

LinksPlatform's Platform.Collections Class Library

1.1 ./Platform.Collections/Arrays/ArrayExtensions.cs

```
1 using System;
2 using System.Collections.Generic;
3 using System.Runtime.CompilerServices;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Collections.Arrays
8 {
9     public static class ArrayExtensions
10     {
11         [MethodImpl(MethodImplOptions.AggressiveInlining)]
12         public static IList<TLink> ShiftRight<TLink>(this TLink[] array) => array.ShiftRight(1);
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         public static IList<TLink> ShiftRight<TLink>(this TLink[] array, int shift)
16         {
17             var restrictions = new TLink[array.Length + shift];
18             Array.Copy(array, 0, restrictions, shift, array.Length);
19             return restrictions;
20         }
21     }
22 }
```

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

```
1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Collections.Arrays
7 {
8     public class ArrayFiller<TElement, TReturnConstant> : ArrayFiller<TElement>
9     {
10         protected readonly TReturnConstant _returnConstant;
11
12         public ArrayFiller(TElement[] array, long offset, TReturnConstant returnConstant) :
13             ↪ base(array, offset) => _returnConstant = returnConstant;
14
15         public ArrayFiller(TElement[] array, TReturnConstant returnConstant) : this(array, 0,
16             ↪ returnConstant) { }
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         public TReturnConstant AddAndReturnConstant(TElement element)
20         {
21             _array[_position++] = element;
22             return _returnConstant;
23         }
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         public TReturnConstant AddFirstAndReturnConstant(IList<TElement> collection)
27         {
28             _array[_position++] = collection[0];
29             return _returnConstant;
30         }
31     }
32 }
```

1.3 ./Platform.Collections/Arrays/ArrayFiller[TElement].cs

```
1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Collections.Arrays
7 {
8     public class ArrayFiller<TElement>
9     {
10         protected readonly TElement[] _array;
11         protected long _position;
12
13         public ArrayFiller(TElement[] array, long offset)
14         {
15             _array = array;
16             _position = offset;
17         }
18
19         public ArrayFiller(TElement[] array) : this(array, 0) { }
20     }
21 }
```

```

21     [MethodImpl(MethodImplOptions.AggressiveInlining)]
22     public void Add(TElement element) => _array[_position++] = element;
23
24     [MethodImpl(MethodImplOptions.AggressiveInlining)]
25     public bool AddAndReturnTrue(TElement element)
26     {
27         _array[_position++] = element;
28         return true;
29     }
30
31     [MethodImpl(MethodImplOptions.AggressiveInlining)]
32     public bool AddFirstAndReturnTrue(IList<TElement> collection)
33     {
34         _array[_position++] = collection[0];
35         return true;
36     }
37 }
38 }

```

1.4 ./Platform.Collections/Arrays/ArrayPool.cs

```

1 using System.Runtime.CompilerServices;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Collections.Arrays
6 {
7     public static class ArrayPool
8     {
9         public static readonly int DefaultSizesAmount = 512;
10        public static readonly int DefaultMaxArraysPerSize = 32;
11
12        [MethodImpl(MethodImplOptions.AggressiveInlining)]
13        public static T[] Allocate<T>(long size) => ArrayPool<T>.ThreadInstance.Allocate(size);
14
15        [MethodImpl(MethodImplOptions.AggressiveInlining)]
16        public static void Free<T>(T[] array) => ArrayPool<T>.ThreadInstance.Free(array);
17    }
18 }

```

1.5 ./Platform.Collections/Arrays/ArrayPool[T].cs

```

1 using System;
2 using System.Collections.Generic;
3 using Platform.Exceptions;
4 using Platform.Disposables;
5 using Platform.Ranges;
6 using Platform.Collections.Stacks;
7
8 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Collections.Arrays
11 {
12     /// <remarks>
13     /// Original idea from
14     /// ↪ http://geekswithblogs.net/blackrob/archive/2014/12/18/array-pooling-in-csharp.aspx
15     /// </remarks>
16     public class ArrayPool<T>
17     {
18         public static readonly T[] Empty = new T[0];
19
20         // May be use Default class for that later.
21         [ThreadStatic]
22         internal static ArrayPool<T> _threadInstance;
23         internal static ArrayPool<T> ThreadInstance { get => _threadInstance ?? (_threadInstance
24             ↪ = new ArrayPool<T>()); }
25
26         private readonly int _maxArraysPerSize;
27         private readonly Dictionary<int, Stack<T[]>> _pool = new Dictionary<int,
28             ↪ Stack<T[]>>(ArrayPool.DefaultSizesAmount);
29
30         public ArrayPool(int maxArraysPerSize) => _maxArraysPerSize = maxArraysPerSize;
31
32         public ArrayPool() : this(ArrayPool.DefaultMaxArraysPerSize) { }
33
34         public Disposable<T[]> AllocateDisposable(long size) => (Allocate(size), Free);
35
36         public Disposable<T[]> Resize(Disposable<T[]> source, long size)
37         {
38             var destination = AllocateDisposable(size);
39             T[] sourceArray = source;
40             T[] destinationArray = destination;

```

```

38     Array.Copy(sourceArray, destinationArray, size < sourceArray.Length ? (int)size :
    ↪ sourceArray.Length);
39     source.Dispose();
40     return destination;
41 }
42
43 public virtual void Clear() => _pool.Clear();
44
45 public virtual T[] Allocate(long size)
46 {
47     Ensure.Always.ArgumentInRange(size, (0, int.MaxValue));
48     return size == 0 ? Empty : _pool.GetOrDefault((int)size)?.PopOrDefault() ?? new
    ↪ T[size];
49 }
50
51 public virtual void Free(T[] array)
52 {
53     Ensure.Always.ArgumentNotNull(array, nameof(array));
54     if (array.Length == 0)
55     {
56         return;
57     }
58     var stack = _pool.GetOrAdd(array.Length, size => new Stack<T[]>(_maxArraysPerSize));
59     if (stack.Count == _maxArraysPerSize) // Stack is full
60     {
61         return;
62     }
63     stack.Push(array);
64 }
65 }
66 }

```

1.6 ./Platform.Collections/Arrays/ArrayString.cs

```

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

```

1.7 ./Platform.Collections/Arrays/CharArrayExtensions.cs

```

1 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3 namespace Platform.Collections.Arrays
4 {
5     public static unsafe class CharArrayExtensions
6     {
7         /// <remarks>
8         /// Based on https://github.com/Microsoft/referencesource/blob/3b1eaf5203992df69de44c783\_
    ↪ a3eda37d3d4cd10/mscorlib/system/string.cs#L833
9         /// </remarks>
10        public static int GenerateHashCode(this char[] array, int offset, int length)
11        {
12            var hashSeed = 5381;
13            var hashAccumulator = hashSeed;
14            fixed (char* pointer = &array[offset])
15            {
16                for (char* s = pointer, last = s + length; s < last; s++)
17                {
18                    hashAccumulator = (hashAccumulator << 5) + hashAccumulator ^ *s;
19                }
20            }
21            return hashAccumulator + (hashSeed * 1566083941);
22        }
23
24        /// <remarks>
25        /// Based on https://github.com/Microsoft/referencesource/blob/3b1eaf5203992df69de44c783\_
    ↪ a3eda37d3d4cd10/mscorlib/system/string.cs#L364
26        /// </remarks>
27        public static bool ContentEqualTo(this char[] left, int leftOffset, int length, char[]
    ↪ right, int rightOffset)

```

```

28     {
29         fixed (char* leftPointer = &left[leftOffset])
30         {
31             fixed (char* rightPointer = &right[rightOffset])
32             {
33                 char* leftPointerCopy = leftPointer, rightPointerCopy = rightPointer;
34                 if (!CheckArraysMainPartForEquality(ref leftPointerCopy, ref
35                     ↪ rightPointerCopy, ref length))
36                 {
37                     return false;
38                 }
39                 CheckArraysRemainderForEquality(ref leftPointerCopy, ref rightPointerCopy,
40                     ↪ ref length);
41                 return length <= 0;
42             }
43         }
44     }
45
46     private static bool CheckArraysMainPartForEquality(ref char* left, ref char* right, ref
47     ↪ int length)
48     {
49         while (length >= 10)
50         {
51             if ((* (int*)left != *(int*)right)
52                 || (*(int*)(left + 2) != *(int*)(right + 2))
53                 || (*(int*)(left + 4) != *(int*)(right + 4))
54                 || (*(int*)(left + 6) != *(int*)(right + 6))
55                 || (*(int*)(left + 8) != *(int*)(right + 8)))
56             {
57                 return false;
58             }
59             left += 10;
60             right += 10;
61             length -= 10;
62         }
63         return true;
64     }
65
66     private static void CheckArraysRemainderForEquality(ref char* left, ref char* right, ref
67     ↪ int length)
68     {
69         // This depends on the fact that the String objects are
70         // always zero terminated and that the terminating zero is not included
71         // in the length. For odd string sizes, the last compare will include
72         // the zero terminator.
73         while (length > 0)
74         {
75             if ((* (int*)left != *(int*)right)
76                 {
77                 break;
78             }
79             left += 2;
80             right += 2;
81             length -= 2;
82         }
83     }
84 }

```

1.8 ./Platform.Collections/Arrays/GenericArrayExtensions.cs

```

1  using System;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Collections.Arrays
6  {
7      public static class GenericArrayExtensions
8      {
9          public static T[] Clone<T>(this T[] array)
10         {
11             var copy = new T[array.Length];
12             Array.Copy(array, 0, copy, 0, array.Length);
13             return copy;
14         }
15     }
16 }

```

1.9 ./Platform.Collections/BitString.cs

```
1 using System;
2 using System.Collections.Concurrent;
3 using System.Collections.Generic;
4 using System.Numerics;
5 using System.Runtime.CompilerServices;
6 using System.Threading.Tasks;
7 using Platform.Exceptions;
8 using Platform.Ranges;
9
10 // ReSharper disable ForCanBeConvertedToForeach
11 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
12
13 namespace Platform.Collections
14 {
15     /// <remarks>
16     /// А что если хранить карту значений, где каждый бит будет означать присутствует ли блок в
17     ///   ↳ 64 бит в массиве значений.
18     /// 64 бита по 0 бит, будут означать отсутствие 64-х блоков по 64 бита. Т.е. упаковка 512
19     ///   ↳ байт в 8 байт.
20     /// Подобный принцип можно применять и к 64-ём блокам и т.п. По сути это карта значений. С
21     ///   ↳ помощью которой можно быстро
22     /// проверять есть ли значения непосредственно далее (ниже по уровню).
23     /// Или как таблица виртуальной памяти где номер блока означает его присутствие и адрес.
24     /// </remarks>
25     public class BitString : IEquatable<BitString>
26     {
27         private static readonly byte[] [] _bitsSetIn16Bits;
28         private long[] _array;
29         private long _length;
30         private long _minPositiveWord;
31         private long _maxPositiveWord;
32
33         public bool this[long index]
34         {
35             get => Get(index);
36             set => Set(index, value);
37         }
38
39         public long Length
40         {
41             get => _length;
42             set
43             {
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)
51                 {
52                     var words = GetWordsCountFromIndex(value);
53                     var oldWords = GetWordsCountFromIndex(_length);
54                     if (words > _array.LongLength)
55                     {
56                         var copy = new long[words];
57                         Array.Copy(_array, copy, _array.LongLength);
58                         _array = copy;
59                     }
60                     else
61                     {
62                         // What is going on here?
63                         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                     }
71                 }
72             }
73             else
74             {
75                 // Looks like minimum and maximum positive words are not updated
76                 throw new NotImplementedException();
77             }
78             _length = value;
79         }
80     }
81 }
```

```
77 #region Constructors
```

```
78 static BitString()
```

```
79 {
```

```
80     _bitsSetIn16Bits = new byte[65536] [];
```

```
81     int i, c, k;
```

```
82     byte bitIndex;
```

```
83     for (i = 0; i < 65536; i++)
```

```
84     {
```

```
85         // Calculating size of array (number of positive bits)
```

```
86         for (c = 0, k = 1; k <= 65536; k <= 1)
```

```
87         {
```

```
88             if ((i & k) == k)
```

```
89             {
```

```
90                 c++;
```

```
91             }
```

```
92         }
```

```
93         var array = new byte[c];
```

```
94         // Adding positive bits indices into array
```

```
95         for (bitIndex = 0, c = 0, k = 1; k <= 65536; k <= 1)
```

```
96         {
```

```
97             if ((i & k) == k)
```

```
98             {
```

```
99                 array[c++] = bitIndex;
```

```
100             }
```

```
101             bitIndex++;
```

```
102         }
```

```
103         _bitsSetIn16Bits[i] = array;
```

```
104     }
```

```
105 }
```

```
106 public BitString(BitString other)
```

```
107 {
```

```
108     Ensure.Always.ArgumentNotNull(other, nameof(other));
```

```
109     _length = other._length;
```

```
110     _array = new long[GetWordsCountFromIndex(_length)];
```

```
111     _minPositiveWord = other._minPositiveWord;
```

```
112     _maxPositiveWord = other._maxPositiveWord;
```

```
113     Array.Copy(other._array, _array, _array.LongLength);
```

```
114 }
```

```
115 public BitString(long length)
```

```
116 {
```

```
117     Ensure.Always.ArgumentInRange(length, GetValidLengthRange(), nameof(length));
```

```
118     _length = length;
```

```
119     _array = new long[GetWordsCountFromIndex(_length)];
```

```
120     MarkBordersAsAllBitsReset();
```

```
121 }
```

```
122 public BitString(long length, bool defaultValue)
```

```
123     : this(length)
```

```
124 {
```

```
125     if (defaultValue)
```

```
126     {
```

```
127         SetAll();
```

```
128     }
```

```
129 }
```

```
130 #endregion
```

```
131 public BitString Not()
```

```
132 {
```

```
133     for (var i = 0; i < _array.Length; i++)
```

```
134     {
```

```
135         _array[i] = ~_array[i];
```

```
136         RefreshBordersByWord(i);
```

```
137     }
```

```
138     return this;
```

```
139 }
```

```
140 public BitString ParallelNot()
```

```
141 {
```

```
142     var processorCount = Environment.ProcessorCount;
```

```
143     if (processorCount <= 1)
```

```
144     {
```

```
145         return Not();
```

```
146     }
```

```
147 }
```

```

155     var partitioner = Partitioner.Create(0, _array.Length, _array.Length /
156         ↪ processorCount);
157     Parallel.ForEach(partitioner.GetDynamicPartitions(), range =>
158     {
159         var maximum = range.Item2;
160         for (var i = range.Item1; i < maximum; i++)
161         {
162             _array[i] = ~_array[i];
163         }
164     });
165     MarkBordersAsAllBitsSet();
166     TryShrinkBorders();
167     return this;
168 }
169 public BitString VectorNot()
170 {
171     if (!Vector.IsHardwareAccelerated)
172     {
173         return Not();
174     }
175     var step = Vector<long>.Count;
176     if (_array.Length < step)
177     {
178         return Not();
179     }
180     VectorNotLoop(_array, step, 0, _array.Length);
181     MarkBordersAsAllBitsSet();
182     TryShrinkBorders();
183     return this;
184 }
185 public BitString ParallelVectorNot()
186 {
187     var processorCount = Environment.ProcessorCount;
188     if (processorCount <= 1 && Vector.IsHardwareAccelerated)
189     {
190         return VectorNot();
191     }
192     if (!Vector.IsHardwareAccelerated)
193     {
194         return Not();
195     }
196     var step = Vector<long>.Count;
197     if (_array.Length < (step * Environment.ProcessorCount))
198     {
199         return VectorNot();
200     }
201     var partitioner = Partitioner.Create(0, _array.Length, _array.Length /
202         ↪ processorCount);
203     Parallel.ForEach(partitioner.GetDynamicPartitions(), range => VectorNotLoop(_array,
204         ↪ step, range.Item1, range.Item2));
205     MarkBordersAsAllBitsSet();
206     TryShrinkBorders();
207     return this;
208 }
209 static private void VectorNotLoop(long[] array, int step, int start, int maximum)
210 {
211     var i = start;
212     var range = maximum - start - 1;
213     var stop = range - (range % step);
214     for (; i < stop; i += step)
215     {
216         var vector = new Vector<long>(array, i);
217         (~vector).CopyTo(array, i);
218     }
219     for (; i < maximum; i++)
220     {
221         array[i] = ~array[i];
222     }
223 }
224 public BitString And(BitString other)
225 {
226     EnsureBitStringHasTheSameSize(other, nameof(other));
227     GetCommonOuterBorders(this, other, out long from, out long to);
228     var otherArray = other._array;
229

```

```

230     for (var i = from; i <= to; i++)
231     {
232         _array[i] &= otherArray[i];
233         RefreshBordersByWord(i);
234     }
235     return this;
236 }
237
238 public BitString ParallelAnd(BitString other)
239 {
240     var processorCount = Environment.ProcessorCount;
241     if (processorCount <= 1)
242     {
243         return And(other);
244     }
245     EnsureBitStringHasTheSameSize(other, nameof(other));
246     GetCommonOuterBorders(this, other, out long from, out long to);
247     var partitioner = Partitioner.Create(from, to + 1, (to - from) / processorCount);
248     Parallel.ForEach(partitioner.GetDynamicPartitions(), range =>
249     {
250         var maximum = range.Item2;
251         for (var i = range.Item1; i < maximum; i++)
252         {
253             _array[i] &= other._array[i];
254         }
255     });
256     MarkBordersAsAllBitsSet();
257     TryShrinkBorders();
258     return this;
259 }
260
261 public BitString VectorAnd(BitString other)
262 {
263     if (!Vector.IsHardwareAccelerated)
264     {
265         return And(other);
266     }
267     var step = Vector<long>.Count;
268     if (_array.Length < step)
269     {
270         return And(other);
271     }
272     EnsureBitStringHasTheSameSize(other, nameof(other));
273     GetCommonOuterBorders(this, other, out long from, out long to);
274     VectorAndLoop(_array, other._array, step, (int)from, (int)(to + 1));
275     MarkBordersAsAllBitsSet();
276     TryShrinkBorders();
277     return this;
278 }
279
280 public BitString ParallelVectorAnd(BitString other)
281 {
282     var processorCount = Environment.ProcessorCount;
283     if (processorCount <= 1 && Vector.IsHardwareAccelerated)
284     {
285         return VectorAnd(other);
286     }
287     if (!Vector.IsHardwareAccelerated)
288     {
289         return And(other);
290     }
291     var step = Vector<long>.Count;
292     if (_array.Length < (step * Environment.ProcessorCount))
293     {
294         return VectorAnd(other);
295     }
296     EnsureBitStringHasTheSameSize(other, nameof(other));
297     GetCommonOuterBorders(this, other, out long from, out long to);
298     var partitioner = Partitioner.Create(from, to + 1, (to - from) / processorCount);
299     Parallel.ForEach(partitioner.GetDynamicPartitions(), range => VectorAndLoop(_array,
300     ↪ other._array, step, (int)range.Item1, (int)range.Item2));
301     MarkBordersAsAllBitsSet();
302     TryShrinkBorders();
303     return this;
304 }
305
306 static private void VectorAndLoop(long[] array, long[] otherArray, int step, int start,
307     ↪ int maximum)

```



```

306 {
307     var i = start;
308     var range = maximum - start - 1;
309     var stop = range - (range % step);
310     for (; i < stop; i += step)
311     {
312         var thisVector = new Vector<long>(array, i);
313         var otherVector = new Vector<long>(otherArray, i);
314         (thisVector & otherVector).CopyTo(array, i);
315     }
316     for (; i < maximum; i++)
317     {
318         array[i] &= otherArray[i];
319     }
320 }
321
322 public BitString Or(BitString other)
323 {
324     EnsureBitStringHasTheSameSize(other, nameof(other));
325     GetCommonOuterBorders(this, other, out long from, out long to);
326     for (var i = from; i <= to; i++)
327     {
328         _array[i] |= other._array[i];
329         RefreshBordersByWord(i);
330     }
331     return this;
332 }
333
334 public BitString ParallelOr(BitString other)
335 {
336     var processorCount = Environment.ProcessorCount;
337     if (processorCount <= 1)
338     {
339         return Or(other);
340     }
341     EnsureBitStringHasTheSameSize(other, nameof(other));
342     GetCommonOuterBorders(this, other, out long from, out long to);
343     var partitioner = Partitioner.Create(from, to + 1, (to - from) / processorCount);
344     Parallel.ForEach(partitioner.GetDynamicPartitions(), range =>
345     {
346         var maximum = range.Item2;
347         for (var i = range.Item1; i < maximum; i++)
348         {
349             _array[i] |= other._array[i];
350         }
351     });
352     MarkBordersAsAllBitsSet();
353     TryShrinkBorders();
354     return this;
355 }
356
357 public BitString VectorOr(BitString other)
358 {
359     if (!Vector.IsHardwareAccelerated)
360     {
361         return Or(other);
362     }
363     var step = Vector<long>.Count;
364     if (_array.Length < step)
365     {
366         return Or(other);
367     }
368     EnsureBitStringHasTheSameSize(other, nameof(other));
369     GetCommonOuterBorders(this, other, out long from, out long to);
370     VectorOrLoop(_array, other._array, step, (int)from, (int)(to + 1));
371     MarkBordersAsAllBitsSet();
372     TryShrinkBorders();
373     return this;
374 }
375
376 public BitString ParallelVectorOr(BitString other)
377 {
378     var processorCount = Environment.ProcessorCount;
379     if (processorCount <= 1 && Vector.IsHardwareAccelerated)
380     {
381         return VectorOr(other);
382     }
383     if (!Vector.IsHardwareAccelerated)
384     {

```

```

385         return Or(other);
386     }
387     var step = Vector<long>.Count;
388     if (_array.Length < (step * Environment.ProcessorCount))
389     {
390         return VectorOr(other);
391     }
392     EnsureBitStringHasTheSameSize(other, nameof(other));
393     GetCommonOuterBorders(this, other, out long from, out long to);
394     var partitioner = Partitioner.Create(from, to + 1, (to - from) / processorCount);
395     Parallel.ForEach(partitioner.GetDynamicPartitions(), range => VectorOrLoop(_array,
396         ↪ other._array, step, (int)range.Item1, (int)range.Item2));
397     MarkBordersAsAllBitsSet();
398     TryShrinkBorders();
399     return this;
400 }
401
402 static private void VectorOrLoop(long[] array, long[] otherArray, int step, int start,
403     ↪ int maximum)
404 {
405     var i = start;
406     var range = maximum - start - 1;
407     var stop = range - (range % step);
408     for (; i < stop; i += step)
409     {
410         var thisVector = new Vector<long>(array, i);
411         var otherVector = new Vector<long>(otherArray, i);
412         (thisVector | otherVector).CopyTo(array, i);
413     }
414     for (; i < maximum; i++)
415     {
416         array[i] |= otherArray[i];
417     }
418 }
419
420 public BitString Xor(BitString other)
421 {
422     EnsureBitStringHasTheSameSize(other, nameof(other));
423     GetCommonOuterBorders(this, other, out long from, out long to);
424     for (var i = from; i <= to; i++)
425     {
426         _array[i] ^= other._array[i];
427         RefreshBordersByWord(i);
428     }
429     return this;
430 }
431
432 public BitString ParallelXor(BitString other)
433 {
434     var processorCount = Environment.ProcessorCount;
435     if (processorCount <= 1)
436     {
437         return Xor(other);
438     }
439     EnsureBitStringHasTheSameSize(other, nameof(other));
440     GetCommonOuterBorders(this, other, out long from, out long to);
441     var partitioner = Partitioner.Create(from, to + 1, (to - from) / processorCount);
442     Parallel.ForEach(partitioner.GetDynamicPartitions(), range =>
443     {
444         var maximum = range.Item2;
445         for (var i = range.Item1; i < maximum; i++)
446         {
447             _array[i] ^= other._array[i];
448         }
449     });
450     MarkBordersAsAllBitsSet();
451     TryShrinkBorders();
452     return this;
453 }
454
455 public BitString VectorXor(BitString other)
456 {
457     if (!Vector.IsHardwareAccelerated)
458     {
459         return Xor(other);
460     }
461     var step = Vector<long>.Count;
462     if (_array.Length < step)

```

```

461     {
462         return Xor(other);
463     }
464     EnsureBitStringHasTheSameSize(other, nameof(other));
465     GetCommonOuterBorders(this, other, out long from, out long to);
466     VectorXorLoop(_array, other._array, step, (int)from, (int)(to + 1));
467     MarkBordersAsAllBitsSet();
468     TryShrinkBorders();
469     return this;
470 }
471
472 public BitString ParallelVectorXor(BitString other)
473 {
474     var processorCount = Environment.ProcessorCount;
475     if (processorCount <= 1 && Vector.IsHardwareAccelerated)
476     {
477         return VectorXor(other);
478     }
479     if (!Vector.IsHardwareAccelerated)
480     {
481         return Xor(other);
482     }
483     var step = Vector<long>.Count;
484     if (_array.Length < (step * Environment.ProcessorCount))
485     {
486         return VectorXor(other);
487     }
488     EnsureBitStringHasTheSameSize(other, nameof(other));
489     GetCommonOuterBorders(this, other, out long from, out long to);
490     var partitioner = Partitioner.Create(from, to + 1, (to - from) / processorCount);
491     Parallel.ForEach(partitioner.GetDynamicPartitions(), range => VectorXorLoop(_array,
492         ↪ other._array, step, (int)range.Item1, (int)range.Item2));
493     MarkBordersAsAllBitsSet();
494     TryShrinkBorders();
495     return this;
496 }
497
498 static private void VectorXorLoop(long[] array, long[] otherArray, int step, int start,
499     ↪ int maximum)
500 {
501     var i = start;
502     var range = maximum - start - 1;
503     var stop = range - (range % step);
504     for (; i < stop; i += step)
505     {
506         var thisVector = new Vector<long>(array, i);
507         var otherVector = new Vector<long>(otherArray, i);
508         (thisVector ^ otherVector).CopyTo(array, i);
509     }
510     for (; i < maximum; i++)
511     {
512         array[i] ^= otherArray[i];
513     }
514 }
515
516 private void RefreshBordersByWord(long wordIndex)
517 {
518     if (_array[wordIndex] == 0)
519     {
520         if (wordIndex == _minPositiveWord && wordIndex != _array.LongLength - 1)
521         {
522             _minPositiveWord++;
523         }
524         if (wordIndex == _maxPositiveWord && wordIndex != 0)
525         {
526             _maxPositiveWord--;
527         }
528     }
529     else
530     {
531         if (wordIndex < _minPositiveWord)
532         {
533             _minPositiveWord = wordIndex;
534         }
535         if (wordIndex > _maxPositiveWord)
536         {
537             _maxPositiveWord = wordIndex;
538         }
539     }
540 }

```

```

538 }
539
540 public bool TryShrinkBorders()
541 {
542     GetBorders(out long from, out long to);
543     while (from <= to && _array[from] == 0)
544     {
545         from++;
546     }
547     if (from > to)
548     {
549         MarkBordersAsAllBitsReset();
550         return true;
551     }
552     while (to >= from && _array[to] == 0)
553     {
554         to--;
555     }
556     if (to < from)
557     {
558         MarkBordersAsAllBitsReset();
559         return true;
560     }
561     var bordersUpdated = from != _minPositiveWord || to != _maxPositiveWord;
562     if (bordersUpdated)
563     {
564         SetBorders(from, to);
565     }
566     return bordersUpdated;
567 }
568
569 [MethodImpl(MethodImplOptions.AggressiveInlining)]
570 public bool Get(long index)
571 {
572     Ensure.Always.ArgumentInRange(index, GetValidIndexRange(), nameof(index));
573     return (_array[GetWordIndexFromIndex(index)] & GetBitMaskFromIndex(index)) != 0;
574 }
575
576 [MethodImpl(MethodImplOptions.AggressiveInlining)]
577 public void Set(long index, bool value)
578 {
579     if (value)
580     {
581         Set(index);
582     }
583     else
584     {
585         Reset(index);
586     }
587 }
588
589 [MethodImpl(MethodImplOptions.AggressiveInlining)]
590 public void Set(long index)
591 {
592     Ensure.Always.ArgumentInRange(index, GetValidIndexRange(), nameof(index));
593     var wordIndex = GetWordIndexFromIndex(index);
594     var mask = GetBitMaskFromIndex(index);
595     _array[wordIndex] |= mask;
596     RefreshBordersByWord(wordIndex);
597 }
598
599 [MethodImpl(MethodImplOptions.AggressiveInlining)]
600 public void Reset(long index)
601 {
602     Ensure.Always.ArgumentInRange(index, GetValidIndexRange(), nameof(index));
603     var wordIndex = GetWordIndexFromIndex(index);
604     var mask = GetBitMaskFromIndex(index);
605     _array[wordIndex] &= ~mask;
606     RefreshBordersByWord(wordIndex);
607 }
608
609 public bool Add(long index)
610 {
611     var wordIndex = GetWordIndexFromIndex(index);
612     var mask = GetBitMaskFromIndex(index);
613     if ((_array[wordIndex] & mask) == 0)
614     {
615         _array[wordIndex] |= mask;
616         RefreshBordersByWord(wordIndex);

```

```

617         return true;
618     }
619     else
620     {
621         return false;
622     }
623 }
624
625 public void SetAll(bool value)
626 {
627     if (value)
628     {
629         SetAll();
630     }
631     else
632     {
633         ResetAll();
634     }
635 }
636
637 public void SetAll()
638 {
639     const long fillValue = unchecked((long)0xffffffffffffffff);
640     var words = GetWordsCountFromIndex(_length);
641     for (var i = 0; i < words; i++)
642     {
643         _array[i] = fillValue;
644     }
645     MarkBordersAsAllBitsSet();
646 }
647
648 public void ResetAll()
649 {
650     const long fillValue = 0;
651     GetBorders(out long from, out long to);
652     for (var i = from; i <= to; i++)
653     {
654         _array[i] = fillValue;
655     }
656     MarkBordersAsAllBitsReset();
657 }
658
659 public List<long> GetSetIndices()
660 {
661     var result = new List<long>();
662     GetBorders(out long from, out long to);
663     for (var i = from; i <= to; i++)
664     {
665         var word = _array[i];
666         if (word != 0)
667         {
668             AppendAllSetBitIndices(result, i, word);
669         }
670     }
671     return result;
672 }
673
674 public List<ulong> GetSetUInt64Indices()
675 {
676     var result = new List<ulong>();
677     GetBorders(out ulong from, out ulong to);
678     for (var i = from; i <= to; i++)
679     {
680         var word = _array[i];
681         if (word != 0)
682         {
683             AppendAllSetBitIndices(result, i, word);
684         }
685     }
686     return result;
687 }
688
689 public long GetFirstSetBitIndex()
690 {
691     var i = _minPositiveWord;
692     var word = _array[i];
693     if (word != 0)
694     {
695         return GetFirstSetBitForWord(i, word);

```

```

696     }
697     return -1;
698 }
699
700 public long GetLastSetBitIndex()
701 {
702     var i = _maxPositiveWord;
703     var word = _array[i];
704     if (word != 0)
705     {
706         return GetLastSetBitForWord(i, word);
707     }
708     return -1;
709 }
710
711 public long CountSetBits()
712 {
713     var total = 0L;
714     GetBorders(out long from, out long to);
715     for (var i = from; i <= to; i++)
716     {
717         var word = _array[i];
718         if (word != 0)
719         {
720             total += CountSetBitsForWord(word);
721         }
722     }
723     return total;
724 }
725
726 public bool HaveCommonBits(BitString other)
727 {
728     EnsureBitStringHasTheSameSize(other, nameof(other));
729     GetCommonInnerBorders(this, other, out long from, out long to);
730     var otherArray = other._array;
731     for (var i = from; i <= to; i++)
732     {
733         var left = _array[i];
734         var right = otherArray[i];
735         if (left != 0 && right != 0 && (left & right) != 0)
736         {
737             return true;
738         }
739     }
740     return false;
741 }
742
743 public long CountCommonBits(BitString other)
744 {
745     EnsureBitStringHasTheSameSize(other, nameof(other));
746     GetCommonInnerBorders(this, other, out long from, out long to);
747     var total = 0L;
748     var otherArray = other._array;
749     for (var i = from; i <= to; i++)
750     {
751         var left = _array[i];
752         var right = otherArray[i];
753         var combined = left & right;
754         if (combined != 0)
755         {
756             total += CountSetBitsForWord(combined);
757         }
758     }
759     return total;
760 }
761
762 public List<long> GetCommonIndices(BitString other)
763 {
764     EnsureBitStringHasTheSameSize(other, nameof(other));
765     GetCommonInnerBorders(this, other, out long from, out long to);
766     var result = new List<long>();
767     var otherArray = other._array;
768     for (var i = from; i <= to; i++)
769     {
770         var left = _array[i];
771         var right = otherArray[i];
772         var combined = left & right;
773         if (combined != 0)
774         {

```

```

775         AppendAllSetBitIndices(result, i, combined);
776     }
777 }
778 return result;
779 }
780
781 public List<ulong> GetCommonUInt64Indices(BitString other)
782 {
783     EnsureBitStringHasTheSameSize(other, nameof(other));
784     GetCommonBorders(this, other, out ulong from, out ulong to);
785     var result = new List<ulong>();
786     var otherArray = other._array;
787     for (var i = from; i <= to; i++)
788     {
789         var left = _array[i];
790         var right = otherArray[i];
791         var combined = left & right;
792         if (combined != 0)
793         {
794             AppendAllSetBitIndices(result, i, combined);
795         }
796     }
797     return result;
798 }
799
800 public long GetFirstCommonBitIndex(BitString other)
801 {
802     EnsureBitStringHasTheSameSize(other, nameof(other));
803     GetCommonInnerBorders(this, other, out long from, out long to);
804     var otherArray = other._array;
805     for (var i = from; i <= to; i++)
806     {
807         var left = _array[i];
808         var right = otherArray[i];
809         var combined = left & right;
810         if (combined != 0)
811         {
812             return GetFirstSetBitForWord(i, combined);
813         }
814     }
815     return -1;
816 }
817
818 public long GetLastCommonBitIndex(BitString other)
819 {
820     EnsureBitStringHasTheSameSize(other, nameof(other));
821     GetCommonInnerBorders(this, other, out long from, out long to);
822     var otherArray = other._array;
823     for (var i = to; i >= from; i--)
824     {
825         var left = _array[i];
826         var right = otherArray[i];
827         var combined = left & right;
828         if (combined != 0)
829         {
830             return GetLastSetBitForWord(i, combined);
831         }
832     }
833     return -1;
834 }
835
836 public override bool Equals(object obj) => obj is BitString @string ? Equals(@string) :
837     ↪ false;
838
839 public bool Equals(BitString other)
840 {
841     if (_length != other._length)
842     {
843         return false;
844     }
845     var otherArray = other._array;
846     if (_array.Length != otherArray.Length)
847     {
848         return false;
849     }
850     if (_minPositiveWord != other._minPositiveWord)
851     {
852         return false;
853     }

```

```

853     if (_maxPositiveWord != other._maxPositiveWord)
854     {
855         return false;
856     }
857     GetCommonBorders(this, other, out ulong from, out ulong to);
858     for (var i = from; i <= to; i++)
859     {
860         if (_array[i] != otherArray[i])
861         {
862             return false;
863         }
864     }
865     return true;
866 }
867
868 [MethodImpl(MethodImplOptions.AggressiveInlining)]
869 private void EnsureBitStringHasTheSameSize(BitString other, string argumentName)
870 {
871     Ensure.Always.ArgumentNotNull(other, argumentName);
872     if (_length != other._length)
873     {
874         throw new ArgumentException("Bit string must be the same size.", argumentName);
875     }
876 }
877
878 [MethodImpl(MethodImplOptions.AggressiveInlining)]
879 private void MarkBordersAsAllBitsReset() => SetBorders(_array.LongLength - 1, 0);
880
881 [MethodImpl(MethodImplOptions.AggressiveInlining)]
882 private void MarkBordersAsAllBitsSet() => SetBorders(0, _array.LongLength - 1);
883
884 [MethodImpl(MethodImplOptions.AggressiveInlining)]
885 private void GetBorders(out long from, out long to)
886 {
887     from = _minPositiveWord;
888     to = _maxPositiveWord;
889 }
890
891 [MethodImpl(MethodImplOptions.AggressiveInlining)]
892 private void GetBorders(out ulong from, out ulong to)
893 {
894     from = (ulong)_minPositiveWord;
895     to = (ulong)_maxPositiveWord;
896 }
897
898 [MethodImpl(MethodImplOptions.AggressiveInlining)]
899 private void SetBorders(long from, long to)
900 {
901     _minPositiveWord = from;
902     _maxPositiveWord = to;
903 }
904
905 [MethodImpl(MethodImplOptions.AggressiveInlining)]
906 private Range<long> GetValidIndexRange() => (0, _length - 1);
907
908 [MethodImpl(MethodImplOptions.AggressiveInlining)]
909 private static Range<long> GetValidLengthRange() => (0, long.MaxValue);
910
911 [MethodImpl(MethodImplOptions.AggressiveInlining)]
912 private static void AppendAllSetBitIndices(List<ulong> result, ulong wordIndex, long
↵ wordValue)
913 {
914     GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
↵ bits32to47, out byte[] bits48to63);
915     AppendAllSetIndices(result, wordIndex, bits00to15, bits16to31, bits32to47,
↵ bits48to63);
916 }
917
918 [MethodImpl(MethodImplOptions.AggressiveInlining)]
919 private static void AppendAllSetBitIndices(List<long> result, long wordIndex, long
↵ wordValue)
920 {
921     GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
↵ bits32to47, out byte[] bits48to63);
922     AppendAllSetBitIndices(result, wordIndex, bits00to15, bits16to31, bits32to47,
↵ bits48to63);
923 }
924
925 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```



```

926 private static long CountSetBitsForWord(long word)
927 {
928     GetBits(word, out byte[] bits00to15, out byte[] bits16to31, out byte[] bits32to47,
929         ↪ out byte[] bits48to63);
930     return bits00to15.LongLength + bits16to31.LongLength + bits32to47.LongLength +
931         ↪ bits48to63.LongLength;
932 }
933 [MethodImpl(MethodImplOptions.AggressiveInlining)]
934 private static long GetFirstSetBitForWord(long wordIndex, long wordValue)
935 {
936     GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
937         ↪ bits32to47, out byte[] bits48to63);
938     return GetFirstSetBit(wordIndex, bits00to15, bits16to31, bits32to47, bits48to63);
939 }
940 [MethodImpl(MethodImplOptions.AggressiveInlining)]
941 private static long GetLastSetBitForWord(long wordIndex, long wordValue)
942 {
943     GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
944         ↪ bits32to47, out byte[] bits48to63);
945     return GetLastSetBit(wordIndex, bits00to15, bits16to31, bits32to47, bits48to63);
946 }
947 private static void AppendAllSetBitIndices(List<long> result, long i, byte[] bits00to15,
948     ↪ byte[] bits16to31, byte[] bits32to47, byte[] bits48to63)
949 {
950     for (var j = 0; j < bits00to15.Length; j++)
951     {
952         result.Add(bits00to15[j] + (i * 64));
953     }
954     for (var j = 0; j < bits16to31.Length; j++)
955     {
956         result.Add(bits16to31[j] + 16 + (i * 64));
957     }
958     for (var j = 0; j < bits32to47.Length; j++)
959     {
960         result.Add(bits32to47[j] + 32 + (i * 64));
961     }
962     for (var j = 0; j < bits48to63.Length; j++)
963     {
964         result.Add(bits48to63[j] + 48 + (i * 64));
965     }
966 }
967 private static void AppendAllSetIndices(List<ulong> result, ulong i, byte[] bits00to15,
968     ↪ byte[] bits16to31, byte[] bits32to47, byte[] bits48to63)
969 {
970     for (var j = 0; j < bits00to15.Length; j++)
971     {
972         result.Add(bits00to15[j] + (i * 64));
973     }
974     for (var j = 0; j < bits16to31.Length; j++)
975     {
976         result.Add(bits16to31[j] + 16UL + (i * 64));
977     }
978     for (var j = 0; j < bits32to47.Length; j++)
979     {
980         result.Add(bits32to47[j] + 32UL + (i * 64));
981     }
982     for (var j = 0; j < bits48to63.Length; j++)
983     {
984         result.Add(bits48to63[j] + 48UL + (i * 64));
985     }
986 }
987 private static long GetFirstSetBit(long i, byte[] bits00to15, byte[] bits16to31, byte[]
988     ↪ bits32to47, byte[] bits48to63)
989 {
990     if (bits00to15.Length > 0)
991     {
992         return bits00to15[0] + (i * 64);
993     }
994     if (bits16to31.Length > 0)
995     {
996         return bits16to31[0] + 16 + (i * 64);
997     }
998 }

```

```

996         if (bits32to47.Length > 0)
997         {
998             return bits32to47[0] + 32 + (i * 64);
999         }
1000         return bits48to63[0] + 48 + (i * 64);
1001     }
1002
1003     private static long GetLastSetBit(long i, byte[] bits00to15, byte[] bits16to31, byte[]
    ↪ bits32to47, byte[] bits48to63)
1004     {
1005         if (bits48to63.Length > 0)
1006         {
1007             return bits48to63[bits48to63.Length - 1] + 48 + (i * 64);
1008         }
1009         if (bits32to47.Length > 0)
1010         {
1011             return bits32to47[bits32to47.Length - 1] + 32 + (i * 64);
1012         }
1013         if (bits16to31.Length > 0)
1014         {
1015             return bits16to31[bits16to31.Length - 1] + 16 + (i * 64);
1016         }
1017         return bits00to15[bits00to15.Length - 1] + (i * 64);
1018     }
1019
1020     private static void GetBits(long word, out byte[] bits00to15, out byte[] bits16to31, out
    ↪ byte[] bits32to47, out byte[] bits48to63)
1021     {
1022         bits00to15 = _bitsSetIn16Bits[word & 0xffffu];
1023         bits16to31 = _bitsSetIn16Bits[(word >> 16) & 0xffffu];
1024         bits32to47 = _bitsSetIn16Bits[(word >> 32) & 0xffffu];
1025         bits48to63 = _bitsSetIn16Bits[(word >> 48) & 0xffffu];
1026     }
1027
1028     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1029     public static void GetCommonInnerBorders(BitString left, BitString right, out long from,
    ↪ out long to)
1030     {
1031         from = Math.Max(left._minPositiveWord, right._minPositiveWord);
1032         to = Math.Min(left._maxPositiveWord, right._maxPositiveWord);
1033     }
1034
1035     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1036     public static void GetCommonOuterBorders(BitString left, BitString right, out long from,
    ↪ out long to)
1037     {
1038         from = Math.Min(left._minPositiveWord, right._minPositiveWord);
1039         to = Math.Max(left._maxPositiveWord, right._maxPositiveWord);
1040     }
1041
1042     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1043     public static void GetCommonBorders(BitString left, BitString right, out ulong from, out
    ↪ ulong to)
1044     {
1045         from = (ulong)Math.Max(left._minPositiveWord, right._minPositiveWord);
1046         to = (ulong)Math.Min(left._maxPositiveWord, right._maxPositiveWord);
1047     }
1048
1049     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1050     public static long GetWordsCountFromIndex(long index) => (index + 63) / 64;
1051
1052     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1053     public static long GetWordIndexFromIndex(long index) => index >> 6;
1054
1055     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1056     public static long GetBitMaskFromIndex(long index) => 1L << (int)(index & 63);
1057
1058     public override int GetHashCode() => base.GetHashCode();
1059
1060     public override string ToString() => base.ToString();
1061 }
1062 }

```

1.10 ./Platform.Collections/BitStringExtensions.cs

```

1  using Platform.Random;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4

```

```

5 namespace Platform.Collections
6 {
7     public static class BitStringExtensions
8     {
9         public static void SetRandomBits(this BitString @string)
10        {
11            for (var i = 0; i < @string.Length; i++)
12            {
13                var value = RandomHelpers.Default.NextBoolean();
14                @string.Set(i, value);
15            }
16        }
17    }
18 }

```

1.11 ./Platform.Collections/Concurrent/ConcurrentQueueExtensions.cs

```

1 using System.Collections.Concurrent;
2 using System.Collections.Generic;
3 using System.Runtime.CompilerServices;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Collections.Concurrent
8 {
9     public static class ConcurrentQueueExtensions
10    {
11        [MethodImpl(MethodImplOptions.AggressiveInlining)]
12        public static IEnumerable<T> DequeueAll<T>(this ConcurrentQueue<T> queue)
13        {
14            while (queue.TryDequeue(out T item))
15            {
16                yield return item;
17            }
18        }
19    }
20 }

```

1.12 ./Platform.Collections/Concurrent/ConcurrentStackExtensions.cs

```

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

```

1.13 ./Platform.Collections/EnsureExtensions.cs

```

1 using System;
2 using System.Collections.Generic;
3 using System.Diagnostics;
4 using System.Runtime.CompilerServices;
5 using Platform.Exceptions;
6 using Platform.Exceptions.ExtensionRoots;
7
8 #pragma warning disable IDE0060 // Remove unused parameter
9 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11 namespace Platform.Collections
12 {
13     public static class EnsureExtensions
14     {
15         #region Always
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public static void ArgumentNotEmpty<T>(this EnsureAlwaysExtensionRoot root,
19         ↪ ICollection<T> argument, string argumentName, string message)
20         {
21             if (argument.IsNullOrEmpty())

```

```

21     {
22         throw new ArgumentException(message, argumentName);
23     }
24 }
25
26 [MethodImpl(MethodImplOptions.AggressiveInlining)]
27 public static void ArgumentNotEmpty<T>(this EnsureAlwaysExtensionRoot root,
    ↪ ICollection<T> argument, string argumentName) => ArgumentNotEmpty(root, argument,
    ↪ argumentName, null);
28
29 [MethodImpl(MethodImplOptions.AggressiveInlining)]
30 public static void ArgumentNotEmpty<T>(this EnsureAlwaysExtensionRoot root,
    ↪ ICollection<T> argument) => ArgumentNotEmpty(root, argument, null, null);
31
32 [MethodImpl(MethodImplOptions.AggressiveInlining)]
33 public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureAlwaysExtensionRoot root,
    ↪ string argument, string argumentName, string message)
34 {
35     if (string.IsNullOrEmpty(argument))
36     {
37         throw new ArgumentException(message, argumentName);
38     }
39 }
40
41 [MethodImpl(MethodImplOptions.AggressiveInlining)]
42 public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureAlwaysExtensionRoot root,
    ↪ string argument, string argumentName) => ArgumentNotEmptyAndNotWhiteSpace(root,
    ↪ argument, argumentName, null);
43
44 [MethodImpl(MethodImplOptions.AggressiveInlining)]
45 public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureAlwaysExtensionRoot root,
    ↪ string argument) => ArgumentNotEmptyAndNotWhiteSpace(root, argument, null, null);
46
47 #endregion
48
49 #region OnDebug
50
51 [Conditional("DEBUG")]
52 public static void ArgumentNotEmpty<T>(this EnsureOnDebugExtensionRoot root,
    ↪ ICollection<T> argument, string argumentName, string message) =>
    ↪ Ensure.Always.ArgumentNotEmpty(argument, argumentName, message);
53
54 [Conditional("DEBUG")]
55 public static void ArgumentNotEmpty<T>(this EnsureOnDebugExtensionRoot root,
    ↪ ICollection<T> argument, string argumentName) =>
    ↪ Ensure.Always.ArgumentNotEmpty(argument, argumentName, null);
56
57 [Conditional("DEBUG")]
58 public static void ArgumentNotEmpty<T>(this EnsureOnDebugExtensionRoot root,
    ↪ ICollection<T> argument) => Ensure.Always.ArgumentNotEmpty(argument, null, null);
59
60 [Conditional("DEBUG")]
61 public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureOnDebugExtensionRoot
    ↪ root, string argument, string argumentName, string message) =>
    ↪ Ensure.Always.ArgumentNotEmptyAndNotWhiteSpace(argument, argumentName, message);
62
63 [Conditional("DEBUG")]
64 public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureOnDebugExtensionRoot
    ↪ root, string argument, string argumentName) =>
    ↪ Ensure.Always.ArgumentNotEmptyAndNotWhiteSpace(argument, argumentName, null);
65
66 [Conditional("DEBUG")]
67 public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureOnDebugExtensionRoot
    ↪ root, string argument) => Ensure.Always.ArgumentNotEmptyAndNotWhiteSpace(argument,
    ↪ null, null);
68
69 #endregion
70 }
71 }

```

1.14 ./Platform.Collections/ICollectionExtensions.cs

```

1 using System.Collections.Generic;
2 using System.Linq;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Collections
7 {

```

```

8     public static class ICollectionExtensions
9     {
10         public static bool IsNullOrEmpty<T>(this ICollection<T> collection) => collection ==
            ↳ null || collection.Count == 0;
11
12         public static bool AllEqualToDefault<T>(this ICollection<T> collection)
13         {
14             var equalityComparer = EqualityComparer<T>.Default;
15             return collection.All(item => equalityComparer.Equals(item, default));
16         }
17     }
18 }

```

1.15 ./Platform.Collections/IDictionaryExtensions.cs

```

1 using System;
2 using System.Collections.Generic;
3 using System.Runtime.CompilerServices;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Collections
8 {
9     public static class IDictionaryExtensions
10    {
11        [MethodImpl(MethodImplOptions.AggressiveInlining)]
12        public static TValue GetOrDefault<TKey, TValue>(this IDictionary<TKey, TValue>
            ↳ dictionary, TKey key)
13        {
14            dictionary.TryGetValue(key, out TValue value);
15            return value;
16        }
17
18        [MethodImpl(MethodImplOptions.AggressiveInlining)]
19        public static TValue GetOrAdd<TKey, TValue>(this IDictionary<TKey, TValue> dictionary,
            ↳ TKey key, Func<TKey, TValue> valueFactory)
20        {
21            if (!dictionary.TryGetValue(key, out TValue value))
22            {
23                value = valueFactory(key);
24                dictionary.Add(key, value);
25                return value;
26            }
27            return value;
28        }
29    }
30 }

```

1.16 ./Platform.Collections/Lists/CharListExtensions.cs

```

1 using System.Collections.Generic;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Collections.Lists
6 {
7     public static class CharListExtensions
8     {
9         /// <remarks>
10        /// Based on https://github.com/Microsoft/referencesource/blob/3b1eaf5203992df69de44c783\_
            ↳ a3eda37d3d4cd10/mscorlib/system/string.cs#L833
11        /// </remarks>
12        public static unsafe int GenerateHashCode(this IList<char> list)
13        {
14            var hashSeed = 5381;
15            var hashAccumulator = hashSeed;
16            for (var i = 0; i < list.Count; i++)
17            {
18                hashAccumulator = (hashAccumulator << 5) + hashAccumulator ^ list[i];
19            }
20            return hashAccumulator + (hashSeed * 1566083941);
21        }
22
23        public static bool EqualTo(this IList<char> left, IList<char> right) =>
            ↳ left.EqualTo(right, ContentEqualTo);
24
25        public static bool ContentEqualTo(this IList<char> left, IList<char> right)
26        {
27            for (var i = left.Count - 1; i >= 0; --i)
28            {
29                if (left[i] != right[i])

```

```

30         {
31             return false;
32         }
33     }
34     return true;
35 }
36 }
37 }

```

1.17 ./Platform.Collections/Lists/IListComparer.cs

```

1 using System.Collections.Generic;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Collections.Lists
6 {
7     public class IListComparer<T> : IComparer<IList<T>>
8     {
9         public int Compare(IList<T> left, IList<T> right) => left.CompareTo(right);
10    }
11 }

```

1.18 ./Platform.Collections/Lists/IListEqualityComparer.cs

```

1 using System.Collections.Generic;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Collections.Lists
6 {
7     public class IListEqualityComparer<T> : IEqualityComparer<IList<T>>
8     {
9         public bool Equals(IList<T> left, IList<T> right) => left.EqualTo(right);
10        public int GetHashCode(IList<T> list) => list.GenerateHashCode();
11    }
12 }

```

1.19 ./Platform.Collections/Lists/IListExtensions.cs

```

1 using System;
2 using System.Collections.Generic;
3 using System.Runtime.CompilerServices;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Collections.Lists
8 {
9     public static class IListExtensions
10    {
11        public static bool AddAndReturnTrue<T>(this IList<T> list, T element)
12        {
13            list.Add(element);
14            return true;
15        }
16
17        public static int GetCountOrZero<T>(this IList<T> list) => list?.Count ?? 0;
18
19        public static bool EqualTo<T>(this IList<T> left, IList<T> right) => EqualTo(left,
20        ↪ right, ContentEqualTo);
21
22        public static bool EqualTo<T>(this IList<T> left, IList<T> right, Func<IList<T>,
23        ↪ IList<T>, bool> contentEqualityComparer)
24        {
25            if (ReferenceEquals(left, right))
26            {
27                return true;
28            }
29            var leftCount = left.GetCountOrZero();
30            var rightCount = right.GetCountOrZero();
31            if (leftCount == 0 && rightCount == 0)
32            {
33                return true;
34            }
35            if (leftCount == 0 || rightCount == 0 || leftCount != rightCount)
36            {
37                return false;
38            }
39            return contentEqualityComparer(left, right);
40        }
41
42        public static bool ContentEqualTo<T>(this IList<T> left, IList<T> right)

```

```

41 {
42     var equalityComparer = EqualityComparer<T>.Default;
43     for (var i = left.Count - 1; i >= 0; --i)
44     {
45         if (!equalityComparer.Equals(left[i], right[i]))
46         {
47             return false;
48         }
49     }
50     return true;
51 }
52
53 public static T[] ToArray<T>(this IList<T> list, Func<T, bool> predicate)
54 {
55     if (list == null)
56     {
57         return null;
58     }
59     var result = new List<T>(list.Count);
60     for (var i = 0; i < list.Count; i++)
61     {
62         if (predicate(list[i]))
63         {
64             result.Add(list[i]);
65         }
66     }
67     return result.ToArray();
68 }
69
70 public static T[] ToArray<T>(this IList<T> list)
71 {
72     var array = new T[list.Count];
73     list.CopyTo(array, 0);
74     return array;
75 }
76
77 public static void ForEach<T>(this IList<T> list, Action<T> action)
78 {
79     for (var i = 0; i < list.Count; i++)
80     {
81         action(list[i]);
82     }
83 }
84
85 /// <remarks>
86 /// Based on http://stackoverflow.com/questions/263400/what-is-the-best-algorithm-for-an
87 /// ↪ -overridden-system-object-gethashcode
88 /// </remarks>
89 public static int GenerateHashCode<T>(this IList<T> list)
90 {
91     var result = 17;
92     for (var i = 0; i < list.Count; i++)
93     {
94         result = unchecked((result * 23) + list[i].GetHashCode());
95     }
96     return result;
97 }
98
99 public static int CompareTo<T>(this IList<T> left, IList<T> right)
100 {
101     var comparer = Comparer<T>.Default;
102     var leftCount = left.GetCountOrZero();
103     var rightCount = right.GetCountOrZero();
104     var intermediateResult = leftCount.CompareTo(rightCount);
105     for (var i = 0; intermediateResult == 0 && i < leftCount; i++)
106     {
107         intermediateResult = comparer.Compare(left[i], right[i]);
108     }
109     return intermediateResult;
110 }
111
112 [MethodImpl(MethodImplOptions.AggressiveInlining)]
113 public static TLink[] SkipFirst<TLink>(this IList<TLink> list) => list.SkipFirst(1);
114
115 [MethodImpl(MethodImplOptions.AggressiveInlining)]
116 public static TLink[] SkipFirst<TLink>(this IList<TLink> list, int skip)
117 {
118     if (list.IsNullOrEmpty() || list.Count <= skip)
119     {

```

```

119         return Array.Empty<TLink>();
120     }
121     var result = new TLink[list.Count - skip];
122     for (int r = skip, w = 0; r < list.Count; r++, w++)
123     {
124         result[w] = list[r];
125     }
126     return result;
127 }
128
129 [MethodImpl(MethodImplOptions.AggressiveInlining)]
130 public static IList<TLink> ShiftRight<TLink>(this IList<TLink> list) =>
131     ↳ list.ShiftRight(1);
132
133 [MethodImpl(MethodImplOptions.AggressiveInlining)]
134 public static IList<TLink> ShiftRight<TLink>(this IList<TLink> list, int shift)
135 {
136     var result = new TLink[list.Count + shift];
137     for (int r = 0, w = shift; r < list.Count; r++, w++)
138     {
139         result[w] = list[r];
140     }
141     return result;
142 }
143 }

```

1.20 ./Platform.Collections/Lists/ListFiller.cs

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Collections.Lists
7  {
8      public class ListFiller<TElement, TReturnConstant>
9      {
10         protected readonly List<TElement> _list;
11         protected readonly TReturnConstant _returnConstant;
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public ListFiller(List<TElement> list, TReturnConstant returnConstant)
15         {
16             _list = list;
17             _returnConstant = returnConstant;
18         }
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         public ListFiller(List<TElement> list) : this(list, default) { }
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         public void Add(TElement element) => _list.Add(element);
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public bool AddAndReturnTrue(TElement element)
28         {
29             _list.Add(element);
30             return true;
31         }
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         public bool AddFirstAndReturnTrue(IList<TElement> list)
35         {
36             _list.Add(list[0]);
37             return true;
38         }
39
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         public TReturnConstant AddAndReturnConstant(TElement element)
42         {
43             _list.Add(element);
44             return _returnConstant;
45         }
46
47         [MethodImpl(MethodImplOptions.AggressiveInlining)]
48         public TReturnConstant AddFirstAndReturnConstant(IList<TElement> list)
49         {
50             _list.Add(list[0]);
51             return _returnConstant;
52         }

```



```

53     [MethodImpl(MethodImplOptions.AggressiveInlining)]
54     public TReturnConstant AddAllValuesAndReturnConstant(IList<TElement> list)
55     {
56         for (int i = 1; i < list.Count; i++)
57         {
58             _list.Add(list[i]);
59         }
60         return _returnConstant;
61     }
62 }
63 }
64 }

```

1.21 ./Platform.Collections/Segments/CharSegment.cs

```

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

```

1.22 ./Platform.Collections/Segments/Segment.cs

```

1  using System;
2  using System.Collections;
3  using System.Collections.Generic;
4  using Platform.Collections.Lists;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7

```

```

8 namespace Platform.Collections.Segments
9 {
10     public class Segment<T> : IEquatable<Segment<T>>, IList<T>
11     {
12         public IList<T> Base { get; }
13         public int Offset { get; }
14         public int Length { get; }
15
16         public Segment(IList<T> @base, int offset, int length)
17         {
18             Base = @base;
19             Offset = offset;
20             Length = length;
21         }
22
23         public override int GetHashCode() => this.GenerateHashCode();
24
25         public virtual bool Equals(Segment<T> other) => this.EqualTo(other);
26
27         public override bool Equals(object obj) => obj is Segment<T> other ? Equals(other) :
28             ↪ false;
29
30         #region IList
31         public T this[int i]
32         {
33             get => Base[Offset + i];
34             set => Base[Offset + i] = value;
35         }
36
37         public int Count => Length;
38
39         public bool IsReadOnly => true;
40
41         public int IndexOf(T item)
42         {
43             var index = Base.IndexOf(item);
44             if (index >= Offset)
45             {
46                 var actualIndex = index - Offset;
47                 if (actualIndex < Length)
48                 {
49                     return actualIndex;
50                 }
51             }
52             return -1;
53         }
54
55         public void Insert(int index, T item) => throw new NotSupportedException();
56
57         public void RemoveAt(int index) => throw new NotSupportedException();
58
59         public void Add(T item) => throw new NotSupportedException();
60
61         public void Clear() => throw new NotSupportedException();
62
63         public bool Contains(T item) => IndexOf(item) >= 0;
64
65         public void CopyTo(T[] array, int arrayIndex)
66         {
67             for (var i = 0; i < Length; i++)
68             {
69                 array[arrayIndex++] = this[i];
70             }
71         }
72
73         public bool Remove(T item) => throw new NotSupportedException();
74
75         public IEnumerator<T> GetEnumerator()
76         {
77             for (var i = 0; i < Length; i++)
78             {
79                 yield return this[i];
80             }
81         }
82
83         IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
84
85         #endregion
86     }

```

```
87 }
```

1.23 ./Platform.Collections/Segments/Walkers/AllSegmentsWalkerBase.cs

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

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

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

1.25 ./Platform.Collections/Segments/Walkers/AllSegmentsWalkerBase[T].cs

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

1.26 ./Platform.Collections/Segments/Walkers/AllSegmentsWalkerExtensions.cs

```
1 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3 namespace Platform.Collections.Segments.Walkers
4 {
5     public static class AllSegmentsWalkerExtensions
6     {
7         public static void WalkAll(this AllSegmentsWalkerBase<char> walker, string @string) =>
8             ↪ walker.WalkAll(@string.ToCharArray());
9         public static void WalkAll<TSegment>(this AllSegmentsWalkerBase<char, TSegment> walker,
10             ↪ string @string) where TSegment : Segment<char> =>
11             ↪ walker.WalkAll(@string.ToCharArray());
12 }
```

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

```

1 using System;
2 using System.Collections.Generic;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Collections.Segments.Walkers
7 {
8     public abstract class DictionaryBasedDuplicateSegmentsWalkerBase<T, TSegment> :
9         ↳ DuplicateSegmentsWalkerBase<T, TSegment>
10        where TSegment : Segment<T>
11    {
12        public static readonly bool DefaultResetDictionaryOnEachWalk;
13
14        private readonly bool _resetDictionaryOnEachWalk;
15        protected IDictionary<TSegment, long> Dictionary;
16
17        protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<TSegment, long>
18            ↳ dictionary, int minimumStringSegmentLength, bool resetDictionaryOnEachWalk)
19            : base(minimumStringSegmentLength)
20        {
21            Dictionary = dictionary;
22            _resetDictionaryOnEachWalk = resetDictionaryOnEachWalk;
23        }
24
25        protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<TSegment, long>
26            ↳ dictionary, int minimumStringSegmentLength) : this(dictionary,
27            ↳ minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
28
29        protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<TSegment, long>
30            ↳ dictionary) : this(dictionary, DefaultMinimumStringSegmentLength,
31            ↳ DefaultResetDictionaryOnEachWalk) { }
32
33        protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength,
34            ↳ bool resetDictionaryOnEachWalk) : this(resetDictionaryOnEachWalk ? null : new
35            ↳ Dictionary<TSegment, long>(), minimumStringSegmentLength, resetDictionaryOnEachWalk)
36            ↳ { }
37
38        protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength) :
39            ↳ this(minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
40
41        protected DictionaryBasedDuplicateSegmentsWalkerBase() :
42            ↳ this(DefaultMinimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
43
44        public override void WalkAll(ICollection<T> elements)
45        {
46            if (_resetDictionaryOnEachWalk)
47            {
48                var capacity = Math.Ceiling(Math.Pow(elements.Count, 2) / 2);
49                Dictionary = new Dictionary<TSegment, long>((int)capacity);
50            }
51            base.WalkAll(elements);
52        }
53
54        protected override long GetSegmentFrequency(TSegment segment) =>
55            ↳ Dictionary.GetOrDefault(segment);
56
57        protected override void SetSegmentFrequency(TSegment segment, long frequency) =>
58            ↳ Dictionary[segment] = frequency;
59    }
60 }

```

1.28 ./Platform.Collections/Segments/Walkers/DictionaryBasedDuplicateSegmentsWalkerBase[T].cs

```

1 using System.Collections.Generic;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Collections.Segments.Walkers
6 {
7     public abstract class DictionaryBasedDuplicateSegmentsWalkerBase<T> :
8         ↳ DictionaryBasedDuplicateSegmentsWalkerBase<T, Segment<T>>
9    {
10        protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<Segment<T>, long>
11            ↳ dictionary, int minimumStringSegmentLength, bool resetDictionaryOnEachWalk) :
12            ↳ base(dictionary, minimumStringSegmentLength, resetDictionaryOnEachWalk) { }
13        protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<Segment<T>, long>
14            ↳ dictionary, int minimumStringSegmentLength) : base(dictionary,
15            ↳ minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
16    }

```

```

11     protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<Segment<T>, long>
    ↪ dictionary) : base(dictionary, DefaultMinimumStringSegmentLength,
    ↪ DefaultResetDictionaryOnEachWalk) { }
12     protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength,
    ↪ bool resetDictionaryOnEachWalk) : base(minimumStringSegmentLength,
    ↪ resetDictionaryOnEachWalk) { }
13     protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength) :
    ↪ base(minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
14     protected DictionaryBasedDuplicateSegmentsWalkerBase() :
    ↪ base(DefaultMinimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
15 }
16 }

```

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

```

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

```

1.30 ./Platform.Collections/Segments/Walkers/DuplicateSegmentsWalkerBase[T].cs

```

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

```

1.31 ./Platform.Collections/Sets/ISetExtensions.cs

```

1 using System.Collections.Generic;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Collections.Sets
6 {
7     public static class ISetExtensions
8     {
9         public static void AddAndReturnVoid<T>(this ISet<T> set, T element) => set.Add(element);
10        public static void RemoveAndReturnVoid<T>(this ISet<T> set, T element) =>
    ↪ set.Remove(element);
11        public static bool DoNotContains<T>(this ISet<T> set, T element) =>
    ↪ !set.Contains(element);
12    }
13 }

```

1.32 ./Platform.Collections/Sets/SetFiller.cs

```

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

```

```

6 namespace Platform.Collections.Sets
7 {
8     public class SetFiller<TElement, TReturnConstant>
9     {
10         protected readonly ISet<TElement> _set;
11         protected readonly TReturnConstant _returnConstant;
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public SetFiller(ISet<TElement> set, TReturnConstant returnConstant)
15         {
16             _set = set;
17             _returnConstant = returnConstant;
18         }
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         public SetFiller(ISet<TElement> set) : this(set, default) { }
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         public void Add(TElement element) => _set.Add(element);
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public bool AddAndReturnTrue(TElement element)
28         {
29             _set.Add(element);
30             return true;
31         }
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         public bool AddFirstAndReturnTrue(ICollection<TElement> list)
35         {
36             _set.Add(list[0]);
37             return true;
38         }
39
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         public TReturnConstant AddAndReturnConstant(TElement element)
42         {
43             _set.Add(element);
44             return _returnConstant;
45         }
46
47         [MethodImpl(MethodImplOptions.AggressiveInlining)]
48         public TReturnConstant AddFirstAndReturnConstant(ICollection<TElement> list)
49         {
50             _set.Add(list[0]);
51             return _returnConstant;
52         }
53     }
54 }

```

1.33 ./Platform.Collections/Stacks/DefaultStack.cs

```

1 using System.Collections.Generic;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Collections.Stacks
6 {
7     public class DefaultStack<TElement> : Stack<TElement>, IStack<TElement>
8     {
9         public bool IsEmpty => Count <= 0;
10     }
11 }

```

1.34 ./Platform.Collections/Stacks/IStack.cs

```

1 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3 namespace Platform.Collections.Stacks
4 {
5     public interface IStack<TElement>
6     {
7         bool IsEmpty { get; }
8         void Push(TElement element);
9         TElement Pop();
10        TElement Peek();
11    }
12 }

```

1.35 ./Platform.Collections/Stacks/IStackExtensions.cs

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

1.36 ./Platform.Collections/Stacks/IStackFactory.cs

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

1.37 ./Platform.Collections/Stacks/StackExtensions.cs

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

1.38 ./Platform.Collections/StringExtensions.cs

```
1 using System;
2 using System.Globalization;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Collections
7 {
8     public static class StringExtensions
9     {
10        public static string CapitalizeFirstLetter(this string @string)
11        {
12            if (@string.IsNullOrEmpty(@string))
13            {
14                return @string;
15            }
16            var chars = @string.ToCharArray();
17            for (var i = 0; i < chars.Length; i++)
18            {
19                var category = char.GetUnicodeCategory(chars[i]);
```

```

20         if (category == UnicodeCategory.UppercaseLetter)
21         {
22             return @string;
23         }
24         if (category == UnicodeCategory.LowercaseLetter)
25         {
26             chars[i] = char.ToUpper(chars[i]);
27             return new string(chars);
28         }
29     }
30     return @string;
31 }
32
33 public static string Truncate(this string @string, int maxLength) =>
34     ↪ string.IsNullOrEmpty(@string) ? @string : @string.Substring(0,
35     ↪ Math.Min(@string.Length, maxLength));
36
37 public static string TrimSingle(this string @string, char charToTrim)
38 {
39     if (!string.IsNullOrEmpty(@string))
40     {
41         if (@string.Length == 1)
42         {
43             if (@string[0] == charToTrim)
44             {
45                 return "";
46             }
47             else
48             {
49                 return @string;
50             }
51         }
52         else
53         {
54             var left = 0;
55             var right = @string.Length - 1;
56             if (@string[left] == charToTrim)
57             {
58                 left++;
59             }
60             if (@string[right] == charToTrim)
61             {
62                 right--;
63             }
64             return @string.Substring(left, right - left + 1);
65         }
66     }
67     else
68     {
69         return @string;
70     }
71 }

```

1.39 ./Platform.Collections/Trees/Node.cs

```

1 using System.Collections.Generic;
2
3 // ReSharper disable ForCanBeConvertedToForeach
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Collections.Trees
7 {
8     public class Node
9     {
10         private Dictionary<object, Node> _childNodes;
11
12         public object Value { get; set; }
13
14         public Dictionary<object, Node> ChildNodes => _childNodes ?? (_childNodes = new
15             ↪ Dictionary<object, Node>());
16
17         public Node this[object key]
18         {
19             get
20             {
21                 var child = GetChild(key);
22                 if (child == null)
23                 {
24                     child = AddChild(key);
25                 }
26             }
27         }
28     }
29 }

```



```

24         }
25         return child;
26     }
27     set => SetChildValue(value, key);
28 }
29
30 public Node(object value) => Value = value;
31
32 public Node() : this(null) { }
33
34 public bool ContainsChild(params object[] keys) => GetChild(keys) != null;
35
36 public Node GetChild(params object[] keys)
37 {
38     var node = this;
39     for (var i = 0; i < keys.Length; i++)
40     {
41         node.ChildNodes.TryGetValue(keys[i], out node);
42         if (node == null)
43         {
44             return null;
45         }
46     }
47     return node;
48 }
49
50 public object GetChildValue(params object[] keys) => GetChild(keys)?.Value;
51
52 public Node AddChild(object key) => AddChild(key, new Node(null));
53
54 public Node AddChild(object key, object value) => AddChild(key, new Node(value));
55
56 public Node AddChild(object key, Node child)
57 {
58     ChildNodes.Add(key, child);
59     return child;
60 }
61
62 public Node SetChild(params object[] keys) => SetChildValue(null, keys);
63
64 public Node SetChild(object key) => SetChildValue(null, key);
65
66 public Node SetChildValue(object value, params object[] keys)
67 {
68     var node = this;
69     for (var i = 0; i < keys.Length; i++)
70     {
71         node = SetChildValue(value, keys[i]);
72     }
73     node.Value = value;
74     return node;
75 }
76
77 public Node SetChildValue(object value, object key)
78 {
79     if (!ChildNodes.TryGetValue(key, out Node child))
80     {
81         child = AddChild(key, value);
82     }
83     child.Value = value;
84     return child;
85 }
86 }
87 }

```

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

```

1 using System;
2 using System.Collections;
3 using Xunit;
4 using Platform.Random;
5
6 namespace Platform.Collections.Tests
7 {
8     public static class BitStringTests
9     {
10         [Fact]
11         public static void BitGetSetTest()
12         {
13             const int n = 250;
14             var bitArray = new BitArray(n);

```

```

15     var bitString = new BitString(n);
16     for (var i = 0; i < n; i++)
17     {
18         var value = RandomHelpers.Default.NextBoolean();
19         bitArray.Set(i, value);
20         bitString.Set(i, value);
21         Assert.Equal(value, bitArray.Get(i));
22         Assert.Equal(value, bitString.Get(i));
23     }
24 }
25
26 [Fact]
27 public static void BitVectorNotTest()
28 {
29     TestToOperationsWithSameMeaning((x, y, w, v) =>
30     {
31         x.VectorNot();
32         w.Not();
33     });
34 }
35
36 [Fact]
37 public static void BitParallelNotTest()
38 {
39     TestToOperationsWithSameMeaning((x, y, w, v) =>
40     {
41         x.ParallelNot();
42         w.Not();
43     });
44 }
45
46 [Fact]
47 public static void BitParallelVectorNotTest()
48 {
49     TestToOperationsWithSameMeaning((x, y, w, v) =>
50     {
51         x.ParallelVectorNot();
52         w.Not();
53     });
54 }
55
56 [Fact]
57 public static void BitVectorAndTest()
58 {
59     TestToOperationsWithSameMeaning((x, y, w, v) =>
60     {
61         x.VectorAnd(y);
62         w.And(v);
63     });
64 }
65
66 [Fact]
67 public static void BitParallelAndTest()
68 {
69     TestToOperationsWithSameMeaning((x, y, w, v) =>
70     {
71         x.ParallelAnd(y);
72         w.And(v);
73     });
74 }
75
76 [Fact]
77 public static void BitParallelVectorAndTest()
78 {
79     TestToOperationsWithSameMeaning((x, y, w, v) =>
80     {
81         x.ParallelVectorAnd(y);
82         w.And(v);
83     });
84 }
85
86 [Fact]
87 public static void BitVectorOrTest()
88 {
89     TestToOperationsWithSameMeaning((x, y, w, v) =>
90     {
91         x.VectorOr(y);
92         w.Or(v);

```

```

93     });
94 }
95
96 [Fact]
97 public static void BitParallelOrTest()
98 {
99     TestToOperationsWithSameMeaning((x, y, w, v) =>
100     {
101         x.ParallelOr(y);
102         w.Or(v);
103     });
104 }
105
106 [Fact]
107 public static void BitParallelVectorOrTest()
108 {
109     TestToOperationsWithSameMeaning((x, y, w, v) =>
110     {
111         x.ParallelVectorOr(y);
112         w.Or(v);
113     });
114 }
115
116 [Fact]
117 public static void BitVectorXorTest()
118 {
119     TestToOperationsWithSameMeaning((x, y, w, v) =>
120     {
121         x.VectorXor(y);
122         w.Xor(v);
123     });
124 }
125
126 [Fact]
127 public static void BitParallelXorTest()
128 {
129     TestToOperationsWithSameMeaning((x, y, w, v) =>
130     {
131         x.ParallelXor(y);
132         w.Xor(v);
133     });
134 }
135
136 [Fact]
137 public static void BitParallelVectorXorTest()
138 {
139     TestToOperationsWithSameMeaning((x, y, w, v) =>
140     {
141         x.ParallelVectorXor(y);
142         w.Xor(v);
143     });
144 }
145
146 private static void TestToOperationsWithSameMeaning(Action<BitString, BitString,
147 ↪ BitString, BitString> test)
148 {
149     const int n = 5654;
150     var x = new BitString(n);
151     var y = new BitString(n);
152     while (x.Equals(y))
153     {
154         x.SetRandomBits();
155         y.SetRandomBits();
156     }
157     var w = new BitString(x);
158     var v = new BitString(y);
159     Assert.False(x.Equals(y));
160     Assert.False(w.Equals(v));
161     Assert.True(x.Equals(w));
162     Assert.True(y.Equals(v));
163     test(x, y, w, v);
164     Assert.True(x.Equals(w));
165 }
166 }

```

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

```

1  using Xunit;
2  using Platform.Collections.Segments;
3
4  namespace Platform.Collections.Tests
5  {
6      public static class CharsSegmentTests
7      {
8          [Fact]
9          public static void GetHashCodeEqualsTest()
10         {
11             const string testString = "test test";
12             var testArray = testString.ToCharArray();
13             var first = new CharSegment(testArray, 0, 4);
14             var firstHashCode = first.GetHashCode();
15             var second = new CharSegment(testArray, 5, 4);
16             var secondHashCode = second.GetHashCode();
17             Assert.Equal(firstHashCode, secondHashCode);
18         }
19
20         [Fact]
21         public static void EqualsTest()
22         {
23             const string testString = "test test";
24             var testArray = testString.ToCharArray();
25             var first = new CharSegment(testArray, 0, 4);
26             var second = new CharSegment(testArray, 5, 4);
27             Assert.True(first.Equals(second));
28         }
29     }
30 }

```

1.42 ./Platform.Collections.Tests/StringTests.cs

```

1  using Xunit;
2
3  namespace Platform.Collections.Tests
4  {
5      public static class StringTests
6      {
7          [Fact]
8          public static void CapitalizeFirstLetterTest()
9          {
10             var source1 = "hello";
11             var result1 = source1.CapitalizeFirstLetter();
12             Assert.Equal("Hello", result1);
13             var source2 = "Hello";
14             var result2 = source2.CapitalizeFirstLetter();
15             Assert.Equal("Hello", result2);
16             var source3 = " hello";
17             var result3 = source3.CapitalizeFirstLetter();
18             Assert.Equal(" Hello", result3);
19         }
20
21         [Fact]
22         public static void TrimSingleTest()
23         {
24             var source1 = "";
25             var result1 = source1.TrimSingle('\');
26             Assert.Equal("", result1);
27             var source2 = " ";
28             var result2 = source2.TrimSingle('\');
29             Assert.Equal("", result2);
30             var source3 = "'hello'";
31             var result3 = source3.TrimSingle('\');
32             Assert.Equal("hello", result3);
33             var source4 = "hello'";
34             var result4 = source4.TrimSingle('\');
35             Assert.Equal("hello", result4);
36             var source5 = "'hello";
37             var result5 = source5.TrimSingle('\');
38             Assert.Equal("hello", result5);
39         }
40     }
41 }

```

Index

- ./Platform.Collections.Tests/BitStringTests.cs, 33
- ./Platform.Collections.Tests/CharsSegmentTests.cs, 35
- ./Platform.Collections.Tests/StringTests.cs, 36
- ./Platform.Collections/Arrays/ArrayExtensions.cs, 1
- ./Platform.Collections/Arrays/ArrayFiller[TElement, TReturnConstant].cs, 1
- ./Platform.Collections/Arrays/ArrayFiller[TElement].cs, 1
- ./Platform.Collections/Arrays/ArrayPool.cs, 2
- ./Platform.Collections/Arrays/ArrayPool[T].cs, 2
- ./Platform.Collections/Arrays/ArrayString.cs, 3
- ./Platform.Collections/Arrays/CharArrayExtensions.cs, 3
- ./Platform.Collections/Arrays/GenericArrayExtensions.cs, 4
- ./Platform.Collections/BitString.cs, 4
- ./Platform.Collections/BitStringExtensions.cs, 18
- ./Platform.Collections/Concurrent/ConcurrentQueueExtensions.cs, 19
- ./Platform.Collections/Concurrent/ConcurrentStackExtensions.cs, 19
- ./Platform.Collections/EnsureExtensions.cs, 19
- ./Platform.Collections/ICollectionExtensions.cs, 20
- ./Platform.Collections/IDictionaryExtensions.cs, 21
- ./Platform.Collections/Lists/CharListExtensions.cs, 21
- ./Platform.Collections/Lists/IListComparer.cs, 22
- ./Platform.Collections/Lists/IListEqualityComparer.cs, 22
- ./Platform.Collections/Lists/IListExtensions.cs, 22
- ./Platform.Collections/Lists/ListFiller.cs, 24
- ./Platform.Collections/Segments/CharSegment.cs, 25
- ./Platform.Collections/Segments/Segment.cs, 25
- ./Platform.Collections/Segments/Walkers/AllSegmentsWalkerBase.cs, 27
- ./Platform.Collections/Segments/Walkers/AllSegmentsWalkerBase[T, TSegment].cs, 27
- ./Platform.Collections/Segments/Walkers/AllSegmentsWalkerBase[T].cs, 27
- ./Platform.Collections/Segments/Walkers/AllSegmentsWalkerExtensions.cs, 27
- ./Platform.Collections/Segments/Walkers/DictionaryBasedDuplicateSegmentsWalkerBase[T, Segment].cs, 27
- ./Platform.Collections/Segments/Walkers/DictionaryBasedDuplicateSegmentsWalkerBase[T].cs, 28
- ./Platform.Collections/Segments/Walkers/DuplicateSegmentsWalkerBase[T, TSegment].cs, 29
- ./Platform.Collections/Segments/Walkers/DuplicateSegmentsWalkerBase[T].cs, 29
- ./Platform.Collections/Sets/ISetExtensions.cs, 29
- ./Platform.Collections/Sets/SetFiller.cs, 29
- ./Platform.Collections/Stacks/DefaultStack.cs, 30
- ./Platform.Collections/Stacks/IStack.cs, 30
- ./Platform.Collections/Stacks/IStackExtensions.cs, 30
- ./Platform.Collections/Stacks/IStackFactory.cs, 31
- ./Platform.Collections/Stacks/StackExtensions.cs, 31
- ./Platform.Collections/StringExtensions.cs, 31
- ./Platform.Collections/Trees/Node.cs, 32