

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     public class BalancedVariantConverter<TLink> : LinksListToSequenceConverterBase<TLink>
9     {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public BalancedVariantConverter(ILinks<TLink> links) : base(links) { }
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public override TLink Convert(ICollection<TLink> sequence)
15         {
16             var length = sequence.Count;
17             if (length < 1)
18             {
19                 return default;
20             }
21             if (length == 1)
22             {
23                 return sequence[0];
24             }
25             // Make copy of next layer
26             if (length > 2)
27             {
28                 // TODO: Try to use stackalloc (which at the moment is not working with
29                 //      ↪ generics) but will be possible with Sigil
30                 var halvedSequence = new TLink[(length / 2) + (length % 2)];
31                 HalveSequence(halvedSequence, sequence, length);
32                 sequence = halvedSequence;
33                 length = halvedSequence.Length;
34             }
35             // Keep creating layer after layer
36             while (length > 2)
37             {
38                 HalveSequence(sequence, sequence, length);
39                 length = (length / 2) + (length % 2);
40             }
41             return _links.GetOrCreate(sequence[0], sequence[1]);
42         }
43
44         [MethodImpl(MethodImplOptions.AggressiveInlining)]
45         private void HalveSequence(ICollection<TLink> destination, ICollection<TLink> source, int length)
46         {
47             var loopedLength = length - (length % 2);
48             for (var i = 0; i < loopedLength; i += 2)
49             {
50                 destination[i / 2] = _links.GetOrCreate(source[i], source[i + 1]);
51             }
52             if (length > loopedLength)
53             {
54                 destination[length / 2] = source[length - 1];
55             }
56         }
57     }
58 }
```

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
16     ///      ↪ Links на этапе сжатия.
17     /// А именно будет создаваться временный список пар необходимых для выполнения сжатия, в
18     ///      ↪ таком случае тип значения элемента массива может быть любым, как char так и ulong.
19 }
```

```

17  ///      Как только список/словарь пар был выявлен можно разом выполнить создание всех этих
18  ↪ пар, а так же разом выполнить замену.
19  /// </remarks>
20  public class CompressingConverter<TLink> : LinksListToSequenceConverterBase<TLink>
21  {
22      private static readonly LinksConstants<TLink> _constants =
23      ↪ Default<LinksConstants<TLink>>.Instance;
24      private static readonly EqualityComparer<TLink> _equalityComparer =
25      ↪ EqualityComparer<TLink>.Default;
26      private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
27
28      private static readonly TLink _zero = default;
29      private static readonly TLink _one = Arithmetic.Increment(_zero);
30
31      private readonly IConverter<IList<TLink>, TLink> _baseConverter;
32      private readonly LinkFrequenciesCache<TLink> _doubletFrequenciesCache;
33      private readonly TLink _minFrequencyToCompress;
34      private readonly bool _doInitialFrequenciesIncrement;
35      private Doublet<TLink> _maxDoublet;
36      private LinkFrequency<TLink> _maxDoubletData;
37
38      private struct HalfDoublet
39      {
40          public TLink Element;
41          public LinkFrequency<TLink> DoubletData;
42
43          [MethodImpl(MethodImplOptions.AggressiveInlining)]
44          public HalfDoublet(TLink element, LinkFrequency<TLink> doubletData)
45          {
46              Element = element;
47              DoubletData = doubletData;
48          }
49
50          public override string ToString() => $"{Element}: ({DoubletData})";
51      }
52
53      [MethodImpl(MethodImplOptions.AggressiveInlining)]
54      public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
55      ↪ baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache)
56      : this(links, baseConverter, doubletFrequenciesCache, _one, true) { }
57
58      [MethodImpl(MethodImplOptions.AggressiveInlining)]
59      public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
60      ↪ baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache, bool
61      ↪ doInitialFrequenciesIncrement)
62      : this(links, baseConverter, doubletFrequenciesCache, _one,
63      ↪ doInitialFrequenciesIncrement) { }
64
65      [MethodImpl(MethodImplOptions.AggressiveInlining)]
66      public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
67      ↪ baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache, TLink
68      ↪ minFrequencyToCompress, bool doInitialFrequenciesIncrement)
69      : base(links)
70      {
71          _baseConverter = baseConverter;
72          _doubletFrequenciesCache = doubletFrequenciesCache;
73          if (_comparer.Compare(minFrequencyToCompress, _one) < 0)
74          {
75              minFrequencyToCompress = _one;
76          }
77          _minFrequencyToCompress = minFrequencyToCompress;
78          _doInitialFrequenciesIncrement = doInitialFrequenciesIncrement;
79          ResetMaxDoublet();
80      }
81
82      [MethodImpl(MethodImplOptions.AggressiveInlining)]
83      public override TLink Convert(IList<TLink> source) =>
84      ↪ _baseConverter.Convert(Compress(source));
85
86      /// <remarks>
87      /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte\_pair\_encoding .
88      /// Faster version (doublets' frequencies dictionary is not recreated).
89      /// </remarks>
90      [MethodImpl(MethodImplOptions.AggressiveInlining)]
91      private IList<TLink> Compress(IList<TLink> sequence)
92      {
93          if (sequence.IsNullOrEmpty())
94          {
95              return null;
96          }
97      }

```

```

87     if (sequence.Count == 1)
88     {
89         return sequence;
90     }
91     if (sequence.Count == 2)
92     {
93         return new[] { _links.GetOrCreate(sequence[0], sequence[1]) };
94     }
95     // TODO: arraypool with min size (to improve cache locality) or stackalloc with Sigil
96     var copy = new HalfDoublet[sequence.Count];
97     Doublet<TLink> doublet = default;
98     for (var i = 1; i < sequence.Count; i++)
99     {
100         doublet = new Doublet<TLink>(sequence[i - 1], sequence[i]);
101         LinkFrequency<TLink> data;
102         if (_doInitialFrequenciesIncrement)
103         {
104             data = _doubletFrequenciesCache.IncrementFrequency(ref doublet);
105         }
106         else
107         {
108             data = _doubletFrequenciesCache.GetFrequency(ref doublet);
109             if (data == null)
110             {
111                 throw new NotSupportedException("If you ask not to increment
112                     ↪ frequencies, it is expected that all frequencies for the sequence
113                     ↪ are prepared.");
114             }
115             copy[i - 1].Element = sequence[i - 1];
116             copy[i - 1].DoubletData = data;
117             UpdateMaxDoublet(ref doublet, data);
118         }
119         copy[sequence.Count - 1].Element = sequence[sequence.Count - 1];
120         copy[sequence.Count - 1].DoubletData = new LinkFrequency<TLink>();
121         if (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
122         {
123             var newLength = ReplaceDoublets(copy);
124             sequence = new TLink[newLength];
125             for (int i = 0; i < newLength; i++)
126             {
127                 sequence[i] = copy[i].Element;
128             }
129         }
130         return sequence;
131     }
132     /// <remarks>
133     /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte\_pair\_encoding
134     /// </remarks>
135     [MethodImpl(MethodImplOptions.AggressiveInlining)]
136     private int ReplaceDoublets(HalfDoublet[] copy)
137     {
138         var oldLength = copy.Length;
139         var newLength = copy.Length;
140         while (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
141         {
142             var maxDoubletSource = _maxDoublet.Source;
143             var maxDoubletTarget = _maxDoublet.Target;
144             if (_equalityComparer.Equals(_maxDoubletData.Link, _constants.Null))
145             {
146                 _maxDoubletData.Link = _links.GetOrCreate(maxDoubletSource,
147                     ↪ maxDoubletTarget);
148             }
149             var maxDoubletReplacementLink = _maxDoubletData.Link;
150             oldLength--;
151             var oldLengthMinusTwo = oldLength - 1;
152             // Substitute all usages
153             int w = 0, r = 0; // (r == read, w == write)
154             for (; r < oldLength; r++)
155             {
156                 if (_equalityComparer.Equals(copy[r].Element, maxDoubletSource) &&
157                     ↪ _equalityComparer.Equals(copy[r + 1].Element, maxDoubletTarget))
158                 {
159                     if (r > 0)
160                     {
161                         var previous = copy[w - 1].Element;
162                         copy[w - 1].DoubletData.DecrementFrequency();

```

```

161         copy[w - 1].DoubletData =
            ↳ _doubletFrequenciesCache.IncrementFrequency(previous,
            ↳ maxDoubletReplacementLink);
162     }
163     if (r < oldLengthMinusTwo)
164     {
165         var next = copy[r + 2].Element;
166         copy[r + 1].DoubletData.DecrementFrequency();
167         copy[w].DoubletData = _doubletFrequenciesCache.IncrementFrequency(maxDoubletReplacementLink,
            ↳ next);
168     }
169     copy[w++].Element = maxDoubletReplacementLink;
170     r++;
171     newLength--;
172 }
173 else
174 {
175     copy[w++] = copy[r];
176 }
177 }
178 if (w < newLength)
179 {
180     copy[w] = copy[r];
181 }
182 oldLength = newLength;
183 ResetMaxDoublet();
184 UpdateMaxDoublet(copy, newLength);
185 }
186 return newLength;
187 }
188
189 [MethodImpl(MethodImplOptions.AggressiveInlining)]
190 private void ResetMaxDoublet()
191 {
192     _maxDoublet = new Doublet<TLink>();
193     _maxDoubletData = new LinkFrequency<TLink>();
194 }
195
196 [MethodImpl(MethodImplOptions.AggressiveInlining)]
197 private void UpdateMaxDoublet(HalfDoublet[] copy, int length)
198 {
199     Doublet<TLink> doublet = default;
200     for (var i = 1; i < length; i++)
201     {
202         doublet = new Doublet<TLink>(copy[i - 1].Element, copy[i].Element);
203         UpdateMaxDoublet(ref doublet, copy[i - 1].DoubletData);
204     }
205 }
206
207 [MethodImpl(MethodImplOptions.AggressiveInlining)]
208 private void UpdateMaxDoublet(ref Doublet<TLink> doublet, LinkFrequency<TLink> data)
209 {
210     var frequency = data.Frequency;
211     var maxFrequency = _maxDoubletData.Frequency;
212     //if (frequency > _minFrequencyToCompress && (maxFrequency < frequency ||
213     ↳ (maxFrequency == frequency && doublet.Source + doublet.Target < /* gives better
214     ↳ compression string data (and gives collisions quickly) */ _maxDoublet.Source +
215     ↳ _maxDoublet.Target)))
216     if (_comparer.Compare(frequency, _minFrequencyToCompress) > 0 &&
217         (_comparer.Compare(maxFrequency, frequency) < 0 ||
218         ↳ (_equalityComparer.Equals(maxFrequency, frequency) &&
219         ↳ _comparer.Compare(Arithmetic.Add(doublet.Source, doublet.Target),
220         ↳ Arithmetic.Add(_maxDoublet.Source, _maxDoublet.Target)) > 0))) /* gives
221         ↳ better stability and better compression on sequent data and even on random
222         ↳ numbers data (but gives collisions anyway) */
223     {
224         _maxDoublet = doublet;
225         _maxDoubletData = data;
226     }
227 }
228 }
229 }
230 }
231 }

```

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     public abstract class LinksListToSequenceConverterBase<TLink> : LinksOperatorBase<TLink>,
10     ↪ IConverter<IList<TLink>, TLink>
11     {
12         [MethodImpl(MethodImplOptions.AggressiveInlining)]
13         protected LinksListToSequenceConverterBase(ILinks<TLink> links) : base(links) { }
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         public abstract TLink Convert(IList<TLink> source);
17     }
18 }

```

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     public class OptimalVariantConverter<TLink> : LinksListToSequenceConverterBase<TLink>
13     {
14         private static readonly EqualityComparer<TLink> _equalityComparer =
15         ↪ EqualityComparer<TLink>.Default;
16         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
17
18         private readonly IConverter<IList<TLink>> _sequenceToItsLocalElementLevelsConverter;
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         public OptimalVariantConverter(ILinks<TLink> links, IConverter<IList<TLink>>
22         ↪ sequenceToItsLocalElementLevelsConverter) : base(links)
23         => _sequenceToItsLocalElementLevelsConverter =
24         ↪ sequenceToItsLocalElementLevelsConverter;
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public OptimalVariantConverter(ILinks<TLink> links, LinkFrequenciesCache<TLink>
28         ↪ linkFrequenciesCache)
29         : this(links, new SequenceToItsLocalElementLevelsConverter<TLink>(links, new Frequen_
30         ↪ ciesCacheBasedLinkToItsFrequencyNumberConverter<TLink>(linkFrequenciesCache))) { }
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         public OptimalVariantConverter(ILinks<TLink> links)
34         : this(links, new LinkFrequenciesCache<TLink>(links, new
35         ↪ TotalSequenceSymbolFrequencyCounter<TLink>(links))) { }
36
37         [MethodImpl(MethodImplOptions.AggressiveInlining)]
38         public override TLink Convert(IList<TLink> sequence)
39         {
40             var length = sequence.Count;
41             if (length == 1)
42             {
43                 return sequence[0];
44             }
45             if (length == 2)
46             {
47                 return _links.GetOrCreate(sequence[0], sequence[1]);
48             }
49             sequence = sequence.ToArray();
50             var levels = _sequenceToItsLocalElementLevelsConverter.Convert(sequence);
51             while (length > 2)
52             {
53                 var levelRepeat = 1;
54                 var currentLevel = levels[0];
55                 var previousLevel = levels[0];
56                 var skipOnce = false;
57                 var w = 0;
58                 for (var i = 1; i < length; i++)
59                 {
60                     if (_equalityComparer.Equals(currentLevel, levels[i]))
61                     {
62                         levelRepeat++;
63                     }
64                 }
65             }
66         }
67     }
68 }

```

```

57         skipOnce = false;
58         if (levelRepeat == 2)
59         {
60             sequence[w] = _links.GetOrCreate(sequence[i - 1], sequence[i]);
61             var newLevel = i >= length - 1 ?
62                 GetPreviousLowerThanCurrentOrCurrent(previousLevel,
63                     ↪ currentLevel) :
64                     i < 2 ?
65                         GetNextLowerThanCurrentOrCurrent(currentLevel, levels[i + 1]) :
66                         GetGreatestNeighbourLowerThanCurrentOrCurrent(previousLevel,
67                             ↪ currentLevel, levels[i + 1]);
68             levels[w] = newLevel;
69             previousLevel = currentLevel;
70             w++;
71             levelRepeat = 0;
72             skipOnce = true;
73         }
74         else if (i == length - 1)
75         {
76             sequence[w] = sequence[i];
77             levels[w] = levels[i];
78             w++;
79         }
80     }
81     else
82     {
83         currentLevel = levels[i];
84         levelRepeat = 1;
85         if (skipOnce)
86         {
87             skipOnce = false;
88         }
89         else
90         {
91             sequence[w] = sequence[i - 1];
92             levels[w] = levels[i - 1];
93             previousLevel = levels[w];
94             w++;
95         }
96         if (i == length - 1)
97         {
98             sequence[w] = sequence[i];
99             levels[w] = levels[i];
100             w++;
101         }
102     }
103     length = w;
104     return _links.GetOrCreate(sequence[0], sequence[1]);
105 }
106
107 [MethodImpl(MethodImplOptions.AggressiveInlining)]
108 private static TLink GetGreatestNeighbourLowerThanCurrentOrCurrent(TLink previous, TLink
109     ↪ current, TLink next)
110 {
111     return _comparer.Compare(previous, next) > 0
112         ? _comparer.Compare(previous, current) < 0 ? previous : current
113         : _comparer.Compare(next, current) < 0 ? next : current;
114 }
115
116 [MethodImpl(MethodImplOptions.AggressiveInlining)]
117 private static TLink GetNextLowerThanCurrentOrCurrent(TLink current, TLink next) =>
118     ↪ _comparer.Compare(next, current) < 0 ? next : current;
119
120 [MethodImpl(MethodImplOptions.AggressiveInlining)]
121 private static TLink GetPreviousLowerThanCurrentOrCurrent(TLink previous, TLink current)
122     ↪ => _comparer.Compare(previous, current) < 0 ? previous : current;

```

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     public class SequenceToItsLocalElementLevelsConverter<TLink> : LinksOperatorBase<TLink>,
        ↳ IConverter<IList<TLink>>
10     {
11         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
12
13         private readonly IConverter<Doublet<TLink>, TLink> _linkToItsFrequencyToNumberConveter;
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         public SequenceToItsLocalElementLevelsConverter(ILinks<TLink> links,
            ↳ IConverter<Doublet<TLink>, TLink> linkToItsFrequencyToNumberConveter) : base(links)
            ↳ => _linkToItsFrequencyToNumberConveter = linkToItsFrequencyToNumberConveter;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         public IList<TLink> Convert(IList<TLink> sequence)
20         {
21             var levels = new TLink[sequence.Count];
22             levels[0] = GetFrequencyNumber(sequence[0], sequence[1]);
23             for (var i = 1; i < sequence.Count - 1; i++)
24             {
25                 var previous = GetFrequencyNumber(sequence[i - 1], sequence[i]);
26                 var next = GetFrequencyNumber(sequence[i], sequence[i + 1]);
27                 levels[i] = _comparer.Compare(previous, next) > 0 ? previous : next;
28             }
29             levels[levels.Length - 1] = GetFrequencyNumber(sequence[sequence.Count - 2],
                ↳ sequence[sequence.Count - 1]);
30             return levels;
31         }
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         public TLink GetFrequencyNumber(TLink source, TLink target) =>
            ↳ _linkToItsFrequencyToNumberConveter.Convert(new Doublet<TLink>(source, target));
35     }
36 }

```

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     public class DefaultSequenceElementCriterionMatcher<TLink> : LinksOperatorBase<TLink>,
        ↳ ICriterionMatcher<TLink>
9     {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public DefaultSequenceElementCriterionMatcher(ILinks<TLink> links) : base(links) { }
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public bool IsMatched(TLink argument) => _links.IsPartialPoint(argument);
15     }
16 }

```

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     public class MarkedSequenceCriterionMatcher<TLink> : ICriterionMatcher<TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
            ↳ EqualityComparer<TLink>.Default;
12
13         private readonly ILinks<TLink> _links;
14         private readonly TLink _sequenceMarkerLink;
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public MarkedSequenceCriterionMatcher(ILinks<TLink> links, TLink sequenceMarkerLink)
18         {
19             _links = links;
20             _sequenceMarkerLink = sequenceMarkerLink;
21         }
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         public bool IsMatched(TLink sequenceCandidate)

```

```

25         => _equalityComparer.Equals(_links.GetSource(sequenceCandidate), _sequenceMarkerLink)
26         || !_equalityComparer.Equals(_links.SearchOrDefault(_sequenceMarkerLink,
27             ↪ sequenceCandidate), _links.Constants.Null);
28     }

```

1.8 ./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     public class DefaultSequenceAppender<TLink> : LinksOperatorBase<TLink>,
12         ↪ ISequenceAppender<TLink>
13     {
14         private static readonly EqualityComparer<TLink> _equalityComparer =
15             ↪ EqualityComparer<TLink>.Default;
16
17         private readonly IStack<TLink> _stack;
18         private readonly ISequenceHeightProvider<TLink> _heightProvider;
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         public DefaultSequenceAppender(ILinks<TLink> links, IStack<TLink> stack,
22             ↪ ISequenceHeightProvider<TLink> heightProvider)
23             : base(links)
24         {
25             _stack = stack;
26             _heightProvider = heightProvider;
27         }
28
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         public TLink Append(TLink sequence, TLink appendant)
31         {
32             var cursor = sequence;
33             var links = _links;
34             while (!_equalityComparer.Equals(_heightProvider.Get(cursor), default))
35             {
36                 var source = links.GetSource(cursor);
37                 var target = links.GetTarget(cursor);
38                 if (_equalityComparer.Equals(_heightProvider.Get(source),
39                     ↪ _heightProvider.Get(target)))
40                 {
41                     break;
42                 }
43                 else
44                 {
45                     _stack.Push(source);
46                     cursor = target;
47                 }
48             }
49             var left = cursor;
50             var right = appendant;
51             while (!_equalityComparer.Equals(cursor = _stack.PopOrDefault(),
52                 ↪ links.Constants.Null))
53             {
54                 right = links.GetOrCreate(left, right);
55                 left = cursor;
56             }
57             return links.GetOrCreate(left, right);
58         }
59     }
60 }

```

1.9 ./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     public class DuplicateSegmentsCounter<TLink> : ICounter<int>
11     {

```



```

12     private readonly IProvider<IList<KeyValuePair<IList<TLink>, IList<TLink>>>>
13         ↪ _duplicateFragmentsProvider;
14
15     [MethodImpl(MethodImplOptions.AggressiveInlining)]
16     public DuplicateSegmentsCounter(IProvider<IList<KeyValuePair<IList<TLink>,
17         ↪ IList<TLink>>>> duplicateFragmentsProvider) => _duplicateFragmentsProvider =
18         ↪ duplicateFragmentsProvider;
19
20     [MethodImpl(MethodImplOptions.AggressiveInlining)]
21     public int Count() => _duplicateFragmentsProvider.Get().Sum(x => x.Value.Count);
22 }
23 }

```

1.10 ./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     public class DuplicateSegmentsProvider<TLink> :
19         ↪ DictionaryBasedDuplicateSegmentsWalkerBase<TLink>,
20         ↪ IProvider<IList<KeyValuePair<IList<TLink>, IList<TLink>>>>
21     {
22         private static readonly UncheckedConverter<TLink, long> _addressToInt64Converter =
23             ↪ UncheckedConverter<TLink, long>.Default;
24         private static readonly UncheckedConverter<TLink, ulong> _addressToUInt64Converter =
25             ↪ UncheckedConverter<TLink, ulong>.Default;
26         private static readonly UncheckedConverter<ulong, TLink> _uInt64ToAddressConverter =
27             ↪ UncheckedConverter<ulong, TLink>.Default;
28
29         private readonly ILinks<TLink> _links;
30         private readonly ILinks<TLink> _sequences;
31         private HashSet<KeyValuePair<IList<TLink>, IList<TLink>>> _groups;
32         private BitString _visited;
33
34         private class ItemEquilityComparer : IEqualityComparer<KeyValuePair<IList<TLink>,
35             ↪ IList<TLink>>>
36         {
37             private readonly IListEqualityComparer<TLink> _listComparer;
38
39             public ItemEquilityComparer() => _listComparer =
40                 ↪ Default<IListEqualityComparer<TLink>>.Instance;
41
42             [MethodImpl(MethodImplOptions.AggressiveInlining)]
43             public bool Equals(KeyValuePair<IList<TLink>, IList<TLink>> left,
44                 ↪ KeyValuePair<IList<TLink>, IList<TLink>> right) =>
45                 ↪ _listComparer.Equals(left.Key, right.Key) && _listComparer.Equals(left.Value,
46                 ↪ right.Value);
47
48             [MethodImpl(MethodImplOptions.AggressiveInlining)]
49             public int GetHashCode(KeyValuePair<IList<TLink>, IList<TLink>> pair) =>
50                 ↪ (_listComparer.GetHashCode(pair.Key),
51                 ↪ _listComparer.GetHashCode(pair.Value)).GetHashCode();
52         }
53
54         private class ItemComparer : IComparer<KeyValuePair<IList<TLink>, IList<TLink>>>
55         {
56             private readonly IListComparer<TLink> _listComparer;
57
58             [MethodImpl(MethodImplOptions.AggressiveInlining)]
59             public ItemComparer() => _listComparer = Default<IListComparer<TLink>>.Instance;
60
61             [MethodImpl(MethodImplOptions.AggressiveInlining)]
62             public int Compare(KeyValuePair<IList<TLink>, IList<TLink>> left,
63                 ↪ KeyValuePair<IList<TLink>, IList<TLink>> right)
64             {
65                 var intermediateResult = _listComparer.Compare(left.Key, right.Key);
66                 if (intermediateResult == 0)

```

```

54         {
55             intermediateResult = _listComparer.Compare(left.Value, right.Value);
56         }
57         return intermediateResult;
58     }
59 }
60
61 [MethodImpl(MethodImplOptions.AggressiveInlining)]
62 public DuplicateSegmentsProvider(ILinks<TLink> links, ILinks<TLink> sequences)
63     : base(minimumStringSegmentLength: 2)
64 {
65     _links = links;
66     _sequences = sequences;
67 }
68
69 [MethodImpl(MethodImplOptions.AggressiveInlining)]
70 public IList<KeyValuePair<IList<TLink>, IList<TLink>>> Get()
71 {
72     _groups = new HashSet<KeyValuePair<IList<TLink>,
73         ↳ IList<TLink>>>(Default<ItemEqualityComparer>.Instance);
74     var links = _links;
75     var count = links.Count();
76     _visited = new BitString(_addressToInt64Converter.Convert(count) + 1L);
77     links.Each(link =>
78     {
79         var linkIndex = links.GetIndex(link);
80         var linkBitIndex = _addressToInt64Converter.Convert(linkIndex);
81         var constants = links.Constants;
82         if (!_visited.Get(linkBitIndex))
83         {
84             var sequenceElements = new List<TLink>();
85             var filler = new ListFiller<TLink, TLink>(sequenceElements, constants.Break);
86             _sequences.Each(filler.AddSkipFirstAndReturnConstant, new
87                 ↳ LinkAddress<TLink>(linkIndex));
88             if (sequenceElements.Count > 2)
89             {
90                 WalkAll(sequenceElements);
91             }
92             return constants.Continue;
93         });
94     var resultList = _groups.ToList();
95     var comparer = Default<ItemComparer>.Instance;
96     resultList.Sort(comparer);
97     #if DEBUG
98     foreach (var item in resultList)
99     {
100         PrintDuplicates(item);
101     }
102     #endif
103     return resultList;
104 }
105
106 [MethodImpl(MethodImplOptions.AggressiveInlining)]
107 protected override Segment<TLink> CreateSegment(IList<TLink> elements, int offset, int
108     ↳ length) => new Segment<TLink>(elements, offset, length);
109
110 [MethodImpl(MethodImplOptions.AggressiveInlining)]
111 protected override void OnDuplicateFound(Segment<TLink> segment)
112 {
113     var duplicates = CollectDuplicatesForSegment(segment);
114     if (duplicates.Count > 1)
115     {
116         _groups.Add(new KeyValuePair<IList<TLink>, IList<TLink>>(segment.ToArray(),
117             ↳ duplicates));
118     }
119 }
120
121 [MethodImpl(MethodImplOptions.AggressiveInlining)]
122 private List<TLink> CollectDuplicatesForSegment(Segment<TLink> segment)
123 {
124     var duplicates = new List<TLink>();
125     var readAsElement = new HashSet<TLink>();
126     var restrictions = segment.ShiftRight();
127     var constants = _links.Constants;
128     restrictions[0] = constants.Any;
129     _sequences.Each(sequence =>
130     {
131         var sequenceIndex = sequence[constants.IndexPart];

```

```

129         duplicates.Add(sequenceIndex);
130         readAsElement.Add(sequenceIndex);
131         return constants.Continue;
132     }, restrictions);
133     if (duplicates.Any(x => _visited.Get(_addressToInt64Converter.Convert(x))))
134     {
135         return new List<TLink>();
136     }
137     foreach (var duplicate in duplicates)
138     {
139         var duplicateBitIndex = _addressToInt64Converter.Convert(duplicate);
140         _visited.Set(duplicateBitIndex);
141     }
142     if (_sequences is Sequences sequencesExperiments)
143     {
144         var partiallyMatched = sequencesExperiments.GetAllPartiallyMatchingSequences4((H_
            ↪ ashSet<ulong>)(object)readAsElement,
            ↪ (IList<ulong>)segment);
145         foreach (var partiallyMatchedSequence in partiallyMatched)
146         {
147             var sequenceIndex =
            ↪ _uInt64ToAddressConverter.Convert(partiallyMatchedSequence);
148             duplicates.Add(sequenceIndex);
149         }
150     }
151     duplicates.Sort();
152     return duplicates;
153 }

154 [MethodImpl(MethodImplOptions.AggressiveInlining)]
155 private void PrintDuplicates(KeyValuePair<IList<TLink>, IList<TLink>> duplicatesItem)
156 {
157     if (!(_links is ILinks<ulong> ulongLinks))
158     {
159         return;
160     }
161     var duplicatesKey = duplicatesItem.Key;
162     var keyString = UnicodeMap.FromLinksToString((IList<ulong>)duplicatesKey);
163     Console.WriteLine($"> {keyString} ({string.Join(", ", duplicatesKey)}");
164     var duplicatesList = duplicatesItem.Value;
165     for (int i = 0; i < duplicatesList.Count; i++)
166     {
167         var sequenceIndex = _addressToUInt64Converter.Convert(duplicatesList[i]);
168         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));
169         Console.WriteLine(formattedSequenceStructure);
170         var sequenceString = UnicodeMap.FromSequenceLinkToString(sequenceIndex,
            ↪ ulongLinks);
171         Console.WriteLine(sequenceString);
172     }
173     Console.WriteLine();
174 }
175 }
176 }
177 }

```

1.11 ./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
13     ↪ between them).
14     /// TODO: Extract interface to implement frequencies storage inside Links storage
15     /// </remarks>
16     public class LinkFrequenciesCache<TLink> : LinksOperatorBase<TLink>
17     {
18         private static readonly EqualityComparer<TLink> _equalityComparer =
19         ↪ EqualityComparer<TLink>.Default;
20         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
21     }
22 }

```

```

20 private static readonly TLink _zero = default;
21 private static readonly TLink _one = Arithmetic.Increment(_zero);
22
23 private readonly Dictionary<Doublet<TLink>, LinkFrequency<TLink>> _doubletsCache;
24 private readonly ICounter<TLink, TLink> _frequencyCounter;
25
26 [MethodImpl(MethodImplOptions.AggressiveInlining)]
27 public LinkFrequenciesCache(ILinks<TLink> links, ICounter<TLink, TLink> frequencyCounter)
28     : base(links)
29 {
30     _doubletsCache = new Dictionary<Doublet<TLink>, LinkFrequency<TLink>>(4096,
31         ↪ DoubletComparer<TLink>.Default);
32     _frequencyCounter = frequencyCounter;
33 }
34
35 [MethodImpl(MethodImplOptions.AggressiveInlining)]
36 public LinkFrequency<TLink> GetFrequency(TLink source, TLink target)
37 {
38     var doublet = new Doublet<TLink>(source, target);
39     return GetFrequency(ref doublet);
40 }
41
42 [MethodImpl(MethodImplOptions.AggressiveInlining)]
43 public LinkFrequency<TLink> GetFrequency(ref Doublet<TLink> doublet)
44 {
45     _doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data);
46     return data;
47 }
48
49 [MethodImpl(MethodImplOptions.AggressiveInlining)]
50 public void IncrementFrequencies(IList<TLink> sequence)
51 {
52     for (var i = 1; i < sequence.Count; i++)
53     {
54         IncrementFrequency(sequence[i - 1], sequence[i]);
55     }
56 }
57
58 [MethodImpl(MethodImplOptions.AggressiveInlining)]
59 public LinkFrequency<TLink> IncrementFrequency(TLink source, TLink target)
60 {
61     var doublet = new Doublet<TLink>(source, target);
62     return IncrementFrequency(ref doublet);
63 }
64
65 [MethodImpl(MethodImplOptions.AggressiveInlining)]
66 public void PrintFrequencies(IList<TLink> sequence)
67 {
68     for (var i = 1; i < sequence.Count; i++)
69     {
70         PrintFrequency(sequence[i - 1], sequence[i]);
71     }
72 }
73
74 [MethodImpl(MethodImplOptions.AggressiveInlining)]
75 public void PrintFrequency(TLink source, TLink target)
76 {
77     var number = GetFrequency(source, target).Frequency;
78     Console.WriteLine("{0},{1}) - {2}", source, target, number);
79 }
80
81 [MethodImpl(MethodImplOptions.AggressiveInlining)]
82 public LinkFrequency<TLink> IncrementFrequency(ref Doublet<TLink> doublet)
83 {
84     if (_doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data))
85     {
86         data.IncrementFrequency();
87     }
88     else
89     {
90         var link = _links.SearchOrDefault(doublet.Source, doublet.Target);
91         data = new LinkFrequency<TLink>(_one, link);
92         if (!_equalityComparer.Equals(link, default))
93         {
94             data.Frequency = Arithmetic.Add(data.Frequency,
95                 ↪ _frequencyCounter.Count(link));
96         }
97         _doubletsCache.Add(doublet, data);
98     }
99 }

```

```

97         return data;
98     }
99
100     [MethodImpl(MethodImplOptions.AggressiveInlining)]
101     public void ValidateFrequencies()
102     {
103         foreach (var entry in _doubletsCache)
104         {
105             var value = entry.Value;
106             var linkIndex = value.Link;
107             if (!_equalityComparer.Equals(linkIndex, default))
108             {
109                 var frequency = value.Frequency;
110                 var count = _frequencyCounter.Count(linkIndex);
111                 // TODO: Why `frequency` always greater than `count` by 1?
112                 if (((_comparer.Compare(frequency, count) > 0) &&
113                     ↪ (_comparer.Compare(Arithmetic.Subtract(frequency, count), _one) > 0))
114                     || ((_comparer.Compare(count, frequency) > 0) &&
115                     ↪ (_comparer.Compare(Arithmetic.Subtract(count, frequency), _one) > 0)))
116                 {
117                     throw new InvalidOperationException("Frequencies validation failed.");
118                 }
119                 //else
120                 //{
121                 //    if (value.Frequency > 0)
122                 //    {
123                 //        var frequency = value.Frequency;
124                 //        linkIndex = _createLink(entry.Key.Source, entry.Key.Target);
125                 //        var count = _countLinkFrequency(linkIndex);
126                 //        if ((frequency > count && frequency - count > 1) || (count > frequency
127                 ↪ && count - frequency > 1))
128                 //            throw new InvalidOperationException("Frequencies validation
129                 ↪ failed.");
130                 //    }
131                 //}
132             }
133         }
134     }
135 }

```

1.12 ./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      public class LinkFrequency<TLink>
9      {
10         public TLink Frequency { get; set; }
11         public TLink Link { get; set; }
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public LinkFrequency(TLink frequency, TLink link)
15         {
16             Frequency = frequency;
17             Link = link;
18         }
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         public LinkFrequency() { }
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         public void IncrementFrequency() => Frequency = Arithmetic<TLink>.Increment(Frequency);
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public void DecrementFrequency() => Frequency = Arithmetic<TLink>.Decrement(Frequency);
28
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]
30         public override string ToString() => $"F: {Frequency}, L: {Link}";
31     }
32 }

```

1.13 ./csharp/Platform.Data.Doublets.Sequences/Frequencies/Cache/LinkTotalsFrequencyValueConverter.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     public class FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<TLink> :
9         ↳ IConverter<Doublet<TLink>, TLink>
10    {
11        private readonly LinkFrequenciesCache<TLink> _cache;
12
13        [MethodImpl(MethodImplOptions.AggressiveInlining)]
14        public
15            ↳ FrequenciesCacheBasedLinkToItsFrequencyNumberConverter(LinkFrequenciesCache<TLink>
16                ↳ cache) => _cache = cache;
17
18        [MethodImpl(MethodImplOptions.AggressiveInlining)]
19        public TLink Convert(Doublet<TLink> source) => _cache.GetFrequency(ref source).Frequency;
20    }
21 }

```

1.14 ./csharp/Platform.Data.Doublets.Sequences/Frequencies/Counters/MarkedSequenceSymbolFrequencyOneOffCounter

```

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     public class MarkedSequenceSymbolFrequencyOneOffCounter<TLink> :
9         ↳ SequenceSymbolFrequencyOneOffCounter<TLink>
10    {
11        private readonly ICriterionMatcher<TLink> _markedSequenceMatcher;
12
13        [MethodImpl(MethodImplOptions.AggressiveInlining)]
14        public MarkedSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links,
15            ↳ ICriterionMatcher<TLink> markedSequenceMatcher, TLink sequenceLink, TLink symbol)
16            : base(links, sequenceLink, symbol)
17            => _markedSequenceMatcher = markedSequenceMatcher;
18
19        [MethodImpl(MethodImplOptions.AggressiveInlining)]
20        public override TLink Count()
21        {
22            if (!_markedSequenceMatcher.IsMatched(_sequenceLink))
23            {
24                return default;
25            }
26            return base.Count();
27        }
28    }
29 }

```

1.15 ./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     public class SequenceSymbolFrequencyOneOffCounter<TLink> : ICounter<TLink>
12     {
13         private static readonly EqualityComparer<TLink> _equalityComparer =
14             ↳ EqualityComparer<TLink>.Default;
15         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
16
17         protected readonly ILinks<TLink> _links;
18         protected readonly TLink _sequenceLink;
19         protected readonly TLink _symbol;
20         protected TLink _total;
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         public SequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links, TLink sequenceLink,
24             ↳ TLink symbol)
25         {
26             _links = links;
27             _sequenceLink = sequenceLink;
28             _symbol = symbol;
29         }
30     }
31 }

```

```

27     _total = default;
28 }
29
30 [MethodImpl(MethodImplOptions.AggressiveInlining)]
31 public virtual TLink Count()
32 {
33     if (_comparer.Compare(_total, default) > 0)
34     {
35         return _total;
36     }
37     StopableSequenceWalker.WalkRight(_sequenceLink, _links.GetSource, _links.GetTarget,
38         ↪ IsElement, VisitElement);
39     return _total;
40 }
41
42 [MethodImpl(MethodImplOptions.AggressiveInlining)]
43 private bool IsElement(TLink x) => _equalityComparer.Equals(x, _symbol) ||
44     ↪ _links.IsPartialPoint(x); // TODO: Use SequenceElementCriteriaMatcher instead of
45     ↪ IsPartialPoint
46
47 [MethodImpl(MethodImplOptions.AggressiveInlining)]
48 private bool VisitElement(TLink element)
49 {
50     if (_equalityComparer.Equals(element, _symbol))
51     {
52         _total = Arithmetic.Increment(_total);
53     }
54     return true;
55 }
56 }
57 }

```

1.16 ./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     public class TotalMarkedSequenceSymbolFrequencyCounter<TLink> : ICounter<TLink, TLink>
9     {
10         private readonly ILinks<TLink> _links;
11         private readonly ICriterionMatcher<TLink> _markedSequenceMatcher;
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public TotalMarkedSequenceSymbolFrequencyCounter(ILinks<TLink> links,
15             ↪ ICriterionMatcher<TLink> markedSequenceMatcher)
16         {
17             _links = links;
18             _markedSequenceMatcher = markedSequenceMatcher;
19         }
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public TLink Count(TLink argument) => new
23             ↪ TotalMarkedSequenceSymbolFrequencyOneOffCounter<TLink>(_links,
24             ↪ _markedSequenceMatcher, argument).Count();
25     }
26 }

```

1.17 ./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     public class TotalMarkedSequenceSymbolFrequencyOneOffCounter<TLink> :
10         ↪ TotalSequenceSymbolFrequencyOneOffCounter<TLink>
11     {
12         private readonly ICriterionMatcher<TLink> _markedSequenceMatcher;
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         public TotalMarkedSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links,
16             ↪ ICriterionMatcher<TLink> markedSequenceMatcher, TLink symbol)
17             : base(links, symbol)
18             => _markedSequenceMatcher = markedSequenceMatcher;
19     }
20 }

```

```

17     [MethodImpl(MethodImplOptions.AggressiveInlining)]
18     protected override void CountSequenceSymbolFrequency(TLink link)
19     {
20     {
21         var symbolFrequencyCounter = new
22             ↪ MarkedSequenceSymbolFrequencyOneOffCounter<TLink>(_links,
23             ↪ _markedSequenceMatcher, link, _symbol);
24         _total = Arithmetic.Add(_total, symbolFrequencyCounter.Count());
25     }
26 }

```

1.18 ./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     public class TotalSequenceSymbolFrequencyCounter<TLink> : ICounter<TLink, TLink>
9     {
10         private readonly ILinks<TLink> _links;
11
12         [MethodImpl(MethodImplOptions.AggressiveInlining)]
13         public TotalSequenceSymbolFrequencyCounter(ILinks<TLink> links) => _links = links;
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         public TLink Count(TLink symbol) => new
17             ↪ TotalSequenceSymbolFrequencyOneOffCounter<TLink>(_links, symbol).Count();
18     }
19 }

```

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

```

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     public class TotalSequenceSymbolFrequencyOneOffCounter<TLink> : ICounter<TLink>
11     {
12         private static readonly EqualityComparer<TLink> _equalityComparer =
13             ↪ EqualityComparer<TLink>.Default;
14         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
15
16         protected readonly ILinks<TLink> _links;
17         protected readonly TLink _symbol;
18         protected readonly HashSet<TLink> _visits;
19         protected TLink _total;
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public TotalSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links, TLink symbol)
23         {
24             _links = links;
25             _symbol = symbol;
26             _visits = new HashSet<TLink>();
27             _total = default;
28         }
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         public TLink Count()
32         {
33             if (_comparer.Compare(_total, default) > 0 || _visits.Count > 0)
34             {
35                 return _total;
36             }
37             CountCore(_symbol);
38             return _total;
39         }
40
41         [MethodImpl(MethodImplOptions.AggressiveInlining)]
42         private void CountCore(TLink link)
43         {
44             var any = _links.Constants.Any;
45             if (_equalityComparer.Equals(_links.Count(any, link), default))

```



```

46         CountSequenceSymbolFrequency(link);
47     }
48     else
49     {
50         _links.Each(EachElementHandler, any, link);
51     }
52 }
53
54 [MethodImpl(MethodImplOptions.AggressiveInlining)]
55 protected virtual void CountSequenceSymbolFrequency(TLink link)
56 {
57     var symbolFrequencyCounter = new SequenceSymbolFrequencyOneOffCounter<TLink>(_links,
58     ↪ link, _symbol);
59     _total = Arithmetic.Add(_total, symbolFrequencyCounter.Count());
60 }
61
62 [MethodImpl(MethodImplOptions.AggressiveInlining)]
63 private TLink EachElementHandler(IList<TLink> doublet)
64 {
65     var constants = _links.Constants;
66     var doubletIndex = doublet[constants.IndexPart];
67     if (_visits.Add(doubletIndex))
68     {
69         CountCore(doubletIndex);
70     }
71     return constants.Continue;
72 }
73 }

```

1.20 ./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     public class CachedSequenceHeightProvider<TLink> : ISequenceHeightProvider<TLink>
11     {
12         private static readonly EqualityComparer<TLink> _equalityComparer =
13         ↪ EqualityComparer<TLink>.Default;
14
15         private readonly TLink _heightPropertyMarker;
16         private readonly ISequenceHeightProvider<TLink> _baseHeightProvider;
17         private readonly IConverter<TLink> _addressToUnaryNumberConverter;
18         private readonly IConverter<TLink> _unaryNumberToAddressConverter;
19         private readonly IProperties<TLink, TLink, TLink> _propertyOperator;
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public CachedSequenceHeightProvider(
23             ISequenceHeightProvider<TLink> baseHeightProvider,
24             IConverter<TLink> addressToUnaryNumberConverter,
25             IConverter<TLink> unaryNumberToAddressConverter,
26             TLink heightPropertyMarker,
27             IProperties<TLink, TLink, TLink> propertyOperator)
28         {
29             _heightPropertyMarker = heightPropertyMarker;
30             _baseHeightProvider = baseHeightProvider;
31             _addressToUnaryNumberConverter = addressToUnaryNumberConverter;
32             _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
33             _propertyOperator = propertyOperator;
34         }
35
36         [MethodImpl(MethodImplOptions.AggressiveInlining)]
37         public TLink Get(TLink sequence)
38         {
39             TLink height;
40             var heightValue = _propertyOperator.GetValue(sequence, _heightPropertyMarker);
41             if (_equalityComparer.Equals(heightValue, default))
42             {
43                 height = _baseHeightProvider.Get(sequence);
44                 heightValue = _addressToUnaryNumberConverter.Convert(height);
45                 _propertyOperator.SetValue(sequence, _heightPropertyMarker, heightValue);
46             }
47             else
48             {
49                 height = _unaryNumberToAddressConverter.Convert(heightValue);

```

```

49     }
50     return height;
51 }
52 }
53 }

```

1.21 ./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     public class DefaultSequenceRightHeightProvider<TLink> : LinksOperatorBase<TLink>,
10         ↳ ISequenceHeightProvider<TLink>
11     {
12         private readonly ICriterionMatcher<TLink> _elementMatcher;
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         public DefaultSequenceRightHeightProvider(ILinks<TLink> links, ICriterionMatcher<TLink>
16             ↳ elementMatcher) : base(links) => _elementMatcher = elementMatcher;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         public TLink Get(TLink sequence)
20         {
21             var height = default(TLink);
22             var pairOrElement = sequence;
23             while (!_elementMatcher.IsMatched(pairOrElement))
24             {
25                 pairOrElement = _links.GetTarget(pairOrElement);
26                 height = Arithmetic.Increment(height);
27             }
28             return height;
29         }
30     }
31 }

```

1.22 ./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     public interface ISequenceHeightProvider<TLink> : IProvider<TLink, TLink>
8     {
9     }
10 }

```

1.23 ./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     public class FrequencyIncrementer<TLink> : LinksOperatorBase<TLink>, IIncrementer<TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
12             ↳ EqualityComparer<TLink>.Default;
13
14         private readonly TLink _frequencyMarker;
15         private readonly TLink _unaryOne;
16         private readonly IIncrementer<TLink> _unaryNumberIncrementer;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         public FrequencyIncrementer(ILinks<TLink> links, TLink frequencyMarker, TLink unaryOne,
20             ↳ IIncrementer<TLink> unaryNumberIncrementer)
21             : base(links)
22         {
23             _frequencyMarker = frequencyMarker;
24             _unaryOne = unaryOne;
25             _unaryNumberIncrementer = unaryNumberIncrementer;
26         }
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

27     public TLink Increment(TLink frequency)
28     {
29         var links = _links;
30         if (_equalityComparer.Equals(frequency, default))
31         {
32             return links.GetOrCreate(_unaryOne, _frequencyMarker);
33         }
34         var incrementedSource =
35             ↪ _unaryNumberIncrementer.Increment(links.GetSource(frequency));
36         return links.GetOrCreate(incrementedSource, _frequencyMarker);
37     }
38 }

```

1.24 ./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      public class UnaryNumberIncrementer<TLink> : LinksOperatorBase<TLink>, IIncrementer<TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
12             ↪ EqualityComparer<TLink>.Default;
13
14         private readonly TLink _unaryOne;
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public UnaryNumberIncrementer(ILinks<TLink> links, TLink unaryOne) : base(links) =>
18             ↪ _unaryOne = unaryOne;
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         public TLink Increment(TLink unaryNumber)
22         {
23             var links = _links;
24             if (_equalityComparer.Equals(unaryNumber, _unaryOne))
25             {
26                 return links.GetOrCreate(_unaryOne, _unaryOne);
27             }
28             var source = links.GetSource(unaryNumber);
29             var target = links.GetTarget(unaryNumber);
30             if (_equalityComparer.Equals(source, target))
31             {
32                 return links.GetOrCreate(unaryNumber, _unaryOne);
33             }
34             else
35             {
36                 return links.GetOrCreate(source, Increment(target));
37             }
38         }
39     }
40 }

```

1.25 ./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      public class CachedFrequencyIncrementingSequenceIndex<TLink> : ISequenceIndex<TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
12             ↪ EqualityComparer<TLink>.Default;
13
14         private readonly LinkFrequenciesCache<TLink> _cache;
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public CachedFrequencyIncrementingSequenceIndex(LinkFrequenciesCache<TLink> cache) =>
18             ↪ _cache = cache;
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         public bool Add(ICollection<TLink> sequence)
22         {
23             var indexed = true;

```

```

22     var i = sequence.Count;
23     while (--i >= 1 && (indexed = IsIndexedWithIncrement(sequence[i - 1], sequence[i])))
24         ↳ { }
25     for (; i >= 1; i--)
26     {
27         _cache.IncrementFrequency(sequence[i - 1], sequence[i]);
28     }
29     return indexed;
30 }
31 [MethodImpl(MethodImplOptions.AggressiveInlining)]
32 private bool IsIndexedWithIncrement(TLink source, TLink target)
33 {
34     var frequency = _cache.GetFrequency(source, target);
35     if (frequency == null)
36     {
37         return false;
38     }
39     var indexed = !_equalityComparer.Equals(frequency.Frequency, default);
40     if (indexed)
41     {
42         _cache.IncrementFrequency(source, target);
43     }
44     return indexed;
45 }
46
47 [MethodImpl(MethodImplOptions.AggressiveInlining)]
48 public bool MightContain(ICollection<TLink> sequence)
49 {
50     var indexed = true;
51     var i = sequence.Count;
52     while (--i >= 1 && (indexed = IsIndexed(sequence[i - 1], sequence[i]))) { }
53     return indexed;
54 }
55
56 [MethodImpl(MethodImplOptions.AggressiveInlining)]
57 private bool IsIndexed(TLink source, TLink target)
58 {
59     var frequency = _cache.GetFrequency(source, target);
60     if (frequency == null)
61     {
62         return false;
63     }
64     return !_equalityComparer.Equals(frequency.Frequency, default);
65 }
66 }
67 }

```

1.26 ./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.Incrementers;
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     public class FrequencyIncrementingSequenceIndex<TLink> : SequenceIndex<TLink>,
11         ↳ ISequenceIndex<TLink>
12     {
13         private static readonly EqualityComparer<TLink> _equalityComparer =
14             ↳ EqualityComparer<TLink>.Default;
15
16         private readonly IProperty<TLink, TLink> _frequencyPropertyOperator;
17         private readonly IIncrementer<TLink> _frequencyIncrementer;
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         public FrequencyIncrementingSequenceIndex(ICollection<TLink> links, IProperty<TLink, TLink>
21             ↳ frequencyPropertyOperator, IIncrementer<TLink> frequencyIncrementer)
22             : base(links)
23         {
24             _frequencyPropertyOperator = frequencyPropertyOperator;
25             _frequencyIncrementer = frequencyIncrementer;
26         }
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         public override bool Add(ICollection<TLink> sequence)
30         {
31             var indexed = true;

```

```

29     var i = sequence.Count;
30     while (--i >= 1 && (indexed = IsIndexedWithIncrement(sequence[i - 1], sequence[i])))
31         ↪ { }
32     for (; i >= 1; i--)
33     {
34         Increment(_links.GetOrCreate(sequence[i - 1], sequence[i]));
35     }
36     return indexed;
37 }
38 [MethodImpl(MethodImplOptions.AggressiveInlining)]
39 private bool IsIndexedWithIncrement(TLink source, TLink target)
40 {
41     var link = _links.SearchOrDefault(source, target);
42     var indexed = !_equalityComparer.Equals(link, default);
43     if (indexed)
44     {
45         Increment(link);
46     }
47     return indexed;
48 }
49
50 [MethodImpl(MethodImplOptions.AggressiveInlining)]
51 private void Increment(TLink link)
52 {
53     var previousFrequency = _frequencyPropertyOperator.Get(link);
54     var frequency = _frequencyIncrementer.Increment(previousFrequency);
55     _frequencyPropertyOperator.Set(link, frequency);
56 }
57 }
58 }

```

1.27 ./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     public interface ISequenceIndex<TLink>
9     {
10         /// <summary>
11         /// Индексирует последовательность глобально, и возвращает значение,
12         /// определяющие была ли запрошенная последовательность проиндексирована ранее.
13         /// </summary>
14         /// <param name="sequence">Последовательность для индексации.</param>
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         bool Add(IList<TLink> sequence);
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         bool MightContain(IList<TLink> sequence);
20     }
21 }

```

1.28 ./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     public class SequenceIndex<TLink> : LinksOperatorBase<TLink>, ISequenceIndex<TLink>
9     {
10         private static readonly EqualityComparer<TLink> _equalityComparer =
11             ↪ EqualityComparer<TLink>.Default;
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public SequenceIndex(ILinks<TLink> links) : base(links) { }
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public virtual bool Add(IList<TLink> sequence)
18         {
19             var indexed = true;
20             var i = sequence.Count;
21             while (--i >= 1 && (indexed =
22                 ↪ !_equalityComparer.Equals(_links.SearchOrDefault(sequence[i - 1], sequence[i]),
23                 ↪ default))) { }
24         }
25     }
26 }

```

```

21         for (; i >= 1; i--)
22         {
23             _links.GetOrCreate(sequence[i - 1], sequence[i]);
24         }
25         return indexed;
26     }
27
28     [MethodImpl(MethodImplOptions.AggressiveInlining)]
29     public virtual bool MightContain(IList<TLink> sequence)
30     {
31         var indexed = true;
32         var i = sequence.Count;
33         while (--i >= 1 && (indexed =
34             ↪ !_equalityComparer.Equals(_links.SearchOrDefault(sequence[i - 1], sequence[i]),
35             ↪ default))) { }
36         return indexed;
37     }
38 }

```

1.29 ./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      public class SynchronizedSequenceIndex<TLink> : ISequenceIndex<TLink>
9      {
10         private static readonly EqualityComparer<TLink> _equalityComparer =
11             ↪ EqualityComparer<TLink>.Default;
12
13         private readonly ISynchronizedLinks<TLink> _links;
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         public SynchronizedSequenceIndex(ISynchronizedLinks<TLink> links) => _links = links;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         public bool Add(IList<TLink> sequence)
20         {
21             var indexed = true;
22             var i = sequence.Count;
23             var links = _links.Unsync;
24             _links.SyncRoot.ExecuteReadOperation(() =>
25             {
26                 while (--i >= 1 && (indexed =
27                     ↪ !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1],
28                     ↪ sequence[i]), default))) { }
29             });
30             if (!indexed)
31             {
32                 _links.SyncRoot.ExecuteWriteOperation(() =>
33                 {
34                     for (; i >= 1; i--)
35                     {
36                         links.GetOrCreate(sequence[i - 1], sequence[i]);
37                     }
38                 });
39             }
40             return indexed;
41         }
42
43         [MethodImpl(MethodImplOptions.AggressiveInlining)]
44         public bool MightContain(IList<TLink> sequence)
45         {
46             var links = _links.Unsync;
47             return _links.SyncRoot.ExecuteReadOperation(() =>
48             {
49                 var indexed = true;
50                 var i = sequence.Count;
51                 while (--i >= 1 && (indexed =
52                     ↪ !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1],
53                     ↪ sequence[i]), default))) { }
54                 return indexed;
55             });
56         }
57     }
58 }

```

1.30 ./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     public class Unindex<TLink> : ISequenceIndex<TLink>
9     {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]
11         public virtual bool Add(IList<TLink> sequence) => false;
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public virtual bool MightContain(IList<TLink> sequence) => true;
15     }
16 }

```

1.31 ./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     public class DecimalToRationalConverter<TLink> : LinksDecoratorBase<TLink>,
12         ↪ IConverter<decimal, TLink>
13     {
14         where TLink: struct
15
16         public readonly BigIntegerToRawNumberSequenceConverter<TLink>
17         ↪ BigIntegerToRawNumberSequenceConverter;
18
19         public DecimalToRationalConverter(ILinks<TLink> links,
20         ↪ BigIntegerToRawNumberSequenceConverter<TLink>
21         ↪ BigIntegerToRawNumberSequenceConverter) : base(links)
22         {
23             BigIntegerToRawNumberSequenceConverter = BigIntegerToRawNumberSequenceConverter;
24         }
25
26         public TLink Convert(decimal @decimal)
27         {
28             var decimalAsString = @decimal.ToString(CultureInfo.InvariantCulture);
29             var dotPosition = decimalAsString.IndexOf('.');
30             var decimalWithoutDots = decimalAsString;
31             int digitsAfterDot = 0;
32             if (dotPosition != -1)
33             {
34                 decimalWithoutDots = decimalWithoutDots.Remove(dotPosition, 1);
35                 digitsAfterDot = decimalAsString.Length - 1 - dotPosition;
36             }
37             BigInteger denominator = new(System.Math.Pow(10, digitsAfterDot));
38             BigInteger numerator = BigInteger.Parse(decimalWithoutDots);
39             BigInteger greatestCommonDivisor;
40             do
41             {
42                 greatestCommonDivisor = BigInteger.GreatestCommonDivisor(numerator, denominator);
43                 numerator /= greatestCommonDivisor;
44                 denominator /= greatestCommonDivisor;
45             }
46             while (greatestCommonDivisor > 1);
47             var numeratorLink = BigIntegerToRawNumberSequenceConverter.Convert(numerator);
48             var denominatorLink = BigIntegerToRawNumberSequenceConverter.Convert(denominator);
49             return _links.GetOrCreate(numeratorLink, denominatorLink);
50         }
51     }
52 }

```

1.32 ./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     public class RationalToDecimalConverter<TLink> : LinksDecoratorBase<TLink>,
    ↪ IConverter<TLink, decimal>
10     where TLink: struct
11     {
12         public readonly RawNumberSequenceToBigIntegerConverter<TLink>
    ↪ RawNumberSequenceToBigIntegerConverter;
13
14         public RationalToDecimalConverter(ILinks<TLink> links,
    ↪ RawNumberSequenceToBigIntegerConverter<TLink>
    ↪ rawNumberSequenceToBigIntegerConverter) : base(links)
15         {
16             RawNumberSequenceToBigIntegerConverter = rawNumberSequenceToBigIntegerConverter;
17         }
18
19         public decimal Convert(TLink rationalNumber)
20         {
21             var numerator = (decimal)RawNumberSequenceToBigIntegerConverter.Convert(_links.GetSo
    ↪ urce(rationalNumber));
22             var denominator = (decimal)RawNumberSequenceToBigIntegerConverter.Convert(_links.Get
    ↪ Target(rationalNumber));
23             return numerator / denominator;
24         }
25     }
26 }

```

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

```

1 using System.Collections.Generic;
2 using System.Numerics;
3 using Platform.Converters;
4 using Platform.Data.Doublets.Decorators;
5 using Platform.Numbers;
6 using Platform.Reflection;
7 using Platform.Unsafe;
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     public class BigIntegerToRawNumberSequenceConverter<TLink> : LinksDecoratorBase<TLink>,
    ↪ IConverter<BigInteger, TLink>
14     where TLink : struct
15     {
16         public static readonly TLink MaximumValue = NumericType<TLink>.MaxValue;
17         public static readonly TLink BitMask = Bit.ShiftRight(MaximumValue, 1);
18         public readonly IConverter<TLink> AddressToNumberConverter;
19         public readonly IConverter<IList<TLink>, TLink> ListToSequenceConverter;
20         public readonly TLink NegativeNumberMarker;
21
22         public BigIntegerToRawNumberSequenceConverter(ILinks<TLink> links, IConverter<TLink>
    ↪ addressToNumberConverter, IConverter<IList<TLink>, TLink> listToSequenceConverter,
    ↪ TLink negativeNumberMarker) : base(links)
23         {
24             AddressToNumberConverter = addressToNumberConverter;
25             ListToSequenceConverter = listToSequenceConverter;
26             NegativeNumberMarker = negativeNumberMarker;
27         }
28
29         private List<TLink> GetRawNumberParts(BigInteger bigInteger)
30         {
31             List<TLink> rawNumbers = new();
32             BigInteger currentBigInt = bigInteger;
33             do
34             {
35                 var bigIntBytes = currentBigInt.ToByteArray();
36                 var bigIntWithBitMask = Bit.And(bigIntBytes.ToStructure<TLink>(), BitMask);
37                 var rawNumber = AddressToNumberConverter.Convert(bigIntWithBitMask);
38                 rawNumbers.Add(rawNumber);
39                 currentBigInt >>= 63;
40             }
41             while (currentBigInt > 0);
42             return rawNumbers;
43         }
44
45         public TLink Convert(BigInteger bigInteger)
46         {
47             var sign = bigInteger.Sign;
48             var number = GetRawNumberParts(sign == -1 ? BigInteger.Negate(bigInteger) :
    ↪ bigInteger);
49             var numberSequence = ListToSequenceConverter.Convert(number);

```



```

50         return sign == -1 ? _links.GetOrCreate(NegativeNumberMarker, numberSequence) :
           ↪ numberSequence;
51     }
52 }
53 }

```

1.34 ./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     public class LongRawNumberSequenceToNumberConverter<TSource, TTarget> :
           ↪ LinksDecoratorBase<TSource>, IConverter<TSource, TTarget>
14     {
15         private static readonly int _bitsPerRawNumber = NumericType<TSource>.BitsSize - 1;
16         private static readonly uncheckedConverter<TSource, TTarget> _sourceToTargetConverter =
           ↪ uncheckedConverter<TSource, TTarget>.Default;
17
18         private readonly IConverter<TSource> _numberToAddressConverter;
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         public LongRawNumberSequenceToNumberConverter(ILinks<TSource> links, IConverter<TSource>
           ↪ numberToAddressConverter) : base(links) => _numberToAddressConverter =
           ↪ numberToAddressConverter;
22
23         [MethodImpl(MethodImplOptions.AggressiveInlining)]
24         public TTarget Convert(TSource source)
25         {
26             var constants = Links.Constants;
27             var externalReferencesRange = constants.ExternalReferencesRange;
28             if (externalReferencesRange.HasValue &&
           ↪ externalReferencesRange.Value.Contains(source))
29             {
30                 return
           ↪ _sourceToTargetConverter.Convert(_numberToAddressConverter.Convert(source));
31             }
32             else
33             {
34                 var pair = Links.GetLink(source);
35                 var walker = new LeftSequenceWalker<TSource>(Links, new DefaultStack<TSource>(),
           ↪ (link) => externalReferencesRange.HasValue &&
           ↪ externalReferencesRange.Value.Contains(link));
36                 TTarget result = default;
37                 foreach (var element in walker.Walk(source))
38                 {
39                     result = Bit.Or(Bit.ShiftLeft(result, _bitsPerRawNumber), Convert(element));
40                 }
41                 return result;
42             }
43         }
44     }
45 }

```

1.35 ./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     public class NumberToLongRawNumberSequenceConverter<TSource, TTarget> :
           ↪ LinksDecoratorBase<TTarget>, IConverter<TSource, TTarget>
13     {
14         private static readonly Comparer<TSource> _comparer = Comparer<TSource>.Default;
15         private static readonly TSource _maximumValue = NumericType<TSource>.MaxValue;
16         private static readonly int _bitsPerRawNumber = NumericType<TTarget>.BitsSize - 1;
17         private static readonly TSource _bitMask = Bit.ShiftRight(_maximumValue,
           ↪ NumericType<TTarget>.BitsSize + 1);

```

```

18     private static readonly TSource _maximumConvertibleAddress = CheckedConverter<TTarget,
19         ↪ TSource>.Default.Convert(Arithmetic.Decrement(Hybrid<TTarget>.ExternalZero));
20     private static readonly UncheckedConverter<TSource, TTarget> _sourceToTargetConverter =
21         ↪ UncheckedConverter<TSource, TTarget>.Default;
22
23     private readonly IConverter<TTarget> _addressToNumberConverter;
24
25     [MethodImpl(MethodImplOptions.AggressiveInlining)]
26     public NumberToLongRawNumberSequenceConverter(ILinks<TTarget> links, IConverter<TTarget>
27         ↪ addressToNumberConverter) : base(links) => _addressToNumberConverter =
28         ↪ addressToNumberConverter;
29
30     [MethodImpl(MethodImplOptions.AggressiveInlining)]
31     public TTarget Convert(TSource source)
32     {
33         if (_comparer.Compare(source, _maximumConvertibleAddress) > 0)
34         {
35             var numberPart = Bit.And(source, _bitMask);
36             var convertedNumber = _addressToNumberConverter.Convert(_sourceToTargetConverter
37                 ↪ .Convert(numberPart));
38             return Links.GetOrCreate(convertedNumber, Convert(Bit.ShiftRight(source,
39                 ↪ _bitsPerRawNumber)));
40         }
41         else
42         {
43             return
44                 ↪ _addressToNumberConverter.Convert(_sourceToTargetConverter.Convert(source));
45         }
46     }
47 }

```

1.36 ./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.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     public class RawNumberSequenceToBigIntegerConverter<TLink> : LinksDecoratorBase<TLink>,
15         ↪ IConverter<TLink, BigInteger>
16     where TLink : struct
17     {
18         public readonly EqualityComparer<TLink> EqualityComparer =
19             ↪ EqualityComparer<TLink>.Default;
20         public readonly IConverter<TLink, TLink> NumberToAddressConverter;
21         public readonly LeftSequenceWalker<TLink> LeftSequenceWalker;
22         public readonly TLink NegativeNumberMarker;
23
24         public RawNumberSequenceToBigIntegerConverter(ILinks<TLink> links, IConverter<TLink,
25             ↪ TLink> numberToAddressConverter, TLink negativeNumberMarker) : base(links)
26         {
27             NumberToAddressConverter = numberToAddressConverter;
28             LeftSequenceWalker = new(links, new DefaultStack<TLink>());
29             NegativeNumberMarker = negativeNumberMarker;
30         }
31
32         public BigInteger Convert(TLink bigInteger)
33         {
34             var sign = 1;
35             var bigIntegerSequence = bigInteger;
36             if (EqualityComparer.Equals(_links.GetSource(bigIntegerSequence),
37                 ↪ NegativeNumberMarker))
38             {
39                 sign = -1;
40                 bigIntegerSequence = _links.GetTarget(bigInteger);
41             }
42             using var enumerator = LeftSequenceWalker.Walk(bigIntegerSequence).GetEnumerator();
43             if (!enumerator.MoveNext())
44             {
45                 throw new Exception("Raw number sequence cannot be empty.");
46             }
47             var nextPart = NumberToAddressConverter.Convert(enumerator.Current);

```

```

44     BigInteger currentBigInt = new(nextPart.ToBytes());
45     while (enumerator.MoveNext())
46     {
47         currentBigInt <= 63;
48         nextPart = NumberToAddressConverter.Convert(enumerator.Current);
49         currentBigInt |= new BigInteger(nextPart.ToBytes());
50     }
51     return sign == -1 ? BigInteger.Negate(currentBigInt) : currentBigInt;
52 }
53 }
54 }

```

1.37 ./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     public class AddressToUnaryNumberConverter<TLink> : LinksOperatorBase<TLink>,
12         ⇨ IConverter<TLink>
13     {
14         private static readonly EqualityComparer<TLink> _equalityComparer =
15             ⇨ EqualityComparer<TLink>.Default;
16         private static readonly TLink _zero = default;
17         private static readonly TLink _one = Arithmetic.Increment(_zero);
18
19         private readonly IConverter<int, TLink> _powerOf2ToUnaryNumberConverter;
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public AddressToUnaryNumberConverter(ILinks<TLink> links, IConverter<int, TLink>
23             ⇨ powerOf2ToUnaryNumberConverter) : base(links) => _powerOf2ToUnaryNumberConverter =
24             ⇨ powerOf2ToUnaryNumberConverter;
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public TLink Convert(TLink number)
28         {
29             var links = _links;
30             var nullConstant = links.Constants.Null;
31             var target = nullConstant;
32             for (var i = 0; !_equalityComparer.Equals(number, _zero) && i <
33                 ⇨ NumericType<TLink>.BitsSize; i++)
34             {
35                 if (_equalityComparer.Equals(Bit.And(number, _one), _one))
36                 {
37                     target = _equalityComparer.Equals(target, nullConstant)
38                         ? _powerOf2ToUnaryNumberConverter.Convert(i)
39                         : links.GetOrCreate(_powerOf2ToUnaryNumberConverter.Convert(i), target);
40                 }
41                 number = Bit.ShiftRight(number, 1);
42             }
43             return target;
44         }
45     }
46 }

```

1.38 ./csharp/Platform.Data.Doublets.Sequences/Numbers/Unary/LinkToItsFrequencyNumberConveter.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     public class LinkToItsFrequencyNumberConveter<TLink> : LinksOperatorBase<TLink>,
12         ⇨ IConverter<Doublet<TLink>, TLink>
13     {
14         private static readonly EqualityComparer<TLink> _equalityComparer =
15             ⇨ EqualityComparer<TLink>.Default;
16
17         private readonly IProperty<TLink, TLink> _frequencyPropertyOperator;
18         private readonly IConverter<TLink> _unaryNumberToAddressConverter;
19     }
20 }

```

```

18     [MethodImpl(MethodImplOptions.AggressiveInlining)]
19     public LinkToItsFrequencyNumberConveter(
20         ILinks<TLink> links,
21         IProperty<TLink, TLink> frequencyPropertyOperator,
22         IConverter<TLink> unaryNumberToAddressConverter)
23         : base(links)
24     {
25         _frequencyPropertyOperator = frequencyPropertyOperator;
26         _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
27     }
28
29     [MethodImpl(MethodImplOptions.AggressiveInlining)]
30     public TLink Convert(Doublet<TLink> doublet)
31     {
32         var links = _links;
33         var link = links.SearchOrDefault(doublet.Source, doublet.Target);
34         if (_equalityComparer.Equals(link, default))
35         {
36             throw new ArgumentException($"Link ({doublet}) not found.", nameof(doublet));
37         }
38         var frequency = _frequencyPropertyOperator.Get(link);
39         if (_equalityComparer.Equals(frequency, default))
40         {
41             return default;
42         }
43         var frequencyNumber = links.GetSource(frequency);
44         return _unaryNumberToAddressConverter.Convert(frequencyNumber);
45     }
46 }
47 }

```

1.39 ./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     public class PowerOf2ToUnaryNumberConverter<TLink> : LinksOperatorBase<TLink>,
12         ↪ IConverter<int, TLink>
13     {
14         private static readonly EqualityComparer<TLink> _equalityComparer =
15             ↪ EqualityComparer<TLink>.Default;
16
17         private readonly TLink[] _unaryNumberPowersOf2;
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         public PowerOf2ToUnaryNumberConverter(ILinks<TLink> links, TLink one) : base(links)
21         {
22             _unaryNumberPowersOf2 = new TLink[64];
23             _unaryNumberPowersOf2[0] = one;
24         }
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public TLink Convert(int power)
28         {
29             Ensure.Always.ArgumentInRange(power, new Range<int>(0, _unaryNumberPowersOf2.Length
30                 ↪ - 1), nameof(power));
31             if (!_equalityComparer.Equals(_unaryNumberPowersOf2[power], default))
32             {
33                 return _unaryNumberPowersOf2[power];
34             }
35             var previousPowerOf2 = Convert(power - 1);
36             var powerOf2 = _links.GetOrCreate(previousPowerOf2, previousPowerOf2);
37             _unaryNumberPowersOf2[power] = powerOf2;
38             return powerOf2;
39         }
40     }
41 }

```

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

```

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     public class UnaryNumberToAddressAddOperationConverter<TLink> : LinksOperatorBase<TLink>,
11         ↳ IConverter<TLink>
12     {
13         private static readonly EqualityComparer<TLink> _equalityComparer =
14             ↳ EqualityComparer<TLink>.Default;
15         private static readonly UncheckedConverter<TLink, ulong> _addressToUInt64Converter =
16             ↳ UncheckedConverter<TLink, ulong>.Default;
17         private static readonly UncheckedConverter<ulong, TLink> _uint64ToAddressConverter =
18             ↳ UncheckedConverter<ulong, TLink>.Default;
19         private static readonly TLink _zero = default;
20         private static readonly TLink _one = Arithmetic.Increment(_zero);
21
22         private readonly Dictionary<TLink, TLink> _unaryToUInt64;
23         private readonly TLink _unaryOne;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         public UnaryNumberToAddressAddOperationConverter(ILinks<TLink> links, TLink unaryOne)
27             : base(links)
28         {
29             _unaryOne = unaryOne;
30             _unaryToUInt64 = CreateUnaryToUInt64Dictionary(links, unaryOne);
31         }
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         public TLink Convert(TLink unaryNumber)
35         {
36             if (_equalityComparer.Equals(unaryNumber, default))
37             {
38                 return default;
39             }
40             if (_equalityComparer.Equals(unaryNumber, _unaryOne))
41             {
42                 return _one;
43             }
44             var links = _links;
45             var source = links.GetSource(unaryNumber);
46             var target = links.GetTarget(unaryNumber);
47             if (_equalityComparer.Equals(source, target))
48             {
49                 return _unaryToUInt64[unaryNumber];
50             }
51             else
52             {
53                 var result = _unaryToUInt64[source];
54                 TLink lastValue;
55                 while (!_unaryToUInt64.TryGetValue(target, out lastValue))
56                 {
57                     source = links.GetSource(target);
58                     result = Arithmetic<TLink>.Add(result, _unaryToUInt64[source]);
59                     target = links.GetTarget(target);
60                 }
61                 result = Arithmetic<TLink>.Add(result, lastValue);
62                 return result;
63             }
64         }
65
66         [MethodImpl(MethodImplOptions.AggressiveInlining)]
67         private static Dictionary<TLink, TLink> CreateUnaryToUInt64Dictionary(ILinks<TLink>
68             ↳ links, TLink unaryOne)
69         {
70             var unaryToUInt64 = new Dictionary<TLink, TLink>
71             {
72                 { unaryOne, _one }
73             };
74             var unary = unaryOne;
75             var number = _one;
76             for (var i = 1; i < 64; i++)
77             {
78                 unary = links.GetOrCreate(unary, unary);
79                 number = Double(number);
80                 unaryToUInt64.Add(unary, number);
81             }
82             return unaryToUInt64;
83         }
84     }
85 }
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100

```

```

80     [MethodImpl(MethodImplOptions.AggressiveInlining)]
81     private static TLink Double(TLink number) =>
        ↪ _uint64ToAddressConverter.Convert(_addressToUInt64Converter.Convert(number) * 2UL);
82 }
83 }

```

1.41 ./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     public class UnaryNumberToAddressOrOperationConverter<TLink> : LinksOperatorBase<TLink>,
        ↪ IConverter<TLink>
12     {
13         private static readonly EqualityComparer<TLink> _equalityComparer =
            ↪ EqualityComparer<TLink>.Default;
14         private static readonly TLink _zero = default;
15         private static readonly TLink _one = Arithmetic.Increment(_zero);
16
17         private readonly IDictionary<TLink, int> _unaryNumberPowerOf2Indicies;
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         public UnaryNumberToAddressOrOperationConverter(ILinks<TLink> links, IConverter<int,
            ↪ TLink> powerOf2ToUnaryNumberConverter) : base(links) => _unaryNumberPowerOf2Indicies
            ↪ = CreateUnaryNumberPowerOf2IndiciesDictionary(powerOf2ToUnaryNumberConverter);
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         public TLink Convert(TLink sourceNumber)
24         {
25             var links = _links;
26             var nullConstant = links.Constants.Null;
27             var source = sourceNumber;
28             var target = nullConstant;
29             if (!_equalityComparer.Equals(source, nullConstant))
30             {
31                 while (true)
32                 {
33                     if (_unaryNumberPowerOf2Indicies.TryGetValue(source, out int powerOf2Index))
34                     {
35                         SetBit(ref target, powerOf2Index);
36                         break;
37                     }
38                     else
39                     {
40                         powerOf2Index = _unaryNumberPowerOf2Indicies[links.GetSource(source)];
41                         SetBit(ref target, powerOf2Index);
42                         source = links.GetTarget(source);
43                     }
44                 }
45             }
46             return target;
47         }
48
49         [MethodImpl(MethodImplOptions.AggressiveInlining)]
50         private static Dictionary<TLink, int>
            ↪ CreateUnaryNumberPowerOf2IndiciesDictionary(IConverter<int, TLink>
            ↪ powerOf2ToUnaryNumberConverter)
51         {
52             var unaryNumberPowerOf2Indicies = new Dictionary<TLink, int>();
53             for (int i = 0; i < NumericType<TLink>.BitsSize; i++)
54             {
55                 unaryNumberPowerOf2Indicies.Add(powerOf2ToUnaryNumberConverter.Convert(i), i);
56             }
57             return unaryNumberPowerOf2Indicies;
58         }
59
60         [MethodImpl(MethodImplOptions.AggressiveInlining)]
61         private static void SetBit(ref TLink target, int powerOf2Index) => target =
            ↪ Bit.Or(target, Bit.ShiftLeft(_one, powerOf2Index));
62     }
63 }

```

1.42 ./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     partial class Sequences
21     {
22         #region Create All Variants (Not Practical)
23
24         /// <remarks>
25         /// Number of links that is needed to generate all variants for
26         /// sequence of length N corresponds to https://oeis.org/A014143/list sequence.
27         /// </remarks>
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         public ulong[] CreateAllVariants2(ulong[] sequence)
30         {
31             return _sync.ExecuteWriteOperation(() =>
32             {
33                 if (sequence.IsNullOrEmpty())
34                 {
35                     return Array.Empty<ulong>();
36                 }
37                 Links.EnsureLinkExists(sequence);
38                 if (sequence.Length == 1)
39                 {
40                     return sequence;
41                 }
42                 return CreateAllVariants2Core(sequence, 0, (ulong)sequence.Length - 1);
43             });
44         }
45
46         [MethodImpl(MethodImplOptions.AggressiveInlining)]
47         private ulong[] CreateAllVariants2Core(ulong[] sequence, ulong startAt, ulong stopAt)
48         {
49             if ((stopAt - startAt) == 0)
50             {
51                 return new[] { sequence[startAt] };
52             }
53             if ((stopAt - startAt) == 1)
54             {
55                 return new[] { Links.Unsync.GetOrCreate(sequence[startAt], sequence[stopAt]) };
56             }
57             var variants = new ulong[Platform.Numbers.Math.Catalan(stopAt - startAt)];
58             var last = 0;
59             for (var splitter = startAt; splitter < stopAt; splitter++)
60             {
61                 var left = CreateAllVariants2Core(sequence, startAt, splitter);
62                 var right = CreateAllVariants2Core(sequence, splitter + 1, stopAt);
63                 for (var i = 0; i < left.Length; i++)
64                 {
65                     for (var j = 0; j < right.Length; j++)
66                     {
67                         var variant = Links.Unsync.GetOrCreate(left[i], right[j]);
68                         if (variant == Constants.Null)
69                         {
70                             throw new NotImplementedException("Creation cancellation is not
71                                 ↪ implemented.");
72                         }
73                         variants[last++] = variant;
74                     }
75                 }
76             }
77             return variants;
78         }
79     }
80 }

```

```

78 [MethodImpl(MethodImplOptions.AggressiveInlining)]
79 public List<ulong> CreateAllVariants1(params ulong[] sequence)
80 {
81     return _sync.ExecuteWriteOperation(() =>
82     {
83         if (sequence.IsNullOrEmpty())
84         {
85             return new List<ulong>();
86         }
87         Links.Unsync.EnsureLinkExists(sequence);
88         if (sequence.Length == 1)
89         {
90             return new List<ulong> { sequence[0] };
91         }
92         var results = new
93             ↳ List<ulong>((int)Platform.Numbers.Math.Catalan((ulong)sequence.Length));
94         return CreateAllVariants1Core(sequence, results);
95     });
96 }
97
98 [MethodImpl(MethodImplOptions.AggressiveInlining)]
99 private List<ulong> CreateAllVariants1Core(ulong[] sequence, List<ulong> results)
100 {
101     if (sequence.Length == 2)
102     {
103         var link = Links.Unsync.GetOrCreate(sequence[0], sequence[1]);
104         if (link == Constants.Null)
105         {
106             throw new NotImplementedException("Creation cancellation is not
107                 ↳ implemented.");
108         }
109         results.Add(link);
110         return results;
111     }
112     var innerSequenceLength = sequence.Length - 1;
113     var innerSequence = new ulong[innerSequenceLength];
114     for (var li = 0; li < innerSequenceLength; li++)
115     {
116         var link = Links.Unsync.GetOrCreate(sequence[li], sequence[li + 1]);
117         if (link == Constants.Null)
118         {
119             throw new NotImplementedException("Creation cancellation is not
120                 ↳ implemented.");
121         }
122         for (var isi = 0; isi < li; isi++)
123         {
124             innerSequence[isi] = sequence[isi];
125         }
126         innerSequence[li] = link;
127         for (var isi = li + 1; isi < innerSequenceLength; isi++)
128         {
129             innerSequence[isi] = sequence[isi + 1];
130         }
131         CreateAllVariants1Core(innerSequence, results);
132     }
133     return results;
134 }
135
136 #endregion
137
138 [MethodImpl(MethodImplOptions.AggressiveInlining)]
139 public HashSet<ulong> Each1(params ulong[] sequence)
140 {
141     var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
142     Each1(link =>
143     {
144         if (!visitedLinks.Contains(link))
145         {
146             visitedLinks.Add(link); // изучить почему случаются повторы
147         }
148         return true;
149     }, sequence);
150     return visitedLinks;
151 }
152
153 [MethodImpl(MethodImplOptions.AggressiveInlining)]
154 private void Each1(Func<ulong, bool> handler, params ulong[] sequence)

```



```

153 {
154     if (sequence.Length == 2)
155     {
156         Links.Unsync.Each(sequence[0], sequence[1], handler);
157     }
158     else
159     {
160         var innerSequenceLength = sequence.Length - 1;
161         for (var li = 0; li < innerSequenceLength; li++)
162         {
163             var left = sequence[li];
164             var right = sequence[li + 1];
165             if (left == 0 && right == 0)
166             {
167                 continue;
168             }
169             var linkIndex = li;
170             ulong[] innerSequence = null;
171             Links.Unsync.Each(doublet =>
172             {
173                 if (innerSequence == null)
174                 {
175                     innerSequence = new ulong[innerSequenceLength];
176                     for (var isi = 0; isi < linkIndex; isi++)
177                     {
178                         innerSequence[isi] = sequence[isi];
179                     }
180                     for (var isi = linkIndex + 1; isi < innerSequenceLength; isi++)
181                     {
182                         innerSequence[isi] = sequence[isi + 1];
183                     }
184                 }
185                 innerSequence[linkIndex] = doublet[Constants.IndexPart];
186                 Each1(handler, innerSequence);
187                 return Constants.Continue;
188             }, Constants.Any, left, right);
189         }
190     }
191 }
192
193 [MethodImpl(MethodImplOptions.AggressiveInlining)]
194 public HashSet<ulong> EachPart(params ulong[] sequence)
195 {
196     var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
197     EachPartCore(link =>
198     {
199         var linkIndex = link[Constants.IndexPart];
200         if (!visitedLinks.Contains(linkIndex))
201         {
202             visitedLinks.Add(linkIndex); // изучить почему случаются повторы
203         }
204         return Constants.Continue;
205     }, sequence);
206     return visitedLinks;
207 }
208
209 [MethodImpl(MethodImplOptions.AggressiveInlining)]
210 public void EachPart(Func<IList<LinkIndex>, LinkIndex> handler, params ulong[] sequence)
211 {
212     var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
213     EachPartCore(link =>
214     {
215         var linkIndex = link[Constants.IndexPart];
216         if (!visitedLinks.Contains(linkIndex))
217         {
218             visitedLinks.Add(linkIndex); // изучить почему случаются повторы
219             return handler(new LinkAddress<LinkIndex>(linkIndex));
220         }
221         return Constants.Continue;
222     }, sequence);
223 }
224
225 [MethodImpl(MethodImplOptions.AggressiveInlining)]
226 private void EachPartCore(Func<IList<LinkIndex>, LinkIndex> handler, params ulong[]
↪ sequence)
227 {
228     if (sequence.IsNullOrEmpty())
229     {

```

```

230         return;
231     }
232     Links.EnsureLinkIsAnyOrExists(sequence);
233     if (sequence.Length == 1)
234     {
235         var link = sequence[0];
236         if (link > 0)
237         {
238             handler(new LinkAddress<LinkIndex>(link));
239         }
240         else
241         {
242             Links.Each(Constants.Any, Constants.Any, handler);
243         }
244     }
245     else if (sequence.Length == 2)
246     {
247         //_links.Each(sequence[0], sequence[1], handler);
248         //  o_|      x_o ...
249         // x_|      |__|
250         Links.Each(sequence[1], Constants.Any, doublet =>
251         {
252             var match = Links.SearchOrDefault(sequence[0], doublet);
253             if (match != Constants.Null)
254             {
255                 handler(new LinkAddress<LinkIndex>(match));
256             }
257             return true;
258         });
259         // |_x      ... x_o
260         // |_o      |__|
261         Links.Each(Constants.Any, sequence[0], doublet =>
262         {
263             var match = Links.SearchOrDefault(doublet, sequence[1]);
264             if (match != 0)
265             {
266                 handler(new LinkAddress<LinkIndex>(match));
267             }
268             return true;
269         });
270         //      . _x o _ .
271         //      |__|
272         PartialStepRight(x => handler(x), sequence[0], sequence[1]);
273     }
274     else
275     {
276         throw new NotImplementedException();
277     }
278 }
279
280 [MethodImpl(MethodImplOptions.AggressiveInlining)]
281 private void PartialStepRight(Action<IList<LinkIndex>> handler, ulong left, ulong right)
282 {
283     Links.Unsync.Each(Constants.Any, left, doublet =>
284     {
285         StepRight(handler, doublet, right);
286         if (left != doublet)
287         {
288             PartialStepRight(handler, doublet, right);
289         }
290         return true;
291     });
292 }
293
294 [MethodImpl(MethodImplOptions.AggressiveInlining)]
295 private void StepRight(Action<IList<LinkIndex>> handler, ulong left, ulong right)
296 {
297     Links.Unsync.Each(left, Constants.Any, rightStep =>
298     {
299         TryStepRightUp(handler, right, rightStep);
300         return true;
301     });
302 }
303
304 [MethodImpl(MethodImplOptions.AggressiveInlining)]
305 private void TryStepRightUp(Action<IList<LinkIndex>> handler, ulong right, ulong
↪ stepFrom)
306 {

```

```

307     var upStep = stepFrom;
308     var firstSource = Links.Unsync.GetTarget(upStep);
309     while (firstSource != right && firstSource != upStep)
310     {
311         upStep = firstSource;
312         firstSource = Links.Unsync.GetSource(upStep);
313     }
314     if (firstSource == right)
315     {
316         handler(new LinkAddress<LinkIndex>(stepFrom));
317     }
318 }
319
320 // TODO: Test
321 [MethodImpl(MethodImplOptions.AggressiveInlining)]
322 private void PartialStepLeft(Action<IList<LinkIndex>> handler, ulong left, ulong right)
323 {
324     Links.Unsync.Each(right, Constants.Any, doublet =>
325     {
326         StepLeft(handler, left, doublet);
327         if (right != doublet)
328         {
329             PartialStepLeft(handler, left, doublet);
330         }
331         return true;
332     });
333 }
334
335 [MethodImpl(MethodImplOptions.AggressiveInlining)]
336 private void StepLeft(Action<IList<LinkIndex>> handler, ulong left, ulong right)
337 {
338     Links.Unsync.Each(Constants.Any, right, leftStep =>
339     {
340         TryStepLeftUp(handler, left, leftStep);
341         return true;
342     });
343 }
344
345 [MethodImpl(MethodImplOptions.AggressiveInlining)]
346 private void TryStepLeftUp(Action<IList<LinkIndex>> handler, ulong left, ulong stepFrom)
347 {
348     var upStep = stepFrom;
349     var firstTarget = Links.Unsync.GetSource(upStep);
350     while (firstTarget != left && firstTarget != upStep)
351     {
352         upStep = firstTarget;
353         firstTarget = Links.Unsync.GetTarget(upStep);
354     }
355     if (firstTarget == left)
356     {
357         handler(new LinkAddress<LinkIndex>(stepFrom));
358     }
359 }
360
361 [MethodImpl(MethodImplOptions.AggressiveInlining)]
362 private bool StartsWith(ulong sequence, ulong link)
363 {
364     var upStep = sequence;
365     var firstSource = Links.Unsync.GetSource(upStep);
366     while (firstSource != link && firstSource != upStep)
367     {
368         upStep = firstSource;
369         firstSource = Links.Unsync.GetSource(upStep);
370     }
371     return firstSource == link;
372 }
373
374 [MethodImpl(MethodImplOptions.AggressiveInlining)]
375 private bool EndsWith(ulong sequence, ulong link)
376 {
377     var upStep = sequence;
378     var lastTarget = Links.Unsync.GetTarget(upStep);
379     while (lastTarget != link && lastTarget != upStep)
380     {
381         upStep = lastTarget;
382         lastTarget = Links.Unsync.GetTarget(upStep);
383     }
384     return lastTarget == link;
385 }

```

```

386 [MethodImpl(MethodImplOptions.AggressiveInlining)]
387 public List<ulong> GetAllMatchingSequences0(params ulong[] sequence)
388 {
389     return _sync.ExecuteReadOperation(() =>
390     {
391         var results = new List<ulong>();
392         if (sequence.Length > 0)
393         {
394             Links.EnsureLinkExists(sequence);
395             var firstElement = sequence[0];
396             if (sequence.Length == 1)
397             {
398                 results.Add(firstElement);
399                 return results;
400             }
401             if (sequence.Length == 2)
402             {
403                 var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
404                 if (doublet != Constants.Null)
405                 {
406                     results.Add(doublet);
407                 }
408                 return results;
409             }
410             var linksInSequence = new HashSet<ulong>(sequence);
411             void handler(IList<LinkIndex> result)
412             {
413                 var resultIndex = result[Links.Constants.IndexPart];
414                 var filterPosition = 0;
415                 StopableSequenceWalker.WalkRight(resultIndex, Links.Unsync.GetSource,
416                 ↪ Links.Unsync.GetTarget,
417                 ↪ x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x,
418                 ↪ x =>
419                 {
420                     if (filterPosition == sequence.Length)
421                     {
422                         filterPosition = -2; // Длиннее чем нужно
423                         return false;
424                     }
425                     if (x != sequence[filterPosition])
426                     {
427                         filterPosition = -1;
428                         return false; // Начинается иначе
429                     }
430                     filterPosition++;
431                     return true;
432                 });
433                 if (filterPosition == sequence.Length)
434                 {
435                     results.Add(resultIndex);
436                 }
437             }
438             if (sequence.Length >= 2)
439             {
440                 StepRight(handler, sequence[0], sequence[1]);
441             }
442             var last = sequence.Length - 2;
443             for (var i = 1; i < last; i++)
444             {
445                 PartialStepRight(handler, sequence[i], sequence[i + 1]);
446             }
447             if (sequence.Length >= 3)
448             {
449                 StepLeft(handler, sequence[sequence.Length - 2],
450                 ↪ sequence[sequence.Length - 1]);
451             }
452             return results;
453         });
454     }
455
456 [MethodImpl(MethodImplOptions.AggressiveInlining)]
457 public HashSet<ulong> GetAllMatchingSequences1(params ulong[] sequence)
458 {
459     return _sync.ExecuteReadOperation(() =>
460     {

```

```

461     var results = new HashSet<ulong>();
462     if (sequence.Length > 0)
463     {
464         Links.EnsureLinkExists(sequence);
465         var firstElement = sequence[0];
466         if (sequence.Length == 1)
467         {
468             results.Add(firstElement);
469             return results;
470         }
471         if (sequence.Length == 2)
472         {
473             var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
474             if (doublet != Constants.Null)
475             {
476                 results.Add(doublet);
477             }
478             return results;
479         }
480         var matcher = new Matcher(this, sequence, results, null);
481         if (sequence.Length >= 2)
482         {
483             StepRight(matcher.AddFullMatchedToResults, sequence[0], sequence[1]);
484         }
485         var last = sequence.Length - 2;
486         for (var i = 1; i < last; i++)
487         {
488             PartialStepRight(matcher.AddFullMatchedToResults, sequence[i],
489                             ↪ sequence[i + 1]);
490         }
491         if (sequence.Length >= 3)
492         {
493             StepLeft(matcher.AddFullMatchedToResults, sequence[sequence.Length - 2],
494                     ↪ sequence[sequence.Length - 1]);
495         }
496         return results;
497     }
498 }
499
500 public const int MaxSequenceFormatSize = 200;
501
502 [MethodImpl(MethodImplOptions.AggressiveInlining)]
503 public string FormatSequence(LinkIndex sequenceLink, params LinkIndex[] knownElements)
504     ↪ => FormatSequence(sequenceLink, (sb, x) => sb.Append(x), true, knownElements);
505
506 [MethodImpl(MethodImplOptions.AggressiveInlining)]
507 public string FormatSequence(LinkIndex sequenceLink, Action<StringBuilder, LinkIndex>
508     ↪ elementToString, bool insertComma, params LinkIndex[] knownElements) =>
509     ↪ Links.SyncRoot.ExecuteReadOperation(() => FormatSequence(Links.Unsync, sequenceLink,
510     ↪ elementToString, insertComma, knownElements));
511
512 [MethodImpl(MethodImplOptions.AggressiveInlining)]
513 private string FormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
514     ↪ Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
515     ↪ LinkIndex[] knownElements)
516 {
517     var linksInSequence = new HashSet<ulong>(knownElements);
518     //var entered = new HashSet<ulong>();
519     var sb = new StringBuilder();
520     sb.Append('{');
521     if (links.Exists(sequenceLink))
522     {
523         StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
524             x => linksInSequence.Contains(x) || links.IsPartialPoint(x), element => //
525             ↪ entered.AddAndReturnVoid, x => { }, entered.DoNotContains
526         {
527             if (insertComma && sb.Length > 1)
528             {
529                 sb.Append(',');
530             }
531             //if (entered.Contains(element))
532             //{
533             //    sb.Append('{');
534             //    elementToString(sb, element);
535             //    sb.Append('}');
536             //}
537             //else

```

```

530         elementToString(sb, element);
531         if (sb.Length < MaxSequenceFormatSize)
532         {
533             return true;
534         }
535         sb.Append(insertComma ? ", ..." : "...");
536         return false;
537     });
538 }
539 sb.Append('}');
540 return sb.ToString();
541 }
542
543 [MethodImpl(MethodImplOptions.AggressiveInlining)]
544 public string SafeFormatSequence(LinkIndex sequenceLink, params LinkIndex[]
    ↪ knownElements) => SafeFormatSequence(sequenceLink, (sb, x) => sb.Append(x), true,
    ↪ knownElements);
545
546 [MethodImpl(MethodImplOptions.AggressiveInlining)]
547 public string SafeFormatSequence(LinkIndex sequenceLink, Action<StringBuilder,
    ↪ LinkIndex> elementToString, bool insertComma, params LinkIndex[] knownElements) =>
    ↪ Links.SyncRoot.ExecuteReadOperation(() => SafeFormatSequence(Links.Unsync,
    ↪ sequenceLink, elementToString, insertComma, knownElements));
548
549 [MethodImpl(MethodImplOptions.AggressiveInlining)]
550 private string SafeFormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
    ↪ Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
    ↪ LinkIndex[] knownElements)
551 {
552     var linksInSequence = new HashSet<ulong>(knownElements);
553     var entered = new HashSet<ulong>();
554     var sb = new StringBuilder();
555     sb.Append('{');
556     if (links.Exists(sequenceLink))
557     {
558         StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
559             x => linksInSequence.Contains(x) || links.IsFullPoint(x),
    ↪ entered.AddAndReturnVoid, x => { }, entered.DoNotContains, element =>
560         {
561             if (insertComma && sb.Length > 1)
562             {
563                 sb.Append(',');
564             }
565             if (entered.Contains(element))
566             {
567                 sb.Append('{');
568                 elementToString(sb, element);
569                 sb.Append('}');
570             }
571             else
572             {
573                 elementToString(sb, element);
574             }
575             if (sb.Length < MaxSequenceFormatSize)
576             {
577                 return true;
578             }
579             sb.Append(insertComma ? ", ..." : "...");
580             return false;
581         });
582     }
583     sb.Append('}');
584     return sb.ToString();
585 }
586
587 [MethodImpl(MethodImplOptions.AggressiveInlining)]
588 public List<ulong> GetAllPartiallyMatchingSequences0(params ulong[] sequence)
589 {
590     return _sync.ExecuteReadOperation(() =>
591     {
592         if (sequence.Length > 0)
593         {
594             Links.EnsureLinkExists(sequence);
595             var results = new HashSet<ulong>();
596             for (var i = 0; i < sequence.Length; i++)
597             {
598                 AllUsagesCore(sequence[i], results);
599             }
600         }
601     });
602 }

```

```

600     var filteredResults = new List<ulong>();
601     var linksInSequence = new HashSet<ulong>(sequence);
602     foreach (var result in results)
603     {
604         var filterPosition = -1;
605         StopableSequenceWalker.WalkRight(result, Links.Unsync.GetSource,
        ↪ Links.Unsync.GetTarget,
        ↪ x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x,
        ↪ x =>
        {
607             if (filterPosition == (sequence.Length - 1))
608             {
609                 return false;
610             }
611             if (filterPosition >= 0)
612             {
613                 if (x == sequence[filterPosition + 1])
614                 {
615                     filterPosition++;
616                 }
617                 else
618                 {
619                     return false;
620                 }
621             }
622             if (filterPosition < 0)
623             {
624                 if (x == sequence[0])
625                 {
626                     filterPosition = 0;
627                 }
628             }
629             return true;
630         });
631         if (filterPosition == (sequence.Length - 1))
632         {
633             filteredResults.Add(result);
634         }
635     }
636     return filteredResults;
637 }
638 return new List<ulong>();
639 });
640 }
641 }
642
643 [MethodImpl(MethodImplOptions.AggressiveInlining)]
644 public HashSet<ulong> GetAllPartiallyMatchingSequences1(params ulong[] sequence)
645 {
646     return _sync.ExecuteReadOperation(() =>
647     {
648         if (sequence.Length > 0)
649         {
650             Links.EnsureLinkExists(sequence);
651             var results = new HashSet<ulong>();
652             for (var i = 0; i < sequence.Length; i++)
653             {
654                 AllUsagesCore(sequence[i], results);
655             }
656             var filteredResults = new HashSet<ulong>();
657             var matcher = new Matcher(this, sequence, filteredResults, null);
658             matcher.AddAllPartialMatchedToResults(results);
659             return filteredResults;
660         }
661         return new HashSet<ulong>();
662     });
663 }
664
665 [MethodImpl(MethodImplOptions.AggressiveInlining)]
666 public bool GetAllPartiallyMatchingSequences2(Func<IList<LinkIndex>, LinkIndex> handler,
667 ↪ params ulong[] sequence)
668 {
669     return _sync.ExecuteReadOperation(() =>
670     {
671         if (sequence.Length > 0)
672         {
673             Links.EnsureLinkExists(sequence);
674
675             var results = new HashSet<ulong>();
676             var filteredResults = new HashSet<ulong>();

```

```

676         var matcher = new Matcher(this, sequence, filteredResults, handler);
677         for (var i = 0; i < sequence.Length; i++)
678         {
679             if (!AllUsagesCore1(sequence[i], results, matcher.HandlePartialMatched))
680             {
681                 return false;
682             }
683         }
684         return true;
685     }
686     return true;
687 });
688 }
689
690 //public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
691 //{
692 //    return Sync.ExecuteReadOperation(() =>
693 //    {
694 //        if (sequence.Length > 0)
695 //        {
696 //            _links.EnsureEachLinkIsAnyOrExists(sequence);
697 //
698 //            var firstResults = new HashSet<ulong>();
699 //            var lastResults = new HashSet<ulong>();
700 //
701 //            var first = sequence.First(x => x != LinksConstants.Any);
702 //            var last = sequence.Last(x => x != LinksConstants.Any);
703 //
704 //            AllUsagesCore(first, firstResults);
705 //            AllUsagesCore(last, lastResults);
706 //
707 //            firstResults.IntersectWith(lastResults);
708 //
709 //            //for (var i = 0; i < sequence.Length; i++)
710 //            //    AllUsagesCore(sequence[i], results);
711 //
712 //            var filteredResults = new HashSet<ulong>();
713 //            var matcher = new Matcher(this, sequence, filteredResults, null);
714 //            matcher.AddAllPartialMatchedToResults(firstResults);
715 //            return filteredResults;
716 //        }
717 //
718 //        return new HashSet<ulong>();
719 //    });
720 //}
721
722 [MethodImpl(MethodImplOptions.AggressiveInlining)]
723 public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
724 {
725     return _sync.ExecuteReadOperation(() =>
726     {
727         if (sequence.Length > 0)
728         {
729             ILinksExtensions.EnsureLinkIsAnyOrExists(Links, sequence);
730             var firstResults = new HashSet<ulong>();
731             var lastResults = new HashSet<ulong>();
732             var first = sequence.First(x => x != Constants.Any);
733             var last = sequence.Last(x => x != Constants.Any);
734             AllUsagesCore(first, firstResults);
735             AllUsagesCore(last, lastResults);
736             firstResults.IntersectWith(lastResults);
737             //for (var i = 0; i < sequence.Length; i++)
738             //    AllUsagesCore(sequence[i], results);
739             var filteredResults = new HashSet<ulong>();
740             var matcher = new Matcher(this, sequence, filteredResults, null);
741             matcher.AddAllPartialMatchedToResults(firstResults);
742             return filteredResults;
743         }
744         return new HashSet<ulong>();
745     });
746 }
747
748 [MethodImpl(MethodImplOptions.AggressiveInlining)]
749 public HashSet<ulong> GetAllPartiallyMatchingSequences4(HashSet<ulong> readAsElements,
750     ↪ IList<ulong> sequence)
751 {
752     return _sync.ExecuteReadOperation(() =>
753     {
754         if (sequence.Count > 0)

```



```

754 {
755     Links.EnsureLinkExists(sequence);
756     var results = new HashSet<LinkIndex>();
757     //var nextResults = new HashSet<ulong>();
758     //for (var i = 0; i < sequence.Length; i++)
759     //{
760         //    AllUsagesCore(sequence[i], nextResults);
761         //    if (results.IsNullOrEmpty())
762         //    {
763             //        results = nextResults;
764             //        nextResults = new HashSet<ulong>();
765         //    }
766         //    else
767         //    {
768             //        results.IntersectWith(nextResults);
769             //        nextResults.Clear();
770         //    }
771     //}
772     var collector1 = new AllUsagesCollector1(Links.Unsync, results);
773     collector1.Collect(Links.Unsync.GetLink(sequence[0]));
774     var next = new HashSet<ulong>();
775     for (var i = 1; i < sequence.Count; i++)
776     {
777         var collector = new AllUsagesCollector1(Links.Unsync, next);
778         collector.Collect(Links.Unsync.GetLink(sequence[i]));
779
780         results.IntersectWith(next);
781         next.Clear();
782     }
783     var filteredResults = new HashSet<ulong>();
784     var matcher = new Matcher(this, sequence, filteredResults, null,
785         ↪ readAsElements);
786     matcher.AddAllPartialMatchedToResultsAndReadAsElements(results.OrderBy(x =>
787         ↪ x)); // OrderBy is a Hack
788     return filteredResults;
789 }
790 });
791 }
792 // Does not work
793 //public HashSet<ulong> GetAllPartiallyMatchingSequences5(HashSet<ulong> readAsElements,
794     ↪ params ulong[] sequence)
795 //{
796     //    var visited = new HashSet<ulong>();
797     //    var results = new HashSet<ulong>();
798     //    var matcher = new Matcher(this, sequence, visited, x => { results.Add(x); return
799     ↪ true; }, readAsElements);
800     //    var last = sequence.Length - 1;
801     //    for (var i = 0; i < last; i++)
802     //    {
803         //        PartialStepRight(matcher.PartialMatch, sequence[i], sequence[i + 1]);
804     //    }
805     //    return results;
806 //}
807 [MethodImpl(MethodImplOptions.AggressiveInlining)]
808 public List<ulong> GetAllPartiallyMatchingSequences(params ulong[] sequence)
809 {
810     return _sync.ExecuteReadOperation(() =>
811     {
812         if (sequence.Length > 0)
813         {
814             Links.EnsureLinkExists(sequence);
815             //var firstElement = sequence[0];
816             //if (sequence.Length == 1)
817             //{
818                 //results.Add(firstElement);
819                 //return results;
820             //}
821             //if (sequence.Length == 2)
822             //{
823                 //var doublet = _links.SearchCore(firstElement, sequence[1]);
824                 //if (doublet != Doublets.Links.Null)
825                 //    results.Add(doublet);
826                 //return results;
827             //}
828             //var lastElement = sequence[sequence.Length - 1];

```

```

828 //Func<ulong, bool> handler = x =>
829 //{
830 //    if (StartsWith(x, firstElement) && EndsWith(x, lastElement))
831 //        results.Add(x);
832 //    return true;
833 //};
834 //if (sequence.Length >= 2)
835 //    StepRight(handler, sequence[0], sequence[1]);
836 //var last = sequence.Length - 2;
837 //for (var i = 1; i < last; i++)
838 //    PartialStepRight(handler, sequence[i], sequence[i + 1]);
839 //if (sequence.Length >= 3)
840 //    StepLeft(handler, sequence[sequence.Length - 2],
841 //        sequence[sequence.Length - 1]);
842 //if (sequence.Length == 1)
843 //    throw new NotImplementedException(); // all sequences, containing
844 //    this element?
845 //if (sequence.Length == 2)
846 //    {
847 //        var results = new List<ulong>();
848 //        PartialStepRight(results.Add, sequence[0], sequence[1]);
849 //        return results;
850 //    }
851 //var matches = new List<List<ulong>>();
852 //var last = sequence.Length - 1;
853 //for (var i = 0; i < last; i++)
854 //    {
855 //        var results = new List<ulong>();
856 //        //StepRight(results.Add, sequence[i], sequence[i + 1]);
857 //        PartialStepRight(results.Add, sequence[i], sequence[i + 1]);
858 //        if (results.Count > 0)
859 //            matches.Add(results);
860 //        else
861 //            return results;
862 //        if (matches.Count == 2)
863 //            {
864 //                var merged = new List<ulong>();
865 //                for (var j = 0; j < matches[0].Count; j++)
866 //                    for (var k = 0; k < matches[1].Count; k++)
867 //                        CloseInnerConnections(merged.Add, matches[0][j],
868 //                            matches[1][k]);
869 //                if (merged.Count > 0)
870 //                    matches = new List<List<ulong>> { merged };
871 //                else
872 //                    return new List<ulong>();
873 //            }
874 //    }
875 //if (matches.Count > 0)
876 //    {
877 //        var usages = new HashSet<ulong>();
878 //        for (int i = 0; i < sequence.Length; i++)
879 //            {
880 //                AllUsagesCore(sequence[i], usages);
881 //            }
882 //        //for (int i = 0; i < matches[0].Count; i++)
883 //        //    AllUsagesCore(matches[0][i], usages);
884 //        //usages.UnionWith(matches[0]);
885 //        return usages.ToList();
886 //    }
887 var firstLinkUsages = new HashSet<ulong>();
888 AllUsagesCore(sequence[0], firstLinkUsages);
889 firstLinkUsages.Add(sequence[0]);
890 //var previousMatchings = firstLinkUsages.ToList(); //new List<ulong>() {
891 //    sequence[0] }; // or all sequences, containing this element?
892 //return GetAllPartiallyMatchingSequencesCore(sequence, firstLinkUsages,
893 //    1).ToList();
894 var results = new HashSet<ulong>();
895 foreach (var match in GetAllPartiallyMatchingSequencesCore(sequence,
896 //    firstLinkUsages, 1))
897 {
898     AllUsagesCore(match, results);
899 }
900 return results.ToList();
901 }
902 return new List<ulong>();
903

```

```

898     });
899 }
900
901 /// <remarks>
902 /// TODO: Может потребоваться ограничение на уровень глубины рекурсии
903 /// </remarks>
904 [MethodImpl(MethodImplOptions.AggressiveInlining)]
905 public HashSet<ulong> AllUsages(ulong link)
906 {
907     return _sync.ExecuteReadOperation(() =>
908     {
909         var usages = new HashSet<ulong>();
910         AllUsagesCore(link, usages);
911         return usages;
912     });
913 }
914
915 // При сборе всех использований (последовательностей) можно сохранять обратный путь к
916 //   ↳ той связи с которой начинался поиск (STTTSSSTT),
917 // причём достаточно одного бита для хранения перехода влево или вправо
918 [MethodImpl(MethodImplOptions.AggressiveInlining)]
919 private void AllUsagesCore(ulong link, HashSet<ulong> usages)
920 {
921     bool handler(ulong doublet)
922     {
923         if (usages.Add(doublet))
924         {
925             AllUsagesCore(doublet, usages);
926         }
927         return true;
928     }
929     Links.Unsync.Each(link, Constants.Any, handler);
930     Links.Unsync.Each(Constants.Any, link, handler);
931 }
932
933 [MethodImpl(MethodImplOptions.AggressiveInlining)]
934 public HashSet<ulong> AllBottomUsages(ulong link)
935 {
936     return _sync.ExecuteReadOperation(() =>
937     {
938         var visits = new HashSet<ulong>();
939         var usages = new HashSet<ulong>();
940         AllBottomUsagesCore(link, visits, usages);
941         return usages;
942     });
943 }
944
945 [MethodImpl(MethodImplOptions.AggressiveInlining)]
946 private void AllBottomUsagesCore(ulong link, HashSet<ulong> visits, HashSet<ulong>
947   ↳ usages)
948 {
949     bool handler(ulong doublet)
950     {
951         if (visits.Add(doublet))
952         {
953             AllBottomUsagesCore(doublet, visits, usages);
954         }
955         return true;
956     }
957     if (Links.Unsync.Count(Constants.Any, link) == 0)
958     {
959         usages.Add(link);
960     }
961     else
962     {
963         Links.Unsync.Each(link, Constants.Any, handler);
964         Links.Unsync.Each(Constants.Any, link, handler);
965     }
966 }
967
968 [MethodImpl(MethodImplOptions.AggressiveInlining)]
969 public ulong CalculateTotalSymbolFrequencyCore(ulong symbol)
970 {
971     if (Options.UseSequenceMarker)
972     {
973         var counter = new TotalMarkedSequenceSymbolFrequencyOneOffCounter<ulong>(Links,
974   ↳ Options.MarkedSequenceMatcher, symbol);
975         return counter.Count();
976     }
977 }

```

```

973     }
974     else
975     {
976         var counter = new TotalSequenceSymbolFrequencyOneOffCounter<ulong>(Links,
977             ↪ symbol);
978         return counter.Count();
979     }
980 }
981 [MethodImpl(MethodImplOptions.AggressiveInlining)]
982 private bool AllUsagesCore1(ulong link, HashSet<ulong> usages, Func<IList<LinkIndex>,
983     ↪ LinkIndex> outerHandler)
984 {
985     bool handler(ulong doublet)
986     {
987         if (usages.Add(doublet))
988         {
989             if (outerHandler(new LinkAddress<LinkIndex>(doublet)) != Constants.Continue)
990             {
991                 return false;
992             }
993             if (!AllUsagesCore1(doublet, usages, outerHandler))
994             {
995                 return false;
996             }
997         }
998         return true;
999     }
1000     return Links.Unsync.Each(link, Constants.Any, handler)
1001         && Links.Unsync.Each(Constants.Any, link, handler);
1002 }
1003 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1004 public void CalculateAllUsages(ulong[] totals)
1005 {
1006     var calculator = new AllUsagesCalculator(Links, totals);
1007     calculator.Calculate();
1008 }
1009 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1010 public void CalculateAllUsages2(ulong[] totals)
1011 {
1012     var calculator = new AllUsagesCalculator2(Links, totals);
1013     calculator.Calculate();
1014 }
1015 }
1016 private class AllUsagesCalculator
1017 {
1018     private readonly SynchronizedLinks<ulong> _links;
1019     private readonly ulong[] _totals;
1020
1021     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1022     public AllUsagesCalculator(SynchronizedLinks<ulong> links, ulong[] totals)
1023     {
1024         _links = links;
1025         _totals = totals;
1026     }
1027
1028     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1029     public void Calculate() => _links.Each(_links.Constants.Any, _links.Constants.Any,
1030         ↪ CalculateCore);
1031
1032     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1033     private bool CalculateCore(ulong link)
1034     {
1035         if (_totals[link] == 0)
1036         {
1037             var total = 1UL;
1038             _totals[link] = total;
1039             var visitedChildren = new HashSet<ulong>();
1040             bool linkCalculator(ulong child)
1041             {
1042                 if (link != child && visitedChildren.Add(child))
1043                 {
1044                     total += _totals[child] == 0 ? 1 : _totals[child];
1045                 }
1046                 return true;
1047             }
1048             _links.Unsync.Each(link, _links.Constants.Any, linkCalculator);

```

```

1049         _links.Unsync.Each(_links.Constants.Any, link, linkCalculator);
1050         _totals[link] = total;
1051     }
1052     return true;
1053 }
1054 }
1055
1056 private class AllUsagesCalculator2
1057 {
1058     private readonly SynchronizedLinks<ulong> _links;
1059     private readonly ulong[] _totals;
1060
1061     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1062     public AllUsagesCalculator2(SynchronizedLinks<ulong> links, ulong[] totals)
1063     {
1064         _links = links;
1065         _totals = totals;
1066     }
1067
1068     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1069     public void Calculate() => _links.Each(_links.Constants.Any, _links.Constants.Any,
1070         ↪ CalculateCore);
1071
1072     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1073     private bool IsElement(ulong link)
1074     {
1075         // _linksInSequence.Contains(link) ||
1076         return _links.Unsync.GetTarget(link) == link || _links.Unsync.GetSource(link) ==
1077             ↪ link;
1078     }
1079
1080     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1081     private bool CalculateCore(ulong link)
1082     {
1083         // TODO: Проработать защиту от заикливания
1084         // Основано на SequenceWalker.WalkLeft
1085         Func<ulong, ulong> getSource = _links.Unsync.GetSource;
1086         Func<ulong, ulong> getTarget = _links.Unsync.GetTarget;
1087         Func<ulong, bool> isElement = IsElement;
1088         void visitLeaf(ulong parent)
1089         {
1090             if (link != parent)
1091             {
1092                 _totals[parent]++;
1093             }
1094         }
1095         void visitNode(ulong parent)
1096         {
1097             if (link != parent)
1098             {
1099                 _totals[parent]++;
1100             }
1101         }
1102         var stack = new Stack();
1103         var element = link;
1104         if (isElement(element))
1105         {
1106             visitLeaf(element);
1107         }
1108         else
1109         {
1110             while (true)
1111             {
1112                 if (isElement(element))
1113                 {
1114                     if (stack.Count == 0)
1115                     {
1116                         break;
1117                     }
1118                     element = stack.Pop();
1119                     var source = getSource(element);
1120                     var target = getTarget(element);
1121                     // Обработка элемента
1122                     if (isElement(target))
1123                     {
1124                         visitLeaf(target);
1125                     }
1126                     if (isElement(source))
1127                     {

```

```

1126         visitLeaf(source);
1127     }
1128     element = source;
1129 }
1130 else
1131 {
1132     stack.Push(element);
1133     visitNode(element);
1134     element = getTarget(element);
1135 }
1136 }
1137 }
1138 _totals[link]++;
1139 return true;
1140 }
1141 }
1142
1143 private class AllUsagesCollector
1144 {
1145     private readonly ILinks<ulong> _links;
1146     private readonly HashSet<ulong> _usages;
1147
1148     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1149     public AllUsagesCollector(ILinks<ulong> links, HashSet<ulong> usages)
1150     {
1151         _links = links;
1152         _usages = usages;
1153     }
1154
1155     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1156     public bool Collect(ulong link)
1157     {
1158         if (_usages.Add(link))
1159         {
1160             _links.Each(link, _links.Constants.Any, Collect);
1161             _links.Each(_links.Constants.Any, link, Collect);
1162         }
1163         return true;
1164     }
1165 }
1166
1167 private class AllUsagesCollector1
1168 {
1169     private readonly ILinks<ulong> _links;
1170     private readonly HashSet<ulong> _usages;
1171     private readonly ulong _continue;
1172
1173     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1174     public AllUsagesCollector1(ILinks<ulong> links, HashSet<ulong> usages)
1175     {
1176         _links = links;
1177         _usages = usages;
1178         _continue = _links.Constants.Continue;
1179     }
1180
1181     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1182     public ulong Collect(ICollection<ulong> link)
1183     {
1184         var linkIndex = _links.GetIndex(link);
1185         if (_usages.Add(linkIndex))
1186         {
1187             _links.Each(Collect, _links.Constants.Any, linkIndex);
1188         }
1189         return _continue;
1190     }
1191 }
1192
1193 private class AllUsagesCollector2
1194 {
1195     private readonly ILinks<ulong> _links;
1196     private readonly BitString _usages;
1197
1198     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1199     public AllUsagesCollector2(ILinks<ulong> links, BitString usages)
1200     {
1201         _links = links;
1202         _usages = usages;
1203     }
1204
1205     [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

1206     public bool Collect(ulong link)
1207     {
1208         if (_usages.Add((long)link))
1209         {
1210             _links.Each(link, _links.Constants.Any, Collect);
1211             _links.Each(_links.Constants.Any, link, Collect);
1212         }
1213         return true;
1214     }
1215 }
1216
1217 private class AllUsagesIntersectingCollector
1218 {
1219     private readonly SynchronizedLinks<ulong> _links;
1220     private readonly HashSet<ulong> _intersectWith;
1221     private readonly HashSet<ulong> _usages;
1222     private readonly HashSet<ulong> _enter;
1223
1224     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1225     public AllUsagesIntersectingCollector(SynchronizedLinks<ulong> links, HashSet<ulong>
        ↪ intersectWith, HashSet<ulong> usages)
1226     {
1227         _links = links;
1228         _intersectWith = intersectWith;
1229         _usages = usages;
1230         _enter = new HashSet<ulong>(); // защита от зацикливания
1231     }
1232
1233     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1234     public bool Collect(ulong link)
1235     {
1236         if (_enter.Add(link))
1237         {
1238             if (_intersectWith.Contains(link))
1239             {
1240                 _usages.Add(link);
1241             }
1242             _links.Unsync.Each(link, _links.Constants.Any, Collect);
1243             _links.Unsync.Each(_links.Constants.Any, link, Collect);
1244         }
1245         return true;
1246     }
1247 }
1248
1249 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1250 private void CloseInnerConnections(Action<IList<LinkIndex>> handler, ulong left, ulong
        ↪ right)
1251 {
1252     TryStepLeftUp(handler, left, right);
1253     TryStepRightUp(handler, right, left);
1254 }
1255
1256 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1257 private void AllCloseConnections(Action<IList<LinkIndex>> handler, ulong left, ulong
        ↪ right)
1258 {
1259     // Direct
1260     if (left == right)
1261     {
1262         handler(new LinkAddress<LinkIndex>(left));
1263     }
1264     var doublet = Links.Unsync.SearchOrDefault(left, right);
1265     if (doublet != Constants.Null)
1266     {
1267         handler(new LinkAddress<LinkIndex>(doublet));
1268     }
1269     // Inner
1270     CloseInnerConnections(handler, left, right);
1271     // Outer
1272     StepLeft(handler, left, right);
1273     StepRight(handler, left, right);
1274     PartialStepRight(handler, left, right);
1275     PartialStepLeft(handler, left, right);
1276 }
1277
1278 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1279 private HashSet<ulong> GetAllPartiallyMatchingSequencesCore(ulong[] sequence,
        ↪ HashSet<ulong> previousMatchings, long startAt)
1280 {

```

```

1281     if (startAt >= sequence.Length) // ?
1282     {
1283         return previousMatchings;
1284     }
1285     var secondLinkUsages = new HashSet<ulong>();
1286     AllUsagesCore(sequence[startAt], secondLinkUsages);
1287     secondLinkUsages.Add(sequence[startAt]);
1288     var matchings = new HashSet<ulong>();
1289     var filler = new SetFiller<LinkIndex, LinkIndex>(matchings, Constants.Continue);
1290     //for (var i = 0; i < previousMatchings.Count; i++)
1291     foreach (var secondLinkUsage in secondLinkUsages)
1292     {
1293         foreach (var previousMatching in previousMatchings)
1294         {
1295             //AllCloseConnections(matchings.AddAndReturnVoid, previousMatching,
1296             //    ↪ secondLinkUsage);
1297             StepRight(filler.AddFirstAndReturnConstant, previousMatching,
1298             //    ↪ secondLinkUsage);
1299             TryStepRightUp(filler.AddFirstAndReturnConstant, secondLinkUsage,
1300             //    ↪ previousMatching);
1301             //PartialStepRight(matchings.AddAndReturnVoid, secondLinkUsage,
1302             //    ↪ sequence[startAt]); // почему-то эта ошибочная запись приводит к
1303             //    ↪ желаемым результатам.
1304             PartialStepRight(filler.AddFirstAndReturnConstant, previousMatching,
1305             //    ↪ secondLinkUsage);
1306         }
1307     }
1308     if (matchings.Count == 0)
1309     {
1310         return matchings;
1311     }
1312     return GetAllPartiallyMatchingSequencesCore(sequence, matchings, startAt + 1); // ??
1313 }
1314
1315 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1316 private static void EnsureEachLinkIsAnyOrZeroOrManyOrExists(SynchronizedLinks<ulong>
1317     ↪ links, params ulong[] sequence)
1318 {
1319     if (sequence == null)
1320     {
1321         return;
1322     }
1323     for (var i = 0; i < sequence.Length; i++)
1324     {
1325         if (sequence[i] != links.Constants.Any && sequence[i] != ZeroOrMany &&
1326             ↪ !links.Exists(sequence[i]))
1327         {
1328             throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
1329             ↪ $"patternSequence[{i}]");
1330         }
1331     }
1332 }
1333
1334 // Pattern Matching -> Key To Triggers
1335 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1336 public HashSet<ulong> MatchPattern(params ulong[] patternSequence)
1337 {
1338     return _sync.ExecuteReadOperation(() =>
1339     {
1340         patternSequence = Simplify(patternSequence);
1341         if (patternSequence.Length > 0)
1342         {
1343             EnsureEachLinkIsAnyOrZeroOrManyOrExists(Links, patternSequence);
1344             var uniqueSequenceElements = new HashSet<ulong>();
1345             for (var i = 0; i < patternSequence.Length; i++)
1346             {
1347                 if (patternSequence[i] != Constants.Any && patternSequence[i] !=
1348                     ↪ ZeroOrMany)
1349                 {
1350                     uniqueSequenceElements.Add(patternSequence[i]);
1351                 }
1352             }
1353             var results = new HashSet<ulong>();
1354             foreach (var uniqueSequenceElement in uniqueSequenceElements)
1355             {
1356                 AllUsagesCore(uniqueSequenceElement, results);
1357             }
1358         }
1359     });
1360 }

```



```

1348         var filteredResults = new HashSet<ulong>();
1349         var matcher = new PatternMatcher(this, patternSequence, filteredResults);
1350         matcher.AddAllPatternMatchedToResults(results);
1351         return filteredResults;
1352     }
1353     return new HashSet<ulong>();
1354 });
1355 }
1356
1357 // Найти все возможные связи между указанным списком связей.
1358 // Находит связи между всеми указанными связями в любом порядке.
1359 // TODO: решить что делать с повторами (когда одни и те же элементы встречаются
1360 // → несколько раз в последовательности)
1361 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1362 public HashSet<ulong> GetAllConnections(params ulong[] linksToConnect)
1363 {
1364     return _sync.ExecuteReadOperation(() =>
1365     {
1366         var results = new HashSet<ulong>();
1367         if (linksToConnect.Length > 0)
1368         {
1369             Links.EnsureLinkExists(linksToConnect);
1370             AllUsagesCore(linksToConnect[0], results);
1371             for (var i = 1; i < linksToConnect.Length; i++)
1372             {
1373                 var next = new HashSet<ulong>();
1374                 AllUsagesCore(linksToConnect[i], next);
1375                 results.IntersectWith(next);
1376             }
1377             return results;
1378         }
1379     });
1380 }
1381 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1382 public HashSet<ulong> GetAllConnections1(params ulong[] linksToConnect)
1383 {
1384     return _sync.ExecuteReadOperation(() =>
1385     {
1386         var results = new HashSet<ulong>();
1387         if (linksToConnect.Length > 0)
1388         {
1389             Links.EnsureLinkExists(linksToConnect);
1390             var collector1 = new AllUsagesCollector(Links.Unsync, results);
1391             collector1.Collect(linksToConnect[0]);
1392             var next = new HashSet<ulong>();
1393             for (var i = 1; i < linksToConnect.Length; i++)
1394             {
1395                 var collector = new AllUsagesCollector(Links.Unsync, next);
1396                 collector.Collect(linksToConnect[i]);
1397                 results.IntersectWith(next);
1398                 next.Clear();
1399             }
1400             return results;
1401         }
1402     });
1403 }
1404 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1405 public HashSet<ulong> GetAllConnections2(params ulong[] linksToConnect)
1406 {
1407     return _sync.ExecuteReadOperation(() =>
1408     {
1409         var results = new HashSet<ulong>();
1410         if (linksToConnect.Length > 0)
1411         {
1412             Links.EnsureLinkExists(linksToConnect);
1413             var collector1 = new AllUsagesCollector(Links, results);
1414             collector1.Collect(linksToConnect[0]);
1415             //AllUsagesCore(linksToConnect[0], results);
1416             for (var i = 1; i < linksToConnect.Length; i++)
1417             {
1418                 var next = new HashSet<ulong>();
1419                 var collector = new AllUsagesIntersectingCollector(Links, results, next);
1420                 collector.Collect(linksToConnect[i]);
1421                 //AllUsagesCore(linksToConnect[i], next);
1422                 //results.IntersectWith(next);
1423                 results = next;
1424             }
1425         }
1426     });
1427 }

```

```

1425     }
1426     }
1427     return results;
1428 });
1429 }
1430
1431 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1432 public List<ulong> GetAllConnections3(params ulong[] linksToConnect)
1433 {
1434     return _sync.ExecuteReadOperation(() =>
1435     {
1436         var results = new BitString((long)Links.Unsync.Count() + 1); // new
1437         ↪ BitArray((int)_links.Total + 1);
1438         if (linksToConnect.Length > 0)
1439         {
1440             Links.EnsureLinkExists(linksToConnect);
1441             var collector1 = new AllUsagesCollector2(Links.Unsync, results);
1442             collector1.Collect(linksToConnect[0]);
1443             for (var i = 1; i < linksToConnect.Length; i++)
1444             {
1445                 var next = new BitString((long)Links.Unsync.Count() + 1); //new
1446                 ↪ BitArray((int)_links.Total + 1);
1447                 var collector = new AllUsagesCollector2(Links.Unsync, next);
1448                 collector.Collect(linksToConnect[i]);
1449                 results = results.And(next);
1450             }
1451             return results.GetSetUInt64Indices();
1452         }
1453     });
1454 }
1455
1456 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1457 private static ulong[] Simplify(ulong[] sequence)
1458 {
1459     // Считаем новый размер последовательности
1460     long newLength = 0;
1461     var zeroOrManyStepped = false;
1462     for (var i = 0; i < sequence.Length; i++)
1463     {
1464         if (sequence[i] == ZeroOrMany)
1465         {
1466             if (zeroOrManyStepped)
1467             {
1468                 continue;
1469             }
1470             zeroOrManyStepped = true;
1471         }
1472         else
1473         {
1474             //if (zeroOrManyStepped) Is it efficient?
1475             zeroOrManyStepped = false;
1476         }
1477         newLength++;
1478     }
1479     // Строим новую последовательность
1480     zeroOrManyStepped = false;
1481     var newSequence = new ulong[newLength];
1482     long j = 0;
1483     for (var i = 0; i < sequence.Length; i++)
1484     {
1485         //var current = zeroOrManyStepped;
1486         //zeroOrManyStepped = patternSequence[i] == zeroOrMany;
1487         //if (current && zeroOrManyStepped)
1488         //    continue;
1489         //var newZeroOrManyStepped = patternSequence[i] == zeroOrMany;
1490         //if (zeroOrManyStepped && newZeroOrManyStepped)
1491         //    continue;
1492         //zeroOrManyStepped = newZeroOrManyStepped;
1493         if (sequence[i] == ZeroOrMany)
1494         {
1495             if (zeroOrManyStepped)
1496             {
1497                 continue;
1498             }
1499             zeroOrManyStepped = true;
1500         }
1501         else
1502         {
1503             //if (zeroOrManyStepped) Is it efficient?

```

```

1502         zeroOrManyStepped = false;
1503     }
1504     newSequence[j++] = sequence[i];
1505 }
1506 return newSequence;
1507 }
1508
1509 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1510 public static void TestSimplify()
1511 {
1512     var sequence = new ulong[] { ZeroOrMany, ZeroOrMany, 2, 3, 4, ZeroOrMany,
1513     ↪ ZeroOrMany, ZeroOrMany, 4, ZeroOrMany, ZeroOrMany, ZeroOrMany };
1514     var simplifiedSequence = Simplify(sequence);
1515 }
1516
1517 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1518 public List<ulong> GetSimilarSequences() => new List<ulong>();
1519
1520 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1521 public void Prediction()
1522 {
1523     //_links
1524     //_sequences
1525 }
1526
1527 #region From Triplets
1528
1529 //public static void DeleteSequence(Link sequence)
1530 //{
1531 //}
1532
1533 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1534 public List<ulong> CollectMatchingSequences(ulong[] links)
1535 {
1536     if (links.Length == 1)
1537     {
1538         throw new InvalidOperationException("Подпоследовательности с одним элементом не
1539         ↪ поддерживаются.");
1540     }
1541     var leftBound = 0;
1542     var rightBound = links.Length - 1;
1543     var left = links[leftBound++];
1544     var right = links[rightBound--];
1545     var results = new List<ulong>();
1546     CollectMatchingSequences(left, leftBound, links, right, rightBound, ref results);
1547     return results;
1548 }
1549
1550 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1551 private void CollectMatchingSequences(ulong leftLink, int leftBound, ulong[]
1552 ↪ middleLinks, ulong rightLink, int rightBound, ref List<ulong> results)
1553 {
1554     var leftLinkTotalReferers = Links.Unsync.Count(leftLink);
1555     var rightLinkTotalReferers = Links.Unsync.Count(rightLink);
1556     if (leftLinkTotalReferers <= rightLinkTotalReferers)
1557     {
1558         var nextLeftLink = middleLinks[leftBound];
1559         var elements = GetRightElements(leftLink, nextLeftLink);
1560         if (leftBound <= rightBound)
1561         {
1562             for (var i = elements.Length - 1; i >= 0; i--)
1563             {
1564                 var element = elements[i];
1565                 if (element != 0)
1566                 {
1567                     CollectMatchingSequences(element, leftBound + 1, middleLinks,
1568                     ↪ rightLink, rightBound, ref results);
1569                 }
1570             }
1571         }
1572     }
1573     else
1574     {
1575         for (var i = elements.Length - 1; i >= 0; i--)
1576         {
1577             var element = elements[i];
1578             if (element != 0)
1579             {
1580                 results.Add(element);
1581             }
1582         }
1583     }
1584 }

```

```

1576     }
1577 }
1578 }
1579 }
1580 else
1581 {
1582     var nextRightLink = middleLinks[rightBound];
1583     var elements = GetLeftElements(rightLink, nextRightLink);
1584     if (leftBound <= rightBound)
1585     {
1586         for (var i = elements.Length - 1; i >= 0; i--)
1587         {
1588             var element = elements[i];
1589             if (element != 0)
1590             {
1591                 CollectMatchingSequences(leftLink, leftBound, middleLinks,
1592                                         ↪ elements[i], rightBound - 1, ref results);
1593             }
1594         }
1595     }
1596     else
1597     {
1598         for (var i = elements.Length - 1; i >= 0; i--)
1599         {
1600             var element = elements[i];
1601             if (element != 0)
1602             {
1603                 results.Add(element);
1604             }
1605         }
1606     }
1607 }
1608 }
1609 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1610 public ulong[] GetRightElements(ulong startLink, ulong rightLink)
1611 {
1612     var result = new ulong[5];
1613     TryStepRight(startLink, rightLink, result, 0);
1614     Links.Each(Constants.Any, startLink, couple =>
1615     {
1616         if (couple != startLink)
1617         {
1618             if (TryStepRight(couple, rightLink, result, 2))
1619             {
1620                 return false;
1621             }
1622         }
1623         return true;
1624     });
1625     if (Links.GetTarget(Links.GetTarget(startLink)) == rightLink)
1626     {
1627         result[4] = startLink;
1628     }
1629     return result;
1630 }
1631 }
1632 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1633 public bool TryStepRight(ulong startLink, ulong rightLink, ulong[] result, int offset)
1634 {
1635     var added = 0;
1636     Links.Each(startLink, Constants.Any, couple =>
1637     {
1638         if (couple != startLink)
1639         {
1640             var coupleTarget = Links.GetTarget(couple);
1641             if (coupleTarget == rightLink)
1642             {
1643                 result[offset] = couple;
1644                 if (++added == 2)
1645                 {
1646                     return false;
1647                 }
1648             }
1649             else if (Links.GetSource(coupleTarget) == rightLink) // coupleTarget.Linker
1650                 ↪ == Net.And &&
1651             {
1652                 result[offset + 1] = couple;
1653             }
1654         }
1655     });

```

```

1652         if (++added == 2)
1653         {
1654             return false;
1655         }
1656     }
1657 }
1658 return true;
1659 });
1660 return added > 0;
1661 }
1662
1663 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1664 public ulong[] GetLeftElements(ulong startLink, ulong leftLink)
1665 {
1666     var result = new ulong[5];
1667     TryStepLeft(startLink, leftLink, result, 0);
1668     Links.Each(startLink, Constants.Any, couple =>
1669     {
1670         if (couple != startLink)
1671         {
1672             if (TryStepLeft(couple, leftLink, result, 2))
1673             {
1674                 return false;
1675             }
1676         }
1677         return true;
1678     });
1679     if (Links.GetSource(Links.GetSource(leftLink)) == startLink)
1680     {
1681         result[4] = leftLink;
1682     }
1683     return result;
1684 }
1685
1686 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1687 public bool TryStepLeft(ulong startLink, ulong leftLink, ulong[] result, int offset)
1688 {
1689     var added = 0;
1690     Links.Each(Constants.Any, startLink, couple =>
1691     {
1692         if (couple != startLink)
1693         {
1694             var coupleSource = Links.GetSource(couple);
1695             if (coupleSource == leftLink)
1696             {
1697                 result[offset] = couple;
1698                 if (++added == 2)
1699                 {
1700                     return false;
1701                 }
1702             }
1703             else if (Links.GetTarget(coupleSource) == leftLink) // coupleSource.Linker
1704                 ↪ == Net.And &&
1705             {
1706                 result[offset + 1] = couple;
1707                 if (++added == 2)
1708                 {
1709                     return false;
1710                 }
1711             }
1712         }
1713         return true;
1714     });
1715     return added > 0;
1716 }
1717 #endregion
1718
1719 #region Walkers
1720
1721 public class PatternMatcher : RightSequenceWalker<ulong>
1722 {
1723     private readonly Sequences _sequences;
1724     private readonly ulong[] _patternSequence;
1725     private readonly HashSet<LinkIndex> _linksInSequence;
1726     private readonly HashSet<LinkIndex> _results;
1727
1728     #region Pattern Match
1729     enum PatternBlockType
1730 
```

```

1731     {
1732         Undefined,
1733         Gap,
1734         Elements
1735     }
1736
1737     struct PatternBlock
1738     {
1739         public PatternBlockType Type;
1740         public long Start;
1741         public long Stop;
1742     }
1743
1744     private readonly List<PatternBlock> _pattern;
1745     private int _patternPosition;
1746     private long _sequencePosition;
1747
1748     #endregion
1749
1750     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1751     public PatternMatcher(Sequences sequences, LinkIndex[] patternSequence,
1752         ↳ HashSet<LinkIndex> results)
1753         : base(sequences.Links.Unsync, new DefaultStack<ulong>())
1754     {
1755         _sequences = sequences;
1756         _patternSequence = patternSequence;
1757         _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
1758             ↳ _sequences.Constants.Any && x != ZeroOrMany));
1759         _results = results;
1760         _pattern = CreateDetailedPattern();
1761     }
1762
1763     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1764     protected override bool IsElement(ulong link) => _linksInSequence.Contains(link) ||
1765         ↳ base.IsElement(link);
1766
1767     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1768     public bool PatternMatch(LinkIndex sequenceToMatch)
1769     {
1770         _patternPosition = 0;
1771         _sequencePosition = 0;
1772         foreach (var part in Walk(sequenceToMatch))
1773         {
1774             if (!PatternMatchCore(part))
1775             {
1776                 break;
1777             }
1778         }
1779         return _patternPosition == _pattern.Count || (_patternPosition == _pattern.Count
1780             ↳ - 1 && _pattern[_patternPosition].Start == 0);
1781     }
1782
1783     [MethodImpl(MethodImplOptions.AggressiveInlining)]
1784     private List<PatternBlock> CreateDetailedPattern()
1785     {
1786         var pattern = new List<PatternBlock>();
1787         var patternBlock = new PatternBlock();
1788         for (var i = 0; i < _patternSequence.Length; i++)
1789         {
1790             if (patternBlock.Type == PatternBlockType.Undefined)
1791             {
1792                 if (_patternSequence[i] == _sequences.Constants.Any)
1793                 {
1794                     patternBlock.Type = PatternBlockType.Gap;
1795                     patternBlock.Start = 1;
1796                     patternBlock.Stop = 1;
1797                 }
1798                 else if (_patternSequence[i] == ZeroOrMany)
1799                 {
1800                     patternBlock.Type = PatternBlockType.Gap;
1801                     patternBlock.Start = 0;
1802                     patternBlock.Stop = long.MaxValue;
1803                 }
1804                 else
1805                 {
1806                     patternBlock.Type = PatternBlockType.Elements;
1807                     patternBlock.Start = i;
1808                     patternBlock.Stop = i;
1809                 }
1810             }
1811         }
1812     }

```

```

1807     else if (patternBlock.Type == PatternBlockType.Elements)
1808     {
1809         if (_patternSequence[i] == _sequences.Constants.Any)
1810         {
1811             pattern.Add(patternBlock);
1812             patternBlock = new PatternBlock
1813             {
1814                 Type = PatternBlockType.Gap,
1815                 Start = 1,
1816                 Stop = 1
1817             };
1818         }
1819         else if (_patternSequence[i] == ZeroOrMany)
1820         {
1821             pattern.Add(patternBlock);
1822             patternBlock = new PatternBlock
1823             {
1824                 Type = PatternBlockType.Gap,
1825                 Start = 0,
1826                 Stop = long.MaxValue
1827             };
1828         }
1829         else
1830         {
1831             patternBlock.Stop = i;
1832         }
1833     }
1834     else // patternBlock.Type == PatternBlockType.Gap
1835     {
1836         if (_patternSequence[i] == _sequences.Constants.Any)
1837         {
1838             patternBlock.Start++;
1839             if (patternBlock.Stop < patternBlock.Start)
1840             {
1841                 patternBlock.Stop = patternBlock.Start;
1842             }
1843         }
1844         else if (_patternSequence[i] == ZeroOrMany)
1845         {
1846             patternBlock.Stop = long.MaxValue;
1847         }
1848         else
1849         {
1850             pattern.Add(patternBlock);
1851             patternBlock = new PatternBlock
1852             {
1853                 Type = PatternBlockType.Elements,
1854                 Start = i,
1855                 Stop = i
1856             };
1857         }
1858     }
1859 }
1860 if (patternBlock.Type != PatternBlockType.Undefined)
1861 {
1862     pattern.Add(patternBlock);
1863 }
1864 return pattern;
1865 }
1866
1867 // match: search for regexp anywhere in text
1868 //int match(char* regexp, char* text)
1869 //{
1870 //    do
1871 //    {
1872 //    } while (*text++ != '\0');
1873 //    return 0;
1874 //}
1875
1876 // matchhere: search for regexp at beginning of text
1877 //int matchhere(char* regexp, char* text)
1878 //{
1879 //    if (regexp[0] == '\0')
1880 //        return 1;
1881 //    if (regexp[1] == '*')
1882 //        return matchstar(regexp[0], regexp + 2, text);
1883 //    if (regexp[0] == '$' && regexp[1] == '\0')
1884 //        return *text == '\0';
1885 //    if (*text != '\0' && (regexp[0] == '.' || regexp[0] == *text))

```

```

1886 //         return matchhere(regexp + 1, text + 1);
1887 //     return 0;
1888 //}
1889
1890 // matchstar: search for c*regexp at beginning of text
1891 //int matchstar(int c, char* regexp, char* text)
1892 //{
1893 //    do
1894 //    {        /* a * matches zero or more instances */
1895 //        if (matchhere(regexp, text))
1896 //            return 1;
1897 //    } while (*text != '\0' && (*text++ == c || c == '.'));
1898 //    return 0;
1899 //}
1900
1901 //private void GetNextPatternElement(out LinkIndex element, out long mininumGap, out
1902 //    ↪ long maximumGap)
1903 //{
1904 //    mininumGap = 0;
1905 //    maximumGap = 0;
1906 //    element = 0;
1907 //    for (; _patternPosition < _patternSequence.Length; _patternPosition++)
1908 //    {
1909 //        if (_patternSequence[_patternPosition] == Doublets.Links.Null)
1910 //            mininumGap++;
1911 //        else if (_patternSequence[_patternPosition] == ZeroOrMany)
1912 //            maximumGap = long.MaxValue;
1913 //        else
1914 //            break;
1915 //    }
1916 //    if (maximumGap < mininumGap)
1917 //        maximumGap = mininumGap;
1918 //}
1919
1920 [MethodImpl(MethodImplOptions.AggressiveInlining)]
1921 private bool PatternMatchCore(LinkIndex element)
1922 {
1923     if (_patternPosition >= _pattern.Count)
1924     {
1925         _patternPosition = -2;
1926         return false;
1927     }
1928     var currentPatternBlock = _pattern[_patternPosition];
1929     if (currentPatternBlock.Type == PatternBlockType.Gap)
1930     {
1931         //var currentMatchingBlockLength = (_sequencePosition -
1932         ↪ _lastMatchedBlockPosition);
1933         if (_sequencePosition < currentPatternBlock.Start)
1934         {
1935             _sequencePosition++;
1936             return true; // Двигаемся дальше
1937         }
1938         // Это последний блок
1939         if (_pattern.Count == _patternPosition + 1)
1940         {
1941             _patternPosition++;
1942             _sequencePosition = 0;
1943             return false; // Полное соответствие
1944         }
1945         else
1946         {
1947             if (_sequencePosition > currentPatternBlock.Stop)
1948             {
1949                 return false; // Соответствие невозможно
1950             }
1951             var nextPatternBlock = _pattern[_patternPosition + 1];
1952             if (_patternSequence[nextPatternBlock.Start] == element)
1953             {
1954                 if (nextPatternBlock.Start < nextPatternBlock.Stop)
1955                 {
1956                     _patternPosition++;
1957                     _sequencePosition = 1;
1958                 }
1959                 else
1960                 {
1961                     _patternPosition += 2;
1962                     _sequencePosition = 0;
1963                 }
1964             }
1965         }
1966     }

```



```

1963     }
1964 }
1965 }
1966 else // currentPatternBlock.Type == PatternBlockType.Elements
1967 {
1968     var patternElementPosition = currentPatternBlock.Start + _sequencePosition;
1969     if (_patternSequence[patternElementPosition] != element)
1970     {
1971         return false; // Соответствие невозможно
1972     }
1973     if (patternElementPosition == currentPatternBlock.Stop)
1974     {
1975         _patternPosition++;
1976         _sequencePosition = 0;
1977     }
1978     else
1979     {
1980         _sequencePosition++;
1981     }
1982 }
1983 return true;
1984 //if (_patternSequence[_patternPosition] != element)
1985 //    return false;
1986 //else
1987 //{
1988 //    _sequencePosition++;
1989 //    _patternPosition++;
1990 //    return true;
1991 //}
1992 ///////
1993 //if (_filterPosition == _patternSequence.Length)
1994 //{
1995 //    _filterPosition = -2; // Длиннее чем нужно
1996 //    return false;
1997 //}
1998 //if (element != _patternSequence[_filterPosition])
1999 //{
2000 //    _filterPosition = -1;
2001 //    return false; // Начинается иначе
2002 //}
2003 //_filterPosition++;
2004 //if (_filterPosition == (_patternSequence.Length - 1))
2005 //    return false;
2006 //if (_filterPosition >= 0)
2007 //{
2008 //    if (element == _patternSequence[_filterPosition + 1])
2009 //        _filterPosition++;
2010 //    else
2011 //        return false;
2012 //}
2013 //if (_filterPosition < 0)
2014 //{
2015 //    if (element == _patternSequence[0])
2016 //        _filterPosition = 0;
2017 //}
2018 }
2019
2020 [MethodImpl(MethodImplOptions.AggressiveInlining)]
2021 public void AddAllPatternMatchedToResults(IEnumerable<ulong> sequencesToMatch)
2022 {
2023     foreach (var sequenceToMatch in sequencesToMatch)
2024     {
2025         if (PatternMatch(sequenceToMatch))
2026         {
2027             _results.Add(sequenceToMatch);
2028         }
2029     }
2030 }
2031 }
2032
2033 #endregion
2034 }
2035 }

```

1.43 ./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 LinkIndex = System.UInt64;
11
12 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
13
14 namespace Platform.Data.Doublets.Sequences
15 {
16     /// <summary>
17     /// Представляет коллекцию последовательностей связей.
18     /// </summary>
19     /// <remarks>
20     /// Обязательно реализовать атомарность каждого публичного метода.
21     ///
22     /// TODO:
23     ///
24     /// !!! Повышение вероятности повторного использования групп (подпоследовательностей),
25     /// через естественную группировку по unicode типам, все whitespace вместе, все символы
26     /// ↪ вместе, все числа вместе и т.п.
27     /// + использовать ровно сбалансированный вариант, чтобы уменьшать вложенность (глубину
28     /// ↪ графа)
29     ///
30     /// х*у - найти все связи между, в последовательностях любой формы, если не стоит
31     /// ↪ ограничитель на то, что является последовательностью, а что нет,
32     /// то находятся любые структуры связей, которые содержат эти элементы именно в таком
33     /// ↪ порядке.
34     ///
35     /// Рост последовательности слева и справа.
36     /// Поиск со звёздочкой.
37     /// URL, PURL - реестр используемых во вне ссылок на ресурсы,
38     /// так же проблема может быть решена при реализации дистанционных триггеров.
39     /// Нужны ли уникальные указатели вообще?
40     /// Что если обращение к информации будет происходить через содержимое всегда?
41     ///
42     /// Писать тесты.
43     ///
44     ///
45     /// Можно убрать зависимость от конкретной реализации Links,
46     /// на зависимость от абстрактного элемента, который может быть представлен несколькими
47     /// ↪ способами.
48     ///
49     /// Можно ли как-то сделать один общий интерфейс
50     ///
51     ///
52     /// Блокчейн и/или гит для распределённой записи транзакций.
53     ///
54     /// </remarks>
55     public partial class Sequences : ILinks<LinkIndex> // IList<string>, IList<LinkIndex[]>
56     ↪ (после завершения реализации Sequences)
57     {
58         /// <summary>Возвращает значение LinkIndex, обозначающее любое количество
59         ↪ связей.</summary>
60         public const LinkIndex ZeroOrMany = LinkIndex.MaxValue;
61
62         public SequencesOptions<LinkIndex> Options { get; }
63         public SynchronizedLinks<LinkIndex> Links { get; }
64         private readonly ISynchronization _sync;
65
66         public LinksConstants<LinkIndex> Constants { get; }
67
68         [MethodImpl(MethodImplOptions.AggressiveInlining)]
69         public Sequences(SynchronizedLinks<LinkIndex> links, SequencesOptions<LinkIndex> options)
70         {
71             Links = links;
72             _sync = links.SyncRoot;
73             Options = options;
74             Options.ValidateOptions();
75             Options.InitOptions(Links);
76             Constants = links.Constants;
77         }
78
79         [MethodImpl(MethodImplOptions.AggressiveInlining)]
80         public Sequences(SynchronizedLinks<LinkIndex> links) : this(links, new
81         ↪ SequencesOptions<LinkIndex>()) { }
82
83         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

76 public bool IsSequence(LinkIndex sequence)
77 {
78     return _sync.ExecuteReadOperation(() =>
79     {
80         if (Options.UseSequenceMarker)
81         {
82             return Options.MarkedSequenceMatcher.IsMatched(sequence);
83         }
84         return !Links.Unsync.IsPartialPoint(sequence);
85     });
86 }
87
88 [MethodImpl(MethodImplOptions.AggressiveInlining)]
89 private LinkIndex GetSequenceByElements(LinkIndex sequence)
90 {
91     if (Options.UseSequenceMarker)
92     {
93         return Links.SearchOrDefault(Options.SequenceMarkerLink, sequence);
94     }
95     return sequence;
96 }
97
98 [MethodImpl(MethodImplOptions.AggressiveInlining)]
99 private LinkIndex GetSequenceElements(LinkIndex sequence)
100 {
101     if (Options.UseSequenceMarker)
102     {
103         var linkContents = new Link<ulong>(Links.GetLink(sequence));
104         if (linkContents.Source == Options.SequenceMarkerLink)
105         {
106             return linkContents.Target;
107         }
108         if (linkContents.Target == Options.SequenceMarkerLink)
109         {
110             return linkContents.Source;
111         }
112     }
113     return sequence;
114 }
115
116 #region Count
117
118 [MethodImpl(MethodImplOptions.AggressiveInlining)]
119 public LinkIndex Count(ICollection<LinkIndex> restrictions)
120 {
121     if (restrictions.IsNullOrEmpty())
122     {
123         return Links.Count(Constants.Any, Options.SequenceMarkerLink, Constants.Any);
124     }
125     if (restrictions.Count == 1) // Первая связь это адрес
126     {
127         var sequenceIndex = restrictions[0];
128         if (sequenceIndex == Constants.Null)
129         {
130             return 0;
131         }
132         if (sequenceIndex == Constants.Any)
133         {
134             return Count(null);
135         }
136         if (Options.UseSequenceMarker)
137         {
138             return Links.Count(Constants.Any, Options.SequenceMarkerLink, sequenceIndex);
139         }
140         return Links.Exists(sequenceIndex) ? 1UL : 0;
141     }
142     throw new NotImplementedException();
143 }
144
145 [MethodImpl(MethodImplOptions.AggressiveInlining)]
146 private LinkIndex CountUsages(params LinkIndex[] restrictions)
147 {
148     if (restrictions.Length == 0)
149     {
150         return 0;
151     }
152     if (restrictions.Length == 1) // Первая связь это адрес
153     {
154         if (restrictions[0] == Constants.Null)

```

```

155     {
156         return 0;
157     }
158     var any = Constants.Any;
159     if (Options.UseSequenceMarker)
160     {
161         var elementsLink = GetSequenceElements(restrictions[0]);
162         var sequenceLink = GetSequenceByElements(elementsLink);
163         if (sequenceLink != Constants.Null)
164         {
165             return Links.Count(any, sequenceLink) + Links.Count(any, elementsLink) -
166                 ↳ 1;
167         }
168         return Links.Count(any, elementsLink);
169     }
170     return Links.Count(any, restrictions[0]);
171 }
172 throw new NotImplementedException();
173 }
174 #endregion
175
176 #region Create
177
178 [MethodImpl(MethodImplOptions.AggressiveInlining)]
179 public LinkIndex Create(ICollection<LinkIndex> restrictions)
180 {
181     return _sync.ExecuteWriteOperation(() =>
182     {
183         if (restrictions.IsNullOrEmpty())
184         {
185             return Constants.Null;
186         }
187         Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
188         return CreateCore(restrictions);
189     });
190 }
191
192 [MethodImpl(MethodImplOptions.AggressiveInlining)]
193 private LinkIndex CreateCore(ICollection<LinkIndex> restrictions)
194 {
195     LinkIndex[] sequence = restrictions.SkipFirst();
196     if (Options.UseIndex)
197     {
198         Options.Index.Add(sequence);
199     }
200     var sequenceRoot = default(LinkIndex);
201     if (Options.EnforceSingleSequenceVersionOnWriteBasedOnExisting)
202     {
203         var matches = Each(restrictions);
204         if (matches.Count > 0)
205         {
206             sequenceRoot = matches[0];
207         }
208     }
209     else if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew)
210     {
211         return CompactCore(sequence);
212     }
213     if (sequenceRoot == default)
214     {
215         sequenceRoot = Options.LinksToSequenceConverter.Convert(sequence);
216     }
217     if (Options.UseSequenceMarker)
218     {
219         return Links.Unsync.GetOrCreate(Options.SequenceMarkerLink, sequenceRoot);
220     }
221     return sequenceRoot; // Возвращаем корень последовательности (т.е. сами элементы)
222 }
223 #endregion
224
225 #region Each
226
227 [MethodImpl(MethodImplOptions.AggressiveInlining)]
228 public List<LinkIndex> Each(ICollection<LinkIndex> sequence)
229 {
230     var results = new List<LinkIndex>();
231     var filler = new ListFiller<LinkIndex, LinkIndex>(results, Constants.Continue);

```

```

233     Each(filler.AddFirstAndReturnConstant, sequence);
234     return results;
235 }
236
237 [MethodImpl(MethodImplOptions.AggressiveInlining)]
238 public LinkIndex Each(Func<IList<LinkIndex>, LinkIndex> handler, IList<LinkIndex>
    ↳ restrictions)
239 {
240     return _sync.ExecuteReadOperation(() =>
241     {
242         if (restrictions.IsNullOrEmpty())
243         {
244             return Constants.Continue;
245         }
246         Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
247         if (restrictions.Count == 1)
248         {
249             var link = restrictions[0];
250             var any = Constants.Any;
251             if (link == any)
252             {
253                 if (Options.UseSequenceMarker)
254                 {
255                     return Links.Unsync.Each(handler, new Link<LinkIndex>(any,
    ↳ Options.SequenceMarkerLink, any));
256                 }
257                 else
258                 {
259                     return Links.Unsync.Each(handler, new Link<LinkIndex>(any, any,
    ↳ any));
260                 }
261             }
262             if (Options.UseSequenceMarker)
263             {
264                 var sequenceLinkValues = Links.Unsync.GetLink(link);
265                 if (sequenceLinkValues[Constants.SourcePart] ==
    ↳ Options.SequenceMarkerLink)
266                 {
267                     link = sequenceLinkValues[Constants.TargetPart];
268                 }
269             }
270             var sequence = Options.Walker.Walk(link).ToArray().ShiftRight();
271             sequence[0] = link;
272             return handler(sequence);
273         }
274         else if (restrictions.Count == 2)
275         {
276             throw new NotImplementedException();
277         }
278         else if (restrictions.Count == 3)
279         {
280             return Links.Unsync.Each(handler, restrictions);
281         }
282         else
283         {
284             var sequence = restrictions.SkipFirst();
285             if (Options.UseIndex && !Options.Index.MightContain(sequence))
286             {
287                 return Constants.Break;
288             }
289             return EachCore(handler, sequence);
290         }
291     });
292 }
293
294 [MethodImpl(MethodImplOptions.AggressiveInlining)]
295 private LinkIndex EachCore(Func<IList<LinkIndex>, LinkIndex> handler, IList<LinkIndex>
    ↳ values)
296 {
297     var matcher = new Matcher(this, values, new HashSet<LinkIndex>(), handler);
298     // TODO: Find out why matcher.HandleFullMatched executed twice for the same sequence
    ↳ Id.
299     Func<IList<LinkIndex>, LinkIndex> innerHandler = Options.UseSequenceMarker ?
    ↳ (Func<IList<LinkIndex>, LinkIndex>)matcher.HandleFullMatchedSequence :
    ↳ matcher.HandleFullMatched;
300     //if (sequence.Length >= 2)
301     if (StepRight(innerHandler, values[0], values[1]) != Constants.Continue)
302     {

```

```

303         return Constants.Break;
304     }
305     var last = values.Count - 2;
306     for (var i = 1; i < last; i++)
307     {
308         if (PartialStepRight(innerHandler, values[i], values[i + 1]) !=
309             ↪ Constants.Continue)
310         {
311             return Constants.Break;
312         }
313     }
314     if (values.Count >= 3)
315     {
316         if (StepLeft(innerHandler, values[values.Count - 2], values[values.Count - 1])
317             ↪ != Constants.Continue)
318         {
319             return Constants.Break;
320         }
321     }
322     return Constants.Continue;
323 }
324 [MethodImpl(MethodImplOptions.AggressiveInlining)]
325 private LinkIndex PartialStepRight(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
326     ↪ left, LinkIndex right)
327 {
328     return Links.Unsync.Each(doublet =>
329     {
330         var doubletIndex = doublet[Constants.IndexPart];
331         if (StepRight(handler, doubletIndex, right) != Constants.Continue)
332         {
333             return Constants.Break;
334         }
335         if (left != doubletIndex)
336         {
337             return PartialStepRight(handler, doubletIndex, right);
338         }
339         return Constants.Continue;
340     }, new Link<LinkIndex>(Constants.Any, Constants.Any, left));
341 }
342 [MethodImpl(MethodImplOptions.AggressiveInlining)]
343 private LinkIndex StepRight(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex left,
344     ↪ LinkIndex right) => Links.Unsync.Each(rightStep => TryStepRightUp(handler, right,
345     ↪ rightStep[Constants.IndexPart]), new Link<LinkIndex>(Constants.Any, left,
346     ↪ Constants.Any));
347 [MethodImpl(MethodImplOptions.AggressiveInlining)]
348 private LinkIndex TryStepRightUp(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
349     ↪ right, LinkIndex stepFrom)
350 {
351     var upStep = stepFrom;
352     var firstSource = Links.Unsync.GetTarget(upStep);
353     while (firstSource != right && firstSource != upStep)
354     {
355         upStep = firstSource;
356         firstSource = Links.Unsync.GetSource(upStep);
357     }
358     if (firstSource == right)
359     {
360         return handler(new LinkAddress<LinkIndex>(stepFrom));
361     }
362     return Constants.Continue;
363 }
364 [MethodImpl(MethodImplOptions.AggressiveInlining)]
365 private LinkIndex StepLeft(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex left,
366     ↪ LinkIndex right) => Links.Unsync.Each(leftStep => TryStepLeftUp(handler, left,
367     ↪ leftStep[Constants.IndexPart]), new Link<LinkIndex>(Constants.Any, Constants.Any,
368     ↪ right));
369 [MethodImpl(MethodImplOptions.AggressiveInlining)]
370 private LinkIndex TryStepLeftUp(Func<IList<LinkIndex>, LinkIndex> handler, LinkIndex
371     ↪ left, LinkIndex stepFrom)
372 {
373     var upStep = stepFrom;
374     var firstTarget = Links.Unsync.GetSource(upStep);
375     while (firstTarget != left && firstTarget != upStep)

```

```

370     {
371         upStep = firstTarget;
372         firstTarget = Links.Unsync.GetTarget(upStep);
373     }
374     if (firstTarget == left)
375     {
376         return handler(new LinkAddress<LinkIndex>(stepFrom));
377     }
378     return Constants.Continue;
379 }
380
381 #endregion
382
383 #region Update
384
385 [MethodImpl(MethodImplOptions.AggressiveInlining)]
386 public LinkIndex Update(IList<LinkIndex> restrictions, IList<LinkIndex> substitution)
387 {
388     var sequence = restrictions.SkipFirst();
389     var newSequence = substitution.SkipFirst();
390     if (sequence.IsNullOrEmpty() && newSequence.IsNullOrEmpty())
391     {
392         return Constants.Null;
393     }
394     if (sequence.IsNullOrEmpty())
395     {
396         return Create(substitution);
397     }
398     if (newSequence.IsNullOrEmpty())
399     {
400         Delete(restrictions);
401         return Constants.Null;
402     }
403     return _sync.ExecuteWriteOperation((Func<ulong>)(() =>
404     {
405         ILinksExtensions.EnsureLinkIsAnyOrExists<ulong>(Links, (IList<ulong>)sequence);
406         Links.EnsureLinkExists(newSequence);
407         return UpdateCore(sequence, newSequence);
408     })));
409 }
410
411 [MethodImpl(MethodImplOptions.AggressiveInlining)]
412 private LinkIndex UpdateCore(IList<LinkIndex> sequence, IList<LinkIndex> newSequence)
413 {
414     LinkIndex bestVariant;
415     if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew &&
416         ↪ !sequence.EqualTo(newSequence))
417     {
418         bestVariant = CompactCore(newSequence);
419     }
420     else
421     {
422         bestVariant = CreateCore(newSequence);
423     }
424     // TODO: Check all options only ones before loop execution
425     // Возможно нужно две версии Each, возвращающий фактические последовательности и с
426     ↪ маркером,
427     // или возможно даже возвращать и тот и тот вариант. С другой стороны все варианты
428     ↪ можно получить имея только фактические последовательности.
429     foreach (var variant in Each(sequence))
430     {
431         if (variant != bestVariant)
432         {
433             UpdateOneCore(variant, bestVariant);
434         }
435     }
436     return bestVariant;
437 }
438
439 [MethodImpl(MethodImplOptions.AggressiveInlining)]
440 private void UpdateOneCore(LinkIndex sequence, LinkIndex newSequence)
441 {
442     if (Options.UseGarbageCollection)
443     {
444         var sequenceElements = GetSequenceElements(sequence);
445         var sequenceElementsContents = new Link<ulong>(Links.GetLink(sequenceElements));
446         var sequenceLink = GetSequenceByElements(sequenceElements);
447         var newSequenceElements = GetSequenceElements(newSequence);
448         var newSequenceLink = GetSequenceByElements(newSequenceElements);

```

```

446     if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
447     {
448         if (sequenceLink != Constants.Null)
449         {
450             Links.Unsync.MergeAndDelete(sequenceLink, newSequenceLink);
451         }
452         Links.Unsync.MergeAndDelete(sequenceElements, newSequenceElements);
453     }
454     ClearGarbage(sequenceElementsContents.Source);
455     ClearGarbage(sequenceElementsContents.Target);
456 }
457 else
458 {
459     if (Options.UseSequenceMarker)
460     {
461         var sequenceElements = GetSequenceElements(sequence);
462         var sequenceLink = GetSequenceByElements(sequenceElements);
463         var newSequenceElements = GetSequenceElements(newSequence);
464         var newSequenceLink = GetSequenceByElements(newSequenceElements);
465         if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
466         {
467             if (sequenceLink != Constants.Null)
468             {
469                 Links.Unsync.MergeAndDelete(sequenceLink, newSequenceLink);
470             }
471             Links.Unsync.MergeAndDelete(sequenceElements, newSequenceElements);
472         }
473     }
474     else
475     {
476         if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
477         {
478             Links.Unsync.MergeAndDelete(sequence, newSequence);
479         }
480     }
481 }
482 }
483
484 #endregion
485
486 #region Delete
487
488 [MethodImpl(MethodImplOptions.AggressiveInlining)]
489 public void Delete(IList<LinkIndex> restrictions)
490 {
491     _sync.ExecuteWriteOperation(() =>
492     {
493         var sequence = restrictions.SkipFirst();
494         // TODO: Check all options only ones before loop execution
495         foreach (var linkToDelete in Each(sequence))
496         {
497             DeleteOneCore(linkToDelete);
498         }
499     });
500 }
501
502 [MethodImpl(MethodImplOptions.AggressiveInlining)]
503 private void DeleteOneCore(LinkIndex link)
504 {
505     if (Options.UseGarbageCollection)
506     {
507         var sequenceElements = GetSequenceElements(link);
508         var sequenceElementsContents = new Link<ulong>(Links.GetLink(sequenceElements));
509         var sequenceLink = GetSequenceByElements(sequenceElements);
510         if (Options.UseCascadeDelete || CountUsages(link) == 0)
511         {
512             if (sequenceLink != Constants.Null)
513             {
514                 Links.Unsync.Delete(sequenceLink);
515             }
516             Links.Unsync.Delete(link);
517         }
518         ClearGarbage(sequenceElementsContents.Source);
519         ClearGarbage(sequenceElementsContents.Target);
520     }
521     else
522     {
523         if (Options.UseSequenceMarker)

```



```

524     {
525         var sequenceElements = GetSequenceElements(link);
526         var sequenceLink = GetSequenceByElements(sequenceElements);
527         if (Options.UseCascadeDelete || CountUsages(link) == 0)
528         {
529             if (sequenceLink != Constants.Null)
530             {
531                 Links.Unsync.Delete(sequenceLink);
532             }
533             Links.Unsync.Delete(link);
534         }
535     }
536     else
537     {
538         if (Options.UseCascadeDelete || CountUsages(link) == 0)
539         {
540             Links.Unsync.Delete(link);
541         }
542     }
543 }
544 }
545
546 #endregion
547
548 #region Compactification
549
550 [MethodImpl(MethodImplOptions.AggressiveInlining)]
551 public void CompactAll()
552 {
553     _sync.ExecuteWriteOperation(() =>
554     {
555         var sequences = Each((LinkAddress<LinkIndex>)Constants.Any);
556         for (int i = 0; i < sequences.Count; i++)
557         {
558             var sequence = this.ToList(sequences[i]);
559             Compact(sequence.ShiftRight());
560         }
561     });
562 }
563
564 /// <remarks>
565 /// bestVariant можно выбирать по максимальному числу использований,
566 /// но сбалансированный позволяет гарантировать уникальность (если есть возможность,
567 /// гарантировать его использование в других местах).
568 ///
569 /// Получается этот метод должен игнорировать Options.EnforceSingleSequenceVersionOnWrite
570 /// </remarks>
571 [MethodImpl(MethodImplOptions.AggressiveInlining)]
572 public LinkIndex Compact(ICollection<LinkIndex> sequence)
573 {
574     return _sync.ExecuteWriteOperation(() =>
575     {
576         if (sequence.IsNullOrEmpty())
577         {
578             return Constants.Null;
579         }
580         Links.EnsureInnerReferenceExists(sequence, nameof(sequence));
581         return CompactCore(sequence);
582     });
583 }
584
585 [MethodImpl(MethodImplOptions.AggressiveInlining)]
586 private LinkIndex CompactCore(ICollection<LinkIndex> sequence) => UpdateCore(sequence,
587     ↪ sequence);
588
589 #endregion
590
591 #region Garbage Collection
592
593 /// <remarks>
594 /// TODO: Добавить дополнительный обработчик / событие CanBeDeleted которое можно
595 ↪ определить извне или в унаследованном классе
596 /// </remarks>
597 [MethodImpl(MethodImplOptions.AggressiveInlining)]
598 private bool IsGarbage(LinkIndex link) => link != Options.SequenceMarkerLink &&
599     ↪ !Links.Unsync.IsPartialPoint(link) && Links.Count(Constants.Any, link) == 0;
600
601 [MethodImpl(MethodImplOptions.AggressiveInlining)]
602 private void ClearGarbage(LinkIndex link)

```

```

600 {
601     if (IsGarbage(link))
602     {
603         var contents = new Link<ulong>(Links.GetLink(link));
604         Links.Unsync.Delete(link);
605         ClearGarbage(contents.Source);
606         ClearGarbage(contents.Target);
607     }
608 }
609
610 #endregion
611
612 #region Walkers
613
614 [MethodImpl(MethodImplOptions.AggressiveInlining)]
615 public bool EachPart(Func<LinkIndex, bool> handler, LinkIndex sequence)
616 {
617     return _sync.ExecuteReadOperation(() =>
618     {
619         var links = Links.Unsync;
620         foreach (var part in Options.Walker.Walk(sequence))
621         {
622             if (!handler(part))
623             {
624                 return false;
625             }
626         }
627         return true;
628     });
629 }
630
631 public class Matcher : RightSequenceWalker<LinkIndex>
632 {
633     private readonly Sequences _sequences;
634     private readonly IList<LinkIndex> _patternSequence;
635     private readonly HashSet<LinkIndex> _linksInSequence;
636     private readonly HashSet<LinkIndex> _results;
637     private readonly Func<IList<LinkIndex>, LinkIndex> _stopableHandler;
638     private readonly HashSet<LinkIndex> _readAsElements;
639     private int _filterPosition;
640
641     [MethodImpl(MethodImplOptions.AggressiveInlining)]
642     public Matcher(Sequences sequences, IList<LinkIndex> patternSequence,
643         ↪ HashSet<LinkIndex> results, Func<IList<LinkIndex>, LinkIndex> stopableHandler,
644         ↪ HashSet<LinkIndex> readAsElements = null)
645         : base(sequences.Links.Unsync, new DefaultStack<LinkIndex>())
646     {
647         _sequences = sequences;
648         _patternSequence = patternSequence;
649         _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
650             ↪ _links.Constants.Any && x != ZeroOrMany));
651         _results = results;
652         _stopableHandler = stopableHandler;
653         _readAsElements = readAsElements;
654     }
655
656     [MethodImpl(MethodImplOptions.AggressiveInlining)]
657     protected override bool IsElement(LinkIndex link) => base.IsElement(link) ||
658         ↪ (_readAsElements != null && _readAsElements.Contains(link)) ||
659         ↪ _linksInSequence.Contains(link);
660
661     [MethodImpl(MethodImplOptions.AggressiveInlining)]
662     public bool FullMatch(LinkIndex sequenceToMatch)
663     {
664         _filterPosition = 0;
665         foreach (var part in Walk(sequenceToMatch))
666         {
667             if (!FullMatchCore(part))
668             {
669                 break;
670             }
671         }
672         return _filterPosition == _patternSequence.Count;
673     }
674
675     [MethodImpl(MethodImplOptions.AggressiveInlining)]
676     private bool FullMatchCore(LinkIndex element)
677     {
678         if (_filterPosition == _patternSequence.Count)
679         {

```

```

675         _filterPosition = -2; // Длиннее чем нужно
676         return false;
677     }
678     if (_patternSequence[_filterPosition] != _links.Constants.Any
679         && element != _patternSequence[_filterPosition])
680     {
681         _filterPosition = -1;
682         return false; // Начинается/Продолжается иначе
683     }
684     _filterPosition++;
685     return true;
686 }
687
688 [MethodImpl(MethodImplOptions.AggressiveInlining)]
689 public void AddFullMatchedToResults(ICollection<LinkIndex> restrictions)
690 {
691     var sequenceToMatch = restrictions[_links.Constants.IndexPart];
692     if (FullMatch(sequenceToMatch))
693     {
694         _results.Add(sequenceToMatch);
695     }
696 }
697
698 [MethodImpl(MethodImplOptions.AggressiveInlining)]
699 public LinkIndex HandleFullMatched(ICollection<LinkIndex> restrictions)
700 {
701     var sequenceToMatch = restrictions[_links.Constants.IndexPart];
702     if (FullMatch(sequenceToMatch) && _results.Add(sequenceToMatch))
703     {
704         return _stopableHandler(new LinkAddress<LinkIndex>(sequenceToMatch));
705     }
706     return _links.Constants.Continue;
707 }
708
709 [MethodImpl(MethodImplOptions.AggressiveInlining)]
710 public LinkIndex HandleFullMatchedSequence(ICollection<LinkIndex> restrictions)
711 {
712     var sequenceToMatch = restrictions[_links.Constants.IndexPart];
713     var sequence = _sequences.GetSequenceByElements(sequenceToMatch);
714     if (sequence != _links.Constants.Null && FullMatch(sequenceToMatch) &&
715         ↪ _results.Add(sequenceToMatch))
716     {
717         return _stopableHandler(new LinkAddress<LinkIndex>(sequence));
718     }
719     return _links.Constants.Continue;
720 }
721
722 /// <remarks>
723 /// TODO: Add support for LinksConstants.Any
724 /// </remarks>
725 [MethodImpl(MethodImplOptions.AggressiveInlining)]
726 public bool PartialMatch(LinkIndex sequenceToMatch)
727 {
728     _filterPosition = -1;
729     foreach (var part in Walk(sequenceToMatch))
730     {
731         if (!PartialMatchCore(part))
732         {
733             break;
734         }
735     }
736     return _filterPosition == _patternSequence.Count - 1;
737 }
738
739 [MethodImpl(MethodImplOptions.AggressiveInlining)]
740 private bool PartialMatchCore(LinkIndex element)
741 {
742     if (_filterPosition == (_patternSequence.Count - 1))
743     {
744         return false; // Нашлось
745     }
746     if (_filterPosition >= 0)
747     {
748         if (element == _patternSequence[_filterPosition + 1])
749         {
750             _filterPosition++;
751         }
752         else
753         {

```

```

753         _filterPosition = -1;
754     }
755 }
756 if (_filterPosition < 0)
757 {
758     if (element == _patternSequence[0])
759     {
760         _filterPosition = 0;
761     }
762 }
763 return true; // Ищем дальше
764 }
765
766 [MethodImpl(MethodImplOptions.AggressiveInlining)]
767 public void AddPartialMatchedToResults(LinkIndex sequenceToMatch)
768 {
769     if (PartialMatch(sequenceToMatch))
770     {
771         _results.Add(sequenceToMatch);
772     }
773 }
774
775 [MethodImpl(MethodImplOptions.AggressiveInlining)]
776 public LinkIndex HandlePartialMatched(ICollection<LinkIndex> restrictions)
777 {
778     var sequenceToMatch = restrictions[_links.Constants.IndexPart];
779     if (PartialMatch(sequenceToMatch))
780     {
781         return _stopableHandler(new LinkAddress<LinkIndex>(sequenceToMatch));
782     }
783     return _links.Constants.Continue;
784 }
785
786 [MethodImpl(MethodImplOptions.AggressiveInlining)]
787 public void AddAllPartialMatchedToResults(IEnumerable<LinkIndex> sequencesToMatch)
788 {
789     foreach (var sequenceToMatch in sequencesToMatch)
790     {
791         if (PartialMatch(sequenceToMatch))
792         {
793             _results.Add(sequenceToMatch);
794         }
795     }
796 }
797
798 [MethodImpl(MethodImplOptions.AggressiveInlining)]
799 public void AddAllPartialMatchedToResultsAndReadAsElements(IEnumerable<LinkIndex>
800 ↪ sequencesToMatch)
801 {
802     foreach (var sequenceToMatch in sequencesToMatch)
803     {
804         if (PartialMatch(sequenceToMatch))
805         {
806             _readAsElements.Add(sequenceToMatch);
807             _results.Add(sequenceToMatch);
808         }
809     }
810 }
811 }
812 #endregion
813 }
814 }

```

1.44 ./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     public static class SequencesExtensions
10     {
11         [MethodImpl(MethodImplOptions.AggressiveInlining)]
12         public static TLink Create<TLink>(this ICollection<TLink> sequences, ICollection<TLink>
13 ↪ groupedSequence)
14     {

```

```

14     var finalSequence = new TLink[groupedSequence.Count];
15     for (var i = 0; i < finalSequence.Length; i++)
16     {
17         var part = groupedSequence[i];
18         finalSequence[i] = part.Length == 1 ? part[0] :
19             ↪ sequences.Create(part.ShiftRight());
20     }
21     return sequences.Create(finalSequence.ShiftRight());
22 }
23 [MethodImpl(MethodImplOptions.AggressiveInlining)]
24 public static IList<TLink> ToList<TLink>(this ILinks<TLink> sequences, TLink sequence)
25 {
26     var list = new List<TLink>();
27     var filler = new ListFiller<TLink, TLink>(list, sequences.Constants.Break);
28     sequences.Each(filler.AddSkipFirstAndReturnConstant, new
29         ↪ LinkAddress<TLink>(sequence));
30     return list;
31 }
32 }

```

1.45 ./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     public class SequencesOptions<TLink> // TODO: To use type parameter <TLink> the
19         ↪ ILinks<TLink> must contain GetConstants function.
20     {
21         private static readonly EqualityComparer<TLink> _equalityComparer =
22             ↪ EqualityComparer<TLink>.Default;
23
24         public TLink SequenceMarkerLink
25         {
26             [MethodImpl(MethodImplOptions.AggressiveInlining)]
27             get;
28             [MethodImpl(MethodImplOptions.AggressiveInlining)]
29             set;
30         }
31
32         public bool UseCascadeUpdate
33         {
34             [MethodImpl(MethodImplOptions.AggressiveInlining)]
35             get;
36             [MethodImpl(MethodImplOptions.AggressiveInlining)]
37             set;
38         }
39
40         public bool UseCascadeDelete
41         {
42             [MethodImpl(MethodImplOptions.AggressiveInlining)]
43             get;
44             [MethodImpl(MethodImplOptions.AggressiveInlining)]
45             set;
46         }
47
48         public bool UseIndex
49         {
50             [MethodImpl(MethodImplOptions.AggressiveInlining)]
51             get;
52             [MethodImpl(MethodImplOptions.AggressiveInlining)]
53             set;
54         } // TODO: Update Index on sequence update/delete.
55
56         public bool UseSequenceMarker
57         {
58             [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

57         get;
58         [MethodImpl(MethodImplOptions.AggressiveInlining)]
59         set;
60     }
61
62     public bool UseCompression
63     {
64         [MethodImpl(MethodImplOptions.AggressiveInlining)]
65         get;
66         [MethodImpl(MethodImplOptions.AggressiveInlining)]
67         set;
68     }
69
70     public bool UseGarbageCollection
71     {
72         [MethodImpl(MethodImplOptions.AggressiveInlining)]
73         get;
74         [MethodImpl(MethodImplOptions.AggressiveInlining)]
75         set;
76     }
77
78     public bool EnforceSingleSequenceVersionOnWriteBasedOnExisting
79     {
80         [MethodImpl(MethodImplOptions.AggressiveInlining)]
81         get;
82         [MethodImpl(MethodImplOptions.AggressiveInlining)]
83         set;
84     }
85
86     public bool EnforceSingleSequenceVersionOnWriteBasedOnNew
87     {
88         [MethodImpl(MethodImplOptions.AggressiveInlining)]
89         get;
90         [MethodImpl(MethodImplOptions.AggressiveInlining)]
91         set;
92     }
93
94     public MarkedSequenceCriterionMatcher<TLink> MarkedSequenceMatcher
95     {
96         [MethodImpl(MethodImplOptions.AggressiveInlining)]
97         get;
98         [MethodImpl(MethodImplOptions.AggressiveInlining)]
99         set;
100     }
101
102     public IConverter<IList<TLink>, TLink> LinksToSequenceConverter
103     {
104         [MethodImpl(MethodImplOptions.AggressiveInlining)]
105         get;
106         [MethodImpl(MethodImplOptions.AggressiveInlining)]
107         set;
108     }
109
110     public ISequenceIndex<TLink> Index
111     {
112         [MethodImpl(MethodImplOptions.AggressiveInlining)]
113         get;
114         [MethodImpl(MethodImplOptions.AggressiveInlining)]
115         set;
116     }
117
118     public ISequenceWalker<TLink> Walker
119     {
120         [MethodImpl(MethodImplOptions.AggressiveInlining)]
121         get;
122         [MethodImpl(MethodImplOptions.AggressiveInlining)]
123         set;
124     }
125
126     public bool ReadFullSequence
127     {
128         [MethodImpl(MethodImplOptions.AggressiveInlining)]
129         get;
130         [MethodImpl(MethodImplOptions.AggressiveInlining)]
131         set;
132     }
133
134     // TODO: Реализовать компактификацию при чтении
135     //public bool EnforceSingleSequenceVersionOnRead { get; set; }
136     //public bool UseRequestMarker { get; set; }
137     //public bool StoreRequestResults { get; set; }

```

```

138 [MethodImpl(MethodImplOptions.AggressiveInlining)]
139 public void InitOptions(ISynchronizedLinks<TLink> links)
140 {
141     if (UseSequenceMarker)
142     {
143         if (_equalityComparer.Equals(SequenceMarkerLink, links.Constants.Null))
144         {
145             SequenceMarkerLink = links.CreatePoint();
146         }
147         else
148         {
149             if (!links.Exists(SequenceMarkerLink))
150             {
151                 var link = links.CreatePoint();
152                 if (!_equalityComparer.Equals(link, SequenceMarkerLink))
153                 {
154                     throw new InvalidOperationException("Cannot recreate sequence marker
155                     ↪ link.");
156                 }
157             }
158         }
159         if (MarkedSequenceMatcher == null)
160         {
161             MarkedSequenceMatcher = new MarkedSequenceCriterionMatcher<TLink>(links,
162             ↪ SequenceMarkerLink);
163         }
164         var balancedVariantConverter = new BalancedVariantConverter<TLink>(links);
165         if (UseCompression)
166         {
167             if (LinksToSequenceConverter == null)
168             {
169                 ICounter<TLink, TLink> totalSequenceSymbolFrequencyCounter;
170                 if (UseSequenceMarker)
171                 {
172                     totalSequenceSymbolFrequencyCounter = new
173                     ↪ TotalMarkedSequenceSymbolFrequencyCounter<TLink>(links,
174                     ↪ MarkedSequenceMatcher);
175                 }
176                 else
177                 {
178                     totalSequenceSymbolFrequencyCounter = new
179                     ↪ TotalSequenceSymbolFrequencyCounter<TLink>(links);
180                 }
181                 var doubletFrequenciesCache = new LinkFrequenciesCache<TLink>(links,
182                 ↪ totalSequenceSymbolFrequencyCounter);
183                 var compressingConverter = new CompressingConverter<TLink>(links,
184                 ↪ balancedVariantConverter, doubletFrequenciesCache);
185                 LinksToSequenceConverter = compressingConverter;
186             }
187         }
188         else
189         {
190             if (LinksToSequenceConverter == null)
191             {
192                 LinksToSequenceConverter = balancedVariantConverter;
193             }
194         }
195         if (UseIndex && Index == null)
196         {
197             Index = new SequenceIndex<TLink>(links);
198         }
199         if (Walker == null)
200         {
201             Walker = new RightSequenceWalker<TLink>(links, new DefaultStack<TLink>());
202         }
203     }
204 }
205
206 [MethodImpl(MethodImplOptions.AggressiveInlining)]
207 public void ValidateOptions()
208 {
209     if (UseGarbageCollection && !UseSequenceMarker)
210     {
211         throw new NotSupportedException("To use garbage collection UseSequenceMarker
212         ↪ option must be on.");
213     }
214 }

```

```
208     }
209 }
```

1.46 ./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     public class DateTimeToLongRawNumberSequenceConverter<TLink> : IConverter<DateTime, TLink>
10     {
11         private readonly IConverter<long, TLink> _int64ToLongRawNumberConverter;
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public DateTimeToLongRawNumberSequenceConverter(IConverter<long, TLink>
15             ↪ int64ToLongRawNumberConverter) => _int64ToLongRawNumberConverter =
16             ↪ int64ToLongRawNumberConverter;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         public TLink Convert(DateTime source) =>
20             ↪ _int64ToLongRawNumberConverter.Convert(source.ToFileTimeUtc());
21     }
22 }
```

1.47 ./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     public class LongRawNumberSequenceToDateTimeConverter<TLink> : IConverter<TLink, DateTime>
10     {
11         private readonly IConverter<TLink, long> _longRawNumberConverterToInt64;
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public LongRawNumberSequenceToDateTimeConverter(IConverter<TLink, long>
15             ↪ longRawNumberConverterToInt64) => _longRawNumberConverterToInt64 =
16             ↪ longRawNumberConverterToInt64;
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         public DateTime Convert(TLink source) =>
20             ↪ DateTime.FromFileTimeUtc(_longRawNumberConverterToInt64.Convert(source));
21     }
22 }
```

1.48 ./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     public static class UInt64LinksExtensions
13     {
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         public static void UseUnicode(this ILinks<ulong> links) => UnicodeMap.InitNew(links);
16     }
17 }
```

1.49 ./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     public class CharToUnicodeSymbolConverter<TLink> : LinksOperatorBase<TLink>,
9         ↪ IConverter<char, TLink>
```



```

9 {
10     private static readonly UncheckedConverter<char, TLink> _charToAddressConverter =
        ↳ UncheckedConverter<char, TLink>.Default;
11
12     private readonly IConverter<TLink> _addressToNumberConverter;
13     private readonly TLink _unicodeSymbolMarker;
14
15     [MethodImpl(MethodImplOptions.AggressiveInlining)]
16     public CharToUnicodeSymbolConverter(ILinks<TLink> links, IConverter<TLink>
        ↳ addressToNumberConverter, TLink unicodeSymbolMarker) : base(links)
17     {
18         _addressToNumberConverter = addressToNumberConverter;
19         _unicodeSymbolMarker = unicodeSymbolMarker;
20     }
21
22     [MethodImpl(MethodImplOptions.AggressiveInlining)]
23     public TLink Convert(char source)
24     {
25         var unaryNumber =
        ↳ _addressToNumberConverter.Convert(_charToAddressConverter.Convert(source));
26         return _links.GetOrCreate(unaryNumber, _unicodeSymbolMarker);
27     }
28 }
29 }

```

1.50 ./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     public class StringToUnicodeSequenceConverter<TLink> : LinksOperatorBase<TLink>,
        ↳ IConverter<string, TLink>
11     {
12         private readonly IConverter<string, IList<TLink>> _stringToUnicodeSymbolListConverter;
13         private readonly IConverter<IList<TLink>, TLink> _unicodeSymbolListToSequenceConverter;
14
15         [MethodImpl(MethodImplOptions.AggressiveInlining)]
16         public StringToUnicodeSequenceConverter(ILinks<TLink> links, IConverter<string,
        ↳ IList<TLink>> stringToUnicodeSymbolListConverter, IConverter<IList<TLink>, TLink>
        ↳ unicodeSymbolListToSequenceConverter) : base(links)
17         {
18             _stringToUnicodeSymbolListConverter = stringToUnicodeSymbolListConverter;
19             _unicodeSymbolListToSequenceConverter = unicodeSymbolListToSequenceConverter;
20         }
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         public StringToUnicodeSequenceConverter(ILinks<TLink> links, IConverter<string,
        ↳ IList<TLink>> stringToUnicodeSymbolListConverter, ISequenceIndex<TLink> index,
        ↳ IConverter<IList<TLink>, TLink> listToSequenceLinkConverter, TLink
        ↳ unicodeSequenceMarker)
24         : this(links, stringToUnicodeSymbolListConverter, new
        ↳ UnicodeSymbolsListToUnicodeSequenceConverter<TLink>(links, index,
        ↳ listToSequenceLinkConverter, unicodeSequenceMarker)) { }
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         public StringToUnicodeSequenceConverter(ILinks<TLink> links, IConverter<char, TLink>
        ↳ charToUnicodeSymbolConverter, ISequenceIndex<TLink> index, IConverter<IList<TLink>,
        ↳ TLink> listToSequenceLinkConverter, TLink unicodeSequenceMarker)
28         : this(links, new
        ↳ StringToUnicodeSymbolsListConverter<TLink>(charToUnicodeSymbolConverter), index,
        ↳ listToSequenceLinkConverter, unicodeSequenceMarker) { }
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         public StringToUnicodeSequenceConverter(ILinks<TLink> links, IConverter<char, TLink>
        ↳ charToUnicodeSymbolConverter, IConverter<IList<TLink>, TLink>
        ↳ listToSequenceLinkConverter, TLink unicodeSequenceMarker)
32         : this(links, charToUnicodeSymbolConverter, new Unindex<TLink>(),
        ↳ listToSequenceLinkConverter, unicodeSequenceMarker) { }
33
34         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35         public StringToUnicodeSequenceConverter(ILinks<TLink> links, IConverter<string,
        ↳ IList<TLink>> stringToUnicodeSymbolListConverter, IConverter<IList<TLink>, TLink>
        ↳ listToSequenceLinkConverter, TLink unicodeSequenceMarker)

```

```

36         : this(links, stringToUnicodeSymbolListConverter, new Unindex<TLink>(),
           ↪ listToSequenceLinkConverter, unicodeSequenceMarker) { }
37
38     [MethodImpl(MethodImplOptions.AggressiveInlining)]
39     public TLink Convert(string source)
40     {
41         var elements = _stringToUnicodeSymbolListConverter.Convert(source);
42         return _unicodeSymbolListToSequenceConverter.Convert(elements);
43     }
44 }
45 }

```

1.51 ./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      public class StringToUnicodeSymbolsListConverter<TLink> : IConverter<string, IList<TLink>>
10     {
11         private readonly IConverter<char, TLink> _charToUnicodeSymbolConverter;
12
13         [MethodImpl(MethodImplOptions.AggressiveInlining)]
14         public StringToUnicodeSymbolsListConverter(IConverter<char, TLink>
           ↪ charToUnicodeSymbolConverter) => _charToUnicodeSymbolConverter =
           ↪ charToUnicodeSymbolConverter;
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public IList<TLink> Convert(string source)
18         {
19             var elements = new TLink[source.Length];
20             for (var i = 0; i < elements.Length; i++)
21             {
22                 elements[i] = _charToUnicodeSymbolConverter.Convert(source[i]);
23             }
24             return elements;
25         }
26     }
27 }

```

1.52 ./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     public class UnicodeMap
13     {
14         public static readonly ulong FirstCharLink = 1;
15         public static readonly ulong LastCharLink = FirstCharLink + char.MaxValue;
16         public static readonly ulong MapSize = 1 + char.MaxValue;
17
18         private readonly ILinks<ulong> _links;
19         private bool _initialized;
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public UnicodeMap(ILinks<ulong> links) => _links = links;
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         public static UnicodeMap InitNew(ILinks<ulong> links)
26         {
27             var map = new UnicodeMap(links);
28             map.Init();
29             return map;
30         }
31
32         [MethodImpl(MethodImplOptions.AggressiveInlining)]
33         public void Init()
34         {
35             if (!_initialized)
36             {

```

```

37         return;
38     }
39     _initialized = true;
40     var firstLink = _links.CreatePoint();
41     if (firstLink != FirstCharLink)
42     {
43         _links.Delete(firstLink);
44     }
45     else
46     {
47         for (var i = FirstCharLink + 1; i <= LastCharLink; i++)
48         {
49             // From NIL to It (NIL -> Character) transformation meaning, (or infinite
50             ↪ amount of NIL characters before actual Character)
51             var createdLink = _links.CreatePoint();
52             _links.Update(createdLink, firstLink, createdLink);
53             if (createdLink != i)
54             {
55                 throw new InvalidOperationException("Unable to initialize UTF 16
56                 ↪ table.");
57             }
58         }
59     }
60     // 0 - null link
61     // 1 - nil character (0 character)
62     // ...
63     // 65536 (0(1) + 65535 = 65536 possible values)
64
65     [MethodImpl(MethodImplOptions.AggressiveInlining)]
66     public static ulong FromCharToLink(char character) => (ulong)character + 1;
67
68     [MethodImpl(MethodImplOptions.AggressiveInlining)]
69     public static char FromLinkToChar(ulong link) => (char)(link - 1);
70
71     [MethodImpl(MethodImplOptions.AggressiveInlining)]
72     public static bool IsCharLink(ulong link) => link <= MapSize;
73
74     [MethodImpl(MethodImplOptions.AggressiveInlining)]
75     public static string FromLinksToString(IList<ulong> linksList)
76     {
77         var sb = new StringBuilder();
78         for (int i = 0; i < linksList.Count; i++)
79         {
80             sb.Append(FromLinkToChar(linksList[i]));
81         }
82         return sb.ToString();
83     }
84
85     [MethodImpl(MethodImplOptions.AggressiveInlining)]
86     public static string FromSequenceLinkToString(ulong link, ILinks<ulong> links)
87     {
88         var sb = new StringBuilder();
89         if (links.Exists(link))
90         {
91             StopableSequenceWalker.WalkRight(link, links.GetSource, links.GetTarget,
92             ↪ x => x <= MapSize || links.GetSource(x) == x || links.GetTarget(x) == x,
93             ↪ element =>
94             {
95                 sb.Append(FromLinkToChar(element));
96                 return true;
97             });
98         }
99         return sb.ToString();
100     }
101
102     [MethodImpl(MethodImplOptions.AggressiveInlining)]
103     public static ulong[] FromCharsToLinkArray(char[] chars) => FromCharsToLinkArray(chars,
104     ↪ chars.Length);
105
106     [MethodImpl(MethodImplOptions.AggressiveInlining)]
107     public static ulong[] FromCharsToLinkArray(char[] chars, int count)
108     {
109         // char array to ulong array
110         var linksSequence = new ulong[count];
111         for (var i = 0; i < count; i++)
112         {

```

```

111         linksSequence[i] = FromCharToLink(chars[i]);
112     }
113     return linksSequence;
114 }
115
116 [MethodImpl(MethodImplOptions.AggressiveInlining)]
117 public static ulong[] FromStringToLinkArray(string sequence)
118 {
119     // char array to ulong array
120     var linksSequence = new ulong[sequence.Length];
121     for (var i = 0; i < sequence.Length; i++)
122     {
123         linksSequence[i] = FromCharToLink(sequence[i]);
124     }
125     return linksSequence;
126 }
127
128 [MethodImpl(MethodImplOptions.AggressiveInlining)]
129 public static List<ulong[]> FromStringToLinkArrayGroups(string sequence)
130 {
131     var result = new List<ulong[]>();
132     var offset = 0;
133     while (offset < sequence.Length)
134     {
135         var currentCategory = CharUnicodeInfo.GetUnicodeCategory(sequence[offset]);
136         var relativeLength = 1;
137         var absoluteLength = offset + relativeLength;
138         while (absoluteLength < sequence.Length &&
139             currentCategory ==
140                 CharUnicodeInfo.GetUnicodeCategory(sequence[absoluteLength]))
141         {
142             relativeLength++;
143             absoluteLength++;
144         }
145         // char array to ulong array
146         var innerSequence = new ulong[relativeLength];
147         var maxLength = offset + relativeLength;
148         for (var i = offset; i < maxLength; i++)
149         {
150             innerSequence[i - offset] = FromCharToLink(sequence[i]);
151         }
152         result.Add(innerSequence);
153         offset += relativeLength;
154     }
155     return result;
156 }
157
158 [MethodImpl(MethodImplOptions.AggressiveInlining)]
159 public static List<ulong[]> FromLinkArrayToLinkArrayGroups(ulong[] array)
160 {
161     var result = new List<ulong[]>();
162     var offset = 0;
163     while (offset < array.Length)
164     {
165         var relativeLength = 1;
166         if (array[offset] <= LastCharLink)
167         {
168             var currentCategory =
169                 CharUnicodeInfo.GetUnicodeCategory(FromLinkToChar(array[offset]));
170             var absoluteLength = offset + relativeLength;
171             while (absoluteLength < array.Length &&
172                 array[absoluteLength] <= LastCharLink &&
173                 currentCategory == CharUnicodeInfo.GetUnicodeCategory(FromLinkToChar(
174                     array[absoluteLength])))
175             {
176                 relativeLength++;
177                 absoluteLength++;
178             }
179         }
180         else
181         {
182             var absoluteLength = offset + relativeLength;
183             while (absoluteLength < array.Length && array[absoluteLength] > LastCharLink)
184             {
185                 relativeLength++;
186                 absoluteLength++;
187             }
188         }
189         // copy array

```

```

187         var innerSequence = new ulong[relativeLength];
188         var maxLength = offset + relativeLength;
189         for (var i = offset; i < maxLength; i++)
190         {
191             innerSequence[i - offset] = array[i];
192         }
193         result.Add(innerSequence);
194         offset += relativeLength;
195     }
196     return result;
197 }
198 }
199 }

```

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

```

1  using System;
2  using System.Runtime.CompilerServices;
3  using Platform.Interfaces;
4  using Platform.Converters;
5  using Platform.Data.Doublets.Sequences.Walkers;
6  using System.Text;
7
8  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Data.Doublets.Unicode
11 {
12     public class UnicodeSequenceToStringConverter<TLink> : LinksOperatorBase<TLink>,
13         ↪ IConverter<TLink, string>
14     {
15         private readonly ICriterionMatcher<TLink> _unicodeSequenceCriterionMatcher;
16         private readonly ISequenceWalker<TLink> _sequenceWalker;
17         private readonly IConverter<TLink, char> _unicodeSymbolToCharConverter;
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         public UnicodeSequenceToStringConverter(ILinks<TLink> links, ICriterionMatcher<TLink>
21             ↪ unicodeSequenceCriterionMatcher, ISequenceWalker<TLink> sequenceWalker,
22             ↪ IConverter<TLink, char> unicodeSymbolToCharConverter) : base(links)
23         {
24             _unicodeSequenceCriterionMatcher = unicodeSequenceCriterionMatcher;
25             _sequenceWalker = sequenceWalker;
26             _unicodeSymbolToCharConverter = unicodeSymbolToCharConverter;
27
28             [MethodImpl(MethodImplOptions.AggressiveInlining)]
29             public string Convert(TLink source)
30             {
31                 if (!_unicodeSequenceCriterionMatcher.IsMatched(source))
32                 {
33                     throw new ArgumentOutOfRangeException(nameof(source), source, "Specified link is
34                         ↪ not a unicode sequence.");
35                 }
36                 var sequence = _links.GetSource(source);
37                 var sb = new StringBuilder();
38                 foreach (var character in _sequenceWalker.Walk(sequence))
39                 {
40                     sb.Append(_unicodeSymbolToCharConverter.Convert(character));
41                 }
42                 return sb.ToString();
43             }
44         }
45     }
46 }

```

1.54 ./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     public class UnicodeSymbolToCharConverter<TLink> : LinksOperatorBase<TLink>,
11         ↪ IConverter<TLink, char>
12     {
13         private static readonly UncheckedConverter<TLink, char> _addressToCharConverter =
14             ↪ UncheckedConverter<TLink, char>.Default;
15
16         private readonly IConverter<TLink> _numberToAddressConverter;
17         private readonly ICriterionMatcher<TLink> _unicodeSymbolCriterionMatcher;

```

```

16     [MethodImpl(MethodImplOptions.AggressiveInlining)]
17     public UnicodeSymbolToCharConverter(ILinks<TLink> links, IConverter<TLink>
18         ↪ numberToAddressConverter, ICriterionMatcher<TLink> unicodeSymbolCriterionMatcher) :
19         ↪ base(links)
20     {
21         _numberToAddressConverter = numberToAddressConverter;
22         _unicodeSymbolCriterionMatcher = unicodeSymbolCriterionMatcher;
23     }
24     [MethodImpl(MethodImplOptions.AggressiveInlining)]
25     public char Convert(TLink source)
26     {
27         if (!_unicodeSymbolCriterionMatcher.IsMatched(source))
28         {
29             throw new ArgumentOutOfRangeException(nameof(source), source, "Specified link is
30                 ↪ not a unicode symbol.");
31         }
32         return _addressToCharConverter.Convert(_numberToAddressConverter.Convert(_links.GetS
33             ↪ ource(source)));
34     }

```

1.55 ./csharp/Platform.Data.Doublets.Sequences/Unicode/UnicodeSymbolsListToUnicodeSequenceConverter.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     public class UnicodeSymbolsListToUnicodeSequenceConverter<TLink> : LinksOperatorBase<TLink>,
11         ↪ IConverter<IList<TLink>, TLink>
12     {
13         private readonly ISequenceIndex<TLink> _index;
14         private readonly IConverter<IList<TLink>, TLink> _listToSequenceLinkConverter;
15         private readonly TLink _unicodeSequenceMarker;
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         public UnicodeSymbolsListToUnicodeSequenceConverter(ILinks<TLink> links,
19             ↪ ISequenceIndex<TLink> index, IConverter<IList<TLink>, TLink>
20             ↪ listToSequenceLinkConverter, TLink unicodeSequenceMarker) : base(links)
21         {
22             _index = index;
23             _listToSequenceLinkConverter = listToSequenceLinkConverter;
24             _unicodeSequenceMarker = unicodeSequenceMarker;
25         }
26
27         [MethodImpl(MethodImplOptions.AggressiveInlining)]
28         public UnicodeSymbolsListToUnicodeSequenceConverter(ILinks<TLink> links,
29             ↪ IConverter<IList<TLink>, TLink> listToSequenceLinkConverter, TLink
30             ↪ unicodeSequenceMarker)
31             : this(links, new Unindex<TLink>(), listToSequenceLinkConverter,
32                 ↪ unicodeSequenceMarker) { }
33
34         [MethodImpl(MethodImplOptions.AggressiveInlining)]
35         public TLink Convert(IList<TLink> list)
36         {
37             _index.Add(list);
38             var sequence = _listToSequenceLinkConverter.Convert(list);
39             return _links.GetOrCreate(sequence, _unicodeSequenceMarker);
40         }
41     }
42 }

```

1.56 ./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     public interface ISequenceWalker<TLink>
9     {
10         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

11         IEnumerable<TLink> Walk(TLink sequence);
12     }
13 }

```

1.57 ./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     public class LeftSequenceWalker<TLink> : SequenceWalkerBase<TLink>
11     {
12         [MethodImpl(MethodImplOptions.AggressiveInlining)]
13         public LeftSequenceWalker(ILinks<TLink> links, IStack<TLink> stack, Func<TLink, bool>
14             isElement) : base(links, stack, isElement) { }
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public LeftSequenceWalker(ILinks<TLink> links, IStack<TLink> stack) : base(links, stack,
18             isElement: links.IsPartialPoint) { }
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         protected override TLink GetNextElementAfterPop(TLink element) =>
22             _links.GetSource(element);
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         protected override TLink GetNextElementAfterPush(TLink element) =>
26             _links.GetTarget(element);
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         protected override IEnumerable<TLink> WalkContents(TLink element)
30         {
31             var links = _links;
32             var parts = links.GetLink(element);
33             var start = links.Constants.SourcePart;
34             for (var i = parts.Count - 1; i >= start; i--)
35             {
36                 var part = parts[i];
37                 if (IsElement(part))
38                 {
39                     yield return part;
40                 }
41             }
42         }
43     }
44 }

```

1.58 ./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     public class LeveledSequenceWalker<TLink> : LinksOperatorBase<TLink>, ISequenceWalker<TLink>
15     {
16         private static readonly EqualityComparer<TLink> _equalityComparer =
17             EqualityComparer<TLink>.Default;
18
19         private readonly Func<TLink, bool> _isElement;
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public LeveledSequenceWalker(ILinks<TLink> links, Func<TLink, bool> isElement) :
23             base(links) => _isElement = isElement;
24
25         [MethodImpl(MethodImplOptions.AggressiveInlining)]
26         public LeveledSequenceWalker(ILinks<TLink> links) : base(links) => _isElement =
27             links.IsPartialPoint;
28
29         [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

27     public IEnumerable<TLink> Walk(TLink sequence) => ToArray(sequence);
28
29     [MethodImpl(MethodImplOptions.AggressiveInlining)]
30     public TLink[] ToArray(TLink sequence)
31     {
32         var length = 1;
33         var array = new TLink[length];
34         array[0] = sequence;
35         if (!_isElement(sequence))
36         {
37             return array;
38         }
39         bool hasElements;
40         do
41         {
42             length *= 2;
43 #if USEARRAYPOOL
44             var nextArray = ArrayPool.Allocate<ulong>(length);
45 #else
46             var nextArray = new TLink[length];
47 #endif
48             hasElements = false;
49             for (var i = 0; i < array.Length; i++)
50             {
51                 var candidate = array[i];
52                 if (_equalityComparer.Equals(array[i], default))
53                 {
54                     continue;
55                 }
56                 var doubletOffset = i * 2;
57                 if (!_isElement(candidate))
58                 {
59                     nextArray[doubletOffset] = candidate;
60                 }
61                 else
62                 {
63                     var links = _links;
64                     var link = links.GetLink(candidate);
65                     var linkSource = links.GetSource(link);
66                     var linkTarget = links.GetTarget(link);
67                     nextArray[doubletOffset] = linkSource;
68                     nextArray[doubletOffset + 1] = linkTarget;
69                     if (!hasElements)
70                     {
71                         hasElements = !(_isElement(linkSource) && _isElement(linkTarget));
72                     }
73                 }
74             }
75 #if USEARRAYPOOL
76             if (array.Length > 1)
77             {
78                 ArrayPool.Free(array);
79             }
80 #endif
81             array = nextArray;
82         }
83         while (hasElements);
84         var filledElementsCount = CountFilledElements(array);
85         if (filledElementsCount == array.Length)
86         {
87             return array;
88         }
89         else
90         {
91             return CopyFilledElements(array, filledElementsCount);
92         }
93     }
94
95     [MethodImpl(MethodImplOptions.AggressiveInlining)]
96     private static TLink[] CopyFilledElements(TLink[] array, int filledElementsCount)
97     {
98         var finalArray = new TLink[filledElementsCount];
99         for (int i = 0, j = 0; i < array.Length; i++)
100         {
101             if (!_equalityComparer.Equals(array[i], default))
102             {
103                 finalArray[j] = array[i];
104                 j++;
105             }

```



```

106     }
107     #if USEARRAYPOOL
108         ArrayPool.Free(array);
109     #endif
110     return finalArray;
111 }
112
113 [MethodImpl(MethodImplOptions.AggressiveInlining)]
114 private static int CountFilledElements(TLink[] array)
115 {
116     var count = 0;
117     for (var i = 0; i < array.Length; i++)
118     {
119         if (!_equalityComparer.Equals(array[i], default))
120         {
121             count++;
122         }
123     }
124     return count;
125 }
126 }
127 }

```

1.59 ./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     public class RightSequenceWalker<TLink> : SequenceWalkerBase<TLink>
11     {
12         [MethodImpl(MethodImplOptions.AggressiveInlining)]
13         public RightSequenceWalker(ILinks<TLink> links, IStack<TLink> stack, Func<TLink, bool>
14             ↪ isElement) : base(links, stack, isElement) { }
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public RightSequenceWalker(ILinks<TLink> links, IStack<TLink> stack) : base(links,
18             ↪ stack, links.IsPartialPoint) { }
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         protected override TLink GetNextElementAfterPop(TLink element) =>
22             ↪ _links.GetTarget(element);
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         protected override TLink GetNextElementAfterPush(TLink element) =>
26             ↪ _links.GetSource(element);
27
28         [MethodImpl(MethodImplOptions.AggressiveInlining)]
29         protected override IEnumerable<TLink> WalkContents(TLink element)
30         {
31             var parts = _links.GetLink(element);
32             for (var i = _links.Constants.SourcePart; i < parts.Count; i++)
33             {
34                 var part = parts[i];
35                 if (IsElement(part))
36                 {
37                     yield return part;
38                 }
39             }
40         }
41     }
42 }

```

1.60 ./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     public abstract class SequenceWalkerBase<TLink> : LinksOperatorBase<TLink>,
11         ↪ ISequenceWalker<TLink>

```

```

11 {
12     private readonly IStack<TLink> _stack;
13     private readonly Func<TLink, bool> _isElement;
14
15     [MethodImpl(MethodImplOptions.AggressiveInlining)]
16     protected SequenceWalkerBase(ILinks<TLink> links, IStack<TLink> stack, Func<TLink, bool>
17     ↪ isElement) : base(links)
18     {
19         _stack = stack;
20         _isElement = isElement;
21     }
22
23     [MethodImpl(MethodImplOptions.AggressiveInlining)]
24     protected SequenceWalkerBase(ILinks<TLink> links, IStack<TLink> stack) : this(links,
25     ↪ stack, links.IsPartialPoint) { }
26
27     [MethodImpl(MethodImplOptions.AggressiveInlining)]
28     public IEnumerable<TLink> Walk(TLink sequence)
29     {
30         _stack.Clear();
31         var element = sequence;
32         if (IsElement(element))
33         {
34             yield return element;
35         }
36         else
37         {
38             while (true)
39             {
40                 if (IsElement(element))
41                 {
42                     if (_stack.IsEmpty)
43                     {
44                         break;
45                     }
46                     element = _stack.Pop();
47                     foreach (var output in WalkContents(element))
48                     {
49                         yield return output;
50                     }
51                     element = GetNextElementAfterPop(element);
52                 }
53                 else
54                 {
55                     _stack.Push(element);
56                     element = GetNextElementAfterPush(element);
57                 }
58             }
59         }
60     }
61
62     [MethodImpl(MethodImplOptions.AggressiveInlining)]
63     protected virtual bool IsElement(TLink elementLink) => _isElement(elementLink);
64
65     [MethodImpl(MethodImplOptions.AggressiveInlining)]
66     protected abstract TLink GetNextElementAfterPop(TLink element);
67
68     [MethodImpl(MethodImplOptions.AggressiveInlining)]
69     protected abstract TLink GetNextElementAfterPush(TLink element);
70
71     [MethodImpl(MethodImplOptions.AggressiveInlining)]
72     protected abstract IEnumerable<TLink> WalkContents(TLink element);
73 }

```

1.61 ./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 TLink = System.UInt64;
11
12 namespace Platform.Data.Doublets.Sequences.Tests
13 {
14     public class BigIntegerConvertersTests

```

```

15 {
16     public ILinks<TLink> CreateLinks() => CreateLinks<TLink>(new IO.TemporaryFile());
17
18     public ILinks<TLink> CreateLinks<TLink>(string dataDbFilename)
19     {
20         var linksConstants = new LinksConstants<TLink>(enableExternalReferencesSupport:
21             ↪ true);
22         return new UnitedMemoryLinks<TLink>(new
23             ↪ FileMappedResizableDirectMemory(dataDbFilename),
24             ↪ UnitedMemoryLinks<TLink>.DefaultLinksSizeStep, linksConstants,
25             ↪ IndexTreeType.Default);
26     }
27
28     [Fact]
29     public void DecimalMaxValueTest()
30     {
31         var links = CreateLinks();
32         BigInteger bigInteger = new(decimal.MaxValue);
33         TLink negativeNumberMarker = links.Create();
34         AddressToRawNumberConverter<TLink> addressToRawNumberConverter = new();
35         RawNumberToAddressConverter<TLink> numberToAddressConverter = new();
36         BalancedVariantConverter<TLink> listToSequenceConverter = new(links);
37         BigIntegerToRawNumberSequenceConverter<TLink> bigIntegerToRawNumberSequenceConverter
38             ↪ = new(links, addressToRawNumberConverter, listToSequenceConverter,
39             ↪ negativeNumberMarker);
40         RawNumberSequenceToBigIntegerConverter<TLink> rawNumberSequenceToBigIntegerConverter
41             ↪ = new(links, numberToAddressConverter, negativeNumberMarker);
42         var bigIntSequence = bigIntegerToRawNumberSequenceConverter.Convert(bigInteger);
43         var bigIntFromSequence =
44             ↪ rawNumberSequenceToBigIntegerConverter.Convert(bigIntSequence);
45         Assert.Equal(bigInteger, bigIntFromSequence);
46     }
47
48     [Fact]
49     public void DecimalMinValueTest()
50     {
51         var links = CreateLinks();
52         BigInteger bigInteger = new(decimal.MinValue);
53         TLink negativeNumberMarker = links.Create();
54         AddressToRawNumberConverter<TLink> addressToRawNumberConverter = new();
55         RawNumberToAddressConverter<TLink> numberToAddressConverter = new();
56         BalancedVariantConverter<TLink> listToSequenceConverter = new(links);
57         BigIntegerToRawNumberSequenceConverter<TLink> bigIntegerToRawNumberSequenceConverter
58             ↪ = new(links, addressToRawNumberConverter, listToSequenceConverter,
59             ↪ negativeNumberMarker);
60         RawNumberSequenceToBigIntegerConverter<TLink> rawNumberSequenceToBigIntegerConverter
61             ↪ = new(links, numberToAddressConverter, negativeNumberMarker);
62         var bigIntSequence = bigIntegerToRawNumberSequenceConverter.Convert(bigInteger);
63         var bigIntFromSequence =
64             ↪ rawNumberSequenceToBigIntegerConverter.Convert(bigIntSequence);
65         Assert.Equal(bigInteger, bigIntFromSequence);
66     }
67
68     [Fact]
69     public void ZeroValueTest()
70     {
71         var links = CreateLinks();
72         BigInteger bigInteger = new(0);
73         TLink negativeNumberMarker = links.Create();
74         AddressToRawNumberConverter<TLink> addressToRawNumberConverter = new();
75         RawNumberToAddressConverter<TLink> numberToAddressConverter = new();
76         BalancedVariantConverter<TLink> listToSequenceConverter = new(links);
77         BigIntegerToRawNumberSequenceConverter<TLink> bigIntegerToRawNumberSequenceConverter
78             ↪ = new(links, addressToRawNumberConverter, listToSequenceConverter,
79             ↪ negativeNumberMarker);
80         RawNumberSequenceToBigIntegerConverter<TLink> rawNumberSequenceToBigIntegerConverter
81             ↪ = new(links, numberToAddressConverter, negativeNumberMarker);
82         var bigIntSequence = bigIntegerToRawNumberSequenceConverter.Convert(bigInteger);
83         var bigIntFromSequence =
84             ↪ rawNumberSequenceToBigIntegerConverter.Convert(bigIntSequence);
85         Assert.Equal(bigInteger, bigIntFromSequence);
86     }
87
88     [Fact]
89     public void OneValueTest()
90     {
91         var links = CreateLinks();
92         BigInteger bigInteger = new(1);

```

```

77     TLink negativeNumberMarker = links.Create();
78     AddressToRawNumberConverter<TLink> addressToRawNumberConverter = new();
79     RawNumberToAddressConverter<TLink> numberToAddressConverter = new();
80     BalancedVariantConverter<TLink> listToSequenceConverter = new(links);
81     BigIntegerToRawNumberSequenceConverter<TLink> bigIntegerToRawNumberSequenceConverter
    ↪ = new(links, addressToRawNumberConverter, listToSequenceConverter,
    ↪ negativeNumberMarker);
82     RawNumberSequenceToBigIntegerConverter<TLink> 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 }

```

1.62 ./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 TLink = 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<TLink> CreateLinks() => CreateLinks<TLink>(new IO.TemporaryFile());
26
27         public static ILinks<TLink> CreateLinks<TLink>(string dataDBFilename)
28         {
29             var linksConstants = new LinksConstants<TLink>(enableExternalReferencesSupport:
    ↪ true);
30             return new UnitedMemoryLinks<TLink>(new
    ↪ FileMappedResizableDirectMemory(dataDBFilename),
    ↪ UnitedMemoryLinks<TLink>.DefaultLinksSizeStep, linksConstants,
    ↪ IndexTreeType.Default);
31         }
32
33         public class ValueCriterionMatcher<TLink> : ICriterionMatcher<TLink>
34         {
35             public readonly ILinks<TLink> Links;
36             public readonly TLink Marker;
37             public ValueCriterionMatcher(ILinks<TLink> links, TLink marker)
38             {
39                 Links = links;
40                 Marker = marker;
41             }
42
43             public bool IsMatched(TLink link) =>
    ↪ EqualityComparer<TLink>.Default.Equals(Links.GetSource(link), Marker);
44         }
45
46         [Fact]
47         public void AppendArrayBug()
48         {
49             ILinks<TLink> links = CreateLinks();
50             TLink zero = default;
51             var markerIndex = Arithmetic.Increment(zero);
52             var meaningRoot = links.GetOrCreate(markerIndex, markerIndex);
53             var sequence = links.Create();
54             sequence = links.Update(sequence, meaningRoot, sequence);
55             var appendant = links.Create();
56             appendant = links.Update(appendant, meaningRoot, appendant);
57             ValueCriterionMatcher<TLink> valueCriterionMatcher = new(links, meaningRoot);

```

```

58     DefaultSequenceRightHeightProvider<ulong> defaultSequenceRightHeightProvider =
        ↪ new(links, valueCriterionMatcher);
59     DefaultSequenceAppender<TLink> defaultSequenceAppender = new(links, new
        ↪ DefaultStack<ulong>(), defaultSequenceRightHeightProvider);
60     var newArray = defaultSequenceAppender.Append(sequence, appendant);
61     var output = links.FormatStructure(newArray, link => link.IsFullPoint(), true);
62     Assert.Equal("(4:(2:1 2) (3:1 3))", output);
63 }
64 }
65 }

```

1.63 ./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 }

```

1.64 ./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,
94             ↳ constants.Itself);
95
96         var unaryNumberToAddressConverter = new
97             ↳ UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);
98         var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links, unaryOne);
99         var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
100             ↳ frequencyMarker, unaryOne, unaryNumberIncrementer);
101         var frequencyPropertyOperator = new PropertyOperator<ulong>(links,
102             ↳ frequencyPropertyMarker, frequencyMarker);
103         var index = new FrequencyIncrementingSequenceIndex<ulong>(links,
104             ↳ frequencyPropertyOperator, frequencyIncrementer);
105         var linkToItsFrequencyNumberConverter = new
106             ↳ LinkToItsFrequencyNumberConverter<ulong>(links, frequencyPropertyOperator,
107             ↳ unaryNumberToAddressConverter);
108         var sequenceToItsLocalElementLevelsConverter = new
109             ↳ SequenceToItsLocalElementLevelsConverter<ulong>(links,
110             ↳ linkToItsFrequencyNumberConverter);
111         var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
112             ↳ sequenceToItsLocalElementLevelsConverter);
113
114         var sequences = new Sequences(links, new SequencesOptions<ulong>() { Walker =
115             ↳ new LevelledSequenceWalker<ulong>(links) });
116
117         ExecuteTest(sequences, sequence, sequenceToItsLocalElementLevelsConverter,
118             ↳ index, optimalVariantConverter);
119     }
120 }
121
122 [Fact]
123 public static void DictionaryBasedFrequencyStoredOptimalVariantSequenceTest()
124 {
125     using (var scope = new TempLinksTestScope(useSequences: false))
126     {

```

```

115     var links = scope.Links;
116
117     links.UseUnicode();
118
119     var sequence = UnicodeMap.FromStringToLinkArray(_sequenceExample);
120
121     var totalSequenceSymbolFrequencyCounter = new
122     ↪ TotalSequenceSymbolFrequencyCounter<ulong>(links);
123
124     var linkFrequenciesCache = new LinkFrequenciesCache<ulong>(links,
125     ↪ totalSequenceSymbolFrequencyCounter);
126
127     var index = new
128     ↪ CachedFrequencyIncrementingSequenceIndex<ulong>(linkFrequenciesCache);
129     var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFreque
130     ↪ ncyNumberConverter<ulong>(linkFrequenciesCache);
131
132     var sequenceToItsLocalElementLevelsConverter = new
133     ↪ SequenceToItsLocalElementLevelsConverter<ulong>(links,
134     ↪ linkToItsFrequencyNumberConverter);
135     var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
136     ↪ sequenceToItsLocalElementLevelsConverter);
137
138     var sequences = new Sequences(links, new SequencesOptions<ulong>() { Walker =
139     ↪ new LevelledSequenceWalker<ulong>(links) });
140
141     ExecuteTest(sequences, sequence, sequenceToItsLocalElementLevelsConverter,
142     ↪ index, optimalVariantConverter);
143 }
144
145 private static void ExecuteTest(Sequences sequences, ulong[] sequence,
146 ↪ SequenceToItsLocalElementLevelsConverter<ulong>
147 ↪ sequenceToItsLocalElementLevelsConverter, ISequenceIndex<ulong> index,
148 ↪ OptimalVariantConverter<ulong> optimalVariantConverter)
149 {
150     index.Add(sequence);
151
152     var optimalVariant = optimalVariantConverter.Convert(sequence);
153
154     var readSequence1 = sequences.ToList(optimalVariant);
155
156     Assert.True(sequence.SequenceEqual(readSequence1));
157 }
158
159 [Fact]
160 public static void SavedSequencesOptimizationTest()
161 {
162     LinksConstants<ulong> constants = new LinksConstants<ulong>((1, long.MaxValue),
163     ↪ (long.MaxValue + 1UL, ulong.MaxValue));
164
165     using (var memory = new HeapResizableDirectMemory())
166     using (var disposableLinks = new UInt64UnitedMemoryLinks(memory,
167     ↪ UInt64UnitedMemoryLinks.DefaultLinksSizeStep, constants, IndexTreeType.Default))
168     {
169         var links = new UInt64Links(disposableLinks);
170
171         var root = links.CreatePoint();
172
173         //var numberToAddressConverter = new RawNumberToAddressConverter<ulong>();
174         var addressToNumberConverter = new AddressToRawNumberConverter<ulong>();
175
176         var unicodeSymbolMarker = links.GetOrCreate(root,
177     ↪ addressToNumberConverter.Convert(1));
178         var unicodeSequenceMarker = links.GetOrCreate(root,
179     ↪ addressToNumberConverter.Convert(2));
180
181         var totalSequenceSymbolFrequencyCounter = new
182     ↪ TotalSequenceSymbolFrequencyCounter<ulong>(links);
183         var linkFrequenciesCache = new LinkFrequenciesCache<ulong>(links,
184     ↪ totalSequenceSymbolFrequencyCounter);
185         var index = new
186     ↪ CachedFrequencyIncrementingSequenceIndex<ulong>(linkFrequenciesCache);
187         var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFreque
188     ↪ ncyNumberConverter<ulong>(linkFrequenciesCache);
189         var sequenceToItsLocalElementLevelsConverter = new
190     ↪ SequenceToItsLocalElementLevelsConverter<ulong>(links,
191     ↪ linkToItsFrequencyNumberConverter);

```

```

171     var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
172         ↪ sequenceToItsLocalElementLevelsConverter);
173
174     var walker = new RightSequenceWalker<ulong>(links, new DefaultStack<ulong>(),
175         ↪ (link) => constants.IsExternalReference(link) || links.IsPartialPoint(link));
176
177     var unicodeSequencesOptions = new SequencesOptions<ulong>()
178     {
179         UseSequenceMarker = true,
180         SequenceMarkerLink = unicodeSequenceMarker,
181         UseIndex = true,
182         Index = index,
183         LinksToSequenceConverter = optimalVariantConverter,
184         Walker = walker,
185         UseGarbageCollection = true
186     };
187
188     var unicodeSequences = new Sequences(new SynchronizedLinks<ulong>(links),
189         ↪ unicodeSequencesOptions);
190
191     // Create some sequences
192     var strings = _loremIpsumExample.Split(new[] { '\n', '\r' },
193         ↪ StringSplitOptions.RemoveEmptyEntries);
194     var arrays = strings.Select(x => x.Select(y =>
195         ↪ addressToNumberConverter.Convert(y)).ToArray()).ToArray();
196     for (int i = 0; i < arrays.Length; i++)
197     {
198         unicodeSequences.Create(arrays[i].ShiftRight());
199     }
200
201     var linksCountAfterCreation = links.Count();
202
203     // get list of sequences links
204     // for each sequence link
205     //     create new sequence version
206     //     if new sequence is not the same as sequence link
207     //         delete sequence link
208     //         collect garbadge
209     unicodeSequences.CompactAll();
210
211     var linksCountAfterCompactification = links.Count();
212
213     Assert.True(linksCountAfterCompactification < linksCountAfterCreation);
214 }
215 }
216 }

```

1.65 ./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 TLink = System.UInt64;
10
11 namespace Platform.Data.Doublets.Sequences.Tests
12 {
13     public class RationalNumbersTests
14     {
15         public ILinks<TLink> CreateLinks() => CreateLinks<TLink>(new IO.TemporaryFile());
16
17         public ILinks<TLink> CreateLinks<TLink>(string dataDbFilename)
18         {
19             var linksConstants = new LinksConstants<TLink>(enableExternalReferencesSupport:
20                 ↪ true);
21             return new UnitedMemoryLinks<TLink>(new
22                 ↪ FileMappedResizableDirectMemory(dataDbFilename),
23                 ↪ UnitedMemoryLinks<TLink>.DefaultLinksSizeStep, linksConstants,
24                 ↪ IndexTreeType.Default);
25         }
26
27         [Fact]
28         public void DecimalMinValueTest()
29         {
30             const decimal @decimal = decimal.MinValue;
31             var links = CreateLinks();

```



```

28     TLink negativeNumberMarker = links.Create();
29     AddressToRawNumberConverter<TLink> addressToRawNumberConverter = new();
30     RawNumberToAddressConverter<TLink> numberToAddressConverter = new();
31     BalancedVariantConverter<TLink> balancedVariantConverter = new(links);
32     BigIntegerToRawNumberSequenceConverter<TLink> bigIntegerToRawNumberSequenceConverter
    ↪ = new(links, addressToRawNumberConverter, balancedVariantConverter,
    ↪ negativeNumberMarker);
33     RawNumberSequenceToBigIntegerConverter<TLink> rawNumberSequenceToBigIntegerConverter
    ↪ = new(links, numberToAddressConverter, negativeNumberMarker);
34     DecimalToRationalConverter<TLink> decimalToRationalConverter = new(links,
    ↪ bigIntegerToRawNumberSequenceConverter);
35     RationalToDecimalConverter<TLink> 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     TLink negativeNumberMarker = links.Create();
47     AddressToRawNumberConverter<TLink> addressToRawNumberConverter = new();
48     RawNumberToAddressConverter<TLink> numberToAddressConverter = new();
49     BalancedVariantConverter<TLink> balancedVariantConverter = new(links);
50     BigIntegerToRawNumberSequenceConverter<TLink> bigIntegerToRawNumberSequenceConverter
    ↪ = new(links, addressToRawNumberConverter, balancedVariantConverter,
    ↪ negativeNumberMarker);
51     RawNumberSequenceToBigIntegerConverter<TLink> rawNumberSequenceToBigIntegerConverter
    ↪ = new(links, numberToAddressConverter, negativeNumberMarker);
52     DecimalToRationalConverter<TLink> decimalToRationalConverter = new(links,
    ↪ bigIntegerToRawNumberSequenceConverter);
53     RationalToDecimalConverter<TLink> 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     TLink negativeNumberMarker = links.Create();
65     AddressToRawNumberConverter<TLink> addressToRawNumberConverter = new();
66     RawNumberToAddressConverter<TLink> numberToAddressConverter = new();
67     BalancedVariantConverter<TLink> balancedVariantConverter = new(links);
68     BigIntegerToRawNumberSequenceConverter<TLink> bigIntegerToRawNumberSequenceConverter
    ↪ = new(links, addressToRawNumberConverter, balancedVariantConverter,
    ↪ negativeNumberMarker);
69     RawNumberSequenceToBigIntegerConverter<TLink> rawNumberSequenceToBigIntegerConverter
    ↪ = new(links, numberToAddressConverter, negativeNumberMarker);
70     DecimalToRationalConverter<TLink> decimalToRationalConverter = new(links,
    ↪ bigIntegerToRawNumberSequenceConverter);
71     RationalToDecimalConverter<TLink> 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     TLink negativeNumberMarker = links.Create();
83     AddressToRawNumberConverter<TLink> addressToRawNumberConverter = new();
84     RawNumberToAddressConverter<TLink> numberToAddressConverter = new();
85     BalancedVariantConverter<TLink> balancedVariantConverter = new(links);
86     BigIntegerToRawNumberSequenceConverter<TLink> bigIntegerToRawNumberSequenceConverter
    ↪ = new(links, addressToRawNumberConverter, balancedVariantConverter,
    ↪ negativeNumberMarker);
87     RawNumberSequenceToBigIntegerConverter<TLink> rawNumberSequenceToBigIntegerConverter
    ↪ = new(links, numberToAddressConverter, negativeNumberMarker);

```

```

88     DecimalToRationalConverter<TLink> decimalToRationalConverter = new(links,
89         ↪ BigIntegerToRawNumberSequenceConverter);
90     RationalToDecimalConverter<TLink> rationalToDecimalConverter = new(links,
91         ↪ rawNumberSequenceToBigIntegerConverter);
92     var rationalNumber = decimalToRationalConverter.Convert(@decimal);
93     var decimalFromRational = rationalToDecimalConverter.Convert(rationalNumber);
94     Assert.Equal(@decimal, decimalFromRational);
95 }
96
97 [Fact]
98 public void DecimalOneTest()
99 {
100     const decimal @decimal = 1;
101     var links = CreateLinks();
102     TLink negativeNumberMarker = links.Create();
103     AddressToRawNumberConverter<TLink> addressToRawNumberConverter = new();
104     RawNumberToAddressConverter<TLink> numberToAddressConverter = new();
105     BalancedVariantConverter<TLink> balancedVariantConverter = new(links);
106     BigIntegerToRawNumberSequenceConverter<TLink> bigIntegerToRawNumberSequenceConverter
107         ↪ = new(links, addressToRawNumberConverter, balancedVariantConverter,
108         ↪ negativeNumberMarker);
109     RawNumberSequenceToBigIntegerConverter<TLink> rawNumberSequenceToBigIntegerConverter
110         ↪ = new(links, numberToAddressConverter, negativeNumberMarker);
111     DecimalToRationalConverter<TLink> decimalToRationalConverter = new(links,
112         ↪ BigIntegerToRawNumberSequenceConverter);
113     RationalToDecimalConverter<TLink> rationalToDecimalConverter = new(links,
114         ↪ rawNumberSequenceToBigIntegerConverter);
115     var rationalNumber = decimalToRationalConverter.Convert(@decimal);
116     var decimalFromRational = rationalToDecimalConverter.Convert(rationalNumber);
117     Assert.Equal(@decimal, decimalFromRational);
118 }
119
120 [Fact]
121 public void DecimalMinusOneTest()
122 {
123     const decimal @decimal = -1;
124     var links = CreateLinks();
125     TLink negativeNumberMarker = links.Create();
126     AddressToRawNumberConverter<TLink> addressToRawNumberConverter = new();
127     RawNumberToAddressConverter<TLink> numberToAddressConverter = new();
128     BalancedVariantConverter<TLink> balancedVariantConverter = new(links);
129     BigIntegerToRawNumberSequenceConverter<TLink> bigIntegerToRawNumberSequenceConverter
130         ↪ = new(links, addressToRawNumberConverter, balancedVariantConverter,
131         ↪ negativeNumberMarker);
132     RawNumberSequenceToBigIntegerConverter<TLink> rawNumberSequenceToBigIntegerConverter
133         ↪ = new(links, numberToAddressConverter, negativeNumberMarker);
134     DecimalToRationalConverter<TLink> decimalToRationalConverter = new(links,
135         ↪ BigIntegerToRawNumberSequenceConverter);
136     RationalToDecimalConverter<TLink> rationalToDecimalConverter = new(links,
137         ↪ rawNumberSequenceToBigIntegerConverter);
138     var rationalNumber = decimalToRationalConverter.Convert(@decimal);
139     var decimalFromRational = rationalToDecimalConverter.Convert(rationalNumber);
140     Assert.Equal(@decimal, decimalFromRational);
141 }
142 }
143 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Diagnostics;
4  using System.Linq;
5  using Xunit;
6  using Platform.Data.Sequences;
7  using Platform.Data.Doublets.Sequences.Converters;
8  using Platform.Data.Doublets.Sequences.Walkers;
9  using Platform.Data.Doublets.Sequences;
10
11 namespace Platform.Data.Doublets.Sequences.Tests
12 {
13     public static class ReadSequenceTests
14     {
15         [Fact]
16         public static void ReadSequenceTest()
17         {
18             const long sequenceLength = 2000;
19
20             using (var scope = new TempLinksTestScope(useSequences: false))
21             {

```

```

22     var links = scope.Links;
23     var sequences = new Sequences(links, new SequencesOptions<ulong> { Walker = new
    ↳ LeveledSequenceWalker<ulong>(links) });
24
25     var sequence = new ulong[sequenceLength];
26     for (var i = 0; i < sequenceLength; i++)
27     {
28         sequence[i] = links.Create();
29     }
30
31     var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
32
33     var sw1 = Stopwatch.StartNew();
34     var balancedVariant = balancedVariantConverter.Convert(sequence); sw1.Stop();
35
36     var sw2 = Stopwatch.StartNew();
37     var readSequence1 = sequences.ToList(balancedVariant); sw2.Stop();
38
39     var sw3 = Stopwatch.StartNew();
40     var readSequence2 = new List<ulong>();
41     SequenceWalker.WalkRight(balancedVariant,
42                             links.GetSource,
43                             links.GetTarget,
44                             links.IsPartialPoint,
45                             readSequence2.Add);
46
47     sw3.Stop();
48
49     Assert.True(sequence.SequenceEqual(readSequence1));
50
51     Assert.True(sequence.SequenceEqual(readSequence2));
52
53     // Assert.True(sw2.Elapsed < sw3.Elapsed);
54
55     Console.WriteLine($"Stack-based walker: {sw3.Elapsed}, Level-based reader:
    ↳ {sw2.Elapsed}");
56
57     for (var i = 0; i < sequenceLength; i++)
58     {
59         links.Delete(sequence[i]);
60     }
61 }
62 }
63 }

```

1.67 ./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 =
22             ↳ Default<LinksConstants<ulong>>.Instance;
23
24         static SequencesTests()
25         {
26             // Trigger static constructor to not mess with performance measurements
27             _ = BitString.GetBitMaskFromIndex(1);
28         }
29
30         [Fact]
31         public static void CreateAllVariantsTest()
32         {
33             const long sequenceLength = 8;
34
35             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,
73 //        ↪ DefaultLinksSizeStep))
74 //    using (var links = new Links(memoryAdapter))
75 //    {
76 //        var sequence = new ulong[sequenceLength];
77 //        for (var i = 0; i < sequenceLength; i++)
78 //            sequence[i] = links.Create(itself, itself);
79 //
80 //        SequencesOptions o = new SequencesOptions();
81 //
82 //        // TODO: Из числа в bool значения o.UseSequenceMarker = ((value & 1) != 0)
83 //        o.
84 //
85 //        var sequences = new Sequences(links);
86 //
87 //        var sw1 = Stopwatch.StartNew();
88 //        var results1 = sequences.CreateAllVariants1(sequence); sw1.Stop();
89 //
90 //        var sw2 = Stopwatch.StartNew();
91 //        var results2 = sequences.CreateAllVariants2(sequence); sw2.Stop();
92 //
93 //        Assert.True(results1.Count > results2.Length);
94 //        Assert.True(sw1.Elapsed > sw2.Elapsed);
95 //
96 //        for (var i = 0; i < sequenceLength; i++)
97 //            links.Delete(sequence[i]);
98 //    }
99 //
100 //    File.Delete(tempFilename);
101 //}
102
103 [Fact]
104 public static void AllVariantsSearchTest()
105 {
106     const long sequenceLength = 8;
107
108     using (var scope = new TempLinksTestScope(useSequences: true))
109     {
110         var links = scope.Links;
111         var sequences = scope.Sequences;
112
113         var sequence = new ulong[sequenceLength];
114         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 [Fact]
158 public static void BalancedVariantSearchTest()
159 {
160     const long sequenceLength = 200;
161
162     using (var scope = new TempLinksTestScope(useSequences: true))
163     {
164         var links = scope.Links;
165         var sequences = scope.Sequences;
166
167         var sequence = new ulong[sequenceLength];
168         for (var i = 0; i < sequenceLength; i++)
169         {
170             sequence[i] = links.Create();
171         }
172
173         var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
174
175         var sw1 = Stopwatch.StartNew();
176         var balancedVariant = balancedVariantConverter.Convert(sequence); sw1.Stop();
177
178         var sw2 = Stopwatch.StartNew();
179         var searchResults2 = sequences.GetAllMatchingSequences0(sequence); sw2.Stop();
180
181         var sw3 = Stopwatch.StartNew();
182         var searchResults3 = sequences.GetAllMatchingSequences1(sequence); sw3.Stop();
183
184         // На количестве в 200 элементов это будет занимать вечность
185         //var sw4 = Stopwatch.StartNew();
186         //var searchResults4 = sequences.Each(sequence); sw4.Stop();
187
188         Assert.True(searchResults2.Count == 1 && balancedVariant == searchResults2[0]);
189
190         Assert.True(searchResults3.Count == 1 && balancedVariant ==
191             ↪ 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 [Fact]
202 public static void AllPartialVariantsSearchTest()
203 {
204     const long sequenceLength = 8;
205
206     using (var scope = new TempLinksTestScope(useSequences: true))
207     {
208         var links = scope.Links;
209         var sequences = scope.Sequences;
210
211         var sequence = new ulong[sequenceLength];
212         for (var i = 0; i < sequenceLength; i++)
213         {
214             sequence[i] = links.Create();
215         }
216
217         var createResults = sequences.CreateAllVariants2(sequence);
218
219         //var createResultsStrings = createResults.Select(x => x + ": " +
220         ↪ 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 =
229         ↪ sequences.GetAllPartiallyMatchingSequences0(partialSequence); sw1.Stop();
230
231         var sw2 = Stopwatch.StartNew();
232         var searchResults2 =
233         ↪ sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();
234
235         //var sw3 = Stopwatch.StartNew();
236         //var searchResults3 =
237         ↪ sequences.GetAllPartiallyMatchingSequences2(partialSequence); sw3.Stop();
238
239         var sw4 = Stopwatch.StartNew();
240         var searchResults4 =
241         ↪ sequences.GetAllPartiallyMatchingSequences3(partialSequence); sw4.Stop();
242
243         //Global.Trash = searchResults3;
244
245         //var searchResults1Strings = searchResults1.Select(x => x + ": " +
246         ↪ sequences.FormatSequence(x)).ToList();
247         //Global.Trash = searchResults1Strings;
248
249         var intersection1 = createResults.Intersect(searchResults1).ToList();
250         Assert.True(intersection1.Count == createResults.Length);
251
252         var intersection2 = createResults.Intersect(searchResults2).ToList();
253         Assert.True(intersection2.Count == createResults.Length);
254
255         var intersection4 = createResults.Intersect(searchResults4).ToList();
256         Assert.True(intersection4.Count == createResults.Length);
257
258         for (var i = 0; i < sequenceLength; i++)
259         {
260             links.Delete(sequence[i]);
261         }
262     }
263 }
264
265 [Fact]
266 public static void BalancedPartialVariantsSearchTest()
267 {
268     const long sequenceLength = 200;
269
270     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         var balancedVariant = balancedVariantConverter.Convert(sequence);
278
279         var partialSequence = new ulong[sequenceLength - 2];
280
281         Array.Copy(sequence, 1, partialSequence, 0, (int)sequenceLength - 2);
282
283         var sw1 = Stopwatch.StartNew();
284         var searchResults1 =
285             ↪ sequences.GetAllPartiallyMatchingSequences0(partialSequence); sw1.Stop();
286
287         var sw2 = Stopwatch.StartNew();
288         var searchResults2 =
289             ↪ sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();
290
291         Assert.True(searchResults1.Count == 1 && balancedVariant == searchResults1[0]);
292
293         Assert.True(searchResults2.Count == 1 && balancedVariant ==
294             ↪ searchResults2.First());
295
296         for (var i = 0; i < sequenceLength; i++)
297         {
298             links.Delete(sequence[i]);
299         }
300     }
301
302     [Fact(Skip = "Correct implementation is pending")]
303     public static void PatternMatchTest()
304     {
305         var zeroOrMany = Sequences.ZeroOrMany;
306
307         using (var scope = new TempLinksTestScope(useSequences: true))
308         {
309             var links = scope.Links;
310             var sequences = scope.Sequences;
311
312             var e1 = links.Create();
313             var e2 = links.Create();
314
315             var sequence = new[]
316             {
317                 e1, e2, e1, e2 // mama / papa
318             };
319
320             var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
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(douplet, 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
379 /// <summary>Imported from https://raw.githubusercontent.com/wiki/Konard/LinksPlatform/%
        ↳ DO%9E-%D1%82%DO%BE%DO%BC%2C-%DO%BA%DO%B0%DO%BA-%DO%B2%D1%81%D1%91-%DO%BD%DO%B0%D1%87
        ↳ %DO%B8%DO%BD%DO%B0%DO%BB%DO%BE%D1%81%D1%8C.md</summary>
380 private static readonly string _exampleText =
381     @"([english
        ↳ version](https://github.com/Konard/LinksPlatform/wiki/About-the-beginning))
382
383 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?
384
385 [![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)
386
387 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?
388
389 [![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)
390
391 А что если повторить? Сделать копию? Создать дубликат? Из одного сделать два? Может это быть
        ↳ так? Инверсия? Отражение? Сумма?
392
393 [![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)
394
395 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?
396
397 [![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)
398

```


399 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?

400

401 `[[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)`

402

403 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?

404

405 `[[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)`

406

407 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?

408

409 `[[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)`

410

411 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?

412

413 `[[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)`

414

415 Допустим у нас есть смысл "связать" и смысл "направления", много ли это нам даёт? Много ли
→ вариантов интерпретаций? 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?

416

417 `[[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)`

418

419 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?

420

421 `[[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)`

422

423 На один уровень внутрь (вниз)? Или на один уровень во вне (вверх)? Или это можно назвать шагом
→ рекурсии или фрактала?

424

425 `[[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)`

426

427 Последовательность? Массив? Список? Множество? Объект? Таблица? Элементы? Цвета? Символы? Буквы?
→ Слово? Цифры? Число? Алфавит? Дерево? Сеть? Граф? Гиперграф?

428

```

429  [![белая обычная и направленная связи со структурой из 8 цветных элементов последовательности,
      ↳ чёрная типизированная связь со структурой из 8 цветных элементов последовательности](https://
      ↳ /raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/12.png "белая обычная и
      ↳ направленная связи со структурой из 8 цветных элементов последовательности, чёрная
      ↳ типизированная связь со структурой из 8 цветных элементов последовательности")](https://raw
      ↳ .githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/12.png)
430
431  ...
432
433  [![анимация](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro-anima
      ↳ tion-500.gif
      ↳ "анимация")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro
      ↳ -animation-500.gif)";
434
435      private static readonly string _exampleLoremIpsumText =
436          @"Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
            ↳ incididunt ut labore et dolore magna aliqua.
437  Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo
            ↳ consequat.";
438
439  [Fact]
440  public static void CompressionTest()
441  {
442      using (var scope = new TempLinksTestScope(useSequences: true))
443      {
444          var links = scope.Links;
445          var sequences = scope.Sequences;
446
447          var e1 = links.Create();
448          var e2 = links.Create();
449
450          var sequence = new[]
451          {
452              e1, e2, e1, e2 // mama / papa / template [(m/p), a] { [1] [2] [1] [2] }
453          };
454
455          var balancedVariantConverter = new BalancedVariantConverter<ulong>(links.Unsync);
456          var totalSequenceSymbolFrequencyCounter = new
            ↳ TotalSequenceSymbolFrequencyCounter<ulong>(links.Unsync);
457          var doubletFrequenciesCache = new LinkFrequenciesCache<ulong>(links.Unsync,
            ↳ totalSequenceSymbolFrequencyCounter);
458          var compressingConverter = new CompressingConverter<ulong>(links.Unsync,
            ↳ balancedVariantConverter, doubletFrequenciesCache);
459
460          var compressedVariant = compressingConverter.Convert(sequence);
461
462          // 1: [1]          (1->1) point
463          // 2: [2]          (2->2) point
464          // 3: [1,2]        (1->2) doublet
465          // 4: [1,2,1,2]    (3->3) doublet
466
467          Assert.True(links.GetSource(links.GetSource(compressedVariant)) == sequence[0]);
468          Assert.True(links.GetTarget(links.GetSource(compressedVariant)) == sequence[1]);
469          Assert.True(links.GetSource(links.GetTarget(compressedVariant)) == sequence[2]);
470          Assert.True(links.GetTarget(links.GetTarget(compressedVariant)) == sequence[3]);
471
472          var source = _constants.SourcePart;
473          var target = _constants.TargetPart;
474
475          Assert.True(links.GetByKeys(compressedVariant, source, source) == sequence[0]);
476          Assert.True(links.GetByKeys(compressedVariant, source, target) == sequence[1]);
477          Assert.True(links.GetByKeys(compressedVariant, target, source) == sequence[2]);
478          Assert.True(links.GetByKeys(compressedVariant, target, target) == sequence[3]);
479
480          // 4 - length of sequence
481          Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 0)
            ↳ == sequence[0]);
482          Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 1)
            ↳ == sequence[1]);
483          Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 2)
            ↳ == sequence[2]);
484          Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 3)
            ↳ == sequence[3]);
485      }
486  }
487
488  [Fact]
489  public static void CompressionEfficiencyTest()

```

```

490 {
491     var strings = _exampleLoremIpsumText.Split(new[] { '\n', '\r' },
        ↳ StringSplitOptions.RemoveEmptyEntries);
492     var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
493     var totalCharacters = arrays.Select(x => x.Length).Sum();
494
495     using (var scope1 = new TempLinksTestScope(useSequences: true))
496     using (var scope2 = new TempLinksTestScope(useSequences: true))
497     using (var scope3 = new TempLinksTestScope(useSequences: true))
498     {
499         scope1.Links.Unsync.UseUnicode();
500         scope2.Links.Unsync.UseUnicode();
501         scope3.Links.Unsync.UseUnicode();
502
503         var balancedVariantConverter1 = new
        ↳ BalancedVariantConverter<ulong>(scope1.Links.Unsync);
504         var totalSequenceSymbolFrequencyCounter = new
        ↳ TotalSequenceSymbolFrequencyCounter<ulong>(scope1.Links.Unsync);
505         var linkFrequenciesCache1 = new LinkFrequenciesCache<ulong>(scope1.Links.Unsync,
        ↳ totalSequenceSymbolFrequencyCounter);
506         var compressor1 = new CompressingConverter<ulong>(scope1.Links.Unsync,
        ↳ balancedVariantConverter1, linkFrequenciesCache1,
        ↳ doInitialFrequenciesIncrement: false);
507
508         //var compressor2 = scope2.Sequences;
509         var compressor3 = scope3.Sequences;
510
511         var constants = Default<LinksConstants<ulong>>.Instance;
512
513         var sequences = compressor3;
514         //var meaningRoot = links.CreatePoint();
515         //var unaryOne = links.CreateAndUpdate(meaningRoot, constants.Itself);
516         //var frequencyMarker = links.CreateAndUpdate(meaningRoot, constants.Itself);
517         //var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot,
        ↳ constants.Itself);
518
519         //var unaryNumberToAddressConverter = new
        ↳ UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);
520         //var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links,
        ↳ unaryOne);
521         //var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
        ↳ frequencyMarker, unaryOne, unaryNumberIncrementer);
522         //var frequencyPropertyOperator = new FrequencyPropertyOperator<ulong>(links,
        ↳ frequencyPropertyMarker, frequencyMarker);
523         //var linkFrequencyIncrementer = new LinkFrequencyIncrementer<ulong>(links,
        ↳ frequencyPropertyOperator, frequencyIncrementer);
524         //var linkToItsFrequencyNumberConverter = new
        ↳ LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
        ↳ unaryNumberToAddressConverter);
525
526         var linkFrequenciesCache3 = new LinkFrequenciesCache<ulong>(scope3.Links.Unsync,
        ↳ totalSequenceSymbolFrequencyCounter);
527
528         var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFreque
        ↳ ncyNumberConverter<ulong>(linkFrequenciesCache3);
529
530         var sequenceToItsLocalElementLevelsConverter = new
        ↳ SequenceToItsLocalElementLevelsConverter<ulong>(scope3.Links.Unsync,
        ↳ linkToItsFrequencyNumberConverter);
531         var optimalVariantConverter = new
        ↳ OptimalVariantConverter<ulong>(scope3.Links.Unsync,
        ↳ sequenceToItsLocalElementLevelsConverter);
532
533         var compressed1 = new ulong[arrays.Length];
534         var compressed2 = new ulong[arrays.Length];
535         var compressed3 = new ulong[arrays.Length];
536
537         var START = 0;
538         var END = arrays.Length;
539
540         //for (int i = START; i < END; i++)
541         //    linkFrequenciesCache1.IncrementFrequencies(arrays[i]);
542
543         var initialCount1 = scope2.Links.Unsync.Count();
544
545         var sw1 = Stopwatch.StartNew();
546
547         for (int i = START; i < END; i++)

```

```

548     {
549         linkFrequenciesCache1.IncrementFrequencies(arrays[i]);
550         compressed1[i] = compressor1.Convert(arrays[i]);
551     }
552
553     var elapsed1 = sw1.Elapsed;
554
555     var balancedVariantConverter2 = new
556     ↪ BalancedVariantConverter<ulong>(scope2.Links.Unsync);
557
558     var initialCount2 = scope2.Links.Unsync.Count();
559
560     var sw2 = Stopwatch.StartNew();
561
562     for (int i = START; i < END; i++)
563     {
564         compressed2[i] = balancedVariantConverter2.Convert(arrays[i]);
565     }
566
567     var elapsed2 = sw2.Elapsed;
568
569     for (int i = START; i < END; i++)
570     {
571         linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
572     }
573
574     var initialCount3 = scope3.Links.Unsync.Count();
575
576     var sw3 = Stopwatch.StartNew();
577
578     for (int i = START; i < END; i++)
579     {
580         //linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
581         compressed3[i] = optimalVariantConverter.Convert(arrays[i]);
582     }
583
584     var elapsed3 = sw3.Elapsed;
585
586     Console.WriteLine($"Compressor: {elapsed1}, Balanced variant: {elapsed2},
587     ↪ Optimal variant: {elapsed3}");
588
589     // Assert.True(elapsed1 > elapsed2);
590
591     // Checks
592     for (int i = START; i < END; i++)
593     {
594         var sequence1 = compressed1[i];
595         var sequence2 = compressed2[i];
596         var sequence3 = compressed3[i];
597
598         var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
599         ↪ scope1.Links.Unsync);
600
601         var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
602         ↪ scope2.Links.Unsync);
603
604         var decompress3 = UnicodeMap.FromSequenceLinkToString(sequence3,
605         ↪ scope3.Links.Unsync);
606
607         var structure1 = scope1.Links.Unsync.FormatStructure(sequence1, link =>
608         ↪ link.IsPartialPoint());
609         var structure2 = scope2.Links.Unsync.FormatStructure(sequence2, link =>
610         ↪ link.IsPartialPoint());
611         var structure3 = scope3.Links.Unsync.FormatStructure(sequence3, link =>
612         ↪ link.IsPartialPoint());
613
614         //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
615         ↪ arrays[i].Length > 3)
616         //    Assert.False(structure1 == structure2);
617         //if (sequence3 != Constants.Null && sequence2 != Constants.Null &&
618         ↪ arrays[i].Length > 3)
619         //    Assert.False(structure3 == structure2);
620
621         Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
622         Assert.True(strings[i] == decompress3 && decompress3 == decompress2);
623     }
624
625     Assert.True((int)(scope1.Links.Unsync.Count() - initialCount1) <
626     ↪ totalCharacters);

```

```

616 Assert.True((int)(scope2.Links.Unsync.Count() - initialCount2) <
    ↳ totalCharacters);
617 Assert.True((int)(scope3.Links.Unsync.Count() - initialCount3) <
    ↳ totalCharacters);
618
619 Console.WriteLine($"{(double)(scope1.Links.Unsync.Count() - initialCount1) /
    ↳ totalCharacters} | {(double)(scope2.Links.Unsync.Count() - initialCount2) /
    ↳ totalCharacters} | {(double)(scope3.Links.Unsync.Count() - initialCount3) /
    ↳ totalCharacters}");
620
621 Assert.True(scope1.Links.Unsync.Count() - initialCount1 <
    ↳ scope2.Links.Unsync.Count() - initialCount2);
622 Assert.True(scope3.Links.Unsync.Count() - initialCount3 <
    ↳ scope2.Links.Unsync.Count() - initialCount2);
623
624 var duplicateProvider1 = new
    ↳ DuplicateSegmentsProvider<ulong>(scope1.Links.Unsync, scope1.Sequences);
625 var duplicateProvider2 = new
    ↳ DuplicateSegmentsProvider<ulong>(scope2.Links.Unsync, scope2.Sequences);
626 var duplicateProvider3 = new
    ↳ DuplicateSegmentsProvider<ulong>(scope3.Links.Unsync, scope3.Sequences);
627
628 var duplicateCounter1 = new DuplicateSegmentsCounter<ulong>(duplicateProvider1);
629 var duplicateCounter2 = new DuplicateSegmentsCounter<ulong>(duplicateProvider2);
630 var duplicateCounter3 = new DuplicateSegmentsCounter<ulong>(duplicateProvider3);
631
632 var duplicates1 = duplicateCounter1.Count();
633
634 ConsoleHelpers.Debug("-----");
635
636 var duplicates2 = duplicateCounter2.Count();
637
638 ConsoleHelpers.Debug("-----");
639
640 var duplicates3 = duplicateCounter3.Count();
641
642 Console.WriteLine($"{duplicates1} | {duplicates2} | {duplicates3}");
643
644 linkFrequenciesCache1.ValidateFrequencies();
645 linkFrequenciesCache3.ValidateFrequencies();
646 }
647 }
648
649 [Fact]
650 public static void CompressionStabilityTest()
651 {
652     // TODO: Fix bug (do a separate test)
653     //const ulong minNumbers = 0;
654     //const ulong maxNumbers = 1000;
655
656     const ulong minNumbers = 10000;
657     const ulong maxNumbers = 12500;
658
659     var strings = new List<string>();
660
661     for (ulong i = minNumbers; i < maxNumbers; i++)
662     {
663         strings.Add(i.ToString());
664     }
665
666     var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
667     var totalCharacters = arrays.Select(x => x.Length).Sum();
668
669     using (var scope1 = new TempLinksTestScope(useSequences: true, sequencesOptions: new
    ↳ SequencesOptions<ulong> { UseCompression = true,
    ↳ EnforceSingleSequenceVersionOnWriteBasedOnExisting = true }))
670     using (var scope2 = new TempLinksTestScope(useSequences: true))
671     {
672         scope1.Links.UseUnicode();
673         scope2.Links.UseUnicode();
674
675         //var compressor1 = new Compressor(scope1.Links.Unsync, scope1.Sequences);
676         var compressor1 = scope1.Sequences;
677         var compressor2 = scope2.Sequences;
678
679         var compressed1 = new ulong[arrays.Length];
680         var compressed2 = new ulong[arrays.Length];
681
682         var sw1 = Stopwatch.StartNew();

```

```

683
684 var START = 0;
685 var END = arrays.Length;
686
687 // Collisions proved (cannot be solved by max doublet comparison, no stable 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 BalancedVariantConverter<ulong>(scope2.Links);
720
721 var sw2 = Stopwatch.StartNew();
722
723 for (int i = START; i < END; i++)
724 {
725     var first = balancedVariantConverter.Convert(arrays[i]);
726     var second = balancedVariantConverter.Convert(arrays[i]);
727
728     if (first == second)
729     {
730         compressed2[i] = first;
731     }
732 }
733
734 var elapsed2 = sw2.Elapsed;
735
736 Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator:
737 ↪ {elapsed2}");
738
739 Assert.True(elapsed1 > elapsed2);
740
741 // Checks
742 for (int i = START; i < END; i++)
743 {
744     var sequence1 = compressed1[i];
745     var sequence2 = compressed2[i];
746
747     if (sequence1 != _constants.Null && sequence2 != _constants.Null)
748     {
749         var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
750 ↪ scope1.Links);
751
752         var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
753 ↪ scope2.Links);
754
755         //var structure1 = scope1.Links.FormatStructure(sequence1, link =>
756 ↪ link.IsPartialPoint());
757         //var structure2 = scope2.Links.FormatStructure(sequence2, link =>
758 ↪ link.IsPartialPoint());
759
760         //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
761 ↪ arrays[i].Length > 3)

```

```

756         // Assert.False(structure1 == structure2);
757
758         Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
759     }
760 }
761
762 Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
763 Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
764
765 Debug.WriteLine($"{(double)(scope1.Links.Count() - UnicodeMap.MapSize) /
    ↳ totalCharacters} | {(double)(scope2.Links.Count() - UnicodeMap.MapSize) /
    ↳ totalCharacters}");
766
767 Assert.True(scope1.Links.Count() <= scope2.Links.Count());
768
769 //compressor1.ValidateFrequencies();
770 }
771 }
772
773 [Fact]
774 public static void RandomNumbersCompressionQualityTest()
775 {
776     const ulong N = 500;
777
778     //const ulong minNumbers = 10000;
779     //const ulong maxNumbers = 20000;
780
781     //var strings = new List<string>();
782
783     //for (ulong i = 0; i < N; i++)
784     //    strings.Add(RandomHelpers.DefaultFactory.NextUInt64(minNumbers,
785     ↳ maxNumbers).ToString());
786
787     var strings = new List<string>();
788
789     for (ulong i = 0; i < N; i++)
790     {
791         strings.Add(RandomHelpers.Default.NextUInt64().ToString());
792     }
793
794     strings = strings.Distinct().ToList();
795
796     var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
797     var totalCharacters = arrays.Select(x => x.Length).Sum();
798
799     using (var scope1 = new TempLinksTestScope(useSequences: true, sequencesOptions: new
800     ↳ SequencesOptions<ulong> { UseCompression = true,
801     ↳ EnforceSingleSequenceVersionOnWriteBasedOnExisting = true }))
802     using (var scope2 = new TempLinksTestScope(useSequences: true))
803     {
804         scope1.Links.UseUnicode();
805         scope2.Links.UseUnicode();
806
807         var compressor1 = scope1.Sequences;
808         var compressor2 = scope2.Sequences;
809
810         var compressed1 = new ulong[arrays.Length];
811         var compressed2 = new ulong[arrays.Length];
812
813         var sw1 = Stopwatch.StartNew();
814
815         var START = 0;
816         var END = arrays.Length;
817
818         for (int i = START; i < END; i++)
819         {
820             compressed1[i] = compressor1.Create(arrays[i].ShiftRight());
821         }
822
823         var elapsed1 = sw1.Elapsed;
824
825         var balancedVariantConverter = new BalancedVariantConverter<ulong>(scope2.Links);
826
827         var sw2 = Stopwatch.StartNew();
828
829         for (int i = START; i < END; i++)
830         {
831             compressed2[i] = balancedVariantConverter.Convert(arrays[i]);
832         }
833     }

```

```

831     var elapsed2 = sw2.Elapsed;
832
833     Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator:
834         ↳ {elapsed2}");
835
836     Assert.True(elapsed1 > elapsed2);
837
838     // Checks
839     for (int i = START; i < END; i++)
840     {
841         var sequence1 = compressed1[i];
842         var sequence2 = compressed2[i];
843
844         if (sequence1 != _constants.Null && sequence2 != _constants.Null)
845         {
846             var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
847                 ↳ scope1.Links);
848
849             var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
850                 ↳ scope2.Links);
851
852             Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
853         }
854     }
855
856     Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
857     Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
858
859     Debug.WriteLine($"{{(double)(scope1.Links.Count() - UnicodeMap.MapSize) /
860         ↳ totalCharacters}} | {{(double)(scope2.Links.Count() - UnicodeMap.MapSize) /
861         ↳ totalCharacters}}");
862
863     // Can be worse than balanced variant
864     //Assert.True(scope1.Links.Count() <= scope2.Links.Count());
865
866     //compressor1.ValidateFrequencies();
867 }
868
869 [Fact]
870 public static void AllTreeBreakDownAtSequencesCreationBugTest()
871 {
872     // Made out of AllPossibleConnectionsTest test.
873
874     //const long sequenceLength = 5; //100% bug
875     const long sequenceLength = 4; //100% bug
876     //const long sequenceLength = 3; //100% _no_bug (ok)
877
878     using (var scope = new TempLinksTestScope(useSequences: true))
879     {
880         var links = scope.Links;
881         var sequences = scope.Sequences;
882
883         var sequence = new ulong[sequenceLength];
884         for (var i = 0; i < sequenceLength; i++)
885         {
886             sequence[i] = links.Create();
887         }
888
889         var createResults = sequences.CreateAllVariants2(sequence);
890
891         Global.Trash = createResults;
892
893         for (var i = 0; i < sequenceLength; i++)
894         {
895             links.Delete(sequence[i]);
896         }
897     }
898 }
899
900 [Fact]
901 public static void AllPossibleConnectionsTest()
902 {
903     const long sequenceLength = 5;
904
905     using (var scope = new TempLinksTestScope(useSequences: true))
906     {
907         var links = scope.Links;
908         var sequences = scope.Sequences;

```



```

905
906     var sequence = new ulong[sequenceLength];
907     for (var i = 0; i < sequenceLength; i++)
908     {
909         sequence[i] = links.Create();
910     }
911
912     var createResults = sequences.CreateAllVariants2(sequence);
913     var reverseResults = sequences.CreateAllVariants2(sequence.Reverse().ToArray());
914
915     for (var i = 0; i < 1; i++)
916     {
917         var sw1 = Stopwatch.StartNew();
918         var searchResults1 = sequences.GetAllConnections(sequence); sw1.Stop();
919
920         var sw2 = Stopwatch.StartNew();
921         var searchResults2 = sequences.GetAllConnections1(sequence); sw2.Stop();
922
923         var sw3 = Stopwatch.StartNew();
924         var searchResults3 = sequences.GetAllConnections2(sequence); sw3.Stop();
925
926         var sw4 = Stopwatch.StartNew();
927         var searchResults4 = sequences.GetAllConnections3(sequence); sw4.Stop();
928
929         Global.Trash = searchResults3;
930         Global.Trash = searchResults4; //-V3008
931
932         var intersection1 = createResults.Intersect(searchResults1).ToList();
933         Assert.True(intersection1.Count == createResults.Length);
934
935         var intersection2 = reverseResults.Intersect(searchResults1).ToList();
936         Assert.True(intersection2.Count == reverseResults.Length);
937
938         var intersection0 = searchResults1.Intersect(searchResults2).ToList();
939         Assert.True(intersection0.Count == searchResults2.Count);
940
941         var intersection3 = searchResults2.Intersect(searchResults3).ToList();
942         Assert.True(intersection3.Count == searchResults3.Count);
943
944         var intersection4 = searchResults3.Intersect(searchResults4).ToList();
945         Assert.True(intersection4.Count == searchResults4.Count);
946     }
947
948     for (var i = 0; i < sequenceLength; i++)
949     {
950         links.Delete(sequence[i]);
951     }
952 }
953
954 [Fact(Skip = "Correct implementation is pending")]
955 public static void CalculateAllUsagesTest()
956 {
957     const long sequenceLength = 3;
958
959     using (var scope = new TempLinksTestScope(useSequences: true))
960     {
961         var links = scope.Links;
962         var sequences = scope.Sequences;
963
964         var sequence = new ulong[sequenceLength];
965         for (var i = 0; i < sequenceLength; i++)
966         {
967             sequence[i] = links.Create();
968         }
969
970         var createResults = sequences.CreateAllVariants2(sequence);
971
972         //var reverseResults =
973         ↪ sequences.CreateAllVariants2(sequence.Reverse().ToArray());
974
975         for (var i = 0; i < 1; i++)
976         {
977             var linksTotalUsages1 = new ulong[links.Count() + 1];
978
979             sequences.CalculateAllUsages(linksTotalUsages1);
980
981             var linksTotalUsages2 = new ulong[links.Count() + 1];
982
983             sequences.CalculateAllUsages2(linksTotalUsages2);

```

```

984         var intersection1 = linksTotalUsages1.Intersect(linksTotalUsages2).ToList();
985         Assert.True(intersection1.Count == linksTotalUsages2.Length);
986     }
987 }
988
989 for (var i = 0; i < sequenceLength; i++)
990 {
991     links.Delete(sequence[i]);
992 }
993 }
994 }
995 }
996 }

```

1.68 ./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 }

```

1.69 ./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<TLink>(this ILinks<TLink>
    ↪ links, int maximumOperationsPerCycle)
158 {
159     var comparer = Comparer<TLink>.Default;

```

```

160     var addressToUInt64Converter = CheckedConverter<TLink, ulong>.Default;
161     var uInt64ToAddressConverter = CheckedConverter<ulong, TLink>.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                 TLink source = uInt64ToAddressConverter.Convert(random.NextUInt64(linksA_
                    ↪ ddressRange));
175                 TLink target = uInt64ToAddressConverter.Convert(random.NextUInt64(linksA_
                    ↪ ddressRange));
176                 ↪ //-V3086
177                 var resultLink = links.GetOrCreate(source, target);
178                 if (comparer.Compare(resultLink,
                    ↪ uInt64ToAddressConverter.Convert(linksCount)) > 0)
179                 {
180                     created++;
181                 }
182             }
183             else
184             {
185                 links.Create();
186                 created++;
187             }
188             Assert.True(created == addressToUInt64Converter.Convert(links.Count()));
189             for (var i = 0; i < N; i++)
190             {
191                 TLink 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 }

```

1.70 ./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
30         private const long Iterations = 10 * 1024;

```

#region Concept

[Fact]

public static void MultipleCreateAndDeleteTest()

```
{
    using (var scope = new Scope<Types<HeapResizableDirectMemory,
        ↳ UInt64UnitedMemoryLinks>>())
    {
        new UInt64Links(scope.Use<ILinks<ulong>>()).TestMultipleRandomCreationsAndDeletions(100);
    }
}
```

[Fact]

public static void CascadeUpdateTest()

```
{
    var itself = _constants.Itself;
    using (var scope = new TempLinksTestScope(useLog: true))
    {
        var links = scope.Links;

        var l1 = links.Create();
        var l2 = links.Create();

        l2 = links.Update(l2, l2, l1, l2);

        links.CreateAndUpdate(l2, itself);
        links.CreateAndUpdate(l2, itself);

        l2 = links.Update(l2, l1);

        links.Delete(l2);

        Global.Trash = links.Count();

        links.Unsync.DisposeIfPossible(); // Close links to access log

        Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scope
            ↳ e.TempTransactionLogFilename);
    }
}
```

[Fact]

public static void BasicTransactionLogTest()

```
{
    using (var scope = new TempLinksTestScope(useLog: true))
    {
        var links = scope.Links;
        var l1 = links.Create();
        var l2 = links.Create();

        Global.Trash = links.Update(l2, l2, l1, l2);

        links.Delete(l1);

        links.Unsync.DisposeIfPossible(); // Close links to access log

        Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scope
            ↳ e.TempTransactionLogFilename);
    }
}
```

[Fact]

public static void TransactionAutoRevertedTest()

```
{
    // Auto Reverted (Because no commit at transaction)
    using (var scope = new TempLinksTestScope(useLog: true))
    {
        var links = scope.Links;
        var transactionsLayer = (UInt64LinksTransactionsLayer)scope.MemoryAdapter;
        using (var transaction = transactionsLayer.BeginTransaction())
        {
            var l1 = links.Create();
            var l2 = links.Create();

            links.Update(l2, l2, l1, l2);
        }

        Assert.Equal(0UL, links.Count());
    }
}
```

```

links.Unsync.DisposeIfPossible();

var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scope.TempTransactionLogFilename);
Assert.Single(transitions);
}
}

[Fact]
public static void TransactionUserCodeErrorNoDataSavedTest()
{
    // User Code Error (Autoreverted), no data saved
    var itself = _constants.Itself;

    TempLinksTestScope lastScope = null;
    try
    {
        using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
            useLog: true))
        {
            var links = scope.Links;
            var transactionsLayer = (UInt64LinksTransactionsLayer)((LinksDisposableDecoratorBase<ulong>)links.Unsync).Links;
            using (var transaction = transactionsLayer.BeginTransaction())
            {
                var l1 = links.CreateAndUpdate(itself, itself);
                var l2 = links.CreateAndUpdate(itself, itself);

                l2 = links.Update(l2, l2, l1, l2);

                links.CreateAndUpdate(l2, itself);
                links.CreateAndUpdate(l2, itself);

                //Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scope.TempTransactionLogFilename);

                l2 = links.Update(l2, l1);

                links.Delete(l2);

                ExceptionThrower();

                transaction.Commit();
            }

            Global.Trash = links.Count();
        }
    }
    catch
    {
        Assert.False(lastScope == null);

        var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(lastScope.TempTransactionLogFilename);

        Assert.True(transitions.Length == 1 && transitions[0].Before.IsNull() &&
            transitions[0].After.IsNull());

        lastScope.DeleteFiles();
    }
}

[Fact]
public static void TransactionUserCodeErrorSomeDataSavedTest()
{
    // User Code Error (Autoreverted), some data saved
    var itself = _constants.Itself;

    TempLinksTestScope lastScope = null;
    try
    {
        ulong l1;
        ulong l2;

        using (var scope = new TempLinksTestScope(useLog: true))
        {
            var links = scope.Links;
            l1 = links.CreateAndUpdate(itself, itself);

```

```

179         l2 = links.CreateAndUpdate(itself, itself);
180
181         l2 = links.Update(l2, l2, l1, l2);
182
183         links.CreateAndUpdate(l2, itself);
184         links.CreateAndUpdate(l2, itself);
185
186         links.Unsync.DisposeIfPossible();
187
188         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(
            ↪ scope.TempTransactionLogFilename);
189     }
190
191     using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
            ↪ useLog: true))
192     {
193         var links = scope.Links;
194         var transactionsLayer = (UInt64LinksTransactionsLayer)links.Unsync;
195         using (var transaction = transactionsLayer.BeginTransaction())
196         {
197             l2 = links.Update(l2, l1);
198
199             links.Delete(l2);
200
201             ExceptionThrower();
202
203             transaction.Commit();
204         }
205
206         Global.Trash = links.Count();
207     }
208 }
209 catch
210 {
211     Assert.False(lastScope == null);
212
213     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(last
        ↪ Scope.TempTransactionLogFilename);
214
215     lastScope.DeleteFiles();
216 }
217 }
218
219 [Fact]
220 public static void TransactionCommit()
221 {
222     var itself = _constants.Itself;
223
224     var tempDatabaseFilename = Path.GetTempFileName();
225     var tempTransactionLogFilename = Path.GetTempFileName();
226
227     // Commit
228     using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
        ↪ UInt64UnitedMemoryLinks(tempDatabaseFilename), tempTransactionLogFilename))
229     using (var links = new UInt64Links(memoryAdapter))
230     {
231         using (var transaction = memoryAdapter.BeginTransaction())
232         {
233             var l1 = links.CreateAndUpdate(itself, itself);
234             var l2 = links.CreateAndUpdate(itself, itself);
235
236             Global.Trash = links.Update(l2, l2, l1, l2);
237
238             links.Delete(l1);
239
240             transaction.Commit();
241         }
242
243         Global.Trash = links.Count();
244     }
245
246     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran
        ↪ sactionLogFilename);
247 }
248
249 [Fact]
250 public static void TransactionDamage()
251 {
252     var itself = _constants.Itself;

```



```

253
254     var tempDatabaseFilename = Path.GetTempFileName();
255     var tempTransactionLogFilename = Path.GetTempFileName();
256
257     // Commit
258     using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
259         ↪ UInt64UnitedMemoryLinks(tempDatabaseFilename), tempTransactionLogFilename))
260     using (var links = new UInt64Links(memoryAdapter))
261     {
262         using (var transaction = memoryAdapter.BeginTransaction())
263         {
264             var l1 = links.CreateAndUpdate(itself, itself);
265             var l2 = links.CreateAndUpdate(itself, itself);
266
267             Global.Trash = links.Update(l2, l2, l1, l2);
268
269             links.Delete(l1);
270
271             transaction.Commit();
272         }
273
274         Global.Trash = links.Count();
275     }
276
277     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran_
278         ↪ sactionLogFilename);
279
280     // Damage database
281
282     FileHelpers.WriteFirst(tempTransactionLogFilename, new
283         ↪ UInt64LinksTransactionsLayer.Transition(new UniqueTimestampFactory(), 555));
284
285     // Try load damaged database
286     try
287     {
288         // TODO: Fix
289         using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
290             ↪ UInt64UnitedMemoryLinks(tempDatabaseFilename), tempTransactionLogFilename))
291         using (var links = new UInt64Links(memoryAdapter))
292         {
293             Global.Trash = links.Count();
294         }
295     }
296     catch (NotSupportedException ex)
297     {
298         Assert.True(ex.Message == "Database is damaged, autorecovery is not supported
299             ↪ yet.");
300     }
301
302     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran_
303         ↪ sactionLogFilename);
304
305     File.Delete(tempDatabaseFilename);
306     File.Delete(tempTransactionLogFilename);
307 }
308
309 [Fact]
310 public static void Bug1Test()
311 {
312     var tempDatabaseFilename = Path.GetTempFileName();
313     var tempTransactionLogFilename = Path.GetTempFileName();
314
315     var itself = _constants.Itself;
316
317     // User Code Error (Autoreverted), some data saved
318     try
319     {
320         ulong l1;
321         ulong l2;
322
323         using (var memory = new UInt64UnitedMemoryLinks(tempDatabaseFilename))
324         using (var memoryAdapter = new UInt64LinksTransactionsLayer(memory,
325             ↪ tempTransactionLogFilename))
326         using (var links = new UInt64Links(memoryAdapter))
327         {
328             l1 = links.CreateAndUpdate(itself, itself);
329             l2 = links.CreateAndUpdate(itself, itself);
330
331             l2 = links.Update(l2, l2, l1, l2);
332         }
333     }
334 }

```

```

325         links.CreateAndUpdate(l2, itself);
326         links.CreateAndUpdate(l2, itself);
327     }
328
329     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(temp ↵
330         ↵ TransactionLogFilename);
331
332     using (var memory = new UInt64UnitedMemoryLinks(tempDatabaseFilename))
333     using (var memoryAdapter = new UInt64LinksTransactionsLayer(memory,
334         ↵ tempTransactionLogFilename))
335     using (var links = new UInt64Links(memoryAdapter))
336     {
337         using (var transaction = memoryAdapter.BeginTransaction())
338         {
339             l2 = links.Update(l2, l1);
340
341             links.Delete(l2);
342
343             ExceptionThrower();
344
345             transaction.Commit();
346         }
347
348         Global.Trash = links.Count();
349     }
350     catch
351     {
352         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(temp ↵
353         ↵ TransactionLogFilename);
354     }
355
356     File.Delete(tempDatabaseFilename);
357     File.Delete(tempTransactionLogFilename);
358 }
359
360 private static void ExceptionThrower() => throw new InvalidOperationException();
361
362 [Fact]
363 public static void PathsTest()
364 {
365     var source = _constants.SourcePart;
366     var target = _constants.TargetPart;
367
368     using (var scope = new TempLinksTestScope())
369     {
370         var links = scope.Links;
371         var l1 = links.CreatePoint();
372         var l2 = links.CreatePoint();
373
374         var r1 = links.GetByKeys(l1, source, target, source);
375         var r2 = links.CheckPathExistence(l2, l2, l2, l2);
376     }
377 }
378
379 [Fact]
380 public static void RecursiveStringFormattingTest()
381 {
382     using (var scope = new TempLinksTestScope(useSequences: true))
383     {
384         var links = scope.Links;
385         var sequences = scope.Sequences; // TODO: Auto use sequences on Sequences getter.
386
387         var a = links.CreatePoint();
388         var b = links.CreatePoint();
389         var c = links.CreatePoint();
390
391         var ab = links.GetOrCreate(a, b);
392         var cb = links.GetOrCreate(c, b);
393         var ac = links.GetOrCreate(a, c);
394
395         a = links.Update(a, c, b);
396         b = links.Update(b, a, c);
397         c = links.Update(c, a, b);
398
399         Debug.WriteLine(links.FormatStructure(ab, link => link.IsFullPoint(), true));
400         Debug.WriteLine(links.FormatStructure(cb, link => link.IsFullPoint(), true));
401         Debug.WriteLine(links.FormatStructure(ac, link => link.IsFullPoint(), true));

```

```

401     Assert.True(links.FormatStructure(cb, link => link.IsFullPoint(), true) ==
402         ↪ "(5:(4:5 (6:5 4)) 6)");
403     Assert.True(links.FormatStructure(ac, link => link.IsFullPoint(), true) ==
404         ↪ "(6:(5:(4:5 6) 6) 4)");
405     Assert.True(links.FormatStructure(ab, link => link.IsFullPoint(), true) ==
406         ↪ "(4:(5:4 (6:5 4)) 6)");
407
408     // TODO: Think how to build balanced syntax tree while formatting structure (eg.
409     ↪ "(4:(5:4 6) (6:5 4))" instead of "(4:(5:4 (6:5 4)) 6)")
410
411     Assert.True(sequences.SafeFormatSequence(cb, DefaultFormatter, false) ==
412         ↪ "{5}{5}{4}{6}");
413     Assert.True(sequences.SafeFormatSequence(ac, DefaultFormatter, false) ==
414         ↪ "{5}{6}{6}{4}");
415     Assert.True(sequences.SafeFormatSequence(ab, DefaultFormatter, false) ==
416         ↪ "{4}{5}{4}{6}");
417 }
418 }
419
420 private static void DefaultFormatter(StringBuilder sb, ulong link)
421 {
422     sb.Append(link.ToString());
423 }
424
425 #endregion
426
427 #region Performance
428
429 /*
430 public static void RunAllPerformanceTests()
431 {
432     try
433     {
434         links.TestLinksInSteps();
435     }
436     catch (Exception ex)
437     {
438         ex.WriteToConsole();
439     }
440
441     return;
442
443     try
444     {
445         //ThreadPool.SetMaxThreads(2, 2);
446
447         // Запускаем все тесты дважды, чтобы первоначальная инициализация не повлияла на
448         ↪ результат
449         // Также это дополнительно помогает в отладке
450         // Увеличивает вероятность попадания информации в кэши
451         for (var i = 0; i < 10; i++)
452         {
453             //0 - 10 ГБ
454             //Каждые 100 МБ срез цифр
455
456             //links.TestGetSourceFunction();
457             //links.TestGetSourceFunctionInParallel();
458             //links.TestGetTargetFunction();
459             //links.TestGetTargetFunctionInParallel();
460             links.Create64BillionLinks();
461
462             links.TestRandomSearchFixed();
463             //links.Create64BillionLinksInParallel();
464             links.TestEachFunction();
465             //links.TestForeach();
466             //links.TestParallelForeach();
467         }
468
469         links.TestDeletionOfAllLinks();
470
471     }
472     catch (Exception ex)
473     {
474         ex.WriteToConsole();
475     }
476 }
477 */
478
479 /*

```

```

472     public static void TestLinksInSteps()
473     {
474         const long gibibyte = 1024 * 1024 * 1024;
475         const long mebibyte = 1024 * 1024;
476
477         var totalLinksToCreate = gibibyte /
↪ Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
478         var linksStep = 102 * mebibyte /
↪ Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
479
480         var creationMeasurements = new List<TimeSpan>();
481         var searchMeasurements = new List<TimeSpan>();
482         var deletionMeasurements = new List<TimeSpan>();
483
484         GetBaseRandomLoopOverhead(linksStep);
485         GetBaseRandomLoopOverhead(linksStep);
486
487         var stepLoopOverhead = GetBaseRandomLoopOverhead(linksStep);
488
489         ConsoleHelpers.Debug("Step loop overhead: {0}.", stepLoopOverhead);
490
491         var loops = totalLinksToCreate / linksStep;
492
493         for (int i = 0; i < loops; i++)
494         {
495             creationMeasurements.Add(Measure(() => links.RunRandomCreations(linksStep)));
496             searchMeasurements.Add(Measure(() => links.RunRandomSearches(linksStep)));
497
498             Console.WriteLine("\rC + S {0}/{1}", i + 1, loops);
499         }
500
501         ConsoleHelpers.Debug();
502
503         for (int i = 0; i < loops; i++)
504         {
505             deletionMeasurements.Add(Measure(() => links.RunRandomDeletions(linksStep)));
506
507             Console.WriteLine("\rD {0}/{1}", i + 1, loops);
508         }
509
510         ConsoleHelpers.Debug();
511
512         ConsoleHelpers.Debug("C S D");
513
514         for (int i = 0; i < loops; i++)
515         {
516             ConsoleHelpers.Debug("{0} {1} {2}", creationMeasurements[i],
↪ searchMeasurements[i], deletionMeasurements[i]);
517         }
518
519         ConsoleHelpers.Debug("C S D (no overhead)");
520
521         for (int i = 0; i < loops; i++)
522         {
523             ConsoleHelpers.Debug("{0} {1} {2}", creationMeasurements[i] - stepLoopOverhead,
↪ searchMeasurements[i] - stepLoopOverhead, deletionMeasurements[i] - stepLoopOverhead);
524         }
525
526         ConsoleHelpers.Debug("All tests done. Total links left in database: {0}.",
↪ links.Total);
527     }
528
529     private static void CreatePoints(this Platform.Links.Data.Core.Doublets.Links links, long
↪ amountToCreate)
530     {
531         for (long i = 0; i < amountToCreate; i++)
532             links.Create(0, 0);
533     }
534
535     private static TimeSpan GetBaseRandomLoopOverhead(long loops)
536     {
537         return Measure(() =>
538         {
539             ulong maxValue = RandomHelpers.DefaultFactory.NextUInt64();
540             ulong result = 0;
541             for (long i = 0; i < loops; i++)
542             {
543                 var source = RandomHelpers.DefaultFactory.NextUInt64(maxValue);
544                 var target = RandomHelpers.DefaultFactory.NextUInt64(maxValue);

```

```

545         result += maxValue + source + target;
546     }
547     Global.Trash = result;
548 });
549 }
550 */
551
552 [Fact(Skip = "performance test")]
553 public static void GetSourceTest()
554 {
555     using (var scope = new TempLinksTestScope())
556     {
557         var links = scope.Links;
558         ConsoleHelpers.Debug("Testing GetSource function with {0} Iterations.",
559             ↪ Iterations);
560
561         ulong counter = 0;
562
563         //var firstLink = links.First();
564         // Создаём одну связь, из которой будет производить считывание
565         var firstLink = links.Create();
566
567         var sw = Stopwatch.StartNew();
568
569         // Тестируем саму функцию
570         for (ulong i = 0; i < Iterations; i++)
571         {
572             counter += links.GetSource(firstLink);
573         }
574
575         var elapsedTime = sw.Elapsed;
576
577         var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
578
579         // Удаляем связь, из которой производилось считывание
580         links.Delete(firstLink);
581
582         ConsoleHelpers.Debug(
583             "{0} Iterations of GetSource function done in {1} ({2} Iterations per
584             ↪ second), counter result: {3}",
585             Iterations, elapsedTime, (long)iterationsPerSecond, counter);
586     }
587
588 [Fact(Skip = "performance test")]
589 public static void GetSourceInParallel()
590 {
591     using (var scope = new TempLinksTestScope())
592     {
593         var links = scope.Links;
594         ConsoleHelpers.Debug("Testing GetSource function with {0} Iterations in
595             ↪ parallel.", Iterations);
596
597         long counter = 0;
598
599         //var firstLink = links.First();
600         var firstLink = links.Create();
601
602         var sw = Stopwatch.StartNew();
603
604         // Тестируем саму функцию
605         Parallel.For(0, Iterations, x =>
606         {
607             Interlocked.Add(ref counter, (long)links.GetSource(firstLink));
608             //Interlocked.Increment(ref counter);
609         });
610
611         var elapsedTime = sw.Elapsed;
612
613         var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
614
615         links.Delete(firstLink);
616
617         ConsoleHelpers.Debug(
618             "{0} Iterations of GetSource function done in {1} ({2} Iterations per
619             ↪ second), counter result: {3}",
620             Iterations, elapsedTime, (long)iterationsPerSecond, counter);
621     }

```

```

620     }
621
622     [Fact(Skip = "performance test")]
623     public static void TestGetTarget()
624     {
625         using (var scope = new TempLinksTestScope())
626         {
627             var links = scope.Links;
628             ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations.",
629                 ↪ Iterations);
630
631             ulong counter = 0;
632
633             //var firstLink = links.First();
634             var firstLink = links.Create();
635
636             var sw = Stopwatch.StartNew();
637
638             for (ulong i = 0; i < Iterations; i++)
639             {
640                 counter += links.GetTarget(firstLink);
641             }
642
643             var elapsedTime = sw.Elapsed;
644
645             var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
646
647             links.Delete(firstLink);
648
649             ConsoleHelpers.Debug(
650                 "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
651                 ↪ second), counter result: {3}",
652                 Iterations, elapsedTime, (long)iterationsPerSecond, counter);
653         }
654     }
655
656     [Fact(Skip = "performance test")]
657     public static void TestGetTargetInParallel()
658     {
659         using (var scope = new TempLinksTestScope())
660         {
661             var links = scope.Links;
662             ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations in
663                 ↪ parallel.", Iterations);
664
665             long counter = 0;
666
667             //var firstLink = links.First();
668             var firstLink = links.Create();
669
670             var sw = Stopwatch.StartNew();
671
672             Parallel.For(0, Iterations, x =>
673             {
674                 Interlocked.Add(ref counter, (long)links.GetTarget(firstLink));
675                 //Interlocked.Increment(ref counter);
676             });
677
678             var elapsedTime = sw.Elapsed;
679
680             var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
681
682             links.Delete(firstLink);
683
684             ConsoleHelpers.Debug(
685                 "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
686                 ↪ second), counter result: {3}",
687                 Iterations, elapsedTime, (long)iterationsPerSecond, counter);
688         }
689     }
690
691     // TODO: Заполнить базу данных перед тестом
692     /*
693     [Fact]
694     public void TestRandomSearchFixed()
695     {
696         var tempFilename = Path.GetTempFileName();
697
698         using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
699             ↪ DefaultLinksSizeStep))

```

```

695         {
696             long iterations = 64 * 1024 * 1024 /
↪ Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
697
698             ulong counter = 0;
699             var maxLink = links.Total;
700
701             ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.", iterations);
702
703             var sw = Stopwatch.StartNew();
704
705             for (var i = iterations; i > 0; i--)
706             {
707                 var source =
↪ RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
708                 var target =
↪ RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
709
710                 counter += links.Search(source, target);
711             }
712
713             var elapsedTime = sw.Elapsed;
714
715             var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;
716
717             ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
↪ Iterations per second), c: {3}", iterations, elapsedTime, (long)iterationsPerSecond,
↪ counter);
718         }
719
720         File.Delete(tempFilename);
721     }*/
722
723     [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
724     public static void TestRandomSearchAll()
725     {
726         using (var scope = new TempLinksTestScope())
727         {
728             var links = scope.Links;
729             ulong counter = 0;
730
731             var maxLink = links.Count();
732
733             var iterations = links.Count();
734
735             ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.",
↪ links.Count());
736
737             var sw = Stopwatch.StartNew();
738
739             for (var i = iterations; i > 0; i--)
740             {
741                 var linksAddressRange = new
↪ Range<ulong>(_constants.InternalReferencesRange.Minimum, maxLink);
742
743                 var source = RandomHelpers.Default.NextUInt64(linksAddressRange);
744                 var target = RandomHelpers.Default.NextUInt64(linksAddressRange);
745
746                 counter += links.SearchOrDefault(source, target);
747             }
748
749             var elapsedTime = sw.Elapsed;
750
751             var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;
752
753             ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
↪ Iterations per second), c: {3}",
↪ iterations, elapsedTime, (long)iterationsPerSecond, counter);
754         }
755     }
756
757     [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
758     public static void TestEach()
759     {
760         using (var scope = new TempLinksTestScope())
761         {
762             var links = scope.Links;
763
764             var counter = new Counter<IList<ulong>, ulong>(links.Constants.Continue);
765
766

```

```

767         ConsoleHelpers.Debug("Testing Each function.");
768
769         var sw = Stopwatch.StartNew();
770
771         links.Each(counter.IncrementAndReturnTrue);
772
773         var elapsedTime = sw.Elapsed;
774
775         var linksPerSecond = counter.Count / elapsedTime.TotalSeconds;
776
777         ConsoleHelpers.Debug("{0} Iterations of Each's handler function done in {1} ({2}
↪         links per second)",
            counter, elapsedTime, (long)linksPerSecond);
778     }
779 }
780
781 /*
782 [Fact]
783 public static void TestForeach()
784 {
785     var tempFilename = Path.GetTempFileName();
786
787     using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
↪     DefaultLinksSizeStep))
788     {
789         ulong counter = 0;
790
791         ConsoleHelpers.Debug("Testing foreach through links.");
792
793         var sw = Stopwatch.StartNew();
794
795         //foreach (var link in links)
796         //{
797             //    counter++;
798         //}
799
800         var elapsedTime = sw.Elapsed;
801
802         var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
803
804         ConsoleHelpers.Debug("{0} Iterations of Foreach's handler block done in {1} ({2}
↪     links per second)", counter, elapsedTime, (long)linksPerSecond);
805     }
806
807     File.Delete(tempFilename);
808 }
809 */
810
811 /*
812 [Fact]
813 public static void TestParallelForeach()
814 {
815     var tempFilename = Path.GetTempFileName();
816
817     using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
↪     DefaultLinksSizeStep))
818     {
819
820         long counter = 0;
821
822         ConsoleHelpers.Debug("Testing parallel foreach through links.");
823
824         var sw = Stopwatch.StartNew();
825
826         //Parallel.ForEach((IEnumerable<ulong>)links, x =>
827         //{
828             //    Interlocked.Increment(ref counter);
829         //});
830
831         var elapsedTime = sw.Elapsed;
832
833         var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
834
835         ConsoleHelpers.Debug("{0} Iterations of Parallel Foreach's handler block done in
↪     {1} ({2} links per second)", counter, elapsedTime, (long)linksPerSecond);
836     }
837
838     File.Delete(tempFilename);
839 }
840 */
841

```



```

842 [Fact(Skip = "performance test")]
843 public static void Create64BillionLinks()
844 {
845     using (var scope = new TempLinksTestScope())
846     {
847         var links = scope.Links;
848         var linksBeforeTest = links.Count();
849
850         long linksToCreate = 64 * 1024 * 1024 / UInt64UnitedMemoryLinks.LinkSizeInBytes;
851
852         ConsoleHelpers.Debug("Creating {0} links.", linksToCreate);
853
854         var elapsedTime = Performance.Measure(() =>
855         {
856             for (long i = 0; i < linksToCreate; i++)
857             {
858                 links.Create();
859             }
860         });
861
862         var linksCreated = links.Count() - linksBeforeTest;
863         var linksPerSecond = linksCreated / elapsedTime.TotalSeconds;
864
865         ConsoleHelpers.Debug("Current links count: {0}.", links.Count());
866
867         ConsoleHelpers.Debug("{0} links created in {1} ({2} links per second)",
868             ↪ linksCreated, elapsedTime,
869             (long)linksPerSecond);
870     }
871 }
872
873 [Fact(Skip = "performance test")]
874 public static void Create64BillionLinksInParallel()
875 {
876     using (var scope = new TempLinksTestScope())
877     {
878         var links = scope.Links;
879         var linksBeforeTest = links.Count();
880
881         var sw = Stopwatch.StartNew();
882
883         long linksToCreate = 64 * 1024 * 1024 / UInt64UnitedMemoryLinks.LinkSizeInBytes;
884
885         ConsoleHelpers.Debug("Creating {0} links in parallel.", linksToCreate);
886
887         Parallel.For(0, linksToCreate, x => links.Create());
888
889         var elapsedTime = sw.Elapsed;
890
891         var linksCreated = links.Count() - linksBeforeTest;
892         var linksPerSecond = linksCreated / elapsedTime.TotalSeconds;
893
894         ConsoleHelpers.Debug("{0} links created in {1} ({2} links per second)",
895             ↪ linksCreated, elapsedTime,
896             (long)linksPerSecond);
897     }
898 }
899
900 [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
901 public static void TestDeletionOfAllLinks()
902 {
903     using (var scope = new TempLinksTestScope())
904     {
905         var links = scope.Links;
906         var linksBeforeTest = links.Count();
907
908         ConsoleHelpers.Debug("Deleting all links");
909
910         var elapsedTime = Performance.Measure(links.DeleteAll);
911
912         var linksDeleted = linksBeforeTest - links.Count();
913         var linksPerSecond = linksDeleted / elapsedTime.TotalSeconds;
914
915         ConsoleHelpers.Debug("{0} links deleted in {1} ({2} links per second)",
916             ↪ linksDeleted, elapsedTime,
917             (long)linksPerSecond);
918     }
919 }

```

```
#endregion
```

```
919 }  
920 }  
921 }
```

1.71 ./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 TLink = System.UInt64;  
9  
10 namespace Platform.Data.Doublets.Sequences.Tests  
11 {  
12     public class UInt64LinksExtensionsTests  
13     {  
14         public static ILinks<TLink> CreateLinks() => CreateLinks<TLink>(new  
15             ↳ Platform.IO.TemporaryFile());  
16  
17         public static ILinks<TLink> CreateLinks<TLink>(string dataDBFilename)  
18         {  
19             var linksConstants = new LinksConstants<TLink>(enableExternalReferencesSupport:  
20                 ↳ true);  
21             return new UnitedMemoryLinks<TLink>(new  
22                 ↳ FileMappedResizableDirectMemory(dataDBFilename),  
23                 ↳ UnitedMemoryLinks<TLink>.DefaultLinksSizeStep, linksConstants,  
24                 ↳ IndexTreeType.Default);  
25         }  
26  
27         [Fact]  
28         public void FormatStructureWithExternalReferenceTest()  
29         {  
30             ILinks<TLink> links = CreateLinks();  
31             TLink zero = default;  
32             var one = Arithmetic.Increment(zero);  
33             var markerIndex = one;  
34             var meaningRoot = links.GetOrCreate(markerIndex, markerIndex);  
35             var numberMarker = links.GetOrCreate(meaningRoot, Arithmetic.Increment(ref  
36                 ↳ markerIndex));  
37             AddressToRawNumberConverter<TLink> addressToNumberConverter = new();  
38             var numberAddress = addressToNumberConverter.Convert(1);  
39             var numberLink = links.GetOrCreate(numberMarker, numberAddress);  
40             var linkNotation = links.FormatStructure(numberLink, link => link.IsFullPoint(),  
41                 ↳ true);  
42             Assert.Equal("(3:(2:1 2) 18446744073709551615)", linkNotation);  
43         }  
44     }  
45 }
```

1.72 ./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             }  
31         }  
32     }  
33 }
```

```

27     }
28     var fromUnaryNumberConverterUsingOrOperation = new
        ↳ UnaryNumberToAddressOrOperationConverter<ulong>(links,
        ↳ powerOf2ToUnaryNumberConverter);
29     var fromUnaryNumberConverterUsingAddOperation = new
        ↳ UnaryNumberToAddressAddOperationConverter<ulong>(links, one);
30     for (int i = 0; i < N; i++)
31     {
32         Assert.Equal(numbers[i],
        ↳ fromUnaryNumberConverterUsingOrOperation.Convert(unaryNumbers[i]));
33         Assert.Equal(numbers[i],
        ↳ fromUnaryNumberConverterUsingAddOperation.Convert(unaryNumbers[i]));
34     }
35 }
36 }
37 }
38 }

```

1.73 ./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
49         private static void TestCharAndUnicodeSymbolConverters(ILinks<ulong> links, ulong
        ↳ meaningRoot, IConverter<ulong> addressToNumberConverter, IConverter<ulong>
        ↳ numberToAddressConverter)
50         {
51             var unicodeSymbolMarker = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);

```

```

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

```

```
103     var unicodeSequenceToStringConverter = new
        ↳ UnicodeSequenceToStringConverter<ulong>(links,
        ↳ unicodeSequenceCriterionMatcher, sequenceWalker,
        ↳ unicodeSymbolToCharConverter);
104
105     var resultingString =
        ↳ unicodeSequenceToStringConverter.Convert(unicodeSequenceLink);
106
107     Assert.Equal(originalString, resultingString);
108 }
109 }
110 }
111 }
```

Index

`./csharp/Platform.Data.Doublets.Sequences.Tests/BigIntegerConvertersTests.cs`, 82
`./csharp/Platform.Data.Doublets.Sequences.Tests/DefaultSequenceAppenderTests.cs`, 84
`./csharp/Platform.Data.Doublets.Sequences.Tests/ILinksExtensionsTests.cs`, 85
`./csharp/Platform.Data.Doublets.Sequences.Tests/OptimalVariantSequenceTests.cs`, 85
`./csharp/Platform.Data.Doublets.Sequences.Tests/RationalNumbersTests.cs`, 88
`./csharp/Platform.Data.Doublets.Sequences.Tests/ReadSequenceTests.cs`, 90
`./csharp/Platform.Data.Doublets.Sequences.Tests/SequencesTests.cs`, 91
`./csharp/Platform.Data.Doublets.Sequences.Tests/TempLinksTestScope.cs`, 106
`./csharp/Platform.Data.Doublets.Sequences.Tests/TestExtensions.cs`, 107
`./csharp/Platform.Data.Doublets.Sequences.Tests/UInt64LinksTests.cs`, 109
`./csharp/Platform.Data.Doublets.Sequences.Tests/UInt64LinksExtensionsTests.cs`, 122
`./csharp/Platform.Data.Doublets.Sequences.Tests/UnaryNumberConvertersTests.cs`, 122
`./csharp/Platform.Data.Doublets.Sequences.Tests/UnicodeConvertersTests.cs`, 123
`./csharp/Platform.Data.Doublets.Sequences/Converters/BalancedVariantConverter.cs`, 1
`./csharp/Platform.Data.Doublets.Sequences/Converters/CompressingConverter.cs`, 1
`./csharp/Platform.Data.Doublets.Sequences/Converters/LinksListToSequenceConverterBase.cs`, 4
`./csharp/Platform.Data.Doublets.Sequences/Converters/OptimalVariantConverter.cs`, 5
`./csharp/Platform.Data.Doublets.Sequences/Converters/SequenceToItsLocalElementLevelsConverter.cs`, 6
`./csharp/Platform.Data.Doublets.Sequences/CriterionMatchers/DefaultSequenceElementCriterionMatcher.cs`, 7
`./csharp/Platform.Data.Doublets.Sequences/CriterionMatchers/MarkedSequenceCriterionMatcher.cs`, 7
`./csharp/Platform.Data.Doublets.Sequences/DefaultSequenceAppender.cs`, 8
`./csharp/Platform.Data.Doublets.Sequences/DuplicateSegmentsCounter.cs`, 8
`./csharp/Platform.Data.Doublets.Sequences/DuplicateSegmentsProvider.cs`, 9
`./csharp/Platform.Data.Doublets.Sequences/Frequencies/Cache/LinkFrequenciesCache.cs`, 11
`./csharp/Platform.Data.Doublets.Sequences/Frequencies/Cache/LinkFrequency.cs`, 13
`./csharp/Platform.Data.Doublets.Sequences/Frequencies/Cache/LinkToItsFrequencyValueConverter.cs`, 13
`./csharp/Platform.Data.Doublets.Sequences/Frequencies/Counters/MarkedSequenceSymbolFrequencyOneOffCounter.cs`, 14
`./csharp/Platform.Data.Doublets.Sequences/Frequencies/Counters/SequenceSymbolFrequencyOneOffCounter.cs`, 14
`./csharp/Platform.Data.Doublets.Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyCounter.cs`, 15
`./csharp/Platform.Data.Doublets.Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyOneOffCounter.cs`, 15
`./csharp/Platform.Data.Doublets.Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyCounter.cs`, 16
`./csharp/Platform.Data.Doublets.Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyOneOffCounter.cs`, 16
`./csharp/Platform.Data.Doublets.Sequences/HeightProviders/CachedSequenceHeightProvider.cs`, 17
`./csharp/Platform.Data.Doublets.Sequences/HeightProviders/DefaultSequenceRightHeightProvider.cs`, 18
`./csharp/Platform.Data.Doublets.Sequences/HeightProviders/ISequenceHeightProvider.cs`, 18
`./csharp/Platform.Data.Doublets.Sequences/Incrementers/FrequencyIncrementer.cs`, 18
`./csharp/Platform.Data.Doublets.Sequences/Incrementers/UnaryNumberIncrementer.cs`, 19
`./csharp/Platform.Data.Doublets.Sequences/Indexes/CachedFrequencyIncrementingSequenceIndex.cs`, 19
`./csharp/Platform.Data.Doublets.Sequences/Indexes/FrequencyIncrementingSequenceIndex.cs`, 20
`./csharp/Platform.Data.Doublets.Sequences/Indexes/ISequenceIndex.cs`, 21
`./csharp/Platform.Data.Doublets.Sequences/Indexes/SequenceIndex.cs`, 21
`./csharp/Platform.Data.Doublets.Sequences/Indexes/SynchronizedSequenceIndex.cs`, 22
`./csharp/Platform.Data.Doublets.Sequences/Indexes/Unindex.cs`, 22
`./csharp/Platform.Data.Doublets.Sequences/Numbers/Rational/DecimalToRationalConverter.cs`, 23
`./csharp/Platform.Data.Doublets.Sequences/Numbers/Rational/RationalToDecimalConverter.cs`, 23
`./csharp/Platform.Data.Doublets.Sequences/Numbers/Raw/BigIntegerToRawNumberSequenceConverter.cs`, 24
`./csharp/Platform.Data.Doublets.Sequences/Numbers/Raw/LongRawNumberSequenceToNumberConverter.cs`, 25
`./csharp/Platform.Data.Doublets.Sequences/Numbers/Raw/NumberToLongRawNumberSequenceConverter.cs`, 25
`./csharp/Platform.Data.Doublets.Sequences/Numbers/Raw/RawNumberSequenceToBigIntegerConverter.cs`, 26
`./csharp/Platform.Data.Doublets.Sequences/Numbers/Raw/AddressToUnaryNumberConverter.cs`, 27
`./csharp/Platform.Data.Doublets.Sequences/Numbers/Unary/LinkToItsFrequencyNumberConverter.cs`, 27
`./csharp/Platform.Data.Doublets.Sequences/Numbers/Unary/PowerOf2ToUnaryNumberConverter.cs`, 28
`./csharp/Platform.Data.Doublets.Sequences/Numbers/Unary/UnaryNumberToAddressAddOperationConverter.cs`, 28
`./csharp/Platform.Data.Doublets.Sequences/Numbers/Unary/UnaryNumberToAddressOrOperationConverter.cs`, 30
`./csharp/Platform.Data.Doublets.Sequences/Sequences.Experiments.cs`, 30
`./csharp/Platform.Data.Doublets.Sequences/Sequences.cs`, 57
`./csharp/Platform.Data.Doublets.Sequences/SequencesExtensions.cs`, 68
`./csharp/Platform.Data.Doublets.Sequences/SequencesOptions.cs`, 69
`./csharp/Platform.Data.Doublets.Sequences/Time/DateTimeToLongRawNumberSequenceConverter.cs`, 72
`./csharp/Platform.Data.Doublets.Sequences/Time/LongRawNumberSequenceToDateTimeConverter.cs`, 72
`./csharp/Platform.Data.Doublets.Sequences/UInt64LinksExtensions.cs`, 72
`./csharp/Platform.Data.Doublets.Sequences/Unicode/CharToUnicodeSymbolConverter.cs`, 72
`./csharp/Platform.Data.Doublets.Sequences/Unicode/StringToUnicodeSequenceConverter.cs`, 73
`./csharp/Platform.Data.Doublets.Sequences/Unicode/StringToUnicodeSymbolsListConverter.cs`, 74
`./csharp/Platform.Data.Doublets.Sequences/Unicode/UnicodeMap.cs`, 74

- ./csharp/Platform.Data.Doublets.Sequences/Unicode/UnicodeSequenceToStringConverter.cs, 77
- ./csharp/Platform.Data.Doublets.Sequences/Unicode/UnicodeSymbolToCharConverter.cs, 77
- ./csharp/Platform.Data.Doublets.Sequences/Unicode/UnicodeSymbolsListToUnicodeSequenceConverter.cs, 78
- ./csharp/Platform.Data.Doublets.Sequences/Walkers/ISequenceWalker.cs, 78
- ./csharp/Platform.Data.Doublets.Sequences/Walkers/LeftSequenceWalker.cs, 79
- ./csharp/Platform.Data.Doublets.Sequences/Walkers/LeveledSequenceWalker.cs, 79
- ./csharp/Platform.Data.Doublets.Sequences/Walkers/RightSequenceWalker.cs, 81
- ./csharp/Platform.Data.Doublets.Sequences/Walkers/SequenceWalkerBase.cs, 81