

## LinksPlatform's Platform.Collections Class Library

### ./Arrays/ArrayFiller[TElement].cs

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

### ./Arrays/ArrayFiller[TElement, TReturnConstant].cs

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

### ./Arrays/ArrayPool.cs

```
1 using System.Runtime.CompilerServices;
2
3 namespace Platform.Collections.Arrays
4 {
5     public static class ArrayPool
6     {
7         public static readonly int DefaultSizesAmount = 512;
8         public static readonly int DefaultMaxArraysPerSize = 32;
```

```

9
10     [MethodImpl(MethodImplOptions.AggressiveInlining)]
11     public static T[] Allocate<T>(long size) => ArrayPool<T>.ThreadInstance.Allocate(size);
12
13     [MethodImpl(MethodImplOptions.AggressiveInlining)]
14     public static void Free<T>(T[] array) => ArrayPool<T>.ThreadInstance.Free(array);
15 }
16 }

```

./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 namespace Platform.Collections.Arrays
9 {
10     /// <remarks>
11     /// Original idea from
12     /// ↪ http://geekswithblogs.net/blackrob/archive/2014/12/18/array-pooling-in-csharp.aspx
13     /// </remarks>
14     public class ArrayPool<T>
15     {
16         public static readonly T[] Empty = new T[0];
17
18         // May be use Default class for that later.
19         [ThreadStatic]
20         internal static ArrayPool<T> _threadInstance;
21         internal static ArrayPool<T> ThreadInstance { get => _threadInstance ?? (_threadInstance
22             ↪ = new ArrayPool<T>()); }
23
24         private readonly int _maxArraysPerSize;
25         private readonly Dictionary<int, Stack<T[]>> _pool = new Dictionary<int,
26             ↪ Stack<T[]>>(ArrayPool.DefaultSizesAmount);
27
28         public ArrayPool(int maxArraysPerSize) => _maxArraysPerSize = maxArraysPerSize;
29
30         public ArrayPool() : this(ArrayPool.DefaultMaxArraysPerSize) { }
31
32         public Disposable<T[]> AllocateDisposable(long size) => (Allocate(size), Free);
33
34         public Disposable<T[]> Resize(Disposable<T[]> source, long size)
35         {
36             var destination = AllocateDisposable(size);
37             T[] sourceArray = source;
38             T[] destinationArray = destination;
39             Array.Copy(sourceArray, destinationArray, size < sourceArray.Length ? (int)size :
40                 ↪ sourceArray.Length);
41             source.Dispose();
42             return destination;
43         }
44
45         public virtual void Clear() => _pool.Clear();
46
47         public virtual T[] Allocate(long size)
48         {
49             Ensure.Always.ArgumentInRange(size, new Range<long>(0, int.MaxValue));
50             return size == 0 ? Empty : _pool.GetOrDefault((int)size)?.PopOrDefault() ?? new
51                 ↪ T[size];
52         }
53
54         public virtual void Free(T[] array)
55         {
56             Ensure.Always.ArgumentNotNull(array, nameof(array));
57             if (array.Length == 0)
58             {
59                 return;
60             }
61             var stack = _pool.GetOrAdd(array.Length, size => new Stack<T[]>(_maxArraysPerSize));
62             if (stack.Count == _maxArraysPerSize) // Stack is full
63             {
64                 return;
65             }
66             stack.Push(array);
67         }
68     }
69 }

```

## ./Arrays/ArrayString.cs

```
1 using Platform.Collections.Segments;
2
3 namespace Platform.Collections.Arrays
4 {
5     public class ArrayString<T> : Segment<T>
6     {
7         public ArrayString(int length) : base(new T[length], 0, length) { }
8         public ArrayString(T[] array) : base(array, 0, array.Length) { }
9         public ArrayString(T[] array, int length) : base(array, 0, length) { }
10    }
11 }
```

## ./Arrays/CharArrayExtensions.cs

```
1 namespace Platform.Collections.Arrays
2 {
3     public static unsafe class CharArrayExtensions
4     {
5         /// <remarks>
6         /// Based on https://github.com/Microsoft/referencesource/blob/3b1eaf5203992df69de44c783\_a3eda37d3d4cd10/mscorlib/system/string.cs#L833
7         /// </remarks>
8         public static int GenerateHashCode(this char[] array, int offset, int length)
9         {
10             var hashSeed = 5381;
11             var hashAccumulator = hashSeed;
12             fixed (char* pointer = &array[offset])
13             {
14                 for (char* s = pointer, last = s + length; s < last; s++)
15                 {
16                     hashAccumulator = (hashAccumulator << 5) + hashAccumulator ^ *s;
17                 }
18             }
19             return hashAccumulator + (hashSeed * 1566083941);
20         }
21
22         /// <remarks>
23         /// Based on https://github.com/Microsoft/referencesource/blob/3b1eaf5203992df69de44c783\_a3eda37d3d4cd10/mscorlib/system/string.cs#L364
24         /// </remarks>
25         public static bool ContentEqualTo(this char[] left, int leftOffset, int length, char[]
26             ↪ right, int rightOffset)
27         {
28             fixed (char* leftPointer = &left[leftOffset])
29             {
30                 fixed (char* rightPointer = &right[rightOffset])
31                 {
32                     char* leftPointerCopy = leftPointer, rightPointerCopy = rightPointer;
33                     if (!CheckArraysMainPartForEquality(ref leftPointerCopy, ref
34                         ↪ rightPointerCopy, ref length))
35                     {
36                         return false;
37                     }
38                     CheckArraysRemainderForEquality(ref leftPointerCopy, ref rightPointerCopy,
39                         ↪ ref length);
40                     return length <= 0;
41                 }
42             }
43         }
44
45         private static bool CheckArraysMainPartForEquality(ref char* left, ref char* right, ref
46             ↪ int length)
47         {
48             while (length >= 10)
49             {
50                 if ((* (int*)left != *(int*)right)
51                     || (*(int*)(left + 2) != *(int*)(right + 2))
52                     || (*(int*)(left + 4) != *(int*)(right + 4))
53                     || (*(int*)(left + 6) != *(int*)(right + 6))
54                     || (*(int*)(left + 8) != *(int*)(right + 8)))
55                 {
56                     return false;
57                 }
58                 left += 10;
59                 right += 10;
60                 length -= 10;
61             }
62             return true;
63         }
64     }
65 }
```

```

59     }
60
61     private static void CheckArraysRemainderForEquality(ref char* left, ref char* right, ref
    ↪ int length)
62     {
63         // This depends on the fact that the String objects are
64         // always zero terminated and that the terminating zero is not included
65         // in the length. For odd string sizes, the last compare will include
66         // the zero terminator.
67         while (length > 0)
68         {
69             if (*(int*)left != *(int*)right)
70             {
71                 break;
72             }
73             left += 2;
74             right += 2;
75             length -= 2;
76         }
77     }
78 }
79 }

```

./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
9  namespace Platform.Collections
10 {
11     /// <remarks>
12     /// А что если хранить карту значений, где каждый бит будет означать присутствует ли блок в
    ↪ 64 бит в массиве значений.
13     /// 64 бита по 0 бит, будут означать отсутствие 64-х блоков по 64 бита. Т.е. упаковка 512
    ↪ байт в 8 байт.
14     /// Подобный принцип можно применять и к 64-ём блокам и т.п. По сути это карта значений. С
    ↪ помощью которой можно быстро
15     /// проверять есть ли значения непосредственно далее (ниже по уровню).
16     /// Или как таблица виртуальной памяти где номер блока означает его присутствие и адрес.
17     /// </remarks>
18     public class BitString
19     {
20         private static readonly byte[] [] BitsSetIn16Bits;
21         private long[] _array;
22         private long _length;
23         private long _minPositiveWord;
24         private long _maxPositiveWord;
25
26         public bool this[long index]
27         {
28             get => Get(index);
29             set => Set(index, value);
30         }
31
32         public long Length
33         {
34             get => _length;
35             set
36             {
37                 if (_length == value)
38                 {
39                     return;
40                 }
41                 Ensure.Always.ArgumentInRange(value, new Range<long>(0, long.MaxValue),
    ↪ nameof(Length));
42                 // Currently we never shrink the array
43                 if (value > _length)
44                 {
45                     var words = GetWordsCountFromIndex(value);
46                     var oldWords = GetWordsCountFromIndex(_length);
47                     if (words > _array.LongLength)
48                     {
49                         var copy = new long[words];
50                         Array.Copy(_array, copy, _array.LongLength);
51                         _array = copy;

```

```

52     }
53     else
54     {
55         // What is going on here?
56         Array.Clear(_array, (int)oldWords, (int)(words - oldWords));
57     }
58     // What is going on here?
59     var mask = (int)(_length % 64);
60     if (mask > 0)
61     {
62         _array[oldWords - 1] &= (1L << mask) - 1;
63     }
64 }
65 else
66 {
67     // Looks like minimum and maximum positive words are not updated
68     throw new NotImplementedException();
69 }
70 _length = value;
71 }
72 }
73
74 #region Constructors
75
76 static BitString()
77 {
78     BitsSetIn16Bits = new byte[65536] [];
79     int i, c, k;
80     byte bitIndex;
81     for (i = 0; i < 65536; i++)
82     {
83         // Calculating size of array (number of positive bits)
84         for (c = 0, k = 1; k <= 65536; k <= 1)
85         {
86             if ((i & k) == k)
87             {
88                 c++;
89             }
90         }
91         var array = new byte[c];
92         // Adding positive bits indices into array
93         for (bitIndex = 0, c = 0, k = 1; k <= 65536; k <= 1)
94         {
95             if ((i & k) == k)
96             {
97                 array[c++] = bitIndex;
98             }
99             bitIndex++;
100         }
101         BitsSetIn16Bits[i] = array;
102     }
103 }
104
105 public BitString(BitString other)
106 {
107     Ensure.Always.ArgumentNotNull(other, nameof(other));
108     _length = other._length;
109     _array = new long[GetWordsCountFromIndex(_length)];
110     _minPositiveWord = other._minPositiveWord;
111     _maxPositiveWord = other._maxPositiveWord;
112     Array.Copy(other._array, _array, _array.LongLength);
113 }
114
115 public BitString(long length)
116 {
117     Ensure.Always.ArgumentInRange(length, GetValidLengthRange(), nameof(length));
118     _length = length;
119     _array = new long[GetWordsCountFromIndex(_length)];
120     MarkBordersAsAllBitsReset();
121 }
122
123 public BitString(long length, bool defaultValue)
124     : this(length)
125 {
126     if (defaultValue)
127     {
128         SetAll();
129     }
130 }

```

```

131
132 #endregion
133
134 public BitString Not()
135 {
136     var words = GetWordsCountFromIndex(_length);
137     for (long i = 0; i < words; i++)
138     {
139         _array[i] = ~_array[i];
140         RefreshBordersByWord(i);
141     }
142     return this;
143 }
144
145 public BitString And(BitString other)
146 {
147     EnsureBitStringHasTheSameSize(other, nameof(other));
148     GetCommonInnerBorders(this, other, out long from, out long to);
149     var otherArray = other._array;
150     for (var i = from; i <= to; i++)
151     {
152         _array[i] &= otherArray[i];
153         RefreshBordersByWord(i);
154     }
155     return this;
156 }
157
158 public BitString Or(BitString other)
159 {
160     EnsureBitStringHasTheSameSize(other, nameof(other));
161     GetCommonOuterBorders(this, other, out long from, out long to);
162     for (var i = from; i <= to; i++)
163     {
164         _array[i] |= other._array[i];
165         RefreshBordersByWord(i);
166     }
167     return this;
168 }
169
170 public BitString Xor(BitString other)
171 {
172     EnsureBitStringHasTheSameSize(other, nameof(other));
173     GetCommonOuterBorders(this, other, out long from, out long to);
174     for (var i = from; i <= to; i++)
175     {
176         _array[i] ^= other._array[i];
177         RefreshBordersByWord(i);
178     }
179     return this;
180 }
181
182 private void RefreshBordersByWord(long wordIndex)
183 {
184     if (_array[wordIndex] == 0)
185     {
186         if (wordIndex == _minPositiveWord && wordIndex != _array.LongLength - 1)
187         {
188             _minPositiveWord++;
189         }
190         if (wordIndex == _maxPositiveWord && wordIndex != 0)
191         {
192             _maxPositiveWord--;
193         }
194     }
195     else
196     {
197         if (wordIndex < _minPositiveWord)
198         {
199             _minPositiveWord = wordIndex;
200         }
201         if (wordIndex > _maxPositiveWord)
202         {
203             _maxPositiveWord = wordIndex;
204         }
205     }
206 }
207
208 public bool TryShrinkBorders()
209 {

```

```

210     GetBorders(out long from, out long to);
211     while (from <= to && _array[from] == 0)
212     {
213         from++;
214     }
215     if (from > to)
216     {
217         MarkBordersAsAllBitsReset();
218         return true;
219     }
220     while (to >= from && _array[to] == 0)
221     {
222         to--;
223     }
224     if (to < from)
225     {
226         MarkBordersAsAllBitsReset();
227         return true;
228     }
229     var bordersUpdated = from != _minPositiveWord || to != _maxPositiveWord;
230     if (bordersUpdated)
231     {
232         SetBorders(from, to);
233     }
234     return bordersUpdated;
235 }
236
237 [MethodImpl(MethodImplOptions.AggressiveInlining)]
238 public bool Get(long index)
239 {
240     Ensure.Always.ArgumentInRange(index, GetValidIndexRange(), nameof(index));
241     return (_array[GetWordIndexFromIndex(index)] & GetBitMaskFromIndex(index)) != 0;
242 }
243
244 [MethodImpl(MethodImplOptions.AggressiveInlining)]
245 public void Set(long index, bool value)
246 {
247     if (value)
248     {
249         Set(index);
250     }
251     else
252     {
253         Reset(index);
254     }
255 }
256
257 [MethodImpl(MethodImplOptions.AggressiveInlining)]
258 public void Set(long index)
259 {
260     Ensure.Always.ArgumentInRange(index, GetValidIndexRange(), nameof(index));
261     var wordIndex = GetWordIndexFromIndex(index);
262     var mask = GetBitMaskFromIndex(index);
263     _array[wordIndex] |= mask;
264     RefreshBordersByWord(wordIndex);
265 }
266
267 [MethodImpl(MethodImplOptions.AggressiveInlining)]
268 public void Reset(long index)
269 {
270     Ensure.Always.ArgumentInRange(index, GetValidIndexRange(), nameof(index));
271     var wordIndex = GetWordIndexFromIndex(index);
272     var mask = GetBitMaskFromIndex(index);
273     _array[wordIndex] &= ~mask;
274     RefreshBordersByWord(wordIndex);
275 }
276
277 public bool Add(long index)
278 {
279     var wordIndex = GetWordIndexFromIndex(index);
280     var mask = GetBitMaskFromIndex(index);
281     if ((_array[wordIndex] & mask) == 0)
282     {
283         _array[wordIndex] |= mask;
284         RefreshBordersByWord(wordIndex);
285         return true;
286     }
287     else
288     {

```

```

289         return false;
290     }
291 }
292
293 public void SetAll(bool value)
294 {
295     if (value)
296     {
297         SetAll();
298     }
299     else
300     {
301         ResetAll();
302     }
303 }
304
305 public void SetAll()
306 {
307     const long fillValue = unchecked((long)0xffffffffffffffff);
308     var words = GetWordsCountFromIndex(_length);
309     for (var i = 0; i < words; i++)
310     {
311         _array[i] = fillValue;
312     }
313     MarkBordersAsAllBitsSet();
314 }
315
316 public void ResetAll()
317 {
318     const long fillValue = 0;
319     GetBorders(out long from, out long to);
320     for (var i = from; i <= to; i++)
321     {
322         _array[i] = fillValue;
323     }
324     MarkBordersAsAllBitsReset();
325 }
326
327 public List<long> GetSetIndices()
328 {
329     var result = new List<long>();
330     GetBorders(out long from, out long to);
331     for (var i = from; i <= to; i++)
332     {
333         var word = _array[i];
334         if (word != 0)
335         {
336             AppendAllSetBitIndices(result, i, word);
337         }
338     }
339     return result;
340 }
341
342 public List<ulong> GetSetUInt64Indices()
343 {
344     var result = new List<ulong>();
345     GetBorders(out ulong from, out ulong to);
346     for (var i = from; i <= to; i++)
347     {
348         var word = _array[i];
349         if (word != 0)
350         {
351             AppendAllSetBitIndices(result, i, word);
352         }
353     }
354     return result;
355 }
356
357 public long GetFirstSetBitIndex()
358 {
359     var i = _minPositiveWord;
360     var word = _array[i];
361     if (word != 0)
362     {
363         return GetFirstSetBitForWord(i, word);
364     }
365     return -1;
366 }
367

```



```

368 public long GetLastSetBitIndex()
369 {
370     var i = _maxPositiveWord;
371     var word = _array[i];
372     if (word != 0)
373     {
374         return GetLastSetBitForWord(i, word);
375     }
376     return -1;
377 }
378
379 public long CountSetBits()
380 {
381     var total = 0L;
382     GetBorders(out long from, out long to);
383     for (var i = from; i <= to; i++)
384     {
385         var word = _array[i];
386         if (word != 0)
387         {
388             total += CountSetBitsForWord(word);
389         }
390     }
391     return total;
392 }
393
394 public bool HaveCommonBits(BitString other)
395 {
396     EnsureBitStringHasTheSameSize(other, nameof(other));
397     GetCommonInnerBorders(this, other, out long from, out long to);
398     var otherArray = other._array;
399     for (var i = from; i <= to; i++)
400     {
401         var left = _array[i];
402         var right = otherArray[i];
403         if (left != 0 && right != 0 && (left & right) != 0)
404         {
405             return true;
406         }
407     }
408     return false;
409 }
410
411 public long CountCommonBits(BitString other)
412 {
413     EnsureBitStringHasTheSameSize(other, nameof(other));
414     GetCommonInnerBorders(this, other, out long from, out long to);
415     var total = 0L;
416     var otherArray = other._array;
417     for (var i = from; i <= to; i++)
418     {
419         var left = _array[i];
420         var right = otherArray[i];
421         var combined = left & right;
422         if (combined != 0)
423         {
424             total += CountSetBitsForWord(combined);
425         }
426     }
427     return total;
428 }
429
430 public List<long> GetCommonIndices(BitString other)
431 {
432     EnsureBitStringHasTheSameSize(other, nameof(other));
433     GetCommonInnerBorders(this, other, out long from, out long to);
434     var result = new List<long>();
435     var otherArray = other._array;
436     for (var i = from; i <= to; i++)
437     {
438         var left = _array[i];
439         var right = otherArray[i];
440         var combined = left & right;
441         if (combined != 0)
442         {
443             AppendAllSetBitIndices(result, i, combined);
444         }
445     }
446     return result;

```

```

447     }
448
449     public List<ulong> GetCommonUInt64Indices(BitString other)
450     {
451         EnsureBitStringHasTheSameSize(other, nameof(other));
452         GetCommonBorders(this, other, out ulong from, out ulong to);
453         var result = new List<ulong>();
454         var otherArray = other._array;
455         for (var i = from; i <= to; i++)
456         {
457             var left = _array[i];
458             var right = otherArray[i];
459             var combined = left & right;
460             if (combined != 0)
461             {
462                 AppendAllSetBitIndices(result, i, combined);
463             }
464         }
465         return result;
466     }
467
468     public long GetFirstCommonBitIndex(BitString other)
469     {
470         EnsureBitStringHasTheSameSize(other, nameof(other));
471         GetCommonInnerBorders(this, other, out long from, out long to);
472         var otherArray = other._array;
473         for (var i = from; i <= to; i++)
474         {
475             var left = _array[i];
476             var right = otherArray[i];
477             var combined = left & right;
478             if (combined != 0)
479             {
480                 return GetFirstSetBitForWord(i, combined);
481             }
482         }
483         return -1;
484     }
485
486     public long GetLastCommonBitIndex(BitString other)
487     {
488         EnsureBitStringHasTheSameSize(other, nameof(other));
489         GetCommonInnerBorders(this, other, out long from, out long to);
490         var otherArray = other._array;
491         for (var i = to; i >= from; i--)
492         {
493             var left = _array[i];
494             var right = otherArray[i];
495             var combined = left & right;
496             if (combined != 0)
497             {
498                 return GetLastSetBitForWord(i, combined);
499             }
500         }
501         return -1;
502     }
503
504     [MethodImpl(MethodImplOptions.AggressiveInlining)]
505     private void EnsureBitStringHasTheSameSize(BitString other, string argumentName)
506     {
507         Ensure.Always.ArgumentNotNull(other, argumentName);
508         if (_length != other._length)
509         {
510             throw new ArgumentException("Bit string must be the same size.", argumentName);
511         }
512     }
513
514     [MethodImpl(MethodImplOptions.AggressiveInlining)]
515     private void MarkBordersAsAllBitsReset() => SetBorders(_array.LongLength - 1, 0);
516
517     [MethodImpl(MethodImplOptions.AggressiveInlining)]
518     private void MarkBordersAsAllBitsSet() => SetBorders(0, _array.LongLength - 1);
519
520     [MethodImpl(MethodImplOptions.AggressiveInlining)]
521     private void GetBorders(out long from, out long to)
522     {
523         from = _minPositiveWord;
524         to = _maxPositiveWord;
525     }

```

```

526 [MethodImpl(MethodImplOptions.AggressiveInlining)]
527 private void GetBorders(out ulong from, out ulong to)
528 {
529     from = (ulong)_minPositiveWord;
530     to = (ulong)_maxPositiveWord;
531 }
532
533 [MethodImpl(MethodImplOptions.AggressiveInlining)]
534 private void SetBorders(long from, long to)
535 {
536     _minPositiveWord = from;
537     _maxPositiveWord = to;
538 }
539
540 [MethodImpl(MethodImplOptions.AggressiveInlining)]
541 private Range<long> GetValidIndexRange() => new Range<long>(0, _length - 1);
542
543 [MethodImpl(MethodImplOptions.AggressiveInlining)]
544 private static Range<long> GetValidLengthRange() => new Range<long>(0, long.MaxValue);
545
546 [MethodImpl(MethodImplOptions.AggressiveInlining)]
547 private static void AppendAllSetBitIndices(List<ulong> result, ulong wordIndex, long
548     ↪ wordValue)
549 {
550     GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
551     ↪ bits32to47, out byte[] bits48to63);
552     AppendAllSetIndices(result, wordIndex, bits00to15, bits16to31, bits32to47,
553     ↪ bits48to63);
554 }
555
556 [MethodImpl(MethodImplOptions.AggressiveInlining)]
557 private static void AppendAllSetBitIndices(List<long> result, long wordIndex, long
558     ↪ wordValue)
559 {
560     GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
561     ↪ bits32to47, out byte[] bits48to63);
562     AppendAllSetIndices(result, wordIndex, bits00to15, bits16to31, bits32to47,
563     ↪ bits48to63);
564 }
565
566 [MethodImpl(MethodImplOptions.AggressiveInlining)]
567 private static long CountSetBitsForWord(long word)
568 {
569     GetBits(word, out byte[] bits00to15, out byte[] bits16to31, out byte[] bits32to47,
570     ↪ out byte[] bits48to63);
571     return bits00to15.LongLength + bits16to31.LongLength + bits32to47.LongLength +
572     ↪ bits48to63.LongLength;
573 }
574
575 [MethodImpl(MethodImplOptions.AggressiveInlining)]
576 private static long GetFirstSetBitForWord(long wordIndex, long wordValue)
577 {
578     GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
579     ↪ bits32to47, out byte[] bits48to63);
580     return GetFirstSetBit(wordIndex, bits00to15, bits16to31, bits32to47, bits48to63);
581 }
582
583 [MethodImpl(MethodImplOptions.AggressiveInlining)]
584 private static long GetLastSetBitForWord(long wordIndex, long wordValue)
585 {
586     GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
587     ↪ bits32to47, out byte[] bits48to63);
588     return GetLastSetBit(wordIndex, bits00to15, bits16to31, bits32to47, bits48to63);
589 }
590
591 private static void AppendAllSetBitIndices(List<long> result, long i, byte[] bits00to15,
592     ↪ byte[] bits16to31, byte[] bits32to47, byte[] bits48to63)
593 {
594     for (var j = 0; j < bits00to15.Length; j++)
595     {
596         result.Add(bits00to15[j] + (i * 64));
597     }
598     for (var j = 0; j < bits16to31.Length; j++)
599     {
600         result.Add(bits16to31[j] + 16 + (i * 64));
601     }
602     for (var j = 0; j < bits32to47.Length; j++)

```

```

593     {
594         result.Add(bits32to47[j] + 32 + (i * 64));
595     }
596     for (var j = 0; j < bits48to63.Length; j++)
597     {
598         result.Add(bits48to63[j] + 48 + (i * 64));
599     }
600 }
601
602 private static void AppendAllSetIndices(List<ulong> result, ulong i, byte[] bits00to15,
↪ byte[] bits16to31, byte[] bits32to47, byte[] bits48to63)
603 {
604     for (var j = 0; j < bits00to15.Length; j++)
605     {
606         result.Add(bits00to15[j] + (i * 64));
607     }
608     for (var j = 0; j < bits16to31.Length; j++)
609     {
610         result.Add(bits16to31[j] + 16UL + (i * 64));
611     }
612     for (var j = 0; j < bits32to47.Length; j++)
613     {
614         result.Add(bits32to47[j] + 32UL + (i * 64));
615     }
616     for (var j = 0; j < bits48to63.Length; j++)
617     {
618         result.Add(bits48to63[j] + 48UL + (i * 64));
619     }
620 }
621
622 private static long GetFirstSetBit(long i, byte[] bits00to15, byte[] bits16to31, byte[]
↪ bits32to47, byte[] bits48to63)
623 {
624     if (bits00to15.Length > 0)
625     {
626         return bits00to15[0] + (i * 64);
627     }
628     if (bits16to31.Length > 0)
629     {
630         return bits16to31[0] + 16 + (i * 64);
631     }
632     if (bits32to47.Length > 0)
633     {
634         return bits32to47[0] + 32 + (i * 64);
635     }
636     return bits48to63[0] + 48 + (i * 64);
637 }
638
639 private static long GetLastSetBit(long i, byte[] bits00to15, byte[] bits16to31, byte[]
↪ bits32to47, byte[] bits48to63)
640 {
641     if (bits48to63.Length > 0)
642     {
643         return bits48to63[bits48to63.Length - 1] + 48 + (i * 64);
644     }
645     if (bits32to47.Length > 0)
646     {
647         return bits32to47[bits32to47.Length - 1] + 32 + (i * 64);
648     }
649     if (bits16to31.Length > 0)
650     {
651         return bits16to31[bits16to31.Length - 1] + 16 + (i * 64);
652     }
653     return bits00to15[bits00to15.Length - 1] + (i * 64);
654 }
655
656 private static void GetBits(long word, out byte[] bits00to15, out byte[] bits16to31, out
↪ byte[] bits32to47, out byte[] bits48to63)
657 {
658     bits00to15 = BitsSetIn16Bits[word & 0xffffu];
659     bits16to31 = BitsSetIn16Bits[(word >> 16) & 0xffffu];
660     bits32to47 = BitsSetIn16Bits[(word >> 32) & 0xffffu];
661     bits48to63 = BitsSetIn16Bits[(word >> 48) & 0xffffu];
662 }
663
664 [MethodImpl(MethodImplOptions.AggressiveInlining)]
665 public static void GetCommonInnerBorders(BitString left, BitString right, out long from,
↪ out long to)

```

```

666     {
667         from = Math.Max(left._minPositiveWord, right._minPositiveWord);
668         to = Math.Min(left._maxPositiveWord, right._maxPositiveWord);
669     }
670
671     [MethodImpl(MethodImplOptions.AggressiveInlining)]
672     public static void GetCommonOuterBorders(BitString left, BitString right, out long from,
        ↪ out long to)
673     {
674         from = Math.Min(left._minPositiveWord, right._minPositiveWord);
675         to = Math.Max(left._maxPositiveWord, right._maxPositiveWord);
676     }
677
678     [MethodImpl(MethodImplOptions.AggressiveInlining)]
679     public static void GetCommonBorders(BitString left, BitString right, out ulong from, out
        ↪ ulong to)
680     {
681         from = (ulong)Math.Max(left._minPositiveWord, right._minPositiveWord);
682         to = (ulong)Math.Min(left._maxPositiveWord, right._maxPositiveWord);
683     }
684
685     [MethodImpl(MethodImplOptions.AggressiveInlining)]
686     public static long GetWordsCountFromIndex(long index) => (index + 63) / 64;
687
688     [MethodImpl(MethodImplOptions.AggressiveInlining)]
689     public static long GetWordIndexFromIndex(long index) => index >> 6;
690
691     [MethodImpl(MethodImplOptions.AggressiveInlining)]
692     public static long GetBitMaskFromIndex(long index) => 1L << (int)(index & 63);
693 }
694 }

```

#### ./Concurrent/ConcurrentQueueExtensions.cs

```

1  using System.Collections.Concurrent;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4
5  namespace Platform.Collections.Concurrent
6  {
7      public static class ConcurrentQueueExtensions
8      {
9          [MethodImpl(MethodImplOptions.AggressiveInlining)]
10         public static IEnumerable<T> DequeueAll<T>(this ConcurrentQueue<T> queue)
11         {
12             while (queue.TryDequeue(out T item))
13             {
14                 yield return item;
15             }
16         }
17     }
18 }

```

#### ./Concurrent/ConcurrentStackExtensions.cs

```

1  using System.Collections.Concurrent;
2  using System.Runtime.CompilerServices;
3
4  namespace Platform.Collections.Concurrent
5  {
6      public static class ConcurrentStackExtensions
7      {
8          [MethodImpl(MethodImplOptions.AggressiveInlining)]
9          public static T PopOrDefault<T>(this ConcurrentStack<T> stack) => stack.TryPop(out T
        ↪ value) ? value : default;
10
11         public static T PeekOrDefault<T>(this ConcurrentStack<T> stack) => stack.TryPeek(out T
        ↪ value) ? value : default;
12     }
13 }

```

#### ./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

```

```

10 namespace Platform.Collections
11 {
12     public static class EnsureExtensions
13     {
14         #region Always
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public static void ArgumentNotEmpty<T>(this EnsureAlwaysExtensionRoot root,
18         ↪ ICollection<T> argument, string argumentName, string message)
19         {
20             if (argument.IsNullOrEmpty())
21             {
22                 throw new ArgumentException(message, argumentName);
23             }
24
25             [MethodImpl(MethodImplOptions.AggressiveInlining)]
26             public static void ArgumentNotEmpty<T>(this EnsureAlwaysExtensionRoot root,
27             ↪ ICollection<T> argument, string argumentName) => ArgumentNotEmpty(root, argument,
28             ↪ argumentName, null);
29
30             [MethodImpl(MethodImplOptions.AggressiveInlining)]
31             public static void ArgumentNotEmpty<T>(this EnsureAlwaysExtensionRoot root,
32             ↪ ICollection<T> argument) => ArgumentNotEmpty(root, argument, null, null);
33
34             [MethodImpl(MethodImplOptions.AggressiveInlining)]
35             public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureAlwaysExtensionRoot root,
36             ↪ string argument, string argumentName, string message)
37             {
38                 if (string.IsNullOrEmptyWhiteSpace(argument))
39                 {
40                     throw new ArgumentException(message, argumentName);
41                 }
42             }
43
44             [MethodImpl(MethodImplOptions.AggressiveInlining)]
45             public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureAlwaysExtensionRoot root,
46             ↪ string argument, string argumentName) => ArgumentNotEmptyAndNotWhiteSpace(root,
47             ↪ argument, argumentName, null);
48
49             [MethodImpl(MethodImplOptions.AggressiveInlining)]
50             public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureAlwaysExtensionRoot root,
51             ↪ string argument) => ArgumentNotEmptyAndNotWhiteSpace(root, argument, null, null);
52
53             #endregion
54
55             #region OnDebug
56
57             [Conditional("DEBUG")]
58             public static void ArgumentNotEmpty<T>(this EnsureOnDebugExtensionRoot root,
59             ↪ ICollection<T> argument, string argumentName, string message) =>
60             ↪ Ensure.Always.ArgumentNotEmpty(argument, argumentName, message);
61
62             [Conditional("DEBUG")]
63             public static void ArgumentNotEmpty<T>(this EnsureOnDebugExtensionRoot root,
64             ↪ ICollection<T> argument, string argumentName) =>
65             ↪ Ensure.Always.ArgumentNotEmpty(argument, argumentName, null);
66
67             [Conditional("DEBUG")]
68             public static void ArgumentNotEmpty<T>(this EnsureOnDebugExtensionRoot root,
69             ↪ ICollection<T> argument) => Ensure.Always.ArgumentNotEmpty(argument, null, null);
70
71             [Conditional("DEBUG")]
72             public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureOnDebugExtensionRoot
73             ↪ root, string argument, string argumentName, string message) =>
74             ↪ Ensure.Always.ArgumentNotEmptyAndNotWhiteSpace(argument, argumentName, message);
75
76             [Conditional("DEBUG")]
77             public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureOnDebugExtensionRoot
78             ↪ root, string argument, string argumentName) =>
79             ↪ Ensure.Always.ArgumentNotEmptyAndNotWhiteSpace(argument, argumentName, null);
80
81             [Conditional("DEBUG")]
82             public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureOnDebugExtensionRoot
83             ↪ root, string argument) => Ensure.Always.ArgumentNotEmptyAndNotWhiteSpace(argument,
84             ↪ null, null);
85
86             #endregion
87         }
88     }
89 }

```

```
69     }
70 }
```

#### ./ICollectionExtensions.cs

```
1 using System.Collections.Generic;
2 using System.Linq;
3
4 namespace Platform.Collections
5 {
6     public static class ICollectionExtensions
7     {
8         public static bool IsNullOrEmpty<T>(this ICollection<T> collection) => collection ==
9             ↪ null || collection.Count == 0;
10
11         public static bool AllEqualToDefault<T>(this ICollection<T> collection)
12         {
13             var equalityComparer = EqualityComparer<T>.Default;
14             return collection.All(item => equalityComparer.Equals(item, default));
15         }
16     }
```

#### ./IDictionaryExtensions.cs

```
1 using System;
2 using System.Collections.Generic;
3 using System.Runtime.CompilerServices;
4
5 namespace Platform.Collections
6 {
7     public static class IDictionaryExtensions
8     {
9         [MethodImpl(MethodImplOptions.AggressiveInlining)]
10        public static TValue GetOrDefault<TKey, TValue>(this IDictionary<TKey, TValue>
11            ↪ dictionary, TKey key)
12        {
13            dictionary.TryGetValue(key, out TValue value);
14            return value;
15        }
16
17        [MethodImpl(MethodImplOptions.AggressiveInlining)]
18        public static TValue GetOrAdd<TKey, TValue>(this IDictionary<TKey, TValue> dictionary,
19            ↪ 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    }
```

#### ./ISetExtensions.cs

```
1 using System.Collections.Generic;
2
3 namespace Platform.Collections
4 {
5     public static class ISetExtensions
6     {
7         public static void AddAndReturnVoid<T>(this ISet<T> set, T element) => set.Add(element);
8         public static void RemoveAndReturnVoid<T>(this ISet<T> set, T element) =>
9             ↪ set.Remove(element);
10        public static bool DoNotContains<T>(this ISet<T> set, T element) =>
11            ↪ !set.Contains(element);
12    }
```

#### ./Lists/CharIListExtensions.cs

```
1 using System.Collections.Generic;
2
3 namespace Platform.Collections.Lists
4 {
5     public static class CharIListExtensions
6     {
7         /// <remarks>
8         /// Based on https://github.com/Microsoft/referencesource/blob/3b1eaf5203992df69de44c783\_
9         ↪ a3eda37d3d4cd10/mscorlib/system/string.cs#L833
10    }
```

```

9      /// </remarks>
10     public static unsafe int GenerateHashCode(this IList<char> list)
11     {
12         var hashSeed = 5381;
13         var hashAccumulator = hashSeed;
14         for (var i = 0; i < list.Count; i++)
15         {
16             hashAccumulator = (hashAccumulator << 5) + hashAccumulator ^ list[i];
17         }
18         return hashAccumulator + (hashSeed * 1566083941);
19     }
20
21     public static bool EqualTo(this IList<char> left, IList<char> right) =>
22     ↪ left.EqualTo(right, ContentEqualTo);
23
24     public static bool ContentEqualTo(this IList<char> left, IList<char> right)
25     {
26         for (var i = left.Count - 1; i >= 0; --i)
27         {
28             if (left[i] != right[i])
29             {
30                 return false;
31             }
32         }
33         return true;
34     }
35 }

```

#### ./Lists/IListComparer.cs

```

1 using System.Collections.Generic;
2
3 namespace Platform.Collections.Lists
4 {
5     public class IListComparer<T> : IComparer<IList<T>>
6     {
7         public int Compare(IList<T> left, IList<T> right) => left.CompareTo(right);
8     }
9 }

```

#### ./Lists/IListEqualityComparer.cs

```

1 using System.Collections.Generic;
2
3 namespace Platform.Collections.Lists
4 {
5     public class IListEqualityComparer<T> : IEqualityComparer<IList<T>>
6     {
7         public bool Equals(IList<T> left, IList<T> right) => left.EqualTo(right);
8         public int GetHashCode(IList<T> list) => list.GenerateHashCode();
9     }
10 }

```

#### ./Lists/IListExtensions.cs

```

1 using System;
2 using System.Collections.Generic;
3
4 namespace Platform.Collections.Lists
5 {
6     public static class IListExtensions
7     {
8         public static bool AddAndReturnTrue<T>(this IList<T> list, T element)
9         {
10             list.Add(element);
11             return true;
12         }
13
14         public static int GetCountOrZero<T>(this IList<T> list) => list?.Count ?? 0;
15
16         public static bool EqualTo<T>(this IList<T> left, IList<T> right) => EqualTo(left,
17 ↪ right, ContentEqualTo);
18
19         public static bool EqualTo<T>(this IList<T> left, IList<T> right, Func<IList<T>,
20 ↪ IList<T>, bool> contentEqualityComparer)
21         {
22             if (ReferenceEquals(left, right))
23             {
24                 return true;
25             }
26             var leftCount = left.GetCountOrZero();
27
28             if (leftCount != right.GetCountOrZero())
29                 return false;
30
31             for (var i = 0; i < leftCount; i++)
32             {
33                 if (!contentEqualityComparer.Equals(left[i], right[i]))
34                     return false;
35             }
36             return true;
37         }
38     }
39 }

```



```

25     var rightCount = right.GetCountOrZero();
26     if (leftCount == 0 && rightCount == 0)
27     {
28         return true;
29     }
30     if (leftCount == 0 || rightCount == 0 || leftCount != rightCount)
31     {
32         return false;
33     }
34     return contentEqualityComparer(left, right);
35 }
36
37 public static bool ContentEqualTo<T>(this IList<T> left, IList<T> right)
38 {
39     var equalityComparer = EqualityComparer<T>.Default;
40     for (var i = left.Count - 1; i >= 0; --i)
41     {
42         if (!equalityComparer.Equals(left[i], right[i]))
43         {
44             return false;
45         }
46     }
47     return true;
48 }
49
50 public static T[] ToArray<T>(this IList<T> list, Func<T, bool> predicate)
51 {
52     if (list == null)
53     {
54         return null;
55     }
56     var result = new List<T>(list.Count);
57     for (var i = 0; i < list.Count; i++)
58     {
59         if (predicate(list[i]))
60         {
61             result.Add(list[i]);
62         }
63     }
64     return result.ToArray();
65 }
66
67 public static T[] ToArray<T>(this IList<T> list)
68 {
69     var array = new T[list.Count];
70     list.CopyTo(array, 0);
71     return array;
72 }
73
74 public static void ForEach<T>(this IList<T> list, Action<T> action)
75 {
76     for (var i = 0; i < list.Count; i++)
77     {
78         action(list[i]);
79     }
80 }
81
82 /// <remarks>
83 /// Based on http://stackoverflow.com/questions/263400/what-is-the-best-algorithm-for-an
84 /// ↪ -overridden-system-object-gethashcode
85 /// </remarks>
86 public static int GenerateHashCode<T>(this IList<T> list)
87 {
88     var result = 17;
89     for (var i = 0; i < list.Count; i++)
90     {
91         result = unchecked((result * 23) + list[i].GetHashCode());
92     }
93     return result;
94 }
95
96 public static int CompareTo<T>(this IList<T> left, IList<T> right)
97 {
98     var comparer = Comparer<T>.Default;
99     var leftCount = left.GetCountOrZero();
100    var rightCount = right.GetCountOrZero();
101    var intermediateResult = leftCount.CompareTo(rightCount);
102    for (var i = 0; intermediateResult == 0 && i < leftCount; i++)

```

```

103         intermediateResult = comparer.Compare(left[i], right[i]);
104     }
105     return intermediateResult;
106 }
107 }
108 }

```

## ./Segments/CharSegment.cs

```

1  using System.Linq;
2  using System.Collections.Generic;
3  using Platform.Collections.Arrays;
4  using Platform.Collections.Lists;
5
6  namespace Platform.Collections.Segments
7  {
8      public class CharSegment : Segment<char>
9      {
10         public CharSegment(IList<char> @base, int offset, int length) : base(@base, offset,
11             ↪ length) { }
12
13         public override int GetHashCode()
14         {
15             // Base can be not an array, but still IList<char>
16             if (Base is char[] baseArray)
17             {
18                 return baseArray.GenerateHashCode(Offset, Length);
19             }
20             else
21             {
22                 return this.GenerateHashCode();
23             }
24         }
25
26         public override bool Equals(Segment<char> other)
27         {
28             bool contentEqualityComparer(IList<char> left, IList<char> right)
29             {
30                 // Base can be not an array, but still IList<char>
31                 if (Base is char[] baseArray && other.Base is char[] otherArray)
32                 {
33                     return baseArray.ContentEqualTo(Offset, Length, otherArray, other.Offset);
34                 }
35                 else
36                 {
37                     return left.ContentEqualTo(right);
38                 }
39             }
40             return this.EqualTo(other, contentEqualityComparer);
41         }
42
43         public static implicit operator string(CharSegment segment)
44         {
45             if (!(segment.Base is char[] array))
46             {
47                 array = segment.Base.ToArray();
48             }
49             return new string(array, segment.Offset, segment.Length);
50         }
51
52         public override string ToString() => this;
53     }
54 }

```

## ./Segments/Segment.cs

```

1  using System;
2  using System.Collections;
3  using System.Collections.Generic;
4  using Platform.Collections.Lists;
5
6  namespace Platform.Collections.Segments
7  {
8      public class Segment<T> : IEquatable<Segment<T>>, IList<T>
9      {
10         public IList<T> Base { get; }
11         public int Offset { get; }
12         public int Length { get; }
13
14         public Segment(IList<T> @base, int offset, int length)
15         {

```

```

16         Base = @base;
17         Offset = offset;
18         Length = length;
19     }
20
21     public override int GetHashCode() => this.GenerateHashCode();
22
23     public virtual bool Equals(Segment<T> other) => this.EqualTo(other);
24
25     public override bool Equals(object obj) => obj is Segment<T> other ? Equals(other) :
    ↪ false;
26
27     #region IList
28
29     public T this[int i]
30     {
31         get => Base[Offset + i];
32         set => Base[Offset + i] = value;
33     }
34
35     public int Count => Length;
36
37     public bool IsReadOnly => true;
38
39     public int IndexOf(T item)
40     {
41         var index = Base.IndexOf(item);
42         if (index >= Offset)
43         {
44             var actualIndex = index - Offset;
45             if (actualIndex < Length)
46             {
47                 return actualIndex;
48             }
49         }
50         return -1;
51     }
52
53     public void Insert(int index, T item) => throw new NotSupportedException();
54
55     public void RemoveAt(int index) => throw new NotSupportedException();
56
57     public void Add(T item) => throw new NotSupportedException();
58
59     public void Clear() => throw new NotSupportedException();
60
61     public bool Contains(T item) => IndexOf(item) >= 0;
62
63     public void CopyTo(T[] array, int arrayIndex)
64     {
65         for (var i = 0; i < Length; i++)
66         {
67             array[arrayIndex++] = this[i];
68         }
69     }
70
71     public bool Remove(T item) => throw new NotSupportedException();
72
73     public IEnumerator<T> GetEnumerator()
74     {
75         for (var i = 0; i < Length; i++)
76         {
77             yield return this[i];
78         }
79     }
80
81     IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
82
83     #endregion
84 }
85 }

```

./Segments/Walkers/AllSegmentsWalkerBase.cs

```

1 namespace Platform.Collections.Segments.Walkers
2 {
3     public abstract class AllSegmentsWalkerBase
4     {
5         public static readonly int DefaultMinimumStringSegmentLength = 2;
6     }
7 }

```

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

```
1 using System.Collections.Generic;
2
3 namespace Platform.Collections.Segments.Walkers
4 {
5     public abstract class AllSegmentsWalkerBase<T> : AllSegmentsWalkerBase<T, Segment<T>>
6     {
7         protected override Segment<T> CreateSegment(IList<T> elements, int offset, int length)
8             => new Segment<T>(elements, offset, length);
9     }
```

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

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

./Segments/Walkers/AllSegmentsWalkerExtensions.cs

```
1 namespace Platform.Collections.Segments.Walkers
2 {
3     public static class AllSegmentsWalkerExtensions
4     {
5         public static void WalkAll(this AllSegmentsWalkerBase<char> walker, string @string) =>
6             walker.WalkAll(@string.ToCharArray());
7         public static void WalkAll<TSegment>(this AllSegmentsWalkerBase<char, TSegment> walker,
8             string @string) where TSegment : Segment<char> =>
9             walker.WalkAll(@string.ToCharArray());
10    }
```

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

```
1 using System.Collections.Generic;
2
3 namespace Platform.Collections.Segments.Walkers
4 {
5     public abstract class DictionaryBasedDuplicateSegmentsWalkerBase<T> :
6         DictionaryBasedDuplicateSegmentsWalkerBase<T, Segment<T>>
7     {
8         protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<Segment<T>, long>
9             dictionary, int minimumStringSegmentLength, bool resetDictionaryOnEachWalk) :
10             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) { }
```

```

11         protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength) :
12             ↪ base(minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
13     }
14 }

```

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

```

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

```

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

```

1  namespace Platform.Collections.Segments.Walkers
2  {
3      public abstract class DuplicateSegmentsWalkerBase<T> : DuplicateSegmentsWalkerBase<T,
4          ↪ Segment<T>>
5      {
6      }
7  }

```

./Segments/Walkers/DuplicateSegmentsWalkerBase[T, TSegment].cs

```
1 namespace Platform.Collections.Segments.Walkers
2 {
3     public abstract class DuplicateSegmentsWalkerBase<T, TSegment> : AllSegmentsWalkerBase<T,
4         ↳ TSegment>
5         where TSegment : Segment<T>
6     {
7         protected DuplicateSegmentsWalkerBase(int minimumStringSegmentLength) :
8             ↳ 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 }
```

./Stacks/DefaultStack.cs

```
1 using System.Collections.Generic;
2
3 namespace Platform.Collections.Stacks
4 {
5     public class DefaultStack<TElement> : Stack<TElement>, IStack<TElement>
6     {
7         public bool IsEmpty => Count <= 0;
8     }
9 }
```

./Stacks/IStack.cs

```
1 namespace Platform.Collections.Stacks
2 {
3     public interface IStack<TElement>
4     {
5         bool IsEmpty { get; }
6         void Push(TElement element);
7         TElement Pop();
8         TElement Peek();
9     }
10 }
```

./Stacks/IStackExtensions.cs

```
1 using System.Runtime.CompilerServices;
2
3 namespace Platform.Collections.Stacks
4 {
5     public static class IStackExtensions
6     {
7         public static void Clear<T>(this IStack<T> stack)
8         {
9             while (!stack.IsEmpty)
10             {
11                 _ = stack.Pop();
12             }
13         }
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         public static T PopOrDefault<T>(this IStack<T> stack) => stack.IsEmpty ? default :
17             ↳ stack.Pop();
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         public static T PeekOrDefault<T>(this IStack<T> stack) => stack.IsEmpty ? default :
21             ↳ stack.Peek();
22     }
23 }
```

### ./Stacks/ISStackFactory.cs

```
1 using Platform.Interfaces;
2 using Platform.Collections.Stacks;
3
4 namespace Platform.Helpers.Collections.Stacks
5 {
6     public interface ISStackFactory<TElement> : IFactory<ISStack<TElement>>
7     {
8     }
9 }
```

### ./Stacks/StackExtensions.cs

```
1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3
4 namespace Platform.Collections.Stacks
5 {
6     public static class StackExtensions
7     {
8         [MethodImpl(MethodImplOptions.AggressiveInlining)]
9         public static T PopOrDefault<T>(this Stack<T> stack) => stack.Count > 0 ? stack.Pop() :
            ↪ default;
10
11         [MethodImpl(MethodImplOptions.AggressiveInlining)]
12         public static T PeekOrDefault<T>(this Stack<T> stack) => stack.Count > 0 ? stack.Peek()
            ↪ : default;
13     }
14 }
```

### ./StringExtensions.cs

```
1 using System;
2 using System.Globalization;
3
4 namespace Platform.Collections
5 {
6     public static class StringExtensions
7     {
8         public static string CapitalizeFirstLetter(this string str)
9         {
10             if (string.IsNullOrEmpty(str))
11             {
12                 return str;
13             }
14             var chars = str.ToCharArray();
15             for (var i = 0; i < chars.Length; i++)
16             {
17                 var category = char.GetUnicodeCategory(chars[i]);
18                 if (category == UnicodeCategory.UppercaseLetter)
19                 {
20                     return str;
21                 }
22                 if (category == UnicodeCategory.LowercaseLetter)
23                 {
24                     chars[i] = char.ToUpper(chars[i]);
25                     return new string(chars);
26                 }
27             }
28             return str;
29         }
30
31         public static string Truncate(this string str, int maxLength) =>
            ↪ string.IsNullOrEmpty(str) ? str : str.Substring(0, Math.Min(str.Length, maxLength));
32     }
33 }
```

### ./Trees/Node.cs

```
1 using System.Collections.Generic;
2
3 // ReSharper disable ForCanBeConvertedToForeach
4
5 namespace Platform.Collections.Trees
6 {
7     public class Node
8     {
9         private Dictionary<object, Node> _childNodes;
10
11         public object Value { get; set; }
12
13         public Dictionary<object, Node> ChildNodes => _childNodes ?? (_childNodes = new
            ↪ Dictionary<object, Node>());
14     }
15 }
```

```

14
15 public Node this[object key]
16 {
17     get
18     {
19         var child = GetChild(key);
20         if (child == null)
21         {
22             child = AddChild(key);
23         }
24         return child;
25     }
26     set => SetChildValue(value, key);
27 }
28
29 public Node(object value) => Value = value;
30
31 public Node() : this(null) { }
32
33 public bool ContainsChild(params object[] keys) => GetChild(keys) != null;
34
35 public Node GetChild(params object[] keys)
36 {
37     var node = this;
38     for (var i = 0; i < keys.Length; i++)
39     {
40         node.ChildNodes.TryGetValue(keys[i], out node);
41         if (node == null)
42         {
43             return null;
44         }
45     }
46     return node;
47 }
48
49 public object GetChildValue(params object[] keys) => GetChild(keys)?.Value;
50
51 public Node AddChild(object key) => AddChild(key, new Node(null));
52
53 public Node AddChild(object key, object value) => AddChild(key, new Node(value));
54
55 public Node AddChild(object key, Node child)
56 {
57     ChildNodes.Add(key, child);
58     return child;
59 }
60
61 public Node SetChild(params object[] keys) => SetChildValue(null, keys);
62
63 public Node SetChild(object key) => SetChildValue(null, key);
64
65 public Node SetChildValue(object value, params object[] keys)
66 {
67     var node = this;
68     for (var i = 0; i < keys.Length; i++)
69     {
70         node = SetChildValue(value, keys[i]);
71     }
72     node.Value = value;
73     return node;
74 }
75
76 public Node SetChildValue(object value, object key)
77 {
78     if (!ChildNodes.TryGetValue(key, out Node child))
79     {
80         child = AddChild(key, value);
81     }
82     child.Value = value;
83     return child;
84 }
85 }
86 }

```



## Index

- ./Arrays/ArrayFiller[TElement, TReturnConstant].cs, 1
- ./Arrays/ArrayFiller[TElement].cs, 1
- ./Arrays/ArrayPool.cs, 1
- ./Arrays/ArrayPool[T].cs, 2
- ./Arrays/ArrayString.cs, 2
- ./Arrays/CharArrayExtensions.cs, 3
- ./BitString.cs, 4
- ./Concurrent/ConcurrentQueueExtensions.cs, 13
- ./Concurrent/ConcurrentStackExtensions.cs, 13
- ./EnsureExtensions.cs, 13
- ./ICollectionExtensions.cs, 15
- ./IDictionaryExtensions.cs, 15
- ./ISetExtensions.cs, 15
- ./Lists/CharListExtensions.cs, 15
- ./Lists/ICollectionComparer.cs, 16
- ./Lists/ICollectionEqualityComparer.cs, 16
- ./Lists/ICollectionExtensions.cs, 16
- ./Segments/CharSegment.cs, 18
- ./Segments/Segment.cs, 18
- ./Segments/Walkers/AllSegmentsWalkerBase.cs, 19
- ./Segments/Walkers/AllSegmentsWalkerBase[T, TSegment].cs, 20
- ./Segments/Walkers/AllSegmentsWalkerBase[T].cs, 19
- ./Segments/Walkers/AllSegmentsWalkerExtensions.cs, 20
- ./Segments/Walkers/DictionaryBasedDuplicateSegmentsWalkerBase[T, Segment].cs, 21
- ./Segments/Walkers/DictionaryBasedDuplicateSegmentsWalkerBase[T].cs, 20
- ./Segments/Walkers/DuplicateSegmentsWalkerBase[T, TSegment].cs, 21
- ./Segments/Walkers/DuplicateSegmentsWalkerBase[T].cs, 21
- ./Stacks/DefaultStack.cs, 22
- ./Stacks/IStack.cs, 22
- ./Stacks/IStackExtensions.cs, 22
- ./Stacks/IStackFactory.cs, 22
- ./Stacks/StackExtensions.cs, 23
- ./StringExtensions.cs, 23
- ./Trees/Node.cs, 23