

LinksPlatform's Platform.Collections Class Library

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

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Collections.Arrays
7  {
8      public class ArrayFiller<TElement>
9      {
10         protected readonly TElement[] _array;
11         protected long _position;
12
13         public ArrayFiller(TElement[] array, long offset)
14         {
15             _array = array;
16             _position = offset;
17         }
18
19         public ArrayFiller(TElement[] array) : this(array, 0) { }
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public void Add(TElement element) => _array[_position++] = element;
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         public bool AddAndReturnTrue(TElement element)
26         {
27             _array[_position++] = element;
28             return true;
29         }
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         public bool AddFirstAndReturnTrue(ICollection<TElement> collection)
33         {
34             _array[_position++] = collection[0];
35             return true;
36         }
37     }
38 }

```

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

```

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

```

./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 }

```

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

```

1 using System;
2 using System.Collections.Generic;
3 using Platform.Exceptions;
4 using Platform.Disposables;
5 using Platform.Ranges;
6 using Platform.Collections.Stacks;
7
8 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Collections.Arrays
11 {
12     /// <remarks>
13     /// Original idea from
14     /// ↪ http://geekswithblogs.net/blackrob/archive/2014/12/18/array-pooling-in-csharp.aspx
15     /// </remarks>
16     public class ArrayPool<T>
17     {
18         public static readonly T[] Empty = new T[0];
19
20         // May be use Default class for that later.
21         [ThreadStatic]
22         internal static ArrayPool<T> _threadInstance;
23         internal static ArrayPool<T> ThreadInstance { get => _threadInstance ?? (_threadInstance
24             ↪ = new ArrayPool<T>()); }
25
26         private readonly int _maxArraysPerSize;
27         private readonly Dictionary<int, Stack<T[]>> _pool = new Dictionary<int,
28             ↪ Stack<T[]>>(ArrayPool.DefaultSizesAmount);
29
30         public ArrayPool(int maxArraysPerSize) => _maxArraysPerSize = maxArraysPerSize;
31
32         public ArrayPool() : this(ArrayPool.DefaultMaxArraysPerSize) { }
33
34         public Disposable<T[]> AllocateDisposable(long size) => (Allocate(size), Free);
35
36         public Disposable<T[]> Resize(Disposable<T[]> source, long size)
37         {
38             var destination = AllocateDisposable(size);
39             T[] sourceArray = source;
40             T[] destinationArray = destination;
41             Array.Copy(sourceArray, destinationArray, size < sourceArray.Length ? (int)size :
42                 ↪ sourceArray.Length);
43             source.Dispose();
44             return destination;
45         }
46
47         public virtual void Clear() => _pool.Clear();
48
49         public virtual T[] Allocate(long size)
50         {
51             Ensure.Always.ArgumentInRange(size, (0, int.MaxValue));
52             return size == 0 ? Empty : _pool.GetOrDefault((int)size)?.PopOrDefault() ?? new
53                 ↪ T[size];
54         }
55
56         public virtual void Free(T[] array)
57         {
58             Ensure.Always.ArgumentNotNull(array, nameof(array));
59             if (array.Length == 0)
60             {
61                 return;
62             }
63             var stack = _pool.GetOrAdd(array.Length, size => new Stack<T[]>(_maxArraysPerSize));
64             if (stack.Count == _maxArraysPerSize) // Stack is full

```

```

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

```

./Platform.Collections/Arrays/ArrayString.cs

```

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

```

./Platform.Collections/Arrays/CharArrayExtensions.cs

```

1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  namespace Platform.Collections.Arrays
4  {
5      public static unsafe class CharArrayExtensions
6      {
7          /// <remarks>
8          /// Based on https://github.com/Microsoft/referencesource/blob/3b1eaf5203992df69de44c783\_a3eda37d3d4cd10/mscorlib/system/string.cs#L833
9          /// </remarks>
10         public static int GenerateHashCode(this char[] array, int offset, int length)
11         {
12             var hashSeed = 5381;
13             var hashAccumulator = hashSeed;
14             fixed (char* pointer = &array[offset])
15             {
16                 for (char* s = pointer, last = s + length; s < last; s++)
17                 {
18                     hashAccumulator = (hashAccumulator << 5) + hashAccumulator ^ *s;
19                 }
20             }
21             return hashAccumulator + (hashSeed * 1566083941);
22         }
23
24         /// <remarks>
25         /// Based on https://github.com/Microsoft/referencesource/blob/3b1eaf5203992df69de44c783\_a3eda37d3d4cd10/mscorlib/system/string.cs#L364
26         /// </remarks>
27         public static bool ContentEqualTo(this char[] left, int leftOffset, int length, char[]
28         ↪ right, int rightOffset)
29         {
30             fixed (char* leftPointer = &left[leftOffset])
31             {
32                 fixed (char* rightPointer = &right[rightOffset])
33                 {
34                     char* leftPointerCopy = leftPointer, rightPointerCopy = rightPointer;
35                     if (!CheckArraysMainPartForEquality(ref leftPointerCopy, ref
36                     ↪ rightPointerCopy, ref length))
37                     {
38                         return false;
39                     }
40                     CheckArraysRemainderForEquality(ref leftPointerCopy, ref rightPointerCopy,
41                     ↪ ref length);
42                     return length <= 0;
43                 }
44             }
45         }
46
47         private static bool CheckArraysMainPartForEquality(ref char* left, ref char* right, ref
48         ↪ int length)
49         {
50             while (length >= 10)
51             {
52                 if ((* (int*)left != *(int*)right)

```

```

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

```

./Platform.Collections/Arrays/GenericArrayExtensions.cs

```

1 using System;
2 using System.Collections.Generic;
3 using System.Text;
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         public static T[] Clone<T>(this T[] array)
12         {
13             var copy = new T[array.Length];
14             Array.Copy(array, 0, copy, 0, array.Length);
15             return copy;
16         }
17     }
18 }

```

./Platform.Collections/BitString.cs

```

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

```

```

21 {
22     private static readonly byte[][] _bitsSetIn16Bits;
23     private long[] _array;
24     private long _length;
25     private long _minPositiveWord;
26     private long _maxPositiveWord;
27
28     public bool this[long index]
29     {
30         get => Get(index);
31         set => Set(index, value);
32     }
33
34     public long Length
35     {
36         get => _length;
37         set
38         {
39             if (_length == value)
40             {
41                 return;
42             }
43             Ensure.Always.ArgumentInRange(value, GetValidLengthRange(), nameof(Length));
44             // Currently we never shrink the array
45             if (value > _length)
46             {
47                 var words = GetWordsCountFromIndex(value);
48                 var oldWords = GetWordsCountFromIndex(_length);
49                 if (words > _array.LongLength)
50                 {
51                     var copy = new long[words];
52                     Array.Copy(_array, copy, _array.LongLength);
53                     _array = copy;
54                 }
55                 else
56                 {
57                     // What is going on here?
58                     Array.Clear(_array, (int)oldWords, (int)(words - oldWords));
59                 }
60                 // What is going on here?
61                 var mask = (int)(_length % 64);
62                 if (mask > 0)
63                 {
64                     _array[oldWords - 1] &= (1L << mask) - 1;
65                 }
66             }
67             else
68             {
69                 // Looks like minimum and maximum positive words are not updated
70                 throw new NotImplementedException();
71             }
72             _length = value;
73         }
74     }
75
76     #region Constructors
77
78     static BitString()
79     {
80         _bitsSetIn16Bits = new byte[65536][];
81         int i, c, k;
82         byte bitIndex;
83         for (i = 0; i < 65536; i++)
84         {
85             // Calculating size of array (number of positive bits)
86             for (c = 0, k = 1; k <= 65536; k <= 1)
87             {
88                 if ((i & k) == k)
89                 {
90                     c++;
91                 }
92             }
93             var array = new byte[c];
94             // Adding positive bits indices into array
95             for (bitIndex = 0, c = 0, k = 1; k <= 65536; k <= 1)
96             {
97                 if ((i & k) == k)
98                 {
99                     array[c++] = bitIndex;

```

```

100         }
101         bitIndex++;
102     }
103     _bitsSetIn16Bits[i] = array;
104 }
105 }
106
107 public BitString(BitString other)
108 {
109     Ensure.Always.ArgumentNotNull(other, nameof(other));
110     _length = other._length;
111     _array = new long[GetWordsCountFromIndex(_length)];
112     _minPositiveWord = other._minPositiveWord;
113     _maxPositiveWord = other._maxPositiveWord;
114     Array.Copy(other._array, _array, _array.LongLength);
115 }
116
117 public BitString(long length)
118 {
119     Ensure.Always.ArgumentInRange(length, GetValidLengthRange(), nameof(length));
120     _length = length;
121     _array = new long[GetWordsCountFromIndex(_length)];
122     MarkBordersAsAllBitsReset();
123 }
124
125 public BitString(long length, bool defaultValue)
126 : this(length)
127 {
128     if (defaultValue)
129     {
130         SetAll();
131     }
132 }
133
134 #endregion
135
136 public BitString Not()
137 {
138     for (var i = 0; i < _array.Length; i++)
139     {
140         _array[i] = ~_array[i];
141         RefreshBordersByWord(i);
142     }
143     return this;
144 }
145
146 public BitString VectorNot()
147 {
148     var thisVector = new Vector<long>(_array);
149     var result = ~thisVector;
150     result.CopyTo(_array, 0);
151     MarkBordersAsAllBitsSet();
152     TryShrinkBorders();
153     return this;
154 }
155
156 public BitString And(BitString other)
157 {
158     EnsureBitStringHasTheSameSize(other, nameof(other));
159     GetCommonInnerBorders(this, other, out long from, out long to);
160     var otherArray = other._array;
161     for (var i = from; i <= to; i++)
162     {
163         _array[i] &= otherArray[i];
164         RefreshBordersByWord(i);
165     }
166     return this;
167 }
168
169 public BitString VectorAnd(BitString other)
170 {
171     EnsureBitStringHasTheSameSize(other, nameof(other));
172     var thisVector = new Vector<long>(_array);
173     var otherVector = new Vector<long>(other._array);
174     var result = thisVector & otherVector;
175     result.CopyTo(_array, 0);
176     MarkBordersAsAllBitsSet();
177     TryShrinkBorders();
178     return this;

```

```

179     }
180
181     public BitString Or(BitString other)
182     {
183         EnsureBitStringHasTheSameSize(other, nameof(other));
184         GetCommonOuterBorders(this, other, out long from, out long to);
185         for (var i = from; i <= to; i++)
186         {
187             _array[i] |= other._array[i];
188             RefreshBordersByWord(i);
189         }
190         return this;
191     }
192
193     public BitString VectorOr(BitString other)
194     {
195         EnsureBitStringHasTheSameSize(other, nameof(other));
196         var thisVector = new Vector<long>(_array);
197         var otherVector = new Vector<long>(other._array);
198         var result = thisVector | otherVector;
199         result.CopyTo(_array, 0);
200         MarkBordersAsAllBitsSet();
201         TryShrinkBorders();
202         return this;
203     }
204
205     public BitString Xor(BitString other)
206     {
207         EnsureBitStringHasTheSameSize(other, nameof(other));
208         GetCommonOuterBorders(this, other, out long from, out long to);
209         for (var i = from; i <= to; i++)
210         {
211             _array[i] ^= other._array[i];
212             RefreshBordersByWord(i);
213         }
214         return this;
215     }
216
217     private void RefreshBordersByWord(long wordIndex)
218     {
219         if (_array[wordIndex] == 0)
220         {
221             if (wordIndex == _minPositiveWord && wordIndex != _array.LongLength - 1)
222             {
223                 _minPositiveWord++;
224             }
225             if (wordIndex == _maxPositiveWord && wordIndex != 0)
226             {
227                 _maxPositiveWord--;
228             }
229         }
230         else
231         {
232             if (wordIndex < _minPositiveWord)
233             {
234                 _minPositiveWord = wordIndex;
235             }
236             if (wordIndex > _maxPositiveWord)
237             {
238                 _maxPositiveWord = wordIndex;
239             }
240         }
241     }
242
243     public bool TryShrinkBorders()
244     {
245         GetBorders(out long from, out long to);
246         while (from <= to && _array[from] == 0)
247         {
248             from++;
249         }
250         if (from > to)
251         {
252             MarkBordersAsAllBitsReset();
253             return true;
254         }
255         while (to >= from && _array[to] == 0)
256         {
257             to--;

```

```

258     }
259     if (to < from)
260     {
261         MarkBordersAsAllBitsReset();
262         return true;
263     }
264     var bordersUpdated = from != _minPositiveWord || to != _maxPositiveWord;
265     if (bordersUpdated)
266     {
267         SetBorders(from, to);
268     }
269     return bordersUpdated;
270 }
271
272 [MethodImpl(MethodImplOptions.AggressiveInlining)]
273 public bool Get(long index)
274 {
275     Ensure.Always.ArgumentInRange(index, GetValidIndexRange(), nameof(index));
276     return (_array[GetWordIndexFromIndex(index)] & GetBitMaskFromIndex(index)) != 0;
277 }
278
279 [MethodImpl(MethodImplOptions.AggressiveInlining)]
280 public void Set(long index, bool value)
281 {
282     if (value)
283     {
284         Set(index);
285     }
286     else
287     {
288         Reset(index);
289     }
290 }
291
292 [MethodImpl(MethodImplOptions.AggressiveInlining)]
293 public void Set(long index)
294 {
295     Ensure.Always.ArgumentInRange(index, GetValidIndexRange(), nameof(index));
296     var wordIndex = GetWordIndexFromIndex(index);
297     var mask = GetBitMaskFromIndex(index);
298     _array[wordIndex] |= mask;
299     RefreshBordersByWord(wordIndex);
300 }
301
302 [MethodImpl(MethodImplOptions.AggressiveInlining)]
303 public void Reset(long index)
304 {
305     Ensure.Always.ArgumentInRange(index, GetValidIndexRange(), nameof(index));
306     var wordIndex = GetWordIndexFromIndex(index);
307     var mask = GetBitMaskFromIndex(index);
308     _array[wordIndex] &= ~mask;
309     RefreshBordersByWord(wordIndex);
310 }
311
312 public bool Add(long index)
313 {
314     var wordIndex = GetWordIndexFromIndex(index);
315     var mask = GetBitMaskFromIndex(index);
316     if ((_array[wordIndex] & mask) == 0)
317     {
318         _array[wordIndex] |= mask;
319         RefreshBordersByWord(wordIndex);
320         return true;
321     }
322     else
323     {
324         return false;
325     }
326 }
327
328 public void SetAll(bool value)
329 {
330     if (value)
331     {
332         SetAll();
333     }
334     else
335     {
336         ResetAll();

```



```

337     }
338 }
339
340 public void SetAll()
341 {
342     const long fillValue = unchecked((long)0xffffffffffffffff);
343     var words = GetWordsCountFromIndex(_length);
344     for (var i = 0; i < words; i++)
345     {
346         _array[i] = fillValue;
347     }
348     MarkBordersAsAllBitsSet();
349 }
350
351 public void ResetAll()
352 {
353     const long fillValue = 0;
354     GetBorders(out long from, out long to);
355     for (var i = from; i <= to; i++)
356     {
357         _array[i] = fillValue;
358     }
359     MarkBordersAsAllBitsReset();
360 }
361
362 public List<long> GetSetIndices()
363 {
364     var result = new List<long>();
365     GetBorders(out long from, out long to);
366     for (var i = from; i <= to; i++)
367     {
368         var word = _array[i];
369         if (word != 0)
370         {
371             AppendAllSetBitIndices(result, i, word);
372         }
373     }
374     return result;
375 }
376
377 public List<ulong> GetSetUInt64Indices()
378 {
379     var result = new List<ulong>();
380     GetBorders(out ulong from, out ulong to);
381     for (var i = from; i <= to; i++)
382     {
383         var word = _array[i];
384         if (word != 0)
385         {
386             AppendAllSetBitIndices(result, i, word);
387         }
388     }
389     return result;
390 }
391
392 public long GetFirstSetBitIndex()
393 {
394     var i = _minPositiveWord;
395     var word = _array[i];
396     if (word != 0)
397     {
398         return GetFirstSetBitForWord(i, word);
399     }
400     return -1;
401 }
402
403 public long GetLastSetBitIndex()
404 {
405     var i = _maxPositiveWord;
406     var word = _array[i];
407     if (word != 0)
408     {
409         return GetLastSetBitForWord(i, word);
410     }
411     return -1;
412 }
413
414 public long CountSetBits()
415 {

```

```

416     var total = 0L;
417     GetBorders(out long from, out long to);
418     for (var i = from; i <= to; i++)
419     {
420         var word = _array[i];
421         if (word != 0)
422         {
423             total += CountSetBitsForWord(word);
424         }
425     }
426     return total;
427 }
428
429 public bool HaveCommonBits(BitString other)
430 {
431     EnsureBitStringHasTheSameSize(other, nameof(other));
432     GetCommonInnerBorders(this, other, out long from, out long to);
433     var otherArray = other._array;
434     for (var i = from; i <= to; i++)
435     {
436         var left = _array[i];
437         var right = otherArray[i];
438         if (left != 0 && right != 0 && (left & right) != 0)
439         {
440             return true;
441         }
442     }
443     return false;
444 }
445
446 public long CountCommonBits(BitString other)
447 {
448     EnsureBitStringHasTheSameSize(other, nameof(other));
449     GetCommonInnerBorders(this, other, out long from, out long to);
450     var total = 0L;
451     var otherArray = other._array;
452     for (var i = from; i <= to; i++)
453     {
454         var left = _array[i];
455         var right = otherArray[i];
456         var combined = left & right;
457         if (combined != 0)
458         {
459             total += CountSetBitsForWord(combined);
460         }
461     }
462     return total;
463 }
464
465 public List<long> GetCommonIndices(BitString other)
466 {
467     EnsureBitStringHasTheSameSize(other, nameof(other));
468     GetCommonInnerBorders(this, other, out long from, out long to);
469     var result = new List<long>();
470     var otherArray = other._array;
471     for (var i = from; i <= to; i++)
472     {
473         var left = _array[i];
474         var right = otherArray[i];
475         var combined = left & right;
476         if (combined != 0)
477         {
478             AppendAllSetBitIndices(result, i, combined);
479         }
480     }
481     return result;
482 }
483
484 public List<ulong> GetCommonUInt64Indices(BitString other)
485 {
486     EnsureBitStringHasTheSameSize(other, nameof(other));
487     GetCommonBorders(this, other, out ulong from, out ulong to);
488     var result = new List<ulong>();
489     var otherArray = other._array;
490     for (var i = from; i <= to; i++)
491     {
492         var left = _array[i];
493         var right = otherArray[i];
494         var combined = left & right;

```

```

495         if (combined != 0)
496         {
497             AppendAllSetBitIndices(result, i, combined);
498         }
499     }
500     return result;
501 }
502
503 public long GetFirstCommonBitIndex(BitString other)
504 {
505     EnsureBitStringHasTheSameSize(other, nameof(other));
506     GetCommonInnerBorders(this, other, out long from, out long to);
507     var otherArray = other._array;
508     for (var i = from; i <= to; i++)
509     {
510         var left = _array[i];
511         var right = otherArray[i];
512         var combined = left & right;
513         if (combined != 0)
514         {
515             return GetFirstSetBitForWord(i, combined);
516         }
517     }
518     return -1;
519 }
520
521 public long GetLastCommonBitIndex(BitString other)
522 {
523     EnsureBitStringHasTheSameSize(other, nameof(other));
524     GetCommonInnerBorders(this, other, out long from, out long to);
525     var otherArray = other._array;
526     for (var i = to; i >= from; i--)
527     {
528         var left = _array[i];
529         var right = otherArray[i];
530         var combined = left & right;
531         if (combined != 0)
532         {
533             return GetLastSetBitForWord(i, combined);
534         }
535     }
536     return -1;
537 }
538
539 public override bool Equals(object obj) => obj is BitString @string ? Equals(@string) :
540     ↪ false;
541
542 public bool Equals(BitString other)
543 {
544     if (_length != other._length)
545     {
546         return false;
547     }
548     if (_array.Length != other._array.Length)
549     {
550         return false;
551     }
552     if (_minPositiveWord != other._minPositiveWord)
553     {
554         return false;
555     }
556     if (_maxPositiveWord != other._maxPositiveWord)
557     {
558         return false;
559     }
560     GetCommonBorders(this, other, out ulong from, out ulong to);
561     for (var i = from; i <= to; i++)
562     {
563         if (_array[i] != other._array[i])
564         {
565             return false;
566         }
567     }
568     return true;
569 }
570
571 [MethodImpl(MethodImplOptions.AggressiveInlining)]
572 private void EnsureBitStringHasTheSameSize(BitString other, string argumentName)
573 {

```

```

573     Ensure.Always.ArgumentNotNull(other, argumentName);
574     if (_length != other._length)
575     {
576         throw new ArgumentException("Bit string must be the same size.", argumentName);
577     }
578 }
579
580 [MethodImpl(MethodImplOptions.AggressiveInlining)]
581 private void MarkBordersAsAllBitsReset() => SetBorders(_array.LongLength - 1, 0);
582
583 [MethodImpl(MethodImplOptions.AggressiveInlining)]
584 private void MarkBordersAsAllBitsSet() => SetBorders(0, _array.LongLength - 1);
585
586 [MethodImpl(MethodImplOptions.AggressiveInlining)]
587 private void GetBorders(out long from, out long to)
588 {
589     from = _minPositiveWord;
590     to = _maxPositiveWord;
591 }
592
593 [MethodImpl(MethodImplOptions.AggressiveInlining)]
594 private void GetBorders(out ulong from, out ulong to)
595 {
596     from = (ulong)_minPositiveWord;
597     to = (ulong)_maxPositiveWord;
598 }
599
600 [MethodImpl(MethodImplOptions.AggressiveInlining)]
601 private void SetBorders(long from, long to)
602 {
603     _minPositiveWord = from;
604     _maxPositiveWord = to;
605 }
606
607 [MethodImpl(MethodImplOptions.AggressiveInlining)]
608 private Range<long> GetValidIndexRange() => (0, _length - 1);
609
610 [MethodImpl(MethodImplOptions.AggressiveInlining)]
611 private static Range<long> GetValidLengthRange() => (0, long.MaxValue);
612
613 [MethodImpl(MethodImplOptions.AggressiveInlining)]
614 private static void AppendAllSetBitIndices(List<ulong> result, ulong wordIndex, long
    ↪ wordValue)
615 {
616     GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
    ↪ bits32to47, out byte[] bits48to63);
617     AppendAllSetIndices(result, wordIndex, bits00to15, bits16to31, bits32to47,
    ↪ bits48to63);
618 }
619
620 [MethodImpl(MethodImplOptions.AggressiveInlining)]
621 private static void AppendAllSetBitIndices(List<long> result, long wordIndex, long
    ↪ wordValue)
622 {
623     GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
    ↪ bits32to47, out byte[] bits48to63);
624     AppendAllSetBitIndices(result, wordIndex, bits00to15, bits16to31, bits32to47,
    ↪ bits48to63);
625 }
626
627 [MethodImpl(MethodImplOptions.AggressiveInlining)]
628 private static long CountSetBitsForWord(long word)
629 {
630     GetBits(word, out byte[] bits00to15, out byte[] bits16to31, out byte[] bits32to47,
    ↪ out byte[] bits48to63);
631     return bits00to15.LongLength + bits16to31.LongLength + bits32to47.LongLength +
    ↪ bits48to63.LongLength;
632 }
633
634 [MethodImpl(MethodImplOptions.AggressiveInlining)]
635 private static long GetFirstSetBitForWord(long wordIndex, long wordValue)
636 {
637     GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
    ↪ bits32to47, out byte[] bits48to63);
638     return GetFirstSetBit(wordIndex, bits00to15, bits16to31, bits32to47, bits48to63);
639 }
640
641 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

642 private static long GetLastSetBitForWord(long wordIndex, long wordValue)
643 {
644     GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
        ↳ bits32to47, out byte[] bits48to63);
645     return GetLastSetBit(wordIndex, bits00to15, bits16to31, bits32to47, bits48to63);
646 }
647
648 private static void AppendAllSetBitIndices(List<long> result, long i, byte[] bits00to15,
        ↳ byte[] bits16to31, byte[] bits32to47, byte[] bits48to63)
649 {
650     for (var j = 0; j < bits00to15.Length; j++)
651     {
652         result.Add(bits00to15[j] + (i * 64));
653     }
654     for (var j = 0; j < bits16to31.Length; j++)
655     {
656         result.Add(bits16to31[j] + 16 + (i * 64));
657     }
658     for (var j = 0; j < bits32to47.Length; j++)
659     {
660         result.Add(bits32to47[j] + 32 + (i * 64));
661     }
662     for (var j = 0; j < bits48to63.Length; j++)
663     {
664         result.Add(bits48to63[j] + 48 + (i * 64));
665     }
666 }
667
668 private static void AppendAllSetIndices(List<ulong> result, ulong i, byte[] bits00to15,
        ↳ byte[] bits16to31, byte[] bits32to47, byte[] bits48to63)
669 {
670     for (var j = 0; j < bits00to15.Length; j++)
671     {
672         result.Add(bits00to15[j] + (i * 64));
673     }
674     for (var j = 0; j < bits16to31.Length; j++)
675     {
676         result.Add(bits16to31[j] + 16UL + (i * 64));
677     }
678     for (var j = 0; j < bits32to47.Length; j++)
679     {
680         result.Add(bits32to47[j] + 32UL + (i * 64));
681     }
682     for (var j = 0; j < bits48to63.Length; j++)
683     {
684         result.Add(bits48to63[j] + 48UL + (i * 64));
685     }
686 }
687
688 private static long GetFirstSetBit(long i, byte[] bits00to15, byte[] bits16to31, byte[]
        ↳ bits32to47, byte[] bits48to63)
689 {
690     if (bits00to15.Length > 0)
691     {
692         return bits00to15[0] + (i * 64);
693     }
694     if (bits16to31.Length > 0)
695     {
696         return bits16to31[0] + 16 + (i * 64);
697     }
698     if (bits32to47.Length > 0)
699     {
700         return bits32to47[0] + 32 + (i * 64);
701     }
702     return bits48to63[0] + 48 + (i * 64);
703 }
704
705 private static long GetLastSetBit(long i, byte[] bits00to15, byte[] bits16to31, byte[]
        ↳ bits32to47, byte[] bits48to63)
706 {
707     if (bits48to63.Length > 0)
708     {
709         return bits48to63[bits48to63.Length - 1] + 48 + (i * 64);
710     }
711     if (bits32to47.Length > 0)
712     {
713         return bits32to47[bits32to47.Length - 1] + 32 + (i * 64);

```

```

714     }
715     if (bits16to31.Length > 0)
716     {
717         return bits16to31[bits16to31.Length - 1] + 16 + (i * 64);
718     }
719     return bits00to15[bits00to15.Length - 1] + (i * 64);
720 }
721
722 private static void GetBits(long word, out byte[] bits00to15, out byte[] bits16to31, out
    ↪ byte[] bits32to47, out byte[] bits48to63)
723 {
724     bits00to15 = _bitsSetIn16Bits[word & 0xffffu];
725     bits16to31 = _bitsSetIn16Bits[(word >> 16) & 0xffffu];
726     bits32to47 = _bitsSetIn16Bits[(word >> 32) & 0xffffu];
727     bits48to63 = _bitsSetIn16Bits[(word >> 48) & 0xffffu];
728 }
729
730 [MethodImpl(MethodImplOptions.AggressiveInlining)]
731 public static void GetCommonInnerBorders(BitString left, BitString right, out long from,
    ↪ out long to)
732 {
733     from = Math.Max(left._minPositiveWord, right._minPositiveWord);
734     to = Math.Min(left._maxPositiveWord, right._maxPositiveWord);
735 }
736
737 [MethodImpl(MethodImplOptions.AggressiveInlining)]
738 public static void GetCommonOuterBorders(BitString left, BitString right, out long from,
    ↪ out long to)
739 {
740     from = Math.Min(left._minPositiveWord, right._minPositiveWord);
741     to = Math.Max(left._maxPositiveWord, right._maxPositiveWord);
742 }
743
744 [MethodImpl(MethodImplOptions.AggressiveInlining)]
745 public static void GetCommonBorders(BitString left, BitString right, out ulong from, out
    ↪ ulong to)
746 {
747     from = (ulong)Math.Max(left._minPositiveWord, right._minPositiveWord);
748     to = (ulong)Math.Min(left._maxPositiveWord, right._maxPositiveWord);
749 }
750
751 [MethodImpl(MethodImplOptions.AggressiveInlining)]
752 public static long GetWordsCountFromIndex(long index) => (index + 63) / 64;
753
754 [MethodImpl(MethodImplOptions.AggressiveInlining)]
755 public static long GetWordIndexFromIndex(long index) => index >> 6;
756
757 [MethodImpl(MethodImplOptions.AggressiveInlining)]
758 public static long GetBitMaskFromIndex(long index) => 1L << (int)(index & 63);
759
760 public override int GetHashCode() => base.GetHashCode();
761
762 public override string ToString() => base.ToString();
763 }
764 }

```

./Platform.Collections/BitStringExtensions.cs

```

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

```

./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 }

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

./Platform.Collections/EnsureExtensions.cs
1 using System;
2 using System.Collections.Generic;
3 using System.Diagnostics;
4 using System.Runtime.CompilerServices;
5 using Platform.Exceptions;
6 using Platform.Exceptions.ExtensionRoots;
7
8 #pragma warning disable IDE0060 // Remove unused parameter
9 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11 namespace Platform.Collections
12 {
13     public static class EnsureExtensions
14     {
15         #region Always
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public static void ArgumentNotEmpty<T>(this EnsureAlwaysExtensionRoot root,
            ↪ ICollection<T> argument, string argumentName, string message)
19         {
20             if (argument.IsNullOrEmpty())
21             {
22                 throw new ArgumentException(message, argumentName);
23             }
24         }
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public static void ArgumentNotEmpty<T>(this EnsureAlwaysExtensionRoot root,
            ↪ ICollection<T> argument, string argumentName) => ArgumentNotEmpty(root, argument,
            ↪ argumentName, null);
28
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         public static void ArgumentNotEmpty<T>(this EnsureAlwaysExtensionRoot root,
            ↪ ICollection<T> argument) => ArgumentNotEmpty(root, argument, null, null);
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureAlwaysExtensionRoot root,
            ↪ string argument, string argumentName, string message)
34         {

```

```

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

```

./Platform.Collections/ICollectionExtensions.cs

```

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

```

./Platform.Collections/IDictionaryExtensions.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4

```



```

5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Collections
8 {
9     public static class IDictionaryExtensions
10    {
11        [MethodImpl(MethodImplOptions.AggressiveInlining)]
12        public static TValue GetOrDefault<TKey, TValue>(this IDictionary<TKey, TValue>
13            ↪ dictionary, TKey key)
14        {
15            dictionary.TryGetValue(key, out TValue value);
16            return value;
17        }
18
19        [MethodImpl(MethodImplOptions.AggressiveInlining)]
20        public static TValue GetOrAdd<TKey, TValue>(this IDictionary<TKey, TValue> dictionary,
21            ↪ TKey key, Func<TKey, TValue> valueFactory)
22        {
23            if (!dictionary.TryGetValue(key, out TValue value))
24            {
25                value = valueFactory(key);
26                dictionary.Add(key, value);
27                return value;
28            }
29            return value;
30        }
31    }
32 }

```

./Platform.Collections/ISetExtensions.cs

```

1 using System.Collections.Generic;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Collections
6 {
7     public static class ISetExtensions
8     {
9         public static void AddAndReturnVoid<T>(this ISet<T> set, T element) => set.Add(element);
10        public static void RemoveAndReturnVoid<T>(this ISet<T> set, T element) =>
11            ↪ set.Remove(element);
12        public static bool DoNotContains<T>(this ISet<T> set, T element) =>
13            ↪ !set.Contains(element);
14    }
15 }

```

./Platform.Collections/Lists/CharListExtensions.cs

```

1 using System.Collections.Generic;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Collections.Lists
6 {
7     public static class CharListExtensions
8     {
9         /// <remarks>
10        /// Based on https://github.com/Microsoft/referencesource/blob/3b1eaf5203992df69de44c783\_a3eda37d3d4cd10/mscorlib/system/string.cs#L833
11        /// </remarks>
12        public static unsafe int GenerateHashCode(this IList<char> list)
13        {
14            var hashSeed = 5381;
15            var hashAccumulator = hashSeed;
16            for (var i = 0; i < list.Count; i++)
17            {
18                hashAccumulator = (hashAccumulator << 5) + hashAccumulator ^ list[i];
19            }
20            return hashAccumulator + (hashSeed * 1566083941);
21        }
22
23        public static bool EqualTo(this IList<char> left, IList<char> right) =>
24            ↪ left.EqualTo(right, ContentEqualTo);
25
26        public static bool ContentEqualTo(this IList<char> left, IList<char> right)
27        {
28            for (var i = left.Count - 1; i >= 0; --i)
29            {
30                if (left[i] != right[i])
31                {

```

```

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

```

./Platform.Collections/Lists/IListComparer.cs

```

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

```

./Platform.Collections/Lists/IListEqualityComparer.cs

```

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

```

./Platform.Collections/Lists/IListExtensions.cs

```

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

```

```

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

```

./Platform.Collections/Segments/CharSegment.cs

```

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

```

```

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

```

./Platform.Collections/Segments/Segment.cs

```

1 using System;
2 using System.Collections;
3 using System.Collections.Generic;
4 using Platform.Collections.Lists;
5
6 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8 namespace Platform.Collections.Segments
9 {
10     public class Segment<T> : IEquatable<Segment<T>>, IList<T>
11     {
12         public IList<T> Base { get; }
13         public int Offset { get; }
14         public int Length { get; }
15
16         public Segment(IList<T> @base, int offset, int length)
17         {
18             Base = @base;
19             Offset = offset;
20             Length = length;
21         }
22
23         public override int GetHashCode() => this.GenerateHashCode();
24
25         public virtual bool Equals(Segment<T> other) => this.EqualTo(other);
26
27         public override bool Equals(object obj) => obj is Segment<T> other ? Equals(other) :
28             ↪ false;
29     }
30 }

```

```

29     #region IList
30
31     public T this[int i]
32     {
33         get => Base[Offset + i];
34         set => Base[Offset + i] = value;
35     }
36
37     public int Count => Length;
38
39     public bool IsReadOnly => true;
40
41     public int IndexOf(T item)
42     {
43         var index = Base.IndexOf(item);
44         if (index >= Offset)
45         {
46             var actualIndex = index - Offset;
47             if (actualIndex < Length)
48             {
49                 return actualIndex;
50             }
51         }
52         return -1;
53     }
54
55     public void Insert(int index, T item) => throw new NotSupportedException();
56
57     public void RemoveAt(int index) => throw new NotSupportedException();
58
59     public void Add(T item) => throw new NotSupportedException();
60
61     public void Clear() => throw new NotSupportedException();
62
63     public bool Contains(T item) => IndexOf(item) >= 0;
64
65     public void CopyTo(T[] array, int arrayIndex)
66     {
67         for (var i = 0; i < Length; i++)
68         {
69             array[arrayIndex++] = this[i];
70         }
71     }
72
73     public bool Remove(T item) => throw new NotSupportedException();
74
75     public IEnumerator<T> GetEnumerator()
76     {
77         for (var i = 0; i < Length; i++)
78         {
79             yield return this[i];
80         }
81     }
82
83     IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
84
85     #endregion
86 }
87 }

```

./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  }

```

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

```

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

```

```

9         protected override Segment<T> CreateSegment(IList<T> elements, int offset, int length)
            => new Segment<T>(elements, offset, length);
10     }
11 }

```

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

```

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

```

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

```

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

```

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

```

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

```

```

13         protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength) :
14             ↪ base(minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
15     }
16 }

```

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

```

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

```

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

```

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

```

```

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

```

```

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

```

```

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

```

```

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

```



```

21         public static T PeekOrDefault<T>(this IStack<T> stack) => stack.IsEmpty ? default :
           ↪ stack.Peek();
22     }
23 }

```

./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 }

```

./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 }

```

./Platform.Collections/StringExtensions.cs

```

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

```

```

40         {
41             if (@string[0] == charToTrim)
42             {
43                 return "";
44             }
45             else
46             {
47                 return @string;
48             }
49         }
50     else
51     {
52         var left = 0;
53         var right = @string.Length - 1;
54         if (@string[left] == charToTrim)
55         {
56             left++;
57         }
58         if (@string[right] == charToTrim)
59         {
60             right--;
61         }
62         return @string.Substring(left, right - left + 1);
63     }
64 }
65 else
66 {
67     return @string;
68 }
69 }
70 }
71 }

```

./Platform.Collections/Trees/Node.cs

```

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

```

```

46     }
47     return node;
48 }
49
50 public object GetChildValue(params object[] keys) => GetChild(keys)?.Value;
51
52 public Node AddChild(object key) => AddChild(key, new Node(null));
53
54 public Node AddChild(object key, object value) => AddChild(key, new Node(value));
55
56 public Node AddChild(object key, Node child)
57 {
58     ChildNodes.Add(key, child);
59     return child;
60 }
61
62 public Node SetChild(params object[] keys) => SetChildValue(null, keys);
63
64 public Node SetChild(object key) => SetChildValue(null, key);
65
66 public Node SetChildValue(object value, params object[] keys)
67 {
68     var node = this;
69     for (var i = 0; i < keys.Length; i++)
70     {
71         node = SetChildValue(value, keys[i]);
72     }
73     node.Value = value;
74     return node;
75 }
76
77 public Node SetChildValue(object value, object key)
78 {
79     if (!ChildNodes.TryGetValue(key, out Node child))
80     {
81         child = AddChild(key, value);
82     }
83     child.Value = value;
84     return child;
85 }
86 }
87 }

```

./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 BitAndTest()
28         {
29             TestToOperationsWithSameMeaning((x, y, w, v) =>
30             {
31                 x.VectorAnd(y);
32                 w.And(v);
33             });
34         }
35
36         [Fact]

```

```

37     public static void BitNotTest()
38     {
39         TestToOperationsWithSameMeaning((x, y, w, v) =>
40         {
41             x.VectorNot();
42             w.Not();
43         });
44     }
45
46     [Fact]
47     public static void BitOrTest()
48     {
49         TestToOperationsWithSameMeaning((x, y, w, v) =>
50         {
51             x.VectorOr(y);
52             w.Or(v);
53         });
54     }
55
56     private static void TestToOperationsWithSameMeaning(Action<BitString, BitString,
57     ↪ BitString, BitString> test)
58     {
59         const int n = 250;
60         var x = new BitString(n);
61         var y = new BitString(n);
62         x.SetRandomBits();
63         y.SetRandomBits();
64         var w = new BitString(x);
65         var v = new BitString(y);
66         test(x, y, w, v);
67         Assert.True(x.Equals(w));
68     }
69 }

```

./Platform.Collections.Tests/CharsSegmentTests.cs

```

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

```

./Platform.Collections.Tests/StringTests.cs

```

1  using Xunit;
2
3  namespace Platform.Collections.Tests
4  {
5      public static class StringTests
6      {
7          [Fact]
8          public static void CapitalizeFirstLetterTest()
9          {
10             var source1 = "hello";
11             var result1 = source1.CapitalizeFirstLetter();

```

```
12     Assert.Equal("Hello", result1);
13     var source2 = "Hello";
14     var result2 = source2.CapitalizeFirstLetter();
15     Assert.Equal("Hello", result2);
16     var source3 = " hello";
17     var result3 = source3.CapitalizeFirstLetter();
18     Assert.Equal(" Hello", result3);
19 }
20
21 [Fact]
22 public static void TrimSingleTest()
23 {
24     var source1 = "";
25     var result1 = source1.TrimSingle('\\');
26     Assert.Equal("", result1);
27     var source2 = " ";
28     var result2 = source2.TrimSingle('\\');
29     Assert.Equal("", result2);
30     var source3 = "'hello'";
31     var result3 = source3.TrimSingle('\\');
32     Assert.Equal("hello", result3);
33     var source4 = "hello'";
34     var result4 = source4.TrimSingle('\\');
35     Assert.Equal("hello", result4);
36     var source5 = "'hello";
37     var result5 = source5.TrimSingle('\\');
38     Assert.Equal("hello", result5);
39 }
40 }
41 }
```

Index

- ./Platform.Collections.Tests/BitStringTests.cs, 27
- ./Platform.Collections.Tests/CharsSegmentTests.cs, 28
- ./Platform.Collections.Tests/StringTests.cs, 28
- ./Platform.Collections/Arrays/ArrayFiller[TElement, TReturnConstant].cs, 1
- ./Platform.Collections/Arrays/ArrayFiller[TElement].cs, 1
- ./Platform.Collections/Arrays/ArrayPool.cs, 1
- ./Platform.Collections/Arrays/ArrayPool[T].cs, 2
- ./Platform.Collections/Arrays/ArrayString.cs, 3
- ./Platform.Collections/Arrays/CharArrayExtensions.cs, 3
- ./Platform.Collections/Arrays/GenericArrayExtensions.cs, 4
- ./Platform.Collections/BitString.cs, 4
- ./Platform.Collections/BitStringExtensions.cs, 14
- ./Platform.Collections/Concurrent/ConcurrentQueueExtensions.cs, 14
- ./Platform.Collections/Concurrent/ConcurrentStackExtensions.cs, 15
- ./Platform.Collections/EnsureExtensions.cs, 15
- ./Platform.Collections/ICollectionExtensions.cs, 16
- ./Platform.Collections/IDictionaryExtensions.cs, 16
- ./Platform.Collections/ISetExtensions.cs, 17
- ./Platform.Collections/Lists/CharListExtensions.cs, 17
- ./Platform.Collections/Lists/IListComparer.cs, 18
- ./Platform.Collections/Lists/IListEqualityComparer.cs, 18
- ./Platform.Collections/Lists/IListExtensions.cs, 18
- ./Platform.Collections/Segments/CharSegment.cs, 19
- ./Platform.Collections/Segments/Segment.cs, 20
- ./Platform.Collections/Segments/Walkers/AllSegmentsWalkerBase.cs, 21
- ./Platform.Collections/Segments/Walkers/AllSegmentsWalkerBase[T, TSegment].cs, 22
- ./Platform.Collections/Segments/Walkers/AllSegmentsWalkerBase[T].cs, 21
- ./Platform.Collections/Segments/Walkers/AllSegmentsWalkerExtensions.cs, 22
- ./Platform.Collections/Segments/Walkers/DictionaryBasedDuplicateSegmentsWalkerBase[T, Segment].cs, 23
- ./Platform.Collections/Segments/Walkers/DictionaryBasedDuplicateSegmentsWalkerBase[T].cs, 22
- ./Platform.Collections/Segments/Walkers/DuplicateSegmentsWalkerBase[T, TSegment].cs, 24
- ./Platform.Collections/Segments/Walkers/DuplicateSegmentsWalkerBase[T].cs, 23
- ./Platform.Collections/Stacks/DefaultStack.cs, 24
- ./Platform.Collections/Stacks/IStack.cs, 24
- ./Platform.Collections/Stacks/IStackExtensions.cs, 24
- ./Platform.Collections/Stacks/IStackFactory.cs, 25
- ./Platform.Collections/Stacks/StackExtensions.cs, 25
- ./Platform.Collections/StringExtensions.cs, 25
- ./Platform.Collections/Trees/Node.cs, 26