1.1 ./csharp/Platform.Data.Doublets.Sequences/Converters/BalancedVariantConverter.cs

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

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

namespace Platform.Data.Doublets.Sequences.Converters
{
    /// <summary>
    /// <para>
    /// Represents the balanced variant converter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksListToSequenceConverterBase{TLinkAddress}"/>
    public class BalancedVariantConverter<TLinkAddress> :
    ↪ LinksListToSequenceConverterBase<TLinkAddress>
    {
        /// <summary>
        /// <para>
        /// Initializes a new <see cref="BalancedVariantConverter"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public BalancedVariantConverter(ILinks<TLinkAddress> links) : base(links) { }

        /// <summary>
        /// <para>
        /// Converts the sequence.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override TLinkAddress Convert(IList<TLinkAddress>? sequence)
        {
            var length = sequence.Count;
            if (length < 1)
            {
                return default;
            }
            if (length == 1)
            {
                return sequence[0];
            }
            // Make copy of next layer
            if (length > 2)
            {
                // TODO: Try to use stackalloc (which at the moment is not working with
                ↪ generics) but will be possible with Sigil
                var halvedSequence = new TLinkAddress[(length / 2) + (length % 2)];
                HalveSequence(halvedSequence, sequence, length);
                sequence = halvedSequence;
                length = halvedSequence.Length;
            }
            // Keep creating layer after layer
            while (length > 2)
            {
                HalveSequence(sequence, sequence, length);
                length = (length / 2) + (length % 2);
            }
            return _links.GetOrCreate(sequence[0], sequence[1]);
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
74          private void HalveSequence(IList<TLinkAddress>? destination, IList<TLinkAddress>?
            ↪  source, int length)
75          {
76              var loopedLength = length - (length % 2);
77              for (var i = 0; i < loopedLength; i += 2)
78              {
79                  destination[i / 2] = _links.GetOrCreate(source[i], source[i + 1]);
80              }
81              if (length > loopedLength)
82              {
83                  destination[length / 2] = source[length - 1];
84              }
85          }
86      }
87  }
```

1.2 ./csharp/Platform.Data.Doublets.Sequences/Converters/CompressingConverter.cs

```
1   using System;
2   using System.Collections.Generic;
3   using System.Runtime.CompilerServices;
4   using Platform.Collections;
5   using Platform.Converters;
6   using Platform.Singletons;
7   using Platform.Numbers;
8   using Platform.Data.Doublets.Sequences.Frequencies.Cache;
9
10  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
11
12  namespace Platform.Data.Doublets.Sequences.Converters
13  {
14      /// <remarks>
15      /// TODO: Возможно будет лучше если алгоритм будет выполняться полностью изолированно от
            ↪  Links на этапе сжатия.
16      ///     А именно будет создаваться временный список пар необходимых для выполнения сжатия, в
            ↪  таком случае тип значения элемента массива может быть любым, как char так и ulong.
17      ///     Как только список/словарь пар был выявлен можно разом выполнить создание всех этих
            ↪  пар, а так же разом выполнить замену.
18      /// </remarks>
19      public class CompressingConverter<TLinkAddress> :
            ↪  LinksListToSequenceConverterBase<TLinkAddress>
20      {
21          private static readonly LinksConstants<TLinkAddress> _constants =
                ↪  Default<LinksConstants<TLinkAddress>>.Instance;
22          private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
                ↪  EqualityComparer<TLinkAddress>.Default;
23          private static readonly Comparer<TLinkAddress> _comparer =
                ↪  Comparer<TLinkAddress>.Default;
24          private static readonly TLinkAddress _zero = default;
25          private static readonly TLinkAddress _one = Arithmetic.Increment(_zero);
26          private readonly IConverter<IList<TLinkAddress>, TLinkAddress> _baseConverter;
27          private readonly LinkFrequenciesCache<TLinkAddress> _doubletFrequenciesCache;
28          private readonly TLinkAddress _minFrequencyToCompress;
29          private readonly bool _doInitialFrequenciesIncrement;
30          private Doublet<TLinkAddress> _maxDoublet;
31          private LinkFrequency<TLinkAddress> _maxDoubletData;
32          private struct HalfDoublet
33          {
34              /// <summary>
35              /// <para>
36              /// The element.
37              /// </para>
38              /// <para></para>
39              /// </summary>
40              public TLinkAddress Element;
41              /// <summary>
42              /// <para>
43              /// The doublet data.
44              /// </para>
45              /// <para></para>
46              /// </summary>
47              public LinkFrequency<TLinkAddress> DoubletData;
48
49              /// <summary>
50              /// <para>
51              /// Initializes a new <see cref="HalfDoublet"/> instance.
52              /// </para>
53              /// <para></para>
54              /// </summary>
55              /// <param name="element">
56              /// <para>A element.</para>
```

```
            /// <para></para>
            /// </param>
            /// <param name="doubletData">
            /// <para>A doublet data.</para>
            /// <para></para>
            /// </param>
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            public HalfDoublet(TLinkAddress element, LinkFrequency<TLinkAddress> doubletData)
            {
                Element = element;
                DoubletData = doubletData;
            }

            /// <summary>
            /// <para>
            /// Returns the string.
            /// </para>
            /// <para></para>
            /// </summary>
            /// <returns>
            /// <para>The string</para>
            /// <para></para>
            /// </returns>
            public override string ToString() => $"{Element}: ({DoubletData})";
        }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="CompressingConverter"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="baseConverter">
        /// <para>A base converter.</para>
        /// <para></para>
        /// </param>
        /// <param name="doubletFrequenciesCache">
        /// <para>A doublet frequencies cache.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public CompressingConverter(ILinks<TLinkAddress> links, IConverter<IList<TLinkAddress>,
        ↪  TLinkAddress> baseConverter, LinkFrequenciesCache<TLinkAddress>
        ↪  doubletFrequenciesCache)
            : this(links, baseConverter, doubletFrequenciesCache, _one, true) { }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="CompressingConverter"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="baseConverter">
        /// <para>A base converter.</para>
        /// <para></para>
        /// </param>
        /// <param name="doubletFrequenciesCache">
        /// <para>A doublet frequencies cache.</para>
        /// <para></para>
        /// </param>
        /// <param name="doInitialFrequenciesIncrement">
        /// <para>A do initial frequencies increment.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public CompressingConverter(ILinks<TLinkAddress> links, IConverter<IList<TLinkAddress>,
        ↪  TLinkAddress> baseConverter, LinkFrequenciesCache<TLinkAddress>
        ↪  doubletFrequenciesCache, bool doInitialFrequenciesIncrement)
            : this(links, baseConverter, doubletFrequenciesCache, _one,
            ↪  doInitialFrequenciesIncrement) { }
```

```
130
131         /// <summary>
132         /// <para>
133         /// Initializes a new <see cref="CompressingConverter"/> instance.
134         /// </para>
135         /// <para></para>
136         /// </summary>
137         /// <param name="links">
138         /// <para>A links.</para>
139         /// <para></para>
140         /// </param>
141         /// <param name="baseConverter">
142         /// <para>A base converter.</para>
143         /// <para></para>
144         /// </param>
145         /// <param name="doubletFrequenciesCache">
146         /// <para>A doublet frequencies cache.</para>
147         /// <para></para>
148         /// </param>
149         /// <param name="minFrequencyToCompress">
150         /// <para>A min frequency to compress.</para>
151         /// <para></para>
152         /// </param>
153         /// <param name="doInitialFrequenciesIncrement">
154         /// <para>A do initial frequencies increment.</para>
155         /// <para></para>
156         /// </param>
157         [MethodImpl(MethodImplOptions.AggressiveInlining)]
158         public CompressingConverter(ILinks<TLinkAddress> links, IConverter<IList<TLinkAddress>,
            ↪ TLinkAddress> baseConverter, LinkFrequenciesCache<TLinkAddress>
            ↪ doubletFrequenciesCache, TLinkAddress minFrequencyToCompress, bool
            ↪ doInitialFrequenciesIncrement)
159             : base(links)
160         {
161             _baseConverter = baseConverter;
162             _doubletFrequenciesCache = doubletFrequenciesCache;
163             if (_comparer.Compare(minFrequencyToCompress, _one) < 0)
164             {
165                 minFrequencyToCompress = _one;
166             }
167             _minFrequencyToCompress = minFrequencyToCompress;
168             _doInitialFrequenciesIncrement = doInitialFrequenciesIncrement;
169             ResetMaxDoublet();
170         }
171
172         /// <summary>
173         /// <para>
174         /// Converts the source.
175         /// </para>
176         /// <para></para>
177         /// </summary>
178         /// <param name="source">
179         /// <para>The source.</para>
180         /// <para></para>
181         /// </param>
182         /// <returns>
183         /// <para>The link</para>
184         /// <para></para>
185         /// </returns>
186         [MethodImpl(MethodImplOptions.AggressiveInlining)]
187         public override TLinkAddress Convert(IList<TLinkAddress>? source) =>
            ↪ _baseConverter.Convert(Compress(source));
188         [MethodImpl(MethodImplOptions.AggressiveInlining)]
189         private IList<TLinkAddress>? Compress(IList<TLinkAddress>? sequence)
190         {
191             if (sequence.IsNullOrEmpty())
192             {
193                 return null;
194             }
195             if (sequence.Count == 1)
196             {
197                 return sequence;
198             }
199             if (sequence.Count == 2)
200             {
201                 return new[] { _links.GetOrCreate(sequence[0], sequence[1]) };
202             }
203             // TODO: arraypool with min size (to improve cache locality) or stackallow with Sigil
```

```
204             var copy = new HalfDoublet[sequence.Count];
205             Doublet<TLinkAddress> doublet = default;
206             for (var i = 1; i < sequence.Count; i++)
207             {
208                 doublet = new Doublet<TLinkAddress>(sequence[i - 1], sequence[i]);
209                 LinkFrequency<TLinkAddress> data;
210                 if (_doInitialFrequenciesIncrement)
211                 {
212                     data = _doubletFrequenciesCache.IncrementFrequency(ref doublet);
213                 }
214                 else
215                 {
216                     data = _doubletFrequenciesCache.GetFrequency(ref doublet);
217                     if (data == null)
218                     {
219                         throw new NotSupportedException("If you ask not to increment
                            ↪ frequencies, it is expected that all frequencies for the sequence
                            ↪ are prepared.");
220                     }
221                 }
222                 copy[i - 1].Element = sequence[i - 1];
223                 copy[i - 1].DoubletData = data;
224                 UpdateMaxDoublet(ref doublet, data);
225             }
226             copy[sequence.Count - 1].Element = sequence[sequence.Count - 1];
227             copy[sequence.Count - 1].DoubletData = new LinkFrequency<TLinkAddress>();
228             if (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
229             {
230                 var newLength = ReplaceDoublets(copy);
231                 sequence = new TLinkAddress[newLength];
232                 for (int i = 0; i < newLength; i++)
233                 {
234                     sequence[i] = copy[i].Element;
235                 }
236             }
237             return sequence;
238         }
239         [MethodImpl(MethodImplOptions.AggressiveInlining)]
240         private int ReplaceDoublets(HalfDoublet[] copy)
241         {
242             var oldLength = copy.Length;
243             var newLength = copy.Length;
244             while (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
245             {
246                 var maxDoubletSource = _maxDoublet.Source;
247                 var maxDoubletTarget = _maxDoublet.Target;
248                 if (_equalityComparer.Equals(_maxDoubletData.Link, _constants.Null))
249                 {
250                     _maxDoubletData.Link = _links.GetOrCreate(maxDoubletSource,
                        ↪ maxDoubletTarget);
251                 }
252                 var maxDoubletReplacementLink = _maxDoubletData.Link;
253                 oldLength--;
254                 var oldLengthMinusTwo = oldLength - 1;
255                 // Substitute all usages
256                 int w = 0, r = 0; // (r == read, w == write)
257                 for (; r < oldLength; r++)
258                 {
259                     if (_equalityComparer.Equals(copy[r].Element, maxDoubletSource) &&
                        ↪ _equalityComparer.Equals(copy[r + 1].Element, maxDoubletTarget))
260                     {
261                         if (r > 0)
262                         {
263                             var previous = copy[w - 1].Element;
264                             copy[w - 1].DoubletData.DecrementFrequency();
265                             copy[w - 1].DoubletData =
                                ↪ _doubletFrequenciesCache.IncrementFrequency(previous,
                                ↪ maxDoubletReplacementLink);
266                         }
267                         if (r < oldLengthMinusTwo)
268                         {
269                             var next = copy[r + 2].Element;
270                             copy[r + 1].DoubletData.DecrementFrequency();
271                             copy[w].DoubletData = _doubletFrequenciesCache.IncrementFrequency(ma
                                ↪ xDoubletReplacementLink,
                                ↪ next);
272                         }
273                         copy[w++].Element = maxDoubletReplacementLink;
```

```
                        r++;
                        newLength--;
                    }
                    else
                    {
                        copy[w++] = copy[r];
                    }
                }
                if (w < newLength)
                {
                    copy[w] = copy[r];
                }
                oldLength = newLength;
                ResetMaxDoublet();
                UpdateMaxDoublet(copy, newLength);
            }
            return newLength;
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void ResetMaxDoublet()
        {
            _maxDoublet = new Doublet<TLinkAddress>();
            _maxDoubletData = new LinkFrequency<TLinkAddress>();
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void UpdateMaxDoublet(HalfDoublet[] copy, int length)
        {
            Doublet<TLinkAddress> doublet = default;
            for (var i = 1; i < length; i++)
            {
                doublet = new Doublet<TLinkAddress>(copy[i - 1].Element, copy[i].Element);
                UpdateMaxDoublet(ref doublet, copy[i - 1].DoubletData);
            }
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private void UpdateMaxDoublet(ref Doublet<TLinkAddress> doublet,
        ↪  LinkFrequency<TLinkAddress> data)
        {
            var frequency = data.Frequency;
            var maxFrequency = _maxDoubletData.Frequency;
            //if (frequency > _minFrequencyToCompress && (maxFrequency < frequency ||
            ↪  (maxFrequency == frequency && doublet.Source + doublet.Target < /* gives better
            ↪  compression string data (and gives collisions quickly) */ _maxDoublet.Source +
            ↪  _maxDoublet.Target)))
            if (_comparer.Compare(frequency, _minFrequencyToCompress) > 0 &&
               (_comparer.Compare(maxFrequency, frequency) < 0 ||
                ↪  (_equalityComparer.Equals(maxFrequency, frequency) &&
                ↪  _comparer.Compare(Arithmetic.Add(doublet.Source, doublet.Target),
                ↪  Arithmetic.Add(_maxDoublet.Source, _maxDoublet.Target)) > 0))) /* gives
                ↪  better stability and better compression on sequent data and even on rundom
                ↪  numbers data (but gives collisions anyway) */
            {
                _maxDoublet = doublet;
                _maxDoubletData = data;
            }
        }
    }
}
```

1.3 ./csharp/Platform.Data.Doublets.Sequences/Converters/LinksListToSequenceConverterBase.cs

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

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

namespace Platform.Data.Doublets.Sequences.Converters
{
    /// <summary>
    /// <para>
    /// Represents the links list to sequence converter base.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksOperatorBase{TLinkAddress}"/>
    /// <seealso cref="IConverter{IList{TLinkAddress}, TLinkAddress}"/>
    public abstract class LinksListToSequenceConverterBase<TLinkAddress> :
    ↪  LinksOperatorBase<TLinkAddress>, IConverter<IList<TLinkAddress>, TLinkAddress>
    {
```

```
        /// <summary>
        /// <para>
        /// Initializes a new <see cref="LinksListToSequenceConverterBase"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected LinksListToSequenceConverterBase(ILinks<TLinkAddress> links) : base(links) { }

        /// <summary>
        /// <para>
        /// Converts the source.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="source">
        /// <para>The source.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public abstract TLinkAddress Convert(IList<TLinkAddress> source);
    }
}
```

1.4 ./csharp/Platform.Data.Doublets.Sequences/Converters/OptimalVariantConverter.cs

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Collections.Lists;
using Platform.Converters;
using Platform.Data.Doublets.Sequences.Frequencies.Cache;
using Platform.Data.Doublets.Sequences.Frequencies.Counters;

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

namespace Platform.Data.Doublets.Sequences.Converters
{
    /// <summary>
    /// <para>
    /// Represents the optimal variant converter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksListToSequenceConverterBase{TLinkAddress}"/>
    public class OptimalVariantConverter<TLinkAddress> :
    ↪ LinksListToSequenceConverterBase<TLinkAddress>
    {
        private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
        ↪ EqualityComparer<TLinkAddress>.Default;
        private static readonly Comparer<TLinkAddress> _comparer =
        ↪ Comparer<TLinkAddress>.Default;
        private readonly IConverter<IList<TLinkAddress>>
        ↪ _sequenceToItsLocalElementLevelsConverter;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="OptimalVariantConverter"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="sequenceToItsLocalElementLevelsConverter">
        /// <para>A sequence to its local element levels converter.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public OptimalVariantConverter(ILinks<TLinkAddress> links,
        ↪ IConverter<IList<TLinkAddress>> sequenceToItsLocalElementLevelsConverter) :
        ↪ base(links)
```

```
        => _sequenceToItsLocalElementLevelsConverter =
        ↪  sequenceToItsLocalElementLevelsConverter;

    /// <summary>
    /// <para>
    /// Initializes a new <see cref="OptimalVariantConverter"/> instance.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <param name="links">
    /// <para>A links.</para>
    /// <para></para>
    /// </param>
    /// <param name="linkFrequenciesCache">
    /// <para>A link frequencies cache.</para>
    /// <para></para>
    /// </param>
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public OptimalVariantConverter(ILinks<TLinkAddress> links,
    ↪  LinkFrequenciesCache<TLinkAddress> linkFrequenciesCache)
        : this(links, new SequenceToItsLocalElementLevelsConverter<TLinkAddress>(links, new
        ↪  FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<TLinkAddress>(linkFrequen
        ↪  ciesCache))) {
        ↪  }

    /// <summary>
    /// <para>
    /// Initializes a new <see cref="OptimalVariantConverter"/> instance.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <param name="links">
    /// <para>A links.</para>
    /// <para></para>
    /// </param>
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public OptimalVariantConverter(ILinks<TLinkAddress> links)
        : this(links, new LinkFrequenciesCache<TLinkAddress>(links, new
        ↪  TotalSequenceSymbolFrequencyCounter<TLinkAddress>(links))) { }

    /// <summary>
    /// <para>
    /// Converts the sequence.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <param name="sequence">
    /// <para>The sequence.</para>
    /// <para></para>
    /// </param>
    /// <returns>
    /// <para>The link</para>
    /// <para></para>
    /// </returns>
    [MethodImpl(MethodImplOptions.AggressiveInlining)]
    public override TLinkAddress Convert(IList<TLinkAddress>? sequence)
    {
        var length = sequence.Count;
        if (length == 1)
        {
            return sequence[0];
        }
        if (length == 2)
        {
            return _links.GetOrCreate(sequence[0], sequence[1]);
        }
        sequence = sequence.ToArray();
        var levels = _sequenceToItsLocalElementLevelsConverter.Convert(sequence);
        while (length > 2)
        {
            var levelRepeat = 1;
            var currentLevel = levels[0];
            var previousLevel = levels[0];
            var skipOnce = false;
            var w = 0;
            for (var i = 1; i < length; i++)
            {
                if (_equalityComparer.Equals(currentLevel, levels[i]))
```

```
                    {
                        levelRepeat++;
                        skipOnce = false;
                        if (levelRepeat == 2)
                        {
                            sequence[w] = _links.GetOrCreate(sequence[i - 1], sequence[i]);
                            var newLevel = i >= length - 1 ?
                                GetPreviousLowerThanCurrentOrCurrent(previousLevel,
                                ↪ currentLevel) :
                                i < 2 ?
                                GetNextLowerThanCurrentOrCurrent(currentLevel, levels[i + 1]) :
                                GetGreatestNeigbourLowerThanCurrentOrCurrent(previousLevel,
                                ↪ currentLevel, levels[i + 1]);
                            levels[w] = newLevel;
                            previousLevel = currentLevel;
                            w++;
                            levelRepeat = 0;
                            skipOnce = true;
                        }
                        else if (i == length - 1)
                        {
                            sequence[w] = sequence[i];
                            levels[w] = levels[i];
                            w++;
                        }
                    }
                    else
                    {
                        currentLevel = levels[i];
                        levelRepeat = 1;
                        if (skipOnce)
                        {
                            skipOnce = false;
                        }
                        else
                        {
                            sequence[w] = sequence[i - 1];
                            levels[w] = levels[i - 1];
                            previousLevel = levels[w];
                            w++;
                        }
                        if (i == length - 1)
                        {
                            sequence[w] = sequence[i];
                            levels[w] = levels[i];
                            w++;
                        }
                    }
                }
                length = w;
            }
            return _links.GetOrCreate(sequence[0], sequence[1]);
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static TLinkAddress GetGreatestNeigbourLowerThanCurrentOrCurrent(TLinkAddress
        ↪ previous, TLinkAddress current, TLinkAddress next)
        {
            return _comparer.Compare(previous, next) > 0
                ? _comparer.Compare(previous, current) < 0 ? previous : current
                : _comparer.Compare(next, current) < 0 ? next : current;
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static TLinkAddress GetNextLowerThanCurrentOrCurrent(TLinkAddress current,
        ↪ TLinkAddress next) => _comparer.Compare(next, current) < 0 ? next : current;
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static TLinkAddress GetPreviousLowerThanCurrentOrCurrent(TLinkAddress previous,
        ↪ TLinkAddress current) => _comparer.Compare(previous, current) < 0 ? previous :
        ↪ current;
    }
}
```

1.5 ./csharp/Platform.Data.Doublets.Sequences/Converters/SequenceToItsLocalElementLevelsConverter.cs

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

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

namespace Platform.Data.Doublets.Sequences.Converters
```

```
{
    /// <summary>
    /// <para>
    /// Represents the sequence to its local element levels converter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksOperatorBase{TLinkAddress}"/>
    /// <seealso cref="IConverter{IList{TLinkAddress}}"/>
    public class SequenceToItsLocalElementLevelsConverter<TLinkAddress> :
    ↪ LinksOperatorBase<TLinkAddress>, IConverter<IList<TLinkAddress>>
    {
        private static readonly Comparer<TLinkAddress> _comparer =
        ↪ Comparer<TLinkAddress>.Default;
        private readonly IConverter<Doublet<TLinkAddress>, TLinkAddress>
        ↪ _linkToItsFrequencyToNumberConveter;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="SequenceToItsLocalElementLevelsConverter"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="linkToItsFrequencyToNumberConveter">
        /// <para>A link to its frequency to number conveter.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public SequenceToItsLocalElementLevelsConverter(ILinks<TLinkAddress> links,
        ↪ IConverter<Doublet<TLinkAddress>, TLinkAddress> linkToItsFrequencyToNumberConveter)
        ↪ : base(links) => _linkToItsFrequencyToNumberConveter =
        ↪ linkToItsFrequencyToNumberConveter;

        /// <summary>
        /// <para>
        /// Converts the sequence.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The levels.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public IList<TLinkAddress>? Convert(IList<TLinkAddress>? sequence)
        {
            var levels = new TLinkAddress[sequence.Count];
            levels[0] = GetFrequencyNumber(sequence[0], sequence[1]);
            for (var i = 1; i < sequence.Count - 1; i++)
            {
                var previous = GetFrequencyNumber(sequence[i - 1], sequence[i]);
                var next = GetFrequencyNumber(sequence[i], sequence[i + 1]);
                levels[i] = _comparer.Compare(previous, next) > 0 ? previous : next;
            }
            levels[levels.Length - 1] = GetFrequencyNumber(sequence[sequence.Count - 2],
            ↪ sequence[sequence.Count - 1]);
            return levels;
        }

        /// <summary>
        /// <para>
        /// Gets the frequency number using the specified source.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="source">
        /// <para>The source.</para>
        /// <para></para>
        /// </param>
        /// <param name="target">
```

```
        /// <para>The target.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLinkAddress GetFrequencyNumber(TLinkAddress source, TLinkAddress target) =>
        ↪ _linkToItsFrequencyToNumberConveter.Convert(new Doublet<TLinkAddress>(source,
        ↪ target));
    }
}
```

1.6 ./csharp/Platform.Data.Doublets.Sequences/CriterionMatchers/DefaultSequenceElementCriterionMatcher.cs

```
using System.Runtime.CompilerServices;
using Platform.Interfaces;

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

namespace Platform.Data.Doublets.Sequences.CriterionMatchers
{
    /// <summary>
    /// <para>
    /// Represents the default sequence element criterion matcher.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksOperatorBase{TLinkAddress}"/>
    /// <seealso cref="ICriterionMatcher{TLinkAddress}"/>
    public class DefaultSequenceElementCriterionMatcher<TLinkAddress> :
    ↪ LinksOperatorBase<TLinkAddress>, ICriterionMatcher<TLinkAddress>
    {
        /// <summary>
        /// <para>
        /// Initializes a new <see cref="DefaultSequenceElementCriterionMatcher"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public DefaultSequenceElementCriterionMatcher(ILinks<TLinkAddress> links) : base(links)
        ↪ { }

        /// <summary>
        /// <para>
        /// Determines whether this instance is matched.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="argument">
        /// <para>The argument.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool IsMatched(TLinkAddress argument) => _links.IsPartialPoint(argument);
    }
}
```

1.7 ./csharp/Platform.Data.Doublets.Sequences/CriterionMatchers/MarkedSequenceCriterionMatcher.cs

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

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

namespace Platform.Data.Doublets.Sequences.CriterionMatchers
{
    /// <summary>
    /// <para>
    /// Represents the marked sequence criterion matcher.
```

```
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="ICriterionMatcher{TLinkAddress}"/>
    public class MarkedSequenceCriterionMatcher<TLinkAddress> : ICriterionMatcher<TLinkAddress>
    {
        private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
        ↪  EqualityComparer<TLinkAddress>.Default;
        private readonly ILinks<TLinkAddress> _links;
        private readonly TLinkAddress _sequenceMarkerLink;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="MarkedSequenceCriterionMatcher"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="sequenceMarkerLink">
        /// <para>A sequence marker link.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public MarkedSequenceCriterionMatcher(ILinks<TLinkAddress> links, TLinkAddress
        ↪  sequenceMarkerLink)
        {
            _links = links;
            _sequenceMarkerLink = sequenceMarkerLink;
        }

        /// <summary>
        /// <para>
        /// Determines whether this instance is matched.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequenceCandidate">
        /// <para>The sequence candidate.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool IsMatched(TLinkAddress sequenceCandidate)
            => _equalityComparer.Equals(_links.GetSource(sequenceCandidate), _sequenceMarkerLink)
            || !_equalityComparer.Equals(_links.SearchOrDefault(_sequenceMarkerLink,
            ↪  sequenceCandidate), _links.Constants.Null);
    }
}
```

1.8 ./csharp/Platform.Data.Doublets.Sequences/CriterionMatchers/UnicodeSequenceMatcher.cs

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

namespace Platform.Data.Doublets.Sequences.CriterionMatchers;

public class UnicodeSequenceMatcher<TLinkAddress> : ICriterionMatcher<TLinkAddress>
{
    public readonly ILinks<TLinkAddress> Storage;
    public readonly TLinkAddress UnicodeSequenceMarker;
    public readonly EqualityComparer<TLinkAddress> EqualityComparer =
    ↪  EqualityComparer<TLinkAddress>.Default;
    public UnicodeSequenceMatcher(ILinks<TLinkAddress> storage, TLinkAddress
    ↪  unicodeSequenceMarker)
    {
        Storage = storage;
        UnicodeSequenceMarker = unicodeSequenceMarker;
    }
    public bool IsMatched(TLinkAddress argument)
    {
        var target = Storage.GetTarget(argument);
        return EqualityComparer.Equals(UnicodeSequenceMarker, argument) ||
        ↪  EqualityComparer.Equals(UnicodeSequenceMarker, target);
    }
```

```
}
```

1.9 ./csharp/Platform.Data.Doublets.Sequences/DefaultSequenceAppender.cs

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Collections.Stacks;
using Platform.Data.Doublets.Sequences.HeightProviders;
using Platform.Data.Sequences;

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

namespace Platform.Data.Doublets.Sequences
{
    /// <summary>
    /// <para>
    /// Represents the default sequence appender.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksOperatorBase{TLinkAddress}"/>
    /// <seealso cref="ISequenceAppender{TLinkAddress}"/>
    public class DefaultSequenceAppender<TLinkAddress> : LinksOperatorBase<TLinkAddress>,
    ↪  ISequenceAppender<TLinkAddress>
    {
        private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
        ↪  EqualityComparer<TLinkAddress>.Default;
        private readonly IStack<TLinkAddress> _stack;
        private readonly ISequenceHeightProvider<TLinkAddress> _heightProvider;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="DefaultSequenceAppender"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="stack">
        /// <para>A stack.</para>
        /// <para></para>
        /// </param>
        /// <param name="heightProvider">
        /// <para>A height provider.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public DefaultSequenceAppender(ILinks<TLinkAddress> links, IStack<TLinkAddress> stack,
        ↪  ISequenceHeightProvider<TLinkAddress> heightProvider)
            : base(links)
        {
            _stack = stack;
            _heightProvider = heightProvider;
        }

        /// <summary>
        /// <para>
        /// Appends the sequence.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <param name="appendant">
        /// <para>The appendant.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLinkAddress Append(TLinkAddress sequence, TLinkAddress appendant)
        {
            var cursor = sequence;
            var links = _links;
```

```
            while (!_equalityComparer.Equals(_heightProvider.Get(cursor), default))
            {
                var source = links.GetSource(cursor);
                var target = links.GetTarget(cursor);
                if (_equalityComparer.Equals(_heightProvider.Get(source),
                ↪  _heightProvider.Get(target)))
                {
                    break;
                }
                else
                {
                    _stack.Push(source);
                    cursor = target;
                }
            }
            var left = cursor;
            var right = appendant;
            while (!_equalityComparer.Equals(cursor = _stack.PopOrDefault(),
            ↪  links.Constants.Null))
            {
                right = links.GetOrCreate(left, right);
                left = cursor;
            }
            return links.GetOrCreate(left, right);
        }
    }
}
```

1.10 ./csharp/Platform.Data.Doublets.Sequences/DuplicateSegmentsCounter.cs

```
using System.Collections.Generic;
using System.Linq;
using System.Runtime.CompilerServices;
using Platform.Interfaces;

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

namespace Platform.Data.Doublets.Sequences
{
    /// <summary>
    /// <para>
    /// Represents the duplicate segments counter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="ICounter{int}"/>
    public class DuplicateSegmentsCounter<TLinkAddress> : ICounter<int>
    {
        private readonly IProvider<IList<KeyValuePair<IList<TLinkAddress>?,
        ↪  IList<TLinkAddress>?>>> _duplicateFragmentsProvider;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="DuplicateSegmentsCounter"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="duplicateFragmentsProvider">
        /// <para>A duplicate fragments provider.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public DuplicateSegmentsCounter(IProvider<IList<KeyValuePair<IList<TLinkAddress>?,
        ↪  IList<TLinkAddress>?>>> duplicateFragmentsProvider) => _duplicateFragmentsProvider =
        ↪  duplicateFragmentsProvider;

        /// <summary>
        /// <para>
        /// Counts this instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The int</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public int Count() => _duplicateFragmentsProvider.Get().Sum(x => x.Value.Count);
    }
}
```

1.11 ./csharp/Platform.Data.Doublets.Sequences/DuplicateSegmentsProvider.cs

```
// using System;
// using System.Linq;
// using System.Collections.Generic;
// using System.Runtime.CompilerServices;
// using Platform.Interfaces;
// using Platform.Collections;
// using Platform.Collections.Lists;
// using Platform.Collections.Segments;
// using Platform.Collections.Segments.Walkers;
// using Platform.Singletons;
// using Platform.Converters;
// using Platform.Data.Doublets.Unicode;
//
// #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
//
// namespace Platform.Data.Doublets.Sequences
// {
//     /// <summary>
//     /// <para>
//     /// Represents the duplicate segments provider.
//     /// </para>
//     /// <para></para>
//     /// </summary>
//     /// <seealso cref="DictionaryBasedDuplicateSegmentsWalkerBase{TLinkAddress}"/>
//     /// <seealso cref="IProvider{IList{KeyValuePair{IList{TLinkAddress},
↪  IList{TLinkAddress}}}}"/>
//     public class DuplicateSegmentsProvider<TLinkAddress> :
↪  DictionaryBasedDuplicateSegmentsWalkerBase<TLinkAddress>,
↪  IProvider<IList<KeyValuePair<IList<TLinkAddress>?, IList<TLinkAddress>?>>>
//     {
//         private static readonly UncheckedConverter<TLinkAddress, long>
↪  _addressToInt64Converter = UncheckedConverter<TLinkAddress, long>.Default;
//         private static readonly UncheckedConverter<TLinkAddress, ulong>
↪  _addressToUInt64Converter = UncheckedConverter<TLinkAddress, ulong>.Default;
//         private static readonly UncheckedConverter<ulong, TLinkAddress>
↪  _uInt64ToAddressConverter = UncheckedConverter<ulong, TLinkAddress>.Default;
//         private readonly ILinks<TLinkAddress> _links;
//         private readonly ILinks<TLinkAddress> _sequences;
//         private HashSet<KeyValuePair<IList<TLinkAddress>?, IList<TLinkAddress>?>> _groups;
//         private BitString _visited;
//         private class ItemEquilityComparer :
↪  IEqualityComparer<KeyValuePair<IList<TLinkAddress>?, IList<TLinkAddress>?>>
//         {
//             private readonly IListEqualityComparer<TLinkAddress> _listComparer;
//
//             /// <summary>
//             /// <para>
//             /// Initializes a new <see cref="ItemEquilityComparer"/> instance.
//             /// </para>
//             /// <para></para>
//             /// </summary>
//             public ItemEquilityComparer() => _listComparer =
↪  Default<IListEqualityComparer<TLinkAddress>>.Instance;
//
//             /// <summary>
//             /// <para>
//             /// Determines whether this instance equals.
//             /// </para>
//             /// <para></para>
//             /// </summary>
//             /// <param name="left">
//             /// <para>The left.</para>
//             /// <para></para>
//             /// </param>
//             /// <param name="right">
//             /// <para>The right.</para>
//             /// <para></para>
//             /// </param>
//             /// <returns>
//             /// <para>The bool</para>
//             /// <para></para>
//             /// </returns>
//             [MethodImpl(MethodImplOptions.AggressiveInlining)]
//             public bool Equals(KeyValuePair<IList<TLinkAddress>?, IList<TLinkAddress>?> left,
↪  KeyValuePair<IList<TLinkAddress>?, IList<TLinkAddress>?> right) =>
↪  _listComparer.Equals(left.Key, right.Key) && _listComparer.Equals(left.Value, right.Value);
```

```
//
//             /// <summary>
//             /// <para>
//             /// Gets the hash code using the specified pair.
//             /// </para>
//             /// <para></para>
//             /// </summary>
//             /// <param name="pair">
//             /// <para>The pair.</para>
//             /// <para></para>
//             /// </param>
//             /// <returns>
//             /// <para>The int</para>
//             /// <para></para>
//             /// </returns>
//             [MethodImpl(MethodImplOptions.AggressiveInlining)]
//             public int GetHashCode(KeyValuePair<IList<TLinkAddress>?, IList<TLinkAddress>?>
    ↪ pair) => (_listComparer.GetHashCode(pair.Key),
    ↪ _listComparer.GetHashCode(pair.Value)).GetHashCode();
//         }
//         private class ItemComparer : IComparer<KeyValuePair<IList<TLinkAddress>?,
    ↪ IList<TLinkAddress>?>>
//         {
//             private readonly IListComparer<TLinkAddress> _listComparer;
//
//             /// <summary>
//             /// <para>
//             /// Initializes a new <see cref="ItemComparer"/> instance.
//             /// </para>
//             /// <para></para>
//             /// </summary>
//             [MethodImpl(MethodImplOptions.AggressiveInlining)]
//             public ItemComparer() => _listComparer =
    ↪ Default<IListComparer<TLinkAddress>>.Instance;
//
//             /// <summary>
//             /// <para>
//             /// Compares the left.
//             /// </para>
//             /// <para></para>
//             /// </summary>
//             /// <param name="left">
//             /// <para>The left.</para>
//             /// <para></para>
//             /// </param>
//             /// <param name="right">
//             /// <para>The right.</para>
//             /// <para></para>
//             /// </param>
//             /// <returns>
//             /// <para>The intermediate result.</para>
//             /// <para></para>
//             /// </returns>
//             [MethodImpl(MethodImplOptions.AggressiveInlining)]
//             public int Compare(KeyValuePair<IList<TLinkAddress>?, IList<TLinkAddress>?> left,
    ↪ KeyValuePair<IList<TLinkAddress>?, IList<TLinkAddress>?> right)
//             {
//                 var intermediateResult = _listComparer.Compare(left.Key, right.Key);
//                 if (intermediateResult == 0)
//                 {
//                     intermediateResult = _listComparer.Compare(left.Value, right.Value);
//                 }
//                 return intermediateResult;
//             }
//         }
//
//         /// <summary>
//         /// <para>
//         /// Initializes a new <see cref="DuplicateSegmentsProvider"/> instance.
//         /// </para>
//         /// <para></para>
//         /// </summary>
//         /// <param name="links">
//         /// <para>A links.</para>
//         /// <para></para>
//         /// </param>
//         /// <param name="sequences">
```

```
//         /// <para>A sequences.</para>
//         /// <para></para>
//         /// </param>
//         [MethodImpl(MethodImplOptions.AggressiveInlining)]
//         public DuplicateSegmentsProvider(ILinks<TLinkAddress> links, ILinks<TLinkAddress>
    ↪ sequences)
//             : base(minimumStringSegmentLength: 2)
//         {
//             _links = links;
//             _sequences = sequences;
//         }
//
//         /// <summary>
//         /// <para>
//         /// Gets this instance.
//         /// </para>
//         /// <para></para>
//         /// </summary>
//         /// <returns>
//         /// <para>The result list.</para>
//         /// <para></para>
//         /// </returns>
//         [MethodImpl(MethodImplOptions.AggressiveInlining)]
//         public IList<KeyValuePair<IList<TLinkAddress>?, IList<TLinkAddress>?>> Get()
//         {
//             _groups = new HashSet<KeyValuePair<IList<TLinkAddress>?,
    ↪ IList<TLinkAddress>?>>(Default<ItemEquilityComparer>.Instance);
//             var links = _links;
//             var count = links.Count();
//             _visited = new BitString(_addressToInt64Converter.Convert(count) + 1L);
//             links.Each(link =>
//             {
//                 var linkIndex = links.GetIndex(link);
//                 var linkBitIndex = _addressToInt64Converter.Convert(linkIndex);
//                 var constants = links.Constants;
//                 if (!_visited.Get(linkBitIndex))
//                 {
//                     var sequenceElements = new List<TLinkAddress>();
//                     var filler = new ListFiller<TLinkAddress, TLinkAddress>(sequenceElements,
    ↪ constants.Break);
//                     _sequences.Each(filler.AddSkipFirstAndReturnConstant, new
    ↪ LinkAddress<TLinkAddress>(linkIndex));
//                     if (sequenceElements.Count > 2)
//                     {
//                         WalkAll(sequenceElements);
//                     }
//                 }
//                 return constants.Continue;
//             });
//             var resultList = _groups.ToList();
//             var comparer = Default<ItemComparer>.Instance;
//             resultList.Sort(comparer);
// #if DEBUG
//             foreach (var item in resultList)
//             {
//                 PrintDuplicates(item);
//             }
// #endif
//             return resultList;
//         }
//
//         /// <summary>
//         /// <para>
//         /// Creates the segment using the specified elements.
//         /// </para>
//         /// <para></para>
//         /// </summary>
//         /// <param name="elements">
//         /// <para>The elements.</para>
//         /// <para></para>
//         /// </param>
//         /// <param name="offset">
//         /// <para>The offset.</para>
//         /// <para></para>
//         /// </param>
//         /// <param name="length">
//         /// <para>The length.</para>
```

```
//         /// <para></para>
//         /// </param>
//         /// <returns>
//         /// <para>A segment of t link</para>
//         /// <para></para>
//         /// </returns>
//         [MethodImpl(MethodImplOptions.AggressiveInlining)]
//         protected override Segment<TLinkAddress> CreateSegment(IList<TLinkAddress>? elements,
    ↪ int offset, int length) => new Segment<TLinkAddress>(elements, offset, length);
//
//         /// <summary>
//         /// <para>
//         /// Ons the dublicate found using the specified segment.
//         /// </para>
//         /// <para></para>
//         /// </summary>
//         /// <param name="segment">
//         /// <para>The segment.</para>
//         /// <para></para>
//         /// </param>
//         [MethodImpl(MethodImplOptions.AggressiveInlining)]
//         protected override void OnDublicateFound(Segment<TLinkAddress> segment)
//         {
//             var duplicates = CollectDuplicatesForSegment(segment);
//             if (duplicates.Count > 1)
//             {
//                 _groups.Add(new KeyValuePair<IList<TLinkAddress>?,
    ↪ IList<TLinkAddress>?>(segment.ToArray(), duplicates));
//             }
//         }
//         [MethodImpl(MethodImplOptions.AggressiveInlining)]
//         private List<TLinkAddress> CollectDuplicatesForSegment(Segment<TLinkAddress> segment)
//         {
//             var duplicates = new List<TLinkAddress>();
//             var readAsElement = new HashSet<TLinkAddress>();
//             var restrictions = segment.ShiftRight();
//             var constants = _links.Constants;
//             restrictions[0] = constants.Any;
//             _sequences.Each(restrictions, sequence =>
//             {
//                 var sequenceIndex = sequence[constants.IndexPart];
//                 duplicates.Add(sequenceIndex);
//                 readAsElement.Add(sequenceIndex);
//                 return constants.Continue;
//             });
//             if (duplicates.Any(x => _visited.Get(_addressToInt64Converter.Convert(x))))
//             {
//                 return new List<TLinkAddress>();
//             }
//             foreach (var duplicate in duplicates)
//             {
//                 var duplicateBitIndex = _addressToInt64Converter.Convert(duplicate);
//                 _visited.Set(duplicateBitIndex);
//             }
//             if (_sequences is Sequences sequencesExperiments)
//             {
//                 var partiallyMatched = sequencesExperiments.GetAllPartiallyMatchingSequences4(
    ↪ ((HashSet<ulong>)(object)readAsElement,
    ↪ (IList<ulong>)segment);
//                 foreach (var partiallyMatchedSequence in partiallyMatched)
//                 {
//                     var sequenceIndex =
    ↪ _uInt64ToAddressConverter.Convert(partiallyMatchedSequence);
//                     duplicates.Add(sequenceIndex);
//                 }
//             }
//             duplicates.Sort();
//             return duplicates;
//         }
//         [MethodImpl(MethodImplOptions.AggressiveInlining)]
//         private void PrintDuplicates(KeyValuePair<IList<TLinkAddress>?, IList<TLinkAddress>?>
    ↪ duplicatesItem)
//         {
//             if (!(_links is ILinks<ulong> ulongLinks))
//             {
//                 return;
//             }
```

```
283 //             var duplicatesKey = duplicatesItem.Key;
284 //             var keyString = UnicodeMap.FromLinksToString((IList<ulong>)duplicatesKey);
285 //             Console.WriteLine($"> {keyString} ({string.Join(", ", duplicatesKey)})");
286 //             var duplicatesList = duplicatesItem.Value;
287 //             for (int i = 0; i < duplicatesList.Count; i++)
288 //             {
289 //                 var sequenceIndex = _addressToUInt64Converter.Convert(duplicatesList[i]);
290 //                 var formatedSequenceStructure = ulongLinks.FormatStructure(sequenceIndex, x
    ↪  => Point<ulong>.IsPartialPoint(x), (sb, link) => _ = UnicodeMap.IsCharLink(link.Index) ?
    ↪  sb.Append(UnicodeMap.FromLinkToChar(link.Index)) : sb.Append(link.Index));
291 //                 Console.WriteLine(formatedSequenceStructure);
292 //                 var sequenceString = UnicodeMap.FromSequenceLinkToString(sequenceIndex,
    ↪  ulongLinks);
293 //                 Console.WriteLine(sequenceString);
294 //             }
295 //             Console.WriteLine();
296 //         }
297 //     }
298 // }
```

1.12 ./csharp/Platform.Data.Doublets.Sequences/Frequencies/Cache/LinkFrequenciesCache.cs

```
1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Interfaces;
5  using Platform.Numbers;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
10 {
11     /// <remarks>
12     /// Can be used to operate with many CompressingConverters (to keep global frequencies data
       ↪  between them).
13     /// TODO: Extract interface to implement frequencies storage inside Links storage
14     /// </remarks>
15     public class LinkFrequenciesCache<TLinkAddress> : LinksOperatorBase<TLinkAddress>
16     {
17         private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
           ↪  EqualityComparer<TLinkAddress>.Default;
18         private static readonly Comparer<TLinkAddress> _comparer =
           ↪  Comparer<TLinkAddress>.Default;
19         private static readonly TLinkAddress _zero = default;
20         private static readonly TLinkAddress _one = Arithmetic.Increment(_zero);
21         private readonly Dictionary<Doublet<TLinkAddress>, LinkFrequency<TLinkAddress>>
           ↪  _doubletsCache;
22         private readonly ICounter<TLinkAddress, TLinkAddress> _frequencyCounter;
23
24         /// <summary>
25         /// <para>
26         /// Initializes a new <see cref="LinkFrequenciesCache"/> instance.
27         /// </para>
28         /// <para></para>
29         /// </summary>
30         /// <param name="links">
31         /// <para>A links.</para>
32         /// <para></para>
33         /// </param>
34         /// <param name="frequencyCounter">
35         /// <para>A frequency counter.</para>
36         /// <para></para>
37         /// </param>
38         [MethodImpl(MethodImplOptions.AggressiveInlining)]
39         public LinkFrequenciesCache(ILinks<TLinkAddress> links, ICounter<TLinkAddress,
           ↪  TLinkAddress> frequencyCounter)
40             : base(links)
41         {
42             _doubletsCache = new Dictionary<Doublet<TLinkAddress>,
               ↪  LinkFrequency<TLinkAddress>>(4096, DoubletComparer<TLinkAddress>.Default);
43             _frequencyCounter = frequencyCounter;
44         }
45
46         /// <summary>
47         /// <para>
48         /// Gets the frequency using the specified source.
49         /// </para>
50         /// <para></para>
51         /// </summary>
52         /// <param name="source">
```

```
        /// <para>The source.</para>
        /// <para></para>
        /// </param>
        /// <param name="target">
        /// <para>The target.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>A link frequency of t link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinkFrequency<TLinkAddress> GetFrequency(TLinkAddress source, TLinkAddress target)
        {
            var doublet = new Doublet<TLinkAddress>(source, target);
            return GetFrequency(ref doublet);
        }

        /// <summary>
        /// <para>
        /// Gets the frequency using the specified doublet.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="doublet">
        /// <para>The doublet.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The data.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinkFrequency<TLinkAddress> GetFrequency(ref Doublet<TLinkAddress> doublet)
        {
            _doubletsCache.TryGetValue(doublet, out LinkFrequency<TLinkAddress> data);
            return data;
        }

        /// <summary>
        /// <para>
        /// Increments the frequencies using the specified sequence.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void IncrementFrequencies(IList<TLinkAddress>? sequence)
        {
            for (var i = 1; i < sequence.Count; i++)
            {
                IncrementFrequency(sequence[i - 1], sequence[i]);
            }
        }

        /// <summary>
        /// <para>
        /// Increments the frequency using the specified source.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="source">
        /// <para>The source.</para>
        /// <para></para>
        /// </param>
        /// <param name="target">
        /// <para>The target.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>A link frequency of t link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
130         public LinkFrequency<TLinkAddress> IncrementFrequency(TLinkAddress source, TLinkAddress
            ↪ target)
131         {
132             var doublet = new Doublet<TLinkAddress>(source, target);
133             return IncrementFrequency(ref doublet);
134         }
135 
136         /// <summary>
137         /// <para>
138         /// Prints the frequencies using the specified sequence.
139         /// </para>
140         /// <para></para>
141         /// </summary>
142         /// <param name="sequence">
143         /// <para>The sequence.</para>
144         /// <para></para>
145         /// </param>
146         [MethodImpl(MethodImplOptions.AggressiveInlining)]
147         public void PrintFrequencies(IList<TLinkAddress>? sequence)
148         {
149             for (var i = 1; i < sequence.Count; i++)
150             {
151                 PrintFrequency(sequence[i - 1], sequence[i]);
152             }
153         }
154 
155         /// <summary>
156         /// <para>
157         /// Prints the frequency using the specified source.
158         /// </para>
159         /// <para></para>
160         /// </summary>
161         /// <param name="source">
162         /// <para>The source.</para>
163         /// <para></para>
164         /// </param>
165         /// <param name="target">
166         /// <para>The target.</para>
167         /// <para></para>
168         /// </param>
169         [MethodImpl(MethodImplOptions.AggressiveInlining)]
170         public void PrintFrequency(TLinkAddress source, TLinkAddress target)
171         {
172             var number = GetFrequency(source, target).Frequency;
173             Console.WriteLine("({0},{1}) - {2}", source, target, number);
174         }
175 
176         /// <summary>
177         /// <para>
178         /// Increments the frequency using the specified doublet.
179         /// </para>
180         /// <para></para>
181         /// </summary>
182         /// <param name="doublet">
183         /// <para>The doublet.</para>
184         /// <para></para>
185         /// </param>
186         /// <returns>
187         /// <para>The data.</para>
188         /// <para></para>
189         /// </returns>
190         [MethodImpl(MethodImplOptions.AggressiveInlining)]
191         public LinkFrequency<TLinkAddress> IncrementFrequency(ref Doublet<TLinkAddress> doublet)
192         {
193             if (_doubletsCache.TryGetValue(doublet, out LinkFrequency<TLinkAddress> data))
194             {
195                 data.IncrementFrequency();
196             }
197             else
198             {
199                 var link = _links.SearchOrDefault(doublet.Source, doublet.Target);
200                 data = new LinkFrequency<TLinkAddress>(_one, link);
201                 if (!_equalityComparer.Equals(link, default))
202                 {
203                     data.Frequency = Arithmetic.Add(data.Frequency,
                        ↪ _frequencyCounter.Count(link));
204                 }
205                 _doubletsCache.Add(doublet, data);
```

```
            }
            return data;
        }

        /// <summary>
        /// <para>
        /// Validates the frequencies.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <exception cref="InvalidOperationException">
        /// <para>Frequencies validation failed.</para>
        /// <para></para>
        /// </exception>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void ValidateFrequencies()
        {
            foreach (var entry in _doubletsCache)
            {
                var value = entry.Value;
                var linkIndex = value.Link;
                if (!_equalityComparer.Equals(linkIndex, default))
                {
                    var frequency = value.Frequency;
                    var count = _frequencyCounter.Count(linkIndex);
                    // TODO: Why `frequency` always greater than `count` by 1?
                    if (((_comparer.Compare(frequency, count) > 0) &&
                     ↪  (_comparer.Compare(Arithmetic.Subtract(frequency, count), _one) > 0))
                      || ((_comparer.Compare(count, frequency) > 0) &&
                       ↪  (_comparer.Compare(Arithmetic.Subtract(count, frequency), _one) > 0)))
                    {
                        throw new InvalidOperationException("Frequencies validation failed.");
                    }
                }
                //else
                //{
                //    if (value.Frequency > 0)
                //    {
                //        var frequency = value.Frequency;
                //        linkIndex = _createLink(entry.Key.Source, entry.Key.Target);
                //        var count = _countLinkFrequency(linkIndex);

                //        if ((frequency > count && frequency - count > 1) || (count > frequency
                ↪  && count - frequency > 1))
                //            throw new InvalidOperationException("Frequencies validation
                ↪  failed.");
                //    }
                //}
            }
        }
    }
}
```

1.13 ./csharp/Platform.Data.Doublets.Sequences/Frequencies/Cache/LinkFrequency.cs

```
using System.Runtime.CompilerServices;
using Platform.Numbers;

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

namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
{
    /// <summary>
    /// <para>
    /// Represents the link frequency.
    /// </para>
    /// <para></para>
    /// </summary>
    public class LinkFrequency<TLinkAddress>
    {
        /// <summary>
        /// <para>
        /// Gets or sets the frequency value.
        /// </para>
        /// <para></para>
        /// </summary>
        public TLinkAddress Frequency { get; set; }
        /// <summary>
        /// <para>
```

```
        /// Gets or sets the link value.
        /// </para>
        /// <para></para>
        /// </summary>
        public TLinkAddress Link { get; set; }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="LinkFrequency"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="frequency">
        /// <para>A frequency.</para>
        /// <para></para>
        /// </param>
        /// <param name="link">
        /// <para>A link.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinkFrequency(TLinkAddress frequency, TLinkAddress link)
        {
            Frequency = frequency;
            Link = link;
        }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="LinkFrequency"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LinkFrequency() { }

        /// <summary>
        /// <para>
        /// Increments the frequency.
        /// </para>
        /// <para></para>
        /// </summary>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void IncrementFrequency() => Frequency =
        ↪  Arithmetic<TLinkAddress>.Increment(Frequency);

        /// <summary>
        /// <para>
        /// Decrements the frequency.
        /// </para>
        /// <para></para>
        /// </summary>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void DecrementFrequency() => Frequency =
        ↪  Arithmetic<TLinkAddress>.Decrement(Frequency);

        /// <summary>
        /// <para>
        /// Returns the string.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The string</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override string ToString() => $"F: {Frequency}, L: {Link}";
    }
}
```

1.14 ./csharp/Platform.Data.Doublets.Sequences/Frequencies/Cache/LinkToItsFrequencyValueConverter.cs

```
using System.Runtime.CompilerServices;
using Platform.Converters;

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

namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
```

```
{
    /// <summary>
    /// <para>
    /// Represents the frequencies cache based link to its frequency number converter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="IConverter{Doublet{TLinkAddress}, TLinkAddress}"/>
    public class FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<TLinkAddress> :
    ↪ IConverter<Doublet<TLinkAddress>, TLinkAddress>
    {
        private readonly LinkFrequenciesCache<TLinkAddress> _cache;

        /// <summary>
        /// <para>
        /// Initializes a new <see
        ↪ cref="FrequenciesCacheBasedLinkToItsFrequencyNumberConverter"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="cache">
        /// <para>A cache.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public FrequenciesCacheBasedLinkToItsFrequencyNumberConverter(LinkFrequenciesCache<TLink」
        ↪ Address> cache) => _cache =
        ↪ cache;

        /// <summary>
        /// <para>
        /// Converts the source.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="source">
        /// <para>The source.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLinkAddress Convert(Doublet<TLinkAddress> source) => _cache.GetFrequency(ref
        ↪ source).Frequency;
    }
}
```

1.15 ./csharp/Platform.Data.Doublets.Sequences/Frequencies/Counters/MarkedSequenceSymbolFrequencyOneOf

```
using System.Runtime.CompilerServices;
using Platform.Interfaces;

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

namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
{
    /// <summary>
    /// <para>
    /// Represents the marked sequence symbol frequency one off counter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="SequenceSymbolFrequencyOneOffCounter{TLinkAddress}"/>
    public class MarkedSequenceSymbolFrequencyOneOffCounter<TLinkAddress> :
    ↪ SequenceSymbolFrequencyOneOffCounter<TLinkAddress>
    {
        private readonly ICriterionMatcher<TLinkAddress> _markedSequenceMatcher;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="MarkedSequenceSymbolFrequencyOneOffCounter"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
```

```
        /// </param>
        /// <param name="markedSequenceMatcher">
        /// <para>A marked sequence matcher.</para>
        /// <para></para>
        /// </param>
        /// <param name="sequenceLink">
        /// <para>A sequence link.</para>
        /// <para></para>
        /// </param>
        /// <param name="symbol">
        /// <para>A symbol.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public MarkedSequenceSymbolFrequencyOneOffCounter(ILinks<TLinkAddress> links,
        ↪  ICriterionMatcher<TLinkAddress> markedSequenceMatcher, TLinkAddress sequenceLink,
        ↪  TLinkAddress symbol)
            : base(links, sequenceLink, symbol)
            => _markedSequenceMatcher = markedSequenceMatcher;

        /// <summary>
        /// <para>
        /// Counts this instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public override TLinkAddress Count()
        {
            if (!_markedSequenceMatcher.IsMatched(_sequenceLink))
            {
                return default;
            }
            return base.Count();
        }
    }
}
```

1.16 ./csharp/Platform.Data.Doublets.Sequences/Frequencies/Counters/SequenceSymbolFrequencyOneOffCounte

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Interfaces;
using Platform.Numbers;
using Platform.Data.Sequences;

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

namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
{
    /// <summary>
    /// <para>
    /// Represents the sequence symbol frequency one off counter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="ICounter{TLinkAddress}"/>
    public class SequenceSymbolFrequencyOneOffCounter<TLinkAddress> : ICounter<TLinkAddress>
    {
        private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
        ↪  EqualityComparer<TLinkAddress>.Default;
        private static readonly Comparer<TLinkAddress> _comparer =
        ↪  Comparer<TLinkAddress>.Default;

        /// <summary>
        /// <para>
        /// The links.
        /// </para>
        /// <para></para>
        /// </summary>
        protected readonly ILinks<TLinkAddress> _links;
        /// <summary>
        /// <para>
        /// The sequence link.
        /// </para>
        /// <para></para>
```

```
        /// </summary>
        protected readonly TLinkAddress _sequenceLink;
        /// <summary>
        /// <para>
        /// The symbol.
        /// </para>
        /// <para></para>
        /// </summary>
        protected readonly TLinkAddress _symbol;
        /// <summary>
        /// <para>
        /// The total.
        /// </para>
        /// <para></para>
        /// </summary>
        protected TLinkAddress _total;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="SequenceSymbolFrequencyOneOffCounter"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="sequenceLink">
        /// <para>A sequence link.</para>
        /// <para></para>
        /// </param>
        /// <param name="symbol">
        /// <para>A symbol.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public SequenceSymbolFrequencyOneOffCounter(ILinks<TLinkAddress> links, TLinkAddress
        ↪  sequenceLink, TLinkAddress symbol)
        {
            _links = links;
            _sequenceLink = sequenceLink;
            _symbol = symbol;
            _total = default;
        }

        /// <summary>
        /// <para>
        /// Counts this instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <returns>
        /// <para>The total.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual TLinkAddress Count()
        {
            if (_comparer.Compare(_total, default) > 0)
            {
                return _total;
            }
            StopableSequenceWalker.WalkRight(_sequenceLink, _links.GetSource, _links.GetTarget,
            ↪  IsElement, VisitElement);
            return _total;
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private bool IsElement(TLinkAddress x) => _equalityComparer.Equals(x, _symbol) ||
        ↪  _links.IsPartialPoint(x); // TODO: Use SequenceElementCreteriaMatcher instead of
        ↪  IsPartialPoint
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private bool VisitElement(TLinkAddress element)
        {
            if (_equalityComparer.Equals(element, _symbol))
            {
                _total = Arithmetic.Increment(_total);
            }
            return true;
```

```
        }
    }
}
```

1.17 ./csharp/Platform.Data.Doublets.Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyC

```
using System.Runtime.CompilerServices;
using Platform.Interfaces;

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

namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
{
    /// <summary>
    /// <para>
    /// Represents the total marked sequence symbol frequency counter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="ICounter{TLinkAddress, TLinkAddress}"/>
    public class TotalMarkedSequenceSymbolFrequencyCounter<TLinkAddress> :
    ↪ ICounter<TLinkAddress, TLinkAddress>
    {
        private readonly ILinks<TLinkAddress> _links;
        private readonly ICriterionMatcher<TLinkAddress> _markedSequenceMatcher;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="TotalMarkedSequenceSymbolFrequencyCounter"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="markedSequenceMatcher">
        /// <para>A marked sequence matcher.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TotalMarkedSequenceSymbolFrequencyCounter(ILinks<TLinkAddress> links,
        ↪ ICriterionMatcher<TLinkAddress> markedSequenceMatcher)
        {
            _links = links;
            _markedSequenceMatcher = markedSequenceMatcher;
        }

        /// <summary>
        /// <para>
        /// Counts the argument.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="argument">
        /// <para>The argument.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLinkAddress Count(TLinkAddress argument) => new
        ↪ TotalMarkedSequenceSymbolFrequencyOneOffCounter<TLinkAddress>(_links,
        ↪ _markedSequenceMatcher, argument).Count();
    }
}
```

1.18 ./csharp/Platform.Data.Doublets.Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyO

```
using System.Runtime.CompilerServices;
using Platform.Interfaces;
using Platform.Numbers;

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

namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
{
    /// <summary>
    /// <para>
```

```
    /// Represents the total marked sequence symbol frequency one off counter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="TotalSequenceSymbolFrequencyOneOffCounter{TLinkAddress}"/>
    public class TotalMarkedSequenceSymbolFrequencyOneOffCounter<TLinkAddress> :
    ↪ TotalSequenceSymbolFrequencyOneOffCounter<TLinkAddress>
    {
        private readonly ICriterionMatcher<TLinkAddress> _markedSequenceMatcher;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="TotalMarkedSequenceSymbolFrequencyOneOffCounter"/>
        ↪ instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="markedSequenceMatcher">
        /// <para>A marked sequence matcher.</para>
        /// <para></para>
        /// </param>
        /// <param name="symbol">
        /// <para>A symbol.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TotalMarkedSequenceSymbolFrequencyOneOffCounter(ILinks<TLinkAddress> links,
        ↪ ICriterionMatcher<TLinkAddress> markedSequenceMatcher, TLinkAddress symbol)
            : base(links, symbol)
            => _markedSequenceMatcher = markedSequenceMatcher;

        /// <summary>
        /// <para>
        /// Counts the sequence symbol frequency using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override void CountSequenceSymbolFrequency(TLinkAddress link)
        {
            var symbolFrequencyCounter = new
            ↪ MarkedSequenceSymbolFrequencyOneOffCounter<TLinkAddress>(_links,
            ↪ _markedSequenceMatcher, link, _symbol);
            _total = Arithmetic.Add(_total, symbolFrequencyCounter.Count());
        }
    }
}
```

1.19 ./csharp/Platform.Data.Doublets.Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyCounter.c

```
using System.Runtime.CompilerServices;
using Platform.Interfaces;

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

namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
{
    /// <summary>
    /// <para>
    /// Represents the total sequence symbol frequency counter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="ICounter{TLinkAddress, TLinkAddress}"/>
    public class TotalSequenceSymbolFrequencyCounter<TLinkAddress> : ICounter<TLinkAddress,
    ↪ TLinkAddress>
    {
        private readonly ILinks<TLinkAddress> _links;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="TotalSequenceSymbolFrequencyCounter"/> instance.
```

```
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TotalSequenceSymbolFrequencyCounter(ILinks<TLinkAddress> links) => _links = links;

        /// <summary>
        /// <para>
        /// Counts the symbol.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="symbol">
        /// <para>The symbol.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLinkAddress Count(TLinkAddress symbol) => new
        ↪  TotalSequenceSymbolFrequencyOneOffCounter<TLinkAddress>(_links, symbol).Count();
    }
}
```

1.20 ./csharp/Platform.Data.Doublets.Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyOneOffC

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Interfaces;
using Platform.Numbers;

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

namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
{
    /// <summary>
    /// <para>
    /// Represents the total sequence symbol frequency one off counter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="ICounter{TLinkAddress}"/>
    public class TotalSequenceSymbolFrequencyOneOffCounter<TLinkAddress> : ICounter<TLinkAddress>
    {
        private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
        ↪  EqualityComparer<TLinkAddress>.Default;
        private static readonly Comparer<TLinkAddress> _comparer =
        ↪  Comparer<TLinkAddress>.Default;

        /// <summary>
        /// <para>
        /// The links.
        /// </para>
        /// <para></para>
        /// </summary>
        protected readonly ILinks<TLinkAddress> _links;
        /// <summary>
        /// <para>
        /// The symbol.
        /// </para>
        /// <para></para>
        /// </summary>
        protected readonly TLinkAddress _symbol;
        /// <summary>
        /// <para>
        /// The visits.
        /// </para>
        /// <para></para>
        /// </summary>
        protected readonly HashSet<TLinkAddress> _visits;
        /// <summary>
        /// <para>
        /// The total.
        /// </para>
```

```
47          /// <para></para>
48          /// </summary>
49          protected TLinkAddress _total;
50
51          /// <summary>
52          /// <para>
53          /// Initializes a new <see cref="TotalSequenceSymbolFrequencyOneOffCounter"/> instance.
54          /// </para>
55          /// <para></para>
56          /// </summary>
57          /// <param name="links">
58          /// <para>A links.</para>
59          /// <para></para>
60          /// </param>
61          /// <param name="symbol">
62          /// <para>A symbol.</para>
63          /// <para></para>
64          /// </param>
65          [MethodImpl(MethodImplOptions.AggressiveInlining)]
66          public TotalSequenceSymbolFrequencyOneOffCounter(ILinks<TLinkAddress> links,
            ↪ TLinkAddress symbol)
67          {
68              _links = links;
69              _symbol = symbol;
70              _visits = new HashSet<TLinkAddress>();
71              _total = default;
72          }
73
74          /// <summary>
75          /// <para>
76          /// Counts this instance.
77          /// </para>
78          /// <para></para>
79          /// </summary>
80          /// <returns>
81          /// <para>The total.</para>
82          /// <para></para>
83          /// </returns>
84          [MethodImpl(MethodImplOptions.AggressiveInlining)]
85          public TLinkAddress Count()
86          {
87              if (_comparer.Compare(_total, default) > 0 || _visits.Count > 0)
88              {
89                  return _total;
90              }
91              CountCore(_symbol);
92              return _total;
93          }
94          [MethodImpl(MethodImplOptions.AggressiveInlining)]
95          private void CountCore(TLinkAddress link)
96          {
97              var any = _links.Constants.Any;
98              if (_equalityComparer.Equals(_links.Count(any, link), default))
99              {
100                 CountSequenceSymbolFrequency(link);
101             }
102             else
103             {
104                 _links.Each(EachElementHandler, any, link);
105             }
106         }
107
108         /// <summary>
109         /// <para>
110         /// Counts the sequence symbol frequency using the specified link.
111         /// </para>
112         /// <para></para>
113         /// </summary>
114         /// <param name="link">
115         /// <para>The link.</para>
116         /// <para></para>
117         /// </param>
118         [MethodImpl(MethodImplOptions.AggressiveInlining)]
119         protected virtual void CountSequenceSymbolFrequency(TLinkAddress link)
120         {
121             var symbolFrequencyCounter = new
                ↪ SequenceSymbolFrequencyOneOffCounter<TLinkAddress>(_links, link, _symbol);
122             _total = Arithmetic.Add(_total, symbolFrequencyCounter.Count());
```

```
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private TLinkAddress EachElementHandler(IList<TLinkAddress>? doublet)
        {
            var constants = _links.Constants;
            var doubletIndex = doublet[constants.IndexPart];
            if (_visits.Add(doubletIndex))
            {
                CountCore(doubletIndex);
            }
            return constants.Continue;
        }
    }
}
```

1.21 ./csharp/Platform.Data.Doublets.Sequences/HeightProviders/CachedSequenceHeightProvider.cs

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Interfaces;
using Platform.Converters;

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

namespace Platform.Data.Doublets.Sequences.HeightProviders
{
    /// <summary>
    /// <para>
    /// Represents the cached sequence height provider.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="ISequenceHeightProvider{TLinkAddress}"/>
    public class CachedSequenceHeightProvider<TLinkAddress> :
    ↪  ISequenceHeightProvider<TLinkAddress>
    {
        private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
        ↪  EqualityComparer<TLinkAddress>.Default;
        private readonly TLinkAddress _heightPropertyMarker;
        private readonly ISequenceHeightProvider<TLinkAddress> _baseHeightProvider;
        private readonly IConverter<TLinkAddress> _addressToUnaryNumberConverter;
        private readonly IConverter<TLinkAddress> _unaryNumberToAddressConverter;
        private readonly IProperties<TLinkAddress, TLinkAddress, TLinkAddress> _propertyOperator;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="CachedSequenceHeightProvider"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="baseHeightProvider">
        /// <para>A base height provider.</para>
        /// <para></para>
        /// </param>
        /// <param name="addressToUnaryNumberConverter">
        /// <para>A address to unary number converter.</para>
        /// <para></para>
        /// </param>
        /// <param name="unaryNumberToAddressConverter">
        /// <para>A unary number to address converter.</para>
        /// <para></para>
        /// </param>
        /// <param name="heightPropertyMarker">
        /// <para>A height property marker.</para>
        /// <para></para>
        /// </param>
        /// <param name="propertyOperator">
        /// <para>A property operator.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public CachedSequenceHeightProvider(
            ISequenceHeightProvider<TLinkAddress> baseHeightProvider,
            IConverter<TLinkAddress> addressToUnaryNumberConverter,
            IConverter<TLinkAddress> unaryNumberToAddressConverter,
            TLinkAddress heightPropertyMarker,
            IProperties<TLinkAddress, TLinkAddress, TLinkAddress> propertyOperator)
        {
            _heightPropertyMarker = heightPropertyMarker;
            _baseHeightProvider = baseHeightProvider;
            _addressToUnaryNumberConverter = addressToUnaryNumberConverter;
```

```
            _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
            _propertyOperator = propertyOperator;
        }

        /// <summary>
        /// <para>
        /// Gets the sequence.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The height.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLinkAddress Get(TLinkAddress sequence)
        {
            TLinkAddress height;
            var heightValue = _propertyOperator.GetValue(sequence, _heightPropertyMarker);
            if (_equalityComparer.Equals(heightValue, default))
            {
                height = _baseHeightProvider.Get(sequence);
                heightValue = _addressToUnaryNumberConverter.Convert(height);
                _propertyOperator.SetValue(sequence, _heightPropertyMarker, heightValue);
            }
            else
            {
                height = _unaryNumberToAddressConverter.Convert(heightValue);
            }
            return height;
        }
    }
}
```

1.22 ./csharp/Platform.Data.Doublets.Sequences/HeightProviders/DefaultSequenceRightHeightProvider.cs

```
using System.Runtime.CompilerServices;
using Platform.Interfaces;
using Platform.Numbers;

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

namespace Platform.Data.Doublets.Sequences.HeightProviders
{
    /// <summary>
    /// <para>
    /// Represents the default sequence right height provider.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksOperatorBase{TLinkAddress}"/>
    /// <seealso cref="ISequenceHeightProvider{TLinkAddress}"/>
    public class DefaultSequenceRightHeightProvider<TLinkAddress> :
    ↪  LinksOperatorBase<TLinkAddress>, ISequenceHeightProvider<TLinkAddress>
    {
        private readonly ICriterionMatcher<TLinkAddress> _elementMatcher;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="DefaultSequenceRightHeightProvider"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="elementMatcher">
        /// <para>A element matcher.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public DefaultSequenceRightHeightProvider(ILinks<TLinkAddress> links,
        ↪  ICriterionMatcher<TLinkAddress> elementMatcher) : base(links) => _elementMatcher =
        ↪  elementMatcher;

```

```
        /// <summary>
        /// <para>
        /// Gets the sequence.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The height.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLinkAddress Get(TLinkAddress sequence)
        {
            var height = default(TLinkAddress);
            var pairOrElement = sequence;
            while (!_elementMatcher.IsMatched(pairOrElement))
            {
                pairOrElement = _links.GetTarget(pairOrElement);
                height = Arithmetic.Increment(height);
            }
            return height;
        }
    }
}
```

1.23 ./csharp/Platform.Data.Doublets.Sequences/HeightProviders/ISequenceHeightProvider.cs

```
using Platform.Interfaces;

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

namespace Platform.Data.Doublets.Sequences.HeightProviders
{
    /// <summary>
    /// <para>
    /// Defines the sequence height provider.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="IProvider{TLinkAddress, TLinkAddress}"/>
    public interface ISequenceHeightProvider<TLinkAddress> : IProvider<TLinkAddress,
    ↪  TLinkAddress>
    {
    }
}
```

1.24 ./csharp/Platform.Data.Doublets.Sequences/Incrementers/FrequencyIncrementer.cs

```
// using System.Collections.Generic;
// using System.Runtime.CompilerServices;
// using Platform.Incrementers;
//
// #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
//
// namespace Platform.Data.Doublets.Incrementers
// {
//     /// <summary>
//     /// <para>
//     /// Represents the frequency incrementer.
//     /// </para>
//     /// <para></para>
//     /// </summary>
//     /// <seealso cref="LinksOperatorBase{TLinkAddress}"/>
//     /// <seealso cref="IIncrementer{TLinkAddress}"/>
//     public class FrequencyIncrementer<TLinkAddress> : LinksOperatorBase<TLinkAddress>,
↪  IIncrementer<TLinkAddress>
//     {
//         private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
↪  EqualityComparer<TLinkAddress>.Default;
//         private readonly TLinkAddress _frequencyMarker;
//         private readonly TLinkAddress _unaryOne;
//         private readonly IIncrementer<TLinkAddress> _unaryNumberIncrementer;
//
//         /// <summary>
//         /// <para>
//         /// Initializes a new <see cref="FrequencyIncrementer"/> instance.
```

```
//         /// </para>
//         /// <para></para>
//         /// </summary>
//         /// <param name="links">
//         /// <para>A links.</para>
//         /// <para></para>
//         /// </param>
//         /// <param name="frequencyMarker">
//         /// <para>A frequency marker.</para>
//         /// <para></para>
//         /// </param>
//         /// <param name="unaryOne">
//         /// <para>A unary one.</para>
//         /// <para></para>
//         /// </param>
//         /// <param name="unaryNumberIncrementer">
//         /// <para>A unary number incrementer.</para>
//         /// <para></para>
//         /// </param>
//         [MethodImpl(MethodImplOptions.AggressiveInlining)]
//         public FrequencyIncrementer(ILinks<TLinkAddress> links, TLinkAddress frequencyMarker,
    ↪  TLinkAddress unaryOne, IIncrementer<TLinkAddress> unaryNumberIncrementer)
//             : base(links)
//         {
//             _frequencyMarker = frequencyMarker;
//             _unaryOne = unaryOne;
//             _unaryNumberIncrementer = unaryNumberIncrementer;
//         }
//
//         /// <summary>
//         /// <para>
//         /// Increments the frequency.
//         /// </para>
//         /// <para></para>
//         /// </summary>
//         /// <param name="frequency">
//         /// <para>The frequency.</para>
//         /// <para></para>
//         /// </param>
//         /// <returns>
//         /// <para>The link</para>
//         /// <para></para>
//         /// </returns>
//         [MethodImpl(MethodImplOptions.AggressiveInlining)]
//         public TLinkAddress Increment(TLinkAddress frequency)
//         {
//             var links = _links;
//             if (_equalityComparer.Equals(frequency, default))
//             {
//                 return links.GetOrCreate(_unaryOne, _frequencyMarker);
//             }
//             var incrementedSource =
    ↪  _unaryNumberIncrementer.Increment(links.GetSource(frequency));
//             return links.GetOrCreate(incrementedSource, _frequencyMarker);
//         }
//     }
// }
```

1.25 ./csharp/Platform.Data.Doublets.Sequences/Incrementers/UnaryNumberIncrementer.cs

```
// using System.Collections.Generic;
// using System.Runtime.CompilerServices;
// using Platform.Incrementers;
//
// #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
//
// namespace Platform.Data.Doublets.Incrementers
// {
//     /// <summary>
//     /// <para>
//     /// Represents the unary number incrementer.
//     /// </para>
//     /// <para></para>
//     /// </summary>
//     /// <seealso cref="LinksOperatorBase{TLinkAddress}"/>
//     /// <seealso cref="IIncrementer{TLinkAddress}"/>
//     public class UnaryNumberIncrementer<TLinkAddress> : LinksOperatorBase<TLinkAddress>,
    ↪  IIncrementer<TLinkAddress>
//     {
```

```
//         private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
    ↪  EqualityComparer<TLinkAddress>.Default;
//         private readonly TLinkAddress _unaryOne;
//
//         /// <summary>
//         /// <para>
//         /// Initializes a new <see cref="UnaryNumberIncrementer"/> instance.
//         /// </para>
//         /// <para></para>
//         /// </summary>
//         /// <param name="links">
//         /// <para>A links.</para>
//         /// <para></para>
//         /// </param>
//         /// <param name="unaryOne">
//         /// <para>A unary one.</para>
//         /// <para></para>
//         /// </param>
//         [MethodImpl(MethodImplOptions.AggressiveInlining)]
//         public UnaryNumberIncrementer(ILinks<TLinkAddress> links, TLinkAddress unaryOne) :
    ↪  base(links) => _unaryOne = unaryOne;
//
//         /// <summary>
//         /// <para>
//         /// Increments the unary number.
//         /// </para>
//         /// <para></para>
//         /// </summary>
//         /// <param name="unaryNumber">
//         /// <para>The unary number.</para>
//         /// <para></para>
//         /// </param>
//         /// <returns>
//         /// <para>The link</para>
//         /// <para></para>
//         /// </returns>
//         [MethodImpl(MethodImplOptions.AggressiveInlining)]
//         public TLinkAddress Increment(TLinkAddress unaryNumber)
//         {
//             var links = _links;
//             if (_equalityComparer.Equals(unaryNumber, _unaryOne))
//             {
//                 return links.GetOrCreate(_unaryOne, _unaryOne);
//             }
//             var source = links.GetSource(unaryNumber);
//             var target = links.GetTarget(unaryNumber);
//             if (_equalityComparer.Equals(source, target))
//             {
//                 return links.GetOrCreate(unaryNumber, _unaryOne);
//             }
//             else
//             {
//                 return links.GetOrCreate(source, Increment(target));
//             }
//         }
//     }
// }
```

1.26 ./csharp/Platform.Data.Doublets.Sequences/Indexes/CachedFrequencyIncrementingSequenceIndex.cs

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Data.Doublets.Sequences.Frequencies.Cache;

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

namespace Platform.Data.Doublets.Sequences.Indexes
{
    /// <summary>
    /// <para>
    /// Represents the cached frequency incrementing sequence index.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="ISequenceIndex{TLinkAddress}"/>
    public class CachedFrequencyIncrementingSequenceIndex<TLinkAddress> :
    ↪  ISequenceIndex<TLinkAddress>
    {
        private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
        ↪  EqualityComparer<TLinkAddress>.Default;
```

```
        private readonly LinkFrequenciesCache<TLinkAddress> _cache;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="CachedFrequencyIncrementingSequenceIndex"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="cache">
        /// <para>A cache.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public CachedFrequencyIncrementingSequenceIndex(LinkFrequenciesCache<TLinkAddress>
        ↪  cache) => _cache = cache;

        /// <summary>
        /// <para>
        /// Determines whether this instance add.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The indexed.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool Add(IList<TLinkAddress>? sequence)
        {
            var indexed = true;
            var i = sequence.Count;
            while (--i >= 1 && (indexed = IsIndexedWithIncrement(sequence[i - 1], sequence[i])))
            ↪  { }
            for (; i >= 1; i--)
            {
                _cache.IncrementFrequency(sequence[i - 1], sequence[i]);
            }
            return indexed;
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private bool IsIndexedWithIncrement(TLinkAddress source, TLinkAddress target)
        {
            var frequency = _cache.GetFrequency(source, target);
            if (frequency == null)
            {
                return false;
            }
            var indexed = !_equalityComparer.Equals(frequency.Frequency, default);
            if (indexed)
            {
                _cache.IncrementFrequency(source, target);
            }
            return indexed;
        }

        /// <summary>
        /// <para>
        /// Determines whether this instance might contain.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The indexed.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool MightContain(IList<TLinkAddress>? sequence)
        {
            var indexed = true;
            var i = sequence.Count;
```

```
            while (--i >= 1 && (indexed = IsIndexed(sequence[i - 1], sequence[i]))) { }
            return indexed;
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private bool IsIndexed(TLinkAddress source, TLinkAddress target)
        {
            var frequency = _cache.GetFrequency(source, target);
            if (frequency == null)
            {
                return false;
            }
            return !_equalityComparer.Equals(frequency.Frequency, default);
        }
    }
}
```

1.27 ./csharp/Platform.Data.Doublets.Sequences/Indexes/FrequencyIncrementingSequenceIndex.cs

```
// using System.Collections.Generic;
// using System.Runtime.CompilerServices;
// using Platform.Interfaces;
// using Platform.Incrementers;
//
// #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
//
// namespace Platform.Data.Doublets.Sequences.Indexes
// {
//     /// <summary>
//     /// <para>
//     /// Represents the frequency incrementing sequence index.
//     /// </para>
//     /// <para></para>
//     /// </summary>
//     /// <seealso cref="SequenceIndex{TLinkAddress}"/>
//     /// <seealso cref="ISequenceIndex{TLinkAddress}"/>
//     public class FrequencyIncrementingSequenceIndex<TLinkAddress> :
    ↪ SequenceIndex<TLinkAddress>, ISequenceIndex<TLinkAddress>
//     {
//         private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
    ↪ EqualityComparer<TLinkAddress>.Default;
//         private readonly IProperty<TLinkAddress, TLinkAddress> _frequencyPropertyOperator;
//         private readonly IIncrementer<TLinkAddress> _frequencyIncrementer;
//
//         /// <summary>
//         /// <para>
//         /// Initializes a new <see cref="FrequencyIncrementingSequenceIndex"/> instance.
//         /// </para>
//         /// <para></para>
//         /// </summary>
//         /// <param name="links">
//         /// <para>A links.</para>
//         /// <para></para>
//         /// </param>
//         /// <param name="frequencyPropertyOperator">
//         /// <para>A frequency property operator.</para>
//         /// <para></para>
//         /// </param>
//         /// <param name="frequencyIncrementer">
//         /// <para>A frequency incrementer.</para>
//         /// <para></para>
//         /// </param>
//         [MethodImpl(MethodImplOptions.AggressiveInlining)]
//         public FrequencyIncrementingSequenceIndex(ILinks<TLinkAddress> links,
    ↪ IProperty<TLinkAddress, TLinkAddress> frequencyPropertyOperator, IIncrementer<TLinkAddress>
    ↪ frequencyIncrementer)
//             : base(links)
//         {
//             _frequencyPropertyOperator = frequencyPropertyOperator;
//             _frequencyIncrementer = frequencyIncrementer;
//         }
//
//         /// <summary>
//         /// <para>
//         /// Determines whether this instance add.
//         /// </para>
//         /// <para></para>
//         /// </summary>
//         /// <param name="sequence">
//         /// <para>The sequence.</para>
```

```
//         /// <para></para>
//         /// </param>
//         /// <returns>
//         /// <para>The indexed.</para>
//         /// <para></para>
//         /// </returns>
//         [MethodImpl(MethodImplOptions.AggressiveInlining)]
//         public override bool Add(IList<TLinkAddress>? sequence)
//         {
//             var indexed = true;
//             var i = sequence.Count;
//             while (--i >= 1 && (indexed = IsIndexedWithIncrement(sequence[i - 1],
    ↪ sequence[i]))) { }
//             for (; i >= 1; i--)
//             {
//                 Increment(_links.GetOrCreate(sequence[i - 1], sequence[i]));
//             }
//             return indexed;
//         }
//         [MethodImpl(MethodImplOptions.AggressiveInlining)]
//         private bool IsIndexedWithIncrement(TLinkAddress source, TLinkAddress target)
//         {
//             var link = _links.SearchOrDefault(source, target);
//             var indexed = !_equalityComparer.Equals(link, default);
//             if (indexed)
//             {
//                 Increment(link);
//             }
//             return indexed;
//         }
//         [MethodImpl(MethodImplOptions.AggressiveInlining)]
//         private void Increment(TLinkAddress link)
//         {
//             var previousFrequency = _frequencyPropertyOperator.Get(link);
//             var frequency = _frequencyIncrementer.Increment(previousFrequency);
//             _frequencyPropertyOperator.Set(link, frequency);
//         }
//     }
// }
```

1.28 ./csharp/Platform.Data.Doublets.Sequences/Indexes/ISequenceIndex.cs

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

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

namespace Platform.Data.Doublets.Sequences.Indexes
{
    /// <summary>
    /// <para>
    /// Defines the sequence index.
    /// </para>
    /// <para></para>
    /// </summary>
    public interface ISequenceIndex<TLinkAddress>
    {
        /// <summary>
        /// Индексирует последовательность глобально, и возвращает значение,
        /// определяющие была ли запрошенная последовательность проиндексирована ранее.
        /// </summary>
        /// <param name="sequence">Последовательность для индексации.</param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        bool Add(IList<TLinkAddress>? sequence);

        /// <summary>
        /// <para>
        /// Determines whether this instance might contain.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
```

```
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        bool MightContain(IList<TLinkAddress>? sequence);
    }
}
```

1.29 ./csharp/Platform.Data.Doublets.Sequences/Indexes/SequenceIndex.cs

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

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

namespace Platform.Data.Doublets.Sequences.Indexes
{
    /// <summary>
    /// <para>
    /// Represents the sequence index.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksOperatorBase{TLinkAddress}"/>
    /// <seealso cref="ISequenceIndex{TLinkAddress}"/>
    public class SequenceIndex<TLinkAddress> : LinksOperatorBase<TLinkAddress>,
    ↪  ISequenceIndex<TLinkAddress>
    {
        private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
        ↪  EqualityComparer<TLinkAddress>.Default;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="SequenceIndex"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public SequenceIndex(ILinks<TLinkAddress> links) : base(links) { }

        /// <summary>
        /// <para>
        /// Determines whether this instance add.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The indexed.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual bool Add(IList<TLinkAddress>? sequence)
        {
            var indexed = true;
            var i = sequence.Count;
            while (--i >= 1 && (indexed =
            ↪  !_equalityComparer.Equals(_links.SearchOrDefault(sequence[i - 1], sequence[i]),
            ↪  default))) { }
            for (; i >= 1; i--)
            {
                _links.GetOrCreate(sequence[i - 1], sequence[i]);
            }
            return indexed;
        }

        /// <summary>
        /// <para>
        /// Determines whether this instance might contain.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
```

```
        /// <returns>
        /// <para>The indexed.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual bool MightContain(IList<TLinkAddress>? sequence)
        {
            var indexed = true;
            var i = sequence.Count;
            while (--i >= 1 && (indexed =
            ↪  !_equalityComparer.Equals(_links.SearchOrDefault(sequence[i - 1], sequence[i]),
            ↪  default))) { }
            return indexed;
        }
    }
}
```

1.30 ./csharp/Platform.Data.Doublets.Sequences/Indexes/SynchronizedSequenceIndex.cs

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

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

namespace Platform.Data.Doublets.Sequences.Indexes
{
    /// <summary>
    /// <para>
    /// Represents the synchronized sequence index.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="ISequenceIndex{TLinkAddress}"/>
    public class SynchronizedSequenceIndex<TLinkAddress> : ISequenceIndex<TLinkAddress>
    {
        private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
        ↪  EqualityComparer<TLinkAddress>.Default;
        private readonly ISynchronizedLinks<TLinkAddress> _links;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="SynchronizedSequenceIndex"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public SynchronizedSequenceIndex(ISynchronizedLinks<TLinkAddress> links) => _links =
        ↪  links;

        /// <summary>
        /// <para>
        /// Determines whether this instance add.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The indexed.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool Add(IList<TLinkAddress>? sequence)
        {
            var indexed = true;
            var i = sequence.Count;
            var links = _links.Unsync;
            _links.SyncRoot.DoRead(() =>
            {
                while (--i >= 1 && (indexed =
                ↪  !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1],
                ↪  sequence[i]), default))) { }
            });
            if (!indexed)
```

```
            {
                _links.SyncRoot.DoWrite(() =>
                {
                    for (; i >= 1; i--)
                    {
                        links.GetOrCreate(sequence[i - 1], sequence[i]);
                    }
                });
            }
            return indexed;
        }

        /// <summary>
        /// <para>
        /// Determines whether this instance might contain.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public bool MightContain(IList<TLinkAddress>? sequence)
        {
            var links = _links.Unsync;
            return _links.SyncRoot.DoRead(() =>
            {
                var indexed = true;
                var i = sequence.Count;
                while (--i >= 1 && (indexed =
                ↪  !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1],
                ↪  sequence[i]), default))) { }
                return indexed;
            });
        }
    }
}
```

1.31 ./csharp/Platform.Data.Doublets.Sequences/Indexes/Unindex.cs

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

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

namespace Platform.Data.Doublets.Sequences.Indexes
{
    /// <summary>
    /// <para>
    /// Represents the unindex.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="ISequenceIndex{TLinkAddress}"/>
    public class Unindex<TLinkAddress> : ISequenceIndex<TLinkAddress>
    {
        /// <summary>
        /// <para>
        /// Determines whether this instance add.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual bool Add(IList<TLinkAddress>? sequence) => false;

        /// <summary>
        /// <para>
```

```
        /// Determines whether this instance might contain.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public virtual bool MightContain(IList<TLinkAddress>? sequence) => true;
    }
}
```

1.32 ./csharp/Platform.Data.Doublets.Sequences/Numbers/Byte/ByteListToRawSequenceConverter.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Drawing;
using System.Linq;
using System.Numerics;
using System.Text;
using Platform.Collections.Stacks;
using Platform.Converters;
using Platform.Data.Doublets.CriterionMatchers;
using Platform.Data.Doublets.Decorators;
using Platform.Data.Doublets.Sequences.Converters;
using Platform.Data.Doublets.Unicode;
using Platform.Numbers;
using Platform.Reflection;
using Platform.Unsafe;

namespace Platform.Data.Doublets.Sequences.Numbers.Byte;

public class ByteListToRawSequenceConverter<TLinkAddress> : LinksDecoratorBase<TLinkAddress>,
    ↪ IConverter<IList<byte>, TLinkAddress> where TLinkAddress : struct
{
    public static readonly TLinkAddress MaximumValue = NumericType<TLinkAddress>.MaxValue;

    public static readonly TLinkAddress BitMask = Bit.ShiftRight(MaximumValue, 1);

    public static readonly int BitsSize = NumericType<TLinkAddress>.BitsSize;

    public readonly IConverter<TLinkAddress> AddressToNumberConverter;
    public readonly IConverter<TLinkAddress> NumberToAddressConverter;

    public readonly IConverter<IList<TLinkAddress>, TLinkAddress> ListToSequenceConverter;

    public readonly BalancedVariantConverter<TLinkAddress> BalancedVariantConverter;

    public readonly IConverter<string, TLinkAddress> StringToUnicodeSequenceConverter;

    public readonly IConverter<TLinkAddress, string> UnicodeSequenceToStringConverter;

    public readonly TLinkAddress Type = default;

    public readonly TLinkAddress ByteArrayLengthType;

    public readonly TLinkAddress ByteArrayType;

    public static readonly UncheckedConverter<Int32, TLinkAddress> IntToTLinkAddressConverter =
        ↪ UncheckedConverter<int, TLinkAddress>.Default;
    public static readonly UncheckedConverter<TLinkAddress, byte> TLinkAddressToByteConverter =
        ↪ UncheckedConverter<TLinkAddress, byte>.Default;
    public ArraySegment<byte> CurrentByteArray;
    public static readonly int BytesInRawNumberCount = BitsSize / 8;
    public readonly UncheckedConverter<byte, TLinkAddress> ByteToTLinkAddressConverter =
        ↪ UncheckedConverter<byte, TLinkAddress>.Default;
    public TLinkAddress EmptyByteArraySequenceType;


    public ByteListToRawSequenceConverter(ILinks<TLinkAddress> links, IConverter<TLinkAddress>
        ↪ addressToNumberConverter, IConverter<TLinkAddress> numberToAddressConverter,
        ↪ IConverter<IList<TLinkAddress>,TLinkAddress> listToSequenceConverter,
        ↪ StringToUnicodeSequenceConverter<TLinkAddress> stringToUnicodeSequenceConverter) :
        ↪ base(links)
    {
        AddressToNumberConverter = addressToNumberConverter;
        NumberToAddressConverter = numberToAddressConverter;
```

```
57          ListToSequenceConverter = listToSequenceConverter;
58          BalancedVariantConverter = new BalancedVariantConverter<TLinkAddress>(_links);
59          StringToUnicodeSequenceConverter = stringToUnicodeSequenceConverter;
60          TLinkAddress Zero = default;
61          Type = Arithmetic.Increment(Zero);
62          ByteArrayLengthType = _links.GetOrCreate(Type,
            ↪  StringToUnicodeSequenceConverter.Convert(nameof(ByteArrayLengthType)));
63          ByteArrayType = _links.GetOrCreate(Type,
            ↪  StringToUnicodeSequenceConverter.Convert(nameof(ByteArrayType)));
64          EmptyByteArraySequenceType = _links.GetOrCreate(Type,
            ↪  StringToUnicodeSequenceConverter.Convert(nameof(EmptyByteArraySequenceType)));
65      }
66
67      private int GetNonSavedBitsCount(int i)
68      {
69          if (i == 0)
70          {
71              return 0;
72          }
73          var nonSavedBitsCount = i % 8;
74          // if (nonSavedBitsCount == 0)
75          // {
76          //     return 1;
77          // }
78          return nonSavedBitsCount;
79      }
80
81      public TLinkAddress Convert(IList<byte> source)
82      {
83          return ListToSequenceConverter.Convert(source.Select(b =>
            ↪  AddressToNumberConverter.Convert(ByteToTLinkAddressConverter.Convert(b))).ToList());
84          // while (byteArrayToSave.Length != 0)
85          // {
86          //     var rawNumber = byteArrayToSave.ToStructure<TLinkAddress>();
87          //     rawNumbers.Add(rawNumber);
88          //     var bytesInRawNumberCount = rawNumbers.Count % 7 == 0 ? BytesInRawNumberCount - 1
            ↪  : BytesInRawNumberCount;
89          //     byteArrayToSave = byteArrayToSave.Skip(bytesInRawNumberCount).ToArray();
90          // }
91          // var processedRawNumbers = new List<TLinkAddress>(rawNumbers.Count);
92          // for (var j = 0; j < rawNumbers.Count; j++)
93          // {
94          //     var processedRawNumber = rawNumbers[j];
95          //     var nonSavedBitsCount = GetNonSavedBitsCount(j);
96          //     if (nonSavedBitsCount != 0)
97          //     {
98          //         processedRawNumber = Bit.ShiftLeft(processedRawNumber, nonSavedBitsCount);
99          //         var nonSavedBits = Bit.ShiftRight(rawNumbers[j - 1], BitsSize -
            ↪  nonSavedBitsCount);
100         //         processedRawNumber = Bit.Or(processedRawNumber, nonSavedBits);
101         //     }
102         //     processedRawNumber = AddressToNumberConverter.Convert(processedRawNumber);
103         //     processedRawNumbers.Add(processedRawNumber);
104         // }
105         // TLinkAddress lastRawNumber = default;
106         // var i = 0;
107         // while (byteArrayToSave.Length != 0)
108         // {
109         //     var rawNumber = byteArrayToSave.ToStructure<TLinkAddress>();
110         //     rawrawNumbers.Add(rawNumber);
111         //     Console.WriteLine("Raw number:");
112         //     Console.WriteLine(TestExtensions.PrettifyBinary<TLinkAddress>(System.Convert.ToSt
            ↪  ring((ushort)(object)rawNumber,
            ↪  2)));
113         //     var nonSavedBitsCount = GetNonSavedBitsCount(i);
114         //     var prevRawNumberWithNonSavedBitsAtStart = Bit.ShiftRight(lastRawNumber, BitsSize
            ↪  - nonSavedBitsCount);
115         //     var processedRawNumber = Bit.ShiftLeft(rawNumber, nonSavedBitsCount);
116         //     processedRawNumber = Bit.Or(processedRawNumber,
            ↪  prevRawNumberWithNonSavedBitsAtStart);
117         //     processedRawNumber = Bit.And(processedRawNumber, BitMask);
118         //     Console.WriteLine("Processed raw number:");
119         //     Console.WriteLine(TestExtensions.PrettifyBinary<TLinkAddress>(System.Convert.ToSt
            ↪  ring((ushort)(object)processedRawNumber,
            ↪  2)));
120         //     processedRawNumber = AddressToNumberConverter.Convert(processedRawNumber);
```

```
121         //     var bytesInRawNumberCount = nonSavedBitsCount == 7 ? BytesInRawNumberCount - 1 :
    ↪  BytesInRawNumberCount;
122         //     byteArrayToSave = byteArrayToSave.Skip(bytesInRawNumberCount).ToArray();
123         //     lastRawNumber = rawNumber;
124         //     i++;
125         //     rawNumbers.Add(processedRawNumber);
126         // }
127         // var notSavedBitsCount = GetNonSavedBitsCount(i);
128         // if ((source.Count != BitsSize) && (source.Count % BytesInRawNumberCount == 0))
129         // {
130         //     lastRawNumber = Bit.ShiftRight(lastRawNumber, BitsSize - notSavedBitsCount);
131         //     lastRawNumber = AddressToNumberConverter.Convert(lastRawNumber);
132         //     rawNumbers.Add(lastRawNumber);
133         // }
134         // Console.WriteLine("Raw numbers");
135         // rawrawNumbers.Reverse();
136         // StringBuilder rawrawNumbersSb = new StringBuilder();
137         // foreach (var rawrawNumber in rawrawNumbers)
138         // {
139         //     rawrawNumbersSb.Append(System.Convert.ToString((ushort)(object)rawrawNumber, 2));
140         // }
141         // Console.WriteLine(rawrawNumbersSb.ToString());
142         //
143         // rawNumbers.Reverse();
144         // Console.WriteLine("Processed raw numbers");
145         // StringBuilder rawNumbersSb = new StringBuilder();
146         // foreach (var rawNumber in rawNumbers)
147         // {
148         //     rawNumbersSb.Append(System.Convert.ToString((ushort)(object)NumberToAddressConver
    ↪  ter.Convert(rawNumber),
    ↪  2));
149         // }
150         // for (int c = 31; c < rawNumbersSb.Length; c += 32)
151         // {
152         //     rawNumbersSb.Remove(c, 1);
153         // }
154         // Console.WriteLine(rawNumbersSb.ToString());
155         // var length = IntToTLinkAddressConverter.Convert(source.Count);
156         // var byteArrayLengthAddress = _links.GetOrCreate(ByteArrayLengthType,
    ↪  AddressToNumberConverter.Convert(length));
157         // var byteArraySequenceAddress = ListToSequenceConverter.Convert(rawNumbers);
158         // return _links.GetOrCreate(ByteArrayType, _links.GetOrCreate(byteArrayLengthAddress,
    ↪  byteArraySequenceAddress));
159
160         // var processedRawNumbers = new List<TLinkAddress>();
161         // if (source.Count == 0)
162         // {
163         //     return Links.GetOrCreate(ByteArrayType, EmptyByteArraySequenceType);
164         // }
165         // var byteArrayToSave = source.ToArray();
166         // TLinkAddress lastRawNumber = default;
167         // var i = 0;
168         // while (byteArrayToSave.Length != 0)
169         // {
170         //     var rawNumber = byteArrayToSave.ToStructure<TLinkAddress>();
171         //     var nonSavedBitsCount = GetNonSavedBitsCount(i);
172         //     var prevRawNumberWithNonSavedBitsAtStart = Bit.ShiftRight(lastRawNumber, BitsSize
    ↪  - nonSavedBitsCount);
173         //     var processedRawNumber = Bit.ShiftLeft(rawNumber, nonSavedBitsCount);
174         //     processedRawNumber = Bit.Or(processedRawNumber,
    ↪  prevRawNumberWithNonSavedBitsAtStart);
175         //     processedRawNumber = Bit.And(processedRawNumber, BitMask);
176         //     processedRawNumber = AddressToNumberConverter.Convert(processedRawNumber);
177         //     var bytesInRawNumberCount = nonSavedBitsCount == 7 ? BytesInRawNumberCount - 1 :
    ↪  BytesInRawNumberCount;
178         //     byteArrayToSave = byteArrayToSave.Skip(bytesInRawNumberCount).ToArray();
179         //     lastRawNumber = rawNumber;
180         //     i++;
181         //     processedRawNumbers.Add(processedRawNumber);
182         // }
183         // var notSavedBitsCount = GetNonSavedBitsCount(i);
184         // if (source.Count % BytesInRawNumberCount == 0)
185         // {
186         //     lastRawNumber = Bit.ShiftRight(lastRawNumber, BitsSize - notSavedBitsCount);
187         //     lastRawNumber = AddressToNumberConverter.Convert(lastRawNumber);
188         //     processedRawNumbers.Add(lastRawNumber);
189         // }
```

```
        // var length = IntToTLinkAddressConverter.Convert(source.Count);
        // var byteArrayLengthAddress = _links.GetOrCreate(ByteArrayLengthType,
        ↪  AddressToNumberConverter.Convert(length));
        // var byteArraySequenceAddress = ListToSequenceConverter.Convert(processedRawNumbers);
        // return _links.GetOrCreate(ByteArrayType, _links.GetOrCreate(byteArrayLengthAddress,
        ↪  byteArraySequenceAddress));


        // List<TLinkAddress> rawNumberList = new(source.Count / BytesInRawNumberCount +
        ↪  source.Count);
        // var byteArray = source.ToArray();
        // var i = 0;
        // TLinkAddress rawNumberWithNonSavedBitsAtStart = default;
        // int lastNotSavedBitsCount = 0;
        // bool hasNotSavedBits = false;
        // while (byteArray.Length != 0)
        // {
        //     // if (i % 8 == 0)
        //     if (i == 0)
        //     {
        //         var rawNumber = byteArray.ToStructure<TLinkAddress>();
        //         hasNotSavedBits = byteArray.Length >= BytesInRawNumberCount;
        //         // var output = TestExtensions.PrettifyBinary<uint>(System.Convert.ToString((
        ↪  ushort)(object)rawNumber,
        ↪  2));
        //         // Console.WriteLine(output);
        //         rawNumberWithNonSavedBitsAtStart = Bit.ShiftRight(rawNumber, BitsSize - 1);
        //         lastNotSavedBitsCount = 1;
        //         rawNumber = Bit.And(rawNumber, BitMask);
        //         var output = TestExtensions.PrettifyBinary<uint>(System.Convert.ToString((ush
        ↪  ort)(object)rawNumber,
        ↪  2));
        //         Console.WriteLine(output);
        //         rawNumber = AddressToNumberConverter.Convert(rawNumber);
        //         // output = TestExtensions.PrettifyBinary<uint>(System.Convert.ToString((usho
        ↪  rt)(object)rawNumber,
        ↪  2));
        //         // Console.WriteLine(output);
        //         rawNumberList.Add(rawNumber);
        //         byteArray = byteArray.Skip(BytesInRawNumberCount).ToArray();
        //     }
        //     else
        //     {
        //         // var lastNotSavedBitsCount = i % 8;
        //         // if (lastNotSavedBitsCount == 0)
        //         // {
        //         //     lastNotSavedBitsCount = 1;
        //         // }
        //         // // if (lastNotSavedBitsCount % BitsSize == 0)
        //         // // {
        //         // //     lastNotSavedBitsCount = 0;
        //         // // }
        //         //
        //
        //         var newNotSavedBitsCount = lastNotSavedBitsCount + 1;
        //         var rawNumber = byteArray.ToStructure<TLinkAddress>();
        //         hasNotSavedBits = byteArray.Length >= BytesInRawNumberCount;
        //         // TODO: Check for lastNotSavedBitsCount == 0
        //         var newNotSavedBits = Bit.ShiftRight(rawNumber, BitsSize -
        ↪  newNotSavedBitsCount);
        //         // Shift left for non saved bits from previous raw number
        //         rawNumber = Bit.ShiftLeft(rawNumber, lastNotSavedBitsCount);
        //         // Put non saved bits at the start
        //         rawNumber = Bit.Or(rawNumber, rawNumberWithNonSavedBitsAtStart);
        //         // Mask last bit
        //         rawNumber = Bit.And(rawNumber, BitMask);
        //         rawNumberWithNonSavedBitsAtStart = newNotSavedBits;
        //         var output = TestExtensions.PrettifyBinary<uint>(System.Convert.ToString((ush
        ↪  ort)(object)rawNumber,
        ↪  2));
        //         Console.WriteLine(output);
        //         rawNumber = AddressToNumberConverter.Convert(rawNumber);
        //         rawNumberList.Add(rawNumber);
        //         var bytesInRawNumberCount = newNotSavedBitsCount % 7 != 0 ?
        ↪  BytesInRawNumberCount : BytesInRawNumberCount - 1;
        //         Console.WriteLine(bytesInRawNumberCount);
        //         Console.WriteLine(newNotSavedBitsCount);
```

```
254          //         Console.WriteLine(newNotSavedBitsCount % 7 != 0);
255          //         Console.WriteLine(newNotSavedBits);
256          //         byteArray = byteArray.Skip(bytesInRawNumberCount).ToArray();
257          //
258          //         if (lastNotSavedBitsCount % 8 == 0)
259          //         {
260          //             lastNotSavedBitsCount = 0;
261          //         }
262          //         lastNotSavedBitsCount++;
263          //     }
264          //     i++;
265          // }
266          // // if(rawNumberWithNonSavedBitsAtStart)
267          // // Console.WriteLine();
268          // Console.WriteLine(lastNotSavedBitsCount);
269          // if (hasNotSavedBits && lastNotSavedBitsCount % 7 != 0)
270          // {
271          //     var output = TestExtensions.PrettifyBinary<uint>(System.Convert.ToString((ushort)
             ↪  (object)rawNumberWithNonSavedBitsAtStart,
             ↪  2));
272          //     Console.WriteLine(output);
273          //     var lastRawNumber =
             ↪  AddressToNumberConverter.Convert(rawNumberWithNonSavedBitsAtStart);
274          //     rawNumberList.Add(lastRawNumber);
275          // }
276
277      }
278  }
```

1.33 ./csharp/Platform.Data.Doublets.Sequences/Numbers/Byte/RawSequenceToByteListConverter.cs

```
1   using System;
2   using System.Collections;
3   using System.Collections.Generic;
4   using System.Linq;
5   using System.Numerics;
6   using System.Text;
7   using Platform.Collections.Stacks;
8   using Platform.Converters;
9   using Platform.Data.Doublets.CriterionMatchers;
10  using Platform.Data.Doublets.Decorators;
11  using Platform.Data.Doublets.Sequences.Converters;
12  using Platform.Data.Doublets.Sequences.Walkers;
13  using Platform.Data.Doublets.Unicode;
14  using Platform.Data.Numbers.Raw;
15  using Platform.Numbers;
16  using Platform.Reflection;
17  using Platform.Unsafe;
18
19  namespace Platform.Data.Doublets.Sequences.Numbers.Byte;
20
21  public class RawSequenceToByteListConverter<TLinkAddress> : LinksDecoratorBase<TLinkAddress>,
    ↪  IConverter<TLinkAddress, IList<byte>> where TLinkAddress : struct
22  {
23      public static readonly EqualityComparer<TLinkAddress> EqualityComparer =
        ↪  EqualityComparer<TLinkAddress>.Default;
24
25      public static readonly TLinkAddress MaximumValue = NumericType<TLinkAddress>.MaxValue;
26
27      public static readonly TLinkAddress BitMask = Bit.ShiftRight(MaximumValue, 1);
28
29      public static readonly int BitsSize = NumericType<TLinkAddress>.BitsSize;
30
31
32      public readonly IConverter<TLinkAddress> NumberToAddressConverter;
33
34      public readonly IConverter<IList<TLinkAddress>, TLinkAddress> ListToSequenceConverter;
35
36      public readonly IConverter<string, TLinkAddress> StringToUnicodeSequenceConverteer;
37
38      public readonly IConverter<TLinkAddress, string> UnicodeSequenceToStringConverteer;
39
40      public readonly BalancedVariantConverter<TLinkAddress> BalancedVariantConverter;
41      public static readonly UncheckedConverter<TLinkAddress, byte> TLinkAddressToByteConverter =
        ↪  UncheckedConverter<TLinkAddress, byte>.Default;
42      public static readonly int BytesInRawNumberCount = BitsSize / 8;
43
44      public readonly TLinkAddress Type;
45
```

```
46      public RawSequenceToByteListConverter(ILinks<TLinkAddress> links, IConverter<TLinkAddress>
        ↪  numberToAddressConverter, IConverter<IList<TLinkAddress>, TLinkAddress>
        ↪  listToSequenceConverter, StringToUnicodeSequenceConverter<TLinkAddress>
        ↪  stringToUnicodeSequenceConverter, UnicodeSequenceToStringConverter<TLinkAddress>
        ↪  unicodeSequenceToStringConverteer) : base(links)
47      {
48          NumberToAddressConverter = numberToAddressConverter;
49          ListToSequenceConverter = listToSequenceConverter;
50          BalancedVariantConverter = new BalancedVariantConverter<TLinkAddress>(links);
51          StringToUnicodeSequenceConverteer = stringToUnicodeSequenceConverter;
52          UnicodeSequenceToStringConverteer = unicodeSequenceToStringConverteer;
53          TLinkAddress Zero = default;
54          Type = Arithmetic.Increment(Zero);
55      }
56
57      private bool IsEmptyArray(TLinkAddress array)
58      {
59          TLinkAddress zero = default;
60          var type = zero.Increment();
61          var emptyArrayTypeUnicodeSequence =
            ↪  StringToUnicodeSequenceConverteer.Convert("EmptyArrayType");
62          var emptyArrayType = _links.SearchOrDefault(type, emptyArrayTypeUnicodeSequence);
63          return EqualityComparer.Equals(emptyArrayType, array);
64      }
65
66      private void EnsureIsByteArrayLength(TLinkAddress byteArrayLengthAddress)
67      {
68          var source = _links.GetSource(byteArrayLengthAddress);
69          TLinkAddress zero = default;
70          var type = Arithmetic.Increment(zero);
71          var byteArrayLengthType = _links.SearchOrDefault(type,
            ↪  StringToUnicodeSequenceConverteer.Convert("ByteArrayLengthType"));
72          if (EqualityComparer.Equals(byteArrayLengthType, default))
73          {
74              throw new Exception("Could not find ByteArrayLengthType");
75          }
76          if (!EqualityComparer.Equals(source, byteArrayLengthType))
77          {
78              throw new Exception("Source must be ByteArrayLengthType");
79          }
80      }
81
82      private int GetByteArrayLength(TLinkAddress byteArrayLinkAddress)
83      {
84          EnsureIsByteArray(byteArrayLinkAddress);
85          var byteArrayValueLinkAddress = Links.GetTarget(byteArrayLinkAddress);
86          var byteArrayLengthLinkAddress = _links.GetSource(byteArrayValueLinkAddress);
87          EnsureIsByteArrayLength(byteArrayLengthLinkAddress);
88          var lengthValue = _links.GetTarget(byteArrayLengthLinkAddress);
89          CheckedConverter<TLinkAddress, int> checkedConverter = CheckedConverter<TLinkAddress,
            ↪  int>.Default;
90          return checkedConverter.Convert(NumberToAddressConverter.Convert(lengthValue));
91      }
92
93      private void EnsureIsByteArray(TLinkAddress possibleByteArray)
94      {
95          var byteArrayType = Links.SearchOrDefault(Type,
            ↪  StringToUnicodeSequenceConverteer.Convert("ByteArrayType"));
96          if(EqualityComparer.Equals(byteArrayType, default))
97          {
98              throw new Exception("ByteArrayType not found in the storage.");
99          }
100         var possibleByteArrayType = Links.GetSource(possibleByteArray);
101         if (!EqualityComparer.Equals(possibleByteArrayType, byteArrayType))
102         {
103             throw new ArgumentException($"{possibleByteArray} is not a byte array.");
104         }
105     }
106
107     private IEnumerator<TLinkAddress> GetRawNumbersEnumerator(TLinkAddress byteArrayLinkAddress)
108     {
109         EnsureIsByteArray(byteArrayLinkAddress);
110         var byteArrayValueLinkAddress = Links.GetTarget(byteArrayLinkAddress);
111         RightSequenceWalker<TLinkAddress> rightSequenceWalker = new(_links, new
            ↪  DefaultStack<TLinkAddress>());
112         var rawNumberSequenceAddress = _links.GetTarget(byteArrayValueLinkAddress);
113         var rawNumberSequence = rightSequenceWalker.Walk(rawNumberSequenceAddress);
114         return rawNumberSequence.GetEnumerator();
```

```
    }

    public IList<byte> Convert(TLinkAddress source)
    {
        return new RightSequenceWalker<TLinkAddress>(Links, new
        ↪  DefaultStack<TLinkAddress>()).Walk(source).Select(address =>
        ↪  NumberToAddressConverter.Convert(address)).Select(address =>
        ↪  TLinkAddressToByteConverter.Convert(address)).ToList();
        // Console.WriteLine("RawSequenceToByteListConverter.Convert");
        // if (IsEmptyArray(source))
        // {
        //     return new List<byte>();
        // }
        // EnsureIsByteArray(source);
        // var byteArrayLength = GetByteArrayLength(source);
        // List<byte> byteList = new(byteArrayLength);
        // var rawNumbersEnumerator = GetRawNumbersEnumerator(source);
        // var i = 0;
        // while (rawNumbersEnumerator.MoveNext())
        // {
        //     Console.WriteLine("Raw number: ");
        //     var rawNumber = NumberToAddressConverter.Convert(rawNumbersEnumerator.Current);
        //     Console.WriteLine(TestExtensions.PrettifyBinary<TLinkAddress>(System.Convert.ToSt
        ↪  ring((ushort)(object)rawNumber,
        ↪  2)));
        //     var nonSavedBitsCount = i % 8;
        //     var isLastRawNumber = (byteArrayLength % BytesInRawNumberCount == 0) &&
        ↪  (byteList.Count == byteArrayLength);
        //     if(isLastRawNumber)
        //     {
        //         rawNumber = Bit.ShiftLeft(rawNumber, 8 - nonSavedBitsCount);
        //         var @byte = TLinkAddressToByteConverter.Convert(rawNumber);
        //         byteList[byteList.Count - 1] = Bit.Or(byteList.Last(), @byte);
        //         break;
        //     }
        //     if (nonSavedBitsCount != 0)
        //     {
        //         var rawNumberWithOnlyNonSavedBits = rawNumber;
        //         rawNumberWithOnlyNonSavedBits = Bit.ShiftLeft(rawNumberWithOnlyNonSavedBits,
        ↪  BitsSize - nonSavedBitsCount);
        //         rawNumberWithOnlyNonSavedBits = Bit.ShiftRight(rawNumberWithOnlyNonSavedBits,
        ↪  BitsSize - nonSavedBitsCount);
        //         rawNumberWithOnlyNonSavedBits = Bit.ShiftLeft(rawNumberWithOnlyNonSavedBits,
        ↪  8 - nonSavedBitsCount);
        //         var @byte =
        ↪  TLinkAddressToByteConverter.Convert(rawNumberWithOnlyNonSavedBits);
        //         byteList[byteList.Count - 1] = Bit.Or(byteList.Last(), @byte);
        //         rawNumber = Bit.ShiftRight(rawNumber, nonSavedBitsCount);
        //     }
        //     var bytesInRawNumber = nonSavedBitsCount == 7 ? BytesInRawNumberCount -1 :
        ↪  BytesInRawNumberCount;
        //     for (int j = 0; (j < bytesInRawNumber) && (byteList.Count < byteArrayLength); j++)
        //     {
        //         var @byte = TLinkAddressToByteConverter.Convert(rawNumber);
        //         byteList.Add(@byte);
        //         rawNumber = Bit.ShiftRight(rawNumber, 8);
        //     }
        //     i++;
        // }
        // return byteList;
    }

    private static byte GetByteWithNotSavedBitsAtEnd(TLinkAddress currentRawNumber, int
    ↪  nonSavedBits)
    {
        var @byte = TLinkAddressToByteConverter.Convert(currentRawNumber);
        @byte = Bit.ShiftLeft(@byte, 8 - nonSavedBits);
        return @byte;
    }
}
```

1.34 ./csharp/Platform.Data.Doublets.Sequences/Numbers/Rational/DecimalToRationalConverter.cs

```
using System.Numerics;
using Platform.Converters;
using Platform.Data.Doublets.Decorators;
using System.Globalization;
using Platform.Data.Doublets.Numbers.Raw;

```

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

namespace Platform.Data.Doublets.Numbers.Rational
{
    /// <summary>
    /// <para>
    /// Represents the decimal to rational converter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksDecoratorBase{TLinkAddress}"/>
    /// <seealso cref="IConverter{decimal, TLinkAddress}"/>
    public class DecimalToRationalConverter<TLinkAddress> : LinksDecoratorBase<TLinkAddress>,
    ↪  IConverter<decimal, TLinkAddress>
        where TLinkAddress: struct
    {
        /// <summary>
        /// <para>
        /// The big integer to raw number sequence converter.
        /// </para>
        /// <para></para>
        /// </summary>
        public readonly BigIntegerToRawNumberSequenceConverter<TLinkAddress>
        ↪  BigIntegerToRawNumberSequenceConverter;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="DecimalToRationalConverter"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="bigIntegerToRawNumberSequenceConverter">
        /// <para>A big integer to raw number sequence converter.</para>
        /// <para></para>
        /// </param>
        public DecimalToRationalConverter(ILinks<TLinkAddress> links,
        ↪  BigIntegerToRawNumberSequenceConverter<TLinkAddress>
        ↪  bigIntegerToRawNumberSequenceConverter) : base(links)
        {
            BigIntegerToRawNumberSequenceConverter = bigIntegerToRawNumberSequenceConverter;
        }

        /// <summary>
        /// <para>
        /// Converts the decimal.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="@decimal">
        /// <para>The decimal.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        public TLinkAddress Convert(decimal @decimal)
        {
            var decimalAsString = @decimal.ToString(CultureInfo.InvariantCulture);
            var dotPosition = decimalAsString.IndexOf('.');
            var decimalWithoutDots = decimalAsString;
            int digitsAfterDot = 0;
            if (dotPosition != -1)
            {
                decimalWithoutDots = decimalWithoutDots.Remove(dotPosition, 1);
                digitsAfterDot = decimalAsString.Length - 1 - dotPosition;
            }
            BigInteger denominator = new(System.Math.Pow(10, digitsAfterDot));
            BigInteger numerator = BigInteger.Parse(decimalWithoutDots);
            BigInteger greatestCommonDivisor;
            do
            {
                greatestCommonDivisor = BigInteger.GreatestCommonDivisor(numerator, denominator);
                numerator /= greatestCommonDivisor;
                denominator /= greatestCommonDivisor;
```

```
            }
            while (greatestCommonDivisor > 1);
            var numeratorLink = BigIntegerToRawNumberSequenceConverter.Convert(numerator);
            var denominatorLink = BigIntegerToRawNumberSequenceConverter.Convert(denominator);
            return _links.GetOrCreate(numeratorLink, denominatorLink);
        }
    }
}
```

1.35 ./csharp/Platform.Data.Doublets.Sequences/Numbers/Rational/RationalToDecimalConverter.cs

```
using Platform.Converters;
using Platform.Data.Doublets.Decorators;
using Platform.Data.Doublets.Numbers.Raw;

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

namespace Platform.Data.Doublets.Numbers.Rational
{
    /// <summary>
    /// <para>
    /// Represents the rational to decimal converter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksDecoratorBase{TLinkAddress}"/>
    /// <seealso cref="IConverter{TLinkAddress, decimal}"/>
    public class RationalToDecimalConverter<TLinkAddress> : LinksDecoratorBase<TLinkAddress>,
    ↪  IConverter<TLinkAddress, decimal>
        where TLinkAddress: struct
    {
        /// <summary>
        /// <para>
        /// The raw number sequence to big integer converter.
        /// </para>
        /// <para></para>
        /// </summary>
        public readonly RawNumberSequenceToBigIntegerConverter<TLinkAddress>
        ↪  RawNumberSequenceToBigIntegerConverter;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="RationalToDecimalConverter"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="rawNumberSequenceToBigIntegerConverter">
        /// <para>A raw number sequence to big integer converter.</para>
        /// <para></para>
        /// </param>
        public RationalToDecimalConverter(ILinks<TLinkAddress> links,
        ↪  RawNumberSequenceToBigIntegerConverter<TLinkAddress>
        ↪  rawNumberSequenceToBigIntegerConverter) : base(links)
        {
            RawNumberSequenceToBigIntegerConverter = rawNumberSequenceToBigIntegerConverter;
        }

        /// <summary>
        /// <para>
        /// Converts the rational number.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="rationalNumber">
        /// <para>The rational number.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The decimal</para>
        /// <para></para>
        /// </returns>
        public decimal Convert(TLinkAddress rationalNumber)
        {
            var numerator = (decimal)RawNumberSequenceToBigIntegerConverter.Convert(_links.GetSo
            ↪  urce(rationalNumber));
```

```
            var denominator = (decimal)RawNumberSequenceToBigIntegerConverter.Convert(_links.Get
            ↪  Target(rationalNumber));
            return numerator / denominator;
        }
    }
}
```

1.36 ./csharp/Platform.Data.Doublets.Sequences/Numbers/Raw/BigIntegerToRawNumberSequenceConverter.cs

```
using System.Collections.Generic;
using System.Numerics;
using System.Runtime.InteropServices;
using Platform.Converters;
using Platform.Data.Doublets.Decorators;
using Platform.Numbers;
using Platform.Reflection;
using Platform.Unsafe;

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

namespace Platform.Data.Doublets.Numbers.Raw
{
    /// <summary>
    /// <para>
    /// Represents the big integer to raw number sequence converter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksDecoratorBase{TLinkAddress}"/>
    /// <seealso cref="IConverter{BigInteger, TLinkAddress}"/>
    public class BigIntegerToRawNumberSequenceConverter<TLinkAddress> :
    ↪  LinksDecoratorBase<TLinkAddress>, IConverter<BigInteger, TLinkAddress>
        where TLinkAddress : struct
    {
        /// <summary>
        /// <para>
        /// The max value.
        /// </para>
        /// <para></para>
        /// </summary>
        public static readonly TLinkAddress MaximumValue = NumericType<TLinkAddress>.MaxValue;
        /// <summary>
        /// <para>
        /// The maximum value.
        /// </para>
        /// <para></para>
        /// </summary>
        public static readonly TLinkAddress BitMask = Bit.ShiftRight(MaximumValue, 1);
        /// <summary>
        /// <para>
        /// The address to number converter.
        /// </para>
        /// <para></para>
        /// </summary>
        public readonly IConverter<TLinkAddress> AddressToNumberConverter;
        /// <summary>
        /// <para>
        /// The list to sequence converter.
        /// </para>
        /// <para></para>
        /// </summary>
        public readonly IConverter<IList<TLinkAddress>, TLinkAddress> ListToSequenceConverter;
        /// <summary>
        /// <para>
        /// The negative number marker.
        /// </para>
        /// <para></para>
        /// </summary>
        public readonly TLinkAddress NegativeNumberMarker;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="BigIntegerToRawNumberSequenceConverter"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
```

```
71          /// <param name="addressToNumberConverter">
72          /// <para>A address to number converter.</para>
73          /// <para></para>
74          /// </param>
75          /// <param name="listToSequenceConverter">
76          /// <para>A list to sequence converter.</para>
77          /// <para></para>
78          /// </param>
79          /// <param name="negativeNumberMarker">
80          /// <para>A negative number marker.</para>
81          /// <para></para>
82          /// </param>
83          public BigIntegerToRawNumberSequenceConverter(ILinks<TLinkAddress> links,
            ↪  IConverter<TLinkAddress> addressToNumberConverter,
            ↪  IConverter<IList<TLinkAddress>,TLinkAddress> listToSequenceConverter, TLinkAddress
            ↪  negativeNumberMarker) : base(links)
84          {
85              AddressToNumberConverter = addressToNumberConverter;
86              ListToSequenceConverter = listToSequenceConverter;
87              NegativeNumberMarker = negativeNumberMarker;
88          }
89          private List<TLinkAddress> GetRawNumberParts(BigInteger bigInteger)
90          {
91              List<TLinkAddress> rawNumbers = new();
92              BigInteger currentBigInt = bigInteger;
93              do
94              {
95                  var bigIntBytes = currentBigInt.ToByteArray();
96                  var bigIntWithBitMask = Bit.And(bigIntBytes.ToStructure<TLinkAddress>(),
                    ↪  BitMask);
97                  var rawNumber = AddressToNumberConverter.Convert(bigIntWithBitMask);
98                  rawNumbers.Add(rawNumber);
99                  currentBigInt >>= (NumericType<TLinkAddress>.BitsSize - 1);
100             }
101             while (currentBigInt > 0);
102             return rawNumbers;
103         }
104
105         /// <summary>
106         /// <para>
107         /// Converts the big integer.
108         /// </para>
109         /// <para></para>
110         /// </summary>
111         /// <param name="bigInteger">
112         /// <para>The big integer.</para>
113         /// <para></para>
114         /// </param>
115         /// <returns>
116         /// <para>The link</para>
117         /// <para></para>
118         /// </returns>
119         public TLinkAddress Convert(BigInteger bigInteger)
120         {
121             var sign = bigInteger.Sign;
122             var number = GetRawNumberParts(sign == -1 ? BigInteger.Negate(bigInteger) :
                ↪  bigInteger);
123             var numberSequence = ListToSequenceConverter.Convert(number);
124             return sign == -1 ? _links.GetOrCreate(NegativeNumberMarker, numberSequence) :
                ↪  numberSequence;
125         }
126     }
127 }
```

1.37 ./csharp/Platform.Data.Doublets.Sequences/Numbers/Raw/LongRawNumberSequenceToNumberConverter.c

```
1  using System.Runtime.CompilerServices;
2  using Platform.Collections.Stacks;
3  using Platform.Converters;
4  using Platform.Numbers;
5  using Platform.Reflection;
6  using Platform.Data.Doublets.Decorators;
7  using Platform.Data.Doublets.Sequences.Walkers;
8
9  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11 namespace Platform.Data.Doublets.Numbers.Raw
12 {
13     /// <summary>
14     /// <para>
```

```
15      /// Represents the long raw number sequence to number converter.
16      /// </para>
17      /// <para></para>
18      /// </summary>
19      /// <seealso cref="LinksDecoratorBase{TSource}"/>
20      /// <seealso cref="IConverter{TSource, TTarget}"/>
21      public class LongRawNumberSequenceToNumberConverter<TSource, TTarget> :
        ↪  LinksDecoratorBase<TSource>, IConverter<TSource, TTarget>
22      {
23          private static readonly int _bitsPerRawNumber = NumericType<TSource>.BitsSize - 1;
24          private static readonly UncheckedConverter<TSource, TTarget> _sourceToTargetConverter =
            ↪  UncheckedConverter<TSource, TTarget>.Default;
25          private readonly IConverter<TSource> _numberToAddressConverter;
26
27          /// <summary>
28          /// <para>
29          /// Initializes a new <see cref="LongRawNumberSequenceToNumberConverter"/> instance.
30          /// </para>
31          /// <para></para>
32          /// </summary>
33          /// <param name="links">
34          /// <para>A links.</para>
35          /// <para></para>
36          /// </param>
37          /// <param name="numberToAddressConverter">
38          /// <para>A number to address converter.</para>
39          /// <para></para>
40          /// </param>
41          [MethodImpl(MethodImplOptions.AggressiveInlining)]
42          public LongRawNumberSequenceToNumberConverter(ILinks<TSource> links, IConverter<TSource>
            ↪  numberToAddressConverter) : base(links) => _numberToAddressConverter =
            ↪  numberToAddressConverter;
43
44          /// <summary>
45          /// <para>
46          /// Converts the source.
47          /// </para>
48          /// <para></para>
49          /// </summary>
50          /// <param name="source">
51          /// <para>The source.</para>
52          /// <para></para>
53          /// </param>
54          /// <returns>
55          /// <para>The target</para>
56          /// <para></para>
57          /// </returns>
58          [MethodImpl(MethodImplOptions.AggressiveInlining)]
59          public TTarget Convert(TSource source)
60          {
61              var constants = Links.Constants;
62              var externalReferencesRange = constants.ExternalReferencesRange;
63              if (externalReferencesRange.HasValue &&
                ↪  externalReferencesRange.Value.Contains(source))
64              {
65                  return
                    ↪  _sourceToTargetConverter.Convert(_numberToAddressConverter.Convert(source));
66              }
67              else
68              {
69                  var pair = Links.GetLink(source);
70                  var walker = new LeftSequenceWalker<TSource>(Links, new DefaultStack<TSource>(),
                    ↪  (link) => externalReferencesRange.HasValue &&
                    ↪  externalReferencesRange.Value.Contains(link));
71                  TTarget result = default;
72                  foreach (var element in walker.Walk(source))
73                  {
74                      result = Bit.Or(Bit.ShiftLeft(result, _bitsPerRawNumber), Convert(element));
75                  }
76                  return result;
77              }
78          }
79      }
80  }
```

1.38 ./csharp/Platform.Data.Doublets.Sequences/Numbers/Raw/NumberToLongRawNumberSequenceConverter.c

```
1   using System.Collections.Generic;
2   using System.Runtime.CompilerServices;
3   using Platform.Converters;
```

```
using Platform.Numbers;
using Platform.Reflection;
using Platform.Data.Doublets.Decorators;

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

namespace Platform.Data.Doublets.Numbers.Raw
{
    /// <summary>
    /// <para>
    /// Represents the number to long raw number sequence converter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksDecoratorBase{TTarget}"/>
    /// <seealso cref="IConverter{TSource, TTarget}"/>
    public class NumberToLongRawNumberSequenceConverter<TSource, TTarget> :
    ↪  LinksDecoratorBase<TTarget>, IConverter<TSource, TTarget>
    {
        private static readonly Comparer<TSource> _comparer = Comparer<TSource>.Default;
        private static readonly TSource _maximumValue = NumericType<TSource>.MaxValue;
        private static readonly int _bitsPerRawNumber = NumericType<TTarget>.BitsSize - 1;
        private static readonly TSource _bitMask = Bit.ShiftRight(_maximumValue,
        ↪  NumericType<TTarget>.BitsSize + 1);
        private static readonly TSource _maximumConvertableAddress = CheckedConverter<TTarget,
        ↪  TSource>.Default.Convert(Arithmetic.Decrement(Hybrid<TTarget>.ExternalZero));
        private static readonly UncheckedConverter<TSource, TTarget> _sourceToTargetConverter =
        ↪  UncheckedConverter<TSource, TTarget>.Default;
        private readonly IConverter<TTarget> _addressToNumberConverter;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="NumberToLongRawNumberSequenceConverter"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="addressToNumberConverter">
        /// <para>A address to number converter.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public NumberToLongRawNumberSequenceConverter(ILinks<TTarget> links, IConverter<TTarget>
        ↪  addressToNumberConverter) : base(links) => _addressToNumberConverter =
        ↪  addressToNumberConverter;

        /// <summary>
        /// <para>
        /// Converts the source.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="source">
        /// <para>The source.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The target</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TTarget Convert(TSource source)
        {
            if (_comparer.Compare(source, _maximumConvertableAddress) > 0)
            {
                var numberPart = Bit.And(source, _bitMask);
                var convertedNumber = _addressToNumberConverter.Convert(_sourceToTargetConverter
                ↪  .Convert(numberPart));
                return Links.GetOrCreate(convertedNumber, Convert(Bit.ShiftRight(source,
                ↪  _bitsPerRawNumber)));
            }
            else
            {
                return
                ↪  _addressToNumberConverter.Convert(_sourceToTargetConverter.Convert(source));
```

```
            }
        }
    }
}
```

1.39 ./csharp/Platform.Data.Doublets.Sequences/Numbers/Raw/RawNumberSequenceToBigIntergerConverter.cs

```
using System;
using System.Collections.Generic;
using System.Numerics;
using Platform.Collections.Stacks;
using Platform.Converters;
using Platform.Data.Doublets.Decorators;
using Platform.Data.Doublets.Sequences.Walkers;
using Platform.Reflection;
using Platform.Unsafe;

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

namespace Platform.Data.Doublets.Numbers.Raw
{
    /// <summary>
    /// <para>
    /// Represents the raw number sequence to big integer converter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksDecoratorBase{TLinkAddress}"/>
    /// <seealso cref="IConverter{TLinkAddress, BigInteger}"/>
    public class RawNumberSequenceToBigIntegerConverter<TLinkAddress> :
    ↪  LinksDecoratorBase<TLinkAddress>, IConverter<TLinkAddress, BigInteger>
        where TLinkAddress : struct
    {
        /// <summary>
        /// <para>
        /// The default.
        /// </para>
        /// <para></para>
        /// </summary>
        public readonly EqualityComparer<TLinkAddress> EqualityComparer =
        ↪  EqualityComparer<TLinkAddress>.Default;
        /// <summary>
        /// <para>
        /// The number to address converter.
        /// </para>
        /// <para></para>
        /// </summary>
        public readonly IConverter<TLinkAddress, TLinkAddress> NumberToAddressConverter;
        /// <summary>
        /// <para>
        /// The left sequence walker.
        /// </para>
        /// <para></para>
        /// </summary>
        public readonly LeftSequenceWalker<TLinkAddress> LeftSequenceWalker;
        /// <summary>
        /// <para>
        /// The negative number marker.
        /// </para>
        /// <para></para>
        /// </summary>
        public readonly TLinkAddress NegativeNumberMarker;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="RawNumberSequenceToBigIntegerConverter"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="numberToAddressConverter">
        /// <para>A number to address converter.</para>
        /// <para></para>
        /// </param>
        /// <param name="negativeNumberMarker">
        /// <para>A negative number marker.</para>
        /// <para></para>
```

```
72          /// </param>
73          public RawNumberSequenceToBigIntegerConverter(ILinks<TLinkAddress> links,
            ↪  IConverter<TLinkAddress, TLinkAddress> numberToAddressConverter, TLinkAddress
            ↪  negativeNumberMarker) : base(links)
74          {
75              NumberToAddressConverter = numberToAddressConverter;
76              LeftSequenceWalker = new(links, new DefaultStack<TLinkAddress>());
77              NegativeNumberMarker = negativeNumberMarker;
78          }
79
80          /// <summary>
81          /// <para>
82          /// Converts the big integer.
83          /// </para>
84          /// <para></para>
85          /// </summary>
86          /// <param name="bigInteger">
87          /// <para>The big integer.</para>
88          /// <para></para>
89          /// </param>
90          /// <exception cref="Exception">
91          /// <para>Raw number sequence cannot be empty.</para>
92          /// <para></para>
93          /// </exception>
94          /// <returns>
95          /// <para>The big integer</para>
96          /// <para></para>
97          /// </returns>
98          public BigInteger Convert(TLinkAddress bigInteger)
99          {
100             var sign = 1;
101             var bigIntegerSequence = bigInteger;
102             if (EqualityComparer.Equals(_links.GetSource(bigIntegerSequence),
                ↪  NegativeNumberMarker))
103             {
104                 sign = -1;
105                 bigIntegerSequence = _links.GetTarget(bigInteger);
106             }
107             using var enumerator = LeftSequenceWalker.Walk(bigIntegerSequence).GetEnumerator();
108             if (!enumerator.MoveNext())
109             {
110                 throw new Exception("Raw number sequence cannot be empty.");
111             }
112             var nextPart = NumberToAddressConverter.Convert(enumerator.Current);
113             BigInteger currentBigInt = new(nextPart.ToBytes());
114             while (enumerator.MoveNext())
115             {
116                 currentBigInt <<= (NumericType<TLinkAddress>.BitsSize - 1);
117                 nextPart = NumberToAddressConverter.Convert(enumerator.Current);
118                 currentBigInt |= new BigInteger(nextPart.ToBytes());
119             }
120             return sign == -1 ? BigInteger.Negate(currentBigInt) : currentBigInt;
121         }
122     }
123 }
```

1.40 ./csharp/Platform.Data.Doublets.Sequences/Numbers/Unary/AddressToUnaryNumberConverter.cs

```
1  using System.Collections.Generic;
2  using Platform.Reflection;
3  using Platform.Converters;
4  using Platform.Numbers;
5  using System.Runtime.CompilerServices;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Data.Doublets.Numbers.Unary
10 {
11     /// <summary>
12     /// <para>
13     /// Represents the address to unary number converter.
14     /// </para>
15     /// <para></para>
16     /// </summary>
17     /// <seealso cref="LinksOperatorBase{TLinkAddress}"/>
18     /// <seealso cref="IConverter{TLinkAddress}"/>
19     public class AddressToUnaryNumberConverter<TLinkAddress> : LinksOperatorBase<TLinkAddress>,
       ↪  IConverter<TLinkAddress>
20     {
```

```
        private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
        ↪  EqualityComparer<TLinkAddress>.Default;
        private static readonly TLinkAddress _zero = default;
        private static readonly TLinkAddress _one = Arithmetic.Increment(_zero);
        private readonly IConverter<int, TLinkAddress> _powerOf2ToUnaryNumberConverter;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="AddressToUnaryNumberConverter"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="powerOf2ToUnaryNumberConverter">
        /// <para>A power of to unary number converter.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public AddressToUnaryNumberConverter(ILinks<TLinkAddress> links, IConverter<int,
        ↪  TLinkAddress> powerOf2ToUnaryNumberConverter) : base(links) =>
        ↪  _powerOf2ToUnaryNumberConverter = powerOf2ToUnaryNumberConverter;

        /// <summary>
        /// <para>
        /// Converts the number.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="number">
        /// <para>The number.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The target.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLinkAddress Convert(TLinkAddress number)
        {
            var links = _links;
            var nullConstant = links.Constants.Null;
            var target = nullConstant;
            for (var i = 0; !_equalityComparer.Equals(number, _zero) && i <
            ↪  NumericType<TLinkAddress>.BitsSize; i++)
            {
                if (_equalityComparer.Equals(Bit.And(number, _one), _one))
                {
                    target = _equalityComparer.Equals(target, nullConstant)
                        ? _powerOf2ToUnaryNumberConverter.Convert(i)
                        : links.GetOrCreate(_powerOf2ToUnaryNumberConverter.Convert(i), target);
                }
                number = Bit.ShiftRight(number, 1);
            }
            return target;
        }
    }
}
```

1.41 ./csharp/Platform.Data.Doublets.Sequences/Numbers/Unary/LinkToItsFrequencyNumberConveter.cs

```
using System;
using System.Collections.Generic;
using Platform.Interfaces;
using Platform.Converters;
using System.Runtime.CompilerServices;

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

namespace Platform.Data.Doublets.Numbers.Unary
{
    /// <summary>
    /// <para>
    /// Represents the link to its frequency number conveter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksOperatorBase{TLinkAddress}"/>
```

```
18      /// <seealso cref="IConverter{Doublet{TLinkAddress}, TLinkAddress}"/>
19      public class LinkToItsFrequencyNumberConveter<TLinkAddress> :
        ↪  LinksOperatorBase<TLinkAddress>, IConverter<Doublet<TLinkAddress>, TLinkAddress>
20      {
21          private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
            ↪  EqualityComparer<TLinkAddress>.Default;
22          private readonly IProperty<TLinkAddress, TLinkAddress> _frequencyPropertyOperator;
23          private readonly IConverter<TLinkAddress> _unaryNumberToAddressConverter;
24
25          /// <summary>
26          /// <para>
27          /// Initializes a new <see cref="LinkToItsFrequencyNumberConveter"/> instance.
28          /// </para>
29          /// <para></para>
30          /// </summary>
31          /// <param name="links">
32          /// <para>A links.</para>
33          /// <para></para>
34          /// </param>
35          /// <param name="frequencyPropertyOperator">
36          /// <para>A frequency property operator.</para>
37          /// <para></para>
38          /// </param>
39          /// <param name="unaryNumberToAddressConverter">
40          /// <para>A unary number to address converter.</para>
41          /// <para></para>
42          /// </param>
43          [MethodImpl(MethodImplOptions.AggressiveInlining)]
44          public LinkToItsFrequencyNumberConveter(
45              ILinks<TLinkAddress> links,
46              IProperty<TLinkAddress, TLinkAddress> frequencyPropertyOperator,
47              IConverter<TLinkAddress> unaryNumberToAddressConverter)
48              : base(links)
49          {
50              _frequencyPropertyOperator = frequencyPropertyOperator;
51              _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
52          }
53
54          /// <summary>
55          /// <para>
56          /// Converts the doublet.
57          /// </para>
58          /// <para></para>
59          /// </summary>
60          /// <param name="doublet">
61          /// <para>The doublet.</para>
62          /// <para></para>
63          /// </param>
64          /// <exception cref="ArgumentException">
65          /// <para>Link ({doublet}) not found. </para>
66          /// <para></para>
67          /// </exception>
68          /// <returns>
69          /// <para>The link</para>
70          /// <para></para>
71          /// </returns>
72          [MethodImpl(MethodImplOptions.AggressiveInlining)]
73          public TLinkAddress Convert(Doublet<TLinkAddress> doublet)
74          {
75              var links = _links;
76              var link = links.SearchOrDefault(doublet.Source, doublet.Target);
77              if (_equalityComparer.Equals(link, default))
78              {
79                  throw new ArgumentException($"Link ({doublet}) not found.", nameof(doublet));
80              }
81              var frequency = _frequencyPropertyOperator.Get(link);
82              if (_equalityComparer.Equals(frequency, default))
83              {
84                  return default;
85              }
86              var frequencyNumber = links.GetSource(frequency);
87              return _unaryNumberToAddressConverter.Convert(frequencyNumber);
88          }
89      }
90  }
```

1.42 ./csharp/Platform.Data.Doublets.Sequences/Numbers/Unary/PowerOf2ToUnaryNumberConverter.cs

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

```
using Platform.Ranges;
using Platform.Converters;
using System.Runtime.CompilerServices;

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

namespace Platform.Data.Doublets.Numbers.Unary
{
    /// <summary>
    /// <para>
    /// Represents the power of to unary number converter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksOperatorBase{TLinkAddress}"/>
    /// <seealso cref="IConverter{int, TLinkAddress}"/>
    public class PowerOf2ToUnaryNumberConverter<TLinkAddress> : LinksOperatorBase<TLinkAddress>,
    ↪  IConverter<int, TLinkAddress>
    {
        private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
        ↪  EqualityComparer<TLinkAddress>.Default;
        private readonly TLinkAddress[] _unaryNumberPowersOf2;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="PowerOf2ToUnaryNumberConverter"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="one">
        /// <para>A one.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public PowerOf2ToUnaryNumberConverter(ILinks<TLinkAddress> links, TLinkAddress one) :
        ↪  base(links)
        {
            _unaryNumberPowersOf2 = new TLinkAddress[64];
            _unaryNumberPowersOf2[0] = one;
        }

        /// <summary>
        /// <para>
        /// Converts the power.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="power">
        /// <para>The power.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The power of.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLinkAddress Convert(int power)
        {
            Ensure.Always.ArgumentInRange(power, new Range<int>(0, _unaryNumberPowersOf2.Length
            ↪  - 1), nameof(power));
            if (!_equalityComparer.Equals(_unaryNumberPowersOf2[power], default))
            {
                return _unaryNumberPowersOf2[power];
            }
            var previousPowerOf2 = Convert(power - 1);
            var powerOf2 = _links.GetOrCreate(previousPowerOf2, previousPowerOf2);
            _unaryNumberPowersOf2[power] = powerOf2;
            return powerOf2;
        }
    }
}
```

1.43 ./csharp/Platform.Data.Doublets.Sequences/Numbers/Unary/UnaryNumberToAddressAddOperationConverte

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

```
using Platform.Converters;
using Platform.Numbers;

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

namespace Platform.Data.Doublets.Numbers.Unary
{
    /// <summary>
    /// <para>
    /// Represents the unary number to address add operation converter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksOperatorBase{TLinkAddress}"/>
    /// <seealso cref="IConverter{TLinkAddress}"/>
    public class UnaryNumberToAddressAddOperationConverter<TLinkAddress> :
    ↪  LinksOperatorBase<TLinkAddress>, IConverter<TLinkAddress>
    {
        private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
        ↪  EqualityComparer<TLinkAddress>.Default;
        private static readonly UncheckedConverter<TLinkAddress, ulong>
        ↪  _addressToUInt64Converter = UncheckedConverter<TLinkAddress, ulong>.Default;
        private static readonly UncheckedConverter<ulong, TLinkAddress>
        ↪  _uInt64ToAddressConverter = UncheckedConverter<ulong, TLinkAddress>.Default;
        private static readonly TLinkAddress _zero = default;
        private static readonly TLinkAddress _one = Arithmetic.Increment(_zero);
        private readonly Dictionary<TLinkAddress, TLinkAddress> _unaryToUInt64;
        private readonly TLinkAddress _unaryOne;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="UnaryNumberToAddressAddOperationConverter"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="unaryOne">
        /// <para>A unary one.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UnaryNumberToAddressAddOperationConverter(ILinks<TLinkAddress> links,
        ↪  TLinkAddress unaryOne)
            : base(links)
        {
            _unaryOne = unaryOne;
            _unaryToUInt64 = CreateUnaryToUInt64Dictionary(links, unaryOne);
        }

        /// <summary>
        /// <para>
        /// Converts the unary number.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="unaryNumber">
        /// <para>The unary number.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLinkAddress Convert(TLinkAddress unaryNumber)
        {
            if (_equalityComparer.Equals(unaryNumber, default))
            {
                return default;
            }
            if (_equalityComparer.Equals(unaryNumber, _unaryOne))
            {
                return _one;
            }
            var links = _links;
            var source = links.GetSource(unaryNumber);
```

```
            var target = links.GetTarget(unaryNumber);
            if (_equalityComparer.Equals(source, target))
            {
                return _unaryToUInt64[unaryNumber];
            }
            else
            {
                var result = _unaryToUInt64[source];
                TLinkAddress lastValue;
                while (!_unaryToUInt64.TryGetValue(target, out lastValue))
                {
                    source = links.GetSource(target);
                    result = Arithmetic<TLinkAddress>.Add(result, _unaryToUInt64[source]);
                    target = links.GetTarget(target);
                }
                result = Arithmetic<TLinkAddress>.Add(result, lastValue);
                return result;
            }
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static Dictionary<TLinkAddress, TLinkAddress>
        ↪  CreateUnaryToUInt64Dictionary(ILinks<TLinkAddress> links, TLinkAddress unaryOne)
        {
            var unaryToUInt64 = new Dictionary<TLinkAddress, TLinkAddress>
            {
                { unaryOne, _one }
            };
            var unary = unaryOne;
            var number = _one;
            for (var i = 1; i < 64; i++)
            {
                unary = links.GetOrCreate(unary, unary);
                number = Double(number);
                unaryToUInt64.Add(unary, number);
            }
            return unaryToUInt64;
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static TLinkAddress Double(TLinkAddress number) =>
        ↪  _uInt64ToAddressConverter.Convert(_addressToUInt64Converter.Convert(number) * 2UL);
    }
}
```

1.44 ./csharp/Platform.Data.Doublets.Sequences/Numbers/Unary/UnaryNumberToAddressOrOperationConverter

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Reflection;
using Platform.Converters;
using Platform.Numbers;

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

namespace Platform.Data.Doublets.Numbers.Unary
{
    /// <summary>
    /// <para>
    /// Represents the unary number to address or operation converter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksOperatorBase{TLinkAddress}"/>
    /// <seealso cref="IConverter{TLinkAddress}"/>
    public class UnaryNumberToAddressOrOperationConverter<TLinkAddress> :
    ↪  LinksOperatorBase<TLinkAddress>, IConverter<TLinkAddress>
    {
        private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
        ↪  EqualityComparer<TLinkAddress>.Default;
        private static readonly TLinkAddress _zero = default;
        private static readonly TLinkAddress _one = Arithmetic.Increment(_zero);
        private readonly IDictionary<TLinkAddress, int> _unaryNumberPowerOf2Indicies;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="UnaryNumberToAddressOrOperationConverter"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
```

```
        /// <para></para>
        /// </param>
        /// <param name="powerOf2ToUnaryNumberConverter">
        /// <para>A power of to unary number converter.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UnaryNumberToAddressOrOperationConverter(ILinks<TLinkAddress> links,
        ↪  IConverter<int, TLinkAddress> powerOf2ToUnaryNumberConverter) : base(links) =>
        ↪  _unaryNumberPowerOf2Indicies =
        ↪  CreateUnaryNumberPowerOf2IndiciesDictionary(powerOf2ToUnaryNumberConverter);

        /// <summary>
        /// <para>
        /// Converts the source number.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sourceNumber">
        /// <para>The source number.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The target.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLinkAddress Convert(TLinkAddress sourceNumber)
        {
            var links = _links;
            var nullConstant = links.Constants.Null;
            var source = sourceNumber;
            var target = nullConstant;
            if (!_equalityComparer.Equals(source, nullConstant))
            {
                while (true)
                {
                    if (_unaryNumberPowerOf2Indicies.TryGetValue(source, out int powerOf2Index))
                    {
                        SetBit(ref target, powerOf2Index);
                        break;
                    }
                    else
                    {
                        powerOf2Index = _unaryNumberPowerOf2Indicies[links.GetSource(source)];
                        SetBit(ref target, powerOf2Index);
                        source = links.GetTarget(source);
                    }
                }
            }
            return target;
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static Dictionary<TLinkAddress, int>
        ↪  CreateUnaryNumberPowerOf2IndiciesDictionary(IConverter<int, TLinkAddress>
        ↪  powerOf2ToUnaryNumberConverter)
        {
            var unaryNumberPowerOf2Indicies = new Dictionary<TLinkAddress, int>();
            for (int i = 0; i < NumericType<TLinkAddress>.BitsSize; i++)
            {
                unaryNumberPowerOf2Indicies.Add(powerOf2ToUnaryNumberConverter.Convert(i), i);
            }
            return unaryNumberPowerOf2Indicies;
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static void SetBit(ref TLinkAddress target, int powerOf2Index) => target =
        ↪  Bit.Or(target, Bit.ShiftLeft(_one, powerOf2Index));
    }
}
```

1.45 ./csharp/Platform.Data.Doublets.Sequences/Sequences.Experiments.cs

```
// using System;
// using System.Collections.Generic;
// using System.Runtime.CompilerServices;
// using System.Linq;
// using System.Text;
// using Platform.Collections;
// using Platform.Collections.Sets;
```

```
// using Platform.Collections.Stacks;
// using Platform.Data.Exceptions;
// using Platform.Data.Sequences;
// using Platform.Data.Doublets.Sequences.Frequencies.Counters;
// using Platform.Data.Doublets.Sequences.Walkers;
// using LinkIndex = System.UInt64;
// using Stack = System.Collections.Generic.Stack<ulong>;
//
// #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
//
// namespace Platform.Data.Doublets.Sequences
// {
//     /// <summary>
//     /// <para>
//     /// Represents the sequences.
//     /// </para>
//     /// <para></para>
//     /// </summary>
//     partial class Sequences
//     {
//         #region Create All Variants (Not Practical)
//
//         /// <remarks>
//         /// Number of links that is needed to generate all variants for
//         /// sequence of length N corresponds to https://oeis.org/A014143/list sequence.
//         /// </remarks>
//         [MethodImpl(MethodImplOptions.AggressiveInlining)]
//         public ulong[] CreateAllVariants2(ulong[] sequence)
//         {
//             return _sync.DoWrite(() =>
//             {
//                 if (sequence.IsNullOrEmpty())
//                 {
//                     return Array.Empty<ulong>();
//                 }
//                 Links.EnsureLinkExists(sequence);
//                 if (sequence.Length == 1)
//                 {
//                     return sequence;
//                 }
//                 return CreateAllVariants2Core(sequence, 0, (ulong)sequence.Length - 1);
//             });
//         }
//         [MethodImpl(MethodImplOptions.AggressiveInlining)]
//         private ulong[] CreateAllVariants2Core(ulong[] sequence, ulong startAt, ulong stopAt)
//         {
//             if ((stopAt - startAt) == 0)
//             {
//                 return new[] { sequence[startAt] };
//             }
//             if ((stopAt - startAt) == 1)
//             {
//                 return new[] { Links.Unsync.GetOrCreate(sequence[startAt], sequence[stopAt])
↪  };
//             }
//             var variants = new ulong[Platform.Numbers.Math.Catalan(stopAt - startAt)];
//             var last = 0;
//             for (var splitter = startAt; splitter < stopAt; splitter++)
//             {
//                 var left = CreateAllVariants2Core(sequence, startAt, splitter);
//                 var right = CreateAllVariants2Core(sequence, splitter + 1, stopAt);
//                 for (var i = 0; i < left.Length; i++)
//                 {
//                     for (var j = 0; j < right.Length; j++)
//                     {
//                         var variant = Links.Unsync.GetOrCreate(left[i], right[j]);
//                         if (variant == Constants.Null)
//                         {
//                             throw new NotImplementedException("Creation cancellation is not
↪  implemented.");
//                         }
//                         variants[last++] = variant;
//                     }
//                 }
//             }
//             return variants;
//         }
```

```
//
//         /// <summary>
//         /// <para>
//         /// Creates the all variants 1 using the specified sequence.
//         /// </para>
//         /// <para></para>
//         /// </summary>
//         /// <param name="sequence">
//         /// <para>The sequence.</para>
//         /// <para></para>
//         /// </param>
//         /// <returns>
//         /// <para>A list of ulong</para>
//         /// <para></para>
//         /// </returns>
//         [MethodImpl(MethodImplOptions.AggressiveInlining)]
//         public List<ulong> CreateAllVariants1(params ulong[] sequence)
//         {
//             return _sync.DoWrite(() =>
//             {
//                 if (sequence.IsNullOrEmpty())
//                 {
//                     return new List<ulong>();
//                 }
//                 Links.Unsync.EnsureLinkExists(sequence);
//                 if (sequence.Length == 1)
//                 {
//                     return new List<ulong> { sequence[0] };
//                 }
//                 var results = new
    ↪  List<ulong>((int)Platform.Numbers.Math.Catalan((ulong)sequence.Length));
//                 return CreateAllVariants1Core(sequence, results);
//             });
//         }
//         [MethodImpl(MethodImplOptions.AggressiveInlining)]
//         private List<ulong> CreateAllVariants1Core(ulong[] sequence, List<ulong> results)
//         {
//             if (sequence.Length == 2)
//             {
//                 var link = Links.Unsync.GetOrCreate(sequence[0], sequence[1]);
//                 if (link == Constants.Null)
//                 {
//                     throw new NotImplementedException("Creation cancellation is not
    ↪  implemented.");
//                 }
//                 results.Add(link);
//                 return results;
//             }
//             var innerSequenceLength = sequence.Length - 1;
//             var innerSequence = new ulong[innerSequenceLength];
//             for (var li = 0; li < innerSequenceLength; li++)
//             {
//                 var link = Links.Unsync.GetOrCreate(sequence[li], sequence[li + 1]);
//                 if (link == Constants.Null)
//                 {
//                     throw new NotImplementedException("Creation cancellation is not
    ↪  implemented.");
//                 }
//                 for (var isi = 0; isi < li; isi++)
//                 {
//                     innerSequence[isi] = sequence[isi];
//                 }
//                 innerSequence[li] = link;
//                 for (var isi = li + 1; isi < innerSequenceLength; isi++)
//                 {
//                     innerSequence[isi] = sequence[isi + 1];
//                 }
//                 CreateAllVariants1Core(innerSequence, results);
//             }
//             return results;
//         }
//
//         #endregion
//
//         /// <summary>
//         /// <para>
//         /// Eaches the 1 using the specified sequence.
```

```
//          /// </para>
//          /// <para></para>
//          /// </summary>
//          /// <param name="sequence">
//          /// <para>The sequence.</para>
//          /// <para></para>
//          /// </param>
//          /// <returns>
//          /// <para>The visited links.</para>
//          /// <para></para>
//          /// </returns>
//          [MethodImpl(MethodImplOptions.AggressiveInlining)]
//          public HashSet<ulong> Each1(params ulong[] sequence)
//          {
//              var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
//              Each1(link =>
//              {
//                  if (!visitedLinks.Contains(link))
//                  {
//                      visitedLinks.Add(link); // изучить почему случаются повторы
//                  }
//                  return true;
//              }, sequence);
//              return visitedLinks;
//          }
//          [MethodImpl(MethodImplOptions.AggressiveInlining)]
//          private void Each1(Func<ulong, bool> handler, params ulong[] sequence)
//          {
//              if (sequence.Length == 2)
//              {
//                  Links.Unsync.Each(sequence[0], sequence[1], handler);
//              }
//              else
//              {
//                  var innerSequenceLength = sequence.Length - 1;
//                  for (var li = 0; li < innerSequenceLength; li++)
//                  {
//                      var left = sequence[li];
//                      var right = sequence[li + 1];
//                      if (left == 0 && right == 0)
//                      {
//                          continue;
//                      }
//                      var linkIndex = li;
//                      ulong[] innerSequence = null;
//                      Links.Unsync.Each(doublet =>
//                      {
//                          if (innerSequence == null)
//                          {
//                              innerSequence = new ulong[innerSequenceLength];
//                              for (var isi = 0; isi < linkIndex; isi++)
//                              {
//                                  innerSequence[isi] = sequence[isi];
//                              }
//                              for (var isi = linkIndex + 1; isi < innerSequenceLength; isi++)
//                              {
//                                  innerSequence[isi] = sequence[isi + 1];
//                              }
//                          }
//                          innerSequence[linkIndex] = doublet[Constants.IndexPart];
//                          Each1(handler, innerSequence);
//                          return Constants.Continue;
//                      }, Constants.Any, left, right);
//                  }
//              }
//          }
//
//          /// <summary>
//          /// <para>
//          /// Eaches the part using the specified sequence.
//          /// </para>
//          /// <para></para>
//          /// </summary>
//          /// <param name="sequence">
//          /// <para>The sequence.</para>
//          /// <para></para>
//          /// </param>
//          /// <returns>
```

```
235 //         /// <para>The visited links.</para>
236 //         /// <para></para>
237 //         /// </returns>
238 //         [MethodImpl(MethodImplOptions.AggressiveInlining)]
239 //         public HashSet<ulong> EachPart(params ulong[] sequence)
240 //         {
241 //             var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
242 //             EachPartCore(link =>
243 //             {
244 //                 var linkIndex = link[Constants.IndexPart];
245 //                 if (!visitedLinks.Contains(linkIndex))
246 //                 {
247 //                     visitedLinks.Add(linkIndex); // изучить почему случаются повторы
248 //                 }
249 //                 return Constants.Continue;
250 //             }, sequence);
251 //             return visitedLinks;
252 //         }
253 //
254 //         /// <summary>
255 //         /// <para>
256 //         /// Eaches the part using the specified handler.
257 //         /// </para>
258 //         /// <para></para>
259 //         /// </summary>
260 //         /// <param name="handler">
261 //         /// <para>The handler.</para>
262 //         /// <para></para>
263 //         /// </param>
264 //         /// <param name="sequence">
265 //         /// <para>The sequence.</para>
266 //         /// <para></para>
267 //         /// </param>
268 //         [MethodImpl(MethodImplOptions.AggressiveInlining)]
269 //         public void EachPart(Func<IList<LinkIndex>, LinkIndex> handler, params ulong[]
    ↪ sequence)
270 //         {
271 //             var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
272 //             EachPartCore(link =>
273 //             {
274 //                 var linkIndex = link[Constants.IndexPart];
275 //                 if (!visitedLinks.Contains(linkIndex))
276 //                 {
277 //                     visitedLinks.Add(linkIndex); // изучить почему случаются повторы
278 //                     return handler(new LinkAddress<LinkIndex>(linkIndex));
279 //                 }
280 //                 return Constants.Continue;
281 //             }, sequence);
282 //         }
283 //         [MethodImpl(MethodImplOptions.AggressiveInlining)]
284 //         private void EachPartCore(Func<IList<LinkIndex>, LinkIndex> handler, params ulong[]
    ↪ sequence)
285 //         {
286 //             if (sequence.IsNullOrEmpty())
287 //             {
288 //                 return;
289 //             }
290 //             Links.EnsureLinkIsAnyOrExists(sequence);
291 //             if (sequence.Length == 1)
292 //             {
293 //                 var link = sequence[0];
294 //                 if (link > 0)
295 //                 {
296 //                     handler(new LinkAddress<LinkIndex>(link));
297 //                 }
298 //                 else
299 //                 {
300 //                     Links.Each(Constants.Any, Constants.Any, handler);
301 //                 }
302 //             }
303 //             else if (sequence.Length == 2)
304 //             {
305 //                 //_links.Each(sequence[0], sequence[1], handler);
306 //                 //  o_|      x_o ...
307 //                 // x_|        |___|
308 //                 Links.Each(sequence[1], Constants.Any, doublet =>
309 //                 {
```

```
//                      var match = Links.SearchOrDefault(sequence[0], doublet);
//                      if (match != Constants.Null)
//                      {
//                          handler(new LinkAddress<LinkIndex>(match));
//                      }
//                      return true;
//                  });
//                  // |_x      ... x_o
//                  //  |_o      |___|
//                  Links.Each(Constants.Any, sequence[0], doublet =>
//                  {
//                      var match = Links.SearchOrDefault(doublet, sequence[1]);
//                      if (match != 0)
//                      {
//                          handler(new LinkAddress<LinkIndex>(match));
//                      }
//                      return true;
//                  });
//                  //          ._x o_.
//                  //           |___|
//                  PartialStepRight(x => handler(x), sequence[0], sequence[1]);
//              }
//              else
//              {
//                  throw new NotImplementedException();
//              }
//          }
//          [MethodImpl(MethodImplOptions.AggressiveInlining)]
//          private void PartialStepRight(Action<IList<LinkIndex>> handler, ulong left, ulong
    ↪  right)
//          {
//              Links.Unsync.Each(Constants.Any, left, doublet =>
//              {
//                  StepRight(handler, doublet, right);
//                  if (left != doublet)
//                  {
//                      PartialStepRight(handler, doublet, right);
//                  }
//                  return true;
//              });
//          }
//          [MethodImpl(MethodImplOptions.AggressiveInlining)]
//          private void StepRight(Action<IList<LinkIndex>> handler, ulong left, ulong right)
//          {
//              Links.Unsync.Each(left, Constants.Any, rightStep =>
//              {
//                  TryStepRightUp(handler, right, rightStep);
//                  return true;
//              });
//          }
//          [MethodImpl(MethodImplOptions.AggressiveInlining)]
//          private void TryStepRightUp(Action<IList<LinkIndex>> handler, ulong right, ulong
    ↪  stepFrom)
//          {
//              var upStep = stepFrom;
//              var firstSource = Links.Unsync.GetTarget(upStep);
//              while (firstSource != right && firstSource != upStep)
//              {
//                  upStep = firstSource;
//                  firstSource = Links.Unsync.GetSource(upStep);
//              }
//              if (firstSource == right)
//              {
//                  handler(new LinkAddress<LinkIndex>(stepFrom));
//              }
//          }
//
//          // TODO: Test
//          [MethodImpl(MethodImplOptions.AggressiveInlining)]
//          private void PartialStepLeft(Action<IList<LinkIndex>> handler, ulong left, ulong
    ↪  right)
//          {
//              Links.Unsync.Each(right, Constants.Any, doublet =>
//              {
//                  StepLeft(handler, left, doublet);
//                  if (right != doublet)
//                  {
```

```
//                      PartialStepLeft(handler, left, doublet);
//                  }
//                  return true;
//              });
//          }
//          [MethodImpl(MethodImplOptions.AggressiveInlining)]
//          private void StepLeft(Action<IList<LinkIndex>> handler, ulong left, ulong right)
//          {
//              Links.Unsync.Each(Constants.Any, right, leftStep =>
//              {
//                  TryStepLeftUp(handler, left, leftStep);
//                  return true;
//              });
//          }
//          [MethodImpl(MethodImplOptions.AggressiveInlining)]
//          private void TryStepLeftUp(Action<IList<LinkIndex>> handler, ulong left, ulong
    ↪  stepFrom)
//          {
//              var upStep = stepFrom;
//              var firstTarget = Links.Unsync.GetSource(upStep);
//              while (firstTarget != left && firstTarget != upStep)
//              {
//                  upStep = firstTarget;
//                  firstTarget = Links.Unsync.GetTarget(upStep);
//              }
//              if (firstTarget == left)
//              {
//                  handler(new LinkAddress<LinkIndex>(stepFrom));
//              }
//          }
//          [MethodImpl(MethodImplOptions.AggressiveInlining)]
//          private bool StartsWith(ulong sequence, ulong link)
//          {
//              var upStep = sequence;
//              var firstSource = Links.Unsync.GetSource(upStep);
//              while (firstSource != link && firstSource != upStep)
//              {
//                  upStep = firstSource;
//                  firstSource = Links.Unsync.GetSource(upStep);
//              }
//              return firstSource == link;
//          }
//          [MethodImpl(MethodImplOptions.AggressiveInlining)]
//          private bool EndsWith(ulong sequence, ulong link)
//          {
//              var upStep = sequence;
//              var lastTarget = Links.Unsync.GetTarget(upStep);
//              while (lastTarget != link && lastTarget != upStep)
//              {
//                  upStep = lastTarget;
//                  lastTarget = Links.Unsync.GetTarget(upStep);
//              }
//              return lastTarget == link;
//          }
//
//          /// <summary>
//          /// <para>
//          /// Gets the all matching sequences 0 using the specified sequence.
//          /// </para>
//          /// <para></para>
//          /// </summary>
//          /// <param name="sequence">
//          /// <para>The sequence.</para>
//          /// <para></para>
//          /// </param>
//          /// <returns>
//          /// <para>A list of ulong</para>
//          /// <para></para>
//          /// </returns>
//          [MethodImpl(MethodImplOptions.AggressiveInlining)]
//          public List<ulong> GetAllMatchingSequences0(params ulong[] sequence)
//          {
//              return _sync.DoRead(() =>
//              {
//                  var results = new List<ulong>();
//                  if (sequence.Length > 0)
//                  {
```

```
//                      Links.EnsureLinkExists(sequence);
//                      var firstElement = sequence[0];
//                      if (sequence.Length == 1)
//                      {
//                          results.Add(firstElement);
//                          return results;
//                      }
//                      if (sequence.Length == 2)
//                      {
//                          var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
//                          if (doublet != Constants.Null)
//                          {
//                              results.Add(doublet);
//                          }
//                          return results;
//                      }
//                      var linksInSequence = new HashSet<ulong>(sequence);
//                      void handler(IList<LinkIndex> result)
//                      {
//                          var resultIndex = result[Links.Constants.IndexPart];
//                          var filterPosition = 0;
//                          StopableSequenceWalker.WalkRight(resultIndex, Links.Unsync.GetSource,
    ↪ Links.Unsync.GetTarget,
//                              x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) ==
    ↪ x, x =>
//                              {
//                                  if (filterPosition == sequence.Length)
//                                  {
//                                      filterPosition = -2; // Длиннее чем нужно
//                                      return false;
//                                  }
//                                  if (x != sequence[filterPosition])
//                                  {
//                                      filterPosition = -1;
//                                      return false; // Начинается иначе
//                                  }
//                                  filterPosition++;
//
//                                  return true;
//                              });
//                          if (filterPosition == sequence.Length)
//                          {
//                              results.Add(resultIndex);
//                          }
//                      }
//                      if (sequence.Length >= 2)
//                      {
//                          StepRight(handler, sequence[0], sequence[1]);
//                      }
//                      var last = sequence.Length - 2;
//                      for (var i = 1; i < last; i++)
//                      {
//                          PartialStepRight(handler, sequence[i], sequence[i + 1]);
//                      }
//                      if (sequence.Length >= 3)
//                      {
//                          StepLeft(handler, sequence[sequence.Length - 2],
    ↪ sequence[sequence.Length - 1]);
//                      }
//                  }
//                  return results;
//              });
//          }
//
//          /// <summary>
//          /// <para>
//          /// Gets the all matching sequences 1 using the specified sequence.
//          /// </para>
//          /// <para></para>
//          /// </summary>
//          /// <param name="sequence">
//          /// <para>The sequence.</para>
//          /// <para></para>
//          /// </param>
//          /// <returns>
//          /// <para>A hash set of ulong</para>
//          /// <para></para>
//          /// </returns>
```

```
//         [MethodImpl(MethodImplOptions.AggressiveInlining)]
//         public HashSet<ulong> GetAllMatchingSequences1(params ulong[] sequence)
//         {
//             return _sync.DoRead(() =>
//             {
//                 var results = new HashSet<ulong>();
//                 if (sequence.Length > 0)
//                 {
//                     Links.EnsureLinkExists(sequence);
//                     var firstElement = sequence[0];
//                     if (sequence.Length == 1)
//                     {
//                         results.Add(firstElement);
//                         return results;
//                     }
//                     if (sequence.Length == 2)
//                     {
//                         var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
//                         if (doublet != Constants.Null)
//                         {
//                             results.Add(doublet);
//                         }
//                         return results;
//                     }
//                     var matcher = new Matcher(this, sequence, results, null);
//                     if (sequence.Length >= 2)
//                     {
//                         StepRight(matcher.AddFullMatchedToResults, sequence[0], sequence[1]);
//                     }
//                     var last = sequence.Length - 2;
//                     for (var i = 1; i < last; i++)
//                     {
//                         PartialStepRight(matcher.AddFullMatchedToResults, sequence[i],
    ↪ sequence[i + 1]);
//                     }
//                     if (sequence.Length >= 3)
//                     {
//                         StepLeft(matcher.AddFullMatchedToResults, sequence[sequence.Length -
    ↪ 2], sequence[sequence.Length - 1]);
//                     }
//                 }
//                 return results;
//             });
//         }
//
//         /// <summary>
//         /// <para>
//         /// The max sequence format size.
//         /// </para>
//         /// <para></para>
//         /// </summary>
//         public const int MaxSequenceFormatSize = 200;
//
//         /// <summary>
//         /// <para>
//         /// Formats the sequence using the specified sequence link.
//         /// </para>
//         /// <para></para>
//         /// </summary>
//         /// <param name="sequenceLink">
//         /// <para>The sequence link.</para>
//         /// <para></para>
//         /// </param>
//         /// <param name="knownElements">
//         /// <para>The known elements.</para>
//         /// <para></para>
//         /// </param>
//         /// <returns>
//         /// <para>The string</para>
//         /// <para></para>
//         /// </returns>
//         [MethodImpl(MethodImplOptions.AggressiveInlining)]
//         public string FormatSequence(LinkIndex sequenceLink, params LinkIndex[]
    ↪ knownElements) => FormatSequence(sequenceLink, (sb, x) => sb.Append(x), true, knownElements);
//
//         /// <summary>
//         /// <para>
```

```
609 //         /// Formats the sequence using the specified sequence link.
610 //         /// </para>
611 //         /// <para></para>
612 //         /// </summary>
613 //         /// <param name="sequenceLink">
614 //         /// <para>The sequence link.</para>
615 //         /// <para></para>
616 //         /// </param>
617 //         /// <param name="elementToString">
618 //         /// <para>The element to string.</para>
619 //         /// <para></para>
620 //         /// </param>
621 //         /// <param name="insertComma">
622 //         /// <para>The insert comma.</para>
623 //         /// <para></para>
624 //         /// </param>
625 //         /// <param name="knownElements">
626 //         /// <para>The known elements.</para>
627 //         /// <para></para>
628 //         /// </param>
629 //         /// <returns>
630 //         /// <para>The string</para>
631 //         /// <para></para>
632 //         /// </returns>
633 //         [MethodImpl(MethodImplOptions.AggressiveInlining)]
634 //         public string FormatSequence(LinkIndex sequenceLink, Action<StringBuilder, LinkIndex>
    ↪ elementToString, bool insertComma, params LinkIndex[] knownElements) =>
    ↪ Links.SyncRoot.DoRead(() => FormatSequence(Links.Unsync, sequenceLink, elementToString,
    ↪ insertComma, knownElements));
635 //         [MethodImpl(MethodImplOptions.AggressiveInlining)]
636 //         private string FormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
    ↪ Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params LinkIndex[]
    ↪ knownElements)
637 //         {
638 //             var linksInSequence = new HashSet<ulong>(knownElements);
639 //             //var entered = new HashSet<ulong>();
640 //             var sb = new StringBuilder();
641 //             sb.Append('{');
642 //             if (links.Exists(sequenceLink))
643 //             {
644 //                 StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource,
    ↪ links.GetTarget,
645 //                     x => linksInSequence.Contains(x) || links.IsPartialPoint(x), element =>
    ↪ // entered.AddAndReturnVoid, x => { }, entered.DoNotContains
646 //                     {
647 //                         if (insertComma && sb.Length > 1)
648 //                         {
649 //                             sb.Append(',');
650 //                         }
651 //                         //if (entered.Contains(element))
652 //                         //{
653 //                         //    sb.Append('{');
654 //                         //    elementToString(sb, element);
655 //                         //    sb.Append('}');
656 //                         //}
657 //                         //else
658 //                         elementToString(sb, element);
659 //                         if (sb.Length < MaxSequenceFormatSize)
660 //                         {
661 //                             return true;
662 //                         }
663 //                         sb.Append(insertComma ? ", ..." : "...");
664 //                         return false;
665 //                     });
666 //             }
667 //             sb.Append('}');
668 //             return sb.ToString();
669 //         }
670 //
671 //         /// <summary>
672 //         /// <para>
673 //         /// Safes the format sequence using the specified sequence link.
674 //         /// </para>
675 //         /// <para></para>
676 //         /// </summary>
677 //         /// <param name="sequenceLink">
678 //         /// <para>The sequence link.</para>
```

```
679 //         /// <para></para>
680 //         /// </param>
681 //         /// <param name="knownElements">
682 //         /// <para>The known elements.</para>
683 //         /// <para></para>
684 //         /// </param>
685 //         /// <returns>
686 //         /// <para>The string</para>
687 //         /// <para></para>
688 //         /// </returns>
689 //         [MethodImpl(MethodImplOptions.AggressiveInlining)]
690 //         public string SafeFormatSequence(LinkIndex sequenceLink, params LinkIndex[]
    ↪ knownElements) => SafeFormatSequence(sequenceLink, (sb, x) => sb.Append(x), true,
    ↪ knownElements);
691 //
692 //         /// <summary>
693 //         /// <para>
694 //         /// Safes the format sequence using the specified sequence link.
695 //         /// </para>
696 //         /// <para></para>
697 //         /// </summary>
698 //         /// <param name="sequenceLink">
699 //         /// <para>The sequence link.</para>
700 //         /// <para></para>
701 //         /// </param>
702 //         /// <param name="elementToString">
703 //         /// <para>The element to string.</para>
704 //         /// <para></para>
705 //         /// </param>
706 //         /// <param name="insertComma">
707 //         /// <para>The insert comma.</para>
708 //         /// <para></para>
709 //         /// </param>
710 //         /// <param name="knownElements">
711 //         /// <para>The known elements.</para>
712 //         /// <para></para>
713 //         /// </param>
714 //         /// <returns>
715 //         /// <para>The string</para>
716 //         /// <para></para>
717 //         /// </returns>
718 //         [MethodImpl(MethodImplOptions.AggressiveInlining)]
719 //         public string SafeFormatSequence(LinkIndex sequenceLink, Action<StringBuilder,
    ↪ LinkIndex> elementToString, bool insertComma, params LinkIndex[] knownElements) =>
    ↪ Links.SyncRoot.DoRead(() => SafeFormatSequence(Links.Unsync, sequenceLink, elementToString,
    ↪ insertComma, knownElements));
720 //         [MethodImpl(MethodImplOptions.AggressiveInlining)]
721 //         private string SafeFormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
    ↪ Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params LinkIndex[]
    ↪ knownElements)
722 //         {
723 //             var linksInSequence = new HashSet<ulong>(knownElements);
724 //             var entered = new HashSet<ulong>();
725 //             var sb = new StringBuilder();
726 //             sb.Append('{');
727 //             if (links.Exists(sequenceLink))
728 //             {
729 //                 StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource,
    ↪ links.GetTarget,
730 //                     x => linksInSequence.Contains(x) || links.IsFullPoint(x),
    ↪ entered.AddAndReturnVoid, x => { }, entered.DoNotContains, element =>
731 //                     {
732 //                         if (insertComma && sb.Length > 1)
733 //                         {
734 //                             sb.Append(',');
735 //                         }
736 //                         if (entered.Contains(element))
737 //                         {
738 //                             sb.Append('{');
739 //                             elementToString(sb, element);
740 //                             sb.Append('}');
741 //                         }
742 //                         else
743 //                         {
744 //                             elementToString(sb, element);
745 //                         }
746 //                         if (sb.Length < MaxSequenceFormatSize)
```

```
//                          {
//                              return true;
//                          }
//                          sb.Append(insertComma ? ", ..." : "...");
//                          return false;
//                      });
//              }
//              sb.Append('}');
//              return sb.ToString();
//          }
//
//          /// <summary>
//          /// <para>
//          /// Gets the all partially matching sequences 0 using the specified sequence.
//          /// </para>
//          /// <para></para>
//          /// </summary>
//          /// <param name="sequence">
//          /// <para>The sequence.</para>
//          /// <para></para>
//          /// </param>
//          /// <returns>
//          /// <para>A list of ulong</para>
//          /// <para></para>
//          /// </returns>
//          [MethodImpl(MethodImplOptions.AggressiveInlining)]
//          public List<ulong> GetAllPartiallyMatchingSequences0(params ulong[] sequence)
//          {
//              return _sync.DoRead(() =>
//              {
//                  if (sequence.Length > 0)
//                  {
//                      Links.EnsureLinkExists(sequence);
//                      var results = new HashSet<ulong>();
//                      for (var i = 0; i < sequence.Length; i++)
//                      {
//                          AllUsagesCore(sequence[i], results);
//                      }
//                      var filteredResults = new List<ulong>();
//                      var linksInSequence = new HashSet<ulong>(sequence);
//                      foreach (var result in results)
//                      {
//                          var filterPosition = -1;
//                          StopableSequenceWalker.WalkRight(result, Links.Unsync.GetSource,
    ↪ Links.Unsync.GetTarget,
//                              x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) ==
    ↪ x, x =>
//                              {
//                                  if (filterPosition == (sequence.Length - 1))
//                                  {
//                                      return false;
//                                  }
//                                  if (filterPosition >= 0)
//                                  {
//                                      if (x == sequence[filterPosition + 1])
//                                      {
//                                          filterPosition++;
//                                      }
//                                      else
//                                      {
//                                          return false;
//                                      }
//                                  }
//                                  if (filterPosition < 0)
//                                  {
//                                      if (x == sequence[0])
//                                      {
//                                          filterPosition = 0;
//                                      }
//                                  }
//                                  return true;
//                              });
//                          if (filterPosition == (sequence.Length - 1))
//                          {
//                              filteredResults.Add(result);
//                          }
//                      }
//                      return filteredResults;
```

```
//                 }
//                 return new List<ulong>();
//             });
//         }
//
//         /// <summary>
//         /// <para>
//         /// Gets the all partially matching sequences 1 using the specified sequence.
//         /// </para>
//         /// <para></para>
//         /// </summary>
//         /// <param name="sequence">
//         /// <para>The sequence.</para>
//         /// <para></para>
//         /// </param>
//         /// <returns>
//         /// <para>A hash set of ulong</para>
//         /// <para></para>
//         /// </returns>
//         [MethodImpl(MethodImplOptions.AggressiveInlining)]
//         public HashSet<ulong> GetAllPartiallyMatchingSequences1(params ulong[] sequence)
//         {
//             return _sync.DoRead(() =>
//             {
//                 if (sequence.Length > 0)
//                 {
//                     Links.EnsureLinkExists(sequence);
//                     var results = new HashSet<ulong>();
//                     for (var i = 0; i < sequence.Length; i++)
//                     {
//                         AllUsagesCore(sequence[i], results);
//                     }
//                     var filteredResults = new HashSet<ulong>();
//                     var matcher = new Matcher(this, sequence, filteredResults, null);
//                     matcher.AddAllPartialMatchedToResults(results);
//                     return filteredResults;
//                 }
//                 return new HashSet<ulong>();
//             });
//         }
//
//         /// <summary>
//         /// <para>
//         /// Determines whether this instance get all partially matching sequences 2.
//         /// </para>
//         /// <para></para>
//         /// </summary>
//         /// <param name="handler">
//         /// <para>The handler.</para>
//         /// <para></para>
//         /// </param>
//         /// <param name="sequence">
//         /// <para>The sequence.</para>
//         /// <para></para>
//         /// </param>
//         /// <returns>
//         /// <para>The bool</para>
//         /// <para></para>
//         /// </returns>
//         [MethodImpl(MethodImplOptions.AggressiveInlining)]
//         public bool GetAllPartiallyMatchingSequences2(Func<IList<LinkIndex>, LinkIndex>
    ↪ handler, params ulong[] sequence)
//         {
//             return _sync.DoRead(() =>
//             {
//                 if (sequence.Length > 0)
//                 {
//                     Links.EnsureLinkExists(sequence);
//
//                     var results = new HashSet<ulong>();
//                     var filteredResults = new HashSet<ulong>();
//                     var matcher = new Matcher(this, sequence, filteredResults, handler);
//                     for (var i = 0; i < sequence.Length; i++)
//                     {
//                         if (!AllUsagesCore1(sequence[i], results,
    ↪ matcher.HandlePartialMatched))
//                         {
```

```
//                                return false;
//                            }
//                        }
//                        return true;
//                    }
//                    return true;
//                });
//            }
//
//            //public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
//            //{
//            //    return Sync.DoRead(() =>
//            //    {
//            //        if (sequence.Length > 0)
//            //        {
//            //            _links.EnsureEachLinkIsAnyOrExists(sequence);
//            //
//            //            var firstResults = new HashSet<ulong>();
//            //            var lastResults = new HashSet<ulong>();
//            //
//            //            var first = sequence.First(x => x != LinksConstants.Any);
//            //            var last = sequence.Last(x => x != LinksConstants.Any);
//            //
//            //            AllUsagesCore(first, firstResults);
//            //            AllUsagesCore(last, lastResults);
//            //
//            //            firstResults.IntersectWith(lastResults);
//            //
//            //            //for (var i = 0; i < sequence.Length; i++)
//            //            //    AllUsagesCore(sequence[i], results);
//            //
//            //            var filteredResults = new HashSet<ulong>();
//            //            var matcher = new Matcher(this, sequence, filteredResults, null);
//            //            matcher.AddAllPartialMatchedToResults(firstResults);
//            //            return filteredResults;
//            //        }
//            //
//            //        return new HashSet<ulong>();
//            //    });
//            //}
//            //
//            /// <summary>
//            /// <para>
//            /// Gets the all partially matching sequences 3 using the specified sequence.
//            /// </para>
//            /// <para></para>
//            /// </summary>
//            /// <param name="sequence">
//            /// <para>The sequence.</para>
//            /// <para></para>
//            /// </param>
//            /// <returns>
//            /// <para>A hash set of ulong</para>
//            /// <para></para>
//            /// </returns>
//            [MethodImpl(MethodImplOptions.AggressiveInlining)]
//            public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
//            {
//                return _sync.DoRead(() =>
//                {
//                    if (sequence.Length > 0)
//                    {
//                        ILinksExtensions.EnsureLinkIsAnyOrExists(Links, sequence);
//                        var firstResults = new HashSet<ulong>();
//                        var lastResults = new HashSet<ulong>();
//                        var first = sequence.First(x => x != Constants.Any);
//                        var last = sequence.Last(x => x != Constants.Any);
//                        AllUsagesCore(first, firstResults);
//                        AllUsagesCore(last, lastResults);
//                        firstResults.IntersectWith(lastResults);
//                        //for (var i = 0; i < sequence.Length; i++)
//                        //    AllUsagesCore(sequence[i], results);
//                        var filteredResults = new HashSet<ulong>();
//                        var matcher = new Matcher(this, sequence, filteredResults, null);
//                        matcher.AddAllPartialMatchedToResults(firstResults);
//                        return filteredResults;
//                    }
//                    return new HashSet<ulong>();
```

```
//             });
//         }
//
//         /// <summary>
//         /// <para>
//         /// Gets the all partially matching sequences 4 using the specified read as elements.
//         /// </para>
//         /// <para></para>
//         /// </summary>
//         /// <param name="readAsElements">
//         /// <para>The read as elements.</para>
//         /// <para></para>
//         /// </param>
//         /// <param name="sequence">
//         /// <para>The sequence.</para>
//         /// <para></para>
//         /// </param>
//         /// <returns>
//         /// <para>A hash set of ulong</para>
//         /// <para></para>
//         /// </returns>
//         [MethodImpl(MethodImplOptions.AggressiveInlining)]
//         public HashSet<ulong> GetAllPartiallyMatchingSequences4(HashSet<ulong>
    ↪ readAsElements, IList<ulong> sequence)
//         {
//             return _sync.DoRead(() =>
//             {
//                 if (sequence.Count > 0)
//                 {
//                     Links.EnsureLinkExists(sequence);
//                     var results = new HashSet<LinkIndex>();
//                     //var nextResults = new HashSet<ulong>();
//                     //for (var i = 0; i < sequence.Length; i++)
//                     //{
//                     //    AllUsagesCore(sequence[i], nextResults);
//                     //    if (results.IsNullOrEmpty())
//                     //    {
//                     //        results = nextResults;
//                     //        nextResults = new HashSet<ulong>();
//                     //    }
//                     //    else
//                     //    {
//                     //        results.IntersectWith(nextResults);
//                     //        nextResults.Clear();
//                     //    }
//                     //}
//                     var collector1 = new AllUsagesCollector1(Links.Unsync, results);
//                     collector1.Collect(Links.Unsync.GetLink(sequence[0]));
//                     var next = new HashSet<ulong>();
//                     for (var i = 1; i < sequence.Count; i++)
//                     {
//                         var collector = new AllUsagesCollector1(Links.Unsync, next);
//                         collector.Collect(Links.Unsync.GetLink(sequence[i]));
//
//                         results.IntersectWith(next);
//                         next.Clear();
//                     }
//                     var filteredResults = new HashSet<ulong>();
//                     var matcher = new Matcher(this, sequence, filteredResults, null,
    ↪ readAsElements);
//                     matcher.AddAllPartialMatchedToResultsAndReadAsElements(results.OrderBy(x
    ↪ => x)); // OrderBy is a Hack
//                     return filteredResults;
//                 }
//                 return new HashSet<ulong>();
//             });
//         }
//
//         // Does not work
//         //public HashSet<ulong> GetAllPartiallyMatchingSequences5(HashSet<ulong>
    ↪ readAsElements, params ulong[] sequence)
//         //{
//         //    var visited = new HashSet<ulong>();
//         //    var results = new HashSet<ulong>();
//         //    var matcher = new Matcher(this, sequence, visited, x => { results.Add(x);
    ↪ return true; }, readAsElements);
//         //    var last = sequence.Length - 1;
```

```
1048 //          //    for (var i = 0; i < last; i++)
1049 //          //    {
1050 //          //        PartialStepRight(matcher.PartialMatch, sequence[i], sequence[i + 1]);
1051 //          //    }
1052 //          //    return results;
1053 //          //}
1054 //
1055 //          /// <summary>
1056 //          /// <para>
1057 //          /// Gets the all partially matching sequences using the specified sequence.
1058 //          /// </para>
1059 //          /// <para></para>
1060 //          /// </summary>
1061 //          /// <param name="sequence">
1062 //          /// <para>The sequence.</para>
1063 //          /// <para></para>
1064 //          /// </param>
1065 //          /// <returns>
1066 //          /// <para>A list of ulong</para>
1067 //          /// <para></para>
1068 //          /// </returns>
1069 //          [MethodImpl(MethodImplOptions.AggressiveInlining)]
1070 //          public List<ulong> GetAllPartiallyMatchingSequences(params ulong[] sequence)
1071 //          {
1072 //              return _sync.DoRead(() =>
1073 //              {
1074 //                  if (sequence.Length > 0)
1075 //                  {
1076 //                      Links.EnsureLinkExists(sequence);
1077 //                      //var firstElement = sequence[0];
1078 //                      //if (sequence.Length == 1)
1079 //                      //{
1080 //                      //    //results.Add(firstElement);
1081 //                      //    return results;
1082 //                      //}
1083 //                      //if (sequence.Length == 2)
1084 //                      //{
1085 //                      //    //var doublet = _links.SearchCore(firstElement, sequence[1]);
1086 //                      //    //if (doublet != Doublets.Links.Null)
1087 //                      //    //    results.Add(doublet);
1088 //                      //    return results;
1089 //                      //}
1090 //                      //var lastElement = sequence[sequence.Length - 1];
1091 //                      //Func<ulong, bool> handler = x =>
1092 //                      //{
1093 //                      //    if (StartsWith(x, firstElement) && EndsWith(x, lastElement))
     ↪  results.Add(x);
1094 //                      //    return true;
1095 //                      //};
1096 //                      //if (sequence.Length >= 2)
1097 //                      //    StepRight(handler, sequence[0], sequence[1]);
1098 //                      //var last = sequence.Length - 2;
1099 //                      //for (var i = 1; i < last; i++)
1100 //                      //    PartialStepRight(handler, sequence[i], sequence[i + 1]);
1101 //                      //if (sequence.Length >= 3)
1102 //                      //    StepLeft(handler, sequence[sequence.Length - 2],
     ↪  sequence[sequence.Length - 1]);
1103 //                      //////if (sequence.Length == 1)
1104 //                      //////{
1105 //                      //////    throw new NotImplementedException(); // all sequences,
     ↪  containing this element?
1106 //                      //////}
1107 //                      //////if (sequence.Length == 2)
1108 //                      //////{
1109 //                      //////    var results = new List<ulong>();
1110 //                      //////    PartialStepRight(results.Add, sequence[0], sequence[1]);
1111 //                      //////    return results;
1112 //                      //////}
1113 //                      //////var matches = new List<List<ulong>>();
1114 //                      //////var last = sequence.Length - 1;
1115 //                      //////for (var i = 0; i < last; i++)
1116 //                      //////{
1117 //                      //////    var results = new List<ulong>();
1118 //                      //////    //StepRight(results.Add, sequence[i], sequence[i + 1]);
1119 //                      //////    PartialStepRight(results.Add, sequence[i], sequence[i + 1]);
1120 //                      //////    if (results.Count > 0)
1121 //                      //////        matches.Add(results);
```

```
//                      //////    else
//                      //////        return results;
//                      //////    if (matches.Count == 2)
//                      //////    {
//                      //////        var merged = new List<ulong>();
//                      //////        for (var j = 0; j < matches[0].Count; j++)
//                      //////            for (var k = 0; k < matches[1].Count; k++)
//                      //////                CloseInnerConnections(merged.Add, matches[0][j],
    ↪ matches[1][k]);
//                      //////        if (merged.Count > 0)
//                      //////            matches = new List<List<ulong>> { merged };
//                      //////        else
//                      //////            return new List<ulong>();
//                      //////    }
//                      //////}
//                      //////if (matches.Count > 0)
//                      //////{
//                      //////    var usages = new HashSet<ulong>();
//                      //////    for (int i = 0; i < sequence.Length; i++)
//                      //////    {
//                      //////        AllUsagesCore(sequence[i], usages);
//                      //////    }
//                      //////    //for (int i = 0; i < matches[0].Count; i++)
//                      //////    //    AllUsagesCore(matches[0][i], usages);
//                      //////    //usages.UnionWith(matches[0]);
//                      //////    return usages.ToList();
//                      //////}
//                      var firstLinkUsages = new HashSet<ulong>();
//                      AllUsagesCore(sequence[0], firstLinkUsages);
//                      firstLinkUsages.Add(sequence[0]);
//                      //var previousMatchings = firstLinkUsages.ToList(); //new List<ulong>() {
    ↪ sequence[0] }; // or all sequences, containing this element?
//                      //return GetAllPartiallyMatchingSequencesCore(sequence, firstLinkUsages,
    ↪ 1).ToList();
//                      var results = new HashSet<ulong>();
//                      foreach (var match in GetAllPartiallyMatchingSequencesCore(sequence,
    ↪ firstLinkUsages, 1))
//                      {
//                          AllUsagesCore(match, results);
//                      }
//                      return results.ToList();
//                  }
//                  return new List<ulong>();
//              });
//          }
//
//          /// <remarks>
//          /// TODO: Может потробоваться ограничение на уровень глубины рекурсии
//          /// </remarks>
//          [MethodImpl(MethodImplOptions.AggressiveInlining)]
//          public HashSet<ulong> AllUsages(ulong link)
//          {
//              return _sync.DoRead(() =>
//              {
//                  var usages = new HashSet<ulong>();
//                  AllUsagesCore(link, usages);
//                  return usages;
//              });
//          }
//
//          // При сборе всех использований (последовательностей) можно сохранять обратный путь к
    ↪ той связи с которой начинался поиск (STTTSSSTT),
//          // причём достаточно одного бита для хранения перехода влево или вправо
//          [MethodImpl(MethodImplOptions.AggressiveInlining)]
//          private void AllUsagesCore(ulong link, HashSet<ulong> usages)
//          {
//              bool handler(ulong doublet)
//              {
//                  if (usages.Add(doublet))
//                  {
//                      AllUsagesCore(doublet, usages);
//                  }
//                  return true;
//              }
//              Links.Unsync.Each(link, Constants.Any, handler);
//              Links.Unsync.Each(Constants.Any, link, handler);
//          }
```

```
//
//         /// <summary>
//         /// <para>
//         /// Alls the bottom usages using the specified link.
//         /// </para>
//         /// <para></para>
//         /// </summary>
//         /// <param name="link">
//         /// <para>The link.</para>
//         /// <para></para>
//         /// </param>
//         /// <returns>
//         /// <para>A hash set of ulong</para>
//         /// <para></para>
//         /// </returns>
//         [MethodImpl(MethodImplOptions.AggressiveInlining)]
//         public HashSet<ulong> AllBottomUsages(ulong link)
//         {
//             return _sync.DoRead(() =>
//             {
//                 var visits = new HashSet<ulong>();
//                 var usages = new HashSet<ulong>();
//                 AllBottomUsagesCore(link, visits, usages);
//                 return usages;
//             });
//         }
//         [MethodImpl(MethodImplOptions.AggressiveInlining)]
//         private void AllBottomUsagesCore(ulong link, HashSet<ulong> visits, HashSet<ulong>
    ↪ usages)
//         {
//             bool handler(ulong doublet)
//             {
//                 if (visits.Add(doublet))
//                 {
//                     AllBottomUsagesCore(doublet, visits, usages);
//                 }
//                 return true;
//             }
//             if (Links.Unsync.Count(Constants.Any, link) == 0)
//             {
//                 usages.Add(link);
//             }
//             else
//             {
//                 Links.Unsync.Each(link, Constants.Any, handler);
//                 Links.Unsync.Each(Constants.Any, link, handler);
//             }
//         }
//
//         /// <summary>
//         /// <para>
//         /// Calculates the total symbol frequency core using the specified symbol.
//         /// </para>
//         /// <para></para>
//         /// </summary>
//         /// <param name="symbol">
//         /// <para>The symbol.</para>
//         /// <para></para>
//         /// </param>
//         /// <returns>
//         /// <para>The ulong</para>
//         /// <para></para>
//         /// </returns>
//         [MethodImpl(MethodImplOptions.AggressiveInlining)]
//         public ulong CalculateTotalSymbolFrequencyCore(ulong symbol)
//         {
//             if (Options.UseSequenceMarker)
//             {
//                 var counter = new
    ↪ TotalMarkedSequenceSymbolFrequencyOneOffCounter<ulong>(Links, Options.MarkedSequenceMatcher,
    ↪ symbol);
//                 return counter.Count();
//             }
//             else
//             {
//                 var counter = new TotalSequenceSymbolFrequencyOneOffCounter<ulong>(Links,
    ↪ symbol);
```

```
//                 return counter.Count();
//             }
//         }
//         [MethodImpl(MethodImplOptions.AggressiveInlining)]
//         private bool AllUsagesCore1(ulong link, HashSet<ulong> usages, Func<IList<LinkIndex>,
    ↪ LinkIndex> outerHandler)
//         {
//             bool handler(ulong doublet)
//             {
//                 if (usages.Add(doublet))
//                 {
//                     if (outerHandler(new LinkAddress<LinkIndex>(doublet)) !=
    ↪ Constants.Continue)
//                     {
//                         return false;
//                     }
//                     if (!AllUsagesCore1(doublet, usages, outerHandler))
//                     {
//                         return false;
//                     }
//                 }
//                 return true;
//             }
//             return Links.Unsync.Each(link, Constants.Any, handler)
//                 && Links.Unsync.Each(Constants.Any, link, handler);
//         }
//
//         /// <summary>
//         /// <para>
//         /// Calculates the all usages using the specified totals.
//         /// </para>
//         /// <para></para>
//         /// </summary>
//         /// <param name="totals">
//         /// <para>The totals.</para>
//         /// <para></para>
//         /// </param>
//         [MethodImpl(MethodImplOptions.AggressiveInlining)]
//         public void CalculateAllUsages(ulong[] totals)
//         {
//             var calculator = new AllUsagesCalculator(Links, totals);
//             calculator.Calculate();
//         }
//
//         /// <summary>
//         /// <para>
//         /// Calculates the all usages 2 using the specified totals.
//         /// </para>
//         /// <para></para>
//         /// </summary>
//         /// <param name="totals">
//         /// <para>The totals.</para>
//         /// <para></para>
//         /// </param>
//         [MethodImpl(MethodImplOptions.AggressiveInlining)]
//         public void CalculateAllUsages2(ulong[] totals)
//         {
//             var calculator = new AllUsagesCalculator2(Links, totals);
//             calculator.Calculate();
//         }
//         private class AllUsagesCalculator
//         {
//             private readonly SynchronizedLinks<ulong> _links;
//             private readonly ulong[] _totals;
//
//             /// <summary>
//             /// <para>
//             /// Initializes a new <see cref="AllUsagesCalculator"/> instance.
//             /// </para>
//             /// <para></para>
//             /// </summary>
//             /// <param name="links">
//             /// <para>A links.</para>
//             /// <para></para>
//             /// </param>
//             /// <param name="totals">
//             /// <para>A totals.</para>
```

```
//             /// <para></para>
//             /// </param>
//             [MethodImpl(MethodImplOptions.AggressiveInlining)]
//             public AllUsagesCalculator(SynchronizedLinks<ulong> links, ulong[] totals)
//             {
//                 _links = links;
//                 _totals = totals;
//             }
//
//             /// <summary>
//             /// <para>
//             /// Calculates this instance.
//             /// </para>
//             /// <para></para>
//             /// </summary>
//             [MethodImpl(MethodImplOptions.AggressiveInlining)]
//             public void Calculate() => _links.Each(_links.Constants.Any,
    ↪ _links.Constants.Any, CalculateCore);
//             [MethodImpl(MethodImplOptions.AggressiveInlining)]
//             private bool CalculateCore(ulong link)
//             {
//                 if (_totals[link] == 0)
//                 {
//                     var total = 1UL;
//                     _totals[link] = total;
//                     var visitedChildren = new HashSet<ulong>();
//                     bool linkCalculator(ulong child)
//                     {
//                         if (link != child && visitedChildren.Add(child))
//                         {
//                             total += _totals[child] == 0 ? 1 : _totals[child];
//                         }
//                         return true;
//                     }
//                     _links.Unsync.Each(link, _links.Constants.Any, linkCalculator);
//                     _links.Unsync.Each(_links.Constants.Any, link, linkCalculator);
//                     _totals[link] = total;
//                 }
//                 return true;
//             }
//         }
//         private class AllUsagesCalculator2
//         {
//             private readonly SynchronizedLinks<ulong> _links;
//             private readonly ulong[] _totals;
//
//             /// <summary>
//             /// <para>
//             /// Initializes a new <see cref="AllUsagesCalculator2"/> instance.
//             /// </para>
//             /// <para></para>
//             /// </summary>
//             /// <param name="links">
//             /// <para>A links.</para>
//             /// <para></para>
//             /// </param>
//             /// <param name="totals">
//             /// <para>A totals.</para>
//             /// <para></para>
//             /// </param>
//             [MethodImpl(MethodImplOptions.AggressiveInlining)]
//             public AllUsagesCalculator2(SynchronizedLinks<ulong> links, ulong[] totals)
//             {
//                 _links = links;
//                 _totals = totals;
//             }
//
//             /// <summary>
//             /// <para>
//             /// Calculates this instance.
//             /// </para>
//             /// <para></para>
//             /// </summary>
//             [MethodImpl(MethodImplOptions.AggressiveInlining)]
//             public void Calculate() => _links.Each(_links.Constants.Any,
    ↪ _links.Constants.Any, CalculateCore);
//             [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
//             private bool IsElement(ulong link)
//             {
//                 //_linksInSequence.Contains(link) ||
//                 return _links.Unsync.GetTarget(link) == link || _links.Unsync.GetSource(link)
↪  == link;
//             }
//             [MethodImpl(MethodImplOptions.AggressiveInlining)]
//             private bool CalculateCore(ulong link)
//             {
//                 // TODO: Проработать защиту от зацикливания
//                 // Основано на SequenceWalker.WalkLeft
//                 Func<ulong, ulong> getSource = _links.Unsync.GetSource;
//                 Func<ulong, ulong> getTarget = _links.Unsync.GetTarget;
//                 Func<ulong, bool> isElement = IsElement;
//                 void visitLeaf(ulong parent)
//                 {
//                     if (link != parent)
//                     {
//                         _totals[parent]++;
//                     }
//                 }
//                 void visitNode(ulong parent)
//                 {
//                     if (link != parent)
//                     {
//                         _totals[parent]++;
//                     }
//                 }
//                 var stack = new Stack();
//                 var element = link;
//                 if (isElement(element))
//                 {
//                     visitLeaf(element);
//                 }
//                 else
//                 {
//                     while (true)
//                     {
//                         if (isElement(element))
//                         {
//                             if (stack.Count == 0)
//                             {
//                                 break;
//                             }
//                             element = stack.Pop();
//                             var source = getSource(element);
//                             var target = getTarget(element);
//                             // Обработка элемента
//                             if (isElement(target))
//                             {
//                                 visitLeaf(target);
//                             }
//                             if (isElement(source))
//                             {
//                                 visitLeaf(source);
//                             }
//                             element = source;
//                         }
//                         else
//                         {
//                             stack.Push(element);
//                             visitNode(element);
//                             element = getTarget(element);
//                         }
//                     }
//                 }
//                 _totals[link]++;
//                 return true;
//             }
//         }
//         private class AllUsagesCollector
//         {
//             private readonly ILinks<ulong> _links;
//             private readonly HashSet<ulong> _usages;
//
//             /// <summary>
//             /// <para>
//             /// Initializes a new <see cref="AllUsagesCollector"/> instance.
```

```
//             /// </para>
//             /// <para></para>
//             /// </summary>
//             /// <param name="links">
//             /// <para>A links.</para>
//             /// <para></para>
//             /// </param>
//             /// <param name="usages">
//             /// <para>A usages.</para>
//             /// <para></para>
//             /// </param>
//             [MethodImpl(MethodImplOptions.AggressiveInlining)]
//             public AllUsagesCollector(ILinks<ulong> links, HashSet<ulong> usages)
//             {
//                 _links = links;
//                 _usages = usages;
//             }
//
//             /// <summary>
//             /// <para>
//             /// Determines whether this instance collect.
//             /// </para>
//             /// <para></para>
//             /// </summary>
//             /// <param name="link">
//             /// <para>The link.</para>
//             /// <para></para>
//             /// </param>
//             /// <returns>
//             /// <para>The bool</para>
//             /// <para></para>
//             /// </returns>
//             [MethodImpl(MethodImplOptions.AggressiveInlining)]
//             public bool Collect(ulong link)
//             {
//                 if (_usages.Add(link))
//                 {
//                     _links.Each(link, _links.Constants.Any, Collect);
//                     _links.Each(_links.Constants.Any, link, Collect);
//                 }
//                 return true;
//             }
//         }
//         private class AllUsagesCollector1
//         {
//             private readonly ILinks<ulong> _links;
//             private readonly HashSet<ulong> _usages;
//             private readonly ulong _continue;
//
//             /// <summary>
//             /// <para>
//             /// Initializes a new <see cref="AllUsagesCollector1"/> instance.
//             /// </para>
//             /// <para></para>
//             /// </summary>
//             /// <param name="links">
//             /// <para>A links.</para>
//             /// <para></para>
//             /// </param>
//             /// <param name="usages">
//             /// <para>A usages.</para>
//             /// <para></para>
//             /// </param>
//             [MethodImpl(MethodImplOptions.AggressiveInlining)]
//             public AllUsagesCollector1(ILinks<ulong> links, HashSet<ulong> usages)
//             {
//                 _links = links;
//                 _usages = usages;
//                 _continue = _links.Constants.Continue;
//             }
//
//             /// <summary>
//             /// <para>
//             /// Collects the link.
//             /// </para>
//             /// <para></para>
//             /// </summary>
//             /// <param name="link">
```

```
//             /// <para>The link.</para>
//             /// <para></para>
//             /// </param>
//             /// <returns>
//             /// <para>The continue.</para>
//             /// <para></para>
//             /// </returns>
//             [MethodImpl(MethodImplOptions.AggressiveInlining)]
//             public ulong Collect(IList<ulong> link)
//             {
//                 var linkIndex = _links.GetIndex(link);
//                 if (_usages.Add(linkIndex))
//                 {
//                     _links.Each(Collect, _links.Constants.Any, linkIndex);
//                 }
//                 return _continue;
//             }
//         }
//         private class AllUsagesCollector2
//         {
//             private readonly ILinks<ulong> _links;
//             private readonly BitString _usages;
//
//             /// <summary>
//             /// <para>
//             /// Initializes a new <see cref="AllUsagesCollector2"/> instance.
//             /// </para>
//             /// <para></para>
//             /// </summary>
//             /// <param name="links">
//             /// <para>A links.</para>
//             /// <para></para>
//             /// </param>
//             /// <param name="usages">
//             /// <para>A usages.</para>
//             /// <para></para>
//             /// </param>
//             [MethodImpl(MethodImplOptions.AggressiveInlining)]
//             public AllUsagesCollector2(ILinks<ulong> links, BitString usages)
//             {
//                 _links = links;
//                 _usages = usages;
//             }
//
//             /// <summary>
//             /// <para>
//             /// Determines whether this instance collect.
//             /// </para>
//             /// <para></para>
//             /// </summary>
//             /// <param name="link">
//             /// <para>The link.</para>
//             /// <para></para>
//             /// </param>
//             /// <returns>
//             /// <para>The bool</para>
//             /// <para></para>
//             /// </returns>
//             [MethodImpl(MethodImplOptions.AggressiveInlining)]
//             public bool Collect(ulong link)
//             {
//                 if (_usages.Add((long)link))
//                 {
//                     _links.Each(link, _links.Constants.Any, Collect);
//                     _links.Each(_links.Constants.Any, link, Collect);
//                 }
//                 return true;
//             }
//         }
//         private class AllUsagesIntersectingCollector
//         {
//             private readonly SynchronizedLinks<ulong> _links;
//             private readonly HashSet<ulong> _intersectWith;
//             private readonly HashSet<ulong> _usages;
//             private readonly HashSet<ulong> _enter;
//
//             /// <summary>
//             /// <para>
```

```
//             /// Initializes a new <see cref="AllUsagesIntersectingCollector"/> instance.
//             /// </para>
//             /// <para></para>
//             /// </summary>
//             /// <param name="links">
//             /// <para>A links.</para>
//             /// <para></para>
//             /// </param>
//             /// <param name="intersectWith">
//             /// <para>A intersect with.</para>
//             /// <para></para>
//             /// </param>
//             /// <param name="usages">
//             /// <para>A usages.</para>
//             /// <para></para>
//             /// </param>
//             [MethodImpl(MethodImplOptions.AggressiveInlining)]
//             public AllUsagesIntersectingCollector(SynchronizedLinks<ulong> links,
    ↪ HashSet<ulong> intersectWith, HashSet<ulong> usages)
//             {
//                 _links = links;
//                 _intersectWith = intersectWith;
//                 _usages = usages;
//                 _enter = new HashSet<ulong>(); // защита от зацикливания
//             }
//
//             /// <summary>
//             /// <para>
//             /// Determines whether this instance collect.
//             /// </para>
//             /// <para></para>
//             /// </summary>
//             /// <param name="link">
//             /// <para>The link.</para>
//             /// <para></para>
//             /// </param>
//             /// <returns>
//             /// <para>The bool</para>
//             /// <para></para>
//             /// </returns>
//             [MethodImpl(MethodImplOptions.AggressiveInlining)]
//             public bool Collect(ulong link)
//             {
//                 if (_enter.Add(link))
//                 {
//                     if (_intersectWith.Contains(link))
//                     {
//                         _usages.Add(link);
//                     }
//                     _links.Unsync.Each(link, _links.Constants.Any, Collect);
//                     _links.Unsync.Each(_links.Constants.Any, link, Collect);
//                 }
//                 return true;
//             }
//         }
//         [MethodImpl(MethodImplOptions.AggressiveInlining)]
//         private void CloseInnerConnections(Action<IList<LinkIndex>> handler, ulong left,
    ↪ ulong right)
//         {
//             TryStepLeftUp(handler, left, right);
//             TryStepRightUp(handler, right, left);
//         }
//         [MethodImpl(MethodImplOptions.AggressiveInlining)]
//         private void AllCloseConnections(Action<IList<LinkIndex>> handler, ulong left, ulong
    ↪ right)
//         {
//             // Direct
//             if (left == right)
//             {
//                 handler(new LinkAddress<LinkIndex>(left));
//             }
//             var doublet = Links.Unsync.SearchOrDefault(left, right);
//             if (doublet != Constants.Null)
//             {
//                 handler(new LinkAddress<LinkIndex>(doublet));
//             }
//             // Inner
```

```
1724  //             CloseInnerConnections(handler, left, right);
1725  //             // Outer
1726  //             StepLeft(handler, left, right);
1727  //             StepRight(handler, left, right);
1728  //             PartialStepRight(handler, left, right);
1729  //             PartialStepLeft(handler, left, right);
1730  //         }
1731  //         [MethodImpl(MethodImplOptions.AggressiveInlining)]
1732  //         private HashSet<ulong> GetAllPartiallyMatchingSequencesCore(ulong[] sequence,
      ↪  HashSet<ulong> previousMatchings, long startAt)
1733  //         {
1734  //             if (startAt >= sequence.Length) // ?
1735  //             {
1736  //                 return previousMatchings;
1737  //             }
1738  //             var secondLinkUsages = new HashSet<ulong>();
1739  //             AllUsagesCore(sequence[startAt], secondLinkUsages);
1740  //             secondLinkUsages.Add(sequence[startAt]);
1741  //             var matchings = new HashSet<ulong>();
1742  //             var filler = new SetFiller<LinkIndex, LinkIndex>(matchings, Constants.Continue);
1743  //             //for (var i = 0; i < previousMatchings.Count; i++)
1744  //             foreach (var secondLinkUsage in secondLinkUsages)
1745  //             {
1746  //                 foreach (var previousMatching in previousMatchings)
1747  //                 {
1748  //                     //AllCloseConnections(matchings.AddAndReturnVoid, previousMatching,
      ↪  secondLinkUsage);
1749  //                     StepRight(filler.AddFirstAndReturnConstant, previousMatching,
      ↪  secondLinkUsage);
1750  //                     TryStepRightUp(filler.AddFirstAndReturnConstant, secondLinkUsage,
      ↪  previousMatching);
1751  //                     //PartialStepRight(matchings.AddAndReturnVoid, secondLinkUsage,
      ↪  sequence[startAt]); // почему-то эта ошибочная запись приводит к желаемым результатам.
1752  //                     PartialStepRight(filler.AddFirstAndReturnConstant, previousMatching,
      ↪  secondLinkUsage);
1753  //                 }
1754  //             }
1755  //             if (matchings.Count == 0)
1756  //             {
1757  //                 return matchings;
1758  //             }
1759  //             return GetAllPartiallyMatchingSequencesCore(sequence, matchings, startAt + 1); //
      ↪  ??
1760  //         }
1761  //         [MethodImpl(MethodImplOptions.AggressiveInlining)]
1762  //         private static void EnsureEachLinkIsAnyOrZeroOrManyOrExists(SynchronizedLinks<ulong>
      ↪  links, params ulong[] sequence)
1763  //         {
1764  //             if (sequence == null)
1765  //             {
1766  //                 return;
1767  //             }
1768  //             for (var i = 0; i < sequence.Length; i++)
1769  //             {
1770  //                 if (sequence[i] != links.Constants.Any && sequence[i] != ZeroOrMany &&
      ↪  !links.Exists(sequence[i]))
1771  //                 {
1772  //                     throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
      ↪  $"patternSequence[{i}]");
1773  //                 }
1774  //             }
1775  //         }
1776  //
1777  //         // Pattern Matching -> Key To Triggers
1778  //         /// <summary>
1779  //         /// <para>
1780  //         /// Matches the pattern using the specified pattern sequence.
1781  //         /// </para>
1782  //         /// <para></para>
1783  //         /// </summary>
1784  //         /// <param name="patternSequence">
1785  //         /// <para>The pattern sequence.</para>
1786  //         /// <para></para>
1787  //         /// </param>
1788  //         /// <returns>
1789  //         /// <para>A hash set of ulong</para>
1790  //         /// <para></para>
```

```
//         /// </returns>
//         [MethodImpl(MethodImplOptions.AggressiveInlining)]
//         public HashSet<ulong> MatchPattern(params ulong[] patternSequence)
//         {
//             return _sync.DoRead(() =>
//             {
//                 patternSequence = Simplify(patternSequence);
//                 if (patternSequence.Length > 0)
//                 {
//                     EnsureEachLinkIsAnyOrZeroOrManyOrExists(Links, patternSequence);
//                     var uniqueSequenceElements = new HashSet<ulong>();
//                     for (var i = 0; i < patternSequence.Length; i++)
//                     {
//                         if (patternSequence[i] != Constants.Any && patternSequence[i] !=
    ↪ ZeroOrMany)
//                         {
//                             uniqueSequenceElements.Add(patternSequence[i]);
//                         }
//                     }
//                     var results = new HashSet<ulong>();
//                     foreach (var uniqueSequenceElement in uniqueSequenceElements)
//                     {
//                         AllUsagesCore(uniqueSequenceElement, results);
//                     }
//                     var filteredResults = new HashSet<ulong>();
//                     var matcher = new PatternMatcher(this, patternSequence, filteredResults);
//                     matcher.AddAllPatternMatchedToResults(results);
//                     return filteredResults;
//                 }
//                 return new HashSet<ulong>();
//             });
//         }
//
//         // Найти все возможные связи между указанным списком связей.
//         // Находит связи между всеми указанными связями в любом порядке.
//         // TODO: решить что делать с повторами (когда одни и те же элементы встречаются
    ↪ несколько раз в последовательности)
//         /// <summary>
//         /// <para>
//         /// Gets the all connections using the specified links to connect.
//         /// </para>
//         /// <para></para>
//         /// </summary>
//         /// <param name="linksToConnect">
//         /// <para>The links to connect.</para>
//         /// <para></para>
//         /// </param>
//         /// <returns>
//         /// <para>A hash set of ulong</para>
//         /// <para></para>
//         /// </returns>
//         [MethodImpl(MethodImplOptions.AggressiveInlining)]
//         public HashSet<ulong> GetAllConnections(params ulong[] linksToConnect)
//         {
//             return _sync.DoRead(() =>
//             {
//                 var results = new HashSet<ulong>();
//                 if (linksToConnect.Length > 0)
//                 {
//                     Links.EnsureLinkExists(linksToConnect);
//                     AllUsagesCore(linksToConnect[0], results);
//                     for (var i = 1; i < linksToConnect.Length; i++)
//                     {
//                         var next = new HashSet<ulong>();
//                         AllUsagesCore(linksToConnect[i], next);
//                         results.IntersectWith(next);
//                     }
//                 }
//                 return results;
//             });
//         }
//
//         /// <summary>
//         /// <para>
//         /// Gets the all connections 1 using the specified links to connect.
//         /// </para>
//         /// <para></para>
```

```
//          /// </summary>
//          /// <param name="linksToConnect">
//          /// <para>The links to connect.</para>
//          /// <para></para>
//          /// </param>
//          /// <returns>
//          /// <para>A hash set of ulong</para>
//          /// <para></para>
//          /// </returns>
//          [MethodImpl(MethodImplOptions.AggressiveInlining)]
//          public HashSet<ulong> GetAllConnections1(params ulong[] linksToConnect)
//          {
//              return _sync.DoRead(() =>
//              {
//                  var results = new HashSet<ulong>();
//                  if (linksToConnect.Length > 0)
//                  {
//                      Links.EnsureLinkExists(linksToConnect);
//                      var collector1 = new AllUsagesCollector(Links.Unsync, results);
//                      collector1.Collect(linksToConnect[0]);
//                      var next = new HashSet<ulong>();
//                      for (var i = 1; i < linksToConnect.Length; i++)
//                      {
//                          var collector = new AllUsagesCollector(Links.Unsync, next);
//                          collector.Collect(linksToConnect[i]);
//                          results.IntersectWith(next);
//                          next.Clear();
//                      }
//                  }
//                  return results;
//              });
//          }
//
//          /// <summary>
//          /// <para>
//          /// Gets the all connections 2 using the specified links to connect.
//          /// </para>
//          /// <para></para>
//          /// </summary>
//          /// <param name="linksToConnect">
//          /// <para>The links to connect.</para>
//          /// <para></para>
//          /// </param>
//          /// <returns>
//          /// <para>A hash set of ulong</para>
//          /// <para></para>
//          /// </returns>
//          [MethodImpl(MethodImplOptions.AggressiveInlining)]
//          public HashSet<ulong> GetAllConnections2(params ulong[] linksToConnect)
//          {
//              return _sync.DoRead(() =>
//              {
//                  var results = new HashSet<ulong>();
//                  if (linksToConnect.Length > 0)
//                  {
//                      Links.EnsureLinkExists(linksToConnect);
//                      var collector1 = new AllUsagesCollector(Links, results);
//                      collector1.Collect(linksToConnect[0]);
//                      //AllUsagesCore(linksToConnect[0], results);
//                      for (var i = 1; i < linksToConnect.Length; i++)
//                      {
//                          var next = new HashSet<ulong>();
//                          var collector = new AllUsagesIntersectingCollector(Links, results,
    ↪  next);
//                          collector.Collect(linksToConnect[i]);
//                          //AllUsagesCore(linksToConnect[i], next);
//                          //results.IntersectWith(next);
//                          results = next;
//                      }
//                  }
//                  return results;
//              });
//          }
//
//          /// <summary>
//          /// <para>
//          /// Gets the all connections 3 using the specified links to connect.
```

```
//         /// </para>
//         /// <para></para>
//         /// </summary>
//         /// <param name="linksToConnect">
//         /// <para>The links to connect.</para>
//         /// <para></para>
//         /// </param>
//         /// <returns>
//         /// <para>A list of ulong</para>
//         /// <para></para>
//         /// </returns>
//         [MethodImpl(MethodImplOptions.AggressiveInlining)]
//         public List<ulong> GetAllConnections3(params ulong[] linksToConnect)
//         {
//             return _sync.DoRead(() =>
//             {
//                 var results = new BitString((long)Links.Unsync.Count() + 1); // new BitArray((int)_links.Total + 1);
//                 if (linksToConnect.Length > 0)
//                 {
//                     Links.EnsureLinkExists(linksToConnect);
//                     var collector1 = new AllUsagesCollector2(Links.Unsync, results);
//                     collector1.Collect(linksToConnect[0]);
//                     for (var i = 1; i < linksToConnect.Length; i++)
//                     {
//                         var next = new BitString((long)Links.Unsync.Count() + 1); //new BitArray((int)_links.Total + 1);
//                         var collector = new AllUsagesCollector2(Links.Unsync, next);
//                         collector.Collect(linksToConnect[i]);
//                         results = results.And(next);
//                     }
//                 }
//                 return results.GetSetUInt64Indices();
//             });
//         }
//         [MethodImpl(MethodImplOptions.AggressiveInlining)]
//         private static ulong[] Simplify(ulong[] sequence)
//         {
//             // Считаем новый размер последовательности
//             long newLength = 0;
//             var zeroOrManyStepped = false;
//             for (var i = 0; i < sequence.Length; i++)
//             {
//                 if (sequence[i] == ZeroOrMany)
//                 {
//                     if (zeroOrManyStepped)
//                     {
//                         continue;
//                     }
//                     zeroOrManyStepped = true;
//                 }
//                 else
//                 {
//                     //if (zeroOrManyStepped) Is it efficient?
//                     zeroOrManyStepped = false;
//                 }
//                 newLength++;
//             }
//             // Строим новую последовательность
//             zeroOrManyStepped = false;
//             var newSequence = new ulong[newLength];
//             long j = 0;
//             for (var i = 0; i < sequence.Length; i++)
//             {
//                 //var current = zeroOrManyStepped;
//                 //zeroOrManyStepped = patternSequence[i] == zeroOrMany;
//                 //if (current && zeroOrManyStepped)
//                 //    continue;
//                 //var newZeroOrManyStepped = patternSequence[i] == zeroOrMany;
//                 //if (zeroOrManyStepped && newZeroOrManyStepped)
//                 //    continue;
//                 //zeroOrManyStepped = newZeroOrManyStepped;
//                 if (sequence[i] == ZeroOrMany)
//                 {
//                     if (zeroOrManyStepped)
//                     {
//                         continue;
```

```
//                      }
//                      zeroOrManyStepped = true;
//                  }
//                  else
//                  {
//                      //if (zeroOrManyStepped) Is it efficient?
//                      zeroOrManyStepped = false;
//                  }
//                  newSequence[j++] = sequence[i];
//              }
//              return newSequence;
//          }
//
//          /// <summary>
//          /// <para>
//          /// Tests the simplify.
//          /// </para>
//          /// <para></para>
//          /// </summary>
//          [MethodImpl(MethodImplOptions.AggressiveInlining)]
//          public static void TestSimplify()
//          {
//              var sequence = new ulong[] { ZeroOrMany, ZeroOrMany, 2, 3, 4, ZeroOrMany,
    ↪  ZeroOrMany, ZeroOrMany, 4, ZeroOrMany, ZeroOrMany, ZeroOrMany };
//              var simplifiedSequence = Simplify(sequence);
//          }
//
//          /// <summary>
//          /// <para>
//          /// Gets the similar sequences.
//          /// </para>
//          /// <para></para>
//          /// </summary>
//          /// <returns>
//          /// <para>A list of ulong</para>
//          /// <para></para>
//          /// </returns>
//          [MethodImpl(MethodImplOptions.AggressiveInlining)]
//          public List<ulong> GetSimilarSequences() => new List<ulong>();
//
//          /// <summary>
//          /// <para>
//          /// Predictions this instance.
//          /// </para>
//          /// <para></para>
//          /// </summary>
//          [MethodImpl(MethodImplOptions.AggressiveInlining)]
//          public void Prediction()
//          {
//              //_links
//              //sequences
//          }
//
//          #region From Triplets
//
//          //public static void DeleteSequence(Link sequence)
//          //{
//          //}
//
//          /// <summary>
//          /// <para>
//          /// Collects the matching sequences using the specified links.
//          /// </para>
//          /// <para></para>
//          /// </summary>
//          /// <param name="links">
//          /// <para>The links.</para>
//          /// <para></para>
//          /// </param>
//          /// <exception cref="InvalidOperationException">
//          /// <para>Подпоследовательности с одним элементом не поддерживаются.</para>
//          /// <para></para>
//          /// </exception>
//          /// <returns>
//          /// <para>The results.</para>
//          /// <para></para>
//          /// </returns>
```

```
//         [MethodImpl(MethodImplOptions.AggressiveInlining)]
//         public List<ulong> CollectMatchingSequences(ulong[] links)
//         {
//             if (links.Length == 1)
//             {
//                 throw new InvalidOperationException("Подпоследовательности с одним элементом
    ↪ не поддерживаются.");
//             }
//             var leftBound = 0;
//             var rightBound = links.Length - 1;
//             var left = links[leftBound++];
//             var right = links[rightBound--];
//             var results = new List<ulong>();
//             CollectMatchingSequences(left, leftBound, links, right, rightBound, ref results);
//             return results;
//         }
//         [MethodImpl(MethodImplOptions.AggressiveInlining)]
//         private void CollectMatchingSequences(ulong leftLink, int leftBound, ulong[]
    ↪ middleLinks, ulong rightLink, int rightBound, ref List<ulong> results)
//         {
//             var leftLinkTotalReferers = Links.Unsync.Count(leftLink);
//             var rightLinkTotalReferers = Links.Unsync.Count(rightLink);
//             if (leftLinkTotalReferers <= rightLinkTotalReferers)
//             {
//                 var nextLeftLink = middleLinks[leftBound];
//                 var elements = GetRightElements(leftLink, nextLeftLink);
//                 if (leftBound <= rightBound)
//                 {
//                     for (var i = elements.Length - 1; i >= 0; i--)
//                     {
//                         var element = elements[i];
//                         if (element != 0)
//                         {
//                             CollectMatchingSequences(element, leftBound + 1, middleLinks,
    ↪ rightLink, rightBound, ref results);
//                         }
//                     }
//                 }
//                 else
//                 {
//                     for (var i = elements.Length - 1; i >= 0; i--)
//                     {
//                         var element = elements[i];
//                         if (element != 0)
//                         {
//                             results.Add(element);
//                         }
//                     }
//                 }
//             }
//             else
//             {
//                 var nextRightLink = middleLinks[rightBound];
//                 var elements = GetLeftElements(rightLink, nextRightLink);
//                 if (leftBound <= rightBound)
//                 {
//                     for (var i = elements.Length - 1; i >= 0; i--)
//                     {
//                         var element = elements[i];
//                         if (element != 0)
//                         {
//                             CollectMatchingSequences(leftLink, leftBound, middleLinks,
    ↪ elements[i], rightBound - 1, ref results);
//                         }
//                     }
//                 }
//                 else
//                 {
//                     for (var i = elements.Length - 1; i >= 0; i--)
//                     {
//                         var element = elements[i];
//                         if (element != 0)
//                         {
//                             results.Add(element);
//                         }
//                     }
//                 }
```

```
//             }
//         }
//
//         /// <summary>
//         /// <para>
//         /// Gets the right elements using the specified start link.
//         /// </para>
//         /// <para></para>
//         /// </summary>
//         /// <param name="startLink">
//         /// <para>The start link.</para>
//         /// <para></para>
//         /// </param>
//         /// <param name="rightLink">
//         /// <para>The right link.</para>
//         /// <para></para>
//         /// </param>
//         /// <returns>
//         /// <para>The result.</para>
//         /// <para></para>
//         /// </returns>
//         [MethodImpl(MethodImplOptions.AggressiveInlining)]
//         public ulong[] GetRightElements(ulong startLink, ulong rightLink)
//         {
//             var result = new ulong[5];
//             TryStepRight(startLink, rightLink, result, 0);
//             Links.Each(Constants.Any, startLink, couple =>
//             {
//                 if (couple != startLink)
//                 {
//                     if (TryStepRight(couple, rightLink, result, 2))
//                     {
//                         return false;
//                     }
//                 }
//                 return true;
//             });
//             if (Links.GetTarget(Links.GetTarget(startLink)) == rightLink)
//             {
//                 result[4] = startLink;
//             }
//             return result;
//         }
//
//         /// <summary>
//         /// <para>
//         /// Determines whether this instance try step right.
//         /// </para>
//         /// <para></para>
//         /// </summary>
//         /// <param name="startLink">
//         /// <para>The start link.</para>
//         /// <para></para>
//         /// </param>
//         /// <param name="rightLink">
//         /// <para>The right link.</para>
//         /// <para></para>
//         /// </param>
//         /// <param name="result">
//         /// <para>The result.</para>
//         /// <para></para>
//         /// </param>
//         /// <param name="offset">
//         /// <para>The offset.</para>
//         /// <para></para>
//         /// </param>
//         /// <returns>
//         /// <para>The bool</para>
//         /// <para></para>
//         /// </returns>
//         [MethodImpl(MethodImplOptions.AggressiveInlining)]
//         public bool TryStepRight(ulong startLink, ulong rightLink, ulong[] result, int offset)
//         {
//             var added = 0;
//             Links.Each(startLink, Constants.Any, couple =>
//             {
//                 if (couple != startLink)
//                 {
```

```
//                      var coupleTarget = Links.GetTarget(couple);
//                      if (coupleTarget == rightLink)
//                      {
//                          result[offset] = couple;
//                          if (++added == 2)
//                          {
//                              return false;
//                          }
//                      }
//                      else if (Links.GetSource(coupleTarget) == rightLink) //
↪  coupleTarget.Linker == Net.And &&
//                      {
//                          result[offset + 1] = couple;
//                          if (++added == 2)
//                          {
//                              return false;
//                          }
//                      }
//                  }
//                  return true;
//              });
//              return added > 0;
//          }
//
//          /// <summary>
//          /// <para>
//          /// Gets the left elements using the specified start link.
//          /// </para>
//          /// <para></para>
//          /// </summary>
//          /// <param name="startLink">
//          /// <para>The start link.</para>
//          /// <para></para>
//          /// </param>
//          /// <param name="leftLink">
//          /// <para>The left link.</para>
//          /// <para></para>
//          /// </param>
//          /// <returns>
//          /// <para>The result.</para>
//          /// <para></para>
//          /// </returns>
//          [MethodImpl(MethodImplOptions.AggressiveInlining)]
//          public ulong[] GetLeftElements(ulong startLink, ulong leftLink)
//          {
//              var result = new ulong[5];
//              TryStepLeft(startLink, leftLink, result, 0);
//              Links.Each(startLink, Constants.Any, couple =>
//              {
//                  if (couple != startLink)
//                  {
//                      if (TryStepLeft(couple, leftLink, result, 2))
//                      {
//                          return false;
//                      }
//                  }
//                  return true;
//              });
//              if (Links.GetSource(Links.GetSource(leftLink)) == startLink)
//              {
//                  result[4] = leftLink;
//              }
//              return result;
//          }
//
//          /// <summary>
//          /// <para>
//          /// Determines whether this instance try step left.
//          /// </para>
//          /// <para></para>
//          /// </summary>
//          /// <param name="startLink">
//          /// <para>The start link.</para>
//          /// <para></para>
//          /// </param>
//          /// <param name="leftLink">
//          /// <para>The left link.</para>
//          /// <para></para>
```

```
//         /// </param>
//         /// <param name="result">
//         /// <para>The result.</para>
//         /// <para></para>
//         /// </param>
//         /// <param name="offset">
//         /// <para>The offset.</para>
//         /// <para></para>
//         /// </param>
//         /// <returns>
//         /// <para>The bool</para>
//         /// <para></para>
//         /// </returns>
//         [MethodImpl(MethodImplOptions.AggressiveInlining)]
//         public bool TryStepLeft(ulong startLink, ulong leftLink, ulong[] result, int offset)
//         {
//             var added = 0;
//             Links.Each(Constants.Any, startLink, couple =>
//             {
//                 if (couple != startLink)
//                 {
//                     var coupleSource = Links.GetSource(couple);
//                     if (coupleSource == leftLink)
//                     {
//                         result[offset] = couple;
//                         if (++added == 2)
//                         {
//                             return false;
//                         }
//                     }
//                     else if (Links.GetTarget(coupleSource) == leftLink) //
    ↪  coupleSource.Linker == Net.And &&
//                     {
//                         result[offset + 1] = couple;
//                         if (++added == 2)
//                         {
//                             return false;
//                         }
//                     }
//                 }
//                 return true;
//             });
//             return added > 0;
//         }
//
//         #endregion
//
//         #region Walkers
//
//         /// <summary>
//         /// <para>
//         /// Represents the pattern matcher.
//         /// </para>
//         /// <para></para>
//         /// </summary>
//         /// <seealso cref="RightSequenceWalker{ulong}"/>
//         public class PatternMatcher : RightSequenceWalker<ulong>
//         {
//             private readonly Sequences _sequences;
//             private readonly ulong[] _patternSequence;
//             private readonly HashSet<LinkIndex> _linksInSequence;
//             private readonly HashSet<LinkIndex> _results;
//
//             #region Pattern Match
//
//             /// <summary>
//             /// <para>
//             /// The pattern block type enum.
//             /// </para>
//             /// <para></para>
//             /// </summary>
//             enum PatternBlockType
//             {
//                 /// <summary>
//                 /// <para>
//                 /// The undefined pattern block type.
//                 /// </para>
//                 /// <para></para>
```

```
//                  /// </summary>
//                  Undefined,
//                  /// <summary>
//                  /// <para>
//                  /// The gap pattern block type.
//                  /// </para>
//                  /// <para></para>
//                  /// </summary>
//                  Gap,
//                  /// <summary>
//                  /// <para>
//                  /// The elements pattern block type.
//                  /// </para>
//                  /// <para></para>
//                  /// </summary>
//                  Elements
//              }
//
//              /// <summary>
//              /// <para>
//              /// The pattern block.
//              /// </para>
//              /// <para></para>
//              /// </summary>
//              struct PatternBlock
//              {
//                  /// <summary>
//                  /// <para>
//                  /// The type.
//                  /// </para>
//                  /// <para></para>
//                  /// </summary>
//                  public PatternBlockType Type;
//                  /// <summary>
//                  /// <para>
//                  /// The start.
//                  /// </para>
//                  /// <para></para>
//                  /// </summary>
//                  public long Start;
//                  /// <summary>
//                  /// <para>
//                  /// The stop.
//                  /// </para>
//                  /// <para></para>
//                  /// </summary>
//                  public long Stop;
//              }
//              private readonly List<PatternBlock> _pattern;
//              private int _patternPosition;
//              private long _sequencePosition;
//
//              #endregion
//
//              /// <summary>
//              /// <para>
//              /// Initializes a new <see cref="PatternMatcher"/> instance.
//              /// </para>
//              /// <para></para>
//              /// </summary>
//              /// <param name="sequences">
//              /// <para>A sequences.</para>
//              /// <para></para>
//              /// </param>
//              /// <param name="patternSequence">
//              /// <para>A pattern sequence.</para>
//              /// <para></para>
//              /// </param>
//              /// <param name="results">
//              /// <para>A results.</para>
//              /// <para></para>
//              /// </param>
//              [MethodImpl(MethodImplOptions.AggressiveInlining)]
//              public PatternMatcher(Sequences sequences, LinkIndex[] patternSequence,
    ↪  HashSet<LinkIndex> results)
//                  : base(sequences.Links.Unsync, new DefaultStack<ulong>())
//              {
```

```
//                 _sequences = sequences;
//                 _patternSequence = patternSequence;
//                 _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
↪ _sequences.Constants.Any && x != ZeroOrMany));
//                 _results = results;
//                 _pattern = CreateDetailedPattern();
//             }
//
//             /// <summary>
//             /// <para>
//             /// Determines whether this instance is element.
//             /// </para>
//             /// <para></para>
//             /// </summary>
//             /// <param name="link">
//             /// <para>The link.</para>
//             /// <para></para>
//             /// </param>
//             /// <returns>
//             /// <para>The bool</para>
//             /// <para></para>
//             /// </returns>
//             [MethodImpl(MethodImplOptions.AggressiveInlining)]
//             protected override bool IsElement(ulong link) => _linksInSequence.Contains(link)
↪ || base.IsElement(link);
//
//             /// <summary>
//             /// <para>
//             /// Determines whether this instance pattern match.
//             /// </para>
//             /// <para></para>
//             /// </summary>
//             /// <param name="sequenceToMatch">
//             /// <para>The sequence to match.</para>
//             /// <para></para>
//             /// </param>
//             /// <returns>
//             /// <para>The bool</para>
//             /// <para></para>
//             /// </returns>
//             [MethodImpl(MethodImplOptions.AggressiveInlining)]
//             public bool PatternMatch(LinkIndex sequenceToMatch)
//             {
//                 _patternPosition = 0;
//                 _sequencePosition = 0;
//                 foreach (var part in Walk(sequenceToMatch))
//                 {
//                     if (!PatternMatchCore(part))
//                     {
//                         break;
//                     }
//                 }
//                 return _patternPosition == _pattern.Count || (_patternPosition ==
↪ _pattern.Count - 1 && _pattern[_patternPosition].Start == 0);
//             }
//             [MethodImpl(MethodImplOptions.AggressiveInlining)]
//             private List<PatternBlock> CreateDetailedPattern()
//             {
//                 var pattern = new List<PatternBlock>();
//                 var patternBlock = new PatternBlock();
//                 for (var i = 0; i < _patternSequence.Length; i++)
//                 {
//                     if (patternBlock.Type == PatternBlockType.Undefined)
//                     {
//                         if (_patternSequence[i] == _sequences.Constants.Any)
//                         {
//                             patternBlock.Type = PatternBlockType.Gap;
//                             patternBlock.Start = 1;
//                             patternBlock.Stop = 1;
//                         }
//                         else if (_patternSequence[i] == ZeroOrMany)
//                         {
//                             patternBlock.Type = PatternBlockType.Gap;
//                             patternBlock.Start = 0;
//                             patternBlock.Stop = long.MaxValue;
//                         }
//                         else
```

```
//                          {
//                              patternBlock.Type = PatternBlockType.Elements;
//                              patternBlock.Start = i;
//                              patternBlock.Stop = i;
//                          }
//                      }
//                      else if (patternBlock.Type == PatternBlockType.Elements)
//                      {
//                          if (_patternSequence[i] == _sequences.Constants.Any)
//                          {
//                              pattern.Add(patternBlock);
//                              patternBlock = new PatternBlock
//                              {
//                                  Type = PatternBlockType.Gap,
//                                  Start = 1,
//                                  Stop = 1
//                              };
//                          }
//                          else if (_patternSequence[i] == ZeroOrMany)
//                          {
//                              pattern.Add(patternBlock);
//                              patternBlock = new PatternBlock
//                              {
//                                  Type = PatternBlockType.Gap,
//                                  Start = 0,
//                                  Stop = long.MaxValue
//                              };
//                          }
//                          else
//                          {
//                              patternBlock.Stop = i;
//                          }
//                      }
//                      else // patternBlock.Type == PatternBlockType.Gap
//                      {
//                          if (_patternSequence[i] == _sequences.Constants.Any)
//                          {
//                              patternBlock.Start++;
//                              if (patternBlock.Stop < patternBlock.Start)
//                              {
//                                  patternBlock.Stop = patternBlock.Start;
//                              }
//                          }
//                          else if (_patternSequence[i] == ZeroOrMany)
//                          {
//                              patternBlock.Stop = long.MaxValue;
//                          }
//                          else
//                          {
//                              pattern.Add(patternBlock);
//                              patternBlock = new PatternBlock
//                              {
//                                  Type = PatternBlockType.Elements,
//                                  Start = i,
//                                  Stop = i
//                              };
//                          }
//                      }
//                  }
//                  if (patternBlock.Type != PatternBlockType.Undefined)
//                  {
//                      pattern.Add(patternBlock);
//                  }
//                  return pattern;
//              }
//
//              // match: search for regexp anywhere in text
//              //int match(char* regexp, char* text)
//              //{
//              //    do
//              //    {
//              //    } while (*text++ != '\0');
//              //    return 0;
//              //}
//
//              // matchhere: search for regexp at beginning of text
//              //int matchhere(char* regexp, char* text)
//              //{
```

```
//              //    if (regexp[0] == '\0')
//              //        return 1;
//              //    if (regexp[1] == '*')
//              //        return matchstar(regexp[0], regexp + 2, text);
//              //    if (regexp[0] == '$' && regexp[1] == '\0')
//              //        return *text == '\0';
//              //    if (*text != '\0' && (regexp[0] == '.' || regexp[0] == *text))
//              //        return matchhere(regexp + 1, text + 1);
//              //    return 0;
//              //}
//              //
//              // matchstar: search for c*regexp at beginning of text
//              //int matchstar(int c, char* regexp, char* text)
//              //{
//              //    do
//              //    {    /* a * matches zero or more instances */
//              //        if (matchhere(regexp, text))
//              //            return 1;
//              //    } while (*text != '\0' && (*text++ == c || c == '.'));
//              //    return 0;
//              //}
//              //
//              //private void GetNextPatternElement(out LinkIndex element, out long mininumGap,
    ↪  out long maximumGap)
//              //{
//              //    mininumGap = 0;
//              //    maximumGap = 0;
//              //    element = 0;
//              //    for (; _patternPosition < _patternSequence.Length; _patternPosition++)
//              //    {
//              //        if (_patternSequence[_patternPosition] == Doublets.Links.Null)
//              //            mininumGap++;
//              //        else if (_patternSequence[_patternPosition] == ZeroOrMany)
//              //            maximumGap = long.MaxValue;
//              //        else
//              //            break;
//              //    }
//              //
//              //    if (maximumGap < mininumGap)
//              //        maximumGap = mininumGap;
//              //}
//              [MethodImpl(MethodImplOptions.AggressiveInlining)]
//              private bool PatternMatchCore(LinkIndex element)
//              {
//                  if (_patternPosition >= _pattern.Count)
//                  {
//                      _patternPosition = -2;
//                      return false;
//                  }
//                  var currentPatternBlock = _pattern[_patternPosition];
//                  if (currentPatternBlock.Type == PatternBlockType.Gap)
//                  {
//                      //var currentMatchingBlockLength = (_sequencePosition -
    ↪  _lastMatchedBlockPosition);
//                      if (_sequencePosition < currentPatternBlock.Start)
//                      {
//                          _sequencePosition++;
//                          return true; // Двигаемся дальше
//                      }
//                      // Это последний блок
//                      if (_pattern.Count == _patternPosition + 1)
//                      {
//                          _patternPosition++;
//                          _sequencePosition = 0;
//                          return false; // Полное соответствие
//                      }
//                      else
//                      {
//                          if (_sequencePosition > currentPatternBlock.Stop)
//                          {
//                              return false; // Соответствие невозможно
//                          }
//                          var nextPatternBlock = _pattern[_patternPosition + 1];
//                          if (_patternSequence[nextPatternBlock.Start] == element)
//                          {
//                              if (nextPatternBlock.Start < nextPatternBlock.Stop)
//                              {
```

```
//                                 _patternPosition++;
//                                 _sequencePosition = 1;
//                             }
//                             else
//                             {
//                                 _patternPosition += 2;
//                                 _sequencePosition = 0;
//                             }
//                         }
//                     }
//                 }
//                 else // currentPatternBlock.Type == PatternBlockType.Elements
//                 {
//                     var patternElementPosition = currentPatternBlock.Start + _sequencePosition;
//                     if (_patternSequence[patternElementPosition] != element)
//                     {
//                         return false; // Соответствие невозможно
//                     }
//                     if (patternElementPosition == currentPatternBlock.Stop)
//                     {
//                         _patternPosition++;
//                         _sequencePosition = 0;
//                     }
//                     else
//                     {
//                         _sequencePosition++;
//                     }
//                 }
//                 return true;
//                 //if (_patternSequence[_patternPosition] != element)
//                 //    return false;
//                 //else
//                 //{
//                 //    _sequencePosition++;
//                 //    _patternPosition++;
//                 //    return true;
//                 //}
//                 ////////
//                 //if (_filterPosition == _patternSequence.Length)
//                 //{
//                 //    _filterPosition = -2; // Длиннее чем нужно
//                 //    return false;
//                 //}
//                 //if (element != _patternSequence[_filterPosition])
//                 //{
//                 //    _filterPosition = -1;
//                 //    return false; // Начинается иначе
//                 //}
//                 //_filterPosition++;
//                 //if (_filterPosition == (_patternSequence.Length - 1))
//                 //    return false;
//                 //if (_filterPosition >= 0)
//                 //{
//                 //    if (element == _patternSequence[_filterPosition + 1])
//                 //        _filterPosition++;
//                 //    else
//                 //        return false;
//                 //}
//                 //if (_filterPosition < 0)
//                 //{
//                 //    if (element == _patternSequence[0])
//                 //        _filterPosition = 0;
//                 //}
//             }
//
//             /// <summary>
//             /// <para>
//             /// Adds the all pattern matched to results using the specified sequences to match.
//             /// </para>
//             /// <para></para>
//             /// </summary>
//             /// <param name="sequencesToMatch">
//             /// <para>The sequences to match.</para>
//             /// <para></para>
//             /// </param>
//             [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
//             public void AddAllPatternMatchedToResults(IEnumerable<ulong> sequencesToMatch)
//             {
//                 foreach (var sequenceToMatch in sequencesToMatch)
//                 {
//                     if (PatternMatch(sequenceToMatch))
//                     {
//                         _results.Add(sequenceToMatch);
//                     }
//                 }
//             }
//
//             #endregion
//     }
// }
```

1.46 ./csharp/Platform.Data.Doublets.Sequences/Sequences.cs

```
// using System;
// using System.Collections.Generic;
// using System.Linq;
// using System.Runtime.CompilerServices;
// using Platform.Collections;
// using Platform.Collections.Lists;
// using Platform.Collections.Stacks;
// using Platform.Threading.Synchronization;
// using Platform.Data.Doublets.Sequences.Walkers;
// using Platform.Delegates;
// using LinkIndex = System.UInt64;
//
// #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
//
// namespace Platform.Data.Doublets.Sequences
// {
//     /// <summary>
//     /// Представляет коллекцию последовательностей связей.
//     /// </summary>
//     /// <remarks>
//     /// Обязательно реализовать атомарность каждого публичного метода.
//     ///
//     /// TODO:
//     ///
//     /// !!! Повышение вероятности повторного использования групп (подпоследовательностей),
//     /// через естественную группировку по unicode типам, все whitespace вместе, все символы вместе, все числа вместе и т.п.
//     /// + использовать ровно сбалансированный вариант, чтобы уменьшать вложенность (глубину графа)
//     ///
//     /// x*y - найти все связи между, в последовательностях любой формы, если не стоит ограничитель на то, что является последовательностью, а что нет,
//     /// то находятся любые структуры связей, которые содержат эти элементы именно в таком порядке.
//     ///
//     /// Рост последовательности слева и справа.
//     /// Поиск со звёздочкой.
//     /// URL, PURL - реестр используемых во вне ссылок на ресурсы,
//     /// так же проблема может быть решена при реализации дистанционных триггеров.
//     /// Нужны ли уникальные указатели вообще?
//     /// Что если обращение к информации будет происходить через содержимое всегда?
//     ///
//     /// Писать тесты.
//     ///
//     ///
//     /// Можно убрать зависимость от конкретной реализации Links,
//     /// на зависимость от абстрактного элемента, который может быть представлен несколькими способами.
//     ///
//     /// Можно ли как-то сделать один общий интерфейс
//     ///
//     ///
//     /// Блокчейн и/или гит для распределённой записи транзакций.
//     ///
//     /// </remarks>
//     public partial class Sequences : ILinks<LinkIndex> // IList<string>, IList<LinkIndex[]> (после завершения реализации Sequences)
//     {
//         /// <summary>Возвращает значение LinkIndex, обозначающее любое количество связей.</summary>
//         public const LinkIndex ZeroOrMany = LinkIndex.MaxValue;
```

```
//
//         /// <summary>
//         /// <para>
//         /// Gets the options value.
//         /// </para>
//         /// <para></para>
//         /// </summary>
//         public SequencesOptions<LinkIndex> Options { get; }
//         /// <summary>
//         /// <para>
//         /// Gets the links value.
//         /// </para>
//         /// <para></para>
//         /// </summary>
//         public SynchronizedLinks<LinkIndex> Links { get; }
//         private readonly ISynchronization _sync;
//
//         /// <summary>
//         /// <para>
//         /// Gets the constants value.
//         /// </para>
//         /// <para></para>
//         /// </summary>
//         public LinksConstants<LinkIndex> Constants { get; }
//
//         /// <summary>
//         /// <para>
//         /// Initializes a new <see cref="Sequences"/> instance.
//         /// </para>
//         /// <para></para>
//         /// </summary>
//         /// <param name="links">
//         /// <para>A links.</para>
//         /// <para></para>
//         /// </param>
//         /// <param name="options">
//         /// <para>A options.</para>
//         /// <para></para>
//         /// </param>
//         [MethodImpl(MethodImplOptions.AggressiveInlining)]
//         public Sequences(SynchronizedLinks<LinkIndex> links, SequencesOptions<LinkIndex>
    ↪  options)
//         {
//             Links = links;
//             _sync = links.SyncRoot;
//             Options = options;
//             Options.ValidateOptions();
//             Options.InitOptions(Links);
//             Constants = links.Constants;
//         }
//
//         /// <summary>
//         /// <para>
//         /// Initializes a new <see cref="Sequences"/> instance.
//         /// </para>
//         /// <para></para>
//         /// </summary>
//         /// <param name="links">
//         /// <para>A links.</para>
//         /// <para></para>
//         /// </param>
//         [MethodImpl(MethodImplOptions.AggressiveInlining)]
//         public Sequences(SynchronizedLinks<LinkIndex> links) : this(links, new
    ↪  SequencesOptions<LinkIndex>()) { }
//
//         /// <summary>
//         /// <para>
//         /// Determines whether this instance is sequence.
//         /// </para>
//         /// <para></para>
//         /// </summary>
//         /// <param name="sequence">
//         /// <para>The sequence.</para>
//         /// <para></para>
//         /// </param>
//         /// <returns>
//         /// <para>The bool</para>
```

```
//          /// <para></para>
//          /// </returns>
//          [MethodImpl(MethodImplOptions.AggressiveInlining)]
//          public bool IsSequence(LinkIndex sequence)
//          {
//              return _sync.DoRead(() =>
//              {
//                  if (Options.UseSequenceMarker)
//                  {
//                      return Options.MarkedSequenceMatcher.IsMatched(sequence);
//                  }
//                  return !Links.Unsync.IsPartialPoint(sequence);
//              });
//          }
//          [MethodImpl(MethodImplOptions.AggressiveInlining)]
//          private LinkIndex GetSequenceByElements(LinkIndex sequence)
//          {
//              if (Options.UseSequenceMarker)
//              {
//                  return Links.SearchOrDefault(Options.SequenceMarkerLink, sequence);
//              }
//              return sequence;
//          }
//          [MethodImpl(MethodImplOptions.AggressiveInlining)]
//          private LinkIndex GetSequenceElements(LinkIndex sequence)
//          {
//              if (Options.UseSequenceMarker)
//              {
//                  var linkContents = new Link<ulong>(Links.GetLink(sequence));
//                  if (linkContents.Source == Options.SequenceMarkerLink)
//                  {
//                      return linkContents.Target;
//                  }
//                  if (linkContents.Target == Options.SequenceMarkerLink)
//                  {
//                      return linkContents.Source;
//                  }
//              }
//              return sequence;
//          }
//
//          #region Count
//
//          /// <summary>
//          /// <para>
//          /// Counts the restriction.
//          /// </para>
//          /// <para></para>
//          /// </summary>
//          /// <param name="restriction">
//          /// <para>The restriction.</para>
//          /// <para></para>
//          /// </param>
//          /// <exception cref="NotImplementedException">
//          /// <para></para>
//          /// <para></para>
//          /// </exception>
//          /// <returns>
//          /// <para>The link index</para>
//          /// <para></para>
//          /// </returns>
//          [MethodImpl(MethodImplOptions.AggressiveInlining)]
//          public LinkIndex Count(IList<LinkIndex>? restriction)
//          {
//              if (restriction.IsNullOrEmpty())
//              {
//                  return Links.Count(Constants.Any, Options.SequenceMarkerLink, Constants.Any);
//              }
//              if (restriction.Count == 1) // Первая связь это адрес
//              {
//                  var sequenceIndex = restriction[0];
//                  if (sequenceIndex == Constants.Null)
//                  {
//                      return 0;
//                  }
//                  if (sequenceIndex == Constants.Any)
//                  {
//                      return Count(null);
```

```
208 //                 }
209 //                 if (Options.UseSequenceMarker)
210 //                 {
211 //                     return Links.Count(Constants.Any, Options.SequenceMarkerLink,
    ↪  sequenceIndex);
212 //                 }
213 //                 return Links.Exists(sequenceIndex) ? 1UL : 0;
214 //             }
215 //             throw new NotImplementedException();
216 //         }
217 //         [MethodImpl(MethodImplOptions.AggressiveInlining)]
218 //         private LinkIndex CountUsages(params LinkIndex[] restriction)
219 //         {
220 //             if (restriction.Length == 0)
221 //             {
222 //                 return 0;
223 //             }
224 //             if (restriction.Length == 1) // Первая связь это адрес
225 //             {
226 //                 if (restriction[0] == Constants.Null)
227 //                 {
228 //                     return 0;
229 //                 }
230 //                 var any = Constants.Any;
231 //                 if (Options.UseSequenceMarker)
232 //                 {
233 //                     var elementsLink = GetSequenceElements(restriction[0]);
234 //                     var sequenceLink = GetSequenceByElements(elementsLink);
235 //                     if (sequenceLink != Constants.Null)
236 //                     {
237 //                         return Links.Count(any, sequenceLink) + Links.Count(any,
    ↪  elementsLink) - 1;
238 //                     }
239 //                     return Links.Count(any, elementsLink);
240 //                 }
241 //                 return Links.Count(any, restriction[0]);
242 //             }
243 //             throw new NotImplementedException();
244 //         }
245 //
246 //         #endregion
247 //
248 //         #region Create
249 //
250 //         /// <summary>
251 //         /// <para>
252 //         /// Creates the restriction.
253 //         /// </para>
254 //         /// <para></para>
255 //         /// </summary>
256 //         /// <param name="restriction">
257 //         /// <para>The restriction.</para>
258 //         /// <para></para>
259 //         /// </param>
260 //         /// <returns>
261 //         /// <para>The link index</para>
262 //         /// <para></para>
263 //         /// </returns>
264 //         [MethodImpl(MethodImplOptions.AggressiveInlining)]
265 //         public LinkIndex Create(IList<LinkIndex>? restriction)
266 //         {
267 //             return _sync.DoWrite(() =>
268 //             {
269 //                 if (restriction.IsNullOrEmpty())
270 //                 {
271 //                     return Constants.Null;
272 //                 }
273 //                 Links.EnsureInnerReferenceExists(restriction, nameof(restriction));
274 //                 return CreateCore(restriction);
275 //             });
276 //         }
277 //         [MethodImpl(MethodImplOptions.AggressiveInlining)]
278 //         private LinkIndex CreateCore(IList<LinkIndex>? restriction)
279 //         {
280 //             LinkIndex[] sequence = restriction.SkipFirst();
281 //             if (Options.UseIndex)
282 //             {
```

```
//                 Options.Index.Add(sequence);
//             }
//             var sequenceRoot = default(LinkIndex);
//             if (Options.EnforceSingleSequenceVersionOnWriteBasedOnExisting)
//             {
//                 var matches = Each(restriction);
//                 if (matches.Count > 0)
//                 {
//                     sequenceRoot = matches[0];
//                 }
//             }
//             else if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew)
//             {
//                 return CompactCore(sequence);
//             }
//             if (sequenceRoot == default)
//             {
//                 sequenceRoot = Options.LinksToSequenceConverter.Convert(sequence);
//             }
//             if (Options.UseSequenceMarker)
//             {
//                 return Links.Unsync.GetOrCreate(Options.SequenceMarkerLink, sequenceRoot);
//             }
//             return sequenceRoot; // Возвращаем корень последовательности (т.е. сами элементы)
//         }
//
//         #endregion
//
//         #region Each
//
//         /// <summary>
//         /// <para>
//         /// Eaches the sequence.
//         /// </para>
//         /// <para></para>
//         /// </summary>
//         /// <param name="sequence">
//         /// <para>The sequence.</para>
//         /// <para></para>
//         /// </param>
//         /// <returns>
//         /// <para>The results.</para>
//         /// <para></para>
//         /// </returns>
//         [MethodImpl(MethodImplOptions.AggressiveInlining)]
//         public List<LinkIndex> Each(IList<LinkIndex> sequence)
//         {
//             var results = new List<LinkIndex>();
//             var filler = new ListFiller<LinkIndex, LinkIndex>(results, Constants.Continue);
//             Each(filler.AddFirstAndReturnConstant, sequence);
//             return results;
//         }
//
//         /// <summary>
//         /// <para>
//         /// Eaches the handler.
//         /// </para>
//         /// <para></para>
//         /// </summary>
//         /// <param name="handler">
//         /// <para>The handler.</para>
//         /// <para></para>
//         /// </param>
//         /// <param name="restriction">
//         /// <para>The restriction.</para>
//         /// <para></para>
//         /// </param>
//         /// <exception cref="NotImplementedException">
//         /// <para></para>
//         /// <para></para>
//         /// </exception>
//         /// <returns>
//         /// <para>The link index</para>
//         /// <para></para>
//         /// </returns>
//         [MethodImpl(MethodImplOptions.AggressiveInlining)]
//         public LinkIndex Each(ReadHandler<LinkIndex> handler, IList<LinkIndex>? restriction)
//         {
```

```
361 //             return _sync.DoRead(() =>
362 //             {
363 //                 if (restriction.IsNullOrEmpty())
364 //                 {
365 //                     return Constants.Continue;
366 //                 }
367 //                 Links.EnsureInnerReferenceExists(restriction, nameof(restriction));
368 //                 if (restriction.Count == 1)
369 //                 {
370 //                     var link = restriction[0];
371 //                     var any = Constants.Any;
372 //                     if (link == any)
373 //                     {
374 //                         if (Options.UseSequenceMarker)
375 //                         {
376 //                             return Links.Unsync.Each(new Link<LinkIndex>(any,
    ↪ Options.SequenceMarkerLink, any), handler);
377 //                         }
378 //                         else
379 //                         {
380 //                             return Links.Unsync.Each(handler, new Link<LinkIndex>(any, any,
    ↪ any));
381 //                         }
382 //                     }
383 //                     if (Options.UseSequenceMarker)
384 //                     {
385 //                         var sequenceLinkValues = Links.Unsync.GetLink(link);
386 //                         if (sequenceLinkValues[Constants.SourcePart] ==
    ↪ Options.SequenceMarkerLink)
387 //                         {
388 //                             link = sequenceLinkValues[Constants.TargetPart];
389 //                         }
390 //                     }
391 //                     var sequence = Options.Walker.Walk(link).ToArray().ShiftRight();
392 //                     sequence[0] = link;
393 //                     return handler(sequence);
394 //                 }
395 //                 else if (restriction.Count == 2)
396 //                 {
397 //                     throw new NotImplementedException();
398 //                 }
399 //                 else if (restriction.Count == 3)
400 //                 {
401 //                     return Links.Unsync.Each(restriction, handler);
402 //                 }
403 //                 else
404 //                 {
405 //                     var sequence = restriction.SkipFirst();
406 //                     if (Options.UseIndex && !Options.Index.MightContain(sequence))
407 //                     {
408 //                         return Constants.Break;
409 //                     }
410 //                     return EachCore(sequence, handler);
411 //                 }
412 //             });
413 //         }
414 //         [MethodImpl(MethodImplOptions.AggressiveInlining)]
415 //         private LinkIndex EachCore(IList<LinkIndex> values, ReadHandler<LinkIndex> handler)
416 //         {
417 //             var matcher = new Matcher(this, values, new HashSet<LinkIndex>(), handler);
418 //             // TODO: Find out why matcher.HandleFullMatched executed twice for the same
    ↪ sequence Id.
419 //             Func<IList<LinkIndex>, LinkIndex> innerHandler = Options.UseSequenceMarker ?
    ↪ (Func<IList<LinkIndex>, LinkIndex>)matcher.HandleFullMatchedSequence :
    ↪ matcher.HandleFullMatched;
420 //             //if (sequence.Length >= 2)
421 //             if (StepRight(innerHandler, values[0], values[1]) != Constants.Continue)
422 //             {
423 //                 return Constants.Break;
424 //             }
425 //             var last = values.Count - 2;
426 //             for (var i = 1; i < last; i++)
427 //             {
428 //                 if (PartialStepRight(innerHandler, values[i], values[i + 1]) !=
    ↪ Constants.Continue)
429 //                 {
430 //                     return Constants.Break;
```

```
//                 }
//             }
//             if (values.Count >= 3)
//             {
//                 if (StepLeft(innerHandler, values[values.Count - 2], values[values.Count -
↪  1]) != Constants.Continue)
//                 {
//                     return Constants.Break;
//                 }
//             }
//             return Constants.Continue;
//         }
//         [MethodImpl(MethodImplOptions.AggressiveInlining)]
//         private LinkIndex PartialStepRight(Func<IList<LinkIndex>, LinkIndex> handler,
↪  LinkIndex left, LinkIndex right)
//         {
//             return Links.Unsync.Each(doublet =>
//             {
//                 var doubletIndex = doublet[Constants.IndexPart];
//                 if (StepRight(handler, doubletIndex, right) != Constants.Continue)
//                 {
//                     return Constants.Break;
//                 }
//                 if (left != doubletIndex)
//                 {
//                     return PartialStepRight(handler, doubletIndex, right);
//                 }
//                 return Constants.Continue;
//             }, new Link<LinkIndex>(Constants.Any, Constants.Any, left));
//         }
//         [MethodImpl(MethodImplOptions.AggressiveInlining)]
//         private LinkIndex StepRight(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
↪  left, LinkIndex right) => Links.Unsync.Each(rightStep => TryStepRightUp(handler, right,
↪  rightStep[Constants.IndexPart]), new Link<LinkIndex>(Constants.Any, left, Constants.Any));
//         [MethodImpl(MethodImplOptions.AggressiveInlining)]
//         private LinkIndex TryStepRightUp(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
↪  right, LinkIndex stepFrom)
//         {
//             var upStep = stepFrom;
//             var firstSource = Links.Unsync.GetTarget(upStep);
//             while (firstSource != right && firstSource != upStep)
//             {
//                 upStep = firstSource;
//                 firstSource = Links.Unsync.GetSource(upStep);
//             }
//             if (firstSource == right)
//             {
//                 return handler(new LinkAddress<LinkIndex>(stepFrom));
//             }
//             return Constants.Continue;
//         }
//         [MethodImpl(MethodImplOptions.AggressiveInlining)]
//         private LinkIndex StepLeft(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex left,
↪  LinkIndex right) => Links.Unsync.Each(leftStep => TryStepLeftUp(handler, left,
↪  leftStep[Constants.IndexPart]), new Link<LinkIndex>(Constants.Any, Constants.Any, right));
//         [MethodImpl(MethodImplOptions.AggressiveInlining)]
//         private LinkIndex TryStepLeftUp(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
↪  left, LinkIndex stepFrom)
//         {
//             var upStep = stepFrom;
//             var firstTarget = Links.Unsync.GetSource(upStep);
//             while (firstTarget != left && firstTarget != upStep)
//             {
//                 upStep = firstTarget;
//                 firstTarget = Links.Unsync.GetTarget(upStep);
//             }
//             if (firstTarget == left)
//             {
//                 return handler(new LinkAddress<LinkIndex>(stepFrom));
//             }
//             return Constants.Continue;
//         }
//
//         #endregion
//
//         #region Update
//
```

```
//         /// <summary>
//         /// <para>
//         /// Updates the restriction.
//         /// </para>
//         /// <para></para>
//         /// </summary>
//         /// <param name="restriction">
//         /// <para>The restriction.</para>
//         /// <para></para>
//         /// </param>
//         /// <param name="substitution">
//         /// <para>The substitution.</para>
//         /// <para></para>
//         /// </param>
//         /// <returns>
//         /// <para>The link index</para>
//         /// <para></para>
//         /// </returns>
//         [MethodImpl(MethodImplOptions.AggressiveInlining)]
//         public LinkIndex Update(IList<LinkIndex>? restriction, IList<LinkIndex>?
    ↪ substitution, WriteHandler<LinkIndex> handler)
//         {
//             var sequence = restriction.SkipFirst();
//             var newSequence = substitution.SkipFirst();
//             if (sequence.IsNullOrEmpty() && newSequence.IsNullOrEmpty())
//             {
//                 return Constants.Null;
//             }
//             if (sequence.IsNullOrEmpty())
//             {
//                 return Create(substitution);
//             }
//             if (newSequence.IsNullOrEmpty())
//             {
//                 Delete(restriction);
//                 return Constants.Null;
//             }
//             return _sync.DoWrite((Func<ulong>)(() =>
//             {
//                 ILinksExtensions.EnsureLinkIsAnyOrExists<ulong>(Links,
    ↪ (IList<ulong>)sequence);
//                 Links.EnsureLinkExists(newSequence);
//                 return UpdateCore(sequence, newSequence);
//             }));
//         }
//         [MethodImpl(MethodImplOptions.AggressiveInlining)]
//         private LinkIndex UpdateCore(IList<LinkIndex> sequence, IList<LinkIndex> newSequence,
    ↪ WriteHandler<LinkIndex> handler)
//         {
//             LinkIndex bestVariant;
//             if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew &&
    ↪ !sequence.EqualTo(newSequence))
//             {
//                 bestVariant = CompactCore(newSequence);
//             }
//             else
//             {
//                 bestVariant = CreateCore(newSequence);
//             }
//             // TODO: Check all options only ones before loop execution
//             // Возможно нужно две версии Each, возвращающий фактические последовательности и
    ↪ с маркером,
//             // или возможно даже возвращать и тот и тот вариант. С другой стороны все
    ↪ варианты можно получить имея только фактические последовательности.
//             foreach (var variant in Each(sequence))
//             {
//                 if (variant != bestVariant)
//                 {
//                     UpdateOneCore(variant, bestVariant);
//                 }
//             }
//             return bestVariant;
//         }
//         [MethodImpl(MethodImplOptions.AggressiveInlining)]
//         private void UpdateOneCore(LinkIndex sequence, LinkIndex newSequence,
    ↪ WriteHandler<LinkIndex> handler)
//         {
```

```
//             if (Options.UseGarbageCollection)
//             {
//                 var sequenceElements = GetSequenceElements(sequence);
//                 var sequenceElementsContents = new
    ↪ Link<ulong>(Links.GetLink(sequenceElements));
//                 var sequenceLink = GetSequenceByElements(sequenceElements);
//                 var newSequenceElements = GetSequenceElements(newSequence);
//                 var newSequenceLink = GetSequenceByElements(newSequenceElements);
//                 if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
//                 {
//                     if (sequenceLink != Constants.Null)
//                     {
//                         Links.Unsync.MergeAndDelete(sequenceLink, newSequenceLink);
//                     }
//                     Links.Unsync.MergeAndDelete(sequenceElements, newSequenceElements);
//                 }
//                 ClearGarbage(sequenceElementsContents.Source);
//                 ClearGarbage(sequenceElementsContents.Target);
//             }
//             else
//             {
//                 if (Options.UseSequenceMarker)
//                 {
//                     var sequenceElements = GetSequenceElements(sequence);
//                     var sequenceLink = GetSequenceByElements(sequenceElements);
//                     var newSequenceElements = GetSequenceElements(newSequence);
//                     var newSequenceLink = GetSequenceByElements(newSequenceElements);
//                     if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
//                     {
//                         if (sequenceLink != Constants.Null)
//                         {
//                             Links.Unsync.MergeAndDelete(sequenceLink, newSequenceLink);
//                         }
//                         Links.Unsync.MergeAndDelete(sequenceElements, newSequenceElements);
//                     }
//                 }
//                 else
//                 {
//                     if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
//                     {
//                         Links.Unsync.MergeAndDelete(sequence, newSequence);
//                     }
//                 }
//             }
//         }
//
//         #endregion
//
//         #region Delete
//
//         /// <summary>
//         /// <para>
//         /// Deletes the restriction.
//         /// </para>
//         /// <para></para>
//         /// </summary>
//         /// <param name="restriction">
//         /// <para>The restriction.</para>
//         /// <para></para>
//         /// </param>
//         [MethodImpl(MethodImplOptions.AggressiveInlining)]
//         public void Delete(IList<LinkIndex>? restriction)
//         {
//             _sync.DoWrite(() =>
//             {
//                 var sequence = restriction.SkipFirst();
//                 // TODO: Check all options only ones before loop execution
//                 foreach (var linkToDelete in Each(sequence))
//                 {
//                     DeleteOneCore(linkToDelete);
//                 }
//             });
//         }
//         [MethodImpl(MethodImplOptions.AggressiveInlining)]
//         private void DeleteOneCore(LinkIndex link)
//         {
//             if (Options.UseGarbageCollection)
```

```
//             {
//                 var sequenceElements = GetSequenceElements(link);
//                 var sequenceElementsContents = new
    ↪ Link<ulong>(Links.GetLink(sequenceElements));
//                 var sequenceLink = GetSequenceByElements(sequenceElements);
//                 if (Options.UseCascadeDelete || CountUsages(link) == 0)
//                 {
//                     if (sequenceLink != Constants.Null)
//                     {
//                         Links.Unsync.Delete(sequenceLink);
//                     }
//                     Links.Unsync.Delete(link);
//                 }
//                 ClearGarbage(sequenceElementsContents.Source);
//                 ClearGarbage(sequenceElementsContents.Target);
//             }
//             else
//             {
//                 if (Options.UseSequenceMarker)
//                 {
//                     var sequenceElements = GetSequenceElements(link);
//                     var sequenceLink = GetSequenceByElements(sequenceElements);
//                     if (Options.UseCascadeDelete || CountUsages(link) == 0)
//                     {
//                         if (sequenceLink != Constants.Null)
//                         {
//                             Links.Unsync.Delete(sequenceLink);
//                         }
//                         Links.Unsync.Delete(link);
//                     }
//                 }
//                 else
//                 {
//                     if (Options.UseCascadeDelete || CountUsages(link) == 0)
//                     {
//                         Links.Unsync.Delete(link);
//                     }
//                 }
//             }
//         }
//
//         #endregion
//
//         #region Compactification
//
//         /// <summary>
//         /// <para>
//         /// Compacts the all.
//         /// </para>
//         /// <para></para>
//         /// </summary>
//         [MethodImpl(MethodImplOptions.AggressiveInlining)]
//         public void CompactAll()
//         {
//             _sync.DoWrite(() =>
//             {
//                 var sequences = Each((LinkAddress<LinkIndex>)Constants.Any);
//                 for (int i = 0; i < sequences.Count; i++)
//                 {
//                     var sequence = this.ToList(sequences[i]);
//                     Compact(sequence.ShiftRight());
//                 }
//             });
//         }
//
//         /// <remarks>
//         /// bestVariant можно выбирать по максимальному числу использований,
//         /// но балансированный позволяет гарантировать уникальность (если есть возможность,
//         /// гарантировать его использование в других местах).
//         ///
//         /// Получается этот метод должен игнорировать
    ↪ Options.EnforceSingleSequenceVersionOnWrite
//         /// </remarks>
//         [MethodImpl(MethodImplOptions.AggressiveInlining)]
//         public LinkIndex Compact(IList<LinkIndex> sequence)
//         {
//             return _sync.DoWrite(() =>
```

```
//             {
//                 if (sequence.IsNullOrEmpty())
//                 {
//                     return Constants.Null;
//                 }
//                 Links.EnsureInnerReferenceExists(sequence, nameof(sequence));
//                 return CompactCore(sequence);
//             });
//         }
//         [MethodImpl(MethodImplOptions.AggressiveInlining)]
//         private LinkIndex CompactCore(IList<LinkIndex> sequence) => UpdateCore(sequence,
    ↪ sequence);
//
//         #endregion
//
//         #region Garbage Collection
//         [MethodImpl(MethodImplOptions.AggressiveInlining)]
//         private bool IsGarbage(LinkIndex link) => link != Options.SequenceMarkerLink &&
    ↪ !Links.Unsync.IsPartialPoint(link) && Links.Count(Constants.Any, link) == 0;
//         [MethodImpl(MethodImplOptions.AggressiveInlining)]
//         private void ClearGarbage(LinkIndex link)
//         {
//             if (IsGarbage(link))
//             {
//                 var contents = new Link<ulong>(Links.GetLink(link));
//                 Links.Unsync.Delete(link);
//                 ClearGarbage(contents.Source);
//                 ClearGarbage(contents.Target);
//             }
//         }
//
//         #endregion
//
//         #region Walkers
//
//         /// <summary>
//         /// <para>
//         /// Determines whether this instance each part.
//         /// </para>
//         /// <para></para>
//         /// </summary>
//         /// <param name="handler">
//         /// <para>The handler.</para>
//         /// <para></para>
//         /// </param>
//         /// <param name="sequence">
//         /// <para>The sequence.</para>
//         /// <para></para>
//         /// </param>
//         /// <returns>
//         /// <para>The bool</para>
//         /// <para></para>
//         /// </returns>
//         [MethodImpl(MethodImplOptions.AggressiveInlining)]
//         public bool EachPart(Func<LinkIndex, bool> handler, LinkIndex sequence)
//         {
//             return _sync.DoRead(() =>
//             {
//                 var links = Links.Unsync;
//                 foreach (var part in Options.Walker.Walk(sequence))
//                 {
//                     if (!handler(part))
//                     {
//                         return false;
//                     }
//                 }
//                 return true;
//             });
//         }
//
//         /// <summary>
//         /// <para>
//         /// Represents the matcher.
//         /// </para>
//         /// <para></para>
//         /// </summary>
//         /// <seealso cref="RightSequenceWalker{LinkIndex}"/>
```

```
//         public class Matcher : RightSequenceWalker<LinkIndex>
//         {
//             private readonly Sequences _sequences;
//             private readonly IList<LinkIndex> _patternSequence;
//             private readonly HashSet<LinkIndex> _linksInSequence;
//             private readonly HashSet<LinkIndex> _results;
//             private readonly ReadHandler<LinkIndex> _stopableHandler;
//             private readonly HashSet<LinkIndex> _readAsElements;
//             private int _filterPosition;
//
//             /// <summary>
//             /// <para>
//             /// Initializes a new <see cref="Matcher"/> instance.
//             /// </para>
//             /// <para></para>
//             /// </summary>
//             /// <param name="sequences">
//             /// <para>A sequences.</para>
//             /// <para></para>
//             /// </param>
//             /// <param name="patternSequence">
//             /// <para>A pattern sequence.</para>
//             /// <para></para>
//             /// </param>
//             /// <param name="results">
//             /// <para>A results.</para>
//             /// <para></para>
//             /// </param>
//             /// <param name="stopableHandler">
//             /// <para>A stopable handler.</para>
//             /// <para></para>
//             /// </param>
//             /// <param name="readAsElements">
//             /// <para>A read as elements.</para>
//             /// <para></para>
//             /// </param>
//             [MethodImpl(MethodImplOptions.AggressiveInlining)]
//             public Matcher(Sequences sequences, IList<LinkIndex> patternSequence,
    ↪ HashSet<LinkIndex> results, ReadHandler<LinkIndex> stopableHandler, HashSet<LinkIndex>
    ↪ readAsElements = null)
//                 : base(sequences.Links.Unsync, new DefaultStack<LinkIndex>())
//             {
//                 _sequences = sequences;
//                 _patternSequence = patternSequence;
//                 _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
    ↪ _links.Constants.Any && x != ZeroOrMany));
//                 _results = results;
//                 _stopableHandler = stopableHandler;
//                 _readAsElements = readAsElements;
//             }
//
//             /// <summary>
//             /// <para>
//             /// Determines whether this instance is element.
//             /// </para>
//             /// <para></para>
//             /// </summary>
//             /// <param name="link">
//             /// <para>The link.</para>
//             /// <para></para>
//             /// </param>
//             /// <returns>
//             /// <para>The bool</para>
//             /// <para></para>
//             /// </returns>
//             [MethodImpl(MethodImplOptions.AggressiveInlining)]
//             protected override bool IsElement(LinkIndex link) => base.IsElement(link) ||
    ↪ (_readAsElements != null && _readAsElements.Contains(link)) ||
    ↪ _linksInSequence.Contains(link);
//
//             /// <summary>
//             /// <para>
//             /// Determines whether this instance full match.
//             /// </para>
//             /// <para></para>
//             /// </summary>
//             /// <param name="sequenceToMatch">
```

```
//              /// <para>The sequence to match.</para>
//              /// <para></para>
//              /// </param>
//              /// <returns>
//              /// <para>The bool</para>
//              /// <para></para>
//              /// </returns>
//              [MethodImpl(MethodImplOptions.AggressiveInlining)]
//              public bool FullMatch(LinkIndex sequenceToMatch)
//              {
//                  _filterPosition = 0;
//                  foreach (var part in Walk(sequenceToMatch))
//                  {
//                      if (!FullMatchCore(part))
//                      {
//                          break;
//                      }
//                  }
//                  return _filterPosition == _patternSequence.Count;
//              }
//              [MethodImpl(MethodImplOptions.AggressiveInlining)]
//              private bool FullMatchCore(LinkIndex element)
//              {
//                  if (_filterPosition == _patternSequence.Count)
//                  {
//                      _filterPosition = -2; // Длиннее чем нужно
//                      return false;
//                  }
//                  if (_patternSequence[_filterPosition] != _links.Constants.Any
//                   && element != _patternSequence[_filterPosition])
//                  {
//                      _filterPosition = -1;
//                      return false; // Начинается/Продолжается иначе
//                  }
//                  _filterPosition++;
//                  return true;
//              }
//
//              /// <summary>
//              /// <para>
//              /// Adds the full matched to results using the specified restriction.
//              /// </para>
//              /// <para></para>
//              /// </summary>
//              /// <param name="restriction">
//              /// <para>The restriction.</para>
//              /// <para></para>
//              /// </param>
//              [MethodImpl(MethodImplOptions.AggressiveInlining)]
//              public void AddFullMatchedToResults(IList<LinkIndex>? restriction)
//              {
//                  var sequenceToMatch = restriction[_links.Constants.IndexPart];
//                  if (FullMatch(sequenceToMatch))
//                  {
//                      _results.Add(sequenceToMatch);
//                  }
//              }
//
//              /// <summary>
//              /// <para>
//              /// Handles the full matched using the specified restriction.
//              /// </para>
//              /// <para></para>
//              /// </summary>
//              /// <param name="restriction">
//              /// <para>The restriction.</para>
//              /// <para></para>
//              /// </param>
//              /// <returns>
//              /// <para>The link index</para>
//              /// <para></para>
//              /// </returns>
//              [MethodImpl(MethodImplOptions.AggressiveInlining)]
//              public LinkIndex HandleFullMatched(IList<LinkIndex>? restriction)
//              {
//                  var sequenceToMatch = restriction[_links.Constants.IndexPart];
//                  if (FullMatch(sequenceToMatch) && _results.Add(sequenceToMatch))
//                  {
```

```
//                     return _stopableHandler(new LinkAddress<LinkIndex>(sequenceToMatch));
//                 }
//                 return _links.Constants.Continue;
//             }
//
//             /// <summary>
//             /// <para>
//             /// Handles the full matched sequence using the specified restriction.
//             /// </para>
//             /// <para></para>
//             /// </summary>
//             /// <param name="restriction">
//             /// <para>The restriction.</para>
//             /// <para></para>
//             /// </param>
//             /// <returns>
//             /// <para>The link index</para>
//             /// <para></para>
//             /// </returns>
//             [MethodImpl(MethodImplOptions.AggressiveInlining)]
//             public LinkIndex HandleFullMatchedSequence(IList<LinkIndex>? restriction)
//             {
//                 var sequenceToMatch = restriction[_links.Constants.IndexPart];
//                 var sequence = _sequences.GetSequenceByElements(sequenceToMatch);
//                 if (sequence != _links.Constants.Null && FullMatch(sequenceToMatch) &&
    ↪ _results.Add(sequenceToMatch))
//                 {
//                     return _stopableHandler(new LinkAddress<LinkIndex>(sequence));
//                 }
//                 return _links.Constants.Continue;
//             }
//
//             /// <remarks>
//             /// TODO: Add support for LinksConstants.Any
//             /// </remarks>
//             [MethodImpl(MethodImplOptions.AggressiveInlining)]
//             public bool PartialMatch(LinkIndex sequenceToMatch)
//             {
//                 _filterPosition = -1;
//                 foreach (var part in Walk(sequenceToMatch))
//                 {
//                     if (!PartialMatchCore(part))
//                     {
//                         break;
//                     }
//                 }
//                 return _filterPosition == _patternSequence.Count - 1;
//             }
//             [MethodImpl(MethodImplOptions.AggressiveInlining)]
//             private bool PartialMatchCore(LinkIndex element)
//             {
//                 if (_filterPosition == (_patternSequence.Count - 1))
//                 {
//                     return false; // Нашлось
//                 }
//                 if (_filterPosition >= 0)
//                 {
//                     if (element == _patternSequence[_filterPosition + 1])
//                     {
//                         _filterPosition++;
//                     }
//                     else
//                     {
//                         _filterPosition = -1;
//                     }
//                 }
//                 if (_filterPosition < 0)
//                 {
//                     if (element == _patternSequence[0])
//                     {
//                         _filterPosition = 0;
//                     }
//                 }
//                 return true; // Ищем дальше
//             }
//
//             /// <summary>
```

```
//              /// <para>
//              /// Adds the partial matched to results using the specified sequence to match.
//              /// </para>
//              /// <para></para>
//              /// </summary>
//              /// <param name="sequenceToMatch">
//              /// <para>The sequence to match.</para>
//              /// <para></para>
//              /// </param>
//              [MethodImpl(MethodImplOptions.AggressiveInlining)]
//              public void AddPartialMatchedToResults(LinkIndex sequenceToMatch)
//              {
//                  if (PartialMatch(sequenceToMatch))
//                  {
//                      _results.Add(sequenceToMatch);
//                  }
//              }
//
//              /// <summary>
//              /// <para>
//              /// Handles the partial matched using the specified restriction.
//              /// </para>
//              /// <para></para>
//              /// </summary>
//              /// <param name="restriction">
//              /// <para>The restriction.</para>
//              /// <para></para>
//              /// </param>
//              /// <returns>
//              /// <para>The link index</para>
//              /// <para></para>
//              /// </returns>
//              [MethodImpl(MethodImplOptions.AggressiveInlining)]
//              public LinkIndex HandlePartialMatched(IList<LinkIndex>? restriction)
//              {
//                  var sequenceToMatch = restriction[_links.Constants.IndexPart];
//                  if (PartialMatch(sequenceToMatch))
//                  {
//                      return _stopableHandler(new LinkAddress<LinkIndex>(sequenceToMatch));
//                  }
//                  return _links.Constants.Continue;
//              }
//
//              /// <summary>
//              /// <para>
//              /// Adds the all partial matched to results using the specified sequences to
    ↪  match.
//              /// </para>
//              /// <para></para>
//              /// </summary>
//              /// <param name="sequencesToMatch">
//              /// <para>The sequences to match.</para>
//              /// <para></para>
//              /// </param>
//              [MethodImpl(MethodImplOptions.AggressiveInlining)]
//              public void AddAllPartialMatchedToResults(IEnumerable<LinkIndex> sequencesToMatch)
//              {
//                  foreach (var sequenceToMatch in sequencesToMatch)
//                  {
//                      if (PartialMatch(sequenceToMatch))
//                      {
//                          _results.Add(sequenceToMatch);
//                      }
//                  }
//              }
//
//              /// <summary>
//              /// <para>
//              /// Adds the all partial matched to results and read as elements using the
    ↪  specified sequences to match.
//              /// </para>
//              /// <para></para>
//              /// </summary>
//              /// <param name="sequencesToMatch">
//              /// <para>The sequences to match.</para>
//              /// <para></para>
//              /// </param>
//              [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
//             public void AddAllPartialMatchedToResultsAndReadAsElements(IEnumerable<LinkIndex>
    ↪  sequencesToMatch)
//             {
//                 foreach (var sequenceToMatch in sequencesToMatch)
//                 {
//                     if (PartialMatch(sequenceToMatch))
//                     {
//                         _readAsElements.Add(sequenceToMatch);
//                         _results.Add(sequenceToMatch);
//                     }
//                 }
//             }
//         }
//
//         #endregion
//     }
// }
```

1.47 ./csharp/Platform.Data.Doublets.Sequences/SequencesExtensions.cs

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

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

namespace Platform.Data.Doublets.Sequences
{
    /// <summary>
    /// <para>
    /// Represents the sequences extensions.
    /// </para>
    /// <para></para>
    /// </summary>
    public static class SequencesExtensions
    {
        /// <summary>
        /// <para>
        /// Creates the sequences.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="TLinkAddress">
        /// <para>The link.</para>
        /// <para></para>
        /// </typeparam>
        /// <param name="sequences">
        /// <para>The sequences.</para>
        /// <para></para>
        /// </param>
        /// <param name="groupedSequence">
        /// <para>The grouped sequence.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static TLinkAddress Create<TLinkAddress>(this ILinks<TLinkAddress> sequences,
        ↪  IList<TLinkAddress[]> groupedSequence)
        {
            var finalSequence = new TLinkAddress[groupedSequence.Count];
            for (var i = 0; i < finalSequence.Length; i++)
            {
                var part = groupedSequence[i];
                finalSequence[i] = part.Length == 1 ? part[0] :
                ↪  sequences.Create(part.ShiftRight());
            }
            return sequences.Create(finalSequence.ShiftRight());
        }

        /// <summary>
        /// <para>
        /// Returns the list using the specified sequences.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <typeparam name="TLinkAddress">
```

```
58          /// <para>The link.</para>
59          /// <para></para>
60          /// </typeparam>
61          /// <param name="sequences">
62          /// <para>The sequences.</para>
63          /// <para></para>
64          /// </param>
65          /// <param name="sequence">
66          /// <para>The sequence.</para>
67          /// <para></para>
68          /// </param>
69          /// <returns>
70          /// <para>The list.</para>
71          /// <para></para>
72          /// </returns>
73          [MethodImpl(MethodImplOptions.AggressiveInlining)]
74          public static IList<TLinkAddress>? ToList<TLinkAddress>(this ILinks<TLinkAddress>
            ↪ sequences, TLinkAddress sequence)
75          {
76              var list = new List<TLinkAddress>();
77              var filler = new ListFiller<TLinkAddress, TLinkAddress>(list,
                ↪ sequences.Constants.Break);
78              sequences.Each(filler.AddSkipFirstAndReturnConstant, new
                ↪ LinkAddress<TLinkAddress>(sequence));
79              return list;
80          }
81      }
82  }
```

1.48 ./csharp/Platform.Data.Doublets.Sequences/SequencesOptions.cs

```
1   using System;
2   using System.Collections.Generic;
3   using Platform.Interfaces;
4   using Platform.Collections.Stacks;
5   using Platform.Converters;
6   using Platform.Data.Doublets.Sequences.Frequencies.Cache;
7   using Platform.Data.Doublets.Sequences.Frequencies.Counters;
8   using Platform.Data.Doublets.Sequences.Converters;
9   using Platform.Data.Doublets.Sequences.Walkers;
10  using Platform.Data.Doublets.Sequences.Indexes;
11  using Platform.Data.Doublets.Sequences.CriterionMatchers;
12  using System.Runtime.CompilerServices;
13
14  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
15
16  namespace Platform.Data.Doublets.Sequences
17  {
18      /// <summary>
19      /// <para>
20      /// Represents the sequences options.
21      /// </para>
22      /// <para></para>
23      /// </summary>
24      public class SequencesOptions<TLinkAddress> // TODO: To use type parameter <TLinkAddress>
        ↪ the ILinks<TLinkAddress> must contain GetConstants function.
25      {
26          private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
            ↪ EqualityComparer<TLinkAddress>.Default;
27
28          /// <summary>
29          /// <para>
30          /// Gets or sets the sequence marker link value.
31          /// </para>
32          /// <para></para>
33          /// </summary>
34          public TLinkAddress SequenceMarkerLink
35          {
36              [MethodImpl(MethodImplOptions.AggressiveInlining)]
37              get;
38              [MethodImpl(MethodImplOptions.AggressiveInlining)]
39              set;
40          }
41
42          /// <summary>
43          /// <para>
44          /// Gets or sets the use cascade update value.
45          /// </para>
46          /// <para></para>
47          /// </summary>
```

```
        public bool UseCascadeUpdate
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

        /// <summary>
        /// <para>
        /// Gets or sets the use cascade delete value.
        /// </para>
        /// <para></para>
        /// </summary>
        public bool UseCascadeDelete
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

        /// <summary>
        /// <para>
        /// Gets or sets the use index value.
        /// </para>
        /// <para></para>
        /// </summary>
        public bool UseIndex
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        } // TODO: Update Index on sequence update/delete.

        /// <summary>
        /// <para>
        /// Gets or sets the use sequence marker value.
        /// </para>
        /// <para></para>
        /// </summary>
        public bool UseSequenceMarker
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

        /// <summary>
        /// <para>
        /// Gets or sets the use compression value.
        /// </para>
        /// <para></para>
        /// </summary>
        public bool UseCompression
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

        /// <summary>
        /// <para>
        /// Gets or sets the use garbage collection value.
        /// </para>
        /// <para></para>
        /// </summary>
        public bool UseGarbageCollection
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

        /// <summary>
        /// <para>
```

```
        /// Gets or sets the enforce single sequence version on write based on existing value.
        /// </para>
        /// <para></para>
        /// </summary>
        public bool EnforceSingleSequenceVersionOnWriteBasedOnExisting
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

        /// <summary>
        /// <para>
        /// Gets or sets the enforce single sequence version on write based on new value.
        /// </para>
        /// <para></para>
        /// </summary>
        public bool EnforceSingleSequenceVersionOnWriteBasedOnNew
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

        /// <summary>
        /// <para>
        /// Gets or sets the marked sequence matcher value.
        /// </para>
        /// <para></para>
        /// </summary>
        public MarkedSequenceCriterionMatcher<TLinkAddress> MarkedSequenceMatcher
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

        /// <summary>
        /// <para>
        /// Gets or sets the links to sequence converter value.
        /// </para>
        /// <para></para>
        /// </summary>
        public IConverter<IList<TLinkAddress>, TLinkAddress> LinksToSequenceConverter
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

        /// <summary>
        /// <para>
        /// Gets or sets the index value.
        /// </para>
        /// <para></para>
        /// </summary>
        public ISequenceIndex<TLinkAddress> Index
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
        }

        /// <summary>
        /// <para>
        /// Gets or sets the walker value.
        /// </para>
        /// <para></para>
        /// </summary>
        public ISequenceWalker<TLinkAddress> Walker
        {
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            get;
            [MethodImpl(MethodImplOptions.AggressiveInlining)]
            set;
```

```
208         }
209
210         /// <summary>
211         /// <para>
212         /// Gets or sets the read full sequence value.
213         /// </para>
214         /// <para></para>
215         /// </summary>
216         public bool ReadFullSequence
217         {
218             [MethodImpl(MethodImplOptions.AggressiveInlining)]
219             get;
220             [MethodImpl(MethodImplOptions.AggressiveInlining)]
221             set;
222         }
223
224         // TODO: Реализовать компактификацию при чтении
225         //public bool EnforceSingleSequenceVersionOnRead { get; set; }
226         //public bool UseRequestMarker { get; set; }
227         //public bool StoreRequestResults { get; set; }
228
229         /// <summary>
230         /// <para>
231         /// Inits the options using the specified links.
232         /// </para>
233         /// <para></para>
234         /// </summary>
235         /// <param name="links">
236         /// <para>The links.</para>
237         /// <para></para>
238         /// </param>
239         /// <exception cref="InvalidOperationException">
240         /// <para>Cannot recreate sequence marker link.</para>
241         /// <para></para>
242         /// </exception>
243         [MethodImpl(MethodImplOptions.AggressiveInlining)]
244         public void InitOptions(ISynchronizedLinks<TLinkAddress> links)
245         {
246             if (UseSequenceMarker)
247             {
248                 if (_equalityComparer.Equals(SequenceMarkerLink, links.Constants.Null))
249                 {
250                     SequenceMarkerLink = links.CreatePoint();
251                 }
252                 else
253                 {
254                     if (!links.Exists(SequenceMarkerLink))
255                     {
256                         var link = links.CreatePoint();
257                         if (!_equalityComparer.Equals(link, SequenceMarkerLink))
258                         {
259                             throw new InvalidOperationException("Cannot recreate sequence marker
                                 ↪  link.");
260                         }
261                     }
262                 }
263                 if (MarkedSequenceMatcher == null)
264                 {
265                     MarkedSequenceMatcher = new
                         ↪  MarkedSequenceCriterionMatcher<TLinkAddress>(links, SequenceMarkerLink);
266                 }
267             }
268             var balancedVariantConverter = new BalancedVariantConverter<TLinkAddress>(links);
269             if (UseCompression)
270             {
271                 if (LinksToSequenceConverter == null)
272                 {
273                     ICounter<TLinkAddress, TLinkAddress> totalSequenceSymbolFrequencyCounter;
274                     if (UseSequenceMarker)
275                     {
276                         totalSequenceSymbolFrequencyCounter = new
                             ↪  TotalMarkedSequenceSymbolFrequencyCounter<TLinkAddress>(links,
                             ↪  MarkedSequenceMatcher);
277                     }
278                     else
279                     {
280                         totalSequenceSymbolFrequencyCounter = new
                             ↪  TotalSequenceSymbolFrequencyCounter<TLinkAddress>(links);
```

```
                    }
                    var doubletFrequenciesCache = new LinkFrequenciesCache<TLinkAddress>(links, totalSequenceSymbolFrequencyCounter);
                    var compressingConverter = new CompressingConverter<TLinkAddress>(links, balancedVariantConverter, doubletFrequenciesCache);
                    LinksToSequenceConverter = compressingConverter;
                }
            }
            else
            {
                if (LinksToSequenceConverter == null)
                {
                    LinksToSequenceConverter = balancedVariantConverter;
                }
            }
            if (UseIndex && Index == null)
            {
                Index = new SequenceIndex<TLinkAddress>(links);
            }
            if (Walker == null)
            {
                Walker = new RightSequenceWalker<TLinkAddress>(links, new DefaultStack<TLinkAddress>());
            }
        }

        /// <summary>
        /// <para>
        /// Validates the options.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <exception cref="NotSupportedException">
        /// <para>To use garbage collection UseSequenceMarker option must be on.</para>
        /// <para></para>
        /// </exception>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void ValidateOptions()
        {
            if (UseGarbageCollection && !UseSequenceMarker)
            {
                throw new NotSupportedException("To use garbage collection UseSequenceMarker option must be on.");
            }
        }
    }
}
```

1.49 ./csharp/Platform.Data.Doublets.Sequences/TestExtensions.cs

```
using System.Text;
using Platform.Reflection;

namespace Platform.Data.Doublets.Sequences;

public class TestExtensions
{
    public static string PrettifyBinary<T>  (string binaryRepresentation)
    {
        var bitsCount = NumericType<T>.BitsSize;
        var sb = new StringBuilder().Append('0', bitsCount - binaryRepresentation.Length).Append(binaryRepresentation).ToString();
        for (var i = 4; i < sb.Length; i += 5)
        {
            sb = sb.Insert(i, " ");
        }
        return sb.ToString();
    }
}
```

1.50 ./csharp/Platform.Data.Doublets.Sequences/Time/DateTimeToLongRawNumberSequenceConverter.cs

```
using System;
using System.Runtime.CompilerServices;
using Platform.Converters;

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

namespace Platform.Data.Doublets.Time
{
```

```
    /// <summary>
    /// <para>
    /// Represents the date time to long raw number sequence converter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="IConverter{DateTime, TLinkAddress}"/>
    public class DateTimeToLongRawNumberSequenceConverter<TLinkAddress> : IConverter<DateTime,
    ↪  TLinkAddress>
    {
        private readonly IConverter<long, TLinkAddress> _int64ToLongRawNumberConverter;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="DateTimeToLongRawNumberSequenceConverter"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="int64ToLongRawNumberConverter">
        /// <para>A int 64 to long raw number converter.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public DateTimeToLongRawNumberSequenceConverter(IConverter<long, TLinkAddress>
        ↪  int64ToLongRawNumberConverter) => _int64ToLongRawNumberConverter =
        ↪  int64ToLongRawNumberConverter;

        /// <summary>
        /// <para>
        /// Converts the source.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="source">
        /// <para>The source.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLinkAddress Convert(DateTime source) =>
        ↪  _int64ToLongRawNumberConverter.Convert(source.ToFileTimeUtc());
    }
}
```

1.51 ./csharp/Platform.Data.Doublets.Sequences/Time/LongRawNumberSequenceToDateTimeConverter.cs

```
using System;
using System.Runtime.CompilerServices;
using Platform.Converters;

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

namespace Platform.Data.Doublets.Time
{
    /// <summary>
    /// <para>
    /// Represents the long raw number sequence to date time converter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="IConverter{TLinkAddress, DateTime}"/>
    public class LongRawNumberSequenceToDateTimeConverter<TLinkAddress> :
    ↪  IConverter<TLinkAddress, DateTime>
    {
        private readonly IConverter<TLinkAddress, long> _longRawNumberConverterToInt64;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="LongRawNumberSequenceToDateTimeConverter"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="longRawNumberConverterToInt64">
        /// <para>A long raw number converter to int 64.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
31          public LongRawNumberSequenceToDateTimeConverter(IConverter<TLinkAddress, long>
            ↪  longRawNumberConverterToInt64) => _longRawNumberConverterToInt64 =
            ↪  longRawNumberConverterToInt64;
32
33          /// <summary>
34          /// <para>
35          /// Converts the source.
36          /// </para>
37          /// <para></para>
38          /// </summary>
39          /// <param name="source">
40          /// <para>The source.</para>
41          /// <para></para>
42          /// </param>
43          /// <returns>
44          /// <para>The date time</para>
45          /// <para></para>
46          /// </returns>
47          [MethodImpl(MethodImplOptions.AggressiveInlining)]
48          public DateTime Convert(TLinkAddress source) =>
            ↪  DateTime.FromFileTimeUtc(_longRawNumberConverterToInt64.Convert(source));
49      }
50  }
```

1.52 ./csharp/Platform.Data.Doublets.Sequences/UInt64LinksExtensions.cs

```
1   using System;
2   using System.Text;
3   using System.Collections.Generic;
4   using System.Runtime.CompilerServices;
5   using Platform.Singletons;
6   using Platform.Data.Doublets.Unicode;
7
8   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10  namespace Platform.Data.Doublets
11  {
12      /// <summary>
13      /// <para>
14      /// Represents the int 64 links extensions.
15      /// </para>
16      /// <para></para>
17      /// </summary>
18      public static class UInt64LinksExtensions
19      {
20          /// <summary>
21          /// <para>
22          /// Uses the unicode using the specified links.
23          /// </para>
24          /// <para></para>
25          /// </summary>
26          /// <param name="links">
27          /// <para>The links.</para>
28          /// <para></para>
29          /// </param>
30          [MethodImpl(MethodImplOptions.AggressiveInlining)]
31          public static void UseUnicode(this ILinks<ulong> links) => UnicodeMap.InitNew(links);
32      }
33  }
```

1.53 ./csharp/Platform.Data.Doublets.Sequences/Unicode/CharToUnicodeSymbolConverter.cs

```
1   using System.Runtime.CompilerServices;
2   using Platform.Converters;
3
4   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6   namespace Platform.Data.Doublets.Unicode
7   {
8       /// <summary>
9       /// <para>
10      /// Represents the char to unicode symbol converter.
11      /// </para>
12      /// <para></para>
13      /// </summary>
14      /// <seealso cref="LinksOperatorBase{TLinkAddress}"/>
15      /// <seealso cref="IConverter{char, TLinkAddress}"/>
16      public class CharToUnicodeSymbolConverter<TLinkAddress> : LinksOperatorBase<TLinkAddress>,
        ↪  IConverter<char, TLinkAddress>
17      {
```

```
        private static readonly UncheckedConverter<char, TLinkAddress> _charToAddressConverter =
        ↪  UncheckedConverter<char, TLinkAddress>.Default;
        private readonly IConverter<TLinkAddress> _addressToNumberConverter;
        private readonly TLinkAddress _unicodeSymbolMarker;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="CharToUnicodeSymbolConverter"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="addressToNumberConverter">
        /// <para>A address to number converter.</para>
        /// <para></para>
        /// </param>
        /// <param name="unicodeSymbolMarker">
        /// <para>A unicode symbol marker.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public CharToUnicodeSymbolConverter(ILinks<TLinkAddress> links, IConverter<TLinkAddress>
        ↪  addressToNumberConverter, TLinkAddress unicodeSymbolMarker) : base(links)
        {
            _addressToNumberConverter = addressToNumberConverter;
            _unicodeSymbolMarker = unicodeSymbolMarker;
        }

        /// <summary>
        /// <para>
        /// Converts the source.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="source">
        /// <para>The source.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLinkAddress Convert(char source)
        {
            var unaryNumber =
            ↪  _addressToNumberConverter.Convert(_charToAddressConverter.Convert(source));
            return _links.GetOrCreate(unaryNumber, _unicodeSymbolMarker);
        }
    }
}
```

1.54 ./csharp/Platform.Data.Doublets.Sequences/Unicode/StringToUnicodeSequenceConverter.cs

```
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Converters;
using Platform.Data.Doublets.Sequences.Indexes;

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

namespace Platform.Data.Doublets.Unicode
{
    /// <summary>
    /// <para>
    /// Represents the string to unicode sequence converter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksOperatorBase{TLinkAddress}"/>
    /// <seealso cref="IConverter{string, TLinkAddress}"/>
    public class StringToUnicodeSequenceConverter<TLinkAddress> :
    ↪  LinksOperatorBase<TLinkAddress>, IConverter<string, TLinkAddress>
    {
        private readonly IConverter<string, IList<TLinkAddress>?>
        ↪  _stringToUnicodeSymbolListConverter;
```

```
21          private readonly IConverter<IList<TLinkAddress>, TLinkAddress>
            ↪  _unicodeSymbolListToSequenceConverter;
22
23          /// <summary>
24          /// <para>
25          /// Initializes a new <see cref="StringToUnicodeSequenceConverter"/> instance.
26          /// </para>
27          /// <para></para>
28          /// </summary>
29          /// <param name="links">
30          /// <para>A links.</para>
31          /// <para></para>
32          /// </param>
33          /// <param name="stringToUnicodeSymbolListConverter">
34          /// <para>A string to unicode symbol list converter.</para>
35          /// <para></para>
36          /// </param>
37          /// <param name="unicodeSymbolListToSequenceConverter">
38          /// <para>A unicode symbol list to sequence converter.</para>
39          /// <para></para>
40          /// </param>
41          [MethodImpl(MethodImplOptions.AggressiveInlining)]
42          public StringToUnicodeSequenceConverter(ILinks<TLinkAddress> links, IConverter<string,
            ↪  IList<TLinkAddress>?> stringToUnicodeSymbolListConverter,
            ↪  IConverter<IList<TLinkAddress>, TLinkAddress> unicodeSymbolListToSequenceConverter)
            ↪  : base(links)
43          {
44              _stringToUnicodeSymbolListConverter = stringToUnicodeSymbolListConverter;
45              _unicodeSymbolListToSequenceConverter = unicodeSymbolListToSequenceConverter;
46          }
47
48          /// <summary>
49          /// <para>
50          /// Initializes a new <see cref="StringToUnicodeSequenceConverter"/> instance.
51          /// </para>
52          /// <para></para>
53          /// </summary>
54          /// <param name="links">
55          /// <para>A links.</para>
56          /// <para></para>
57          /// </param>
58          /// <param name="stringToUnicodeSymbolListConverter">
59          /// <para>A string to unicode symbol list converter.</para>
60          /// <para></para>
61          /// </param>
62          /// <param name="index">
63          /// <para>A index.</para>
64          /// <para></para>
65          /// </param>
66          /// <param name="listToSequenceLinkConverter">
67          /// <para>A list to sequence link converter.</para>
68          /// <para></para>
69          /// </param>
70          /// <param name="unicodeSequenceMarker">
71          /// <para>A unicode sequence marker.</para>
72          /// <para></para>
73          /// </param>
74          [MethodImpl(MethodImplOptions.AggressiveInlining)]
75          public StringToUnicodeSequenceConverter(ILinks<TLinkAddress> links, IConverter<string,
            ↪  IList<TLinkAddress>?> stringToUnicodeSymbolListConverter,
            ↪  ISequenceIndex<TLinkAddress> index, IConverter<IList<TLinkAddress>, TLinkAddress>
            ↪  listToSequenceLinkConverter, TLinkAddress unicodeSequenceMarker)
76              : this(links, stringToUnicodeSymbolListConverter, new
                ↪  UnicodeSymbolsListToUnicodeSequenceConverter<TLinkAddress>(links, index,
                ↪  listToSequenceLinkConverter, unicodeSequenceMarker)) { }
77
78          /// <summary>
79          /// <para>
80          /// Initializes a new <see cref="StringToUnicodeSequenceConverter"/> instance.
81          /// </para>
82          /// <para></para>
83          /// </summary>
84          /// <param name="links">
85          /// <para>A links.</para>
86          /// <para></para>
87          /// </param>
88          /// <param name="charToUnicodeSymbolConverter">
89          /// <para>A char to unicode symbol converter.</para>
```

```
90          /// <para></para>
91          /// </param>
92          /// <param name="index">
93          /// <para>A index.</para>
94          /// <para></para>
95          /// </param>
96          /// <param name="listToSequenceLinkConverter">
97          /// <para>A list to sequence link converter.</para>
98          /// <para></para>
99          /// </param>
100         /// <param name="unicodeSequenceMarker">
101         /// <para>A unicode sequence marker.</para>
102         /// <para></para>
103         /// </param>
104         [MethodImpl(MethodImplOptions.AggressiveInlining)]
105         public StringToUnicodeSequenceConverter(ILinks<TLinkAddress> links, IConverter<char,
            ↪ TLinkAddress> charToUnicodeSymbolConverter, ISequenceIndex<TLinkAddress> index,
            ↪ IConverter<IList<TLinkAddress>, TLinkAddress> listToSequenceLinkConverter,
            ↪ TLinkAddress unicodeSequenceMarker)
106             : this(links, new
                ↪ StringToUnicodeSymbolsListConverter<TLinkAddress>(charToUnicodeSymbolConverter),
                ↪ index, listToSequenceLinkConverter, unicodeSequenceMarker) { }
107
108         /// <summary>
109         /// <para>
110         /// Initializes a new <see cref="StringToUnicodeSequenceConverter"/> instance.
111         /// </para>
112         /// <para></para>
113         /// </summary>
114         /// <param name="links">
115         /// <para>A links.</para>
116         /// <para></para>
117         /// </param>
118         /// <param name="charToUnicodeSymbolConverter">
119         /// <para>A char to unicode symbol converter.</para>
120         /// <para></para>
121         /// </param>
122         /// <param name="listToSequenceLinkConverter">
123         /// <para>A list to sequence link converter.</para>
124         /// <para></para>
125         /// </param>
126         /// <param name="unicodeSequenceMarker">
127         /// <para>A unicode sequence marker.</para>
128         /// <para></para>
129         /// </param>
130         [MethodImpl(MethodImplOptions.AggressiveInlining)]
131         public StringToUnicodeSequenceConverter(ILinks<TLinkAddress> links, IConverter<char,
            ↪ TLinkAddress> charToUnicodeSymbolConverter, IConverter<IList<TLinkAddress>,
            ↪ TLinkAddress> listToSequenceLinkConverter, TLinkAddress unicodeSequenceMarker)
132             : this(links, charToUnicodeSymbolConverter, new Unindex<TLinkAddress>(),
                ↪ listToSequenceLinkConverter, unicodeSequenceMarker) { }
133
134         /// <summary>
135         /// <para>
136         /// Initializes a new <see cref="StringToUnicodeSequenceConverter"/> instance.
137         /// </para>
138         /// <para></para>
139         /// </summary>
140         /// <param name="links">
141         /// <para>A links.</para>
142         /// <para></para>
143         /// </param>
144         /// <param name="stringToUnicodeSymbolListConverter">
145         /// <para>A string to unicode symbol list converter.</para>
146         /// <para></para>
147         /// </param>
148         /// <param name="listToSequenceLinkConverter">
149         /// <para>A list to sequence link converter.</para>
150         /// <para></para>
151         /// </param>
152         /// <param name="unicodeSequenceMarker">
153         /// <para>A unicode sequence marker.</para>
154         /// <para></para>
155         /// </param>
156         [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
157         public StringToUnicodeSequenceConverter(ILinks<TLinkAddress> links, IConverter<string,
            ↪  IList<TLinkAddress>?> stringToUnicodeSymbolListConverter,
            ↪  IConverter<IList<TLinkAddress>, TLinkAddress> listToSequenceLinkConverter,
            ↪  TLinkAddress unicodeSequenceMarker)
158             : this(links, stringToUnicodeSymbolListConverter, new Unindex<TLinkAddress>(),
                ↪  listToSequenceLinkConverter, unicodeSequenceMarker) { }
159 
160         /// <summary>
161         /// <para>
162         /// Converts the source.
163         /// </para>
164         /// <para></para>
165         /// </summary>
166         /// <param name="source">
167         /// <para>The source.</para>
168         /// <para></para>
169         /// </param>
170         /// <returns>
171         /// <para>The link</para>
172         /// <para></para>
173         /// </returns>
174         [MethodImpl(MethodImplOptions.AggressiveInlining)]
175         public TLinkAddress Convert(string source)
176         {
177             var elements = _stringToUnicodeSymbolListConverter.Convert(source);
178             return _unicodeSymbolListToSequenceConverter.Convert(elements);
179         }
180     }
181 }
```

1.55 ./csharp/Platform.Data.Doublets.Sequences/Unicode/StringToUnicodeSymbolsListConverter.cs

```
1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using Platform.Converters;
4 
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6 
7  namespace Platform.Data.Doublets.Unicode
8  {
9      /// <summary>
10     /// <para>
11     /// Represents the string to unicode symbols list converter.
12     /// </para>
13     /// <para></para>
14     /// </summary>
15     /// <seealso cref="IConverter{string, IList{TLinkAddress}}"/>
16     public class StringToUnicodeSymbolsListConverter<TLinkAddress> : IConverter<string,
       ↪  IList<TLinkAddress>?>
17     {
18         private readonly IConverter<char, TLinkAddress> _charToUnicodeSymbolConverter;
19 
20         /// <summary>
21         /// <para>
22         /// Initializes a new <see cref="StringToUnicodeSymbolsListConverter"/> instance.
23         /// </para>
24         /// <para></para>
25         /// </summary>
26         /// <param name="charToUnicodeSymbolConverter">
27         /// <para>A char to unicode symbol converter.</para>
28         /// <para></para>
29         /// </param>
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         public StringToUnicodeSymbolsListConverter(IConverter<char, TLinkAddress>
           ↪  charToUnicodeSymbolConverter) => _charToUnicodeSymbolConverter =
           ↪  charToUnicodeSymbolConverter;
32 
33         /// <summary>
34         /// <para>
35         /// Converts the source.
36         /// </para>
37         /// <para></para>
38         /// </summary>
39         /// <param name="source">
40         /// <para>The source.</para>
41         /// <para></para>
42         /// </param>
43         /// <returns>
44         /// <para>The elements.</para>
45         /// <para></para>
```

```
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public IList<TLinkAddress>? Convert(string source)
        {
            var elements = new TLinkAddress[source.Length];
            for (var i = 0; i < elements.Length; i++)
            {
                elements[i] = _charToUnicodeSymbolConverter.Convert(source[i]);
            }
            return elements;
        }
    }
}
```

1.56 ./csharp/Platform.Data.Doublets.Sequences/Unicode/UnicodeMap.cs

```
using System;
using System.Collections.Generic;
using System.Globalization;
using System.Runtime.CompilerServices;
using System.Text;
using Platform.Data.Sequences;

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

namespace Platform.Data.Doublets.Unicode
{
    /// <summary>
    /// <para>
    /// Represents the unicode map.
    /// </para>
    /// <para></para>
    /// </summary>
    public class UnicodeMap
    {
        /// <summary>
        /// <para>
        /// The first char link.
        /// </para>
        /// <para></para>
        /// </summary>
        public static readonly ulong FirstCharLink = 1;
        /// <summary>
        /// <para>
        /// The max value.
        /// </para>
        /// <para></para>
        /// </summary>
        public static readonly ulong LastCharLink = FirstCharLink + char.MaxValue;
        /// <summary>
        /// <para>
        /// The max value.
        /// </para>
        /// <para></para>
        /// </summary>
        public static readonly ulong MapSize = 1 + char.MaxValue;
        private readonly ILinks<ulong> _links;
        private bool _initialized;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="UnicodeMap"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UnicodeMap(ILinks<ulong> links) => _links = links;

        /// <summary>
        /// <para>
        /// Inits the new using the specified links.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>The links.</para>
```

```
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The map.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static UnicodeMap InitNew(ILinks<ulong> links)
        {
            var map = new UnicodeMap(links);
            map.Init();
            return map;
        }

        /// <summary>
        /// <para>
        /// Inits this instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <exception cref="InvalidOperationException">
        /// <para>Unable to initialize UTF 16 table.</para>
        /// <para></para>
        /// </exception>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public void Init()
        {
            if (_initialized)
            {
                return;
            }
            _initialized = true;
            var firstLink = _links.CreatePoint();
            if (firstLink != FirstCharLink)
            {
                _links.Delete(firstLink);
            }
            else
            {
                for (var i = FirstCharLink + 1; i <= LastCharLink; i++)
                {
                    // From NIL to It (NIL -> Character) transformation meaning, (or infinite
                    ↪  amount of NIL characters before actual Character)
                    var createdLink = _links.CreatePoint();
                    _links.Update(createdLink, firstLink, createdLink);
                    if (createdLink != i)
                    {
                        throw new InvalidOperationException("Unable to initialize UTF 16
                        ↪  table.");
                    }
                }
            }
        }

        // 0 - null link
        // 1 - nil character (0 character)
        // ...
        // 65536 (0(1) + 65535 = 65536 possible values)

        /// <summary>
        /// <para>
        /// Creates the char to link using the specified character.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="character">
        /// <para>The character.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The ulong</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static ulong FromCharToLink(char character) => (ulong)character + 1;

        /// <summary>
        /// <para>
```

```
        /// Creates the link to char using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The char</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static char FromLinkToChar(ulong link) => (char)(link - 1);

        /// <summary>
        /// <para>
        /// Determines whether is char link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static bool IsCharLink(ulong link) => link <= MapSize;

        /// <summary>
        /// <para>
        /// Creates the links to string using the specified links list.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="linksList">
        /// <para>The links list.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The string</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static string FromLinksToString(IList<ulong> linksList)
        {
            var sb = new StringBuilder();
            for (int i = 0; i < linksList.Count; i++)
            {
                sb.Append(FromLinkToChar(linksList[i]));
            }
            return sb.ToString();
        }

        /// <summary>
        /// <para>
        /// Creates the sequence link to string using the specified link.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="link">
        /// <para>The link.</para>
        /// <para></para>
        /// </param>
        /// <param name="links">
        /// <para>The links.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The string</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static string FromSequenceLinkToString(ulong link, ILinks<ulong> links)
        {
```

```
219             var sb = new StringBuilder();
220             if (links.Exists(link))
221             {
222                 StopableSequenceWalker.WalkRight(link, links.GetSource, links.GetTarget,
223                     x => x <= MapSize || links.GetSource(x) == x || links.GetTarget(x) == x,
                        ↪ element =>
224                     {
225                         sb.Append(FromLinkToChar(element));
226                         return true;
227                     });
228             }
229             return sb.ToString();
230         }
231 
232         /// <summary>
233         /// <para>
234         /// Creates the chars to link array using the specified chars.
235         /// </para>
236         /// <para></para>
237         /// </summary>
238         /// <param name="chars">
239         /// <para>The chars.</para>
240         /// <para></para>
241         /// </param>
242         /// <returns>
243         /// <para>The ulong array</para>
244         /// <para></para>
245         /// </returns>
246         [MethodImpl(MethodImplOptions.AggressiveInlining)]
247         public static ulong[] FromCharsToLinkArray(char[] chars) => FromCharsToLinkArray(chars,
            ↪ chars.Length);
248 
249         /// <summary>
250         /// <para>
251         /// Creates the chars to link array using the specified chars.
252         /// </para>
253         /// <para></para>
254         /// </summary>
255         /// <param name="chars">
256         /// <para>The chars.</para>
257         /// <para></para>
258         /// </param>
259         /// <param name="count">
260         /// <para>The count.</para>
261         /// <para></para>
262         /// </param>
263         /// <returns>
264         /// <para>The links sequence.</para>
265         /// <para></para>
266         /// </returns>
267         [MethodImpl(MethodImplOptions.AggressiveInlining)]
268         public static ulong[] FromCharsToLinkArray(char[] chars, int count)
269         {
270             // char array to ulong array
271             var linksSequence = new ulong[count];
272             for (var i = 0; i < count; i++)
273             {
274                 linksSequence[i] = FromCharToLink(chars[i]);
275             }
276             return linksSequence;
277         }
278 
279         /// <summary>
280         /// <para>
281         /// Creates the string to link array using the specified sequence.
282         /// </para>
283         /// <para></para>
284         /// </summary>
285         /// <param name="sequence">
286         /// <para>The sequence.</para>
287         /// <para></para>
288         /// </param>
289         /// <returns>
290         /// <para>The links sequence.</para>
291         /// <para></para>
292         /// </returns>
293         [MethodImpl(MethodImplOptions.AggressiveInlining)]
294         public static ulong[] FromStringToLinkArray(string sequence)
```

```
        {
            // char array to ulong array
            var linksSequence = new ulong[sequence.Length];
            for (var i = 0; i < sequence.Length; i++)
            {
                linksSequence[i] = FromCharToLink(sequence[i]);
            }
            return linksSequence;
        }

        /// <summary>
        /// <para>
        /// Creates the string to link array groups using the specified sequence.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The result.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static List<ulong[]> FromStringToLinkArrayGroups(string sequence)
        {
            var result = new List<ulong[]>();
            var offset = 0;
            while (offset < sequence.Length)
            {
                var currentCategory = CharUnicodeInfo.GetUnicodeCategory(sequence[offset]);
                var relativeLength = 1;
                var absoluteLength = offset + relativeLength;
                while (absoluteLength < sequence.Length &&
                       currentCategory ==
                       ↪ CharUnicodeInfo.GetUnicodeCategory(sequence[absoluteLength]))
                {
                    relativeLength++;
                    absoluteLength++;
                }
                // char array to ulong array
                var innerSequence = new ulong[relativeLength];
                var maxLength = offset + relativeLength;
                for (var i = offset; i < maxLength; i++)
                {
                    innerSequence[i - offset] = FromCharToLink(sequence[i]);
                }
                result.Add(innerSequence);
                offset += relativeLength;
            }
            return result;
        }

        /// <summary>
        /// <para>
        /// Creates the link array to link array groups using the specified array.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="array">
        /// <para>The array.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The result.</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public static List<ulong[]> FromLinkArrayToLinkArrayGroups(ulong[] array)
        {
            var result = new List<ulong[]>();
            var offset = 0;
            while (offset < array.Length)
            {
                var relativeLength = 1;
                if (array[offset] <= LastCharLink)
                {
```

```
                    var currentCategory =
                    ↪  CharUnicodeInfo.GetUnicodeCategory(FromLinkToChar(array[offset]));
                    var absoluteLength = offset + relativeLength;
                    while (absoluteLength < array.Length &&
                           array[absoluteLength] <= LastCharLink &&
                           currentCategory == CharUnicodeInfo.GetUnicodeCategory(FromLinkToChar(
                           ↪  array[absoluteLength])))
                    {
                        relativeLength++;
                        absoluteLength++;
                    }
                }
                else
                {
                    var absoluteLength = offset + relativeLength;
                    while (absoluteLength < array.Length && array[absoluteLength] > LastCharLink)
                    {
                        relativeLength++;
                        absoluteLength++;
                    }
                }
                // copy array
                var innerSequence = new ulong[relativeLength];
                var maxLength = offset + relativeLength;
                for (var i = offset; i < maxLength; i++)
                {
                    innerSequence[i - offset] = array[i];
                }
                result.Add(innerSequence);
                offset += relativeLength;
            }
            return result;
        }
    }
}
```

1.57 ./csharp/Platform.Data.Doublets.Sequences/Unicode/UnicodeSequenceToStringConverter.cs

```
using System;
using System.Collections.Generic;
using System.Runtime.CompilerServices;
using Platform.Interfaces;
using Platform.Converters;
using Platform.Data.Doublets.Sequences.Walkers;
using System.Text;
using Platform.Data.Doublets.Sequences.CriterionMatchers;

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

namespace Platform.Data.Doublets.Unicode
{
    /// <summary>
    /// <para>
    /// Represents the unicode sequence to string converter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksOperatorBase{TLinkAddress}"/>
    /// <seealso cref="IConverter{TLinkAddress, string}"/>
    public class UnicodeSequenceToStringConverter<TLinkAddress> :
    ↪  LinksOperatorBase<TLinkAddress>, IConverter<TLinkAddress, string>
    {
        private readonly ICriterionMatcher<TLinkAddress> _unicodeSequenceCriterionMatcher;
        private readonly ISequenceWalker<TLinkAddress> _sequenceWalker;
        private readonly IConverter<TLinkAddress, char> _unicodeSymbolToCharConverter;
        private readonly TLinkAddress _unicodeSequenceMarker;


        /// <summary>
        /// <para>
        /// Initializes a new <see cref="UnicodeSequenceToStringConverter"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="unicodeSequenceCriterionMatcher">
        /// <para>A unicode sequence criterion matcher.</para>
        /// <para></para>
```

```
        /// </param>
        /// <param name="sequenceWalker">
        /// <para>A sequence walker.</para>
        /// <para></para>
        /// </param>
        /// <param name="unicodeSymbolToCharConverter">
        /// <para>A unicode symbol to char converter.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UnicodeSequenceToStringConverter(ILinks<TLinkAddress> links,
        ↪  ICriterionMatcher<TLinkAddress> unicodeSequenceCriterionMatcher,
        ↪  ISequenceWalker<TLinkAddress> sequenceWalker, IConverter<TLinkAddress, char>
        ↪  unicodeSymbolToCharConverter, TLinkAddress unicodeSequenceMarker) : base(links)
        {
            _unicodeSequenceCriterionMatcher = unicodeSequenceCriterionMatcher;
            _sequenceWalker = sequenceWalker;
            _unicodeSymbolToCharConverter = unicodeSymbolToCharConverter;
            _unicodeSequenceMarker = unicodeSequenceMarker;
        }

        public UnicodeSequenceToStringConverter(ILinks<TLinkAddress> links,
        ↪  ISequenceWalker<TLinkAddress> sequenceWalker, IConverter<TLinkAddress, char>
        ↪  unicodeSymbolToCharConverter, TLinkAddress unicodeSequenceMarker): this(links, new
        ↪  UnicodeSequenceMatcher<TLinkAddress>(links, unicodeSequenceMarker), sequenceWalker,
        ↪  unicodeSymbolToCharConverter, unicodeSequenceMarker){}


            /// <summary>
        /// <para>
        /// Converts the source.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="source">
        /// <para>The source.</para>
        /// <para></para>
        /// </param>
        /// <exception cref="ArgumentOutOfRangeException">
        /// <para>Specified link is not a unicode sequence.</para>
        /// <para></para>
        /// </exception>
        /// <returns>
        /// <para>The string</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public string Convert(TLinkAddress source)
        {
            if (!_unicodeSequenceCriterionMatcher.IsMatched(source))
            {
                throw new ArgumentOutOfRangeException(nameof(source), source, "Specified link is
                ↪  not a unicode sequence.");
            }
            if(EqualityComparer<TLinkAddress>.Default.Equals(_unicodeSequenceMarker, source))
            {
                return String.Empty;
            }
            var sequence = _links.GetSource(source);
            var sb = new StringBuilder();
            foreach(var character in _sequenceWalker.Walk(sequence))
            {
                sb.Append(_unicodeSymbolToCharConverter.Convert(character));
            }
            return sb.ToString();
        }
    }
}
```

1.58 ./csharp/Platform.Data.Doublets.Sequences/Unicode/UnicodeSymbolToCharConverter.cs

```
using System;
using System.Runtime.CompilerServices;
using Platform.Interfaces;
using Platform.Converters;

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

namespace Platform.Data.Doublets.Unicode
{
```

```
    /// <summary>
    /// <para>
    /// Represents the unicode symbol to char converter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksOperatorBase{TLinkAddress}"/>
    /// <seealso cref="IConverter{TLinkAddress, char}"/>
    public class UnicodeSymbolToCharConverter<TLinkAddress> : LinksOperatorBase<TLinkAddress>, IConverter<TLinkAddress, char>
    {
        private static readonly UncheckedConverter<TLinkAddress, char> _addressToCharConverter = UncheckedConverter<TLinkAddress, char>.Default;
        private readonly IConverter<TLinkAddress> _numberToAddressConverter;
        private readonly ICriterionMatcher<TLinkAddress> _unicodeSymbolCriterionMatcher;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="UnicodeSymbolToCharConverter"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="numberToAddressConverter">
        /// <para>A number to address converter.</para>
        /// <para></para>
        /// </param>
        /// <param name="unicodeSymbolCriterionMatcher">
        /// <para>A unicode symbol criterion matcher.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UnicodeSymbolToCharConverter(ILinks<TLinkAddress> links, IConverter<TLinkAddress> numberToAddressConverter, ICriterionMatcher<TLinkAddress> unicodeSymbolCriterionMatcher) : base(links)
        {
            _numberToAddressConverter = numberToAddressConverter;
            _unicodeSymbolCriterionMatcher = unicodeSymbolCriterionMatcher;
        }

        /// <summary>
        /// <para>
        /// Converts the source.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="source">
        /// <para>The source.</para>
        /// <para></para>
        /// </param>
        /// <exception cref="ArgumentOutOfRangeException">
        /// <para>Specified link is not a unicode symbol.</para>
        /// <para></para>
        /// </exception>
        /// <returns>
        /// <para>The char</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public char Convert(TLinkAddress source)
        {
            if (!_unicodeSymbolCriterionMatcher.IsMatched(source))
            {
                throw new ArgumentOutOfRangeException(nameof(source), source, "Specified link is not a unicode symbol.");
            }
            return _addressToCharConverter.Convert(_numberToAddressConverter.Convert(_links.GetSource(source)));
        }
    }
}
```

1.59 ./csharp/Platform.Data.Doublets.Sequences/Unicode/UnicodeSymbolsListToUnicodeSequenceConverter.cs

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

```
using Platform.Collections;
using Platform.Converters;
using Platform.Data.Doublets.Sequences.Indexes;

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

namespace Platform.Data.Doublets.Unicode
{
    /// <summary>
    /// <para>
    /// Represents the unicode symbols list to unicode sequence converter.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksOperatorBase{TLinkAddress}"/>
    /// <seealso cref="IConverter{IList{TLinkAddress}, TLinkAddress}"/>
    public class UnicodeSymbolsListToUnicodeSequenceConverter<TLinkAddress> :
    ↪ LinksOperatorBase<TLinkAddress>, IConverter<IList<TLinkAddress>, TLinkAddress>
    {
        private readonly ISequenceIndex<TLinkAddress> _index;
        private readonly IConverter<IList<TLinkAddress>, TLinkAddress>
        ↪ _listToSequenceLinkConverter;
        private readonly TLinkAddress _unicodeSequenceMarker;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="UnicodeSymbolsListToUnicodeSequenceConverter"/>
        ↪ instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="index">
        /// <para>A index.</para>
        /// <para></para>
        /// </param>
        /// <param name="listToSequenceLinkConverter">
        /// <para>A list to sequence link converter.</para>
        /// <para></para>
        /// </param>
        /// <param name="unicodeSequenceMarker">
        /// <para>A unicode sequence marker.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public UnicodeSymbolsListToUnicodeSequenceConverter(ILinks<TLinkAddress> links,
        ↪ ISequenceIndex<TLinkAddress> index, IConverter<IList<TLinkAddress>, TLinkAddress>
        ↪ listToSequenceLinkConverter, TLinkAddress unicodeSequenceMarker) : base(links)
        {
            _index = index;
            _listToSequenceLinkConverter = listToSequenceLinkConverter;
            _unicodeSequenceMarker = unicodeSequenceMarker;
        }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="UnicodeSymbolsListToUnicodeSequenceConverter"/>
        ↪ instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="listToSequenceLinkConverter">
        /// <para>A list to sequence link converter.</para>
        /// <para></para>
        /// </param>
        /// <param name="unicodeSequenceMarker">
        /// <para>A unicode sequence marker.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
```

```
74          public UnicodeSymbolsListToUnicodeSequenceConverter(ILinks<TLinkAddress> links,
            ↪  IConverter<IList<TLinkAddress>, TLinkAddress> listToSequenceLinkConverter,
            ↪  TLinkAddress unicodeSequenceMarker)
75              : this(links, new Unindex<TLinkAddress>(), listToSequenceLinkConverter,
                ↪  unicodeSequenceMarker) { }
76
77          /// <summary>
78          /// <para>
79          /// Converts the list.
80          /// </para>
81          /// <para></para>
82          /// </summary>
83          /// <param name="list">
84          /// <para>The list.</para>
85          /// <para></para>
86          /// </param>
87          /// <returns>
88          /// <para>The link</para>
89          /// <para></para>
90          /// </returns>
91          [MethodImpl(MethodImplOptions.AggressiveInlining)]
92          public TLinkAddress Convert(IList<TLinkAddress>? list)
93          {
94              if (list.IsNullOrEmpty())
95              {
96                  return _unicodeSequenceMarker;
97              }
98              _index.Add(list);
99              var sequence = _listToSequenceLinkConverter.Convert(list);
100             return _links.GetOrCreate(sequence, _unicodeSequenceMarker);
101         }
102     }
103 }
```

1.60 ./csharp/Platform.Data.Doublets.Sequences/Walkers/ISequenceWalker.cs

```
1   using System.Collections.Generic;
2   using System.Runtime.CompilerServices;
3
4   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6   namespace Platform.Data.Doublets.Sequences.Walkers
7   {
8       /// <summary>
9       /// <para>
10      /// Defines the sequence walker.
11      /// </para>
12      /// <para></para>
13      /// </summary>
14      public interface ISequenceWalker<TLinkAddress>
15      {
16          /// <summary>
17          /// <para>
18          /// Walks the sequence.
19          /// </para>
20          /// <para></para>
21          /// </summary>
22          /// <param name="sequence">
23          /// <para>The sequence.</para>
24          /// <para></para>
25          /// </param>
26          /// <returns>
27          /// <para>An enumerable of t link</para>
28          /// <para></para>
29          /// </returns>
30          [MethodImpl(MethodImplOptions.AggressiveInlining)]
31          IEnumerable<TLinkAddress> Walk(TLinkAddress sequence);
32      }
33  }
```

1.61 ./csharp/Platform.Data.Doublets.Sequences/Walkers/LeftSequenceWalker.cs

```
1   using System;
2   using System.Collections.Generic;
3   using System.Runtime.CompilerServices;
4   using Platform.Collections.Stacks;
5
6   #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8   namespace Platform.Data.Doublets.Sequences.Walkers
```

```
{
    /// <summary>
    /// <para>
    /// Represents the left sequence walker.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="SequenceWalkerBase{TLinkAddress}"/>
    public class LeftSequenceWalker<TLinkAddress> : SequenceWalkerBase<TLinkAddress>
    {
        /// <summary>
        /// <para>
        /// Initializes a new <see cref="LeftSequenceWalker"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="stack">
        /// <para>A stack.</para>
        /// <para></para>
        /// </param>
        /// <param name="isElement">
        /// <para>A is element.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LeftSequenceWalker(ILinks<TLinkAddress> links, IStack<TLinkAddress> stack,
        ↪  Func<TLinkAddress, bool> isElement) : base(links, stack, isElement) { }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="LeftSequenceWalker"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="stack">
        /// <para>A stack.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LeftSequenceWalker(ILinks<TLinkAddress> links, IStack<TLinkAddress> stack) :
        ↪  base(links, stack, links.IsPartialPoint) { }

        /// <summary>
        /// <para>
        /// Gets the next element after pop using the specified element.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="element">
        /// <para>The element.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress GetNextElementAfterPop(TLinkAddress element) =>
        ↪  _links.GetSource(element);

        /// <summary>
        /// <para>
        /// Gets the next element after push using the specified element.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="element">
        /// <para>The element.</para>
        /// <para></para>
        /// </param>
```

```
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress GetNextElementAfterPush(TLinkAddress element) =>
        ↪ _links.GetTarget(element);

        /// <summary>
        /// <para>
        /// Walks the contents using the specified element.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="element">
        /// <para>The element.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>An enumerable of t link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override IEnumerable<TLinkAddress> WalkContents(TLinkAddress element)
        {
            var links = _links;
            var parts = links.GetLink(element);
            var start = links.Constants.SourcePart;
            for (var i = parts.Count - 1; i >= start; i--)
            {
                var part = parts[i];
                if (IsElement(part))
                {
                    yield return part;
                }
            }
        }
    }
}
```

1.62 ./csharp/Platform.Data.Doublets.Sequences/Walkers/LeveledSequenceWalker.cs

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

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

//#define USEARRAYPOOL
#if USEARRAYPOOL
using Platform.Collections;
#endif

namespace Platform.Data.Doublets.Sequences.Walkers
{
    /// <summary>
    /// <para>
    /// Represents the leveled sequence walker.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksOperatorBase{TLinkAddress}"/>
    /// <seealso cref="ISequenceWalker{TLinkAddress}"/>
    public class LeveledSequenceWalker<TLinkAddress> : LinksOperatorBase<TLinkAddress>,
    ↪ ISequenceWalker<TLinkAddress>
    {
        private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
        ↪ EqualityComparer<TLinkAddress>.Default;
        private readonly Func<TLinkAddress, bool> _isElement;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="LeveledSequenceWalker"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
```

```
        /// <param name="isElement">
        /// <para>A is element.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LeveledSequenceWalker(ILinks<TLinkAddress> links, Func<TLinkAddress, bool>
        ↪  isElement) : base(links) => _isElement = isElement;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="LeveledSequenceWalker"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public LeveledSequenceWalker(ILinks<TLinkAddress> links) : base(links) => _isElement =
        ↪  _links.IsPartialPoint;

        /// <summary>
        /// <para>
        /// Walks the sequence.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>An enumerable of t link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public IEnumerable<TLinkAddress> Walk(TLinkAddress sequence) => ToArray(sequence);

        /// <summary>
        /// <para>
        /// Returns the array using the specified sequence.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link array</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public TLinkAddress[] ToArray(TLinkAddress sequence)
        {
            var length = 1;
            var array = new TLinkAddress[length];
            array[0] = sequence;
            if (_isElement(sequence))
            {
                return array;
            }
            bool hasElements;
            do
            {
                length *= 2;
#if USEARRAYPOOL
                var nextArray = ArrayPool.Allocate<ulong>(length);
#else
                var nextArray = new TLinkAddress[length];
#endif
                hasElements = false;
                for (var i = 0; i < array.Length; i++)
                {
                    var candidate = array[i];
                    if (_equalityComparer.Equals(array[i], default))
                    {
                        continue;
```

```
                    }
                    var doubletOffset = i * 2;
                    if (_isElement(candidate))
                    {
                        nextArray[doubletOffset] = candidate;
                    }
                    else
                    {
                        var links = _links;
                        var link = links.GetLink(candidate);
                        var linkSource = links.GetSource(link);
                        var linkTarget = links.GetTarget(link);
                        nextArray[doubletOffset] = linkSource;
                        nextArray[doubletOffset + 1] = linkTarget;
                        if (!hasElements)
                        {
                            hasElements = !(_isElement(linkSource) && _isElement(linkTarget));
                        }
                    }
                }
#if USEARRAYPOOL
                if (array.Length > 1)
                {
                    ArrayPool.Free(array);
                }
#endif
                array = nextArray;
            }
            while (hasElements);
            var filledElementsCount = CountFilledElements(array);
            if (filledElementsCount == array.Length)
            {
                return array;
            }
            else
            {
                return CopyFilledElements(array, filledElementsCount);
            }
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static TLinkAddress[] CopyFilledElements(TLinkAddress[] array, int
        ↪ filledElementsCount)
        {
            var finalArray = new TLinkAddress[filledElementsCount];
            for (int i = 0, j = 0; i < array.Length; i++)
            {
                if (!_equalityComparer.Equals(array[i], default))
                {
                    finalArray[j] = array[i];
                    j++;
                }
            }
#if USEARRAYPOOL
            ArrayPool.Free(array);
#endif
            return finalArray;
        }
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        private static int CountFilledElements(TLinkAddress[] array)
        {
            var count = 0;
            for (var i = 0; i < array.Length; i++)
            {
                if (!_equalityComparer.Equals(array[i], default))
                {
                    count++;
                }
            }
            return count;
        }
    }
}
```

1.63 ./csharp/Platform.Data.Doublets.Sequences/Walkers/RightSequenceWalker.cs

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

```

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

namespace Platform.Data.Doublets.Sequences.Walkers
{
    /// <summary>
    /// <para>
    /// Represents the right sequence walker.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="SequenceWalkerBase{TLinkAddress}"/>
    public class RightSequenceWalker<TLinkAddress> : SequenceWalkerBase<TLinkAddress>
    {
        /// <summary>
        /// <para>
        /// Initializes a new <see cref="RightSequenceWalker"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="stack">
        /// <para>A stack.</para>
        /// <para></para>
        /// </param>
        /// <param name="isElement">
        /// <para>A is element.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public RightSequenceWalker(ILinks<TLinkAddress> links, IStack<TLinkAddress> stack,
        ↪  Func<TLinkAddress, bool> isElement) : base(links, stack, isElement) { }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="RightSequenceWalker"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="stack">
        /// <para>A stack.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public RightSequenceWalker(ILinks<TLinkAddress> links, IStack<TLinkAddress> stack) :
        ↪  base(links, stack, links.IsPartialPoint) { }

        /// <summary>
        /// <para>
        /// Gets the next element after pop using the specified element.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="element">
        /// <para>The element.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress GetNextElementAfterPop(TLinkAddress element) =>
        ↪  _links.GetTarget(element);

        /// <summary>
        /// <para>
        /// Gets the next element after push using the specified element.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="element">
```

```
        /// <para>The element.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override TLinkAddress GetNextElementAfterPush(TLinkAddress element) =>
        ↪  _links.GetSource(element);

        /// <summary>
        /// <para>
        /// Walks the contents using the specified element.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="element">
        /// <para>The element.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>An enumerable of t link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected override IEnumerable<TLinkAddress> WalkContents(TLinkAddress element)
        {
            var parts = _links.GetLink(element);
            for (var i = _links.Constants.SourcePart; i < parts.Count; i++)
            {
                var part = parts[i];
                if (IsElement(part))
                {
                    yield return part;
                }
            }
        }
    }
}
```

1.64 ./csharp/Platform.Data.Doublets.Sequences/Walkers/SequenceWalkerBase.cs

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

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

namespace Platform.Data.Doublets.Sequences.Walkers
{
    /// <summary>
    /// <para>
    /// Represents the sequence walker base.
    /// </para>
    /// <para></para>
    /// </summary>
    /// <seealso cref="LinksOperatorBase{TLinkAddress}"/>
    /// <seealso cref="ISequenceWalker{TLinkAddress}"/>
    public abstract class SequenceWalkerBase<TLinkAddress> : LinksOperatorBase<TLinkAddress>,
    ↪  ISequenceWalker<TLinkAddress>
    {
        private readonly IStack<TLinkAddress> _stack;
        private readonly Func<TLinkAddress, bool> _isElement;

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="SequenceWalkerBase"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="stack">
        /// <para>A stack.</para>
        /// <para></para>
        /// </param>
```

```
        /// <param name="isElement">
        /// <para>A is element.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected SequenceWalkerBase(ILinks<TLinkAddress> links, IStack<TLinkAddress> stack,
        ↪  Func<TLinkAddress, bool> isElement) : base(links)
        {
            _stack = stack;
            _isElement = isElement;
        }

        /// <summary>
        /// <para>
        /// Initializes a new <see cref="SequenceWalkerBase"/> instance.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="links">
        /// <para>A links.</para>
        /// <para></para>
        /// </param>
        /// <param name="stack">
        /// <para>A stack.</para>
        /// <para></para>
        /// </param>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected SequenceWalkerBase(ILinks<TLinkAddress> links, IStack<TLinkAddress> stack) :
        ↪  this(links, stack, links.IsPartialPoint) { }

        /// <summary>
        /// <para>
        /// Walks the sequence.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="sequence">
        /// <para>The sequence.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>An enumerable of t link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        public IEnumerable<TLinkAddress> Walk(TLinkAddress sequence)
        {
            _stack.Clear();
            var element = sequence;
            if (IsElement(element))
            {
                yield return element;
            }
            else
            {
                while (true)
                {
                    if (IsElement(element))
                    {
                        if (_stack.IsEmpty)
                        {
                            break;
                        }
                        element = _stack.Pop();
                        foreach (var output in WalkContents(element))
                        {
                            yield return output;
                        }
                        element = GetNextElementAfterPop(element);
                    }
                    else
                    {
                        _stack.Push(element);
                        element = GetNextElementAfterPush(element);
                    }
                }
            }
        }
```

```

        /// <summary>
        /// <para>
        /// Determines whether this instance is element.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="elementLink">
        /// <para>The element link.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The bool</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected virtual bool IsElement(TLinkAddress elementLink) => _isElement(elementLink);

        /// <summary>
        /// <para>
        /// Gets the next element after pop using the specified element.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="element">
        /// <para>The element.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract TLinkAddress GetNextElementAfterPop(TLinkAddress element);

        /// <summary>
        /// <para>
        /// Gets the next element after push using the specified element.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="element">
        /// <para>The element.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>The link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract TLinkAddress GetNextElementAfterPush(TLinkAddress element);

        /// <summary>
        /// <para>
        /// Walks the contents using the specified element.
        /// </para>
        /// <para></para>
        /// </summary>
        /// <param name="element">
        /// <para>The element.</para>
        /// <para></para>
        /// </param>
        /// <returns>
        /// <para>An enumerable of t link</para>
        /// <para></para>
        /// </returns>
        [MethodImpl(MethodImplOptions.AggressiveInlining)]
        protected abstract IEnumerable<TLinkAddress> WalkContents(TLinkAddress element);
    }
}
```

1.65 ./csharp/Platform.Data.Doublets.Sequences.Tests/BigIntegerConvertersTests.cs

```
using System.Collections.Generic;
using System.Numerics;
using Platform.Data.Doublets.Memory;
using Platform.Data.Doublets.Memory.United.Generic;
using Platform.Data.Doublets.Numbers.Raw;
using Platform.Data.Doublets.Sequences.Converters;
```

```
using Platform.Data.Numbers.Raw;
using Platform.Memory;
using Xunit;
using TLinkAddress = System.UInt64;

namespace Platform.Data.Doublets.Sequences.Tests
{
    public class BigIntegerConvertersTests
    {
        public ILinks<TLinkAddress> CreateLinks() => CreateLinks<TLinkAddress>(new
        ↪  IO.TemporaryFile());

        public ILinks<TLinkAddress> CreateLinks<TLinkAddress>(string dataDbFilename)
        {
            var linksConstants = new
            ↪  LinksConstants<TLinkAddress>(enableExternalReferencesSupport: true);
            return new UnitedMemoryLinks<TLinkAddress>(new
            ↪  FileMappedResizableDirectMemory(dataDbFilename),
            ↪  UnitedMemoryLinks<TLinkAddress>.DefaultLinksSizeStep, linksConstants,
            ↪  IndexTreeType.Default);
        }

        [Fact]
        public void DecimalMaxValueTest()
        {
            var links = CreateLinks();
            BigInteger bigInteger = new(decimal.MaxValue);
            TLinkAddress negativeNumberMarker = links.Create();
            AddressToRawNumberConverter<TLinkAddress> addressToRawNumberConverter = new();
            RawNumberToAddressConverter<TLinkAddress> numberToAddressConverter = new();
            BalancedVariantConverter<TLinkAddress> listToSequenceConverter = new(links);
            BigIntegerToRawNumberSequenceConverter<TLinkAddress>
            ↪  bigIntegerToRawNumberSequenceConverter = new(links, addressToRawNumberConverter,
            ↪  listToSequenceConverter, negativeNumberMarker);
            RawNumberSequenceToBigIntegerConverter<TLinkAddress>
            ↪  rawNumberSequenceToBigIntegerConverter = new(links, numberToAddressConverter,
            ↪  negativeNumberMarker);
            var bigIntSequence = bigIntegerToRawNumberSequenceConverter.Convert(bigInteger);
            var bigIntFromSequence =
            ↪  rawNumberSequenceToBigIntegerConverter.Convert(bigIntSequence);
            Assert.Equal(bigInteger, bigIntFromSequence);
        }

        [Fact]
        public void DecimalMinValueTest()
        {
            var links = CreateLinks();
            BigInteger bigInteger = new(decimal.MinValue);
            TLinkAddress negativeNumberMarker = links.Create();
            AddressToRawNumberConverter<TLinkAddress> addressToRawNumberConverter = new();
            RawNumberToAddressConverter<TLinkAddress> numberToAddressConverter = new();
            BalancedVariantConverter<TLinkAddress> listToSequenceConverter = new(links);
            BigIntegerToRawNumberSequenceConverter<TLinkAddress>
            ↪  bigIntegerToRawNumberSequenceConverter = new(links, addressToRawNumberConverter,
            ↪  listToSequenceConverter, negativeNumberMarker);
            RawNumberSequenceToBigIntegerConverter<TLinkAddress>
            ↪  rawNumberSequenceToBigIntegerConverter = new(links, numberToAddressConverter,
            ↪  negativeNumberMarker);
            var bigIntSequence = bigIntegerToRawNumberSequenceConverter.Convert(bigInteger);
            var bigIntFromSequence =
            ↪  rawNumberSequenceToBigIntegerConverter.Convert(bigIntSequence);
            Assert.Equal(bigInteger, bigIntFromSequence);
        }

        [Fact]
        public void ZeroValueTest()
        {
            var links = CreateLinks();
            BigInteger bigInteger = new(0);
            TLinkAddress negativeNumberMarker = links.Create();
            AddressToRawNumberConverter<TLinkAddress> addressToRawNumberConverter = new();
            RawNumberToAddressConverter<TLinkAddress> numberToAddressConverter = new();
            BalancedVariantConverter<TLinkAddress> listToSequenceConverter = new(links);
            BigIntegerToRawNumberSequenceConverter<TLinkAddress>
            ↪  bigIntegerToRawNumberSequenceConverter = new(links, addressToRawNumberConverter,
            ↪  listToSequenceConverter, negativeNumberMarker);
```

```
            RawNumberSequenceToBigIntegerConverter<TLinkAddress>
            ↪ rawNumberSequenceToBigIntegerConverter = new(links, numberToAddressConverter,
            ↪ negativeNumberMarker);
            var bigIntSequence = bigIntegerToRawNumberSequenceConverter.Convert(bigInteger);
            var bigIntFromSequence =
            ↪ rawNumberSequenceToBigIntegerConverter.Convert(bigIntSequence);
            Assert.Equal(bigInteger, bigIntFromSequence);
        }

        [Fact]
        public void OneValueTest()
        {
            var links = CreateLinks();
            BigInteger bigInteger = new(1);
            TLinkAddress negativeNumberMarker = links.Create();
            AddressToRawNumberConverter<TLinkAddress> addressToRawNumberConverter = new();
            RawNumberToAddressConverter<TLinkAddress> numberToAddressConverter = new();
            BalancedVariantConverter<TLinkAddress> listToSequenceConverter = new(links);
            BigIntegerToRawNumberSequenceConverter<TLinkAddress>
            ↪ bigIntegerToRawNumberSequenceConverter = new(links, addressToRawNumberConverter,
            ↪ listToSequenceConverter, negativeNumberMarker);
            RawNumberSequenceToBigIntegerConverter<TLinkAddress>
            ↪ rawNumberSequenceToBigIntegerConverter = new(links, numberToAddressConverter,
            ↪ negativeNumberMarker);
            var bigIntSequence = bigIntegerToRawNumberSequenceConverter.Convert(bigInteger);
            var bigIntFromSequence =
            ↪ rawNumberSequenceToBigIntegerConverter.Convert(bigIntSequence);
            Assert.Equal(bigInteger, bigIntFromSequence);
        }
    }
}
```

1.66 ./csharp/Platform.Data.Doublets.Sequences.Tests/ByteConvertersTests.cs

```
using System;
using System.Collections;
using System.Collections.Generic;
using System.Linq;
using System.Numerics;
using System.Text;
using Platform.Collections.Lists;
using Platform.Collections.Stacks;
using Platform.Converters;
using Platform.Data.Doublets.CriterionMatchers;
using Platform.Data.Doublets.Memory;
using Platform.Data.Doublets.Memory.United.Generic;
using Platform.Data.Doublets.Numbers.Raw;
using Platform.Data.Doublets.Sequences.Converters;
using Platform.Data.Doublets.Sequences.Numbers.Byte;
using Platform.Data.Doublets.Sequences.Walkers;
using Platform.Data.Doublets.Unicode;
using Platform.Data.Numbers.Raw;
using Platform.Memory;
using Platform.Numbers;
using Platform.Reflection;
using Xunit;
using TLinkAddress = System.UInt16;

namespace Platform.Data.Doublets.Sequences.Tests
{
    public class ByteConvertersTests
    {
        private readonly ILinks<TLinkAddress> Storage;
        private static readonly AddressToRawNumberConverter<TLinkAddress>
        ↪ _addressToRawNumberConverter = new();
        private static readonly RawNumberToAddressConverter<TLinkAddress>
        ↪ _rawNumberToAddressConverter = new();
        private readonly BalancedVariantConverter<TLinkAddress> _listToSequenceConverter;
        private readonly ByteListToRawSequenceConverter<TLinkAddress>
        ↪ _byteListToRawSequenceConverter;
        private readonly RawSequenceToByteListConverter<TLinkAddress>
        ↪ _rawSequenceToByteListConverter;

        public ByteConvertersTests()
        {
            var linksConstants = new
            ↪ LinksConstants<TLinkAddress>(enableExternalReferencesSupport: true);
            var storageFileName = new IO.TemporaryFile().Filename;
            var storageMemory = new FileMappedResizableDirectMemory(storageFileName);
```

```
41              Storage = new UnitedMemoryLinks<TLinkAddress>(storageMemory,
                ↪  UnitedMemoryLinks<TLinkAddress>.DefaultLinksSizeStep, linksConstants,
                ↪  IndexTreeType.Default);
42              _listToSequenceConverter = new BalancedVariantConverter<TLinkAddress>(Storage);
43              TLinkAddress zero = default;
44              TLinkAddress one = Arithmetic.Increment(zero);
45              var type = Storage.GetOrCreate(one, one);
46              var typeIndex = type;
47              var unicodeSymbolType = Storage.GetOrCreate(type, Arithmetic.Increment(ref
                ↪  typeIndex));
48              var unicodeSequenceType = Storage.GetOrCreate(type, Arithmetic.Increment(ref
                ↪  typeIndex));
49              BalancedVariantConverter<TLinkAddress> balancedVariantConverter = new(Storage);
50              AddressToRawNumberConverter<TLinkAddress> addressToRawNumberConverter = new();
51              RawNumberToAddressConverter<TLinkAddress> rawNumberToAddressConverter = new();
52              TargetMatcher<TLinkAddress> unicodeSymbolCriterionMatcher = new(Storage,
                ↪  unicodeSymbolType);
53              TargetMatcher<TLinkAddress> unicodeSequenceCriterionMatcher = new(Storage,
                ↪  unicodeSequenceType);
54              CharToUnicodeSymbolConverter<TLinkAddress> charToUnicodeSymbolConverter =
55                  new(Storage, addressToRawNumberConverter, unicodeSymbolType);
56              UnicodeSymbolToCharConverter<TLinkAddress> unicodeSymbolToCharConverter =
57                  new(Storage, rawNumberToAddressConverter, unicodeSymbolCriterionMatcher);
58              var stringToUnicodeSequenceConverter = new
                ↪  StringToUnicodeSequenceConverter<TLinkAddress>(Storage,
                ↪  charToUnicodeSymbolConverter,
59                  balancedVariantConverter, unicodeSequenceType);
60              RightSequenceWalker<TLinkAddress> unicodeSymbolSequenceWalker = new(Storage, new
                ↪  DefaultStack<TLinkAddress>(), unicodeSymbolCriterionMatcher.IsMatched);
61              UnicodeSequenceToStringConverter<TLinkAddress> unicodeSequenceToStringConverter =
                ↪  new UnicodeSequenceToStringConverter<TLinkAddress>(Storage,
                ↪  unicodeSequenceCriterionMatcher, unicodeSymbolSequenceWalker,
                ↪  unicodeSymbolToCharConverter, unicodeSequenceType);
62              _byteListToRawSequenceConverter = new
                ↪  ByteListToRawSequenceConverter<TLinkAddress>(Storage,
                ↪  _addressToRawNumberConverter, _rawNumberToAddressConverter,
                ↪  _listToSequenceConverter, stringToUnicodeSequenceConverter);
63              _rawSequenceToByteListConverter = new
                ↪  RawSequenceToByteListConverter<TLinkAddress>(Storage,
                ↪  _rawNumberToAddressConverter, _listToSequenceConverter,
                ↪  stringToUnicodeSequenceConverter, unicodeSequenceToStringConverter);
64          }
65
66          private static byte[] GetRandomArray(int length)
67          {
68              byte[] array = new byte[length];
69              new System.Random(61267).NextBytes(array);
70              return array;
71          }
72
73          // [InlineData(new byte[]{})]
74          [InlineData(new byte[]{0})]
75          [InlineData(new byte[]{0, 0})]
76          [InlineData(new byte[]{0, 0, 0, 0})]
77          [InlineData(new byte[]{1})]
78          [InlineData(new byte[]{1, 1})]
79          [InlineData(new byte[]{1, 1, 1, 1})]
80          [InlineData(new byte[]{1, 1, 1, 1, 1, 1})]
81          [InlineData(new byte[]{1, 1, 1, 1, 1, 1, 1, 1, 1, 1})]
82          [InlineData(new byte[]{255, 255})]
83          [InlineData(new byte[]{255, 255, 255, 255, 255})]
84          [Theory]
85          public void FixedArraysTest(byte[] byteArray)
86          {
87              Test(byteArray);
88          }
89
90          [InlineData(1)]
91          [InlineData(2)]
92          [InlineData(3)]
93          [InlineData(4)]
94          [InlineData(5)]
95          [InlineData(6)]
96          [InlineData(7)]
97          [InlineData(8)]
98          [InlineData(9)]
99          [InlineData(10)]
100         [InlineData(11)]
```

```
        [InlineData(12)]
        [InlineData(13)]
        [InlineData(14)]
        [InlineData(15)]
        [InlineData(16)]
        [InlineData(17)]
        [InlineData(18)]
        [InlineData(19)]
        [InlineData(20)]
        [InlineData(21)]
        [InlineData(22)]
        [InlineData(23)]
        [InlineData(24)]
        [InlineData(25)]
        [InlineData(26)]
        [InlineData(27)]
        [InlineData(28)]
        [InlineData(29)]
        [InlineData(30)]
        [InlineData(31)]
        [InlineData(32)]
        [Theory]
        public void RandomArrayTest(int length)
        {
            var byteArray = GetRandomArray(length);
            Test(byteArray);
        }

        public void Test(byte[] byteArray)
        {
            var byteListRawSequence = _byteListToRawSequenceConverter.Convert(byteArray);
            Console.WriteLine();
            var byteListFromConverter =
            ↪  _rawSequenceToByteListConverter.Convert(byteListRawSequence);
            Console.WriteLine("Original");
            foreach (var b in byteArray)
            {
                Console.WriteLine(TestExtensions.PrettifyBinary<byte>(Convert.ToString(b, 2)));
            }
            Console.WriteLine();
            Console.WriteLine("From converter:");
            foreach (var b in byteListFromConverter)
            {
                Console.WriteLine(TestExtensions.PrettifyBinary<byte>(Convert.ToString(b, 2)));
            }
            Assert.Equal(byteArray, byteListFromConverter.ToArray());
        }


    }
}
```

1.67 ./csharp/Platform.Data.Doublets.Sequences.Tests/DefaultSequenceAppenderTests.cs

```
using System.Collections.Generic;
using Platform.Collections.Stacks;
using Platform.Data.Doublets.Memory;
using Platform.Data.Doublets.Memory.United.Generic;
using Platform.Data.Doublets.Sequences;
using Platform.Data.Doublets.Sequences.HeightProviders;
using Platform.Data.Numbers.Raw;
using Platform.Interfaces;
using Platform.Memory;
using Platform.Numbers;
using Xunit;
using Xunit.Abstractions;
using TLinkAddress = System.UInt64;

namespace Platform.Data.Doublets.Sequences.Tests
{
    public class DefaultSequenceAppenderTests
    {
        private readonly ITestOutputHelper _output;

        public DefaultSequenceAppenderTests(ITestOutputHelper output)
        {
            _output = output;
        }
        public static ILinks<TLinkAddress> CreateLinks() => CreateLinks<TLinkAddress>(new
        ↪  IO.TemporaryFile());

```

```
27          public static ILinks<TLinkAddress> CreateLinks<TLinkAddress>(string dataDBFilename)
28          {
29              var linksConstants = new
                ↪  LinksConstants<TLinkAddress>(enableExternalReferencesSupport: true);
30              return new UnitedMemoryLinks<TLinkAddress>(new
                ↪  FileMappedResizableDirectMemory(dataDBFilename),
                ↪  UnitedMemoryLinks<TLinkAddress>.DefaultLinksSizeStep, linksConstants,
                ↪  IndexTreeType.Default);
31          }
32
33          public class ValueCriterionMatcher<TLinkAddress> : ICriterionMatcher<TLinkAddress>
34          {
35              public readonly ILinks<TLinkAddress> Links;
36              public readonly TLinkAddress Marker;
37              public ValueCriterionMatcher(ILinks<TLinkAddress> links, TLinkAddress marker)
38              {
39                  Links = links;
40                  Marker = marker;
41              }
42
43              public bool IsMatched(TLinkAddress link) =>
                ↪  EqualityComparer<TLinkAddress>.Default.Equals(Links.GetSource(link), Marker);
44          }
45
46          [Fact]
47          public void AppendArrayBug()
48          {
49              ILinks<TLinkAddress> links = CreateLinks();
50              TLinkAddress zero = default;
51              var markerIndex = Arithmetic.Increment(zero);
52              var meaningRoot = links.GetOrCreate(markerIndex, markerIndex);
53              var sequence = links.Create();
54              sequence = links.Update(sequence, meaningRoot, sequence);
55              var appendant = links.Create();
56              appendant = links.Update(appendant, meaningRoot, appendant);
57              ValueCriterionMatcher<TLinkAddress> valueCriterionMatcher = new(links, meaningRoot);
58              DefaultSequenceRightHeightProvider<ulong> defaultSequenceRightHeightProvider =
                ↪  new(links, valueCriterionMatcher);
59              DefaultSequenceAppender<TLinkAddress> defaultSequenceAppender = new(links, new
                ↪  DefaultStack<ulong>(), defaultSequenceRightHeightProvider);
60              var newArray = defaultSequenceAppender.Append(sequence, appendant);
61              var output = links.FormatStructure(newArray, link => link.IsFullPoint(), true);
62              Assert.Equal("(4:(2:1 2) (3:1 3))", output);
63          }
64      }
65  }
```

1.68 ./csharp/Platform.Data.Doublets.Sequences.Tests/ILinksExtensionsTests.cs

```
1   // using Xunit;
2   //
3   // namespace Platform.Data.Doublets.Sequences.Tests
4   // {
5   //     public class ILinksExtensionsTests
6   //     {
7   //         [Fact]
8   //         public void FormatTest()
9   //         {
10  //             using (var scope = new TempLinksTestScope())
11  //             {
12  //                 var links = scope.Links;
13  //                 var link = links.Create();
14  //                 var linkString = links.Format(link);
15  //                 Assert.Equal("(1: 1 1)", linkString);
16  //             }
17  //         }
18  //     }
19  // }
```

1.69 ./csharp/Platform.Data.Doublets.Sequences.Tests/OptimalVariantSequenceTests.cs

```
1   // using System;
2   // using System.Linq;
3   // using Xunit;
4   // using Platform.Collections.Stacks;
5   // using Platform.Collections.Arrays;
6   // using Platform.Memory;
7   // using Platform.Data.Numbers.Raw;
8   // using Platform.Data.Doublets.Sequences;
9   // using Platform.Data.Doublets.Sequences.Frequencies.Cache;
```

```
// using Platform.Data.Doublets.Sequences.Frequencies.Counters;
// using Platform.Data.Doublets.Sequences.Converters;
// using Platform.Data.Doublets.PropertyOperators;
// using Platform.Data.Doublets.Incrementers;
// using Platform.Data.Doublets.Sequences.Walkers;
// using Platform.Data.Doublets.Sequences.Indexes;
// using Platform.Data.Doublets.Unicode;
// using Platform.Data.Doublets.Numbers.Unary;
// using Platform.Data.Doublets.Decorators;
// using Platform.Data.Doublets.Memory.United.Specific;
// using Platform.Data.Doublets.Memory;
//
// namespace Platform.Data.Doublets.Sequences.Tests
// {
//     public static class OptimalVariantSequenceTests
//     {
//         private static readonly string _sequenceExample = "зеленела зелёная зелень";
//         private static readonly string _loremIpsumExample = @"Lorem ipsum dolor sit amet,
↪  consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna
↪  aliqua.
// Facilisi nullam vehicula ipsum a arcu cursus vitae congue mauris.
// Et malesuada fames ac turpis egestas sed.
// Eget velit aliquet sagittis id consectetur purus.
// Dignissim cras tincidunt lobortis feugiat vivamus.
// Vitae aliquet nec ullamcorper sit.
// Lectus quam id leo in vitae.
// Tortor dignissim convallis aenean et tortor at risus viverra adipiscing.
// Sed risus ultricies tristique nulla aliquet enim tortor at auctor.
// Integer eget aliquet nibh praesent tristique.
// Vitae congue eu consequat ac felis donec et odio.
// Tristique et egestas quis ipsum suspendisse.
// Suspendisse potenti nullam ac tortor vitae purus faucibus ornare.
// Nulla facilisi etiam dignissim diam quis enim lobortis scelerisque.
// Imperdiet proin fermentum leo vel orci.
// In ante metus dictum at tempor commodo.
// Nisi lacus sed viverra tellus in.
// Quam vulputate dignissim suspendisse in.
// Elit scelerisque mauris pellentesque pulvinar pellentesque habitant morbi tristique senectus.
// Gravida cum sociis natoque penatibus et magnis dis parturient.
// Risus quis varius quam quisque id diam.
// Congue nisi vitae suscipit tellus mauris a diam maecenas.
// Eget nunc scelerisque viverra mauris in aliquam sem fringilla.
// Pharetra vel turpis nunc eget lorem dolor sed viverra.
// Mattis pellentesque id nibh tortor id aliquet.
// Purus non enim praesent elementum facilisis leo vel.
// Etiam sit amet nisl purus in mollis nunc sed.
// Tortor at auctor urna nunc id cursus metus aliquam.
// Volutpat odio facilisis mauris sit amet.
// Turpis egestas pretium aenean pharetra magna ac placerat.
// Fermentum dui faucibus in ornare quam viverra orci sagittis eu.
// Porttitor leo a diam sollicitudin tempor id eu.
// Volutpat sed cras ornare arcu dui.
// Ut aliquam purus sit amet luctus venenatis lectus magna.
// Aliquet risus feugiat in ante metus dictum at.
// Mattis nunc sed blandit libero.
// Elit pellentesque habitant morbi tristique senectus et netus.
// Nibh sit amet commodo nulla facilisi nullam vehicula ipsum a.
// Enim sit amet venenatis urna cursus eget nunc scelerisque viverra.
// Amet venenatis urna cursus eget nunc scelerisque viverra mauris in.
// Diam donec adipiscing tristique risus nec feugiat.
// Pulvinar mattis nunc sed blandit libero volutpat.
// Cras fermentum odio eu feugiat pretium nibh ipsum.
// In nulla posuere sollicitudin aliquam ultrices sagittis orci a.
// Mauris pellentesque pulvinar pellentesque habitant morbi tristique senectus et.
// A iaculis at erat pellentesque.
// Morbi blandit cursus risus at ultrices mi tempus imperdiet nulla.
// Eget lorem dolor sed viverra ipsum nunc.
// Leo a diam sollicitudin tempor id eu.
// Interdum consectetur libero id faucibus nisl tincidunt eget nullam non.";
//
//         [Fact]
//         public static void LinksBasedFrequencyStoredOptimalVariantSequenceTest()
//         {
//             using (var scope = new TempLinksTestScope(useSequences: false))
//             {
//                 var links = scope.Links;
//                 var constants = links.Constants;
//
```

```
//                 links.UseUnicode();
//
//                 var sequence = UnicodeMap.FromStringToLinkArray(_sequenceExample);
//
//                 var meaningRoot = links.CreatePoint();
//                 var unaryOne = links.CreateAndUpdate(meaningRoot, constants.Itself);
//                 var frequencyMarker = links.CreateAndUpdate(meaningRoot, constants.Itself);
//                 var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot,
    ↪ constants.Itself);
//
//                 var unaryNumberToAddressConverter = new
    ↪ UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);
//                 var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links,
    ↪ unaryOne);
//                 var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
    ↪ frequencyMarker, unaryOne, unaryNumberIncrementer);
//                 var frequencyPropertyOperator = new PropertyOperator<ulong>(links,
    ↪ frequencyPropertyMarker, frequencyMarker);
//                 var index = new FrequencyIncrementingSequenceIndex<ulong>(links,
    ↪ frequencyPropertyOperator, frequencyIncrementer);
//                 var linkToItsFrequencyNumberConverter = new
    ↪ LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
    ↪ unaryNumberToAddressConverter);
//                 var sequenceToItsLocalElementLevelsConverter = new
    ↪ SequenceToItsLocalElementLevelsConverter<ulong>(links, linkToItsFrequencyNumberConverter);
//                 var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
    ↪ sequenceToItsLocalElementLevelsConverter);
//
//                 var sequences = new Sequences(links, new SequencesOptions<ulong>() { Walker =
    ↪ new LeveledSequenceWalker<ulong>(links) });
//
//                 ExecuteTest(sequences, sequence, sequenceToItsLocalElementLevelsConverter,
    ↪ index, optimalVariantConverter);
//             }
//         }
//
//         [Fact]
//         public static void DictionaryBasedFrequencyStoredOptimalVariantSequenceTest()
//         {
//             using (var scope = new TempLinksTestScope(useSequences: false))
//             {
//                 var links = scope.Links;
//
//                 links.UseUnicode();
//
//                 var sequence = UnicodeMap.FromStringToLinkArray(_sequenceExample);
//
//                 var totalSequenceSymbolFrequencyCounter = new
    ↪ TotalSequenceSymbolFrequencyCounter<ulong>(links);
//
//                 var linkFrequenciesCache = new LinkFrequenciesCache<ulong>(links,
    ↪ totalSequenceSymbolFrequencyCounter);
//
//                 var index = new
    ↪ CachedFrequencyIncrementingSequenceIndex<ulong>(linkFrequenciesCache);
//                 var linkToItsFrequencyNumberConverter = new
    ↪ FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<ulong>(linkFrequenciesCache);
//
//                 var sequenceToItsLocalElementLevelsConverter = new
    ↪ SequenceToItsLocalElementLevelsConverter<ulong>(links, linkToItsFrequencyNumberConverter);
//                 var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
    ↪ sequenceToItsLocalElementLevelsConverter);
//
//                 var sequences = new Sequences(links, new SequencesOptions<ulong>() { Walker =
    ↪ new LeveledSequenceWalker<ulong>(links) });
//
//                 ExecuteTest(sequences, sequence, sequenceToItsLocalElementLevelsConverter,
    ↪ index, optimalVariantConverter);
//             }
//         }
//         private static void ExecuteTest(Sequences sequences, ulong[] sequence,
    ↪ SequenceToItsLocalElementLevelsConverter<ulong> sequenceToItsLocalElementLevelsConverter,
    ↪ ISequenceIndex<ulong> index, OptimalVariantConverter<ulong> optimalVariantConverter)
//         {
//             index.Add(sequence);
//
```

```
//             var optimalVariant = optimalVariantConverter.Convert(sequence);
//
//             var readSequence1 = sequences.ToList(optimalVariant);
//
//             Assert.True(sequence.SequenceEqual(readSequence1));
//         }
//
//         [Fact]
//         public static void SavedSequencesOptimizationTest()
//         {
//             LinksConstants<ulong> constants = new LinksConstants<ulong>((1, long.MaxValue),
    ↪ (long.MaxValue + 1UL, ulong.MaxValue));
//
//             using (var memory = new HeapResizableDirectMemory())
//             using (var disposableLinks = new UInt64UnitedMemoryLinks(memory,
    ↪ UInt64UnitedMemoryLinks.DefaultLinksSizeStep, constants, IndexTreeType.Default))
//             {
//                 var links = new UInt64Links(disposableLinks);
//
//                 var root = links.CreatePoint();
//
//                 //var numberToAddressConverter = new RawNumberToAddressConverter<ulong>();
//                 var addressToNumberConverter = new AddressToRawNumberConverter<ulong>();
//
//                 var unicodeSymbolMarker = links.GetOrCreate(root,
    ↪ addressToNumberConverter.Convert(1));
//                 var unicodeSequenceMarker = links.GetOrCreate(root,
    ↪ addressToNumberConverter.Convert(2));
//
//                 var totalSequenceSymbolFrequencyCounter = new
    ↪ TotalSequenceSymbolFrequencyCounter<ulong>(links);
//                 var linkFrequenciesCache = new LinkFrequenciesCache<ulong>(links,
    ↪ totalSequenceSymbolFrequencyCounter);
//                 var index = new
    ↪ CachedFrequencyIncrementingSequenceIndex<ulong>(linkFrequenciesCache);
//                 var linkToItsFrequencyNumberConverter = new
    ↪ FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<ulong>(linkFrequenciesCache);
//                 var sequenceToItsLocalElementLevelsConverter = new
    ↪ SequenceToItsLocalElementLevelsConverter<ulong>(links, linkToItsFrequencyNumberConverter);
//                 var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
    ↪ sequenceToItsLocalElementLevelsConverter);
//
//                 var walker = new RightSequenceWalker<ulong>(links, new DefaultStack<ulong>(),
    ↪ (link) => constants.IsExternalReference(link) || links.IsPartialPoint(link));
//
//                 var unicodeSequencesOptions = new SequencesOptions<ulong>()
//                 {
//                     UseSequenceMarker = true,
//                     SequenceMarkerLink = unicodeSequenceMarker,
//                     UseIndex = true,
//                     Index = index,
//                     LinksToSequenceConverter = optimalVariantConverter,
//                     Walker = walker,
//                     UseGarbageCollection = true
//                 };
//
//                 var unicodeSequences = new Sequences(new SynchronizedLinks<ulong>(links),
    ↪ unicodeSequencesOptions);
//
//                 // Create some sequences
//                 var strings = _loremIpsumExample.Split(new[] { '\n', '\r' },
    ↪ StringSplitOptions.RemoveEmptyEntries);
//                 var arrays = strings.Select(x => x.Select(y =>
    ↪ addressToNumberConverter.Convert(y)).ToArray()).ToArray();
//                 for (int i = 0; i < arrays.Length; i++)
//                 {
//                     unicodeSequences.Create(arrays[i].ShiftRight());
//                 }
//
//                 var linksCountAfterCreation = links.Count();
//
//                 // get list of sequences links
//                 // for each sequence link
//                 //   create new sequence version
//                 //   if new sequence is not the same as sequence link
//                 //     delete sequence link
```

```
202 //                  //     collect garbadge
203 //                  unicodeSequences.CompactAll();
204 //
205 //                  var linksCountAfterCompactification = links.Count();
206 //
207 //                  Assert.True(linksCountAfterCompactification < linksCountAfterCreation);
208 //              }
209 //          }
210 //      }
211 // }
```

1.70 ./csharp/Platform.Data.Doublets.Sequences.Tests/RationalNumbersTests.cs

```
1  using Platform.Data.Doublets.Memory;
2  using Platform.Data.Doublets.Memory.United.Generic;
3  using Platform.Data.Doublets.Numbers.Rational;
4  using Platform.Data.Doublets.Numbers.Raw;
5  using Platform.Data.Doublets.Sequences.Converters;
6  using Platform.Data.Numbers.Raw;
7  using Platform.Memory;
8  using Xunit;
9  using TLinkAddress = System.UInt64;
10
11 namespace Platform.Data.Doublets.Sequences.Tests
12 {
13     public class RationalNumbersTests
14     {
15         public ILinks<TLinkAddress> CreateLinks() => CreateLinks<TLinkAddress>(new
           ↪  IO.TemporaryFile());
16
17         public ILinks<TLinkAddress> CreateLinks<TLinkAddress>(string dataDbFilename)
18         {
19             var linksConstants = new
               ↪  LinksConstants<TLinkAddress>(enableExternalReferencesSupport: true);
20             return new UnitedMemoryLinks<TLinkAddress>(new
               ↪  FileMappedResizableDirectMemory(dataDbFilename),
               ↪  UnitedMemoryLinks<TLinkAddress>.DefaultLinksSizeStep, linksConstants,
               ↪  IndexTreeType.Default);
21         }
22
23         [Fact]
24         public void DecimalMinValueTest()
25         {
26             const decimal @decimal = decimal.MinValue;
27             var links = CreateLinks();
28             TLinkAddress negativeNumberMarker = links.Create();
29             AddressToRawNumberConverter<TLinkAddress> addressToRawNumberConverter = new();
30             RawNumberToAddressConverter<TLinkAddress> numberToAddressConverter = new();
31             BalancedVariantConverter<TLinkAddress> balancedVariantConverter = new(links);
32             BigIntegerToRawNumberSequenceConverter<TLinkAddress>
               ↪  bigIntegerToRawNumberSequenceConverter = new(links, addressToRawNumberConverter,
               ↪  balancedVariantConverter, negativeNumberMarker);
33             RawNumberSequenceToBigIntegerConverter<TLinkAddress>
               ↪  rawNumberSequenceToBigIntegerConverter = new(links, numberToAddressConverter,
               ↪  negativeNumberMarker);
34             DecimalToRationalConverter<TLinkAddress> decimalToRationalConverter = new(links,
               ↪  bigIntegerToRawNumberSequenceConverter);
35             RationalToDecimalConverter<TLinkAddress> rationalToDecimalConverter = new(links,
               ↪  rawNumberSequenceToBigIntegerConverter);
36             var rationalNumber = decimalToRationalConverter.Convert(@decimal);
37             var decimalFromRational = rationalToDecimalConverter.Convert(rationalNumber);
38            Assert.Equal(@decimal, decimalFromRational);
39         }
40
41         [Fact]
42         public void DecimalMaxValueTest()
43         {
44             const decimal @decimal = decimal.MaxValue;
45             var links = CreateLinks();
46             TLinkAddress negativeNumberMarker = links.Create();
47             AddressToRawNumberConverter<TLinkAddress> addressToRawNumberConverter = new();
48             RawNumberToAddressConverter<TLinkAddress> numberToAddressConverter = new();
49             BalancedVariantConverter<TLinkAddress> balancedVariantConverter = new(links);
50             BigIntegerToRawNumberSequenceConverter<TLinkAddress>
               ↪  bigIntegerToRawNumberSequenceConverter = new(links, addressToRawNumberConverter,
               ↪  balancedVariantConverter, negativeNumberMarker);
51             RawNumberSequenceToBigIntegerConverter<TLinkAddress>
               ↪  rawNumberSequenceToBigIntegerConverter = new(links, numberToAddressConverter,
               ↪  negativeNumberMarker);
```

```
52              DecimalToRationalConverter<TLinkAddress> decimalToRationalConverter = new(links,
                ↪  bigIntegerToRawNumberSequenceConverter);
53              RationalToDecimalConverter<TLinkAddress> rationalToDecimalConverter = new(links,
                ↪  rawNumberSequenceToBigIntegerConverter);
54              var rationalNumber = decimalToRationalConverter.Convert(@decimal);
55              var decimalFromRational = rationalToDecimalConverter.Convert(rationalNumber);
56              Assert.Equal(@decimal, decimalFromRational);
57          }
58
59          [Fact]
60          public void DecimalPositiveHalfTest()
61          {
62              const decimal @decimal = 0.5M;
63              var links = CreateLinks();
64              TLinkAddress negativeNumberMarker = links.Create();
65              AddressToRawNumberConverter<TLinkAddress> addressToRawNumberConverter = new();
66              RawNumberToAddressConverter<TLinkAddress> numberToAddressConverter = new();
67              BalancedVariantConverter<TLinkAddress> balancedVariantConverter = new(links);
68              BigIntegerToRawNumberSequenceConverter<TLinkAddress>
                ↪  bigIntegerToRawNumberSequenceConverter = new(links, addressToRawNumberConverter,
                ↪  balancedVariantConverter, negativeNumberMarker);
69              RawNumberSequenceToBigIntegerConverter<TLinkAddress>
                ↪  rawNumberSequenceToBigIntegerConverter = new(links, numberToAddressConverter,
                ↪  negativeNumberMarker);
70              DecimalToRationalConverter<TLinkAddress> decimalToRationalConverter = new(links,
                ↪  bigIntegerToRawNumberSequenceConverter);
71              RationalToDecimalConverter<TLinkAddress> rationalToDecimalConverter = new(links,
                ↪  rawNumberSequenceToBigIntegerConverter);
72              var rationalNumber = decimalToRationalConverter.Convert(@decimal);
73              var decimalFromRational = rationalToDecimalConverter.Convert(rationalNumber);
74              Assert.Equal(@decimal, decimalFromRational);
75          }
76
77          [Fact]
78          public void DecimalNegativeHalfTest()
79          {
80              const decimal @decimal = -0.5M;
81              var links = CreateLinks();
82              TLinkAddress negativeNumberMarker = links.Create();
83              AddressToRawNumberConverter<TLinkAddress> addressToRawNumberConverter = new();
84              RawNumberToAddressConverter<TLinkAddress> numberToAddressConverter = new();
85              BalancedVariantConverter<TLinkAddress> balancedVariantConverter = new(links);
86              BigIntegerToRawNumberSequenceConverter<TLinkAddress>
                ↪  bigIntegerToRawNumberSequenceConverter = new(links, addressToRawNumberConverter,
                ↪  balancedVariantConverter, negativeNumberMarker);
87              RawNumberSequenceToBigIntegerConverter<TLinkAddress>
                ↪  rawNumberSequenceToBigIntegerConverter = new(links, numberToAddressConverter,
                ↪  negativeNumberMarker);
88              DecimalToRationalConverter<TLinkAddress> decimalToRationalConverter = new(links,
                ↪  bigIntegerToRawNumberSequenceConverter);
89              RationalToDecimalConverter<TLinkAddress> rationalToDecimalConverter = new(links,
                ↪  rawNumberSequenceToBigIntegerConverter);
90              var rationalNumber = decimalToRationalConverter.Convert(@decimal);
91              var decimalFromRational = rationalToDecimalConverter.Convert(rationalNumber);
92              Assert.Equal(@decimal, decimalFromRational);
93          }
94
95          [Fact]
96          public void DecimalOneTest()
97          {
98              const decimal @decimal = 1;
99              var links = CreateLinks();
100             TLinkAddress negativeNumberMarker = links.Create();
101             AddressToRawNumberConverter<TLinkAddress> addressToRawNumberConverter = new();
102             RawNumberToAddressConverter<TLinkAddress> numberToAddressConverter = new();
103             BalancedVariantConverter<TLinkAddress> balancedVariantConverter = new(links);
104             BigIntegerToRawNumberSequenceConverter<TLinkAddress>
                ↪  bigIntegerToRawNumberSequenceConverter = new(links, addressToRawNumberConverter,
                ↪  balancedVariantConverter, negativeNumberMarker);
105             RawNumberSequenceToBigIntegerConverter<TLinkAddress>
                ↪  rawNumberSequenceToBigIntegerConverter = new(links, numberToAddressConverter,
                ↪  negativeNumberMarker);
106             DecimalToRationalConverter<TLinkAddress> decimalToRationalConverter = new(links,
                ↪  bigIntegerToRawNumberSequenceConverter);
107             RationalToDecimalConverter<TLinkAddress> rationalToDecimalConverter = new(links,
                ↪  rawNumberSequenceToBigIntegerConverter);
108             var rationalNumber = decimalToRationalConverter.Convert(@decimal);
```

```
            var decimalFromRational = rationalToDecimalConverter.Convert(rationalNumber);
            Assert.Equal(@decimal, decimalFromRational);
        }

        [Fact]
        public void DecimalMinusOneTest()
        {
            const decimal @decimal = -1;
            var links = CreateLinks();
            TLinkAddress negativeNumberMarker = links.Create();
            AddressToRawNumberConverter<TLinkAddress> addressToRawNumberConverter = new();
            RawNumberToAddressConverter<TLinkAddress> numberToAddressConverter = new();
            BalancedVariantConverter<TLinkAddress> balancedVariantConverter = new(links);
            BigIntegerToRawNumberSequenceConverter<TLinkAddress>
            ↪ bigIntegerToRawNumberSequenceConverter = new(links, addressToRawNumberConverter,
            ↪ balancedVariantConverter, negativeNumberMarker);
            RawNumberSequenceToBigIntegerConverter<TLinkAddress>
            ↪ rawNumberSequenceToBigIntegerConverter = new(links, numberToAddressConverter,
            ↪ negativeNumberMarker);
            DecimalToRationalConverter<TLinkAddress> decimalToRationalConverter = new(links,
            ↪ bigIntegerToRawNumberSequenceConverter);
            RationalToDecimalConverter<TLinkAddress> rationalToDecimalConverter = new(links,
            ↪ rawNumberSequenceToBigIntegerConverter);
            var rationalNumber = decimalToRationalConverter.Convert(@decimal);
            var decimalFromRational = rationalToDecimalConverter.Convert(rationalNumber);
            Assert.Equal(@decimal, decimalFromRational);
        }
    }
}
```

1.71 ./csharp/Platform.Data.Doublets.Sequences.Tests/ReadSequenceTests.cs

```
// using System;
// using System.Collections.Generic;
// using System.Diagnostics;
// using System.Linq;
// using Xunit;
// using Platform.Data.Sequences;
// using Platform.Data.Doublets.Sequences.Converters;
// using Platform.Data.Doublets.Sequences.Walkers;
// using Platform.Data.Doublets.Sequences;
//
// namespace Platform.Data.Doublets.Sequences.Tests
// {
//     public static class ReadSequenceTests
//     {
//         [Fact]
//         public static void ReadSequenceTest()
//         {
//             const long sequenceLength = 2000;
//
//             using (var scope = new TempLinksTestScope(useSequences: false))
//             {
//                 var links = scope.Links;
//                 var sequences = new Sequences(links, new SequencesOptions<ulong> { Walker =
    ↪  new LeveledSequenceWalker<ulong>(links) });
//
//                 var sequence = new ulong[sequenceLength];
//                 for (var i = 0; i < sequenceLength; i++)
//                 {
//                     sequence[i] = links.Create();
//                 }
//
//                 var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
//
//                 var sw1 = Stopwatch.StartNew();
//                 var balancedVariant = balancedVariantConverter.Convert(sequence); sw1.Stop();
//
//                 var sw2 = Stopwatch.StartNew();
//                 var readSequence1 = sequences.ToList(balancedVariant); sw2.Stop();
//
//                 var sw3 = Stopwatch.StartNew();
//                 var readSequence2 = new List<ulong>();
//                 SequenceWalker.WalkRight(balancedVariant,
//                                          links.GetSource,
//                                          links.GetTarget,
//                                          links.IsPartialPoint,
//                                          readSequence2.Add);
```

```
//                 sw3.Stop();
//
//                 Assert.True(sequence.SequenceEqual(readSequence1));
//
//                 Assert.True(sequence.SequenceEqual(readSequence2));
//
//                 // Assert.True(sw2.Elapsed < sw3.Elapsed);
//
//                 Console.WriteLine($"Stack-based walker: {sw3.Elapsed}, Level-based reader:
↪  {sw2.Elapsed}");
//
//                 for (var i = 0; i < sequenceLength; i++)
//                 {
//                     links.Delete(sequence[i]);
//                 }
//             }
//         }
//     }
// }
```

1.72 ./csharp/Platform.Data.Doublets.Sequences.Tests/SequencesTests.cs

```
// using System;
// using System.Collections.Generic;
// using System.Diagnostics;
// using System.Linq;
// using Xunit;
// using Platform.Collections;
// using Platform.Collections.Arrays;
// using Platform.Random;
// using Platform.IO;
// using Platform.Singletons;
// using Platform.Data.Doublets.Sequences;
// using Platform.Data.Doublets.Sequences.Frequencies.Cache;
// using Platform.Data.Doublets.Sequences.Frequencies.Counters;
// using Platform.Data.Doublets.Sequences.Converters;
// using Platform.Data.Doublets.Unicode;
//
// namespace Platform.Data.Doublets.Sequences.Tests
// {
//     public static class SequencesTests
//     {
//         private static readonly LinksConstants<ulong> _constants =
↪  Default<LinksConstants<ulong>>.Instance;
//
//         static SequencesTests()
//         {
//             // Trigger static constructor to not mess with perfomance measurements
//             _ = BitString.GetBitMaskFromIndex(1);
//         }
//
//         [Fact]
//         public static void CreateAllVariantsTest()
//         {
//             const long sequenceLength = 8;
//
//             using (var scope = new TempLinksTestScope(useSequences: true))
//             {
//                 var links = scope.Links;
//                 var sequences = scope.Sequences;
//
//                 var sequence = new ulong[sequenceLength];
//                 for (var i = 0; i < sequenceLength; i++)
//                 {
//                     sequence[i] = links.Create();
//                 }
//
//                 var sw1 = Stopwatch.StartNew();
//                 var results1 = sequences.CreateAllVariants1(sequence); sw1.Stop();
//
//                 var sw2 = Stopwatch.StartNew();
//                 var results2 = sequences.CreateAllVariants2(sequence); sw2.Stop();
//
//                 Assert.True(results1.Count > results2.Length);
//                 Assert.True(sw1.Elapsed > sw2.Elapsed);
//
//                 for (var i = 0; i < sequenceLength; i++)
//                 {
//                     links.Delete(sequence[i]);
```

```
//                 }
//
//                 Assert.True(links.Count() == 0);
//             }
//         }
//
//         //[Fact]
//         //public void CUDTest()
//         //{
//         //    var tempFilename = Path.GetTempFileName();
//         //
//         //    const long sequenceLength = 8;
//         //
//         //    const ulong itself = LinksConstants.Itself;
//         //
//         //    using (var memoryAdapter = new ResizableDirectMemoryLinks(tempFilename,
↪ DefaultLinksSizeStep))
//         //    using (var links = new Links(memoryAdapter))
//         //    {
//         //        var sequence = new ulong[sequenceLength];
//         //        for (var i = 0; i < sequenceLength; i++)
//         //            sequence[i] = links.Create(itself, itself);
//         //
//         //        SequencesOptions o = new SequencesOptions();
//         //
//         // TODO: Из числа в bool значения o.UseSequenceMarker = ((value & 1) != 0)
//         //        o.
//         //
//         //        var sequences = new Sequences(links);
//         //
//         //        var sw1 = Stopwatch.StartNew();
//         //        var results1 = sequences.CreateAllVariants1(sequence); sw1.Stop();
//         //
//         //        var sw2 = Stopwatch.StartNew();
//         //        var results2 = sequences.CreateAllVariants2(sequence); sw2.Stop();
//         //
//         //        Assert.True(results1.Count > results2.Length);
//         //        Assert.True(sw1.Elapsed > sw2.Elapsed);
//         //
//         //        for (var i = 0; i < sequenceLength; i++)
//         //            links.Delete(sequence[i]);
//         //    }
//         //
//         //    File.Delete(tempFilename);
//         //}
//         //
//         [Fact]
//         public static void AllVariantsSearchTest()
//         {
//             const long sequenceLength = 8;
//
//             using (var scope = new TempLinksTestScope(useSequences: true))
//             {
//                 var links = scope.Links;
//                 var sequences = scope.Sequences;
//
//                 var sequence = new ulong[sequenceLength];
//                 for (var i = 0; i < sequenceLength; i++)
//                 {
//                     sequence[i] = links.Create();
//                 }
//
//                 var createResults =
↪ sequences.CreateAllVariants2(sequence).Distinct().ToArray();
//
//                 //for (int i = 0; i < createResults.Length; i++)
//                 //    sequences.Create(createResults[i]);
//
//                 var sw0 = Stopwatch.StartNew();
//                 var searchResults0 = sequences.GetAllMatchingSequences0(sequence); sw0.Stop();
//
//                 var sw1 = Stopwatch.StartNew();
//                 var searchResults1 = sequences.GetAllMatchingSequences1(sequence); sw1.Stop();
//
//                 var sw2 = Stopwatch.StartNew();
//                 var searchResults2 = sequences.Each1(sequence); sw2.Stop();
//
```

```
//                  var sw3 = Stopwatch.StartNew();
//                  var searchResults3 = sequences.Each(sequence.ShiftRight()); sw3.Stop();
//
//                  var intersection0 = createResults.Intersect(searchResults0).ToList();
//                  Assert.True(intersection0.Count == searchResults0.Count);
//                  Assert.True(intersection0.Count == createResults.Length);
//
//                  var intersection1 = createResults.Intersect(searchResults1).ToList();
//                  Assert.True(intersection1.Count == searchResults1.Count);
//                  Assert.True(intersection1.Count == createResults.Length);
//
//                  var intersection2 = createResults.Intersect(searchResults2).ToList();
//                  Assert.True(intersection2.Count == searchResults2.Count);
//                  Assert.True(intersection2.Count == createResults.Length);
//
//                  var intersection3 = createResults.Intersect(searchResults3).ToList();
//                  Assert.True(intersection3.Count == searchResults3.Count);
//                  Assert.True(intersection3.Count == createResults.Length);
//
//                  for (var i = 0; i < sequenceLength; i++)
//                  {
//                      links.Delete(sequence[i]);
//                  }
//              }
//          }
//
//          [Fact]
//          public static void BalancedVariantSearchTest()
//          {
//              const long sequenceLength = 200;
//
//              using (var scope = new TempLinksTestScope(useSequences: true))
//              {
//                  var links = scope.Links;
//                  var sequences = scope.Sequences;
//
//                  var sequence = new ulong[sequenceLength];
//                  for (var i = 0; i < sequenceLength; i++)
//                  {
//                      sequence[i] = links.Create();
//                  }
//
//                  var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
//
//                  var sw1 = Stopwatch.StartNew();
//                  var balancedVariant = balancedVariantConverter.Convert(sequence); sw1.Stop();
//
//                  var sw2 = Stopwatch.StartNew();
//                  var searchResults2 = sequences.GetAllMatchingSequences0(sequence); sw2.Stop();
//
//                  var sw3 = Stopwatch.StartNew();
//                  var searchResults3 = sequences.GetAllMatchingSequences1(sequence); sw3.Stop();
//
//                  // На количестве в 200 элементов это будет занимать вечность
//                  //var sw4 = Stopwatch.StartNew();
//                  //var searchResults4 = sequences.Each(sequence); sw4.Stop();
//
//                  Assert.True(searchResults2.Count == 1 && balancedVariant ==
    ↪ searchResults2[0]);
//
//                  Assert.True(searchResults3.Count == 1 && balancedVariant ==
    ↪ searchResults3.First());
//
//                  //Assert.True(sw1.Elapsed < sw2.Elapsed);
//
//                  for (var i = 0; i < sequenceLength; i++)
//                  {
//                      links.Delete(sequence[i]);
//                  }
//              }
//          }
//
//          [Fact]
//          public static void AllPartialVariantsSearchTest()
//          {
//              const long sequenceLength = 8;
//
```

```
//             using (var scope = new TempLinksTestScope(useSequences: true))
//             {
//                 var links = scope.Links;
//                 var sequences = scope.Sequences;
//
//                 var sequence = new ulong[sequenceLength];
//                 for (var i = 0; i < sequenceLength; i++)
//                 {
//                     sequence[i] = links.Create();
//                 }
//
//                 var createResults = sequences.CreateAllVariants2(sequence);
//
//                 //var createResultsStrings = createResults.Select(x => x + ": " +
↪  sequences.FormatSequence(x)).ToList();
//                 //Global.Trash = createResultsStrings;
//
//                 var partialSequence = new ulong[sequenceLength - 2];
//
//                 Array.Copy(sequence, 1, partialSequence, 0, (int)sequenceLength - 2);
//
//                 var sw1 = Stopwatch.StartNew();
//                 var searchResults1 =
↪  sequences.GetAllPartiallyMatchingSequences0(partialSequence); sw1.Stop();
//
//                 var sw2 = Stopwatch.StartNew();
//                 var searchResults2 =
↪  sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();
//
//                 //var sw3 = Stopwatch.StartNew();
//                 //var searchResults3 =
↪  sequences.GetAllPartiallyMatchingSequences2(partialSequence); sw3.Stop();
//
//                 var sw4 = Stopwatch.StartNew();
//                 var searchResults4 =
↪  sequences.GetAllPartiallyMatchingSequences3(partialSequence); sw4.Stop();
//
//                 //Global.Trash = searchResults3;
//
//                 //var searchResults1Strings = searchResults1.Select(x => x + ": " +
↪  sequences.FormatSequence(x)).ToList();
//                 //Global.Trash = searchResults1Strings;
//
//                 var intersection1 = createResults.Intersect(searchResults1).ToList();
//                 Assert.True(intersection1.Count == createResults.Length);
//
//                 var intersection2 = createResults.Intersect(searchResults2).ToList();
//                 Assert.True(intersection2.Count == createResults.Length);
//
//                 var intersection4 = createResults.Intersect(searchResults4).ToList();
//                 Assert.True(intersection4.Count == createResults.Length);
//
//                 for (var i = 0; i < sequenceLength; i++)
//                 {
//                     links.Delete(sequence[i]);
//                 }
//             }
//         }
//
//         [Fact]
//         public static void BalancedPartialVariantsSearchTest()
//         {
//             const long sequenceLength = 200;
//
//             using (var scope = new TempLinksTestScope(useSequences: true))
//             {
//                 var links = scope.Links;
//                 var sequences = scope.Sequences;
//
//                 var sequence = new ulong[sequenceLength];
//                 for (var i = 0; i < sequenceLength; i++)
//                 {
//                     sequence[i] = links.Create();
//                 }
//
//                 var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
//
```

```
278 //                 var balancedVariant = balancedVariantConverter.Convert(sequence);
279 //
280 //                 var partialSequence = new ulong[sequenceLength - 2];
281 //
282 //                 Array.Copy(sequence, 1, partialSequence, 0, (int)sequenceLength - 2);
283 //
284 //                 var sw1 = Stopwatch.StartNew();
285 //                 var searchResults1 =
    ↪ sequences.GetAllPartiallyMatchingSequences0(partialSequence); sw1.Stop();
286 //
287 //                 var sw2 = Stopwatch.StartNew();
288 //                 var searchResults2 =
    ↪ sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();
289 //
290 //                 Assert.True(searchResults1.Count == 1 && balancedVariant ==
    ↪ searchResults1[0]);
291 //
292 //                 Assert.True(searchResults2.Count == 1 && balancedVariant ==
    ↪ searchResults2.First());
293 //
294 //                 for (var i = 0; i < sequenceLength; i++)
295 //                 {
296 //                     links.Delete(sequence[i]);
297 //                 }
298 //             }
299 //         }
300 //
301 //         [Fact(Skip = "Correct implementation is pending")]
302 //         public static void PatternMatchTest()
303 //         {
304 //             var zeroOrMany = Sequences.ZeroOrMany;
305 //
306 //             using (var scope = new TempLinksTestScope(useSequences: true))
307 //             {
308 //                 var links = scope.Links;
309 //                 var sequences = scope.Sequences;
310 //
311 //                 var e1 = links.Create();
312 //                 var e2 = links.Create();
313 //
314 //                 var sequence = new[]
315 //                 {
316 //                     e1, e2, e1, e2 // mama / papa
317 //                 };
318 //
319 //                 var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
320 //
321 //                 var balancedVariant = balancedVariantConverter.Convert(sequence);
322 //
323 //                 // 1: [1]
324 //                 // 2: [2]
325 //                 // 3: [1,2]
326 //                 // 4: [1,2,1,2]
327 //
328 //                 var doublet = links.GetSource(balancedVariant);
329 //
330 //                 var matchedSequences1 = sequences.MatchPattern(e2, e1, zeroOrMany);
331 //
332 //                 Assert.True(matchedSequences1.Count == 0);
333 //
334 //                 var matchedSequences2 = sequences.MatchPattern(zeroOrMany, e2, e1);
335 //
336 //                 Assert.True(matchedSequences2.Count == 0);
337 //
338 //                 var matchedSequences3 = sequences.MatchPattern(e1, zeroOrMany, e1);
339 //
340 //                 Assert.True(matchedSequences3.Count == 0);
341 //
342 //                 var matchedSequences4 = sequences.MatchPattern(e1, zeroOrMany, e2);
343 //
344 //                 Assert.Contains(doublet, matchedSequences4);
345 //                 Assert.Contains(balancedVariant, matchedSequences4);
346 //
347 //                 for (var i = 0; i < sequence.Length; i++)
348 //                 {
349 //                     links.Delete(sequence[i]);
350 //                 }
```

```
//             }
//         }
//
//         [Fact]
//         public static void IndexTest()
//         {
//             using (var scope = new TempLinksTestScope(new SequencesOptions<ulong> { UseIndex = true }, useSequences: true))
//             {
//                 var links = scope.Links;
//                 var sequences = scope.Sequences;
//                 var index = sequences.Options.Index;
//
//                 var e1 = links.Create();
//                 var e2 = links.Create();
//
//                 var sequence = new[]
//                 {
//                     e1, e2, e1, e2 // mama / papa
//                 };
//
//                 Assert.False(index.MightContain(sequence));
//
//                 index.Add(sequence);
//
//                 Assert.True(index.MightContain(sequence));
//             }
//         }
//         private static readonly string _exampleText =
//             @"([english version](https://github.com/Konard/LinksPlatform/wiki/About-the-beginning))
//
// Обозначение пустоты, какое оно? Темнота ли это? Там где отсутствие света, отсутствие фотонов (носителей света)? Или это то, что полностью отражает свет? Пустой белый лист бумаги? Там где есть место для нового начала? Разве пустота это не характеристика пространства? Пространство это то, что можно чем-то наполнить?
//
// [![чёрное пространство, белое пространство](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/1.png ""чёрное пространство, белое пространство"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/1.png)
//
// Что может быть минимальным рисунком, образом, графикой? Может быть это точка? Это ли простейшая форма? Но есть ли у точки размер? Цвет? Масса? Координаты? Время существования?
//
// [![чёрное пространство, чёрная точка](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/2.png ""чёрное пространство, чёрная точка"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/2.png)
//
// А что если повторить? Сделать копию? Создать дубликат? Из одного сделать два? Может это быть так? Инверсия? Отражение? Сумма?
//
// [![белая точка, чёрная точка](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/3.png ""белая точка, чёрная точка"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/3.png)
//
// А что если мы вообразим движение? Нужно ли время? Каким самым коротким будет путь? Что будет если этот путь зафиксировать? Запомнить след? Как две точки становятся линией? Чертой? Гранью? Разделителем? Единицей?
//
// [![две белые точки, чёрная вертикальная линия](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/4.png ""две белые точки, чёрная вертикальная линия"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/4.png)
//
// Можно ли замкнуть движение? Может ли это быть кругом? Можно ли замкнуть время? Или остаётся только спираль? Но что если замкнуть предел? Создать ограничение, разделение? Получится замкнутая область? Полностью отделённая от всего остального? Но что это всё остальное? Что можно делить? В каком направлении? Ничего или всё? Пустота или полнота? Начало или конец? Или может быть это единица и ноль? Дуальность? Противоположность? А что будет с кругом если у него нет размера? Будет ли круг точкой? Точка состоящая из точек?
//
// [![белая вертикальная линия, чёрный круг](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/5.png ""белая вертикальная линия, чёрный круг"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/5.png)
```

```
//
// Как ещё можно использовать грань, черту, линию? А что если она может что-то соединять, может тогда её нужно повернуть? Почему то, что перпендикулярно вертикальному горизонтально? Горизонт? Инвертирует ли это смысл? Что такое смысл? Из чего состоит смысл? Существует ли элементарная единица смысла?
//
// [![белый круг, чёрная горизонтальная линия](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/6.png ""белый круг, чёрная горизонтальная линия"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/6.png)
//
// Соединять, допустим, а какой смысл в этом есть ещё? Что если помимо смысла ""соединить, связать"", есть ещё и смысл направления ""от начала к концу""? От предка к потомку? От родителя к ребёнку? От общего к частному?
//
// [![белая горизонтальная линия, чёрная горизонтальная стрелка](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/7.png ""белая горизонтальная линия, чёрная горизонтальная стрелка"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/7.png)
//
// Шаг назад. Возьмём опять отделённую область, которая лишь та же замкнутая линия, что ещё она может представлять собой? Объект? Но в чём его суть? Разве не в том, что у него есть граница, разделяющая внутреннее и внешнее? Допустим связь, стрелка, линия соединяет два объекта, как бы это выглядело?
//
// [![белая связь, чёрная направленная связь](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/8.png ""белая связь, чёрная направленная связь"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/8.png)
//
// Допустим у нас есть смысл ""связать"" и смысл ""направления"", много ли это нам даёт? Много ли вариантов интерпретации? А что если уточнить, каким именно образом выполнена связь? Что если можно задать ей чёткий, конкретный смысл? Что это будет? Тип? Глагол? Связка? Действие? Трансформация? Переход из состояния в состояние? Или всё это и есть объект, суть которого в его конечном состоянии, если конечно конец определён направлением?
//
// [![белая обычная и направленная связи, чёрная типизированная связь](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/9.png ""белая обычная и направленная связи, чёрная типизированная связь"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/9.png)
//
// А что если всё это время, мы смотрели на суть как бы снаружи? Можно ли взглянуть на это изнутри? Что будет внутри объектов? Объекты ли это? Или это связи? Может ли эта структура описать сама себя? Но что тогда получится, разве это не рекурсия? Может это фрактал?
//
// [![белая обычная и направленная связи с рекурсивной внутренней структурой, чёрная типизированная связь с рекурсивной внутренней структурой](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/10.png ""белая обычная и направленная связи с рекурсивной внутренней структурой, чёрная типизированная связь с рекурсивной внутренней структурой"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/10.png)
//
// На один уровень внутрь (вниз)? Или на один уровень во вне (вверх)? Или это можно назвать шагом рекурсии или фрактала?
//
// [![белая обычная и направленная связи с двойной рекурсивной внутренней структурой, чёрная типизированная связь с двойной рекурсивной внутренней структурой](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/11.png ""белая обычная и направленная связи с двойной рекурсивной внутренней структурой, чёрная типизированная связь с двойной рекурсивной внутренней структурой"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/11.png)
//
// Последовательность? Массив? Список? Множество? Объект? Таблица? Элементы? Цвета? Символы? Буквы? Слово? Цифры? Число? Алфавит? Дерево? Сеть? Граф? Гиперграф?
//
// [![белая обычная и направленная связи со структурой из 8 цветных элементов последовательности, чёрная типизированная связь со структурой из 8 цветных элементов последовательности](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/12.png ""белая обычная и направленная связи со структурой из 8 цветных элементов последовательности, чёрная типизированная связь со структурой из 8 цветных элементов последовательности"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/12.png)
//
// ...
//
```

```
// [![анимация](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro-animation-500.gif ""анимация"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro-animation-500.gif)";
//         private static readonly string _exampleLoremIpsumText =
//             @"Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna aliqua.
// Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.";
//
//         [Fact]
//         public static void CompressionTest()
//         {
//             using (var scope = new TempLinksTestScope(useSequences: true))
//             {
//                 var links = scope.Links;
//                 var sequences = scope.Sequences;
//
//                 var e1 = links.Create();
//                 var e2 = links.Create();
//
//                 var sequence = new[]
//                 {
//                     e1, e2, e1, e2 // mama / papa / template [(m/p), a] { [1] [2] [1] [2] }
//                 };
//
//                 var balancedVariantConverter = new BalancedVariantConverter<ulong>(links.Unsync);
//                 var totalSequenceSymbolFrequencyCounter = new TotalSequenceSymbolFrequencyCounter<ulong>(links.Unsync);
//                 var doubletFrequenciesCache = new LinkFrequenciesCache<ulong>(links.Unsync, totalSequenceSymbolFrequencyCounter);
//                 var compressingConverter = new CompressingConverter<ulong>(links.Unsync, balancedVariantConverter, doubletFrequenciesCache);
//
//                 var compressedVariant = compressingConverter.Convert(sequence);
//
//                 // 1: [1]       (1->1) point
//                 // 2: [2]       (2->2) point
//                 // 3: [1,2]     (1->2) doublet
//                 // 4: [1,2,1,2] (3->3) doublet
//
//                 Assert.True(links.GetSource(links.GetSource(compressedVariant)) == sequence[0]);
//                 Assert.True(links.GetTarget(links.GetSource(compressedVariant)) == sequence[1]);
//                 Assert.True(links.GetSource(links.GetTarget(compressedVariant)) == sequence[2]);
//                 Assert.True(links.GetTarget(links.GetTarget(compressedVariant)) == sequence[3]);
//
//                 var source = _constants.SourcePart;
//                 var target = _constants.TargetPart;
//
//                 Assert.True(links.GetByKeys(compressedVariant, source, source) == sequence[0]);
//                 Assert.True(links.GetByKeys(compressedVariant, source, target) == sequence[1]);
//                 Assert.True(links.GetByKeys(compressedVariant, target, source) == sequence[2]);
//                 Assert.True(links.GetByKeys(compressedVariant, target, target) == sequence[3]);
//
//                 // 4 - length of sequence
//                 Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 0) == sequence[0]);
//                 Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 1) == sequence[1]);
//                 Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 2) == sequence[2]);
//                 Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 3) == sequence[3]);
//             }
//         }
//
//         [Fact]
```

```
//         public static void CompressionEfficiencyTest()
//         {
//             var strings = _exampleLoremIpsumText.Split(new[] { '\n', '\r' },
↪  StringSplitOptions.RemoveEmptyEntries);
//             var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
//             var totalCharacters = arrays.Select(x => x.Length).Sum();
//
//             using (var scope1 = new TempLinksTestScope(useSequences: true))
//             using (var scope2 = new TempLinksTestScope(useSequences: true))
//             using (var scope3 = new TempLinksTestScope(useSequences: true))
//             {
//                 scope1.Links.Unsync.UseUnicode();
//                 scope2.Links.Unsync.UseUnicode();
//                 scope3.Links.Unsync.UseUnicode();
//
//                 var balancedVariantConverter1 = new
↪  BalancedVariantConverter<ulong>(scope1.Links.Unsync);
//                 var totalSequenceSymbolFrequencyCounter = new
↪  TotalSequenceSymbolFrequencyCounter<ulong>(scope1.Links.Unsync);
//                 var linkFrequenciesCache1 = new
↪  LinkFrequenciesCache<ulong>(scope1.Links.Unsync, totalSequenceSymbolFrequencyCounter);
//                 var compressor1 = new CompressingConverter<ulong>(scope1.Links.Unsync,
↪  balancedVariantConverter1, linkFrequenciesCache1, doInitialFrequenciesIncrement: false);
//
//                 //var compressor2 = scope2.Sequences;
//                 var compressor3 = scope3.Sequences;
//
//                 var constants = Default<LinksConstants<ulong>>.Instance;
//
//                 var sequences = compressor3;
//                 //var meaningRoot = links.CreatePoint();
//                 //var unaryOne = links.CreateAndUpdate(meaningRoot, constants.Itself);
//                 //var frequencyMarker = links.CreateAndUpdate(meaningRoot, constants.Itself);
//                 //var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot,
↪  constants.Itself);
//
//                 //var unaryNumberToAddressConverter = new
↪  UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);
//                 //var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links,
↪  unaryOne);
//                 //var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
↪  frequencyMarker, unaryOne, unaryNumberIncrementer);
//                 //var frequencyPropertyOperator = new FrequencyPropertyOperator<ulong>(links,
↪  frequencyPropertyMarker, frequencyMarker);
//                 //var linkFrequencyIncrementer = new LinkFrequencyIncrementer<ulong>(links,
↪  frequencyPropertyOperator, frequencyIncrementer);
//                 //var linkToItsFrequencyNumberConverter = new
↪  LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
↪  unaryNumberToAddressConverter);
//
//                 var linkFrequenciesCache3 = new
↪  LinkFrequenciesCache<ulong>(scope3.Links.Unsync, totalSequenceSymbolFrequencyCounter);
//
//                 var linkToItsFrequencyNumberConverter = new
↪  FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<ulong>(linkFrequenciesCache3);
//
//                 var sequenceToItsLocalElementLevelsConverter = new
↪  SequenceToItsLocalElementLevelsConverter<ulong>(scope3.Links.Unsync,
↪  linkToItsFrequencyNumberConverter);
//                 var optimalVariantConverter = new
↪  OptimalVariantConverter<ulong>(scope3.Links.Unsync,
↪  sequenceToItsLocalElementLevelsConverter);
//
//                 var compressed1 = new ulong[arrays.Length];
//                 var compressed2 = new ulong[arrays.Length];
//                 var compressed3 = new ulong[arrays.Length];
//
//                 var START = 0;
//                 var END = arrays.Length;
//
//                 //for (int i = START; i < END; i++)
//                 //    linkFrequenciesCache1.IncrementFrequencies(arrays[i]);
//
//                 var initialCount1 = scope2.Links.Unsync.Count();
//
//                 var sw1 = Stopwatch.StartNew();
```

```
//
//                 for (int i = START; i < END; i++)
//                 {
//                     linkFrequenciesCache1.IncrementFrequencies(arrays[i]);
//                     compressed1[i] = compressor1.Convert(arrays[i]);
//                 }
//
//                 var elapsed1 = sw1.Elapsed;
//
//                 var balancedVariantConverter2 = new
↪  BalancedVariantConverter<ulong>(scope2.Links.Unsync);
//
//                 var initialCount2 = scope2.Links.Unsync.Count();
//
//                 var sw2 = Stopwatch.StartNew();
//
//                 for (int i = START; i < END; i++)
//                 {
//                     compressed2[i] = balancedVariantConverter2.Convert(arrays[i]);
//                 }
//
//                 var elapsed2 = sw2.Elapsed;
//
//                 for (int i = START; i < END; i++)
//                 {
//                     linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
//                 }
//
//                 var initialCount3 = scope3.Links.Unsync.Count();
//
//                 var sw3 = Stopwatch.StartNew();
//
//                 for (int i = START; i < END; i++)
//                 {
//                     //linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
//                     compressed3[i] = optimalVariantConverter.Convert(arrays[i]);
//                 }
//
//                 var elapsed3 = sw3.Elapsed;
//
//                 Console.WriteLine($"Compressor: {elapsed1}, Balanced variant: {elapsed2},
↪  Optimal variant: {elapsed3}");
//
//                 // Assert.True(elapsed1 > elapsed2);
//
//                 // Checks
//                 for (int i = START; i < END; i++)
//                 {
//                     var sequence1 = compressed1[i];
//                     var sequence2 = compressed2[i];
//                     var sequence3 = compressed3[i];
//
//                     var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
↪  scope1.Links.Unsync);
//
//                     var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
↪  scope2.Links.Unsync);
//
//                     var decompress3 = UnicodeMap.FromSequenceLinkToString(sequence3,
↪  scope3.Links.Unsync);
//
//                     var structure1 = scope1.Links.Unsync.FormatStructure(sequence1, link =>
↪  link.IsPartialPoint());
//                     var structure2 = scope2.Links.Unsync.FormatStructure(sequence2, link =>
↪  link.IsPartialPoint());
//                     var structure3 = scope3.Links.Unsync.FormatStructure(sequence3, link =>
↪  link.IsPartialPoint());
//
//                     //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
↪  arrays[i].Length > 3)
//                     //    Assert.False(structure1 == structure2);
//                     //if (sequence3 != Constants.Null && sequence2 != Constants.Null &&
↪  arrays[i].Length > 3)
//                     //    Assert.False(structure3 == structure2);
//
//                     Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
//                     Assert.True(strings[i] == decompress3 && decompress3 == decompress2);
```

```
//                 }
//
//                 Assert.True((int)(scope1.Links.Unsync.Count() - initialCount1) <
↪  totalCharacters);
//                 Assert.True((int)(scope2.Links.Unsync.Count() - initialCount2) <
↪  totalCharacters);
//                 Assert.True((int)(scope3.Links.Unsync.Count() - initialCount3) <
↪  totalCharacters);
//
//                 Console.WriteLine($"{(double)(scope1.Links.Unsync.Count() - initialCount1) /
↪  totalCharacters} | {(double)(scope2.Links.Unsync.Count() - initialCount2) / totalCharacters}
↪  | {(double)(scope3.Links.Unsync.Count() - initialCount3) / totalCharacters}");
//
//                 Assert.True(scope1.Links.Unsync.Count() - initialCount1 <
↪  scope2.Links.Unsync.Count() - initialCount2);
//                 Assert.True(scope3.Links.Unsync.Count() - initialCount3 <
↪  scope2.Links.Unsync.Count() - initialCount2);
//
//                 var duplicateProvider1 = new
↪  DuplicateSegmentsProvider<ulong>(scope1.Links.Unsync, scope1.Sequences);
//                 var duplicateProvider2 = new
↪  DuplicateSegmentsProvider<ulong>(scope2.Links.Unsync, scope2.Sequences);
//                 var duplicateProvider3 = new
↪  DuplicateSegmentsProvider<ulong>(scope3.Links.Unsync, scope3.Sequences);
//
//                 var duplicateCounter1 = new
↪  DuplicateSegmentsCounter<ulong>(duplicateProvider1);
//                 var duplicateCounter2 = new
↪  DuplicateSegmentsCounter<ulong>(duplicateProvider2);
//                 var duplicateCounter3 = new
↪  DuplicateSegmentsCounter<ulong>(duplicateProvider3);
//
//                 var duplicates1 = duplicateCounter1.Count();
//
//                 ConsoleHelpers.Debug("------");
//
//                 var duplicates2 = duplicateCounter2.Count();
//
//                 ConsoleHelpers.Debug("------");
//
//                 var duplicates3 = duplicateCounter3.Count();
//
//                 Console.WriteLine($"{duplicates1} | {duplicates2} | {duplicates3}");
//
//                 linkFrequenciesCache1.ValidateFrequencies();
//                 linkFrequenciesCache3.ValidateFrequencies();
//             }
//         }
//
//         [Fact]
//         public static void CompressionStabilityTest()
//         {
//             // TODO: Fix bug (do a separate test)
//             //const ulong minNumbers = 0;
//             //const ulong maxNumbers = 1000;
//
//             const ulong minNumbers = 10000;
//             const ulong maxNumbers = 12500;
//
//             var strings = new List<string>();
//
//             for (ulong i = minNumbers; i < maxNumbers; i++)
//             {
//                 strings.Add(i.ToString());
//             }
//
//             var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
//             var totalCharacters = arrays.Select(x => x.Length).Sum();
//
//             using (var scope1 = new TempLinksTestScope(useSequences: true, sequencesOptions:
↪  new SequencesOptions<ulong> { UseCompression = true,
↪  EnforceSingleSequenceVersionOnWriteBasedOnExisting = true }))
//             using (var scope2 = new TempLinksTestScope(useSequences: true))
//             {
//                 scope1.Links.UseUnicode();
//                 scope2.Links.UseUnicode();
```

```
671 //
672 //                  //var compressor1 = new Compressor(scope1.Links.Unsync, scope1.Sequences);
673 //                  var compressor1 = scope1.Sequences;
674 //                  var compressor2 = scope2.Sequences;
675 //
676 //                  var compressed1 = new ulong[arrays.Length];
677 //                  var compressed2 = new ulong[arrays.Length];
678 //
679 //                  var sw1 = Stopwatch.StartNew();
680 //
681 //                  var START = 0;
682 //                  var END = arrays.Length;
683 //
684 //                  // Collisions proved (cannot be solved by max doublet comparison, no stable
    ↪  rule)
685 //                  // Stability issue starts at 10001 or 11000
686 //                  //for (int i = START; i < END; i++)
687 //                  //{
688 //                  //    var first = compressor1.Compress(arrays[i]);
689 //                  //    var second = compressor1.Compress(arrays[i]);
690 //
691 //                  //    if (first == second)
692 //                  //        compressed1[i] = first;
693 //                  //    else
694 //                  //    {
695 //                  //        // TODO: Find a solution for this case
696 //                  //    }
697 //                  //}
698 //
699 //                  for (int i = START; i < END; i++)
700 //                  {
701 //                      var first = compressor1.Create(arrays[i].ShiftRight());
702 //                      var second = compressor1.Create(arrays[i].ShiftRight());
703 //
704 //                      if (first == second)
705 //                      {
706 //                          compressed1[i] = first;
707 //                      }
708 //                      else
709 //                      {
710 //                          // TODO: Find a solution for this case
711 //                      }
712 //                  }
713 //
714 //                  var elapsed1 = sw1.Elapsed;
715 //
716 //                  var balancedVariantConverter = new
    ↪  BalancedVariantConverter<ulong>(scope2.Links);
717 //
718 //                  var sw2 = Stopwatch.StartNew();
719 //
720 //                  for (int i = START; i < END; i++)
721 //                  {
722 //                      var first = balancedVariantConverter.Convert(arrays[i]);
723 //                      var second = balancedVariantConverter.Convert(arrays[i]);
724 //
725 //                      if (first == second)
726 //                      {
727 //                          compressed2[i] = first;
728 //                      }
729 //                  }
730 //
731 //                  var elapsed2 = sw2.Elapsed;
732 //
733 //                  Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator:
    ↪  {elapsed2}");
734 //
735 //                  Assert.True(elapsed1 > elapsed2);
736 //
737 //                  // Checks
738 //                  for (int i = START; i < END; i++)
739 //                  {
740 //                      var sequence1 = compressed1[i];
741 //                      var sequence2 = compressed2[i];
742 //
743 //                      if (sequence1 != _constants.Null && sequence2 != _constants.Null)
744 //                      {
```

```
//                          var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
↪ scope1.Links);
//
//                          var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
↪ scope2.Links);
//
//                          //var structure1 = scope1.Links.FormatStructure(sequence1, link =>
↪ link.IsPartialPoint());
//                          //var structure2 = scope2.Links.FormatStructure(sequence2, link =>
↪ link.IsPartialPoint());
//
//                          //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
↪ arrays[i].Length > 3)
//                          //    Assert.False(structure1 == structure2);
//
//                          Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
//                      }
//                  }
//
//                  Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) <
↪ totalCharacters);
//                  Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) <
↪ totalCharacters);
//
//                  Debug.WriteLine($"{(double)(scope1.Links.Count() - UnicodeMap.MapSize) /
↪ totalCharacters} | {(double)(scope2.Links.Count() - UnicodeMap.MapSize) / totalCharacters}");
//
//                  Assert.True(scope1.Links.Count() <= scope2.Links.Count());
//
//                  //compressor1.ValidateFrequencies();
//              }
//          }
//
//          [Fact]
//          public static void RundomNumbersCompressionQualityTest()
//          {
//              const ulong N = 500;
//
//              //const ulong minNumbers = 10000;
//              //const ulong maxNumbers = 20000;
//
//              //var strings = new List<string>();
//
//              //for (ulong i = 0; i < N; i++)
//              //    strings.Add(RandomHelpers.DefaultFactory.NextUInt64(minNumbers,
↪ maxNumbers).ToString());
//
//              var strings = new List<string>();
//
//              for (ulong i = 0; i < N; i++)
//              {
//                  strings.Add(RandomHelpers.Default.NextUInt64().ToString());
//              }
//
//              strings = strings.Distinct().ToList();
//
//              var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
//              var totalCharacters = arrays.Select(x => x.Length).Sum();
//
//              using (var scope1 = new TempLinksTestScope(useSequences: true, sequencesOptions:
↪ new SequencesOptions<ulong> { UseCompression = true,
↪ EnforceSingleSequenceVersionOnWriteBasedOnExisting = true }))
//              using (var scope2 = new TempLinksTestScope(useSequences: true))
//              {
//                  scope1.Links.UseUnicode();
//                  scope2.Links.UseUnicode();
//
//                  var compressor1 = scope1.Sequences;
//                  var compressor2 = scope2.Sequences;
//
//                  var compressed1 = new ulong[arrays.Length];
//                  var compressed2 = new ulong[arrays.Length];
//
//                  var sw1 = Stopwatch.StartNew();
//
//                  var START = 0;
//                  var END = arrays.Length;
```

```
//
//                 for (int i = START; i < END; i++)
//                 {
//                     compressed1[i] = compressor1.Create(arrays[i].ShiftRight());
//                 }
//
//                 var elapsed1 = sw1.Elapsed;
//
//                 var balancedVariantConverter = new
↪  BalancedVariantConverter<ulong>(scope2.Links);
//
//                 var sw2 = Stopwatch.StartNew();
//
//                 for (int i = START; i < END; i++)
//                 {
//                     compressed2[i] = balancedVariantConverter.Convert(arrays[i]);
//                 }
//
//                 var elapsed2 = sw2.Elapsed;
//
//                 Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator:
↪  {elapsed2}");
//
//                 Assert.True(elapsed1 > elapsed2);
//
//                 // Checks
//                 for (int i = START; i < END; i++)
//                 {
//                     var sequence1 = compressed1[i];
//                     var sequence2 = compressed2[i];
//
//                     if (sequence1 != _constants.Null && sequence2 != _constants.Null)
//                     {
//                         var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
↪  scope1.Links);
//
//                         var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
↪  scope2.Links);
//
//                         Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
//                     }
//                 }
//
//                 Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) <
↪  totalCharacters);
//                 Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) <
↪  totalCharacters);
//
//                 Debug.WriteLine($"{(double)(scope1.Links.Count() - UnicodeMap.MapSize) /
↪  totalCharacters} | {(double)(scope2.Links.Count() - UnicodeMap.MapSize) / totalCharacters}");
//
//                 // Can be worse than balanced variant
//                 //Assert.True(scope1.Links.Count() <= scope2.Links.Count());
//
//                 //compressor1.ValidateFrequencies();
//             }
//         }
//
//         [Fact]
//         public static void AllTreeBreakDownAtSequencesCreationBugTest()
//         {
//             // Made out of AllPossibleConnectionsTest test.
//
//             //const long sequenceLength = 5; //100% bug
//             const long sequenceLength = 4; //100% bug
//             //const long sequenceLength = 3; //100% _no_bug (ok)
//
//             using (var scope = new TempLinksTestScope(useSequences: true))
//             {
//                 var links = scope.Links;
//                 var sequences = scope.Sequences;
//
//                 var sequence = new ulong[sequenceLength];
//                 for (var i = 0; i < sequenceLength; i++)
//                 {
//                     sequence[i] = links.Create();
//                 }
```

```
//
//                  var createResults = sequences.CreateAllVariants2(sequence);
//
//                  Global.Trash = createResults;
//
//                  for (var i = 0; i < sequenceLength; i++)
//                  {
//                      links.Delete(sequence[i]);
//                  }
//              }
//          }
//
//          [Fact]
//          public static void AllPossibleConnectionsTest()
//          {
//              const long sequenceLength = 5;
//
//              using (var scope = new TempLinksTestScope(useSequences: true))
//              {
//                  var links = scope.Links;
//                  var sequences = scope.Sequences;
//
//                  var sequence = new ulong[sequenceLength];
//                  for (var i = 0; i < sequenceLength; i++)
//                  {
//                      sequence[i] = links.Create();
//                  }
//
//                  var createResults = sequences.CreateAllVariants2(sequence);
//                  var reverseResults =
    ↪ sequences.CreateAllVariants2(sequence.Reverse().ToArray());
//
//                  for (var i = 0; i < 1; i++)
//                  {
//                      var sw1 = Stopwatch.StartNew();
//                      var searchResults1 = sequences.GetAllConnections(sequence); sw1.Stop();
//
//                      var sw2 = Stopwatch.StartNew();
//                      var searchResults2 = sequences.GetAllConnections1(sequence); sw2.Stop();
//
//                      var sw3 = Stopwatch.StartNew();
//                      var searchResults3 = sequences.GetAllConnections2(sequence); sw3.Stop();
//
//                      var sw4 = Stopwatch.StartNew();
//                      var searchResults4 = sequences.GetAllConnections3(sequence); sw4.Stop();
//
//                      Global.Trash = searchResults3;
//                      Global.Trash = searchResults4; //-V3008
//
//                      var intersection1 = createResults.Intersect(searchResults1).ToList();
//                      Assert.True(intersection1.Count == createResults.Length);
//
//                      var intersection2 = reverseResults.Intersect(searchResults1).ToList();
//                      Assert.True(intersection2.Count == reverseResults.Length);
//
//                      var intersection0 = searchResults1.Intersect(searchResults2).ToList();
//                      Assert.True(intersection0.Count == searchResults2.Count);
//
//                      var intersection3 = searchResults2.Intersect(searchResults3).ToList();
//                      Assert.True(intersection3.Count == searchResults3.Count);
//
//                      var intersection4 = searchResults3.Intersect(searchResults4).ToList();
//                      Assert.True(intersection4.Count == searchResults4.Count);
//                  }
//
//                  for (var i = 0; i < sequenceLength; i++)
//                  {
//                      links.Delete(sequence[i]);
//                  }
//              }
//          }
//
//          [Fact(Skip = "Correct implementation is pending")]
//          public static void CalculateAllUsagesTest()
//          {
//              const long sequenceLength = 3;
//
```

```
//             using (var scope = new TempLinksTestScope(useSequences: true))
//             {
//                 var links = scope.Links;
//                 var sequences = scope.Sequences;
//
//                 var sequence = new ulong[sequenceLength];
//                 for (var i = 0; i < sequenceLength; i++)
//                 {
//                     sequence[i] = links.Create();
//                 }
//
//                 var createResults = sequences.CreateAllVariants2(sequence);
//
//                 //var reverseResults =
↪  sequences.CreateAllVariants2(sequence.Reverse().ToArray());
//
//                 for (var i = 0; i < 1; i++)
//                 {
//                     var linksTotalUsages1 = new ulong[links.Count() + 1];
//
//                     sequences.CalculateAllUsages(linksTotalUsages1);
//
//                     var linksTotalUsages2 = new ulong[links.Count() + 1];
//
//                     sequences.CalculateAllUsages2(linksTotalUsages2);
//
//                     var intersection1 =
↪  linksTotalUsages1.Intersect(linksTotalUsages2).ToList();
//                     Assert.True(intersection1.Count == linksTotalUsages2.Length);
//                 }
//
//                 for (var i = 0; i < sequenceLength; i++)
//                 {
//                     links.Delete(sequence[i]);
//                 }
//             }
//         }
//     }
// }
```

1.73 ./csharp/Platform.Data.Doublets.Sequences.Tests/TempLinksTestScope.cs

```
// using System.IO;
// using Platform.Disposables;
// using Platform.Data.Doublets.Sequences;
// using Platform.Data.Doublets.Decorators;
// using Platform.Data.Doublets.Memory.United.Specific;
// using Platform.Data.Doublets.Memory.Split.Specific;
// using Platform.Memory;
//
// namespace Platform.Data.Doublets.Sequences.Tests
// {
//     public class TempLinksTestScope : DisposableBase
//     {
//         public ILinks<ulong> MemoryAdapter { get; }
//         public SynchronizedLinks<ulong> Links { get; }
//         public Sequences Sequences { get; }
//         public string TempFilename { get; }
//         public string TempTransactionLogFilename { get; }
//         private readonly bool _deleteFiles;
//
//         public TempLinksTestScope(bool deleteFiles = true, bool useSequences = false, bool
↪  useLog = false) : this(new SequencesOptions<ulong>(), deleteFiles, useSequences, useLog) { }
//
//         public TempLinksTestScope(SequencesOptions<ulong> sequencesOptions, bool deleteFiles
↪  = true, bool useSequences = false, bool useLog = false)
//         {
//             _deleteFiles = deleteFiles;
//             TempFilename = Path.GetTempFileName();
//             TempTransactionLogFilename = Path.GetTempFileName();
//             //var coreMemoryAdapter = new UInt64UnitedMemoryLinks(TempFilename);
//             var coreMemoryAdapter = new UInt64SplitMemoryLinks(new
↪  FileMappedResizableDirectMemory(TempFilename), new
↪  FileMappedResizableDirectMemory(Path.ChangeExtension(TempFilename, "indexes")),
↪  UInt64SplitMemoryLinks.DefaultLinksSizeStep, new LinksConstants<ulong>(),
↪  Memory.IndexTreeType.Default, useLinkedList: true);
```

```
//             MemoryAdapter = useLog ? (ILinks<ulong>)new
↪  UInt64LinksTransactionsLayer(coreMemoryAdapter, TempTransactionLogFilename) :
↪  coreMemoryAdapter;
//             Links = new SynchronizedLinks<ulong>(new UInt64Links(MemoryAdapter));
//             if (useSequences)
//             {
//                 Sequences = new Sequences(Links, sequencesOptions);
//             }
//         }
//
//         protected override void Dispose(bool manual, bool wasDisposed)
//         {
//             if (!wasDisposed)
//             {
//                 Links.Unsync.DisposeIfPossible();
//                 if (_deleteFiles)
//                 {
//                     DeleteFiles();
//                 }
//             }
//         }
//
//         public void DeleteFiles()
//         {
//             File.Delete(TempFilename);
//             File.Delete(TempTransactionLogFilename);
//         }
//     }
// }
```

1.74 ./csharp/Platform.Data.Doublets.Sequences.Tests/TestExtensions.cs

```
// using System.Collections.Generic;
// using Xunit;
// using Platform.Ranges;
// using Platform.Numbers;
// using Platform.Random;
// using Platform.Setters;
// using Platform.Converters;
//
// namespace Platform.Data.Doublets.Sequences.Tests
// {
//     public static class TestExtensions
//     {
//         public static void TestCRUDOperations<T>(this ILinks<T> links)
//         {
//             var constants = links.Constants;
//
//             var equalityComparer = EqualityComparer<T>.Default;
//
//             var zero = default(T);
//             var one = Arithmetic.Increment(zero);
//
//             // Create Link
//             Assert.True(equalityComparer.Equals(links.Count(), zero));
//
//             var setter = new Setter<T>(constants.Null);
//             links.Each(constants.Any, constants.Any, setter.SetAndReturnTrue);
//
//             Assert.True(equalityComparer.Equals(setter.Result, constants.Null));
//
//             var linkAddress = links.Create();
//
//             var link = new Link<T>(links.GetLink(linkAddress));
//
//             Assert.True(link.Count == 3);
//             Assert.True(equalityComparer.Equals(link.Index, linkAddress));
//             Assert.True(equalityComparer.Equals(link.Source, constants.Null));
//             Assert.True(equalityComparer.Equals(link.Target, constants.Null));
//
//             Assert.True(equalityComparer.Equals(links.Count(), one));
//
//             // Get first link
//             setter = new Setter<T>(constants.Null);
//             links.Each(constants.Any, constants.Any, setter.SetAndReturnFalse);
//
//             Assert.True(equalityComparer.Equals(setter.Result, linkAddress));
//
//             // Update link to reference itself
```

```
//              links.Update(linkAddress, linkAddress, linkAddress);
//
//              link = new Link<T>(links.GetLink(linkAddress));
//
//              Assert.True(equalityComparer.Equals(link.Source, linkAddress));
//              Assert.True(equalityComparer.Equals(link.Target, linkAddress));
//
//              // Update link to reference null (prepare for delete)
//              var updated = links.Update(linkAddress, constants.Null, constants.Null);
//
//              Assert.True(equalityComparer.Equals(updated, linkAddress));
//
//              link = new Link<T>(links.GetLink(linkAddress));
//
//              Assert.True(equalityComparer.Equals(link.Source, constants.Null));
//              Assert.True(equalityComparer.Equals(link.Target, constants.Null));
//
//              // Delete link
//              links.Delete(linkAddress);
//
//              Assert.True(equalityComparer.Equals(links.Count(), zero));
//
//              setter = new Setter<T>(constants.Null);
//              links.Each(constants.Any, constants.Any, setter.SetAndReturnTrue);
//
//              Assert.True(equalityComparer.Equals(setter.Result, constants.Null));
//          }
//
//          public static void TestRawNumbersCRUDOperations<T>(this ILinks<T> links)
//          {
//              // Constants
//              var constants = links.Constants;
//              var equalityComparer = EqualityComparer<T>.Default;
//
//              var zero = default(T);
//              var one = Arithmetic.Increment(zero);
//              var two = Arithmetic.Increment(one);
//
//              var h106E = new Hybrid<T>(106L, isExternal: true);
//              var h107E = new Hybrid<T>(-char.ConvertFromUtf32(107)[0]);
//              var h108E = new Hybrid<T>(-108L);
//
//              Assert.Equal(106L, h106E.AbsoluteValue);
//              Assert.Equal(107L, h107E.AbsoluteValue);
//              Assert.Equal(108L, h108E.AbsoluteValue);
//
//              // Create Link (External -> External)
//              var linkAddress1 = links.Create();
//
//              links.Update(linkAddress1, h106E, h108E);
//
//              var link1 = new Link<T>(links.GetLink(linkAddress1));
//
//              Assert.True(equalityComparer.Equals(link1.Source, h106E));
//              Assert.True(equalityComparer.Equals(link1.Target, h108E));
//
//              // Create Link (Internal -> External)
//              var linkAddress2 = links.Create();
//
//              links.Update(linkAddress2, linkAddress1, h108E);
//
//              var link2 = new Link<T>(links.GetLink(linkAddress2));
//
//              Assert.True(equalityComparer.Equals(link2.Source, linkAddress1));
//              Assert.True(equalityComparer.Equals(link2.Target, h108E));
//
//              // Create Link (Internal -> Internal)
//              var linkAddress3 = links.Create();
//
//              links.Update(linkAddress3, linkAddress1, linkAddress2);
//
//              var link3 = new Link<T>(links.GetLink(linkAddress3));
//
//              Assert.True(equalityComparer.Equals(link3.Source, linkAddress1));
//              Assert.True(equalityComparer.Equals(link3.Target, linkAddress2));
//
//              // Search for created link
//              var setter1 = new Setter<T>(constants.Null);
```

```
126 //             links.Each(h106E, h108E, setter1.SetAndReturnFalse);
127 //
128 //             Assert.True(equalityComparer.Equals(setter1.Result, linkAddress1));
129 //
130 //             // Search for nonexistent link
131 //             var setter2 = new Setter<T>(constants.Null);
132 //             links.Each(h106E, h107E, setter2.SetAndReturnFalse);
133 //
134 //             Assert.True(equalityComparer.Equals(setter2.Result, constants.Null));
135 //
136 //             // Update link to reference null (prepare for delete)
137 //             var updated = links.Update(linkAddress3, constants.Null, constants.Null);
138 //
139 //             Assert.True(equalityComparer.Equals(updated, linkAddress3));
140 //
141 //             link3 = new Link<T>(links.GetLink(linkAddress3));
142 //
143 //             Assert.True(equalityComparer.Equals(link3.Source, constants.Null));
144 //             Assert.True(equalityComparer.Equals(link3.Target, constants.Null));
145 //
146 //             // Delete link
147 //             links.Delete(linkAddress3);
148 //
149 //             Assert.True(equalityComparer.Equals(links.Count(), two));
150 //
151 //             var setter3 = new Setter<T>(constants.Null);
152 //             links.Each(constants.Any, constants.Any, setter3.SetAndReturnTrue);
153 //
154 //             Assert.True(equalityComparer.Equals(setter3.Result, linkAddress2));
155 //         }
156 //
157 //         public static void TestMultipleRandomCreationsAndDeletions<TLinkAddress>(this
    ↪  ILinks<TLinkAddress> links, int maximumOperationsPerCycle)
158 //         {
159 //             var comparer = Comparer<TLinkAddress>.Default;
160 //             var addressToUInt64Converter = CheckedConverter<TLinkAddress, ulong>.Default;
161 //             var uInt64ToAddressConverter = CheckedConverter<ulong, TLinkAddress>.Default;
162 //             for (var N = 1; N < maximumOperationsPerCycle; N++)
163 //             {
164 //                 var random = new System.Random(N);
165 //                 var created = 0UL;
166 //                 var deleted = 0UL;
167 //                 for (var i = 0; i < N; i++)
168 //                 {
169 //                     var linksCount = addressToUInt64Converter.Convert(links.Count());
170 //                     var createPoint = random.NextBoolean();
171 //                     if (linksCount >= 2 && createPoint)
172 //                     {
173 //                         var linksAddressRange = new Range<ulong>(1, linksCount);
174 //                         TLinkAddress source =
    ↪  uInt64ToAddressConverter.Convert(random.NextUInt64(linksAddressRange));
175 //                         TLinkAddress target =
    ↪  uInt64ToAddressConverter.Convert(random.NextUInt64(linksAddressRange)); //-V3086
176 //                         var resultLink = links.GetOrCreate(source, target);
177 //                         if (comparer.Compare(resultLink,
    ↪  uInt64ToAddressConverter.Convert(linksCount)) > 0)
178 //                         {
179 //                             created++;
180 //                         }
181 //                     }
182 //                     else
183 //                     {
184 //                         links.Create();
185 //                         created++;
186 //                     }
187 //                 }
188 //                 Assert.True(created == addressToUInt64Converter.Convert(links.Count()));
189 //                 for (var i = 0; i < N; i++)
190 //                 {
191 //                     TLinkAddress link = uInt64ToAddressConverter.Convert((ulong)i + 1UL);
192 //                     if (links.Exists(link))
193 //                     {
194 //                         links.Delete(link);
195 //                         deleted++;
196 //                     }
197 //                 }
198 //                 Assert.True(addressToUInt64Converter.Convert(links.Count()) == 0L);
```

```
//             }
//         }
//     }
// }
```

1.75 ./csharp/Platform.Data.Doublets.Sequences.Tests/UInt64LinksTests.cs

```
// using System;
// using System.Collections.Generic;
// using System.Diagnostics;
// using System.IO;
// using System.Text;
// using System.Threading;
// using System.Threading.Tasks;
// using Xunit;
// using Platform.Disposables;
// using Platform.Ranges;
// using Platform.Random;
// using Platform.Timestamps;
// using Platform.Reflection;
// using Platform.Singletons;
// using Platform.Scopes;
// using Platform.Counters;
// using Platform.Diagnostics;
// using Platform.IO;
// using Platform.Memory;
// using Platform.Data.Doublets.Decorators;
// using Platform.Data.Doublets.Memory.United.Specific;
//
// namespace Platform.Data.Doublets.Sequences.Tests
// {
//     public static class UInt64LinksTests
//     {
//         private static readonly LinksConstants<ulong> _constants =
    ↪  Default<LinksConstants<ulong>>.Instance;
//         private const long Iterations = 10 * 1024;
//
//         #region Concept
//
//         [Fact]
//         public static void MultipleCreateAndDeleteTest()
//         {
//             using (var scope = new Scope<Types<HeapResizableDirectMemory,
    ↪  UInt64UnitedMemoryLinks>>())
//             {
//                 new
    ↪  UInt64Links(scope.Use<ILinks<ulong>>()).TestMultipleRandomCreationsAndDeletions(100);
//             }
//         }
//
//         [Fact]
//         public static void CascadeUpdateTest()
//         {
//             var itself = _constants.Itself;
//             using (var scope = new TempLinksTestScope(useLog: true))
//             {
//                 var links = scope.Links;
//
//                 var l1 = links.Create();
//                 var l2 = links.Create();
//
//                 l2 = links.Update(l2, l2, l1, l2);
//
//                 links.CreateAndUpdate(l2, itself);
//                 links.CreateAndUpdate(l2, itself);
//
//                 l2 = links.Update(l2, l1);
//
//                 links.Delete(l2);
//
//                 Global.Trash = links.Count();
//
//                 links.Unsync.DisposeIfPossible(); // Close links to access log
//
//                 Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(s
    ↪  cope.TempTransactionLogFilename);
//             }
//         }
```

```
//
//         [Fact]
//         public static void BasicTransactionLogTest()
//         {
//             using (var scope = new TempLinksTestScope(useLog: true))
//             {
//                 var links = scope.Links;
//                 var l1 = links.Create();
//                 var l2 = links.Create();
//
//                 Global.Trash = links.Update(l2, l2, l1, l2);
//
//                 links.Delete(l1);
//
//                 links.Unsync.DisposeIfPossible(); // Close links to access log
//
//                 Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(s
↪ cope.TempTransactionLogFilename);
//             }
//         }
//
//         [Fact]
//         public static void TransactionAutoRevertedTest()
//         {
//             // Auto Reverted (Because no commit at transaction)
//             using (var scope = new TempLinksTestScope(useLog: true))
//             {
//                 var links = scope.Links;
//                 var transactionsLayer = (UInt64LinksTransactionsLayer)scope.MemoryAdapter;
//                 using (var transaction = transactionsLayer.BeginTransaction())
//                 {
//                     var l1 = links.Create();
//                     var l2 = links.Create();
//
//                     links.Update(l2, l2, l1, l2);
//                 }
//
//                 Assert.Equal(0UL, links.Count());
//
//                 links.Unsync.DisposeIfPossible();
//
//                 var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition
↪ >(scope.TempTransactionLogFilename);
//                 Assert.Single(transitions);
//             }
//         }
//
//         [Fact]
//         public static void TransactionUserCodeErrorNoDataSavedTest()
//         {
//             // User Code Error (Autoreverted), no data saved
//             var itself = _constants.Itself;
//
//             TempLinksTestScope lastScope = null;
//             try
//             {
//                 using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
↪ useLog: true))
//                 {
//                     var links = scope.Links;
//                     var transactionsLayer =
↪ (UInt64LinksTransactionsLayer)((LinksDisposableDecoratorBase<ulong>)links.Unsync).Links;
//                     using (var transaction = transactionsLayer.BeginTransaction())
//                     {
//                         var l1 = links.CreateAndUpdate(itself, itself);
//                         var l2 = links.CreateAndUpdate(itself, itself);
//
//                         l2 = links.Update(l2, l2, l1, l2);
//
//                         links.CreateAndUpdate(l2, itself);
//                         links.CreateAndUpdate(l2, itself);
//
//                         //Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Tra
↪ nsition>(scope.TempTransactionLogFilename);
//
//                         l2 = links.Update(l2, l1);
//
```

```
//                          links.Delete(l2);
//
//                          ExceptionThrower();
//
//                          transaction.Commit();
//                      }
//
//                      Global.Trash = links.Count();
//                  }
//              }
//              catch
//              {
//                  Assert.False(lastScope == null);
//
//                  var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition
↪ >(lastScope.TempTransactionLogFilename);
//
//                  Assert.True(transitions.Length == 1 && transitions[0].Before.IsNull() &&
↪ transitions[0].After.IsNull());
//
//                  lastScope.DeleteFiles();
//              }
//          }
//
//          [Fact]
//          public static void TransactionUserCodeErrorSomeDataSavedTest()
//          {
//              // User Code Error (Autoreverted), some data saved
//              var itself = _constants.Itself;
//
//              TempLinksTestScope lastScope = null;
//              try
//              {
//                  ulong l1;
//                  ulong l2;
//
//                  using (var scope = new TempLinksTestScope(useLog: true))
//                  {
//                      var links = scope.Links;
//                      l1 = links.CreateAndUpdate(itself, itself);
//                      l2 = links.CreateAndUpdate(itself, itself);
//
//                      l2 = links.Update(l2, l2, l1, l2);
//
//                      links.CreateAndUpdate(l2, itself);
//                      links.CreateAndUpdate(l2, itself);
//
//                      links.Unsync.DisposeIfPossible();
//
//                      Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transitio
↪ n>(scope.TempTransactionLogFilename);
//                  }
//
//                  using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
↪ useLog: true))
//                  {
//                      var links = scope.Links;
//                      var transactionsLayer = (UInt64LinksTransactionsLayer)links.Unsync;
//                      using (var transaction = transactionsLayer.BeginTransaction())
//                      {
//                          l2 = links.Update(l2, l1);
//
//                          links.Delete(l2);
//
//                          ExceptionThrower();
//
//                          transaction.Commit();
//                      }
//
//                      Global.Trash = links.Count();
//                  }
//              }
//              catch
//              {
//                  Assert.False(lastScope == null);
//
```

```
//                 Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(l
↪ astScope.TempTransactionLogFilename);
//
//                 lastScope.DeleteFiles();
//             }
//         }
//
//         [Fact]
//         public static void TransactionCommit()
//         {
//             var itself = _constants.Itself;
//
//             var tempDatabaseFilename = Path.GetTempFileName();
//             var tempTransactionLogFilename = Path.GetTempFileName();
//
//             // Commit
//             using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
↪ UInt64UnitedMemoryLinks(tempDatabaseFilename), tempTransactionLogFilename))
//             using (var links = new UInt64Links(memoryAdapter))
//             {
//                 using (var transaction = memoryAdapter.BeginTransaction())
//                 {
//                     var l1 = links.CreateAndUpdate(itself, itself);
//                     var l2 = links.CreateAndUpdate(itself, itself);
//
//                     Global.Trash = links.Update(l2, l2, l1, l2);
//
//                     links.Delete(l1);
//
//                     transaction.Commit();
//                 }
//
//                 Global.Trash = links.Count();
//             }
//
//             Global.Trash =
↪ FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTransactionLogFilename);
//         }
//
//         [Fact]
//         public static void TransactionDamage()
//         {
//             var itself = _constants.Itself;
//
//             var tempDatabaseFilename = Path.GetTempFileName();
//             var tempTransactionLogFilename = Path.GetTempFileName();
//
//             // Commit
//             using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
↪ UInt64UnitedMemoryLinks(tempDatabaseFilename), tempTransactionLogFilename))
//             using (var links = new UInt64Links(memoryAdapter))
//             {
//                 using (var transaction = memoryAdapter.BeginTransaction())
//                 {
//                     var l1 = links.CreateAndUpdate(itself, itself);
//                     var l2 = links.CreateAndUpdate(itself, itself);
//
//                     Global.Trash = links.Update(l2, l2, l1, l2);
//
//                     links.Delete(l1);
//
//                     transaction.Commit();
//                 }
//
//                 Global.Trash = links.Count();
//             }
//
//             Global.Trash =
↪ FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTransactionLogFilename);
//
//             // Damage database
//
//             FileHelpers.WriteFirst(tempTransactionLogFilename, new
↪ UInt64LinksTransactionsLayer.Transition(new UniqueTimestampFactory(), 555));
//
//             // Try load damaged database
//             try
```

```
//             {
//                 // TODO: Fix
//                 using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
↪  UInt64UnitedMemoryLinks(tempDatabaseFilename), tempTransactionLogFilename))
//                 using (var links = new UInt64Links(memoryAdapter))
//                 {
//                     Global.Trash = links.Count();
//                 }
//             }
//             catch (NotSupportedException ex)
//             {
//                 Assert.True(ex.Message == "Database is damaged, autorecovery is not supported
↪  yet.");
//             }
//
//             Global.Trash =
↪  FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTransactionLogFilename);
//
//             File.Delete(tempDatabaseFilename);
//             File.Delete(tempTransactionLogFilename);
//         }
//
//         [Fact]
//         public static void Bug1Test()
//         {
//             var tempDatabaseFilename = Path.GetTempFileName();
//             var tempTransactionLogFilename = Path.GetTempFileName();
//
//             var itself = _constants.Itself;
//
//             // User Code Error (Autoreverted), some data saved
//             try
//             {
//                 ulong l1;
//                 ulong l2;
//
//                 using (var memory = new UInt64UnitedMemoryLinks(tempDatabaseFilename))
//                 using (var memoryAdapter = new UInt64LinksTransactionsLayer(memory,
↪  tempTransactionLogFilename))
//                 using (var links = new UInt64Links(memoryAdapter))
//                 {
//                     l1 = links.CreateAndUpdate(itself, itself);
//                     l2 = links.CreateAndUpdate(itself, itself);
//
//                     l2 = links.Update(l2, l2, l1, l2);
//
//                     links.CreateAndUpdate(l2, itself);
//                     links.CreateAndUpdate(l2, itself);
//                 }
//
//                 Global.Trash =
↪  FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTransactionLogFilename);
//
//                 using (var memory = new UInt64UnitedMemoryLinks(tempDatabaseFilename))
//                 using (var memoryAdapter = new UInt64LinksTransactionsLayer(memory,
↪  tempTransactionLogFilename))
//                 using (var links = new UInt64Links(memoryAdapter))
//                 {
//                     using (var transaction = memoryAdapter.BeginTransaction())
//                     {
//                         l2 = links.Update(l2, l1);
//
//                         links.Delete(l2);
//
//                         ExceptionThrower();
//
//                         transaction.Commit();
//                     }
//
//                     Global.Trash = links.Count();
//                 }
//             }
//             catch
//             {
//                 Global.Trash =
↪  FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTransactionLogFilename);
//             }
```

```
//
//             File.Delete(tempDatabaseFilename);
//             File.Delete(tempTransactionLogFilename);
//         }
//         private static void ExceptionThrower() => throw new InvalidOperationException();
//
//         [Fact]
//         public static void PathsTest()
//         {
//             var source = _constants.SourcePart;
//             var target = _constants.TargetPart;
//
//             using (var scope = new TempLinksTestScope())
//             {
//                 var links = scope.Links;
//                 var l1 = links.CreatePoint();
//                 var l2 = links.CreatePoint();
//
//                 var r1 = links.GetByKeys(l1, source, target, source);
//                 var r2 = links.CheckPathExistance(l2, l2, l2, l2);
//             }
//         }
//
//         [Fact]
//         public static void RecursiveStringFormattingTest()
//         {
//             using (var scope = new TempLinksTestScope(useSequences: true))
//             {
//                 var links = scope.Links;
//                 var sequences = scope.Sequences; // TODO: Auto use sequences on Sequences getter.
//
//                 var a = links.CreatePoint();
//                 var b = links.CreatePoint();
//                 var c = links.CreatePoint();
//
//                 var ab = links.GetOrCreate(a, b);
//                 var cb = links.GetOrCreate(c, b);
//                 var ac = links.GetOrCreate(a, c);
//
//                 a = links.Update(a, c, b);
//                 b = links.Update(b, a, c);
//                 c = links.Update(c, a, b);
//
//                 Debug.WriteLine(links.FormatStructure(ab, link => link.IsFullPoint(), true));
//                 Debug.WriteLine(links.FormatStructure(cb, link => link.IsFullPoint(), true));
//                 Debug.WriteLine(links.FormatStructure(ac, link => link.IsFullPoint(), true));
//
//                 Assert.True(links.FormatStructure(cb, link => link.IsFullPoint(), true) == "(5:(4:5 (6:5 4)) 6)");
//                 Assert.True(links.FormatStructure(ac, link => link.IsFullPoint(), true) == "(6:(5:(4:5 6) 6) 4)");
//                 Assert.True(links.FormatStructure(ab, link => link.IsFullPoint(), true) == "(4:(5:4 (6:5 4)) 6)");
//
//                 // TODO: Think how to build balanced syntax tree while formatting structure (eg. "(4:(5:4 6) (6:5 4)") instead of "(4:(5:4 (6:5 4)) 6)"
//
//                 Assert.True(sequences.SafeFormatSequence(cb, DefaultFormatter, false) == "{{5}{5}{4}{6}}");
//                 Assert.True(sequences.SafeFormatSequence(ac, DefaultFormatter, false) == "{{5}{6}{6}{4}}");
//                 Assert.True(sequences.SafeFormatSequence(ab, DefaultFormatter, false) == "{{4}{5}{4}{6}}");
//             }
//         }
//         private static void DefaultFormatter(StringBuilder sb, ulong link)
//         {
//             sb.Append(link.ToString());
//         }
//
//         #endregion
//
//         #region Performance
//
//         /*
//         public static void RunAllPerformanceTests()
```

```
//        {
//            try
//            {
//                links.TestLinksInSteps();
//            }
//            catch (Exception ex)
//            {
//                ex.WriteToConsole();
//            }
//
//            return;
//
//            try
//            {
//                //ThreadPool.SetMaxThreads(2, 2);
//
//                // Запускаем все тесты дважды, чтобы первоначальная инициализация не повлияла на результат
//                // Также это дополнительно помогает в отладке
//                // Увеличивает вероятность попадания информации в кэши
//                for (var i = 0; i < 10; i++)
//                {
//                    //0 - 10 ГБ
//                    //Каждые 100 МБ срез цифр
//
//                    //links.TestGetSourceFunction();
//                    //links.TestGetSourceFunctionInParallel();
//                    //links.TestGetTargetFunction();
//                    //links.TestGetTargetFunctionInParallel();
//                    links.Create64BillionLinks();
//
//                    links.TestRandomSearchFixed();
//                    //links.Create64BillionLinksInParallel();
//                    links.TestEachFunction();
//                    //links.TestForeach();
//                    //links.TestParallelForeach();
//                }
//
//                links.TestDeletionOfAllLinks();
//
//            }
//            catch (Exception ex)
//            {
//                ex.WriteToConsole();
//            }
//        }*/
//
//         /*
//        public static void TestLinksInSteps()
//        {
//            const long gibibyte = 1024 * 1024 * 1024;
//            const long mebibyte = 1024 * 1024;
//
//            var totalLinksToCreate = gibibyte / Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
//            var linksStep = 102 * mebibyte / Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
//
//            var creationMeasurements = new List<TimeSpan>();
//            var searchMeasuremets = new List<TimeSpan>();
//            var deletionMeasurements = new List<TimeSpan>();
//
//            GetBaseRandomLoopOverhead(linksStep);
//            GetBaseRandomLoopOverhead(linksStep);
//
//            var stepLoopOverhead = GetBaseRandomLoopOverhead(linksStep);
//
//            ConsoleHelpers.Debug("Step loop overhead: {0}.", stepLoopOverhead);
//
//            var loops = totalLinksToCreate / linksStep;
//
//            for (int i = 0; i < loops; i++)
//            {
//                creationMeasurements.Add(Measure(() => links.RunRandomCreations(linksStep)));
//                searchMeasuremets.Add(Measure(() => links.RunRandomSearches(linksStep)));
//
//                Console.Write("\rC + S {0}/{1}", i + 1, loops);
//            }
```

```
//
//            ConsoleHelpers.Debug();
//
//            for (int i = 0; i < loops; i++)
//            {
//                deletionMeasurements.Add(Measure(() => links.RunRandomDeletions(linksStep)));
//
//                Console.Write("\rD {0}/{1}", i + 1, loops);
//            }
//
//            ConsoleHelpers.Debug();
//
//            ConsoleHelpers.Debug("C S D");
//
//            for (int i = 0; i < loops; i++)
//            {
//                ConsoleHelpers.Debug("{0} {1} {2}", creationMeasurements[i],
↪  searchMeasuremets[i], deletionMeasurements[i]);
//            }
//
//            ConsoleHelpers.Debug("C S D (no overhead)");
//
//            for (int i = 0; i < loops; i++)
//            {
//                ConsoleHelpers.Debug("{0} {1} {2}", creationMeasurements[i] -
↪  stepLoopOverhead, searchMeasuremets[i] - stepLoopOverhead, deletionMeasurements[i] -
↪  stepLoopOverhead);
//            }
//
//            ConsoleHelpers.Debug("All tests done. Total links left in database: {0}.",
↪  links.Total);
//        }
//
//        private static void CreatePoints(this Platform.Links.Data.Core.Doublets.Links links,
↪  long amountToCreate)
//        {
//            for (long i = 0; i < amountToCreate; i++)
//                links.Create(0, 0);
//        }
//
//         private static TimeSpan GetBaseRandomLoopOverhead(long loops)
//         {
//             return Measure(() =>
//             {
//                 ulong maxValue = RandomHelpers.DefaultFactory.NextUInt64();
//                 ulong result = 0;
//                 for (long i = 0; i < loops; i++)
//                 {
//                     var source = RandomHelpers.DefaultFactory.NextUInt64(maxValue);
//                     var target = RandomHelpers.DefaultFactory.NextUInt64(maxValue);
//
//                     result += maxValue + source + target;
//                 }
//                 Global.Trash = result;
//             });
//         }
//          */
//
//         [Fact(Skip = "performance test")]
//         public static void GetSourceTest()
//         {
//             using (var scope = new TempLinksTestScope())
//             {
//                 var links = scope.Links;
//                 ConsoleHelpers.Debug("Testing GetSource function with {0} Iterations.",
↪  Iterations);
//
//                 ulong counter = 0;
//
//                 //var firstLink = links.First();
//                 // Создаём одну связь, из которой будет производить считывание
//                 var firstLink = links.Create();
//
//                 var sw = Stopwatch.StartNew();
//
//                 // Тестируем саму функцию
//                 for (ulong i = 0; i < Iterations; i++)
```

```
//                 {
//                     counter += links.GetSource(firstLink);
//                 }
//
//                 var elapsedTime = sw.Elapsed;
//
//                 var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
//
//                 // Удаляем связь, из которой производилось считывание
//                 links.Delete(firstLink);
//
//                 ConsoleHelpers.Debug(
//                     "{0} Iterations of GetSource function done in {1} ({2} Iterations per second), counter result: {3}",
//                     Iterations, elapsedTime, (long)iterationsPerSecond, counter);
//             }
//         }
//
//         [Fact(Skip = "performance test")]
//         public static void GetSourceInParallel()
//         {
//             using (var scope = new TempLinksTestScope())
//             {
//                 var links = scope.Links;
//                 ConsoleHelpers.Debug("Testing GetSource function with {0} Iterations in parallel.", Iterations);
//
//                 long counter = 0;
//
//                 //var firstLink = links.First();
//                 var firstLink = links.Create();
//
//                 var sw = Stopwatch.StartNew();
//
//                 // Тестируем саму функцию
//                 Parallel.For(0, Iterations, x =>
//                 {
//                     Interlocked.Add(ref counter, (long)links.GetSource(firstLink));
//                     //Interlocked.Increment(ref counter);
//                 });
//
//                 var elapsedTime = sw.Elapsed;
//
//                 var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
//
//                 links.Delete(firstLink);
//
//                 ConsoleHelpers.Debug(
//                     "{0} Iterations of GetSource function done in {1} ({2} Iterations per second), counter result: {3}",
//                     Iterations, elapsedTime, (long)iterationsPerSecond, counter);
//             }
//         }
//
//         [Fact(Skip = "performance test")]
//         public static void TestGetTarget()
//         {
//             using (var scope = new TempLinksTestScope())
//             {
//                 var links = scope.Links;
//                 ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations.", Iterations);
//
//                 ulong counter = 0;
//
//                 //var firstLink = links.First();
//                 var firstLink = links.Create();
//
//                 var sw = Stopwatch.StartNew();
//
//                 for (ulong i = 0; i < Iterations; i++)
//                 {
//                     counter += links.GetTarget(firstLink);
//                 }
//
//                 var elapsedTime = sw.Elapsed;
//
```

```
//                 var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
//
//                 links.Delete(firstLink);
//
//                 ConsoleHelpers.Debug(
//                     "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
    ↪ second), counter result: {3}",
//                     Iterations, elapsedTime, (long)iterationsPerSecond, counter);
//             }
//         }
//
//         [Fact(Skip = "performance test")]
//         public static void TestGetTargetInParallel()
//         {
//             using (var scope = new TempLinksTestScope())
//             {
//                 var links = scope.Links;
//                 ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations in
    ↪ parallel.", Iterations);
//
//                 long counter = 0;
//
//                 //var firstLink = links.First();
//                 var firstLink = links.Create();
//
//                 var sw = Stopwatch.StartNew();
//
//                 Parallel.For(0, Iterations, x =>
//                 {
//                     Interlocked.Add(ref counter, (long)links.GetTarget(firstLink));
//                     //Interlocked.Increment(ref counter);
//                 });
//
//                 var elapsedTime = sw.Elapsed;
//
//                 var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
//
//                 links.Delete(firstLink);
//
//                 ConsoleHelpers.Debug(
//                     "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
    ↪ second), counter result: {3}",
//                     Iterations, elapsedTime, (long)iterationsPerSecond, counter);
//             }
//         }
//
//         // TODO: Заполнить базу данных перед тестом
//         /*
//         [Fact]
//         public void TestRandomSearchFixed()
//         {
//             var tempFilename = Path.GetTempFileName();
//
//             using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
    ↪ DefaultLinksSizeStep))
//             {
//                 long iterations = 64 * 1024 * 1024 /
    ↪ Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
//
//                 ulong counter = 0;
//                 var maxLink = links.Total;
//
//                 ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.",
    ↪ iterations);
//
//                 var sw = Stopwatch.StartNew();
//
//                 for (var i = iterations; i > 0; i--)
//                 {
//                     var source =
    ↪ RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
//                     var target =
    ↪ RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
//
//                     counter += links.Search(source, target);
//                 }
//
```

```
//                 var elapsedTime = sw.Elapsed;
//
//                 var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;
//
//                 ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2} Iterations per second), c: {3}", iterations, elapsedTime, (long)iterationsPerSecond, counter);
//             }
//
//             File.Delete(tempFilename);
//         }*/
//
//         [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
//         public static void TestRandomSearchAll()
//         {
//             using (var scope = new TempLinksTestScope())
//             {
//                 var links = scope.Links;
//                 ulong counter = 0;
//
//                 var maxLink = links.Count();
//
//                 var iterations = links.Count();
//
//                 ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.", links.Count());
//
//                 var sw = Stopwatch.StartNew();
//
//                 for (var i = iterations; i > 0; i--)
//                 {
//                     var linksAddressRange = new Range<ulong>(_constants.InternalReferencesRange.Minimum, maxLink);
//
//                     var source = RandomHelpers.Default.NextUInt64(linksAddressRange);
//                     var target = RandomHelpers.Default.NextUInt64(linksAddressRange);
//
//                     counter += links.SearchOrDefault(source, target);
//                 }
//
//                 var elapsedTime = sw.Elapsed;
//
//                 var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;
//
//                 ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2} Iterations per second), c: {3}",
//                      iterations, elapsedTime, (long)iterationsPerSecond, counter);
//             }
//         }
//
//         [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
//         public static void TestEach()
//         {
//             using (var scope = new TempLinksTestScope())
//             {
//                 var links = scope.Links;
//
//                 var counter = new Counter<IList<ulong>, ulong>(links.Constants.Continue);
//
//                 ConsoleHelpers.Debug("Testing Each function.");
//
//                 var sw = Stopwatch.StartNew();
//
//                 links.Each(counter.IncrementAndReturnTrue);
//
//                 var elapsedTime = sw.Elapsed;
//
//                 var linksPerSecond = counter.Count / elapsedTime.TotalSeconds;
//
//                 ConsoleHelpers.Debug("{0} Iterations of Each's handler function done in {1} ({2} links per second)",
//                     counter, elapsedTime, (long)linksPerSecond);
//             }
//         }
//
//         /*
//         [Fact]
```

```
//         public static void TestForeach()
//         {
//             var tempFilename = Path.GetTempFileName();
//
//             using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
↪  DefaultLinksSizeStep))
//             {
//                 ulong counter = 0;
//
//                 ConsoleHelpers.Debug("Testing foreach through links.");
//
//                 var sw = Stopwatch.StartNew();
//
//                 //foreach (var link in links)
//                 //{
//                 //    counter++;
//                 //}
//
//                 var elapsedTime = sw.Elapsed;
//
//                 var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
//
//                 ConsoleHelpers.Debug("{0} Iterations of Foreach's handler block done in {1}
↪  ({2} links per second)", counter, elapsedTime, (long)linksPerSecond);
//             }
//
//             File.Delete(tempFilename);
//         }
//         */
//
//         /*
//         [Fact]
//         public static void TestParallelForeach()
//         {
//             var tempFilename = Path.GetTempFileName();
//
//             using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
↪  DefaultLinksSizeStep))
//             {
//
//                 long counter = 0;
//
//                 ConsoleHelpers.Debug("Testing parallel foreach through links.");
//
//                 var sw = Stopwatch.StartNew();
//
//                 //Parallel.ForEach((IEnumerable<ulong>)links, x =>
//                 //{
//                 //    Interlocked.Increment(ref counter);
//                 //});
//
//                 var elapsedTime = sw.Elapsed;
//
//                 var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
//
//                 ConsoleHelpers.Debug("{0} Iterations of Parallel Foreach's handler block done
↪  in {1} ({2} links per second)", counter, elapsedTime, (long)linksPerSecond);
//             }
//
//             File.Delete(tempFilename);
//         }
//         */
//
//         [Fact(Skip = "performance test")]
//         public static void Create64BillionLinks()
//         {
//             using (var scope = new TempLinksTestScope())
//             {
//                 var links = scope.Links;
//                 var linksBeforeTest = links.Count();
//
//                 long linksToCreate = 64 * 1024 * 1024 /
↪  UInt64UnitedMemoryLinks.LinkSizeInBytes;
//
//                 ConsoleHelpers.Debug("Creating {0} links.", linksToCreate);
//
//                 var elapsedTime = Performance.Measure(() =>
```

```
853 //                 {
854 //                     for (long i = 0; i < linksToCreate; i++)
855 //                     {
856 //                         links.Create();
857 //                     }
858 //                 });
859 //
860 //                 var linksCreated = links.Count() - linksBeforeTest;
861 //                 var linksPerSecond = linksCreated / elapsedTime.TotalSeconds;
862 //
863 //                 ConsoleHelpers.Debug("Current links count: {0}.", links.Count());
864 //
865 //                 ConsoleHelpers.Debug("{0} links created in {1} ({2} links per second)",
    ↪  linksCreated, elapsedTime,
866 //                     (long)linksPerSecond);
867 //             }
868 //         }
869 //
870 //         [Fact(Skip = "performance test")]
871 //         public static void Create64BillionLinksInParallel()
872 //         {
873 //             using (var scope = new TempLinksTestScope())
874 //             {
875 //                 var links = scope.Links;
876 //                 var linksBeforeTest = links.Count();
877 //
878 //                 var sw = Stopwatch.StartNew();
879 //
880 //                 long linksToCreate = 64 * 1024 * 1024 /
    ↪  UInt64UnitedMemoryLinks.LinkSizeInBytes;
881 //
882 //                 ConsoleHelpers.Debug("Creating {0} links in parallel.", linksToCreate);
883 //
884 //                 Parallel.For(0, linksToCreate, x => links.Create());
885 //
886 //                 var elapsedTime = sw.Elapsed;
887 //
888 //                 var linksCreated = links.Count() - linksBeforeTest;
889 //                 var linksPerSecond = linksCreated / elapsedTime.TotalSeconds;
890 //
891 //                 ConsoleHelpers.Debug("{0} links created in {1} ({2} links per second)",
    ↪  linksCreated, elapsedTime,
892 //                     (long)linksPerSecond);
893 //             }
894 //         }
895 //
896 //         [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
897 //         public static void TestDeletionOfAllLinks()
898 //         {
899 //             using (var scope = new TempLinksTestScope())
900 //             {
901 //                 var links = scope.Links;
902 //                 var linksBeforeTest = links.Count();
903 //
904 //                 ConsoleHelpers.Debug("Deleting all links");
905 //
906 //                 var elapsedTime = Performance.Measure(links.DeleteAll);
907 //
908 //                 var linksDeleted = linksBeforeTest - links.Count();
909 //                 var linksPerSecond = linksDeleted / elapsedTime.TotalSeconds;
910 //
911 //                 ConsoleHelpers.Debug("{0} links deleted in {1} ({2} links per second)",
    ↪  linksDeleted, elapsedTime,
912 //                     (long)linksPerSecond);
913 //             }
914 //         }
915 //
916 //         #endregion
917 //     }
918 // }
```

1.76 ./csharp/Platform.Data.Doublets.Sequences.Tests/Uint64LinksExtensionsTests.cs

```
1 using Platform.Data.Doublets.Memory;
2 using Platform.Data.Doublets.Memory.United.Generic;
3 using Platform.Data.Numbers.Raw;
4 using Platform.Memory;
5 using Platform.Numbers;
6 using Xunit;
```

```
using Xunit.Abstractions;
using TLinkAddress = System.UInt64;

namespace Platform.Data.Doublets.Sequences.Tests
{
    public class Uint64LinksExtensionsTests
    {
        public static ILinks<TLinkAddress> CreateLinks() => CreateLinks<TLinkAddress>(new
        ↪  Platform.IO.TemporaryFile());

        public static ILinks<TLinkAddress> CreateLinks<TLinkAddress>(string dataDBFilename)
        {
            var linksConstants = new
            ↪  LinksConstants<TLinkAddress>(enableExternalReferencesSupport: true);
            return new UnitedMemoryLinks<TLinkAddress>(new
            ↪  FileMappedResizableDirectMemory(dataDBFilename),
            ↪  UnitedMemoryLinks<TLinkAddress>.DefaultLinksSizeStep, linksConstants,
            ↪  IndexTreeType.Default);
        }
        [Fact]
        public void FormatStructureWithExternalReferenceTest()
        {
            ILinks<TLinkAddress> links = CreateLinks();
            TLinkAddress zero = default;
            var one = Arithmetic.Increment(zero);
            var markerIndex = one;
            var meaningRoot = links.GetOrCreate(markerIndex, markerIndex);
            var numberMarker = links.GetOrCreate(meaningRoot, Arithmetic.Increment(ref
            ↪  markerIndex));
            AddressToRawNumberConverter<TLinkAddress> addressToNumberConverter = new();
            var numberAddress = addressToNumberConverter.Convert(1);
            var numberLink = links.GetOrCreate(numberMarker, numberAddress);
            var linkNotation = links.FormatStructure(numberLink, link => link.IsFullPoint(),
            ↪  true);
            Assert.Equal("(3:(2:1 2) 18446744073709551615)", linkNotation);
        }
    }
}
```

1.77 ./csharp/Platform.Data.Doublets.Sequences.Tests/UnaryNumberConvertersTests.cs

```
// using Xunit;
// using Platform.Random;
// using Platform.Data.Doublets.Numbers.Unary;
//
// namespace Platform.Data.Doublets.Sequences.Tests
// {
//     public static class UnaryNumberConvertersTests
//     {
//         [Fact]
//         public static void ConvertersTest()
//         {
//             using (var scope = new TempLinksTestScope())
//             {
//                 const int N = 10;
//                 var links = scope.Links;
//                 var meaningRoot = links.CreatePoint();
//                 var one = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
//                 var powerOf2ToUnaryNumberConverter = new
    ↪  PowerOf2ToUnaryNumberConverter<ulong>(links, one);
//                 var toUnaryNumberConverter = new AddressToUnaryNumberConverter<ulong>(links,
    ↪  powerOf2ToUnaryNumberConverter);
//                 var random = new System.Random(0);
//                 ulong[] numbers = new ulong[N];
//                 ulong[] unaryNumbers = new ulong[N];
//                 for (int i = 0; i < N; i++)
//                 {
//                     numbers[i] = random.NextUInt64();
//                     unaryNumbers[i] = toUnaryNumberConverter.Convert(numbers[i]);
//                 }
//                 var fromUnaryNumberConverterUsingOrOperation = new
    ↪  UnaryNumberToAddressOrOperationConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
//                 var fromUnaryNumberConverterUsingAddOperation = new
    ↪  UnaryNumberToAddressAddOperationConverter<ulong>(links, one);
//                 for (int i = 0; i < N; i++)
//                 {
//                     Assert.Equal(numbers[i],
    ↪  fromUnaryNumberConverterUsingOrOperation.Convert(unaryNumbers[i]));
```

```
// Assert.Equal(numbers[i], fromUnaryNumberConverterUsingAddOperation.Convert(unaryNumbers[i]));
//                 }
//             }
//         }
//     }
// }
```

1.78 ./csharp/Platform.Data.Doublets.Sequences.Tests/UnicodeConvertersTests.cs

```
// using Xunit;
// using Platform.Converters;
// using Platform.Memory;
// using Platform.Reflection;
// using Platform.Scopes;
// using Platform.Data.Numbers.Raw;
// using Platform.Data.Doublets.Incrementers;
// using Platform.Data.Doublets.Numbers.Unary;
// using Platform.Data.Doublets.PropertyOperators;
// using Platform.Data.Doublets.Sequences.Converters;
// using Platform.Data.Doublets.Sequences.Indexes;
// using Platform.Data.Doublets.Sequences.Walkers;
// using Platform.Data.Doublets.Unicode;
// using Platform.Data.Doublets.Memory.United.Generic;
// using Platform.Data.Doublets.CriterionMatchers;
//
// namespace Platform.Data.Doublets.Sequences.Tests
// {
//     public static class UnicodeConvertersTests
//     {
//         [Fact]
//         public static void CharAndUnaryNumberUnicodeSymbolConvertersTest()
//         {
//             using (var scope = new TempLinksTestScope())
//             {
//                 var links = scope.Links;
//                 var meaningRoot = links.CreatePoint();
//                 var one = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
//                 var powerOf2ToUnaryNumberConverter = new PowerOf2ToUnaryNumberConverter<ulong>(links, one);
//                 var addressToUnaryNumberConverter = new AddressToUnaryNumberConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
//                 var unaryNumberToAddressConverter = new UnaryNumberToAddressOrOperationConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
//                 TestCharAndUnicodeSymbolConverters(links, meaningRoot, addressToUnaryNumberConverter, unaryNumberToAddressConverter);
//             }
//         }
//
//         [Fact]
//         public static void CharAndRawNumberUnicodeSymbolConvertersTest()
//         {
//             using (var scope = new Scope<Types<HeapResizableDirectMemory, UnitedMemoryLinks<ulong>>>())
//             {
//                 var links = scope.Use<ILinks<ulong>>();
//                 var meaningRoot = links.CreatePoint();
//                 var addressToRawNumberConverter = new AddressToRawNumberConverter<ulong>();
//                 var rawNumberToAddressConverter = new RawNumberToAddressConverter<ulong>();
//                 TestCharAndUnicodeSymbolConverters(links, meaningRoot, addressToRawNumberConverter, rawNumberToAddressConverter);
//             }
//         }
//         private static void TestCharAndUnicodeSymbolConverters(ILinks<ulong> links, ulong meaningRoot, IConverter<ulong> addressToNumberConverter, IConverter<ulong> numberToAddressConverter)
//         {
//             var unicodeSymbolMarker = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
//             var charToUnicodeSymbolConverter = new CharToUnicodeSymbolConverter<ulong>(links, addressToNumberConverter, unicodeSymbolMarker);
//             var originalCharacter = 'H';
//             var characterLink = charToUnicodeSymbolConverter.Convert(originalCharacter);
//             var unicodeSymbolCriterionMatcher = new TargetMatcher<ulong>(links, unicodeSymbolMarker);
//             var unicodeSymbolToCharConverter = new UnicodeSymbolToCharConverter<ulong>(links, numberToAddressConverter, unicodeSymbolCriterionMatcher);
//             var resultingCharacter = unicodeSymbolToCharConverter.Convert(characterLink);
```

```
57  //             Assert.Equal(originalCharacter, resultingCharacter);
58  //         }
59  //
60  //         [Fact]
61  //         public static void StringAndUnicodeSequenceConvertersTest()
62  //         {
63  //             using (var scope = new TempLinksTestScope())
64  //             {
65  //                 var links = scope.Links;
66  //
67  //                 var itself = links.Constants.Itself;
68  //
69  //                 var meaningRoot = links.CreatePoint();
70  //                 var unaryOne = links.CreateAndUpdate(meaningRoot, itself);
71  //                 var unicodeSymbolMarker = links.CreateAndUpdate(meaningRoot, itself);
72  //                 var unicodeSequenceMarker = links.CreateAndUpdate(meaningRoot, itself);
73  //                 var frequencyMarker = links.CreateAndUpdate(meaningRoot, itself);
74  //                 var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot, itself);
75  //
76  //                 var powerOf2ToUnaryNumberConverter = new
    ↪ PowerOf2ToUnaryNumberConverter<ulong>(links, unaryOne);
77  //                 var addressToUnaryNumberConverter = new
    ↪ AddressToUnaryNumberConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
78  //                 var charToUnicodeSymbolConverter = new
    ↪ CharToUnicodeSymbolConverter<ulong>(links, addressToUnaryNumberConverter,
    ↪ unicodeSymbolMarker);
79  //
80  //                 var unaryNumberToAddressConverter = new
    ↪ UnaryNumberToAddressOrOperationConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
81  //                 var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links,
    ↪ unaryOne);
82  //                 var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
    ↪ frequencyMarker, unaryOne, unaryNumberIncrementer);
83  //                 var frequencyPropertyOperator = new PropertyOperator<ulong>(links,
    ↪ frequencyPropertyMarker, frequencyMarker);
84  //                 var index = new FrequencyIncrementingSequenceIndex<ulong>(links,
    ↪ frequencyPropertyOperator, frequencyIncrementer);
85  //                 var linkToItsFrequencyNumberConverter = new
    ↪ LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
    ↪ unaryNumberToAddressConverter);
86  //                 var sequenceToItsLocalElementLevelsConverter = new
    ↪ SequenceToItsLocalElementLevelsConverter<ulong>(links, linkToItsFrequencyNumberConverter);
87  //                 var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
    ↪ sequenceToItsLocalElementLevelsConverter);
88  //
89  //                 var stringToUnicodeSequenceConverter = new
    ↪ StringToUnicodeSequenceConverter<ulong>(links, charToUnicodeSymbolConverter, index,
    ↪ optimalVariantConverter, unicodeSequenceMarker);
90  //
91  //                 var originalString = "Hello";
92  //
93  //                 var unicodeSequenceLink =
    ↪ stringToUnicodeSequenceConverter.Convert(originalString);
94  //
95  //                 var unicodeSymbolCriterionMatcher = new TargetMatcher<ulong>(links,
    ↪ unicodeSymbolMarker);
96  //                 var unicodeSymbolToCharConverter = new
    ↪ UnicodeSymbolToCharConverter<ulong>(links, unaryNumberToAddressConverter,
    ↪ unicodeSymbolCriterionMatcher);
97  //
98  //                 var unicodeSequenceCriterionMatcher = new TargetMatcher<ulong>(links,
    ↪ unicodeSequenceMarker);
99  //
100 //                 var sequenceWalker = new LeveledSequenceWalker<ulong>(links,
    ↪ unicodeSymbolCriterionMatcher.IsMatched);
101 //
102 //                 var unicodeSequenceToStringConverter = new
    ↪ UnicodeSequenceToStringConverter<ulong>(links, unicodeSequenceCriterionMatcher,
    ↪ sequenceWalker, unicodeSymbolToCharConverter);
103 //
104 //                 var resultingString =
    ↪ unicodeSequenceToStringConverter.Convert(unicodeSequenceLink);
105 //
106 //                 Assert.Equal(originalString, resultingString);
107 //             }
```

```
108 //             }
109 //         }
110 // }
```

Index