

LinksPlatform's Platform.Collections Class Library

1.1 ./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(ICollection<TElement> collection)
27         {
28             _array[_position++] = collection[0];
29             return _returnConstant;
30         }
31     }
32 }
```

1.2 ./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         [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(ICollection<TElement> collection)
33         {
34             _array[_position++] = collection[0];
35             return true;
36         }
37     }
38 }
```

1.3 ./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 }

```

1.4 ./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;
41             Array.Copy(sourceArray, destinationArray, size < sourceArray.Length ? (int)size :
42                 ↪ sourceArray.Length);
43             source.Dispose();
44             return destination;
45         }
46
47         public virtual void Clear() => _pool.Clear();
48
49         public virtual T[] Allocate(long size)
50         {
51             Ensure.Always.ArgumentInRange(size, (0, int.MaxValue));
52             return size == 0 ? Empty : _pool.GetOrDefault((int)size)?.PopOrDefault() ?? new
53                 ↪ T[size];
54         }
55
56         public virtual void Free(T[] array)
57         {
58             Ensure.Always.ArgumentNotNull(array, nameof(array));
59             if (array.Length == 0)
60             {
61                 return;
62             }
63             var stack = _pool.GetOrAdd(array.Length, size => new Stack<T[]>(_maxArraysPerSize));
64             if (stack.Count == _maxArraysPerSize) // Stack is full

```

```

60         {
61             return;
62         }
63         stack.Push(array);
64     }
65 }
66 }

```

1.5 ./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 }

```

1.6 ./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[]
28        ↪ right, int rightOffset)
29        {
30            fixed (char* leftPointer = &left[leftOffset])
31            {
32                fixed (char* rightPointer = &right[rightOffset])
33                {
34                    char* leftPointerCopy = leftPointer, rightPointerCopy = rightPointer;
35                    if (!CheckArraysMainPartForEquality(ref leftPointerCopy, ref
36                    ↪ rightPointerCopy, ref length))
37                    {
38                        return false;
39                    }
40                    CheckArraysRemainderForEquality(ref leftPointerCopy, ref rightPointerCopy,
41                    ↪ ref length);
42                    return length <= 0;
43                }
44            }
45        }
46
47        private static bool CheckArraysMainPartForEquality(ref char* left, ref char* right, ref
48        ↪ int length)
49        {
50            while (length >= 10)
51            {
52                if ((* (int*)left != *(int*)right)

```

```

49         || (*(int*)(left + 2)) != (*(int*)(right + 2))
50         || (*(int*)(left + 4)) != (*(int*)(right + 4))
51         || (*(int*)(left + 6)) != (*(int*)(right + 6))
52         || (*(int*)(left + 8)) != (*(int*)(right + 8)))
53     {
54         return false;
55     }
56     left += 10;
57     right += 10;
58     length -= 10;
59 }
60 return true;
61 }
62
63 private static void CheckArraysRemainderForEquality(ref char* left, ref char* right, ref
→ int length)
64 {
65     // This depends on the fact that the String objects are
66     // always zero terminated and that the terminating zero is not included
67     // in the length. For odd string sizes, the last compare will include
68     // the zero terminator.
69     while (length > 0)
70     {
71         if (*(int*)left != *(int*)right)
72         {
73             break;
74         }
75         left += 2;
76         right += 2;
77         length -= 2;
78     }
79 }
80 }
81 }

```

1.7 ./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 }

```

1.8 ./Platform.Collections/BitString.cs

```

1 using System;
2 using System.Collections.Concurrent;
3 using System.Collections.Generic;
4 using System.Linq;
5 using System.Numerics;
6 using System.Runtime.CompilerServices;
7 using System.Threading.Tasks;
8 using Platform.Exceptions;
9 using Platform.Ranges;
10
11 // ReSharper disable ForCanBeConvertedToForeach
12 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
13
14 namespace Platform.Collections
15 {
16     /// <remarks>
17     /// А что если хранить карту значений, где каждый бит будет означать присутствует ли блок в
→ 64 бит в массиве значений.
18     /// 64 бита по 0 бит, будут означать отсутствие 64-х блоков по 64 бита. Т.е. упаковка 512
→ байт в 8 байт.
19     /// Подобный принцип можно применять и к 64-ём блокам и т.п. По сути это карта значений. С
→ помощью которой можно быстро
20     /// проверять есть ли значения непосредственно далее (ниже по уровню).
21     /// Или как таблица виртуальной памяти где номер блока означает его присутствие и адрес.
22     /// </remarks>
23     public class BitString : IEquatable<BitString>

```

```

24 {
25     private static readonly byte[][] _bitsSetIn16Bits;
26     private long[] _array;
27     private long _length;
28     private long _minPositiveWord;
29     private long _maxPositiveWord;
30
31     public bool this[long index]
32     {
33         get => Get(index);
34         set => Set(index, value);
35     }
36
37     public long Length
38     {
39         get => _length;
40         set
41         {
42             if (_length == value)
43             {
44                 return;
45             }
46             Ensure.Always.ArgumentInRange(value, GetValidLengthRange(), nameof(Length));
47             // Currently we never shrink the array
48             if (value > _length)
49             {
50                 var words = GetWordsCountFromIndex(value);
51                 var oldWords = GetWordsCountFromIndex(_length);
52                 if (words > _array.LongLength)
53                 {
54                     var copy = new long[words];
55                     Array.Copy(_array, copy, _array.LongLength);
56                     _array = copy;
57                 }
58                 else
59                 {
60                     // What is going on here?
61                     Array.Clear(_array, (int)oldWords, (int)(words - oldWords));
62                 }
63                 // What is going on here?
64                 var mask = (int)(_length % 64);
65                 if (mask > 0)
66                 {
67                     _array[oldWords - 1] &= (1L << mask) - 1;
68                 }
69             }
70             else
71             {
72                 // Looks like minimum and maximum positive words are not updated
73                 throw new NotImplementedException();
74             }
75             _length = value;
76         }
77     }
78
79     #region Constructors
80
81     static BitString()
82     {
83         _bitsSetIn16Bits = new byte[65536][];
84         int i, c, k;
85         byte bitIndex;
86         for (i = 0; i < 65536; i++)
87         {
88             // Calculating size of array (number of positive bits)
89             for (c = 0, k = 1; k <= 65536; k <= 1)
90             {
91                 if ((i & k) == k)
92                 {
93                     c++;
94                 }
95             }
96             var array = new byte[c];
97             // Adding positive bits indices into array
98             for (bitIndex = 0, c = 0, k = 1; k <= 65536; k <= 1)
99             {
100                 if ((i & k) == k)
101                 {
102                     array[c++] = bitIndex;

```

```

103         }
104         bitIndex++;
105     }
106     _bitsSetIn16Bits[i] = array;
107 }
108 }
109
110 public BitString(BitString other)
111 {
112     Ensure.Always.ArgumentNotNull(other, nameof(other));
113     _length = other._length;
114     _array = new long[GetWordsCountFromIndex(_length)];
115     _minPositiveWord = other._minPositiveWord;
116     _maxPositiveWord = other._maxPositiveWord;
117     Array.Copy(other._array, _array, _array.LongLength);
118 }
119
120 public BitString(long length)
121 {
122     Ensure.Always.ArgumentInRange(length, GetValidLengthRange(), nameof(length));
123     _length = length;
124     _array = new long[GetWordsCountFromIndex(_length)];
125     MarkBordersAsAllBitsReset();
126 }
127
128 public BitString(long length, bool defaultValue)
129     : this(length)
130 {
131     if (defaultValue)
132     {
133         SetAll();
134     }
135 }
136
137 #endregion
138
139 public BitString Not()
140 {
141     for (var i = 0; i < _array.Length; i++)
142     {
143         _array[i] = ~_array[i];
144         RefreshBordersByWord(i);
145     }
146     return this;
147 }
148
149 public BitString ParallelNot()
150 {
151     var processorCount = Environment.ProcessorCount;
152     if (processorCount <= 1)
153     {
154         return Not();
155     }
156     var partitioner = Partitioner.Create(0, _array.Length, _array.Length /
157 ↪ processorCount);
158     Parallel.ForEach(partitioner.GetDynamicPartitions(), range =>
159     {
160         var maximum = range.Item2;
161         for (var i = range.Item1; i < maximum; i++)
162         {
163             _array[i] = ~_array[i];
164         }
165     });
166     MarkBordersAsAllBitsSet();
167     TryShrinkBorders();
168     return this;
169 }
170
171 public BitString VectorNot()
172 {
173     if (!Vector.IsHardwareAccelerated)
174     {
175         return Not();
176     }
177     var step = Vector<long>.Count;
178     if (_array.Length < step)
179     {
180         return Not();
181     }

```

```

181     VectorNotLoop(_array, step, 0, _array.Length);
182     MarkBordersAsAllBitsSet();
183     TryShrinkBorders();
184     return this;
185 }
186
187 public BitString ParallelVectorNot()
188 {
189     var processorCount = Environment.ProcessorCount;
190     if (processorCount <= 1 && Vector.IsHardwareAccelerated)
191     {
192         return VectorNot();
193     }
194     if (!Vector.IsHardwareAccelerated)
195     {
196         return Not();
197     }
198     var step = Vector<long>.Count;
199     if (_array.Length < (step * Environment.ProcessorCount))
200     {
201         return VectorNot();
202     }
203     var partitioner = Partitioner.Create(0, _array.Length, _array.Length /
204         ↪ processorCount);
205     Parallel.ForEach(partitioner.GetDynamicPartitions(), range => VectorNotLoop(_array,
206         ↪ step, range.Item1, range.Item2));
207     MarkBordersAsAllBitsSet();
208     TryShrinkBorders();
209     return this;
210 }
211
212 static private void VectorNotLoop(long[] array, int step, int start, int maximum)
213 {
214     var i = start;
215     var range = maximum - start - 1;
216     var stop = range - (range % step);
217     for (; i < stop; i += step)
218     {
219         var vector = new Vector<long>(array, i);
220         (~vector).CopyTo(array, i);
221     }
222     for (; i < maximum; i++)
223     {
224         array[i] = ~array[i];
225     }
226 }
227
228 public BitString And(BitString other)
229 {
230     EnsureBitStringHasTheSameSize(other, nameof(other));
231     GetCommonOuterBorders(this, other, out long from, out long to);
232     var otherArray = other._array;
233     for (var i = from; i <= to; i++)
234     {
235         _array[i] &= otherArray[i];
236         RefreshBordersByWord(i);
237     }
238     return this;
239 }
240
241 public BitString ParallelAnd(BitString other)
242 {
243     var processorCount = Environment.ProcessorCount;
244     if (processorCount <= 1)
245     {
246         return And(other);
247     }
248     EnsureBitStringHasTheSameSize(other, nameof(other));
249     GetCommonOuterBorders(this, other, out long from, out long to);
250     var partitioner = Partitioner.Create(from, to + 1, (to - from) / processorCount);
251     Parallel.ForEach(partitioner.GetDynamicPartitions(), range =>
252     {
253         var maximum = range.Item2;
254         for (var i = range.Item1; i < maximum; i++)
255         {
256             _array[i] &= other._array[i];
257         }
258     });
259     MarkBordersAsAllBitsSet();

```

```

258     TryShrinkBorders();
259     return this;
260 }
261
262 public BitString VectorAnd(BitString other)
263 {
264     if (!Vector.IsHardwareAccelerated)
265     {
266         return And(other);
267     }
268     var step = Vector<long>.Count;
269     if (_array.Length < step)
270     {
271         return And(other);
272     }
273     EnsureBitStringHasTheSameSize(other, nameof(other));
274     GetCommonOuterBorders(this, other, out long from, out long to);
275     VectorAndLoop(_array, other._array, step, (int)from, (int)(to + 1));
276     MarkBordersAsAllBitsSet();
277     TryShrinkBorders();
278     return this;
279 }
280
281 public BitString ParallelVectorAnd(BitString other)
282 {
283     var processorCount = Environment.ProcessorCount;
284     if (processorCount <= 1 && Vector.IsHardwareAccelerated)
285     {
286         return VectorAnd(other);
287     }
288     if (!Vector.IsHardwareAccelerated)
289     {
290         return And(other);
291     }
292     var step = Vector<long>.Count;
293     if (_array.Length < (step * Environment.ProcessorCount))
294     {
295         return VectorAnd(other);
296     }
297     EnsureBitStringHasTheSameSize(other, nameof(other));
298     GetCommonOuterBorders(this, other, out long from, out long to);
299     var partitioner = Partitioner.Create(from, to + 1, (to - from) / processorCount);
300     Parallel.ForEach(partitioner.GetDynamicPartitions(), range => VectorAndLoop(_array,
301     ↪ other._array, step, (int)range.Item1, (int)range.Item2));
302     MarkBordersAsAllBitsSet();
303     TryShrinkBorders();
304     return this;
305 }
306
307 static private void VectorAndLoop(long[] array, long[] otherArray, int step, int start,
308 ↪ int maximum)
309 {
310     var i = start;
311     var range = maximum - start - 1;
312     var stop = range - (range % step);
313     for (; i < stop; i += step)
314     {
315         var thisVector = new Vector<long>(array, i);
316         var otherVector = new Vector<long>(otherArray, i);
317         (thisVector & otherVector).CopyTo(array, i);
318     }
319     for (; i < maximum; i++)
320     {
321         array[i] &= otherArray[i];
322     }
323 }
324
325 public BitString Or(BitString other)
326 {
327     EnsureBitStringHasTheSameSize(other, nameof(other));
328     GetCommonOuterBorders(this, other, out long from, out long to);
329     for (var i = from; i <= to; i++)
330     {
331         _array[i] |= other._array[i];
332         RefreshBordersByWord(i);
333     }
334     return this;
335 }

```



```

335 public BitString ParallelOr(BitString other)
336 {
337     var processorCount = Environment.ProcessorCount;
338     if (processorCount <= 1)
339     {
340         return Or(other);
341     }
342     EnsureBitStringHasTheSameSize(other, nameof(other));
343     GetCommonOuterBorders(this, other, out long from, out long to);
344     var partitioner = Partitioner.Create(from, to + 1, (to - from) / processorCount);
345     Parallel.ForEach(partitioner.GetDynamicPartitions(), range =>
346     {
347         var maximum = range.Item2;
348         for (var i = range.Item1; i < maximum; i++)
349         {
350             _array[i] |= other._array[i];
351         }
352     });
353     MarkBordersAsAllBitsSet();
354     TryShrinkBorders();
355     return this;
356 }
357
358 public BitString VectorOr(BitString other)
359 {
360     if (!Vector.IsHardwareAccelerated)
361     {
362         return Or(other);
363     }
364     var step = Vector<long>.Count;
365     if (_array.Length < step)
366     {
367         return Or(other);
368     }
369     EnsureBitStringHasTheSameSize(other, nameof(other));
370     GetCommonOuterBorders(this, other, out long from, out long to);
371     VectorOrLoop(_array, other._array, step, (int)from, (int)(to + 1));
372     MarkBordersAsAllBitsSet();
373     TryShrinkBorders();
374     return this;
375 }
376
377 public BitString ParallelVectorOr(BitString other)
378 {
379     var processorCount = Environment.ProcessorCount;
380     if (processorCount <= 1 && Vector.IsHardwareAccelerated)
381     {
382         return VectorOr(other);
383     }
384     if (!Vector.IsHardwareAccelerated)
385     {
386         return Or(other);
387     }
388     var step = Vector<long>.Count;
389     if (_array.Length < (step * Environment.ProcessorCount))
390     {
391         return VectorOr(other);
392     }
393     EnsureBitStringHasTheSameSize(other, nameof(other));
394     GetCommonOuterBorders(this, other, out long from, out long to);
395     var partitioner = Partitioner.Create(from, to + 1, (to - from) / processorCount);
396     Parallel.ForEach(partitioner.GetDynamicPartitions(), range => VectorOrLoop(_array,
397     ↪ other._array, step, (int)range.Item1, (int)range.Item2));
398     MarkBordersAsAllBitsSet();
399     TryShrinkBorders();
400     return this;
401 }
402
403 static private void VectorOrLoop(long[] array, long[] otherArray, int step, int start,
404 ↪ int maximum)
405 {
406     var i = start;
407     var range = maximum - start - 1;
408     var stop = range - (range % step);
409     for (; i < stop; i += step)
410     {
411         var thisVector = new Vector<long>(array, i);
412         var otherVector = new Vector<long>(otherArray, i);

```

```

411         (thisVector | otherVector).CopyTo(array, i);
412     }
413     for (; i < maximum; i++)
414     {
415         array[i] |= otherArray[i];
416     }
417 }
418
419 public BitString Xor(BitString other)
420 {
421     EnsureBitStringHasTheSameSize(other, nameof(other));
422     GetCommonOuterBorders(this, other, out long from, out long to);
423     for (var i = from; i <= to; i++)
424     {
425         _array[i] ^= other._array[i];
426         RefreshBordersByWord(i);
427     }
428     return this;
429 }
430
431 public BitString ParallelXor(BitString other)
432 {
433     var processorCount = Environment.ProcessorCount;
434     if (processorCount <= 1)
435     {
436         return Xor(other);
437     }
438     EnsureBitStringHasTheSameSize(other, nameof(other));
439     GetCommonOuterBorders(this, other, out long from, out long to);
440     var partitioner = Partitioner.Create(from, to + 1, (to - from) / processorCount);
441     Parallel.ForEach(partitioner.GetDynamicPartitions(), range =>
442     {
443         var maximum = range.Item2;
444         for (var i = range.Item1; i < maximum; i++)
445         {
446             _array[i] ^= other._array[i];
447         }
448     });
449     MarkBordersAsAllBitsSet();
450     TryShrinkBorders();
451     return this;
452 }
453
454 public BitString VectorXor(BitString other)
455 {
456     if (!Vector.IsHardwareAccelerated)
457     {
458         return Xor(other);
459     }
460     var step = Vector<long>.Count;
461     if (_array.Length < step)
462     {
463         return Xor(other);
464     }
465     EnsureBitStringHasTheSameSize(other, nameof(other));
466     GetCommonOuterBorders(this, other, out long from, out long to);
467     VectorXorLoop(_array, other._array, step, (int)from, (int)(to + 1));
468     MarkBordersAsAllBitsSet();
469     TryShrinkBorders();
470     return this;
471 }
472
473 public BitString ParallelVectorXor(BitString other)
474 {
475     var processorCount = Environment.ProcessorCount;
476     if (processorCount <= 1 && Vector.IsHardwareAccelerated)
477     {
478         return VectorXor(other);
479     }
480     if (!Vector.IsHardwareAccelerated)
481     {
482         return Xor(other);
483     }
484     var step = Vector<long>.Count;
485     if (_array.Length < (step * Environment.ProcessorCount))
486     {
487         return VectorXor(other);
488     }
489     EnsureBitStringHasTheSameSize(other, nameof(other));

```

```

490     GetCommonOuterBorders(this, other, out long from, out long to);
491     var partitioner = Partitioner.Create(from, to + 1, (to - from) / processorCount);
492     var x = partitioner.GetDynamicPartitions();
493     var array = x.ToArray();
494     Parallel.ForEach(array, range => VectorXorLoop(_array, other._array, step,
495         ↪ (int)range.Item1, (int)range.Item2));
496     MarkBordersAsAllBitsSet();
497     TryShrinkBorders();
498     return this;
499 }
500
501 static private void VectorXorLoop(long[] array, long[] otherArray, int step, int start,
502     ↪ int maximum)
503 {
504     var i = start;
505     var range = maximum - start - 1;
506     var stop = range - (range % step);
507     for (; i < stop; i += step)
508     {
509         var thisVector = new Vector<long>(array, i);
510         var otherVector = new Vector<long>(otherArray, i);
511         (thisVector ^ otherVector).CopyTo(array, i);
512     }
513     for (; i < maximum; i++)
514     {
515         array[i] ^= otherArray[i];
516     }
517 }
518
519 private void RefreshBordersByWord(long wordIndex)
520 {
521     if (_array[wordIndex] == 0)
522     {
523         if (wordIndex == _minPositiveWord && wordIndex != _array.LongLength - 1)
524         {
525             _minPositiveWord++;
526         }
527         if (wordIndex == _maxPositiveWord && wordIndex != 0)
528         {
529             _maxPositiveWord--;
530         }
531     }
532     else
533     {
534         if (wordIndex < _minPositiveWord)
535         {
536             _minPositiveWord = wordIndex;
537         }
538         if (wordIndex > _maxPositiveWord)
539         {
540             _maxPositiveWord = wordIndex;
541         }
542     }
543 }
544
545 public bool TryShrinkBorders()
546 {
547     GetBorders(out long from, out long to);
548     while (from <= to && _array[from] == 0)
549     {
550         from++;
551     }
552     if (from > to)
553     {
554         MarkBordersAsAllBitsReset();
555         return true;
556     }
557     while (to >= from && _array[to] == 0)
558     {
559         to--;
560     }
561     if (to < from)
562     {
563         MarkBordersAsAllBitsReset();
564         return true;
565     }
566     var bordersUpdated = from != _minPositiveWord || to != _maxPositiveWord;
567     if (bordersUpdated)
568     {

```

```

567         SetBorders(from, to);
568     }
569     return bordersUpdated;
570 }
571
572 [MethodImpl(MethodImplOptions.AggressiveInlining)]
573 public bool Get(long index)
574 {
575     Ensure.Always.ArgumentInRange(index, GetValidIndexRange(), nameof(index));
576     return (_array[GetWordIndexFromIndex(index)] & GetBitMaskFromIndex(index)) != 0;
577 }
578
579 [MethodImpl(MethodImplOptions.AggressiveInlining)]
580 public void Set(long index, bool value)
581 {
582     if (value)
583     {
584         Set(index);
585     }
586     else
587     {
588         Reset(index);
589     }
590 }
591
592 [MethodImpl(MethodImplOptions.AggressiveInlining)]
593 public void Set(long index)
594 {
595     Ensure.Always.ArgumentInRange(index, GetValidIndexRange(), nameof(index));
596     var wordIndex = GetWordIndexFromIndex(index);
597     var mask = GetBitMaskFromIndex(index);
598     _array[wordIndex] |= mask;
599     RefreshBordersByWord(wordIndex);
600 }
601
602 [MethodImpl(MethodImplOptions.AggressiveInlining)]
603 public void Reset(long index)
604 {
605     Ensure.Always.ArgumentInRange(index, GetValidIndexRange(), nameof(index));
606     var wordIndex = GetWordIndexFromIndex(index);
607     var mask = GetBitMaskFromIndex(index);
608     _array[wordIndex] &= ~mask;
609     RefreshBordersByWord(wordIndex);
610 }
611
612 public bool Add(long index)
613 {
614     var wordIndex = GetWordIndexFromIndex(index);
615     var mask = GetBitMaskFromIndex(index);
616     if ((_array[wordIndex] & mask) == 0)
617     {
618         _array[wordIndex] |= mask;
619         RefreshBordersByWord(wordIndex);
620         return true;
621     }
622     else
623     {
624         return false;
625     }
626 }
627
628 public void SetAll(bool value)
629 {
630     if (value)
631     {
632         SetAll();
633     }
634     else
635     {
636         ResetAll();
637     }
638 }
639
640 public void SetAll()
641 {
642     const long fillValue = unchecked((long)0xffffffffffffffff);
643     var words = GetWordsCountFromIndex(_length);
644     for (var i = 0; i < words; i++)
645     {

```

```

646         _array[i] = fillValue;
647     }
648     MarkBordersAsAllBitsSet();
649 }
650
651 public void ResetAll()
652 {
653     const long fillValue = 0;
654     GetBorders(out long from, out long to);
655     for (var i = from; i <= to; i++)
656     {
657         _array[i] = fillValue;
658     }
659     MarkBordersAsAllBitsReset();
660 }
661
662 public List<long> GetSetIndices()
663 {
664     var result = new List<long>();
665     GetBorders(out long from, out long to);
666     for (var i = from; i <= to; i++)
667     {
668         var word = _array[i];
669         if (word != 0)
670         {
671             AppendAllSetBitIndices(result, i, word);
672         }
673     }
674     return result;
675 }
676
677 public List<ulong> GetSetUInt64Indices()
678 {
679     var result = new List<ulong>();
680     GetBorders(out ulong from, out ulong to);
681     for (var i = from; i <= to; i++)
682     {
683         var word = _array[i];
684         if (word != 0)
685         {
686             AppendAllSetBitIndices(result, i, word);
687         }
688     }
689     return result;
690 }
691
692 public long GetFirstSetBitIndex()
693 {
694     var i = _minPositiveWord;
695     var word = _array[i];
696     if (word != 0)
697     {
698         return GetFirstSetBitForWord(i, word);
699     }
700     return -1;
701 }
702
703 public long GetLastSetBitIndex()
704 {
705     var i = _maxPositiveWord;
706     var word = _array[i];
707     if (word != 0)
708     {
709         return GetLastSetBitForWord(i, word);
710     }
711     return -1;
712 }
713
714 public long CountSetBits()
715 {
716     var total = 0L;
717     GetBorders(out long from, out long to);
718     for (var i = from; i <= to; i++)
719     {
720         var word = _array[i];
721         if (word != 0)
722         {
723             total += CountSetBitsForWord(word);
724         }
725     }
726 }

```

```

725     }
726     return total;
727 }
728
729 public bool HaveCommonBits(BitString other)
730 {
731     EnsureBitStringHasTheSameSize(other, nameof(other));
732     GetCommonInnerBorders(this, other, out long from, out long to);
733     var otherArray = other._array;
734     for (var i = from; i <= to; i++)
735     {
736         var left = _array[i];
737         var right = otherArray[i];
738         if (left != 0 && right != 0 && (left & right) != 0)
739         {
740             return true;
741         }
742     }
743     return false;
744 }
745
746 public long CountCommonBits(BitString other)
747 {
748     EnsureBitStringHasTheSameSize(other, nameof(other));
749     GetCommonInnerBorders(this, other, out long from, out long to);
750     var total = 0L;
751     var otherArray = other._array;
752     for (var i = from; i <= to; i++)
753     {
754         var left = _array[i];
755         var right = otherArray[i];
756         var combined = left & right;
757         if (combined != 0)
758         {
759             total += CountSetBitsForWord(combined);
760         }
761     }
762     return total;
763 }
764
765 public List<long> GetCommonIndices(BitString other)
766 {
767     EnsureBitStringHasTheSameSize(other, nameof(other));
768     GetCommonInnerBorders(this, other, out long from, out long to);
769     var result = new List<long>();
770     var otherArray = other._array;
771     for (var i = from; i <= to; i++)
772     {
773         var left = _array[i];
774         var right = otherArray[i];
775         var combined = left & right;
776         if (combined != 0)
777         {
778             AppendAllSetBitIndices(result, i, combined);
779         }
780     }
781     return result;
782 }
783
784 public List<ulong> GetCommonUInt64Indices(BitString other)
785 {
786     EnsureBitStringHasTheSameSize(other, nameof(other));
787     GetCommonBorders(this, other, out ulong from, out ulong to);
788     var result = new List<ulong>();
789     var otherArray = other._array;
790     for (var i = from; i <= to; i++)
791     {
792         var left = _array[i];
793         var right = otherArray[i];
794         var combined = left & right;
795         if (combined != 0)
796         {
797             AppendAllSetBitIndices(result, i, combined);
798         }
799     }
800     return result;
801 }
802
803 public long GetFirstCommonBitIndex(BitString other)

```

```

804 {
805     EnsureBitStringHasTheSameSize(other, nameof(other));
806     GetCommonInnerBorders(this, other, out long from, out long to);
807     var otherArray = other._array;
808     for (var i = from; i <= to; i++)
809     {
810         var left = _array[i];
811         var right = otherArray[i];
812         var combined = left & right;
813         if (combined != 0)
814         {
815             return GetFirstSetBitForWord(i, combined);
816         }
817     }
818     return -1;
819 }
820
821 public long GetLastCommonBitIndex(BitString other)
822 {
823     EnsureBitStringHasTheSameSize(other, nameof(other));
824     GetCommonInnerBorders(this, other, out long from, out long to);
825     var otherArray = other._array;
826     for (var i = to; i >= from; i--)
827     {
828         var left = _array[i];
829         var right = otherArray[i];
830         var combined = left & right;
831         if (combined != 0)
832         {
833             return GetLastSetBitForWord(i, combined);
834         }
835     }
836     return -1;
837 }
838
839 public override bool Equals(object obj) => obj is BitString @string ? Equals(@string) :
840     ↪ false;
841
842 public bool Equals(BitString other)
843 {
844     if (_length != other._length)
845     {
846         return false;
847     }
848     var otherArray = other._array;
849     if (_array.Length != otherArray.Length)
850     {
851         return false;
852     }
853     if (_minPositiveWord != other._minPositiveWord)
854     {
855         return false;
856     }
857     if (_maxPositiveWord != other._maxPositiveWord)
858     {
859         return false;
860     }
861     GetCommonBorders(this, other, out ulong from, out ulong to);
862     for (var i = from; i <= to; i++)
863     {
864         if (_array[i] != otherArray[i])
865         {
866             return false;
867         }
868     }
869     return true;
870 }
871
872 [MethodImpl(MethodImplOptions.AggressiveInlining)]
873 private void EnsureBitStringHasTheSameSize(BitString other, string argumentName)
874 {
875     Ensure.Always.ArgumentNotNull(other, argumentName);
876     if (_length != other._length)
877     {
878         throw new ArgumentException("Bit string must be the same size.", argumentName);
879     }
880 }
881
882 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

882 private void MarkBordersAsAllBitsReset() => SetBorders(_array.LongLength - 1, 0);
883
884 [MethodImpl(MethodImplOptions.AggressiveInlining)]
885 private void MarkBordersAsAllBitsSet() => SetBorders(0, _array.LongLength - 1);
886
887 [MethodImpl(MethodImplOptions.AggressiveInlining)]
888 private void GetBorders(out long from, out long to)
889 {
890     from = _minPositiveWord;
891     to = _maxPositiveWord;
892 }
893
894 [MethodImpl(MethodImplOptions.AggressiveInlining)]
895 private void GetBorders(out ulong from, out ulong to)
896 {
897     from = (ulong)_minPositiveWord;
898     to = (ulong)_maxPositiveWord;
899 }
900
901 [MethodImpl(MethodImplOptions.AggressiveInlining)]
902 private void SetBorders(long from, long to)
903 {
904     _minPositiveWord = from;
905     _maxPositiveWord = to;
906 }
907
908 [MethodImpl(MethodImplOptions.AggressiveInlining)]
909 private Range<long> GetValidIndexRange() => (0, _length - 1);
910
911 [MethodImpl(MethodImplOptions.AggressiveInlining)]
912 private static Range<long> GetValidLengthRange() => (0, long.MaxValue);
913
914 [MethodImpl(MethodImplOptions.AggressiveInlining)]
915 private static void AppendAllSetBitIndices(List<ulong> result, ulong wordIndex, long
    ↪ wordValue)
916 {
917     GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
    ↪ bits32to47, out byte[] bits48to63);
918     AppendAllSetIndices(result, wordIndex, bits00to15, bits16to31, bits32to47,
    ↪ bits48to63);
919 }
920
921 [MethodImpl(MethodImplOptions.AggressiveInlining)]
922 private static void AppendAllSetBitIndices(List<long> result, long wordIndex, long
    ↪ wordValue)
923 {
924     GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
    ↪ bits32to47, out byte[] bits48to63);
925     AppendAllSetBitIndices(result, wordIndex, bits00to15, bits16to31, bits32to47,
    ↪ bits48to63);
926 }
927
928 [MethodImpl(MethodImplOptions.AggressiveInlining)]
929 private static long CountSetBitsForWord(long word)
930 {
931     GetBits(word, out byte[] bits00to15, out byte[] bits16to31, out byte[] bits32to47,
    ↪ out byte[] bits48to63);
932     return bits00to15.LongLength + bits16to31.LongLength + bits32to47.LongLength +
    ↪ bits48to63.LongLength;
933 }
934
935 [MethodImpl(MethodImplOptions.AggressiveInlining)]
936 private static long GetFirstSetBitForWord(long wordIndex, long wordValue)
937 {
938     GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
    ↪ bits32to47, out byte[] bits48to63);
939     return GetFirstSetBit(wordIndex, bits00to15, bits16to31, bits32to47, bits48to63);
940 }
941
942 [MethodImpl(MethodImplOptions.AggressiveInlining)]
943 private static long GetLastSetBitForWord(long wordIndex, long wordValue)
944 {
945     GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
    ↪ bits32to47, out byte[] bits48to63);
946     return GetLastSetBit(wordIndex, bits00to15, bits16to31, bits32to47, bits48to63);
947 }
948

```



```

private static void AppendAllSetBitIndices(List<long> result, long i, byte[] bits00to15,
→ byte[] bits16to31, byte[] bits32to47, byte[] bits48to63)
{
    for (var j = 0; j < bits00to15.Length; j++)
    {
        result.Add(bits00to15[j] + (i * 64));
    }
    for (var j = 0; j < bits16to31.Length; j++)
    {
        result.Add(bits16to31[j] + 16 + (i * 64));
    }
    for (var j = 0; j < bits32to47.Length; j++)
    {
        result.Add(bits32to47[j] + 32 + (i * 64));
    }
    for (var j = 0; j < bits48to63.Length; j++)
    {
        result.Add(bits48to63[j] + 48 + (i * 64));
    }
}

private static void AppendAllSetIndices(List<ulong> result, ulong i, byte[] bits00to15,
→ byte[] bits16to31, byte[] bits32to47, byte[] bits48to63)
{
    for (var j = 0; j < bits00to15.Length; j++)
    {
        result.Add(bits00to15[j] + (i * 64));
    }
    for (var j = 0; j < bits16to31.Length; j++)
    {
        result.Add(bits16to31[j] + 16UL + (i * 64));
    }
    for (var j = 0; j < bits32to47.Length; j++)
    {
        result.Add(bits32to47[j] + 32UL + (i * 64));
    }
    for (var j = 0; j < bits48to63.Length; j++)
    {
        result.Add(bits48to63[j] + 48UL + (i * 64));
    }
}

private static long GetFirstSetBit(long i, byte[] bits00to15, byte[] bits16to31, byte[]
→ bits32to47, byte[] bits48to63)
{
    if (bits00to15.Length > 0)
    {
        return bits00to15[0] + (i * 64);
    }
    if (bits16to31.Length > 0)
    {
        return bits16to31[0] + 16 + (i * 64);
    }
    if (bits32to47.Length > 0)
    {
        return bits32to47[0] + 32 + (i * 64);
    }
    return bits48to63[0] + 48 + (i * 64);
}

private static long GetLastSetBit(long i, byte[] bits00to15, byte[] bits16to31, byte[]
→ bits32to47, byte[] bits48to63)
{
    if (bits48to63.Length > 0)
    {
        return bits48to63[bits48to63.Length - 1] + 48 + (i * 64);
    }
    if (bits32to47.Length > 0)
    {
        return bits32to47[bits32to47.Length - 1] + 32 + (i * 64);
    }
    if (bits16to31.Length > 0)
    {
        return bits16to31[bits16to31.Length - 1] + 16 + (i * 64);
    }
    return bits00to15[bits00to15.Length - 1] + (i * 64);
}

```

```

1023 private static void GetBits(long word, out byte[] bits00to15, out byte[] bits16to31, out
1024     ↳ byte[] bits32to47, out byte[] bits48to63)
1025 {
1026     bits00to15 = _bitsSetIn16Bits[word & 0xffffu];
1027     bits16to31 = _bitsSetIn16Bits[(word >> 16) & 0xffffu];
1028     bits32to47 = _bitsSetIn16Bits[(word >> 32) & 0xffffu];
1029     bits48to63 = _bitsSetIn16Bits[(word >> 48) & 0xffffu];
1030 }
1031 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1032 public static void GetCommonInnerBorders(BitString left, BitString right, out long from,
1033     ↳ out long to)
1034 {
1035     from = Math.Max(left._minPositiveWord, right._minPositiveWord);
1036     to = Math.Min(left._maxPositiveWord, right._maxPositiveWord);
1037 }
1038 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1039 public static void GetCommonOuterBorders(BitString left, BitString right, out long from,
1040     ↳ out long to)
1041 {
1042     from = Math.Min(left._minPositiveWord, right._minPositiveWord);
1043     to = Math.Max(left._maxPositiveWord, right._maxPositiveWord);
1044 }
1045 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1046 public static void GetCommonBorders(BitString left, BitString right, out ulong from, out
1047     ↳ ulong to)
1048 {
1049     from = (ulong)Math.Max(left._minPositiveWord, right._minPositiveWord);
1050     to = (ulong)Math.Min(left._maxPositiveWord, right._maxPositiveWord);
1051 }
1052 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1053 public static long GetWordsCountFromIndex(long index) => (index + 63) / 64;
1054 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1055 public static long GetWordIndexFromIndex(long index) => index >> 6;
1056 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1057 public static long GetBitMaskFromIndex(long index) => 1L << (int)(index & 63);
1058 public override int GetHashCode() => base.GetHashCode();
1059 public override string ToString() => base.ToString();
1060 }
1061 }
1062 }
1063 }
1064 }
1065 }

```

1.9 ./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 }

```

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

```

1 using System.Collections.Concurrent;
2 using System.Collections.Generic;
3 using System.Runtime.CompilerServices;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Collections.Concurrent
8 {
9     public static class ConcurrentQueueExtensions
10     {

```

```

11     [MethodImpl(MethodImplOptions.AggressiveInlining)]
12     public static IEnumerable<T> DequeueAll<T>(this ConcurrentQueue<T> queue)
13     {
14         while (queue.TryDequeue(out T item))
15         {
16             yield return item;
17         }
18     }
19 }
20 }

```

1.11 ./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
            ↪ value) ? value : default;
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public static T PeekOrDefault<T>(this ConcurrentStack<T> stack) => stack.TryPeek(out T
            ↪ value) ? value : default;
15     }
16 }

```

1.12 ./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,
            ↪ ICollection<T> argument, string argumentName, string message)
19         {
20             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,
43         ↪ string argument, string argumentName) => ArgumentNotEmptyAndNotWhiteSpace(root,
44         ↪ argument, argumentName, null);
45
46     [MethodImpl(MethodImplOptions.AggressiveInlining)]
47     public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureAlwaysExtensionRoot root,
48         ↪ string argument) => ArgumentNotEmptyAndNotWhiteSpace(root, argument, null, null);
49
50     #endregion
51
52     #region OnDebug
53
54     [Conditional("DEBUG")]
55     public static void ArgumentNotEmpty<T>(this EnsureOnDebugExtensionRoot root,
56         ↪ ICollection<T> argument, string argumentName, string message) =>
57         ↪ Ensure.Always.ArgumentNotEmpty(argument, argumentName, message);
58
59     [Conditional("DEBUG")]
60     public static void ArgumentNotEmpty<T>(this EnsureOnDebugExtensionRoot root,
61         ↪ ICollection<T> argument, string argumentName) =>
62         ↪ Ensure.Always.ArgumentNotEmpty(argument, argumentName, null);
63
64     [Conditional("DEBUG")]
65     public static void ArgumentNotEmpty<T>(this EnsureOnDebugExtensionRoot root,
66         ↪ ICollection<T> argument) => Ensure.Always.ArgumentNotEmpty(argument, null, null);
67
68     [Conditional("DEBUG")]
69     public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureOnDebugExtensionRoot
70         ↪ root, string argument, string argumentName, string message) =>
71         ↪ Ensure.Always.ArgumentNotEmptyAndNotWhiteSpace(argument, argumentName, message);
72
73     [Conditional("DEBUG")]
74     public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureOnDebugExtensionRoot
75         ↪ root, string argument, string argumentName) =>
76         ↪ Ensure.Always.ArgumentNotEmptyAndNotWhiteSpace(argument, argumentName, null);
77
78     [Conditional("DEBUG")]
79     public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureOnDebugExtensionRoot
80         ↪ root, string argument) => Ensure.Always.ArgumentNotEmptyAndNotWhiteSpace(argument,
81         ↪ null, null);
82
83     #endregion
84 }
85 }

```

1.13 ./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 ==
11             ↪ null || collection.Count == 0;
12
13         public static bool AllEqualToDefault<T>(this ICollection<T> collection)
14         {
15             var equalityComparer = EqualityComparer<T>.Default;
16             return collection.All(item => equalityComparer.Equals(item, default));
17         }
18     }
19 }

```

1.14 ./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 }

```

1.15 ./Platform.Collections/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
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 }

```

1.16 ./Platform.Collections/Lists/CharIListExtensions.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 CharIListExtensions
8     {
9         /// <remarks>
10        /// Based on https://github.com/Microsoft/referencesource/blob/3b1eaf5203992df69de44c783a3eda37d3d4cd10/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
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Collections.Lists
7 {
8     public static class IListExtensions
9     {
10        public static bool AddAndReturnTrue<T>(this IList<T> list, T element)
11        {
12            list.Add(element);
13            return true;
14        }
15
16        public static int GetCountOrZero<T>(this IList<T> list) => list?.Count ?? 0;
17
18        public static bool EqualTo<T>(this IList<T> left, IList<T> right) => EqualTo(left,
19        ↪ right, ContentEqualTo);
20
21        public static bool EqualTo<T>(this IList<T> left, IList<T> right, Func<IList<T>,
22        ↪ IList<T>, bool> contentEqualityComparer)
23        {
24            if (ReferenceEquals(left, right))
25            {
26                return true;
27            }
28            var leftCount = left.GetCountOrZero();
29            var rightCount = right.GetCountOrZero();
30            if (leftCount == 0 && rightCount == 0)
31            {
32                return true;
33            }
34            if (leftCount == 0 || rightCount == 0 || leftCount != rightCount)
35            {
36                return false;
37            }
38            return contentEqualityComparer(left, right);
39        }
40
41        public static bool ContentEqualTo<T>(this IList<T> left, IList<T> right)
42        {
43            var equalityComparer = EqualityComparer<T>.Default;
44            for (var i = left.Count - 1; i >= 0; --i)
45            {
46                if (!equalityComparer.Equals(left[i], right[i]))
47                {
48                    return false;
49                }
50            }
51            return true;
52        }
53    }
54 }
```

```

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

```

1.20 ./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()

```

```

15     {
16         // Base can be not an array, but still IList<char>
17         if (Base is char[] baseArray)
18         {
19             return baseArray.GenerateHashCode(Offset, Length);
20         }
21         else
22         {
23             return this.GenerateHashCode();
24         }
25     }
26
27     public override bool Equals(Segment<char> other)
28     {
29         bool contentEqualityComparer(IList<char> left, IList<char> right)
30         {
31             // Base can be not an array, but still IList<char>
32             if (Base is char[] baseArray && other.Base is char[] otherArray)
33             {
34                 return baseArray.ContentEqualTo(Offset, Length, otherArray, other.Offset);
35             }
36             else
37             {
38                 return left.ContentEqualTo(right);
39             }
40         }
41         return this.EqualTo(other, contentEqualityComparer);
42     }
43
44     public static implicit operator string(CharSegment segment)
45     {
46         if (!(segment.Base is char[] array))
47         {
48             array = segment.Base.ToArray();
49         }
50         return new string(array, segment.Offset, segment.Length);
51     }
52
53     public override string ToString() => this;
54 }
55 }

```

1.21 ./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

```

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

```

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

```

1.23 ./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     public virtual void WalkAll(ICollection<T> elements)
17     {
18         for (int offset = 0, maxOffset = elements.Count - _minimumStringSegmentLength;
19             ↪ offset <= maxOffset; offset++)
20         {
21             for (int length = _minimumStringSegmentLength, maxLength = elements.Count -
22                 ↪ offset; length <= maxLength; length++)
23             {
24                 Iteration(CreateSegment(elements, offset, length));
25             }
26         }
27     }
28
29     protected abstract TSegment CreateSegment(ICollection<T> elements, int offset, int length);
30
31     protected abstract void Iteration(TSegment segment);
32 }

```

1.24 ./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     }
12 }

```

1.25 ./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     }
13 }

```

1.26 ./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 }

```

```

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

```

1.27 ./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> :
        ↪ DictionaryBasedDuplicateSegmentsWalkerBase<T, Segment<T>>
8     {
9         protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<Segment<T>, long>
        ↪ dictionary, int minimumStringSegmentLength, bool resetDictionaryOnEachWalk) :
        ↪ base(dictionary, minimumStringSegmentLength, resetDictionaryOnEachWalk) { }
10        protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<Segment<T>, long>
        ↪ dictionary, int minimumStringSegmentLength) : base(dictionary,
        ↪ minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
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 }

```

1.28 ./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 }

```

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

```

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

```

1.30 ./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 }

```

1.31 ./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 }

```

1.32 ./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 }

```

1.33 ./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 }
```

1.34 ./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 }
```

1.35 ./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                 }
48             }
49             return @string;
50         }
51     }
52 }
```

```

45         else
46         {
47             return @string;
48         }
49     }
50     else
51     {
52         var left = 0;
53         var right = @string.Length - 1;
54         if (@string[left] == charToTrim)
55         {
56             left++;
57         }
58         if (@string[right] == charToTrim)
59         {
60             right--;
61         }
62         return @string.Substring(left, right - left + 1);
63     }
64 }
65 else
66 {
67     return @string;
68 }
69 }
70 }
71 }

```

1.36 ./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                 return child;
27             }
28             set => SetChildValue(value, key);
29         }
30
31         public Node(object value) => Value = value;
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 }

```

1.37 ./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 }

```

1.38 ./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 }

```

```
29     }
30 }
```

1.39 ./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, 31
- ./Platform.Collections.Tests/CharsSegmentTests.cs, 33
- ./Platform.Collections.Tests/StringTests.cs, 34
- ./Platform.Collections/Arrays/ArrayFiller[TElement, TReturnConstant].cs, 1
- ./Platform.Collections/Arrays/ArrayFiller[TElement].cs, 1
- ./Platform.Collections/Arrays/ArrayPool.cs, 1
- ./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, 18
- ./Platform.Collections/Concurrent/ConcurrentStackExtensions.cs, 19
- ./Platform.Collections/EnsureExtensions.cs, 19
- ./Platform.Collections/ICollectionExtensions.cs, 20
- ./Platform.Collections/IDictionaryExtensions.cs, 20
- ./Platform.Collections/ISetExtensions.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/Segments/CharSegment.cs, 23
- ./Platform.Collections/Segments/Segment.cs, 24
- ./Platform.Collections/Segments/Walkers/AllSegmentsWalkerBase.cs, 25
- ./Platform.Collections/Segments/Walkers/AllSegmentsWalkerBase[T, TSegment].cs, 25
- ./Platform.Collections/Segments/Walkers/AllSegmentsWalkerBase[T].cs, 26
- ./Platform.Collections/Segments/Walkers/AllSegmentsWalkerExtensions.cs, 26
- ./Platform.Collections/Segments/Walkers/DictionaryBasedDuplicateSegmentsWalkerBase[T, Segment].cs, 26
- ./Platform.Collections/Segments/Walkers/DictionaryBasedDuplicateSegmentsWalkerBase[T].cs, 27
- ./Platform.Collections/Segments/Walkers/DuplicateSegmentsWalkerBase[T, TSegment].cs, 27
- ./Platform.Collections/Segments/Walkers/DuplicateSegmentsWalkerBase[T].cs, 28
- ./Platform.Collections/Stacks/DefaultStack.cs, 28
- ./Platform.Collections/Stacks/IStack.cs, 28
- ./Platform.Collections/Stacks/IStackExtensions.cs, 28
- ./Platform.Collections/Stacks/IStackFactory.cs, 28
- ./Platform.Collections/Stacks/StackExtensions.cs, 29
- ./Platform.Collections/StringExtensions.cs, 29
- ./Platform.Collections/Trees/Node.cs, 30