

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

```

55     [MethodImpl(MethodImplOptions.AggressiveInlining)]
56     public virtual void Free(T[] array)
57     {
58         if (array.IsNullOrEmpty())
59         {
60             return;
61         }
62         var stack = _pool.GetOrAdd(array.LongLength, size => new
        ↪ Stack<T[]>(_maxArraysPerSize));
63         if (stack.Count == _maxArraysPerSize) // Stack is full
64         {
65             return;
66         }
67         stack.Push(array);
68     }
69 }
70 }

```

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         ↪ a3eda37d3d4cd10/mscorlib/system/string.cs#L833
12         /// </remarks>
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public static int GenerateHashCode(this char[] array, int offset, int length)
15         {
16             var hashSeed = 5381;
17             var hashAccumulator = hashSeed;
18             fixed (char* pointer = &array[offset])
19             {
20                 for (char* s = pointer, last = s + length; s < last; s++)
21                 {
22                     hashAccumulator = (hashAccumulator << 5) + hashAccumulator ^ *s;
23                 }
24             }
25             return hashAccumulator + (hashSeed * 1566083941);
26         }
27
28         /// <remarks>
29         /// Based on https://github.com/Microsoft/referencesource/blob/3b1eaf5203992df69de44c783\_a3eda37d3d4cd10/mscorlib/system/string.cs#L364
30         ↪ a3eda37d3d4cd10/mscorlib/system/string.cs#L364
31         /// </remarks>
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         public static bool ContentEqualTo(this char[] left, int leftOffset, int length, char[]
34         ↪ right, int rightOffset)
35         {
36             fixed (char* leftPointer = &left[leftOffset])
37             {
38                 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
49 [MethodImpl(MethodImplOptions.AggressiveInlining)]
50 private static bool CheckArraysMainPartForEquality(ref char* left, ref char* right, ref
51     ↪ int length)
52 {
53     while (length >= 10)
54     {
55         if ((* (int*)left != * (int*)right)
56             || (* (int*)(left + 2) != * (int*)(right + 2))
57             || (* (int*)(left + 4) != * (int*)(right + 4))
58             || (* (int*)(left + 6) != * (int*)(right + 6))
59             || (* (int*)(left + 8) != * (int*)(right + 8)))
60         {
61             return false;
62         }
63         left += 10;
64         right += 10;
65         length -= 10;
66     }
67     return true;
68 }
69
70 [MethodImpl(MethodImplOptions.AggressiveInlining)]
71 private static void CheckArraysRemainderForEquality(ref char* left, ref char* right, ref
72     ↪ int length)
73 {
74     // This depends on the fact that the String objects are
75     // always zero terminated and that the terminating zero is not included
76     // in the length. For odd string sizes, the last compare will include
77     // the zero terminator.
78     while (length > 0)
79     {
80         if ((* (int*)left != * (int*)right)
81             {
82                 break;
83             }
84         left += 2;
85         right += 2;
86         length -= 2;
87     }
88 }
89 }

```

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                         Array.Clear(_array, (int)oldWords, (int)(words - oldWords));
68                     }
69                     // 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
144 #endregion
145

```

```

146 [MethodImpl(MethodImplOptions.AggressiveInlining)]
147 public BitString Not()
148 {
149     for (var i = 0L; i < _array.LongLength; i++)
150     {
151         _array[i] = ~_array[i];
152         RefreshBordersByWord(i);
153     }
154     return this;
155 }
156
157 [MethodImpl(MethodImplOptions.AggressiveInlining)]
158 public BitString ParallelNot()
159 {
160     var threads = Environment.ProcessorCount / 2;
161     if (threads <= 1)
162     {
163         return Not();
164     }
165     var partitioner = Partitioner.Create(0L, _array.LongLength, _array.LongLength /
166 ↪ threads);
167     Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions() {
168 ↪ MaxDegreeOfParallelism = threads }, range =>
169     {
170         var maximum = range.Item2;
171         for (var i = range.Item1; i < maximum; i++)
172         {
173             _array[i] = ~_array[i];
174         }
175     });
176     MarkBordersAsAllBitsSet();
177     TryShrinkBorders();
178     return this;
179 }
180
181 [MethodImpl(MethodImplOptions.AggressiveInlining)]
182 public BitString VectorNot()
183 {
184     if (!Vector.IsHardwareAccelerated || _array.LongLength >= int.MaxValue)
185     {
186         return Not();
187     }
188     var step = Vector<long>.Count;
189     if (_array.Length < step)
190     {
191         return Not();
192     }
193     VectorNotLoop(_array, step, 0, _array.Length);
194     MarkBordersAsAllBitsSet();
195     TryShrinkBorders();
196     return this;
197 }
198
199 [MethodImpl(MethodImplOptions.AggressiveInlining)]
200 public BitString ParallelVectorNot()
201 {
202     var threads = Environment.ProcessorCount / 2;
203     if (threads <= 1)
204     {
205         return VectorNot();
206     }
207     if (!Vector.IsHardwareAccelerated)
208     {
209         return ParallelNot();
210     }
211     var step = Vector<long>.Count;
212     if (_array.Length < (step * threads))
213     {
214         return VectorNot();
215     }
216     var partitioner = Partitioner.Create(0, _array.Length, _array.Length / threads);
217     Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions() {
218 ↪ MaxDegreeOfParallelism = threads }, range => VectorNotLoop(_array, step,
219 ↪ range.Item1, range.Item2));
220     MarkBordersAsAllBitsSet();
221     TryShrinkBorders();
222     return this;
223 }

```

```

220 [MethodImpl(MethodImplOptions.AggressiveInlining)]
221 static private void VectorNotLoop(long[] array, int step, int start, int maximum)
222 {
223     var i = start;
224     var range = maximum - start - 1;
225     var stop = range - (range % step);
226     for (; i < stop; i += step)
227     {
228         (~new Vector<long>(array, i)).CopyTo(array, i);
229     }
230     for (; i < maximum; i++)
231     {
232         array[i] = ~array[i];
233     }
234 }
235
236 [MethodImpl(MethodImplOptions.AggressiveInlining)]
237 public BitString And(BitString other)
238 {
239     EnsureBitStringHasTheSameSize(other, nameof(other));
240     GetCommonOuterBorders(this, other, out long from, out long to);
241     var otherArray = other._array;
242     for (var i = from; i <= to; i++)
243     {
244         _array[i] &= otherArray[i];
245         RefreshBordersByWord(i);
246     }
247     return this;
248 }
249
250 [MethodImpl(MethodImplOptions.AggressiveInlining)]
251 public BitString ParallelAnd(BitString other)
252 {
253     var threads = Environment.ProcessorCount / 2;
254     if (threads <= 1)
255     {
256         return And(other);
257     }
258     EnsureBitStringHasTheSameSize(other, nameof(other));
259     GetCommonOuterBorders(this, other, out long from, out long to);
260     var partitioner = Partitioner.Create(from, to + 1, (to - from) / threads);
261     Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions() {
262         ↪ MaxDegreeOfParallelism = threads }, range =>
263     {
264         var maximum = range.Item2;
265         for (var i = range.Item1; i < maximum; i++)
266         {
267             _array[i] &= other._array[i];
268         }
269     });
270     MarkBordersAsAllBitsSet();
271     TryShrinkBorders();
272     return this;
273 }
274
275 [MethodImpl(MethodImplOptions.AggressiveInlining)]
276 public BitString VectorAnd(BitString other)
277 {
278     if (!Vector.IsHardwareAccelerated || _array.LongLength >= int.MaxValue)
279     {
280         return And(other);
281     }
282     var step = Vector<long>.Count;
283     if (_array.Length < step)
284     {
285         return And(other);
286     }
287     EnsureBitStringHasTheSameSize(other, nameof(other));
288     GetCommonOuterBorders(this, other, out int from, out int to);
289     VectorAndLoop(_array, other._array, step, from, to + 1);
290     MarkBordersAsAllBitsSet();
291     TryShrinkBorders();
292     return this;
293 }
294
295 [MethodImpl(MethodImplOptions.AggressiveInlining)]
296 public BitString ParallelVectorAnd(BitString other)

```



```

297 {
298     var threads = Environment.ProcessorCount / 2;
299     if (threads <= 1)
300     {
301         return VectorAnd(other);
302     }
303     if (!Vector.IsHardwareAccelerated)
304     {
305         return ParallelAnd(other);
306     }
307     var step = Vector<long>.Count;
308     if (_array.Length < (step * threads))
309     {
310         return VectorAnd(other);
311     }
312     EnsureBitStringHasTheSameSize(other, nameof(other));
313     GetCommonOuterBorders(this, other, out int from, out int to);
314     var partitioner = Partitioner.Create(from, to + 1, (to - from) / threads);
315     Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions() {
        ↪ MaxDegreeOfParallelism = threads }, range => VectorAndLoop(_array, other._array,
        ↪ step, range.Item1, range.Item2));
316     MarkBordersAsAllBitsSet();
317     TryShrinkBorders();
318     return this;
319 }
320
321 [MethodImpl(MethodImplOptions.AggressiveInlining)]
322 static private void VectorAndLoop(long[] array, long[] otherArray, int step, int start,
    ↪ int maximum)
323 {
324     var i = start;
325     var range = maximum - start - 1;
326     var stop = range - (range % step);
327     for (; i < stop; i += step)
328     {
329         (new Vector<long>(array, i) & new Vector<long>(otherArray, i)).CopyTo(array, i);
330     }
331     for (; i < maximum; i++)
332     {
333         array[i] &= otherArray[i];
334     }
335 }
336
337 [MethodImpl(MethodImplOptions.AggressiveInlining)]
338 public BitString Or(BitString other)
339 {
340     EnsureBitStringHasTheSameSize(other, nameof(other));
341     GetCommonOuterBorders(this, other, out long from, out long to);
342     for (var i = from; i <= to; i++)
343     {
344         _array[i] |= other._array[i];
345         RefreshBordersByWord(i);
346     }
347     return this;
348 }
349
350 [MethodImpl(MethodImplOptions.AggressiveInlining)]
351 public BitString ParallelOr(BitString other)
352 {
353     var threads = Environment.ProcessorCount / 2;
354     if (threads <= 1)
355     {
356         return Or(other);
357     }
358     EnsureBitStringHasTheSameSize(other, nameof(other));
359     GetCommonOuterBorders(this, other, out long from, out long to);
360     var partitioner = Partitioner.Create(from, to + 1, (to - from) / threads);
361     Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions() {
        ↪ MaxDegreeOfParallelism = threads }, range =>
362     {
363         var maximum = range.Item2;
364         for (var i = range.Item1; i < maximum; i++)
365         {
366             _array[i] |= other._array[i];
367         }
368     });
369     MarkBordersAsAllBitsSet();
370     TryShrinkBorders();

```

```

371     return this;
372 }
373
374 [MethodImpl(MethodImplOptions.AggressiveInlining)]
375 public BitString VectorOr(BitString other)
376 {
377     if (!Vector.IsHardwareAccelerated || _array.LongLength >= int.MaxValue)
378     {
379         return Or(other);
380     }
381     var step = Vector<long>.Count;
382     if (_array.Length < step)
383     {
384         return Or(other);
385     }
386     EnsureBitStringHasTheSameSize(other, nameof(other));
387     GetCommonOuterBorders(this, other, out int from, out int to);
388     VectorOrLoop(_array, other._array, step, from, to + 1);
389     MarkBordersAsAllBitsSet();
390     TryShrinkBorders();
391     return this;
392 }
393
394 [MethodImpl(MethodImplOptions.AggressiveInlining)]
395 public BitString ParallelVectorOr(BitString other)
396 {
397     var threads = Environment.ProcessorCount / 2;
398     if (threads <= 1)
399     {
400         return VectorOr(other);
401     }
402     if (!Vector.IsHardwareAccelerated)
403     {
404         return ParallelOr(other);
405     }
406     var step = Vector<long>.Count;
407     if (_array.Length < (step * threads))
408     {
409         return VectorOr(other);
410     }
411     EnsureBitStringHasTheSameSize(other, nameof(other));
412     GetCommonOuterBorders(this, other, out int from, out int to);
413     var partitioner = Partitioner.Create(from, to + 1, (to - from) / threads);
414     Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions() {
415         ↪ MaxDegreeOfParallelism = threads }, range => VectorOrLoop(_array, other._array,
416         ↪ step, range.Item1, range.Item2));
417     MarkBordersAsAllBitsSet();
418     TryShrinkBorders();
419     return this;
420 }
421
422 [MethodImpl(MethodImplOptions.AggressiveInlining)]
423 static private void VectorOrLoop(long[] array, long[] otherArray, int step, int start,
424     ↪ int maximum)
425 {
426     var i = start;
427     var range = maximum - start - 1;
428     var stop = range - (range % step);
429     for (; i < stop; i += step)
430     {
431         (new Vector<long>(array, i) | new Vector<long>(otherArray, i)).CopyTo(array, i);
432     }
433     for (; i < maximum; i++)
434     {
435         array[i] |= otherArray[i];
436     }
437 }
438
439 [MethodImpl(MethodImplOptions.AggressiveInlining)]
440 public BitString Xor(BitString other)
441 {
442     EnsureBitStringHasTheSameSize(other, nameof(other));
443     GetCommonOuterBorders(this, other, out long from, out long to);
444     for (var i = from; i <= to; i++)
445     {
446         _array[i] ^= other._array[i];
447         RefreshBordersByWord(i);
448     }

```

```

446         return this;
447     }
448
449     [MethodImpl(MethodImplOptions.AggressiveInlining)]
450     public BitString ParallelXor(BitString other)
451     {
452         var threads = Environment.ProcessorCount / 2;
453         if (threads <= 1)
454         {
455             return Xor(other);
456         }
457         EnsureBitStringHasTheSameSize(other, nameof(other));
458         GetCommonOuterBorders(this, other, out long from, out long to);
459         var partitioner = Partitioner.Create(from, to + 1, (to - from) / threads);
460         Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions() {
461             ↪ MaxDegreeOfParallelism = threads }, range =>
462         {
463             var maximum = range.Item2;
464             for (var i = range.Item1; i < maximum; i++)
465             {
466                 _array[i] ^= other._array[i];
467             }
468         });
469         MarkBordersAsAllBitsSet();
470         TryShrinkBorders();
471         return this;
472     }
473
474     [MethodImpl(MethodImplOptions.AggressiveInlining)]
475     public BitString VectorXor(BitString other)
476     {
477         if (!Vector.IsHardwareAccelerated || _array.LongLength >= int.MaxValue)
478         {
479             return Xor(other);
480         }
481         var step = Vector<long>.Count;
482         if (_array.Length < step)
483         {
484             return Xor(other);
485         }
486         EnsureBitStringHasTheSameSize(other, nameof(other));
487         GetCommonOuterBorders(this, other, out int from, out int to);
488         VectorXorLoop(_array, other._array, step, from, to + 1);
489         MarkBordersAsAllBitsSet();
490         TryShrinkBorders();
491         return this;
492     }
493
494     [MethodImpl(MethodImplOptions.AggressiveInlining)]
495     public BitString ParallelVectorXor(BitString other)
496     {
497         var threads = Environment.ProcessorCount / 2;
498         if (threads <= 1)
499         {
500             return VectorXor(other);
501         }
502         if (!Vector.IsHardwareAccelerated)
503         {
504             return ParallelXor(other);
505         }
506         var step = Vector<long>.Count;
507         if (_array.Length < (step * threads))
508         {
509             return VectorXor(other);
510         }
511         EnsureBitStringHasTheSameSize(other, nameof(other));
512         GetCommonOuterBorders(this, other, out int from, out int to);
513         var partitioner = Partitioner.Create(from, to + 1, (to - from) / threads);
514         Parallel.ForEach(partitioner.GetDynamicPartitions(), new ParallelOptions() {
515             ↪ MaxDegreeOfParallelism = threads }, range => VectorXorLoop(_array, other._array,
516             ↪ step, range.Item1, range.Item2));
517         MarkBordersAsAllBitsSet();
518         TryShrinkBorders();
519         return this;
520     }
521
522     [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

520 static private void VectorXorLoop(long[] array, long[] otherArray, int step, int start,
521     ↪ int maximum)
522 {
523     var i = start;
524     var range = maximum - start - 1;
525     var stop = range - (range % step);
526     for (; i < stop; i += step)
527     {
528         (new Vector<long>(array, i) ^ new Vector<long>(otherArray, i)).CopyTo(array, i);
529     }
530     for (; i < maximum; i++)
531     {
532         array[i] ^= otherArray[i];
533     }
534 }
535 [MethodImpl(MethodImplOptions.AggressiveInlining)]
536 private void RefreshBordersByWord(long wordIndex)
537 {
538     if (_array[wordIndex] == 0)
539     {
540         if (wordIndex == _minPositiveWord && wordIndex != _array.LongLength - 1)
541         {
542             _minPositiveWord++;
543         }
544         if (wordIndex == _maxPositiveWord && wordIndex != 0)
545         {
546             _maxPositiveWord--;
547         }
548     }
549     else
550     {
551         if (wordIndex < _minPositiveWord)
552         {
553             _minPositiveWord = wordIndex;
554         }
555         if (wordIndex > _maxPositiveWord)
556         {
557             _maxPositiveWord = wordIndex;
558         }
559     }
560 }
561 [MethodImpl(MethodImplOptions.AggressiveInlining)]
562 public bool TryShrinkBorders()
563 {
564     GetBorders(out long from, out long to);
565     while (from <= to && _array[from] == 0)
566     {
567         from++;
568     }
569     if (from > to)
570     {
571         MarkBordersAsAllBitsReset();
572         return true;
573     }
574     while (to >= from && _array[to] == 0)
575     {
576         to--;
577     }
578     if (to < from)
579     {
580         MarkBordersAsAllBitsReset();
581         return true;
582     }
583     var bordersUpdated = from != _minPositiveWord || to != _maxPositiveWord;
584     if (bordersUpdated)
585     {
586         SetBorders(from, to);
587     }
588     return bordersUpdated;
589 }
590 [MethodImpl(MethodImplOptions.AggressiveInlining)]
591 public bool Get(long index)
592 {
593     Ensure.Always.ArgumentInRange(index, GetValidIndexRange(), nameof(index));
594     return (_array[GetWordIndexFromIndex(index)] & GetBitMaskFromIndex(index)) != 0;
595 }
596 }
597

```

```

598 [MethodImpl(MethodImplOptions.AggressiveInlining)]
599 public void Set(long index, bool value)
600 {
601     if (value)
602     {
603         Set(index);
604     }
605     else
606     {
607         Reset(index);
608     }
609 }
610
611 [MethodImpl(MethodImplOptions.AggressiveInlining)]
612 public void Set(long index)
613 {
614     Ensure.Always.ArgumentInRange(index, GetValidIndexRange(), nameof(index));
615     var wordIndex = GetWordIndexFromIndex(index);
616     var mask = GetBitMaskFromIndex(index);
617     _array[wordIndex] |= mask;
618     RefreshBordersByWord(wordIndex);
619 }
620
621 [MethodImpl(MethodImplOptions.AggressiveInlining)]
622 public void Reset(long index)
623 {
624     Ensure.Always.ArgumentInRange(index, GetValidIndexRange(), nameof(index));
625     var wordIndex = GetWordIndexFromIndex(index);
626     var mask = GetBitMaskFromIndex(index);
627     _array[wordIndex] &= ~mask;
628     RefreshBordersByWord(wordIndex);
629 }
630
631 [MethodImpl(MethodImplOptions.AggressiveInlining)]
632 public bool Add(long index)
633 {
634     var wordIndex = GetWordIndexFromIndex(index);
635     var mask = GetBitMaskFromIndex(index);
636     if ((_array[wordIndex] & mask) == 0)
637     {
638         _array[wordIndex] |= mask;
639         RefreshBordersByWord(wordIndex);
640         return true;
641     }
642     else
643     {
644         return false;
645     }
646 }
647
648 [MethodImpl(MethodImplOptions.AggressiveInlining)]
649 public void SetAll(bool value)
650 {
651     if (value)
652     {
653         SetAll();
654     }
655     else
656     {
657         ResetAll();
658     }
659 }
660
661 [MethodImpl(MethodImplOptions.AggressiveInlining)]
662 public void SetAll()
663 {
664     const long fillValue = unchecked((long)0xffffffffffffffff);
665     var words = GetWordsCountFromIndex(_length);
666     for (var i = 0; i < words; i++)
667     {
668         _array[i] = fillValue;
669     }
670     MarkBordersAsAllBitsSet();
671 }
672
673 [MethodImpl(MethodImplOptions.AggressiveInlining)]
674 public void ResetAll()
675 {
676

```

```

677     const long fillValue = 0;
678     GetBorders(out long from, out long to);
679     for (var i = from; i <= to; i++)
680     {
681         _array[i] = fillValue;
682     }
683     MarkBordersAsAllBitsReset();
684 }
685
686 [MethodImpl(MethodImplOptions.AggressiveInlining)]
687 public List<long> GetSetIndices()
688 {
689     var result = new List<long>();
690     GetBorders(out long from, out long to);
691     for (var i = from; i <= to; i++)
692     {
693         var word = _array[i];
694         if (word != 0)
695         {
696             AppendAllSetBitIndices(result, i, word);
697         }
698     }
699     return result;
700 }
701
702 [MethodImpl(MethodImplOptions.AggressiveInlining)]
703 public List<ulong> GetSetUInt64Indices()
704 {
705     var result = new List<ulong>();
706     GetBorders(out ulong from, out ulong to);
707     for (var i = from; i <= to; i++)
708     {
709         var word = _array[i];
710         if (word != 0)
711         {
712             AppendAllSetBitIndices(result, i, word);
713         }
714     }
715     return result;
716 }
717
718 [MethodImpl(MethodImplOptions.AggressiveInlining)]
719 public long GetFirstSetBitIndex()
720 {
721     var i = _minPositiveWord;
722     var word = _array[i];
723     if (word != 0)
724     {
725         return GetFirstSetBitForWord(i, word);
726     }
727     return -1;
728 }
729
730 [MethodImpl(MethodImplOptions.AggressiveInlining)]
731 public long GetLastSetBitIndex()
732 {
733     var i = _maxPositiveWord;
734     var word = _array[i];
735     if (word != 0)
736     {
737         return GetLastSetBitForWord(i, word);
738     }
739     return -1;
740 }
741
742 [MethodImpl(MethodImplOptions.AggressiveInlining)]
743 public long CountSetBits()
744 {
745     var total = 0L;
746     GetBorders(out long from, out long to);
747     for (var i = from; i <= to; i++)
748     {
749         var word = _array[i];
750         if (word != 0)
751         {
752             total += CountSetBitsForWord(word);
753         }
754     }
755     return total;

```

```

756     }
757
758     [MethodImpl(MethodImplOptions.AggressiveInlining)]
759     public bool HaveCommonBits(BitString other)
760     {
761         EnsureBitStringHasTheSameSize(other, nameof(other));
762         GetCommonInnerBorders(this, other, out long from, out long to);
763         var otherArray = other._array;
764         for (var i = from; i <= to; i++)
765         {
766             var left = _array[i];
767             var right = otherArray[i];
768             if (left != 0 && right != 0 && (left & right) != 0)
769             {
770                 return true;
771             }
772         }
773         return false;
774     }
775
776     [MethodImpl(MethodImplOptions.AggressiveInlining)]
777     public long CountCommonBits(BitString other)
778     {
779         EnsureBitStringHasTheSameSize(other, nameof(other));
780         GetCommonInnerBorders(this, other, out long from, out long to);
781         var total = 0L;
782         var otherArray = other._array;
783         for (var i = from; i <= to; i++)
784         {
785             var left = _array[i];
786             var right = otherArray[i];
787             var combined = left & right;
788             if (combined != 0)
789             {
790                 total += CountSetBitsForWord(combined);
791             }
792         }
793         return total;
794     }
795
796     [MethodImpl(MethodImplOptions.AggressiveInlining)]
797     public List<long> GetCommonIndices(BitString other)
798     {
799         EnsureBitStringHasTheSameSize(other, nameof(other));
800         GetCommonInnerBorders(this, other, out long from, out long to);
801         var result = new List<long>();
802         var otherArray = other._array;
803         for (var i = from; i <= to; i++)
804         {
805             var left = _array[i];
806             var right = otherArray[i];
807             var combined = left & right;
808             if (combined != 0)
809             {
810                 AppendAllSetBitIndices(result, i, combined);
811             }
812         }
813         return result;
814     }
815
816     [MethodImpl(MethodImplOptions.AggressiveInlining)]
817     public List<ulong> GetCommonUInt64Indices(BitString other)
818     {
819         EnsureBitStringHasTheSameSize(other, nameof(other));
820         GetCommonBorders(this, other, out ulong from, out ulong to);
821         var result = new List<ulong>();
822         var otherArray = other._array;
823         for (var i = from; i <= to; i++)
824         {
825             var left = _array[i];
826             var right = otherArray[i];
827             var combined = left & right;
828             if (combined != 0)
829             {
830                 AppendAllSetBitIndices(result, i, combined);
831             }
832         }
833         return result;
834     }

```

```

835 [MethodImpl(MethodImplOptions.AggressiveInlining)]
836 public long GetFirstCommonBitIndex(BitString other)
837 {
838     EnsureBitStringHasTheSameSize(other, nameof(other));
839     GetCommonInnerBorders(this, other, out long from, out long to);
840     var otherArray = other._array;
841     for (var i = from; i <= to; i++)
842     {
843         var left = _array[i];
844         var right = otherArray[i];
845         var combined = left & right;
846         if (combined != 0)
847         {
848             return GetFirstSetBitForWord(i, combined);
849         }
850     }
851     return -1;
852 }
853
854 [MethodImpl(MethodImplOptions.AggressiveInlining)]
855 public long GetLastCommonBitIndex(BitString other)
856 {
857     EnsureBitStringHasTheSameSize(other, nameof(other));
858     GetCommonInnerBorders(this, other, out long from, out long to);
859     var otherArray = other._array;
860     for (var i = to; i >= from; i--)
861     {
862         var left = _array[i];
863         var right = otherArray[i];
864         var combined = left & right;
865         if (combined != 0)
866         {
867             return GetLastSetBitForWord(i, combined);
868         }
869     }
870     return -1;
871 }
872
873 [MethodImpl(MethodImplOptions.AggressiveInlining)]
874 public override bool Equals(object obj) => obj is BitString @string ? Equals(@string) :
875     => false;
876
877 [MethodImpl(MethodImplOptions.AggressiveInlining)]
878 public bool Equals(BitString other)
879 {
880     if (_length != other._length)
881     {
882         return false;
883     }
884     var otherArray = other._array;
885     if (_array.Length != otherArray.Length)
886     {
887         return false;
888     }
889     if (_minPositiveWord != other._minPositiveWord)
890     {
891         return false;
892     }
893     if (_maxPositiveWord != other._maxPositiveWord)
894     {
895         return false;
896     }
897     GetCommonBorders(this, other, out ulong from, out ulong to);
898     for (var i = from; i <= to; i++)
899     {
900         if (_array[i] != otherArray[i])
901         {
902             return false;
903         }
904     }
905     return true;
906 }
907
908 [MethodImpl(MethodImplOptions.AggressiveInlining)]
909 private void EnsureBitStringHasTheSameSize(BitString other, string argumentName)
910 {
911     Ensure.Always.ArgumentNotNull(other, argumentName);
912     if (_length != other._length)

```



```

913     {
914         throw new ArgumentException("Bit string must be the same size.", argumentName);
915     }
916 }
917
918 [MethodImpl(MethodImplOptions.AggressiveInlining)]
919 private void MarkBordersAsAllBitsReset() => SetBorders(_array.LongLength - 1, 0);
920
921 [MethodImpl(MethodImplOptions.AggressiveInlining)]
922 private void MarkBordersAsAllBitsSet() => SetBorders(0, _array.LongLength - 1);
923
924 [MethodImpl(MethodImplOptions.AggressiveInlining)]
925 private void GetBorders(out long from, out long to)
926 {
927     from = _minPositiveWord;
928     to = _maxPositiveWord;
929 }
930
931 [MethodImpl(MethodImplOptions.AggressiveInlining)]
932 private void GetBorders(out ulong from, out ulong to)
933 {
934     from = (ulong)_minPositiveWord;
935     to = (ulong)_maxPositiveWord;
936 }
937
938 [MethodImpl(MethodImplOptions.AggressiveInlining)]
939 private void SetBorders(long from, long to)
940 {
941     _minPositiveWord = from;
942     _maxPositiveWord = to;
943 }
944
945 [MethodImpl(MethodImplOptions.AggressiveInlining)]
946 private Range<long> GetValidIndexRange() => (0, _length - 1);
947
948 [MethodImpl(MethodImplOptions.AggressiveInlining)]
949 private static Range<long> GetValidLengthRange() => (0, long.MaxValue);
950
951 [MethodImpl(MethodImplOptions.AggressiveInlining)]
952 private static void AppendAllSetBitIndices(List<ulong> result, ulong wordIndex, long
    ↪ wordValue)
953 {
954     GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
    ↪ bits32to47, out byte[] bits48to63);
955     AppendAllSetIndices(result, wordIndex, bits00to15, bits16to31, bits32to47,
    ↪ bits48to63);
956 }
957
958 [MethodImpl(MethodImplOptions.AggressiveInlining)]
959 private static void AppendAllSetBitIndices(List<long> result, long wordIndex, long
    ↪ wordValue)
960 {
961     GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
    ↪ bits32to47, out byte[] bits48to63);
962     AppendAllSetBitIndices(result, wordIndex, bits00to15, bits16to31, bits32to47,
    ↪ bits48to63);
963 }
964
965 [MethodImpl(MethodImplOptions.AggressiveInlining)]
966 private static long CountSetBitsForWord(long word)
967 {
968     GetBits(word, out byte[] bits00to15, out byte[] bits16to31, out byte[] bits32to47,
    ↪ out byte[] bits48to63);
969     return bits00to15.LongLength + bits16to31.LongLength + bits32to47.LongLength +
    ↪ bits48to63.LongLength;
970 }
971
972 [MethodImpl(MethodImplOptions.AggressiveInlining)]
973 private static long GetFirstSetBitForWord(long wordIndex, long wordValue)
974 {
975     GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
    ↪ bits32to47, out byte[] bits48to63);
976     return GetFirstSetBit(wordIndex, bits00to15, bits16to31, bits32to47, bits48to63);
977 }
978
979 [MethodImpl(MethodImplOptions.AggressiveInlining)]
980 private static long GetLastSetBitForWord(long wordIndex, long wordValue)
981 {

```

```

982         GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
983             ↪ bits32to47, out byte[] bits48to63);
984         return GetLastSetBit(wordIndex, bits00to15, bits16to31, bits32to47, bits48to63);
985     }
986     [MethodImpl(MethodImplOptions.AggressiveInlining)]
987     private static void AppendAllSetBitIndices(List<long> result, long i, byte[] bits00to15,
988         ↪ byte[] bits16to31, byte[] bits32to47, byte[] bits48to63)
989     {
990         for (var j = 0; j < bits00to15.Length; j++)
991         {
992             result.Add(bits00to15[j] + (i * 64));
993         }
994         for (var j = 0; j < bits16to31.Length; j++)
995         {
996             result.Add(bits16to31[j] + 16 + (i * 64));
997         }
998         for (var j = 0; j < bits32to47.Length; j++)
999         {
1000             result.Add(bits32to47[j] + 32 + (i * 64));
1001         }
1002         for (var j = 0; j < bits48to63.Length; j++)
1003         {
1004             result.Add(bits48to63[j] + 48 + (i * 64));
1005         }
1006     }
1007     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1008     private static void AppendAllSetIndices(List<ulong> result, ulong i, byte[] bits00to15,
1009         ↪ byte[] bits16to31, byte[] bits32to47, byte[] bits48to63)
1010     {
1011         for (var j = 0; j < bits00to15.Length; j++)
1012         {
1013             result.Add(bits00to15[j] + (i * 64));
1014         }
1015         for (var j = 0; j < bits16to31.Length; j++)
1016         {
1017             result.Add(bits16to31[j] + 16UL + (i * 64));
1018         }
1019         for (var j = 0; j < bits32to47.Length; j++)
1020         {
1021             result.Add(bits32to47[j] + 32UL + (i * 64));
1022         }
1023         for (var j = 0; j < bits48to63.Length; j++)
1024         {
1025             result.Add(bits48to63[j] + 48UL + (i * 64));
1026         }
1027     }
1028     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1029     private static long GetFirstSetBit(long i, byte[] bits00to15, byte[] bits16to31, byte[]
1030         ↪ bits32to47, byte[] bits48to63)
1031     {
1032         if (bits00to15.Length > 0)
1033         {
1034             return bits00to15[0] + (i * 64);
1035         }
1036         if (bits16to31.Length > 0)
1037         {
1038             return bits16to31[0] + 16 + (i * 64);
1039         }
1040         if (bits32to47.Length > 0)
1041         {
1042             return bits32to47[0] + 32 + (i * 64);
1043         }
1044         return bits48to63[0] + 48 + (i * 64);
1045     }
1046     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1047     private static long GetLastSetBit(long i, byte[] bits00to15, byte[] bits16to31, byte[]
1048         ↪ bits32to47, byte[] bits48to63)
1049     {
1050         if (bits48to63.Length > 0)
1051         {
1052             return bits48to63[bits48to63.Length - 1] + 48 + (i * 64);
1053         }
1054         if (bits32to47.Length > 0)

```

```

1054     {
1055         return bits32to47[bits32to47.Length - 1] + 32 + (i * 64);
1056     }
1057     if (bits16to31.Length > 0)
1058     {
1059         return bits16to31[bits16to31.Length - 1] + 16 + (i * 64);
1060     }
1061     return bits00to15[bits00to15.Length - 1] + (i * 64);
1062 }
1063
1064 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1065 private static void GetBits(long word, out byte[] bits00to15, out byte[] bits16to31, out
    ↪ byte[] bits32to47, out byte[] bits48to63)
1066 {
1067     bits00to15 = _bitsSetIn16Bits[word & 0xffffu];
1068     bits16to31 = _bitsSetIn16Bits[(word >> 16) & 0xffffu];
1069     bits32to47 = _bitsSetIn16Bits[(word >> 32) & 0xffffu];
1070     bits48to63 = _bitsSetIn16Bits[(word >> 48) & 0xffffu];
1071 }
1072
1073 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1074 public static void GetCommonInnerBorders(BitString left, BitString right, out long from,
    ↪ out long to)
1075 {
1076     from = Math.Max(left._minPositiveWord, right._minPositiveWord);
1077     to = Math.Min(left._maxPositiveWord, right._maxPositiveWord);
1078 }
1079
1080 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1081 public static void GetCommonOuterBorders(BitString left, BitString right, out long from,
    ↪ out long to)
1082 {
1083     from = Math.Min(left._minPositiveWord, right._minPositiveWord);
1084     to = Math.Max(left._maxPositiveWord, right._maxPositiveWord);
1085 }
1086
1087 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1088 public static void GetCommonOuterBorders(BitString left, BitString right, out int from,
    ↪ out int to)
1089 {
1090     from = (int)Math.Min(left._minPositiveWord, right._minPositiveWord);
1091     to = (int)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 }

```

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 }

```

```

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

```

1.13 ./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 ==
    ↳ null || collection.Count == 0;
13
14     [MethodImpl(MethodImplOptions.AggressiveInlining)]
15     public static bool AllEqualToDefault<T>(this ICollection<T> collection)
16     {
17         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>
    ↳ dictionary, TKey key)
13        {
14            dictionary.TryGetValue(key, out TValue value);
15            return value;
16        }
17
18        [MethodImpl(MethodImplOptions.AggressiveInlining)]
19        public static TValue GetOrAdd<TKey, TValue>(this IDictionary<TKey, TValue> dictionary,
    ↳ TKey key, Func<TKey, TValue> valueFactory)
20        {
21            if (!dictionary.TryGetValue(key, out TValue value))
22            {
23                value = valueFactory(key);
24                dictionary.Add(key, value);
25                return value;
26            }
27            return value;
28        }
29    }
30 }

```

1.15 ./Platform.Collections/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) =>
    ↳ left.EqualTo(right, ContentEqualTo);
27
28        [MethodImpl(MethodImplOptions.AggressiveInlining)]
29        public static bool ContentEqualTo(this IList<char> left, IList<char> right)

```

```

30     {
31         for (var i = left.Count - 1; i >= 0; --i)
32         {
33             if (left[i] != right[i])
34             {
35                 return false;
36             }
37         }
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         }
34     }
35 }

```

```

31     var leftCount = left.GetCountOrZero();
32     var rightCount = right.GetCountOrZero();
33     if (leftCount == 0 && rightCount == 0)
34     {
35         return true;
36     }
37     if (leftCount == 0 || rightCount == 0 || leftCount != rightCount)
38     {
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)

```



```

109 {
110     var comparer = Comparer<T>.Default;
111     var leftCount = left.GetCountOrZero();
112     var rightCount = right.GetCountOrZero();
113     var intermediateResult = leftCount.CompareTo(rightCount);
114     for (var i = 0; intermediateResult == 0 && i < leftCount; i++)
115     {
116         intermediateResult = comparer.Compare(left[i], right[i]);
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(IList<TElement> list)
49     {
50         _list.Add(list[0]);
51         return _returnConstant;
52     }
53
54     [MethodImpl(MethodImplOptions.AggressiveInlining)]
55     public TReturnConstant AddAllValuesAndReturnConstant(IList<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(IList<char> @base, int offset, int length) : base(@base, offset,
15             ↪ length) { }
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public override int GetHashCode()
19         {
20             // Base can be not an array, but still IList<char>
21             if (Base is char[] baseArray)
22             {
23                 return baseArray.GenerateHashCode(Offset, Length);
24             }
25             else
26             {
27                 return this.GenerateHashCode();
28             }
29         }
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         public override bool Equals(Segment<char> other)
33         {
34             bool contentEqualityComparer(IList<char> left, IList<char> right)
35             {
36                 // Base can be not an array, but still IList<char>
37                 if (Base is char[] baseArray && other.Base is char[] otherArray)
38                 {
39                     return baseArray.ContentEqualTo(Offset, Length, otherArray, other.Offset);
40                 }
41                 else
42                 {
43                     return left.ContentEqualTo(right);
44                 }
45             }
46             return this.EqualTo(other, contentEqualityComparer);
47         }
48     }
49 }

```

```

46     }
47
48     [MethodImpl(MethodImplOptions.AggressiveInlining)]
49     public static implicit operator string(CharSegment segment)
50     {
51         if (!(segment.Base is char[] array))
52         {
53             array = segment.Base.ToArray();
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         }
39     }
40 }

```

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         private readonly bool _resetDictionaryOnEachWalk;
16         protected IDictionary<TSegment, long> Dictionary;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<TSegment, long>
20             ↪ dictionary, int minimumStringSegmentLength, bool resetDictionaryOnEachWalk)
21             : base(minimumStringSegmentLength)
22         {
23             Dictionary = dictionary;
24             _resetDictionaryOnEachWalk = resetDictionaryOnEachWalk;
25         }
26
27         [MethodImpl(MethodImplOptions.AggressiveInlining)]
28         protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<TSegment, long>
29             ↪ dictionary, int minimumStringSegmentLength) : this(dictionary,
30             ↪ minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<TSegment, long>
34             ↪ dictionary) : this(dictionary, DefaultMinimumStringSegmentLength,
35             ↪ DefaultResetDictionaryOnEachWalk) { }
36
37         [MethodImpl(MethodImplOptions.AggressiveInlining)]
38         protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength,
39             ↪ bool resetDictionaryOnEachWalk) : this(resetDictionaryOnEachWalk ? null : new
40             ↪ Dictionary<TSegment, long>(), minimumStringSegmentLength, resetDictionaryOnEachWalk)
41             ↪ { }
42
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength) :
45             ↪ this(minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
46
47         [MethodImpl(MethodImplOptions.AggressiveInlining)]
48         protected DictionaryBasedDuplicateSegmentsWalkerBase() :
49             ↪ this(DefaultMinimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
50
51         [MethodImpl(MethodImplOptions.AggressiveInlining)]
52         public override void WalkAll(ICollection<T> elements)
53         {
54             if (_resetDictionaryOnEachWalk)
55             {
56                 var capacity = Math.Ceiling(Math.Pow(elements.Count, 2) / 2);
57                 Dictionary = new Dictionary<TSegment, long>((int)capacity);
58             }
59             base.WalkAll(elements);
60         }
61
62         [MethodImpl(MethodImplOptions.AggressiveInlining)]
63         protected override long GetSegmentFrequency(TSegment segment) =>
64             ↪ Dictionary.GetOrDefault(segment);
65
66         [MethodImpl(MethodImplOptions.AggressiveInlining)]
67         protected override void SetSegmentFrequency(TSegment segment, long frequency) =>
68             ↪ Dictionary[segment] = frequency;
69     }
70 }

```

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)]

```

```

11     protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<Segment<T>, long>
    ↪ dictionary, int minimumStringSegmentLength, bool resetDictionaryOnEachWalk) :
    ↪ base(dictionary, minimumStringSegmentLength, resetDictionaryOnEachWalk) { }
12
13     [MethodImpl(MethodImplOptions.AggressiveInlining)]
14     protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<Segment<T>, long>
    ↪ dictionary, int minimumStringSegmentLength) : base(dictionary,
    ↪ minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
15
16     [MethodImpl(MethodImplOptions.AggressiveInlining)]
17     protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<Segment<T>, long>
    ↪ dictionary) : base(dictionary, DefaultMinimumStringSegmentLength,
    ↪ DefaultResetDictionaryOnEachWalk) { }
18
19     [MethodImpl(MethodImplOptions.AggressiveInlining)]
20     protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength,
    ↪ bool resetDictionaryOnEachWalk) : base(minimumStringSegmentLength,
    ↪ resetDictionaryOnEachWalk) { }
21
22     [MethodImpl(MethodImplOptions.AggressiveInlining)]
23     protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength) :
    ↪ base(minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
24
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>
    ↪ where TSegment : Segment<T>
8     {
9         [MethodImpl(MethodImplOptions.AggressiveInlining)]
10        protected DuplicateSegmentsWalkerBase(int minimumStringSegmentLength) :
    ↪ base(minimumStringSegmentLength) { }
11
12        [MethodImpl(MethodImplOptions.AggressiveInlining)]
13        protected DuplicateSegmentsWalkerBase() : base(DefaultMinimumStringSegmentLength) { }
14
15        [MethodImpl(MethodImplOptions.AggressiveInlining)]
16        protected override void Iteration(TSegment segment)
17        {
18            var frequency = GetSegmentFrequency(segment);
19            if (frequency == 1)
20            {
21                OnDuplicateFound(segment);
22            }
23            SetSegmentFrequency(segment, frequency + 1);
24        }
25
26        [MethodImpl(MethodImplOptions.AggressiveInlining)]
27        protected abstract void OnDuplicateFound(TSegment segment);
28
29        [MethodImpl(MethodImplOptions.AggressiveInlining)]
30        protected abstract long GetSegmentFrequency(TSegment segment);
31
32        [MethodImpl(MethodImplOptions.AggressiveInlining)]
33        protected abstract void SetSegmentFrequency(TSegment segment, long frequency);
34    }
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(ICollection<TElement> list)
35         {
36             _set.Add(list[0]);
37             return true;
38         }
39
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         public TReturnConstant AddAndReturnConstant(TElement element)
42         {
43             _set.Add(element);
44             return _returnConstant;
45         }
46
47         [MethodImpl(MethodImplOptions.AggressiveInlining)]
48         public TReturnConstant AddFirstAndReturnConstant(ICollection<TElement> list)
49         {
50             _set.Add(list[0]);
51             return _returnConstant;
52         }
53     }
54 }
```


1.32 ./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             }
55         }
56     }
57 }

```

```

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 firstHashCode = new CharSegment(testArray, 0, 4).GetHashCode();
14         var secondHashCode = new CharSegment(testArray, 5, 4).GetHashCode();
15         Assert.Equal(firstHashCode, secondHashCode);
16     }
17
18     [Fact]
19     public static void EqualsTest()
20     {
21         const string testString = "test test";
22         var testArray = testString.ToCharArray();
23         var first = new CharSegment(testArray, 0, 4);
24         var second = new CharSegment(testArray, 5, 4);
25         Assert.True(first.Equals(second));
26     }
27 }
28 }

```

1.41 ./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             Assert.Equal("Hello", "hello".CapitalizeFirstLetter());
11             Assert.Equal("Hello", "Hello".CapitalizeFirstLetter());
12             Assert.Equal(" Hello", " hello".CapitalizeFirstLetter());
13         }
14
15         [Fact]
16         public static void TrimSingleTest()
17         {
18             Assert.Equal("", "".TrimSingle('\'));
19             Assert.Equal("", "''.TrimSingle('\'));
20             Assert.Equal("hello", "'hello'".TrimSingle('\'));
21             Assert.Equal("hello", "hello'".TrimSingle('\'));
22             Assert.Equal("hello", "'hello".TrimSingle('\'));
23         }
24     }
25 }

```

Index

- ./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, 32
- ./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, 34
- ./Platform.Collections/StringExtensions.cs, 34
- ./Platform.Collections/Trees/Node.cs, 35