

LinksPlatform's Platform.Data.Doublets.Sequences Class Library

1.1 ./csharp/Platform.Data.Doublets.Sequences/Converters/BalancedVariantConverter.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.Converters
7  {
8      /// <summary>
9      /// <para>
10     /// Represents the balanced variant converter.
11     /// </para>
12     /// <para></para>
13     /// </summary>
14     /// <seealso cref="LinkedListToSequenceConverterBase{TLinkAddress}"/>
15     public class BalancedVariantConverter<TLinkAddress> :
16         ↳ LinkedListToSequenceConverterBase<TLinkAddress>
17     {
18         /// <summary>
19         /// <para>
20         /// Initializes a new <see cref="BalancedVariantConverter"/> instance.
21         /// </para>
22         /// <para></para>
23         /// </summary>
24         /// <param name="links">
25         /// <para>A links.</para>
26         /// <para></para>
27         /// </param>
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         public BalancedVariantConverter(ILinks<TLinkAddress> links) : base(links) { }
30
31         /// <summary>
32         /// <para>
33         /// Converts the sequence.
34         /// </para>
35         /// <para></para>
36         /// </summary>
37         /// <param name="sequence">
38         /// <para>The sequence.</para>
39         /// <para></para>
40         /// </param>
41         /// <returns>
42         /// <para>The link</para>
43         /// <para></para>
44         /// </returns>
45         [MethodImpl(MethodImplOptions.AggressiveInlining)]
46         public override TLinkAddress Convert(ICollection<TLinkAddress>? sequence)
47         {
48             var length = sequence.Count;
49             if (length < 1)
50             {
51                 return default;
52             }
53             if (length == 1)
54             {
55                 return sequence[0];
56             }
57             // Make copy of next layer
58             if (length > 2)
59             {
60                 // TODO: Try to use stackalloc (which at the moment is not working with
61                 ↳ generics) but will be possible with Sigil
62                 var halvedSequence = new TLinkAddress[(length / 2) + (length % 2)];
63                 HalveSequence(halvedSequence, sequence, length);
64                 sequence = halvedSequence;
65                 length = halvedSequence.Length;
66             }
67             // Keep creating layer after layer
68             while (length > 2)
69             {
70                 HalveSequence(sequence, sequence, length);
71                 length = (length / 2) + (length % 2);
72             }
73             return _links.GetOrCreate(sequence[0], sequence[1]);
74         }
75     }
76     [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

74     private void HalveSequence(ICollection<TLinkAddress>? destination, ICollection<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> :
    ↪ LinkedListToSequenceConverterBase<TLinkAddress>
20     {
21         private static readonly LinkConstants<TLinkAddress> _constants =
    ↪ Default<LinkConstants<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<ICollection<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>

```

```

57     /// <para></para>
58     /// </param>
59     /// <param name="doubletData">
60     /// <para>A doublet data.</para>
61     /// <para></para>
62     /// </param>
63     [MethodImpl(MethodImplOptions.AggressiveInlining)]
64     public HalfDoublet(TLinkAddress element, LinkFrequency<TLinkAddress> doubletData)
65     {
66         Element = element;
67         DoubletData = doubletData;
68     }
69
70     /// <summary>
71     /// <para>
72     /// Returns the string.
73     /// </para>
74     /// <para></para>
75     /// </summary>
76     /// <returns>
77     /// <para>The string</para>
78     /// <para></para>
79     /// </returns>
80     public override string ToString() => $"{Element}: ({DoubletData})";
81 }
82
83 /// <summary>
84 /// <para>
85 /// Initializes a new <see cref="CompressingConverter"/> instance.
86 /// </para>
87 /// <para></para>
88 /// </summary>
89 /// <param name="links">
90 /// <para>A links.</para>
91 /// <para></para>
92 /// </param>
93 /// <param name="baseConverter">
94 /// <para>A base converter.</para>
95 /// <para></para>
96 /// </param>
97 /// <param name="doubletFrequenciesCache">
98 /// <para>A doublet frequencies cache.</para>
99 /// <para></para>
100 /// </param>
101 [MethodImpl(MethodImplOptions.AggressiveInlining)]
102 public CompressingConverter(ILinks<TLinkAddress> links, IConverter<IList<TLinkAddress>,
    ↪ TLinkAddress> baseConverter, LinkFrequenciesCache<TLinkAddress>
    ↪ doubletFrequenciesCache)
    : this(links, baseConverter, doubletFrequenciesCache, _one, true) { }
103
104 /// <summary>
105 /// <para>
106 /// Initializes a new <see cref="CompressingConverter"/> instance.
107 /// </para>
108 /// <para></para>
109 /// </summary>
110 /// <param name="links">
111 /// <para>A links.</para>
112 /// <para></para>
113 /// </param>
114 /// <param name="baseConverter">
115 /// <para>A base converter.</para>
116 /// <para></para>
117 /// </param>
118 /// <param name="doubletFrequenciesCache">
119 /// <para>A doublet frequencies cache.</para>
120 /// <para></para>
121 /// </param>
122 /// <param name="doInitialFrequenciesIncrement">
123 /// <para>A do initial frequencies increment.</para>
124 /// <para></para>
125 /// </param>
126 [MethodImpl(MethodImplOptions.AggressiveInlining)]
127 public CompressingConverter(ILinks<TLinkAddress> links, IConverter<IList<TLinkAddress>,
    ↪ TLinkAddress> baseConverter, LinkFrequenciesCache<TLinkAddress>
    ↪ doubletFrequenciesCache, bool doInitialFrequenciesIncrement)
    : this(links, baseConverter, doubletFrequenciesCache, _one,
    ↪ doInitialFrequenciesIncrement) { }
128
129

```

```

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>,
159     ↪ TLinkAddress> baseConverter, LinkFrequenciesCache<TLinkAddress>
160     ↪ doubletFrequenciesCache, TLinkAddress minFrequencyToCompress, bool
161     ↪ doInitialFrequenciesIncrement)
162         : base(links)
163     {
164         _baseConverter = baseConverter;
165         _doubletFrequenciesCache = doubletFrequenciesCache;
166         if (_comparer.Compare(minFrequencyToCompress, _one) < 0)
167         {
168             minFrequencyToCompress = _one;
169         }
170         _minFrequencyToCompress = minFrequencyToCompress;
171         _doInitialFrequenciesIncrement = doInitialFrequenciesIncrement;
172         ResetMaxDoublet();
173     }
174
175     /// <summary>
176     /// <para>
177     /// Converts the source.
178     /// </para>
179     /// </summary>
180     /// <param name="source">
181     /// <para>The source.</para>
182     /// <para></para>
183     /// </param>
184     /// <returns>
185     /// <para>The link</para>
186     /// <para></para>
187     /// </returns>
188     [MethodImpl(MethodImplOptions.AggressiveInlining)]
189     public override TLinkAddress Convert(IList<TLinkAddress>? source) =>
190     ↪ _baseConverter.Convert(Compress(source));
191     [MethodImpl(MethodImplOptions.AggressiveInlining)]
192     private IList<TLinkAddress>? Compress(IList<TLinkAddress>? sequence)
193     {
194         if (sequence.IsNullOrEmpty())
195         {
196             return null;
197         }
198         if (sequence.Count == 1)
199         {
200             return sequence;
201         }
202         if (sequence.Count == 2)
203         {
204             return new[] { _links.GetOrCreate(sequence[0], sequence[1]) };
205         }
206         // 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
220                 ↪ frequencies, it is expected that all frequencies for the sequence
221                 ↪ are prepared.");
222         }
223     }
224     copy[i - 1].Element = sequence[i - 1];
225     copy[i - 1].DoubletData = data;
226     UpdateMaxDoublet(ref doublet, data);
227 }
228 copy[sequence.Count - 1].Element = sequence[sequence.Count - 1];
229 copy[sequence.Count - 1].DoubletData = new LinkFrequency<TLinkAddress>();
230 if (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
231 {
232     var newLength = ReplaceDoublets(copy);
233     sequence = new TLinkAddress[newLength];
234     for (int i = 0; i < newLength; i++)
235     {
236         sequence[i] = copy[i].Element;
237     }
238     return sequence;
239 }
240 [MethodImpl(MethodImplOptions.AggressiveInlining)]
241 private int ReplaceDoublets(HalfDoublet[] copy)
242 {
243     var oldLength = copy.Length;
244     var newLength = copy.Length;
245     while (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
246     {
247         var maxDoubletSource = _maxDoublet.Source;
248         var maxDoubletTarget = _maxDoublet.Target;
249         if (_equalityComparer.Equals(_maxDoubletData.Link, _constants.Null))
250         {
251             _maxDoubletData.Link = _links.GetOrCreate(maxDoubletSource,
252                 ↪ maxDoubletTarget);
253         }
254         var maxDoubletReplacementLink = _maxDoubletData.Link;
255         oldLength--;
256         var oldLengthMinusTwo = oldLength - 1;
257         // Substitute all usages
258         int w = 0, r = 0; // (r == read, w == write)
259         for (; r < oldLength; r++)
260         {
261             if (_equalityComparer.Equals(copy[r].Element, maxDoubletSource) &&
262                 ↪ _equalityComparer.Equals(copy[r + 1].Element, maxDoubletTarget))
263             {
264                 if (r > 0)
265                 {
266                     var previous = copy[w - 1].Element;
267                     copy[w - 1].DoubletData.DecrementFrequency();
268                     copy[w - 1].DoubletData =
269                         ↪ _doubletFrequenciesCache.IncrementFrequency(previous,
270                             ↪ maxDoubletReplacementLink);
271                 }
272                 if (r < oldLengthMinusTwo)
273                 {
274                     var next = copy[r + 2].Element;
275                     copy[r + 1].DoubletData.DecrementFrequency();
276                     copy[w].DoubletData = _doubletFrequenciesCache.IncrementFrequency(max
277                         ↪ xDoubletReplacementLink,
278                         ↪ next);
279                 }
280                 copy[w++].Element = maxDoubletReplacementLink;

```

```

274         r++;
275         newLength--;
276     }
277     else
278     {
279         copy[w++] = copy[r];
280     }
281 }
282 if (w < newLength)
283 {
284     copy[w] = copy[r];
285 }
286 oldLength = newLength;
287 ResetMaxDoublet();
288 UpdateMaxDoublet(copy, newLength);
289 }
290 return newLength;
291 }
292 [MethodImpl(MethodImplOptions.AggressiveInlining)]
293 private void ResetMaxDoublet()
294 {
295     _maxDoublet = new Doublet<TLinkAddress>();
296     _maxDoubletData = new LinkFrequency<TLinkAddress>();
297 }
298 [MethodImpl(MethodImplOptions.AggressiveInlining)]
299 private void UpdateMaxDoublet(HalfDoublet[] copy, int length)
300 {
301     Doublet<TLinkAddress> doublet = default;
302     for (var i = 1; i < length; i++)
303     {
304         doublet = new Doublet<TLinkAddress>(copy[i - 1].Element, copy[i].Element);
305         UpdateMaxDoublet(ref doublet, copy[i - 1].DoubletData);
306     }
307 }
308 [MethodImpl(MethodImplOptions.AggressiveInlining)]
309 private void UpdateMaxDoublet(ref Doublet<TLinkAddress> doublet,
310     ↪ LinkFrequency<TLinkAddress> data)
311 {
312     var frequency = data.Frequency;
313     var maxFrequency = _maxDoubletData.Frequency;
314     //if (frequency > _minFrequencyToCompress && (maxFrequency < frequency ||
315     ↪ (maxFrequency == frequency && doublet.Source + doublet.Target < /* gives better
316     ↪ compression string data (and gives collisions quickly) */ _maxDoublet.Source +
317     ↪ _maxDoublet.Target)))
318     if (_comparer.Compare(frequency, _minFrequencyToCompress) > 0 &&
319     ↪ (_comparer.Compare(maxFrequency, frequency) < 0 ||
320     ↪ (_equalityComparer.Equals(maxFrequency, frequency) &&
321     ↪ _comparer.Compare(Arithmetic.Add(doublet.Source, doublet.Target),
322     ↪ Arithmetic.Add(_maxDoublet.Source, _maxDoublet.Target)) > 0))) /* gives
323     ↪ better stability and better compression on sequent data and even on random
324     ↪ numbers data (but gives collisions anyway) */
325     {
326         _maxDoublet = doublet;
327         _maxDoubletData = data;
328     }
329 }
330 }
331 }
332 }

```

1.3 ./csharp/Platform.Data.Doublets.Sequences/Converters/LinksListToSequenceConverterBase.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.Sequences.Converters
8 {
9     /// <summary>
10     /// <para>
11     /// Represents the links list to sequence converter base.
12     /// </para>
13     /// <para></para>
14     /// </summary>
15     /// <seealso cref="LinksOperatorBase{TLinkAddress}"/>
16     /// <seealso cref="IConverter{IList{TLinkAddress}, TLinkAddress}"/>
17     public abstract class LinksListToSequenceConverterBase<TLinkAddress> :
18     ↪ LinksOperatorBase<TLinkAddress>, IConverter<IList<TLinkAddress>, TLinkAddress>
19     {

```

```

19     /// <summary>
20     /// <para>
21     /// Initializes a new <see cref="LinksListToSequenceConverterBase"/> instance.
22     /// </para>
23     /// <para></para>
24     /// </summary>
25     /// <param name="links">
26     /// <para>A links.</para>
27     /// <para></para>
28     /// </param>
29     [MethodImpl(MethodImplOptions.AggressiveInlining)]
30     protected LinksListToSequenceConverterBase(ILinks<TLinkAddress> links) : base(links) { }
31
32     /// <summary>
33     /// <para>
34     /// Converts the source.
35     /// </para>
36     /// <para></para>
37     /// </summary>
38     /// <param name="source">
39     /// <para>The source.</para>
40     /// <para></para>
41     /// </param>
42     /// <returns>
43     /// <para>The link</para>
44     /// <para></para>
45     /// </returns>
46     [MethodImpl(MethodImplOptions.AggressiveInlining)]
47     public abstract TLinkAddress Convert(IList<TLinkAddress> source);
48 }
49 }

```

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

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3 using Platform.Collections.Lists;
4 using Platform.Converters;
5 using Platform.Data.Doublets.Sequences.Frequencies.Cache;
6 using Platform.Data.Doublets.Sequences.Frequencies.Counters;
7
8 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Data.Doublets.Sequences.Converters
11 {
12     /// <summary>
13     /// <para>
14     /// Represents the optimal variant converter.
15     /// </para>
16     /// <para></para>
17     /// </summary>
18     /// <seealso cref="LinksListToSequenceConverterBase{TLinkAddress}"/>
19     public class OptimalVariantConverter<TLinkAddress> :
20         ↳ LinksListToSequenceConverterBase<TLinkAddress>
21     {
22         private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
23             ↳ EqualityComparer<TLinkAddress>.Default;
24         private static readonly Comparer<TLinkAddress> _comparer =
25             ↳ Comparer<TLinkAddress>.Default;
26         private readonly IConverter<IList<TLinkAddress>>
27             ↳ _sequenceToItsLocalElementLevelsConverter;
28
29         /// <summary>
30         /// <para>
31         /// Initializes a new <see cref="OptimalVariantConverter"/> instance.
32         /// </para>
33         /// <para></para>
34         /// </summary>
35         /// <param name="links">
36         /// <para>A links.</para>
37         /// <para></para>
38         /// </param>
39         /// <param name="sequenceToItsLocalElementLevelsConverter">
40         /// <para>A sequence to its local element levels converter.</para>
41         /// <para></para>
42         /// </param>
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         public OptimalVariantConverter(ILinks<TLinkAddress> links,
45             ↳ IConverter<IList<TLinkAddress>> sequenceToItsLocalElementLevelsConverter) :
46             ↳ base(links)

```

```

41     => _sequenceToItsLocalElementLevelsConverter =
42         ↳ sequenceToItsLocalElementLevelsConverter;
43
44     /// <summary>
45     /// <para>
46     /// Initializes a new <see cref="OptimalVariantConverter"/> instance.
47     /// </para>
48     /// </summary>
49     /// <param name="links">
50     /// <para>A links.</para>
51     /// </para>
52     /// </param>
53     /// <param name="linkFrequenciesCache">
54     /// <para>A link frequencies cache.</para>
55     /// </param>
56     [MethodImpl(MethodImplOptions.AggressiveInlining)]
57     public OptimalVariantConverter(ILinks<TLinkAddress> links,
58         ↳ LinkFrequenciesCache<TLinkAddress> linkFrequenciesCache)
59         : this(links, new SequenceToItsLocalElementLevelsConverter<TLinkAddress>(links, new
60             ↳ FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<TLinkAddress>(linkFrequen
61             ↳ ciesCache))) {
62         ↳ }
63
64     /// <summary>
65     /// <para>
66     /// Initializes a new <see cref="OptimalVariantConverter"/> instance.
67     /// </para>
68     /// </summary>
69     /// <param name="links">
70     /// <para>A links.</para>
71     /// </para>
72     /// </param>
73     [MethodImpl(MethodImplOptions.AggressiveInlining)]
74     public OptimalVariantConverter(ILinks<TLinkAddress> links)
75         : this(links, new LinkFrequenciesCache<TLinkAddress>(links, new
76             ↳ TotalSequenceSymbolFrequencyCounter<TLinkAddress>(links))) { }
77
78     /// <summary>
79     /// <para>
80     /// Converts the sequence.
81     /// </para>
82     /// </summary>
83     /// <param name="sequence">
84     /// <para>The sequence.</para>
85     /// </para>
86     /// </param>
87     /// <returns>
88     /// <para>The link</para>
89     /// </returns>
90     [MethodImpl(MethodImplOptions.AggressiveInlining)]
91     public override TLinkAddress Convert(IList<TLinkAddress>? sequence)
92     {
93         var length = sequence.Count;
94         if (length == 1)
95         {
96             return sequence[0];
97         }
98         if (length == 2)
99         {
100             return _links.GetOrCreate(sequence[0], sequence[1]);
101         }
102         sequence = sequence.ToArray();
103         var levels = _sequenceToItsLocalElementLevelsConverter.Convert(sequence);
104         while (length > 2)
105         {
106             var levelRepeat = 1;
107             var currentLevel = levels[0];
108             var previousLevel = levels[0];
109             var skipOnce = false;
110             var w = 0;
111             for (var i = 1; i < length; i++)
112             {
113                 if (_equalityComparer.Equals(currentLevel, levels[i]))

```



```

113     {
114         levelRepeat++;
115         skipOnce = false;
116         if (levelRepeat == 2)
117         {
118             sequence[w] = _links.GetOrCreate(sequence[i - 1], sequence[i]);
119             var newLevel = i >= length - 1 ?
120                 GetPreviousLowerThanCurrentOrCurrent(previousLevel,
121                     ↪ currentLevel) :
122                 i < 2 ?
123                 GetNextLowerThanCurrentOrCurrent(currentLevel, levels[i + 1]) :
124                 GetGreatestNeighbourLowerThanCurrentOrCurrent(previousLevel,
125                     ↪ currentLevel, levels[i + 1]);
126             levels[w] = newLevel;
127             previousLevel = currentLevel;
128             w++;
129             levelRepeat = 0;
130             skipOnce = true;
131         }
132         else if (i == length - 1)
133         {
134             sequence[w] = sequence[i];
135             levels[w] = levels[i];
136             w++;
137         }
138     }
139     else
140     {
141         currentLevel = levels[i];
142         levelRepeat = 1;
143         if (skipOnce)
144         {
145             skipOnce = false;
146         }
147         else
148         {
149             sequence[w] = sequence[i - 1];
150             levels[w] = levels[i - 1];
151             previousLevel = levels[w];
152             w++;
153         }
154         if (i == length - 1)
155         {
156             sequence[w] = sequence[i];
157             levels[w] = levels[i];
158             w++;
159         }
160     }
161     }
162     length = w;
163     return _links.GetOrCreate(sequence[0], sequence[1]);
164 }
165 [MethodImpl(MethodImplOptions.AggressiveInlining)]
166 private static TLinkAddress GetGreatestNeighbourLowerThanCurrentOrCurrent(TLinkAddress
167     ↪ previous, TLinkAddress current, TLinkAddress next)
168 {
169     return _comparer.Compare(previous, next) > 0
170         ? _comparer.Compare(previous, current) < 0 ? previous : current
171         : _comparer.Compare(next, current) < 0 ? next : current;
172 }
173 [MethodImpl(MethodImplOptions.AggressiveInlining)]
174 private static TLinkAddress GetNextLowerThanCurrentOrCurrent(TLinkAddress current,
175     ↪ TLinkAddress next) => _comparer.Compare(next, current) < 0 ? next : current;
176 [MethodImpl(MethodImplOptions.AggressiveInlining)]
177 private static TLinkAddress GetPreviousLowerThanCurrentOrCurrent(TLinkAddress previous,
178     ↪ TLinkAddress current) => _comparer.Compare(previous, current) < 0 ? previous :
179     ↪ current;
180 }
181 }

```

1.5 ./csharp/Platform.Data.Doublets.Sequences/Converters/SequenceToltsLocalElementLevelsConverter.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.Sequences.Converters

```

```

8 {
9     /// <summary>
10    /// <para>
11    /// Represents the sequence to its local element levels converter.
12    /// </para>
13    /// <para></para>
14    /// </summary>
15    /// <seealso cref="LinksOperatorBase{TLinkAddress}"/>
16    /// <seealso cref="IConverter{IList{TLinkAddress}}"/>
17    public class SequenceToItsLocalElementLevelsConverter<TLinkAddress> :
18    ↪ LinksOperatorBase<TLinkAddress>, IConverter<IList<TLinkAddress>>
19    {
20        private static readonly Comparer<TLinkAddress> _comparer =
21        ↪ Comparer<TLinkAddress>.Default;
22        private readonly IConverter<Doublet<TLinkAddress>, TLinkAddress>
23        ↪ _linkToItsFrequencyToNumberConveter;
24
25        /// <summary>
26        /// <para>
27        /// Initializes a new <see cref="SequenceToItsLocalElementLevelsConverter"/> 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="linkToItsFrequencyToNumberConveter">
36        /// <para>A link to its frequency to number conveter.</para>
37        /// <para></para>
38        /// </param>
39        [MethodImpl(MethodImplOptions.AggressiveInlining)]
40        public SequenceToItsLocalElementLevelsConverter(ILinks<TLinkAddress> links,
41        ↪ IConverter<Doublet<TLinkAddress>, TLinkAddress> linkToItsFrequencyToNumberConveter)
42        ↪ : base(links) => _linkToItsFrequencyToNumberConveter =
43        ↪ linkToItsFrequencyToNumberConveter;
44
45        /// <summary>
46        /// <para>
47        /// Converts the sequence.
48        /// </para>
49        /// <para></para>
50        /// </summary>
51        /// <param name="sequence">
52        /// <para>The sequence.</para>
53        /// <para></para>
54        /// </param>
55        /// <returns>
56        /// <para>The levels.</para>
57        /// <para></para>
58        /// </returns>
59        [MethodImpl(MethodImplOptions.AggressiveInlining)]
60        public IList<TLinkAddress>? Convert(IList<TLinkAddress>? sequence)
61        {
62            var levels = new TLinkAddress[sequence.Count];
63            levels[0] = GetFrequencyNumber(sequence[0], sequence[1]);
64            for (var i = 1; i < sequence.Count - 1; i++)
65            {
66                var previous = GetFrequencyNumber(sequence[i - 1], sequence[i]);
67                var next = GetFrequencyNumber(sequence[i], sequence[i + 1]);
68                levels[i] = _comparer.Compare(previous, next) > 0 ? previous : next;
69            }
70            levels[levels.Length - 1] = GetFrequencyNumber(sequence[sequence.Count - 2],
71            ↪ sequence[sequence.Count - 1]);
72            return levels;
73        }
74
75        /// <summary>
76        /// <para>
77        /// Gets the frequency number using the specified source.
78        /// </para>
79        /// <para></para>
80        /// </summary>
81        /// <param name="source">
82        /// <para>The source.</para>
83        /// <para></para>
84        /// </param>
85        /// <param name="target">

```

```

79     /// <para>The target.</para>
80     /// <para></para>
81     /// </param>
82     /// <returns>
83     /// <para>The link</para>
84     /// <para></para>
85     /// </returns>
86     [MethodImpl(MethodImplOptions.AggressiveInlining)]
87     public TLinkAddress GetFrequencyNumber(TLinkAddress source, TLinkAddress target) =>
88         ↪ _linkToItsFrequencyToNumberConveter.Convert(new Doublet<TLinkAddress>(source,
89         ↪ target));
88 }
89 }

```

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

```

1  using System.Runtime.CompilerServices;
2  using Platform.Interfaces;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Sequences.CriterionMatchers
7  {
8      /// <summary>
9      /// <para>
10     /// Represents the default sequence element criterion matcher.
11     /// </para>
12     /// <para></para>
13     /// </summary>
14     /// <seealso cref="LinksOperatorBase{TLinkAddress}"/>
15     /// <seealso cref="ICriterionMatcher{TLinkAddress}"/>
16     public class DefaultSequenceElementCriterionMatcher<TLinkAddress> :
17         ↪ LinksOperatorBase<TLinkAddress>, ICriterionMatcher<TLinkAddress>
18     {
19         /// <summary>
20         /// <para>
21         /// Initializes a new <see cref="DefaultSequenceElementCriterionMatcher"/> instance.
22         /// </para>
23         /// <para></para>
24         /// </summary>
25         /// <param name="links">
26         /// <para>A links.</para>
27         /// <para></para>
28         /// </param>
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         public DefaultSequenceElementCriterionMatcher(ILinks<TLinkAddress> links) : base(links)
31         ↪ { }
32
33         /// <summary>
34         /// <para>
35         /// Determines whether this instance is matched.
36         /// </para>
37         /// <para></para>
38         /// </summary>
39         /// <param name="argument">
40         /// <para>The argument.</para>
41         /// <para></para>
42         /// </param>
43         /// <returns>
44         /// <para>The bool</para>
45         /// <para></para>
46         /// </returns>
47         [MethodImpl(MethodImplOptions.AggressiveInlining)]
48         public bool IsMatched(TLinkAddress argument) => _links.IsPartialPoint(argument);
49     }
50 }

```

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

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using Platform.Interfaces;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Sequences.CriterionMatchers
8  {
9      /// <summary>
10     /// <para>
11     /// Represents the marked sequence criterion matcher.

```

```

12  /// </para>
13  /// <para></para>
14  /// </summary>
15  /// <seealso cref="ICriterionMatcher{TLinkAddress}"/>
16  public class MarkedSequenceCriterionMatcher<TLinkAddress> : ICriterionMatcher<TLinkAddress>
17  {
18      private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
19          ↪ EqualityComparer<TLinkAddress>.Default;
20      private readonly ILinks<TLinkAddress> _links;
21      private readonly TLinkAddress _sequenceMarkerLink;
22
23      /// <summary>
24      /// <para>
25      /// Initializes a new <see cref="MarkedSequenceCriterionMatcher"/> instance.
26      /// </para>
27      /// </summary>
28      /// <param name="links">
29      /// <para>A links.</para>
30      /// </param>
31      /// <param name="sequenceMarkerLink">
32      /// <para>A sequence marker link.</para>
33      /// </param>
34      [MethodImpl(MethodImplOptions.AggressiveInlining)]
35      public MarkedSequenceCriterionMatcher(ILinks<TLinkAddress> links, TLinkAddress
36          ↪ sequenceMarkerLink)
37      {
38          _links = links;
39          _sequenceMarkerLink = sequenceMarkerLink;
40      }
41
42      /// <summary>
43      /// <para>
44      /// Determines whether this instance is matched.
45      /// </para>
46      /// </summary>
47      /// <param name="sequenceCandidate">
48      /// <para>The sequence candidate.</para>
49      /// </param>
50      /// <returns>
51      /// <para>The bool</para>
52      /// </returns>
53      [MethodImpl(MethodImplOptions.AggressiveInlining)]
54      public bool IsMatched(TLinkAddress sequenceCandidate)
55      => _equalityComparer.Equals(_links.GetSource(sequenceCandidate), _sequenceMarkerLink)
56      || !_equalityComparer.Equals(_links.SearchOrDefault(_sequenceMarkerLink,
57          ↪ sequenceCandidate), _links.Constants.Null);
58  }
59
60 }

```

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

```

1  using System.Collections.Generic;
2  using Platform.Interfaces;
3
4  namespace Platform.Data.Doublets.Sequences.CriterionMatchers;
5
6  public class UnicodeSequenceMatcher<TLinkAddress> : ICriterionMatcher<TLinkAddress>
7  {
8      public readonly ILinks<TLinkAddress> Storage;
9      public readonly TLinkAddress UnicodeSequenceMarker;
10     public readonly EqualityComparer<TLinkAddress> EqualityComparer =
11         ↪ EqualityComparer<TLinkAddress>.Default;
12     public UnicodeSequenceMatcher(ILinks<TLinkAddress> storage, TLinkAddress
13         ↪ unicodeSequenceMarker)
14     {
15         Storage = storage;
16         UnicodeSequenceMarker = unicodeSequenceMarker;
17     }
18     public bool IsMatched(TLinkAddress argument)
19     {
20         var target = Storage.GetTarget(argument);
21         return EqualityComparer.Equals(UnicodeSequenceMarker, argument) ||
22             ↪ EqualityComparer.Equals(UnicodeSequenceMarker, target);
23     }
24 }

```

21 }

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

```
1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3 using Platform.Collections.Stacks;
4 using Platform.Data.Doublets.Sequences.HeightProviders;
5 using Platform.Data.Sequences;
6
7 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9 namespace Platform.Data.Doublets.Sequences
10 {
11     /// <summary>
12     /// <para>
13     /// Represents the default sequence appender.
14     /// </para>
15     /// <para></para>
16     /// </summary>
17     /// <seealso cref="LinksOperatorBase{TLinkAddress}"/>
18     /// <seealso cref="ISequenceAppender{TLinkAddress}"/>
19     public class DefaultSequenceAppender<TLinkAddress> : LinksOperatorBase<TLinkAddress>,
20         ↳ ISequenceAppender<TLinkAddress>
21     {
22         private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
23             ↳ EqualityComparer<TLinkAddress>.Default;
24         private readonly IStack<TLinkAddress> _stack;
25         private readonly ISequenceHeightProvider<TLinkAddress> _heightProvider;
26
27         /// <summary>
28         /// <para>
29         /// Initializes a new <see cref="DefaultSequenceAppender"/> 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="stack">
38         /// <para>A stack.</para>
39         /// <para></para>
40         /// </param>
41         /// <param name="heightProvider">
42         /// <para>A height provider.</para>
43         /// <para></para>
44         /// </param>
45         [MethodImpl(MethodImplOptions.AggressiveInlining)]
46         public DefaultSequenceAppender(ILinks<TLinkAddress> links, IStack<TLinkAddress> stack,
47             ↳ ISequenceHeightProvider<TLinkAddress> heightProvider)
48             : base(links)
49         {
50             _stack = stack;
51             _heightProvider = heightProvider;
52         }
53
54         /// <summary>
55         /// <para>
56         /// Appends the sequence.
57         /// </para>
58         /// <para></para>
59         /// </summary>
60         /// <param name="sequence">
61         /// <para>The sequence.</para>
62         /// <para></para>
63         /// </param>
64         /// <param name="appendant">
65         /// <para>The appendant.</para>
66         /// <para></para>
67         /// </param>
68         /// <returns>
69         /// <para>The link</para>
70         /// <para></para>
71         /// </returns>
72         [MethodImpl(MethodImplOptions.AggressiveInlining)]
73         public TLinkAddress Append(TLinkAddress sequence, TLinkAddress appendant)
74         {
75             var cursor = sequence;
76             var links = _links;
```

```

74         while (!_equalityComparer.Equals(_heightProvider.Get(cursor), default))
75         {
76             var source = links.GetSource(cursor);
77             var target = links.GetTarget(cursor);
78             if (_equalityComparer.Equals(_heightProvider.Get(source),
79                 ↪ _heightProvider.Get(target)))
80             {
81                 break;
82             }
83             else
84             {
85                 _stack.Push(source);
86                 cursor = target;
87             }
88         }
89         var left = cursor;
90         var right = appendant;
91         while (!_equalityComparer.Equals(cursor = _stack.PopOrDefault(),
92             ↪ links.Constants.Null))
93         {
94             right = links.GetOrCreate(left, right);
95             left = cursor;
96         }
97         return links.GetOrCreate(left, right);
98     }
99 }

```

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

```

1  using System.Collections.Generic;
2  using System.Linq;
3  using System.Runtime.CompilerServices;
4  using Platform.Interfaces;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.Sequences
9  {
10     /// <summary>
11     /// <para>
12     /// Represents the duplicate segments counter.
13     /// </para>
14     /// <para></para>
15     /// </summary>
16     /// <seealso cref="ICounter{int}"/>
17     public class DuplicateSegmentsCounter<TLinkAddress> : ICounter<int>
18     {
19         private readonly IProvider<IList<KeyValuePair<IList<TLinkAddress>?,
20             ↪ IList<TLinkAddress>?>>> _duplicateFragmentsProvider;
21
22         /// <summary>
23         /// <para>
24         /// Initializes a new <see cref="DuplicateSegmentsCounter"/> instance.
25         /// </para>
26         /// <para></para>
27         /// </summary>
28         /// <param name="duplicateFragmentsProvider">
29         /// <para>A duplicate fragments provider.</para>
30         /// <para></para>
31         /// </param>
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         public DuplicateSegmentsCounter(IProvider<IList<KeyValuePair<IList<TLinkAddress>?,
34             ↪ IList<TLinkAddress>?>>> duplicateFragmentsProvider) => _duplicateFragmentsProvider =
35             ↪ duplicateFragmentsProvider;
36
37         /// <summary>
38         /// <para>
39         /// Counts this instance.
40         /// </para>
41         /// <para></para>
42         /// </summary>
43         /// <returns>
44         /// <para>The int</para>
45         /// <para></para>
46         /// </returns>
47         [MethodImpl(MethodImplOptions.AggressiveInlining)]
48         public int Count() => _duplicateFragmentsProvider.Get().Sum(x => x.Value.Count);
49     }
50 }

```

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

```

1  // using System;
2  // using System.Linq;
3  // using System.Collections.Generic;
4  // using System.Runtime.CompilerServices;
5  // using Platform.Interfaces;
6  // using Platform.Collections;
7  // using Platform.Collections.Lists;
8  // using Platform.Collections.Segments;
9  // using Platform.Collections.Segments.Walkers;
10 // using Platform.Singletons;
11 // using Platform.Converters;
12 // using Platform.Data.Doublets.Unicode;
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 duplicate segments provider.
21 //     /// </para>
22 //     /// <para></para>
23 //     /// </summary>
24 //     /// <seealso cref="DictionaryBasedDuplicateSegmentsWalkerBase{TLinkAddress}"/>
25 //     /// <seealso cref="IProvider{IList{KeyValuePair{IList{TLinkAddress},
    ↵  IList{TLinkAddress}}}" />
26 //     public class DuplicateSegmentsProvider<TLinkAddress> :
    ↵  DictionaryBasedDuplicateSegmentsWalkerBase<TLinkAddress>,
    ↵  IProvider<IList<KeyValuePair<IList<TLinkAddress>?, IList<TLinkAddress>?>>>
27 //     {
28 //         private static readonly UncheckedConverter<TLinkAddress, long>
    ↵  _addressToInt64Converter = UncheckedConverter<TLinkAddress, long>.Default;
29 //         private static readonly UncheckedConverter<TLinkAddress, ulong>
    ↵  _addressToUInt64Converter = UncheckedConverter<TLinkAddress, ulong>.Default;
30 //         private static readonly UncheckedConverter<ulong, TLinkAddress>
    ↵  _uInt64ToAddressConverter = UncheckedConverter<ulong, TLinkAddress>.Default;
31 //         private readonly IList<TLinkAddress> _links;
32 //         private readonly IList<TLinkAddress> _sequences;
33 //         private HashSet<KeyValuePair<IList<TLinkAddress>?, IList<TLinkAddress>?>> _groups;
34 //         private BitString _visited;
35 //         private class ItemEquilityComparer :
    ↵  IEqualityComparer<KeyValuePair<IList<TLinkAddress>?, IList<TLinkAddress>?>>
36 //         {
37 //             private readonly IListEqualityComparer<TLinkAddress> _listComparer;
38 //
39 //             /// <summary>
40 //             /// <para>
41 //             /// Initializes a new <see cref="ItemEquilityComparer"/> instance.
42 //             /// </para>
43 //             /// <para></para>
44 //             /// </summary>
45 //             public ItemEquilityComparer() => _listComparer =
    ↵  Default<IListEqualityComparer<TLinkAddress>>.Instance;
46 //
47 //             /// <summary>
48 //             /// <para>
49 //             /// Determines whether this instance equals.
50 //             /// </para>
51 //             /// <para></para>
52 //             /// </summary>
53 //             /// <param name="left">
54 //             /// <para>The left.</para>
55 //             /// <para></para>
56 //             /// </param>
57 //             /// <param name="right">
58 //             /// <para>The right.</para>
59 //             /// <para></para>
60 //             /// </param>
61 //             /// <returns>
62 //             /// <para>The bool</para>
63 //             /// <para></para>
64 //             /// </returns>
65 //             [MethodImpl(MethodImplOptions.AggressiveInlining)]
66 //             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);

```

```

67 //
68 //          /// <summary>
69 //          /// <para>
70 //          /// Gets the hash code using the specified pair.
71 //          /// </para>
72 //          /// <para></para>
73 //          /// </summary>
74 //          /// <param name="pair">
75 //          /// <para>The pair.</para>
76 //          /// <para></para>
77 //          /// </param>
78 //          /// <returns>
79 //          /// <para>The int</para>
80 //          /// <para></para>
81 //          /// </returns>
82 //          [MethodImpl(MethodImplOptions.AggressiveInlining)]
83 //          public int GetHashCode(KeyValuePair<IList<TLinkAddress>?, IList<TLinkAddress>?>
↵ pair) => (_listComparer.GetHashCode(pair.Key),
↵ _listComparer.GetHashCode(pair.Value)).GetHashCode();
84 //          }
85 //          private class ItemComparer : IComparer<KeyValuePair<IList<TLinkAddress>?,
↵ IList<TLinkAddress>?>>
86 //          {
87 //              private readonly IListComparer<TLinkAddress> _listComparer;
88 //
89 //              /// <summary>
90 //              /// <para>
91 //              /// Initializes a new <see cref="ItemComparer"/> instance.
92 //              /// </para>
93 //              /// <para></para>
94 //              /// </summary>
95 //              [MethodImpl(MethodImplOptions.AggressiveInlining)]
96 //              public ItemComparer() => _listComparer =
↵ Default<IListComparer<TLinkAddress>>.Instance;
97 //
98 //              /// <summary>
99 //              /// <para>
100 //              /// Compares the left.
101 //              /// </para>
102 //              /// <para></para>
103 //              /// </summary>
104 //              /// <param name="left">
105 //              /// <para>The left.</para>
106 //              /// <para></para>
107 //              /// </param>
108 //              /// <param name="right">
109 //              /// <para>The right.</para>
110 //              /// <para></para>
111 //              /// </param>
112 //              /// <returns>
113 //              /// <para>The intermediate result.</para>
114 //              /// <para></para>
115 //              /// </returns>
116 //              [MethodImpl(MethodImplOptions.AggressiveInlining)]
117 //              public int Compare(KeyValuePair<IList<TLinkAddress>?, IList<TLinkAddress>?> left,
↵ KeyValuePair<IList<TLinkAddress>?, IList<TLinkAddress>?> right)
118 //              {
119 //                  var intermediateResult = _listComparer.Compare(left.Key, right.Key);
120 //                  if (intermediateResult == 0)
121 //                  {
122 //                      intermediateResult = _listComparer.Compare(left.Value, right.Value);
123 //                  }
124 //                  return intermediateResult;
125 //              }
126 //          }
127 //
128 //          /// <summary>
129 //          /// <para>
130 //          /// Initializes a new <see cref="DuplicateSegmentsProvider"/> instance.
131 //          /// </para>
132 //          /// <para></para>
133 //          /// </summary>
134 //          /// <param name="links">
135 //          /// <para>A links.</para>
136 //          /// <para></para>
137 //          /// </param>
138 //          /// <param name="sequences">

```



```

139 //          /// <para>A sequences.</para>
140 //          /// <para></para>
141 //          /// </param>
142 //          [MethodImpl(MethodImplOptions.AggressiveInlining)]
143 //          public DuplicateSegmentsProvider(ILinks<TLinkAddress> links, ILinks<TLinkAddress>
↵ sequences)
144 //              : base(minimumStringSegmentLength: 2)
145 //          {
146 //              _links = links;
147 //              _sequences = sequences;
148 //          }
149 //
150 //          /// <summary>
151 //          /// <para>
152 //          /// Gets this instance.
153 //          /// </para>
154 //          /// <para></para>
155 //          /// </summary>
156 //          /// <returns>
157 //          /// <para>The result list.</para>
158 //          /// <para></para>
159 //          /// </returns>
160 //          [MethodImpl(MethodImplOptions.AggressiveInlining)]
161 //          public IList<KeyValuePair<IList<TLinkAddress>?, IList<TLinkAddress>?>> Get()
162 //          {
163 //              _groups = new HashSet<KeyValuePair<IList<TLinkAddress>?,
↵ IList<TLinkAddress>?>>(Default<ItemEquilityComparer>.Instance);
164 //              var links = _links;
165 //              var count = links.Count();
166 //              _visited = new BitString(_addressToInt64Converter.Convert(count) + 1L);
167 //              links.Each(link =>
168 //              {
169 //                  var linkIndex = links.GetIndex(link);
170 //                  var linkBitIndex = _addressToInt64Converter.Convert(linkIndex);
171 //                  var constants = links.Constants;
172 //                  if (!_visited.Get(linkBitIndex))
173 //                  {
174 //                      var sequenceElements = new List<TLinkAddress>();
175 //                      var filler = new ListFiller<TLinkAddress, TLinkAddress>(sequenceElements,
↵ constants.Break);
176 //                      _sequences.Each(filler.AddSkipFirstAndReturnConstant, new
↵ LinkAddress<TLinkAddress>(linkIndex));
177 //                      if (sequenceElements.Count > 2)
178 //                      {
179 //                          WalkAll(sequenceElements);
180 //                      }
181 //                      return constants.Continue;
182 //                  }
183 //              });
184 //              var resultList = _groups.ToList();
185 //              var comparer = Default<ItemComparer>.Instance;
186 //              resultList.Sort(comparer);
187 //              #if DEBUG
188 //              foreach (var item in resultList)
189 //              {
190 //                  PrintDuplicates(item);
191 //              }
192 //              #endif
193 //              return resultList;
194 //          }
195 //
196 //          /// <summary>
197 //          /// <para>
198 //          /// Creates the segment using the specified elements.
199 //          /// </para>
200 //          /// <para></para>
201 //          /// </summary>
202 //          /// <param name="elements">
203 //          /// <para>The elements.</para>
204 //          /// <para></para>
205 //          /// </param>
206 //          /// <param name="offset">
207 //          /// <para>The offset.</para>
208 //          /// <para></para>
209 //          /// </param>
210 //          /// <param name="length">
211 //          /// <para>The length.</para>

```

```

212 //          /// <para></para>
213 //          /// </param>
214 //          /// <returns>
215 //          /// <para>A segment of t link</para>
216 //          /// <para></para>
217 //          /// </returns>
218 //          [MethodImpl(MethodImplOptions.AggressiveInlining)]
219 //          protected override Segment<TLinkAddress> CreateSegment(IList<TLinkAddress>? elements,
↵ int offset, int length) => new Segment<TLinkAddress>(elements, offset, length);
220 //
221 //          /// <summary>
222 //          /// <para>
223 //          /// Ons the dubuplicate found using the specified segment.
224 //          /// </para>
225 //          /// <para></para>
226 //          /// </summary>
227 //          /// <param name="segment">
228 //          /// <para>The segment.</para>
229 //          /// <para></para>
230 //          /// </param>
231 //          [MethodImpl(MethodImplOptions.AggressiveInlining)]
232 //          protected override void OnDubuplicateFound(Segment<TLinkAddress> segment)
233 //          {
234 //              var duplicates = CollectDuplicatessForSegment(segment);
235 //              if (duplicates.Count > 1)
236 //              {
237 //                  _groups.Add(new KeyValuePair<IList<TLinkAddress>?,
↵ IList<TLinkAddress>?>(segment.ToArray(), duplicates));
238 //              }
239 //          }
240 //          [MethodImpl(MethodImplOptions.AggressiveInlining)]
241 //          private List<TLinkAddress> CollectDuplicatessForSegment(Segment<TLinkAddress> segment)
242 //          {
243 //              var duplicates = new List<TLinkAddress>();
244 //              var readAsElement = new HashSet<TLinkAddress>();
245 //              var restrictions = segment.ShiftRight();
246 //              var constants = _links.Constants;
247 //              restrictions[0] = constants.Any;
248 //              _sequences.Each(restrictions, sequence =>
249 //              {
250 //                  var sequenceIndex = sequence[constants.IndexPart];
251 //                  duplicates.Add(sequenceIndex);
252 //                  readAsElement.Add(sequenceIndex);
253 //                  return constants.Continue;
254 //              });
255 //              if (duplicates.Any(x => _visited.Get(_addressToInt64Converter.Convert(x))))
256 //              {
257 //                  return new List<TLinkAddress>();
258 //              }
259 //              foreach (var duplicate in duplicates)
260 //              {
261 //                  var duplicateBitIndex = _addressToInt64Converter.Convert(duplicate);
262 //                  _visited.Set(duplicateBitIndex);
263 //              }
264 //              if (_sequences is Sequences sequencesExperiments)
265 //              {
266 //                  var partiallyMatched = sequencesExperiments.GetAllPartiallyMatchingSequences4_1
↵ ((HashSet<ulong>)(object)readAsElement,
↵ (IList<ulong>)segment);
267 //                  foreach (var partiallyMatchedSequence in partiallyMatched)
268 //                  {
269 //                      var sequenceIndex =
↵ _uInt64ToAddressConverter.Convert(partiallyMatchedSequence);
270 //                      duplicates.Add(sequenceIndex);
271 //                  }
272 //              }
273 //              duplicates.Sort();
274 //              return duplicates;
275 //          }
276 //          [MethodImpl(MethodImplOptions.AggressiveInlining)]
277 //          private void PrintDuplicatess(KeyValuePair<IList<TLinkAddress>?, IList<TLinkAddress>?>
↵ duplicatesItem)
278 //          {
279 //              if (!(_links is ILinks<ulong> ulongLinks))
280 //              {
281 //                  return;
282 //              }

```

```

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 formattedSequenceStructure = 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(formattedSequenceStructure);
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">

```

```

53     /// <para>The source.</para>
54     /// <para></para>
55     /// </param>
56     /// <param name="target">
57     /// <para>The target.</para>
58     /// <para></para>
59     /// </param>
60     /// <returns>
61     /// <para>A link frequency of t link</para>
62     /// <para></para>
63     /// </returns>
64     [MethodImpl(MethodImplOptions.AggressiveInlining)]
65     public LinkFrequency<TLinkAddress> GetFrequency(TLinkAddress source, TLinkAddress target)
66     {
67         var doublet = new Doublet<TLinkAddress>(source, target);
68         return GetFrequency(ref doublet);
69     }
70
71     /// <summary>
72     /// <para>
73     /// Gets the frequency using the specified doublet.
74     /// </para>
75     /// <para></para>
76     /// </summary>
77     /// <param name="doublet">
78     /// <para>The doublet.</para>
79     /// <para></para>
80     /// </param>
81     /// <returns>
82     /// <para>The data.</para>
83     /// <para></para>
84     /// </returns>
85     [MethodImpl(MethodImplOptions.AggressiveInlining)]
86     public LinkFrequency<TLinkAddress> GetFrequency(ref Doublet<TLinkAddress> doublet)
87     {
88         _doubletsCache.TryGetValue(doublet, out LinkFrequency<TLinkAddress> data);
89         return data;
90     }
91
92     /// <summary>
93     /// <para>
94     /// Increments the frequencies using the specified sequence.
95     /// </para>
96     /// <para></para>
97     /// </summary>
98     /// <param name="sequence">
99     /// <para>The sequence.</para>
100    /// <para></para>
101    /// </param>
102    [MethodImpl(MethodImplOptions.AggressiveInlining)]
103    public void IncrementFrequencies(ICollection<TLinkAddress>? sequence)
104    {
105        for (var i = 1; i < sequence.Count; i++)
106        {
107            IncrementFrequency(sequence[i - 1], sequence[i]);
108        }
109    }
110
111    /// <summary>
112    /// <para>
113    /// Increments the frequency using the specified source.
114    /// </para>
115    /// <para></para>
116    /// </summary>
117    /// <param name="source">
118    /// <para>The source.</para>
119    /// <para></para>
120    /// </param>
121    /// <param name="target">
122    /// <para>The target.</para>
123    /// <para></para>
124    /// </param>
125    /// <returns>
126    /// <para>A link frequency of t link</para>
127    /// <para></para>
128    /// </returns>
129    [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(ICollection<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);

```

```

206     }
207     return data;
208 }
209
210 /// <summary>
211 /// <para>
212 /// Validates the frequencies.
213 /// </para>
214 /// <para></para>
215 /// </summary>
216 /// <exception cref="InvalidOperationException">
217 /// <para>Frequencies validation failed.</para>
218 /// <para></para>
219 /// </exception>
220 [MethodImpl(MethodImplOptions.AggressiveInlining)]
221 public void ValidateFrequencies()
222 {
223     foreach (var entry in _doubletsCache)
224     {
225         var value = entry.Value;
226         var linkIndex = value.Link;
227         if (!_equalityComparer.Equals(linkIndex, default))
228         {
229             var frequency = value.Frequency;
230             var count = _frequencyCounter.Count(linkIndex);
231             // TODO: Why `frequency` always greater than `count` by 1?
232             if (((_comparer.Compare(frequency, count) > 0) &&
233                 ↪ (_comparer.Compare(Arithmetic.Subtract(frequency, count), _one) > 0))
234                 || ((_comparer.Compare(count, frequency) > 0) &&
235                 ↪ (_comparer.Compare(Arithmetic.Subtract(count, frequency), _one) > 0)))
236             {
237                 throw new InvalidOperationException("Frequencies validation failed.");
238             }
239             //else
240             //{
241             //    if (value.Frequency > 0)
242             //    {
243             //        var frequency = value.Frequency;
244             //        linkIndex = _createLink(entry.Key.Source, entry.Key.Target);
245             //        var count = _countLinkFrequency(linkIndex);
246             //        if ((frequency > count && frequency - count > 1) || (count > frequency
247             //            ↪ && count - frequency > 1))
248             //            throw new InvalidOperationException("Frequencies validation
249             //            ↪ failed.");
250             //    }
251             //}
252         }
253     }
254 }

```

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

```

1 using System.Runtime.CompilerServices;
2 using Platform.Numbers;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
7 {
8     /// <summary>
9     /// <para>
10    /// Represents the link frequency.
11    /// </para>
12    /// <para></para>
13    /// </summary>
14    public class LinkFrequency<TLinkAddress>
15    {
16        /// <summary>
17        /// <para>
18        /// Gets or sets the frequency value.
19        /// </para>
20        /// <para></para>
21        /// </summary>
22        public TLinkAddress Frequency { get; set; }
23        /// <summary>
24        /// <para>

```

```

25     /// Gets or sets the link value.
26     /// </para>
27     /// <para></para>
28     /// </summary>
29     public TLinkAddress Link { get; set; }
30
31     /// <summary>
32     /// <para>
33     /// Initializes a new <see cref="LinkFrequency"/> instance.
34     /// </para>
35     /// <para></para>
36     /// </summary>
37     /// <param name="frequency">
38     /// <para>A frequency.</para>
39     /// <para></para>
40     /// </param>
41     /// <param name="link">
42     /// <para>A link.</para>
43     /// <para></para>
44     /// </param>
45     [MethodImpl(MethodImplOptions.AggressiveInlining)]
46     public LinkFrequency(TLinkAddress frequency, TLinkAddress link)
47     {
48         Frequency = frequency;
49         Link = link;
50     }
51
52     /// <summary>
53     /// <para>
54     /// Initializes a new <see cref="LinkFrequency"/> instance.
55     /// </para>
56     /// <para></para>
57     /// </summary>
58     [MethodImpl(MethodImplOptions.AggressiveInlining)]
59     public LinkFrequency() { }
60
61     /// <summary>
62     /// <para>
63     /// Increments the frequency.
64     /// </para>
65     /// <para></para>
66     /// </summary>
67     [MethodImpl(MethodImplOptions.AggressiveInlining)]
68     public void IncrementFrequency() => Frequency =
        ↪ Arithmetic<TLinkAddress>.Increment(Frequency);
69
70     /// <summary>
71     /// <para>
72     /// Decrements the frequency.
73     /// </para>
74     /// <para></para>
75     /// </summary>
76     [MethodImpl(MethodImplOptions.AggressiveInlining)]
77     public void DecrementFrequency() => Frequency =
        ↪ Arithmetic<TLinkAddress>.Decrement(Frequency);
78
79     /// <summary>
80     /// <para>
81     /// Returns the string.
82     /// </para>
83     /// <para></para>
84     /// </summary>
85     /// <returns>
86     /// <para>The string</para>
87     /// <para></para>
88     /// </returns>
89     [MethodImpl(MethodImplOptions.AggressiveInlining)]
90     public override string ToString() => $"F: {Frequency}, L: {Link}";
91 }
92 }

```

1.14 ./csharp/Platform.Data.Doublets.Sequences/Frequencies/Cache/LinkToltsFrequencyValueConverter.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.Sequences.Frequencies.Cache

```

```

7 {
8     /// <summary>
9     /// <para>
10    /// Represents the frequencies cache based link to its frequency number converter.
11    /// </para>
12    /// <para></para>
13    /// </summary>
14    /// <seealso cref="IConverter{Doublet{TLinkAddress}, TLinkAddress}"/>
15    public class FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<TLinkAddress> :
        ↳ IConverter<Doublet<TLinkAddress>, TLinkAddress>
16    {
17        private readonly LinkFrequenciesCache<TLinkAddress> _cache;
18
19        /// <summary>
20        /// <para>
21        /// Initializes a new <see
22        ↳ cref="FrequenciesCacheBasedLinkToItsFrequencyNumberConverter"/> instance.
23        /// </para>
24        /// <para></para>
25        /// </summary>
26        /// <param name="cache">
27        /// <para>A cache.</para>
28        /// <para></para>
29        /// </param>
30        [MethodImpl(MethodImplOptions.AggressiveInlining)]
31        public FrequenciesCacheBasedLinkToItsFrequencyNumberConverter(LinkFrequenciesCache<TLink
32        ↳ Address> cache) => _cache =
33        ↳ cache;
34
35        /// <summary>
36        /// <para>
37        /// Converts the source.
38        /// </para>
39        /// <para></para>
40        /// </summary>
41        /// <param name="source">
42        /// <para>The source.</para>
43        /// <para></para>
44        /// </param>
45        /// <returns>
46        /// <para>The link</para>
47        /// <para></para>
48        /// </returns>
49        [MethodImpl(MethodImplOptions.AggressiveInlining)]
50        public TLinkAddress Convert(Doublet<TLinkAddress> source) => _cache.GetFrequency(ref
51        ↳ source).Frequency;
52    }
53 }

```

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

```

1 using System.Runtime.CompilerServices;
2 using Platform.Interfaces;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
7 {
8     /// <summary>
9     /// <para>
10    /// Represents the marked sequence symbol frequency one off counter.
11    /// </para>
12    /// <para></para>
13    /// </summary>
14    /// <seealso cref="SequenceSymbolFrequencyOneOffCounter{TLinkAddress}"/>
15    public class MarkedSequenceSymbolFrequencyOneOffCounter<TLinkAddress> :
        ↳ SequenceSymbolFrequencyOneOffCounter<TLinkAddress>
16    {
17        private readonly ICriterionMatcher<TLinkAddress> _markedSequenceMatcher;
18
19        /// <summary>
20        /// <para>
21        /// Initializes a new <see cref="MarkedSequenceSymbolFrequencyOneOffCounter"/> instance.
22        /// </para>
23        /// <para></para>
24        /// </summary>
25        /// <param name="links">
26        /// <para>A links.</para>
27        /// <para></para>

```



```

28     /// </param>
29     /// <param name="markedSequenceMatcher">
30     /// <para>A marked sequence matcher.</para>
31     /// <para></para>
32     /// </param>
33     /// <param name="sequenceLink">
34     /// <para>A sequence link.</para>
35     /// <para></para>
36     /// </param>
37     /// <param name="symbol">
38     /// <para>A symbol.</para>
39     /// <para></para>
40     /// </param>
41     [MethodImpl(MethodImplOptions.AggressiveInlining)]
42     public MarkedSequenceSymbolFrequencyOneOffCounter(ILinks<TLinkAddress> links,
43         ↪ ICriterionMatcher<TLinkAddress> markedSequenceMatcher, TLinkAddress sequenceLink,
44         ↪ TLinkAddress symbol)
45         : base(links, sequenceLink, symbol)
46         => _markedSequenceMatcher = markedSequenceMatcher;
47
48     /// <summary>
49     /// <para>
50     /// Counts this instance.
51     /// </para>
52     /// <para></para>
53     /// </summary>
54     /// <returns>
55     /// <para>The link</para>
56     /// <para></para>
57     /// </returns>
58     [MethodImpl(MethodImplOptions.AggressiveInlining)]
59     public override TLinkAddress Count()
60     {
61         if (!_markedSequenceMatcher.IsMatched(_sequenceLink))
62         {
63             return default;
64         }
65         return base.Count();
66     }
67 }
68 }
```

1.16 `./csharp/Platform.Data.Doublets.Sequences/Frequencies/Counters/SequenceSymbolFrequencyOneOffCounter`

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3 using Platform.Interfaces;
4 using Platform.Numbers;
5 using Platform.Data.Sequences;
6
7 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9 namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
10 {
11     /// <summary>
12     /// <para>
13     /// Represents the sequence symbol frequency one off counter.
14     /// </para>
15     /// <para></para>
16     /// </summary>
17     /// <seealso cref="ICounter{TLinkAddress}"/>
18     public class SequenceSymbolFrequencyOneOffCounter<TLinkAddress> : ICounter<TLinkAddress>
19     {
20         private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
21             ↳ EqualityComparer<TLinkAddress>.Default;
22         private static readonly Comparer<TLinkAddress> _comparer =
23             ↳ Comparer<TLinkAddress>.Default;
24
25         /// <summary>
26         /// <para>
27         /// The links.
28         /// </para>
29         /// <para></para>
30         /// </summary>
31         protected readonly ILinks<TLinkAddress> _links;
32
33         /// <summary>
34         /// <para>
35         /// The sequence link.
36         /// </para>
37         /// <para></para>
38         /// </summary>
39         protected readonly ISequenceLink<TLinkAddress> _sequenceLink;
40     }
41 }

```

```

35     /// </summary>
36     protected readonly TLinkAddress _sequenceLink;
37     /// <summary>
38     /// <para>
39     /// The symbol.
40     /// </para>
41     /// <para></para>
42     /// </summary>
43     protected readonly TLinkAddress _symbol;
44     /// <summary>
45     /// <para>
46     /// The total.
47     /// </para>
48     /// <para></para>
49     /// </summary>
50     protected TLinkAddress _total;
51
52     /// <summary>
53     /// <para>
54     /// Initializes a new <see cref="SequenceSymbolFrequencyOneOffCounter"/> instance.
55     /// </para>
56     /// <para></para>
57     /// </summary>
58     /// <param name="links">
59     /// <para>A links.</para>
60     /// <para></para>
61     /// </param>
62     /// <param name="sequenceLink">
63     /// <para>A sequence link.</para>
64     /// <para></para>
65     /// </param>
66     /// <param name="symbol">
67     /// <para>A symbol.</para>
68     /// <para></para>
69     /// </param>
70     [MethodImpl(MethodImplOptions.AggressiveInlining)]
71     public SequenceSymbolFrequencyOneOffCounter(ILinks<TLinkAddress> links, TLinkAddress
72     ↪ sequenceLink, TLinkAddress symbol)
73     {
74         _links = links;
75         _sequenceLink = sequenceLink;
76         _symbol = symbol;
77         _total = default;
78     }
79
80     /// <summary>
81     /// <para>
82     /// Counts this instance.
83     /// </para>
84     /// <para></para>
85     /// </summary>
86     /// <returns>
87     /// <para>The total.</para>
88     /// <para></para>
89     /// </returns>
90     [MethodImpl(MethodImplOptions.AggressiveInlining)]
91     public virtual TLinkAddress Count()
92     {
93         if (_comparer.Compare(_total, default) > 0)
94         {
95             return _total;
96         }
97         StopableSequenceWalker.WalkRight(_sequenceLink, _links.GetSource, _links.GetTarget,
98         ↪ IsElement, VisitElement);
99         return _total;
100     }
101
102     [MethodImpl(MethodImplOptions.AggressiveInlining)]
103     private bool IsElement(TLinkAddress x) => _equalityComparer.Equals(x, _symbol) ||
104     ↪ _links.IsPartialPoint(x); // TODO: Use SequenceElementCriteriaMatcher instead of
105     ↪ IsPartialPoint
106
107     [MethodImpl(MethodImplOptions.AggressiveInlining)]
108     private bool VisitElement(TLinkAddress element)
109     {
110         if (_equalityComparer.Equals(element, _symbol))
111         {
112             _total = Arithmetic.Increment(_total);
113         }
114         return true;
115     }

```

```

109     }
110 }
111 }

```

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

```

1  using System.Runtime.CompilerServices;
2  using Platform.Interfaces;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
7  {
8      /// <summary>
9      /// <para>
10     /// Represents the total marked sequence symbol frequency counter.
11     /// </para>
12     /// <para></para>
13     /// </summary>
14     /// <seealso cref="ICounter{TLinkAddress, TLinkAddress}"/>
15     public class TotalMarkedSequenceSymbolFrequencyCounter<TLinkAddress> :
16         ↪ ICounter<TLinkAddress, TLinkAddress>
17     {
18         private readonly ILinks<TLinkAddress> _links;
19         private readonly ICriterionMatcher<TLinkAddress> _markedSequenceMatcher;
20
21         /// <summary>
22         /// <para>
23         /// Initializes a new <see cref="TotalMarkedSequenceSymbolFrequencyCounter"/> instance.
24         /// </para>
25         /// <para></para>
26         /// </summary>
27         /// <param name="links">
28         /// <para>A links.</para>
29         /// </param>
30         /// <param name="markedSequenceMatcher">
31         /// <para>A marked sequence matcher.</para>
32         /// </param>
33         /// </param>
34         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35         public TotalMarkedSequenceSymbolFrequencyCounter(ILinks<TLinkAddress> links,
36             ↪ ICriterionMatcher<TLinkAddress> markedSequenceMatcher)
37         {
38             _links = links;
39             _markedSequenceMatcher = markedSequenceMatcher;
40
41             /// <summary>
42             /// <para>
43             /// Counts the argument.
44             /// </para>
45             /// <para></para>
46             /// </summary>
47             /// <param name="argument">
48             /// <para>The argument.</para>
49             /// </param>
50             /// </param>
51             /// <returns>
52             /// <para>The link</para>
53             /// </returns>
54             [MethodImpl(MethodImplOptions.AggressiveInlining)]
55             public TLinkAddress Count(TLinkAddress argument) => new
56                 ↪ TotalMarkedSequenceSymbolFrequencyOneOffCounter<TLinkAddress>(_links,
57                 ↪ _markedSequenceMatcher, argument).Count();
58     }
59 }

```

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

```

1  using System.Runtime.CompilerServices;
2  using Platform.Interfaces;
3  using Platform.Numbers;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
8  {
9      /// <summary>
10     /// <para>

```

```

11     /// Represents the total marked sequence symbol frequency one off counter.
12     /// </para>
13     /// <para></para>
14     /// </summary>
15     /// <seealso cref="TotalSequenceSymbolFrequencyOneOffCounter{TLinkAddress}"/>
16     public class TotalMarkedSequenceSymbolFrequencyOneOffCounter<TLinkAddress> :
17         ↳ TotalSequenceSymbolFrequencyOneOffCounter<TLinkAddress>
18     {
19         private readonly ICriterionMatcher<TLinkAddress> _markedSequenceMatcher;
20
21         /// <summary>
22         /// <para>
23         ///     Initializes a new <see cref="TotalMarkedSequenceSymbolFrequencyOneOffCounter"/>
24         ///     ↳ instance.
25         /// </para>
26         /// <para></para>
27         /// </summary>
28         /// <param name="links">
29         /// <para>A links.</para>
30         /// <para></para>
31         /// </param>
32         /// <param name="markedSequenceMatcher">
33         /// <para>A marked sequence matcher.</para>
34         /// <para></para>
35         /// </param>
36         /// <param name="symbol">
37         /// <para>A symbol.</para>
38         /// <para></para>
39         /// </param>
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         public TotalMarkedSequenceSymbolFrequencyOneOffCounter(ILinks<TLinkAddress> links,
42             ↳ ICriterionMatcher<TLinkAddress> markedSequenceMatcher, TLinkAddress symbol)
43             : base(links, symbol)
44             => _markedSequenceMatcher = markedSequenceMatcher;
45
46         /// <summary>
47         /// <para>
48         ///     Counts the sequence symbol frequency using the specified link.
49         /// </para>
50         /// <para></para>
51         /// </summary>
52         /// <param name="link">
53         /// <para>The link.</para>
54         /// <para></para>
55         /// </param>
56         [MethodImpl(MethodImplOptions.AggressiveInlining)]
57         protected override void CountSequenceSymbolFrequency(TLinkAddress link)
58         {
59             var symbolFrequencyCounter = new
60                 ↳ MarkedSequenceSymbolFrequencyOneOffCounter<TLinkAddress>(_links,
61                 ↳ _markedSequenceMatcher, link, _symbol);
62             _total = Arithmetic.Add(_total, symbolFrequencyCounter.Count());
63         }
64     }
65 }

```

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

```

1 using System.Runtime.CompilerServices;
2 using Platform.Interfaces;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
7 {
8     /// <summary>
9     /// <para>
10     ///     Represents the total sequence symbol frequency counter.
11     /// </para>
12     /// <para></para>
13     /// </summary>
14     /// <seealso cref="ICounter{TLinkAddress, TLinkAddress}"/>
15     public class TotalSequenceSymbolFrequencyCounter<TLinkAddress> : ICounter<TLinkAddress,
16         ↳ TLinkAddress>
17     {
18         private readonly ILinks<TLinkAddress> _links;
19
20         /// <summary>
21         /// <para>
22         ///     Initializes a new <see cref="TotalSequenceSymbolFrequencyCounter"/> instance.

```

```

22     /// </para>
23     /// <para></para>
24     /// </summary>
25     /// <param name="links">
26     /// <para>A links.</para>
27     /// <para></para>
28     /// </param>
29     [MethodImpl(MethodImplOptions.AggressiveInlining)]
30     public TotalSequenceSymbolFrequencyCounter(ILinks<TLinkAddress> links) => _links = links;
31
32     /// <summary>
33     /// <para>
34     /// Counts the symbol.
35     /// </para>
36     /// <para></para>
37     /// </summary>
38     /// <param name="symbol">
39     /// <para>The symbol.</para>
40     /// <para></para>
41     /// </param>
42     /// <returns>
43     /// <para>The link</para>
44     /// <para></para>
45     /// </returns>
46     [MethodImpl(MethodImplOptions.AggressiveInlining)]
47     public TLinkAddress Count(TLinkAddress symbol) => new
48     {
49         ↪ TotalSequenceSymbolFrequencyOneOffCounter<TLinkAddress>(_links, symbol).Count();
50     }
51 }

```

1.20 ./csharp/Platform.Data.Doublets.Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyOneOffCounter.cs

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using Platform.Interfaces;
4  using Platform.Numbers;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
9  {
10     /// <summary>
11     /// <para>
12     /// Represents the total sequence symbol frequency one off counter.
13     /// </para>
14     /// <para></para>
15     /// </summary>
16     /// <seealso cref="ICounter{TLinkAddress}"/>
17     public class TotalSequenceSymbolFrequencyOneOffCounter<TLinkAddress> : ICounter<TLinkAddress>
18     {
19         private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
20         ↪ EqualityComparer<TLinkAddress>.Default;
21         private static readonly Comparer<TLinkAddress> _comparer =
22         ↪ Comparer<TLinkAddress>.Default;
23
24         /// <summary>
25         /// <para>
26         /// The links.
27         /// </para>
28         /// <para></para>
29         /// </summary>
30         protected readonly ILinks<TLinkAddress> _links;
31
32         /// <summary>
33         /// <para>
34         /// The symbol.
35         /// </para>
36         /// <para></para>
37         /// </summary>
38         protected readonly TLinkAddress _symbol;
39
40         /// <summary>
41         /// <para>
42         /// The visits.
43         /// </para>
44         /// <para></para>
45         /// </summary>
46         protected readonly HashSet<TLinkAddress> _visits;
47
48         /// <summary>
49         /// <para>
50         /// The total.
51         /// </para>
52     }
53 }

```

```

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

```

```

123     }
124     [MethodImpl(MethodImplOptions.AggressiveInlining)]
125     private TLinkAddress EachElementHandler(ICollection<TLinkAddress>? doublet)
126     {
127         var constants = _links.Constants;
128         var doubletIndex = doublet[constants.IndexPart];
129         if (_visits.Add(doubletIndex))
130         {
131             CountCore(doubletIndex);
132         }
133         return constants.Continue;
134     }
135 }
136 }

```

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

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3 using Platform.Interfaces;
4 using Platform.Converters;
5
6 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8 namespace Platform.Data.Doublets.Sequences.HeightProviders
9 {
10     /// <summary>
11     /// <para>
12     /// Represents the cached sequence height provider.
13     /// </para>
14     /// <para></para>
15     /// </summary>
16     /// <seealso cref="ISequenceHeightProvider{TLinkAddress}"/>
17     public class CachedSequenceHeightProvider<TLinkAddress> :
18         ↳ ISequenceHeightProvider<TLinkAddress>
19     {
20         private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
21             ↳ EqualityComparer<TLinkAddress>.Default;
22         private readonly TLinkAddress _heightPropertyMarker;
23         private readonly ISequenceHeightProvider<TLinkAddress> _baseHeightProvider;
24         private readonly IConverter<TLinkAddress> _addressToUnaryNumberConverter;
25         private readonly IConverter<TLinkAddress> _unaryNumberToAddressConverter;
26         private readonly IProperties<TLinkAddress, TLinkAddress, TLinkAddress> _propertyOperator;
27
28         /// <summary>
29         /// <para>
30         /// Initializes a new <see cref="CachedSequenceHeightProvider"/> instance.
31         /// </para>
32         /// <para></para>
33         /// </summary>
34         /// <param name="baseHeightProvider">
35         /// <para>A base height provider.</para>
36         /// <para></para>
37         /// </param>
38         /// <param name="addressToUnaryNumberConverter">
39         /// <para>A address to unary number converter.</para>
40         /// <para></para>
41         /// </param>
42         /// <param name="unaryNumberToAddressConverter">
43         /// <para>A unary number to address converter.</para>
44         /// <para></para>
45         /// </param>
46         /// <param name="heightPropertyMarker">
47         /// <para>A height property marker.</para>
48         /// <para></para>
49         /// </param>
50         /// <param name="propertyOperator">
51         /// <para>A property operator.</para>
52         /// <para></para>
53         /// </param>
54         [MethodImpl(MethodImplOptions.AggressiveInlining)]
55         public CachedSequenceHeightProvider(
56             ISequenceHeightProvider<TLinkAddress> baseHeightProvider,
57             IConverter<TLinkAddress> addressToUnaryNumberConverter,
58             IConverter<TLinkAddress> unaryNumberToAddressConverter,
59             TLinkAddress heightPropertyMarker,
60             IProperties<TLinkAddress, TLinkAddress, TLinkAddress> propertyOperator)
61         {
62             _heightPropertyMarker = heightPropertyMarker;
63             _baseHeightProvider = baseHeightProvider;
64             _addressToUnaryNumberConverter = addressToUnaryNumberConverter;

```

```

63         _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
64         _propertyOperator = propertyOperator;
65     }
66
67     /// <summary>
68     /// <para>
69     /// Gets the sequence.
70     /// </para>
71     /// <para></para>
72     /// </summary>
73     /// <param name="sequence">
74     /// <para>The sequence.</para>
75     /// <para></para>
76     /// </param>
77     /// <returns>
78     /// <para>The height.</para>
79     /// <para></para>
80     /// </returns>
81     [MethodImpl(MethodImplOptions.AggressiveInlining)]
82     public TLinkAddress Get(TLinkAddress sequence)
83     {
84         TLinkAddress height;
85         var heightValue = _propertyOperator.GetValue(sequence, _heightPropertyMarker);
86         if (_equalityComparer.Equals(heightValue, default))
87         {
88             height = _baseHeightProvider.Get(sequence);
89             heightValue = _addressToUnaryNumberConverter.Convert(height);
90             _propertyOperator.SetValue(sequence, _heightPropertyMarker, heightValue);
91         }
92         else
93         {
94             height = _unaryNumberToAddressConverter.Convert(heightValue);
95         }
96         return height;
97     }
98 }
99 }

```

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

```

1  using System.Runtime.CompilerServices;
2  using Platform.Interfaces;
3  using Platform.Numbers;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Sequences.HeightProviders
8  {
9      /// <summary>
10     /// <para>
11     /// Represents the default sequence right height provider.
12     /// </para>
13     /// <para></para>
14     /// </summary>
15     /// <seealso cref="LinksOperatorBase{TLinkAddress}">
16     /// <seealso cref="ISequenceHeightProvider{TLinkAddress}">
17     public class DefaultSequenceRightHeightProvider<TLinkAddress> :
18         ↳ LinksOperatorBase<TLinkAddress>, ISequenceHeightProvider<TLinkAddress>
19     {
20         private readonly ICriterionMatcher<TLinkAddress> _elementMatcher;
21
22         /// <summary>
23         /// <para>
24         /// Initializes a new <see cref="DefaultSequenceRightHeightProvider"/> instance.
25         /// </para>
26         /// <para></para>
27         /// </summary>
28         /// <param name="links">
29         /// <para>A links.</para>
30         /// <para></para>
31         /// </param>
32         /// <param name="elementMatcher">
33         /// <para>A element matcher.</para>
34         /// <para></para>
35         /// </param>
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         public DefaultSequenceRightHeightProvider(ILinks<TLinkAddress> links,
38             ↳ ICriterionMatcher<TLinkAddress> elementMatcher) : base(links) => _elementMatcher =
39             ↳ elementMatcher;

```



```

38     /// <summary>
39     /// <para>
40     /// Gets the sequence.
41     /// </para>
42     /// <para></para>
43     /// </summary>
44     /// <param name="sequence">
45     /// <para>The sequence.</para>
46     /// <para></para>
47     /// </param>
48     /// <returns>
49     /// <para>The height.</para>
50     /// <para></para>
51     /// </returns>
52     [MethodImpl(MethodImplOptions.AggressiveInlining)]
53     public TLinkAddress Get(TLinkAddress sequence)
54     {
55         var height = default(TLinkAddress);
56         var pairOrElement = sequence;
57         while (!_elementMatcher.IsMatched(pairOrElement))
58         {
59             pairOrElement = _links.GetTarget(pairOrElement);
60             height = Arithmetic.Increment(height);
61         }
62         return height;
63     }
64 }
65 }

```

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

```

1 using Platform.Interfaces;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Data.Doublets.Sequences.HeightProviders
6 {
7     /// <summary>
8     /// <para>
9     /// Defines the sequence height provider.
10    /// </para>
11    /// <para></para>
12    /// </summary>
13    /// <seealso cref="IProvider{TLinkAddress, TLinkAddress}"/>
14    public interface ISequenceHeightProvider<TLinkAddress> : IProvider<TLinkAddress,
15        ↪ TLinkAddress>
16    {
17    }
18 }

```

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

```

1 // using System.Collections.Generic;
2 // using System.Runtime.CompilerServices;
3 // using Platform.Incrementers;
4 //
5 // #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6 //
7 // namespace Platform.Data.Doublets.Incrementers
8 // {
9 //     /// <summary>
10 //     /// <para>
11 //     /// Represents the frequency incrementer.
12 //     /// </para>
13 //     /// <para></para>
14 //     /// </summary>
15 //     /// <seealso cref="LinksOperatorBase{TLinkAddress}"/>
16 //     /// <seealso cref="IIncrementer{TLinkAddress}"/>
17 //     public class FrequencyIncrementer<TLinkAddress> : LinksOperatorBase<TLinkAddress>,
18 //         ↪ IIncrementer<TLinkAddress>
19 //     {
20 //         private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
21 //         ↪ EqualityComparer<TLinkAddress>.Default;
22 //         private readonly TLinkAddress _frequencyMarker;
23 //         private readonly TLinkAddress _unaryOne;
24 //         private readonly IIncrementer<TLinkAddress> _unaryNumberIncrementer;
25 //
26 //         /// <summary>
27 //         /// <para>
28 //         /// Initializes a new <see cref="FrequencyIncrementer"/> 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="frequencyMarker">
35 //          /// <para>A frequency marker.</para>
36 //          /// <para></para>
37 //          /// </param>
38 //          /// <param name="unaryOne">
39 //          /// <para>A unary one.</para>
40 //          /// <para></para>
41 //          /// </param>
42 //          /// <param name="unaryNumberIncrementer">
43 //          /// <para>A unary number incrementer.</para>
44 //          /// <para></para>
45 //          /// </param>
46 //          [MethodImpl(MethodImplOptions.AggressiveInlining)]
47 //          public FrequencyIncrementer(ILinks<TLinkAddress> links, TLinkAddress frequencyMarker,
↪ TLinkAddress unaryOne, IIncrementer<TLinkAddress> unaryNumberIncrementer)
48 //              : base(links)
49 //          {
50 //              _frequencyMarker = frequencyMarker;
51 //              _unaryOne = unaryOne;
52 //              _unaryNumberIncrementer = unaryNumberIncrementer;
53 //          }
54 //
55 //          /// <summary>
56 //          /// <para>
57 //          /// Increments the frequency.
58 //          /// </para>
59 //          /// <para></para>
60 //          /// </summary>
61 //          /// <param name="frequency">
62 //          /// <para>The frequency.</para>
63 //          /// <para></para>
64 //          /// </param>
65 //          /// <returns>
66 //          /// <para>The link</para>
67 //          /// <para></para>
68 //          /// </returns>
69 //          [MethodImpl(MethodImplOptions.AggressiveInlining)]
70 //          public TLinkAddress Increment(TLinkAddress frequency)
71 //          {
72 //              var links = _links;
73 //              if (_equalityComparer.Equals(frequency, default))
74 //              {
75 //                  return links.GetOrCreate(_unaryOne, _frequencyMarker);
76 //              }
77 //              var incrementedSource =
↪ _unaryNumberIncrementer.Increment(links.GetSource(frequency));
78 //              return links.GetOrCreate(incrementedSource, _frequencyMarker);
79 //          }
80 //      }
81 // }

```

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

```

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

```

```

19 //         private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
    ↳ EqualityComparer<TLinkAddress>.Default;
20 //         private readonly TLinkAddress _unaryOne;
21 //
22 //         /// <summary>
23 //         /// <para>
24 //         /// Initializes a new <see cref="UnaryNumberIncrementer"/> instance.
25 //         /// </para>
26 //         /// <para></para>
27 //         /// </summary>
28 //         /// <param name="links">
29 //         /// <para>A links.</para>
30 //         /// <para></para>
31 //         /// </param>
32 //         /// <param name="unaryOne">
33 //         /// <para>A unary one.</para>
34 //         /// <para></para>
35 //         /// </param>
36 //         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37 //         public UnaryNumberIncrementer(ILinks<TLinkAddress> links, TLinkAddress unaryOne) :
    ↳ base(links) => _unaryOne = unaryOne;
38 //
39 //         /// <summary>
40 //         /// <para>
41 //         /// Increments the unary number.
42 //         /// </para>
43 //         /// <para></para>
44 //         /// </summary>
45 //         /// <param name="unaryNumber">
46 //         /// <para>The unary number.</para>
47 //         /// <para></para>
48 //         /// </param>
49 //         /// <returns>
50 //         /// <para>The link</para>
51 //         /// <para></para>
52 //         /// </returns>
53 //         [MethodImpl(MethodImplOptions.AggressiveInlining)]
54 //         public TLinkAddress Increment(TLinkAddress unaryNumber)
55 //         {
56 //             var links = _links;
57 //             if (_equalityComparer.Equals(unaryNumber, _unaryOne))
58 //             {
59 //                 return links.GetOrCreate(_unaryOne, _unaryOne);
60 //             }
61 //             var source = links.GetSource(unaryNumber);
62 //             var target = links.GetTarget(unaryNumber);
63 //             if (_equalityComparer.Equals(source, target))
64 //             {
65 //                 return links.GetOrCreate(unaryNumber, _unaryOne);
66 //             }
67 //             else
68 //             {
69 //                 return links.GetOrCreate(source, Increment(target));
70 //             }
71 //         }
72 //     }
73 // }

```

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

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3 using Platform.Data.Doublets.Sequences.Frequencies.Cache;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Data.Doublets.Sequences.Indexes
8 {
9     /// <summary>
10    /// <para>
11    /// Represents the cached frequency incrementing sequence index.
12    /// </para>
13    /// <para></para>
14    /// </summary>
15    /// <seealso cref="ISequenceIndex{TLinkAddress}"/>
16    public class CachedFrequencyIncrementingSequenceIndex<TLinkAddress> :
    ↳ ISequenceIndex<TLinkAddress>
17    {
18        private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
    ↳ EqualityComparer<TLinkAddress>.Default;

```

```

19 private readonly LinkFrequenciesCache<TLinkAddress> _cache;
20
21 /// <summary>
22 /// <para>
23 /// Initializes a new <see cref="CachedFrequencyIncrementingSequenceIndex"/> instance.
24 /// </para>
25 /// <para></para>
26 /// </summary>
27 /// <param name="cache">
28 /// <para>A cache.</para>
29 /// <para></para>
30 /// </param>
31 [MethodImpl(MethodImplOptions.AggressiveInlining)]
32 public CachedFrequencyIncrementingSequenceIndex(LinkFrequenciesCache<TLinkAddress>
    ↪ cache) => _cache = cache;
33
34 /// <summary>
35 /// <para>
36 /// Determines whether this instance add.
37 /// </para>
38 /// <para></para>
39 /// </summary>
40 /// <param name="sequence">
41 /// <para>The sequence.</para>
42 /// <para></para>
43 /// </param>
44 /// <returns>
45 /// <para>The indexed.</para>
46 /// <para></para>
47 /// </returns>
48 [MethodImpl(MethodImplOptions.AggressiveInlining)]
49 public bool Add(IList<TLinkAddress>? sequence)
50 {
51     var indexed = true;
52     var i = sequence.Count;
53     while (--i >= 1 && (indexed = IsIndexedWithIncrement(sequence[i - 1], sequence[i])))
54         ↪ { }
55     for (; i >= 1; i--)
56     {
57         _cache.IncrementFrequency(sequence[i - 1], sequence[i]);
58     }
59     return indexed;
60 }
61 [MethodImpl(MethodImplOptions.AggressiveInlining)]
62 private bool IsIndexedWithIncrement(TLinkAddress source, TLinkAddress target)
63 {
64     var frequency = _cache.GetFrequency(source, target);
65     if (frequency == null)
66     {
67         return false;
68     }
69     var indexed = !_equalityComparer.Equals(frequency.Frequency, default);
70     if (indexed)
71     {
72         _cache.IncrementFrequency(source, target);
73     }
74     return indexed;
75 }
76
77 /// <summary>
78 /// <para>
79 /// Determines whether this instance might contain.
80 /// </para>
81 /// <para></para>
82 /// </summary>
83 /// <param name="sequence">
84 /// <para>The sequence.</para>
85 /// <para></para>
86 /// </param>
87 /// <returns>
88 /// <para>The indexed.</para>
89 /// <para></para>
90 /// </returns>
91 [MethodImpl(MethodImplOptions.AggressiveInlining)]
92 public bool MightContain(IList<TLinkAddress>? sequence)
93 {
94     var indexed = true;
95     var i = sequence.Count;

```

```

95         while (--i >= 1 && (indexed = IsIndexed(sequence[i - 1], sequence[i]))) { }
96         return indexed;
97     }
98     [MethodImpl(MethodImplOptions.AggressiveInlining)]
99     private bool IsIndexed(TLinkAddress source, TLinkAddress target)
100     {
101         var frequency = _cache.GetFrequency(source, target);
102         if (frequency == null)
103         {
104             return false;
105         }
106         return !_equalityComparer.Equals(frequency.Frequency, default);
107     }
108 }
109 }

```

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

```

1  // using System.Collections.Generic;
2  // using System.Runtime.CompilerServices;
3  // using Platform.Interfaces;
4  // using Platform.Incremeters;
5  //
6  // #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7  //
8  // namespace Platform.Data.Doublets.Sequences.Indexes
9  // {
10 //     /// <summary>
11 //     /// <para>
12 //     /// Represents the frequency incrementing sequence index.
13 //     /// </para>
14 //     /// <para></para>
15 //     /// </summary>
16 //     /// <seealso cref="SequenceIndex{TLinkAddress}"/>
17 //     /// <seealso cref="ISequenceIndex{TLinkAddress}"/>
18 //     public class FrequencyIncrementingSequenceIndex<TLinkAddress> :
19 //     ↪ SequenceIndex<TLinkAddress>, ISequenceIndex<TLinkAddress>
20 //     {
21 //         private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
22 //         ↪ EqualityComparer<TLinkAddress>.Default;
23 //         private readonly IProperty<TLinkAddress, TLinkAddress> _frequencyPropertyOperator;
24 //         private readonly IIncrementer<TLinkAddress> _frequencyIncrementer;
25 //
26 //         /// <summary>
27 //         /// <para>
28 //         /// Initializes a new <see cref="FrequencyIncrementingSequenceIndex"/> instance.
29 //         /// </para>
30 //         /// <para></para>
31 //         /// </summary>
32 //         /// <param name="links">
33 //         /// <para>A links.</para>
34 //         /// <para></para>
35 //         /// </param>
36 //         /// <param name="frequencyPropertyOperator">
37 //         /// <para>A frequency property operator.</para>
38 //         /// <para></para>
39 //         /// </param>
40 //         /// <param name="frequencyIncrementer">
41 //         /// <para>A frequency incrementer.</para>
42 //         /// <para></para>
43 //         /// </param>
44 //         [MethodImpl(MethodImplOptions.AggressiveInlining)]
45 //         public FrequencyIncrementingSequenceIndex(ILinks<TLinkAddress> links,
46 //         ↪ IProperty<TLinkAddress, TLinkAddress> frequencyPropertyOperator, IIncrementer<TLinkAddress>
47 //         ↪ frequencyIncrementer)
48 //         : base(links)
49 //         {
50 //             _frequencyPropertyOperator = frequencyPropertyOperator;
51 //             _frequencyIncrementer = frequencyIncrementer;
52 //         }
53 //
54 //         /// <summary>
55 //         /// <para>
56 //         /// Determines whether this instance add.
57 //         /// </para>
58 //         /// <para></para>
59 //         /// </summary>
60 //         /// <param name="sequence">
61 //         /// <para>The sequence.</para>

```

```

58 //      /// <para></para>
59 //      /// </param>
60 //      /// <returns>
61 //      /// <para>The indexed.</para>
62 //      /// <para></para>
63 //      /// </returns>
64 //      [MethodImpl(MethodImplOptions.AggressiveInlining)]
65 //      public override bool Add(ICollection<TLinkAddress>? sequence)
66 //      {
67 //          var indexed = true;
68 //          var i = sequence.Count;
69 //          while (--i >= 1 && (indexed = IsIndexedWithIncrement(sequence[i - 1],
↵ sequence[i]))) { }
70 //          for (; i >= 1; i--)
71 //          {
72 //              Increment(_links.GetOrCreate(sequence[i - 1], sequence[i]));
73 //          }
74 //          return indexed;
75 //      }
76 //      [MethodImpl(MethodImplOptions.AggressiveInlining)]
77 //      private bool IsIndexedWithIncrement(TLinkAddress source, TLinkAddress target)
78 //      {
79 //          var link = _links.SearchOrDefault(source, target);
80 //          var indexed = !_equalityComparer.Equals(link, default);
81 //          if (indexed)
82 //          {
83 //              Increment(link);
84 //          }
85 //          return indexed;
86 //      }
87 //      [MethodImpl(MethodImplOptions.AggressiveInlining)]
88 //      private void Increment(TLinkAddress link)
89 //      {
90 //          var previousFrequency = _frequencyPropertyOperator.Get(link);
91 //          var frequency = _frequencyIncrementer.Increment(previousFrequency);
92 //          _frequencyPropertyOperator.Set(link, frequency);
93 //      }
94 //      }
95 // }

```

1.28 ./csharp/Platform.Data.Doublets.Sequences/Indexes/ISequenceIndex.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.Indexes
7 {
8     /// <summary>
9     /// <para>
10     /// Defines the sequence index.
11     /// </para>
12     /// <para></para>
13     /// </summary>
14     public interface ISequenceIndex<TLinkAddress>
15     {
16         /// <summary>
17         /// Индексирует последовательность глобально, и возвращает значение,
18         /// определяющие была ли запрошенная последовательность проиндексирована ранее.
19         /// </summary>
20         /// <param name="sequence">Последовательность для индексации.</param>
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         bool Add(ICollection<TLinkAddress>? sequence);
23
24         /// <summary>
25         /// <para>
26         /// Determines whether this instance might contain.
27         /// </para>
28         /// <para></para>
29         /// </summary>
30         /// <param name="sequence">
31         /// <para>The sequence.</para>
32         /// <para></para>
33         /// </param>
34         /// <returns>
35         /// <para>The bool</para>
36         /// <para></para>
37         /// </returns>

```

```

38     [MethodImpl(MethodImplOptions.AggressiveInlining)]
39     bool MightContain(IList<TLinkAddress>? sequence);
40 }
41 }

```

1.29 ./csharp/Platform.Data.Doublets.Sequences/Indexes/SequenceIndex.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.Indexes
7  {
8      /// <summary>
9      /// <para>
10     /// Represents the sequence index.
11     /// </para>
12     /// <para></para>
13     /// </summary>
14     /// <seealso cref="LinksOperatorBase{TLinkAddress}"/>
15     /// <seealso cref="ISequenceIndex{TLinkAddress}"/>
16     public class SequenceIndex<TLinkAddress> : LinksOperatorBase<TLinkAddress>,
17     ↪ ISequenceIndex<TLinkAddress>
18     {
19         private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
20         ↪ EqualityComparer<TLinkAddress>.Default;
21
22         /// <summary>
23         /// <para>
24         /// Initializes a new <see cref="SequenceIndex"/> instance.
25         /// </para>
26         /// <para></para>
27         /// </summary>
28         /// <param name="links">
29         /// <para>A links.</para>
30         /// <para></para>
31         /// </param>
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         public SequenceIndex(ILinks<TLinkAddress> links) : base(links) { }
34
35         /// <summary>
36         /// <para>
37         /// Determines whether this instance add.
38         /// </para>
39         /// <para></para>
40         /// </summary>
41         /// <param name="sequence">
42         /// <para>The sequence.</para>
43         /// <para></para>
44         /// </param>
45         /// <returns>
46         /// <para>The indexed.</para>
47         /// <para></para>
48         /// </returns>
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         public virtual bool Add(IList<TLinkAddress>? sequence)
51         {
52             var indexed = true;
53             var i = sequence.Count;
54             while (--i >= 1 && (indexed =
55             ↪ !_equalityComparer.Equals(_links.SearchOrDefault(sequence[i - 1], sequence[i]),
56             ↪ default))) { }
57             for (; i >= 1; i--)
58             {
59                 _links.GetOrCreate(sequence[i - 1], sequence[i]);
60             }
61             return indexed;
62         }
63
64         /// <summary>
65         /// <para>
66         /// Determines whether this instance might contain.
67         /// </para>
68         /// <para></para>
69         /// </summary>
70         /// <param name="sequence">
71         /// <para>The sequence.</para>
72         /// <para></para>
73         /// </param>

```

```

70     /// <returns>
71     /// <para>The indexed.</para>
72     /// <para></para>
73     /// </returns>
74     [MethodImpl(MethodImplOptions.AggressiveInlining)]
75     public virtual bool MightContain(IList<TLinkAddress>? sequence)
76     {
77         var indexed = true;
78         var i = sequence.Count;
79         while (--i >= 1 && (indexed =
80             ↪ !_equalityComparer.Equals(_links.SearchOrDefault(sequence[i - 1], sequence[i]),
81             ↪ default))) { }
82         return indexed;
83     }
84 }

```

1.30 ./csharp/Platform.Data.Doublets.Sequences/Indexes/SynchronizedSequenceIndex.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.Indexes
7 {
8     /// <summary>
9     /// <para>
10    /// Represents the synchronized sequence index.
11    /// </para>
12    /// <para></para>
13    /// </summary>
14    /// <seealso cref="ISequenceIndex{TLinkAddress}"/>
15    public class SynchronizedSequenceIndex<TLinkAddress> : ISequenceIndex<TLinkAddress>
16    {
17        private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
18            EqualityComparer<TLinkAddress>.Default;
19        private readonly ISynchronizedLinks<TLinkAddress> _links;
20
21        /// <summary>
22        /// <para>
23        /// Initializes a new <see cref="SynchronizedSequenceIndex"/> instance.
24        /// </para>
25        /// <para></para>
26        /// </summary>
27        /// <param name="links">
28        /// <para>A links.</para>
29        /// <para></para>
30        /// </param>
31        [MethodImpl(MethodImplOptions.AggressiveInlining)]
32        public SynchronizedSequenceIndex(ISynchronizedLinks<TLinkAddress> links) => _links =
33            links;
34
35        /// <summary>
36        /// <para>
37        /// Determines whether this instance add.
38        /// </para>
39        /// <para></para>
40        /// </summary>
41        /// <param name="sequence">
42        /// <para>The sequence.</para>
43        /// <para></para>
44        /// </param>
45        /// <returns>
46        /// <para>The indexed.</para>
47        /// <para></para>
48        /// </returns>
49        [MethodImpl(MethodImplOptions.AggressiveInlining)]
50        public bool Add(ICollection<TLinkAddress>? sequence)
51        {
52            var indexed = true;
53            var i = sequence.Count;
54            var links = _links.Unsync;
55            _links.SyncRoot.DoRead(() =>
56            {
57                while (--i >= 0 && (indexed =
58                    !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1],
59                        sequence[i]), default))) { }
60            });
61            if (!indexed)
62            {
63                // ...
64            }
65        }
66    }
67 }

```



```

58     {
59         _links.SyncRoot.DoWrite(() =>
60         {
61             for (; i >= 1; i--)
62             {
63                 links.GetOrCreate(sequence[i - 1], sequence[i]);
64             }
65         });
66     }
67     return indexed;
68 }
69
70 /// <summary>
71 /// <para>
72 /// Determines whether this instance might contain.
73 /// </para>
74 /// <para></para>
75 /// </summary>
76 /// <param name="sequence">
77 /// <para>The sequence.</para>
78 /// <para></para>
79 /// </param>
80 /// <returns>
81 /// <para>The bool</para>
82 /// <para></para>
83 /// </returns>
84 [MethodImpl(MethodImplOptions.AggressiveInlining)]
85 public bool MightContain(ICollection<TLinkAddress>? sequence)
86 {
87     var links = _links.Unsync;
88     return _links.SyncRoot.DoRead(() =>
89     {
90         var indexed = true;
91         var i = sequence.Count;
92         while (--i >= 1 && (indexed =
93             ↪ !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1],
94             ↪ sequence[i]), default))) { }
95         return indexed;
96     });
97 }

```

1.31 ./csharp/Platform.Data.Doublets.Sequences/Indexes/Unindex.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.Indexes
7 {
8     /// <summary>
9     /// <para>
10    /// Represents the unindex.
11    /// </para>
12    /// <para></para>
13    /// </summary>
14    /// <seealso cref="ISequenceIndex{TLinkAddress}"/>
15    public class Unindex<TLinkAddress> : ISequenceIndex<TLinkAddress>
16    {
17        /// <summary>
18        /// <para>
19        /// Determines whether this instance add.
20        /// </para>
21        /// <para></para>
22        /// </summary>
23        /// <param name="sequence">
24        /// <para>The sequence.</para>
25        /// <para></para>
26        /// </param>
27        /// <returns>
28        /// <para>The bool</para>
29        /// <para></para>
30        /// </returns>
31        [MethodImpl(MethodImplOptions.AggressiveInlining)]
32        public virtual bool Add(ICollection<TLinkAddress>? sequence) => false;
33
34        /// <summary>
35        /// <para>

```

```

36     /// Determines whether this instance might contain.
37     /// </para>
38     /// <para></para>
39     /// </summary>
40     /// <param name="sequence">
41     /// <para>The sequence.</para>
42     /// <para></para>
43     /// </param>
44     /// <returns>
45     /// <para>The bool</para>
46     /// <para></para>
47     /// </returns>
48     [MethodImpl(MethodImplOptions.AggressiveInlining)]
49     public virtual bool MightContain(ICollection<TLinkAddress>? sequence) => true;
50 }
51 }

```

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

```

1  using System;
2  using System.Collections;
3  using System.Collections.Generic;
4  using System.Drawing;
5  using System.Linq;
6  using System.Numerics;
7  using System.Text;
8  using Platform.Collections.Stacks;
9  using Platform.Converters;
10 using Platform.Data.Doublets.CriterionMatchers;
11 using Platform.Data.Doublets.Decorators;
12 using Platform.Data.Doublets.Sequences.Converters;
13 using Platform.Data.Doublets.Unicode;
14 using Platform.Numbers;
15 using Platform.Reflection;
16 using Platform.Unsafe;
17
18 namespace Platform.Data.Doublets.Sequences.Numbers.Byte;
19
20 public class ByteListToRawSequenceConverter<TLinkAddress> : LinksDecoratorBase<TLinkAddress>,
    ↳ IConverter<ICollection<byte>, TLinkAddress> where TLinkAddress : struct
21 {
22     public static readonly TLinkAddress MaximumValue = NumericType<TLinkAddress>.MaxValue;
23
24     public static readonly TLinkAddress BitMask = Bit.ShiftRight(MaximumValue, 1);
25
26     public static readonly int BitsSize = NumericType<TLinkAddress>.BitsSize;
27
28     public readonly IConverter<TLinkAddress> AddressToNumberConverter;
29     public readonly IConverter<TLinkAddress> NumberToAddressConverter;
30
31     public readonly IConverter<ICollection<TLinkAddress>, TLinkAddress> ListToSequenceConverter;
32
33     public readonly BalancedVariantConverter<TLinkAddress> BalancedVariantConverter;
34
35     public readonly IConverter<string, TLinkAddress> StringToUnicodeSequenceConverter;
36
37     public readonly IConverter<TLinkAddress, string> UnicodeSequenceToStringConverter;
38
39     public readonly TLinkAddress Type = default;
40
41     public readonly TLinkAddress ByteArrayLengthType;
42
43     public readonly TLinkAddress ByteArrayType;
44
45     public static readonly unchecked Converter<Int32, TLinkAddress> IntToTLinkAddressConverter =
    ↳ unchecked Converter<int, TLinkAddress>.Default;
46     public static readonly unchecked Converter<TLinkAddress, byte> TLinkAddressToByteConverter =
    ↳ unchecked Converter<TLinkAddress, byte>.Default;
47     public ArraySegment<byte> CurrentByteArray;
48     public static readonly int BytesInRawNumberCount = BitsSize / 8;
49     public readonly unchecked Converter<byte, TLinkAddress> ByteToTLinkAddressConverter =
    ↳ unchecked Converter<byte, TLinkAddress>.Default;
50     public TLinkAddress EmptyByteArraySequenceType;
51
52
53     public ByteListToRawSequenceConverter(ILinks<TLinkAddress> links, IConverter<TLinkAddress>
    ↳ addressToNumberConverter, IConverter<TLinkAddress> numberToAddressConverter,
    ↳ IConverter<ICollection<TLinkAddress>, TLinkAddress> listToSequenceConverter,
    ↳ StringToUnicodeSequenceConverter<TLinkAddress> stringToUnicodeSequenceConverter) :
    ↳ base(links)
54     {
55         AddressToNumberConverter = addressToNumberConverter;
56         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(ICollection<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
89     //     : BytesInRawNumberCount;
90     //     byteArrayToSave = byteArrayToSave.Skip(bytesInRawNumberCount).ToArray();
91     // }
92     // var processedRawNumbers = new List<TLinkAddress>(rawNumbers.Count);
93     // for (var j = 0; j < rawNumbers.Count; j++)
94     // {
95     //     var processedRawNumber = rawNumbers[j];
96     //     var nonSavedBitsCount = GetNonSavedBitsCount(j);
97     //     if (nonSavedBitsCount != 0)
98     //     {
99     //         processedRawNumber = Bit.ShiftLeft(processedRawNumber, nonSavedBitsCount);
100     //         var nonSavedBits = Bit.ShiftRight(rawNumbers[j - 1], BitsSize -
101     ↳ nonSavedBitsCount);
102     //         processedRawNumber = Bit.Or(processedRawNumber, nonSavedBits);
103     //     }
104     //     processedRawNumber = AddressToNumberConverter.Convert(processedRawNumber);
105     //     processedRawNumbers.Add(processedRawNumber);
106     // }
107     // TLinkAddress lastRawNumber = default;
108     // var i = 0;
109     // while (byteArrayToSave.Length != 0)
110     // {
111     //     var rawNumber = byteArrayToSave.ToStructure<TLinkAddress>();
112     //     rawNumbers.Add(rawNumber);
113     //     Console.WriteLine("Raw number:");
114     //     Console.WriteLine(TestExtensions.PrettifyBinary<TLinkAddress>(System.Convert.ToSt
115     ↳ ring((ushort)(object)rawNumber,
116     ↳ 2)));
117     //     var nonSavedBitsCount = GetNonSavedBitsCount(i);
118     //     var prevRawNumberWithNonSavedBitsAtStart = Bit.ShiftRight(lastRawNumber, BitsSize
119     ↳ - nonSavedBitsCount);
120     //     var processedRawNumber = Bit.ShiftLeft(rawNumber, nonSavedBitsCount);
121     //     processedRawNumber = Bit.Or(processedRawNumber,
122     ↳ prevRawNumberWithNonSavedBitsAtStart);
123     //     processedRawNumber = Bit.And(processedRawNumber, BitMask);
124     //     Console.WriteLine("Processed raw number:");
125     //     Console.WriteLine(TestExtensions.PrettifyBinary<TLinkAddress>(System.Convert.ToSt
126     ↳ ring((ushort)(object)processedRawNumber,
127     ↳ 2)));
128     //     processedRawNumber = AddressToNumberConverter.Convert(processedRawNumber);

```

```

121 //     var bytesInRawNumberCount = nonSavedBitsCount == 7 ? BytesInRawNumberCount - 1 :
122     ↳ BytesInRawNumberCount;
123 //     byteArrayToSave = byteArrayToSave.Skip(bytesInRawNumberCount).ToArray();
124 //     lastRawNumber = rawNumber;
125 //     i++;
126 //     rawNumbers.Add(processedRawNumber);
127 // }
128 // var notSavedBitsCount = GetNonSavedBitsCount(i);
129 // if ((source.Count != BitsSize) && (source.Count % BytesInRawNumberCount == 0))
130 // {
131 //     lastRawNumber = Bit.ShiftRight(lastRawNumber, BitsSize - notSavedBitsCount);
132 //     lastRawNumber = AddressToNumberConverter.Convert(lastRawNumber);
133 //     rawNumbers.Add(lastRawNumber);
134 // }
135 // Console.WriteLine("Raw numbers");
136 // rawrawNumbers.Reverse();
137 // StringBuilder rawrawNumbersSb = new StringBuilder();
138 // foreach (var rawrawNumber in rawrawNumbers)
139 // {
140 //     rawrawNumbersSb.Append(System.Convert.ToString((ushort)(object)rawrawNumber, 2));
141 // }
142 // Console.WriteLine(rawrawNumbersSb.ToString());
143 //
144 // rawNumbers.Reverse();
145 // Console.WriteLine("Processed raw numbers");
146 // StringBuilder rawNumbersSb = new StringBuilder();
147 // foreach (var rawNumber in rawNumbers)
148 // {
149 //     rawNumbersSb.Append(System.Convert.ToString((ushort)(object)NumberToAddressConver
150     ↳ ter.Convert(rawNumber),
151     ↳ 2));
152 // }
153 // for (int c = 31; c < rawNumbersSb.Length; c += 32)
154 // {
155 //     rawNumbersSb.Remove(c, 1);
156 // }
157 // Console.WriteLine(rawNumbersSb.ToString());
158 // var length = IntToTLinkAddressConverter.Convert(source.Count);
159 // var byteArrayLengthAddress = _links.GetOrCreate(ByteArrayLengthType,
160     ↳ AddressToNumberConverter.Convert(length));
161 // var byteArraySequenceAddress = ListToSequenceConverter.Convert(rawNumbers);
162 // return _links.GetOrCreate(ByteArrayType, _links.GetOrCreate(byteArrayLengthAddress,
163     ↳ byteArraySequenceAddress));
164
165 // var processedRawNumbers = new List<TLinkAddress>();
166 // if (source.Count == 0)
167 // {
168 //     return Links.GetOrCreate(ByteArrayType, EmptyByteArraySequenceType);
169 // }
170 // var byteArrayToSave = source.ToArray();
171 // TLinkAddress lastRawNumber = default;
172 // var i = 0;
173 // while (byteArrayToSave.Length != 0)
174 // {
175 //     var rawNumber = byteArrayToSave.ToStructure<TLinkAddress>();
176 //     var nonSavedBitsCount = GetNonSavedBitsCount(i);
177 //     var prevRawNumberWithNonSavedBitsAtStart = Bit.ShiftRight(lastRawNumber, BitsSize
178     ↳ - nonSavedBitsCount);
179 //     var processedRawNumber = Bit.ShiftLeft(rawNumber, nonSavedBitsCount);
180 //     processedRawNumber = Bit.Or(processedRawNumber,
181     ↳ prevRawNumberWithNonSavedBitsAtStart);
182 //     processedRawNumber = Bit.And(processedRawNumber, BitMask);
183 //     processedRawNumber = AddressToNumberConverter.Convert(processedRawNumber);
184 //     var bytesInRawNumberCount = nonSavedBitsCount == 7 ? BytesInRawNumberCount - 1 :
185     ↳ BytesInRawNumberCount;
186 //     byteArrayToSave = byteArrayToSave.Skip(bytesInRawNumberCount).ToArray();
187 //     lastRawNumber = rawNumber;
188 //     i++;
189 //     processedRawNumbers.Add(processedRawNumber);
190 // }
191 // var notSavedBitsCount = GetNonSavedBitsCount(i);
192 // if (source.Count % BytesInRawNumberCount == 0)
193 // {
194 //     lastRawNumber = Bit.ShiftRight(lastRawNumber, BitsSize - notSavedBitsCount);
195 //     lastRawNumber = AddressToNumberConverter.Convert(lastRawNumber);
196 //     processedRawNumbers.Add(lastRawNumber);
197 // }

```

```

190 // var length = IntToTLinkAddressConverter.Convert(source.Count);
191 // var byteArrayLengthAddress = _links.GetOrCreate(ByteArrayLengthType,
    → AddressToNumberConverter.Convert(length));
192 // var byteArraySequenceAddress = ListToSequenceConverter.Convert(processedRawNumbers);
193 // return _links.GetOrCreate(ByteArrayType, _links.GetOrCreate(byteArrayLengthAddress,
    → byteArraySequenceAddress));

194
195
196 // List<TLinkAddress> rawNumberList = new(source.Count / BytesInRawNumberCount +
    → source.Count);
197 // var byteArray = source.ToArray();
198 // var i = 0;
199 // TLinkAddress rawNumberWithNonSavedBitsAtStart = default;
200 // int lastNotSavedBitsCount = 0;
201 // bool hasNotSavedBits = false;
202 // while (byteArray.Length != 0)
203 // {
204 //     // if (i % 8 == 0)
205 //     // if (i == 0)
206 //     {
207 //         var rawNumber = byteArray.ToStructure<TLinkAddress>();
208 //         hasNotSavedBits = byteArray.Length >= BytesInRawNumberCount;
209 //         // var output = TestExtensions.PrettifyBinary<uint>(System.Convert.ToString((
    → ushort)(object)rawNumber,
    → 2));
210 //         // Console.WriteLine(output);
211 //         rawNumberWithNonSavedBitsAtStart = Bit.ShiftRight(rawNumber, BitsSize - 1);
212 //         lastNotSavedBitsCount = 1;
213 //         rawNumber = Bit.And(rawNumber, BitMask);
214 //         var output = TestExtensions.PrettifyBinary<uint>(System.Convert.ToString((ush
    → ort)(object)rawNumber,
    → 2));
215 //         Console.WriteLine(output);
216 //         rawNumber = AddressToNumberConverter.Convert(rawNumber);
217 //         // output = TestExtensions.PrettifyBinary<uint>(System.Convert.ToString((usho
    → rt)(object)rawNumber,
    → 2));
218 //         // Console.WriteLine(output);
219 //         rawNumberList.Add(rawNumber);
220 //         byteArray = byteArray.Skip(BytesInRawNumberCount).ToArray();
221 //     }
222 //     else
223 //     {
224 //         // var lastNotSavedBitsCount = i % 8;
225 //         // if (lastNotSavedBitsCount == 0)
226 //         // {
227 //         //     lastNotSavedBitsCount = 1;
228 //         // }
229 //         // // if (lastNotSavedBitsCount % BitsSize == 0)
230 //         // // {
231 //         // //     lastNotSavedBitsCount = 0;
232 //         // // }
233 //         //
234 //         //
235 //         var newNotSavedBitsCount = lastNotSavedBitsCount + 1;
236 //         var rawNumber = byteArray.ToStructure<TLinkAddress>();
237 //         hasNotSavedBits = byteArray.Length >= BytesInRawNumberCount;
238 //         // TODO: Check for lastNotSavedBitsCount == 0
239 //         var newNotSavedBits = Bit.ShiftRight(rawNumber, BitsSize -
    → newNotSavedBitsCount);
240 //         // Shift left for non saved bits from previous raw number
241 //         rawNumber = Bit.ShiftLeft(rawNumber, lastNotSavedBitsCount);
242 //         // Put non saved bits at the start
243 //         rawNumber = Bit.Or(rawNumber, rawNumberWithNonSavedBitsAtStart);
244 //         // Mask last bit
245 //         rawNumber = Bit.And(rawNumber, BitMask);
246 //         rawNumberWithNonSavedBitsAtStart = newNotSavedBits;
247 //         var output = TestExtensions.PrettifyBinary<uint>(System.Convert.ToString((ush
    → ort)(object)rawNumber,
    → 2));
248 //         Console.WriteLine(output);
249 //         rawNumber = AddressToNumberConverter.Convert(rawNumber);
250 //         rawNumberList.Add(rawNumber);
251 //         var bytesInRawNumberCount = newNotSavedBitsCount % 7 != 0 ?
    → BytesInRawNumberCount : BytesInRawNumberCount - 1;
252 //         Console.WriteLine(bytesInRawNumberCount);
253 //         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)j
272         ↪ (object)rowNumberWithNonSavedBitsAtStart,
273         ↪ 2));
274         // Console.WriteLine(output);
275         // var lastRawNumber =
276         ↪ AddressToNumberConverter.Convert(rowNumberWithNonSavedBitsAtStart);
277         // rowNumberList.Add(lastRawNumber);
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>,
22 ↪ IConverter<TLinkAddress, IList<byte>> where TLinkAddress : struct
23 {
24     public static readonly EqualityComparer<TLinkAddress> EqualityComparer =
25     ↪ EqualityComparer<TLinkAddress>.Default;
26
27     public static readonly TLinkAddress MaximumValue = NumericType<TLinkAddress>.MaxValue;
28
29     public static readonly TLinkAddress BitMask = Bit.ShiftRight(MaximumValue, 1);
30
31     public static readonly int BitsSize = NumericType<TLinkAddress>.BitsSize;
32
33     public readonly IConverter<TLinkAddress> NumberToAddressConverter;
34
35     public readonly IConverter<IList<TLinkAddress>, TLinkAddress> ListToSequenceConverter;
36
37     public readonly IConverter<string, TLinkAddress> StringToUnicodeSequenceConverter;
38
39     public readonly IConverter<TLinkAddress, string> UnicodeSequenceToStringConverter;
40
41     public readonly BalancedVariantConverter<TLinkAddress> BalancedVariantConverter;
42     public static readonly UncheckedConverter<TLinkAddress, byte> TLinkAddressToByteConverter =
43     ↪ UncheckedConverter<TLinkAddress, byte>.Default;
44     public static readonly int BytesInRawNumberCount = BitsSize / 8;
45
46     public readonly TLinkAddress Type;
47
48     public RawSequenceToByteListConverter(TLinkAddress type,
49     ↪ IConverter<TLinkAddress> numberToAddressConverter,
50     ↪ IConverter<IList<TLinkAddress>, TLinkAddress> listToSequenceConverter,
51     ↪ IConverter<string, TLinkAddress> stringToUnicodeSequenceConverter,
52     ↪ IConverter<TLinkAddress, string> unicodeSequenceToStringConverter,
53     ↪ BalancedVariantConverter<TLinkAddress> balancedVariantConverter,
54     ↪ UncheckedConverter<TLinkAddress, byte> tLinkAddressToByteConverter,
55     ↪ int bytesInRawNumberCount)
56     {
57         Type = type;
58         NumberToAddressConverter = numberToAddressConverter;
59         ListToSequenceConverter = listToSequenceConverter;
60         StringToUnicodeSequenceConverter = stringToUnicodeSequenceConverter;
61         UnicodeSequenceToStringConverter = unicodeSequenceToStringConverter;
62         BalancedVariantConverter = balancedVariantConverter;
63         TLinkAddressToByteConverter = tLinkAddressToByteConverter;
64         BytesInRawNumberCount = bytesInRawNumberCount;
65     }
66
67     public IList<byte> Convert(TLinkAddress value)
68     {
69         return Convert(value, 0);
70     }
71
72     public IList<byte> Convert(TLinkAddress value, int start)
73     {
74         var result = new byte[BytesInRawNumberCount];
75         for (int i = start; i < result.Length; i++)
76         {
77             result[i] = TLinkAddressToByteConverter.Convert(value);
78         }
79         return result;
80     }
81
82     public TLinkAddress Convert(IList<byte> value)
83     {
84         return Convert(value, 0);
85     }
86
87     public TLinkAddress Convert(IList<byte> value, int start)
88     {
89         var result = NumberToAddressConverter.Convert(0);
90         for (int i = start; i < value.Count; i++)
91         {
92             result = NumberToAddressConverter.Convert(result | (value[i] << (i - start) * 8));
93         }
94         return result;
95     }
96
97     public string Convert(TLinkAddress value)
98     {
99         return Convert(value, 0);
100    }
101
102    public string Convert(TLinkAddress value, int start)
103    {
104        var result = new string[BytesInRawNumberCount];
105        for (int i = start; i < result.Length; i++)
106        {
107            result[i] = Convert(value, i);
108        }
109        return string.Concat(result);
110    }
111
112    public byte Convert(TLinkAddress value, int start)
113    {
114        return TLinkAddressToByteConverter.Convert(value);
115    }
116
117    public TLinkAddress Convert(string value)
118    {
119        return Convert(value, 0);
120    }
121
122    public TLinkAddress Convert(string value, int start)
123    {
124        var result = NumberToAddressConverter.Convert(0);
125        for (int i = start; i < value.Length; i++)
126        {
127            result = NumberToAddressConverter.Convert(result | (value[i] << (i - start) * 8));
128        }
129        return result;
130    }
131
132    public TLinkAddress Convert(char value)
133    {
134        return Convert(value, 0);
135    }
136
137    public TLinkAddress Convert(char value, int start)
138    {
139        var result = NumberToAddressConverter.Convert(0);
140        for (int i = start; i < value.Length; i++)
141        {
142            result = NumberToAddressConverter.Convert(result | (value[i] << (i - start) * 8));
143        }
144        return result;
145    }
146
147    public TLinkAddress Convert(ushort value)
148    {
149        return Convert(value, 0);
150    }
151
152    public TLinkAddress Convert(ushort value, int start)
153    {
154        var result = NumberToAddressConverter.Convert(0);
155        for (int i = start; i < value.Length; i++)
156        {
157            result = NumberToAddressConverter.Convert(result | (value[i] << (i - start) * 8));
158        }
159        return result;
160    }
161
162    public TLinkAddress Convert(int value)
163    {
164        return Convert(value, 0);
165    }
166
167    public TLinkAddress Convert(int value, int start)
168    {
169        var result = NumberToAddressConverter.Convert(0);
170        for (int i = start; i < value.Length; i++)
171        {
172            result = NumberToAddressConverter.Convert(result | (value[i] << (i - start) * 8));
173        }
174        return result;
175    }
176
177    public TLinkAddress Convert(long value)
178    {
179        return Convert(value, 0);
180    }
181
182    public TLinkAddress Convert(long value, int start)
183    {
184        var result = NumberToAddressConverter.Convert(0);
185        for (int i = start; i < value.Length; i++)
186        {
187            result = NumberToAddressConverter.Convert(result | (value[i] << (i - start) * 8));
188        }
189        return result;
190    }
191
192    public TLinkAddress Convert(ulong value)
193    {
194        return Convert(value, 0);
195    }
196
197    public TLinkAddress Convert(ulong value, int start)
198    {
199        var result = NumberToAddressConverter.Convert(0);
200        for (int i = start; i < value.Length; i++)
201        {
202            result = NumberToAddressConverter.Convert(result | (value[i] << (i - start) * 8));
203        }
204        return result;
205    }
206
207    public TLinkAddress Convert(byte value)
208    {
209        return Convert(value, 0);
210    }
211
212    public TLinkAddress Convert(byte value, int start)
213    {
214        var result = NumberToAddressConverter.Convert(0);
215        for (int i = start; i < value.Length; i++)
216        {
217            result = NumberToAddressConverter.Convert(result | (value[i] << (i - start) * 8));
218        }
219        return result;
220    }
221
222    public TLinkAddress Convert(sbyte value)
223    {
224        return Convert(value, 0);
225    }
226
227    public TLinkAddress Convert(sbyte value, int start)
228    {
229        var result = NumberToAddressConverter.Convert(0);
230        for (int i = start; i < value.Length; i++)
231        {
232            result = NumberToAddressConverter.Convert(result | (value[i] << (i - start) * 8));
233        }
234        return result;
235    }
236
237    public TLinkAddress Convert(short value)
238    {
239        return Convert(value, 0);
240    }
241
242    public TLinkAddress Convert(short value, int start)
243    {
244        var result = NumberToAddressConverter.Convert(0);
245        for (int i = start; i < value.Length; i++)
246        {
247            result = NumberToAddressConverter.Convert(result | (value[i] << (i - start) * 8));
248        }
249        return result;
250    }
251
252    public TLinkAddress Convert(ushort value)
253    {
254        return Convert(value, 0);
255    }
256
257    public TLinkAddress Convert(ushort value, int start)
258    {
259        var result = NumberToAddressConverter.Convert(0);
260        for (int i = start; i < value.Length; i++)
261        {
262            result = NumberToAddressConverter.Convert(result | (value[i] << (i - start) * 8));
263        }
264        return result;
265    }
266
267    public TLinkAddress Convert(int value)
268    {
269        return Convert(value, 0);
270    }
271
272    public TLinkAddress Convert(int value, int start)
273    {
274        var result = NumberToAddressConverter.Convert(0);
275        for (int i = start; i < value.Length; i++)
276        {
277            result = NumberToAddressConverter.Convert(result | (value[i] << (i - start) * 8));
278        }
279        return result;
280    }
281
282    public TLinkAddress Convert(long value)
283    {
284        return Convert(value, 0);
285    }
286
287    public TLinkAddress Convert(long value, int start)
288    {
289        var result = NumberToAddressConverter.Convert(0);
290        for (int i = start; i < value.Length; i++)
291        {
292            result = NumberToAddressConverter.Convert(result | (value[i] << (i - start) * 8));
293        }
294        return result;
295    }
296
297    public TLinkAddress Convert(ulong value)
298    {
299        return Convert(value, 0);
300    }
301
302    public TLinkAddress Convert(ulong value, int start)
303    {
304        var result = NumberToAddressConverter.Convert(0);
305        for (int i = start; i < value.Length; i++)
306        {
307            result = NumberToAddressConverter.Convert(result | (value[i] << (i - start) * 8));
308        }
309        return result;
310    }
311
312    public TLinkAddress Convert(byte value)
313    {
314        return Convert(value, 0);
315    }
316
317    public TLinkAddress Convert(byte value, int start)
318    {
319        var result = NumberToAddressConverter.Convert(0);
320        for (int i = start; i < value.Length; i++)
321        {
322            result = NumberToAddressConverter.Convert(result | (value[i] << (i - start) * 8));
323        }
324        return result;
325    }
326
327    public TLinkAddress Convert(sbyte value)
328    {
329        return Convert(value, 0);
330    }
331
332    public TLinkAddress Convert(sbyte value, int start)
333    {
334        var result = NumberToAddressConverter.Convert(0);
335        for (int i = start; i < value.Length; i++)
336        {
337            result = NumberToAddressConverter.Convert(result | (value[i] << (i - start) * 8));
338        }
339        return result;
340    }
341
342    public TLinkAddress Convert(short value)
343    {
344        return Convert(value, 0);
345    }
346
347    public TLinkAddress Convert(short value, int start)
348    {
349        var result = NumberToAddressConverter.Convert(0);
350        for (int i = start; i < value.Length; i++)
351        {
352            result = NumberToAddressConverter.Convert(result | (value[i] << (i - start) * 8));
353        }
354        return result;
355    }
356
357    public TLinkAddress Convert(ushort value)
358    {
359        return Convert(value, 0);
360    }
361
362    public TLinkAddress Convert(ushort value, int start)
363    {
364        var result = NumberToAddressConverter.Convert(0);
365        for (int i = start; i < value.Length; i++)
366        {
367            result = NumberToAddressConverter.Convert(result | (value[i] << (i - start) * 8));
368        }
369        return result;
370    }
371
372    public TLinkAddress Convert(int value)
373    {
374        return Convert(value, 0);
375    }
376
377    public TLinkAddress Convert(int value, int start)
378    {
379        var result = NumberToAddressConverter.Convert(0);
380        for (int i = start; i < value.Length; i++)
381        {
382            result = NumberToAddressConverter.Convert(result | (value[i] << (i - start) * 8));
383        }
384        return result;
385    }
386
387    public TLinkAddress Convert(long value)
388    {
389        return Convert(value, 0);
390    }
391
392    public TLinkAddress Convert(long value, int start)
393    {
394        var result = NumberToAddressConverter.Convert(0);
395        for (int i = start; i < value.Length; i++)
396        {
397            result = NumberToAddressConverter.Convert(result | (value[i] << (i - start) * 8));
398        }
399        return result;
400    }
401
402    public TLinkAddress Convert(ulong value)
403    {
404        return Convert(value, 0);
405    }
406
407    public TLinkAddress Convert(ulong value, int start)
408    {
409        var result = NumberToAddressConverter.Convert(0);
410        for (int i = start; i < value.Length; i++)
411        {
412            result = NumberToAddressConverter.Convert(result | (value[i] << (i - start) * 8));
413        }
414        return result;
415    }
416
417    public TLinkAddress Convert(byte value)
418    {
419        return Convert(value, 0);
420    }
421
422    public TLinkAddress Convert(byte value, int start)
423    {
424        var result = NumberToAddressConverter.Convert(0);
425        for (int i = start; i < value.Length; i++)
426        {
427            result = NumberToAddressConverter.Convert(result | (value[i] << (i - start) * 8));
428        }
429        return result;
430    }
431
432    public TLinkAddress Convert(sbyte value)
433    {
434        return Convert(value, 0);
435    }
436
437    public TLinkAddress Convert(sbyte value, int start)
438    {
439        var result = NumberToAddressConverter.Convert(0);
440        for (int i = start; i < value.Length; i++)
441        {
442            result = NumberToAddressConverter.Convert(result | (value[i] << (i - start) * 8));
443        }
444        return result;
445    }
446
447    public TLinkAddress Convert(short value)
448    {
449        return Convert(value, 0);
450    }
451
452    public TLinkAddress Convert(short value, int start)
453    {
454        var result = NumberToAddressConverter.Convert(0);
455        for (int i = start; i < value.Length; i++)
456        {
457            result = NumberToAddressConverter.Convert(result | (value[i] << (i - start) * 8));
458        }
459        return result;
460    }
461
462    public TLinkAddress Convert(ushort value)
463    {
464        return Convert(value, 0);
465    }
466
467    public TLinkAddress Convert(ushort value, int start)
468    {
469        var result = NumberToAddressConverter.Convert(0);
470        for (int i = start; i < value.Length; i++)
471        {
472            result = NumberToAddressConverter.Convert(result | (value[i] << (i - start) * 8));
473        }
474        return result;
475    }
476
477    public TLinkAddress Convert(int value)
478    {
479        return Convert(value, 0);
480    }
481
482    public TLinkAddress Convert(int value, int start)
483    {
484        var result = NumberToAddressConverter.Convert(0);
485        for (int i = start; i < value.Length; i++)
486        {
487            result = NumberToAddressConverter.Convert(result | (value[i] << (i - start) * 8));
488        }
489        return result;
490    }
491
492    public TLinkAddress Convert(long value)
493    {
494        return Convert(value, 0);
495    }
496
497    public TLinkAddress Convert(long value, int start)
498    {
499        var result = NumberToAddressConverter.Convert(0);
500        for (int i = start; i < value.Length; i++)
501        {
502            result = NumberToAddressConverter.Convert(result | (value[i] << (i - start) * 8));
503        }
504        return result;
505    }
506
507    public TLinkAddress Convert(ulong value)
508    {
509        return Convert(value, 0);
510    }
511
512    public TLinkAddress Convert(ulong value, int start)
513    {
514        var result = NumberToAddressConverter.Convert(0);
515        for (int i = start; i < value.Length; i++)
516        {
517            result = NumberToAddressConverter.Convert(result | (value[i] << (i - start) * 8));
518        }
519        return result;
520    }
521
522    public TLinkAddress Convert(byte value)
523    {
524        return Convert(value, 0);
525    }
526
527    public TLinkAddress Convert(byte value, int start)
528    {
529        var result = NumberToAddressConverter.Convert(0);
530        for (int i = start; i < value.Length; i++)
531        {
532            result = NumberToAddressConverter.Convert(result | (value[i] << (i - start) * 8));
533        }
534        return result;
535    }
536
537    public TLinkAddress Convert(sbyte value)
538    {
539        return Convert(value, 0);
540    }
541
542    public TLinkAddress Convert(sbyte value, int start)
543    {
544        var result = NumberToAddressConverter.Convert(0);
545        for (int i = start; i < value.Length; i++)
546        {
547            result = NumberToAddressConverter.Convert(result | (value[i] << (i - start) * 8));
548        }
549        return result;
550    }
551
552    public TLinkAddress Convert(short value)
553    {
554        return Convert(value, 0);
555    }
556
557    public TLinkAddress Convert(short value, int start)
558    {
559        var result = NumberToAddressConverter.Convert(0);
560        for (int i = start; i < value.Length; i++)
561        {
562            result = NumberToAddressConverter.Convert(result | (value[i] << (i - start) * 8));
563        }
564        return result;
565    }
566
567    public TLinkAddress Convert(ushort value)
568    {
569        return Convert(value, 0);
570    }
571
572    public TLinkAddress Convert(ushort value, int start)
573    {
574        var result = NumberToAddressConverter.Convert(0);
575        for (int i = start; i < value.Length; i++)
576        {
577            result = NumberToAddressConverter.Convert(result | (value[i] << (i - start) * 8));
578        }
579        return result;
580    }
581
582    public TLinkAddress Convert(int value)
583    {
584        return Convert(value, 0);
585    }
586
587    public TLinkAddress Convert(int value, int start)
588    {
589        var result = NumberToAddressConverter.Convert(0);
590        for (int i = start; i < value.Length; i++)
591        {
592            result = NumberToAddressConverter.Convert(result | (value[i] << (i - start) * 8));
593        }
594        return result;
595    }
596
597    public TLinkAddress Convert(long value)
598    {
599        return Convert(value, 0);
600    }
601
602    public TLinkAddress Convert(long value, int start)
603    {
604        var result = NumberToAddressConverter.Convert(0);
605        for (int i = start; i < value.Length; i++)
606        {
607            result = NumberToAddressConverter.Convert(result | (value[i] << (i - start) * 8));
608        }
609        return result;
610    }
611
612    public TLinkAddress Convert(ulong value)
613    {
614        return Convert(value, 0);
615    }
616
617    public TLinkAddress Convert(ulong value, int start)
618    {
619        var result = NumberToAddressConverter.Convert(0);
620        for (int i = start; i < value.Length; i++)
621        {
622            result = NumberToAddressConverter.Convert(result | (value[i] << (i - start) * 8));
623        }
624        return result;
625    }
626
627    public TLinkAddress Convert(byte value)
628    {
629        return Convert(value, 0);
630    }
631
632    public TLinkAddress Convert(byte value, int start)
633    {
634        var result = NumberToAddressConverter.Convert(0);
635        for (int i = start; i < value.Length; i++)
636        {
637            result = NumberToAddressConverter.Convert(result | (value[i] << (i - start) * 8));
638        }
639        return result;
640    }
641
642    public TLinkAddress Convert(sbyte value)
643    {
644        return Convert(value, 0);
645    }
646
647    public TLinkAddress Convert(sbyte value, int start)
648    {
649        var result = NumberToAddressConverter.Convert(0);
650        for (int i = start; i < value.Length; i++)
651        {
652            result = NumberToAddressConverter.Convert(result | (value[i] << (i - start) * 8));
653        }
654        return result;
655    }
656
657    public TLinkAddress Convert(short value)
658    {
659        return Convert(value, 0);
660    }
661
662    public TLinkAddress Convert(short value, int start)
663    {
664        var result = NumberToAddressConverter.Convert(0);
665        for (int i = start; i < value.Length; i++)
666        {
667            result = NumberToAddressConverter.Convert(result | (value[i] << (i - start) * 8));
668        }
669        return result;
670    }
671
672    public TLinkAddress Convert(ushort value)
673    {
674        return Convert(value, 0);
675    }
676
677    public TLinkAddress Convert(ushort value, int start)
678    {
679        var result = NumberToAddressConverter.Convert(0);
680        for (int i = start; i < value.Length; i++)
681        {
682            result = NumberToAddressConverter.Convert(result | (value[i] << (i - start) * 8));
683        }
684        return result;
685    }
686
687    public TLinkAddress Convert(int value)
688    {
689        return Convert(value, 0);
690    }
691
692    public TLinkAddress Convert(int value, int start)
693    {
694        var result = NumberToAddressConverter.Convert(0);
695        for (int i = start; i < value.Length; i++)
696        {
697            result = NumberToAddressConverter.Convert(result | (value[i] << (i - start) * 8));
698        }
699        return result;
700    }
701
702    public TLinkAddress Convert(long value)
703    {
704        return Convert(value, 0);
705    }
706
707    public TLinkAddress Convert(long value, int start)
708    {
709        var result = NumberToAddressConverter.Convert(0);
710        for (int i = start; i < value.Length; i++)
711        {
712            result = NumberToAddressConverter.Convert(result | (value[i] << (i - start) * 8));
713        }
714        return result;
715    }
716
717    public TLinkAddress Convert(ulong value)
718    {
719        return Convert(value, 0);
720    }
721
722    public TLinkAddress Convert(ulong value, int start)
723    {
724        var result = NumberToAddressConverter.Convert(0);
725        for (int i = start; i < value.Length; i++)
726        {
727            result = NumberToAddressConverter.Convert(result | (value[i] << (i - start) * 8));
728        }
729        return result;
730    }
731
732    public TLinkAddress Convert(byte value)
733    {
734        return Convert(value, 0);
735    }
736
737    public TLinkAddress Convert(byte value, int start)
738    {
739        var result = NumberToAddressConverter.Convert(0);
740        for (int i = start; i < value.Length; i++)
741        {
742            result = NumberToAddressConverter.Convert(result | (value[i] << (i - start) * 8));
743        }
744        return result;
745    }
746
747    public TLinkAddress Convert(sbyte value)
748    {
749        return Convert(value, 0);
750    }
751
752    public TLinkAddress Convert(sbyte value, int start)
753    {
754        var result = NumberToAddressConverter.Convert(0);
755        for (int i = start; i < value.Length; i++)
756        {
757            result = NumberToAddressConverter.Convert(result | (value[i] << (i - start) * 8));
758        }
759        return result;
760    }
761
762    public TLinkAddress Convert(short value)
763    {
764        return Convert(value, 0);
765    }
766
767    public TLinkAddress Convert(short value, int start)
768    {
769        var result = NumberToAddressConverter.Convert(0);
770        for (int i = start; i < value.Length; i++)
771        {
772            result = NumberToAddressConverter.Convert(result | (value[i] << (i - start) * 8));
773        }
774        return result;
775    }
776
777    public TLinkAddress Convert(ushort value)
778    {
779        return Convert(value, 0);
780    }
781
782    public TLinkAddress Convert(ushort value, int start)
783    {
784        var result = NumberToAddressConverter.Convert(0);
785        for (int i = start; i < value.Length; i++)
786        {
787            result = NumberToAddressConverter.Convert(result | (value[i] << (i - start) * 8));
788        }
789        return result;
790    }
791
792    public TLinkAddress Convert(int value)
793    {
794        return Convert(value, 0);
795    }
796
797    public TLinkAddress Convert(int value, int start)
798    {
799        var result = NumberToAddressConverter.Convert(0);
800        for (int i = start; i < value.Length; i++)
801        {
802            result = NumberToAddressConverter.Convert(result | (value[i] << (i - start) * 8));
803        }
804        return result;
805    }
806
807    public TLinkAddress Convert(long value)
808    {
809        return Convert(value, 0);
810    }
811
812    public TLinkAddress Convert(long value, int start)
813    {
814        var result = NumberToAddressConverter.Convert(0);
815        for (int i = start; i < value.Length; i++)
816        {
817            result = NumberToAddressConverter.Convert(result | (value[i] << (i - start) * 8));
818        }
819        return result;
820    }
821
822    public TLinkAddress Convert(ulong value)
823    {
824        return Convert(value, 0);
825    }
826
827    public TLinkAddress Convert(ulong value, int start)
828    {
829        var result = NumberToAddressConverter.Convert(0);
830        for (int i = start; i < value.Length; i++)
831        {
832            result = NumberToAddressConverter.Convert(result | (value[i] << (i - start) * 8));
833        }
834        return result;
835    }
836
837    public TLinkAddress Convert(byte value)
838    {
839        return Convert(value, 0);
840    }
841
842    public TLinkAddress Convert(byte value, int start)
843    {
844        var result = NumberToAddressConverter.Convert(0);
845        for (int i = start; i < value.Length; i++)
846        {
847            result = NumberToAddressConverter.Convert(result | (value[i] << (i - start) * 8));
848        }
849        return result;
850    }
851
852    public TLinkAddress Convert(sbyte value)
853    {
854        return Convert(value, 0);
855    }
856
857    public TLinkAddress Convert(sbyte value, int start)
858    {
859        var result = NumberToAddressConverter.Convert(0);
860        for (int i = start; i < value.Length; i++)
861        {
862            result = NumberToAddressConverter.Convert(result | (value[i] << (i - start) * 8));
863        }
864        return result;
865    }
866
867    public TLinkAddress Convert(short value)
868    {
869        return Convert(value, 0);
870    }
871
872    public TLinkAddress Convert(short value, int start)
873    {
874        var result = NumberToAddressConverter.Convert(0);
875        for (int i = start; i < value.Length; i++)
876        {
877            result = NumberToAddressConverter.Convert(result | (value[i] << (i - start) * 8));
878        }
879        return result;
880    }
881
882    public TLinkAddress Convert(ushort value)
883    {
884        return Convert(value, 0);
885    }
886
887    public TLinkAddress Convert(ushort value, int start)
888    {
889        var result = NumberToAddressConverter.Convert(0);
890        for (int i = start; i < value.Length; i++)
891        {
892            result = NumberToAddressConverter.Convert(result | (value[i] << (i - start) * 8));
893        }
894        return result;
895    }
896
897    public TLinkAddress Convert(int value)
898    {
899        return Convert(value, 0);
900    }
901
902    public TLinkAddress Convert(int value, int start)
903    {
904        var result = NumberToAddressConverter.Convert(0);
905        for (int i = start; i < value.Length; i++)
906        {
907            result = NumberToAddressConverter.Convert(result | (value[i] << (i - start) * 8));
908        }
909        return result;
910    }
911
912    public TLinkAddress Convert(long value)
913    {
914        return Convert(value, 0);
915    }
916
917    public TLinkAddress Convert(long value, int start)
918    {
919        var result = NumberToAddressConverter.Convert(0);
920        for (int i = start; i < value.Length; i++)
921        {
922            result = NumberToAddressConverter.Convert(result | (value[i] << (i - start) * 8));
923        }
924        return result;
925    }
926
927    public TLinkAddress Convert(ulong value)
928    {
929        return Convert(value, 0);
930    }
931
932    public TLinkAddress Convert(ulong value, int start)
933    {
934        var result = NumberToAddressConverter.Convert(0);
935        for (int i = start; i < value.Length; i++)
936        {
937            result = NumberToAddressConverter.Convert(result | (value[i] << (i - start) * 8));
938        }
939        return result;
940    }
941
942    public TLinkAddress Convert(byte value)
943    {
944        return Convert(value, 0);
945    }
946
947    public TLinkAddress Convert(byte value, int start)
948    {
949        var result = NumberToAddressConverter.Convert(0);
950        for (int i = start; i < value.Length; i++)
951        {
952            result = NumberToAddressConverter.Convert(result | (value[i] << (i - start) * 8));
953        }
954        return result;
955    }
956
957    public TLinkAddress Convert(sbyte value)
958    {
959        return Convert(value, 0);
960    }
961
962    public TLinkAddress Convert(sbyte value, int start)
963    {
964        var result = NumberToAddressConverter.Convert(0);
965        for (int i = start; i < value.Length; i++)
966        {
967            result = NumberToAddressConverter.Convert(result | (value[i] << (i - start) * 8));
968        }
969        return result;
970    }
971
972    public TLinkAddress Convert(short value)
973    {
974        return Convert(value, 0);
975    }
976
977    public TLinkAddress Convert(short value, int start)
978    {
979        var result = NumberToAddressConverter.Convert(0);
980        for (int i = start; i < value.Length; i++)
981        {
982            result = NumberToAddressConverter.Convert(result | (value[i] << (i - start) * 8));
983        }
984        return result;
985    }
986
987    public TLinkAddress Convert(ushort value)
988    {
989        return Convert(value, 0);
990    }
991
992    public TLinkAddress Convert(ushort value, int start)
993    {
994        var result = NumberToAddressConverter.Convert(0);
995        for (int i = start; i < value.Length; i++)
996        {
997            result = NumberToAddressConverter.Convert(result | (value[i] << (i - start) * 8));
998        }
999        return result;
1000    }
1001
1002    public TLinkAddress Convert(int value)
1003    {
1004        return Convert(value, 0);
1005    }
1006
1007    public TLinkAddress Convert(int value, int start)
1008    {
1009        var result = NumberToAddressConverter.Convert(0);
1010        for (int i = start; i < value.Length; i++)
1011        {
1012            result = NumberToAddressConverter.Convert(result | (value[i] << (i - start) * 8));
1013        }
1014        return result;
1015    }
1016
1017    public TLinkAddress Convert(long value)
1018    {
1019        return Convert(value, 0);
1020    }
1021
1022    public TLinkAddress Convert(long value, int start)
1023    {
1024        var result = NumberToAddressConverter.Convert(0);
1025        for (int i = start; i < value.Length; i++)
1026        {
1027            result = NumberToAddressConverter.Convert(result | (value[i] << (i - start) * 8));
1028        }
1029        return result;
1030    }
1031
1032    public TLinkAddress Convert(ulong value)
1033    {
1034        return Convert(value, 0);
1035    }
1036
1037    public TLinkAddress Convert(ulong value, int start)
1038    {
1039        var result = NumberToAddressConverter.Convert(0);
1040        for (int i = start; i < value.Length; i++)
1041        {
1042            result = NumberToAddressConverter.Convert(result | (value[i] << (i - start) * 8));
1043        }
1044        return result;
1045    }
1046
1047    public TLinkAddress Convert(byte value)
1048    {
1049        return Convert(value, 0);
1050    }
1051
1052    public TLinkAddress Convert(byte value, int start)
1053    {
1054        var result = NumberToAddressConverter.Convert(0);
1055        for (int i = start; i < value.Length; i++)
1056        {
1057            result = NumberToAddressConverter.Convert(result | (value[i] << (i - start) * 8));
1058        }
1059        return result;
1060    }
1061
1062    public TLinkAddress Convert(sbyte value)
1063    {
1064        return Convert(value, 0);
1065    }
1066
1067    public TLinkAddress Convert(sbyte value, int start)
1068    {
1069        var result = NumberToAddressConverter.Convert(0);
1070        for (int i = start; i < value.Length; i++)
1071        {
1072            result = NumberToAddressConverter.Convert(result | (value[i] << (i - start) * 8));
1073        }
1074        return result;
1075    }
1076
1077    public TLinkAddress Convert(short value)
1078    {
1079        return Convert(value, 0);
1080    }
1081
1082    public TLinkAddress Convert(short value, int
```

```

46 public RawSequenceToByteListConverter(ILinks<TLinkAddress> links, IConverter<TLinkAddress>
    ↳ numberToAddressConverter, IConverter<IList<TLinkAddress>, TLinkAddress>
    ↳ listToSequenceConverter, StringToUnicodeSequenceConverter<TLinkAddress>
    ↳ stringToUnicodeSequenceConverter, UnicodeSequenceToStringConverter<TLinkAddress>
    ↳ unicodeSequenceToStringConverter) : base(links)
47 {
48     NumberToAddressConverter = numberToAddressConverter;
49     ListToSequenceConverter = listToSequenceConverter;
50     BalancedVariantConverter = new BalancedVariantConverter<TLinkAddress>(links);
51     StringToUnicodeSequenceConverter = stringToUnicodeSequenceConverter;
52     UnicodeSequenceToStringConverter = unicodeSequenceToStringConverter;
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 =
        ↳ StringToUnicodeSequenceConverter.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,
        ↳ StringToUnicodeSequenceConverter.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,
        ↳ StringToUnicodeSequenceConverter.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();

```

```

115 }
116
117 public IList<byte> Convert(TLinkAddress source)
118 {
119     return new RightSequenceWalker<TLinkAddress>(Links, new
        ↳ DefaultStack<TLinkAddress>()).Walk(source).Select(address =>
        ↳ NumberToAddressConverter.Convert(address)).Select(address =>
        ↳ TLinkAddressToByteConverter.Convert(address)).ToList();
120 // Console.WriteLine("RawSequenceToByteListConverter.Convert");
121 // if (IsEmptyArray(source))
122 // {
123 //     return new List<byte>();
124 // }
125 // EnsureIsByteArray(source);
126 // var byteArrayLength = GetByteArrayLength(source);
127 // List<byte> byteList = new(byteArrayLength);
128 // var rawNumbersEnumerator = GetRawNumbersEnumerator(source);
129 // var i = 0;
130 // while (rawNumbersEnumerator.MoveNext())
131 // {
132 //     Console.WriteLine("Raw number: ");
133 //     var rawNumber = NumberToAddressConverter.Convert(rawNumbersEnumerator.Current);
134 //     Console.WriteLine(TestExtensions.PrettifyBinary<TLinkAddress>(System.Convert.ToSt
        ↳ ring((ushort)(object)rawNumber,
        ↳ 2)));
135 //     var nonSavedBitsCount = i % 8;
136 //     var isLastRawNumber = (byteArrayLength % BytesInRawNumberCount == 0) &&
        ↳ (byteList.Count == byteArrayLength);
137 //     if(isLastRawNumber)
138 //     {
139 //         rawNumber = Bit.ShiftLeft(rawNumber, 8 - nonSavedBitsCount);
140 //         var @byte = TLinkAddressToByteConverter.Convert(rawNumber);
141 //         byteList[byteList.Count - 1] = Bit.Or(byteList.Last(), @byte);
142 //         break;
143 //     }
144 //     if (nonSavedBitsCount != 0)
145 //     {
146 //         var rawNumberWithOnlyNonSavedBits = rawNumber;
147 //         rawNumberWithOnlyNonSavedBits = Bit.ShiftLeft(rawNumberWithOnlyNonSavedBits,
        ↳ BitsSize - nonSavedBitsCount);
148 //         rawNumberWithOnlyNonSavedBits = Bit.ShiftRight(rawNumberWithOnlyNonSavedBits,
        ↳ BitsSize - nonSavedBitsCount);
149 //         rawNumberWithOnlyNonSavedBits = Bit.ShiftLeft(rawNumberWithOnlyNonSavedBits,
        ↳ 8 - nonSavedBitsCount);
150 //         var @byte =
        ↳ TLinkAddressToByteConverter.Convert(rawNumberWithOnlyNonSavedBits);
151 //         byteList[byteList.Count - 1] = Bit.Or(byteList.Last(), @byte);
152 //         rawNumber = Bit.ShiftRight(rawNumber, nonSavedBitsCount);
153 //     }
154 //     var bytesInRawNumber = nonSavedBitsCount == 7 ? BytesInRawNumberCount - 1 :
        ↳ BytesInRawNumberCount;
155 //     for (int j = 0; (j < bytesInRawNumber) && (byteList.Count < byteArrayLength); j++)
156 //     {
157 //         var @byte = TLinkAddressToByteConverter.Convert(rawNumber);
158 //         byteList.Add(@byte);
159 //         rawNumber = Bit.ShiftRight(rawNumber, 8);
160 //     }
161 //     i++;
162 // }
163 // return byteList;
164 }
165
166 private static byte GetByteWithNotSavedBitsAtEnd(TLinkAddress currentRawNumber, int
        ↳ nonSavedBits)
167 {
168     var @byte = TLinkAddressToByteConverter.Convert(currentRawNumber);
169     @byte = Bit.ShiftLeft(@byte, 8 - nonSavedBits);
170     return @byte;
171 }
172 }

```

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

```

1 using System.Numerics;
2 using Platform.Converters;
3 using Platform.Data.Doublets.Decorators;
4 using System.Globalization;
5 using Platform.Data.Doublets.Numbers.Raw;
6

```



```

7 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9 namespace Platform.Data.Doublets.Numbers.Rational
10 {
11     /// <summary>
12     /// <para>
13     /// Represents the decimal to rational converter.
14     /// </para>
15     /// <para></para>
16     /// </summary>
17     /// <seealso cref="LinksDecoratorBase{TLinkAddress}"/>
18     /// <seealso cref="IConverter{decimal, TLinkAddress}"/>
19     public class DecimalToRationalConverter<TLinkAddress> : LinksDecoratorBase<TLinkAddress>,
20         ↳ IConverter<decimal, TLinkAddress>
21         where TLinkAddress: struct
22     {
23         /// <summary>
24         /// <para>
25         /// The big integer to raw number sequence converter.
26         /// </para>
27         /// <para></para>
28         /// </summary>
29         public readonly BigIntegerToRawNumberSequenceConverter<TLinkAddress>
30             ↳ BigIntegerToRawNumberSequenceConverter;
31
32         /// <summary>
33         /// <para>
34         /// Initializes a new <see cref="DecimalToRationalConverter"/> instance.
35         /// </para>
36         /// <para></para>
37         /// </summary>
38         /// <param name="links">
39         /// <para>A links.</para>
40         /// <para></para>
41         /// </param>
42         /// <param name="bigIntegerToRawNumberSequenceConverter">
43         /// <para>A big integer to raw number sequence converter.</para>
44         /// <para></para>
45         /// </param>
46         public DecimalToRationalConverter(ILinks<TLinkAddress> links,
47             ↳ BigIntegerToRawNumberSequenceConverter<TLinkAddress>
48             ↳ bigIntegerToRawNumberSequenceConverter) : base(links)
49         {
50             BigIntegerToRawNumberSequenceConverter = bigIntegerToRawNumberSequenceConverter;
51         }
52
53         /// <summary>
54         /// <para>
55         /// Converts the decimal.
56         /// </para>
57         /// <para></para>
58         /// </summary>
59         /// <param name="@decimal">
60         /// <para>The decimal.</para>
61         /// <para></para>
62         /// </param>
63         /// <returns>
64         /// <para>The link</para>
65         /// <para></para>
66         /// </returns>
67         public TLinkAddress Convert(decimal @decimal)
68         {
69             var decimalAsString = @decimal.ToString(CultureInfo.InvariantCulture);
70             var dotPosition = decimalAsString.IndexOf('.');
71             var decimalWithoutDots = decimalAsString;
72             int digitsAfterDot = 0;
73             if (dotPosition != -1)
74             {
75                 decimalWithoutDots = decimalWithoutDots.Remove(dotPosition, 1);
76                 digitsAfterDot = decimalAsString.Length - 1 - dotPosition;
77             }
78             BigInteger denominator = new(System.Math.Pow(10, digitsAfterDot));
79             BigInteger numerator = BigInteger.Parse(decimalWithoutDots);
80             BigInteger greatestCommonDivisor;
81             do
82             {
83                 greatestCommonDivisor = BigInteger.GreatestCommonDivisor(numerator, denominator);
84                 numerator /= greatestCommonDivisor;
85                 denominator /= greatestCommonDivisor;
86             } while (greatestCommonDivisor != 1);
87         }
88     }
89 }

```

```

82     }
83     while (greatestCommonDivisor > 1);
84     var numeratorLink = BigIntegerToRawNumberSequenceConverter.Convert(numerator);
85     var denominatorLink = BigIntegerToRawNumberSequenceConverter.Convert(denominator);
86     return _links.GetOrCreate(numeratorLink, denominatorLink);
87 }
88 }
89 }

```

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

```

1  using Platform.Converters;
2  using Platform.Data.Doublets.Decorators;
3  using Platform.Data.Doublets.Numbers.Raw;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Numbers.Rational
8  {
9      /// <summary>
10     /// <para>
11     /// Represents the rational to decimal converter.
12     /// </para>
13     /// <para></para>
14     /// </summary>
15     /// <seealso cref="LinksDecoratorBase{TLinkAddress}"/>
16     /// <seealso cref="IConverter{TLinkAddress, decimal}"/>
17     public class RationalToDecimalConverter<TLinkAddress> : LinksDecoratorBase<TLinkAddress>,
18     ↪ IConverter<TLinkAddress, decimal>
19     where TLinkAddress: struct
20     {
21         /// <summary>
22         /// <para>
23         /// The raw number sequence to big integer converter.
24         /// </para>
25         /// <para></para>
26         /// </summary>
27         public readonly RawNumberSequenceToBigIntegerConverter<TLinkAddress>
28         ↪ RawNumberSequenceToBigIntegerConverter;
29
30         /// <summary>
31         /// <para>
32         /// Initializes a new <see cref="RationalToDecimalConverter"/> instance.
33         /// </para>
34         /// <para></para>
35         /// </summary>
36         /// <param name="links">
37         /// <para>A links.</para>
38         /// <para></para>
39         /// </param>
40         /// <param name="rawNumberSequenceToBigIntegerConverter">
41         /// <para>A raw number sequence to big integer converter.</para>
42         /// <para></para>
43         /// </param>
44         public RationalToDecimalConverter(ILinks<TLinkAddress> links,
45         ↪ RawNumberSequenceToBigIntegerConverter<TLinkAddress>
46         ↪ rawNumberSequenceToBigIntegerConverter) : base(links)
47         {
48             RawNumberSequenceToBigIntegerConverter = rawNumberSequenceToBigIntegerConverter;
49         }
50
51         /// <summary>
52         /// <para>
53         /// Converts the rational number.
54         /// </para>
55         /// <para></para>
56         /// </summary>
57         /// <param name="rationalNumber">
58         /// <para>The rational number.</para>
59         /// <para></para>
60         /// </param>
61         /// <returns>
62         /// <para>The decimal</para>
63         /// <para></para>
64         /// </returns>
65         public decimal Convert(TLinkAddress rationalNumber)
66         {
67             var numerator = (decimal)RawNumberSequenceToBigIntegerConverter.Convert(_links.GetSo
68             ↪ urce(rationalNumber));

```

```

64         var denominator = (decimal)RawNumberSequenceToBigIntegerConverter.Convert(_links.Get_|
        ↪ Target(rationalNumber));
65         return numerator / denominator;
66     }
67 }
68 }

```

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

```

1  using System.Collections.Generic;
2  using System.Numerics;
3  using System.Runtime.InteropServices;
4  using Platform.Converters;
5  using Platform.Data.Doublets.Decorators;
6  using Platform.Numbers;
7  using Platform.Reflection;
8  using Platform.Unsafe;
9
10 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
11
12 namespace Platform.Data.Doublets.Numbers.Raw
13 {
14     /// <summary>
15     /// <para>
16     /// Represents the big integer to raw number sequence converter.
17     /// </para>
18     /// <para></para>
19     /// </summary>
20     /// <seealso cref="LinksDecoratorBase{TLinkAddress}"/>
21     /// <seealso cref="IConverter{BigInteger, TLinkAddress}"/>
22     public class BigIntegerToRawNumberSequenceConverter<TLinkAddress> :
23     ↪ LinksDecoratorBase<TLinkAddress>, IConverter<BigInteger, TLinkAddress>
24     where TLinkAddress : struct
25     {
26         /// <summary>
27         /// <para>
28         /// The max value.
29         /// </para>
30         /// <para></para>
31         /// </summary>
32         public static readonly TLinkAddress MaximumValue = NumericType<TLinkAddress>.MaxValue;
33         /// <summary>
34         /// <para>
35         /// The maximum value.
36         /// </para>
37         /// <para></para>
38         /// </summary>
39         public static readonly TLinkAddress BitMask = Bit.ShiftRight(MaximumValue, 1);
40         /// <summary>
41         /// <para>
42         /// The address to number converter.
43         /// </para>
44         /// <para></para>
45         /// </summary>
46         public readonly IConverter<TLinkAddress> AddressToNumberConverter;
47         /// <summary>
48         /// <para>
49         /// The list to sequence converter.
50         /// </para>
51         /// <para></para>
52         /// </summary>
53         public readonly IConverter<IList<TLinkAddress>, TLinkAddress> ListToSequenceConverter;
54         /// <summary>
55         /// <para>
56         /// The negative number marker.
57         /// </para>
58         /// <para></para>
59         /// </summary>
60         public readonly TLinkAddress NegativeNumberMarker;
61
62         /// <summary>
63         /// <para>
64         /// Initializes a new <see cref="BigIntegerToRawNumberSequenceConverter"/> instance.
65         /// </para>
66         /// <para></para>
67         /// </summary>
68         /// <param name="links">
69         /// <para>A links.</para>
70         /// <para></para>
71         /// </param>

```

```

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

```

```

4 using Platform.Numbers;
5 using Platform.Reflection;
6 using Platform.Data.Doublets.Decorators;
7
8 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Data.Doublets.Numbers.Raw
11 {
12     /// <summary>
13     /// <para>
14     /// Represents the number to long raw number sequence converter.
15     /// </para>
16     /// <para></para>
17     /// </summary>
18     /// <seealso cref="LinksDecoratorBase{TTarget}"/>
19     /// <seealso cref="IConverter{TSource, TTarget}"/>
20     public class NumberToLongRawNumberSequenceConverter<TSource, TTarget> :
21         ↳ LinksDecoratorBase<TTarget>, IConverter<TSource, TTarget>
22     {
23         private static readonly Comparer<TSource> _comparer = Comparer<TSource>.Default;
24         private static readonly TSource _maximumValue = NumericType<TSource>.MaxValue;
25         private static readonly int _bitsPerRawNumber = NumericType<TTarget>.BitsSize - 1;
26         private static readonly TSource _bitMask = Bit.ShiftRight(_maximumValue,
27             ↳ NumericType<TTarget>.BitsSize + 1);
28         private static readonly TSource _maximumConvertibleAddress = CheckedConverter<TTarget,
29             ↳ TSource>.Default.Convert(Arithmetic.Decrement(Hybrid<TTarget>.ExternalZero));
30         private static readonly uncheckedConverter<TSource, TTarget> _sourceToTargetConverter =
31             ↳ uncheckedConverter<TSource, TTarget>.Default;
32         private readonly IConverter<TTarget> _addressToNumberConverter;
33
34         /// <summary>
35         /// <para>
36         /// Initializes a new <see cref="NumberToLongRawNumberSequenceConverter"/> instance.
37         /// </para>
38         /// <para></para>
39         /// </summary>
40         /// <param name="links">
41         /// <para>A links.</para>
42         /// <para></para>
43         /// </param>
44         /// <param name="addressToNumberConverter">
45         /// <para>A address to number converter.</para>
46         /// <para></para>
47         /// </param>
48         [MethodImpl(MethodImplOptions.AggressiveInlining)]
49         public NumberToLongRawNumberSequenceConverter(ILinks<TTarget> links, IConverter<TTarget>
50             ↳ addressToNumberConverter) : base(links) => _addressToNumberConverter =
51             ↳ addressToNumberConverter;
52
53         /// <summary>
54         /// <para>
55         /// Converts the source.
56         /// </para>
57         /// <para></para>
58         /// </summary>
59         /// <param name="source">
60         /// <para>The source.</para>
61         /// <para></para>
62         /// </param>
63         /// <returns>
64         /// <para>The target</para>
65         /// <para></para>
66         /// </returns>
67         [MethodImpl(MethodImplOptions.AggressiveInlining)]
68         public TTarget Convert(TSource source)
69         {
70             if (_comparer.Compare(source, _maximumConvertibleAddress) > 0)
71             {
72                 var numberPart = Bit.And(source, _bitMask);
73                 var convertedNumber = _addressToNumberConverter.Convert(_sourceToTargetConverter
74                     ↳ .Convert(numberPart));
75                 return Links.GetOrCreate(convertedNumber, Convert(Bit.ShiftRight(source,
76                     ↳ _bitsPerRawNumber)));
77             }
78             else
79             {
80                 return
81                     ↳ _addressToNumberConverter.Convert(_sourceToTargetConverter.Convert(source));
82             }
83         }
84     }
85 }

```

```

73     }
74 }
75 }
76 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Numerics;
4  using Platform.Collections.Stacks;
5  using Platform.Converters;
6  using Platform.Data.Doublets.Decorators;
7  using Platform.Data.Doublets.Sequences.Walkers;
8  using Platform.Reflection;
9  using Platform.Unsafe;
10
11 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
12
13 namespace Platform.Data.Doublets.Numbers.Raw
14 {
15     /// <summary>
16     /// <para>
17     /// Represents the raw number sequence to big integer converter.
18     /// </para>
19     /// <para></para>
20     /// </summary>
21     /// <seealso cref="LinksDecoratorBase{TLinkAddress}"/>
22     /// <seealso cref="IConverter{TLinkAddress, BigInteger}"/>
23     public class RawNumberSequenceToBigIntegerConverter<TLinkAddress> :
24         ↳ LinksDecoratorBase<TLinkAddress>, IConverter<TLinkAddress, BigInteger>
25     {
26         /// <summary>
27         /// <para>
28         /// The default.
29         /// </para>
30         /// <para></para>
31         /// </summary>
32         public readonly EqualityComparer<TLinkAddress> EqualityComparer =
33             ↳ EqualityComparer<TLinkAddress>.Default;
34         /// <summary>
35         /// <para>
36         /// The number to address converter.
37         /// </para>
38         /// <para></para>
39         /// </summary>
40         public readonly IConverter<TLinkAddress, TLinkAddress> NumberToAddressConverter;
41         /// <summary>
42         /// <para>
43         /// The left sequence walker.
44         /// </para>
45         /// <para></para>
46         /// </summary>
47         public readonly LeftSequenceWalker<TLinkAddress> LeftSequenceWalker;
48         /// <summary>
49         /// <para>
50         /// The negative number marker.
51         /// </para>
52         /// <para></para>
53         /// </summary>
54         public readonly TLinkAddress NegativeNumberMarker;
55
56         /// <summary>
57         /// <para>
58         /// Initializes a new <see cref="RawNumberSequenceToBigIntegerConverter"/> instance.
59         /// </para>
60         /// <para></para>
61         /// </summary>
62         /// <param name="links">
63         /// <para>A links.</para>
64         /// <para></para>
65         /// </param>
66         /// <param name="numberToAddressConverter">
67         /// <para>A number to address converter.</para>
68         /// <para></para>
69         /// </param>
70         /// <param name="negativeNumberMarker">
71         /// <para>A negative number marker.</para>
72         /// <para></para>

```

```

22    /// </param>
23    public RawNumberSequenceToBigIntegerConverter(ILinks<TLinkAddress> links,
    ↪ IConverter<TLinkAddress, TLinkAddress> numberToAddressConverter, TLinkAddress
    ↪ negativeNumberMarker) : base(links)
24    {
25        NumberToAddressConverter = numberToAddressConverter;
26        LeftSequenceWalker = new(links, new DefaultStack<TLinkAddress>());
27        NegativeNumberMarker = negativeNumberMarker;
28    }
29
30    /// <summary>
31    /// <para>
32    /// Converts the big integer.
33    /// </para>
34    /// <para></para>
35    /// </summary>
36    /// <param name="bigInteger">
37    /// <para>The big integer.</para>
38    /// <para></para>
39    /// </param>
40    /// <exception cref="Exception">
41    /// <para>Raw number sequence cannot be empty.</para>
42    /// <para></para>
43    /// </exception>
44    /// <returns>
45    /// <para>The big integer</para>
46    /// <para></para>
47    /// </returns>
48    public BigInteger Convert(TLinkAddress bigInteger)
49    {
50        var sign = 1;
51        var bigIntegerSequence = bigInteger;
52        if (EqualityComparer.Equals(_links.GetSource(bigIntegerSequence),
    ↪ NegativeNumberMarker))
53        {
54            sign = -1;
55            bigIntegerSequence = _links.GetTarget(bigInteger);
56        }
57        using var enumerator = LeftSequenceWalker.Walk(bigIntegerSequence).GetEnumerator();
58        if (!enumerator.MoveNext())
59        {
60            throw new Exception("Raw number sequence cannot be empty.");
61        }
62        var nextPart = NumberToAddressConverter.Convert(enumerator.Current);
63        BigInteger currentBigInt = new(nextPart.ToBytes());
64        while (enumerator.MoveNext())
65        {
66            currentBigInt <= (NumericType<TLinkAddress>.BitsSize - 1);
67            nextPart = NumberToAddressConverter.Convert(enumerator.Current);
68            currentBigInt |= new BigInteger(nextPart.ToBytes());
69        }
70        return sign == -1 ? BigInteger.Negate(currentBigInt) : currentBigInt;
71    }
72 }
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
101
102
103
104
105
106
107
108
109
110
111
112
113
114
115
116
117
118
119
120
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     {

```



```

21 private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
    ↳ EqualityComparer<TLinkAddress>.Default;
22 private static readonly TLinkAddress _zero = default;
23 private static readonly TLinkAddress _one = Arithmetic.Increment(_zero);
24 private readonly IConverter<int, TLinkAddress> _powerOf2ToUnaryNumberConverter;
25
26 /// <summary>
27 /// <para>
28 /// Initializes a new <see cref="AddressToUnaryNumberConverter"/> instance.
29 /// </para>
30 /// <para></para>
31 /// </summary>
32 /// <param name="links">
33 /// <para>A links.</para>
34 /// <para></para>
35 /// </param>
36 /// <param name="powerOf2ToUnaryNumberConverter">
37 /// <para>A power of 2 to unary number converter.</para>
38 /// <para></para>
39 /// </param>
40 [MethodImpl(MethodImplOptions.AggressiveInlining)]
41 public AddressToUnaryNumberConverter(ILinks<TLinkAddress> links, IConverter<int,
    ↳ TLinkAddress> powerOf2ToUnaryNumberConverter) : base(links) =>
    ↳ _powerOf2ToUnaryNumberConverter = powerOf2ToUnaryNumberConverter;
42
43 /// <summary>
44 /// <para>
45 /// Converts the number.
46 /// </para>
47 /// <para></para>
48 /// </summary>
49 /// <param name="number">
50 /// <para>The number.</para>
51 /// <para></para>
52 /// </param>
53 /// <returns>
54 /// <para>The target.</para>
55 /// <para></para>
56 /// </returns>
57 [MethodImpl(MethodImplOptions.AggressiveInlining)]
58 public TLinkAddress Convert(TLinkAddress number)
59 {
60     var links = _links;
61     var nullConstant = links.Constants.Null;
62     var target = nullConstant;
63     for (var i = 0; !_equalityComparer.Equals(number, _zero) && i <
    ↳ NumericType<TLinkAddress>.BitsSize; i++)
64     {
65         if (_equalityComparer.Equals(Bit.And(number, _one), _one))
66         {
67             target = _equalityComparer.Equals(target, nullConstant)
68                 ? _powerOf2ToUnaryNumberConverter.Convert(i)
69                 : links.GetOrCreate(_powerOf2ToUnaryNumberConverter.Convert(i), target);
70         }
71         number = Bit.ShiftRight(number, 1);
72     }
73     return target;
74 }
75 }
76 }

```

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

```

1 using System;
2 using System.Collections.Generic;
3 using Platform.Interfaces;
4 using Platform.Converters;
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 link to its frequency number converter.
14     /// </para>
15     /// <para></para>
16     /// </summary>
17     /// <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;

```

```

3 using Platform.Ranges;
4 using Platform.Converters;
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 power of to unary number converter.
14     /// </para>
15     /// <para></para>
16     /// </summary>
17     /// <seealso cref="LinksOperatorBase{TLinkAddress}"/>
18     /// <seealso cref="IConverter<int, TLinkAddress>"/>
19     public class PowerOf2ToUnaryNumberConverter<TLinkAddress> : LinksOperatorBase<TLinkAddress>,
20     ↪ IConverter<int, TLinkAddress>
21     {
22         private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
23         ↪ EqualityComparer<TLinkAddress>.Default;
24         private readonly TLinkAddress[] _unaryNumberPowersOf2;
25
26         /// <summary>
27         /// <para>
28         /// Initializes a new <see cref="PowerOf2ToUnaryNumberConverter"/> instance.
29         /// </para>
30         /// <para></para>
31         /// </summary>
32         /// <param name="links">
33         /// <para>A links.</para>
34         /// <para></para>
35         /// </param>
36         /// <param name="one">
37         /// <para>A one.</para>
38         /// <para></para>
39         /// </param>
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         public PowerOf2ToUnaryNumberConverter(ILinks<TLinkAddress> links, TLinkAddress one) :
42         ↪ base(links)
43         {
44             _unaryNumberPowersOf2 = new TLinkAddress[64];
45             _unaryNumberPowersOf2[0] = one;
46         }
47
48         /// <summary>
49         /// <para>
50         /// Converts the power.
51         /// </para>
52         /// <para></para>
53         /// </summary>
54         /// <param name="power">
55         /// <para>The power.</para>
56         /// <para></para>
57         /// </param>
58         /// <returns>
59         /// <para>The power of.</para>
60         /// <para></para>
61         /// </returns>
62         [MethodImpl(MethodImplOptions.AggressiveInlining)]
63         public TLinkAddress Convert(int power)
64         {
65             Ensure.Always.ArgumentInRange(power, new Range<int>(0, _unaryNumberPowersOf2.Length
66             ↪ - 1), nameof(power));
67             if (!_equalityComparer.Equals(_unaryNumberPowersOf2[power], default))
68             {
69                 return _unaryNumberPowersOf2[power];
70             }
71             var previousPowerOf2 = Convert(power - 1);
72             var powerOf2 = _links.GetOrCreate(previousPowerOf2, previousPowerOf2);
73             _unaryNumberPowersOf2[power] = powerOf2;
74             return powerOf2;
75         }
76     }
77 }

```

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

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;

```

```

3 using Platform.Converters;
4 using Platform.Numbers;
5
6 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8 namespace Platform.Data.Doublets.Numbers.Unary
9 {
10     /// <summary>
11     /// <para>
12     /// Represents the unary number to address add operation converter.
13     /// </para>
14     /// <para></para>
15     /// </summary>
16     /// <seealso cref="LinksOperatorBase{TLinkAddress}"/>
17     /// <seealso cref="IConverter{TLinkAddress}"/>
18     public class UnaryNumberToAddressAddOperationConverter<TLinkAddress> :
19         ↳ LinksOperatorBase<TLinkAddress>, IConverter<TLinkAddress>
20     {
21         private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
22             ↳ EqualityComparer<TLinkAddress>.Default;
23         private static readonly UncheckedConverter<TLinkAddress, ulong>
24             ↳ _addressToUInt64Converter = UncheckedConverter<TLinkAddress, ulong>.Default;
25         private static readonly UncheckedConverter<ulong, TLinkAddress>
26             ↳ _uInt64ToAddressConverter = UncheckedConverter<ulong, TLinkAddress>.Default;
27         private static readonly TLinkAddress _zero = default;
28         private static readonly TLinkAddress _one = Arithmetic.Increment(_zero);
29         private readonly Dictionary<TLinkAddress, TLinkAddress> _unaryToUInt64;
30         private readonly TLinkAddress _unaryOne;
31
32         /// <summary>
33         /// <para>
34         /// Initializes a new <see cref="UnaryNumberToAddressAddOperationConverter"/> instance.
35         /// </para>
36         /// <para></para>
37         /// </summary>
38         /// <param name="links">
39         /// <para>A links.</para>
40         /// <para></para>
41         /// </param>
42         /// <param name="unaryOne">
43         /// <para>A unary one.</para>
44         /// <para></para>
45         /// </param>
46         [MethodImpl(MethodImplOptions.AggressiveInlining)]
47         public UnaryNumberToAddressAddOperationConverter(ILinks<TLinkAddress> links,
48             ↳ TLinkAddress unaryOne)
49             : base(links)
50         {
51             _unaryOne = unaryOne;
52             _unaryToUInt64 = CreateUnaryToUInt64Dictionary(links, unaryOne);
53         }
54
55         /// <summary>
56         /// <para>
57         /// Converts the unary number.
58         /// </para>
59         /// <para></para>
60         /// </summary>
61         /// <param name="unaryNumber">
62         /// <para>The unary number.</para>
63         /// <para></para>
64         /// </param>
65         /// <returns>
66         /// <para>The link</para>
67         /// <para></para>
68         /// </returns>
69         [MethodImpl(MethodImplOptions.AggressiveInlining)]
70         public TLinkAddress Convert(TLinkAddress unaryNumber)
71         {
72             if (_equalityComparer.Equals(unaryNumber, default))
73             {
74                 return default;
75             }
76             if (_equalityComparer.Equals(unaryNumber, _unaryOne))
77             {
78                 return _one;
79             }
80             var links = _links;
81             var source = links.GetSource(unaryNumber);

```

```

77     var target = links.GetTarget(unaryNumber);
78     if (_equalityComparer.Equals(source, target))
79     {
80         return _unaryToUInt64[unaryNumber];
81     }
82     else
83     {
84         var result = _unaryToUInt64[source];
85         TLinkAddress lastValue;
86         while (!_unaryToUInt64.TryGetValue(target, out lastValue))
87         {
88             source = links.GetSource(target);
89             result = Arithmetic<TLinkAddress>.Add(result, _unaryToUInt64[source]);
90             target = links.GetTarget(target);
91         }
92         result = Arithmetic<TLinkAddress>.Add(result, lastValue);
93         return result;
94     }
95 }
96 [MethodImpl(MethodImplOptions.AggressiveInlining)]
97 private static Dictionary<TLinkAddress, TLinkAddress>
98     ↪ CreateUnaryToUInt64Dictionary(ILinks<TLinkAddress> links, TLinkAddress unaryOne)
99 {
100     var unaryToUInt64 = new Dictionary<TLinkAddress, TLinkAddress>
101     {
102         { unaryOne, _one }
103     };
104     var unary = unaryOne;
105     var number = _one;
106     for (var i = 1; i < 64; i++)
107     {
108         unary = links.GetOrCreate(unary, unary);
109         number = Double(number);
110         unaryToUInt64.Add(unary, number);
111     }
112     return unaryToUInt64;
113 }
114 [MethodImpl(MethodImplOptions.AggressiveInlining)]
115 private static TLinkAddress Double(TLinkAddress number) =>
116     ↪ _uInt64ToAddressConverter.Convert(_addressToUInt64Converter.Convert(number) * 2UL);
117 }

```

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

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3 using Platform.Reflection;
4 using Platform.Converters;
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.Numbers.Unary
10 {
11     /// <summary>
12     /// <para>
13     /// Represents the unary number to address or operation converter.
14     /// </para>
15     /// <para></para>
16     /// </summary>
17     /// <seealso cref="LinksOperatorBase{TLinkAddress}"/>
18     /// <seealso cref="IConverter{TLinkAddress}"/>
19     public class UnaryNumberToAddressOrOperationConverter<TLinkAddress> :
20         ↪ LinksOperatorBase<TLinkAddress>, IConverter<TLinkAddress>
21     {
22         private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
23             ↪ EqualityComparer<TLinkAddress>.Default;
24         private static readonly TLinkAddress _zero = default;
25         private static readonly TLinkAddress _one = Arithmetic.Increment(_zero);
26         private readonly IDictionary<TLinkAddress, int> _unaryNumberPowerOf2Indicies;
27
28         /// <summary>
29         /// <para>
30         /// Initializes a new <see cref="UnaryNumberToAddressOrOperationConverter"/> instance.
31         /// </para>
32         /// <para></para>
33         /// </summary>
34         /// <param name="links">
35         /// <para>A links.</para>

```

```

34     /// <para></para>
35     /// </param>
36     /// <param name="powerOf2ToUnaryNumberConverter">
37     /// <para>A power of to unary number converter.</para>
38     /// <para></para>
39     /// </param>
40     [MethodImpl(MethodImplOptions.AggressiveInlining)]
41     public UnaryNumberToAddressOrOperationConverter(ILinks<TLinkAddress> links,
42         ↪ IConverter<int, TLinkAddress> powerOf2ToUnaryNumberConverter) : base(links) =>
43         ↪ _unaryNumberPowerOf2Indicies =
44         ↪ CreateUnaryNumberPowerOf2IndiciesDictionary(powerOf2ToUnaryNumberConverter);
45
46     /// <summary>
47     /// <para>
48     /// Converts the source number.
49     /// </para>
50     /// <para></para>
51     /// </summary>
52     /// <param name="sourceNumber">
53     /// <para>The source number.</para>
54     /// <para></para>
55     /// </param>
56     /// <returns>
57     /// <para>The target.</para>
58     /// <para></para>
59     /// </returns>
60     [MethodImpl(MethodImplOptions.AggressiveInlining)]
61     public TLinkAddress Convert(TLinkAddress sourceNumber)
62     {
63         var links = _links;
64         var nullConstant = links.Constants.Null;
65         var source = sourceNumber;
66         var target = nullConstant;
67         if (!_equalityComparer.Equals(source, nullConstant))
68         {
69             while (true)
70             {
71                 if (_unaryNumberPowerOf2Indicies.TryGetValue(source, out int powerOf2Index))
72                 {
73                     SetBit(ref target, powerOf2Index);
74                     break;
75                 }
76                 else
77                 {
78                     powerOf2Index = _unaryNumberPowerOf2Indicies[links.GetSource(source)];
79                     SetBit(ref target, powerOf2Index);
80                     source = links.GetTarget(source);
81                 }
82             }
83             return target;
84         }
85     }
86     [MethodImpl(MethodImplOptions.AggressiveInlining)]
87     private static Dictionary<TLinkAddress, int>
88     ↪ CreateUnaryNumberPowerOf2IndiciesDictionary(IConverter<int, TLinkAddress>
89     ↪ powerOf2ToUnaryNumberConverter)
90     {
91         var unaryNumberPowerOf2Indicies = new Dictionary<TLinkAddress, int>();
92         for (int i = 0; i < NumericType<TLinkAddress>.BitsSize; i++)
93         {
94             unaryNumberPowerOf2Indicies.Add(powerOf2ToUnaryNumberConverter.Convert(i), i);
95         }
96         return unaryNumberPowerOf2Indicies;
97     }
98     [MethodImpl(MethodImplOptions.AggressiveInlining)]
99     private static void SetBit(ref TLinkAddress target, int powerOf2Index) => target =
100     ↪ Bit.Or(target, Bit.ShiftLeft(_one, powerOf2Index));
101 }

```

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

```

1  // using System;
2  // using System.Collections.Generic;
3  // using System.Runtime.CompilerServices;
4  // using System.Linq;
5  // using System.Text;
6  // using Platform.Collections;
7  // using Platform.Collections.Sets;

```

```

8 // using Platform.Collections.Stacks;
9 // using Platform.Data.Exceptions;
10 // using Platform.Data.Sequences;
11 // using Platform.Data.Doublets.Sequences.Frequencies.Counters;
12 // using Platform.Data.Doublets.Sequences.Walkers;
13 // using LinkIndex = System.UInt64;
14 // using Stack = System.Collections.Generic.Stack<ulong>;
15 //
16 // #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
17 //
18 // namespace Platform.Data.Doublets.Sequences
19 // {
20 //     /// <summary>
21 //     /// <para>
22 //     /// Represents the sequences.
23 //     /// </para>
24 //     /// <para></para>
25 //     /// </summary>
26 //     partial class Sequences
27 //     {
28 //         #region Create All Variants (Not Practical)
29 //
30 //         /// <remarks>
31 //         /// Number of links that is needed to generate all variants for
32 //         /// sequence of length N corresponds to https://oeis.org/A014143/list sequence.
33 //         /// </remarks>
34 //         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35 //         public ulong[] CreateAllVariants2(ulong[] sequence)
36 //         {
37 //             return _sync.DoWrite(() =>
38 //             {
39 //                 if (sequence.IsNullOrEmpty())
40 //                 {
41 //                     return Array.Empty<ulong>();
42 //                 }
43 //                 Links.EnsureLinkExists(sequence);
44 //                 if (sequence.Length == 1)
45 //                 {
46 //                     return sequence;
47 //                 }
48 //                 return CreateAllVariants2Core(sequence, 0, (ulong)sequence.Length - 1);
49 //             });
50 //         }
51 //         [MethodImpl(MethodImplOptions.AggressiveInlining)]
52 //         private ulong[] CreateAllVariants2Core(ulong[] sequence, ulong startAt, ulong stopAt)
53 //         {
54 //             if ((stopAt - startAt) == 0)
55 //             {
56 //                 return new[] { sequence[startAt] };
57 //             }
58 //             if ((stopAt - startAt) == 1)
59 //             {
60 //                 return new[] { Links.Unsync.GetOrCreate(sequence[startAt], sequence[stopAt])
61 // ↵ };
62 //             }
63 //             var variants = new ulong[Platform.Numbers.Math.Catalan(stopAt - startAt)];
64 //             var last = 0;
65 //             for (var splitter = startAt; splitter < stopAt; splitter++)
66 //             {
67 //                 var left = CreateAllVariants2Core(sequence, startAt, splitter);
68 //                 var right = CreateAllVariants2Core(sequence, splitter + 1, stopAt);
69 //                 for (var i = 0; i < left.Length; i++)
70 //                 {
71 //                     for (var j = 0; j < right.Length; j++)
72 //                     {
73 //                         var variant = Links.Unsync.GetOrCreate(left[i], right[j]);
74 //                         if (variant == Constants.Null)
75 //                         {
76 //                             throw new NotImplementedException("Creation cancellation is not
77 ↵ implemented.");
78 //                         }
79 //                         variants[last++] = variant;
80 //                     }
81 //                 }
82 //             }
83 //             return variants;
84 //         }
85 //     }
86 // }

```

```

83 //
84 //      /// <summary>
85 //      /// <para>
86 //      /// Creates the all variants 1 using the specified sequence.
87 //      /// </para>
88 //      /// <para></para>
89 //      /// </summary>
90 //      /// <param name="sequence">
91 //      /// <para>The sequence.</para>
92 //      /// <para></para>
93 //      /// </param>
94 //      /// <returns>
95 //      /// <para>A list of ulong</para>
96 //      /// <para></para>
97 //      /// </returns>
98 //      [MethodImpl(MethodImplOptions.AggressiveInlining)]
99 //      public List<ulong> CreateAllVariants1(params ulong[] sequence)
100 //      {
101 //          return _sync.DoWrite(() =>
102 //          {
103 //              if (sequence.IsNullOrEmpty())
104 //              {
105 //                  return new List<ulong>();
106 //              }
107 //              Links.Unsync.EnsureLinkExists(sequence);
108 //              if (sequence.Length == 1)
109 //              {
110 //                  return new List<ulong> { sequence[0] };
111 //              }
112 //              var results = new
113 //      ↪ List<ulong>((int)Platform.Numbers.Math.Catalan((ulong)sequence.Length));
114 //              return CreateAllVariants1Core(sequence, results);
115 //          });
116 //      [MethodImpl(MethodImplOptions.AggressiveInlining)]
117 //      private List<ulong> CreateAllVariants1Core(ulong[] sequence, List<ulong> results)
118 //      {
119 //          if (sequence.Length == 2)
120 //          {
121 //              var link = Links.Unsync.GetOrCreate(sequence[0], sequence[1]);
122 //              if (link == Constants.Null)
123 //              {
124 //                  throw new NotImplementedException("Creation cancellation is not
125 //      ↪ implemented.");
126 //              }
127 //              results.Add(link);
128 //              return results;
129 //          }
130 //          var innerSequenceLength = sequence.Length - 1;
131 //          var innerSequence = new ulong[innerSequenceLength];
132 //          for (var li = 0; li < innerSequenceLength; li++)
133 //          {
134 //              var link = Links.Unsync.GetOrCreate(sequence[li], sequence[li + 1]);
135 //              if (link == Constants.Null)
136 //              {
137 //                  throw new NotImplementedException("Creation cancellation is not
138 //      ↪ implemented.");
139 //              }
140 //              for (var isi = 0; isi < li; isi++)
141 //              {
142 //                  innerSequence[isi] = sequence[isi];
143 //              }
144 //              innerSequence[li] = link;
145 //              for (var isi = li + 1; isi < innerSequenceLength; isi++)
146 //              {
147 //                  innerSequence[isi] = sequence[isi + 1];
148 //              }
149 //              CreateAllVariants1Core(innerSequence, results);
150 //          }
151 //          return results;
152 //      }
153 //
154 //      #endregion
155 //
156 //      /// <summary>
157 //      /// <para>
158 //      /// Eaches the 1 using the specified sequence.

```



```

157 //      /// </para>
158 //      /// <para></para>
159 //      /// </summary>
160 //      /// <param name="sequence">
161 //      /// <para>The sequence.</para>
162 //      /// <para></para>
163 //      /// </param>
164 //      /// <returns>
165 //      /// <para>The visited links.</para>
166 //      /// <para></para>
167 //      /// </returns>
168 //      [MethodImpl(MethodImplOptions.AggressiveInlining)]
169 //      public HashSet<ulong> Each1(params ulong[] sequence)
170 //      {
171 //          var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
172 //          Each1(link =>
173 //              {
174 //                  if (!visitedLinks.Contains(link))
175 //                  {
176 //                      visitedLinks.Add(link); // изучить почему случаются повторы
177 //                  }
178 //                  return true;
179 //              }, sequence);
180 //          return visitedLinks;
181 //      }
182 //      [MethodImpl(MethodImplOptions.AggressiveInlining)]
183 //      private void Each1(Func<ulong, bool> handler, params ulong[] sequence)
184 //      {
185 //          if (sequence.Length == 2)
186 //          {
187 //              Links.Unsync.Each(sequence[0], sequence[1], handler);
188 //          }
189 //          else
190 //          {
191 //              var innerSequenceLength = sequence.Length - 1;
192 //              for (var li = 0; li < innerSequenceLength; li++)
193 //              {
194 //                  var left = sequence[li];
195 //                  var right = sequence[li + 1];
196 //                  if (left == 0 && right == 0)
197 //                  {
198 //                      continue;
199 //                  }
200 //                  var linkIndex = li;
201 //                  ulong[] innerSequence = null;
202 //                  Links.Unsync.Each(doublet =>
203 //                      {
204 //                          if (innerSequence == null)
205 //                          {
206 //                              innerSequence = new ulong[innerSequenceLength];
207 //                              for (var isi = 0; isi < linkIndex; isi++)
208 //                              {
209 //                                  innerSequence[isi] = sequence[isi];
210 //                              }
211 //                              for (var isi = linkIndex + 1; isi < innerSequenceLength; isi++)
212 //                              {
213 //                                  innerSequence[isi] = sequence[isi + 1];
214 //                              }
215 //                              innerSequence[linkIndex] = doublet[Constants.IndexPart];
216 //                              Each1(handler, innerSequence);
217 //                              return Constants.Continue;
218 //                          }, Constants.Any, left, right);
219 //                      }
220 //              }
221 //          }
222 //      }
223 //
224 //      /// <summary>
225 //      /// <para>
226 //      /// Eaches the part using the specified sequence.
227 //      /// </para>
228 //      /// <para></para>
229 //      /// </summary>
230 //      /// <param name="sequence">
231 //      /// <para>The sequence.</para>
232 //      /// <para></para>
233 //      /// </param>
234 //      /// <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 //              {

```

```

310 //         var match = Links.SearchDefault(sequence[0], doublet);
311 //         if (match != Constants.Null)
312 //         {
313 //             handler(new LinkAddress<LinkIndex>(match));
314 //         }
315 //         return true;
316 //     });
317 //     // |_x          ... x_o
318 //     // |_o          |__|
319 //     Links.Each(Constants.Any, sequence[0], doublet =>
320 //     {
321 //         var match = Links.SearchOrDefault(doublet, sequence[1]);
322 //         if (match != 0)
323 //         {
324 //             handler(new LinkAddress<LinkIndex>(match));
325 //         }
326 //         return true;
327 //     });
328 //     //         ..x o_.
329 //     //         |__|
330 //     PartialStepRight(x => handler(x), sequence[0], sequence[1]);
331 // }
332 // else
333 // {
334 //     throw new NotImplementedException();
335 // }
336 // }
337 // [MethodImpl(MethodImplOptions.AggressiveInlining)]
338 // private void PartialStepRight(Action<IList<LinkIndex>> handler, ulong left, ulong
↵ right)
339 // {
340 //     Links.Unsync.Each(Constants.Any, left, doublet =>
341 //     {
342 //         StepRight(handler, doublet, right);
343 //         if (left != doublet)
344 //         {
345 //             PartialStepRight(handler, doublet, right);
346 //         }
347 //         return true;
348 //     });
349 // }
350 // [MethodImpl(MethodImplOptions.AggressiveInlining)]
351 // private void StepRight(Action<IList<LinkIndex>> handler, ulong left, ulong right)
352 // {
353 //     Links.Unsync.Each(left, Constants.Any, rightStep =>
354 //     {
355 //         TryStepRightUp(handler, right, rightStep);
356 //         return true;
357 //     });
358 // }
359 // [MethodImpl(MethodImplOptions.AggressiveInlining)]
360 // private void TryStepRightUp(Action<IList<LinkIndex>> handler, ulong right, ulong
↵ stepFrom)
361 // {
362 //     var upStep = stepFrom;
363 //     var firstSource = Links.Unsync.GetTarget(upStep);
364 //     while (firstSource != right && firstSource != upStep)
365 //     {
366 //         upStep = firstSource;
367 //         firstSource = Links.Unsync.GetSource(upStep);
368 //     }
369 //     if (firstSource == right)
370 //     {
371 //         handler(new LinkAddress<LinkIndex>(stepFrom));
372 //     }
373 // }
374 //
375 // // TODO: Test
376 // [MethodImpl(MethodImplOptions.AggressiveInlining)]
377 // private void PartialStepLeft(Action<IList<LinkIndex>> handler, ulong left, ulong
↵ right)
378 // {
379 //     Links.Unsync.Each(right, Constants.Any, doublet =>
380 //     {
381 //         StepLeft(handler, left, doublet);
382 //         if (right != doublet)
383 //         {

```

```

384 //         PartialStepLeft(handler, left, doublet);
385 //     }
386 //     return true;
387 // });
388 // }
389 // [MethodImpl(MethodImplOptions.AggressiveInlining)]
390 // private void StepLeft(Action<IList<LinkIndex>> handler, ulong left, ulong right)
391 // {
392 //     Links.Unsync.Each(Constants.Any, right, leftStep =>
393 //     {
394 //         TryStepLeftUp(handler, left, leftStep);
395 //         return true;
396 //     });
397 // }
398 // [MethodImpl(MethodImplOptions.AggressiveInlining)]
399 // private void TryStepLeftUp(Action<IList<LinkIndex>> handler, ulong left, ulong
↵ stepFrom)
400 // {
401 //     var upStep = stepFrom;
402 //     var firstTarget = Links.Unsync.GetSource(upStep);
403 //     while (firstTarget != left && firstTarget != upStep)
404 //     {
405 //         upStep = firstTarget;
406 //         firstTarget = Links.Unsync.GetTarget(upStep);
407 //     }
408 //     if (firstTarget == left)
409 //     {
410 //         handler(new LinkAddress<LinkIndex>(stepFrom));
411 //     }
412 // }
413 // [MethodImpl(MethodImplOptions.AggressiveInlining)]
414 // private bool StartsWith(ulong sequence, ulong link)
415 // {
416 //     var upStep = sequence;
417 //     var firstSource = Links.Unsync.GetSource(upStep);
418 //     while (firstSource != link && firstSource != upStep)
419 //     {
420 //         upStep = firstSource;
421 //         firstSource = Links.Unsync.GetSource(upStep);
422 //     }
423 //     return firstSource == link;
424 // }
425 // [MethodImpl(MethodImplOptions.AggressiveInlining)]
426 // private bool EndsWith(ulong sequence, ulong link)
427 // {
428 //     var upStep = sequence;
429 //     var lastTarget = Links.Unsync.GetTarget(upStep);
430 //     while (lastTarget != link && lastTarget != upStep)
431 //     {
432 //         upStep = lastTarget;
433 //         lastTarget = Links.Unsync.GetTarget(upStep);
434 //     }
435 //     return lastTarget == link;
436 // }
437 //
438 // /// <summary>
439 // /// <para>
440 // /// Gets the all matching sequences 0 using the specified sequence.
441 // /// </para>
442 // /// <para></para>
443 // /// </summary>
444 // /// <param name="sequence">
445 // /// <para>The sequence.</para>
446 // /// <para></para>
447 // /// </param>
448 // /// <returns>
449 // /// <para>A list of ulong</para>
450 // /// <para></para>
451 // /// </returns>
452 // [MethodImpl(MethodImplOptions.AggressiveInlining)]
453 // public List<ulong> GetAllMatchingSequences0(params ulong[] sequence)
454 // {
455 //     return _sync.DoRead(() =>
456 //     {
457 //         var results = new List<ulong>();
458 //         if (sequence.Length > 0)
459 //         {

```

```

460 // Links.EnsureLinkExists(sequence);
461 // var firstElement = sequence[0];
462 // if (sequence.Length == 1)
463 // {
464 //     results.Add(firstElement);
465 //     return results;
466 // }
467 // if (sequence.Length == 2)
468 // {
469 //     var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
470 //     if (doublet != Constants.Null)
471 //     {
472 //         results.Add(doublet);
473 //     }
474 //     return results;
475 // }
476 // var linksInSequence = new HashSet<ulong>(sequence);
477 // void handler(ICollection<LinkIndex> result)
478 // {
479 //     var resultIndex = result[Links.Constants.IndexPart];
480 //     var filterPosition = 0;
481 //     StopableSequenceWalker.WalkRight(resultIndex, Links.Unsync.GetSource,
↳ Links.Unsync.GetTarget,
482 //     x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) ==
↳ x, x =>
483 //     {
484 //         if (filterPosition == sequence.Length)
485 //         {
486 //             filterPosition = -2; // Длиннее чем нужно
487 //             return false;
488 //         }
489 //         if (x != sequence[filterPosition])
490 //         {
491 //             filterPosition = -1;
492 //             return false; // Начинается иначе
493 //         }
494 //         filterPosition++;
495 //         return true;
496 //     });
497 //     if (filterPosition == sequence.Length)
498 //     {
499 //         results.Add(resultIndex);
500 //     }
501 // }
502 // if (sequence.Length >= 2)
503 // {
504 //     StepRight(handler, sequence[0], sequence[1]);
505 // }
506 // var last = sequence.Length - 2;
507 // for (var i = 1; i < last; i++)
508 // {
509 //     PartialStepRight(handler, sequence[i], sequence[i + 1]);
510 // }
511 // if (sequence.Length >= 3)
512 // {
513 //     StepLeft(handler, sequence[sequence.Length - 2],
↳ sequence[sequence.Length - 1]);
514 //     }
515 // }
516 // }
517 // return results;
518 // });
519 // }
520 //
521 // /// <summary>
522 // /// <para>
523 // /// Gets the all matching sequences 1 using the specified sequence.
524 // /// </para>
525 // /// <para></para>
526 // /// </summary>
527 // /// <param name="sequence">
528 // /// <para>The sequence.</para>
529 // /// <para></para>
530 // /// </param>
531 // /// <returns>
532 // /// <para>A hash set of ulong</para>
533 // /// <para></para>
534 // /// </returns>

```

```

535 // [MethodImpl(MethodImplOptions.AggressiveInlining)]
536 // public HashSet<ulong> GetAllMatchingSequences1(params ulong[] sequence)
537 // {
538 //     return _sync.DoRead(() =>
539 //     {
540 //         var results = new HashSet<ulong>();
541 //         if (sequence.Length > 0)
542 //         {
543 //             Links.EnsureLinkExists(sequence);
544 //             var firstElement = sequence[0];
545 //             if (sequence.Length == 1)
546 //             {
547 //                 results.Add(firstElement);
548 //                 return results;
549 //             }
550 //             if (sequence.Length == 2)
551 //             {
552 //                 var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
553 //                 if (doublet != Constants.Null)
554 //                 {
555 //                     results.Add(doublet);
556 //                 }
557 //                 return results;
558 //             }
559 //             var matcher = new Matcher(this, sequence, results, null);
560 //             if (sequence.Length >= 2)
561 //             {
562 //                 StepRight(matcher.AddFullMatchedToResults, sequence[0], sequence[1]);
563 //             }
564 //             var last = sequence.Length - 2;
565 //             for (var i = 1; i < last; i++)
566 //             {
567 //                 PartialStepRight(matcher.AddFullMatchedToResults, sequence[i],
↵ sequence[i + 1]);
568 //             }
569 //             if (sequence.Length >= 3)
570 //             {
571 //                 StepLeft(matcher.AddFullMatchedToResults, sequence[sequence.Length -
↵ 2], sequence[sequence.Length - 1]);
572 //             }
573 //         }
574 //         return results;
575 //     });
576 // }
577 //
578 // /// <summary>
579 // /// <para>
580 // /// The max sequence format size.
581 // /// </para>
582 // /// <para></para>
583 // /// </summary>
584 // public const int MaxSequenceFormatSize = 200;
585 //
586 // /// <summary>
587 // /// <para>
588 // /// Formats the sequence using the specified sequence link.
589 // /// </para>
590 // /// <para></para>
591 // /// </summary>
592 // /// <param name="sequenceLink">
593 // /// <para>The sequence link.</para>
594 // /// <para></para>
595 // /// </param>
596 // /// <param name="knownElements">
597 // /// <para>The known elements.</para>
598 // /// <para></para>
599 // /// </param>
600 // /// <returns>
601 // /// <para>The string</para>
602 // /// <para></para>
603 // /// </returns>
604 // [MethodImpl(MethodImplOptions.AggressiveInlining)]
605 // public string FormatSequence(LinkIndex sequenceLink, params LinkIndex[]
↵ knownElements) => FormatSequence(sequenceLink, (sb, x) => sb.Append(x), true, knownElements);
606 //
607 // /// <summary>
608 // /// <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)

```



```

747 // {
748 //     return true;
749 // }
750 // sb.Append(insertComma ? ", ..." : "...");
751 // return false;
752 // });
753 // }
754 // sb.Append('}');
755 // return sb.ToString();
756 // }
757 //
758 // /// <summary>
759 // /// <para>
760 // /// Gets the all partially matching sequences 0 using the specified sequence.
761 // /// </para>
762 // /// <para></para>
763 // /// </summary>
764 // /// <param name="sequence">
765 // /// <para>The sequence.</para>
766 // /// <para></para>
767 // /// </param>
768 // /// <returns>
769 // /// <para>A list of ulong</para>
770 // /// <para></para>
771 // /// </returns>
772 // [MethodImpl(MethodImplOptions.AggressiveInlining)]
773 // public List<ulong> GetAllPartiallyMatchingSequences0(params ulong[] sequence)
774 // {
775 //     return _sync.DoRead(() =>
776 //     {
777 //         if (sequence.Length > 0)
778 //         {
779 //             Links.EnsureLinkExists(sequence);
780 //             var results = new HashSet<ulong>();
781 //             for (var i = 0; i < sequence.Length; i++)
782 //             {
783 //                 AllUsagesCore(sequence[i], results);
784 //             }
785 //             var filteredResults = new List<ulong>();
786 //             var linksInSequence = new HashSet<ulong>(sequence);
787 //             foreach (var result in results)
788 //             {
789 //                 var filterPosition = -1;
790 //                 StopableSequenceWalker.WalkRight(result, Links.Unsync.GetSource,
791 // ↪ Links.Unsync.GetTarget,
792 // ↪ x, x =>
793 //                 {
794 //                     if (filterPosition == (sequence.Length - 1))
795 //                     {
796 //                         return false;
797 //                     }
798 //                     if (filterPosition >= 0)
799 //                     {
800 //                         if (x == sequence[filterPosition + 1])
801 //                         {
802 //                             filterPosition++;
803 //                         }
804 //                         else
805 //                         {
806 //                             return false;
807 //                         }
808 //                     }
809 //                     if (filterPosition < 0)
810 //                     {
811 //                         if (x == sequence[0])
812 //                         {
813 //                             filterPosition = 0;
814 //                         }
815 //                     }
816 //                     return true;
817 //                 });
818 //                 if (filterPosition == (sequence.Length - 1))
819 //                 {
820 //                     filteredResults.Add(result);
821 //                 }
822 //             }
823 //             return filteredResults;

```

```

823 //      }
824 //      return new List<ulong>();
825 //  });
826 //  }
827 //
828 //      /// <summary>
829 //      /// <para>
830 //      /// Gets the all partially matching sequences 1 using the specified sequence.
831 //      /// </para>
832 //      /// <para></para>
833 //      /// </summary>
834 //      /// <param name="sequence">
835 //      /// <para>The sequence.</para>
836 //      /// <para></para>
837 //      /// </param>
838 //      /// <returns>
839 //      /// <para>A hash set of ulong</para>
840 //      /// <para></para>
841 //      /// </returns>
842 //      [MethodImpl(MethodImplOptions.AggressiveInlining)]
843 //      public HashSet<ulong> GetAllPartiallyMatchingSequences1(params ulong[] sequence)
844 //      {
845 //          return _sync.DoRead(() =>
846 //          {
847 //              if (sequence.Length > 0)
848 //              {
849 //                  Links.EnsureLinkExists(sequence);
850 //                  var results = new HashSet<ulong>();
851 //                  for (var i = 0; i < sequence.Length; i++)
852 //                  {
853 //                      AllUsagesCore(sequence[i], results);
854 //                  }
855 //                  var filteredResults = new HashSet<ulong>();
856 //                  var matcher = new Matcher(this, sequence, filteredResults, null);
857 //                  matcher.AddAllPartialMatchedToResults(results);
858 //                  return filteredResults;
859 //              }
860 //              return new HashSet<ulong>();
861 //          });
862 //      }
863 //
864 //      /// <summary>
865 //      /// <para>
866 //      /// Determines whether this instance get all partially matching sequences 2.
867 //      /// </para>
868 //      /// <para></para>
869 //      /// </summary>
870 //      /// <param name="handler">
871 //      /// <para>The handler.</para>
872 //      /// <para></para>
873 //      /// </param>
874 //      /// <param name="sequence">
875 //      /// <para>The sequence.</para>
876 //      /// <para></para>
877 //      /// </param>
878 //      /// <returns>
879 //      /// <para>The bool</para>
880 //      /// <para></para>
881 //      /// </returns>
882 //      [MethodImpl(MethodImplOptions.AggressiveInlining)]
883 //      public bool GetAllPartiallyMatchingSequences2(Func<IList<LinkIndex>, LinkIndex>
↵ handler, params ulong[] sequence)
884 //      {
885 //          return _sync.DoRead(() =>
886 //          {
887 //              if (sequence.Length > 0)
888 //              {
889 //                  Links.EnsureLinkExists(sequence);
890 //
891 //                  var results = new HashSet<ulong>();
892 //                  var filteredResults = new HashSet<ulong>();
893 //                  var matcher = new Matcher(this, sequence, filteredResults, handler);
894 //                  for (var i = 0; i < sequence.Length; i++)
895 //                  {
896 //                      if (!AllUsagesCore1(sequence[i], results,
↵ matcher.HandlePartialMatched))
897 //                      {

```

```

898 //         return false;
899 //     }
900 // }
901 //     return true;
902 // }
903 //     return true;
904 // });
905 // }
906 //
907 // //public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
908 // //{
909 // //    return Sync.DoRead(() =>
910 // //    {
911 // //        if (sequence.Length > 0)
912 // //        {
913 // //            _links.EnsureEachLinkIsAnyOrExists(sequence);
914 // //
915 // //            var firstResults = new HashSet<ulong>();
916 // //            var lastResults = new HashSet<ulong>();
917 // //
918 // //            var first = sequence.First(x => x != LinksConstants.Any);
919 // //            var last = sequence.Last(x => x != LinksConstants.Any);
920 // //
921 // //            AllUsagesCore(first, firstResults);
922 // //            AllUsagesCore(last, lastResults);
923 // //
924 // //            firstResults.IntersectWith(lastResults);
925 // //
926 // //            //for (var i = 0; i < sequence.Length; i++)
927 // //            //    AllUsagesCore(sequence[i], results);
928 // //
929 // //            var filteredResults = new HashSet<ulong>();
930 // //            var matcher = new Matcher(this, sequence, filteredResults, null);
931 // //            matcher.AddAllPartialMatchedToResults(firstResults);
932 // //            return filteredResults;
933 // //        }
934 // //
935 // //        return new HashSet<ulong>();
936 // //    });
937 // //}
938 //
939 // /// <summary>
940 // /// <para>
941 // /// Gets the all partially matching sequences 3 using the specified sequence.
942 // /// </para>
943 // /// <para></para>
944 // /// </summary>
945 // /// <param name="sequence">
946 // /// <para>The sequence.</para>
947 // /// <para></para>
948 // /// </param>
949 // /// <returns>
950 // /// <para>A hash set of ulong</para>
951 // /// <para></para>
952 // /// </returns>
953 // [MethodImpl(MethodImplOptions.AggressiveInlining)]
954 // public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
955 // {
956 //     return _sync.DoRead(() =>
957 //     {
958 //         if (sequence.Length > 0)
959 //         {
960 //             ILinksExtensions.EnsureLinkIsAnyOrExists(Links, sequence);
961 //             var firstResults = new HashSet<ulong>();
962 //             var lastResults = new HashSet<ulong>();
963 //             var first = sequence.First(x => x != Constants.Any);
964 //             var last = sequence.Last(x => x != Constants.Any);
965 //             AllUsagesCore(first, firstResults);
966 //             AllUsagesCore(last, lastResults);
967 //             firstResults.IntersectWith(lastResults);
968 //             //for (var i = 0; i < sequence.Length; i++)
969 //             //    AllUsagesCore(sequence[i], results);
970 //             var filteredResults = new HashSet<ulong>();
971 //             var matcher = new Matcher(this, sequence, filteredResults, null);
972 //             matcher.AddAllPartialMatchedToResults(firstResults);
973 //             return filteredResults;
974 //         }
975 //         return new HashSet<ulong>();

```

```

976 //     });
977 // }
978 //
979 //     /// <summary>
980 //     /// <para>
981 //     /// Gets the all partially matching sequences 4 using the specified read as elements.
982 //     /// </para>
983 //     /// <para></para>
984 //     /// </summary>
985 //     /// <param name="readAsElements">
986 //     /// <para>The read as elements.</para>
987 //     /// <para></para>
988 //     /// </param>
989 //     /// <param name="sequence">
990 //     /// <para>The sequence.</para>
991 //     /// <para></para>
992 //     /// </param>
993 //     /// <returns>
994 //     /// <para>A hash set of ulong</para>
995 //     /// <para></para>
996 //     /// </returns>
997 //     [MethodImpl(MethodImplOptions.AggressiveInlining)]
998 //     public HashSet<ulong> GetAllPartiallyMatchingSequences4(HashSet<ulong>
↵ readAsElements, IList<ulong> sequence)
999 //     {
1000 //         return _sync.DoRead(() =>
1001 //         {
1002 //             if (sequence.Count > 0)
1003 //             {
1004 //                 Links.EnsureLinkExists(sequence);
1005 //                 var results = new HashSet<LinkIndex>();
1006 //                 //var nextResults = new HashSet<ulong>();
1007 //                 //for (var i = 0; i < sequence.Length; i++)
1008 //                 //{
1009 //                     // AllUsagesCore(sequence[i], nextResults);
1010 //                     // if (results.IsNullOrEmpty())
1011 //                     // {
1012 //                         // results = nextResults;
1013 //                         // nextResults = new HashSet<ulong>();
1014 //                     // }
1015 //                     // else
1016 //                     // {
1017 //                         // results.IntersectWith(nextResults);
1018 //                         // nextResults.Clear();
1019 //                     // }
1020 //                 //}
1021 //                 var collector1 = new AllUsagesCollector1(Links.Unsync, results);
1022 //                 collector1.Collect(Links.Unsync.GetLink(sequence[0]));
1023 //                 var next = new HashSet<ulong>();
1024 //                 for (var i = 1; i < sequence.Count; i++)
1025 //                 {
1026 //                     var collector = new AllUsagesCollector1(Links.Unsync, next);
1027 //                     collector.Collect(Links.Unsync.GetLink(sequence[i]));
1028 //
1029 //                     results.IntersectWith(next);
1030 //                     next.Clear();
1031 //                 }
1032 //                 var filteredResults = new HashSet<ulong>();
1033 //                 var matcher = new Matcher(this, sequence, filteredResults, null,
↵ readAsElements);
1034 //                 matcher.AddAllPartialMatchedToResultsAndReadAsElements(results.OrderBy(x
↵ => x)); // OrderBy is a Hack
1035 //                 return filteredResults;
1036 //             }
1037 //             return new HashSet<ulong>();
1038 //         });
1039 //     }
1040 //
1041 //     // Does not work
1042 //     //public HashSet<ulong> GetAllPartiallyMatchingSequences5(HashSet<ulong>
↵ readAsElements, params ulong[] sequence)
1043 //     //{
1044 //         // var visited = new HashSet<ulong>();
1045 //         // var results = new HashSet<ulong>();
1046 //         // var matcher = new Matcher(this, sequence, visited, x => { results.Add(x);
↵ return true; }, readAsElements);
1047 //         // 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))
1094 //                     //        results.Add(x);
1095 //                     //    return true;
1096 //                 };
1097 //                 //if (sequence.Length >= 2)
1098 //                 //    StepRight(handler, sequence[0], sequence[1]);
1099 //                 //var last = sequence.Length - 2;
1100 //                 //for (var i = 1; i < last; i++)
1101 //                 //    PartialStepRight(handler, sequence[i], sequence[i + 1]);
1102 //                 //if (sequence.Length >= 3)
1103 //                 //    StepLeft(handler, sequence[sequence.Length - 2],
1104 //                 //        sequence[sequence.Length - 1]);
1105 //                 //if (sequence.Length == 1)
1106 //                 //{
1107 //                     throw new NotImplementedException(); // all sequences,
1108 //                     containing this element?
1109 //                 }
1110 //                 //if (sequence.Length == 2)
1111 //                 //{
1112 //                     var results = new List<ulong>();
1113 //                     PartialStepRight(results.Add, sequence[0], sequence[1]);
1114 //                     return results;
1115 //                 }
1116 //                 //var matches = new List<List<ulong>>();
1117 //                 //var last = sequence.Length - 1;
1118 //                 //for (var i = 0; i < last; i++)
1119 //                 //{
1120 //                     var results = new List<ulong>();
1121 //                     //StepRight(results.Add, sequence[i], sequence[i + 1]);
1122 //                     PartialStepRight(results.Add, sequence[i], sequence[i + 1]);
1123 //                     if (results.Count > 0)
1124 //                         matches.Add(results);
1125 //                 }
1126 //                 return matches;
1127 //             }
1128 //             return new List<List<ulong>>();
1129 //         });
1130 //     }

```

```

1122 //          // else
1123 //          //         return results;
1124 //          //         if (matches.Count == 2)
1125 //          //         {
1126 //          //             var merged = new List<ulong>();
1127 //          //             for (var j = 0; j < matches[0].Count; j++)
1128 //          //                 for (var k = 0; k < matches[1].Count; k++)
1129 //          //                     CloseInnerConnections(merged.Add, matches[0][j],
↵ matches[1][k]);
1130 //          //             if (merged.Count > 0)
1131 //          //                 matches = new List<List<ulong>> { merged };
1132 //          //             else
1133 //          //                 return new List<ulong>();
1134 //          //             }
1135 //          //         }
1136 //          //         if (matches.Count > 0)
1137 //          //         {
1138 //          //             var usages = new HashSet<ulong>();
1139 //          //             for (int i = 0; i < sequence.Length; i++)
1140 //          //             {
1141 //          //                 AllUsagesCore(sequence[i], usages);
1142 //          //             }
1143 //          //             //for (int i = 0; i < matches[0].Count; i++)
1144 //          //                 // AllUsagesCore(matches[0][i], usages);
1145 //          //                 //usages.UnionWith(matches[0]);
1146 //          //                 return usages.ToList();
1147 //          //             }
1148 //          //             var firstLinkUsages = new HashSet<ulong>();
1149 //          //             AllUsagesCore(sequence[0], firstLinkUsages);
1150 //          //             firstLinkUsages.Add(sequence[0]);
1151 //          //             //var previousMatchings = firstLinkUsages.ToList(); //new List<ulong>() {
↵ sequence[0] }; // or all sequences, containing this element?
1152 //          //             //return GetAllPartiallyMatchingSequencesCore(sequence, firstLinkUsages,
↵ 1).ToList();
1153 //          //             var results = new HashSet<ulong>();
1154 //          //             foreach (var match in GetAllPartiallyMatchingSequencesCore(sequence,
↵ firstLinkUsages, 1))
1155 //          //             {
1156 //          //                 AllUsagesCore(match, results);
1157 //          //             }
1158 //          //             return results.ToList();
1159 //          //         }
1160 //          //         return new List<ulong>();
1161 //          //     });
1162 //    }
1163 //
1164 //    /// <remarks>
1165 //    /// TODO: Может потребоваться ограничение на уровень глубины рекурсии
1166 //    /// </remarks>
1167 //    [MethodImpl(MethodImplOptions.AggressiveInlining)]
1168 //    public HashSet<ulong> AllUsages(ulong link)
1169 //    {
1170 //        return _sync.DoRead(() =>
1171 //        {
1172 //            var usages = new HashSet<ulong>();
1173 //            AllUsagesCore(link, usages);
1174 //            return usages;
1175 //        });
1176 //    }
1177 //
1178 //    // При сборе всех использований (последовательностей) можно сохранять обратный путь к
↵ той связи с которой начинался поиск (STTTSSSTT),
1179 //    // причём достаточно одного бита для хранения перехода влево или вправо
1180 //    [MethodImpl(MethodImplOptions.AggressiveInlining)]
1181 //    private void AllUsagesCore(ulong link, HashSet<ulong> usages)
1182 //    {
1183 //        bool handler(ulong doublet)
1184 //        {
1185 //            if (usages.Add(doublet))
1186 //            {
1187 //                AllUsagesCore(doublet, usages);
1188 //            }
1189 //            return true;
1190 //        }
1191 //        Links.Unsync.Each(link, Constants.Any, handler);
1192 //        Links.Unsync.Each(Constants.Any, link, handler);
1193 //    }

```

```

1194 //
1195 //      /// <summary>
1196 //      /// <para>
1197 //      /// Alls the bottom usages using the specified link.
1198 //      /// </para>
1199 //      /// <para></para>
1200 //      /// </summary>
1201 //      /// <param name="link">
1202 //      /// <para>The link.</para>
1203 //      /// <para></para>
1204 //      /// </param>
1205 //      /// <returns>
1206 //      /// <para>A hash set of ulong</para>
1207 //      /// <para></para>
1208 //      /// </returns>
1209 //      [MethodImpl(MethodImplOptions.AggressiveInlining)]
1210 //      public HashSet<ulong> AllBottomUsages(ulong link)
1211 //      {
1212 //          return _sync.DoRead(() =>
1213 //          {
1214 //              var visits = new HashSet<ulong>();
1215 //              var usages = new HashSet<ulong>();
1216 //              AllBottomUsagesCore(link, visits, usages);
1217 //              return usages;
1218 //          });
1219 //      }
1220 //      [MethodImpl(MethodImplOptions.AggressiveInlining)]
1221 //      private void AllBottomUsagesCore(ulong link, HashSet<ulong> visits, HashSet<ulong>
1222 //      ↪ usages)
1223 //      {
1224 //          bool handler(ulong doublet)
1225 //          {
1226 //              if (visits.Add(doublet))
1227 //              {
1228 //                  AllBottomUsagesCore(doublet, visits, usages);
1229 //              }
1230 //              return true;
1231 //          }
1232 //          if (Links.Unsync.Count(Constants.Any, link) == 0)
1233 //          {
1234 //              usages.Add(link);
1235 //          }
1236 //          else
1237 //          {
1238 //              Links.Unsync.Each(link, Constants.Any, handler);
1239 //              Links.Unsync.Each(Constants.Any, link, handler);
1240 //          }
1241 //      }
1242 //
1243 //      /// <summary>
1244 //      /// <para>
1245 //      /// Calculates the total symbol frequency core using the specified symbol.
1246 //      /// </para>
1247 //      /// <para></para>
1248 //      /// </summary>
1249 //      /// <param name="symbol">
1250 //      /// <para>The symbol.</para>
1251 //      /// <para></para>
1252 //      /// </param>
1253 //      /// <returns>
1254 //      /// <para>The ulong</para>
1255 //      /// <para></para>
1256 //      /// </returns>
1257 //      [MethodImpl(MethodImplOptions.AggressiveInlining)]
1258 //      public ulong CalculateTotalSymbolFrequencyCore(ulong symbol)
1259 //      {
1260 //          if (Options.UseSequenceMarker)
1261 //          {
1262 //              var counter = new
1263 //              ↪ TotalMarkedSequenceSymbolFrequencyOneOffCounter<ulong>(Links, Options.MarkedSequenceMatcher,
1264 //              ↪ symbol);
1265 //              return counter.Count();
1266 //          }
1267 //          else
1268 //          {
1269 //              var counter = new TotalSequenceSymbolFrequencyOneOffCounter<ulong>(Links,
1270 //              ↪ symbol);

```

```

1267 //         return counter.Count();
1268 //     }
1269 // }
1270 // [MethodImpl(MethodImplOptions.AggressiveInlining)]
1271 // private bool AllUsagesCore1(ulong link, HashSet<ulong> usages, Func<IList<LinkIndex>,
↵ LinkIndex> outerHandler)
1272 // {
1273 //     bool handler(ulong doublet)
1274 //     {
1275 //         if (usages.Add(doublet))
1276 //         {
1277 //             if (outerHandler(new LinkAddress<LinkIndex>(doublet)) !=
↵ Constants.Continue)
1278 //             {
1279 //                 return false;
1280 //             }
1281 //             if (!AllUsagesCore1(doublet, usages, outerHandler))
1282 //             {
1283 //                 return false;
1284 //             }
1285 //         }
1286 //         return true;
1287 //     }
1288 //     return Links.Unsync.Each(link, Constants.Any, handler)
1289 //         && Links.Unsync.Each(Constants.Any, link, handler);
1290 // }
1291 //
1292 // /// <summary>
1293 // /// <para>
1294 // /// Calculates the all usages using the specified totals.
1295 // /// </para>
1296 // /// <para></para>
1297 // /// </summary>
1298 // /// <param name="totals">
1299 // /// <para>The totals.</para>
1300 // /// <para></para>
1301 // /// </param>
1302 // [MethodImpl(MethodImplOptions.AggressiveInlining)]
1303 // public void CalculateAllUsages(ulong[] totals)
1304 // {
1305 //     var calculator = new AllUsagesCalculator(Links, totals);
1306 //     calculator.Calculate();
1307 // }
1308 //
1309 // /// <summary>
1310 // /// <para>
1311 // /// Calculates the all usages 2 using the specified totals.
1312 // /// </para>
1313 // /// <para></para>
1314 // /// </summary>
1315 // /// <param name="totals">
1316 // /// <para>The totals.</para>
1317 // /// <para></para>
1318 // /// </param>
1319 // [MethodImpl(MethodImplOptions.AggressiveInlining)]
1320 // public void CalculateAllUsages2(ulong[] totals)
1321 // {
1322 //     var calculator = new AllUsagesCalculator2(Links, totals);
1323 //     calculator.Calculate();
1324 // }
1325 // private class AllUsagesCalculator
1326 // {
1327 //     private readonly SynchronizedLinks<ulong> _links;
1328 //     private readonly ulong[] _totals;
1329 //
1330 //     /// <summary>
1331 //     /// <para>
1332 //     /// Initializes a new <see cref="AllUsagesCalculator"/> instance.
1333 //     /// </para>
1334 //     /// <para></para>
1335 //     /// </summary>
1336 //     /// <param name="links">
1337 //     /// <para>A links.</para>
1338 //     /// <para></para>
1339 //     /// </param>
1340 //     /// <param name="totals">
1341 //     /// <para>A totals.</para>

```



```

1342 //          /// <para></para>
1343 //          /// </param>
1344 //          [MethodImpl(MethodImplOptions.AggressiveInlining)]
1345 //          public AllUsagesCalculator(SynchronizedLinks<ulong> links, ulong[] totals)
1346 //          {
1347 //              _links = links;
1348 //              _totals = totals;
1349 //          }
1350 //
1351 //          /// <summary>
1352 //          /// <para>
1353 //          /// Calculates this instance.
1354 //          /// </para>
1355 //          /// <para></para>
1356 //          /// </summary>
1357 //          [MethodImpl(MethodImplOptions.AggressiveInlining)]
1358 //          public void Calculate() => _links.Each(_links.Constants.Any,
↵      _links.Constants.Any, CalculateCore);
1359 //          [MethodImpl(MethodImplOptions.AggressiveInlining)]
1360 //          private bool CalculateCore(ulong link)
1361 //          {
1362 //              if (_totals[link] == 0)
1363 //              {
1364 //                  var total = 1UL;
1365 //                  _totals[link] = total;
1366 //                  var visitedChildren = new HashSet<ulong>();
1367 //                  bool linkCalculator(ulong child)
1368 //                  {
1369 //                      if (link != child && visitedChildren.Add(child))
1370 //                      {
1371 //                          total += _totals[child] == 0 ? 1 : _totals[child];
1372 //                      }
1373 //                      return true;
1374 //                  }
1375 //                  _links.Unsync.Each(link, _links.Constants.Any, linkCalculator);
1376 //                  _links.Unsync.Each(_links.Constants.Any, link, linkCalculator);
1377 //                  _totals[link] = total;
1378 //              }
1379 //              return true;
1380 //          }
1381 //      }
1382 //      private class AllUsagesCalculator2
1383 //      {
1384 //          private readonly SynchronizedLinks<ulong> _links;
1385 //          private readonly ulong[] _totals;
1386 //
1387 //          /// <summary>
1388 //          /// <para>
1389 //          /// Initializes a new <see cref="AllUsagesCalculator2"/> instance.
1390 //          /// </para>
1391 //          /// <para></para>
1392 //          /// </summary>
1393 //          /// <param name="links">
1394 //          /// <para>A links.</para>
1395 //          /// <para></para>
1396 //          /// </param>
1397 //          /// <param name="totals">
1398 //          /// <para>A totals.</para>
1399 //          /// <para></para>
1400 //          /// </param>
1401 //          [MethodImpl(MethodImplOptions.AggressiveInlining)]
1402 //          public AllUsagesCalculator2(SynchronizedLinks<ulong> links, ulong[] totals)
1403 //          {
1404 //              _links = links;
1405 //              _totals = totals;
1406 //          }
1407 //
1408 //          /// <summary>
1409 //          /// <para>
1410 //          /// Calculates this instance.
1411 //          /// </para>
1412 //          /// <para></para>
1413 //          /// </summary>
1414 //          [MethodImpl(MethodImplOptions.AggressiveInlining)]
1415 //          public void Calculate() => _links.Each(_links.Constants.Any,
↵      _links.Constants.Any, CalculateCore);
1416 //          [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

1417 // private bool IsElement(ulong link)
1418 // {
1419 //     //_linksInSequence.Contains(link) ||
1420 //     return _links.Unsync.GetTarget(link) == link || _links.Unsync.GetSource(link)
    == link;
1421 // }
1422 // [MethodImpl(MethodImplOptions.AggressiveInlining)]
1423 // private bool CalculateCore(ulong link)
1424 // {
1425 //     // TODO: Проработать защиту от заикливания
1426 //     // Основано на SequenceWalker.WalkLeft
1427 //     Func<ulong, ulong> getSource = _links.Unsync.GetSource;
1428 //     Func<ulong, ulong> getTarget = _links.Unsync.GetTarget;
1429 //     Func<ulong, bool> isElement = IsElement;
1430 //     void visitLeaf(ulong parent)
1431 //     {
1432 //         if (link != parent)
1433 //         {
1434 //             _totals[parent]++;
1435 //         }
1436 //     }
1437 //     void visitNode(ulong parent)
1438 //     {
1439 //         if (link != parent)
1440 //         {
1441 //             _totals[parent]++;
1442 //         }
1443 //     }
1444 //     var stack = new Stack();
1445 //     var element = link;
1446 //     if (isElement(element))
1447 //     {
1448 //         visitLeaf(element);
1449 //     }
1450 //     else
1451 //     {
1452 //         while (true)
1453 //         {
1454 //             if (isElement(element))
1455 //             {
1456 //                 if (stack.Count == 0)
1457 //                 {
1458 //                     break;
1459 //                 }
1460 //                 element = stack.Pop();
1461 //                 var source = getSource(element);
1462 //                 var target = getTarget(element);
1463 //                 // Обработка элемента
1464 //                 if (isElement(target))
1465 //                 {
1466 //                     visitLeaf(target);
1467 //                 }
1468 //                 if (isElement(source))
1469 //                 {
1470 //                     visitLeaf(source);
1471 //                 }
1472 //                 element = source;
1473 //             }
1474 //             else
1475 //             {
1476 //                 stack.Push(element);
1477 //                 visitNode(element);
1478 //                 element = getTarget(element);
1479 //             }
1480 //         }
1481 //     }
1482 //     _totals[link]++;
1483 //     return true;
1484 // }
1485 // }
1486 // private class AllUsagesCollector
1487 // {
1488 //     private readonly ILinks<ulong> _links;
1489 //     private readonly HashSet<ulong> _usages;
1490 //
1491 //     /// <summary>
1492 //     /// <para>
1493 //     /// Initializes a new <see cref="AllUsagesCollector"/> instance.

```

```

1494 //      /// </para>
1495 //      /// <para></para>
1496 //      /// </summary>
1497 //      /// <param name="links">
1498 //      /// <para>A links.</para>
1499 //      /// <para></para>
1500 //      /// </param>
1501 //      /// <param name="usages">
1502 //      /// <para>A usages.</para>
1503 //      /// <para></para>
1504 //      /// </param>
1505 //      [MethodImpl(MethodImplOptions.AggressiveInlining)]
1506 //      public AllUsagesCollector(ILinks<ulong> links, HashSet<ulong> usages)
1507 //      {
1508 //          _links = links;
1509 //          _usages = usages;
1510 //      }
1511 //
1512 //      /// <summary>
1513 //      /// <para>
1514 //      /// Determines whether this instance collect.
1515 //      /// </para>
1516 //      /// <para></para>
1517 //      /// </summary>
1518 //      /// <param name="link">
1519 //      /// <para>The link.</para>
1520 //      /// <para></para>
1521 //      /// </param>
1522 //      /// <returns>
1523 //      /// <para>The bool</para>
1524 //      /// <para></para>
1525 //      /// </returns>
1526 //      [MethodImpl(MethodImplOptions.AggressiveInlining)]
1527 //      public bool Collect(ulong link)
1528 //      {
1529 //          if (_usages.Add(link))
1530 //          {
1531 //              _links.Each(link, _links.Constants.Any, Collect);
1532 //              _links.Each(_links.Constants.Any, link, Collect);
1533 //          }
1534 //          return true;
1535 //      }
1536 //  }
1537 //  private class AllUsagesCollector1
1538 //  {
1539 //      private readonly ILinks<ulong> _links;
1540 //      private readonly HashSet<ulong> _usages;
1541 //      private readonly ulong _continue;
1542 //
1543 //      /// <summary>
1544 //      /// <para>
1545 //      /// Initializes a new <see cref="AllUsagesCollector1"/> instance.
1546 //      /// </para>
1547 //      /// <para></para>
1548 //      /// </summary>
1549 //      /// <param name="links">
1550 //      /// <para>A links.</para>
1551 //      /// <para></para>
1552 //      /// </param>
1553 //      /// <param name="usages">
1554 //      /// <para>A usages.</para>
1555 //      /// <para></para>
1556 //      /// </param>
1557 //      [MethodImpl(MethodImplOptions.AggressiveInlining)]
1558 //      public AllUsagesCollector1(ILinks<ulong> links, HashSet<ulong> usages)
1559 //      {
1560 //          _links = links;
1561 //          _usages = usages;
1562 //          _continue = _links.Constants.Continue;
1563 //      }
1564 //
1565 //      /// <summary>
1566 //      /// <para>
1567 //      /// Collects the link.
1568 //      /// </para>
1569 //      /// <para></para>
1570 //      /// </summary>
1571 //      /// <param name="link">

```

```

1572 //          /// <para>The link.</para>
1573 //          /// <para></para>
1574 //          /// </param>
1575 //          /// <returns>
1576 //          /// <para>The continue.</para>
1577 //          /// <para></para>
1578 //          /// </returns>
1579 //          [MethodImpl(MethodImplOptions.AggressiveInlining)]
1580 //          public ulong Collect(ICollection<ulong> link)
1581 //          {
1582 //              var linkIndex = _links.GetIndex(link);
1583 //              if (_usages.Add(linkIndex))
1584 //              {
1585 //                  _links.Each(Collect, _links.Constants.Any, linkIndex);
1586 //              }
1587 //              return _continue;
1588 //          }
1589 //      }
1590 //      private class AllUsagesCollector2
1591 //      {
1592 //          private readonly ICollection<ulong> _links;
1593 //          private readonly BitString _usages;
1594 //
1595 //          /// <summary>
1596 //          /// <para>
1597 //          /// Initializes a new <see cref="AllUsagesCollector2"/> instance.
1598 //          /// </para>
1599 //          /// <para></para>
1600 //          /// </summary>
1601 //          /// <param name="links">
1602 //          /// <para>A links.</para>
1603 //          /// <para></para>
1604 //          /// </param>
1605 //          /// <param name="usages">
1606 //          /// <para>A usages.</para>
1607 //          /// <para></para>
1608 //          /// </param>
1609 //          [MethodImpl(MethodImplOptions.AggressiveInlining)]
1610 //          public AllUsagesCollector2(ICollection<ulong> links, BitString usages)
1611 //          {
1612 //              _links = links;
1613 //              _usages = usages;
1614 //          }
1615 //
1616 //          /// <summary>
1617 //          /// <para>
1618 //          /// Determines whether this instance collect.
1619 //          /// </para>
1620 //          /// <para></para>
1621 //          /// </summary>
1622 //          /// <param name="link">
1623 //          /// <para>The link.</para>
1624 //          /// <para></para>
1625 //          /// </param>
1626 //          /// <returns>
1627 //          /// <para>The bool</para>
1628 //          /// <para></para>
1629 //          /// </returns>
1630 //          [MethodImpl(MethodImplOptions.AggressiveInlining)]
1631 //          public bool Collect(ulong link)
1632 //          {
1633 //              if (_usages.Add((long)link))
1634 //              {
1635 //                  _links.Each(link, _links.Constants.Any, Collect);
1636 //                  _links.Each(_links.Constants.Any, link, Collect);
1637 //              }
1638 //              return true;
1639 //          }
1640 //      }
1641 //      private class AllUsagesIntersectingCollector
1642 //      {
1643 //          private readonly SynchronizedLinks<ulong> _links;
1644 //          private readonly HashSet<ulong> _intersectWith;
1645 //          private readonly HashSet<ulong> _usages;
1646 //          private readonly HashSet<ulong> _enter;
1647 //
1648 //          /// <summary>
1649 //          /// <para>

```

```

1650 //          /// Initializes a new <see cref="AllUsagesIntersectingCollector"/> instance.
1651 //          /// </para>
1652 //          /// <para></para>
1653 //          /// </summary>
1654 //          /// <param name="links">
1655 //          /// <para>A links.</para>
1656 //          /// <para></para>
1657 //          /// </param>
1658 //          /// <param name="intersectWith">
1659 //          /// <para>A intersect with.</para>
1660 //          /// <para></para>
1661 //          /// </param>
1662 //          /// <param name="usages">
1663 //          /// <para>A usages.</para>
1664 //          /// <para></para>
1665 //          /// </param>
1666 //          [MethodImpl(MethodImplOptions.AggressiveInlining)]
1667 //          public AllUsagesIntersectingCollector(SynchronizedLinks<ulong> links,
↵      HashSet<ulong> intersectWith, HashSet<ulong> usages)
1668 //          {
1669 //              _links = links;
1670 //              _intersectWith = intersectWith;
1671 //              _usages = usages;
1672 //              _enter = new HashSet<ulong>(); // защита от зацикливания
1673 //          }
1674 //
1675 //          /// <summary>
1676 //          /// <para>
1677 //          /// Determines whether this instance collect.
1678 //          /// </para>
1679 //          /// <para></para>
1680 //          /// </summary>
1681 //          /// <param name="link">
1682 //          /// <para>The link.</para>
1683 //          /// <para></para>
1684 //          /// </param>
1685 //          /// <returns>
1686 //          /// <para>The bool</para>
1687 //          /// <para></para>
1688 //          /// </returns>
1689 //          [MethodImpl(MethodImplOptions.AggressiveInlining)]
1690 //          public bool Collect(ulong link)
1691 //          {
1692 //              if (_enter.Add(link))
1693 //              {
1694 //                  if (_intersectWith.Contains(link))
1695 //                  {
1696 //                      _usages.Add(link);
1697 //                  }
1698 //                  _links.Unsync.Each(link, _links.Constants.Any, Collect);
1699 //                  _links.Unsync.Each(_links.Constants.Any, link, Collect);
1700 //              }
1701 //              return true;
1702 //          }
1703 //      }
1704 //      [MethodImpl(MethodImplOptions.AggressiveInlining)]
1705 //      private void CloseInnerConnections(Action<IList<LinkIndex>> handler, ulong left,
↵      ulong right)
1706 //      {
1707 //          TryStepLeftUp(handler, left, right);
1708 //          TryStepRightUp(handler, right, left);
1709 //      }
1710 //      [MethodImpl(MethodImplOptions.AggressiveInlining)]
1711 //      private void AllCloseConnections(Action<IList<LinkIndex>> handler, ulong left, ulong
↵      right)
1712 //      {
1713 //          // Direct
1714 //          if (left == right)
1715 //          {
1716 //              handler(new LinkAddress<LinkIndex>(left));
1717 //          }
1718 //          var doublet = Links.Unsync.SearchOrDefault(left, right);
1719 //          if (doublet != Constants.Null)
1720 //          {
1721 //              handler(new LinkAddress<LinkIndex>(doublet));
1722 //          }
1723 //          // 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>

```

```

1791 //      /// </returns>
1792 //      [MethodImpl(MethodImplOptions.AggressiveInlining)]
1793 //      public HashSet<ulong> MatchPattern(params ulong[] patternSequence)
1794 //      {
1795 //          return _sync.DoRead(() =>
1796 //          {
1797 //              patternSequence = Simplify(patternSequence);
1798 //              if (patternSequence.Length > 0)
1799 //              {
1800 //                  EnsureEachLinkIsAnyOrZeroOrManyOrExists(Links, patternSequence);
1801 //                  var uniqueSequenceElements = new HashSet<ulong>();
1802 //                  for (var i = 0; i < patternSequence.Length; i++)
1803 //                  {
1804 //                      if (patternSequence[i] != Constants.Any && patternSequence[i] !=
ZeroOrMany)
1805 //                      {
1806 //                          uniqueSequenceElements.Add(patternSequence[i]);
1807 //                      }
1808 //                  }
1809 //                  var results = new HashSet<ulong>();
1810 //                  foreach (var uniqueSequenceElement in uniqueSequenceElements)
1811 //                  {
1812 //                      AllUsagesCore(uniqueSequenceElement, results);
1813 //                  }
1814 //                  var filteredResults = new HashSet<ulong>();
1815 //                  var matcher = new PatternMatcher(this, patternSequence, filteredResults);
1816 //                  matcher.AddAllPatternMatchedToResults(results);
1817 //                  return filteredResults;
1818 //              }
1819 //              return new HashSet<ulong>();
1820 //          });
1821 //      }
1822 //
1823 //      // Найти все возможные связи между указанным списком связей.
1824 //      // Находит связи между всеми указанными связями в любом порядке.
1825 //      // TODO: решить что делать с повторами (когда одни и те же элементы встречаются
несколько раз в последовательности)
1826 //      /// <summary>
1827 //      /// <para>
1828 //      /// Gets the all connections using the specified links to connect.
1829 //      /// </para>
1830 //      /// <para></para>
1831 //      /// </summary>
1832 //      /// <param name="linksToConnect">
1833 //      /// <para>The links to connect.</para>
1834 //      /// <para></para>
1835 //      /// </param>
1836 //      /// <returns>
1837 //      /// <para>A hash set of ulong</para>
1838 //      /// <para></para>
1839 //      /// </returns>
1840 //      [MethodImpl(MethodImplOptions.AggressiveInlining)]
1841 //      public HashSet<ulong> GetAllConnections(params ulong[] linksToConnect)
1842 //      {
1843 //          return _sync.DoRead(() =>
1844 //          {
1845 //              var results = new HashSet<ulong>();
1846 //              if (linksToConnect.Length > 0)
1847 //              {
1848 //                  Links.EnsureLinkExists(linksToConnect);
1849 //                  AllUsagesCore(linksToConnect[0], results);
1850 //                  for (var i = 1; i < linksToConnect.Length; i++)
1851 //                  {
1852 //                      var next = new HashSet<ulong>();
1853 //                      AllUsagesCore(linksToConnect[i], next);
1854 //                      results.IntersectWith(next);
1855 //                  }
1856 //              }
1857 //              return results;
1858 //          });
1859 //      }
1860 //
1861 //      /// <summary>
1862 //      /// <para>
1863 //      /// Gets the all connections 1 using the specified links to connect.
1864 //      /// </para>
1865 //      /// <para></para>

```

```

1866 //      /// </summary>
1867 //      /// <param name="linksToConnect">
1868 //      /// <para>The links to connect.</para>
1869 //      /// <para></para>
1870 //      /// </param>
1871 //      /// <returns>
1872 //      /// <para>A hash set of ulong</para>
1873 //      /// <para></para>
1874 //      /// </returns>
1875 //      [MethodImpl(MethodImplOptions.AggressiveInlining)]
1876 //      public HashSet<ulong> GetAllConnections1(params ulong[] linksToConnect)
1877 //      {
1878 //          return _sync.DoRead(() =>
1879 //          {
1880 //              var results = new HashSet<ulong>();
1881 //              if (linksToConnect.Length > 0)
1882 //              {
1883 //                  Links.EnsureLinkExists(linksToConnect);
1884 //                  var collector1 = new AllUsagesCollector(Links.Unsync, results);
1885 //                  collector1.Collect(linksToConnect[0]);
1886 //                  var next = new HashSet<ulong>();
1887 //                  for (var i = 1; i < linksToConnect.Length; i++)
1888 //                  {
1889 //                      var collector = new AllUsagesCollector(Links.Unsync, next);
1890 //                      collector.Collect(linksToConnect[i]);
1891 //                      results.IntersectWith(next);
1892 //                      next.Clear();
1893 //                  }
1894 //              }
1895 //              return results;
1896 //          });
1897 //      }
1898 //
1899 //      /// <summary>
1900 //      /// <para>
1901 //      /// Gets the all connections 2 using the specified links to connect.
1902 //      /// </para>
1903 //      /// <para></para>
1904 //      /// </summary>
1905 //      /// <param name="linksToConnect">
1906 //      /// <para>The links to connect.</para>
1907 //      /// <para></para>
1908 //      /// </param>
1909 //      /// <returns>
1910 //      /// <para>A hash set of ulong</para>
1911 //      /// <para></para>
1912 //      /// </returns>
1913 //      [MethodImpl(MethodImplOptions.AggressiveInlining)]
1914 //      public HashSet<ulong> GetAllConnections2(params ulong[] linksToConnect)
1915 //      {
1916 //          return _sync.DoRead(() =>
1917 //          {
1918 //              var results = new HashSet<ulong>();
1919 //              if (linksToConnect.Length > 0)
1920 //              {
1921 //                  Links.EnsureLinkExists(linksToConnect);
1922 //                  var collector1 = new AllUsagesCollector(Links, results);
1923 //                  collector1.Collect(linksToConnect[0]);
1924 //                  //AllUsagesCore(linksToConnect[0], results);
1925 //                  for (var i = 1; i < linksToConnect.Length; i++)
1926 //                  {
1927 //                      var next = new HashSet<ulong>();
1928 //                      var collector = new AllUsagesIntersectingCollector(Links, results,
1929 //                      next);
1930 //                      collector.Collect(linksToConnect[i]);
1931 //                      //AllUsagesCore(linksToConnect[i], next);
1932 //                      //results.IntersectWith(next);
1933 //                      results = next;
1934 //                  }
1935 //              }
1936 //              return results;
1937 //          });
1938 //      }
1939 //
1940 //      /// <summary>
1941 //      /// <para>
1942 //      /// Gets the all connections 3 using the specified links to connect.

```



```

1942 //      /// </para>
1943 //      /// <para></para>
1944 //      /// </summary>
1945 //      /// <param name="linksToConnect">
1946 //      /// <para>The links to connect.</para>
1947 //      /// <para></para>
1948 //      /// </param>
1949 //      /// <returns>
1950 //      /// <para>A list of ulong</para>
1951 //      /// <para></para>
1952 //      /// </returns>
1953 //      [MethodImpl(MethodImplOptions.AggressiveInlining)]
1954 //      public List<ulong> GetAllConnections3(params ulong[] linksToConnect)
1955 //      {
1956 //          return _sync.DoRead(() =>
1957 //          {
1958 //              var results = new BitString((long)Links.Unsync.Count() + 1); // new
1959 //              BitArray((int)_links.Total + 1);
1960 //              if (linksToConnect.Length > 0)
1961 //              {
1962 //                  Links.EnsureLinkExists(linksToConnect);
1963 //                  var collector1 = new AllUsagesCollector2(Links.Unsync, results);
1964 //                  collector1.Collect(linksToConnect[0]);
1965 //                  for (var i = 1; i < linksToConnect.Length; i++)
1966 //                  {
1967 //                      var next = new BitString((long)Links.Unsync.Count() + 1); //new
1968 //                      BitArray((int)_links.Total + 1);
1969 //                      var collector = new AllUsagesCollector2(Links.Unsync, next);
1970 //                      collector.Collect(linksToConnect[i]);
1971 //                      results = results.And(next);
1972 //                  }
1973 //                  return results.GetSetUInt64Indices();
1974 //              });
1975 //          }
1976 //      [MethodImpl(MethodImplOptions.AggressiveInlining)]
1977 //      private static ulong[] Simplify(ulong[] sequence)
1978 //      {
1979 //          // Считаем новый размер последовательности
1980 //          long newLength = 0;
1981 //          var zeroOrManyStepped = false;
1982 //          for (var i = 0; i < sequence.Length; i++)
1983 //          {
1984 //              if (sequence[i] == ZeroOrMany)
1985 //              {
1986 //                  if (zeroOrManyStepped)
1987 //                  {
1988 //                      continue;
1989 //                  }
1990 //                  zeroOrManyStepped = true;
1991 //              }
1992 //              else
1993 //              {
1994 //                  //if (zeroOrManyStepped) Is it efficient?
1995 //                  zeroOrManyStepped = false;
1996 //              }
1997 //              newLength++;
1998 //          }
1999 //          // Строим новую последовательность
2000 //          zeroOrManyStepped = false;
2001 //          var newSequence = new ulong[newLength];
2002 //          long j = 0;
2003 //          for (var i = 0; i < sequence.Length; i++)
2004 //          {
2005 //              //var current = zeroOrManyStepped;
2006 //              //zeroOrManyStepped = patternSequence[i] == zeroOrMany;
2007 //              //if (current && zeroOrManyStepped)
2008 //              //    continue;
2009 //              //var newZeroOrManyStepped = patternSequence[i] == zeroOrMany;
2010 //              //if (zeroOrManyStepped && newZeroOrManyStepped)
2011 //              //    continue;
2012 //              //zeroOrManyStepped = newZeroOrManyStepped;
2013 //              if (sequence[i] == ZeroOrMany)
2014 //              {
2015 //                  if (zeroOrManyStepped)
2016 //                  {
2017 //                      continue;
2018 //                  }
2019 //                  newSequence[j] = sequence[i];
2020 //                  j++;
2021 //              }
2022 //          }
2023 //          return newSequence;
2024 //      }
2025 //      }
2026 //      }
2027 //      }
2028 //      }
2029 //      }
2030 //      }
2031 //      }
2032 //      }
2033 //      }
2034 //      }
2035 //      }
2036 //      }
2037 //      }
2038 //      }
2039 //      }
2040 //      }
2041 //      }
2042 //      }
2043 //      }
2044 //      }
2045 //      }
2046 //      }
2047 //      }
2048 //      }
2049 //      }
2050 //      }
2051 //      }
2052 //      }
2053 //      }
2054 //      }
2055 //      }
2056 //      }
2057 //      }
2058 //      }
2059 //      }
2060 //      }
2061 //      }
2062 //      }
2063 //      }
2064 //      }
2065 //      }
2066 //      }
2067 //      }
2068 //      }
2069 //      }
2070 //      }
2071 //      }
2072 //      }
2073 //      }
2074 //      }
2075 //      }
2076 //      }
2077 //      }
2078 //      }
2079 //      }
2080 //      }
2081 //      }
2082 //      }
2083 //      }
2084 //      }
2085 //      }
2086 //      }
2087 //      }
2088 //      }
2089 //      }
2090 //      }
2091 //      }
2092 //      }
2093 //      }
2094 //      }
2095 //      }
2096 //      }
2097 //      }
2098 //      }
2099 //      }
2100 //      }
2101 //      }
2102 //      }
2103 //      }
2104 //      }
2105 //      }
2106 //      }
2107 //      }
2108 //      }
2109 //      }
2110 //      }
2111 //      }
2112 //      }
2113 //      }
2114 //      }
2115 //      }
2116 //      }
2117 //      }
2118 //      }
2119 //      }
2120 //      }
2121 //      }
2122 //      }
2123 //      }
2124 //      }
2125 //      }
2126 //      }
2127 //      }
2128 //      }
2129 //      }
2130 //      }
2131 //      }
2132 //      }
2133 //      }
2134 //      }
2135 //      }
2136 //      }
2137 //      }
2138 //      }
2139 //      }
2140 //      }
2141 //      }
2142 //      }
2143 //      }
2144 //      }
2145 //      }
2146 //      }
2147 //      }
2148 //      }
2149 //      }
2150 //      }
2151 //      }
2152 //      }
2153 //      }
2154 //      }
2155 //      }
2156 //      }
2157 //      }
2158 //      }
2159 //      }
2160 //      }
2161 //      }
2162 //      }
2163 //      }
2164 //      }
2165 //      }
2166 //      }
2167 //      }
2168 //      }
2169 //      }
2170 //      }
2171 //      }
2172 //      }
2173 //      }
2174 //      }
2175 //      }
2176 //      }
2177 //      }
2178 //      }
2179 //      }
2180 //      }
2181 //      }
2182 //      }
2183 //      }
2184 //      }
2185 //      }
2186 //      }
2187 //      }
2188 //      }
2189 //      }
2190 //      }
2191 //      }
2192 //      }
2193 //      }
2194 //      }
2195 //      }
2196 //      }
2197 //      }
2198 //      }
2199 //      }
2200 //      }
2201 //      }
2202 //      }
2203 //      }
2204 //      }
2205 //      }
2206 //      }
2207 //      }
2208 //      }
2209 //      }
2210 //      }
2211 //      }
2212 //      }
2213 //      }
2214 //      }
2215 //      }
2216 //      }
2217 //      }
2218 //      }
2219 //      }
2220 //      }
2221 //      }
2222 //      }
2223 //      }
2224 //      }
2225 //      }
2226 //      }
2227 //      }
2228 //      }
2229 //      }
2230 //      }
2231 //      }
2232 //      }
2233 //      }
2234 //      }
2235 //      }
2236 //      }
2237 //      }
2238 //      }
2239 //      }
2240 //      }
2241 //      }
2242 //      }
2243 //      }
2244 //      }
2245 //      }
2246 //      }
2247 //      }
2248 //      }
2249 //      }
2250 //      }
2251 //      }
2252 //      }
2253 //      }
2254 //      }
2255 //      }
2256 //      }
2257 //      }
2258 //      }
2259 //      }
2260 //      }
2261 //      }
2262 //      }
2263 //      }
2264 //      }
2265 //      }
2266 //      }
2267 //      }
2268 //      }
2269 //      }
2270 //      }
2271 //      }
2272 //      }
2273 //      }
2274 //      }
2275 //      }
2276 //      }
2277 //      }
2278 //      }
2279 //      }
2280 //      }
2281 //      }
2282 //      }
2283 //      }
2284 //      }
2285 //      }
2286 //      }
2287 //      }
2288 //      }
2289 //      }
2290 //      }
2291 //      }
2292 //      }
2293 //      }
2294 //      }
2295 //      }
2296 //      }
2297 //      }
2298 //      }
2299 //      }
2300 //      }
2301 //      }
2302 //      }
2303 //      }
2304 //      }
2305 //      }
2306 //      }
2307 //      }
2308 //      }
2309 //      }
2310 //      }
2311 //      }
2312 //      }
2313 //      }
2314 //      }
2315 //      }
2316 //      }
2317 //      }
2318 //      }
2319 //      }
2320 //      }
2321 //      }
2322 //      }
2323 //      }
2324 //      }
2325 //      }
2326 //      }
2327 //      }
2328 //      }
2329 //      }
2330 //      }
2331 //      }
2332 //      }
2333 //      }
2334 //      }
2335 //      }
2336 //      }
2337 //      }
2338 //      }
2339 //      }
2340 //      }
2341 //      }
2342 //      }
2343 //      }
2344 //      }
2345 //      }
2346 //      }
2347 //      }
2348 //      }
2349 //      }
2350 //      }
2351 //      }
2352 //      }
2353 //      }
2354 //      }
2355 //      }
2356 //      }
2357 //      }
2358 //      }
2359 //      }
2360 //      }
2361 //      }
2362 //      }
2363 //      }
2364 //      }
2365 //      }
2366 //      }
2367 //      }
2368 //      }
2369 //      }
2370 //      }
2371 //      }
2372 //      }
2373 //      }
2374 //      }
2375 //      }
2376 //      }
2377 //      }
2378 //      }
2379 //      }
2380 //      }
2381 //      }
2382 //      }
2383 //      }
2384 //     
```

```

2017 // }
2018 //     zeroOrManyStepped = true;
2019 // }
2020 // else
2021 // {
2022 //     //if (zeroOrManyStepped) Is it efficient?
2023 //     zeroOrManyStepped = false;
2024 // }
2025 //     newSequence[j++] = sequence[i];
2026 // }
2027 // return newSequence;
2028 // }
2029 //
2030 // /// <summary>
2031 // /// <para>
2032 // /// Tests the simplify.
2033 // /// </para>
2034 // /// <para></para>
2035 // /// </summary>
2036 // [MethodImpl(MethodImplOptions.AggressiveInlining)]
2037 // public static void TestSimplify()
2038 // {
2039 //     var sequence = new ulong[] { ZeroOrMany, ZeroOrMany, 2, 3, 4, ZeroOrMany,
↵ ZeroOrMany, ZeroOrMany, 4, ZeroOrMany, ZeroOrMany, ZeroOrMany };
2040 //     var simplifiedSequence = Simplify(sequence);
2041 // }
2042 //
2043 // /// <summary>
2044 // /// <para>
2045 // /// Gets the similar sequences.
2046 // /// </para>
2047 // /// <para></para>
2048 // /// </summary>
2049 // /// <returns>
2050 // /// <para>A list of ulong</para>
2051 // /// <para></para>
2052 // /// </returns>
2053 // [MethodImpl(MethodImplOptions.AggressiveInlining)]
2054 // public List<ulong> GetSimilarSequences() => new List<ulong>();
2055 //
2056 // /// <summary>
2057 // /// <para>
2058 // /// Predictions this instance.
2059 // /// </para>
2060 // /// <para></para>
2061 // /// </summary>
2062 // [MethodImpl(MethodImplOptions.AggressiveInlining)]
2063 // public void Prediction()
2064 // {
2065 //     //_links
2066 //     //sequences
2067 // }
2068 //
2069 // #region From Triplets
2070 //
2071 // //public static void DeleteSequence(Link sequence)
2072 // //{
2073 // //}
2074 //
2075 // /// <summary>
2076 // /// <para>
2077 // /// Collects the matching sequences using the specified links.
2078 // /// </para>
2079 // /// <para></para>
2080 // /// </summary>
2081 // /// <param name="links">
2082 // /// <para>The links.</para>
2083 // /// <para></para>
2084 // /// </param>
2085 // /// <exception cref="InvalidOperationException">
2086 // /// <para>Подпоследовательности с одним элементом не поддерживаются.</para>
2087 // /// <para></para>
2088 // /// </exception>
2089 // /// <returns>
2090 // /// <para>The results.</para>
2091 // /// <para></para>
2092 // /// </returns>

```

```

2093 // [MethodImpl(MethodImplOptions.AggressiveInlining)]
2094 // public List<ulong> CollectMatchingSequences(ulong[] links)
2095 // {
2096 //     if (links.Length == 1)
2097 //     {
2098 //         throw new InvalidOperationException("Подпоследовательности с одним элементом
↵ не поддерживаются.");
2099 //     }
2100 //     var leftBound = 0;
2101 //     var rightBound = links.Length - 1;
2102 //     var left = links[leftBound++];
2103 //     var right = links[rightBound--];
2104 //     var results = new List<ulong>();
2105 //     CollectMatchingSequences(left, leftBound, links, right, rightBound, ref results);
2106 //     return results;
2107 // }
2108 // [MethodImpl(MethodImplOptions.AggressiveInlining)]
2109 // private void CollectMatchingSequences(ulong leftLink, int leftBound, ulong[]
↵ middleLinks, ulong rightLink, int rightBound, ref List<ulong> results)
2110 // {
2111 //     var leftLinkTotalReferers = Links.Unsync.Count(leftLink);
2112 //     var rightLinkTotalReferers = Links.Unsync.Count(rightLink);
2113 //     if (leftLinkTotalReferers <= rightLinkTotalReferers)
2114 //     {
2115 //         var nextLeftLink = middleLinks[leftBound];
2116 //         var elements = GetRightElements(leftLink, nextLeftLink);
2117 //         if (leftBound <= rightBound)
2118 //         {
2119 //             for (var i = elements.Length - 1; i >= 0; i--)
2120 //             {
2121 //                 var element = elements[i];
2122 //                 if (element != 0)
2123 //                 {
2124 //                     CollectMatchingSequences(element, leftBound + 1, middleLinks,
↵ rightLink, rightBound, ref results);
2125 //                 }
2126 //             }
2127 //         }
2128 //         else
2129 //         {
2130 //             for (var i = elements.Length - 1; i >= 0; i--)
2131 //             {
2132 //                 var element = elements[i];
2133 //                 if (element != 0)
2134 //                 {
2135 //                     results.Add(element);
2136 //                 }
2137 //             }
2138 //         }
2139 //     }
2140 //     else
2141 //     {
2142 //         var nextRightLink = middleLinks[rightBound];
2143 //         var elements = GetLeftElements(rightLink, nextRightLink);
2144 //         if (leftBound <= rightBound)
2145 //         {
2146 //             for (var i = elements.Length - 1; i >= 0; i--)
2147 //             {
2148 //                 var element = elements[i];
2149 //                 if (element != 0)
2150 //                 {
2151 //                     CollectMatchingSequences(leftLink, leftBound, middleLinks,
↵ elements[i], rightBound - 1, ref results);
2152 //                 }
2153 //             }
2154 //         }
2155 //         else
2156 //         {
2157 //             for (var i = elements.Length - 1; i >= 0; i--)
2158 //             {
2159 //                 var element = elements[i];
2160 //                 if (element != 0)
2161 //                 {
2162 //                     results.Add(element);
2163 //                 }
2164 //             }
2165 //         }

```

```

2166 //      }
2167 //    }
2168 //
2169 //    /// <summary>
2170 //    /// <para>
2171 //    /// Gets the right elements using the specified start link.
2172 //    /// </para>
2173 //    /// <para></para>
2174 //    /// </summary>
2175 //    /// <param name="startLink">
2176 //    /// <para>The start link.</para>
2177 //    /// <para></para>
2178 //    /// </param>
2179 //    /// <param name="rightLink">
2180 //    /// <para>The right link.</para>
2181 //    /// <para></para>
2182 //    /// </param>
2183 //    /// <returns>
2184 //    /// <para>The result.</para>
2185 //    /// <para></para>
2186 //    /// </returns>
2187 //    [MethodImpl(MethodImplOptions.AggressiveInlining)]
2188 //    public ulong[] GetRightElements(ulong startLink, ulong rightLink)
2189 //    {
2190 //        var result = new ulong[5];
2191 //        TryStepRight(startLink, rightLink, result, 0);
2192 //        Links.Each(Constants.Any, startLink, couple =>
2193 //        {
2194 //            if (couple != startLink)
2195 //            {
2196 //                if (TryStepRight(couple, rightLink, result, 2))
2197 //                {
2198 //                    return false;
2199 //                }
2200 //            }
2201 //            return true;
2202 //        });
2203 //        if (Links.GetTarget(Links.GetTarget(startLink)) == rightLink)
2204 //        {
2205 //            result[4] = startLink;
2206 //        }
2207 //        return result;
2208 //    }
2209 //
2210 //    /// <summary>
2211 //    /// <para>
2212 //    /// Determines whether this instance try step right.
2213 //    /// </para>
2214 //    /// <para></para>
2215 //    /// </summary>
2216 //    /// <param name="startLink">
2217 //    /// <para>The start link.</para>
2218 //    /// <para></para>
2219 //    /// </param>
2220 //    /// <param name="rightLink">
2221 //    /// <para>The right link.</para>
2222 //    /// <para></para>
2223 //    /// </param>
2224 //    /// <param name="result">
2225 //    /// <para>The result.</para>
2226 //    /// <para></para>
2227 //    /// </param>
2228 //    /// <param name="offset">
2229 //    /// <para>The offset.</para>
2230 //    /// <para></para>
2231 //    /// </param>
2232 //    /// <returns>
2233 //    /// <para>The bool</para>
2234 //    /// <para></para>
2235 //    /// </returns>
2236 //    [MethodImpl(MethodImplOptions.AggressiveInlining)]
2237 //    public bool TryStepRight(ulong startLink, ulong rightLink, ulong[] result, int offset)
2238 //    {
2239 //        var added = 0;
2240 //        Links.Each(startLink, Constants.Any, couple =>
2241 //        {
2242 //            if (couple != startLink)
2243 //            {

```

```

2244 //         var coupleTarget = Links.GetTarget(couple);
2245 //         if (coupleTarget == rightLink)
2246 //         {
2247 //             result[offset] = couple;
2248 //             if (++added == 2)
2249 //             {
2250 //                 return false;
2251 //             }
2252 //         }
2253 //         else if (Links.GetSource(coupleTarget) == rightLink) //
↪ coupleTarget.Linker == Net.And &&
2254 //         {
2255 //             result[offset + 1] = couple;
2256 //             if (++added == 2)
2257 //             {
2258 //                 return false;
2259 //             }
2260 //         }
2261 //     }
2262 //     return true;
2263 // });
2264 //     return added > 0;
2265 // }
2266 //
2267 //     /// <summary>
2268 //     /// <para>
2269 //     /// Gets the left elements using the specified start link.
2270 //     /// </para>
2271 //     /// <para></para>
2272 //     /// </summary>
2273 //     /// <param name="startLink">
2274 //     /// <para>The start link.</para>
2275 //     /// <para></para>
2276 //     /// </param>
2277 //     /// <param name="leftLink">
2278 //     /// <para>The left link.</para>
2279 //     /// <para></para>
2280 //     /// </param>
2281 //     /// <returns>
2282 //     /// <para>The result.</para>
2283 //     /// <para></para>
2284 //     /// </returns>
2285 //     [MethodImpl(MethodImplOptions.AggressiveInlining)]
2286 //     public ulong[] GetLeftElements(ulong startLink, ulong leftLink)
2287 //     {
2288 //         var result = new ulong[5];
2289 //         TryStepLeft(startLink, leftLink, result, 0);
2290 //         Links.Each(startLink, Constants.Any, couple =>
2291 //         {
2292 //             if (couple != startLink)
2293 //             {
2294 //                 if (TryStepLeft(couple, leftLink, result, 2))
2295 //                 {
2296 //                     return false;
2297 //                 }
2298 //             }
2299 //             return true;
2300 //         });
2301 //         if (Links.GetSource(Links.GetSource(leftLink)) == startLink)
2302 //         {
2303 //             result[4] = leftLink;
2304 //         }
2305 //         return result;
2306 //     }
2307 //
2308 //     /// <summary>
2309 //     /// <para>
2310 //     /// Determines whether this instance try step left.
2311 //     /// </para>
2312 //     /// <para></para>
2313 //     /// </summary>
2314 //     /// <param name="startLink">
2315 //     /// <para>The start link.</para>
2316 //     /// <para></para>
2317 //     /// </param>
2318 //     /// <param name="leftLink">
2319 //     /// <para>The left link.</para>
2320 //     /// <para></para>

```

```

2321 //      /// </param>
2322 //      /// <param name="result">
2323 //      /// <para>The result.</para>
2324 //      /// <para></para>
2325 //      /// </param>
2326 //      /// <param name="offset">
2327 //      /// <para>The offset.</para>
2328 //      /// <para></para>
2329 //      /// </param>
2330 //      /// <returns>
2331 //      /// <para>The bool</para>
2332 //      /// <para></para>
2333 //      /// </returns>
2334 //      [MethodImpl(MethodImplOptions.AggressiveInlining)]
2335 //      public bool TryStepLeft(ulong startLink, ulong leftLink, ulong[] result, int offset)
2336 //      {
2337 //          var added = 0;
2338 //          Links.Each(Constants.Any, startLink, couple =>
2339 //          {
2340 //              if (couple != startLink)
2341 //              {
2342 //                  var coupleSource = Links.GetSource(couple);
2343 //                  if (coupleSource == leftLink)
2344 //                  {
2345 //                      result[offset] = couple;
2346 //                      if (++added == 2)
2347 //                      {
2348 //                          return false;
2349 //                      }
2350 //                      else if (Links.GetTarget(coupleSource) == leftLink) //
2351 //                      ↪ coupleSource.Linker == Net.And &&
2352 //                      {
2353 //                          result[offset + 1] = couple;
2354 //                          if (++added == 2)
2355 //                          {
2356 //                              return false;
2357 //                          }
2358 //                      }
2359 //                  }
2360 //                  return true;
2361 //              });
2362 //              return added > 0;
2363 //          }
2364 //      }
2365 //      #endregion
2366 //
2367 //      #region Walkers
2368 //
2369 //      /// <summary>
2370 //      /// <para>
2371 //      /// Represents the pattern matcher.
2372 //      /// </para>
2373 //      /// <para></para>
2374 //      /// </summary>
2375 //      /// <seealso cref="RightSequenceWalker{ulong}" />
2376 //      public class PatternMatcher : RightSequenceWalker<ulong>
2377 //      {
2378 //          private readonly Sequences _sequences;
2379 //          private readonly ulong[] _patternSequence;
2380 //          private readonly HashSet<LinkIndex> _linksInSequence;
2381 //          private readonly HashSet<LinkIndex> _results;
2382 //
2383 //          #region Pattern Match
2384 //
2385 //          /// <summary>
2386 //          /// <para>
2387 //          /// The pattern block type enum.
2388 //          /// </para>
2389 //          /// <para></para>
2390 //          /// </summary>
2391 //          enum PatternBlockType
2392 //          {
2393 //              /// <summary>
2394 //              /// <para>
2395 //              /// The undefined pattern block type.
2396 //              /// </para>
2397 //              /// <para></para>

```

```

2398 //          /// </summary>
2399 //          Undefined,
2400 //          /// <summary>
2401 //          /// <para>
2402 //          /// The gap pattern block type.
2403 //          /// </para>
2404 //          /// <para></para>
2405 //          /// </summary>
2406 //          Gap,
2407 //          /// <summary>
2408 //          /// <para>
2409 //          /// The elements pattern block type.
2410 //          /// </para>
2411 //          /// <para></para>
2412 //          /// </summary>
2413 //          Elements
2414 //      }
2415 //
2416 //      /// <summary>
2417 //      /// <para>
2418 //      /// The pattern block.
2419 //      /// </para>
2420 //      /// <para></para>
2421 //      /// </summary>
2422 //      struct PatternBlock
2423 //      {
2424 //          /// <summary>
2425 //          /// <para>
2426 //          /// The type.
2427 //          /// </para>
2428 //          /// <para></para>
2429 //          /// </summary>
2430 //          public PatternBlockType Type;
2431 //          /// <summary>
2432 //          /// <para>
2433 //          /// The start.
2434 //          /// </para>
2435 //          /// <para></para>
2436 //          /// </summary>
2437 //          public long Start;
2438 //          /// <summary>
2439 //          /// <para>
2440 //          /// The stop.
2441 //          /// </para>
2442 //          /// <para></para>
2443 //          /// </summary>
2444 //          public long Stop;
2445 //      }
2446 //      private readonly List<PatternBlock> _pattern;
2447 //      private int _patternPosition;
2448 //      private long _sequencePosition;
2449 //
2450 //      #endregion
2451 //
2452 //      /// <summary>
2453 //      /// <para>
2454 //      /// Initializes a new <see cref="PatternMatcher"/> instance.
2455 //      /// </para>
2456 //      /// <para></para>
2457 //      /// </summary>
2458 //      /// <param name="sequences">
2459 //      /// <para>A sequences.</para>
2460 //      /// <para></para>
2461 //      /// </param>
2462 //      /// <param name="patternSequence">
2463 //      /// <para>A pattern sequence.</para>
2464 //      /// <para></para>
2465 //      /// </param>
2466 //      /// <param name="results">
2467 //      /// <para>A results.</para>
2468 //      /// <para></para>
2469 //      /// </param>
2470 //      [MethodImpl(MethodImplOptions.AggressiveInlining)]
2471 //      public PatternMatcher(Sequences sequences, LinkIndex[] patternSequence,
2472 //      ↪      HashSet<LinkIndex> results)
2473 //      : base(sequences.Links.Unsync, new DefaultStack<ulong>())
2474 //      {

```

```

2474 //         _sequences = sequences;
2475 //         _patternSequence = patternSequence;
2476 //         _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
↪ _sequences.Constants.Any && x != ZeroOrMany));
2477 //         _results = results;
2478 //         _pattern = CreateDetailedPattern();
2479 //     }
2480 //
2481 //     /// <summary>
2482 //     /// <para>
2483 //     /// Determines whether this instance is element.
2484 //     /// </para>
2485 //     /// <para></para>
2486 //     /// </summary>
2487 //     /// <param name="link">
2488 //     /// <para>The link.</para>
2489 //     /// <para></para>
2490 //     /// </param>
2491 //     /// <returns>
2492 //     /// <para>The bool</para>
2493 //     /// <para></para>
2494 //     /// </returns>
2495 //     [MethodImpl(MethodImplOptions.AggressiveInlining)]
2496 //     protected override bool IsElement(ulong link) => _linksInSequence.Contains(link)
↪ || base.IsElement(link);
2497 //
2498 //     /// <summary>
2499 //     /// <para>
2500 //     /// Determines whether this instance pattern match.
2501 //     /// </para>
2502 //     /// <para></para>
2503 //     /// </summary>
2504 //     /// <param name="sequenceToMatch">
2505 //     /// <para>The sequence to match.</para>
2506 //     /// <para></para>
2507 //     /// </param>
2508 //     /// <returns>
2509 //     /// <para>The bool</para>
2510 //     /// <para></para>
2511 //     /// </returns>
2512 //     [MethodImpl(MethodImplOptions.AggressiveInlining)]
2513 //     public bool PatternMatch(LinkIndex sequenceToMatch)
2514 //     {
2515 //         _patternPosition = 0;
2516 //         _sequencePosition = 0;
2517 //         foreach (var part in Walk(sequenceToMatch))
2518 //         {
2519 //             if (!PatternMatchCore(part))
2520 //             {
2521 //                 break;
2522 //             }
2523 //         }
2524 //         return _patternPosition == _pattern.Count || (_patternPosition ==
↪ _pattern.Count - 1 && _pattern[_patternPosition].Start == 0);
2525 //     }
2526 //     [MethodImpl(MethodImplOptions.AggressiveInlining)]
2527 //     private List<PatternBlock> CreateDetailedPattern()
2528 //     {
2529 //         var pattern = new List<PatternBlock>();
2530 //         var patternBlock = new PatternBlock();
2531 //         for (var i = 0; i < _patternSequence.Length; i++)
2532 //         {
2533 //             if (patternBlock.Type == PatternBlockType.Undefined)
2534 //             {
2535 //                 if (_patternSequence[i] == _sequences.Constants.Any)
2536 //                 {
2537 //                     patternBlock.Type = PatternBlockType.Gap;
2538 //                     patternBlock.Start = 1;
2539 //                     patternBlock.Stop = 1;
2540 //                 }
2541 //                 else if (_patternSequence[i] == ZeroOrMany)
2542 //                 {
2543 //                     patternBlock.Type = PatternBlockType.Gap;
2544 //                     patternBlock.Start = 0;
2545 //                     patternBlock.Stop = long.MaxValue;
2546 //                 }
2547 //                 else

```



```

2548 //
2549 //
2550 //
2551 //
2552 //
2553 //
2554 //
2555 //
2556 //
2557 //
2558 //
2559 //
2560 //
2561 //
2562 //
2563 //
2564 //
2565 //
2566 //
2567 //
2568 //
2569 //
2570 //
2571 //
2572 //
2573 //
2574 //
2575 //
2576 //
2577 //
2578 //
2579 //
2580 //
2581 //
2582 //
2583 //
2584 //
2585 //
2586 //
2587 //
2588 //
2589 //
2590 //
2591 //
2592 //
2593 //
2594 //
2595 //
2596 //
2597 //
2598 //
2599 //
2600 //
2601 //
2602 //
2603 //
2604 //
2605 //
2606 //
2607 //
2608 //
2609 //
2610 //
2611 //
2612 //
2613 //
2614 //
2615 //
2616 //
2617 //
2618 //
2619 //
2620 //
2621 //
2622 //
2623 //
2624 //
2625 //

```

```

{
    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)
//{

```

```

2626 // // if (regexp[0] == '\0')
2627 // // return 1;
2628 // // if (regexp[1] == '*')
2629 // // return matchstar(regexp[0], regexp + 2, text);
2630 // // if (regexp[0] == '$' && regexp[1] == '\0')
2631 // // return *text == '\0';
2632 // // if (*text != '\0' && (regexp[0] == '.' || regexp[0] == *text))
2633 // // return matchhere(regexp + 1, text + 1);
2634 // // return 0;
2635 // //}
2636 //
2637 // // matchstar: search for c*regexp at beginning of text
2638 // //int matchstar(int c, char* regexp, char* text)
2639 // //{
2640 // // do
2641 // // { /* a * matches zero or more instances */
2642 // // if (matchhere(regexp, text))
2643 // // return 1;
2644 // // } while (*text != '\0' && (*text++ == c || c == '.'));
2645 // // return 0;
2646 // //}
2647 //
2648 // //private void GetNextPatternElement(out LinkIndex element, out long mininumGap,
↵ out long maximumGap)
2649 // //{
2650 // // mininumGap = 0;
2651 // // maximumGap = 0;
2652 // // element = 0;
2653 // // for (; _patternPosition < _patternSequence.Length; _patternPosition++)
2654 // // {
2655 // // if (_patternSequence[_patternPosition] == Doublets.Links.Null)
2656 // // mininumGap++;
2657 // // else if (_patternSequence[_patternPosition] == ZeroOrMany)
2658 // // maximumGap = long.MaxValue;
2659 // // else
2660 // // break;
2661 // // }
2662 // //
2663 // // if (maximumGap < mininumGap)
2664 // // maximumGap = mininumGap;
2665 // //}
2666 // [MethodImpl(MethodImplOptions.AggressiveInlining)]
2667 // private bool PatternMatchCore(LinkIndex element)
2668 // {
2669 // if (_patternPosition >= _pattern.Count)
2670 // {
2671 // _patternPosition = -2;
2672 // return false;
2673 // }
2674 // var currentPatternBlock = _pattern[_patternPosition];
2675 // if (currentPatternBlock.Type == PatternBlockType.Gap)
2676 // {
2677 // //var currentMatchingBlockLength = (_sequencePosition -
↵ _lastMatchedBlockPosition);
2678 // if (_sequencePosition < currentPatternBlock.Start)
2679 // {
2680 // _sequencePosition++;
2681 // return true; // Двигаемся дальше
2682 // }
2683 // // Это последний блок
2684 // if (_pattern.Count == _patternPosition + 1)
2685 // {
2686 // _patternPosition++;
2687 // _sequencePosition = 0;
2688 // return false; // Полное соответствие
2689 // }
2690 // else
2691 // {
2692 // if (_sequencePosition > currentPatternBlock.Stop)
2693 // {
2694 // return false; // Соответствие невозможно
2695 // }
2696 // var nextPatternBlock = _pattern[_patternPosition + 1];
2697 // if (_patternSequence[nextPatternBlock.Start] == element)
2698 // {
2699 // if (nextPatternBlock.Start < nextPatternBlock.Stop)
2700 // {

```

```

2701 //                                     _patternPosition++;
2702 //                                     _sequencePosition = 1;
2703 //                                     }
2704 //                                     else
2705 //                                     {
2706 //                                     _patternPosition += 2;
2707 //                                     _sequencePosition = 0;
2708 //                                     }
2709 //                                     }
2710 //                                     }
2711 //                                     }
2712 //                                     else // currentPatternBlock.Type == PatternBlockType.Elements
2713 //                                     {
2714 //                                     var patternElementPosition = currentPatternBlock.Start +
↪ _sequencePosition;
2715 //                                     if (_patternSequence[patternElementPosition] != element)
2716 //                                     {
2717 //                                     return false; // Соответствие невозможно
2718 //                                     }
2719 //                                     if (patternElementPosition == currentPatternBlock.Stop)
2720 //                                     {
2721 //                                     _patternPosition++;
2722 //                                     _sequencePosition = 0;
2723 //                                     }
2724 //                                     else
2725 //                                     {
2726 //                                     _sequencePosition++;
2727 //                                     }
2728 //                                     }
2729 //                                     return true;
2730 //                                     //if (_patternSequence[_patternPosition] != element)
2731 //                                     //    return false;
2732 //                                     //else
2733 //                                     //{
2734 //                                     //    _sequencePosition++;
2735 //                                     //    _patternPosition++;
2736 //                                     //    return true;
2737 //                                     //}
2738 //                                     //if (_filterPosition == _patternSequence.Length)
2739 //                                     //{
2740 //                                     //    _filterPosition = -2; // Длиннее чем нужно
2741 //                                     //    return false;
2742 //                                     //}
2743 //                                     //if (element != _patternSequence[_filterPosition])
2744 //                                     //{
2745 //                                     //    _filterPosition = -1;
2746 //                                     //    return false; // Начинается иначе
2747 //                                     //}
2748 //                                     //filterPosition++;
2749 //                                     //if (_filterPosition == (_patternSequence.Length - 1))
2750 //                                     //    return false;
2751 //                                     //if (_filterPosition >= 0)
2752 //                                     //{
2753 //                                     //    if (element == _patternSequence[_filterPosition + 1])
2754 //                                     //        _filterPosition++;
2755 //                                     //    else
2756 //                                     //        return false;
2757 //                                     //}
2758 //                                     //if (_filterPosition < 0)
2759 //                                     //{
2760 //                                     //    if (element == _patternSequence[0])
2761 //                                     //        _filterPosition = 0;
2762 //                                     //}
2763 //                                     }
2764 //                                     }
2765 //                                     /// <summary>
2766 //                                     /// <para>
2767 //                                     /// Adds the all pattern matched to results using the specified sequences to
2768 ↪ match.
2769 //                                     /// </para>
2770 //                                     /// <para></para>
2771 //                                     /// </summary>
2772 //                                     /// <param name="sequencesToMatch">
2773 //                                     /// <para>The sequences to match.</para>
2774 //                                     /// <para></para>
2775 //                                     /// </param>
2776 //                                     [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

2777 //         public void AddAllPatternMatchedToResults(IEnumerable<ulong> sequencesToMatch)
2778 //         {
2779 //             foreach (var sequenceToMatch in sequencesToMatch)
2780 //             {
2781 //                 if (PatternMatch(sequenceToMatch))
2782 //                 {
2783 //                     _results.Add(sequenceToMatch);
2784 //                 }
2785 //             }
2786 //         }
2787 //     }
2788 //
2789 //     #endregion
2790 // }
2791 // }

```

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

```

1  // using System;
2  // using System.Collections.Generic;
3  // using System.Linq;
4  // using System.Runtime.CompilerServices;
5  // using Platform.Collections;
6  // using Platform.Collections.Lists;
7  // using Platform.Collections.Stacks;
8  // using Platform.Threading.Synchronization;
9  // using Platform.Data.Doublets.Sequences.Walkers;
10 // using Platform.Delegates;
11 // using LinkIndex = System.UInt64;
12 //
13 // #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
14 //
15 // namespace Platform.Data.Doublets.Sequences
16 // {
17 //     /// <summary>
18 //     /// Представляет коллекцию последовательностей связей.
19 //     /// </summary>
20 //     /// <remarks>
21 //     /// Обязательно реализовать атомарность каждого публичного метода.
22 //     ///
23 //     /// TODO:
24 //     ///
25 //     /// !!! Повышение вероятности повторного использования групп (подпоследовательностей),
26 //     /// через естественную группировку по unicode типам, все whitespace вместе, все символы
27 //     /// вместе, все числа вместе и т.п.
28 //     /// + использовать ровно сбалансированный вариант, чтобы уменьшать вложенность (глубину
29 //     /// графа)
30 //     ///
31 //     /// x*y - найти все связи между, в последовательностях любой формы, если не стоит
32 //     /// ограничитель на то, что является последовательностью, а что нет,
33 //     /// то находятся любые структуры связей, которые содержат эти элементы именно в таком
34 //     /// порядке.
35 //     ///
36 //     /// Рост последовательности слева и справа.
37 //     /// Поиск со звёздочкой.
38 //     /// URL, PURL - реестр используемых во вне ссылок на ресурсы,
39 //     /// так же проблема может быть решена при реализации дистанционных триггеров.
40 //     /// Нужны ли уникальные указатели вообще?
41 //     /// Что если обращение к информации будет происходить через содержимое всегда?
42 //     ///
43 //     /// Писать тесты.
44 //     ///
45 //     ///
46 //     /// Можно убрать зависимость от конкретной реализации Links,
47 //     /// на зависимость от абстрактного элемента, который может быть представлен несколькими
48 //     /// способами.
49 //     ///
50 //     /// Можно ли как-то сделать один общий интерфейс
51 //     ///
52 //     ///
53 //     /// Блокчейн и/или гит для распределённой записи транзакций.
54 //     ///
55 //     /// </remarks>
56 //     public partial class Sequences : ILinks<LinkIndex> // IList<string>, IList<LinkIndex[]>
57 //     (после завершения реализации Sequences)
58 //     {
59 //         /// <summary>Возвращает значение LinkIndex, обозначающее любое количество
60 //         /// связей.</summary>
61 //         public const LinkIndex ZeroOrMany = LinkIndex.MaxValue;

```

```

55 //
56 //      /// <summary>
57 //      /// <para>
58 //      /// Gets the options value.
59 //      /// </para>
60 //      /// <para></para>
61 //      /// </summary>
62 //      public SequencesOptions<LinkIndex> Options { get; }
63 //      /// <summary>
64 //      /// <para>
65 //      /// Gets the links value.
66 //      /// </para>
67 //      /// <para></para>
68 //      /// </summary>
69 //      public SynchronizedLinks<LinkIndex> Links { get; }
70 //      private readonly ISynchronization _sync;
71 //
72 //      /// <summary>
73 //      /// <para>
74 //      /// Gets the constants value.
75 //      /// </para>
76 //      /// <para></para>
77 //      /// </summary>
78 //      public LinksConstants<LinkIndex> Constants { get; }
79 //
80 //      /// <summary>
81 //      /// <para>
82 //      /// Initializes a new <see cref="Sequences"/> instance.
83 //      /// </para>
84 //      /// <para></para>
85 //      /// </summary>
86 //      /// <param name="links">
87 //      /// <para>A links.</para>
88 //      /// <para></para>
89 //      /// </param>
90 //      /// <param name="options">
91 //      /// <para>A options.</para>
92 //      /// <para></para>
93 //      /// </param>
94 //      [MethodImpl(MethodImplOptions.AggressiveInlining)]
95 //      public Sequences(SynchronizedLinks<LinkIndex> links, SequencesOptions<LinkIndex>
↪ options)
96 //      {
97 //          Links = links;
98 //          _sync = links.SyncRoot;
99 //          Options = options;
100 //          Options.ValidateOptions();
101 //          Options.InitOptions(Links);
102 //          Constants = links.Constants;
103 //      }
104 //
105 //      /// <summary>
106 //      /// <para>
107 //      /// Initializes a new <see cref="Sequences"/> instance.
108 //      /// </para>
109 //      /// <para></para>
110 //      /// </summary>
111 //      /// <param name="links">
112 //      /// <para>A links.</para>
113 //      /// <para></para>
114 //      /// </param>
115 //      [MethodImpl(MethodImplOptions.AggressiveInlining)]
116 //      public Sequences(SynchronizedLinks<LinkIndex> links) : this(links, new
↪ SequencesOptions<LinkIndex>()) { }
117 //
118 //      /// <summary>
119 //      /// <para>
120 //      /// Determines whether this instance is sequence.
121 //      /// </para>
122 //      /// <para></para>
123 //      /// </summary>
124 //      /// <param name="sequence">
125 //      /// <para>The sequence.</para>
126 //      /// <para></para>
127 //      /// </param>
128 //      /// <returns>
129 //      /// <para>The bool</para>

```

```

130 //      /// <para></para>
131 //      /// </returns>
132 //      [MethodImpl(MethodImplOptions.AggressiveInlining)]
133 //      public bool IsSequence(LinkIndex sequence)
134 //      {
135 //          return _sync.DoRead(() =>
136 //          {
137 //              if (Options.UseSequenceMarker)
138 //              {
139 //                  return Options.MarkedSequenceMatcher.IsMatched(sequence);
140 //              }
141 //              return !Links.Unsync.IsPartialPoint(sequence);
142 //          });
143 //      }
144 //      [MethodImpl(MethodImplOptions.AggressiveInlining)]
145 //      private LinkIndex GetSequenceByElements(LinkIndex sequence)
146 //      {
147 //          if (Options.UseSequenceMarker)
148 //          {
149 //              return Links.SearchOrDefault(Options.SequenceMarkerLink, sequence);
150 //          }
151 //          return sequence;
152 //      }
153 //      [MethodImpl(MethodImplOptions.AggressiveInlining)]
154 //      private LinkIndex GetSequenceElements(LinkIndex sequence)
155 //      {
156 //          if (Options.UseSequenceMarker)
157 //          {
158 //              var linkContents = new Link<ulong>(Links.GetLink(sequence));
159 //              if (linkContents.Source == Options.SequenceMarkerLink)
160 //              {
161 //                  return linkContents.Target;
162 //              }
163 //              if (linkContents.Target == Options.SequenceMarkerLink)
164 //              {
165 //                  return linkContents.Source;
166 //              }
167 //          }
168 //          return sequence;
169 //      }
170 //
171 //      #region Count
172 //
173 //      /// <summary>
174 //      /// <para>
175 //      /// Counts the restriction.
176 //      /// </para>
177 //      /// <para></para>
178 //      /// </summary>
179 //      /// <param name="restriction">
180 //      /// <para>The restriction.</para>
181 //      /// <para></para>
182 //      /// </param>
183 //      /// <exception cref="NotImplementedException">
184 //      /// <para></para>
185 //      /// <para></para>
186 //      /// </exception>
187 //      /// <returns>
188 //      /// <para>The link index</para>
189 //      /// <para></para>
190 //      /// </returns>
191 //      [MethodImpl(MethodImplOptions.AggressiveInlining)]
192 //      public LinkIndex Count(ICollection<LinkIndex>? restriction)
193 //      {
194 //          if (restriction.IsNullOrEmpty())
195 //          {
196 //              return Links.Count(Constants.Any, Options.SequenceMarkerLink, Constants.Any);
197 //          }
198 //          if (restriction.Count == 1) // Первая связь это адрес
199 //          {
200 //              var sequenceIndex = restriction[0];
201 //              if (sequenceIndex == Constants.Null)
202 //              {
203 //                  return 0;
204 //              }
205 //              if (sequenceIndex == Constants.Any)
206 //              {
207 //                  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(ICollection<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(ICollection<LinkIndex>? restriction)
279 // {
280 //     LinkIndex[] sequence = restriction.SkipFirst();
281 //     if (Options.UseIndex)
282 //     {

```

```

283 //         Options.Index.Add(sequence);
284 //     }
285 //     var sequenceRoot = default(LinkIndex);
286 //     if (Options.EnforceSingleSequenceVersionOnWriteBasedOnExisting)
287 //     {
288 //         var matches = Each(restriction);
289 //         if (matches.Count > 0)
290 //         {
291 //             sequenceRoot = matches[0];
292 //         }
293 //     }
294 //     else if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew)
295 //     {
296 //         return CompactCore(sequence);
297 //     }
298 //     if (sequenceRoot == default)
299 //     {
300 //         sequenceRoot = Options.LinksToSequenceConverter.Convert(sequence);
301 //     }
302 //     if (Options.UseSequenceMarker)
303 //     {
304 //         return Links.Unsync.GetOrCreate(Options.SequenceMarkerLink, sequenceRoot);
305 //     }
306 //     return sequenceRoot; // Возвращаем корень последовательности (т.е. сами элементы)
307 // }
308 //
309 // #endregion
310 //
311 // #region Each
312 //
313 //     /// <summary>
314 //     /// <para>
315 //     /// Eaches the sequence.
316 //     /// </para>
317 //     /// <para></para>
318 //     /// </summary>
319 //     /// <param name="sequence">
320 //     /// <para>The sequence.</para>
321 //     /// <para></para>
322 //     /// </param>
323 //     /// <returns>
324 //     /// <para>The results.</para>
325 //     /// <para></para>
326 //     /// </returns>
327 //     [MethodImpl(MethodImplOptions.AggressiveInlining)]
328 //     public List<LinkIndex> Each(IList<LinkIndex> sequence)
329 //     {
330 //         var results = new List<LinkIndex>();
331 //         var filler = new ListFiller<LinkIndex, LinkIndex>(results, Constants.Continue);
332 //         Each(filler.AddFirstAndReturnConstant, sequence);
333 //         return results;
334 //     }
335 //
336 //     /// <summary>
337 //     /// <para>
338 //     /// Eaches the handler.
339 //     /// </para>
340 //     /// <para></para>
341 //     /// </summary>
342 //     /// <param name="handler">
343 //     /// <para>The handler.</para>
344 //     /// <para></para>
345 //     /// </param>
346 //     /// <param name="restriction">
347 //     /// <para>The restriction.</para>
348 //     /// <para></para>
349 //     /// </param>
350 //     /// <exception cref="NotImplementedException">
351 //     /// <para></para>
352 //     /// <para></para>
353 //     /// </exception>
354 //     /// <returns>
355 //     /// <para>The link index</para>
356 //     /// <para></para>
357 //     /// </returns>
358 //     [MethodImpl(MethodImplOptions.AggressiveInlining)]
359 //     public LinkIndex Each(ReadHandler<LinkIndex> handler, IList<LinkIndex>? restriction)
360 //     {

```



```

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(ICollection<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<ICollection<LinkIndex>, LinkIndex> innerHandler = Options.UseSequenceMarker ?
↵ (Func<ICollection<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;

```

```

431 //         }
432 //     }
433 //     if (values.Count >= 3)
434 //     {
435 //         if (StepLeft(innerHandler, values[values.Count - 2], values[values.Count -
↳ 1]) != Constants.Continue)
436 //         {
437 //             return Constants.Break;
438 //         }
439 //     }
440 //     return Constants.Continue;
441 // }
442 // [MethodImpl(MethodImplOptions.AggressiveInlining)]
443 // private LinkIndex PartialStepRight(Func<IList<LinkIndex>, LinkIndex> handler,
↳ LinkIndex left, LinkIndex right)
444 // {
445 //     return Links.Unsync.Each(doublet =>
446 //     {
447 //         var doubletIndex = doublet[Constants.IndexPart];
448 //         if (StepRight(handler, doubletIndex, right) != Constants.Continue)
449 //         {
450 //             return Constants.Break;
451 //         }
452 //         if (left != doubletIndex)
453 //         {
454 //             return PartialStepRight(handler, doubletIndex, right);
455 //         }
456 //         return Constants.Continue;
457 //     }, new Link<LinkIndex>(Constants.Any, Constants.Any, left));
458 // }
459 // [MethodImpl(MethodImplOptions.AggressiveInlining)]
460 // 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));
461 // [MethodImpl(MethodImplOptions.AggressiveInlining)]
462 // private LinkIndex TryStepRightUp(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
↳ right, LinkIndex stepFrom)
463 // {
464 //     var upStep = stepFrom;
465 //     var firstSource = Links.Unsync.GetTarget(upStep);
466 //     while (firstSource != right && firstSource != upStep)
467 //     {
468 //         upStep = firstSource;
469 //         firstSource = Links.Unsync.GetSource(upStep);
470 //     }
471 //     if (firstSource == right)
472 //     {
473 //         return handler(new LinkAddress<LinkIndex>(stepFrom));
474 //     }
475 //     return Constants.Continue;
476 // }
477 // [MethodImpl(MethodImplOptions.AggressiveInlining)]
478 // 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));
479 // [MethodImpl(MethodImplOptions.AggressiveInlining)]
480 // private LinkIndex TryStepLeftUp(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
↳ left, LinkIndex stepFrom)
481 // {
482 //     var upStep = stepFrom;
483 //     var firstTarget = Links.Unsync.GetSource(upStep);
484 //     while (firstTarget != left && firstTarget != upStep)
485 //     {
486 //         upStep = firstTarget;
487 //         firstTarget = Links.Unsync.GetTarget(upStep);
488 //     }
489 //     if (firstTarget == left)
490 //     {
491 //         return handler(new LinkAddress<LinkIndex>(stepFrom));
492 //     }
493 //     return Constants.Continue;
494 // }
495 //
496 // #endregion
497 //
498 // #region Update
499 //

```

```

500 //      /// <summary>
501 //      /// <para>
502 //      /// Updates the restriction.
503 //      /// </para>
504 //      /// <para></para>
505 //      /// </summary>
506 //      /// <param name="restriction">
507 //      /// <para>The restriction.</para>
508 //      /// <para></para>
509 //      /// </param>
510 //      /// <param name="substitution">
511 //      /// <para>The substitution.</para>
512 //      /// <para></para>
513 //      /// </param>
514 //      /// <returns>
515 //      /// <para>The link index</para>
516 //      /// <para></para>
517 //      /// </returns>
518 //      [MethodImpl(MethodImplOptions.AggressiveInlining)]
519 //      public LinkIndex Update(ICollection<LinkIndex>? restriction, ICollection<LinkIndex>?
    ↪ substitution, WriteHandler<LinkIndex> handler)
520 //      {
521 //          var sequence = restriction.SkipFirst();
522 //          var newSequence = substitution.SkipFirst();
523 //          if (sequence.IsNullOrEmpty() && newSequence.IsNullOrEmpty())
524 //          {
525 //              return Constants.Null;
526 //          }
527 //          if (sequence.IsNullOrEmpty())
528 //          {
529 //              return Create(substitution);
530 //          }
531 //          if (newSequence.IsNullOrEmpty())
532 //          {
533 //              Delete(restriction);
534 //              return Constants.Null;
535 //          }
536 //          return _sync.DoWrite((Func<ulong>)(() =>
537 //          {
538 //              ILinksExtensions.EnsureLinkIsAnyOrExists<ulong>(Links,
    ↪ (ICollection<ulong>)sequence);
539 //              Links.EnsureLinkExists(newSequence);
540 //              return UpdateCore(sequence, newSequence);
541 //          }));
542 //          [MethodImpl(MethodImplOptions.AggressiveInlining)]
543 //          private LinkIndex UpdateCore(ICollection<LinkIndex> sequence, ICollection<LinkIndex> newSequence,
544 //          ↪ WriteHandler<LinkIndex> handler)
545 //          {
546 //              LinkIndex bestVariant;
547 //              if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew &&
    ↪ !sequence.EqualTo(newSequence))
548 //              {
549 //                  bestVariant = CompactCore(newSequence);
550 //              }
551 //              else
552 //              {
553 //                  bestVariant = CreateCore(newSequence);
554 //              }
555 //              // TODO: Check all options only ones before loop execution
556 //              // Возможно нужно две версии Each, возвращающий фактические последовательности и
    ↪ с маркером,
557 //              // или возможно даже возвращать и тот и тот вариант. С другой стороны все
    ↪ варианты можно получить имея только фактические последовательности.
558 //              foreach (var variant in Each(sequence))
559 //              {
560 //                  if (variant != bestVariant)
561 //                  {
562 //                      UpdateOneCore(variant, bestVariant);
563 //                  }
564 //              }
565 //              return bestVariant;
566 //          }
567 //          [MethodImpl(MethodImplOptions.AggressiveInlining)]
568 //          private void UpdateOneCore(LinkIndex sequence, LinkIndex newSequence,
    ↪ WriteHandler<LinkIndex> handler)
569 //          {

```

```

570 //         if (Options.UseGarbageCollection)
571 //         {
572 //             var sequenceElements = GetSequenceElements(sequence);
573 //             var sequenceElementsContents = new
↵ Link<ulong>(Links.GetLink(sequenceElements));
574 //             var sequenceLink = GetSequenceByElements(sequenceElements);
575 //             var newSequenceElements = GetSequenceElements(newSequence);
576 //             var newSequenceLink = GetSequenceByElements(newSequenceElements);
577 //             if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
578 //             {
579 //                 if (sequenceLink != Constants.Null)
580 //                 {
581 //                     Links.Unsync.MergeAndDelete(sequenceLink, newSequenceLink);
582 //                 }
583 //                 Links.Unsync.MergeAndDelete(sequenceElements, newSequenceElements);
584 //             }
585 //             ClearGarbage(sequenceElementsContents.Source);
586 //             ClearGarbage(sequenceElementsContents.Target);
587 //         }
588 //     else
589 //     {
590 //         if (Options.UseSequenceMarker)
591 //         {
592 //             var sequenceElements = GetSequenceElements(sequence);
593 //             var sequenceLink = GetSequenceByElements(sequenceElements);
594 //             var newSequenceElements = GetSequenceElements(newSequence);
595 //             var newSequenceLink = GetSequenceByElements(newSequenceElements);
596 //             if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
597 //             {
598 //                 if (sequenceLink != Constants.Null)
599 //                 {
600 //                     Links.Unsync.MergeAndDelete(sequenceLink, newSequenceLink);
601 //                 }
602 //                 Links.Unsync.MergeAndDelete(sequenceElements, newSequenceElements);
603 //             }
604 //         }
605 //     else
606 //     {
607 //         if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
608 //         {
609 //             Links.Unsync.MergeAndDelete(sequence, newSequence);
610 //         }
611 //     }
612 // }
613 //
614 // #endregion
615 //
616 // #region Delete
617 //
618 // /// <summary>
619 // /// <para>
620 // /// Deletes the restriction.
621 // /// </para>
622 // /// <para></para>
623 // /// </summary>
624 // /// <param name="restriction">
625 // /// <para>The restriction.</para>
626 // /// <para></para>
627 // /// </param>
628 // [MethodImpl(MethodImplOptions.AggressiveInlining)]
629 // public void Delete(ICollection<LinkIndex>? restriction)
630 // {
631 //     _sync.DoWrite(() =>
632 //     {
633 //         var sequence = restriction.SkipFirst();
634 //         // TODO: Check all options only ones before loop execution
635 //         foreach (var linkToDelete in Each(sequence))
636 //         {
637 //             DeleteOneCore(linkToDelete);
638 //         }
639 //     });
640 // }
641 // [MethodImpl(MethodImplOptions.AggressiveInlining)]
642 // private void DeleteOneCore(LinkIndex link)
643 // {
644 //     if (Options.UseGarbageCollection)
645 //

```

```

646 //      {
647 //          var sequenceElements = GetSequenceElements(link);
648 //          var sequenceElementsContents = new
↪ Link<ulong>(Links.GetLink(sequenceElements));
649 //          var sequenceLink = GetSequenceByElements(sequenceElements);
650 //          if (Options.UseCascadeDelete || CountUsages(link) == 0)
651 //          {
652 //              if (sequenceLink != Constants.Null)
653 //              {
654 //                  Links.Unsync.Delete(sequenceLink);
655 //              }
656 //              Links.Unsync.Delete(link);
657 //          }
658 //          ClearGarbage(sequenceElementsContents.Source);
659 //          ClearGarbage(sequenceElementsContents.Target);
660 //      }
661 //      else
662 //      {
663 //          if (Options.UseSequenceMarker)
664 //          {
665 //              var sequenceElements = GetSequenceElements(link);
666 //              var sequenceLink = GetSequenceByElements(sequenceElements);
667 //              if (Options.UseCascadeDelete || CountUsages(link) == 0)
668 //              {
669 //                  if (sequenceLink != Constants.Null)
670 //                  {
671 //                      Links.Unsync.Delete(sequenceLink);
672 //                  }
673 //                  Links.Unsync.Delete(link);
674 //              }
675 //          }
676 //          else
677 //          {
678 //              if (Options.UseCascadeDelete || CountUsages(link) == 0)
679 //              {
680 //                  Links.Unsync.Delete(link);
681 //              }
682 //          }
683 //      }
684 //  }
685 //
686 //  #endregion
687 //
688 //  #region Compactification
689 //
690 //  /// <summary>
691 //  /// <para>
692 //  /// Compacts the all.
693 //  /// </para>
694 //  /// <para></para>
695 //  /// </summary>
696 //  [MethodImpl(MethodImplOptions.AggressiveInlining)]
697 //  public void CompactAll()
698 //  {
699 //      _sync.DoWrite(() =>
700 //      {
701 //          var sequences = Each((LinkAddress<LinkIndex>)Constants.Any);
702 //          for (int i = 0; i < sequences.Count; i++)
703 //          {
704 //              var sequence = this.ToList(sequences[i]);
705 //              Compact(sequence.ShiftRight());
706 //          }
707 //      });
708 //  }
709 //
710 //  /// <remarks>
711 //  /// bestVariant можно выбирать по максимальному числу использований,
712 //  /// но балансированный позволяет гарантировать уникальность (если есть возможность,
713 //  /// гарантировать его использование в других местах).
714 //  ///
715 //  /// Получается этот метод должен игнорировать
↪ Options.EnforceSingleSequenceVersionOnWrite
716 //  /// </remarks>
717 //  [MethodImpl(MethodImplOptions.AggressiveInlining)]
718 //  public LinkIndex Compact(ICollection<LinkIndex> sequence)
719 //  {
720 //      return _sync.DoWrite(() =>

```

```

721 //      {
722 //          if (sequence.IsNullOrEmpty())
723 //          {
724 //              return Constants.Null;
725 //          }
726 //          Links.EnsureInnerReferenceExists(sequence, nameof(sequence));
727 //          return CompactCore(sequence);
728 //      });
729 //  }
730 //  [MethodImpl(MethodImplOptions.AggressiveInlining)]
731 //  private LinkIndex CompactCore(ICollection<LinkIndex> sequence) => UpdateCore(sequence,
↵ sequence);
732 //
733 //      #endregion
734 //
735 //      #region Garbage Collection
736 //      [MethodImpl(MethodImplOptions.AggressiveInlining)]
737 //      private bool IsGarbage(LinkIndex link) => link != Options.SequenceMarkerLink &&
↵ !Links.Unsync.IsPartialPoint(link) && Links.Count(Constants.Any, link) == 0;
738 //      [MethodImpl(MethodImplOptions.AggressiveInlining)]
739 //      private void ClearGarbage(LinkIndex link)
740 //      {
741 //          if (IsGarbage(link))
742 //          {
743 //              var contents = new Link<ulong>(Links.GetLink(link));
744 //              Links.Unsync.Delete(link);
745 //              ClearGarbage(contents.Source);
746 //              ClearGarbage(contents.Target);
747 //          }
748 //      }
749 //
750 //      #endregion
751 //
752 //      #region Walkers
753 //
754 //      /// <summary>
755 //      /// <para>
756 //      /// Determines whether this instance each part.
757 //      /// </para>
758 //      /// <para></para>
759 //      /// </summary>
760 //      /// <param name="handler">
761 //      /// <para>The handler.</para>
762 //      /// <para></para>
763 //      /// </param>
764 //      /// <param name="sequence">
765 //      /// <para>The sequence.</para>
766 //      /// <para></para>
767 //      /// </param>
768 //      /// <returns>
769 //      /// <para>The bool</para>
770 //      /// <para></para>
771 //      /// </returns>
772 //      [MethodImpl(MethodImplOptions.AggressiveInlining)]
773 //      public bool EachPart(Func<LinkIndex, bool> handler, LinkIndex sequence)
774 //      {
775 //          return _sync.DoRead(() =>
776 //          {
777 //              var links = Links.Unsync;
778 //              foreach (var part in Options.Walker.Walk(sequence))
779 //              {
780 //                  if (!handler(part))
781 //                  {
782 //                      return false;
783 //                  }
784 //              }
785 //              return true;
786 //          });
787 //      }
788 //
789 //      /// <summary>
790 //      /// <para>
791 //      /// Represents the matcher.
792 //      /// </para>
793 //      /// <para></para>
794 //      /// </summary>
795 //      /// <seealso cref="RightSequenceWalker{LinkIndex}" />

```

```

796 // public class Matcher : RightSequenceWalker<LinkIndex>
797 // {
798 //     private readonly Sequences _sequences;
799 //     private readonly IList<LinkIndex> _patternSequence;
800 //     private readonly HashSet<LinkIndex> _linksInSequence;
801 //     private readonly HashSet<LinkIndex> _results;
802 //     private readonly ReadHandler<LinkIndex> _stopableHandler;
803 //     private readonly HashSet<LinkIndex> _readAsElements;
804 //     private int _filterPosition;
805 //
806 //     /// <summary>
807 //     /// <para>
808 //     /// Initializes a new <see cref="Matcher"/> instance.
809 //     /// </para>
810 //     /// <para></para>
811 //     /// </summary>
812 //     /// <param name="sequences">
813 //     /// <para>A sequences.</para>
814 //     /// <para></para>
815 //     /// </param>
816 //     /// <param name="patternSequence">
817 //     /// <para>A pattern sequence.</para>
818 //     /// <para></para>
819 //     /// </param>
820 //     /// <param name="results">
821 //     /// <para>A results.</para>
822 //     /// <para></para>
823 //     /// </param>
824 //     /// <param name="stopableHandler">
825 //     /// <para>A stopable handler.</para>
826 //     /// <para></para>
827 //     /// </param>
828 //     /// <param name="readAsElements">
829 //     /// <para>A read as elements.</para>
830 //     /// <para></para>
831 //     /// </param>
832 //     [MethodImpl(MethodImplOptions.AggressiveInlining)]
833 //     public Matcher(Sequences sequences, IList<LinkIndex> patternSequence,
↳ HashSet<LinkIndex> results, ReadHandler<LinkIndex> stopableHandler, HashSet<LinkIndex>
↳ readAsElements = null)
834 //         : base(sequences.Links.Unsync, new DefaultStack<LinkIndex>())
835 //     {
836 //         _sequences = sequences;
837 //         _patternSequence = patternSequence;
838 //         _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
↳ _links.Constants.Any && x != ZeroOrMany));
839 //         _results = results;
840 //         _stopableHandler = stopableHandler;
841 //         _readAsElements = readAsElements;
842 //     }
843 //
844 //     /// <summary>
845 //     /// <para>
846 //     /// Determines whether this instance is element.
847 //     /// </para>
848 //     /// <para></para>
849 //     /// </summary>
850 //     /// <param name="link">
851 //     /// <para>The link.</para>
852 //     /// <para></para>
853 //     /// </param>
854 //     /// <returns>
855 //     /// <para>The bool</para>
856 //     /// <para></para>
857 //     /// </returns>
858 //     [MethodImpl(MethodImplOptions.AggressiveInlining)]
859 //     protected override bool IsElement(LinkIndex link) => base.IsElement(link) ||
↳ (_readAsElements != null && _readAsElements.Contains(link)) ||
↳ _linksInSequence.Contains(link);
860 //
861 //     /// <summary>
862 //     /// <para>
863 //     /// Determines whether this instance full match.
864 //     /// </para>
865 //     /// <para></para>
866 //     /// </summary>
867 //     /// <param name="sequenceToMatch">

```

```

868 //      /// <para>The sequence to match.</para>
869 //      /// <para></para>
870 //      /// </param>
871 //      /// <returns>
872 //      /// <para>The bool</para>
873 //      /// <para></para>
874 //      /// </returns>
875 //      [MethodImpl(MethodImplOptions.AggressiveInlining)]
876 //      public bool FullMatch(LinkIndex sequenceToMatch)
877 //      {
878 //          _filterPosition = 0;
879 //          foreach (var part in Walk(sequenceToMatch))
880 //          {
881 //              if (!FullMatchCore(part))
882 //              {
883 //                  break;
884 //              }
885 //          }
886 //          return _filterPosition == _patternSequence.Count;
887 //      }
888 //      [MethodImpl(MethodImplOptions.AggressiveInlining)]
889 //      private bool FullMatchCore(LinkIndex element)
890 //      {
891 //          if (_filterPosition == _patternSequence.Count)
892 //          {
893 //              _filterPosition = -2; // Длиннее чем нужно
894 //              return false;
895 //          }
896 //          if (_patternSequence[_filterPosition] != _links.Constants.Any
897 //              && element != _patternSequence[_filterPosition])
898 //          {
899 //              _filterPosition = -1;
900 //              return false; // Начинается/Продолжается иначе
901 //          }
902 //          _filterPosition++;
903 //          return true;
904 //      }
905 //
906 //      /// <summary>
907 //      /// <para>
908 //      /// Adds the full matched to results using the specified restriction.
909 //      /// </para>
910 //      /// <para></para>
911 //      /// </summary>
912 //      /// <param name="restriction">
913 //      /// <para>The restriction.</para>
914 //      /// <para></para>
915 //      /// </param>
916 //      [MethodImpl(MethodImplOptions.AggressiveInlining)]
917 //      public void AddFullMatchedToResults(ICollection<LinkIndex>? restriction)
918 //      {
919 //          var sequenceToMatch = restriction[_links.Constants.IndexPart];
920 //          if (FullMatch(sequenceToMatch))
921 //          {
922 //              _results.Add(sequenceToMatch);
923 //          }
924 //      }
925 //
926 //      /// <summary>
927 //      /// <para>
928 //      /// Handles the full matched using the specified restriction.
929 //      /// </para>
930 //      /// <para></para>
931 //      /// </summary>
932 //      /// <param name="restriction">
933 //      /// <para>The restriction.</para>
934 //      /// <para></para>
935 //      /// </param>
936 //      /// <returns>
937 //      /// <para>The link index</para>
938 //      /// <para></para>
939 //      /// </returns>
940 //      [MethodImpl(MethodImplOptions.AggressiveInlining)]
941 //      public LinkIndex HandleFullMatched(ICollection<LinkIndex>? restriction)
942 //      {
943 //          var sequenceToMatch = restriction[_links.Constants.IndexPart];
944 //          if (FullMatch(sequenceToMatch) && _results.Add(sequenceToMatch))
945 //          {

```



```

946 //         return _stopableHandler(new LinkAddress<LinkIndex>(sequenceToMatch));
947 //     }
948 //     return _links.Constants.Continue;
949 // }
950 //
951 //     /// <summary>
952 //     /// <para>
953 //     /// Handles the full matched sequence using the specified restriction.
954 //     /// </para>
955 //     /// <para></para>
956 //     /// </summary>
957 //     /// <param name="restriction">
958 //     /// <para>The restriction.</para>
959 //     /// <para></para>
960 //     /// </param>
961 //     /// <returns>
962 //     /// <para>The link index</para>
963 //     /// <para></para>
964 //     /// </returns>
965 //     [MethodImpl(MethodImplOptions.AggressiveInlining)]
966 //     public LinkIndex HandleFullMatchedSequence(ICollection<LinkIndex>? restriction)
967 //     {
968 //         var sequenceToMatch = restriction[_links.Constants.IndexPart];
969 //         var sequence = _sequences.GetSequenceByElements(sequenceToMatch);
970 //         if (sequence != _links.Constants.Null && FullMatch(sequenceToMatch) &&
↵ _results.Add(sequenceToMatch))
971 //         {
972 //             return _stopableHandler(new LinkAddress<LinkIndex>(sequence));
973 //         }
974 //         return _links.Constants.Continue;
975 //     }
976 //
977 //     /// <remarks>
978 //     /// TODO: Add support for LinksConstants.Any
979 //     /// </remarks>
980 //     [MethodImpl(MethodImplOptions.AggressiveInlining)]
981 //     public bool PartialMatch(LinkIndex sequenceToMatch)
982 //     {
983 //         _filterPosition = -1;
984 //         foreach (var part in Walk(sequenceToMatch))
985 //         {
986 //             if (!PartialMatchCore(part))
987 //             {
988 //                 break;
989 //             }
990 //         }
991 //         return _filterPosition == _patternSequence.Count - 1;
992 //     }
993 //     [MethodImpl(MethodImplOptions.AggressiveInlining)]
994 //     private bool PartialMatchCore(LinkIndex element)
995 //     {
996 //         if (_filterPosition == (_patternSequence.Count - 1))
997 //         {
998 //             return false; // Нашлось
999 //         }
1000 //         if (_filterPosition >= 0)
1001 //         {
1002 //             if (element == _patternSequence[_filterPosition + 1])
1003 //             {
1004 //                 _filterPosition++;
1005 //             }
1006 //             else
1007 //             {
1008 //                 _filterPosition = -1;
1009 //             }
1010 //         }
1011 //         if (_filterPosition < 0)
1012 //         {
1013 //             if (element == _patternSequence[0])
1014 //             {
1015 //                 _filterPosition = 0;
1016 //             }
1017 //         }
1018 //         return true; // Ищем дальше
1019 //     }
1020 //
1021 //     /// <summary>

```

```

1022 //          /// <para>
1023 //          /// Adds the partial matched to results using the specified sequence to match.
1024 //          /// </para>
1025 //          /// <para></para>
1026 //          /// </summary>
1027 //          /// <param name="sequenceToMatch">
1028 //          /// <para>The sequence to match.</para>
1029 //          /// <para></para>
1030 //          /// </param>
1031 //          [MethodImpl(MethodImplOptions.AggressiveInlining)]
1032 //          public void AddPartialMatchedToResults(LinkIndex sequenceToMatch)
1033 //          {
1034 //              if (PartialMatch(sequenceToMatch))
1035 //              {
1036 //                  _results.Add(sequenceToMatch);
1037 //              }
1038 //          }
1039 //
1040 //          /// <summary>
1041 //          /// <para>
1042 //          /// Handles the partial matched using the specified restriction.
1043 //          /// </para>
1044 //          /// <para></para>
1045 //          /// </summary>
1046 //          /// <param name="restriction">
1047 //          /// <para>The restriction.</para>
1048 //          /// <para></para>
1049 //          /// </param>
1050 //          /// <returns>
1051 //          /// <para>The link index</para>
1052 //          /// <para></para>
1053 //          /// </returns>
1054 //          [MethodImpl(MethodImplOptions.AggressiveInlining)]
1055 //          public LinkIndex HandlePartialMatched(ICollection<LinkIndex>? restriction)
1056 //          {
1057 //              var sequenceToMatch = restriction[_links.Constants.IndexPart];
1058 //              if (PartialMatch(sequenceToMatch))
1059 //              {
1060 //                  return _stopableHandler(new LinkAddress<LinkIndex>(sequenceToMatch));
1061 //              }
1062 //              return _links.Constants.Continue;
1063 //          }
1064 //
1065 //          /// <summary>
1066 //          /// <para>
1067 //          /// Adds the all partial matched to results using the specified sequences to
1068 //          ↪ match.
1069 //          /// </para>
1070 //          /// <para></para>
1071 //          /// </summary>
1072 //          /// <param name="sequencesToMatch">
1073 //          /// <para>The sequences to match.</para>
1074 //          /// <para></para>
1075 //          /// </param>
1076 //          [MethodImpl(MethodImplOptions.AggressiveInlining)]
1077 //          public void AddAllPartialMatchedToResults(IEnumerable<LinkIndex> sequencesToMatch)
1078 //          {
1079 //              foreach (var sequenceToMatch in sequencesToMatch)
1080 //              {
1081 //                  if (PartialMatch(sequenceToMatch))
1082 //                  {
1083 //                      _results.Add(sequenceToMatch);
1084 //                  }
1085 //              }
1086 //          }
1087 //
1088 //          /// <summary>
1089 //          /// <para>
1090 //          /// Adds the all partial matched to results and read as elements using the
1091 //          ↪ specified sequences to match.
1092 //          /// </para>
1093 //          /// <para></para>
1094 //          /// </summary>
1095 //          /// <param name="sequencesToMatch">
1096 //          /// <para>The sequences to match.</para>
1097 //          /// <para></para>
1098 //          /// </param>
1099 //          [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

1098 //         public void AddAllPartialMatchedToResultsAndReadAsElements(IEnumerable<LinkIndex>
↵ sequencesToMatch)
1099 //         {
1100 //             foreach (var sequenceToMatch in sequencesToMatch)
1101 //             {
1102 //                 if (PartialMatch(sequenceToMatch))
1103 //                 {
1104 //                     _readAsElements.Add(sequenceToMatch);
1105 //                     _results.Add(sequenceToMatch);
1106 //                 }
1107 //             }
1108 //         }
1109 //     }
1110 //
1111 //     #endregion
1112 // }
1113 // }

```

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

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3 using Platform.Collections.Lists;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Data.Doublets.Sequences
8 {
9     /// <summary>
10    /// <para>
11    /// Represents the sequences extensions.
12    /// </para>
13    /// <para></para>
14    /// </summary>
15    public static class SequencesExtensions
16    {
17        /// <summary>
18        /// <para>
19        /// Creates the sequences.
20        /// </para>
21        /// <para></para>
22        /// </summary>
23        /// <typeparam name="TLinkAddress">
24        /// <para>The link.</para>
25        /// <para></para>
26        /// </typeparam>
27        /// <param name="sequences">
28        /// <para>The sequences.</para>
29        /// <para></para>
30        /// </param>
31        /// <param name="groupedSequence">
32        /// <para>The grouped sequence.</para>
33        /// <para></para>
34        /// </param>
35        /// <returns>
36        /// <para>The link</para>
37        /// <para></para>
38        /// </returns>
39        [MethodImpl(MethodImplOptions.AggressiveInlining)]
40        public static TLinkAddress Create<TLinkAddress>(this ILinks<TLinkAddress> sequences,
↵ IList<TLinkAddress[]> groupedSequence)
41        {
42            var finalSequence = new TLinkAddress[groupedSequence.Count];
43            for (var i = 0; i < finalSequence.Length; i++)
44            {
45                var part = groupedSequence[i];
46                finalSequence[i] = part.Length == 1 ? part[0] :
↵ sequences.Create(part.ShiftRight());
47            }
48            return sequences.Create(finalSequence.ShiftRight());
49        }
50
51        /// <summary>
52        /// <para>
53        /// Returns the list using the specified sequences.
54        /// </para>
55        /// <para></para>
56        /// </summary>
57        /// <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>

```

```

48 public bool UseCascadeUpdate
49 {
50     [MethodImpl(MethodImplOptions.AggressiveInlining)]
51     get;
52     [MethodImpl(MethodImplOptions.AggressiveInlining)]
53     set;
54 }
55
56 /// <summary>
57 /// <para>
58 /// Gets or sets the use cascade delete value.
59 /// </para>
60 /// <para></para>
61 /// </summary>
62 public bool UseCascadeDelete
63 {
64     [MethodImpl(MethodImplOptions.AggressiveInlining)]
65     get;
66     [MethodImpl(MethodImplOptions.AggressiveInlining)]
67     set;
68 }
69
70 /// <summary>
71 /// <para>
72 /// Gets or sets the use index value.
73 /// </para>
74 /// <para></para>
75 /// </summary>
76 public bool UseIndex
77 {
78     [MethodImpl(MethodImplOptions.AggressiveInlining)]
79     get;
80     [MethodImpl(MethodImplOptions.AggressiveInlining)]
81     set;
82 } // TODO: Update Index on sequence update/delete.
83
84 /// <summary>
85 /// <para>
86 /// Gets or sets the use sequence marker value.
87 /// </para>
88 /// <para></para>
89 /// </summary>
90 public bool UseSequenceMarker
91 {
92     [MethodImpl(MethodImplOptions.AggressiveInlining)]
93     get;
94     [MethodImpl(MethodImplOptions.AggressiveInlining)]
95     set;
96 }
97
98 /// <summary>
99 /// <para>
100 /// Gets or sets the use compression value.
101 /// </para>
102 /// <para></para>
103 /// </summary>
104 public bool UseCompression
105 {
106     [MethodImpl(MethodImplOptions.AggressiveInlining)]
107     get;
108     [MethodImpl(MethodImplOptions.AggressiveInlining)]
109     set;
110 }
111
112 /// <summary>
113 /// <para>
114 /// Gets or sets the use garbage collection value.
115 /// </para>
116 /// <para></para>
117 /// </summary>
118 public bool UseGarbageCollection
119 {
120     [MethodImpl(MethodImplOptions.AggressiveInlining)]
121     get;
122     [MethodImpl(MethodImplOptions.AggressiveInlining)]
123     set;
124 }
125
126 /// <summary>
127 /// <para>

```

```

128     /// Gets or sets the enforce single sequence version on write based on existing value.
129     /// </para>
130     /// <para></para>
131     /// </summary>
132     public bool EnforceSingleSequenceVersionOnWriteBasedOnExisting
133     {
134         [MethodImpl(MethodImplOptions.AggressiveInlining)]
135         get;
136         [MethodImpl(MethodImplOptions.AggressiveInlining)]
137         set;
138     }
139
140     /// <summary>
141     /// <para>
142     /// Gets or sets the enforce single sequence version on write based on new value.
143     /// </para>
144     /// <para></para>
145     /// </summary>
146     public bool EnforceSingleSequenceVersionOnWriteBasedOnNew
147     {
148         [MethodImpl(MethodImplOptions.AggressiveInlining)]
149         get;
150         [MethodImpl(MethodImplOptions.AggressiveInlining)]
151         set;
152     }
153
154     /// <summary>
155     /// <para>
156     /// Gets or sets the marked sequence matcher value.
157     /// </para>
158     /// <para></para>
159     /// </summary>
160     public MarkedSequenceCriterionMatcher<TLinkAddress> MarkedSequenceMatcher
161     {
162         [MethodImpl(MethodImplOptions.AggressiveInlining)]
163         get;
164         [MethodImpl(MethodImplOptions.AggressiveInlining)]
165         set;
166     }
167
168     /// <summary>
169     /// <para>
170     /// Gets or sets the links to sequence converter value.
171     /// </para>
172     /// <para></para>
173     /// </summary>
174     public IConverter<IList<TLinkAddress>, TLinkAddress> LinksToSequenceConverter
175     {
176         [MethodImpl(MethodImplOptions.AggressiveInlining)]
177         get;
178         [MethodImpl(MethodImplOptions.AggressiveInlining)]
179         set;
180     }
181
182     /// <summary>
183     /// <para>
184     /// Gets or sets the index value.
185     /// </para>
186     /// <para></para>
187     /// </summary>
188     public ISequenceIndex<TLinkAddress> Index
189     {
190         [MethodImpl(MethodImplOptions.AggressiveInlining)]
191         get;
192         [MethodImpl(MethodImplOptions.AggressiveInlining)]
193         set;
194     }
195
196     /// <summary>
197     /// <para>
198     /// Gets or sets the walker value.
199     /// </para>
200     /// <para></para>
201     /// </summary>
202     public ISequenceWalker<TLinkAddress> Walker
203     {
204         [MethodImpl(MethodImplOptions.AggressiveInlining)]
205         get;
206         [MethodImpl(MethodImplOptions.AggressiveInlining)]
207         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
260                             ↪ link.");
261                     }
262                 }
263             }
264             if (MarkedSequenceMatcher == null)
265             {
266                 MarkedSequenceMatcher = new
267                     ↪ MarkedSequenceCriterionMatcher<TLinkAddress>(links, SequenceMarkerLink);
268             }
269         }
270         var balancedVariantConverter = new BalancedVariantConverter<TLinkAddress>(links);
271         if (UseCompression)
272         {
273             if (LinksToSequenceConverter == null)
274             {
275                 ICounter<TLinkAddress, TLinkAddress> totalSequenceSymbolFrequencyCounter;
276                 if (UseSequenceMarker)
277                 {
278                     totalSequenceSymbolFrequencyCounter = new
279                         ↪ TotalMarkedSequenceSymbolFrequencyCounter<TLinkAddress>(links,
280                             ↪ MarkedSequenceMatcher);
281                 }
282                 else
283                 {
284                     totalSequenceSymbolFrequencyCounter = new
285                         ↪ TotalSequenceSymbolFrequencyCounter<TLinkAddress>(links);
286                 }
287             }
288         }
289     }
290 
```

```

281     }
282     var doubletFrequenciesCache = new LinkFrequenciesCache<TLinkAddress>(links,
    ↪ totalSequenceSymbolFrequencyCounter);
283     var compressingConverter = new CompressingConverter<TLinkAddress>(links,
    ↪ balancedVariantConverter, doubletFrequenciesCache);
284     LinksToSequenceConverter = compressingConverter;
285 }
286 else
287 {
288     if (LinksToSequenceConverter == null)
289     {
290         LinksToSequenceConverter = balancedVariantConverter;
291     }
292 }
293 if (UseIndex && Index == null)
294 {
295     Index = new SequenceIndex<TLinkAddress>(links);
296 }
297 if (Walker == null)
298 {
299     Walker = new RightSequenceWalker<TLinkAddress>(links, new
300     ↪ DefaultStack<TLinkAddress>());
301 }
302 }
303
304 /// <summary>
305 /// <para>
306 /// Validates the options.
307 /// </para>
308 /// <para></para>
309 /// </summary>
310 /// <exception cref="NotSupportedException">
311 /// <para>To use garbage collection UseSequenceMarker option must be on.</para>
312 /// <para></para>
313 /// </exception>
314 [MethodImpl(MethodImplOptions.AggressiveInlining)]
315 public void ValidateOptions()
316 {
317     if (UseGarbageCollection && !UseSequenceMarker)
318     {
319         throw new NotSupportedException("To use garbage collection UseSequenceMarker
    ↪ option must be on.");
320     }
321 }
322 }
323 }

```

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

```

1 using System.Text;
2 using Platform.Reflection;
3
4 namespace Platform.Data.Doublets.Sequences;
5
6 public class TestExtensions
7 {
8     public static string PrettifyBinary<T> (string binaryRepresentation)
9     {
10         var bitsCount = NumericType<T>.BitsSize;
11         var sb = new StringBuilder().Append('0', bitsCount -
    ↪ binaryRepresentation.Length).Append(binaryRepresentation).ToString();
12         for (var i = 4; i < sb.Length; i += 5)
13         {
14             sb = sb.Insert(i, " ");
15         }
16         return sb.ToString();
17     }
18 }

```

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

```

1 using System;
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.Time
8 {

```



```

9      /// <summary>
10     /// <para>
11     /// Represents the date time to long raw number sequence converter.
12     /// </para>
13     /// <para></para>
14     /// </summary>
15     /// <seealso cref="IConverter{DateTime, TLinkAddress}"/>
16     public class DateTimeToLongRawNumberSequenceConverter<TLinkAddress> : IConverter<DateTime,
    ↪ TLinkAddress>
17     {
18         private readonly IConverter<long, TLinkAddress> _int64ToLongRawNumberConverter;
19
20         /// <summary>
21         /// <para>
22         /// Initializes a new <see cref="DateTimeToLongRawNumberSequenceConverter"/> instance.
23         /// </para>
24         /// <para></para>
25         /// </summary>
26         /// <param name="int64ToLongRawNumberConverter">
27         /// <para>A int 64 to long raw number converter.</para>
28         /// <para></para>
29         /// </param>
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         public DateTimeToLongRawNumberSequenceConverter(IConverter<long, TLinkAddress>
    ↪ int64ToLongRawNumberConverter) => _int64ToLongRawNumberConverter =
    ↪ int64ToLongRawNumberConverter;
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 link</para>
45         /// <para></para>
46         /// </returns>
47         [MethodImpl(MethodImplOptions.AggressiveInlining)]
48         public TLinkAddress Convert(DateTime source) =>
    ↪ _int64ToLongRawNumberConverter.Convert(source.ToFileTimeUtc());
49     }
50 }

```

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

```

1  using System;
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.Time
8  {
9      /// <summary>
10     /// <para>
11     /// Represents the long raw number sequence to date time converter.
12     /// </para>
13     /// <para></para>
14     /// </summary>
15     /// <seealso cref="IConverter{TLinkAddress, DateTime}"/>
16     public class LongRawNumberSequenceToDateTimeConverter<TLinkAddress> :
    ↪ IConverter<TLinkAddress, DateTime>
17     {
18         private readonly IConverter<TLinkAddress, long> _longRawNumberConverterToInt64;
19
20         /// <summary>
21         /// <para>
22         /// Initializes a new <see cref="LongRawNumberSequenceToDateTimeConverter"/> instance.
23         /// </para>
24         /// <para></para>
25         /// </summary>
26         /// <param name="longRawNumberConverterToInt64">
27         /// <para>A long raw number converter to int 64.</para>
28         /// <para></para>
29         /// </param>
30         [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     {

```

```

18     private static readonly UncheckedConverter<char, TLinkAddress> _charToAddressConverter =
19         ↪ UncheckedConverter<char, TLinkAddress>.Default;
20     private readonly IConverter<TLinkAddress> _addressToNumberConverter;
21     private readonly TLinkAddress _unicodeSymbolMarker;
22
23     /// <summary>
24     /// <para>
25     /// Initializes a new <see cref="CharToUnicodeSymbolConverter"/> instance.
26     /// </para>
27     /// </summary>
28     /// <param name="links">
29     /// <para>A links.</para>
30     /// </param>
31     /// <param name="addressToNumberConverter">
32     /// <para>A address to number converter.</para>
33     /// </param>
34     /// <param name="unicodeSymbolMarker">
35     /// <para>A unicode symbol marker.</para>
36     /// </param>
37     [MethodImpl(MethodImplOptions.AggressiveInlining)]
38     public CharToUnicodeSymbolConverter(ILinks<TLinkAddress> links, IConverter<TLinkAddress>
39         ↪ addressToNumberConverter, TLinkAddress unicodeSymbolMarker) : base(links)
40     {
41         _addressToNumberConverter = addressToNumberConverter;
42         _unicodeSymbolMarker = unicodeSymbolMarker;
43     }
44
45     /// <summary>
46     /// <para>
47     /// Converts the source.
48     /// </para>
49     /// </summary>
50     /// <param name="source">
51     /// <para>The source.</para>
52     /// </param>
53     /// <returns>
54     /// <para>The link</para>
55     /// </returns>
56     [MethodImpl(MethodImplOptions.AggressiveInlining)]
57     public TLinkAddress Convert(char source)
58     {
59         var unaryNumber =
60             ↪ _addressToNumberConverter.Convert(_charToAddressConverter.Convert(source));
61         return _links.GetOrCreate(unaryNumber, _unicodeSymbolMarker);
62     }
63 }
64
65 }
66
67 }
68 }

```

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

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3 using Platform.Converters;
4 using Platform.Data.Doublets.Sequences.Indexes;
5
6 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8 namespace Platform.Data.Doublets.Unicode
9 {
10     /// <summary>
11     /// <para>
12     /// Represents the string to unicode sequence converter.
13     /// </para>
14     /// </summary>
15     /// <seealso cref="LinksOperatorBase{TLinkAddress}"/>
16     /// <seealso cref="IConverter{string, TLinkAddress}"/>
17     public class StringToUnicodeSequenceConverter<TLinkAddress> :
18         ↪ LinksOperator<TLinkAddress>, IConverter<string, TLinkAddress>
19     {
20         private readonly IConverter<string, IList<TLinkAddress>?>
21             ↪ _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>

```

```

46     /// </returns>
47     [MethodImpl(MethodImplOptions.AggressiveInlining)]
48     public IList<TLinkAddress>? Convert(string source)
49     {
50         var elements = new TLinkAddress[source.Length];
51         for (var i = 0; i < elements.Length; i++)
52         {
53             elements[i] = _charToUnicodeSymbolConverter.Convert(source[i]);
54         }
55         return elements;
56     }
57 }
58 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Globalization;
4  using System.Runtime.CompilerServices;
5  using System.Text;
6  using Platform.Data.Sequences;
7
8  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Data.Doublets.Unicode
11 {
12     /// <summary>
13     /// <para>
14     /// Represents the unicode map.
15     /// </para>
16     /// <para></para>
17     /// </summary>
18     public class UnicodeMap
19     {
20         /// <summary>
21         /// <para>
22         /// The first char link.
23         /// </para>
24         /// <para></para>
25         /// </summary>
26         public static readonly ulong FirstCharLink = 1;
27         /// <summary>
28         /// <para>
29         /// The max value.
30         /// </para>
31         /// <para></para>
32         /// </summary>
33         public static readonly ulong LastCharLink = FirstCharLink + char.MaxValue;
34         /// <summary>
35         /// <para>
36         /// The max value.
37         /// </para>
38         /// <para></para>
39         /// </summary>
40         public static readonly ulong MapSize = 1 + char.MaxValue;
41         private readonly ILinks<ulong> _links;
42         private bool _initialized;
43
44         /// <summary>
45         /// <para>
46         /// Initializes a new <see cref="UnicodeMap"/> instance.
47         /// </para>
48         /// <para></para>
49         /// </summary>
50         /// <param name="links">
51         /// <para>A links.</para>
52         /// <para></para>
53         /// </param>
54         [MethodImpl(MethodImplOptions.AggressiveInlining)]
55         public UnicodeMap(ILinks<ulong> links) => _links = links;
56
57         /// <summary>
58         /// <para>
59         /// Inits the new using the specified links.
60         /// </para>
61         /// <para></para>
62         /// </summary>
63         /// <param name="links">
64         /// <para>The links.</para>

```

```

65     /// <para></para>
66     /// </param>
67     /// <returns>
68     /// <para>The map.</para>
69     /// <para></para>
70     /// </returns>
71     [MethodImpl(MethodImplOptions.AggressiveInlining)]
72     public static UnicodeMap InitNew(ILinks<ulong> links)
73     {
74         var map = new UnicodeMap(links);
75         map.Init();
76         return map;
77     }
78
79     /// <summary>
80     /// <para>
81     /// Inits this instance.
82     /// </para>
83     /// <para></para>
84     /// </summary>
85     /// <exception cref="InvalidOperationException">
86     /// <para>Unable to initialize UTF 16 table.</para>
87     /// <para></para>
88     /// </exception>
89     [MethodImpl(MethodImplOptions.AggressiveInlining)]
90     public void Init()
91     {
92         if (_initialized)
93         {
94             return;
95         }
96         _initialized = true;
97         var firstLink = _links.CreatePoint();
98         if (firstLink != FirstCharLink)
99         {
100             _links.Delete(firstLink);
101         }
102         else
103         {
104             for (var i = FirstCharLink + 1; i <= LastCharLink; i++)
105             {
106                 // From NIL to It (NIL -> Character) transformation meaning, (or infinite
107                 // ↳ amount of NIL characters before actual Character)
108                 var createdLink = _links.CreatePoint();
109                 _links.Update(createdLink, firstLink, createdLink);
110                 if (createdLink != i)
111                 {
112                     throw new InvalidOperationException("Unable to initialize UTF 16
113                     ↳ table.");
114                 }
115             }
116         }
117     }
118
119     // 0 - null link
120     // 1 - nil character (0 character)
121     // ...
122     // 65536 (0(1) + 65535 = 65536 possible values)
123
124     /// <summary>
125     /// <para>
126     /// Creates the char to link using the specified character.
127     /// </para>
128     /// <para></para>
129     /// </summary>
130     /// <param name="character">
131     /// <para>The character.</para>
132     /// <para></para>
133     /// </param>
134     /// <returns>
135     /// <para>The ulong</para>
136     /// <para></para>
137     /// </returns>
138     [MethodImpl(MethodImplOptions.AggressiveInlining)]
139     public static ulong FromCharToLink(char character) => (ulong)character + 1;
140
141     /// <summary>
142     /// <para>

```



```

141     /// Creates the link to char using the specified link.
142     /// </para>
143     /// <para></para>
144     /// </summary>
145     /// <param name="link">
146     /// <para>The link.</para>
147     /// <para></para>
148     /// </param>
149     /// <returns>
150     /// <para>The char</para>
151     /// <para></para>
152     /// </returns>
153     [MethodImpl(MethodImplOptions.AggressiveInlining)]
154     public static char FromLinkToChar(ulong link) => (char)(link - 1);
155
156     /// <summary>
157     /// <para>
158     /// Determines whether is char link.
159     /// </para>
160     /// <para></para>
161     /// </summary>
162     /// <param name="link">
163     /// <para>The link.</para>
164     /// <para></para>
165     /// </param>
166     /// <returns>
167     /// <para>The bool</para>
168     /// <para></para>
169     /// </returns>
170     [MethodImpl(MethodImplOptions.AggressiveInlining)]
171     public static bool IsCharLink(ulong link) => link <= MapSize;
172
173     /// <summary>
174     /// <para>
175     /// Creates the links to string using the specified links list.
176     /// </para>
177     /// <para></para>
178     /// </summary>
179     /// <param name="linksList">
180     /// <para>The links list.</para>
181     /// <para></para>
182     /// </param>
183     /// <returns>
184     /// <para>The string</para>
185     /// <para></para>
186     /// </returns>
187     [MethodImpl(MethodImplOptions.AggressiveInlining)]
188     public static string FromLinksToString(IList<ulong> linksList)
189     {
190         var sb = new StringBuilder();
191         for (int i = 0; i < linksList.Count; i++)
192         {
193             sb.Append(FromLinkToChar(linksList[i]));
194         }
195         return sb.ToString();
196     }
197
198     /// <summary>
199     /// <para>
200     /// Creates the sequence link to string using the specified link.
201     /// </para>
202     /// <para></para>
203     /// </summary>
204     /// <param name="link">
205     /// <para>The link.</para>
206     /// <para></para>
207     /// </param>
208     /// <param name="links">
209     /// <para>The links.</para>
210     /// <para></para>
211     /// </param>
212     /// <returns>
213     /// <para>The string</para>
214     /// <para></para>
215     /// </returns>
216     [MethodImpl(MethodImplOptions.AggressiveInlining)]
217     public static string FromSequenceLinkToString(ulong link, ILinks<ulong> links)
218     {

```

```

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,
224             ↪ element =>
225             {
226                 sb.Append(FromLinkToChar(element));
227                 return true;
228             }
229     }
230     return sb.ToString();
231 }
232
233 /// <summary>
234 /// <para>
235 /// Creates the chars to link array using the specified chars.
236 /// </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,
248     ↪ chars.Length);
249
250 /// <summary>
251 /// <para>
252 /// Creates the chars to link array using the specified chars.
253 /// </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)

```

```

295 {
296     // char array to ulong array
297     var linksSequence = new ulong[sequence.Length];
298     for (var i = 0; i < sequence.Length; i++)
299     {
300         linksSequence[i] = FromCharToLink(sequence[i]);
301     }
302     return linksSequence;
303 }
304
305 /// <summary>
306 /// <para>
307 /// Creates the string to link array groups using the specified sequence.
308 /// </para>
309 /// <para></para>
310 /// </summary>
311 /// <param name="sequence">
312 /// <para>The sequence.</para>
313 /// <para></para>
314 /// </param>
315 /// <returns>
316 /// <para>The result.</para>
317 /// <para></para>
318 /// </returns>
319 [MethodImpl(MethodImplOptions.AggressiveInlining)]
320 public static List<ulong[]> FromStringToLinkArrayGroups(string sequence)
321 {
322     var result = new List<ulong[]>();
323     var offset = 0;
324     while (offset < sequence.Length)
325     {
326         var currentCategory = CharUnicodeInfo.GetUnicodeCategory(sequence[offset]);
327         var relativeLength = 1;
328         var absoluteLength = offset + relativeLength;
329         while (absoluteLength < sequence.Length &&
330             currentCategory ==
331                 CharUnicodeInfo.GetUnicodeCategory(sequence[absoluteLength]))
332         {
333             relativeLength++;
334             absoluteLength++;
335         }
336         // char array to ulong array
337         var innerSequence = new ulong[relativeLength];
338         var maxLength = offset + relativeLength;
339         for (var i = offset; i < maxLength; i++)
340         {
341             innerSequence[i - offset] = FromCharToLink(sequence[i]);
342         }
343         result.Add(innerSequence);
344         offset += relativeLength;
345     }
346     return result;
347 }
348
349 /// <summary>
350 /// <para>
351 /// Creates the link array to link array groups using the specified array.
352 /// </para>
353 /// <para></para>
354 /// </summary>
355 /// <param name="array">
356 /// <para>The array.</para>
357 /// <para></para>
358 /// </param>
359 /// <returns>
360 /// <para>The result.</para>
361 /// <para></para>
362 /// </returns>
363 [MethodImpl(MethodImplOptions.AggressiveInlining)]
364 public static List<ulong[]> FromLinkArrayToLinkArrayGroups(ulong[] array)
365 {
366     var result = new List<ulong[]>();
367     var offset = 0;
368     while (offset < array.Length)
369     {
370         var relativeLength = 1;
371         if (array[offset] <= LastCharLink)
372         {

```

```

372         var currentCategory =
373             ↳ CharUnicodeInfo.GetUnicodeCategory(FromLinkToChar(array[offset]));
374         var absoluteLength = offset + relativeLength;
375         while (absoluteLength < array.Length &&
376             array[absoluteLength] <= LastCharLink &&
377             currentCategory == CharUnicodeInfo.GetUnicodeCategory(FromLinkToChar(
378                 ↳ array[absoluteLength])))
379         {
380             relativeLength++;
381             absoluteLength++;
382         }
383     }
384     else
385     {
386         var absoluteLength = offset + relativeLength;
387         while (absoluteLength < array.Length && array[absoluteLength] > LastCharLink)
388         {
389             relativeLength++;
390             absoluteLength++;
391         }
392         // copy array
393         var innerSequence = new ulong[relativeLength];
394         var maxLength = offset + relativeLength;
395         for (var i = offset; i < maxLength; i++)
396         {
397             innerSequence[i - offset] = array[i];
398         }
399         result.Add(innerSequence);
400         offset += relativeLength;
401     }
402     return result;
403 }
404 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Interfaces;
5  using Platform.Converters;
6  using Platform.Data.Doublets.Sequences.Walkers;
7  using System.Text;
8  using Platform.Data.Doublets.Sequences.CriterionMatchers;
9
10 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
11
12 namespace Platform.Data.Doublets.Unicode
13 {
14     /// <summary>
15     /// <para>
16     /// Represents the unicode sequence to string converter.
17     /// </para>
18     /// <para></para>
19     /// </summary>
20     /// <seealso cref="LinksOperatorBase{TLinkAddress}"/>
21     /// <seealso cref="IConverter{TLinkAddress, string}"/>
22     public class UnicodeSequenceToStringConverter<TLinkAddress> :
23         ↳ LinksOperatorBase<TLinkAddress>, IConverter<TLinkAddress, string>
24     {
25         private readonly ICriterionMatcher<TLinkAddress> _unicodeSequenceCriterionMatcher;
26         private readonly ISequenceWalker<TLinkAddress> _sequenceWalker;
27         private readonly IConverter<TLinkAddress, char> _unicodeSymbolToCharConverter;
28         private readonly TLinkAddress _unicodeSequenceMarker;
29
30         /// <summary>
31         /// <para>
32         /// Initializes a new <see cref="UnicodeSequenceToStringConverter"/> instance.
33         /// </para>
34         /// <para></para>
35         /// </summary>
36         /// <param name="links">
37         /// <para>A links.</para>
38         /// <para></para>
39         /// </param>
40         /// <param name="unicodeSequenceCriterionMatcher">
41         /// <para>A unicode sequence criterion matcher.</para>
42         /// <para></para>

```

```

43     /// </param>
44     /// <param name="sequenceWalker">
45     /// <para>A sequence walker.</para>
46     /// </para></param>
47     /// <param name="unicodeSymbolToCharConverter">
48     /// <para>A unicode symbol to char converter.</para>
49     /// </para></param>
50     /// </param>
51     [MethodImpl(MethodImplOptions.AggressiveInlining)]
52     public UnicodeSequenceToStringConverter(ILinks<TLinkAddress> links,
53     ↪ ICriterionMatcher<TLinkAddress> unicodeSequenceCriterionMatcher,
54     ↪ ISequenceWalker<TLinkAddress> sequenceWalker, IConverter<TLinkAddress, char>
55     ↪ unicodeSymbolToCharConverter, TLinkAddress unicodeSequenceMarker) : base(links)
56     {
57         _unicodeSequenceCriterionMatcher = unicodeSequenceCriterionMatcher;
58         _sequenceWalker = sequenceWalker;
59         _unicodeSymbolToCharConverter = unicodeSymbolToCharConverter;
60         _unicodeSequenceMarker = unicodeSequenceMarker;
61     }
62
63     public UnicodeSequenceToStringConverter(ILinks<TLinkAddress> links,
64     ↪ ISequenceWalker<TLinkAddress> sequenceWalker, IConverter<TLinkAddress, char>
65     ↪ unicodeSymbolToCharConverter, TLinkAddress unicodeSequenceMarker): this(links, new
66     ↪ UnicodeSequenceMatcher<TLinkAddress>(links, unicodeSequenceMarker), sequenceWalker,
67     ↪ unicodeSymbolToCharConverter, unicodeSequenceMarker){}
68
69     /// <summary>
70     /// <para>
71     /// Converts the source.
72     /// </para>
73     /// </summary>
74     /// <param name="source">
75     /// <para>The source.</para>
76     /// </para></param>
77     /// <exception cref="ArgumentOutOfRangeException">
78     /// <para>Specified link is not a unicode sequence.</para>
79     /// </exception>
80     /// <returns>
81     /// <para>The string</para>
82     /// </para></returns>
83     [MethodImpl(MethodImplOptions.AggressiveInlining)]
84     public string Convert(TLinkAddress source)
85     {
86         if (!_unicodeSequenceCriterionMatcher.IsMatched(source))
87         {
88             throw new ArgumentOutOfRangeException(nameof(source), source, "Specified link is
89             ↪ not a unicode sequence.");
90         }
91         if (EqualityComparer<TLinkAddress>.Default.Equals(_unicodeSequenceMarker, source))
92         {
93             return String.Empty;
94         }
95         var sequence = _links.GetSource(source);
96         var sb = new StringBuilder();
97         foreach (var character in _sequenceWalker.Walk(sequence))
98         {
99             sb.Append(_unicodeSymbolToCharConverter.Convert(character));
100         }
101         return sb.ToString();
102     }

```

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

```

1 using System;
2 using System.Runtime.CompilerServices;
3 using Platform.Interfaces;
4 using Platform.Converters;
5
6 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8 namespace Platform.Data.Doublets.Unicode
9 {

```

```

10  /// <summary>
11  /// <para>
12  /// Represents the unicode symbol to char converter.
13  /// </para>
14  /// <para></para>
15  /// </summary>
16  /// <seealso cref="LinksOperatorBase{TLinkAddress}"/>
17  /// <seealso cref="IConverter{TLinkAddress, char}"/>
18  public class UnicodeSymbolToCharConverter<TLinkAddress> : LinksOperatorBase<TLinkAddress>,
    ↳ IConverter<TLinkAddress, char>
19  {
20      private static readonly UncheckedConverter<TLinkAddress, char> _addressToCharConverter =
    ↳ UncheckedConverter<TLinkAddress, char>.Default;
21      private readonly IConverter<TLinkAddress> _numberToAddressConverter;
22      private readonly ICriterionMatcher<TLinkAddress> _unicodeSymbolCriterionMatcher;
23
24      /// <summary>
25      /// <para>
26      /// Initializes a new <see cref="UnicodeSymbolToCharConverter"/> 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="numberToAddressConverter">
35      /// <para>A number to address converter.</para>
36      /// <para></para>
37      /// </param>
38      /// <param name="unicodeSymbolCriterionMatcher">
39      /// <para>A unicode symbol criterion matcher.</para>
40      /// <para></para>
41      /// </param>
42      [MethodImpl(MethodImplOptions.AggressiveInlining)]
43      public UnicodeSymbolToCharConverter(ILinks<TLinkAddress> links, IConverter<TLinkAddress>
    ↳ numberToAddressConverter, ICriterionMatcher<TLinkAddress>
    ↳ unicodeSymbolCriterionMatcher) : base(links)
44      {
45          _numberToAddressConverter = numberToAddressConverter;
46          _unicodeSymbolCriterionMatcher = unicodeSymbolCriterionMatcher;
47      }
48
49      /// <summary>
50      /// <para>
51      /// Converts the source.
52      /// </para>
53      /// <para></para>
54      /// </summary>
55      /// <param name="source">
56      /// <para>The source.</para>
57      /// <para></para>
58      /// </param>
59      /// <exception cref="ArgumentOutOfRangeException">
60      /// <para>Specified link is not a unicode symbol.</para>
61      /// <para></para>
62      /// </exception>
63      /// <returns>
64      /// <para>The char</para>
65      /// <para></para>
66      /// </returns>
67      [MethodImpl(MethodImplOptions.AggressiveInlining)]
68      public char Convert(TLinkAddress source)
69      {
70          if (!_unicodeSymbolCriterionMatcher.IsMatched(source))
71          {
72              throw new ArgumentOutOfRangeException(nameof(source), source, "Specified link is
    ↳ not a unicode symbol.");
73          }
74          return _addressToCharConverter.Convert(_numberToAddressConverter.Convert(_links.GetS_
    ↳ ource(source)));
75      }
76  }
77 }

```

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

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;

```

```

3 using Platform.Collections;
4 using Platform.Converters;
5 using Platform.Data.Doublets.Sequences.Indexes;
6
7 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9 namespace Platform.Data.Doublets.Unicode
10 {
11     /// <summary>
12     /// <para>
13     /// Represents the unicode symbols list to unicode sequence converter.
14     /// </para>
15     /// <para></para>
16     /// </summary>
17     /// <seealso cref="LinksOperatorBase{TLinkAddress}"/>
18     /// <seealso cref="IConverter<IList{TLinkAddress}, TLinkAddress>"/>
19     public class UnicodeSymbolsListToUnicodeSequenceConverter<TLinkAddress> :
20         ↳ LinksOperatorBase<TLinkAddress>, IConverter<IList<TLinkAddress>, TLinkAddress>
21     {
22         private readonly ISequenceIndex<TLinkAddress> _index;
23         private readonly IConverter<IList<TLinkAddress>, TLinkAddress>
24             ↳ _listToSequenceLinkConverter;
25         private readonly TLinkAddress _unicodeSequenceMarker;
26
27         /// <summary>
28         /// <para>
29         /// Initializes a new <see cref="UnicodeSymbolsListToUnicodeSequenceConverter"/>
30         ↳ instance.
31         /// </para>
32         /// <para></para>
33         /// </summary>
34         /// <param name="links">
35         /// <para>A links.</para>
36         /// <para></para>
37         /// </param>
38         /// <param name="index">
39         /// <para>A index.</para>
40         /// <para></para>
41         /// </param>
42         /// <param name="listToSequenceLinkConverter">
43         /// <para>A list to sequence link converter.</para>
44         /// <para></para>
45         /// </param>
46         /// <param name="unicodeSequenceMarker">
47         /// <para>A unicode sequence marker.</para>
48         /// <para></para>
49         /// </param>
50         [MethodImpl(MethodImplOptions.AggressiveInlining)]
51         public UnicodeSymbolsListToUnicodeSequenceConverter(ILinks<TLinkAddress> links,
52             ↳ ISequenceIndex<TLinkAddress> index, IConverter<IList<TLinkAddress>, TLinkAddress>
53             ↳ listToSequenceLinkConverter, TLinkAddress unicodeSequenceMarker) : base(links)
54         {
55             _index = index;
56             _listToSequenceLinkConverter = listToSequenceLinkConverter;
57             _unicodeSequenceMarker = unicodeSequenceMarker;
58         }
59
60         /// <summary>
61         /// <para>
62         /// Initializes a new <see cref="UnicodeSymbolsListToUnicodeSequenceConverter"/>
63         ↳ instance.
64         /// </para>
65         /// <para></para>
66         /// </summary>
67         /// <param name="links">
68         /// <para>A links.</para>
69         /// <para></para>
70         /// </param>
71         /// <param name="listToSequenceLinkConverter">
72         /// <para>A list to sequence link converter.</para>
73         /// <para></para>
74         /// </param>
75         /// <param name="unicodeSequenceMarker">
76         /// <para>A unicode sequence marker.</para>
77         /// <para></para>
78         /// </param>
79         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

74     public UnicodeSymbolsListToUnicodeSequenceConverter(ILinks<TLinkAddress> links,
75         ↳ IConverter<IList<TLinkAddress>, TLinkAddress> listToSequenceLinkConverter,
76         ↳ TLinkAddress unicodeSequenceMarker)
77         : this(links, new Unindex<TLinkAddress>(), listToSequenceLinkConverter,
78             ↳ unicodeSequenceMarker) { }
79
80     /// <summary>
81     /// <para>
82     /// Converts the list.
83     /// </para>
84     /// <para></para>
85     /// </summary>
86     /// <param name="list">
87     /// <para>The list.</para>
88     /// </param>
89     /// <returns>
90     /// <para>The link</para>
91     /// <para></para>
92     /// </returns>
93     [MethodImpl(MethodImplOptions.AggressiveInlining)]
94     public TLinkAddress Convert(IList<TLinkAddress>? list)
95     {
96         if (list.IsNullOrEmpty())
97         {
98             return _unicodeSequenceMarker;
99         }
100         _index.Add(list);
101         var sequence = _listToSequenceLinkConverter.Convert(list);
102         return _links.GetOrCreate(sequence, _unicodeSequenceMarker);
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         /// </param>
25         /// <returns>
26         /// <para>An enumerable of t link</para>
27         /// <para></para>
28         /// </returns>
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         IEnumerable<TLinkAddress> Walk(TLinkAddress sequence);
31     }
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

```



```

9 {
10     /// <summary>
11     /// <para>
12     /// Represents the left sequence walker.
13     /// </para>
14     /// <para></para>
15     /// </summary>
16     /// <seealso cref="SequenceWalkerBase{TLinkAddress}"/>
17     public class LeftSequenceWalker<TLinkAddress> : SequenceWalkerBase<TLinkAddress>
18     {
19         /// <summary>
20         /// <para>
21         /// Initializes a new <see cref="LeftSequenceWalker"/> instance.
22         /// </para>
23         /// <para></para>
24         /// </summary>
25         /// <param name="links">
26         /// <para>A links.</para>
27         /// <para></para>
28         /// </param>
29         /// <param name="stack">
30         /// <para>A stack.</para>
31         /// <para></para>
32         /// </param>
33         /// <param name="isElement">
34         /// <para>A is element.</para>
35         /// <para></para>
36         /// </param>
37         [MethodImpl(MethodImplOptions.AggressiveInlining)]
38         public LeftSequenceWalker(ILinks<TLinkAddress> links, IStack<TLinkAddress> stack,
39             ↪ Func<TLinkAddress, bool> isElement) : base(links, stack, isElement) { }
40
41         /// <summary>
42         /// <para>
43         /// Initializes a new <see cref="LeftSequenceWalker"/> instance.
44         /// </para>
45         /// <para></para>
46         /// </summary>
47         /// <param name="links">
48         /// <para>A links.</para>
49         /// <para></para>
50         /// </param>
51         /// <param name="stack">
52         /// <para>A stack.</para>
53         /// <para></para>
54         /// </param>
55         [MethodImpl(MethodImplOptions.AggressiveInlining)]
56         public LeftSequenceWalker(ILinks<TLinkAddress> links, IStack<TLinkAddress> stack) :
57             ↪ base(links, stack, links.IsPartialPoint) { }
58
59         /// <summary>
60         /// <para>
61         /// Gets the next element after pop using the specified element.
62         /// </para>
63         /// <para></para>
64         /// </summary>
65         /// <param name="element">
66         /// <para>The element.</para>
67         /// <para></para>
68         /// </param>
69         /// <returns>
70         /// <para>The link</para>
71         /// <para></para>
72         /// </returns>
73         [MethodImpl(MethodImplOptions.AggressiveInlining)]
74         protected override TLinkAddress GetNextElementAfterPop(TLinkAddress element) =>
75             ↪ _links.GetSource(element);
76
77         /// <summary>
78         /// <para>
79         /// Gets the next element after push using the specified element.
80         /// </para>
81         /// <para></para>
82         /// </summary>
83         /// <param name="element">
84         /// <para>The element.</para>
85         /// <para></para>
86         /// </param>

```

```

84     /// <returns>
85     /// <para>The link</para>
86     /// <para></para>
87     /// </returns>
88     [MethodImpl(MethodImplOptions.AggressiveInlining)]
89     protected override TLinkAddress GetNextElementAfterPush(TLinkAddress element) =>
90         ↪ _links.GetTarget(element);
91
92     /// <summary>
93     /// <para>
94     /// Walks the contents using the specified element.
95     /// </para>
96     /// <para></para>
97     /// </summary>
98     /// <param name="element">
99     /// <para>The element.</para>
100    /// <para></para>
101    /// </param>
102    /// <returns>
103    /// <para>An enumerable of t link</para>
104    /// <para></para>
105    /// </returns>
106    [MethodImpl(MethodImplOptions.AggressiveInlining)]
107    protected override IEnumerable<TLinkAddress> WalkContents(TLinkAddress element)
108    {
109        var links = _links;
110        var parts = links.GetLink(element);
111        var start = links.Constants.SourcePart;
112        for (var i = parts.Count - 1; i >= start; i--)
113        {
114            var part = parts[i];
115            if (IsElement(part))
116            {
117                yield return part;
118            }
119        }
120    }
121 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  // #define USEARRAYPOOL
8  #if USEARRAYPOOL
9  using Platform.Collections;
10 #endif
11
12 namespace Platform.Data.Doublets.Sequences.Walkers
13 {
14     /// <summary>
15     /// <para>
16     /// Represents the leveled sequence walker.
17     /// </para>
18     /// <para></para>
19     /// </summary>
20     /// <seealso cref="LinksOperatorBase{TLinkAddress}"/>
21     /// <seealso cref="ISequenceWalker{TLinkAddress}"/>
22     public class LeveledSequenceWalker<TLinkAddress> : LinksOperatorBase<TLinkAddress>,
23         ↪ ISequenceWalker<TLinkAddress>
24     {
25         private static readonly EqualityComparer<TLinkAddress> _equalityComparer =
26             ↪ EqualityComparer<TLinkAddress>.Default;
27         private readonly Func<TLinkAddress, bool> _isElement;
28
29         /// <summary>
30         /// <para>
31         /// Initializes a new <see cref="LeveledSequenceWalker"/> instance.
32         /// </para>
33         /// <para></para>
34         /// </summary>
35         /// <param name="links">
36         /// <para>A links.</para>
37         /// <para></para>
38         /// </param>

```

```

37     /// <param name="isElement">
38     /// <para>A is element.</para>
39     /// <para></para>
40     /// </param>
41     [MethodImpl(MethodImplOptions.AggressiveInlining)]
42     public LeveledSequenceWalker(ILinks<TLinkAddress> links, Func<TLinkAddress, bool>
43     ↪ isElement) : base(links) => _isElement = isElement;
44
45     /// <summary>
46     /// <para>
47     /// Initializes a new <see cref="LeveledSequenceWalker"/> instance.
48     /// </para>
49     /// <para></para>
50     /// </summary>
51     /// <param name="links">
52     /// <para>A links.</para>
53     /// <para></para>
54     /// </param>
55     [MethodImpl(MethodImplOptions.AggressiveInlining)]
56     public LeveledSequenceWalker(ILinks<TLinkAddress> links) : base(links) => _isElement =
57     ↪ _links.IsPartialPoint;
58
59     /// <summary>
60     /// <para>
61     /// Walks the sequence.
62     /// </para>
63     /// <para></para>
64     /// </summary>
65     /// <param name="sequence">
66     /// <para>The sequence.</para>
67     /// <para></para>
68     /// </param>
69     /// <returns>
70     /// <para>An enumerable of t link</para>
71     /// <para></para>
72     /// </returns>
73     [MethodImpl(MethodImplOptions.AggressiveInlining)]
74     public IEnumerable<TLinkAddress> Walk(TLinkAddress sequence) => ToArray(sequence);
75
76     /// <summary>
77     /// <para>
78     /// Returns the array using the specified sequence.
79     /// </para>
80     /// <para></para>
81     /// </summary>
82     /// <param name="sequence">
83     /// <para>The sequence.</para>
84     /// <para></para>
85     /// </param>
86     /// <returns>
87     /// <para>The link array</para>
88     /// <para></para>
89     /// </returns>
90     [MethodImpl(MethodImplOptions.AggressiveInlining)]
91     public TLinkAddress[] ToArray(TLinkAddress sequence)
92     {
93         var length = 1;
94         var array = new TLinkAddress[length];
95         array[0] = sequence;
96         if (_isElement(sequence))
97         {
98             return array;
99         }
100         bool hasElements;
101         do
102         {
103             length *= 2;
104             #if USEARRAYPOOL
105             var nextArray = ArrayPool.Allocate<ulong>(length);
106             #else
107             var nextArray = new TLinkAddress[length];
108             #endif
109             hasElements = false;
110             for (var i = 0; i < array.Length; i++)
111             {
112                 var candidate = array[i];
113                 if (_equalityComparer.Equals(array[i], default))
114                 {
115                     continue;

```

```

114     }
115     var doubletOffset = i * 2;
116     if (_isElement(candidate))
117     {
118         nextArray[doubletOffset] = candidate;
119     }
120     else
121     {
122         var links = _links;
123         var link = links.GetLink(candidate);
124         var linkSource = links.GetSource(link);
125         var linkTarget = links.GetTarget(link);
126         nextArray[doubletOffset] = linkSource;
127         nextArray[doubletOffset + 1] = linkTarget;
128         if (!hasElements)
129         {
130             hasElements = !(_isElement(linkSource) && _isElement(linkTarget));
131         }
132     }
133 }
134 #if USEARRAYPOOL
135     if (array.Length > 1)
136     {
137         ArrayPool.Free(array);
138     }
139 #endif
140     array = nextArray;
141 }
142 while (hasElements);
143 var filledElementsCount = CountFilledElements(array);
144 if (filledElementsCount == array.Length)
145 {
146     return array;
147 }
148 else
149 {
150     return CopyFilledElements(array, filledElementsCount);
151 }
152 }
153 [MethodImpl(MethodImplOptions.AggressiveInlining)]
154 private static TLinkAddress[] CopyFilledElements(TLinkAddress[] array, int
    ↪ filledElementsCount)
155 {
156     var finalArray = new TLinkAddress[filledElementsCount];
157     for (int i = 0, j = 0; i < array.Length; i++)
158     {
159         if (!_equalityComparer.Equals(array[i], default))
160         {
161             finalArray[j] = array[i];
162             j++;
163         }
164     }
165     #if USEARRAYPOOL
166         ArrayPool.Free(array);
167     #endif
168     return finalArray;
169 }
170 [MethodImpl(MethodImplOptions.AggressiveInlining)]
171 private static int CountFilledElements(TLinkAddress[] array)
172 {
173     var count = 0;
174     for (var i = 0; i < array.Length; i++)
175     {
176         if (!_equalityComparer.Equals(array[i], default))
177         {
178             count++;
179         }
180     }
181     return count;
182 }
183 }
184 }

```

1.63 ./csharp/Platform.Data.Doublets.Sequences/Walkers/RightSequenceWalker.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
9 {
10     /// <summary>
11     /// <para>
12     /// Represents the right sequence walker.
13     /// </para>
14     /// <para></para>
15     /// </summary>
16     /// <seealso cref="SequenceWalkerBase{TLinkAddress}"/>
17     public class RightSequenceWalker<TLinkAddress> : SequenceWalkerBase<TLinkAddress>
18     {
19         /// <summary>
20         /// <para>
21         /// Initializes a new <see cref="RightSequenceWalker"/> instance.
22         /// </para>
23         /// <para></para>
24         /// </summary>
25         /// <param name="links">
26         /// <para>A links.</para>
27         /// <para></para>
28         /// </param>
29         /// <param name="stack">
30         /// <para>A stack.</para>
31         /// <para></para>
32         /// </param>
33         /// <param name="isElement">
34         /// <para>A is element.</para>
35         /// <para></para>
36         /// </param>
37         [MethodImpl(MethodImplOptions.AggressiveInlining)]
38         public RightSequenceWalker(ILinks<TLinkAddress> links, IStack<TLinkAddress> stack,
39             ↪ Func<TLinkAddress, bool> isElement) : base(links, stack, isElement) { }
40
41         /// <summary>
42         /// <para>
43         /// Initializes a new <see cref="RightSequenceWalker"/> instance.
44         /// </para>
45         /// <para></para>
46         /// </summary>
47         /// <param name="links">
48         /// <para>A links.</para>
49         /// <para></para>
50         /// </param>
51         /// <param name="stack">
52         /// <para>A stack.</para>
53         /// <para></para>
54         /// </param>
55         [MethodImpl(MethodImplOptions.AggressiveInlining)]
56         public RightSequenceWalker(ILinks<TLinkAddress> links, IStack<TLinkAddress> stack) :
57             ↪ base(links, stack, links.IsPartialPoint) { }
58
59         /// <summary>
60         /// <para>
61         /// Gets the next element after pop using the specified element.
62         /// </para>
63         /// <para></para>
64         /// </summary>
65         /// <param name="element">
66         /// <para>The element.</para>
67         /// <para></para>
68         /// </param>
69         /// <returns>
70         /// <para>The link</para>
71         /// <para></para>
72         /// </returns>
73         [MethodImpl(MethodImplOptions.AggressiveInlining)]
74         protected override TLinkAddress GetNextElementAfterPop(TLinkAddress element) =>
75             ↪ _links.GetTarget(element);
76
77         /// <summary>
78         /// <para>
79         /// Gets the next element after push using the specified element.
80         /// </para>
81         /// <para></para>
82         /// </summary>
83         /// <param name="element">

```

```

81     /// <para>The element.</para>
82     /// <para></para>
83     /// </param>
84     /// <returns>
85     /// <para>The link</para>
86     /// <para></para>
87     /// </returns>
88     [MethodImpl(MethodImplOptions.AggressiveInlining)]
89     protected override TLinkAddress GetNextElementAfterPush(TLinkAddress element) =>
90         ↪ _links.GetSource(element);
91
92     /// <summary>
93     /// <para>
94     /// Walks the contents using the specified element.
95     /// </para>
96     /// <para></para>
97     /// </summary>
98     /// <param name="element">
99     /// <para>The element.</para>
100    /// <para></para>
101    /// </param>
102    /// <returns>
103    /// <para>An enumerable of t link</para>
104    /// <para></para>
105    /// </returns>
106    [MethodImpl(MethodImplOptions.AggressiveInlining)]
107    protected override IEnumerable<TLinkAddress> WalkContents(TLinkAddress element)
108    {
109        var parts = _links.GetLink(element);
110        for (var i = _links.Constants.SourcePart; i < parts.Count; i++)
111        {
112            var part = parts[i];
113            if (IsElement(part))
114            {
115                yield return part;
116            }
117        }
118    }
119 }

```

1.64 ./csharp/Platform.Data.Doublets.Sequences/Walkers/SequenceWalkerBase.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
9  {
10     /// <summary>
11     /// <para>
12     /// Represents the sequence walker base.
13     /// </para>
14     /// <para></para>
15     /// </summary>
16     /// <seealso cref="LinksOperatorBase{TLinkAddress}">
17     /// <seealso cref="ISequenceWalker{TLinkAddress}">
18     public abstract class SequenceWalkerBase<TLinkAddress> : LinksOperatorBase<TLinkAddress>,
19         ↪ ISequenceWalker<TLinkAddress>
20     {
21         private readonly IStack<TLinkAddress> _stack;
22         private readonly Func<TLinkAddress, bool> _isElement;
23
24         /// <summary>
25         /// <para>
26         /// Initializes a new <see cref="SequenceWalkerBase"/> 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="stack">
35         /// <para>A stack.</para>
36         /// <para></para>
37         /// </param>

```

```

37     /// <param name="isElement">
38     /// <para>A is element.</para>
39     /// <para></para>
40     /// </param>
41     [MethodImpl(MethodImplOptions.AggressiveInlining)]
42     protected SequenceWalkerBase(ILinks<TLinkAddress> links, IStack<TLinkAddress> stack,
43     ↪ Func<TLinkAddress, bool> isElement) : base(links)
44     {
45         _stack = stack;
46         _isElement = isElement;
47     }
48     /// <summary>
49     /// <para>
50     /// Initializes a new <see cref="SequenceWalkerBase"/> 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="stack">
59     /// <para>A stack.</para>
60     /// <para></para>
61     /// </param>
62     [MethodImpl(MethodImplOptions.AggressiveInlining)]
63     protected SequenceWalkerBase(ILinks<TLinkAddress> links, IStack<TLinkAddress> stack) :
64     ↪ this(links, stack, links.IsPartialPoint) { }
65     /// <summary>
66     /// <para>
67     /// Walks the sequence.
68     /// </para>
69     /// <para></para>
70     /// </summary>
71     /// <param name="sequence">
72     /// <para>The sequence.</para>
73     /// <para></para>
74     /// </param>
75     /// <returns>
76     /// <para>An enumerable of t link</para>
77     /// <para></para>
78     /// </returns>
79     [MethodImpl(MethodImplOptions.AggressiveInlining)]
80     public IEnumerable<TLinkAddress> Walk(TLinkAddress sequence)
81     {
82         _stack.Clear();
83         var element = sequence;
84         if (IsElement(element))
85         {
86             yield return element;
87         }
88         else
89         {
90             while (true)
91             {
92                 if (IsElement(element))
93                 {
94                     if (_stack.IsEmpty)
95                     {
96                         break;
97                     }
98                     element = _stack.Pop();
99                     foreach (var output in WalkContents(element))
100                     {
101                         yield return output;
102                     }
103                     element = GetNextElementAfterPop(element);
104                 }
105                 else
106                 {
107                     _stack.Push(element);
108                     element = GetNextElementAfterPush(element);
109                 }
110             }
111         }
112     }

```

```

113
114     /// <summary>
115     /// <para>
116     /// Determines whether this instance is element.
117     /// </para>
118     /// <para></para>
119     /// </summary>
120     /// <param name="elementLink">
121     /// <para>The element link.</para>
122     /// <para></para>
123     /// </param>
124     /// <returns>
125     /// <para>The bool</para>
126     /// <para></para>
127     /// </returns>
128     [MethodImpl(MethodImplOptions.AggressiveInlining)]
129     protected virtual bool IsElement(TLinkAddress elementLink) => _isElement(elementLink);
130
131     /// <summary>
132     /// <para>
133     /// Gets the next element after pop using the specified element.
134     /// </para>
135     /// <para></para>
136     /// </summary>
137     /// <param name="element">
138     /// <para>The element.</para>
139     /// <para></para>
140     /// </param>
141     /// <returns>
142     /// <para>The link</para>
143     /// <para></para>
144     /// </returns>
145     [MethodImpl(MethodImplOptions.AggressiveInlining)]
146     protected abstract TLinkAddress GetNextElementAfterPop(TLinkAddress element);
147
148     /// <summary>
149     /// <para>
150     /// Gets the next element after push using the specified element.
151     /// </para>
152     /// <para></para>
153     /// </summary>
154     /// <param name="element">
155     /// <para>The element.</para>
156     /// <para></para>
157     /// </param>
158     /// <returns>
159     /// <para>The link</para>
160     /// <para></para>
161     /// </returns>
162     [MethodImpl(MethodImplOptions.AggressiveInlining)]
163     protected abstract TLinkAddress GetNextElementAfterPush(TLinkAddress element);
164
165     /// <summary>
166     /// <para>
167     /// Walks the contents using the specified element.
168     /// </para>
169     /// <para></para>
170     /// </summary>
171     /// <param name="element">
172     /// <para>The element.</para>
173     /// <para></para>
174     /// </param>
175     /// <returns>
176     /// <para>An enumerable of t link</para>
177     /// <para></para>
178     /// </returns>
179     [MethodImpl(MethodImplOptions.AggressiveInlining)]
180     protected abstract IEnumerable<TLinkAddress> WalkContents(TLinkAddress element);
181 }
182 }

```

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

```

1 using System.Collections.Generic;
2 using System.Numerics;
3 using Platform.Data.Doublets.Memory;
4 using Platform.Data.Doublets.Memory.United.Generic;
5 using Platform.Data.Doublets.Numbers.Raw;
6 using Platform.Data.Doublets.Sequences.Converters;

```



```

7 using Platform.Data.Numbers.Raw;
8 using Platform.Memory;
9 using Xunit;
10 using TLinkAddress = System.UInt64;
11
12 namespace Platform.Data.Doublets.Sequences.Tests
13 {
14     public class BigIntegerConvertersTests
15     {
16         public ILinks<TLinkAddress> CreateLinks() => CreateLinks<TLinkAddress>(new
            ↳ IO.TemporaryFile());
17
18         public ILinks<TLinkAddress> CreateLinks<TLinkAddress>(string dataDbFilename)
19         {
20             var linksConstants = new
                ↳ LinksConstants<TLinkAddress>(enableExternalReferencesSupport: true);
21             return new UnitedMemoryLinks<TLinkAddress>(new
                ↳ FileMappedResizableDirectMemory(dataDbFilename),
                ↳ UnitedMemoryLinks<TLinkAddress>.DefaultLinksSizeStep, linksConstants,
                ↳ IndexTreeType.Default);
22         }
23
24         [Fact]
25         public void DecimalMaxValueTest()
26         {
27             var links = CreateLinks();
28             BigInteger bigInteger = new(decimal.MaxValue);
29             TLinkAddress negativeNumberMarker = links.Create();
30             AddressToRawNumberConverter<TLinkAddress> addressToRawNumberConverter = new();
31             RawNumberToAddressConverter<TLinkAddress> numberToAddressConverter = new();
32             BalancedVariantConverter<TLinkAddress> listToSequenceConverter = new(links);
33             BigIntegerToRawNumberSequenceConverter<TLinkAddress>
                ↳ bigIntegerToRawNumberSequenceConverter = new(links, addressToRawNumberConverter,
                ↳ listToSequenceConverter, negativeNumberMarker);
34             RawNumberSequenceToBigIntegerConverter<TLinkAddress>
                ↳ rawNumberSequenceToBigIntegerConverter = new(links, numberToAddressConverter,
                ↳ negativeNumberMarker);
35             var bigIntSequence = bigIntegerToRawNumberSequenceConverter.Convert(bigInteger);
36             var bigIntFromSequence =
                ↳ rawNumberSequenceToBigIntegerConverter.Convert(bigIntSequence);
37             Assert.Equal(bigInteger, bigIntFromSequence);
38         }
39
40         [Fact]
41         public void DecimalMinValueTest()
42         {
43             var links = CreateLinks();
44             BigInteger bigInteger = new(decimal.MinValue);
45             TLinkAddress negativeNumberMarker = links.Create();
46             AddressToRawNumberConverter<TLinkAddress> addressToRawNumberConverter = new();
47             RawNumberToAddressConverter<TLinkAddress> numberToAddressConverter = new();
48             BalancedVariantConverter<TLinkAddress> listToSequenceConverter = new(links);
49             BigIntegerToRawNumberSequenceConverter<TLinkAddress>
                ↳ bigIntegerToRawNumberSequenceConverter = new(links, addressToRawNumberConverter,
                ↳ listToSequenceConverter, negativeNumberMarker);
50             RawNumberSequenceToBigIntegerConverter<TLinkAddress>
                ↳ rawNumberSequenceToBigIntegerConverter = new(links, numberToAddressConverter,
                ↳ negativeNumberMarker);
51             var bigIntSequence = bigIntegerToRawNumberSequenceConverter.Convert(bigInteger);
52             var bigIntFromSequence =
                ↳ rawNumberSequenceToBigIntegerConverter.Convert(bigIntSequence);
53             Assert.Equal(bigInteger, bigIntFromSequence);
54         }
55
56         [Fact]
57         public void ZeroValueTest()
58         {
59             var links = CreateLinks();
60             BigInteger bigInteger = new(0);
61             TLinkAddress negativeNumberMarker = links.Create();
62             AddressToRawNumberConverter<TLinkAddress> addressToRawNumberConverter = new();
63             RawNumberToAddressConverter<TLinkAddress> numberToAddressConverter = new();
64             BalancedVariantConverter<TLinkAddress> listToSequenceConverter = new(links);
65             BigIntegerToRawNumberSequenceConverter<TLinkAddress>
                ↳ bigIntegerToRawNumberSequenceConverter = new(links, addressToRawNumberConverter,
                ↳ listToSequenceConverter, negativeNumberMarker);

```

```

66         RawNumberSequenceToBigIntegerConverter<TLinkAddress>
        ↪ rawNumberSequenceToBigIntegerConverter = new(links, numberToAddressConverter,
        ↪ negativeNumberMarker);
67     var bigIntSequence = bigIntegerToRawNumberSequenceConverter.Convert(bigInteger);
68     var bigIntFromSequence =
        ↪ rawNumberSequenceToBigIntegerConverter.Convert(bigIntSequence);
69     Assert.Equal(bigInteger, bigIntFromSequence);
70 }
71
72 [Fact]
73 public void OneValueTest()
74 {
75     var links = CreateLinks();
76     BigInteger bigInteger = new(1);
77     TLinkAddress negativeNumberMarker = links.Create();
78     AddressToRawNumberConverter<TLinkAddress> addressToRawNumberConverter = new();
79     RawNumberToAddressConverter<TLinkAddress> numberToAddressConverter = new();
80     BalancedVariantConverter<TLinkAddress> listToSequenceConverter = new(links);
81     BigIntegerToRawNumberSequenceConverter<TLinkAddress>
        ↪ bigIntegerToRawNumberSequenceConverter = new(links, addressToRawNumberConverter,
        ↪ listToSequenceConverter, negativeNumberMarker);
82     RawNumberSequenceToBigIntegerConverter<TLinkAddress>
        ↪ rawNumberSequenceToBigIntegerConverter = new(links, numberToAddressConverter,
        ↪ negativeNumberMarker);
83     var bigIntSequence = bigIntegerToRawNumberSequenceConverter.Convert(bigInteger);
84     var bigIntFromSequence =
        ↪ rawNumberSequenceToBigIntegerConverter.Convert(bigIntSequence);
85     Assert.Equal(bigInteger, bigIntFromSequence);
86 }
87 }
88 }

```

1.66 ./csharp/Platform.Data.Doublets.Sequences.Tests/ByteConvertersTests.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.Lists;
8  using Platform.Collections.Stacks;
9  using Platform.Converters;
10 using Platform.Data.Doublets.CriterionMatchers;
11 using Platform.Data.Doublets.Memory;
12 using Platform.Data.Doublets.Memory.United.Generic;
13 using Platform.Data.Doublets.Numbers.Raw;
14 using Platform.Data.Doublets.Sequences.Converters;
15 using Platform.Data.Doublets.Sequences.Numbers.Byte;
16 using Platform.Data.Doublets.Sequences.Walkers;
17 using Platform.Data.Doublets.Unicode;
18 using Platform.Data.Numbers.Raw;
19 using Platform.Memory;
20 using Platform.Numbers;
21 using Platform.Reflection;
22 using Xunit;
23 using TLinkAddress = System.UInt16;
24
25 namespace Platform.Data.Doublets.Sequences.Tests
26 {
27     public class ByteConvertersTests
28     {
29         private readonly ILinks<TLinkAddress> Storage;
30         private static readonly AddressToRawNumberConverter<TLinkAddress>
        ↪ _addressToRawNumberConverter = new();
31         private static readonly RawNumberToAddressConverter<TLinkAddress>
        ↪ _rawNumberToAddressConverter = new();
32         private readonly BalancedVariantConverter<TLinkAddress> _listToSequenceConverter;
33         private readonly ByteListToRawSequenceConverter<TLinkAddress>
        ↪ _byteListToRawSequenceConverter;
34         private readonly RawSequenceToByteListConverter<TLinkAddress>
        ↪ _rawSequenceToByteListConverter;
35
36         public ByteConvertersTests()
37         {
38             var linksConstants = new
        ↪ LinksConstants<TLinkAddress>(enableExternalReferencesSupport: true);
39             var storageFileName = new IO.TemporaryFile().Filename;
40             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)]

```

```

101 [InlineData(12)]
102 [InlineData(13)]
103 [InlineData(14)]
104 [InlineData(15)]
105 [InlineData(16)]
106 [InlineData(17)]
107 [InlineData(18)]
108 [InlineData(19)]
109 [InlineData(20)]
110 [InlineData(21)]
111 [InlineData(22)]
112 [InlineData(23)]
113 [InlineData(24)]
114 [InlineData(25)]
115 [InlineData(26)]
116 [InlineData(27)]
117 [InlineData(28)]
118 [InlineData(29)]
119 [InlineData(30)]
120 [InlineData(31)]
121 [InlineData(32)]
122 [Theory]
123 public void RandomArrayTest(int length)
124 {
125     var byteArray = GetRandomArray(length);
126     Test(byteArray);
127 }
128
129 public void Test(byte[] byteArray)
130 {
131     var byteListRawSequence = _byteListToRawSequenceConverter.Convert(byteArray);
132     Console.WriteLine();
133     var byteListFromConverter =
134         ↪ _rawSequenceToByteListConverter.Convert(byteListRawSequence);
135     Console.WriteLine("Original");
136     foreach (var b in byteArray)
137     {
138         Console.WriteLine(TestExtensions.PrettifyBinary<byte>(Convert.ToString(b, 2)));
139     }
140     Console.WriteLine();
141     Console.WriteLine("From converter:");
142     foreach (var b in byteListFromConverter)
143     {
144         Console.WriteLine(TestExtensions.PrettifyBinary<byte>(Convert.ToString(b, 2)));
145     }
146     Assert.Equal(byteArray, byteListFromConverter.ToArray());
147 }
148
149 }
150 }

```

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

```

1 using System.Collections.Generic;
2 using Platform.Collections.Stacks;
3 using Platform.Data.Doublets.Memory;
4 using Platform.Data.Doublets.Memory.United.Generic;
5 using Platform.Data.Doublets.Sequences;
6 using Platform.Data.Doublets.Sequences.HeightProviders;
7 using Platform.Data.Numbers.Raw;
8 using Platform.Interfaces;
9 using Platform.Memory;
10 using Platform.Numbers;
11 using Xunit;
12 using Xunit.Abstractions;
13 using TLinkAddress = System.UInt64;
14
15 namespace Platform.Data.Doublets.Sequences.Tests
16 {
17     public class DefaultSequenceAppenderTests
18     {
19         private readonly ITestOutputHelper _output;
20
21         public DefaultSequenceAppenderTests(ITestOutputHelper output)
22         {
23             _output = output;
24         }
25         public static ILinks<TLinkAddress> CreateLinks() => CreateLinks<TLinkAddress>(new
26             ↪ IO.TemporaryFile());

```

```

27     public static ILinks<TLinkAddress> CreateLinks<TLinkAddress>(string dataDBFilename)
28     {
29         var linksConstants = new
30         ↪ LinksConstants<TLinkAddress>(enableExternalReferencesSupport: true);
31         return new UnitedMemoryLinks<TLinkAddress>(new
32         ↪ FileMappedResizableDirectMemory(dataDBFilename),
33         ↪ UnitedMemoryLinks<TLinkAddress>.DefaultLinksSizeStep, linksConstants,
34         ↪ IndexTreeType.Default);
35     }
36
37     public class ValueCriterionMatcher<TLinkAddress> : ICriterionMatcher<TLinkAddress>
38     {
39         public readonly ILinks<TLinkAddress> Links;
40         public readonly TLinkAddress Marker;
41         public ValueCriterionMatcher(ILinks<TLinkAddress> links, TLinkAddress marker)
42         {
43             Links = links;
44             Marker = marker;
45         }
46
47         public bool IsMatched(TLinkAddress link) =>
48         ↪ EqualityComparer<TLinkAddress>.Default.Equals(Links.GetSource(link), Marker);
49     }
50
51     [Fact]
52     public void AppendArrayBug()
53     {
54         ILinks<TLinkAddress> links = CreateLinks();
55         TLinkAddress zero = default;
56         var markerIndex = Arithmetic.Increment(zero);
57         var meaningRoot = links.GetOrCreate(markerIndex, markerIndex);
58         var sequence = links.Create();
59         sequence = links.Update(sequence, meaningRoot, sequence);
60         var appendant = links.Create();
61         appendant = links.Update(appendant, meaningRoot, appendant);
62         ValueCriterionMatcher<TLinkAddress> valueCriterionMatcher = new(links, meaningRoot);
63         DefaultSequenceRightHeightProvider<ulong> defaultSequenceRightHeightProvider =
64         ↪ new(links, valueCriterionMatcher);
65         DefaultSequenceAppender<TLinkAddress> defaultSequenceAppender = new(links, new
66         ↪ DefaultStack<ulong>(), defaultSequenceRightHeightProvider);
67         var newArray = defaultSequenceAppender.Append(sequence, appendant);
68         var output = links.FormatStructure(newArray, link => link.IsFullPoint(), true);
69         Assert.Equal("(4:(2:1 2) (3:1 3))", output);
70     }
71 }
72
73 }
74
75 }

```

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;

```

```

10 // using Platform.Data.Doublets.Sequences.Frequencies.Counters;
11 // using Platform.Data.Doublets.Sequences.Converters;
12 // using Platform.Data.Doublets.PropertyOperators;
13 // using Platform.Data.Doublets.Incrementers;
14 // using Platform.Data.Doublets.Sequences.Walkers;
15 // using Platform.Data.Doublets.Sequences.Indexes;
16 // using Platform.Data.Doublets.Unicode;
17 // using Platform.Data.Doublets.Numbers.Unary;
18 // using Platform.Data.Doublets.Decorators;
19 // using Platform.Data.Doublets.Memory.United.Specific;
20 // using Platform.Data.Doublets.Memory;
21 //
22 // namespace Platform.Data.Doublets.Sequences.Tests
23 // {
24 //     public static class OptimalVariantSequenceTests
25 //     {
26 //         private static readonly string _sequenceExample = "зеленела зелёная зелень";
27 //         private static readonly string _loremIpsumExample = @"Lorem ipsum dolor sit amet,
↳ consectetur adipiscing elit, sed do eiusmod tempor incididunt ut labore et dolore magna
↳ aliqua.
28 // Facilisi nullam vehicula ipsum a arcu cursus vitae congue mauris.
29 // Et malesuada fames ac turpis egestas sed.
30 // Eget velit aliquet sagittis id consectetur purus.
31 // Dignissim cras tincidunt lobortis feugiat vivamus.
32 // Vitae aliquet nec ullamcorper sit.
33 // Lectus quam id leo in vitae.
34 // Tortor dignissim convallis aenean et tortor at risus viverra adipiscing.
35 // Sed risus ultricies tristique nulla aliquet enim tortor at auctor.
36 // Integer eget aliquet nibh praesent tristique.
37 // Vitae congue eu consequat ac felis donec et odio.
38 // Tristique et egestas quis ipsum suspendisse.
39 // Suspendisse potenti nullam ac tortor vitae purus faucibus ornare.
40 // Nulla facilisi etiam dignissim diam quis enim lobortis scelerisque.
41 // Imperdiet proin fermentum leo vel orci.
42 // In ante metus dictum at tempor commodo.
43 // Nisi lacus sed viverra tellus in.
44 // Quam vulputate dignissim suspendisse in.
45 // Elit scelerisque mauris pellentesque pulvinar pellentesque habitant morbi tristique senectus.
46 // Gravida cum sociis natoque penatibus et magnis dis parturient.
47 // Risus quis varius quam quisque id diam.
48 // Congue nisi vitae suscipit tellus mauris a diam maecenas.
49 // Eget nunc scelerisque viverra mauris in aliquam sem fringilla.
50 // Pharetra vel turpis nunc eget lorem dolor sed viverra.
51 // Mattis pellentesque id nibh tortor id aliquet.
52 // Purus non enim praesent elementum facilisis leo vel.
53 // Etiam sit amet nisl purus in mollis nunc sed.
54 // Tortor at auctor urna nunc id cursus metus aliquam.
55 // Volutpat odio facilisis mauris sit amet.
56 // Turpis egestas pretium aenean pharetra magna ac placerat.
57 // Fermentum dui faucibus in ornare quam viverra orci sagittis eu.
58 // Porttitor leo a diam sollicitudin tempor id eu.
59 // Volutpat sed cras ornare arcu dui.
60 // Ut aliquam purus sit amet luctus venenatis lectus magna.
61 // Aliquet risus feugiat in ante metus dictum at.
62 // Mattis nunc sed blandit libero.
63 // Elit pellentesque habitant morbi tristique senectus et netus.
64 // Nibh sit amet commodo nulla facilisi nullam vehicula ipsum a.
65 // Enim sit amet venenatis urna cursus eget nunc scelerisque viverra.
66 // Amet venenatis urna cursus eget nunc scelerisque viverra mauris in.
67 // Diam donec adipiscing tristique risus nec feugiat.
68 // Pulvinar mattis nunc sed blandit libero volutpat.
69 // Cras fermentum odio eu feugiat pretium nibh ipsum.
70 // In nulla posuere sollicitudin aliquam ultrices sagittis orci a.
71 // Mauris pellentesque pulvinar pellentesque habitant morbi tristique senectus et.
72 // A iaculis at erat pellentesque.
73 // Morbi blandit cursus risus at ultrices mi tempus imperdiet nulla.
74 // Eget lorem dolor sed viverra ipsum nunc.
75 // Leo a diam sollicitudin tempor id eu.
76 // Interdum consectetur libero id faucibus nisl tincidunt eget nullam non.";
77 //
78 //     [Fact]
79 //     public static void LinksBasedFrequencyStoredOptimalVariantSequenceTest()
80 //     {
81 //         using (var scope = new TempLinksTestScope(useSequences: false))
82 //         {
83 //             var links = scope.Links;
84 //             var constants = links.Constants;
85 //

```

```

86 //         links.UseUnicode();
87 //
88 //         var sequence = UnicodeMap.FromStringToLinkArray(_sequenceExample);
89 //
90 //         var meaningRoot = links.CreatePoint();
91 //         var unaryOne = links.CreateAndUpdate(meaningRoot, constants.Itself);
92 //         var frequencyMarker = links.CreateAndUpdate(meaningRoot, constants.Itself);
93 //         var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot,
↳ constants.Itself);
94 //
95 //         var unaryNumberToAddressConverter = new
↳ UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);
96 //         var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links,
↳ unaryOne);
97 //         var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
↳ frequencyMarker, unaryOne, unaryNumberIncrementer);
98 //         var frequencyPropertyOperator = new PropertyOperator<ulong>(links,
↳ frequencyPropertyMarker, frequencyMarker);
99 //         var index = new FrequencyIncrementingSequenceIndex<ulong>(links,
↳ frequencyPropertyOperator, frequencyIncrementer);
100 //         var linkToItsFrequencyNumberConverter = new
↳ LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
↳ unaryNumberToAddressConverter);
101 //         var sequenceToItsLocalElementLevelsConverter = new
↳ SequenceToItsLocalElementLevelsConverter<ulong>(links, linkToItsFrequencyNumberConverter);
102 //         var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
↳ sequenceToItsLocalElementLevelsConverter);
103 //
104 //         var sequences = new Sequences(links, new SequencesOptions<ulong>() { Walker =
↳ new LeveledSequenceWalker<ulong>(links) });
105 //
106 //         ExecuteTest(sequences, sequence, sequenceToItsLocalElementLevelsConverter,
↳ index, optimalVariantConverter);
107 //     }
108 // }
109 //
110 // [Fact]
111 // public static void DictionaryBasedFrequencyStoredOptimalVariantSequenceTest()
112 // {
113 //     using (var scope = new TempLinksTestScope(useSequences: false))
114 //     {
115 //         var links = scope.Links;
116 //
117 //         links.UseUnicode();
118 //
119 //         var sequence = UnicodeMap.FromStringToLinkArray(_sequenceExample);
120 //
121 //         var totalSequenceSymbolFrequencyCounter = new
↳ TotalSequenceSymbolFrequencyCounter<ulong>(links);
122 //
123 //         var linkFrequenciesCache = new LinkFrequenciesCache<ulong>(links,
↳ totalSequenceSymbolFrequencyCounter);
124 //
125 //         var index = new
↳ CachedFrequencyIncrementingSequenceIndex<ulong>(linkFrequenciesCache);
126 //         var linkToItsFrequencyNumberConverter = new
↳ FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<ulong>(linkFrequenciesCache);
127 //
128 //         var sequenceToItsLocalElementLevelsConverter = new
↳ SequenceToItsLocalElementLevelsConverter<ulong>(links, linkToItsFrequencyNumberConverter);
129 //         var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
↳ sequenceToItsLocalElementLevelsConverter);
130 //
131 //         var sequences = new Sequences(links, new SequencesOptions<ulong>() { Walker =
↳ new LeveledSequenceWalker<ulong>(links) });
132 //
133 //         ExecuteTest(sequences, sequence, sequenceToItsLocalElementLevelsConverter,
↳ index, optimalVariantConverter);
134 //     }
135 // }
136 // private static void ExecuteTest(Sequences sequences, ulong[] sequence,
↳ SequenceToItsLocalElementLevelsConverter<ulong> sequenceToItsLocalElementLevelsConverter,
↳ ISequenceIndex<ulong> index, OptimalVariantConverter<ulong> optimalVariantConverter)
137 // {
138 //     index.Add(sequence);
139 // }

```

```

140 //         var optimalVariant = optimalVariantConverter.Convert(sequence);
141 //
142 //         var readSequence1 = sequences.ToList(optimalVariant);
143 //
144 //         Assert.True(sequence.SequenceEqual(readSequence1));
145 //     }
146 //
147 //     [Fact]
148 //     public static void SavedSequencesOptimizationTest()
149 //     {
150 //         LinksConstants<ulong> constants = new LinksConstants<ulong>((1, long.MaxValue),
151 //         ↪ (long.MaxValue + 1UL, ulong.MaxValue));
152 //
153 //         using (var memory = new HeapResizableDirectMemory())
154 //         ↪ using (var disposableLinks = new UInt64UnitedMemoryLinks(memory,
155 //         ↪ UInt64UnitedMemoryLinks.DefaultLinksSizeStep, constants, IndexTreeType.Default))
156 //         ↪ {
157 //             var links = new UInt64Links(disposableLinks);
158 //
159 //             var root = links.CreatePoint();
160 //
161 //             //var numberToAddressConverter = new RawNumberToAddressConverter<ulong>();
162 //             var addressToNumberConverter = new AddressToRawNumberConverter<ulong>();
163 //
164 //             var unicodeSymbolMarker = links.GetOrCreate(root,
165 //             ↪ addressToNumberConverter.Convert(1));
166 //             var unicodeSequenceMarker = links.GetOrCreate(root,
167 //             ↪ addressToNumberConverter.Convert(2));
168 //
169 //             var totalSequenceSymbolFrequencyCounter = new
170 //             ↪ TotalSequenceSymbolFrequencyCounter<ulong>(links);
171 //             var linkFrequenciesCache = new LinkFrequenciesCache<ulong>(links,
172 //             ↪ totalSequenceSymbolFrequencyCounter);
173 //             var index = new
174 //             ↪ CachedFrequencyIncrementingSequenceIndex<ulong>(linkFrequenciesCache);
175 //             var linkToItsFrequencyNumberConverter = new
176 //             ↪ FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<ulong>(linkFrequenciesCache);
177 //             var sequenceToItsLocalElementLevelsConverter = new
178 //             ↪ SequenceToItsLocalElementLevelsConverter<ulong>(links, linkToItsFrequencyNumberConverter);
179 //             var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
180 //             ↪ sequenceToItsLocalElementLevelsConverter);
181 //
182 //             var walker = new RightSequenceWalker<ulong>(links, new DefaultStack<ulong>(),
183 //             ↪ (link) => constants.IsExternalReference(link) || links.IsPartialPoint(link));
184 //
185 //             var unicodeSequencesOptions = new SequencesOptions<ulong>()
186 //             ↪ {
187 //                 UseSequenceMarker = true,
188 //                 SequenceMarkerLink = unicodeSequenceMarker,
189 //                 UseIndex = true,
190 //                 Index = index,
191 //                 LinksToSequenceConverter = optimalVariantConverter,
192 //                 Walker = walker,
193 //                 UseGarbageCollection = true
194 //             };
195 //
196 //             var unicodeSequences = new Sequences(new SynchronizedLinks<ulong>(links),
197 //             ↪ unicodeSequencesOptions);
198 //
199 //             // Create some sequences
200 //             var strings = _loremIpsumExample.Split(new[] { '\n', '\r' },
201 //             ↪ StringSplitOptions.RemoveEmptyEntries);
202 //             var arrays = strings.Select(x => x.Select(y =>
203 //             ↪ addressToNumberConverter.Convert(y)).ToArray()).ToArray();
204 //             for (int i = 0; i < arrays.Length; i++)
205 //             ↪ {
206 //                 unicodeSequences.Create(arrays[i].ShiftRight());
207 //             }
208 //
209 //             var linksCountAfterCreation = links.Count();
210 //
211 //             // get list of sequences links
212 //             // for each sequence link
213 //             //     create new sequence version
214 //             //     if new sequence is not the same as sequence link
215 //             //     delete sequence link

```



```

202 //                // collect garbage
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 { Walker =
→ new LeveledSequenceWalker(links) });
//
//                 var sequence = new ulong[sequenceLength];
//                 for (var i = 0; i < sequenceLength; i++)
//                 {
//                     sequence[i] = links.Create();
//                 }
//
//                 var balancedVariantConverter = new BalancedVariantConverter(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();
//                 SequenceWalker.WalkRight(balancedVariant,
//                                         links.GetSource,
//                                         links.GetTarget,
//                                         links.IsPartialPoint,
//                                         readSequence2.Add);
//             }
//         }
//     }
// }
```

```

46 //                 sw3.Stop();
47 //
48 //                 Assert.True(sequence.SequenceEqual(readSequence1));
49 //
50 //                 Assert.True(sequence.SequenceEqual(readSequence2));
51 //
52 //                 // Assert.True(sw2.Elapsed < sw3.Elapsed);
53 //
54 //                 Console.WriteLine($"Stack-based walker: {sw3.Elapsed}, Level-based reader:
↵ {sw2.Elapsed}");
55 //
56 //                 for (var i = 0; i < sequenceLength; i++)
57 //                 {
58 //                     links.Delete(sequence[i]);
59 //                 }
60 //             }
61 //         }
62 //     }
63 // }

```

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

```

1 // using System;
2 // using System.Collections.Generic;
3 // using System.Diagnostics;
4 // using System.Linq;
5 // using Xunit;
6 // using Platform.Collections;
7 // using Platform.Collections.Arrays;
8 // using Platform.Random;
9 // using Platform.IO;
10 // using Platform.Singletons;
11 // using Platform.Data.Doublets.Sequences;
12 // using Platform.Data.Doublets.Sequences.Frequencies.Cache;
13 // using Platform.Data.Doublets.Sequences.Frequencies.Counters;
14 // using Platform.Data.Doublets.Sequences.Converters;
15 // using Platform.Data.Doublets.Unicode;
16 //
17 // namespace Platform.Data.Doublets.Sequences.Tests
18 // {
19 //     public static class SequencesTests
20 //     {
21 //         private static readonly LinksConstants<ulong> _constants =
↵ Default<LinksConstants<ulong>>.Instance;
22 //
23 //         static SequencesTests()
24 //         {
25 //             // Trigger static constructor to not mess with performance measurements
26 //             _ = BitString.GetBitMaskFromIndex(1);
27 //         }
28 //
29 //         [Fact]
30 //         public static void CreateAllVariantsTest()
31 //         {
32 //             const long sequenceLength = 8;
33 //
34 //             using (var scope = new TempLinksTestScope(useSequences: true))
35 //             {
36 //                 var links = scope.Links;
37 //                 var sequences = scope.Sequences;
38 //
39 //                 var sequence = new ulong[sequenceLength];
40 //                 for (var i = 0; i < sequenceLength; i++)
41 //                 {
42 //                     sequence[i] = links.Create();
43 //                 }
44 //
45 //                 var sw1 = Stopwatch.StartNew();
46 //                 var results1 = sequences.CreateAllVariants1(sequence); sw1.Stop();
47 //
48 //                 var sw2 = Stopwatch.StartNew();
49 //                 var results2 = sequences.CreateAllVariants2(sequence); sw2.Stop();
50 //
51 //                 Assert.True(results1.Count > results2.Length);
52 //                 Assert.True(sw1.Elapsed > sw2.Elapsed);
53 //
54 //                 for (var i = 0; i < sequenceLength; i++)
55 //                 {
56 //                     links.Delete(sequence[i]);

```

```

57 //      }
58 //
59 //      Assert.True(links.Count() == 0);
60 //  }
61 //  }
62 //
63 //  //[Fact]
64 //  //public void CUDTest()
65 //  //{
66 //      //  var tempFilename = Path.GetTempFileName();
67 //
68 //      //  const long sequenceLength = 8;
69 //
70 //      //  const ulong itself = LinksConstants.Itself;
71 //
72 //      //  using (var memoryAdapter = new ResizableDirectMemoryLinks(tempFilename,
↵ DefaultLinksSizeStep))
73 //      //      using (var links = new Links(memoryAdapter))
74 //      //      {
75 //          //          var sequence = new ulong[sequenceLength];
76 //          //          for (var i = 0; i < sequenceLength; i++)
77 //          //              sequence[i] = links.Create(itself, itself);
78 //
79 //          //          SequencesOptions o = new SequencesOptions();
80 //
81 //          //  TODO: Из числа в bool значения o.UseSequenceMarker = ((value & 1) != 0)
82 //          //          o.
83 //
84 //          //          var sequences = new Sequences(links);
85 //
86 //          //          var sw1 = Stopwatch.StartNew();
87 //          //          var results1 = sequences.CreateAllVariants1(sequence); sw1.Stop();
88 //
89 //          //          var sw2 = Stopwatch.StartNew();
90 //          //          var results2 = sequences.CreateAllVariants2(sequence); sw2.Stop();
91 //
92 //          //          Assert.True(results1.Count > results2.Length);
93 //          //          Assert.True(sw1.Elapsed > sw2.Elapsed);
94 //
95 //          //          for (var i = 0; i < sequenceLength; i++)
96 //          //              links.Delete(sequence[i]);
97 //      //      }
98 //
99 //      //      File.Delete(tempFilename);
100 //  //  }
101 //
102 //  [Fact]
103 //  public static void AllVariantsSearchTest()
104 //  {
105 //      //      const long sequenceLength = 8;
106 //
107 //      //      using (var scope = new TempLinksTestScope(useSequences: true))
108 //      //      {
109 //          //          var links = scope.Links;
110 //          //          var sequences = scope.Sequences;
111 //
112 //          //          var sequence = new ulong[sequenceLength];
113 //          //          for (var i = 0; i < sequenceLength; i++)
114 //          //          {
115 //              //              sequence[i] = links.Create();
116 //          //          }
117 //
118 //          //          var createResults =
↵ sequences.CreateAllVariants2(sequence).Distinct().ToArray();
119 //
120 //          //          //for (int i = 0; i < createResults.Length; i++)
121 //          //          //      sequences.Create(createResults[i]);
122 //
123 //          //          var sw0 = Stopwatch.StartNew();
124 //          //          var searchResults0 = sequences.GetAllMatchingSequences0(sequence); sw0.Stop();
125 //
126 //          //          var sw1 = Stopwatch.StartNew();
127 //          //          var searchResults1 = sequences.GetAllMatchingSequences1(sequence); sw1.Stop();
128 //
129 //          //          var sw2 = Stopwatch.StartNew();
130 //          //          var searchResults2 = sequences.Each1(sequence); sw2.Stop();
131 //

```

```

132 //         var sw3 = Stopwatch.StartNew();
133 //         var searchResults3 = sequences.Each(sequence.ShiftRight()); sw3.Stop();
134 //
135 //         var intersection0 = createResults.Intersect(searchResults0).ToList();
136 //         Assert.True(intersection0.Count == searchResults0.Count);
137 //         Assert.True(intersection0.Count == createResults.Length);
138 //
139 //         var intersection1 = createResults.Intersect(searchResults1).ToList();
140 //         Assert.True(intersection1.Count == searchResults1.Count);
141 //         Assert.True(intersection1.Count == createResults.Length);
142 //
143 //         var intersection2 = createResults.Intersect(searchResults2).ToList();
144 //         Assert.True(intersection2.Count == searchResults2.Count);
145 //         Assert.True(intersection2.Count == createResults.Length);
146 //
147 //         var intersection3 = createResults.Intersect(searchResults3).ToList();
148 //         Assert.True(intersection3.Count == searchResults3.Count);
149 //         Assert.True(intersection3.Count == createResults.Length);
150 //
151 //         for (var i = 0; i < sequenceLength; i++)
152 //         {
153 //             links.Delete(sequence[i]);
154 //         }
155 //     }
156 // }
157 //
158 // [Fact]
159 // public static void BalancedVariantSearchTest()
160 // {
161 //     const long sequenceLength = 200;
162 //
163 //     using (var scope = new TempLinksTestScope(useSequences: true))
164 //     {
165 //         var links = scope.Links;
166 //         var sequences = scope.Sequences;
167 //
168 //         var sequence = new ulong[sequenceLength];
169 //         for (var i = 0; i < sequenceLength; i++)
170 //         {
171 //             sequence[i] = links.Create();
172 //         }
173 //
174 //         var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
175 //
176 //         var sw1 = Stopwatch.StartNew();
177 //         var balancedVariant = balancedVariantConverter.Convert(sequence); sw1.Stop();
178 //
179 //         var sw2 = Stopwatch.StartNew();
180 //         var searchResults2 = sequences.GetAllMatchingSequences0(sequence); sw2.Stop();
181 //
182 //         var sw3 = Stopwatch.StartNew();
183 //         var searchResults3 = sequences.GetAllMatchingSequences1(sequence); sw3.Stop();
184 //
185 //         // На количестве в 200 элементов это будет занимать вечность
186 //         //var sw4 = Stopwatch.StartNew();
187 //         //var searchResults4 = sequences.Each(sequence); sw4.Stop();
188 //
189 //         Assert.True(searchResults2.Count == 1 && balancedVariant ==
↵ searchResults2[0]);
190 //
191 //         Assert.True(searchResults3.Count == 1 && balancedVariant ==
↵ searchResults3.First());
192 //
193 //         //Assert.True(sw1.Elapsed < sw2.Elapsed);
194 //
195 //         for (var i = 0; i < sequenceLength; i++)
196 //         {
197 //             links.Delete(sequence[i]);
198 //         }
199 //     }
200 // }
201 //
202 // [Fact]
203 // public static void AllPartialVariantsSearchTest()
204 // {
205 //     const long sequenceLength = 8;
206 //

```

```

207 //         using (var scope = new TempLinksTestScope(useSequences: true))
208 //         {
209 //             var links = scope.Links;
210 //             var sequences = scope.Sequences;
211 //
212 //             var sequence = new ulong[sequenceLength];
213 //             for (var i = 0; i < sequenceLength; i++)
214 //             {
215 //                 sequence[i] = links.Create();
216 //             }
217 //
218 //             var createResults = sequences.CreateAllVariants2(sequence);
219 //
220 //             //var createResultsStrings = createResults.Select(x => x + ": " +
↵ sequences.FormatSequence(x)).ToList();
221 //             //Global.Trash = createResultsStrings;
222 //
223 //             var partialSequence = new ulong[sequenceLength - 2];
224 //
225 //             Array.Copy(sequence, 1, partialSequence, 0, (int)sequenceLength - 2);
226 //
227 //             var sw1 = Stopwatch.StartNew();
228 //             var searchResults1 =
↵ sequences.GetAllPartiallyMatchingSequences0(partialSequence); sw1.Stop();
229 //
230 //             var sw2 = Stopwatch.StartNew();
231 //             var searchResults2 =
↵ sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();
232 //
233 //             //var sw3 = Stopwatch.StartNew();
234 //             //var searchResults3 =
↵ sequences.GetAllPartiallyMatchingSequences2(partialSequence); sw3.Stop();
235 //
236 //             var sw4 = Stopwatch.StartNew();
237 //             var searchResults4 =
↵ sequences.GetAllPartiallyMatchingSequences3(partialSequence); sw4.Stop();
238 //
239 //             //Global.Trash = searchResults3;
240 //
241 //             //var searchResults1Strings = searchResults1.Select(x => x + ": " +
↵ sequences.FormatSequence(x)).ToList();
242 //             //Global.Trash = searchResults1Strings;
243 //
244 //             var intersection1 = createResults.Intersect(searchResults1).ToList();
245 //             Assert.True(intersection1.Count == createResults.Length);
246 //
247 //             var intersection2 = createResults.Intersect(searchResults2).ToList();
248 //             Assert.True(intersection2.Count == createResults.Length);
249 //
250 //             var intersection4 = createResults.Intersect(searchResults4).ToList();
251 //             Assert.True(intersection4.Count == createResults.Length);
252 //
253 //             for (var i = 0; i < sequenceLength; i++)
254 //             {
255 //                 links.Delete(sequence[i]);
256 //             }
257 //         }
258 //     }
259 //
260 //     [Fact]
261 //     public static void BalancedPartialVariantsSearchTest()
262 //     {
263 //         const long sequenceLength = 200;
264 //
265 //         using (var scope = new TempLinksTestScope(useSequences: true))
266 //         {
267 //             var links = scope.Links;
268 //             var sequences = scope.Sequences;
269 //
270 //             var sequence = new ulong[sequenceLength];
271 //             for (var i = 0; i < sequenceLength; i++)
272 //             {
273 //                 sequence[i] = links.Create();
274 //             }
275 //
276 //             var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
277 //

```

```

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 //         }

```



```

351 //     }
352 // }
353 //
354 // [Fact]
355 // public static void IndexTest()
356 // {
357 //     using (var scope = new TempLinksTestScope(new SequencesOptions<ulong> { UseIndex
    ↳ = true }, useSequences: true))
358 //     {
359 //         var links = scope.Links;
360 //         var sequences = scope.Sequences;
361 //         var index = sequences.Options.Index;
362 //
363 //         var e1 = links.Create();
364 //         var e2 = links.Create();
365 //
366 //         var sequence = new[]
367 //         {
368 //             e1, e2, e1, e2 // mama / papa
369 //         };
370 //
371 //         Assert.False(index.MightContain(sequence));
372 //
373 //         index.Add(sequence);
374 //
375 //         Assert.True(index.MightContain(sequence));
376 //     }
377 // }
378 // private static readonly string _exampleText =
379 //     @"([english
    ↳ version](https://github.com/Konard/LinksPlatform/wiki/About-the-beginning))
380 //
381 // 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?
382 //
383 // [![чёрное пространство, белое
    ↳ пространство](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/1.png
    ↳ "чёрное пространство, белое пространство")](https://raw.githubusercontent.com/Konard/Links
    ↳ Platform/master/doc/Intro/1.png)
384 //
385 // 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?
386 //
387 // [![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)
388 //
389 // А что если повторить? Сделать копию? Создать дубликат? Из одного сделать два? Может это быть
    ↳ так? Инверсия? Отражение? Сумма?
390 //
391 // [![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)
392 //
393 // 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?
394 //
395 // [![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)
396 //
397 // 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?
398 //
399 // [![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)

```

```

400 //
401 // 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?
402 //
403 // ![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)
404 //
405 // 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?
406 //
407 // ![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)
408 //
409 // 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?
410 //
411 // ![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)
412 //
413 // 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?
414 //
415 // ![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)
416 //
417 // 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?
418 //
419 // ![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)
420 //
421 // На один уровень внутрь (вниз)? Или на один уровень во вне (вверх)? Или это можно назвать
↳ шагом рекурсии или фрактала?
422 //
423 // ![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)
424 //
425 // Последовательность? Массив? Список? Множество? Объект? Таблица? Элементы? Цвета? Символы?
↳ Буквы? Слово? Цифры? Число? Алфавит? Дерево? Сеть? Граф? Гиперграф?
426 //
427 // ![белая обычная и направленная связи со структурой из 8 цветных элементов
↳ последовательности, чёрная типизированная связь со структурой из 8 цветных элементов последо
↳ вательности](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)
428 //
429 // ...
430 //

```

```

431 // [![анимация](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro-an
    ↳ imation-500.gif
    ↳ ""анимация"")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro
    ↳ -animation-500.gif)";
432 //         private static readonly string _exampleLoremIpsumText =
433 //             @"Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
    ↳ incididunt ut labore et dolore magna aliqua.
434 // Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea
    ↳ commodo consequat.";
435 //
436 //         [Fact]
437 //         public static void CompressionTest()
438 //         {
439 //             using (var scope = new TempLinksTestScope(useSequences: true))
440 //             {
441 //                 var links = scope.Links;
442 //                 var sequences = scope.Sequences;
443 //
444 //                 var e1 = links.Create();
445 //                 var e2 = links.Create();
446 //
447 //                 var sequence = new[]
448 //                 {
449 //                     e1, e2, e1, e2 // mama / papa / template [(m/p), a] { [1] [2] [1] [2] }
450 //                 };
451 //
452 //                 var balancedVariantConverter = new
    ↳ BalancedVariantConverter<ulong>(links.Unsync);
453 //                 var totalSequenceSymbolFrequencyCounter = new
    ↳ TotalSequenceSymbolFrequencyCounter<ulong>(links.Unsync);
454 //                 var doubletFrequenciesCache = new LinkFrequenciesCache<ulong>(links.Unsync,
    ↳ totalSequenceSymbolFrequencyCounter);
455 //                 var compressingConverter = new CompressingConverter<ulong>(links.Unsync,
    ↳ balancedVariantConverter, doubletFrequenciesCache);
456 //
457 //                 var compressedVariant = compressingConverter.Convert(sequence);
458 //
459 //                 // 1: [1]          (1->1) point
460 //                 // 2: [2]          (2->2) point
461 //                 // 3: [1,2]        (1->2) doublet
462 //                 // 4: [1,2,1,2]    (3->3) doublet
463 //
464 //                 Assert.True(links.GetSource(links.GetSource(compressedVariant)) ==
    ↳ sequence[0]);
465 //                 Assert.True(links.GetTarget(links.GetSource(compressedVariant)) ==
    ↳ sequence[1]);
466 //                 Assert.True(links.GetSource(links.GetTarget(compressedVariant)) ==
    ↳ sequence[2]);
467 //                 Assert.True(links.GetTarget(links.GetTarget(compressedVariant)) ==
    ↳ sequence[3]);
468 //
469 //                 var source = _constants.SourcePart;
470 //                 var target = _constants.TargetPart;
471 //
472 //                 Assert.True(links.GetByKeys(compressedVariant, source, source) ==
    ↳ sequence[0]);
473 //                 Assert.True(links.GetByKeys(compressedVariant, source, target) ==
    ↳ sequence[1]);
474 //                 Assert.True(links.GetByKeys(compressedVariant, target, source) ==
    ↳ sequence[2]);
475 //                 Assert.True(links.GetByKeys(compressedVariant, target, target) ==
    ↳ sequence[3]);
476 //
477 //                 // 4 - length of sequence
478 //                 Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4,
    ↳ 0) == sequence[0]);
479 //                 Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4,
    ↳ 1) == sequence[1]);
480 //                 Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4,
    ↳ 2) == sequence[2]);
481 //                 Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4,
    ↳ 3) == sequence[3]);
482 //             }
483 //         }
484 //
485 //         [Fact]

```

```

486 //         public static void CompressionEfficiencyTest()
487 //         {
488 //             var strings = _exampleLoremIpsumText.Split(new[] { '\n', '\r' },
↳ StringSplitOptions.RemoveEmptyEntries);
489 //             var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
490 //             var totalCharacters = arrays.Select(x => x.Length).Sum();
491 //
492 //             using (var scope1 = new TempLinksTestScope(useSequences: true))
493 //             using (var scope2 = new TempLinksTestScope(useSequences: true))
494 //             using (var scope3 = new TempLinksTestScope(useSequences: true))
495 //             {
496 //                 scope1.Links.Unsync.UseUnicode();
497 //                 scope2.Links.Unsync.UseUnicode();
498 //                 scope3.Links.Unsync.UseUnicode();
499 //
500 //                 var balancedVariantConverter1 = new
↳ BalancedVariantConverter<ulong>(scope1.Links.Unsync);
501 //                 var totalSequenceSymbolFrequencyCounter = new
↳ TotalSequenceSymbolFrequencyCounter<ulong>(scope1.Links.Unsync);
502 //                 var linkFrequenciesCache1 = new
↳ LinkFrequenciesCache<ulong>(scope1.Links.Unsync, totalSequenceSymbolFrequencyCounter);
503 //                 var compressor1 = new CompressingConverter<ulong>(scope1.Links.Unsync,
↳ balancedVariantConverter1, linkFrequenciesCache1, doInitialFrequenciesIncrement: false);
504 //
505 //                 //var compressor2 = scope2.Sequences;
506 //                 var compressor3 = scope3.Sequences;
507 //
508 //                 var constants = Default<LinksConstants<ulong>>.Instance;
509 //
510 //                 var sequences = compressor3;
511 //                 //var meaningRoot = links.CreatePoint();
512 //                 //var unaryOne = links.CreateAndUpdate(meaningRoot, constants.Itself);
513 //                 //var frequencyMarker = links.CreateAndUpdate(meaningRoot, constants.Itself);
514 //                 //var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot,
↳ constants.Itself);
515 //
516 //                 //var unaryNumberToAddressConverter = new
↳ UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);
517 //                 //var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links,
↳ unaryOne);
518 //                 //var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
↳ frequencyMarker, unaryOne, unaryNumberIncrementer);
519 //                 //var frequencyPropertyOperator = new FrequencyPropertyOperator<ulong>(links,
↳ frequencyPropertyMarker, frequencyMarker);
520 //                 //var linkFrequencyIncrementer = new LinkFrequencyIncrementer<ulong>(links,
↳ frequencyPropertyOperator, frequencyIncrementer);
521 //                 //var linkToItsFrequencyNumberConverter = new
↳ LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
↳ unaryNumberToAddressConverter);
522 //
523 //                 var linkFrequenciesCache3 = new
↳ LinkFrequenciesCache<ulong>(scope3.Links.Unsync, totalSequenceSymbolFrequencyCounter);
524 //
525 //                 var linkToItsFrequencyNumberConverter = new
↳ FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<ulong>(linkFrequenciesCache3);
526 //
527 //                 var sequenceToItsLocalElementLevelsConverter = new
↳ SequenceToItsLocalElementLevelsConverter<ulong>(scope3.Links.Unsync,
↳ linkToItsFrequencyNumberConverter);
528 //                 var optimalVariantConverter = new
↳ OptimalVariantConverter<ulong>(scope3.Links.Unsync,
↳ sequenceToItsLocalElementLevelsConverter);
529 //
530 //                 var compressed1 = new ulong[arrays.Length];
531 //                 var compressed2 = new ulong[arrays.Length];
532 //                 var compressed3 = new ulong[arrays.Length];
533 //
534 //                 var START = 0;
535 //                 var END = arrays.Length;
536 //
537 //                 //for (int i = START; i < END; i++)
538 //                 //     linkFrequenciesCache1.IncrementFrequencies(arrays[i]);
539 //
540 //                 var initialCount1 = scope2.Links.Unsync.Count();
541 //
542 //                 var sw1 = Stopwatch.StartNew();

```

```

543 //
544 //         for (int i = START; i < END; i++)
545 //         {
546 //             linkFrequenciesCache1.IncrementFrequencies(arrays[i]);
547 //             compressed1[i] = compressor1.Convert(arrays[i]);
548 //         }
549 //
550 //         var elapsed1 = sw1.Elapsed;
551 //
552 //         var balancedVariantConverter2 = new
↵ BalancedVariantConverter<ulong>(scope2.Links.Unsync);
553 //
554 //         var initialCount2 = scope2.Links.Unsync.Count();
555 //
556 //         var sw2 = Stopwatch.StartNew();
557 //
558 //         for (int i = START; i < END; i++)
559 //         {
560 //             compressed2[i] = balancedVariantConverter2.Convert(arrays[i]);
561 //         }
562 //
563 //         var elapsed2 = sw2.Elapsed;
564 //
565 //         for (int i = START; i < END; i++)
566 //         {
567 //             linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
568 //         }
569 //
570 //         var initialCount3 = scope3.Links.Unsync.Count();
571 //
572 //         var sw3 = Stopwatch.StartNew();
573 //
574 //         for (int i = START; i < END; i++)
575 //         {
576 //             //linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
577 //             compressed3[i] = optimalVariantConverter.Convert(arrays[i]);
578 //         }
579 //
580 //         var elapsed3 = sw3.Elapsed;
581 //
582 //         Console.WriteLine($"Compressor: {elapsed1}, Balanced variant: {elapsed2},
↵ Optimal variant: {elapsed3}");
583 //
584 //         // Assert.True(elapsed1 > elapsed2);
585 //
586 //         // Checks
587 //         for (int i = START; i < END; i++)
588 //         {
589 //             var sequence1 = compressed1[i];
590 //             var sequence2 = compressed2[i];
591 //             var sequence3 = compressed3[i];
592 //
593 //             var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
↵ scope1.Links.Unsync);
594 //
595 //             var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
↵ scope2.Links.Unsync);
596 //
597 //             var decompress3 = UnicodeMap.FromSequenceLinkToString(sequence3,
↵ scope3.Links.Unsync);
598 //
599 //             var structure1 = scope1.Links.Unsync.FormatStructure(sequence1, link =>
↵ link.IsPartialPoint());
600 //             var structure2 = scope2.Links.Unsync.FormatStructure(sequence2, link =>
↵ link.IsPartialPoint());
601 //             var structure3 = scope3.Links.Unsync.FormatStructure(sequence3, link =>
↵ link.IsPartialPoint());
602 //
603 //             //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
↵ arrays[i].Length > 3)
604 //                 // Assert.False(structure1 == structure2);
605 //             //if (sequence3 != Constants.Null && sequence2 != Constants.Null &&
↵ arrays[i].Length > 3)
606 //                 // Assert.False(structure3 == structure2);
607 //
608 //             Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
609 //             Assert.True(strings[i] == decompress3 && decompress3 == decompress2);

```

```

610 //         }
611 //
612 //         Assert.True((int)(scope1.Links.Unsync.Count() - initialCount1) <
↳ totalCharacters);
613 //         Assert.True((int)(scope2.Links.Unsync.Count() - initialCount2) <
↳ totalCharacters);
614 //         Assert.True((int)(scope3.Links.Unsync.Count() - initialCount3) <
↳ totalCharacters);
615 //
616 //         Console.WriteLine($"{(double)(scope1.Links.Unsync.Count() - initialCount1) /
↳ totalCharacters} | {(double)(scope2.Links.Unsync.Count() - initialCount2) / totalCharacters}
↳ | {(double)(scope3.Links.Unsync.Count() - initialCount3) / totalCharacters}");
617 //
618 //         Assert.True(scope1.Links.Unsync.Count() - initialCount1 <
↳ scope2.Links.Unsync.Count() - initialCount2);
619 //         Assert.True(scope3.Links.Unsync.Count() - initialCount3 <
↳ scope2.Links.Unsync.Count() - initialCount2);
620 //
621 //         var duplicateProvider1 = new
↳ DuplicateSegmentsProvider<ulong>(scope1.Links.Unsync, scope1.Sequences);
622 //         var duplicateProvider2 = new
↳ DuplicateSegmentsProvider<ulong>(scope2.Links.Unsync, scope2.Sequences);
623 //         var duplicateProvider3 = new
↳ DuplicateSegmentsProvider<ulong>(scope3.Links.Unsync, scope3.Sequences);
624 //
625 //         var duplicateCounter1 = new
↳ DuplicateSegmentsCounter<ulong>(duplicateProvider1);
626 //         var duplicateCounter2 = new
↳ DuplicateSegmentsCounter<ulong>(duplicateProvider2);
627 //         var duplicateCounter3 = new
↳ DuplicateSegmentsCounter<ulong>(duplicateProvider3);
628 //
629 //         var duplicates1 = duplicateCounter1.Count();
630 //
631 //         ConsoleHelpers.Debug("-----");
632 //
633 //         var duplicates2 = duplicateCounter2.Count();
634 //
635 //         ConsoleHelpers.Debug("-----");
636 //
637 //         var duplicates3 = duplicateCounter3.Count();
638 //
639 //         Console.WriteLine($"{duplicates1} | {duplicates2} | {duplicates3}");
640 //
641 //         linkFrequenciesCache1.ValidateFrequencies();
642 //         linkFrequenciesCache3.ValidateFrequencies();
643 //     }
644 // }
645 //
646 // [Fact]
647 // public static void CompressionStabilityTest()
648 // {
649 //     // TODO: Fix bug (do a separate test)
650 //     //const ulong minNumbers = 0;
651 //     //const ulong maxNumbers = 1000;
652 //
653 //     const ulong minNumbers = 10000;
654 //     const ulong maxNumbers = 12500;
655 //
656 //     var strings = new List<string>();
657 //
658 //     for (ulong i = minNumbers; i < maxNumbers; i++)
659 //     {
660 //         strings.Add(i.ToString());
661 //     }
662 //
663 //     var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
664 //     var totalCharacters = arrays.Select(x => x.Length).Sum();
665 //
666 //     using (var scope1 = new TempLinksTestScope(useSequences: true, sequencesOptions:
↳ new SequencesOptions<ulong> { UseCompression = true,
↳ EnforceSingleSequenceVersionOnWriteBasedOnExisting = true }))
667 //     using (var scope2 = new TempLinksTestScope(useSequences: true))
668 //     {
669 //         scope1.Links.UseUnicode();
670 //         scope2.Links.UseUnicode();

```

```

671 //
672 //
673 // //var compressor1 = new Compressor(scope1.Links.Unsync, scope1.Sequences);
674 // var compressor1 = scope1.Sequences;
675 // var compressor2 = scope2.Sequences;
676 //
677 // var compressed1 = new ulong[arrays.Length];
678 // var compressed2 = new ulong[arrays.Length];
679 //
680 //
681 // var sw1 = Stopwatch.StartNew();
682 //
683 // var START = 0;
684 // var END = arrays.Length;
685 //
686 // // Collisions proved (cannot be solved by max doublet comparison, no stable
687 // ↪ rule)
688 // // Stability issue starts at 10001 or 11000
689 // //for (int i = START; i < END; i++)
690 // //{
691 // //    var first = compressor1.Compress(arrays[i]);
692 // //    var second = compressor1.Compress(arrays[i]);
693 // //
694 // //    if (first == second)
695 // //        compressed1[i] = first;
696 // //    else
697 // //    {
698 // //        // TODO: Find a solution for this case
699 // //    }
700 // //}
701 //
702 // for (int i = START; i < END; i++)
703 // {
704 //     var first = compressor1.Create(arrays[i].ShiftRight());
705 //     var second = compressor1.Create(arrays[i].ShiftRight());
706 //
707 //     if (first == second)
708 //     {
709 //         compressed1[i] = first;
710 //     }
711 //     else
712 //     {
713 //         // TODO: Find a solution for this case
714 //     }
715 // }
716 //
717 // var elapsed1 = sw1.Elapsed;
718 //
719 // var balancedVariantConverter = new
720 // ↪ BalancedVariantConverter<ulong>(scope2.Links);
721 //
722 // var sw2 = Stopwatch.StartNew();
723 //
724 // for (int i = START; i < END; i++)
725 // {
726 //     var first = balancedVariantConverter.Convert(arrays[i]);
727 //     var second = balancedVariantConverter.Convert(arrays[i]);
728 //
729 //     if (first == second)
730 //     {
731 //         compressed2[i] = first;
732 //     }
733 // }
734 //
735 // var elapsed2 = sw2.Elapsed;
736 //
737 // Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator:
738 // ↪ {elapsed2}");
739 //
740 // Assert.True(elapsed1 > elapsed2);
741 //
742 // // Checks
743 // for (int i = START; i < END; i++)
744 // {
745 //     var sequence1 = compressed1[i];
746 //     var sequence2 = compressed2[i];
747 //
748 //     if (sequence1 != _constants.Null && sequence2 != _constants.Null)
749 //     {

```

```
745 // var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,  
→ scope1.Links);  
746 //  
747 // var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,  
→ scope2.Links);  
748 //  
749 // //var structure1 = scope1.Links.FormatStructure(sequence1, link =>  
→ link.IsPartialPoint());  
750 // //var structure2 = scope2.Links.FormatStructure(sequence2, link =>  
→ link.IsPartialPoint());  
751 //  
752 // //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&  
→ arrays[i].Length > 3)  
753 // // Assert.False(structure1 == structure2);  
754 //  
755 // Assert.True(strings[i] == decompress1 && decompress1 == decompress2);  
756 // }  
757 // }  
758 //  
759 // Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) <  
→ totalCharacters);  
760 // Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) <  
→ totalCharacters);  
761 //  
762 // Debug.WriteLine($"{(double)(scope1.Links.Count() - UnicodeMap.MapSize) /  
→ totalCharacters} | {(double)(scope2.Links.Count() - UnicodeMap.MapSize) / totalCharacters}");  
763 //  
764 // Assert.True(scope1.Links.Count() <= scope2.Links.Count());  
765 //  
766 // //compressor1.ValidateFrequencies();  
767 // }  
768 // }  
769 //  
770 [Fact]  
771 public static void RandomNumbersCompressionQualityTest()  
772 {  
773     const ulong N = 500;  
774 //  
775 //const ulong minNumbers = 10000;  
776 //const ulong maxNumbers = 20000;  
777 //  
778 //var strings = new List<string>();  
779 //  
780 //for (ulong i = 0; i < N; i++)  
781 //    strings.Add(RandomHelpers.DefaultFactory.NextUInt64(minNumbers,  
→ maxNumbers).ToString());  
782 //  
783 //var strings = new List<string>();  
784 //  
785 //for (ulong i = 0; i < N; i++)  
786 //{  
787 //    strings.Add(RandomHelpers.Default.NextUInt64().ToString());  
788 //}  
789 //  
790 //strings = strings.Distinct().ToList();  
791 //  
792 //var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();  
793 //var totalCharacters = arrays.Select(x => x.Length).Sum();  
794 //  
795 //using (var scope1 = new TempLinksTestScope(useSequences: true, sequencesOptions:  
→ new SequencesOptions<ulong> { UseCompression = true,  
→ EnforceSingleSequenceVersionOnWriteBasedOnExisting = true }))  
796 //    using (var scope2 = new TempLinksTestScope(useSequences: true))  
797 //    {  
798 //        scope1.Links.UseUnicode();  
799 //        scope2.Links.UseUnicode();  
800 //  
801 //var compressor1 = scope1.Sequences;  
802 //var compressor2 = scope2.Sequences;  
803 //  
804 //var compressed1 = new ulong[arrays.Length];  
805 //var compressed2 = new ulong[arrays.Length];  
806 //  
807 //var sw1 = Stopwatch.StartNew();  
808 //  
809 //var START = 0;  
810 //var END = arrays.Length;
```



```

811 //
812 //         for (int i = START; i < END; i++)
813 //         {
814 //             compressed1[i] = compressor1.Create(arrays[i].ShiftRight());
815 //         }
816 //
817 //         var elapsed1 = sw1.Elapsed;
818 //
819 //         var balancedVariantConverter = new
↵ BalancedVariantConverter<ulong>(scope2.Links);
820 //
821 //         var sw2 = Stopwatch.StartNew();
822 //
823 //         for (int i = START; i < END; i++)
824 //         {
825 //             compressed2[i] = balancedVariantConverter.Convert(arrays[i]);
826 //         }
827 //
828 //         var elapsed2 = sw2.Elapsed;
829 //
830 //         Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator:
↵ {elapsed2}");
831 //
832 //         Assert.True(elapsed1 > elapsed2);
833 //
834 //         // Checks
835 //         for (int i = START; i < END; i++)
836 //         {
837 //             var sequence1 = compressed1[i];
838 //             var sequence2 = compressed2[i];
839 //
840 //             if (sequence1 != _constants.Null && sequence2 != _constants.Null)
841 //             {
842 //                 var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
↵ scope1.Links);
843 //
844 //                 var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
↵ scope2.Links);
845 //
846 //                 Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
847 //             }
848 //         }
849 //
850 //         Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) <
↵ totalCharacters);
851 //         Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) <
↵ totalCharacters);
852 //
853 //         Debug.WriteLine($"{{(double)(scope1.Links.Count() - UnicodeMap.MapSize) /
↵ totalCharacters}} | {{(double)(scope2.Links.Count() - UnicodeMap.MapSize) / totalCharacters}}");
854 //
855 //         // Can be worse than balanced variant
856 //         //Assert.True(scope1.Links.Count() <= scope2.Links.Count());
857 //
858 //         //compressor1.ValidateFrequencies();
859 //     }
860 // }
861 //
862 // [Fact]
863 // public static void AllTreeBreakDownAtSequencesCreationBugTest()
864 // {
865 //     // Made out of AllPossibleConnectionsTest test.
866 //
867 //     //const long sequenceLength = 5; //100% bug
868 //     const long sequenceLength = 4; //100% bug
869 //     //const long sequenceLength = 3; //100% _no_bug (ok)
870 //
871 //     using (var scope = new TempLinksTestScope(useSequences: true))
872 //     {
873 //         var links = scope.Links;
874 //         var sequences = scope.Sequences;
875 //
876 //         var sequence = new ulong[sequenceLength];
877 //         for (var i = 0; i < sequenceLength; i++)
878 //         {
879 //             sequence[i] = links.Create();
880 //         }

```

```

881 //
882 //         var createResults = sequences.CreateAllVariants2(sequence);
883 //
884 //         Global.Trash = createResults;
885 //
886 //         for (var i = 0; i < sequenceLength; i++)
887 //         {
888 //             links.Delete(sequence[i]);
889 //         }
890 //     }
891 // }
892 //
893 // [Fact]
894 // public static void AllPossibleConnectionsTest()
895 // {
896 //     const long sequenceLength = 5;
897 //
898 //     using (var scope = new TempLinksTestScope(useSequences: true))
899 //     {
900 //         var links = scope.Links;
901 //         var sequences = scope.Sequences;
902 //
903 //         var sequence = new ulong[sequenceLength];
904 //         for (var i = 0; i < sequenceLength; i++)
905 //         {
906 //             sequence[i] = links.Create();
907 //         }
908 //
909 //         var createResults = sequences.CreateAllVariants2(sequence);
910 //         var reverseResults =
↵ sequences.CreateAllVariants2(sequence.Reverse().ToArray());
911 //
912 //         for (var i = 0; i < 1; i++)
913 //         {
914 //             var sw1 = Stopwatch.StartNew();
915 //             var searchResults1 = sequences.GetAllConnections(sequence); sw1.Stop();
916 //
917 //             var sw2 = Stopwatch.StartNew();
918 //             var searchResults2 = sequences.GetAllConnections1(sequence); sw2.Stop();
919 //
920 //             var sw3 = Stopwatch.StartNew();
921 //             var searchResults3 = sequences.GetAllConnections2(sequence); sw3.Stop();
922 //
923 //             var sw4 = Stopwatch.StartNew();
924 //             var searchResults4 = sequences.GetAllConnections3(sequence); sw4.Stop();
925 //
926 //             Global.Trash = searchResults3;
927 //             Global.Trash = searchResults4; //-V3008
928 //
929 //             var intersection1 = createResults.Intersect(searchResults1).ToList();
930 //             Assert.True(intersection1.Count == createResults.Length);
931 //
932 //             var intersection2 = reverseResults.Intersect(searchResults1).ToList();
933 //             Assert.True(intersection2.Count == reverseResults.Length);
934 //
935 //             var intersection0 = searchResults1.Intersect(searchResults2).ToList();
936 //             Assert.True(intersection0.Count == searchResults2.Count);
937 //
938 //             var intersection3 = searchResults2.Intersect(searchResults3).ToList();
939 //             Assert.True(intersection3.Count == searchResults3.Count);
940 //
941 //             var intersection4 = searchResults3.Intersect(searchResults4).ToList();
942 //             Assert.True(intersection4.Count == searchResults4.Count);
943 //         }
944 //
945 //         for (var i = 0; i < sequenceLength; i++)
946 //         {
947 //             links.Delete(sequence[i]);
948 //         }
949 //     }
950 // }
951 //
952 // [Fact(Skip = "Correct implementation is pending")]
953 // public static void CalculateAllUsagesTest()
954 // {
955 //     const long sequenceLength = 3;
956 //

```

```

957 //         using (var scope = new TempLinksTestScope(useSequences: true))
958 //         {
959 //             var links = scope.Links;
960 //             var sequences = scope.Sequences;
961 //
962 //             var sequence = new ulong[sequenceLength];
963 //             for (var i = 0; i < sequenceLength; i++)
964 //             {
965 //                 sequence[i] = links.Create();
966 //             }
967 //
968 //             var createResults = sequences.CreateAllVariants2(sequence);
969 //
970 //             //var reverseResults =
↵ sequences.CreateAllVariants2(sequence.Reverse().ToArray());
971 //
972 //             for (var i = 0; i < 1; i++)
973 //             {
974 //                 var linksTotalUsages1 = new ulong[links.Count() + 1];
975 //
976 //                 sequences.CalculateAllUsages(linksTotalUsages1);
977 //
978 //                 var linksTotalUsages2 = new ulong[links.Count() + 1];
979 //
980 //                 sequences.CalculateAllUsages2(linksTotalUsages2);
981 //
982 //                 var intersection1 =
↵ linksTotalUsages1.Intersect(linksTotalUsages2).ToList();
983 //                 Assert.True(intersection1.Count == linksTotalUsages2.Length);
984 //             }
985 //
986 //             for (var i = 0; i < sequenceLength; i++)
987 //             {
988 //                 links.Delete(sequence[i]);
989 //             }
990 //         }
991 //     }
992 // }
993 // }

```

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

```

1 // using System.IO;
2 // using Platform.Disposables;
3 // using Platform.Data.Doublets.Sequences;
4 // using Platform.Data.Doublets.Decorators;
5 // using Platform.Data.Doublets.Memory.United.Specific;
6 // using Platform.Data.Doublets.Memory.Split.Specific;
7 // using Platform.Memory;
8 //
9 // namespace Platform.Data.Doublets.Sequences.Tests
10 // {
11 //     public class TempLinksTestScope : DisposableBase
12 //     {
13 //         public ILinks<ulong> MemoryAdapter { get; }
14 //         public SynchronizedLinks<ulong> Links { get; }
15 //         public Sequences Sequences { get; }
16 //         public string TempFilename { get; }
17 //         public string TempTransactionLogFilename { get; }
18 //         private readonly bool _deleteFiles;
19 //
20 //         public TempLinksTestScope(bool deleteFiles = true, bool useSequences = false, bool
↵ useLog = false) : this(new SequencesOptions<ulong>(), deleteFiles, useSequences, useLog) { }
21 //
22 //         public TempLinksTestScope(SequencesOptions<ulong> sequencesOptions, bool deleteFiles
↵ = true, bool useSequences = false, bool useLog = false)
23 //         {
24 //             _deleteFiles = deleteFiles;
25 //             TempFilename = Path.GetTempFileName();
26 //             TempTransactionLogFilename = Path.GetTempFileName();
27 //             //var coreMemoryAdapter = new UInt64UnitedMemoryLinks(TempFilename);
28 //             var coreMemoryAdapter = new UInt64SplitMemoryLinks(new
↵ FileMappedResizableDirectMemory(TempFilename), new
↵ FileMappedResizableDirectMemory(Path.ChangeExtension(TempFilename, "indexes")),
↵ UInt64SplitMemoryLinks.DefaultLinksSizeStep, new LinksConstants<ulong>(),
↵ Memory.IndexTreeType.Default, useLinkedList: true);

```

```

29 //          MemoryAdapter = useLog ? (ILinks<ulong>)new
↳ UInt64LinksTransactionsLayer(coreMemoryAdapter, TempTransactionLogFilename) :
↳ coreMemoryAdapter;
30 //          Links = new SynchronizedLinks<ulong>(new UInt64Links(MemoryAdapter));
31 //          if (useSequences)
32 //          {
33 //              Sequences = new Sequences(Links, sequencesOptions);
34 //          }
35 //      }
36 //
37 //      protected override void Dispose(bool manual, bool wasDisposed)
38 //      {
39 //          if (!wasDisposed)
40 //          {
41 //              Links.Unsync.DisposeIfPossible();
42 //              if (_deleteFiles)
43 //              {
44 //                  DeleteFiles();
45 //              }
46 //          }
47 //      }
48 //
49 //      public void DeleteFiles()
50 //      {
51 //          File.Delete(TempFilename);
52 //          File.Delete(TempTransactionLogFilename);
53 //      }
54 //  }
55 // }

```

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

```

1 // using System.Collections.Generic;
2 // using Xunit;
3 // using Platform.Ranges;
4 // using Platform.Numbers;
5 // using Platform.Random;
6 // using Platform.Setters;
7 // using Platform.Converters;
8 //
9 // namespace Platform.Data.Doublets.Sequences.Tests
10 // {
11 //     public static class TestExtensions
12 //     {
13 //         public static void TestCRUDOperations<T>(this ILinks<T> links)
14 //         {
15 //             var constants = links.Constants;
16 //
17 //             var equalityComparer = EqualityComparer<T>.Default;
18 //
19 //             var zero = default(T);
20 //             var one = Arithmetic.Increment(zero);
21 //
22 //             // Create Link
23 //             Assert.True(equalityComparer.Equals(links.Count(), zero));
24 //
25 //             var setter = new Setter<T>(constants.Null);
26 //             links.Each(constants.Any, constants.Any, setter.SetAndReturnTrue);
27 //
28 //             Assert.True(equalityComparer.Equals(setter.Result, constants.Null));
29 //
30 //             var linkAddress = links.Create();
31 //
32 //             var link = new Link<T>(links.GetLink(linkAddress));
33 //
34 //             Assert.True(link.Count == 3);
35 //             Assert.True(equalityComparer.Equals(link.Index, linkAddress));
36 //             Assert.True(equalityComparer.Equals(link.Source, constants.Null));
37 //             Assert.True(equalityComparer.Equals(link.Target, constants.Null));
38 //
39 //             Assert.True(equalityComparer.Equals(links.Count(), one));
40 //
41 //             // Get first link
42 //             setter = new Setter<T>(constants.Null);
43 //             links.Each(constants.Any, constants.Any, setter.SetAndReturnFalse);
44 //
45 //             Assert.True(equalityComparer.Equals(setter.Result, linkAddress));
46 //
47 //             // Update link to reference itself

```

```

48 //         links.Update(linkAddress, linkAddress, linkAddress);
49 //
50 //         link = new Link<T>(links.GetLink(linkAddress));
51 //
52 //         Assert.True(equalityComparer.Equals(link.Source, linkAddress));
53 //         Assert.True(equalityComparer.Equals(link.Target, linkAddress));
54 //
55 //         // Update link to reference null (prepare for delete)
56 //         var updated = links.Update(linkAddress, constants.Null, constants.Null);
57 //
58 //         Assert.True(equalityComparer.Equals(updated, linkAddress));
59 //
60 //         link = new Link<T>(links.GetLink(linkAddress));
61 //
62 //         Assert.True(equalityComparer.Equals(link.Source, constants.Null));
63 //         Assert.True(equalityComparer.Equals(link.Target, constants.Null));
64 //
65 //         // Delete link
66 //         links.Delete(linkAddress);
67 //
68 //         Assert.True(equalityComparer.Equals(links.Count(), zero));
69 //
70 //         setter = new Setter<T>(constants.Null);
71 //         links.Each(constants.Any, constants.Any, setter.SetAndReturnTrue);
72 //
73 //         Assert.True(equalityComparer.Equals(setter.Result, constants.Null));
74 //     }
75 //
76 // public static void TestRawNumbersCRUDOperations<T>(this ILinks<T> links)
77 // {
78 //     // Constants
79 //     var constants = links.Constants;
80 //     var equalityComparer = EqualityComparer<T>.Default;
81 //
82 //     var zero = default(T);
83 //     var one = Arithmetic.Increment(zero);
84 //     var two = Arithmetic.Increment(one);
85 //
86 //     var h106E = new Hybrid<T>(106L, isExternal: true);
87 //     var h107E = new Hybrid<T>(-char.ConvertFromUtf32(107)[0]);
88 //     var h108E = new Hybrid<T>(-108L);
89 //
90 //     Assert.Equal(106L, h106E.AbsoluteValue);
91 //     Assert.Equal(107L, h107E.AbsoluteValue);
92 //     Assert.Equal(108L, h108E.AbsoluteValue);
93 //
94 //     // Create Link (External -> External)
95 //     var linkAddress1 = links.Create();
96 //
97 //     links.Update(linkAddress1, h106E, h108E);
98 //
99 //     var link1 = new Link<T>(links.GetLink(linkAddress1));
100 //
101 //     Assert.True(equalityComparer.Equals(link1.Source, h106E));
102 //     Assert.True(equalityComparer.Equals(link1.Target, h108E));
103 //
104 //     // Create Link (Internal -> External)
105 //     var linkAddress2 = links.Create();
106 //
107 //     links.Update(linkAddress2, linkAddress1, h108E);
108 //
109 //     var link2 = new Link<T>(links.GetLink(linkAddress2));
110 //
111 //     Assert.True(equalityComparer.Equals(link2.Source, linkAddress1));
112 //     Assert.True(equalityComparer.Equals(link2.Target, h108E));
113 //
114 //     // Create Link (Internal -> Internal)
115 //     var linkAddress3 = links.Create();
116 //
117 //     links.Update(linkAddress3, linkAddress1, linkAddress2);
118 //
119 //     var link3 = new Link<T>(links.GetLink(linkAddress3));
120 //
121 //     Assert.True(equalityComparer.Equals(link3.Source, linkAddress1));
122 //     Assert.True(equalityComparer.Equals(link3.Target, linkAddress2));
123 //
124 //     // Search for created link
125 //     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);

```

```

199 //      }
200 //    }
201 //  }
202 // }

```

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

```

1  // using System;
2  // using System.Collections.Generic;
3  // using System.Diagnostics;
4  // using System.IO;
5  // using System.Text;
6  // using System.Threading;
7  // using System.Threading.Tasks;
8  // using Xunit;
9  // using Platform.Disposables;
10 // using Platform.Ranges;
11 // using Platform.Random;
12 // using Platform.Timestamps;
13 // using Platform.Reflection;
14 // using Platform.Singletons;
15 // using Platform.Scopes;
16 // using Platform.Counters;
17 // using Platform.Diagnostics;
18 // using Platform.IO;
19 // using Platform.Memory;
20 // using Platform.Data.Doublets.Decorators;
21 // using Platform.Data.Doublets.Memory.United.Specific;
22 //
23 // namespace Platform.Data.Doublets.Sequences.Tests
24 // {
25 //     public static class UInt64LinksTests
26 //     {
27 //         private static readonly LinksConstants<ulong> _constants =
28 //         ↪ Default<LinksConstants<ulong>>.Instance;
29 //         private const long Iterations = 10 * 1024;
30 //
31 //         #region Concept
32 //
33 //         [Fact]
34 //         public static void MultipleCreateAndDeleteTest()
35 //         {
36 //             using (var scope = new Scope<Types<HeapResizableDirectMemory,
37 //             ↪ UInt64UnitedMemoryLinks>>())
38 //             {
39 //                 new
40 //             ↪ UInt64Links(scope.Use<ILinks<ulong>>()).TestMultipleRandomCreationsAndDeletions(100);
41 //             }
42 //         }
43 //
44 //         [Fact]
45 //         public static void CascadeUpdateTest()
46 //         {
47 //             var itself = _constants.Itself;
48 //             using (var scope = new TempLinksTestScope(useLog: true))
49 //             {
50 //                 var links = scope.Links;
51 //
52 //                 var l1 = links.Create();
53 //                 var l2 = links.Create();
54 //
55 //                 l2 = links.Update(l2, l2, l1, l2);
56 //
57 //                 links.CreateAndUpdate(l2, itself);
58 //                 links.CreateAndUpdate(l2, itself);
59 //
60 //                 l2 = links.Update(l2, l1);
61 //
62 //                 links.Delete(l2);
63 //
64 //                 Global.Trash = links.Count();
65 //
66 //                 links.Unsync.DisposeIfPossible(); // Close links to access log
67 //
68 //                 Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scope.TempTransactionLogFilename);
69 //             }
70 //         }
71 //     }
72 // }

```

```

68 //
69 // [Fact]
70 // public static void BasicTransactionLogTest()
71 // {
72 //     using (var scope = new TempLinksTestScope(useLog: true))
73 //     {
74 //         var links = scope.Links;
75 //         var l1 = links.Create();
76 //         var l2 = links.Create();
77 //
78 //         Global.Trash = links.Update(l2, l2, l1, l2);
79 //
80 //         links.Delete(l1);
81 //
82 //         links.Unsync.DisposeIfPossible(); // Close links to access log
83 //
84 //         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scope.TempTransactionLogFilename);
85 //     }
86 // }
87 //
88 // [Fact]
89 // public static void TransactionAutoRevertedTest()
90 // {
91 //     // Auto Reverted (Because no commit at transaction)
92 //     using (var scope = new TempLinksTestScope(useLog: true))
93 //     {
94 //         var links = scope.Links;
95 //         var transactionsLayer = (UInt64LinksTransactionsLayer)scope.MemoryAdapter;
96 //         using (var transaction = transactionsLayer.BeginTransaction())
97 //         {
98 //             var l1 = links.Create();
99 //             var l2 = links.Create();
100 //
101 //             links.Update(l2, l2, l1, l2);
102 //         }
103 //
104 //         Assert.Equal(0UL, links.Count());
105 //
106 //         links.Unsync.DisposeIfPossible();
107 //
108 //         var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scope.TempTransactionLogFilename);
109 //         Assert.Single(transitions);
110 //     }
111 // }
112 //
113 // [Fact]
114 // public static void TransactionUserCodeErrorNoDataSavedTest()
115 // {
116 //     // User Code Error (Autoreverted), no data saved
117 //     var itself = _constants.Itself;
118 //
119 //     TempLinksTestScope lastScope = null;
120 //     try
121 //     {
122 //         using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false, useLog: true))
123 //         {
124 //             var links = scope.Links;
125 //             var transactionsLayer = (UInt64LinksTransactionsLayer)((LinksDisposableDecoratorBase<ulong>)links.Unsync).Links;
126 //             using (var transaction = transactionsLayer.BeginTransaction())
127 //             {
128 //                 var l1 = links.CreateAndUpdate(itself, itself);
129 //                 var l2 = links.CreateAndUpdate(itself, itself);
130 //
131 //                 l2 = links.Update(l2, l2, l1, l2);
132 //
133 //                 links.CreateAndUpdate(l2, itself);
134 //                 links.CreateAndUpdate(l2, itself);
135 //
136 //                 //Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scope.TempTransactionLogFilename);
137 //
138 //                 l2 = links.Update(l2, l1);
139 //

```



```

140 //                links.Delete(l2);
141 //
142 //                ExceptionThrower();
143 //
144 //                transaction.Commit();
145 //            }
146 //
147 //                Global.Trash = links.Count();
148 //            }
149 //        }
150 //    catch
151 //    {
152 //        Assert.False(lastScope == null);
153 //
154 //        var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>
↳ >(lastScope.TempTransactionLogFilename);
155 //
156 //        Assert.True(transitions.Length == 1 && transitions[0].Before.IsNull() &&
↳ transitions[0].After.IsNull());
157 //
158 //        lastScope.DeleteFiles();
159 //    }
160 // }
161 //
162 // [Fact]
163 // public static void TransactionUserCodeErrorSomeDataSavedTest()
164 // {
165 //     // User Code Error (Autoreverted), some data saved
166 //     var itself = _constants.Itself;
167 //
168 //     TempLinksTestScope lastScope = null;
169 //     try
170 //     {
171 //         ulong l1;
172 //         ulong l2;
173 //
174 //         using (var scope = new TempLinksTestScope(useLog: true))
175 //         {
176 //             var links = scope.Links;
177 //             l1 = links.CreateAndUpdate(itself, itself);
178 //             l2 = links.CreateAndUpdate(itself, itself);
179 //
180 //             l2 = links.Update(l2, l2, l1, l2);
181 //
182 //             links.CreateAndUpdate(l2, itself);
183 //             links.CreateAndUpdate(l2, itself);
184 //
185 //             links.Unsync.DisposeIfPossible();
186 //
187 //             Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transitio
↳ n>(scope.TempTransactionLogFilename);
188 //         }
189 //
190 //         using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
↳ useLog: true))
191 //         {
192 //             var links = scope.Links;
193 //             var transactionsLayer = (UInt64LinksTransactionsLayer)links.Unsync;
194 //             using (var transaction = transactionsLayer.BeginTransaction())
195 //             {
196 //                 l2 = links.Update(l2, l1);
197 //
198 //                 links.Delete(l2);
199 //
200 //                 ExceptionThrower();
201 //
202 //                 transaction.Commit();
203 //             }
204 //
205 //             Global.Trash = links.Count();
206 //         }
207 //     }
208 // catch
209 // {
210 //     Assert.False(lastScope == null);
211 // }

```

```

212 // Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(l
↪ astScope.TempTransactionLogFilename);
213 //
214 // lastScope.DeleteFiles();
215 // }
216 // }
217 //
218 // [Fact]
219 // public static void TransactionCommit()
220 // {
221 //     var itself = _constants.Itself;
222 //
223 //     var tempDatabaseFilename = Path.GetTempFileName();
224 //     var tempTransactionLogFilename = Path.GetTempFileName();
225 //
226 //     // Commit
227 //     using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
↪ UInt64UnitedMemoryLinks(tempDatabaseFilename), tempTransactionLogFilename))
228 //     using (var links = new UInt64Links(memoryAdapter))
229 //     {
230 //         using (var transaction = memoryAdapter.BeginTransaction())
231 //         {
232 //             var l1 = links.CreateAndUpdate(itself, itself);
233 //             var l2 = links.CreateAndUpdate(itself, itself);
234 //
235 //             Global.Trash = links.Update(l2, l2, l1, l2);
236 //
237 //             links.Delete(l1);
238 //
239 //             transaction.Commit();
240 //         }
241 //
242 //         Global.Trash = links.Count();
243 //     }
244 //
245 //     Global.Trash =
↪ FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTransactionLogFilename);
246 // }
247 //
248 // [Fact]
249 // public static void TransactionDamage()
250 // {
251 //     var itself = _constants.Itself;
252 //
253 //     var tempDatabaseFilename = Path.GetTempFileName();
254 //     var tempTransactionLogFilename = Path.GetTempFileName();
255 //
256 //     // Commit
257 //     using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
↪ UInt64UnitedMemoryLinks(tempDatabaseFilename), tempTransactionLogFilename))
258 //     using (var links = new UInt64Links(memoryAdapter))
259 //     {
260 //         using (var transaction = memoryAdapter.BeginTransaction())
261 //         {
262 //             var l1 = links.CreateAndUpdate(itself, itself);
263 //             var l2 = links.CreateAndUpdate(itself, itself);
264 //
265 //             Global.Trash = links.Update(l2, l2, l1, l2);
266 //
267 //             links.Delete(l1);
268 //
269 //             transaction.Commit();
270 //         }
271 //
272 //         Global.Trash = links.Count();
273 //     }
274 //
275 //     Global.Trash =
↪ FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTransactionLogFilename);
276 //
277 //     // Damage database
278 //
279 //     FileHelpers.WriteFirst(tempTransactionLogFilename, new
↪ UInt64LinksTransactionsLayer.Transition(new UniqueTimestampFactory(), 555));
280 //
281 //     // Try load damaged database
282 //     try

```

```

283 //      {
284 //          // TODO: Fix
285 //          using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
↳ UInt64UnitedMemoryLinks(tempDatabaseFilename), tempTransactionLogFilename))
286 //          using (var links = new UInt64Links(memoryAdapter))
287 //          {
288 //              Global.Trash = links.Count();
289 //          }
290 //      }
291 //      catch (NotSupportedException ex)
292 //      {
293 //          Assert.True(ex.Message == "Database is damaged, autorecovery is not supported
↳ yet.");
294 //      }
295 //
296 //      Global.Trash =
↳ FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTransactionLogFilename);
297 //
298 //      File.Delete(tempDatabaseFilename);
299 //      File.Delete(tempTransactionLogFilename);
300 //  }
301 //
302 //  [Fact]
303 //  public static void Bug1Test()
304 //  {
305 //      var tempDatabaseFilename = Path.GetTempFileName();
306 //      var tempTransactionLogFilename = Path.GetTempFileName();
307 //
308 //      var itself = _constants.Itself;
309 //
310 //      // User Code Error (Autoreverted), some data saved
311 //      try
312 //      {
313 //          ulong l1;
314 //          ulong l2;
315 //
316 //          using (var memory = new UInt64UnitedMemoryLinks(tempDatabaseFilename))
317 //          using (var memoryAdapter = new UInt64LinksTransactionsLayer(memory,
↳ tempTransactionLogFilename))
318 //          using (var links = new UInt64Links(memoryAdapter))
319 //          {
320 //              l1 = links.CreateAndUpdate(itself, itself);
321 //              l2 = links.CreateAndUpdate(itself, itself);
322 //
323 //              l2 = links.Update(l2, l2, l1, l2);
324 //
325 //              links.CreateAndUpdate(l2, itself);
326 //              links.CreateAndUpdate(l2, itself);
327 //          }
328 //
329 //          Global.Trash =
↳ FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTransactionLogFilename);
330 //
331 //          using (var memory = new UInt64UnitedMemoryLinks(tempDatabaseFilename))
332 //          using (var memoryAdapter = new UInt64LinksTransactionsLayer(memory,
↳ tempTransactionLogFilename))
333 //          using (var links = new UInt64Links(memoryAdapter))
334 //          {
335 //              using (var transaction = memoryAdapter.BeginTransaction())
336 //              {
337 //                  l2 = links.Update(l2, l1);
338 //
339 //                  links.Delete(l2);
340 //
341 //                  ExceptionThrower();
342 //
343 //                  transaction.Commit();
344 //              }
345 //
346 //              Global.Trash = links.Count();
347 //          }
348 //      }
349 //      catch
350 //      {
351 //          Global.Trash =
↳ FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTransactionLogFilename);
352 //      }

```

```

353 //
354 //         File.Delete(tempDatabaseFilename);
355 //         File.Delete(tempTransactionLogFilename);
356 //     }
357 //     private static void ExceptionThrower() => throw new InvalidOperationException();
358 //
359 //     [Fact]
360 //     public static void PathsTest()
361 //     {
362 //         var source = _constants.SourcePart;
363 //         var target = _constants.TargetPart;
364 //
365 //         using (var scope = new TempLinksTestScope())
366 //         {
367 //             var links = scope.Links;
368 //             var l1 = links.CreatePoint();
369 //             var l2 = links.CreatePoint();
370 //
371 //             var r1 = links.GetByKeys(l1, source, target, source);
372 //             var r2 = links.CheckPathExistence(l2, l2, l2, l2);
373 //         }
374 //     }
375 //
376 //     [Fact]
377 //     public static void RecursiveStringFormattingTest()
378 //     {
379 //         using (var scope = new TempLinksTestScope(useSequences: true))
380 //         {
381 //             var links = scope.Links;
382 //             var sequences = scope.Sequences; // TODO: Auto use sequences on Sequences
383 //
384 //             var a = links.CreatePoint();
385 //             var b = links.CreatePoint();
386 //             var c = links.CreatePoint();
387 //
388 //             var ab = links.GetOrCreate(a, b);
389 //             var cb = links.GetOrCreate(c, b);
390 //             var ac = links.GetOrCreate(a, c);
391 //
392 //             a = links.Update(a, c, b);
393 //             b = links.Update(b, a, c);
394 //             c = links.Update(c, a, b);
395 //
396 //             Debug.WriteLine(links.FormatStructure(ab, link => link.IsFullPoint(), true));
397 //             Debug.WriteLine(links.FormatStructure(cb, link => link.IsFullPoint(), true));
398 //             Debug.WriteLine(links.FormatStructure(ac, link => link.IsFullPoint(), true));
399 //
400 //             Assert.True(links.FormatStructure(cb, link => link.IsFullPoint(), true) ==
401 // ↪ "5:(4:5 (6:5 4)) 6");
402 //             Assert.True(links.FormatStructure(ac, link => link.IsFullPoint(), true) ==
403 // ↪ "6:(5:(4:5 6) 6) 4");
404 //             Assert.True(links.FormatStructure(ab, link => link.IsFullPoint(), true) ==
405 // ↪ "4:(5:4 (6:5 4)) 6");
406 //
407 //             // TODO: Think how to build balanced syntax tree while formatting structure
408 //             // (eg. "4:(5:4 6) (6:5 4)" instead of "4:(5:4 (6:5 4)) 6")
409 //
410 //             Assert.True(sequences.SafeFormatSequence(cb, DefaultFormatter, false) ==
411 // ↪ "{{5}}{{5}}{{4}}{{6}}}");
412 //             Assert.True(sequences.SafeFormatSequence(ac, DefaultFormatter, false) ==
413 // ↪ "{{5}}{{6}}{{6}}{{4}}}");
414 //             Assert.True(sequences.SafeFormatSequence(ab, DefaultFormatter, false) ==
415 // ↪ "{{4}}{{5}}{{4}}{{6}}}");
416 //         }
417 //     }
418 //     private static void DefaultFormatter(StringBuilder sb, ulong link)
419 //     {
420 //         sb.Append(link.ToString());
421 //     }
422 //
423 //     #endregion
424 //
425 //     #region Performance
426 //
427 //     /*
428 //     public static void RunAllPerformanceTests()

```

```

422 // {
423 //     try
424 //     {
425 //         links.TestLinksInSteps();
426 //     }
427 //     catch (Exception ex)
428 //     {
429 //         ex.WriteToConsole();
430 //     }
431 //
432 //     return;
433 //
434 //     try
435 //     {
436 //         //ThreadPool.SetMaxThreads(2, 2);
437 //
438 //         // Запускаем все тесты дважды, чтобы первоначальная инициализация не повлияла
↪ на результат
439 //         // Также это дополнительно помогает в отладке
440 //         // Увеличивает вероятность попадания информации в кэши
441 //         for (var i = 0; i < 10; i++)
442 //         {
443 //             //0 - 10 ГБ
444 //             //Каждые 100 МБ срез цифр
445 //
446 //             //links.TestGetSourceFunction();
447 //             //links.TestGetSourceFunctionInParallel();
448 //             //links.TestGetTargetFunction();
449 //             //links.TestGetTargetFunctionInParallel();
450 //             links.Create64BillionLinks();
451 //
452 //             links.TestRandomSearchFixed();
453 //             //links.Create64BillionLinksInParallel();
454 //             links.TestEachFunction();
455 //             //links.TestForeach();
456 //             //links.TestParallelForeach();
457 //         }
458 //
459 //         links.TestDeletionOfAllLinks();
460 //
461 //     }
462 //     catch (Exception ex)
463 //     {
464 //         ex.WriteToConsole();
465 //     }
466 // }*/
467 //
468 // /*
469 // public static void TestLinksInSteps()
470 // {
471 //     const long gibibyte = 1024 * 1024 * 1024;
472 //     const long mebibyte = 1024 * 1024;
473 //
474 //     var totalLinksToCreate = gibibyte /
↪ Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
475 //     var linksStep = 102 * mebibyte /
↪ Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
476 //
477 //     var creationMeasurements = new List<TimeSpan>();
478 //     var searchMeasurements = new List<TimeSpan>();
479 //     var deletionMeasurements = new List<TimeSpan>();
480 //
481 //     GetBaseRandomLoopOverhead(linksStep);
482 //     GetBaseRandomLoopOverhead(linksStep);
483 //
484 //     var stepLoopOverhead = GetBaseRandomLoopOverhead(linksStep);
485 //
486 //     ConsoleHelpers.Debug("Step loop overhead: {0}.", stepLoopOverhead);
487 //
488 //     var loops = totalLinksToCreate / linksStep;
489 //
490 //     for (int i = 0; i < loops; i++)
491 //     {
492 //         creationMeasurements.Add(Measure(() => links.RunRandomCreations(linksStep)));
493 //         searchMeasurements.Add(Measure(() => links.RunRandomSearches(linksStep)));
494 //
495 //         Console.Write("\rC + S {0}/{1}", i + 1, loops);
496 //     }

```

```

497 //
498 //         ConsoleHelpers.Debug();
499 //
500 //         for (int i = 0; i < loops; i++)
501 //         {
502 //             deletionMeasurements.Add(Measure(() => links.RunRandomDeletions(linksStep)));
503 //
504 //             Console.Write("\rD {0}/{1}", i + 1, loops);
505 //         }
506 //
507 //         ConsoleHelpers.Debug();
508 //
509 //         ConsoleHelpers.Debug("C S D");
510 //
511 //         for (int i = 0; i < loops; i++)
512 //         {
513 //             ConsoleHelpers.Debug("{0} {1} {2}", creationMeasurements[i],
↵ searchMeasurements[i], deletionMeasurements[i]);
514 //         }
515 //
516 //         ConsoleHelpers.Debug("C S D (no overhead)");
517 //
518 //         for (int i = 0; i < loops; i++)
519 //         {
520 //             ConsoleHelpers.Debug("{0} {1} {2}", creationMeasurements[i] -
↵ stepLoopOverhead, searchMeasurements[i] - stepLoopOverhead, deletionMeasurements[i] -
↵ stepLoopOverhead);
521 //         }
522 //
523 //         ConsoleHelpers.Debug("All tests done. Total links left in database: {0}.",
↵ links.Total);
524 //     }
525 //
526 //     private static void CreatePoints(this Platform.Links.Data.Core.Doublets.Links links,
↵ long amountToCreate)
527 //     {
528 //         for (long i = 0; i < amountToCreate; i++)
529 //             links.Create(0, 0);
530 //     }
531 //
532 //     private static TimeSpan GetBaseRandomLoopOverhead(long loops)
533 //     {
534 //         return Measure(() =>
535 //         {
536 //             ulong maxValue = RandomHelpers.DefaultFactory.NextUInt64();
537 //             ulong result = 0;
538 //             for (long i = 0; i < loops; i++)
539 //             {
540 //                 var source = RandomHelpers.DefaultFactory.NextUInt64(maxValue);
541 //                 var target = RandomHelpers.DefaultFactory.NextUInt64(maxValue);
542 //
543 //                 result += maxValue + source + target;
544 //             }
545 //             Global.Trash = result;
546 //         });
547 //     }
548 //     */
549 //
550 //     [Fact(Skip = "performance test")]
551 //     public static void GetSourceTest()
552 //     {
553 //         using (var scope = new TempLinksTestScope())
554 //         {
555 //             var links = scope.Links;
556 //             ConsoleHelpers.Debug("Testing GetSource function with {0} Iterations.",
↵ Iterations);
557 //
558 //             ulong counter = 0;
559 //
560 //             //var firstLink = links.First();
561 //             // Создаём одну связь, из которой будет производить считывание
562 //             var firstLink = links.Create();
563 //
564 //             var sw = Stopwatch.StartNew();
565 //
566 //             // Тестируем саму функцию
567 //             for (ulong i = 0; i < Iterations; i++)

```

```

568 //      {
569 //          counter += links.GetSource(firstLink);
570 //      }
571 //
572 //      var elapsedTime = sw.Elapsed;
573 //
574 //      var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
575 //
576 //      // Удаляем связь, из которой производилось считывание
577 //      links.Delete(firstLink);
578 //
579 //      ConsoleHelpers.Debug(
580 //          "{0} Iterations of GetSource function done in {1} ({2} Iterations per
↵ second), counter result: {3}",
581 //              Iterations, elapsedTime, (long)iterationsPerSecond, counter);
582 //      }
583 //
584 //
585 //      [Fact(Skip = "performance test")]
586 //      public static void GetSourceInParallel()
587 //      {
588 //          using (var scope = new TempLinksTestScope())
589 //          {
590 //              var links = scope.Links;
591 //              ConsoleHelpers.Debug("Testing GetSource function with {0} Iterations in
↵ parallel.", Iterations);
592 //
593 //              long counter = 0;
594 //
595 //              //var firstLink = links.First();
596 //              var firstLink = links.Create();
597 //
598 //              var sw = Stopwatch.StartNew();
599 //
600 //              // Тестируем саму функцию
601 //              Parallel.For(0, Iterations, x =>
602 //              {
603 //                  Interlocked.Add(ref counter, (long)links.GetSource(firstLink));
604 //                  //Interlocked.Increment(ref counter);
605 //              });
606 //
607 //              var elapsedTime = sw.Elapsed;
608 //
609 //              var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
610 //
611 //              links.Delete(firstLink);
612 //
613 //              ConsoleHelpers.Debug(
614 //                  "{0} Iterations of GetSource function done in {1} ({2} Iterations per
↵ second), counter result: {3}",
615 //                      Iterations, elapsedTime, (long)iterationsPerSecond, counter);
616 //              }
617 //
618 //
619 //      [Fact(Skip = "performance test")]
620 //      public static void TestGetTarget()
621 //      {
622 //          using (var scope = new TempLinksTestScope())
623 //          {
624 //              var links = scope.Links;
625 //              ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations.",
↵ Iterations);
626 //
627 //              ulong counter = 0;
628 //
629 //              //var firstLink = links.First();
630 //              var firstLink = links.Create();
631 //
632 //              var sw = Stopwatch.StartNew();
633 //
634 //              for (ulong i = 0; i < Iterations; i++)
635 //              {
636 //                  counter += links.GetTarget(firstLink);
637 //              }
638 //
639 //              var elapsedTime = sw.Elapsed;
640 //

```

```

641 //                var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
642 //
643 //                links.Delete(firstLink);
644 //
645 //                ConsoleHelpers.Debug(
646 //                    "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
↵ second), counter result: {3}",
647 //                        Iterations, elapsedTime, (long)iterationsPerSecond, counter);
648 //                }
649 //            }
650 //
651 //            [Fact(Skip = "performance test")]
652 //            public static void TestGetTargetInParallel()
653 //            {
654 //                using (var scope = new TempLinksTestScope())
655 //                {
656 //                    var links = scope.Links;
657 //                    ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations in
↵ parallel.", Iterations);
658 //
659 //                    long counter = 0;
660 //
661 //                    //var firstLink = links.First();
662 //                    var firstLink = links.Create();
663 //
664 //                    var sw = Stopwatch.StartNew();
665 //
666 //                    Parallel.For(0, Iterations, x =>
667 //                    {
668 //                        Interlocked.Add(ref counter, (long)links.GetTarget(firstLink));
669 //                        //Interlocked.Increment(ref counter);
670 //                    });
671 //
672 //                    var elapsedTime = sw.Elapsed;
673 //
674 //                    var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
675 //
676 //                    links.Delete(firstLink);
677 //
678 //                    ConsoleHelpers.Debug(
679 //                        "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
↵ second), counter result: {3}",
680 //                            Iterations, elapsedTime, (long)iterationsPerSecond, counter);
681 //                    }
682 //                }
683 //
684 //                // TODO: Заполнить базу данных перед тестом
685 //                /*
686 //                [Fact]
687 //                public void TestRandomSearchFixed()
688 //                {
689 //                    var tempFilename = Path.GetTempFileName();
690 //
691 //                    using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
↵ DefaultLinksSizeStep))
692 //                    {
693 //                        long iterations = 64 * 1024 * 1024 /
↵ Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
694 //
695 //                        ulong counter = 0;
696 //                        var maxLink = links.Total;
697 //
698 //                        ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.",
↵ iterations);
699 //
700 //                        var sw = Stopwatch.StartNew();
701 //
702 //                        for (var i = iterations; i > 0; i--)
703 //                        {
704 //                            var source =
↵ RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
705 //                            var target =
↵ RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
706 //
707 //                            counter += links.Search(source, target);
708 //                        }
709 //

```



```

710 //         var elapsedTime = sw.Elapsed;
711 //
712 //         var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;
713 //
714 //         ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
↵ Iterations per second), c: {3}", iterations, elapsedTime, (long)iterationsPerSecond,
↵ counter);
715 //     }
716 //
717 //     File.Delete(tempFilename);
718 // }*/
719 //
720 // [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
721 // public static void TestRandomSearchAll()
722 // {
723 //     using (var scope = new TempLinksTestScope())
724 //     {
725 //         var links = scope.Links;
726 //         ulong counter = 0;
727 //
728 //         var maxLink = links.Count();
729 //
730 //         var iterations = links.Count();
731 //
732 //         ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.",
↵ links.Count());
733 //
734 //         var sw = Stopwatch.StartNew();
735 //
736 //         for (var i = iterations; i > 0; i--)
737 //         {
738 //             var linksAddressRange = new
↵ Range<ulong>(_constants.InternalReferencesRange.Minimum, maxLink);
739 //
740 //             var source = RandomHelpers.Default.NextUInt64(linksAddressRange);
741 //             var target = RandomHelpers.Default.NextUInt64(linksAddressRange);
742 //
743 //             counter += links.SearchOrDefault(source, target);
744 //         }
745 //
746 //         var elapsedTime = sw.Elapsed;
747 //
748 //         var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;
749 //
750 //         ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
↵ Iterations per second), c: {3}",
↵ iterations, elapsedTime, (long)iterationsPerSecond, counter);
751 //     }
752 // }
753 //
754 // [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
755 // public static void TestEach()
756 // {
757 //     using (var scope = new TempLinksTestScope())
758 //     {
759 //         var links = scope.Links;
760 //
761 //         var counter = new Counter<IList<ulong>, ulong>(links.Constants.Continue);
762 //
763 //         ConsoleHelpers.Debug("Testing Each function.");
764 //
765 //         var sw = Stopwatch.StartNew();
766 //
767 //         links.Each(counter.IncrementAndReturnTrue);
768 //
769 //         var elapsedTime = sw.Elapsed;
770 //
771 //         var linksPerSecond = counter.Count / elapsedTime.TotalSeconds;
772 //
773 //         ConsoleHelpers.Debug("{0} Iterations of Each's handler function done in {1}
↵ ({2} links per second)",
↵ counter, elapsedTime, (long)linksPerSecond);
774 //     }
775 // }
776 //
777 // }
778 //
779 // /*
780 // [Fact]

```

```

781 //         public static void TestForeach()
782 //         {
783 //             var tempFilename = Path.GetTempFileName();
784 //
785 //             using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
↵ DefaultLinksSizeStep))
786 //             {
787 //                 ulong counter = 0;
788 //
789 //                 ConsoleHelpers.Debug("Testing foreach through links.");
790 //
791 //                 var sw = Stopwatch.StartNew();
792 //
793 //                 //foreach (var link in links)
794 //                 //{
795 //                     counter++;
796 //                 //}
797 //
798 //                 var elapsedTime = sw.Elapsed;
799 //
800 //                 var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
801 //
802 //                 ConsoleHelpers.Debug("{0} Iterations of Foreach's handler block done in {1}
↵ ({2} links per second)", counter, elapsedTime, (long)linksPerSecond);
803 //             }
804 //
805 //             File.Delete(tempFilename);
806 //         }
807 //     */
808 //
809 //     /*
810 //     [Fact]
811 //     public static void TestParallelForeach()
812 //     {
813 //         var tempFilename = Path.GetTempFileName();
814 //
815 //         using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
↵ DefaultLinksSizeStep))
816 //         {
817 //
818 //             long counter = 0;
819 //
820 //             ConsoleHelpers.Debug("Testing parallel foreach through links.");
821 //
822 //             var sw = Stopwatch.StartNew();
823 //
824 //             //Parallel.ForEach((IEnumerable<ulong>)links, x =>
825 //             //{
826 //                 Interlocked.Increment(ref counter);
827 //             //});
828 //
829 //             var elapsedTime = sw.Elapsed;
830 //
831 //             var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
832 //
833 //             ConsoleHelpers.Debug("{0} Iterations of Parallel Foreach's handler block done
↵ in {1} ({2} links per second)", counter, elapsedTime, (long)linksPerSecond);
834 //         }
835 //
836 //         File.Delete(tempFilename);
837 //     }
838 //     */
839 //
840 //     [Fact(Skip = "performance test")]
841 //     public static void Create64BillionLinks()
842 //     {
843 //         using (var scope = new TempLinksTestScope())
844 //         {
845 //             var links = scope.Links;
846 //             var linksBeforeTest = links.Count();
847 //
848 //             long linksToCreate = 64 * 1024 * 1024 /
↵ UInt64UnitedMemoryLinks.LinkSizeInBytes;
849 //
850 //             ConsoleHelpers.Debug("Creating {0} links.", linksToCreate);
851 //
852 //             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;

```

```

7  using Xunit.Abstractions;
8  using TLinkAddress = System.UInt64;
9
10 namespace Platform.Data.Doublets.Sequences.Tests
11 {
12     public class UInt64LinksExtensionsTests
13     {
14         public static ILinks<TLinkAddress> CreateLinks() => CreateLinks<TLinkAddress>(new
            ↳ Platform.IO.TemporaryFile());
15
16         public static ILinks<TLinkAddress> CreateLinks<TLinkAddress>(string dataDBFilename)
17         {
18             var linksConstants = new
                ↳ LinksConstants<TLinkAddress>(enableExternalReferencesSupport: true);
19             return new UnitedMemoryLinks<TLinkAddress>(new
                ↳ FileMappedResizableDirectMemory(dataDBFilename),
                ↳ UnitedMemoryLinks<TLinkAddress>.DefaultLinksSizeStep, linksConstants,
                ↳ IndexTreeType.Default);
20         }
21         [Fact]
22         public void FormatStructureWithExternalReferenceTest()
23         {
24             ILinks<TLinkAddress> links = CreateLinks();
25             TLinkAddress zero = default;
26             var one = Arithmetic.Increment(zero);
27             var markerIndex = one;
28             var meaningRoot = links.GetOrCreate(markerIndex, markerIndex);
29             var numberMarker = links.GetOrCreate(meaningRoot, Arithmetic.Increment(ref
                ↳ markerIndex));
30             AddressToRawNumberConverter<TLinkAddress> addressToNumberConverter = new();
31             var numberAddress = addressToNumberConverter.Convert(1);
32             var numberLink = links.GetOrCreate(numberMarker, numberAddress);
33             var linkNotation = links.FormatStructure(numberLink, link => link.IsFullPoint(),
                ↳ true);
34             Assert.Equal("(3:(2:1 2) 18446744073709551615)", linkNotation);
35         }
36     }
37 }

```

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

```

1  // using Xunit;
2  // using Platform.Random;
3  // using Platform.Data.Doublets.Numbers.Unary;
4  //
5  // namespace Platform.Data.Doublets.Sequences.Tests
6  // {
7  //     public static class UnaryNumberConvertersTests
8  //     {
9  //         [Fact]
10         //         public static void ConvertersTest()
11         //         {
12             //             using (var scope = new TempLinksTestScope())
13             //             {
14                 //                 const int N = 10;
15                 //                 var links = scope.Links;
16                 //                 var meaningRoot = links.CreatePoint();
17                 //                 var one = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
18                 //                 var powerOf2ToUnaryNumberConverter = new
19                 ↳ PowerOf2ToUnaryNumberConverter<ulong>(links, one);
20                 //                 var toUnaryNumberConverter = new AddressToUnaryNumberConverter<ulong>(links,
21                 ↳ powerOf2ToUnaryNumberConverter);
22                 //                 var random = new System.Random(0);
23                 //                 ulong[] numbers = new ulong[N];
24                 //                 ulong[] unaryNumbers = new ulong[N];
25                 //                 for (int i = 0; i < N; i++)
26                 //                 {
27                     //                     numbers[i] = random.NextUInt64();
28                     //                     unaryNumbers[i] = toUnaryNumberConverter.Convert(numbers[i]);
29                 //                 }
30                 //                 var fromUnaryNumberConverterUsingOrOperation = new
31                 ↳ UnaryNumberToAddressOrOperationConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
32                 //                 var fromUnaryNumberConverterUsingAddOperation = new
33                 ↳ UnaryNumberToAddressAddOperationConverter<ulong>(links, one);
34                 //                 for (int i = 0; i < N; i++)
35                 //                 {
36                     //                     Assert.Equal(numbers[i],
37                     ↳ fromUnaryNumberConverterUsingOrOperation.Convert(unaryNumbers[i]));
38                 //                 }
39             //             }
40         //         }
41     //     }
42 // }

```

```

33 // Assert.Equal(numbers[i],
    ↳ fromUnaryNumberConverterUsingAddOperation.Convert(unaryNumbers[i]));
34 // }
35 // }
36 // }
37 // }
38 // }

```

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

```

1 // using Xunit;
2 // using Platform.Converters;
3 // using Platform.Memory;
4 // using Platform.Reflection;
5 // using Platform.Scopes;
6 // using Platform.Data.Numbers.Raw;
7 // using Platform.Data.Doublets.Incrementers;
8 // using Platform.Data.Doublets.Numbers.Unary;
9 // using Platform.Data.Doublets.PropertyOperators;
10 // using Platform.Data.Doublets.Sequences.Converters;
11 // using Platform.Data.Doublets.Sequences.Indexes;
12 // using Platform.Data.Doublets.Sequences.Walkers;
13 // using Platform.Data.Doublets.Unicode;
14 // using Platform.Data.Doublets.Memory.United.Generic;
15 // using Platform.Data.Doublets.CriterionMatchers;
16 //
17 // namespace Platform.Data.Doublets.Sequences.Tests
18 // {
19 //     public static class UnicodeConvertersTests
20 //     {
21 //         [Fact]
22 //         public static void CharAndUnaryNumberUnicodeSymbolConvertersTest()
23 //         {
24 //             using (var scope = new TempLinksTestScope())
25 //             {
26 //                 var links = scope.Links;
27 //                 var meaningRoot = links.CreatePoint();
28 //                 var one = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
29 //                 var powerOf2ToUnaryNumberConverter = new
    ↳ PowerOf2ToUnaryNumberConverter<ulong>(links, one);
30 //                 var addressToUnaryNumberConverter = new
    ↳ AddressToUnaryNumberConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
31 //                 var unaryNumberToAddressConverter = new
    ↳ UnaryNumberToAddressOrOperationConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
32 //                 TestCharAndUnicodeSymbolConverters(links, meaningRoot,
    ↳ addressToUnaryNumberConverter, unaryNumberToAddressConverter);
33 //             }
34 //         }
35 //
36 //         [Fact]
37 //         public static void CharAndRawNumberUnicodeSymbolConvertersTest()
38 //         {
39 //             using (var scope = new Scope<Types<HeapResizableDirectMemory,
    ↳ UnitedMemoryLinks<ulong>>>())
40 //             {
41 //                 var links = scope.Use<ILinks<ulong>>();
42 //                 var meaningRoot = links.CreatePoint();
43 //                 var addressToRawNumberConverter = new AddressToRawNumberConverter<ulong>();
44 //                 var rawNumberToAddressConverter = new RawNumberToAddressConverter<ulong>();
45 //                 TestCharAndUnicodeSymbolConverters(links, meaningRoot,
    ↳ addressToRawNumberConverter, rawNumberToAddressConverter);
46 //             }
47 //         }
48 //         private static void TestCharAndUnicodeSymbolConverters(ILinks<ulong> links, ulong
    ↳ meaningRoot, IConverter<ulong> addressToNumberConverter, IConverter<ulong>
    ↳ numberToAddressConverter)
49 //         {
50 //             var unicodeSymbolMarker = links.CreateAndUpdate(meaningRoot,
    ↳ links.Constants.Itself);
51 //             var charToUnicodeSymbolConverter = new CharToUnicodeSymbolConverter<ulong>(links,
    ↳ addressToNumberConverter, unicodeSymbolMarker);
52 //             var originalCharacter = 'H';
53 //             var characterLink = charToUnicodeSymbolConverter.Convert(originalCharacter);
54 //             var unicodeSymbolCriterionMatcher = new TargetMatcher<ulong>(links,
    ↳ unicodeSymbolMarker);
55 //             var unicodeSymbolToCharConverter = new UnicodeSymbolToCharConverter<ulong>(links,
    ↳ numberToAddressConverter, unicodeSymbolCriterionMatcher);
56 //             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
↳ LinkToItsFrequencyNumberConverter<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

`./csharp/Platform.Data.Doublets.Sequences.Tests/BigIntegerConvertersTests.cs`, 144
`./csharp/Platform.Data.Doublets.Sequences.Tests/ByteConvertersTests.cs`, 146
`./csharp/Platform.Data.Doublets.Sequences.Tests/DefaultSequenceAppenderTests.cs`, 148
`./csharp/Platform.Data.Doublets.Sequences.Tests/ILinksExtensionsTests.cs`, 149
`./csharp/Platform.Data.Doublets.Sequences.Tests/OptimalVariantSequenceTests.cs`, 149
`./csharp/Platform.Data.Doublets.Sequences.Tests/RationalNumbersTests.cs`, 153
`./csharp/Platform.Data.Doublets.Sequences.Tests/ReadSequenceTests.cs`, 155
`./csharp/Platform.Data.Doublets.Sequences.Tests/SequencesTests.cs`, 156
`./csharp/Platform.Data.Doublets.Sequences.Tests/TempLinksTestScope.cs`, 171
`./csharp/Platform.Data.Doublets.Sequences.Tests/TestExtensions.cs`, 172
`./csharp/Platform.Data.Doublets.Sequences.Tests/UInt64LinksTests.cs`, 175
`./csharp/Platform.Data.Doublets.Sequences.Tests/UInt64LinksExtensionsTests.cs`, 187
`./csharp/Platform.Data.Doublets.Sequences.Tests/UnaryNumberConvertersTests.cs`, 188
`./csharp/Platform.Data.Doublets.Sequences.Tests/UnicodeConvertersTests.cs`, 189
`./csharp/Platform.Data.Doublets.Sequences/Converters/BalancedVariantConverter.cs`, 1
`./csharp/Platform.Data.Doublets.Sequences/Converters/CompressingConverter.cs`, 2
`./csharp/Platform.Data.Doublets.Sequences/Converters/LinksListToSequenceConverterBase.cs`, 6
`./csharp/Platform.Data.Doublets.Sequences/Converters/OptimalVariantConverter.cs`, 7
`./csharp/Platform.Data.Doublets.Sequences/Converters/SequenceToToltsLocalElementLevelsConverter.cs`, 9
`./csharp/Platform.Data.Doublets.Sequences/CriterionMatchers/DefaultSequenceElementCriterionMatcher.cs`, 11
`./csharp/Platform.Data.Doublets.Sequences/CriterionMatchers/MarkedSequenceCriterionMatcher.cs`, 11
`./csharp/Platform.Data.Doublets.Sequences/CriterionMatchers/UnicodeSequenceMatcher.cs`, 12
`./csharp/Platform.Data.Doublets.Sequences/DefaultSequenceAppender.cs`, 13
`./csharp/Platform.Data.Doublets.Sequences/DuplicateSegmentsCounter.cs`, 14
`./csharp/Platform.Data.Doublets.Sequences/DuplicateSegmentsProvider.cs`, 15
`./csharp/Platform.Data.Doublets.Sequences/Frequencies/Cache/LinkFrequenciesCache.cs`, 19
`./csharp/Platform.Data.Doublets.Sequences/Frequencies/Cache/LinkFrequency.cs`, 22
`./csharp/Platform.Data.Doublets.Sequences/Frequencies/Cache/LinkToToltsFrequencyValueConverter.cs`, 23
`./csharp/Platform.Data.Doublets.Sequences/Frequencies/Counters/MarkedSequenceSymbolFrequencyOneOffCounter.cs`, 24
`./csharp/Platform.Data.Doublets.Sequences/Frequencies/Counters/SequenceSymbolFrequencyOneOffCounter.cs`, 25
`./csharp/Platform.Data.Doublets.Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyCounter.cs`, 27
`./csharp/Platform.Data.Doublets.Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyOneOffCounter.cs`, 27
`./csharp/Platform.Data.Doublets.Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyCounter.cs`, 28
`./csharp/Platform.Data.Doublets.Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyOneOffCounter.cs`, 29
`./csharp/Platform.Data.Doublets.Sequences/HeightProviders/CachedSequenceHeightProvider.cs`, 31
`./csharp/Platform.Data.Doublets.Sequences/HeightProviders/DefaultSequenceRightHeightProvider.cs`, 32
`./csharp/Platform.Data.Doublets.Sequences/HeightProviders/ISequenceHeightProvider.cs`, 33
`./csharp/Platform.Data.Doublets.Sequences/Incrementers/FrequencyIncrementer.cs`, 33
`./csharp/Platform.Data.Doublets.Sequences/Incrementers/UnaryNumberIncrementer.cs`, 34
`./csharp/Platform.Data.Doublets.Sequences/Indexes/CachedFrequencyIncrementingSequenceIndex.cs`, 35
`./csharp/Platform.Data.Doublets.Sequences/Indexes/FrequencyIncrementingSequenceIndex.cs`, 37
`./csharp/Platform.Data.Doublets.Sequences/Indexes/ISequenceIndex.cs`, 38
`./csharp/Platform.Data.Doublets.Sequences/Indexes/SequenceIndex.cs`, 39
`./csharp/Platform.Data.Doublets.Sequences/Indexes/SynchronizedSequenceIndex.cs`, 40
`./csharp/Platform.Data.Doublets.Sequences/Indexes/Unindex.cs`, 41
`./csharp/Platform.Data.Doublets.Sequences/Numbers/Byte/ByteListToRawSequenceConverter.cs`, 42
`./csharp/Platform.Data.Doublets.Sequences/Numbers/Byte/RawSequenceToByteListConverter.cs`, 46
`./csharp/Platform.Data.Doublets.Sequences/Numbers/Rational/DecimalToRationalConverter.cs`, 48
`./csharp/Platform.Data.Doublets.Sequences/Numbers/Rational/RationalToDecimalConverter.cs`, 50
`./csharp/Platform.Data.Doublets.Sequences/Numbers/Raw/BigIntegerToRawNumberSequenceConverter.cs`, 51
`./csharp/Platform.Data.Doublets.Sequences/Numbers/Raw/LongRawNumberSequenceToNumberConverter.cs`, 52
`./csharp/Platform.Data.Doublets.Sequences/Numbers/Raw/NumberToLongRawNumberSequenceConverter.cs`, 53
`./csharp/Platform.Data.Doublets.Sequences/Numbers/Raw/RawNumberSequenceToBigIntegerConverter.cs`, 55
`./csharp/Platform.Data.Doublets.Sequences/Numbers/Unary/AddressToUnaryNumberConverter.cs`, 56
`./csharp/Platform.Data.Doublets.Sequences/Numbers/Unary/LinkToToltsFrequencyNumberConverter.cs`, 57
`./csharp/Platform.Data.Doublets.Sequences/Numbers/Unary/PowerOf2ToUnaryNumberConverter.cs`, 58
`./csharp/Platform.Data.Doublets.Sequences/Numbers/Unary/UnaryNumberToAddressAddOperationConverter.cs`, 59
`./csharp/Platform.Data.Doublets.Sequences/Numbers/Unary/UnaryNumberToAddressOrOperationConverter.cs`, 61
`./csharp/Platform.Data.Doublets.Sequences/Sequences.Experiments.cs`, 62
`./csharp/Platform.Data.Doublets.Sequences/Sequences.cs`, 100
`./csharp/Platform.Data.Doublets.Sequences/SequencesExtensions.cs`, 115
`./csharp/Platform.Data.Doublets.Sequences/SequencesOptions.cs`, 116
`./csharp/Platform.Data.Doublets.Sequences/TestExtensions.cs`, 120
`./csharp/Platform.Data.Doublets.Sequences/Time/DateTimeToLongRawNumberSequenceConverter.cs`, 120
`./csharp/Platform.Data.Doublets.Sequences/Time/LongRawNumberSequenceToDateTimeConverter.cs`, 121

- ./csharp/Platform.Data.Doublets.Sequences/UInt64LinksExtensions.cs, 122
- ./csharp/Platform.Data.Doublets.Sequences/Unicode/CharToUnicodeSymbolConverter.cs, 122
- ./csharp/Platform.Data.Doublets.Sequences/Unicode/StringToUnicodeSequenceConverter.cs, 123
- ./csharp/Platform.Data.Doublets.Sequences/Unicode/StringToUnicodeSymbolsListConverter.cs, 126
- ./csharp/Platform.Data.Doublets.Sequences/Unicode/UnicodeMap.cs, 127
- ./csharp/Platform.Data.Doublets.Sequences/Unicode/UnicodeSequenceToStringConverter.cs, 132
- ./csharp/Platform.Data.Doublets.Sequences/Unicode/UnicodeSymbolToCharConverter.cs, 133
- ./csharp/Platform.Data.Doublets.Sequences/Unicode/UnicodeSymbolsListToUnicodeSequenceConverter.cs, 134
- ./csharp/Platform.Data.Doublets.Sequences/Walkers/ISequenceWalker.cs, 136
- ./csharp/Platform.Data.Doublets.Sequences/Walkers/LeftSequenceWalker.cs, 136
- ./csharp/Platform.Data.Doublets.Sequences/Walkers/LeveledSequenceWalker.cs, 138
- ./csharp/Platform.Data.Doublets.Sequences/Walkers/RightSequenceWalker.cs, 140
- ./csharp/Platform.Data.Doublets.Sequences/Walkers/SequenceWalkerBase.cs, 142