

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         [MethodImpl(MethodImplOptions.AggressiveInlining)]
13         public ArrayFiller(TElement[] array, long offset, TReturnConstant returnConstant) :
14             ↪ base(array, offset) => _returnConstant = returnConstant;
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public ArrayFiller(TElement[] array, TReturnConstant returnConstant) : this(array, 0,
18             ↪ returnConstant) { }
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         public TReturnConstant AddAndReturnConstant(TElement element)
22         {
23             _array[_position++] = element;
24             return _returnConstant;
25         }
26
27         [MethodImpl(MethodImplOptions.AggressiveInlining)]
28         public TReturnConstant AddFirstAndReturnConstant(ICollection<TElement> collection)
29         {
30             _array[_position++] = collection[0];
31             return _returnConstant;
32         }
33     }
34 }
```

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         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public ArrayFiller(TElement[] array, long offset)
15         {
16             _array = array;
17             _position = offset;
18         }
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         public ArrayFiller(TElement[] array) : this(array, 0) { }
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         public void Add(TElement element) => _array[_position++] = element;
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public bool AddAndReturnTrue(TElement element)
28         {
29             _array[_position++] = element;
30             return true;
31         }
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         public bool AddFirstAndReturnTrue(ICollection<TElement> collection)
35         {
36             _array[_position++] = collection[0];
37             return true;
38         }
39     }
40 }
```

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 System.Runtime.CompilerServices;
4 using Platform.Exceptions;
5 using Platform.Disposables;
6 using Platform.Ranges;
7 using Platform.Collections.Stacks;
8
9 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11 namespace Platform.Collections.Arrays
12 {
13     /// <remarks>
14     /// Original idea from
15     ↪ http://geekswithblogs.net/blackrob/archive/2014/12/18/array-pooling-in-csharp.aspx
16     /// </remarks>
17     public class ArrayPool<T>
18     {
19         public static readonly T[] Empty = Array.Empty<T>();
20
21         // May be use Default class for that later.
22         [ThreadStatic]
23         internal static ArrayPool<T> _threadInstance;
24         internal static ArrayPool<T> ThreadInstance { get => _threadInstance ?? (_threadInstance
25             ↪ = new ArrayPool<T>()); }
26
27         private readonly int _maxArraysPerSize;
28         private readonly Dictionary<int, Stack<T[]>> _pool = new Dictionary<int,
29             ↪ Stack<T[]>>(ArrayPool.DefaultSizesAmount);
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         public ArrayPool(int maxArraysPerSize) => _maxArraysPerSize = maxArraysPerSize;
33
34         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35         public ArrayPool() : this(ArrayPool.DefaultMaxArraysPerSize) { }
36
37         [MethodImpl(MethodImplOptions.AggressiveInlining)]
38         public Disposable<T[]> AllocatedDisposable(long size) => (Allocate(size), Free);
39
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         public Disposable<T[]> Resize(Disposable<T[]> source, long size)
42         {
43             var destination = AllocatedDisposable(size);
44             T[] sourceArray = source;
45             T[] destinationArray = destination;
46             Array.Copy(sourceArray, destinationArray, size < sourceArray.Length ? (int)size :
47                 ↪ sourceArray.Length);
48             source.Dispose();
49             return destination;
50         }
51
52         [MethodImpl(MethodImplOptions.AggressiveInlining)]
53         public virtual void Clear() => _pool.Clear();
54
55         [MethodImpl(MethodImplOptions.AggressiveInlining)]
56         public virtual T[] Allocate(long size)
57         {
58             Ensure.Always.ArgumentInRange(size, (0, int.MaxValue));
59         }
60     }
61 }
```

```

55         return size == 0 ? Empty : _pool.GetOrDefault((int)size)?.PopOrDefault() ?? new
           ↪ T[size];
56     }
57
58     [MethodImpl(MethodImplOptions.AggressiveInlining)]
59     public virtual void Free(T[] array)
60     {
61         Ensure.Always.ArgumentNotNull(array, nameof(array));
62         if (array.Length == 0)
63         {
64             return;
65         }
66         var stack = _pool.GetOrAdd(array.Length, size => new Stack<T[]>(_maxArraysPerSize));
67         if (stack.Count == _maxArraysPerSize) // Stack is full
68         {
69             return;
70         }
71         stack.Push(array);
72     }
73 }
74 }

```

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

```

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

```

1.6 ./Platform.Collections/Arrays/CharArrayExtensions.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 unsafe class CharArrayExtensions
8      {
9          /// <remarks>
10         /// Based on https://github.com/Microsoft/referencesource/blob/3b1eaf5203992df69de44c783\_a3eda37d3d4cd10/mscorlib/system/string.cs#L833
11         /// </remarks>
12         [MethodImpl(MethodImplOptions.AggressiveInlining)]
13         public static int GenerateHashCode(this char[] array, int offset, int length)
14         {
15             var hashSeed = 5381;
16             var hashAccumulator = hashSeed;
17             fixed (char* pointer = &array[offset])
18             {
19                 for (char* s = pointer, last = s + length; s < last; s++)
20                 {
21                     hashAccumulator = (hashAccumulator << 5) + hashAccumulator ^ *s;
22                 }
23             }
24             return hashAccumulator + (hashSeed * 1566083941);
25         }
26
27         /// <remarks>
28         /// Based on https://github.com/Microsoft/referencesource/blob/3b1eaf5203992df69de44c783\_a3eda37d3d4cd10/mscorlib/system/string.cs#L364
29         /// </remarks>
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         public static bool ContentEqualTo(this char[] left, int leftOffset, int length, char[]
           ↪ right, int rightOffset)
32         {

```

```

33     fixed (char* leftPointer = &left[leftOffset])
34     {
35         fixed (char* rightPointer = &right[rightOffset])
36         {
37             char* leftPointerCopy = leftPointer, rightPointerCopy = rightPointer;
38             if (!CheckArraysMainPartForEquality(ref leftPointerCopy, ref
39                 ↪ rightPointerCopy, ref length))
40             {
41                 return false;
42             }
43             CheckArraysRemainderForEquality(ref leftPointerCopy, ref rightPointerCopy,
44                 ↪ ref length);
45             return length <= 0;
46         }
47     }
48     [MethodImpl(MethodImplOptions.AggressiveInlining)]
49     private static bool CheckArraysMainPartForEquality(ref char* left, ref char* right, ref
50     ↪ int length)
51     {
52         while (length >= 10)
53         {
54             if ((* (int*)left != * (int*)right)
55                 || (* (int*)(left + 2) != * (int*)(right + 2))
56                 || (* (int*)(left + 4) != * (int*)(right + 4))
57                 || (* (int*)(left + 6) != * (int*)(right + 6))
58                 || (* (int*)(left + 8) != * (int*)(right + 8)))
59             {
60                 return false;
61             }
62             left += 10;
63             right += 10;
64             length -= 10;
65         }
66         return true;
67     }
68     [MethodImpl(MethodImplOptions.AggressiveInlining)]
69     private static void CheckArraysRemainderForEquality(ref char* left, ref char* right, ref
70     ↪ int length)
71     {
72         // This depends on the fact that the String objects are
73         // always zero terminated and that the terminating zero is not included
74         // in the length. For odd string sizes, the last compare will include
75         // the zero terminator.
76         while (length > 0)
77         {
78             if ((* (int*)left != * (int*)right)
79                 {
80                 break;
81             }
82             left += 2;
83             right += 2;
84             length -= 2;
85         }
86     }
87 }

```

1.7 ./Platform.Collections/Arrays/GenericArrayExtensions.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 GenericArrayExtensions
10     {
11         [MethodImpl(MethodImplOptions.AggressiveInlining)]
12         public static T[] Clone<T>(this T[] array)
13         {
14             var copy = new T[array.Length];
15             Array.Copy(array, 0, copy, 0, array.Length);
16             return copy;
17         }
18     }

```

```

19     [MethodImpl(MethodImplOptions.AggressiveInlining)]
20     public static IList<T> ShiftRight<T>(this T[] array) => array.ShiftRight(1);
21
22     [MethodImpl(MethodImplOptions.AggressiveInlining)]
23     public static IList<T> ShiftRight<T>(this T[] array, int shift)
24     {
25         var restrictions = new T[array.Length + shift];
26         Array.Copy(array, 0, restrictions, shift, array.Length);
27         return restrictions;
28     }
29 }
30 }

```

1.8 ./Platform.Collections/BitString.cs

```

1  using System;
2  using System.Collections.Concurrent;
3  using System.Collections.Generic;
4  using System.Numerics;
5  using System.Runtime.CompilerServices;
6  using System.Threading.Tasks;
7  using Platform.Exceptions;
8  using Platform.Ranges;
9
10 // ReSharper disable ForCanBeConvertedToForeach
11 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
12
13 namespace Platform.Collections
14 {
15     /// <remarks>
16     /// А что если хранить карту значений, где каждый бит будет означать присутствует ли блок в
17     /// → 64 бит в массиве значений.
18     /// 64 бита по 0 бит, будут означать отсутствие 64-х блоков по 64 бита. Т.е. упаковка 512
19     /// → байт в 8 байт.
20     /// Подобный принцип можно применять и к 64-ём блокам и т.п. По сути это карта значений. С
21     /// → помощью которой можно быстро
22     /// проверять есть ли значения непосредственно далее (ниже по уровню).
23     /// Или как таблица виртуальной памяти где номер блока означает его присутствие и адрес.
24     /// </remarks>
25     public class BitString : IEquatable<BitString>
26     {
27         private static readonly byte[] [] _bitsSetIn16Bits;
28         private long[] _array;
29         private long _length;
30         private long _minPositiveWord;
31         private long _maxPositiveWord;
32
33         public bool this[long index]
34         {
35             [MethodImpl(MethodImplOptions.AggressiveInlining)]
36             get => Get(index);
37             [MethodImpl(MethodImplOptions.AggressiveInlining)]
38             set => Set(index, value);
39         }
40
41         public long Length
42         {
43             [MethodImpl(MethodImplOptions.AggressiveInlining)]
44             get => _length;
45             [MethodImpl(MethodImplOptions.AggressiveInlining)]
46             set
47             {
48                 if (_length == value)
49                 {
50                     return;
51                 }
52                 Ensure.Always.ArgumentInRange(value, GetValidLengthRange(), nameof(Length));
53                 // Currently we never shrink the array
54                 if (value > _length)
55                 {
56                     var words = GetWordsCountFromIndex(value);
57                     var oldWords = GetWordsCountFromIndex(_length);
58                     if (words > _array.LongLength)
59                     {
60                         var copy = new long[words];
61                         Array.Copy(_array, copy, _array.LongLength);
62                         _array = copy;
63                     }
64                     else
65                     {
66                         // What is going on here?
67                     }
68                 }
69             }
70         }
71     }
72 }

```

```

64         Array.Clear(_array, (int)oldWords, (int)(words - oldWords));
65     }
66     // What is going on here?
67     var mask = (int)(_length % 64);
68     if (mask > 0)
69     {
70         _array[oldWords - 1] &= (1L << mask) - 1;
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
82 #region Constructors
83
84 [MethodImpl(MethodImplOptions.AggressiveInlining)]
85 static BitString()
86 {
87     _bitsSetIn16Bits = new byte[65536][];
88     int i, c, k;
89     byte bitIndex;
90     for (i = 0; i < 65536; i++)
91     {
92         // Calculating size of array (number of positive bits)
93         for (c = 0, k = 1; k <= 65536; k <= 1)
94         {
95             if ((i & k) == k)
96             {
97                 c++;
98             }
99         }
100         var array = new byte[c];
101         // Adding positive bits indices into array
102         for (bitIndex = 0, c = 0, k = 1; k <= 65536; k <= 1)
103         {
104             if ((i & k) == k)
105             {
106                 array[c++] = bitIndex;
107             }
108             bitIndex++;
109         }
110         _bitsSetIn16Bits[i] = array;
111     }
112 }
113
114 [MethodImpl(MethodImplOptions.AggressiveInlining)]
115 public BitString(BitString other)
116 {
117     Ensure.Always.ArgumentNotNull(other, nameof(other));
118     _length = other._length;
119     _array = new long[GetWordsCountFromIndex(_length)];
120     _minPositiveWord = other._minPositiveWord;
121     _maxPositiveWord = other._maxPositiveWord;
122     Array.Copy(other._array, _array, _array.LongLength);
123 }
124
125 [MethodImpl(MethodImplOptions.AggressiveInlining)]
126 public BitString(long length)
127 {
128     Ensure.Always.ArgumentInRange(length, GetValidLengthRange(), nameof(length));
129     _length = length;
130     _array = new long[GetWordsCountFromIndex(_length)];
131     MarkBordersAsAllBitsReset();
132 }
133
134 [MethodImpl(MethodImplOptions.AggressiveInlining)]
135 public BitString(long length, bool defaultValue)
136     : this(length)
137 {
138     if (defaultValue)
139     {
140         SetAll();
141     }
142 }

```

```

143 #endregion
144
145 [MethodImpl(MethodImplOptions.AggressiveInlining)]
146 public BitString Not()
147 {
148     for (var i = 0; i < _array.Length; i++)
149     {
150         _array[i] = ~_array[i];
151         RefreshBordersByWord(i);
152     }
153     return this;
154 }
155
156 [MethodImpl(MethodImplOptions.AggressiveInlining)]
157 public BitString ParallelNot()
158 {
159     var processorCount = Environment.ProcessorCount;
160     if (processorCount <= 1)
161     {
162         return Not();
163     }
164     var partitioner = Partitioner.Create(0, _array.Length, _array.Length /
165         ↪ processorCount);
166     Parallel.ForEach(partitioner.GetDynamicPartitions(), range =>
167     {
168         var maximum = range.Item2;
169         for (var i = range.Item1; i < maximum; i++)
170         {
171             _array[i] = ~_array[i];
172         }
173     });
174     MarkBordersAsAllBitsSet();
175     TryShrinkBorders();
176     return this;
177 }
178
179 [MethodImpl(MethodImplOptions.AggressiveInlining)]
180 public BitString VectorNot()
181 {
182     if (!Vector.IsHardwareAccelerated)
183     {
184         return Not();
185     }
186     var step = Vector<long>.Count;
187     if (_array.Length < step)
188     {
189         return Not();
190     }
191     VectorNotLoop(_array, step, 0, _array.Length);
192     MarkBordersAsAllBitsSet();
193     TryShrinkBorders();
194     return this;
195 }
196
197 [MethodImpl(MethodImplOptions.AggressiveInlining)]
198 public BitString ParallelVectorNot()
199 {
200     var processorCount = Environment.ProcessorCount;
201     if (processorCount <= 1 && Vector.IsHardwareAccelerated)
202     {
203         return VectorNot();
204     }
205     if (!Vector.IsHardwareAccelerated)
206     {
207         return Not();
208     }
209     var step = Vector<long>.Count;
210     if (_array.Length < (step * Environment.ProcessorCount))
211     {
212         return VectorNot();
213     }
214     var partitioner = Partitioner.Create(0, _array.Length, _array.Length /
215         ↪ processorCount);
216     Parallel.ForEach(partitioner.GetDynamicPartitions(), range => VectorNotLoop(_array,
217         ↪ step, range.Item1, range.Item2));
218     MarkBordersAsAllBitsSet();
219     TryShrinkBorders();

```

```

218         return this;
219     }
220
221     [MethodImpl(MethodImplOptions.AggressiveInlining)]
222     static private void VectorNotLoop(long[] array, int step, int start, int maximum)
223     {
224         var i = start;
225         var range = maximum - start - 1;
226         var stop = range - (range % step);
227         for (; i < stop; i += step)
228         {
229             var vector = new Vector<long>(array, i);
230             (~vector).CopyTo(array, i);
231         }
232         for (; i < maximum; i++)
233         {
234             array[i] = ~array[i];
235         }
236     }
237
238     [MethodImpl(MethodImplOptions.AggressiveInlining)]
239     public BitString And(BitString other)
240     {
241         EnsureBitStringHasTheSameSize(other, nameof(other));
242         GetCommonOuterBorders(this, other, out long from, out long to);
243         var otherArray = other._array;
244         for (var i = from; i <= to; i++)
245         {
246             _array[i] &= otherArray[i];
247             RefreshBordersByWord(i);
248         }
249         return this;
250     }
251
252     [MethodImpl(MethodImplOptions.AggressiveInlining)]
253     public BitString ParallelAnd(BitString other)
254     {
255         var processorCount = Environment.ProcessorCount;
256         if (processorCount <= 1)
257         {
258             return And(other);
259         }
260         EnsureBitStringHasTheSameSize(other, nameof(other));
261         GetCommonOuterBorders(this, other, out long from, out long to);
262         var partitioner = Partitioner.Create(from, to + 1, (to - from) / processorCount);
263         Parallel.ForEach(partitioner.GetDynamicPartitions(), range =>
264         {
265             var maximum = range.Item2;
266             for (var i = range.Item1; i < maximum; i++)
267             {
268                 _array[i] &= other._array[i];
269             }
270         });
271         MarkBordersAsAllBitsSet();
272         TryShrinkBorders();
273         return this;
274     }
275
276     [MethodImpl(MethodImplOptions.AggressiveInlining)]
277     public BitString VectorAnd(BitString other)
278     {
279         if (!Vector.IsHardwareAccelerated)
280         {
281             return And(other);
282         }
283         var step = Vector<long>.Count;
284         if (_array.Length < step)
285         {
286             return And(other);
287         }
288         EnsureBitStringHasTheSameSize(other, nameof(other));
289         GetCommonOuterBorders(this, other, out long from, out long to);
290         VectorAndLoop(_array, other._array, step, (int)from, (int)(to + 1));
291         MarkBordersAsAllBitsSet();
292         TryShrinkBorders();
293         return this;
294     }
295
296     [MethodImpl(MethodImplOptions.AggressiveInlining)]

```



```

297 public BitString ParallelVectorAnd(BitString other)
298 {
299     var processorCount = Environment.ProcessorCount;
300     if (processorCount <= 1 && Vector.IsHardwareAccelerated)
301     {
302         return VectorAnd(other);
303     }
304     if (!Vector.IsHardwareAccelerated)
305     {
306         return And(other);
307     }
308     var step = Vector<long>.Count;
309     if (_array.Length < (step * Environment.ProcessorCount))
310     {
311         return VectorAnd(other);
312     }
313     EnsureBitStringHasTheSameSize(other, nameof(other));
314     GetCommonOuterBorders(this, other, out long from, out long to);
315     var partitioner = Partitioner.Create(from, to + 1, (to - from) / processorCount);
316     Parallel.ForEach(partitioner.GetDynamicPartitions(), range => VectorAndLoop(_array,
317         ↪ other._array, step, (int)range.Item1, (int)range.Item2));
318     MarkBordersAsAllBitsSet();
319     TryShrinkBorders();
320     return this;
321 }
322 [MethodImpl(MethodImplOptions.AggressiveInlining)]
323 static private void VectorAndLoop(long[] array, long[] otherArray, int step, int start,
324     ↪ int maximum)
325 {
326     var i = start;
327     var range = maximum - start - 1;
328     var stop = range - (range % step);
329     for (; i < stop; i += step)
330     {
331         var thisVector = new Vector<long>(array, i);
332         var otherVector = new Vector<long>(otherArray, i);
333         (thisVector & otherVector).CopyTo(array, i);
334     }
335     for (; i < maximum; i++)
336     {
337         array[i] &= otherArray[i];
338     }
339 }
340 [MethodImpl(MethodImplOptions.AggressiveInlining)]
341 public BitString Or(BitString other)
342 {
343     EnsureBitStringHasTheSameSize(other, nameof(other));
344     GetCommonOuterBorders(this, other, out long from, out long to);
345     for (var i = from; i <= to; i++)
346     {
347         _array[i] |= other._array[i];
348         RefreshBordersByWord(i);
349     }
350     return this;
351 }
352 [MethodImpl(MethodImplOptions.AggressiveInlining)]
353 public BitString ParallelOr(BitString other)
354 {
355     var processorCount = Environment.ProcessorCount;
356     if (processorCount <= 1)
357     {
358         return Or(other);
359     }
360     EnsureBitStringHasTheSameSize(other, nameof(other));
361     GetCommonOuterBorders(this, other, out long from, out long to);
362     var partitioner = Partitioner.Create(from, to + 1, (to - from) / processorCount);
363     Parallel.ForEach(partitioner.GetDynamicPartitions(), range =>
364     {
365         var maximum = range.Item2;
366         for (var i = range.Item1; i < maximum; i++)
367         {
368             _array[i] |= other._array[i];
369         }
370     });
371     MarkBordersAsAllBitsSet();
372 }

```

```

373     TryShrinkBorders();
374     return this;
375 }
376
377 [MethodImpl(MethodImplOptions.AggressiveInlining)]
378 public BitString VectorOr(BitString other)
379 {
380     if (!Vector.IsHardwareAccelerated)
381     {
382         return Or(other);
383     }
384     var step = Vector<long>.Count;
385     if (_array.Length < step)
386     {
387         return Or(other);
388     }
389     EnsureBitStringHasTheSameSize(other, nameof(other));
390     GetCommonOuterBorders(this, other, out long from, out long to);
391     VectorOrLoop(_array, other._array, step, (int)from, (int)(to + 1));
392     MarkBordersAsAllBitsSet();
393     TryShrinkBorders();
394     return this;
395 }
396
397 [MethodImpl(MethodImplOptions.AggressiveInlining)]
398 public BitString ParallelVectorOr(BitString other)
399 {
400     var processorCount = Environment.ProcessorCount;
401     if (processorCount <= 1 && Vector.IsHardwareAccelerated)
402     {
403         return VectorOr(other);
404     }
405     if (!Vector.IsHardwareAccelerated)
406     {
407         return Or(other);
408     }
409     var step = Vector<long>.Count;
410     if (_array.Length < (step * Environment.ProcessorCount))
411     {
412         return VectorOr(other);
413     }
414     EnsureBitStringHasTheSameSize(other, nameof(other));
415     GetCommonOuterBorders(this, other, out long from, out long to);
416     var partitioner = Partitioner.Create(from, to + 1, (to - from) / processorCount);
417     Parallel.ForEach(partitioner.GetDynamicPartitions(), range => VectorOrLoop(_array,
418 ↪ other._array, step, (int)range.Item1, (int)range.Item2));
419     MarkBordersAsAllBitsSet();
420     TryShrinkBorders();
421     return this;
422 }
423
424 [MethodImpl(MethodImplOptions.AggressiveInlining)]
425 static private void VectorOrLoop(long[] array, long[] otherArray, int step, int start,
426 ↪ int maximum)
427 {
428     var i = start;
429     var range = maximum - start - 1;
430     var stop = range - (range % step);
431     for (; i < stop; i += step)
432     {
433         var thisVector = new Vector<long>(array, i);
434         var otherVector = new Vector<long>(otherArray, i);
435         (thisVector | otherVector).CopyTo(array, i);
436     }
437     for (; i < maximum; i++)
438     {
439         array[i] |= otherArray[i];
440     }
441 }
442
443 [MethodImpl(MethodImplOptions.AggressiveInlining)]
444 public BitString Xor(BitString other)
445 {
446     EnsureBitStringHasTheSameSize(other, nameof(other));
447     GetCommonOuterBorders(this, other, out long from, out long to);
448     for (var i = from; i <= to; i++)
449     {
450         _array[i] ^= other._array[i];
451     }
452 }

```

```

449         RefreshBordersByWord(i);
450     }
451     return this;
452 }
453
454 [MethodImpl(MethodImplOptions.AggressiveInlining)]
455 public BitString ParallelXor(BitString other)
456 {
457     var processorCount = Environment.ProcessorCount;
458     if (processorCount <= 1)
459     {
460         return Xor(other);
461     }
462     EnsureBitStringHasTheSameSize(other, nameof(other));
463     GetCommonOuterBorders(this, other, out long from, out long to);
464     var partitioner = Partitioner.Create(from, to + 1, (to - from) / processorCount);
465     Parallel.ForEach(partitioner.GetDynamicPartitions(), range =>
466     {
467         var maximum = range.Item2;
468         for (var i = range.Item1; i < maximum; i++)
469         {
470             _array[i] ^= other._array[i];
471         }
472     });
473     MarkBordersAsAllBitsSet();
474     TryShrinkBorders();
475     return this;
476 }
477
478 [MethodImpl(MethodImplOptions.AggressiveInlining)]
479 public BitString VectorXor(BitString other)
480 {
481     if (!Vector.IsHardwareAccelerated)
482     {
483         return Xor(other);
484     }
485     var step = Vector<long>.Count;
486     if (_array.Length < step)
487     {
488         return Xor(other);
489     }
490     EnsureBitStringHasTheSameSize(other, nameof(other));
491     GetCommonOuterBorders(this, other, out long from, out long to);
492     VectorXorLoop(_array, other._array, step, (int)from, (int)(to + 1));
493     MarkBordersAsAllBitsSet();
494     TryShrinkBorders();
495     return this;
496 }
497
498 [MethodImpl(MethodImplOptions.AggressiveInlining)]
499 public BitString ParallelVectorXor(BitString other)
500 {
501     var processorCount = Environment.ProcessorCount;
502     if (processorCount <= 1 && Vector.IsHardwareAccelerated)
503     {
504         return VectorXor(other);
505     }
506     if (!Vector.IsHardwareAccelerated)
507     {
508         return Xor(other);
509     }
510     var step = Vector<long>.Count;
511     if (_array.Length < (step * Environment.ProcessorCount))
512     {
513         return VectorXor(other);
514     }
515     EnsureBitStringHasTheSameSize(other, nameof(other));
516     GetCommonOuterBorders(this, other, out long from, out long to);
517     var partitioner = Partitioner.Create(from, to + 1, (to - from) / processorCount);
518     Parallel.ForEach(partitioner.GetDynamicPartitions(), range => VectorXorLoop(_array,
519         ↪ other._array, step, (int)range.Item1, (int)range.Item2));
519     MarkBordersAsAllBitsSet();
520     TryShrinkBorders();
521     return this;
522 }
523
524 [MethodImpl(MethodImplOptions.AggressiveInlining)]
525 static private void VectorXorLoop(long[] array, long[] otherArray, int step, int start,
526     ↪ int maximum)

```

```

526 {
527     var i = start;
528     var range = maximum - start - 1;
529     var stop = range - (range % step);
530     for (; i < stop; i += step)
531     {
532         var thisVector = new Vector<long>(array, i);
533         var otherVector = new Vector<long>(otherArray, i);
534         (thisVector ^ otherVector).CopyTo(array, i);
535     }
536     for (; i < maximum; i++)
537     {
538         array[i] ^= otherArray[i];
539     }
540 }
541
542 [MethodImpl(MethodImplOptions.AggressiveInlining)]
543 private void RefreshBordersByWord(long wordIndex)
544 {
545     if (_array[wordIndex] == 0)
546     {
547         if (wordIndex == _minPositiveWord && wordIndex != _array.LongLength - 1)
548         {
549             _minPositiveWord++;
550         }
551         if (wordIndex == _maxPositiveWord && wordIndex != 0)
552         {
553             _maxPositiveWord--;
554         }
555     }
556     else
557     {
558         if (wordIndex < _minPositiveWord)
559         {
560             _minPositiveWord = wordIndex;
561         }
562         if (wordIndex > _maxPositiveWord)
563         {
564             _maxPositiveWord = wordIndex;
565         }
566     }
567 }
568
569 [MethodImpl(MethodImplOptions.AggressiveInlining)]
570 public bool TryShrinkBorders()
571 {
572     GetBorders(out long from, out long to);
573     while (from <= to && _array[from] == 0)
574     {
575         from++;
576     }
577     if (from > to)
578     {
579         MarkBordersAsAllBitsReset();
580         return true;
581     }
582     while (to >= from && _array[to] == 0)
583     {
584         to--;
585     }
586     if (to < from)
587     {
588         MarkBordersAsAllBitsReset();
589         return true;
590     }
591     var bordersUpdated = from != _minPositiveWord || to != _maxPositiveWord;
592     if (bordersUpdated)
593     {
594         SetBorders(from, to);
595     }
596     return bordersUpdated;
597 }
598
599 [MethodImpl(MethodImplOptions.AggressiveInlining)]
600 public bool Get(long index)
601 {
602     Ensure.Always.ArgumentInRange(index, GetValidIndexRange(), nameof(index));
603     return (_array[GetWordIndexFromIndex(index)] & GetBitMaskFromIndex(index)) != 0;
604 }

```

```

605 [MethodImpl(MethodImplOptions.AggressiveInlining)]
606 public void Set(long index, bool value)
607 {
608     if (value)
609     {
610         Set(index);
611     }
612     else
613     {
614         Reset(index);
615     }
616 }
617
618 [MethodImpl(MethodImplOptions.AggressiveInlining)]
619 public void Set(long index)
620 {
621     Ensure.Always.ArgumentInRange(index, GetValidIndexRange(), nameof(index));
622     var wordIndex = GetWordIndexFromIndex(index);
623     var mask = GetBitMaskFromIndex(index);
624     _array[wordIndex] |= mask;
625     RefreshBordersByWord(wordIndex);
626 }
627
628 [MethodImpl(MethodImplOptions.AggressiveInlining)]
629 public void Reset(long index)
630 {
631     Ensure.Always.ArgumentInRange(index, GetValidIndexRange(), nameof(index));
632     var wordIndex = GetWordIndexFromIndex(index);
633     var mask = GetBitMaskFromIndex(index);
634     _array[wordIndex] &= ~mask;
635     RefreshBordersByWord(wordIndex);
636 }
637
638 [MethodImpl(MethodImplOptions.AggressiveInlining)]
639 public bool Add(long index)
640 {
641     var wordIndex = GetWordIndexFromIndex(index);
642     var mask = GetBitMaskFromIndex(index);
643     if ((_array[wordIndex] & mask) == 0)
644     {
645         _array[wordIndex] |= mask;
646         RefreshBordersByWord(wordIndex);
647         return true;
648     }
649     else
650     {
651         return false;
652     }
653 }
654
655 [MethodImpl(MethodImplOptions.AggressiveInlining)]
656 public void SetAll(bool value)
657 {
658     if (value)
659     {
660         SetAll();
661     }
662     else
663     {
664         ResetAll();
665     }
666 }
667
668 [MethodImpl(MethodImplOptions.AggressiveInlining)]
669 public void SetAll()
670 {
671     const long fillValue = unchecked((long)0xffffffffffffffff);
672     var words = GetWordsCountFromIndex(_length);
673     for (var i = 0; i < words; i++)
674     {
675         _array[i] = fillValue;
676     }
677     MarkBordersAsAllBitsSet();
678 }
679
680 [MethodImpl(MethodImplOptions.AggressiveInlining)]
681 public void ResetAll()
682 {
683

```

```

684     const long fillValue = 0;
685     GetBorders(out long from, out long to);
686     for (var i = from; i <= to; i++)
687     {
688         _array[i] = fillValue;
689     }
690     MarkBordersAsAllBitsReset();
691 }
692
693 [MethodImpl(MethodImplOptions.AggressiveInlining)]
694 public List<long> GetSetIndices()
695 {
696     var result = new List<long>();
697     GetBorders(out long from, out long to);
698     for (var i = from; i <= to; i++)
699     {
700         var word = _array[i];
701         if (word != 0)
702         {
703             AppendAllSetBitIndices(result, i, word);
704         }
705     }
706     return result;
707 }
708
709 [MethodImpl(MethodImplOptions.AggressiveInlining)]
710 public List<ulong> GetSetUInt64Indices()
711 {
712     var result = new List<ulong>();
713     GetBorders(out ulong from, out ulong to);
714     for (var i = from; i <= to; i++)
715     {
716         var word = _array[i];
717         if (word != 0)
718         {
719             AppendAllSetBitIndices(result, i, word);
720         }
721     }
722     return result;
723 }
724
725 [MethodImpl(MethodImplOptions.AggressiveInlining)]
726 public long GetFirstSetBitIndex()
727 {
728     var i = _minPositiveWord;
729     var word = _array[i];
730     if (word != 0)
731     {
732         return GetFirstSetBitForWord(i, word);
733     }
734     return -1;
735 }
736
737 [MethodImpl(MethodImplOptions.AggressiveInlining)]
738 public long GetLastSetBitIndex()
739 {
740     var i = _maxPositiveWord;
741     var word = _array[i];
742     if (word != 0)
743     {
744         return GetLastSetBitForWord(i, word);
745     }
746     return -1;
747 }
748
749 [MethodImpl(MethodImplOptions.AggressiveInlining)]
750 public long CountSetBits()
751 {
752     var total = 0L;
753     GetBorders(out long from, out long to);
754     for (var i = from; i <= to; i++)
755     {
756         var word = _array[i];
757         if (word != 0)
758         {
759             total += CountSetBitsForWord(word);
760         }
761     }
762     return total;

```

```

763     }
764
765     [MethodImpl(MethodImplOptions.AggressiveInlining)]
766     public bool HaveCommonBits(BitString other)
767     {
768         EnsureBitStringHasTheSameSize(other, nameof(other));
769         GetCommonInnerBorders(this, other, out long from, out long to);
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             if (left != 0 && right != 0 && (left & right) != 0)
776             {
777                 return true;
778             }
779         }
780         return false;
781     }
782
783     [MethodImpl(MethodImplOptions.AggressiveInlining)]
784     public long CountCommonBits(BitString other)
785     {
786         EnsureBitStringHasTheSameSize(other, nameof(other));
787         GetCommonInnerBorders(this, other, out long from, out long to);
788         var total = 0L;
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                 total += CountSetBitsForWord(combined);
798             }
799         }
800         return total;
801     }
802
803     [MethodImpl(MethodImplOptions.AggressiveInlining)]
804     public List<long> GetCommonIndices(BitString other)
805     {
806         EnsureBitStringHasTheSameSize(other, nameof(other));
807         GetCommonInnerBorders(this, other, out long from, out long to);
808         var result = new List<long>();
809         var otherArray = other._array;
810         for (var i = from; i <= to; i++)
811         {
812             var left = _array[i];
813             var right = otherArray[i];
814             var combined = left & right;
815             if (combined != 0)
816             {
817                 AppendAllSetBitIndices(result, i, combined);
818             }
819         }
820         return result;
821     }
822
823     [MethodImpl(MethodImplOptions.AggressiveInlining)]
824     public List<ulong> GetCommonUInt64Indices(BitString other)
825     {
826         EnsureBitStringHasTheSameSize(other, nameof(other));
827         GetCommonBorders(this, other, out ulong from, out ulong to);
828         var result = new List<ulong>();
829         var otherArray = other._array;
830         for (var i = from; i <= to; i++)
831         {
832             var left = _array[i];
833             var right = otherArray[i];
834             var combined = left & right;
835             if (combined != 0)
836             {
837                 AppendAllSetBitIndices(result, i, combined);
838             }
839         }
840         return result;
841     }

```

```

842 [MethodImpl(MethodImplOptions.AggressiveInlining)]
843 public long GetFirstCommonBitIndex(BitString other)
844 {
845     EnsureBitStringHasTheSameSize(other, nameof(other));
846     GetCommonInnerBorders(this, other, out long from, out long to);
847     var otherArray = other._array;
848     for (var i = from; i <= to; i++)
849     {
850         var left = _array[i];
851         var right = otherArray[i];
852         var combined = left & right;
853         if (combined != 0)
854         {
855             return GetFirstSetBitForWord(i, combined);
856         }
857     }
858     return -1;
859 }
860
861 [MethodImpl(MethodImplOptions.AggressiveInlining)]
862 public long GetLastCommonBitIndex(BitString other)
863 {
864     EnsureBitStringHasTheSameSize(other, nameof(other));
865     GetCommonInnerBorders(this, other, out long from, out long to);
866     var otherArray = other._array;
867     for (var i = to; i >= from; i--)
868     {
869         var left = _array[i];
870         var right = otherArray[i];
871         var combined = left & right;
872         if (combined != 0)
873         {
874             return GetLastSetBitForWord(i, combined);
875         }
876     }
877     return -1;
878 }
879
880 [MethodImpl(MethodImplOptions.AggressiveInlining)]
881 public override bool Equals(object obj) => obj is BitString @string ? Equals(@string) :
882     => false;
883
884 [MethodImpl(MethodImplOptions.AggressiveInlining)]
885 public bool Equals(BitString other)
886 {
887     if (_length != other._length)
888     {
889         return false;
890     }
891     var otherArray = other._array;
892     if (_array.Length != otherArray.Length)
893     {
894         return false;
895     }
896     if (_minPositiveWord != other._minPositiveWord)
897     {
898         return false;
899     }
900     if (_maxPositiveWord != other._maxPositiveWord)
901     {
902         return false;
903     }
904     GetCommonBorders(this, other, out ulong from, out ulong to);
905     for (var i = from; i <= to; i++)
906     {
907         if (_array[i] != otherArray[i])
908         {
909             return false;
910         }
911     }
912     return true;
913 }
914
915 [MethodImpl(MethodImplOptions.AggressiveInlining)]
916 private void EnsureBitStringHasTheSameSize(BitString other, string argumentName)
917 {
918     Ensure.Always.ArgumentNotNull(other, argumentName);
919     if (_length != other._length)

```



```

920     {
921         throw new ArgumentException("Bit string must be the same size.", argumentName);
922     }
923 }
924
925 [MethodImpl(MethodImplOptions.AggressiveInlining)]
926 private void MarkBordersAsAllBitsReset() => SetBorders(_array.LongLength - 1, 0);
927
928 [MethodImpl(MethodImplOptions.AggressiveInlining)]
929 private void MarkBordersAsAllBitsSet() => SetBorders(0, _array.LongLength - 1);
930
931 [MethodImpl(MethodImplOptions.AggressiveInlining)]
932 private void GetBorders(out long from, out long to)
933 {
934     from = _minPositiveWord;
935     to = _maxPositiveWord;
936 }
937
938 [MethodImpl(MethodImplOptions.AggressiveInlining)]
939 private void GetBorders(out ulong from, out ulong to)
940 {
941     from = (ulong)_minPositiveWord;
942     to = (ulong)_maxPositiveWord;
943 }
944
945 [MethodImpl(MethodImplOptions.AggressiveInlining)]
946 private void SetBorders(long from, long to)
947 {
948     _minPositiveWord = from;
949     _maxPositiveWord = to;
950 }
951
952 [MethodImpl(MethodImplOptions.AggressiveInlining)]
953 private Range<long> GetValidIndexRange() => (0, _length - 1);
954
955 [MethodImpl(MethodImplOptions.AggressiveInlining)]
956 private static Range<long> GetValidLengthRange() => (0, long.MaxValue);
957
958 [MethodImpl(MethodImplOptions.AggressiveInlining)]
959 private static void AppendAllSetBitIndices(List<ulong> result, ulong wordIndex, long
    ↪ wordValue)
960 {
961     GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
    ↪ bits32to47, out byte[] bits48to63);
962     AppendAllSetIndices(result, wordIndex, bits00to15, bits16to31, bits32to47,
    ↪ bits48to63);
963 }
964
965 [MethodImpl(MethodImplOptions.AggressiveInlining)]
966 private static void AppendAllSetBitIndices(List<long> result, long wordIndex, long
    ↪ wordValue)
967 {
968     GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
    ↪ bits32to47, out byte[] bits48to63);
969     AppendAllSetBitIndices(result, wordIndex, bits00to15, bits16to31, bits32to47,
    ↪ bits48to63);
970 }
971
972 [MethodImpl(MethodImplOptions.AggressiveInlining)]
973 private static long CountSetBitsForWord(long word)
974 {
975     GetBits(word, out byte[] bits00to15, out byte[] bits16to31, out byte[] bits32to47,
    ↪ out byte[] bits48to63);
976     return bits00to15.LongLength + bits16to31.LongLength + bits32to47.LongLength +
    ↪ bits48to63.LongLength;
977 }
978
979 [MethodImpl(MethodImplOptions.AggressiveInlining)]
980 private static long GetFirstSetBitForWord(long wordIndex, long wordValue)
981 {
982     GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
    ↪ bits32to47, out byte[] bits48to63);
983     return GetFirstSetBit(wordIndex, bits00to15, bits16to31, bits32to47, bits48to63);
984 }
985
986 [MethodImpl(MethodImplOptions.AggressiveInlining)]
987 private static long GetLastSetBitForWord(long wordIndex, long wordValue)
988 {

```

```

989         GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
990             ↪ bits32to47, out byte[] bits48to63);
991         return GetLastSetBit(wordIndex, bits00to15, bits16to31, bits32to47, bits48to63);
992     }
993     [MethodImpl(MethodImplOptions.AggressiveInlining)]
994     private static void AppendAllSetBitIndices(List<long> result, long i, byte[] bits00to15,
995     ↪ byte[] bits16to31, byte[] bits32to47, byte[] bits48to63)
996     {
997         for (var j = 0; j < bits00to15.Length; j++)
998         {
999             result.Add(bits00to15[j] + (i * 64));
1000         }
1001         for (var j = 0; j < bits16to31.Length; j++)
1002         {
1003             result.Add(bits16to31[j] + 16 + (i * 64));
1004         }
1005         for (var j = 0; j < bits32to47.Length; j++)
1006         {
1007             result.Add(bits32to47[j] + 32 + (i * 64));
1008         }
1009         for (var j = 0; j < bits48to63.Length; j++)
1010         {
1011             result.Add(bits48to63[j] + 48 + (i * 64));
1012         }
1013     }
1014     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1015     private static void AppendAllSetIndices(List<ulong> result, ulong i, byte[] bits00to15,
1016     ↪ byte[] bits16to31, byte[] bits32to47, byte[] bits48to63)
1017     {
1018         for (var j = 0; j < bits00to15.Length; j++)
1019         {
1020             result.Add(bits00to15[j] + (i * 64));
1021         }
1022         for (var j = 0; j < bits16to31.Length; j++)
1023         {
1024             result.Add(bits16to31[j] + 16UL + (i * 64));
1025         }
1026         for (var j = 0; j < bits32to47.Length; j++)
1027         {
1028             result.Add(bits32to47[j] + 32UL + (i * 64));
1029         }
1030         for (var j = 0; j < bits48to63.Length; j++)
1031         {
1032             result.Add(bits48to63[j] + 48UL + (i * 64));
1033         }
1034     }
1035     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1036     private static long GetFirstSetBit(long i, byte[] bits00to15, byte[] bits16to31, byte[]
1037     ↪ bits32to47, byte[] bits48to63)
1038     {
1039         if (bits00to15.Length > 0)
1040         {
1041             return bits00to15[0] + (i * 64);
1042         }
1043         if (bits16to31.Length > 0)
1044         {
1045             return bits16to31[0] + 16 + (i * 64);
1046         }
1047         if (bits32to47.Length > 0)
1048         {
1049             return bits32to47[0] + 32 + (i * 64);
1050         }
1051         return bits48to63[0] + 48 + (i * 64);
1052     }
1053     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1054     private static long GetLastSetBit(long i, byte[] bits00to15, byte[] bits16to31, byte[]
1055     ↪ bits32to47, byte[] bits48to63)
1056     {
1057         if (bits48to63.Length > 0)
1058         {
1059             return bits48to63[bits48to63.Length - 1] + 48 + (i * 64);
1060         }
1061         if (bits32to47.Length > 0)

```

```

1061     {
1062         return bits32to47[bits32to47.Length - 1] + 32 + (i * 64);
1063     }
1064     if (bits16to31.Length > 0)
1065     {
1066         return bits16to31[bits16to31.Length - 1] + 16 + (i * 64);
1067     }
1068     return bits00to15[bits00to15.Length - 1] + (i * 64);
1069 }
1070
1071 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1072 private static void GetBits(long word, out byte[] bits00to15, out byte[] bits16to31, out
    ↪ byte[] bits32to47, out byte[] bits48to63)
1073 {
1074     bits00to15 = _bitsSetIn16Bits[word & 0xffffu];
1075     bits16to31 = _bitsSetIn16Bits[(word >> 16) & 0xffffu];
1076     bits32to47 = _bitsSetIn16Bits[(word >> 32) & 0xffffu];
1077     bits48to63 = _bitsSetIn16Bits[(word >> 48) & 0xffffu];
1078 }
1079
1080 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1081 public static void GetCommonInnerBorders(BitString left, BitString right, out long from,
    ↪ out long to)
1082 {
1083     from = Math.Max(left._minPositiveWord, right._minPositiveWord);
1084     to = Math.Min(left._maxPositiveWord, right._maxPositiveWord);
1085 }
1086
1087 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1088 public static void GetCommonOuterBorders(BitString left, BitString right, out long from,
    ↪ out long to)
1089 {
1090     from = Math.Min(left._minPositiveWord, right._minPositiveWord);
1091     to = Math.Max(left._maxPositiveWord, right._maxPositiveWord);
1092 }
1093
1094 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1095 public static void GetCommonBorders(BitString left, BitString right, out ulong from, out
    ↪ ulong to)
1096 {
1097     from = (ulong)Math.Max(left._minPositiveWord, right._minPositiveWord);
1098     to = (ulong)Math.Min(left._maxPositiveWord, right._maxPositiveWord);
1099 }
1100
1101 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1102 public static long GetWordsCountFromIndex(long index) => (index + 63) / 64;
1103
1104 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1105 public static long GetWordIndexFromIndex(long index) => index >> 6;
1106
1107 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1108 public static long GetBitMaskFromIndex(long index) => 1L << (int)(index & 63);
1109
1110 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1111 public override int GetHashCode() => base.GetHashCode();
1112
1113 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1114 public override string ToString() => base.ToString();
1115 }
1116 }

```

1.9 ./Platform.Collections/BitStringExtensions.cs

```

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

```

```

18     }
19 }
20 }

```

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
12         ↪ value) ? value : default;
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         public static T PeekOrDefault<T>(this ConcurrentStack<T> stack) => stack.TryPeek(out T
16         ↪ value) ? value : default;
17     }
18 }

```

1.12 ./Platform.Collections/EnsureExtensions.cs

```

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

```

```

30     public static void ArgumentNotEmpty<T>(this EnsureAlwaysExtensionRoot root,
31         ↪ ICollection<T> argument) => ArgumentNotEmpty(root, argument, null, null);
32
33     [MethodImpl(MethodImplOptions.AggressiveInlining)]
34     public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureAlwaysExtensionRoot root,
35         ↪ string argument, string argumentName, string message)
36     {
37         if (string.IsNullOrEmpty(argument))
38         {
39             throw new ArgumentException(message, argumentName);
40         }
41     }
42
43     [MethodImpl(MethodImplOptions.AggressiveInlining)]
44     public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureAlwaysExtensionRoot root,
45         ↪ string argument, string argumentName) => ArgumentNotEmptyAndNotWhiteSpace(root,
46         ↪ argument, argumentName, null);
47
48     [MethodImpl(MethodImplOptions.AggressiveInlining)]
49     public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureAlwaysExtensionRoot root,
50         ↪ string argument) => ArgumentNotEmptyAndNotWhiteSpace(root, argument, null, null);
51
52     #endregion
53
54     #region OnDebug
55
56     [Conditional("DEBUG")]
57     public static void ArgumentNotEmpty<T>(this EnsureOnDebugExtensionRoot root,
58         ↪ ICollection<T> argument, string argumentName, string message) =>
59         ↪ Ensure.Always.ArgumentNotEmpty(argument, argumentName, message);
60
61     [Conditional("DEBUG")]
62     public static void ArgumentNotEmpty<T>(this EnsureOnDebugExtensionRoot root,
63         ↪ ICollection<T> argument, string argumentName) =>
64         ↪ Ensure.Always.ArgumentNotEmpty(argument, argumentName, null);
65
66     [Conditional("DEBUG")]
67     public static void ArgumentNotEmpty<T>(this EnsureOnDebugExtensionRoot root,
68         ↪ ICollection<T> argument) => Ensure.Always.ArgumentNotEmpty(argument, null, null);
69
70     [Conditional("DEBUG")]
71     public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureOnDebugExtensionRoot
72         ↪ root, string argument, string argumentName, string message) =>
73         ↪ Ensure.Always.ArgumentNotEmptyAndNotWhiteSpace(argument, argumentName, message);
74
75     [Conditional("DEBUG")]
76     public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureOnDebugExtensionRoot
77         ↪ root, string argument, string argumentName) =>
78         ↪ Ensure.Always.ArgumentNotEmptyAndNotWhiteSpace(argument, argumentName, null);
79
80     [Conditional("DEBUG")]
81     public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureOnDebugExtensionRoot
82         ↪ root, string argument) => Ensure.Always.ArgumentNotEmptyAndNotWhiteSpace(argument,
83         ↪ null, null);
84
85     #endregion
86 }
87 }

```

1.13 ./Platform.Collections/ICollectionExtensions.cs

```

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

```

```

18         return collection.All(item => equalityComparer.Equals(item, default));
19     }
20 }
21 }

```

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>
13            ↪ dictionary, TKey key)
14        {
15            dictionary.TryGetValue(key, out TValue value);
16            return value;
17        }
18
19        [MethodImpl(MethodImplOptions.AggressiveInlining)]
20        public static TValue GetOrAdd<TKey, TValue>(this IDictionary<TKey, TValue> dictionary,
21            ↪ TKey key, Func<TKey, TValue> valueFactory)
22        {
23            if (!dictionary.TryGetValue(key, out TValue value))
24            {
25                value = valueFactory(key);
26                dictionary.Add(key, value);
27                return value;
28            }
29            return value;
30        }
31    }
32 }

```

1.15 ./Platform.Collections/Lists/CharListExtensions.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 static class CharListExtensions
9     {
10        /// <remarks>
11        /// Based on https://github.com/Microsoft/referencesource/blob/3b1eaf5203992df69de44c783\_a3eda37d3d4cd10/mscorlib/system/string.cs#L833
12        /// </remarks>
13        [MethodImpl(MethodImplOptions.AggressiveInlining)]
14        public static unsafe int GenerateHashCode(this IList<char> list)
15        {
16            var hashSeed = 5381;
17            var hashAccumulator = hashSeed;
18            for (var i = 0; i < list.Count; i++)
19            {
20                hashAccumulator = (hashAccumulator << 5) + hashAccumulator ^ list[i];
21            }
22            return hashAccumulator + (hashSeed * 1566083941);
23        }
24
25        [MethodImpl(MethodImplOptions.AggressiveInlining)]
26        public static bool EqualTo(this IList<char> left, IList<char> right) =>
27            ↪ left.EqualTo(right, ContentEqualTo);
28
29        [MethodImpl(MethodImplOptions.AggressiveInlining)]
30        public static bool ContentEqualTo(this IList<char> left, IList<char> right)
31        {
32            for (var i = left.Count - 1; i >= 0; --i)
33            {
34                if (left[i] != right[i])
35                {
36                    return false;
37                }
38            }
39        }
40    }
41 }

```

```

38         return true;
39     }
40 }
41 }

```

1.16 ./Platform.Collections/Lists/IListComparer.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 IListComparer<T> : IComparer<IList<T>>
9     {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public int Compare(IList<T> left, IList<T> right) => left.CompareTo(right);
12     }
13 }

```

1.17 ./Platform.Collections/Lists/IListEqualityComparer.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 IListEqualityComparer<T> : IEqualityComparer<IList<T>>
9     {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public bool Equals(IList<T> left, IList<T> right) => left.EqualTo(right);
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public int GetHashCode(IList<T> list) => list.GenerateHashCode();
15     }
16 }

```

1.18 ./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         [MethodImpl(MethodImplOptions.AggressiveInlining)]
12         public static bool AddAndReturnTrue<T>(this IList<T> list, T element)
13         {
14             list.Add(element);
15             return true;
16         }
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         public static int GetCountOrZero<T>(this IList<T> list) => list?.Count ?? 0;
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public static bool EqualTo<T>(this IList<T> left, IList<T> right) => EqualTo(left,
23             ↪ right, ContentEqualTo);
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         public static bool EqualTo<T>(this IList<T> left, IList<T> right, Func<IList<T>,
27             ↪ IList<T>, bool> contentEqualityComparer)
28         {
29             if (ReferenceEquals(left, right))
30             {
31                 return true;
32             }
33             var leftCount = left.GetCountOrZero();
34             var rightCount = right.GetCountOrZero();
35             if (leftCount == 0 && rightCount == 0)
36             {
37                 return true;
38             }
39             if (leftCount == 0 || rightCount == 0 || leftCount != rightCount)
40             {

```

```

39         return false;
40     }
41     return contentEqualityComparer(left, right);
42 }
43
44 [MethodImpl(MethodImplOptions.AggressiveInlining)]
45 public static bool ContentEqualTo<T>(this IList<T> left, IList<T> right)
46 {
47     var equalityComparer = EqualityComparer<T>.Default;
48     for (var i = left.Count - 1; i >= 0; --i)
49     {
50         if (!equalityComparer.Equals(left[i], right[i]))
51         {
52             return false;
53         }
54     }
55     return true;
56 }
57
58 [MethodImpl(MethodImplOptions.AggressiveInlining)]
59 public static T[] ToArray<T>(this IList<T> list, Func<T, bool> predicate)
60 {
61     if (list == null)
62     {
63         return null;
64     }
65     var result = new List<T>(list.Count);
66     for (var i = 0; i < list.Count; i++)
67     {
68         if (predicate(list[i]))
69         {
70             result.Add(list[i]);
71         }
72     }
73     return result.ToArray();
74 }
75
76 [MethodImpl(MethodImplOptions.AggressiveInlining)]
77 public static T[] ToArray<T>(this IList<T> list)
78 {
79     var array = new T[list.Count];
80     list.CopyTo(array, 0);
81     return array;
82 }
83
84 [MethodImpl(MethodImplOptions.AggressiveInlining)]
85 public static void ForEach<T>(this IList<T> list, Action<T> action)
86 {
87     for (var i = 0; i < list.Count; i++)
88     {
89         action(list[i]);
90     }
91 }
92
93 /// <remarks>
94 /// Based on http://stackoverflow.com/questions/263400/what-is-the-best-algorithm-for-an
95 /// ↪ -overridden-system-object-gethashcode
96 /// </remarks>
97 [MethodImpl(MethodImplOptions.AggressiveInlining)]
98 public static int GenerateHashCode<T>(this IList<T> list)
99 {
100     var result = 17;
101     for (var i = 0; i < list.Count; i++)
102     {
103         result = unchecked((result * 23) + list[i].GetHashCode());
104     }
105     return result;
106 }
107
108 [MethodImpl(MethodImplOptions.AggressiveInlining)]
109 public static int CompareTo<T>(this IList<T> left, IList<T> right)
110 {
111     var comparer = Comparer<T>.Default;
112     var leftCount = left.GetCountOrZero();
113     var rightCount = right.GetCountOrZero();
114     var intermediateResult = leftCount.CompareTo(rightCount);
115     for (var i = 0; intermediateResult == 0 && i < leftCount; i++)
116     {
117         intermediateResult = comparer.Compare(left[i], right[i]);
118     }
119 }

```



```

117     }
118     return intermediateResult;
119 }
120
121 [MethodImpl(MethodImplOptions.AggressiveInlining)]
122 public static T[] SkipFirst<T>(this IList<T> list) => list.SkipFirst(1);
123
124 [MethodImpl(MethodImplOptions.AggressiveInlining)]
125 public static T[] SkipFirst<T>(this IList<T> list, int skip)
126 {
127     if (list.IsNullOrEmpty() || list.Count <= skip)
128     {
129         return Array.Empty<T>();
130     }
131     var result = new T[list.Count - skip];
132     for (int r = skip, w = 0; r < list.Count; r++, w++)
133     {
134         result[w] = list[r];
135     }
136     return result;
137 }
138
139 [MethodImpl(MethodImplOptions.AggressiveInlining)]
140 public static IList<T> ShiftRight<T>(this IList<T> list) => list.ShiftRight(1);
141
142 [MethodImpl(MethodImplOptions.AggressiveInlining)]
143 public static IList<T> ShiftRight<T>(this IList<T> list, int shift)
144 {
145     var result = new T[list.Count + shift];
146     for (int r = 0, w = shift; r < list.Count; r++, w++)
147     {
148         result[w] = list[r];
149     }
150     return result;
151 }
152 }
153 }

```

1.19 ./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(ICollection<TElement> list)
49     {
50         _list.Add(list[0]);
51         return _returnConstant;
52     }
53
54     [MethodImpl(MethodImplOptions.AggressiveInlining)]
55     public TReturnConstant AddAllValuesAndReturnConstant(ICollection<TElement> list)
56     {
57         for (int i = 1; i < list.Count; i++)
58         {
59             _list.Add(list[i]);
60         }
61         return _returnConstant;
62     }
63 }
64 }

```

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

```

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

```

```

54     }
55     return new string(array, segment.Offset, segment.Length);
56 }
57
58 [MethodImpl(MethodImplOptions.AggressiveInlining)]
59 public override string ToString() => this;
60 }
61 }

```

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

```

1  using System;
2  using System.Collections;
3  using System.Collections.Generic;
4  using System.Runtime.CompilerServices;
5  using Platform.Collections.Lists;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Collections.Segments
10 {
11     public class Segment<T> : IEquatable<Segment<T>>, IList<T>
12     {
13         public IList<T> Base
14         {
15             [MethodImpl(MethodImplOptions.AggressiveInlining)]
16             get;
17         }
18         public int Offset
19         {
20             [MethodImpl(MethodImplOptions.AggressiveInlining)]
21             get;
22         }
23         public int Length
24         {
25             [MethodImpl(MethodImplOptions.AggressiveInlining)]
26             get;
27         }
28
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         public Segment(IList<T> @base, int offset, int length)
31         {
32             Base = @base;
33             Offset = offset;
34             Length = length;
35         }
36
37         [MethodImpl(MethodImplOptions.AggressiveInlining)]
38         public override int GetHashCode() => this.GenerateHashCode();
39
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         public virtual bool Equals(Segment<T> other) => this.EqualTo(other);
42
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         public override bool Equals(object obj) => obj is Segment<T> other ? Equals(other) :
45             ↪ false;
46
47         #region IList
48         public T this[int i]
49         {
50             [MethodImpl(MethodImplOptions.AggressiveInlining)]
51             get => Base[Offset + i];
52             [MethodImpl(MethodImplOptions.AggressiveInlining)]
53             set => Base[Offset + i] = value;
54         }
55
56         public int Count
57         {
58             [MethodImpl(MethodImplOptions.AggressiveInlining)]
59             get => Length;
60         }
61
62         public bool IsReadOnly
63         {
64             [MethodImpl(MethodImplOptions.AggressiveInlining)]
65             get => true;
66         }
67
68         [MethodImpl(MethodImplOptions.AggressiveInlining)]
69         public int IndexOf(T item)
70         {

```

```

71     var index = Base.IndexOf(item);
72     if (index >= Offset)
73     {
74         var actualIndex = index - Offset;
75         if (actualIndex < Length)
76         {
77             return actualIndex;
78         }
79     }
80     return -1;
81 }
82
83 [MethodImpl(MethodImplOptions.AggressiveInlining)]
84 public void Insert(int index, T item) => throw new NotSupportedException();
85
86 [MethodImpl(MethodImplOptions.AggressiveInlining)]
87 public void RemoveAt(int index) => throw new NotSupportedException();
88
89 [MethodImpl(MethodImplOptions.AggressiveInlining)]
90 public void Add(T item) => throw new NotSupportedException();
91
92 [MethodImpl(MethodImplOptions.AggressiveInlining)]
93 public void Clear() => throw new NotSupportedException();
94
95 [MethodImpl(MethodImplOptions.AggressiveInlining)]
96 public bool Contains(T item) => IndexOf(item) >= 0;
97
98 [MethodImpl(MethodImplOptions.AggressiveInlining)]
99 public void CopyTo(T[] array, int arrayIndex)
100 {
101     for (var i = 0; i < Length; i++)
102     {
103         array[arrayIndex++] = this[i];
104     }
105 }
106
107 [MethodImpl(MethodImplOptions.AggressiveInlining)]
108 public bool Remove(T item) => throw new NotSupportedException();
109
110 [MethodImpl(MethodImplOptions.AggressiveInlining)]
111 public IEnumerator<T> GetEnumerator()
112 {
113     for (var i = 0; i < Length; i++)
114     {
115         yield return this[i];
116     }
117 }
118
119 [MethodImpl(MethodImplOptions.AggressiveInlining)]
120 IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
121
122 #endregion
123 }
124 }

```

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  using System.Runtime.CompilerServices;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Collections.Segments.Walkers
7  {
8      public abstract class AllSegmentsWalkerBase<T, TSegment> : AllSegmentsWalkerBase
9          where TSegment : Segment<T>
10     {
11         private readonly int _minimumStringSegmentLength;
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

14     protected AllSegmentsWalkerBase(int minimumStringSegmentLength) =>
15         ↪ _minimumStringSegmentLength = minimumStringSegmentLength;
16
17     [MethodImpl(MethodImplOptions.AggressiveInlining)]
18     protected AllSegmentsWalkerBase() : this(DefaultMinimumStringSegmentLength) { }
19
20     [MethodImpl(MethodImplOptions.AggressiveInlining)]
21     public virtual void WalkAll(ICollection<T> elements)
22     {
23         for (int offset = 0, maxOffset = elements.Count - _minimumStringSegmentLength;
24             ↪ offset <= maxOffset; offset++)
25         {
26             for (int length = _minimumStringSegmentLength, maxLength = elements.Count -
27                 ↪ offset; length <= maxLength; length++)
28             {
29                 Iteration(CreateSegment(elements, offset, length));
30             }
31         }
32
33     [MethodImpl(MethodImplOptions.AggressiveInlining)]
34     protected abstract TSegment CreateSegment(ICollection<T> elements, int offset, int length);
35
36     [MethodImpl(MethodImplOptions.AggressiveInlining)]
37     protected abstract void Iteration(TSegment segment);
38 }

```

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

```

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

```

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

```

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

```

1.26 ./Platform.Collections/Segments/Walkers/DictionaryBasedDuplicateSegmentsWalkerBase[T, Segment].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.Segments.Walkers
8  {
9      public abstract class DictionaryBasedDuplicateSegmentsWalkerBase<T, TSegment> :
10         ↪ DuplicateSegmentsWalkerBase<T, TSegment>
11         where TSegment : Segment<T>
12     {
13         public static readonly bool DefaultResetDictionaryOnEachWalk;
14     }
15 }

```

```

14 private readonly bool _resetDictionaryOnEachWalk;
15 protected IDictionary<TSegment, long> Dictionary;
16
17 [MethodImpl(MethodImplOptions.AggressiveInlining)]
18 protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<TSegment, long>
19     ↪ dictionary, int minimumStringSegmentLength, bool resetDictionaryOnEachWalk)
20     : base(minimumStringSegmentLength)
21 {
22     Dictionary = dictionary;
23     _resetDictionaryOnEachWalk = resetDictionaryOnEachWalk;
24 }
25
26 [MethodImpl(MethodImplOptions.AggressiveInlining)]
27 protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<TSegment, long>
28     ↪ dictionary, int minimumStringSegmentLength) : this(dictionary,
29     ↪ minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
30
31 [MethodImpl(MethodImplOptions.AggressiveInlining)]
32 protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<TSegment, long>
33     ↪ dictionary) : this(dictionary, DefaultMinimumStringSegmentLength,
34     ↪ DefaultResetDictionaryOnEachWalk) { }
35
36 [MethodImpl(MethodImplOptions.AggressiveInlining)]
37 protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength,
38     ↪ bool resetDictionaryOnEachWalk) : this(resetDictionaryOnEachWalk ? null : new
39     ↪ Dictionary<TSegment, long>(), minimumStringSegmentLength, resetDictionaryOnEachWalk)
40     : { }
41
42 [MethodImpl(MethodImplOptions.AggressiveInlining)]
43 protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength) :
44     ↪ this(minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
45
46 [MethodImpl(MethodImplOptions.AggressiveInlining)]
47 protected DictionaryBasedDuplicateSegmentsWalkerBase() :
48     ↪ this(DefaultMinimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
49
50 [MethodImpl(MethodImplOptions.AggressiveInlining)]
51 public override void WalkAll(ICollection<T> elements)
52 {
53     if (_resetDictionaryOnEachWalk)
54     {
55         var capacity = Math.Ceiling(Math.Pow(elements.Count, 2) / 2);
56         Dictionary = new Dictionary<TSegment, long>((int)capacity);
57     }
58     base.WalkAll(elements);
59 }
60
61 [MethodImpl(MethodImplOptions.AggressiveInlining)]
62 protected override long GetSegmentFrequency(TSegment segment) =>
63     ↪ Dictionary.GetOrDefault(segment);
64
65 [MethodImpl(MethodImplOptions.AggressiveInlining)]
66 protected override void SetSegmentFrequency(TSegment segment, long frequency) =>
67     ↪ Dictionary[segment] = frequency;
68 }
69 }

```

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

```

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

```

```

16     [MethodImpl(MethodImplOptions.AggressiveInlining)]
17     protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<Segment<T>, long>
    ↪ dictionary) : base(dictionary, DefaultMinimumStringSegmentLength,
    ↪ DefaultResetDictionaryOnEachWalk) { }

18
19     [MethodImpl(MethodImplOptions.AggressiveInlining)]
20     protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength,
    ↪ bool resetDictionaryOnEachWalk) : base(minimumStringSegmentLength,
    ↪ resetDictionaryOnEachWalk) { }

21
22     [MethodImpl(MethodImplOptions.AggressiveInlining)]
23     protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength) :
    ↪ base(minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }

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

27 }
28 }

```

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

```

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

```

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,
    ↪ Segment<T>>
6     {
7     }
8 }

```

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

```

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

```

```

7 {
8     public static class ISetExtensions
9     {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public static void AddAndReturnVoid<T>(this ISet<T> set, T element) => set.Add(element);
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public static void RemoveAndReturnVoid<T>(this ISet<T> set, T element) =>
15             ↪ set.Remove(element);
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public static bool DoNotContains<T>(this ISet<T> set, T element) =>
19             ↪ !set.Contains(element);
20     }
21 }

```

1.31 ./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(IList<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(IList<TElement> list)
49         {
50             _set.Add(list[0]);
51             return _returnConstant;
52         }
53     }
54 }

```

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

```

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

```



```

8     public class DefaultStack<TElement> : Stack<TElement>, IStack<TElement>
9     {
10         public bool IsEmpty
11         {
12             [MethodImpl(MethodImplOptions.AggressiveInlining)]
13             get => Count <= 0;
14         }
15     }
16 }

```

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

```

1 using System.Runtime.CompilerServices;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Collections.Stacks
6 {
7     public interface IStack<TElement>
8     {
9         bool IsEmpty
10         {
11             [MethodImpl(MethodImplOptions.AggressiveInlining)]
12             get;
13         }
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         void Push(TElement element);
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         TElement Pop();
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         TElement Peek();
23     }
24 }

```

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

```

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

```

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

```

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

```

1.36 ./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() :
            ↪ default;
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public static T PeekOrDefault<T>(this Stack<T> stack) => stack.Count > 0 ? stack.Peek()
            ↪ : default;
15     }
16 }
```

1.37 ./Platform.Collections/StringExtensions.cs

```
1 using System;
2 using System.Globalization;
3 using System.Runtime.CompilerServices;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Collections
8 {
9     public static class StringExtensions
10    {
11        [MethodImpl(MethodImplOptions.AggressiveInlining)]
12        public static string CapitalizeFirstLetter(this string @string)
13        {
14            if (string.IsNullOrEmpty(@string))
15            {
16                return @string;
17            }
18            var chars = @string.ToCharArray();
19            for (var i = 0; i < chars.Length; i++)
20            {
21                var category = char.GetUnicodeCategory(chars[i]);
22                if (category == UnicodeCategory.UppercaseLetter)
23                {
24                    return @string;
25                }
26                if (category == UnicodeCategory.LowercaseLetter)
27                {
28                    chars[i] = char.ToUpper(chars[i]);
29                    return new string(chars);
30                }
31            }
32            return @string;
33        }
34
35        [MethodImpl(MethodImplOptions.AggressiveInlining)]
36        public static string Truncate(this string @string, int maxLength) =>
            ↪ string.IsNullOrEmpty(@string) ? @string : @string.Substring(0,
            ↪ Math.Min(@string.Length, maxLength));
37
38        [MethodImpl(MethodImplOptions.AggressiveInlining)]
39        public static string TrimSingle(this string @string, char charToTrim)
40        {
41            if (!string.IsNullOrEmpty(@string))
42            {
43                if (@string.Length == 1)
44                {
45                    if (@string[0] == charToTrim)
46                    {
47                        return "";
48                    }
49                    else
50                    {
51                        return @string;
52                    }
53                }
54                else
55                {
56                    var left = 0;
```

```

57         var right = @string.Length - 1;
58         if (@string[left] == charToTrim)
59         {
60             left++;
61         }
62         if (@string[right] == charToTrim)
63         {
64             right--;
65         }
66         return @string.Substring(left, right - left + 1);
67     }
68 }
69 else
70 {
71     return @string;
72 }
73 }
74 }
75 }

```

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

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  // ReSharper disable ForCanBeConvertedToForeach
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Collections.Trees
8  {
9      public class Node
10     {
11         private Dictionary<object, Node> _childNodes;
12
13         public object Value
14         {
15             [MethodImpl(MethodImplOptions.AggressiveInlining)]
16             get;
17             [MethodImpl(MethodImplOptions.AggressiveInlining)]
18             set;
19         }
20
21         public Dictionary<object, Node> ChildNodes
22         {
23             [MethodImpl(MethodImplOptions.AggressiveInlining)]
24             get => _childNodes ?? (_childNodes = new Dictionary<object, Node>());
25         }
26
27         public Node this[object key]
28         {
29             [MethodImpl(MethodImplOptions.AggressiveInlining)]
30             get
31             {
32                 var child = GetChild(key);
33                 if (child == null)
34                 {
35                     child = AddChild(key);
36                 }
37                 return child;
38             }
39             [MethodImpl(MethodImplOptions.AggressiveInlining)]
40             set => SetChildValue(value, key);
41         }
42
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         public Node(object value) => Value = value;
45
46         [MethodImpl(MethodImplOptions.AggressiveInlining)]
47         public Node() : this(null) { }
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         public bool ContainsChild(params object[] keys) => GetChild(keys) != null;
51
52         [MethodImpl(MethodImplOptions.AggressiveInlining)]
53         public Node GetChild(params object[] keys)
54         {
55             var node = this;
56             for (var i = 0; i < keys.Length; i++)
57             {
58                 node.ChildNodes.TryGetValue(keys[i], out node);
59                 if (node == null)

```

```

60         {
61             return null;
62         }
63     }
64     return node;
65 }
66
67 [MethodImpl(MethodImplOptions.AggressiveInlining)]
68 public object GetChildValue(params object[] keys) => GetChild(keys)?.Value;
69
70 [MethodImpl(MethodImplOptions.AggressiveInlining)]
71 public Node AddChild(object key) => AddChild(key, new Node(null));
72
73 [MethodImpl(MethodImplOptions.AggressiveInlining)]
74 public Node AddChild(object key, object value) => AddChild(key, new Node(value));
75
76 [MethodImpl(MethodImplOptions.AggressiveInlining)]
77 public Node AddChild(object key, Node child)
78 {
79     ChildNodes.Add(key, child);
80     return child;
81 }
82
83 [MethodImpl(MethodImplOptions.AggressiveInlining)]
84 public Node SetChild(params object[] keys) => SetChildValue(null, keys);
85
86 [MethodImpl(MethodImplOptions.AggressiveInlining)]
87 public Node SetChild(object key) => SetChildValue(null, key);
88
89 [MethodImpl(MethodImplOptions.AggressiveInlining)]
90 public Node SetChildValue(object value, params object[] keys)
91 {
92     var node = this;
93     for (var i = 0; i < keys.Length; i++)
94     {
95         node = SetChildValue(value, keys[i]);
96     }
97     node.Value = value;
98     return node;
99 }
100
101 [MethodImpl(MethodImplOptions.AggressiveInlining)]
102 public Node SetChildValue(object value, object key)
103 {
104     if (!ChildNodes.TryGetValue(key, out Node child))
105     {
106         child = AddChild(key, value);
107     }
108     child.Value = value;
109     return child;
110 }
111 }
112 }

```

1.39 ./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 }

```

1.40 ./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 }

```

1.41 ./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, 36
- ./Platform.Collections.Tests/CharsSegmentTests.cs, 38
- ./Platform.Collections.Tests/StringTests.cs, 39
- ./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, 5
- ./Platform.Collections/BitStringExtensions.cs, 19
- ./Platform.Collections/Concurrent/ConcurrentQueueExtensions.cs, 20
- ./Platform.Collections/Concurrent/ConcurrentStackExtensions.cs, 20
- ./Platform.Collections/EnsureExtensions.cs, 20
- ./Platform.Collections/ICollectionExtensions.cs, 21
- ./Platform.Collections/IDictionaryExtensions.cs, 22
- ./Platform.Collections/Lists/CharListExtensions.cs, 22
- ./Platform.Collections/Lists/IListComparer.cs, 23
- ./Platform.Collections/Lists/IListEqualityComparer.cs, 23
- ./Platform.Collections/Lists/IListExtensions.cs, 23
- ./Platform.Collections/Lists/ListFiller.cs, 25
- ./Platform.Collections/Segments/CharSegment.cs, 26
- ./Platform.Collections/Segments/Segment.cs, 27
- ./Platform.Collections/Segments/Walkers/AllSegmentsWalkerBase.cs, 28
- ./Platform.Collections/Segments/Walkers/AllSegmentsWalkerBase[T, TSegment].cs, 28
- ./Platform.Collections/Segments/Walkers/AllSegmentsWalkerBase[T].cs, 29
- ./Platform.Collections/Segments/Walkers/AllSegmentsWalkerExtensions.cs, 29
- ./Platform.Collections/Segments/Walkers/DictionaryBasedDuplicateSegmentsWalkerBase[T, Segment].cs, 29
- ./Platform.Collections/Segments/Walkers/DictionaryBasedDuplicateSegmentsWalkerBase[T].cs, 30
- ./Platform.Collections/Segments/Walkers/DuplicateSegmentsWalkerBase[T, TSegment].cs, 31
- ./Platform.Collections/Segments/Walkers/DuplicateSegmentsWalkerBase[T].cs, 31
- ./Platform.Collections/Sets/ISetExtensions.cs, 31
- ./Platform.Collections/Sets/SetFiller.cs, 32
- ./Platform.Collections/Stacks/DefaultStack.cs, 32
- ./Platform.Collections/Stacks/IStack.cs, 33
- ./Platform.Collections/Stacks/IStackExtensions.cs, 33
- ./Platform.Collections/Stacks/IStackFactory.cs, 33
- ./Platform.Collections/Stacks/StackExtensions.cs, 33
- ./Platform.Collections/StringExtensions.cs, 34
- ./Platform.Collections/Trees/Node.cs, 35