LinksPlatform's Platform.Collections Class Library

./Arrays/ArrayFiller[TElement].cs

```csharp
using System.Collections.Generic;
using System.Runtime.CompilerServices;

namespace Platform.Collections.Arrays
{
    public class ArrayFiller<TElement>
    {
        protected readonly TElement[] _array;
        protected long _position;

        public ArrayFiller(TElement[] array, long offset)
        {
            _array = array;
            _position = offset;
        }

        public ArrayFiller(TElement[] array) : this(array, 0) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Add(TElement element) => _array[_position++] = element;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool AddAndReturnTrue(TElement element)
        {
            _array[_position++] = element;
            return true;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool AddFirstAndReturnTrue(IList<TElement> collection)
        {
            _array[_position++] = collection[0];
            return true;
        }
    }
}
```

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

```csharp
using System.Collections.Generic;
using System.Runtime.CompilerServices;

namespace Platform.Collections.Arrays
{
    public class ArrayFiller<TElement, TReturnConstant> : ArrayFiller<TElement>
    {
        protected readonly TReturnConstant _returnConstant;

        public ArrayFiller(TElement[] array, long offset, TReturnConstant returnConstant) :
        ↪    base(array, offset) => _returnConstant = returnConstant;

        public ArrayFiller(TElement[] array, TReturnConstant returnConstant) : this(array, 0,
        ↪    returnConstant) { }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TReturnConstant AddAndReturnConstant(TElement element)
        {
            _array[_position++] = element;
            return _returnConstant;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TReturnConstant AddFirstAndReturnConstant(IList<TElement> collection)
        {
            _array[_position++] = collection[0];
            return _returnConstant;
        }
    }
}
```

./Arrays/ArrayPool.cs

```csharp
using System.Runtime.CompilerServices;

namespace Platform.Collections.Arrays
{
    public static class ArrayPool
    {
        public static readonly int DefaultSizesAmount = 512;
        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>
13      public class ArrayPool<T>
14      {
15          public static readonly T[] Empty = new T[0];
16
17          // May be use Default class for that later.
18          [ThreadStatic]
19          internal static ArrayPool<T> _threadInstance;
20          internal static ArrayPool<T> ThreadInstance { get => _threadInstance ?? (_threadInstance
               ↪   = new ArrayPool<T>()); }
21
22          private readonly int _maxArraysPerSize;
23          private readonly Dictionary<int, Stack<T[]>> _pool = new Dictionary<int,
               ↪   Stack<T[]>>(ArrayPool.DefaultSizesAmount);
24
25          public ArrayPool(int maxArraysPerSize) => _maxArraysPerSize = maxArraysPerSize;
26
27          public ArrayPool() : this(ArrayPool.DefaultMaxArraysPerSize) { }
28
29          public Disposable<T[]> AllocateDisposable(long size) => (Allocate(size), Free);
30
31          public Disposable<T[]> Resize(Disposable<T[]> source, long size)
32          {
33              var destination = AllocateDisposable(size);
34              T[] sourceArray = source;
35              T[] destinationArray = destination;
36              Array.Copy(sourceArray, destinationArray, size < sourceArray.Length ? (int)size :
                   ↪   sourceArray.Length);
37              source.Dispose();
38              return destination;
39          }
40
41          public virtual void Clear() => _pool.Clear();
42
43          public virtual T[] Allocate(long size)
44          {
45              Ensure.Always.ArgumentInRange(size, new Range<long>(0, int.MaxValue));
46              return size == 0 ? Empty : _pool.GetOrDefault((int)size)?.PopOrDefault() ?? new
                   ↪   T[size];
47          }
48
49          public virtual void Free(T[] array)
50          {
51              Ensure.Always.ArgumentNotNull(array, nameof(array));
52              if (array.Length == 0)
53              {
54                  return;
55              }
56              var stack = _pool.GetOrAdd(array.Length, size => new Stack<T[]>(_maxArraysPerSize));
57              if (stack.Count == _maxArraysPerSize) // Stack is full
58              {
59                  return;
60              }
61              stack.Push(array);
62          }
63      }
64  }
```

./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[]
           ↪  right, int rightOffset)
26         {
27             fixed (char* leftPointer = &left[leftOffset])
28             {
29                 fixed (char* rightPointer = &right[rightOffset])
30                 {
31                     char* leftPointerCopy = leftPointer, rightPointerCopy = rightPointer;
32                     if (!CheckArraysMainPartForEquality(ref leftPointerCopy, ref
                       ↪  rightPointerCopy, ref length))
33                     {
34                         return false;
35                     }
36                     CheckArraysRemainderForEquality(ref leftPointerCopy, ref rightPointerCopy,
                       ↪   ref length);
37                     return length <= 0;
38                 }
39             }
40         }
41
42         private static bool CheckArraysMainPartForEquality(ref char* left, ref char* right, ref
           ↪   int length)
43         {
44             while (length >= 10)
45             {
46                 if ((*(int*)left != *(int*)right)
47                  || (*(int*)(left + 2) != *(int*)(right + 2))
48                  || (*(int*)(left + 4) != *(int*)(right + 4))
49                  || (*(int*)(left + 6) != *(int*)(right + 6))
50                  || (*(int*)(left + 8) != *(int*)(right + 8)))
51                 {
52                     return false;
53                 }
54                 left += 10;
55                 right += 10;
56                 length -= 10;
57             }
58             return true;
```

```csharp
            }

        private static void CheckArraysRemainderForEquality(ref char* left, ref char* right, ref
          ↪  int length)
        {
            // This depends on the fact that the String objects are
            // always zero terminated and that the terminating zero is not included
            // in the length. For odd string sizes, the last compare will include
            // the zero terminator.
            while (length > 0)
            {
                if (*(int*)left != *(int*)right)
                {
                    break;
                }
                left += 2;
                right += 2;
                length -= 2;
            }
        }
    }
}
```

./BitString.cs

```csharp
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Exceptions;
using Platform.Ranges;

// ReSharper disable ForCanBeConvertedToForeach

namespace Platform.Collections
{
    /// <remarks>
    /// А что если хранить карту значений, где каждый бит будет означать присутствует ли блок в
      ↪  64 бит в массиве значений.
    /// 64 бита по 0 бит, будут означать отсутствие 64-х блоков по 64 бита. Т.е. упаковка 512
      ↪  байт в 8 байт.
    /// Подобный принцип можно применять и к 64-ём блокам и т.п. По сути это карта значений. С
      ↪  помощью которой можно быстро
    /// проверять есть ли значения непосредственно далее (ниже по уровню).
    /// Или как таблица виртуальной памяти где номер блока означает его присутствие и адрес.
    /// </remarks>
    public class BitString
    {
        private static readonly byte[][] BitsSetIn16Bits;
        private long[] _array;
        private long _length;
        private long _minPositiveWord;
        private long _maxPositiveWord;

        public bool this[long index]
        {
            get => Get(index);
            set => Set(index, value);
        }

        public long Length
        {
            get => _length;
            set
            {
                if (_length == value)
                {
                    return;
                }
                Ensure.Always.ArgumentInRange(value, new Range<long>(0, long.MaxValue),
                  ↪  nameof(Length));
                // Currently we never shrink the array
                if (value > _length)
                {
                    var words = GetWordsCountFromIndex(value);
                    var oldWords = GetWordsCountFromIndex(_length);
                    if (words > _array.LongLength)
                    {
                        var copy = new long[words];
                        Array.Copy(_array, copy, _array.LongLength);
                        _array = copy;
```

```csharp
                    }
                    else
                    {
                        // What is going on here?
                        Array.Clear(_array, (int)oldWords, (int)(words - oldWords));
                    }
                    // What is going on here?
                    var mask = (int)(_length % 64);
                    if (mask > 0)
                    {
                        _array[oldWords - 1] &= (1L << mask) - 1;
                    }
                }
                else
                {
                    // Looks like minimum and maximum positive words are not updated
                    throw new NotImplementedException();
                }
                _length = value;
            }
        }

        #region Constructors

        static BitString()
        {
            BitsSetIn16Bits = new byte[65536][];
            int i, c, k;
            byte bitIndex;
            for (i = 0; i < 65536; i++)
            {
                // Calculating size of array (number of positive bits)
                for (c = 0, k = 1; k <= 65536; k <<= 1)
                {
                    if ((i & k) == k)
                    {
                        c++;
                    }
                }
                var array = new byte[c];
                // Adding positive bits indices into array
                for (bitIndex = 0, c = 0, k = 1; k <= 65536; k <<= 1)
                {
                    if ((i & k) == k)
                    {
                        array[c++] = bitIndex;
                    }
                    bitIndex++;
                }
                BitsSetIn16Bits[i] = array;
            }
        }

        public BitString(BitString other)
        {
            Ensure.Always.ArgumentNotNull(other, nameof(other));
            _length = other._length;
            _array = new long[GetWordsCountFromIndex(_length)];
            _minPositiveWord = other._minPositiveWord;
            _maxPositiveWord = other._maxPositiveWord;
            Array.Copy(other._array, _array, _array.LongLength);
        }

        public BitString(long length)
        {
            Ensure.Always.ArgumentInRange(length, GetValidLengthRange(), nameof(length));
            _length = length;
            _array = new long[GetWordsCountFromIndex(_length)];
            MarkBordersAsAllBitsReset();
        }

        public BitString(long length, bool defaultValue)
            : this(length)
        {
            if (defaultValue)
            {
                SetAll();
            }
        }
```

```csharp
        #endregion

        public BitString Not()
        {
            var words = GetWordsCountFromIndex(_length);
            for (long i = 0; i < words; i++)
            {
                _array[i] = ~_array[i];
                RefreshBordersByWord(i);
            }
            return this;
        }

        public BitString And(BitString other)
        {
            EnsureBitStringHasTheSameSize(other, nameof(other));
            GetCommonInnerBorders(this, other, out long from, out long to);
            var otherArray = other._array;
            for (var i = from; i <= to; i++)
            {
                _array[i] &= otherArray[i];
                RefreshBordersByWord(i);
            }
            return this;
        }

        public BitString Or(BitString other)
        {
            EnsureBitStringHasTheSameSize(other, nameof(other));
            GetCommonOuterBorders(this, other, out long from, out long to);
            for (var i = from; i <= to; i++)
            {
                _array[i] |= other._array[i];
                RefreshBordersByWord(i);
            }
            return this;
        }

        public BitString Xor(BitString other)
        {
            EnsureBitStringHasTheSameSize(other, nameof(other));
            GetCommonOuterBorders(this, other, out long from, out long to);
            for (var i = from; i <= to; i++)
            {
                _array[i] ^= other._array[i];
                RefreshBordersByWord(i);
            }
            return this;
        }

        private void RefreshBordersByWord(long wordIndex)
        {
            if (_array[wordIndex] == 0)
            {
                if (wordIndex == _minPositiveWord && wordIndex != _array.LongLength - 1)
                {
                    _minPositiveWord++;
                }
                if (wordIndex == _maxPositiveWord && wordIndex != 0)
                {
                    _maxPositiveWord--;
                }
            }
            else
            {
                if (wordIndex < _minPositiveWord)
                {
                    _minPositiveWord = wordIndex;
                }
                if (wordIndex > _maxPositiveWord)
                {
                    _maxPositiveWord = wordIndex;
                }
            }
        }

        public bool TryShrinkBorders()
        {
```

```csharp
            GetBorders(out long from, out long to);
            while (from <= to && _array[from] == 0)
            {
                from++;
            }
            if (from > to)
            {
                MarkBordersAsAllBitsReset();
                return true;
            }
            while (to >= from && _array[to] == 0)
            {
                to--;
            }
            if (to < from)
            {
                MarkBordersAsAllBitsReset();
                return true;
            }
            var bordersUpdated = from != _minPositiveWord || to != _maxPositiveWord;
            if (bordersUpdated)
            {
                SetBorders(from, to);
            }
            return bordersUpdated;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool Get(long index)
        {
            Ensure.Always.ArgumentInRange(index, GetValidIndexRange(), nameof(index));
            return (_array[GetWordIndexFromIndex(index)] & GetBitMaskFromIndex(index)) != 0;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Set(long index, bool value)
        {
            if (value)
            {
                Set(index);
            }
            else
            {
                Reset(index);
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Set(long index)
        {
            Ensure.Always.ArgumentInRange(index, GetValidIndexRange(), nameof(index));
            var wordIndex = GetWordIndexFromIndex(index);
            var mask = GetBitMaskFromIndex(index);
            _array[wordIndex] |= mask;
            RefreshBordersByWord(wordIndex);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Reset(long index)
        {
            Ensure.Always.ArgumentInRange(index, GetValidIndexRange(), nameof(index));
            var wordIndex = GetWordIndexFromIndex(index);
            var mask = GetBitMaskFromIndex(index);
            _array[wordIndex] &= ~mask;
            RefreshBordersByWord(wordIndex);
        }

        public bool Add(long index)
        {
            var wordIndex = GetWordIndexFromIndex(index);
            var mask = GetBitMaskFromIndex(index);
            if ((_array[wordIndex] & mask) == 0)
            {
                _array[wordIndex] |= mask;
                RefreshBordersByWord(wordIndex);
                return true;
            }
            else
            {
```

```csharp
                    return false;
                }
            }

            public void SetAll(bool value)
            {
                if (value)
                {
                    SetAll();
                }
                else
                {
                    ResetAll();
                }
            }

            public void SetAll()
            {
                const long fillValue = unchecked((long)0xffffffffffffffff);
                var words = GetWordsCountFromIndex(_length);
                for (var i = 0; i < words; i++)
                {
                    _array[i] = fillValue;
                }
                MarkBordersAsAllBitsSet();
            }

            public void ResetAll()
            {
                const long fillValue = 0;
                GetBorders(out long from, out long to);
                for (var i = from; i <= to; i++)
                {
                    _array[i] = fillValue;
                }
                MarkBordersAsAllBitsReset();
            }

            public List<long> GetSetIndices()
            {
                var result = new List<long>();
                GetBorders(out long from, out long to);
                for (var i = from; i <= to; i++)
                {
                    var word = _array[i];
                    if (word != 0)
                    {
                        AppendAllSetBitIndices(result, i, word);
                    }
                }
                return result;
            }

            public List<ulong> GetSetUInt64Indices()
            {
                var result = new List<ulong>();
                GetBorders(out ulong from, out ulong to);
                for (var i = from; i <= to; i++)
                {
                    var word = _array[i];
                    if (word != 0)
                    {
                        AppendAllSetBitIndices(result, i, word);
                    }
                }
                return result;
            }

            public long GetFirstSetBitIndex()
            {
                var i = _minPositiveWord;
                var word = _array[i];
                if (word != 0)
                {
                    return GetFirstSetBitForWord(i, word);
                }
                return -1;
            }
```

```csharp
        public long GetLastSetBitIndex()
        {
            var i = _maxPositiveWord;
            var word = _array[i];
            if (word != 0)
            {
                return GetLastSetBitForWord(i, word);
            }
            return -1;
        }

        public long CountSetBits()
        {
            var total = 0L;
            GetBorders(out long from, out long to);
            for (var i = from; i <= to; i++)
            {
                var word = _array[i];
                if (word != 0)
                {
                    total += CountSetBitsForWord(word);
                }
            }
            return total;
        }

        public bool HaveCommonBits(BitString other)
        {
            EnsureBitStringHasTheSameSize(other, nameof(other));
            GetCommonInnerBorders(this, other, out long from, out long to);
            var otherArray = other._array;
            for (var i = from; i <= to; i++)
            {
                var left = _array[i];
                var right = otherArray[i];
                if (left != 0 && right != 0 && (left & right) != 0)
                {
                    return true;
                }
            }
            return false;
        }

        public long CountCommonBits(BitString other)
        {
            EnsureBitStringHasTheSameSize(other, nameof(other));
            GetCommonInnerBorders(this, other, out long from, out long to);
            var total = 0L;
            var otherArray = other._array;
            for (var i = from; i <= to; i++)
            {
                var left = _array[i];
                var right = otherArray[i];
                var combined = left & right;
                if (combined != 0)
                {
                    total += CountSetBitsForWord(combined);
                }
            }
            return total;
        }

        public List<long> GetCommonIndices(BitString other)
        {
            EnsureBitStringHasTheSameSize(other, nameof(other));
            GetCommonInnerBorders(this, other, out long from, out long to);
            var result = new List<long>();
            var otherArray = other._array;
            for (var i = from; i <= to; i++)
            {
                var left = _array[i];
                var right = otherArray[i];
                var combined = left & right;
                if (combined != 0)
                {
                    AppendAllSetBitIndices(result, i, combined);
                }
            }
            return result;
```

```csharp
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        }
```

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

```csharp
                {
                    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);
        }

        private static void GetBits(long word, out byte[] bits00to15, out byte[] bits16to31, out
        ↪   byte[] bits32to47, out byte[] bits48to63)
        {
            bits00to15 = BitsSetIn16Bits[word & 0xffffu];
            bits16to31 = BitsSetIn16Bits[(word >> 16) & 0xffffu];
            bits32to47 = BitsSetIn16Bits[(word >> 32) & 0xffffu];
            bits48to63 = BitsSetIn16Bits[(word >> 48) & 0xffffu];
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        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
```

```csharp
namespace Platform.Collections
{
    public static class EnsureExtensions
    {
        #region Always

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void ArgumentNotEmpty<T>(this EnsureAlwaysExtensionRoot root,
            ICollection<T> argument, string argumentName, string message)
        {
            if (argument.IsNullOrEmpty())
            {
                throw new ArgumentException(message, argumentName);
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void ArgumentNotEmpty<T>(this EnsureAlwaysExtensionRoot root,
            ICollection<T> argument, string argumentName) => ArgumentNotEmpty(root, argument,
            argumentName, null);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void ArgumentNotEmpty<T>(this EnsureAlwaysExtensionRoot root,
            ICollection<T> argument) => ArgumentNotEmpty(root, argument, null, null);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureAlwaysExtensionRoot root,
            string argument, string argumentName, string message)
        {
            if (string.IsNullOrWhiteSpace(argument))
            {
                throw new ArgumentException(message, argumentName);
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureAlwaysExtensionRoot root,
            string argument, string argumentName) => ArgumentNotEmptyAndNotWhiteSpace(root,
            argument, argumentName, null);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureAlwaysExtensionRoot root,
            string argument) => ArgumentNotEmptyAndNotWhiteSpace(root, argument, null, null);

        #endregion

        #region OnDebug

        [Conditional("DEBUG")]
        public static void ArgumentNotEmpty<T>(this EnsureOnDebugExtensionRoot root,
            ICollection<T> argument, string argumentName, string message) =>
            Ensure.Always.ArgumentNotEmpty(argument, argumentName, message);

        [Conditional("DEBUG")]
        public static void ArgumentNotEmpty<T>(this EnsureOnDebugExtensionRoot root,
            ICollection<T> argument, string argumentName) =>
            Ensure.Always.ArgumentNotEmpty(argument, argumentName, null);

        [Conditional("DEBUG")]
        public static void ArgumentNotEmpty<T>(this EnsureOnDebugExtensionRoot root,
            ICollection<T> argument) => Ensure.Always.ArgumentNotEmpty(argument, null, null);

        [Conditional("DEBUG")]
        public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureOnDebugExtensionRoot
            root, string argument, string argumentName, string message) =>
            Ensure.Always.ArgumentNotEmptyAndNotWhiteSpace(argument, argumentName, message);

        [Conditional("DEBUG")]
        public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureOnDebugExtensionRoot
            root, string argument, string argumentName) =>
            Ensure.Always.ArgumentNotEmptyAndNotWhiteSpace(argument, argumentName, null);

        [Conditional("DEBUG")]
        public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureOnDebugExtensionRoot
            root, string argument) => Ensure.Always.ArgumentNotEmptyAndNotWhiteSpace(argument,
            null, null);

        #endregion
```

```
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 ==
         ↪   null || collection.Count == 0;
9
10           public static bool AllEqualToDefault<T>(this ICollection<T> collection)
11           {
12               var equalityComparer = EqualityComparer<T>.Default;
13               return collection.All(item => equalityComparer.Equals(item, default));
14           }
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>
         ↪   dictionary, TKey key)
11           {
12               dictionary.TryGetValue(key, out TValue value);
13               return value;
14           }
15
16           [MethodImpl(MethodImplOptions.AggressiveInlining)]
17           public static TValue GetOrAdd<TKey, TValue>(this IDictionary<TKey, TValue> dictionary,
         ↪   TKey key, Func<TKey, TValue> valueFactory)
18           {
19               if (!dictionary.TryGetValue(key, out TValue value))
20               {
21                   value = valueFactory(key);
22                   dictionary.Add(key, value);
23                   return value;
24               }
25               return value;
26           }
27       }
28   }
```

./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) =>
         ↪   set.Remove(element);
9            public static bool DoNotContains<T>(this ISet<T> set, T element) =>
         ↪   !set.Contains(element);
10       }
11   }
```

./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
         ↪   a3eda37d3d4cd10/mscorlib/system/string.cs#L833
```

```
 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) =>
   ↪            left.EqualTo(right, ContentEqualTo);
22
23              public static bool ContentEqualTo(this IList<char> left, IList<char> right)
24              {
25                  for (var i = left.Count - 1; i >= 0; --i)
26                  {
27                      if (left[i] != right[i])
28                      {
29                          return false;
30                      }
31                  }
32                  return true;
33              }
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,
   ↪          right, ContentEqualTo);
17
18          public static bool EqualTo<T>(this IList<T> left, IList<T> right, Func<IList<T>,
   ↪          IList<T>, bool> contentEqualityComparer)
19          {
20              if (ReferenceEquals(left, right))
21              {
22                  return true;
23              }
24              var leftCount = left.GetCountOrZero();
```

```csharp
            var rightCount = right.GetCountOrZero();
            if (leftCount == 0 && rightCount == 0)
            {
                return true;
            }
            if (leftCount == 0 || rightCount == 0 || leftCount != rightCount)
            {
                return false;
            }
            return contentEqualityComparer(left, right);
        }

        public static bool ContentEqualTo<T>(this IList<T> left, IList<T> right)
        {
            var equalityComparer = EqualityComparer<T>.Default;
            for (var i = left.Count - 1; i >= 0; --i)
            {
                if (!equalityComparer.Equals(left[i], right[i]))
                {
                    return false;
                }
            }
            return true;
        }

        public static T[] ToArray<T>(this IList<T> list, Func<T, bool> predicate)
        {
            if (list == null)
            {
                return null;
            }
            var result = new List<T>(list.Count);
            for (var i = 0; i < list.Count; i++)
            {
                if (predicate(list[i]))
                {
                    result.Add(list[i]);
                }
            }
            return result.ToArray();
        }

        public static T[] ToArray<T>(this IList<T> list)
        {
            var array = new T[list.Count];
            list.CopyTo(array, 0);
            return array;
        }

        public static void ForEach<T>(this IList<T> list, Action<T> action)
        {
            for (var i = 0; i < list.Count; i++)
            {
                action(list[i]);
            }
        }

        /// <remarks>
        /// Based on http://stackoverflow.com/questions/263400/what-is-the-best-algorithm-for-an↵
        ↪   -overridden-system-object-gethashcode
        /// </remarks>
        public static int GenerateHashCode<T>(this IList<T> list)
        {
            var result = 17;
            for (var i = 0; i < list.Count; i++)
            {
                result = unchecked((result * 23) + list[i].GetHashCode());
            }
            return result;
        }

        public static int CompareTo<T>(this IList<T> left, IList<T> right)
        {
            var comparer = Comparer<T>.Default;
            var leftCount = left.GetCountOrZero();
            var rightCount = right.GetCountOrZero();
            var intermediateResult = leftCount.CompareTo(rightCount);
            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

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

./Segments/Segment.cs

```csharp
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           {
```

```csharp
                Base = @base;
                Offset = offset;
                Length = length;
            }

            public override int GetHashCode() => this.GenerateHashCode();

            public virtual bool Equals(Segment<T> other) => this.EqualTo(other);

            public override bool Equals(object obj) => obj is Segment<T> other ? Equals(other) :
            ↪  false;

            #region IList

            public T this[int i]
            {
                get => Base[Offset + i];
                set => Base[Offset + i] = value;
            }

            public int Count => Length;

            public bool IsReadOnly => true;

            public int IndexOf(T item)
            {
                var index = Base.IndexOf(item);
                if (index >= Offset)
                {
                    var actualIndex = index - Offset;
                    if (actualIndex < Length)
                    {
                        return actualIndex;
                    }
                }
                return -1;
            }

            public void Insert(int index, T item) => throw new NotSupportedException();

            public void RemoveAt(int index) => throw new NotSupportedException();

            public void Add(T item) => throw new NotSupportedException();

            public void Clear() => throw new NotSupportedException();

            public bool Contains(T item) => IndexOf(item) >= 0;

            public void CopyTo(T[] array, int arrayIndex)
            {
                for (var i = 0; i < Length; i++)
                {
                    array[arrayIndex++] = this[i];
                }
            }

            public bool Remove(T item) => throw new NotSupportedException();

            public IEnumerator<T> GetEnumerator()
            {
                for (var i = 0; i < Length; i++)
                {
                    yield return this[i];
                }
            }

            IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();

            #endregion
        }
    }
```

./Segments/Walkers/AllSegmentsWalkerBase.cs

```csharp
namespace Platform.Collections.Segments.Walkers
{
    public abstract class AllSegmentsWalkerBase
    {
        public static readonly int DefaultMinimumStringSegmentLength = 2;
    }
}
```

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

```csharp
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)
              => new Segment<T>(elements, offset, length);
8      }
9  }
```

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

```csharp
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) =>
              _minimumStringSegmentLength = minimumStringSegmentLength;
11
12         protected AllSegmentsWalkerBase() : this(DefaultMinimumStringSegmentLength) { }
13
14         public virtual void WalkAll(IList<T> elements)
15         {
16             for (int offset = 0, maxOffset = elements.Count - _minimumStringSegmentLength;
                  offset <= maxOffset; offset++)
17             {
18                 for (int length = _minimumStringSegmentLength, maxLength = elements.Count -
                      offset; length <= maxLength; length++)
19                 {
20                     Iteration(CreateSegment(elements, offset, length));
21                 }
22             }
23         }
24
25         protected abstract TSegment CreateSegment(IList<T> elements, int offset, int length);
26
27         protected abstract void Iteration(TSegment segment);
28     }
29 }
```

./Segments/Walkers/AllSegmentsWalkerExtensions.cs

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

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

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

```
11        protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength) :
   ↪    base(minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
12        protected DictionaryBasedDuplicateSegmentsWalkerBase() :
   ↪    base(DefaultMinimumStringSegmentLength, 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> :
   ↪    DuplicateSegmentsWalkerBase<T, TSegment>
7           where TSegment : Segment<T>
8       {
9           public static readonly bool DefaultResetDictionaryOnEachWalk;
10
11          private readonly bool _resetDictionaryOnEachWalk;
12          protected IDictionary<TSegment, long> Dictionary;
13
14          protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<TSegment, long>
   ↪    dictionary, int minimumStringSegmentLength, bool resetDictionaryOnEachWalk)
15              : base(minimumStringSegmentLength)
16          {
17              Dictionary = dictionary;
18              _resetDictionaryOnEachWalk = resetDictionaryOnEachWalk;
19          }
20
21          protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<TSegment, long>
   ↪    dictionary, int minimumStringSegmentLength) : this(dictionary,
   ↪    minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
22
23          protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<TSegment, long>
   ↪    dictionary) : this(dictionary, DefaultMinimumStringSegmentLength,
   ↪    DefaultResetDictionaryOnEachWalk) { }
24
25          protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength,
   ↪    bool resetDictionaryOnEachWalk) : this(resetDictionaryOnEachWalk ? null : new
   ↪    Dictionary<TSegment, long>(), minimumStringSegmentLength, resetDictionaryOnEachWalk)
   ↪    { }
26
27          protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength) :
   ↪    this(minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
28
29          protected DictionaryBasedDuplicateSegmentsWalkerBase() :
   ↪    this(DefaultMinimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
30
31          public override void WalkAll(IList<T> elements)
32          {
33              if (_resetDictionaryOnEachWalk)
34              {
35                  var capacity = Math.Ceiling(Math.Pow(elements.Count, 2) / 2);
36                  Dictionary = new Dictionary<TSegment, long>((int)capacity);
37              }
38              base.WalkAll(elements);
39          }
40
41          protected override long GetSegmentFrequency(TSegment segment) =>
   ↪    Dictionary.GetOrDefault(segment);
42
43          protected override void SetSegmentFrequency(TSegment segment, long frequency) =>
   ↪    Dictionary[segment] = frequency;
44      }
45  }
```

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

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

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

```csharp
namespace Platform.Collections.Segments.Walkers
{
    public abstract class DuplicateSegmentsWalkerBase<T, TSegment> : AllSegmentsWalkerBase<T,
        TSegment>
        where TSegment : Segment<T>
    {
        protected DuplicateSegmentsWalkerBase(int minimumStringSegmentLength) :
            base(minimumStringSegmentLength) { }

        protected DuplicateSegmentsWalkerBase() : base(DefaultMinimumStringSegmentLength) { }

        protected override void Iteration(TSegment segment)
        {
            var frequency = GetSegmentFrequency(segment);
            if (frequency == 1)
            {
                OnDublicateFound(segment);
            }
            SetSegmentFrequency(segment, frequency + 1);
        }

        protected abstract void OnDublicateFound(TSegment segment);
        protected abstract long GetSegmentFrequency(TSegment segment);
        protected abstract void SetSegmentFrequency(TSegment segment, long frequency);
    }
}
```

./Stacks/DefaultStack.cs

```csharp
using System.Collections.Generic;

namespace Platform.Collections.Stacks
{
    public class DefaultStack<TElement> : Stack<TElement>, IStack<TElement>
    {
        public bool IsEmpty => Count <= 0;
    }
}
```

./Stacks/IStack.cs

```csharp
namespace Platform.Collections.Stacks
{
    public interface IStack<TElement>
    {
        bool IsEmpty { get; }
        void Push(TElement element);
        TElement Pop();
        TElement Peek();
    }
}
```

./Stacks/IStackExtensions.cs

```csharp
using System.Runtime.CompilerServices;

namespace Platform.Collections.Stacks
{
    public static class IStackExtensions
    {
        public static void Clear<T>(this IStack<T> stack)
        {
            while (!stack.IsEmpty)
            {
                _ = stack.Pop();
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T PopOrDefault<T>(this IStack<T> stack) => stack.IsEmpty ? default :
            stack.Pop();

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T PeekOrDefault<T>(this IStack<T> stack) => stack.IsEmpty ? default :
            stack.Peek();
    }
}
```

./Stacks/IStackFactory.cs

```csharp
using Platform.Interfaces;

namespace Platform.Collections.Stacks
{
    public interface IStackFactory<TElement> : IFactory<IStack<TElement>>
    {
    }
}
```

./Stacks/StackExtensions.cs

```csharp
using System.Collections.Generic;
using System.Runtime.CompilerServices;

namespace Platform.Collections.Stacks
{
    public static class StackExtensions
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T PopOrDefault<T>(this Stack<T> stack) => stack.Count > 0 ? stack.Pop() :
            default;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T PeekOrDefault<T>(this Stack<T> stack) => stack.Count > 0 ? stack.Peek()
            : default;
    }
}
```

./StringExtensions.cs

```csharp
using System;
using System.Globalization;

namespace Platform.Collections
{
    public static class StringExtensions
    {
        public static string CapitalizeFirstLetter(this string str)
        {
            if (string.IsNullOrWhiteSpace(str))
            {
                return str;
            }
            var chars = str.ToCharArray();
            for (var i = 0; i < chars.Length; i++)
            {
                var category = char.GetUnicodeCategory(chars[i]);
                if (category == UnicodeCategory.UppercaseLetter)
                {
                    return str;
                }
                if (category == UnicodeCategory.LowercaseLetter)
                {
                    chars[i] = char.ToUpper(chars[i]);
                    return new string(chars);
                }
            }
            return str;
        }

        public static string Truncate(this string str, int maxLength) =>
            string.IsNullOrEmpty(str) ? str : str.Substring(0, Math.Min(str.Length, maxLength));
    }
}
```

./Trees/Node.cs

```csharp
using System.Collections.Generic;

// ReSharper disable ForCanBeConvertedToForeach

namespace Platform.Collections.Trees
{
    public class Node
    {
        private Dictionary<object, Node> _childNodes;

        public object Value { get; set; }

        public Dictionary<object, Node> ChildNodes => _childNodes ?? (_childNodes = new
            Dictionary<object, Node>());
```

```csharp
        public Node this[object key]
        {
            get
            {
                var child = GetChild(key);
                if (child == null)
                {
                    child = AddChild(key);
                }
                return child;
            }
            set => SetChildValue(value, key);
        }

        public Node(object value) => Value = value;

        public Node() : this(null) { }

        public bool ContainsChild(params object[] keys) => GetChild(keys) != null;

        public Node GetChild(params object[] keys)
        {
            var node = this;
            for (var i = 0; i < keys.Length; i++)
            {
                node.ChildNodes.TryGetValue(keys[i], out node);
                if (node == null)
                {
                    return null;
                }
            }
            return node;
        }

        public object GetChildValue(params object[] keys) => GetChild(keys)?.Value;

        public Node AddChild(object key) => AddChild(key, new Node(null));

        public Node AddChild(object key, object value) => AddChild(key, new Node(value));

        public Node AddChild(object key, Node child)
        {
            ChildNodes.Add(key, child);
            return child;
        }

        public Node SetChild(params object[] keys) => SetChildValue(null, keys);

        public Node SetChild(object key) => SetChildValue(null, key);

        public Node SetChildValue(object value, params object[] keys)
        {
            var node = this;
            for (var i = 0; i < keys.Length; i++)
            {
                node = SetChildValue(value, keys[i]);
            }
            node.Value = value;
            return node;
        }

        public Node SetChildValue(object value, object key)
        {
            if (!ChildNodes.TryGetValue(key, out Node child))
            {
                child = AddChild(key, value);
            }
            child.Value = value;
            return child;
        }
    }
}
```

Index