

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, new Range<long>(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/BitString.cs

```

1 using System;
2 using System.Collections.Generic;
3 using System.Runtime.CompilerServices;
4 using Platform.Exceptions;
5 using Platform.Ranges;
6
7 // ReSharper disable ForCanBeConvertedToForeach
8 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Collections
11 {
12     /// <remarks>
13     /// А что если хранить карту значений, где каждый бит будет означать присутствует ли блок в
→ 64 бит в массиве значений.
14     /// 64 бита по 0 бит, будут означать отсутствие 64-х блоков по 64 бита. Т.е. упаковка 512
→ байт в 8 байт.
15     /// Подобный принцип можно применять и к 64-ём блокам и т.п. По сути это карта значений. С
→ помощью которой можно быстро
16     /// проверять есть ли значения непосредственно далее (ниже по уровню).
17     /// Или как таблица виртуальной памяти где номер блока означает его присутствие и адрес.
18     /// </remarks>
19     public class BitString
20     {
21         private static readonly byte[] [] _bitsSetIn16Bits;
22         private long[] _array;
23         private long _length;
24         private long _minPositiveWord;
25         private long _maxPositiveWord;
26
27         public bool this[long index]
28         {
29             get => Get(index);
30             set => Set(index, value);
31         }
32
33         public long Length
34         {
35             get => _length;
36             set
37             {
38                 if (_length == value)
39                 {
40                     return;
41                 }

```

```

42     Ensure.Always.ArgumentInRange(value, new Range<long>(0, long.MaxValue),
43         ↳ 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             // What is going on here?
60             var mask = (int)(_length % 64);
61             if (mask > 0)
62             {
63                 _array[oldWords - 1] &= (1L << mask) - 1;
64             }
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 #region Constructors
76
77 static BitString()
78 {
79     _bitsSetIn16Bits = new byte[65536][];
80     int i, c, k;
81     byte bitIndex;
82     for (i = 0; i < 65536; i++)
83     {
84         // Calculating size of array (number of positive bits)
85         for (c = 0, k = 1; k <= 65536; k <= 1)
86         {
87             if ((i & k) == k)
88             {
89                 c++;
90             }
91         }
92         var array = new byte[c];
93         // Adding positive bits indices into array
94         for (bitIndex = 0, c = 0, k = 1; k <= 65536; k <= 1)
95         {
96             if ((i & k) == k)
97             {
98                 array[c++] = bitIndex;
99             }
100             bitIndex++;
101         }
102         _bitsSetIn16Bits[i] = array;
103     }
104 }
105
106 public BitString(BitString other)
107 {
108     Ensure.Always.ArgumentNotNull(other, nameof(other));
109     _length = other._length;
110     _array = new long[GetWordsCountFromIndex(_length)];
111     _minPositiveWord = other._minPositiveWord;
112     _maxPositiveWord = other._maxPositiveWord;
113     Array.Copy(other._array, _array, _array.LongLength);
114 }
115
116 public BitString(long length)
117 {
118     Ensure.Always.ArgumentInRange(length, GetValidLengthRange(), nameof(length));
119     _length = length;

```

```

120     _array = new long[GetWordsCountFromIndex(_length)];
121     MarkBordersAsAllBitsReset();
122 }
123
124 public BitString(long length, bool defaultValue)
125     : this(length)
126 {
127     if (defaultValue)
128     {
129         SetAll();
130     }
131 }
132
133 #endregion
134
135 public BitString Not()
136 {
137     var words = GetWordsCountFromIndex(_length);
138     for (long i = 0; i < words; i++)
139     {
140         _array[i] = ~_array[i];
141         RefreshBordersByWord(i);
142     }
143     return this;
144 }
145
146 public BitString And(BitString other)
147 {
148     EnsureBitStringHasTheSameSize(other, nameof(other));
149     GetCommonInnerBorders(this, other, out long from, out long to);
150     var otherArray = other._array;
151     for (var i = from; i <= to; i++)
152     {
153         _array[i] &= otherArray[i];
154         RefreshBordersByWord(i);
155     }
156     return this;
157 }
158
159 public BitString Or(BitString other)
160 {
161     EnsureBitStringHasTheSameSize(other, nameof(other));
162     GetCommonOuterBorders(this, other, out long from, out long to);
163     for (var i = from; i <= to; i++)
164     {
165         _array[i] |= other._array[i];
166         RefreshBordersByWord(i);
167     }
168     return this;
169 }
170
171 public BitString Xor(BitString other)
172 {
173     EnsureBitStringHasTheSameSize(other, nameof(other));
174     GetCommonOuterBorders(this, other, out long from, out long to);
175     for (var i = from; i <= to; i++)
176     {
177         _array[i] ^= other._array[i];
178         RefreshBordersByWord(i);
179     }
180     return this;
181 }
182
183 private void RefreshBordersByWord(long wordIndex)
184 {
185     if (_array[wordIndex] == 0)
186     {
187         if (wordIndex == _minPositiveWord && wordIndex != _array.LongLength - 1)
188         {
189             _minPositiveWord++;
190         }
191         if (wordIndex == _maxPositiveWord && wordIndex != 0)
192         {
193             _maxPositiveWord--;
194         }
195     }
196     else
197     {
198         if (wordIndex < _minPositiveWord)

```

```

199         {
200             _minPositiveWord = wordIndex;
201         }
202         if (wordIndex > _maxPositiveWord)
203         {
204             _maxPositiveWord = wordIndex;
205         }
206     }
207 }
208
209 public bool TryShrinkBorders()
210 {
211     GetBorders(out long from, out long to);
212     while (from <= to && _array[from] == 0)
213     {
214         from++;
215     }
216     if (from > to)
217     {
218         MarkBordersAsAllBitsReset();
219         return true;
220     }
221     while (to >= from && _array[to] == 0)
222     {
223         to--;
224     }
225     if (to < from)
226     {
227         MarkBordersAsAllBitsReset();
228         return true;
229     }
230     var bordersUpdated = from != _minPositiveWord || to != _maxPositiveWord;
231     if (bordersUpdated)
232     {
233         SetBorders(from, to);
234     }
235     return bordersUpdated;
236 }
237
238 [MethodImpl(MethodImplOptions.AggressiveInlining)]
239 public bool Get(long index)
240 {
241     Ensure.Always.ArgumentInRange(index, GetValidIndexRange(), nameof(index));
242     return (_array[GetWordIndexFromIndex(index)] & GetBitMaskFromIndex(index)) != 0;
243 }
244
245 [MethodImpl(MethodImplOptions.AggressiveInlining)]
246 public void Set(long index, bool value)
247 {
248     if (value)
249     {
250         Set(index);
251     }
252     else
253     {
254         Reset(index);
255     }
256 }
257
258 [MethodImpl(MethodImplOptions.AggressiveInlining)]
259 public void Set(long index)
260 {
261     Ensure.Always.ArgumentInRange(index, GetValidIndexRange(), nameof(index));
262     var wordIndex = GetWordIndexFromIndex(index);
263     var mask = GetBitMaskFromIndex(index);
264     _array[wordIndex] |= mask;
265     RefreshBordersByWord(wordIndex);
266 }
267
268 [MethodImpl(MethodImplOptions.AggressiveInlining)]
269 public void Reset(long index)
270 {
271     Ensure.Always.ArgumentInRange(index, GetValidIndexRange(), nameof(index));
272     var wordIndex = GetWordIndexFromIndex(index);
273     var mask = GetBitMaskFromIndex(index);
274     _array[wordIndex] &= ~mask;
275     RefreshBordersByWord(wordIndex);
276 }
277

```

```

278 public bool Add(long index)
279 {
280     var wordIndex = GetWordIndexFromIndex(index);
281     var mask = GetBitMaskFromIndex(index);
282     if ((_array[wordIndex] & mask) == 0)
283     {
284         _array[wordIndex] |= mask;
285         RefreshBordersByWord(wordIndex);
286         return true;
287     }
288     else
289     {
290         return false;
291     }
292 }
293
294 public void SetAll(bool value)
295 {
296     if (value)
297     {
298         SetAll();
299     }
300     else
301     {
302         ResetAll();
303     }
304 }
305
306 public void SetAll()
307 {
308     const long fillValue = unchecked((long)0xffffffffffffffff);
309     var words = GetWordsCountFromIndex(_length);
310     for (var i = 0; i < words; i++)
311     {
312         _array[i] = fillValue;
313     }
314     MarkBordersAsAllBitsSet();
315 }
316
317 public void ResetAll()
318 {
319     const long fillValue = 0;
320     GetBorders(out long from, out long to);
321     for (var i = from; i <= to; i++)
322     {
323         _array[i] = fillValue;
324     }
325     MarkBordersAsAllBitsReset();
326 }
327
328 public List<long> GetSetIndices()
329 {
330     var result = new List<long>();
331     GetBorders(out long from, out long to);
332     for (var i = from; i <= to; i++)
333     {
334         var word = _array[i];
335         if (word != 0)
336         {
337             AppendAllSetBitIndices(result, i, word);
338         }
339     }
340     return result;
341 }
342
343 public List<ulong> GetSetUInt64Indices()
344 {
345     var result = new List<ulong>();
346     GetBorders(out ulong from, out ulong to);
347     for (var i = from; i <= to; i++)
348     {
349         var word = _array[i];
350         if (word != 0)
351         {
352             AppendAllSetBitIndices(result, i, word);
353         }
354     }
355     return result;
356 }

```



```

357
358 public long GetFirstSetBitIndex()
359 {
360     var i = _minPositiveWord;
361     var word = _array[i];
362     if (word != 0)
363     {
364         return GetFirstSetBitForWord(i, word);
365     }
366     return -1;
367 }
368
369 public long GetLastSetBitIndex()
370 {
371     var i = _maxPositiveWord;
372     var word = _array[i];
373     if (word != 0)
374     {
375         return GetLastSetBitForWord(i, word);
376     }
377     return -1;
378 }
379
380 public long CountSetBits()
381 {
382     var total = 0L;
383     GetBorders(out long from, out long to);
384     for (var i = from; i <= to; i++)
385     {
386         var word = _array[i];
387         if (word != 0)
388         {
389             total += CountSetBitsForWord(word);
390         }
391     }
392     return total;
393 }
394
395 public bool HaveCommonBits(BitString other)
396 {
397     EnsureBitStringHasTheSameSize(other, nameof(other));
398     GetCommonInnerBorders(this, other, out long from, out long to);
399     var otherArray = other._array;
400     for (var i = from; i <= to; i++)
401     {
402         var left = _array[i];
403         var right = otherArray[i];
404         if (left != 0 && right != 0 && (left & right) != 0)
405         {
406             return true;
407         }
408     }
409     return false;
410 }
411
412 public long CountCommonBits(BitString other)
413 {
414     EnsureBitStringHasTheSameSize(other, nameof(other));
415     GetCommonInnerBorders(this, other, out long from, out long to);
416     var total = 0L;
417     var otherArray = other._array;
418     for (var i = from; i <= to; i++)
419     {
420         var left = _array[i];
421         var right = otherArray[i];
422         var combined = left & right;
423         if (combined != 0)
424         {
425             total += CountSetBitsForWord(combined);
426         }
427     }
428     return total;
429 }
430
431 public List<long> GetCommonIndices(BitString other)
432 {
433     EnsureBitStringHasTheSameSize(other, nameof(other));
434     GetCommonInnerBorders(this, other, out long from, out long to);
435     var result = new List<long>();

```

```

436     var otherArray = other._array;
437     for (var i = from; i <= to; i++)
438     {
439         var left = _array[i];
440         var right = otherArray[i];
441         var combined = left & right;
442         if (combined != 0)
443         {
444             AppendAllSetBitIndices(result, i, combined);
445         }
446     }
447     return result;
448 }
449
450 public List<ulong> GetCommonUInt64Indices(BitString other)
451 {
452     EnsureBitStringHasTheSameSize(other, nameof(other));
453     GetCommonBorders(this, other, out ulong from, out ulong to);
454     var result = new List<ulong>();
455     var otherArray = other._array;
456     for (var i = from; i <= to; i++)
457     {
458         var left = _array[i];
459         var right = otherArray[i];
460         var combined = left & right;
461         if (combined != 0)
462         {
463             AppendAllSetBitIndices(result, i, combined);
464         }
465     }
466     return result;
467 }
468
469 public long GetFirstCommonBitIndex(BitString other)
470 {
471     EnsureBitStringHasTheSameSize(other, nameof(other));
472     GetCommonInnerBorders(this, other, out long from, out long to);
473     var otherArray = other._array;
474     for (var i = from; i <= to; i++)
475     {
476         var left = _array[i];
477         var right = otherArray[i];
478         var combined = left & right;
479         if (combined != 0)
480         {
481             return GetFirstSetBitForWord(i, combined);
482         }
483     }
484     return -1;
485 }
486
487 public long GetLastCommonBitIndex(BitString other)
488 {
489     EnsureBitStringHasTheSameSize(other, nameof(other));
490     GetCommonInnerBorders(this, other, out long from, out long to);
491     var otherArray = other._array;
492     for (var i = to; i >= from; i--)
493     {
494         var left = _array[i];
495         var right = otherArray[i];
496         var combined = left & right;
497         if (combined != 0)
498         {
499             return GetLastSetBitForWord(i, combined);
500         }
501     }
502     return -1;
503 }
504
505 [MethodImpl(MethodImplOptions.AggressiveInlining)]
506 private void EnsureBitStringHasTheSameSize(BitString other, string argumentName)
507 {
508     Ensure.Always.ArgumentNotNull(other, argumentName);
509     if (_length != other._length)
510     {
511         throw new ArgumentException("Bit string must be the same size.", argumentName);
512     }
513 }
514

```

```

515 [MethodImpl(MethodImplOptions.AggressiveInlining)]
516 private void MarkBordersAsAllBitsReset() => SetBorders(_array.LongLength - 1, 0);
517
518 [MethodImpl(MethodImplOptions.AggressiveInlining)]
519 private void MarkBordersAsAllBitsSet() => SetBorders(0, _array.LongLength - 1);
520
521 [MethodImpl(MethodImplOptions.AggressiveInlining)]
522 private void GetBorders(out long from, out long to)
523 {
524     from = _minPositiveWord;
525     to = _maxPositiveWord;
526 }
527
528 [MethodImpl(MethodImplOptions.AggressiveInlining)]
529 private void GetBorders(out ulong from, out ulong to)
530 {
531     from = (ulong)_minPositiveWord;
532     to = (ulong)_maxPositiveWord;
533 }
534
535 [MethodImpl(MethodImplOptions.AggressiveInlining)]
536 private void SetBorders(long from, long to)
537 {
538     _minPositiveWord = from;
539     _maxPositiveWord = to;
540 }
541
542 [MethodImpl(MethodImplOptions.AggressiveInlining)]
543 private Range<long> GetValidIndexRange() => new Range<long>(0, _length - 1);
544
545 [MethodImpl(MethodImplOptions.AggressiveInlining)]
546 private static Range<long> GetValidLengthRange() => new Range<long>(0, long.MaxValue);
547
548 [MethodImpl(MethodImplOptions.AggressiveInlining)]
549 private static void AppendAllSetBitIndices(List<ulong> result, ulong wordIndex, long
    ↪ wordValue)
550 {
551     GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
    ↪ bits32to47, out byte[] bits48to63);
552     AppendAllSetIndices(result, wordIndex, bits00to15, bits16to31, bits32to47,
    ↪ bits48to63);
553 }
554
555 [MethodImpl(MethodImplOptions.AggressiveInlining)]
556 private static void AppendAllSetBitIndices(List<long> result, long wordIndex, long
    ↪ wordValue)
557 {
558     GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
    ↪ bits32to47, out byte[] bits48to63);
559     AppendAllSetIndices(result, wordIndex, bits00to15, bits16to31, bits32to47,
    ↪ bits48to63);
560 }
561
562 [MethodImpl(MethodImplOptions.AggressiveInlining)]
563 private static long CountSetBitsForWord(long word)
564 {
565     GetBits(word, out byte[] bits00to15, out byte[] bits16to31, out byte[] bits32to47,
    ↪ out byte[] bits48to63);
566     return bits00to15.LongLength + bits16to31.LongLength + bits32to47.LongLength +
    ↪ bits48to63.LongLength;
567 }
568
569 [MethodImpl(MethodImplOptions.AggressiveInlining)]
570 private static long GetFirstSetBitForWord(long wordIndex, long wordValue)
571 {
572     GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
    ↪ bits32to47, out byte[] bits48to63);
573     return GetFirstSetBit(wordIndex, bits00to15, bits16to31, bits32to47, bits48to63);
574 }
575
576 [MethodImpl(MethodImplOptions.AggressiveInlining)]
577 private static long GetLastSetBitForWord(long wordIndex, long wordValue)
578 {
579     GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
    ↪ bits32to47, out byte[] bits48to63);
580     return GetLastSetBit(wordIndex, bits00to15, bits16to31, bits32to47, bits48to63);
581 }
582

```

```

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

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

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

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

```

```

657 private static void GetBits(long word, out byte[] bits00to15, out byte[] bits16to31, out
    ↳ byte[] bits32to47, out byte[] bits48to63)
658 {
659     bits00to15 = _bitsSetIn16Bits[word & 0xffffu];
660     bits16to31 = _bitsSetIn16Bits[(word >> 16) & 0xffffu];
661     bits32to47 = _bitsSetIn16Bits[(word >> 32) & 0xffffu];
662     bits48to63 = _bitsSetIn16Bits[(word >> 48) & 0xffffu];
663 }
664
665 [MethodImpl(MethodImplOptions.AggressiveInlining)]
666 public static void GetCommonInnerBorders(BitString left, BitString right, out long from,
    ↳ out long to)
667 {
668     from = Math.Max(left._minPositiveWord, right._minPositiveWord);
669     to = Math.Min(left._maxPositiveWord, right._maxPositiveWord);
670 }
671
672 [MethodImpl(MethodImplOptions.AggressiveInlining)]
673 public static void GetCommonOuterBorders(BitString left, BitString right, out long from,
    ↳ out long to)
674 {
675     from = Math.Min(left._minPositiveWord, right._minPositiveWord);
676     to = Math.Max(left._maxPositiveWord, right._maxPositiveWord);
677 }
678
679 [MethodImpl(MethodImplOptions.AggressiveInlining)]
680 public static void GetCommonBorders(BitString left, BitString right, out ulong from, out
    ↳ ulong to)
681 {
682     from = (ulong)Math.Max(left._minPositiveWord, right._minPositiveWord);
683     to = (ulong)Math.Min(left._maxPositiveWord, right._maxPositiveWord);
684 }
685
686 [MethodImpl(MethodImplOptions.AggressiveInlining)]
687 public static long GetWordsCountFromIndex(long index) => (index + 63) / 64;
688
689 [MethodImpl(MethodImplOptions.AggressiveInlining)]
690 public static long GetWordIndexFromIndex(long index) => index >> 6;
691
692 [MethodImpl(MethodImplOptions.AggressiveInlining)]
693 public static long GetBitMaskFromIndex(long index) => 1L << (int)(index & 63);
694 }
695 }

```

./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         public static T PeekOrDefault<T>(this ConcurrentStack<T> stack) => stack.TryPeek(out T
13         ↪ value) ? value : default;
14     }
15 }

```

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

```

```

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 }

```

./Platform.Collections/ICollectionExtensions.cs

```

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

```

./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 }

```

./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/CharIListExtensions.cs

```

1 using System.Collections.Generic;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Collections.Lists
6 {
7     public static class CharIListExtensions
8     {
9         /// <remarks>
10        /// Based on https://github.com/Microsoft/referencesource/blob/3b1eaf5203992df69de44c783a3eda37d3d4cd10/mscorlib/system/string.cs#L833
11        ↪
12        /// </remarks>
13        public static unsafe int GenerateHashCode(this IList<char> list)
14        {
15            var hashSeed = 5381;
16            var hashAccumulator = hashSeed;
17            for (var i = 0; i < list.Count; i++)
18            {
19                hashAccumulator = (hashAccumulator << 5) + hashAccumulator ^ list[i];
20            }
21            return hashAccumulator + (hashSeed * 1566083941);
22        }
23
24        public static bool EqualTo(this IList<char> left, IList<char> right) =>
25            ↪ left.EqualTo(right, ContentEqualTo);
26
27        public static bool ContentEqualTo(this IList<char> left, IList<char> right)
28        {
29            for (var i = left.Count - 1; i >= 0; --i)
30            {
31                if (left[i] != right[i])
32                {
33                    return false;
34                }
35            }
36            return true;
37        }
38    }
39 }

```

./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 }

```



```
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;
44             for (var i = left.Count - 1; i >= 0; --i)
45             {
46                 if (!equalityComparer.Equals(left[i], right[i]))
47                 {
48                     return false;
49                 }
50             }
51             return true;
52         }
53
54         public static T[] ToArray<T>(this IList<T> list, Func<T, bool> predicate)
55         {
56             if (list == null)
57             {
58                 return null;
59             }
60             var result = new List<T>(list.Count);
61             for (var i = 0; i < list.Count; i++)
62             {
63                 if (predicate(list[i]))
64                 {
65                     result.Add(list[i]);
66                 }
67             }
68             return result.ToArray();
69         }
70
71         public static T[] ToArray<T>(this IList<T> list)
72         {
73             var array = new T[list.Count];
74             list.CopyTo(array, 0);
75             return array;
76         }
77     }
78 }
```

```

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         }
43     }
44 }

```

```

39         }
40     }
41     return this.EqualTo(other, contentEqualityComparer);
42 }
43
44 public static implicit operator string(CharSegment segment)
45 {
46     if (!(segment.Base is char[] array))
47     {
48         array = segment.Base.ToArray();
49     }
50     return new string(array, segment.Offset, segment.Length);
51 }
52
53 public override string ToString() => this;
54 }
55 }

```

./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) :
            ↪ false;
28
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)
10             ↪ => new Segment<T>(elements, offset, length);
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) =>
13             ↪ _minimumStringSegmentLength = minimumStringSegmentLength;
14
15         protected AllSegmentsWalkerBase() : this(DefaultMinimumStringSegmentLength) { }
16
17         public virtual void WalkAll(IList<T> elements)
18         {
19             for (int offset = 0, maxOffset = elements.Count - _minimumStringSegmentLength;
20                 ↪ offset <= maxOffset; offset++)
21             {
22                 for (int length = _minimumStringSegmentLength, maxLength = elements.Count -
23                     ↪ offset; length <= maxLength; length++)
24                 {
25                     Iteration(CreateSegment(elements, offset, length));
26                 }
27             }
28         }
29     }

```

```

23         }
24     }
25 }
26
27     protected abstract TSegment CreateSegment(IList<T> elements, int offset, int length);
28
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) =>
8              ↪ walker.WalkAll(@string.ToCharArray());
9          public static void WalkAll<TSegment>(this AllSegmentsWalkerBase<char> walker,
10             ↪ string @string) where TSegment : Segment<char> =>
11             ↪ walker.WalkAll(@string.ToCharArray());
12     }
13 }

```

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

```

1  using System.Collections.Generic;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Collections.Segments.Walkers
6  {
7      public abstract class DictionaryBasedDuplicateSegmentsWalkerBase<T> :
8          ↪ DictionaryBasedDuplicateSegmentsWalkerBase<T, Segment<T>>
9      {
10         protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<Segment<T>, long>
11             ↪ dictionary, int minimumStringSegmentLength, bool resetDictionaryOnEachWalk) :
12             ↪ base(dictionary, minimumStringSegmentLength, resetDictionaryOnEachWalk) { }
13         protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<Segment<T>, long>
14             ↪ dictionary, int minimumStringSegmentLength) : base(dictionary,
15             ↪ minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
16         protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<Segment<T>, long>
17             ↪ dictionary) : base(dictionary, DefaultMinimumStringSegmentLength,
18             ↪ DefaultResetDictionaryOnEachWalk) { }
19         protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength,
20             ↪ bool resetDictionaryOnEachWalk) : base(minimumStringSegmentLength,
21             ↪ resetDictionaryOnEachWalk) { }
22         protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength) :
23             ↪ base(minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
24         protected DictionaryBasedDuplicateSegmentsWalkerBase() :
25             ↪ base(DefaultMinimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
26     }
27 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Collections.Segments.Walkers
7  {
8      public abstract class DictionaryBasedDuplicateSegmentsWalkerBase<T, TSegment> :
9          ↪ DuplicateSegmentsWalkerBase<T, TSegment>
10         where TSegment : Segment<T>
11     {
12         public static readonly bool DefaultResetDictionaryOnEachWalk;
13
14         private readonly bool _resetDictionaryOnEachWalk;
15         protected IDictionary<TSegment, long> Dictionary;
16
17         protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<TSegment, long>
18             ↪ dictionary, int minimumStringSegmentLength, bool resetDictionaryOnEachWalk)
19             : base(minimumStringSegmentLength)
20         {
21             Dictionary = dictionary;
22             _resetDictionaryOnEachWalk = resetDictionaryOnEachWalk;
23         }
24     }
25 }

```

```

22
23     protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<TSegment, long>
    ↪ dictionary, int minimumStringSegmentLength) : this(dictionary,
    ↪ minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
24
25     protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<TSegment, long>
    ↪ dictionary) : this(dictionary, DefaultMinimumStringSegmentLength,
    ↪ DefaultResetDictionaryOnEachWalk) { }
26
27     protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength,
    ↪ bool resetDictionaryOnEachWalk) : this(resetDictionaryOnEachWalk ? null : new
    ↪ Dictionary<TSegment, long>(), minimumStringSegmentLength, resetDictionaryOnEachWalk)
    ↪ { }
28
29     protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength) :
    ↪ this(minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
30
31     protected DictionaryBasedDuplicateSegmentsWalkerBase() :
    ↪ this(DefaultMinimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
32
33     public override void WalkAll(ICollection<T> elements)
34     {
35         if (_resetDictionaryOnEachWalk)
36         {
37             var capacity = Math.Ceiling(Math.Pow(elements.Count, 2) / 2);
38             Dictionary = new Dictionary<TSegment, long>((int)capacity);
39         }
40         base.WalkAll(elements);
41     }
42
43     protected override long GetSegmentFrequency(TSegment segment) =>
    ↪ Dictionary.GetOrDefault(segment);
44
45     protected override void SetSegmentFrequency(TSegment segment, long frequency) =>
    ↪ Dictionary[segment] = frequency;
46 }
47 }

```

./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 }

```

./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 :
19        ↪ stack.Pop();
20
21        [MethodImpl(MethodImplOptions.AggressiveInlining)]
22        public static T PeekOrDefault<T>(this IStack<T> stack) => stack.IsEmpty ? default :
23        ↪ stack.Peek();
24    }
25 }
```

./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() :
12        ↪ default;
13    }
14 }
```

```

12         [MethodImpl(MethodImplOptions.AggressiveInlining)]
13         public static T PeekOrDefault<T>(this Stack<T> stack) => stack.Count > 0 ? stack.Peek()
14         ↪ : 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 str)
11         {
12             if (string.IsNullOrEmpty(str))
13             {
14                 return str;
15             }
16             var chars = str.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 str;
23                 }
24                 if (category == UnicodeCategory.LowercaseLetter)
25                 {
26                     chars[i] = char.ToUpper(chars[i]);
27                     return new string(chars);
28                 }
29             }
30             return str;
31         }
32
33         public static string Truncate(this string str, int maxLength) =>
34         ↪ string.IsNullOrEmpty(str) ? str : str.Substring(0, Math.Min(str.Length, maxLength));
35     }
36 }

```

./Platform.Collections/Trees/Node.cs

```

1  using System.Collections.Generic;
2
3  // ReSharper disable ForCanBeConvertedToForeach
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Collections.Trees
7  {
8      public class Node
9      {
10         private Dictionary<object, Node> _childNodes;
11
12         public object Value { get; set; }
13
14         public Dictionary<object, Node> ChildNodes => _childNodes ?? (_childNodes = new
15         ↪ Dictionary<object, Node>());
16
17         public Node this[object key]
18         {
19             get
20             {
21                 var child = GetChild(key);
22                 if (child == null)
23                 {
24                     child = AddChild(key);
25                 }
26                 return child;
27             }
28             set => SetChildValue(value, key);
29         }
30
31         public Node(object value) => Value = value;
32
33         public Node() : this(null) { }
34     }
35 }

```



```

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

```

./Platform.Collections.Tests/BitStringTests.cs

```

1  using System.Collections;
2  using Xunit;
3  using Platform.Random;
4
5  namespace Platform.Collections.Tests
6  {
7      public static class BitStringTests
8      {
9          [Fact]
10         public static void BitGetSetTest()
11         {
12             const int n = 250;
13             var bitArray = new BitArray(n);
14             var bitString = new BitString(n);
15             for (var i = 0; i < n; i++)
16             {
17                 var value = RandomHelpers.Default.NextBoolean();
18                 bitArray.Set(i, value);
19                 bitString.Set(i, value);
20                 Assert.Equal(value, bitArray.Get(i));
21                 Assert.Equal(value, bitString.Get(i));
22             }
23         }
24     }
25 }

```

```
24     }
25 }
```

./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 }
```

Index

- ./Platform.Collections.Tests/BitStringTests.cs, 25
- ./Platform.Collections.Tests/CharsSegmentTests.cs, 26
- ./Platform.Collections.Tests/StringTests.cs, 26
- ./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/BitString.cs, 4
- ./Platform.Collections/Concurrent/ConcurrentQueueExtensions.cs, 13
- ./Platform.Collections/Concurrent/ConcurrentStackExtensions.cs, 13
- ./Platform.Collections/EnsureExtensions.cs, 14
- ./Platform.Collections/ICollectionExtensions.cs, 15
- ./Platform.Collections/IDictionaryExtensions.cs, 15
- ./Platform.Collections/ISetExtensions.cs, 15
- ./Platform.Collections/Lists/CharListExtensions.cs, 16
- ./Platform.Collections/Lists/IListComparer.cs, 16
- ./Platform.Collections/Lists/IListEqualityComparer.cs, 16
- ./Platform.Collections/Lists/IListExtensions.cs, 17
- ./Platform.Collections/Segments/CharSegment.cs, 18
- ./Platform.Collections/Segments/Segment.cs, 19
- ./Platform.Collections/Segments/Walkers/AllSegmentsWalkerBase.cs, 20
- ./Platform.Collections/Segments/Walkers/AllSegmentsWalkerBase[T, TSegment].cs, 20
- ./Platform.Collections/Segments/Walkers/AllSegmentsWalkerBase[T].cs, 20
- ./Platform.Collections/Segments/Walkers/AllSegmentsWalkerExtensions.cs, 21
- ./Platform.Collections/Segments/Walkers/DictionaryBasedDuplicateSegmentsWalkerBase[T, Segment].cs, 21
- ./Platform.Collections/Segments/Walkers/DictionaryBasedDuplicateSegmentsWalkerBase[T].cs, 21
- ./Platform.Collections/Segments/Walkers/DuplicateSegmentsWalkerBase[T, TSegment].cs, 22
- ./Platform.Collections/Segments/Walkers/DuplicateSegmentsWalkerBase[T].cs, 22
- ./Platform.Collections/Stacks/DefaultStack.cs, 22
- ./Platform.Collections/Stacks/IStack.cs, 23
- ./Platform.Collections/Stacks/IStackExtensions.cs, 23
- ./Platform.Collections/Stacks/IStackFactory.cs, 23
- ./Platform.Collections/Stacks/StackExtensions.cs, 23
- ./Platform.Collections/StringExtensions.cs, 24
- ./Platform.Collections/Trees/Node.cs, 24