95
                 if (!ChildNodes.TryGetValue(key, out Node child))
96
                     child = AddChild(key, value);
99
                 child.Value = value;
                 return child;
101
             }
102
        }
103
104
1.39
       ./csharp/Platform.Collections.Tests/ArrayTests.cs
   using Xunit;
    using Platform.Collections.Arrays;
 3
    namespace Platform.Collections.Tests
 4
 5
        public class ArrayTests
             [Fact]
             public void GetElementTest()
10
                 var nullArray = (int[])null;
1.1
                 Assert . Equal(0, nullArray . GetElementOrDefault(1));
12
                 Assert.False(nullArray.TryGetElement(1, out int element));
                 Assert.Equal(0, element)
14
                 var array = new int[] { 1, 2, 3 };
15
                 Assert.Equal(3, array.GetElementOrDefault(2));
16
                 Assert.True(array.TryGetElement(2, out element));
17
                 Assert.Equal(3, element);
18
                 Assert.Equal(0, array.GetElementOrDefault(10));
19
                 Assert.False(array.TryGetElement(10, out element));
21
                 Assert.Equal(0, element);
             }
22
        }
23
    }
^{24}
      ./csharp/Platform.Collections.Tests/BitStringTests.cs
    using System;
    using System.Collections;
    using Xunit;
    using Platform.Random;
 4
    namespace Platform.Collections.Tests
 7
        public static class BitStringTests
 8
 9
             [Fact]
10
             public static void BitGetSetTest()
11
                 const int n = 250;
13
                 var bitArray = new BitArray(n);
14
                 var bitString = new BitString(n);
15
                 for (var i = 0; i < n; i++)</pre>
16
17
                     var value = RandomHelpers.Default.NextBoolean();
                     bitArray.Set(i, value);
19
                     bitString.Set(i, value);
20
                     Assert.Equal(value, bitArray.Get(i));
                     Assert.Equal(value, bitString.Get(i));
22
                 }
23
             }
25
             [Fact]
```

```
public static void BitVectorNotTest()
    TestToOperationsWithSameMeaning((x, y, w, v) =>
        x.VectorNot();
        w.Not();
    });
}
[Fact]
public static void BitParallelNotTest()
    TestToOperationsWithSameMeaning((x, y, w, v) =>
        x.ParallelNot();
        w.Not();
    });
}
[Fact]
public static void BitParallelVectorNotTest()
    TestToOperationsWithSameMeaning((x, y, w, v) =>
        x.ParallelVectorNot();
        w.Not();
    });
}
[Fact]
public static void BitVectorAndTest()
    TestToOperationsWithSameMeaning((x, y, w, v) =>
        x. VectorAnd(y);
        w.And(v);
    });
}
[Fact]
public static void BitParallelAndTest()
    TestToOperationsWithSameMeaning((x, y, w, v) =>
        x.ParallelAnd(y);
        w.And(v);
    });
}
[Fact]
public static void BitParallelVectorAndTest()
    TestToOperationsWithSameMeaning((x, y, w, v) =>
        x.ParallelVectorAnd(y);
        w.And(v);
    });
}
[Fact]
public static void BitVectorOrTest()
    TestToOperationsWithSameMeaning((x, y, w, v) =>
        x.VectorOr(y);
        w.Or(v);
    });
}
[Fact]
public static void BitParallelOrTest()
    TestToOperationsWithSameMeaning((x, y, w, v) =>
        x.ParallelOr(y);
        w.Or(v);
    });
}
```

29 30

32

33

34 35

36

38

39 40 41

42

43

44 45

46

47 48

49

51

52

53

55

57 58 59

60

61

62

64

66

67 68

69 70

 $71 \\ 72$

73

 $\frac{74}{75}$

76 77

78

79 80

82

83

84 85

86

88

89 90

91

92

93

95

96

97 98

99

101

102

```
105
             [Fact]
             public static void BitParallelVectorOrTest()
107
108
                  TestToOperationsWithSameMeaning((x, y, w, v) =>
109
110
                      x.ParallelVectorOr(y);
111
                      w.Or(v);
112
                  });
             }
114
115
             [Fact]
             public static void BitVectorXorTest()
117
118
119
                  TestToOperationsWithSameMeaning((x, y, w, v) =>
120
                      x.VectorXor(y);
121
                      w.Xor(v);
                  });
123
             }
124
125
             [Fact]
126
             public static void BitParallelXorTest()
127
                  TestToOperationsWithSameMeaning((x, y, w, v) =>
129
130
131
                      x.ParallelXor(y);
132
                      w.Xor(v);
                  });
133
             }
134
135
             [Fact]
136
137
             public static void BitParallelVectorXorTest()
138
                  TestToOperationsWithSameMeaning((x, y, w, v) =>
139
140
                      x.ParallelVectorXor(y);
                      w.Xor(v);
142
                  });
143
             }
144
145
             private static void TestToOperationsWithSameMeaning(Action<BitString, BitString,
146
                 BitString, BitString> test)
147
                  const int n = 5654;
148
                  var x = new BitString(n);
                  var y = new BitString(n);
150
                 while (x.Equals(y))
151
152
                      x.SetRandomBits();
                      y.SetRandomBits();
154
                  }
155
156
                  var w = new BitString(x);
                  var v = new BitString(y);
157
                  Assert.False(x.Equals(y));
158
                  Assert.False(w.Equals(v));
159
                  Assert.True(x.Equals(w));
                  Assert.True(y.Equals(v));
161
                 test(x, y, w, v);
Assert.True(x.Equals(w));
162
             }
164
         }
165
166
       ./csharp/Platform.Collections.Tests/CharsSegmentTests.cs
1.41
    using Xunit;
 1
    using Platform.Collections.Segments;
 3
    namespace Platform.Collections.Tests
 4
 5
         public static class CharsSegmentTests
 7
             [Fact]
             public static void GetHashCodeEqualsTest()
10
                  const string testString = "test test";
11
                  var testArray = testString.ToCharArray();
                  var firstHashCode = new CharSegment(testArray, 0, 4).GetHashCode();
13
                  var secondHashCode = new CharSegment(testArray, 5, 4) GetHashCode();
```

```
Assert.Equal(firstHashCode, secondHashCode);
15
              }
16
17
              [Fact]
              public static void EqualsTest()
19
20
                   const string testString = "test test";
21
                   var testArray = testString.ToCharArray();
22
                   var first = new CharSegment(testArray, 0, 4);
23
                   var second = new CharSegment(testArray, 5, 4);
                   Assert.True(first.Equals(second));
25
              }
26
27
         }
    }
1.42
      ./csharp/Platform.Collections.Tests/ListTests.cs
    using System.Collections.Generic;
    using Xunit;
    using Platform.Collections.Lists;
4
    namespace Platform.Collections.Tests
         public class ListTests
8
9
              [Fact]
10
              public void GetElementTest()
11
                   var nullList = (IList<int>)null;
                   Assert.Equal(0, nullList.GetElementOrDefault(1));
14
                   Assert.False(nullList.TryGetElement(1, out int element));
15
                   Assert.Equal(0, element)
                   var list = new List<int>() { 1, 2, 3 };
17
                   Assert.Equal(3, list.GetElementOrDefault(2));
18
                   Assert.True(list.TryGetElement(2, out element));
                   Assert.Equal(3, element);
20
                   Assert.Equal(0, list.GetElementOrDefault(10));
21
                   Assert.False(list.TryGetElement(10, out element));
22
23
                   Assert.Equal(0, element);
              }
24
         }
25
    }
      ./csharp/Platform.Collections.Tests/StringTests.cs
1.43
   using Xunit;
    namespace Platform.Collections.Tests
3
4
         public static class StringTests
              [Fact]
              public static void CapitalizeFirstLetterTest()
                   Assert.Equal("Hello", "hello".CapitalizeFirstLetter());
Assert.Equal("Hello", "Hello".CapitalizeFirstLetter());
10
11
                   Assert.Equal(" Hello", " hello".CapitalizeFirstLetter());
              }
14
              [Fact]
15
              public static void TrimSingleTest()
16
17
                  Assert.Equal("", "'".TrimSingle('\''));
Assert.Equal("", "''".TrimSingle('\''));
Assert.Equal("hello", "'hello'".TrimSingle('\''));
Assert.Equal("hello", "hello'".TrimSingle('\''));
Assert.Equal("hello", "'hello".TrimSingle('\''));
19
20
21
22
              }
23
         }
^{24}
```

}

Index ./csharp/Platform.Collections.Tests/ArrayTests.cs, 42 ./csharp/Platform.Collections.Tests/BitStringTests.cs, 42 ./csharp/Platform.Collections.Tests/CharsSegmentTests.cs, 44 ./csharp/Platform.Collections.Tests/ListTests.cs, 45 ./csharp/Platform.Collections.Tests/StringTests.cs, 45 ./csharp/Platform.Collections/Arrays/ArrayFiller[TElement, TReturnConstant].cs, 1 ./csharp/Platform.Collections/Arrays/ArrayFiller[TElement].cs, 1 ./csharp/Platform Collections/Arrays/ArrayPool.cs, 2 ./csharp/Platform.Collections/Arrays/ArrayPool[T].cs, 2 ./csharp/Platform Collections/Arrays/ArrayString.cs, 3 ./csharp/Platform.Collections/Arrays/CharArrayExtensions.cs, 3 ./csharp/Platform.Collections/Arrays/GenericArrayExtensions.cs, 4 ./csharp/Platform.Collections/BitString.cs, 8 ./csharp/Platform.Collections/BitStringExtensions.cs, 23 ./csharp/Platform.Collections/Concurrent/ConcurrentQueueExtensions.cs, 23 ./csharp/Platform.Collections/Concurrent/ConcurrentStackExtensions.cs, 24 ./csharp/Platform.Collections/EnsureExtensions.cs, 24 ./csharp/Platform.Collections/ICollectionExtensions.cs, 25 ./csharp/Platform.Collections/IDictionaryExtensions.cs, 25 ./csharp/Platform.Collections/Lists/CharlListExtensions.cs, 26 ./csharp/Platform.Collections/Lists/IListComparer.cs, 26 ./csharp/Platform.Collections/Lists/IListEqualityComparer.cs, 27 ./csharp/Platform.Collections/Lists/IListExtensions.cs, 27 ./csharp/Platform.Collections/Lists/ListFiller.cs, 30 /csharp/Platform Collections/Segments/CharSegment.cs, 31 ./csharp/Platform.Collections/Segments/Segment.cs, 32 ./csharp/Platform.Collections/Segments/Walkers/AllSegmentsWalkerBase.cs, 33 $./csharp/Platform.Collections/Segments/Walkers/AllSegmentsWalkerBase[T,\ TSegment].cs,\ 33$./csharp/Platform.Collections/Segments/Walkers/AllSegmentsWalkerBase[T].cs, 34 /csharp/Platform.Collections/Segments/Walkers/AllSegmentsWalkerExtensions.cs, 34 ./csharp/Platform.Collections/Segments/Walkers/DictionaryBasedDuplicateSegmentsWalkerBase[T, Segment].cs, 34 ./csharp/Platform.Collections/Segments/Walkers/DictionaryBasedDuplicateSegmentsWalkerBase[T].cs, 35 /csharp/Platform.Collections/Segments/Walkers/DuplicateSegmentsWalkerBase[T, TSegment].cs, 36 ./csharp/Platform.Collections/Segments/Walkers/DuplicateSegmentsWalkerBase[T].cs, 36 ./csharp/Platform.Collections/Sets/ISetExtensions.cs, 36

./csharp/Platform.Collections/Sets/SetFiller.cs, 37

./csharp/Platform.Collections/Stacks/DefaultStack.cs, 38 ./csharp/Platform.Collections/Stacks/IStack.cs, 38

./csharp/Platform.Collections/Stacks/IStackExtensions.cs, 39 ./csharp/Platform.Collections/Stacks/IStackFactory.cs, 39 ./csharp/Platform.Collections/Stacks/StackExtensions.cs, 39 ./csharp/Platform.Collections/StringExtensions.cs, 40 ./csharp/Platform.Collections/Trees/Node.cs, 40