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

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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;
        }
    }
}
```

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

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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;
        }
    }
}
```

1.3 ./Platform.Collections/Arrays/ArrayPool.cs

```
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

```

```
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);
    }
}
```

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

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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, (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);
        }
    }
}
```

1.5 ./Platform.Collections/Arrays/ArrayString.cs

```
using Platform.Collections.Segments;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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) { }
    }
}
```

1.6 ./Platform.Collections/Arrays/CharArrayExtensions.cs

```
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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;
            }
        }
    }
}
```

1.7 ./Platform.Collections/Arrays/GenericArrayExtensions.cs

```
using System;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Collections.Arrays
{
    public static class GenericArrayExtensions
    {
        public static T[] Clone<T>(this T[] array)
        {
            var copy = new T[array.Length];
            Array.Copy(array, 0, copy, 0, array.Length);
            return copy;
        }
    }
}
```

1.8 ./Platform.Collections/BitString.cs

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

// ReSharper disable ForCanBeConvertedToForeach
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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 : IEquatable<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, GetValidLengthRange(), 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();
            }
        }

        #endregion

        public BitString Not()
        {
            for (var i = 0; i < _array.Length; i++)
            {
                _array[i] = ~_array[i];
                RefreshBordersByWord(i);
            }
            return this;
        }

        public BitString VectorNot()
        {
            if (_array.Length != Vector<long>.Count)
                return Not();
            var thisVector = new Vector<long>(_array);
            var result = ~thisVector;
            result.CopyTo(_array, 0);
            MarkBordersAsAllBitsSet();
            TryShrinkBorders();
            return this;
        }

        public BitString And(BitString other)
        {
            EnsureBitStringHasTheSameSize(other, nameof(other));
            GetCommonOuterBorders(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 VectorAnd(BitString other)
        {
            if (_array.Length != Vector<long>.Count)
                return And(other);
            EnsureBitStringHasTheSameSize(other, nameof(other));
            var thisVector = new Vector<long>(_array);
            var otherVector = new Vector<long>(other._array);
            var result = thisVector & otherVector;
            result.CopyTo(_array, 0);
            MarkBordersAsAllBitsSet();
```

```
            TryShrinkBorders();
            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 VectorOr(BitString other)
        {
            if (_array.Length != Vector<long>.Count)
                return Or(other);
            EnsureBitStringHasTheSameSize(other, nameof(other));
            var thisVector = new Vector<long>(_array);
            var otherVector = new Vector<long>(other._array);
            var result = thisVector | otherVector;
            result.CopyTo(_array, 0);
            MarkBordersAsAllBitsSet();
            TryShrinkBorders();
            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()
        {
            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;
        }

        public override bool Equals(object obj) => obj is BitString @string ? Equals(@string) :
        ↪  false;

        public bool Equals(BitString other)
        {
            if (_length != other._length)
            {
                return false;
            }
            var otherArray = other._array;
            if (_array.Length != otherArray.Length)
            {
                return false;
            }
            if (_minPositiveWord != other._minPositiveWord)
            {
                return false;
            }
            if (_maxPositiveWord != other._maxPositiveWord)
            {
                return false;
            }
            GetCommonBorders(this, other, out ulong from, out ulong to);
            for (var i = from; i <= to; i++)
            {
                if (_array[i] != otherArray[i])
                {
                    return false;
                }
            }
            return true;
```

```
575         }
576
577         [MethodImpl(MethodImplOptions.AggressiveInlining)]
578         private void EnsureBitStringHasTheSameSize(BitString other, string argumentName)
579         {
580             Ensure.Always.ArgumentNotNull(other, argumentName);
581             if (_length != other._length)
582             {
583                 throw new ArgumentException("Bit string must be the same size.", argumentName);
584             }
585         }
586
587         [MethodImpl(MethodImplOptions.AggressiveInlining)]
588         private void MarkBordersAsAllBitsReset() => SetBorders(_array.LongLength - 1, 0);
589
590         [MethodImpl(MethodImplOptions.AggressiveInlining)]
591         private void MarkBordersAsAllBitsSet() => SetBorders(0, _array.LongLength - 1);
592
593         [MethodImpl(MethodImplOptions.AggressiveInlining)]
594         private void GetBorders(out long from, out long to)
595         {
596             from = _minPositiveWord;
597             to = _maxPositiveWord;
598         }
599
600         [MethodImpl(MethodImplOptions.AggressiveInlining)]
601         private void GetBorders(out ulong from, out ulong to)
602         {
603             from = (ulong)_minPositiveWord;
604             to = (ulong)_maxPositiveWord;
605         }
606
607         [MethodImpl(MethodImplOptions.AggressiveInlining)]
608         private void SetBorders(long from, long to)
609         {
610             _minPositiveWord = from;
611             _maxPositiveWord = to;
612         }
613
614         [MethodImpl(MethodImplOptions.AggressiveInlining)]
615         private Range<long> GetValidIndexRange() => (0, _length - 1);
616
617         [MethodImpl(MethodImplOptions.AggressiveInlining)]
618         private static Range<long> GetValidLengthRange() => (0, long.MaxValue);
619
620         [MethodImpl(MethodImplOptions.AggressiveInlining)]
621         private static void AppendAllSetBitIndices(List<ulong> result, ulong wordIndex, long
    ↪  wordValue)
622         {
623             GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
    ↪  bits32to47, out byte[] bits48to63);
624             AppendAllSetIndices(result, wordIndex, bits00to15, bits16to31, bits32to47,
    ↪  bits48to63);
625         }
626
627         [MethodImpl(MethodImplOptions.AggressiveInlining)]
628         private static void AppendAllSetBitIndices(List<long> result, long wordIndex, long
    ↪  wordValue)
629         {
630             GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
    ↪  bits32to47, out byte[] bits48to63);
631             AppendAllSetBitIndices(result, wordIndex, bits00to15, bits16to31, bits32to47,
    ↪  bits48to63);
632         }
633
634         [MethodImpl(MethodImplOptions.AggressiveInlining)]
635         private static long CountSetBitsForWord(long word)
636         {
637             GetBits(word, out byte[] bits00to15, out byte[] bits16to31, out byte[] bits32to47,
    ↪  out byte[] bits48to63);
638             return bits00to15.LongLength + bits16to31.LongLength + bits32to47.LongLength +
    ↪  bits48to63.LongLength;
639         }
640
641         [MethodImpl(MethodImplOptions.AggressiveInlining)]
642         private static long GetFirstSetBitForWord(long wordIndex, long wordValue)
643         {
644             GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
    ↪  bits32to47, out byte[] bits48to63);
```

```
645             return GetFirstSetBit(wordIndex, bits00to15, bits16to31, bits32to47, bits48to63);
646         }
647 
648         [MethodImpl(MethodImplOptions.AggressiveInlining)]
649         private static long GetLastSetBitForWord(long wordIndex, long wordValue)
650         {
651             GetBits(wordValue, out byte[] bits00to15, out byte[] bits16to31, out byte[]
                ↪  bits32to47, out byte[] bits48to63);
652             return GetLastSetBit(wordIndex, bits00to15, bits16to31, bits32to47, bits48to63);
653         }
654 
655         private static void AppendAllSetBitIndices(List<long> result, long i, byte[] bits00to15,
            ↪  byte[] bits16to31, byte[] bits32to47, byte[] bits48to63)
656         {
657             for (var j = 0; j < bits00to15.Length; j++)
658             {
659                 result.Add(bits00to15[j] + (i * 64));
660             }
661             for (var j = 0; j < bits16to31.Length; j++)
662             {
663                 result.Add(bits16to31[j] + 16 + (i * 64));
664             }
665             for (var j = 0; j < bits32to47.Length; j++)
666             {
667                 result.Add(bits32to47[j] + 32 + (i * 64));
668             }
669             for (var j = 0; j < bits48to63.Length; j++)
670             {
671                 result.Add(bits48to63[j] + 48 + (i * 64));
672             }
673         }
674 
675         private static void AppendAllSetIndices(List<ulong> result, ulong i, byte[] bits00to15,
            ↪  byte[] bits16to31, byte[] bits32to47, byte[] bits48to63)
676         {
677             for (var j = 0; j < bits00to15.Length; j++)
678             {
679                 result.Add(bits00to15[j] + (i * 64));
680             }
681             for (var j = 0; j < bits16to31.Length; j++)
682             {
683                 result.Add(bits16to31[j] + 16UL + (i * 64));
684             }
685             for (var j = 0; j < bits32to47.Length; j++)
686             {
687                 result.Add(bits32to47[j] + 32UL + (i * 64));
688             }
689             for (var j = 0; j < bits48to63.Length; j++)
690             {
691                 result.Add(bits48to63[j] + 48UL + (i * 64));
692             }
693         }
694 
695         private static long GetFirstSetBit(long i, byte[] bits00to15, byte[] bits16to31, byte[]
            ↪  bits32to47, byte[] bits48to63)
696         {
697             if (bits00to15.Length > 0)
698             {
699                 return bits00to15[0] + (i * 64);
700             }
701             if (bits16to31.Length > 0)
702             {
703                 return bits16to31[0] + 16 + (i * 64);
704             }
705             if (bits32to47.Length > 0)
706             {
707                 return bits32to47[0] + 32 + (i * 64);
708             }
709             return bits48to63[0] + 48 + (i * 64);
710         }
711 
712         private static long GetLastSetBit(long i, byte[] bits00to15, byte[] bits16to31, byte[]
            ↪  bits32to47, byte[] bits48to63)
713         {
714             if (bits48to63.Length > 0)
715             {
716                 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)
        {
            from = Math.Max(left._minPositiveWord, right._minPositiveWord);
            to = Math.Min(left._maxPositiveWord, right._maxPositiveWord);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void GetCommonOuterBorders(BitString left, BitString right, out long from, out long to)
        {
            from = Math.Min(left._minPositiveWord, right._minPositiveWord);
            to = Math.Max(left._maxPositiveWord, right._maxPositiveWord);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static void GetCommonBorders(BitString left, BitString right, out ulong from, out ulong to)
        {
            from = (ulong)Math.Max(left._minPositiveWord, right._minPositiveWord);
            to = (ulong)Math.Min(left._maxPositiveWord, right._maxPositiveWord);
        }

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static long GetWordsCountFromIndex(long index) => (index + 63) / 64;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static long GetWordIndexFromIndex(long index) => index >> 6;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static long GetBitMaskFromIndex(long index) => 1L << (int)(index & 63);

        public override int GetHashCode() => base.GetHashCode();

        public override string ToString() => base.ToString();
    }
}
```

1.9 ./Platform.Collections/BitStringExtensions.cs

```
using Platform.Random;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Collections
{
    public static class BitStringExtensions
    {
        public static void SetRandomBits(this BitString @string)
        {
            for (var i = 0; i < @string.Length; i++)
            {
                var value = RandomHelpers.Default.NextBoolean();
                @string.Set(i, value);
            }
        }
    }
}
```

1.10 ./Platform.Collections/Concurrent/ConcurrentQueueExtensions.cs

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Collections.Concurrent
{
    public static class ConcurrentQueueExtensions
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static IEnumerable<T> DequeueAll<T>(this ConcurrentQueue<T> queue)
        {
            while (queue.TryDequeue(out T item))
            {
                yield return item;
            }
        }
    }
}
```

1.11 ./Platform.Collections/Concurrent/ConcurrentStackExtensions.cs

```
using System.Collections.Concurrent;
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Collections.Concurrent
{
    public static class ConcurrentStackExtensions
    {
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T PopOrDefault<T>(this ConcurrentStack<T> stack) => stack.TryPop(out T
        ↪  value) ? value : default;

        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static T PeekOrDefault<T>(this ConcurrentStack<T> stack) => stack.TryPeek(out T
        ↪  value) ? value : default;
    }
}
```

1.12 ./Platform.Collections/EnsureExtensions.cs

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Runtime.CompilerServices;
using Platform.Exceptions;
using Platform.Exceptions.ExtensionRoots;

#pragma warning disable IDE0060 // Remove unused parameter
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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)]
```

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

1.13 ./Platform.Collections/ICollectionExtensions.cs

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

1.14 ./Platform.Collections/IDictionaryExtensions.cs

```
1   using System;
2   using System.Collections.Generic;
```

```
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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;
        }
    }
}
```

1.15 ./Platform.Collections/ISetExtensions.cs

```
using System.Collections.Generic;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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);
    }
}
```

1.16 ./Platform.Collections/Lists/CharIListExtensions.cs

```
using System.Collections.Generic;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

namespace Platform.Collections.Lists
{
    public static class CharIListExtensions
    {
        /// <remarks>
        /// Based on https://github.com/Microsoft/referencesource/blob/3b1eaf5203992df69de44c783
        ↪  a3eda37d3d4cd10/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;
        }
    }
}
```

1.17 ./Platform.Collections/Lists/IListComparer.cs

```
using System.Collections.Generic;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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

1.18 ./Platform.Collections/Lists/IListEqualityComparer.cs

```
using System.Collections.Generic;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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();
    }
}
```

1.19 ./Platform.Collections/Lists/IListExtensions.cs

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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)
```

```
40          {
41              var equalityComparer = EqualityComparer<T>.Default;
42              for (var i = left.Count - 1; i >= 0; --i)
43              {
44                  if (!equalityComparer.Equals(left[i], right[i]))
45                  {
46                      return false;
47                  }
48              }
49              return true;
50          }
51
52          public static T[] ToArray<T>(this IList<T> list, Func<T, bool> predicate)
53          {
54              if (list == null)
55              {
56                  return null;
57              }
58              var result = new List<T>(list.Count);
59              for (var i = 0; i < list.Count; i++)
60              {
61                  if (predicate(list[i]))
62                  {
63                      result.Add(list[i]);
64                  }
65              }
66              return result.ToArray();
67          }
68
69          public static T[] ToArray<T>(this IList<T> list)
70          {
71              var array = new T[list.Count];
72              list.CopyTo(array, 0);
73              return array;
74          }
75
76          public static void ForEach<T>(this IList<T> list, Action<T> action)
77          {
78              for (var i = 0; i < list.Count; i++)
79              {
80                  action(list[i]);
81              }
82          }
83
84          /// <remarks>
85          /// Based on http://stackoverflow.com/questions/263400/what-is-the-best-algorithm-for-an
            ↪  -overridden-system-object-gethashcode
86          /// </remarks>
87          public static int GenerateHashCode<T>(this IList<T> list)
88          {
89              var result = 17;
90              for (var i = 0; i < list.Count; i++)
91              {
92                  result = unchecked((result * 23) + list[i].GetHashCode());
93              }
94              return result;
95          }
96
97          public static int CompareTo<T>(this IList<T> left, IList<T> right)
98          {
99              var comparer = Comparer<T>.Default;
100             var leftCount = left.GetCountOrZero();
101             var rightCount = right.GetCountOrZero();
102             var intermediateResult = leftCount.CompareTo(rightCount);
103             for (var i = 0; intermediateResult == 0 && i < leftCount; i++)
104             {
105                 intermediateResult = comparer.Compare(left[i], right[i]);
106             }
107             return intermediateResult;
108         }
109     }
110 }
```

1.20 ./Platform.Collections/Segments/CharSegment.cs

```
1  using System.Linq;
2  using System.Collections.Generic;
3  using Platform.Collections.Arrays;
4  using Platform.Collections.Lists;
5
```

```
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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;
    }
}
```

1.21 ./Platform.Collections/Segments/Segment.cs

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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);

```

```
27          public override bool Equals(object obj) => obj is Segment<T> other ? Equals(other) :
            ↪  false;
28
29          #region IList
30
31          public T this[int i]
32          {
33              get => Base[Offset + i];
34              set => Base[Offset + i] = value;
35          }
36
37          public int Count => Length;
38
39          public bool IsReadOnly => true;
40
41          public int IndexOf(T item)
42          {
43              var index = Base.IndexOf(item);
44              if (index >= Offset)
45              {
46                  var actualIndex = index - Offset;
47                  if (actualIndex < Length)
48                  {
49                      return actualIndex;
50                  }
51              }
52              return -1;
53          }
54
55          public void Insert(int index, T item) => throw new NotSupportedException();
56
57          public void RemoveAt(int index) => throw new NotSupportedException();
58
59          public void Add(T item) => throw new NotSupportedException();
60
61          public void Clear() => throw new NotSupportedException();
62
63          public bool Contains(T item) => IndexOf(item) >= 0;
64
65          public void CopyTo(T[] array, int arrayIndex)
66          {
67              for (var i = 0; i < Length; i++)
68              {
69                  array[arrayIndex++] = this[i];
70              }
71          }
72
73          public bool Remove(T item) => throw new NotSupportedException();
74
75          public IEnumerator<T> GetEnumerator()
76          {
77              for (var i = 0; i < Length; i++)
78              {
79                  yield return this[i];
80              }
81          }
82
83          IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
84
85          #endregion
86      }
87  }
```

1.22 ./Platform.Collections/Segments/Walkers/AllSegmentsWalkerBase.cs

```
1   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3   namespace Platform.Collections.Segments.Walkers
4   {
5       public abstract class AllSegmentsWalkerBase
6       {
7           public static readonly int DefaultMinimumStringSegmentLength = 2;
8       }
9   }
```

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

```
1   using System.Collections.Generic;
2
3   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5   namespace Platform.Collections.Segments.Walkers
```

```
{
    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);
    }
}
```

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

```
using System.Collections.Generic;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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);
    }
}
```

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

```
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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());
    }
}
```

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

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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;
    }
}
```

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

```
using System.Collections.Generic;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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) { }
    }
}
```

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

```
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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);
    }
}
```

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

```
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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

1.30 ./Platform.Collections/Stacks/DefaultStack.cs

```
using System.Collections.Generic;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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

1.31 ./Platform.Collections/Stacks/IStack.cs

```
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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

1.32 ./Platform.Collections/Stacks/IStackExtensions.cs

```
using System.Runtime.CompilerServices;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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();
    }
}
```

1.33 ./Platform.Collections/Stacks/IStackFactory.cs

```
using Platform.Interfaces;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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

1.34 ./Platform.Collections/Stacks/StackExtensions.cs

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

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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;
    }
}
```

1.35 ./Platform.Collections/StringExtensions.cs

```
using System;
using System.Globalization;

#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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

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

```

```
        public static string TrimSingle(this string @string, char charToTrim)
        {
            if (!string.IsNullOrEmpty(@string))
            {
                if (@string.Length == 1)
                {
                    if (@string[0] == charToTrim)
                    {
                        return "";
                    }
                    else
                    {
                        return @string;
                    }
                }
                else
                {
                    var left = 0;
                    var right = @string.Length - 1;
                    if (@string[left] == charToTrim)
                    {
                        left++;
                    }
                    if (@string[right] == charToTrim)
                    {
                        right--;
                    }
                    return @string.Substring(left, right - left + 1);
                }
            }
            else
            {
                return @string;
            }
        }
    }
}
```

1.36 ./Platform.Collections/Trees/Node.cs

```
using System.Collections.Generic;

// ReSharper disable ForCanBeConvertedToForeach
#pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

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>());

        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;
        }
    }
}
```

1.37 ./Platform.Collections.Tests/BitStringTests.cs

```
using System;
using System.Collections;
using Xunit;
using Platform.Random;

namespace Platform.Collections.Tests
{
    public static class BitStringTests
    {
        [Fact]
        public static void BitGetSetTest()
        {
            const int n = 250;
            var bitArray = new BitArray(n);
            var bitString = new BitString(n);
            for (var i = 0; i < n; i++)
            {
                var value = RandomHelpers.Default.NextBoolean();
                bitArray.Set(i, value);
                bitString.Set(i, value);
                Assert.Equal(value, bitArray.Get(i));
                Assert.Equal(value, bitString.Get(i));
            }
        }

        [Fact]
        public static void BitAndTest()
        {
            TestToOperationsWithSameMeaning((x, y, w, v) =>
            {
                x.VectorAnd(y);
```

```
                w.And(v);
            });
        }

        [Fact]
        public static void BitNotTest()
        {
            TestToOperationsWithSameMeaning((x, y, w, v) =>
            {
                x.VectorNot();
                w.Not();
            });
        }

        [Fact]
        public static void BitOrTest()
        {
            TestToOperationsWithSameMeaning((x, y, w, v) =>
            {
                x.VectorOr(y);
                w.Or(v);
            });
        }

        private static void TestToOperationsWithSameMeaning(Action<BitString, BitString,
        ↪  BitString, BitString> test)
        {
            const int n = 250;
            var x = new BitString(n);
            var y = new BitString(n);
            while (x.Equals(y))
            {
                x.SetRandomBits();
                y.SetRandomBits();
            }
            var w = new BitString(x);
            var v = new BitString(y);
            Assert.False(x.Equals(y));
            Assert.False(w.Equals(v));
            Assert.True(x.Equals(w));
            Assert.True(y.Equals(v));
            test(x, y, w, v);
            Assert.True(x.Equals(w));
        }
    }
}
```

1.38 ./Platform.Collections.Tests/CharsSegmentTests.cs

```
using Xunit;
using Platform.Collections.Segments;

namespace Platform.Collections.Tests
{
    public static class CharsSegmentTests
    {
        [Fact]
        public static void GetHashCodeEqualsTest()
        {
            const string testString = "test test";
            var testArray = testString.ToCharArray();
            var first = new CharSegment(testArray, 0, 4);
            var firstHashCode = first.GetHashCode();
            var second = new CharSegment(testArray, 5, 4);
            var secondHashCode = second.GetHashCode();
            Assert.Equal(firstHashCode, secondHashCode);
        }

        [Fact]
        public static void EqualsTest()
        {
            const string testString = "test test";
            var testArray = testString.ToCharArray();
            var first = new CharSegment(testArray, 0, 4);
            var second = new CharSegment(testArray, 5, 4);
            Assert.True(first.Equals(second));
        }
    }
}
```

1.39 ./Platform.Collections.Tests/StringTests.cs

```
using Xunit;

namespace Platform.Collections.Tests
{
    public static class StringTests
    {
        [Fact]
        public static void CapitalizeFirstLetterTest()
        {
            var source1 = "hello";
            var result1 = source1.CapitalizeFirstLetter();
            Assert.Equal("Hello", result1);
            var source2 = "Hello";
            var result2 = source2.CapitalizeFirstLetter();
            Assert.Equal("Hello", result2);
            var source3 = "  hello";
            var result3 = source3.CapitalizeFirstLetter();
            Assert.Equal("  Hello", result3);
        }

        [Fact]
        public static void TrimSingleTest()
        {
            var source1 = "'";
            var result1 = source1.TrimSingle('\'');
            Assert.Equal("", result1);
            var source2 = "''";
            var result2 = source2.TrimSingle('\'');
            Assert.Equal("", result2);
            var source3 = "'hello'";
            var result3 = source3.TrimSingle('\'');
            Assert.Equal("hello", result3);
            var source4 = "hello'";
            var result4 = source4.TrimSingle('\'');
            Assert.Equal("hello", result4);
            var source5 = "'hello";
            var result5 = source5.TrimSingle('\'');
            Assert.Equal("hello", result5);
        }
    }
}
```

Index