./Arrays/ArrayFiller[TElement].cs

```
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

```
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

```
using System.Runtime.CompilerServices;

namespace Platform.Collections.Arrays
{
    public static class ArrayPool
    {
        public static readonly int DefaultSizesAmount = 512;
        public static readonly int DefaultMaxArraysPerSize = 32;
```

```

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T[] Allocate<T>(long size) => ArrayPool<T>.ThreadInstance.Allocate(size);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void Free<T>(T[] array) => ArrayPool<T>.ThreadInstance.Free(array);
    }
}
```

./Arrays/ArrayPool[T].cs

```
using System;
using System.Collections.Generic;
using Platform.Exceptions;
using Platform.Disposables;
using Platform.Ranges;
using Platform.Collections.Stacks;

namespace Platform.Collections.Arrays
{
    /// <remarks>
    /// Original idea from
    ↪ http://geekswithblogs.net/blackrob/archive/2014/12/18/array-pooling-in-csharp.aspx
    /// </remarks>
    public class ArrayPool<T>
    {
        public static readonly T[] Empty = new T[0];

        // May be use Default class for that later.
        [ThreadStatic]
        internal static ArrayPool<T> _threadInstance;
        internal static ArrayPool<T> ThreadInstance { get => _threadInstance ?? (_threadInstance
        ↪  = new ArrayPool<T>()); }

        private readonly int _maxArraysPerSize;
        private readonly Dictionary<int, Stack<T[]>> _pool = new Dictionary<int,
        ↪  Stack<T[]>>(ArrayPool.DefaultSizesAmount);

        public ArrayPool(int maxArraysPerSize) => _maxArraysPerSize = maxArraysPerSize;

        public ArrayPool() : this(ArrayPool.DefaultMaxArraysPerSize) { }

        public Disposable<T[]> AllocateDisposable(long size) => (Allocate(size), Free);

        public Disposable<T[]> Resize(Disposable<T[]> source, long size)
        {
            var destination = AllocateDisposable(size);
            T[] sourceArray = source;
            T[] destinationArray = destination;
            Array.Copy(sourceArray, destinationArray, size < sourceArray.Length ? (int)size :
            ↪  sourceArray.Length);
            source.Dispose();
            return destination;
        }

        public virtual void Clear() => _pool.Clear();

        public virtual T[] Allocate(long size)
        {
            Ensure.Always.ArgumentInRange(size, new Range<long>(0, int.MaxValue));
            return size == 0 ? Empty : _pool.GetOrDefault((int)size)?.PopOrDefault() ?? new
            ↪  T[size];
        }

        public virtual void Free(T[] array)
        {
            Ensure.Always.ArgumentNotNull(array, nameof(array));
            if (array.Length == 0)
            {
                return;
            }
            var stack = _pool.GetOrAdd(array.Length, size => new Stack<T[]>(_maxArraysPerSize));
            if (stack.Count == _maxArraysPerSize) // Stack is full
            {
                return;
            }
            stack.Push(array);
        }
    }
}
```

./Arrays/ArrayString.cs

```
using Platform.Collections.Segments;

namespace Platform.Collections.Arrays
{
    public class ArrayString<T> : Segment<T>
    {
        public ArrayString(int length) : base(new T[length], 0, length) { }
        public ArrayString(T[] array) : base(array, 0, array.Length) { }
        public ArrayString(T[] array, int length) : base(array, 0, length) { }
    }
}
```

./Arrays/CharArrayExtensions.cs

```
namespace Platform.Collections.Arrays
{
    public static unsafe class CharArrayExtensions
    {
        /// <remarks>
        /// Based on https://github.com/Microsoft/referencesource/blob/3b1eaf5203992df69de44c783a3eda37d3d4cd10/mscorlib/system/string.cs#L833
        /// </remarks>
        public static int GenerateHashCode(this char[] array, int offset, int length)
        {
            var hashSeed = 5381;
            var hashAccumulator = hashSeed;
            fixed (char* pointer = &array[offset])
            {
                for (char* s = pointer, last = s + length; s < last; s++)
                {
                    hashAccumulator = (hashAccumulator << 5) + hashAccumulator ^ *s;
                }
            }
            return hashAccumulator + (hashSeed * 1566083941);
        }

        /// <remarks>
        /// Based on https://github.com/Microsoft/referencesource/blob/3b1eaf5203992df69de44c783a3eda37d3d4cd10/mscorlib/system/string.cs#L364
        /// </remarks>
        public static bool ContentEqualTo(this char[] left, int leftOffset, int length, char[] right, int rightOffset)
        {
            fixed (char* leftPointer = &left[leftOffset])
            {
                fixed (char* rightPointer = &right[rightOffset])
                {
                    char* leftPointerCopy = leftPointer, rightPointerCopy = rightPointer;
                    if (!CheckArraysMainPartForEquality(ref leftPointerCopy, ref rightPointerCopy, ref length))
                    {
                        return false;
                    }
                    CheckArraysRemainderForEquality(ref leftPointerCopy, ref rightPointerCopy, ref length);
                    return length <= 0;
                }
            }
        }

        private static bool CheckArraysMainPartForEquality(ref char* left, ref char* right, ref int length)
        {
            while (length >= 10)
            {
                if ((*(int*)left != *(int*)right)
                 || (*(int*)(left + 2) != *(int*)(right + 2))
                 || (*(int*)(left + 4) != *(int*)(right + 4))
                 || (*(int*)(left + 6) != *(int*)(right + 6))
                 || (*(int*)(left + 8) != *(int*)(right + 8)))
                {
                    return false;
                }
                left += 10;
                right += 10;
                length -= 10;
            }
            return true;
```

```
        }

        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

```
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;
```

```
                }
                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();
        }
    }
```

```
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         {
```

```
            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
            {
```

```
                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;
        }

```

```
        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;
```

```
        }

        public List<ulong> GetCommonUInt64Indices(BitString other)
        {
            EnsureBitStringHasTheSameSize(other, nameof(other));
            GetCommonBorders(this, other, out ulong from, out ulong to);
            var result = new List<ulong>();
            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;
        }

        public long GetFirstCommonBitIndex(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];
                var combined = left & right;
                if (combined != 0)
                {
                    return GetFirstSetBitForWord(i, combined);
                }
            }
            return -1;
        }

        public long GetLastCommonBitIndex(BitString other)
        {
            EnsureBitStringHasTheSameSize(other, nameof(other));
            GetCommonInnerBorders(this, other, out long from, out long to);
            var otherArray = other._array;
            for (var i = to; i >= from; i--)
            {
                var left = _array[i];
                var right = otherArray[i];
                var combined = left & right;
                if (combined != 0)
                {
                    return GetLastSetBitForWord(i, combined);
                }
            }
            return -1;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void EnsureBitStringHasTheSameSize(BitString other, string argumentName)
        {
            Ensure.Always.ArgumentNotNull(other, argumentName);
            if (_length != other._length)
            {
                throw new ArgumentException("Bit string must be the same size.", argumentName);
            }
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void MarkBordersAsAllBitsReset() => SetBorders(_array.LongLength - 1, 0);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void MarkBordersAsAllBitsSet() => SetBorders(0, _array.LongLength - 1);

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void GetBorders(out long from, out long to)
        {
            from = _minPositiveWord;
            to = _maxPositiveWord;
        }
```

```
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++)
```

```
            {
                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 
```

```
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,
            ↪  ICollection<T> argument, string argumentName, string message)
18          {
19              if (argument.IsNullOrEmpty())
20              {
21                  throw new ArgumentException(message, argumentName);
22              }
23          }
24
25          [MethodImpl(MethodImplOptions.AggressiveInlining)]
26          public static void ArgumentNotEmpty<T>(this EnsureAlwaysExtensionRoot root,
            ↪  ICollection<T> argument, string argumentName) => ArgumentNotEmpty(root, argument,
            ↪  argumentName, null);
27
28          [MethodImpl(MethodImplOptions.AggressiveInlining)]
29          public static void ArgumentNotEmpty<T>(this EnsureAlwaysExtensionRoot root,
            ↪  ICollection<T> argument) => ArgumentNotEmpty(root, argument, null, null);
30
31          [MethodImpl(MethodImplOptions.AggressiveInlining)]
32          public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureAlwaysExtensionRoot root,
            ↪  string argument, string argumentName, string message)
33          {
34              if (string.IsNullOrWhiteSpace(argument))
35              {
36                  throw new ArgumentException(message, argumentName);
37              }
38          }
39
40          [MethodImpl(MethodImplOptions.AggressiveInlining)]
41          public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureAlwaysExtensionRoot root,
            ↪  string argument, string argumentName) => ArgumentNotEmptyAndNotWhiteSpace(root,
            ↪  argument, argumentName, null);
42
43          [MethodImpl(MethodImplOptions.AggressiveInlining)]
44          public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureAlwaysExtensionRoot root,
            ↪  string argument) => ArgumentNotEmptyAndNotWhiteSpace(root, argument, null, null);
45
46          #endregion
47
48          #region OnDebug
49
50          [Conditional("DEBUG")]
51          public static void ArgumentNotEmpty<T>(this EnsureOnDebugExtensionRoot root,
            ↪  ICollection<T> argument, string argumentName, string message) =>
            ↪  Ensure.Always.ArgumentNotEmpty(argument, argumentName, message);
52
53          [Conditional("DEBUG")]
54          public static void ArgumentNotEmpty<T>(this EnsureOnDebugExtensionRoot root,
            ↪  ICollection<T> argument, string argumentName) =>
            ↪  Ensure.Always.ArgumentNotEmpty(argument, argumentName, null);
55
56          [Conditional("DEBUG")]
57          public static void ArgumentNotEmpty<T>(this EnsureOnDebugExtensionRoot root,
            ↪  ICollection<T> argument) => Ensure.Always.ArgumentNotEmpty(argument, null, null);
58
59          [Conditional("DEBUG")]
60          public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureOnDebugExtensionRoot
            ↪  root, string argument, string argumentName, string message) =>
            ↪  Ensure.Always.ArgumentNotEmptyAndNotWhiteSpace(argument, argumentName, message);
61
62          [Conditional("DEBUG")]
63          public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureOnDebugExtensionRoot
            ↪  root, string argument, string argumentName) =>
            ↪  Ensure.Always.ArgumentNotEmptyAndNotWhiteSpace(argument, argumentName, null);
64
65          [Conditional("DEBUG")]
66          public static void ArgumentNotEmptyAndNotWhiteSpace(this EnsureOnDebugExtensionRoot
            ↪  root, string argument) => Ensure.Always.ArgumentNotEmptyAndNotWhiteSpace(argument,
            ↪  null, null);
67
68          #endregion
```

```
    }
}
```

./ICollectionExtensions.cs
```
using System.Collections.Generic;
using System.Linq;

namespace Platform.Collections
{
    public static class ICollectionExtensions
    {
        public static bool IsNullOrEmpty<T>(this ICollection<T> collection) => collection == null || collection.Count == 0;

        public static bool AllEqualToDefault<T>(this ICollection<T> collection)
        {
            var equalityComparer = EqualityComparer<T>.Default;
            return collection.All(item => equalityComparer.Equals(item, default));
        }
    }
}
```

./IDictionaryExtensions.cs
```
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;

namespace Platform.Collections
{
    public static class IDictionaryExtensions
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TValue GetOrDefault<TKey, TValue>(this IDictionary<TKey, TValue> dictionary, TKey key)
        {
            dictionary.TryGetValue(key, out TValue value);
            return value;
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TValue GetOrAdd<TKey, TValue>(this IDictionary<TKey, TValue> dictionary, TKey key, Func<TKey, TValue> valueFactory)
        {
            if (!dictionary.TryGetValue(key, out TValue value))
            {
                value = valueFactory(key);
                dictionary.Add(key, value);
                return value;
            }
            return value;
        }
    }
}
```

./ISetExtensions.cs
```
using System.Collections.Generic;

namespace Platform.Collections
{
    public static class ISetExtensions
    {
        public static void AddAndReturnVoid<T>(this ISet<T> set, T element) => set.Add(element);
        public static void RemoveAndReturnVoid<T>(this ISet<T> set, T element) => set.Remove(element);
        public static bool DoNotContains<T>(this ISet<T> set, T element) => !set.Contains(element);
    }
}
```

./Lists/CharIListExtensions.cs
```
using System.Collections.Generic;

namespace Platform.Collections.Lists
{
    public static class CharIListExtensions
    {
        /// <remarks>
        /// Based on https://github.com/Microsoft/referencesource/blob/3b1eaf5203992df69de44c783a3eda37d3d4cd10/mscorlib/system/string.cs#L833
```

```
        /// </remarks>
        public static unsafe int GenerateHashCode(this IList<char> list)
        {
            var hashSeed = 5381;
            var hashAccumulator = hashSeed;
            for (var i = 0; i < list.Count; i++)
            {
                hashAccumulator = (hashAccumulator << 5) + hashAccumulator ^ list[i];
            }
            return hashAccumulator + (hashSeed * 1566083941);
        }

        public static bool EqualTo(this IList<char> left, IList<char> right) =>
        ↪ left.EqualTo(right, ContentEqualTo);

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

./Lists/IListComparer.cs

```
using System.Collections.Generic;

namespace Platform.Collections.Lists
{
    public class IListComparer<T> : IComparer<IList<T>>
    {
        public int Compare(IList<T> left, IList<T> right) => left.CompareTo(right);
    }
}
```

./Lists/IListEqualityComparer.cs

```
using System.Collections.Generic;

namespace Platform.Collections.Lists
{
    public class IListEqualityComparer<T> : IEqualityComparer<IList<T>>
    {
        public bool Equals(IList<T> left, IList<T> right) => left.EqualTo(right);
        public int GetHashCode(IList<T> list) => list.GenerateHashCode();
    }
}
```

./Lists/IListExtensions.cs

```
using System;
using System.Collections.Generic;

namespace Platform.Collections.Lists
{
    public static class IListExtensions
    {
        public static bool AddAndReturnTrue<T>(this IList<T> list, T element)
        {
            list.Add(element);
            return true;
        }

        public static int GetCountOrZero<T>(this IList<T> list) => list?.Count ?? 0;

        public static bool EqualTo<T>(this IList<T> left, IList<T> right) => EqualTo(left,
        ↪ right, ContentEqualTo);

        public static bool EqualTo<T>(this IList<T> left, IList<T> right, Func<IList<T>,
        ↪ IList<T>, bool> contentEqualityComparer)
        {
            if (ReferenceEquals(left, right))
            {
                return true;
            }
            var leftCount = left.GetCountOrZero();
```

```
            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++)
            {
```

```
                intermediateResult = comparer.Compare(left[i], right[i]);
            }
            return intermediateResult;
        }
    }
}
```

./Segments/CharSegment.cs

```
using System.Linq;
using System.Collections.Generic;
using Platform.Collections.Arrays;
using Platform.Collections.Lists;

namespace Platform.Collections.Segments
{
    public class CharSegment : Segment<char>
    {
        public CharSegment(IList<char> @base, int offset, int length) : base(@base, offset,
        ↪  length) { }

        public override int GetHashCode()
        {
            // Base can be not an array, but still IList<char>
            if (Base is char[] baseArray)
            {
                return baseArray.GenerateHashCode(Offset, Length);
            }
            else
            {
                return this.GenerateHashCode();
            }
        }

        public override bool Equals(Segment<char> other)
        {
            bool contentEqualityComparer(IList<char> left, IList<char> right)
            {
                // Base can be not an array, but still IList<char>
                if (Base is char[] baseArray && other.Base is char[] otherArray)
                {
                    return baseArray.ContentEqualTo(Offset, Length, otherArray, other.Offset);
                }
                else
                {
                    return left.ContentEqualTo(right);
                }
            }
            return this.EqualTo(other, contentEqualityComparer);
        }

        public static implicit operator string(CharSegment segment)
        {
            if (!(segment.Base is char[] array))
            {
                array = segment.Base.ToArray();
            }
            return new string(array, segment.Offset, segment.Length);
        }

        public override string ToString() => this;
    }
}
```

./Segments/Segment.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using Platform.Collections.Lists;

namespace Platform.Collections.Segments
{
    public class Segment<T> : IEquatable<Segment<T>>, IList<T>
    {
        public IList<T> Base { get; }
        public int Offset { get; }
        public int Length { get; }

        public Segment(IList<T> @base, int offset, int length)
        {
```

```
            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

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

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

```
using System.Collections.Generic;

namespace Platform.Collections.Segments.Walkers
{
    public abstract class AllSegmentsWalkerBase<T> : AllSegmentsWalkerBase<T, Segment<T>>
    {
        protected override Segment<T> CreateSegment(IList<T> elements, int offset, int length)
        ↪  => new Segment<T>(elements, offset, length);
    }
}
```

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

```
using System.Collections.Generic;

namespace Platform.Collections.Segments.Walkers
{
    public abstract class AllSegmentsWalkerBase<T, TSegment> : AllSegmentsWalkerBase
        where TSegment : Segment<T>
    {
        private readonly int _minimumStringSegmentLength;

        protected AllSegmentsWalkerBase(int minimumStringSegmentLength) =>
        ↪  _minimumStringSegmentLength = minimumStringSegmentLength;

        protected AllSegmentsWalkerBase() : this(DefaultMinimumStringSegmentLength) { }

        public virtual void WalkAll(IList<T> elements)
        {
            for (int offset = 0, maxOffset = elements.Count - _minimumStringSegmentLength;
            ↪  offset <= maxOffset; offset++)
            {
                for (int length = _minimumStringSegmentLength, maxLength = elements.Count -
                ↪  offset; length <= maxLength; length++)
                {
                    Iteration(CreateSegment(elements, offset, length));
                }
            }
        }

        protected abstract TSegment CreateSegment(IList<T> elements, int offset, int length);

        protected abstract void Iteration(TSegment segment);
    }
}
```

./Segments/Walkers/AllSegmentsWalkerExtensions.cs

```
namespace Platform.Collections.Segments.Walkers
{
    public static class AllSegmentsWalkerExtensions
    {
        public static void WalkAll(this AllSegmentsWalkerBase<char> walker, string @string) =>
        ↪  walker.WalkAll(@string.ToCharArray());
        public static void WalkAll<TSegment>(this AllSegmentsWalkerBase<char, TSegment> walker,
        ↪  string @string) where TSegment : Segment<char> =>
        ↪  walker.WalkAll(@string.ToCharArray());
    }
}
```

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

```
using System.Collections.Generic;

namespace Platform.Collections.Segments.Walkers
{
    public abstract class DictionaryBasedDuplicateSegmentsWalkerBase<T> :
    ↪  DictionaryBasedDuplicateSegmentsWalkerBase<T, Segment<T>>
    {
        protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<Segment<T>, long>
        ↪  dictionary, int minimumStringSegmentLength, bool resetDictionaryOnEachWalk) :
        ↪  base(dictionary, minimumStringSegmentLength, resetDictionaryOnEachWalk) { }
        protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<Segment<T>, long>
        ↪  dictionary, int minimumStringSegmentLength) : base(dictionary,
        ↪  minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
        protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<Segment<T>, long>
        ↪  dictionary) : base(dictionary, DefaultMinimumStringSegmentLength,
        ↪  DefaultResetDictionaryOnEachWalk) { }
        protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength,
        ↪  bool resetDictionaryOnEachWalk) : base(minimumStringSegmentLength,
        ↪  resetDictionaryOnEachWalk) { }
```

```
        protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength) : base(minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
        protected DictionaryBasedDuplicateSegmentsWalkerBase() : base(DefaultMinimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }
    }
}
```

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

```
using System;
using System.Collections.Generic;

namespace Platform.Collections.Segments.Walkers
{
    public abstract class DictionaryBasedDuplicateSegmentsWalkerBase<T, TSegment> : DuplicateSegmentsWalkerBase<T, TSegment>
        where TSegment : Segment<T>
    {
        public static readonly bool DefaultResetDictionaryOnEachWalk;

        private readonly bool _resetDictionaryOnEachWalk;
        protected IDictionary<TSegment, long> Dictionary;

        protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<TSegment, long> dictionary, int minimumStringSegmentLength, bool resetDictionaryOnEachWalk)
            : base(minimumStringSegmentLength)
        {
            Dictionary = dictionary;
            _resetDictionaryOnEachWalk = resetDictionaryOnEachWalk;
        }

        protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<TSegment, long> dictionary, int minimumStringSegmentLength) : this(dictionary, minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }

        protected DictionaryBasedDuplicateSegmentsWalkerBase(IDictionary<TSegment, long> dictionary) : this(dictionary, DefaultMinimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }

        protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength, bool resetDictionaryOnEachWalk) : this(resetDictionaryOnEachWalk ? null : new Dictionary<TSegment, long>(), minimumStringSegmentLength, resetDictionaryOnEachWalk) { }

        protected DictionaryBasedDuplicateSegmentsWalkerBase(int minimumStringSegmentLength) : this(minimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }

        protected DictionaryBasedDuplicateSegmentsWalkerBase() : this(DefaultMinimumStringSegmentLength, DefaultResetDictionaryOnEachWalk) { }

        public override void WalkAll(IList<T> elements)
        {
            if (_resetDictionaryOnEachWalk)
            {
                var capacity = Math.Ceiling(Math.Pow(elements.Count, 2) / 2);
                Dictionary = new Dictionary<TSegment, long>((int)capacity);
            }
            base.WalkAll(elements);
        }

        protected override long GetSegmentFrequency(TSegment segment) => Dictionary.GetOrDefault(segment);

        protected override void SetSegmentFrequency(TSegment segment, long frequency) => Dictionary[segment] = frequency;
    }
}
```

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

```
namespace Platform.Collections.Segments.Walkers
{
    public abstract class DuplicateSegmentsWalkerBase<T> : DuplicateSegmentsWalkerBase<T, Segment<T>>
    {
    }
}
```

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

```
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

```
using System.Collections.Generic;

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

./Stacks/IStack.cs

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

./Stacks/IStackExtensions.cs

```
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
```
using Platform.Interfaces;
using Platform.Collections.Stacks;

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

./Stacks/StackExtensions.cs
```
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
```
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
```
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>());
```

```
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