

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

./Platform.Data.Doublets/Decorators/LinksCascadeUniquenessAndUsagesResolver.cs

```
1 namespace Platform.Data.Doublets.Decorators
2 {
3     public class LinksCascadeUniquenessAndUsagesResolver<TLink> : LinksUniquenessResolver<TLink>
4     {
5         public LinksCascadeUniquenessAndUsagesResolver(ILinks<TLink> links) : base(links) { }
6
7         protected override TLink ResolveAddressChangeConflict(TLink oldLinkAddress, TLink
8             ↪ newLinkAddress)
9         {
10             Links.MergeUsages(oldLinkAddress, newLinkAddress);
11             return base.ResolveAddressChangeConflict(oldLinkAddress, newLinkAddress);
12         }
13     }
```

./Platform.Data.Doublets/Decorators/LinksCascadeUsagesResolver.cs

```
1 namespace Platform.Data.Doublets.Decorators
2 {
3     /// <remarks>
4     /// <para>Must be used in conjunction with NonNullContentsLinkDeletionResolver.</para>
5     /// <para>Должен использоваться вместе с NonNullContentsLinkDeletionResolver.</para>
6     /// </remarks>
7     public class LinksCascadeUsagesResolver<TLink> : LinksDecoratorBase<TLink>
8     {
9         public LinksCascadeUsagesResolver(ILinks<TLink> links) : base(links) { }
10
11         public override void Delete(TLink linkIndex)
12         {
13             this.DeleteAllUsages(linkIndex);
14             Links.Delete(linkIndex);
15         }
16     }
17 }
```

./Platform.Data.Doublets/Decorators/LinksDecoratorBase.cs

```
1 using System;
2 using System.Collections.Generic;
3 using Platform.Data.Constants;
4
5 namespace Platform.Data.Doublets.Decorators
6 {
7     public abstract class LinksDecoratorBase<TLink> : LinksOperatorBase<TLink>, ILinks<TLink>
8     {
9         public LinksCombinedConstants<TLink, TLink, int> Constants { get; }
10         protected LinksDecoratorBase(ILinks<TLink> links) : base(links) => Constants =
11             ↪ links.Constants;
12         public virtual TLink Count(IList<TLink> restriction) => Links.Count(restriction);
13         public virtual TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
14             ↪ => Links.Each(handler, restrictions);
15         public virtual TLink Create() => Links.Create();
16         public virtual TLink Update(IList<TLink> restrictions) => Links.Update(restrictions);
17         public virtual void Delete(TLink link) => Links.Delete(link);
18     }
19 }
```

./Platform.Data.Doublets/Decorators/LinksDisposableDecoratorBase.cs

```
1 using System;
2 using System.Collections.Generic;
3 using Platform.Disposables;
4 using Platform.Data.Constants;
5
6 namespace Platform.Data.Doublets.Decorators
7 {
8     public abstract class LinksDisposableDecoratorBase<TLink> : DisposableBase, ILinks<TLink>
9     {
10         public LinksCombinedConstants<TLink, TLink, int> Constants { get; }
11
12         public ILinks<TLink> Links { get; }
13
14         protected LinksDisposableDecoratorBase(ILinks<TLink> links)
15         {
16             Links = links;
17             Constants = links.Constants;
18         }
19
20         public virtual TLink Count(IList<TLink> restriction) => Links.Count(restriction);
21     }
```

```

22     public virtual TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
23         => Links.Each(handler, restrictions);
24
25     public virtual TLink Create() => Links.Create();
26
27     public virtual TLink Update(IList<TLink> restrictions) => Links.Update(restrictions);
28
29     public virtual void Delete(TLink link) => Links.Delete(link);
30
31     protected override bool AllowMultipleDisposeCalls => true;
32
33     protected override void Dispose(bool manual, bool wasDisposed)
34     {
35         if (!wasDisposed)
36         {
37             Links.DisposeIfPossible();
38         }
39     }
40 }

```

./Platform.Data.Doublets/Decorators/LinksInnerReferenceExistenceValidator.cs

```

1  using System;
2  using System.Collections.Generic;
3
4  namespace Platform.Data.Doublets.Decorators
5  {
6      // TODO: Make LinksExternalReferenceValidator. A layer that checks each link to exist or to
7      // be external (hybrid link's raw number).
8      public class LinksInnerReferenceExistenceValidator<TLink> : LinksDecoratorBase<TLink>
9      {
10         public LinksInnerReferenceExistenceValidator(ILinks<TLink> links) : base(links) { }
11
12         public override TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
13         {
14             Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
15             return Links.Each(handler, restrictions);
16         }
17
18         public override TLink Count(IList<TLink> restriction)
19         {
20             Links.EnsureInnerReferenceExists(restriction, nameof(restriction));
21             return Links.Count(restriction);
22         }
23
24         public override TLink Update(IList<TLink> restrictions)
25         {
26             // TODO: Possible values: null, ExistentLink or NonExistentHybrid(ExternalReference)
27             Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
28             return Links.Update(restrictions);
29         }
30
31         public override void Delete(TLink link)
32         {
33             Links.EnsureLinkExists(link, nameof(link));
34             Links.Delete(link);
35         }
36     }

```

./Platform.Data.Doublets/Decorators/LinksItselfConstantToSelfReferenceResolver.cs

```

1  using System;
2  using System.Collections.Generic;
3
4  namespace Platform.Data.Doublets.Decorators
5  {
6      public class LinksItselfConstantToSelfReferenceResolver<TLink> : LinksDecoratorBase<TLink>
7      {
8         private static readonly EqualityComparer<TLink> _equalityComparer =
9             => EqualityComparer<TLink>.Default;
10
11         public LinksItselfConstantToSelfReferenceResolver(ILinks<TLink> links) : base(links) { }
12
13         public override TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
14         {
15             var constants = Constants;
16             var itselfConstant = constants.Itself;
17             var indexPartConstant = constants.IndexPart;
18             var sourcePartConstant = constants.SourcePart;

```

```

18     var targetPartConstant = constants.TargetPart;
19     var restrictionsCount = restrictions.Count;
20     if (!_equalityComparer.Equals(constants.Any, itselfConstant)
21         && (((restrictionsCount > indexPartConstant) &&
22             ↪ _equalityComparer.Equals(restrictions[indexPartConstant], itselfConstant))
23         || ((restrictionsCount > sourcePartConstant) &&
24             ↪ _equalityComparer.Equals(restrictions[sourcePartConstant], itselfConstant))
25         || ((restrictionsCount > targetPartConstant) &&
26             ↪ _equalityComparer.Equals(restrictions[targetPartConstant], itselfConstant))))
27     {
28         // Itself constant is not supported for Each method right now, skipping execution
29         return constants.Continue;
30     }
31     return Links.Each(handler, restrictions);
32 }
33 }

```

./Platform.Data.Doublets/Decorators/LinksNonExistentDependenciesCreator.cs

```

1  using System.Collections.Generic;
2
3  namespace Platform.Data.Doublets.Decorators
4  {
5      /// <remarks>
6      /// Not practical if newSource and newTarget are too big.
7      /// To be able to use practical version we should allow to create link at any specific
8      ↪ location inside ResizableDirectMemoryLinks.
9      /// This in turn will require to implement not a list of empty links, but a list of ranges
10     ↪ to store it more efficiently.
11     /// </remarks>
12     public class LinksNonExistentDependenciesCreator<TLink> : LinksDecoratorBase<TLink>
13     {
14         public LinksNonExistentDependenciesCreator(ILinks<TLink> links) : base(links) { }
15
16         public override TLink Update(IList<TLink> restrictions)
17         {
18             var constants = Constants;
19             Links.EnsureCreated(restrictions[constants.SourcePart],
20                 ↪ restrictions[constants.TargetPart]);
21             return Links.Update(restrictions);
22         }
23     }
24 }

```

./Platform.Data.Doublets/Decorators/LinksNullConstantToSelfReferenceResolver.cs

```

1  using System.Collections.Generic;
2
3  namespace Platform.Data.Doublets.Decorators
4  {
5      public class LinksNullConstantToSelfReferenceResolver<TLink> : LinksDecoratorBase<TLink>
6      {
7          public LinksNullConstantToSelfReferenceResolver(ILinks<TLink> links) : base(links) { }
8
9          public override TLink Create()
10         {
11             var link = Links.Create();
12             return Links.Update(link, link, link);
13         }
14
15         public override TLink Update(IList<TLink> restrictions) =>
16             ↪ Links.Update(Links.ResolveConstantAsSelfReference(Constants.Null, restrictions));
17     }
18 }

```

./Platform.Data.Doublets/Decorators/LinksUniquenessResolver.cs

```

1  using System.Collections.Generic;
2
3  namespace Platform.Data.Doublets.Decorators
4  {
5      public class LinksUniquenessResolver<TLink> : LinksDecoratorBase<TLink>
6      {
7          private static readonly EqualityComparer<TLink> _equalityComparer =
8             ↪ EqualityComparer<TLink>.Default;
9
10         public LinksUniquenessResolver(ILinks<TLink> links) : base(links) { }

```

```

10
11     public override TLink Update(ICollection<TLink> restrictions)
12     {
13         var newLinkAddress = Links.SearchOrDefault(restrictions[Constants.SourcePart],
14             ↪ restrictions[Constants.TargetPart]);
15         if (!_equalityComparer.Equals(newLinkAddress, default))
16         {
17             return Links.Update(restrictions);
18         }
19         return ResolveAddressChangeConflict(restrictions[Constants.IndexPart],
20             ↪ newLinkAddress);
21     }
22
23     protected virtual TLink ResolveAddressChangeConflict(TLink oldLinkAddress, TLink
24     ↪ newLinkAddress)
25     {
26         if (!_equalityComparer.Equals(oldLinkAddress, newLinkAddress) &&
27             ↪ Links.Exists(oldLinkAddress))
28         {
29             Delete(oldLinkAddress);
30         }
31         return newLinkAddress;
32     }
33 }

```

./Platform.Data.Doublets/Decorators/LinksUniquenessValidator.cs

```

1 using System.Collections.Generic;
2
3 namespace Platform.Data.Doublets.Decorators
4 {
5     public class LinksUniquenessValidator<TLink> : LinksDecoratorBase<TLink>
6     {
7         public LinksUniquenessValidator(ICollection<TLink> links) : base(links) { }
8
9         public override TLink Update(ICollection<TLink> restrictions)
10        {
11            Links.EnsureDoesNotExists(restrictions[Constants.SourcePart],
12                ↪ restrictions[Constants.TargetPart]);
13            return Links.Update(restrictions);
14        }
15    }

```

./Platform.Data.Doublets/Decorators/LinksUsagesValidator.cs

```

1 using System.Collections.Generic;
2
3 namespace Platform.Data.Doublets.Decorators
4 {
5     public class LinksUsagesValidator<TLink> : LinksDecoratorBase<TLink>
6     {
7         public LinksUsagesValidator(ICollection<TLink> links) : base(links) { }
8
9         public override TLink Update(ICollection<TLink> restrictions)
10        {
11            Links.EnsureNoUsages(restrictions[Constants.IndexPart]);
12            return Links.Update(restrictions);
13        }
14
15        public override void Delete(TLink link)
16        {
17            Links.EnsureNoUsages(link);
18            Links.Delete(link);
19        }
20    }
21 }

```

./Platform.Data.Doublets/Decorators/NonNullContentsLinkDeletionResolver.cs

```

1 namespace Platform.Data.Doublets.Decorators
2 {
3     public class NonNullContentsLinkDeletionResolver<TLink> : LinksDecoratorBase<TLink>
4     {
5         public NonNullContentsLinkDeletionResolver(ICollection<TLink> links) : base(links) { }
6
7         public override void Delete(TLink linkIndex)
8         {
9             Links.EnforceResetValues(linkIndex);
10            Links.Delete(linkIndex);

```

```

11     }
12 }
13 }

```

./Platform.Data.Doublets/Decorators/UInt64Links.cs

```

1  using System;
2  using System.Collections.Generic;
3  using Platform.Collections;
4
5  namespace Platform.Data.Doublets.Decorators
6  {
7      /// <summary>
8      /// Представляет объект для работы с базой данных (файлом) в формате Links (массива связей).
9      /// </summary>
10     /// <remarks>
11     /// Возможные оптимизации:
12     /// Объединение в одном поле Source и Target с уменьшением до 32 бит.
13     ///     + меньше объём БД
14     ///     - меньше производительность
15     ///     - больше ограничение на количество связей в БД)
16     /// Ленивое хранение размеров поддеревьев (расчитываемое по мере использования БД)
17     ///     + меньше объём БД
18     ///     - больше сложность
19     ///
20     /// Текущее теоретическое ограничение на индекс связи, из-за использования 5 бит в размере
21     ///     ↳ поддеревьев для AVL баланса и флагов нитей: 2 в степени(64 минус 5 равно 59 ) равно 576
22     ///     ↳ 460 752 303 423 488
23     /// Желательно реализовать поддержку переключения между деревьями и битовыми индексами
24     ///     ↳ (битовыми строками) - вариант матрицы (выстраиваемой лениво).
25     ///
26     /// Решить отключать ли проверки при компиляции под Release. Т.е. исключения будут
27     ///     ↳ выбрасываться только при #if DEBUG
28     /// </remarks>
29     public class UInt64Links : LinksDisposableDecoratorBase<ulong>
30     {
31         public UInt64Links(ILinks<ulong> links) : base(links) { }
32
33         public override ulong Each(Func<IList<ulong>, ulong> handler, IList<ulong> restrictions)
34         {
35             this.EnsureLinkIsAnyOrExists(restrictions);
36             return Links.Each(handler, restrictions);
37         }
38
39         public override ulong Create() => Links.CreatePoint();
40
41         public override ulong Update(IList<ulong> restrictions)
42         {
43             var constants = Constants;
44             var nullConstant = constants.Null;
45             if (restrictions.IsNullOrEmpty())
46             {
47                 return nullConstant;
48             }
49             // TODO: Looks like this is a common type of exceptions linked with restrictions
50             //     ↳ support
51             if (restrictions.Count != 3)
52             {
53                 throw new NotSupportedException();
54             }
55             var indexPartConstant = constants.IndexPart;
56             var updatedLink = restrictions[indexPartConstant];
57             this.EnsureLinkExists(updatedLink,
58                 ↳ $"{nameof(restrictions)}[{nameof(indexPartConstant)}]");
59             var sourcePartConstant = constants.SourcePart;
60             var newSource = restrictions[sourcePartConstant];
61             this.EnsureLinkIsItselfOrExists(newSource,
62                 ↳ $"{nameof(restrictions)}[{nameof(sourcePartConstant)}]");
63             var targetPartConstant = constants.TargetPart;
64             var newTarget = restrictions[targetPartConstant];
65             this.EnsureLinkIsItselfOrExists(newTarget,
66                 ↳ $"{nameof(restrictions)}[{nameof(targetPartConstant)}]");
67             var existedLink = nullConstant;
68             var itselfConstant = constants.Itself;
69             if (newSource != itselfConstant && newTarget != itselfConstant)
70             {
71                 existedLink = this.SearchOrDefault(newSource, newTarget);
72             }
73             if (existedLink == nullConstant)
74             {

```

```

67         var before = Links.GetLink(updatedLink);
68         if (before[sourcePartConstant] != newSource || before[targetPartConstant] !=
        ↪ newTarget)
69         {
70             Links.Update(updatedLink, newSource == itselfConstant ? updatedLink :
        ↪ newSource,
71                                     newTarget == itselfConstant ? updatedLink :
        ↪ newTarget);
72         }
73         return updatedLink;
74     }
75     else
76     {
77         return this.MergeAndDelete(updatedLink, existedLink);
78     }
79 }
80
81 public override void Delete(ulong linkIndex)
82 {
83     Links.EnsureLinkExists(linkIndex);
84     Links.EnforceResetValues(linkIndex);
85     this.DeleteAllUsages(linkIndex);
86     Links.Delete(linkIndex);
87 }
88 }
89 }

```

# ./Platform.Data.Doublets/Decorators/UniLinks.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Linq;
4  using Platform.Collections;
5  using Platform.Collections.Arrays;
6  using Platform.Collections.Lists;
7  using Platform.Data.Universal;
8
9  namespace Platform.Data.Doublets.Decorators
10 {
11     /// <remarks>
12     /// What does empty pattern (for condition or substitution) mean? Nothing or Everything?
13     /// Now we go with nothing. And nothing is something one, but empty, and cannot be changed
14     ↪ by itself. But can cause creation (update from nothing) or deletion (update to nothing).
15     ///
16     /// TODO: Decide to change to IDoubletLinks or not to change. (Better to create
17     ↪ DefaultUniLinksBase, that contains logic itself and can be implemented using both
18     ↪ IDoubletLinks and ILinks.)
19     /// </remarks>
20     internal class UniLinks<TLink> : LinksDecoratorBase<TLink>, IUniLinks<TLink>
21     {
22         private static readonly EqualityComparer<TLink> _equalityComparer =
23         ↪ EqualityComparer<TLink>.Default;
24
25         public UniLinks(ILinks<TLink> links) : base(links) { }
26
27         private struct Transition
28         {
29             public IList<TLink> Before;
30             public IList<TLink> After;
31
32             public Transition(IList<TLink> before, IList<TLink> after)
33             {
34                 Before = before;
35                 After = after;
36             }
37         }
38
39         //public static readonly TLink NullConstant = Use<LinksCombinedConstants<TLink, TLink,
40         ↪ int>>.Single.Null;
41         //public static readonly IReadOnlyList<TLink> NullLink = new
42         ↪ ReadOnlyCollection<TLink>(new List<TLink> { NullConstant, NullConstant, NullConstant
43         ↪ });
44
45         // TODO: Подумать о том, как реализовать древовидный Restriction и Substitution
46         ↪ (Links-Expression)
47         public TLink Trigger(IList<TLink> restriction, Func<IList<TLink>, IList<TLink>, TLink>
48         ↪ matchedHandler, IList<TLink> substitution, Func<IList<TLink>, IList<TLink>, TLink>
49         ↪ substitutedHandler)
50         {
51             ///List<Transition> transitions = null;

```

```

42     ///if (!restriction.IsNullOrEmpty())
43     ///{
44     ///    // Есть причина делать проход (чтение)
45     ///    if (matchedHandler != null)
46     ///    {
47     ///        if (!substitution.IsNullOrEmpty())
48     ///        {
49     ///            // restriction => { 0, 0, 0 } | { 0 } // Create
50     ///            // substitution => { itself, 0, 0 } | { itself, itself, itself } //
    → Create / Update
51     ///            // substitution => { 0, 0, 0 } | { 0 } // Delete
52     ///            transitions = new List<Transition>();
53     ///            if (Equals(substitution[Constants.IndexPart], Constants.Null))
54     ///            {
55     ///                // If index is Null, that means we always ignore every other
    → value (they are also Null by definition)
56     ///                var matchDecision = matchedHandler(, NullLink);
57     ///                if (Equals(matchDecision, Constants.Break))
58     ///                    return false;
59     ///                if (!Equals(matchDecision, Constants.Skip))
60     ///                    transitions.Add(new Transition(matchedLink, newValue));
61     ///            }
62     ///            else
63     ///            {
64     ///                Func<T, bool> handler;
65     ///                handler = link =>
66     ///                {
67     ///                    var matchedLink = Memory.GetLinkValue(link);
68     ///                    var newValue = Memory.GetLinkValue(link);
69     ///                    newValue[Constants.IndexPart] = Constants.Itself;
70     ///                    newValue[Constants.SourcePart] =
    → Equals(substitution[Constants.SourcePart], Constants.Itself) ?
    → matchedLink[Constants.IndexPart] : substitution[Constants.SourcePart];
71     ///                    newValue[Constants.TargetPart] =
    → Equals(substitution[Constants.TargetPart], Constants.Itself) ?
    → matchedLink[Constants.IndexPart] : substitution[Constants.TargetPart];
72     ///                    var matchDecision = matchedHandler(matchedLink, newValue);
73     ///                    if (Equals(matchDecision, Constants.Break))
74     ///                        return false;
75     ///                    if (!Equals(matchDecision, Constants.Skip))
76     ///                        transitions.Add(new Transition(matchedLink, newValue));
77     ///                    return true;
78     ///                };
79     ///                if (!Memory.Each(handler, restriction))
80     ///                    return Constants.Break;
81     ///            }
82     ///        }
83     ///        else
84     ///        {
85     ///            Func<T, bool> handler = link =>
86     ///            {
87     ///                var matchedLink = Memory.GetLinkValue(link);
88     ///                var matchDecision = matchedHandler(matchedLink, matchedLink);
89     ///                return !Equals(matchDecision, Constants.Break);
90     ///            };
91     ///            if (!Memory.Each(handler, restriction))
92     ///                return Constants.Break;
93     ///        }
94     ///    }
95     ///    else
96     ///    {
97     ///        if (substitution != null)
98     ///        {
99     ///            transitions = new List<IList<T>>>();
100     ///            Func<T, bool> handler = link =>
101     ///            {
102     ///                var matchedLink = Memory.GetLinkValue(link);
103     ///                transitions.Add(matchedLink);
104     ///                return true;
105     ///            };
106     ///            if (!Memory.Each(handler, restriction))
107     ///                return Constants.Break;
108     ///        }
109     ///        else
110     ///        {
111     ///            return Constants.Continue;
112     ///        }

```

```

113         }
114     }
115     if (substitution != null)
116     {
117         // Есть причина делать замену (запись)
118         if (substitutedHandler != null)
119         {
120             //
121         }
122         else
123         {
124             //
125         }
126     }
127     return Constants.Continue;
128
129     if (restriction.IsNullOrEmpty()) // Create
130     {
131         substitution[Constants.IndexPart] = Memory.AllocateLink();
132         Memory.SetLinkValue(substitution);
133     }
134     //else if (substitution.IsNullOrEmpty()) // Delete
135     {
136         Memory.FreeLink(restriction[Constants.IndexPart]);
137     }
138     //else if (restriction.EqualTo(substitution)) // Read or ("repeat" the state) // Each
139     {
140         // No need to collect links to list
141         // Skip == Continue
142         // No need to check substitutedHandler
143         if (!Memory.Each(link => !Equals(matchedHandler(Memory.GetLinkValue(link)),
144             ↪ Constants.Break), restriction))
145         {
146             return Constants.Break;
147         }
148     }
149     //else // Update
150     {
151         //List<IList<T>> matchedLinks = null;
152         if (matchedHandler != null)
153         {
154             matchedLinks = new List<IList<T>>();
155             Func<T, bool> handler = link =>
156             {
157                 var matchedLink = Memory.GetLinkValue(link);
158                 var matchDecision = matchedHandler(matchedLink);
159                 if (Equals(matchDecision, Constants.Break))
160                     return false;
161                 if (!Equals(matchDecision, Constants.Skip))
162                     matchedLinks.Add(matchedLink);
163                 return true;
164             };
165             if (!Memory.Each(handler, restriction))
166                 return Constants.Break;
167         }
168         if (!matchedLinks.IsNullOrEmpty())
169         {
170             var totalMatchedLinks = matchedLinks.Count;
171             for (var i = 0; i < totalMatchedLinks; i++)
172             {
173                 var matchedLink = matchedLinks[i];
174                 if (substitutedHandler != null)
175                 {
176                     var newValue = new List<T>(); // TODO: Prepare value to update here
177                     // TODO: Decide is it actually needed to use Before and After
178                     ↪ substitution handling.
179                     var substitutedDecision = substitutedHandler(matchedLink,
180                     ↪ newValue);
181                     if (Equals(substitutedDecision, Constants.Break))
182                         return Constants.Break;
183                     if (Equals(substitutedDecision, Constants.Continue))
184                     {
185                         // Actual update here
186                         Memory.SetLinkValue(newValue);
187                     }
188                     if (Equals(substitutedDecision, Constants.Skip))
189                     {
190                         // Cancel the update. TODO: decide use separate Cancel
191                         ↪ constant or Skip is enough?
192                     }
193                 }
194             }
195         }
196     }

```



```

186         //      }
187         //    }
188         //}
189         return Constants.Continue;
190     }
191
192     public TLink Trigger(IList<TLink> patternOrCondition, Func<IList<TLink>, TLink>
    ↪ matchHandler, IList<TLink> substitution, Func<IList<TLink>, IList<TLink>, TLink>
    ↪ substitutionHandler)
193     {
194         if (patternOrCondition.IsNullOrEmpty() && substitution.IsNullOrEmpty())
195         {
196             return Constants.Continue;
197         }
198         else if (patternOrCondition.EqualTo(substitution)) // Should be Each here TODO:
    ↪ Check if it is a correct condition
199         {
200             // Or it only applies to trigger without matchHandler.
201             throw new NotImplementedException();
202         }
203         else if (!substitution.IsNullOrEmpty()) // Creation
204         {
205             var before = ArrayPool<TLink>.Empty;
206             // Что должно означать False здесь? Остановиться (перестать идти) или пропустить
    ↪ (пройти мимо) или пустить (взять)?
207             if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
    ↪ Constants.Break))
208             {
209                 return Constants.Break;
210             }
211             var after = (IList<TLink>)substitution.ToArray();
212             if (_equalityComparer.Equals(after[0], default))
213             {
214                 var newLink = Links.Create();
215                 after[0] = newLink;
216             }
217             if (substitution.Count == 1)
218             {
219                 after = Links.GetLink(substitution[0]);
220             }
221             else if (substitution.Count == 3)
222             {
223                 Links.Update(after);
224             }
225             else
226             {
227                 throw new NotSupportedException();
228             }
229             if (matchHandler != null)
230             {
231                 return substitutionHandler(before, after);
232             }
233             return Constants.Continue;
234         }
235         else if (!patternOrCondition.IsNullOrEmpty()) // Deletion
236         {
237             if (patternOrCondition.Count == 1)
238             {
239                 var linkToDelete = patternOrCondition[0];
240                 var before = Links.GetLink(linkToDelete);
241                 if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
    ↪ Constants.Break))
242                 {
243                     return Constants.Break;
244                 }
245                 var after = ArrayPool<TLink>.Empty;
246                 Links.Update(linkToDelete, Constants.Null, Constants.Null);
247                 Links.Delete(linkToDelete);
248                 if (matchHandler != null)
249                 {
250                     return substitutionHandler(before, after);
251                 }
252                 return Constants.Continue;
253             }
254             else
255             {
256                 throw new NotSupportedException();
257             }
258         }
259     }

```



```

4 namespace Platform.Data.Doublets
5 {
6     /// <remarks>
7     /// TODO: Может стоит попробовать ref во всех методах (IRefEqualityComparer)
8     /// 2x faster with comparer
9     /// </remarks>
10    public class DoubletComparer<T> : IEqualityComparer<Doublet<T>>
11    {
12        public static readonly DoubletComparer<T> Default = new DoubletComparer<T>();
13
14        [MethodImpl(MethodImplOptions.AggressiveInlining)]
15        public bool Equals(Doublet<T> x, Doublet<T> y) => x.Equals(y);
16
17        [MethodImpl(MethodImplOptions.AggressiveInlining)]
18        public int GetHashCode(Doublet<T> obj) => obj.GetHashCode();
19    }
20 }

```

./Platform.Data.Doublets/Doublet.cs

```

1 using System;
2 using System.Collections.Generic;
3
4 namespace Platform.Data.Doublets
5 {
6     public struct Doublet<T> : IEquatable<Doublet<T>>
7     {
8         private static readonly EqualityComparer<T> _equalityComparer =
9             EqualityComparer<T>.Default;
10
11         public T Source { get; set; }
12         public T Target { get; set; }
13
14         public Doublet(T source, T target)
15         {
16             Source = source;
17             Target = target;
18         }
19
20         public override string ToString() => $"{Source}->{Target}";
21
22         public bool Equals(Doublet<T> other) => _equalityComparer.Equals(Source, other.Source)
23             && _equalityComparer.Equals(Target, other.Target);
24
25         public override bool Equals(object obj) => obj is Doublet<T> doublet ?
26             base.Equals(doublet) : false;
27
28         public override int GetHashCode() => (Source, Target).GetHashCode();
29     }
30 }

```

./Platform.Data.Doublets/Hybrid.cs

```

1 using System;
2 using System.Reflection;
3 using Platform.Reflection;
4 using Platform.Converters;
5 using Platform.Exceptions;
6
7 namespace Platform.Data.Doublets
8 {
9     public class Hybrid<T>
10     {
11         public readonly T Value;
12         public bool IsNothing => Convert.ToInt64(To.Signed(Value)) == 0;
13         public bool IsInternal => Convert.ToInt64(To.Signed(Value)) > 0;
14         public bool IsExternal => Convert.ToInt64(To.Signed(Value)) < 0;
15         public long AbsoluteValue => Numbers.Math.Abs(Convert.ToInt64(To.Signed(Value)));
16
17         public Hybrid(T value)
18         {
19             Ensure.Always.IsUnsignedInteger<T>();
20             Value = value;
21         }
22
23         public Hybrid(object value) => Value = To.UnsignedAs<T>(Convert.ChangeType(value,
24             Type<T>.SignedVersion));
25
26         public Hybrid(object value, bool isExternal)
27         {
28             var signedType = Type<T>.SignedVersion;
29             var signedValue = Convert.ChangeType(value, signedType);
30         }
31     }
32 }

```

```

29     var abs = typeof(Numbers.Math).GetTypeInfo().GetMethod("Abs").MakeGenericMethod(sign
    ↪ edType);
30     var negate = typeof(Numbers.Math).GetTypeInfo().GetMethod("Negate").MakeGenericMetho
    ↪ d(signedType);
31     var absoluteValue = abs.Invoke(null, new[] { signedValue });
32     var resultValue = isExternal ? negate.Invoke(null, new[] { absoluteValue }) :
    ↪ absoluteValue;
33     Value = To.UnsignedAs<T>(resultValue);
34 }
35
36 public static implicit operator Hybrid<T>(T integer) => new Hybrid<T>(integer);
37
38 public static explicit operator Hybrid<T>(ulong integer) => new Hybrid<T>(integer);
39
40 public static explicit operator Hybrid<T>(long integer) => new Hybrid<T>(integer);
41
42 public static explicit operator Hybrid<T>(uint integer) => new Hybrid<T>(integer);
43
44 public static explicit operator Hybrid<T>(int integer) => new Hybrid<T>(integer);
45
46 public static explicit operator Hybrid<T>(ushort integer) => new Hybrid<T>(integer);
47
48 public static explicit operator Hybrid<T>(short integer) => new Hybrid<T>(integer);
49
50 public static explicit operator Hybrid<T>(byte integer) => new Hybrid<T>(integer);
51
52 public static explicit operator Hybrid<T>(sbyte integer) => new Hybrid<T>(integer);
53
54 public static implicit operator T(Hybrid<T> hybrid) => hybrid.Value;
55
56 public static explicit operator ulong(Hybrid<T> hybrid) =>
    ↪ Convert.ToUInt64(hybrid.Value);
57
58 public static explicit operator long(Hybrid<T> hybrid) => hybrid.AbsoluteValue;
59
60 public static explicit operator uint(Hybrid<T> hybrid) => Convert.ToUInt32(hybrid.Value);
61
62 public static explicit operator int(Hybrid<T> hybrid) =>
    ↪ Convert.ToInt32(hybrid.AbsoluteValue);
63
64 public static explicit operator ushort(Hybrid<T> hybrid) =>
    ↪ Convert.ToUInt16(hybrid.Value);
65
66 public static explicit operator short(Hybrid<T> hybrid) =>
    ↪ Convert.ToInt16(hybrid.AbsoluteValue);
67
68 public static explicit operator byte(Hybrid<T> hybrid) => Convert.ToByte(hybrid.Value);
69
70 public static explicit operator sbyte(Hybrid<T> hybrid) =>
    ↪ Convert.ToSByte(hybrid.AbsoluteValue);
71
72 public override string ToString() => IsNothing ? default(T) == null ? "Nothing" :
    ↪ default(T).ToString() : IsExternal ? $"{AbsoluteValue}" : Value.ToString();
73 }
74 }

```

./Platform.Data.Doublets/ILinks.cs

```

1 using Platform.Data.Constants;
2
3 namespace Platform.Data.Doublets
4 {
5     public interface ILinks<TLink> : ILinks<TLink, LinksCombinedConstants<TLink, TLink, int>>
6     {
7     }
8 }

```

./Platform.Data.Doublets/ILinksExtensions.cs

```

1 using System;
2 using System.Collections;
3 using System.Collections.Generic;
4 using System.Linq;
5 using System.Runtime.CompilerServices;
6 using Platform.Ranges;
7 using Platform.Collections.Arrays;
8 using Platform.Numbers;
9 using Platform.Random;
10 using Platform.Setters;
11 using Platform.Data.Exceptions;
12
13 namespace Platform.Data.Doublets

```

```

14 {
15     public static class ILinksExtensions
16     {
17         public static void RunRandomCreations<TLink>(this ILinks<TLink> links, long
18             ↳ amountOfCreations)
19         {
20             for (long i = 0; i < amountOfCreations; i++)
21             {
22                 var linksAddressRange = new Range<ulong>(0, (Integer<TLink>)links.Count());
23                 Integer<TLink> source = RandomHelpers.Default.NextUInt64(linksAddressRange);
24                 Integer<TLink> target = RandomHelpers.Default.NextUInt64(linksAddressRange);
25                 links.CreateAndUpdate(source, target);
26             }
27         }
28         public static void RunRandomSearches<TLink>(this ILinks<TLink> links, long
29             ↳ amountOfSearches)
30         {
31             for (long i = 0; i < amountOfSearches; i++)
32             {
33                 var linkAddressRange = new Range<ulong>(1, (Integer<TLink>)links.Count());
34                 Integer<TLink> source = RandomHelpers.Default.NextUInt64(linkAddressRange);
35                 Integer<TLink> target = RandomHelpers.Default.NextUInt64(linkAddressRange);
36                 links.SearchOrDefault(source, target);
37             }
38         }
39         public static void RunRandomDeletions<TLink>(this ILinks<TLink> links, long
40             ↳ amountOfDeletions)
41         {
42             var min = (ulong)amountOfDeletions > (Integer<TLink>)links.Count() ? 1 :
43                 ↳ (Integer<TLink>)links.Count() - (ulong)amountOfDeletions;
44             for (long i = 0; i < amountOfDeletions; i++)
45             {
46                 var linksAddressRange = new Range<ulong>(min, (Integer<TLink>)links.Count());
47                 Integer<TLink> link = RandomHelpers.Default.NextUInt64(linksAddressRange);
48                 links.Delete(link);
49                 if ((Integer<TLink>)links.Count() < min)
50                 {
51                     break;
52                 }
53             }
54         }
55         /// <remarks>
56         /// TODO: Возможно есть очень простой способ это сделать.
57         /// (Например просто удалить файл, или изменить его размер таким образом,
58         /// чтобы удалился весь контент)
59         /// Например через _header->AllocatedLinks в ResizableDirectMemoryLinks
60         /// </remarks>
61         public static void DeleteAll<TLink>(this ILinks<TLink> links)
62         {
63             var equalityComparer = EqualityComparer<TLink>.Default;
64             var comparer = Comparer<TLink>.Default;
65             for (var i = links.Count(); comparer.Compare(i, default) > 0; i =
66                 ↳ Arithmetic.Decrement(i))
67             {
68                 links.Delete(i);
69                 if (!equalityComparer.Equals(links.Count(), Arithmetic.Decrement(i)))
70                 {
71                     i = links.Count();
72                 }
73             }
74         }
75         public static TLink First<TLink>(this ILinks<TLink> links)
76         {
77             TLink firstLink = default;
78             var equalityComparer = EqualityComparer<TLink>.Default;
79             if (equalityComparer.Equals(links.Count(), default))
80             {
81                 throw new Exception("В хранилище нет связей.");
82             }
83             links.Each(links.Constants.Any, links.Constants.Any, link =>
84             {
85                 firstLink = link[links.Constants.IndexPart];
86                 return links.Constants.Break;
87             });
88         }
89     }
90 }

```

```

87         if (equalityComparer.Equals(firstLink, default))
88         {
89             throw new Exception("В процессе поиска по хранилищу не было найдено связей.");
90         }
91         return firstLink;
92     }
93
94     public static bool IsInnerReference<TLink>(this ILinks<TLink> links, TLink reference)
95     {
96         var constants = links.Constants;
97         var comparer = Comparer<TLink>.Default;
98         return comparer.Compare(constants.MinPossibleIndex, reference) >= 0 &&
99             ⇨ comparer.Compare(reference, constants.MaxPossibleIndex) <= 0;
100     }
101
102     #region Paths
103
104     /// <remarks>
105     /// TODO: Как так? Как то что ниже может быть корректно?
106     /// Скорее всего практически не применимо
107     /// Предполагалось, что можно было конвертировать формируемый в проходе через
108     ⇨ SequenceWalker
109     /// Stack в конкретный путь из Source, Target до связи, но это не всегда так.
110     /// TODO: Возможно нужен метод, который именно выбрасывает исключения (EnsurePathExists)
111     /// </remarks>
112     public static bool CheckPathExistence<TLink>(this ILinks<TLink> links, params TLink[]
113     ⇨ path)
114     {
115         var current = path[0];
116         //EnsureLinkExists(current, "path");
117         if (!links.Exists(current))
118         {
119             return false;
120         }
121         var equalityComparer = EqualityComparer<TLink>.Default;
122         var constants = links.Constants;
123         for (var i = 1; i < path.Length; i++)
124         {
125             var next = path[i];
126             var values = links.GetLink(current);
127             var source = values[constants.SourcePart];
128             var target = values[constants.TargetPart];
129             if (equalityComparer.Equals(source, target) && equalityComparer.Equals(source,
130             ⇨ next))
131             {
132                 //throw new Exception(string.Format("Невозможно выбрать путь, так как и
133                 ⇨ Source и Target совпадают с элементом пути {0}.", next));
134                 return false;
135             }
136             if (!equalityComparer.Equals(next, source) && !equalityComparer.Equals(next,
137             ⇨ target))
138             {
139                 //throw new Exception(string.Format("Невозможно продолжить путь через
140                 ⇨ элемент пути {0}", next));
141                 return false;
142             }
143             current = next;
144         }
145         return true;
146     }
147
148     /// <remarks>
149     /// Может потребовать дополнительного стека для PathElement's при использовании
150     ⇨ SequenceWalker.
151     /// </remarks>
152     public static TLink GetByKeyes<TLink>(this ILinks<TLink> links, TLink root, params int[]
153     ⇨ path)
154     {
155         links.EnsureLinkExists(root, "root");
156         var currentLink = root;
157         for (var i = 0; i < path.Length; i++)
158         {
159             currentLink = links.GetLink(currentLink)[path[i]];
160         }
161         return currentLink;
162     }
163
164     public static TLink GetSquareMatrixSequenceElementByIndex<TLink>(this ILinks<TLink>
165     ⇨ links, TLink root, ulong size, ulong index)

```

```

156 {
157     var constants = links.Constants;
158     var source = constants.SourcePart;
159     var target = constants.TargetPart;
160     if (!Numbers.Math.IsPowerOfTwo(size))
161     {
162         throw new ArgumentOutOfRangeException(nameof(size), "Sequences with sizes other
        ↳ than powers of two are not supported.");
163     }
164     var path = new BitArray(BitConverter.GetBytes(index));
165     var length = Bit.GetLowestPosition(size);
166     links.EnsureLinkExists(root, "root");
167     var currentLink = root;
168     for (var i = length - 1; i >= 0; i--)
169     {
170         currentLink = links.GetLink(currentLink)[path[i] ? target : source];
171     }
172     return currentLink;
173 }
174
175 #endregion
176
177 /// <summary>
178 /// Возвращает индекс указанной связи.
179 /// </summary>
180 /// <param name="links">Хранилище связей.</param>
181 /// <param name="link">Связь представленная списком, состоящим из её адреса и
182 ↳ содержимого.</param>
183 /// <returns>Индекс начальной связи для указанной связи.</returns>
184 [MethodImpl(MethodImplOptions.AggressiveInlining)]
185 public static TLink GetIndex<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
186 ↳ link[links.Constants.IndexPart];
187
188 /// <summary>
189 /// Возвращает индекс начальной (Source) связи для указанной связи.
190 /// </summary>
191 /// <param name="links">Хранилище связей.</param>
192 /// <param name="link">Индекс связи.</param>
193 /// <returns>Индекс начальной связи для указанной связи.</returns>
194 [MethodImpl(MethodImplOptions.AggressiveInlining)]
195 public static TLink GetSource<TLink>(this ILinks<TLink> links, TLink link) =>
196 ↳ links.GetLink(link)[links.Constants.SourcePart];
197
198 /// <summary>
199 /// Возвращает индекс начальной (Source) связи для указанной связи.
200 /// </summary>
201 /// <param name="links">Хранилище связей.</param>
202 /// <param name="link">Связь представленная списком, состоящим из её адреса и
203 ↳ содержимого.</param>
204 /// <returns>Индекс начальной связи для указанной связи.</returns>
205 [MethodImpl(MethodImplOptions.AggressiveInlining)]
206 public static TLink GetSource<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
207 ↳ link[links.Constants.SourcePart];
208
209 /// <summary>
210 /// Возвращает индекс конечной (Target) связи для указанной связи.
211 /// </summary>
212 /// <param name="links">Хранилище связей.</param>
213 /// <param name="link">Индекс связи.</param>
214 /// <returns>Индекс конечной связи для указанной связи.</returns>
215 [MethodImpl(MethodImplOptions.AggressiveInlining)]
216 public static TLink GetTarget<TLink>(this ILinks<TLink> links, TLink link) =>
217 ↳ links.GetLink(link)[links.Constants.TargetPart];
218
219 /// <summary>
220 /// Возвращает индекс конечной (Target) связи для указанной связи.
221 /// </summary>
222 /// <param name="links">Хранилище связей.</param>
223 /// <param name="link">Связь представленная списком, состоящим из её адреса и
224 ↳ содержимого.</param>
225 /// <returns>Индекс конечной связи для указанной связи.</returns>
226 [MethodImpl(MethodImplOptions.AggressiveInlining)]
227 public static TLink GetTarget<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
228 ↳ link[links.Constants.TargetPart];
229
230 /// <summary>
231 /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
232 ↳ (handler) для каждой подходящей связи.

```

```

224 /// </summary>
225 /// <param name="links">Хранилище связей.</param>
226 /// <param name="handler">Обработчик каждой подходящей связи.</param>
227 /// <param name="restrictions">Ограничения на содержимое связей. Каждое ограничение
    ↳ может иметь значения: Constants.Null - 0-я связь, обозначающая ссылку на пустоту,
    ↳ Any - отсутствие ограничения, 1..∞ конкретный адрес связи.</param>
228 /// <returns>True, в случае если проход по связям не был прерван и False в обратном
    ↳ случае.</returns>
229 [MethodImpl(MethodImplOptions.AggressiveInlining)]
230 public static bool Each<TLink>(this ILinks<TLink> links, Func<IList<TLink>, TLink>
    ↳ handler, params TLink[] restrictions)
231 => EqualityComparer<TLink>.Default.Equals(links.Each(handler, restrictions),
    ↳ links.Constants.Continue);
232
233 /// <summary>
234 /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
    ↳ (handler) для каждой подходящей связи.
235 /// </summary>
236 /// <param name="links">Хранилище связей.</param>
237 /// <param name="source">Значение, определяющее соответствующие шаблону связи.
    ↳ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве начала,
    ↳ Constants.Any - любое начало, 1..∞ конкретное начало)</param>
238 /// <param name="target">Значение, определяющее соответствующие шаблону связи.
    ↳ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве конца,
    ↳ Constants.Any - любой конец, 1..∞ конкретный конец)</param>
239 /// <param name="handler">Обработчик каждой подходящей связи.</param>
240 /// <returns>True, в случае если проход по связям не был прерван и False в обратном
    ↳ случае.</returns>
241 [MethodImpl(MethodImplOptions.AggressiveInlining)]
242 public static bool Each<TLink>(this ILinks<TLink> links, TLink source, TLink target,
    ↳ Func<TLink, bool> handler)
243 {
244     var constants = links.Constants;
245     return links.Each(link => handler(link[constants.IndexPart]) ? constants.Continue :
    ↳ constants.Break, constants.Any, source, target);
246 }
247
248 /// <summary>
249 /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
    ↳ (handler) для каждой подходящей связи.
250 /// </summary>
251 /// <param name="links">Хранилище связей.</param>
252 /// <param name="source">Значение, определяющее соответствующие шаблону связи.
    ↳ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве начала,
    ↳ Constants.Any - любое начало, 1..∞ конкретное начало)</param>
253 /// <param name="target">Значение, определяющее соответствующие шаблону связи.
    ↳ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве конца,
    ↳ Constants.Any - любой конец, 1..∞ конкретный конец)</param>
254 /// <param name="handler">Обработчик каждой подходящей связи.</param>
255 /// <returns>True, в случае если проход по связям не был прерван и False в обратном
    ↳ случае.</returns>
256 [MethodImpl(MethodImplOptions.AggressiveInlining)]
257 public static bool Each<TLink>(this ILinks<TLink> links, TLink source, TLink target,
    ↳ Func<IList<TLink>, TLink> handler)
258 {
259     var constants = links.Constants;
260     return links.Each(handler, constants.Any, source, target);
261 }
262
263 [MethodImpl(MethodImplOptions.AggressiveInlining)]
264 public static IList<IList<TLink>> All<TLink>(this ILinks<TLink> links, params TLink[]
    ↳ restrictions)
265 {
266     long arraySize = (Integer<TLink>)links.Count(restrictions);
267     var array = new IList<TLink>[arraySize];
268     if (arraySize > 0)
269     {
270         var filler = new ArrayFiller<IList<TLink>, TLink>(array,
            ↳ links.Constants.Continue);
271         links.Each(filler.AddAndReturnConstant, restrictions);
272     }
273     return array;
274 }
275
276 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```



```

277 public static IList<TLink> AllIndices<TLink>(this ILinks<TLink> links, params TLink[]
    ↳ restrictions)
278 {
279     long arraySize = (Integer<TLink>)links.Count(restrictions);
280     var array = new TLink[arraySize];
281     if (arraySize > 0)
282     {
283         var filler = new ArrayFiller<TLink, TLink>(array, links.Constants.Continue);
284         links.Each(filler.AddFirstAndReturnConstant, restrictions);
285     }
286     return array;
287 }
288
289 /// <summary>
290 /// Возвращает значение, определяющее существует ли связь с указанными началом и концом
    ↳ в хранилище связей.
291 /// </summary>
292 /// <param name="links">Хранилище связей.</param>
293 /// <param name="source">Начало связи.</param>
294 /// <param name="target">Конец связи.</param>
295 /// <returns>Значение, определяющее существует ли связь.</returns>
296 [MethodImpl(MethodImplOptions.AggressiveInlining)]
297 public static bool Exists<TLink>(this ILinks<TLink> links, TLink source, TLink target)
    ↳ => Comparer<TLink>.Default.Compare(links.Count(links.Constants.Any, source, target),
    ↳ default) > 0;
298
299 #region Ensure
300 // TODO: May be move to EnsureExtensions or make it both there and here
301
302 [MethodImpl(MethodImplOptions.AggressiveInlining)]
303 public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links, TLink
    ↳ reference, string argumentName)
304 {
305     if (links.IsInnerReference(reference) && !links.Exists(reference))
306     {
307         throw new ArgumentLinkDoesNotExistsException<TLink>(reference, argumentName);
308     }
309 }
310
311 [MethodImpl(MethodImplOptions.AggressiveInlining)]
312 public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links,
    ↳ IList<TLink> restrictions, string argumentName)
313 {
314     for (int i = 0; i < restrictions.Count; i++)
315     {
316         links.EnsureInnerReferenceExists(restrictions[i], argumentName);
317     }
318 }
319
320 [MethodImpl(MethodImplOptions.AggressiveInlining)]
321 public static void EnsureLinkIsAnyOrExists<TLink>(this ILinks<TLink> links, IList<TLink>
    ↳ restrictions)
322 {
323     for (int i = 0; i < restrictions.Count; i++)
324     {
325         links.EnsureLinkIsAnyOrExists(restrictions[i], nameof(restrictions));
326     }
327 }
328
329 [MethodImpl(MethodImplOptions.AggressiveInlining)]
330 public static void EnsureLinkIsAnyOrExists<TLink>(this ILinks<TLink> links, TLink link,
    ↳ string argumentName)
331 {
332     var equalityComparer = EqualityComparer<TLink>.Default;
333     if (!equalityComparer.Equals(link, links.Constants.Any) && !links.Exists(link))
334     {
335         throw new ArgumentLinkDoesNotExistsException<TLink>(link, argumentName);
336     }
337 }
338
339 [MethodImpl(MethodImplOptions.AggressiveInlining)]
340 public static void EnsureLinkIsItselfOrExists<TLink>(this ILinks<TLink> links, TLink
    ↳ link, string argumentName)
341 {
342     var equalityComparer = EqualityComparer<TLink>.Default;
343     if (!equalityComparer.Equals(link, links.Constants.Itself) && !links.Exists(link))
344     {
345         throw new ArgumentLinkDoesNotExistsException<TLink>(link, argumentName);

```

```

346     }
347 }
348
349 /// <param name="links">Хранилище связей.</param>
350 [MethodImpl(MethodImplOptions.AggressiveInlining)]
351 public static void EnsureDoesNotExists<TLink>(this ILinks<TLink> links, TLink source,
    ↪ TLink target)
352 {
353     if (links.Exists(source, target))
354     {
355         throw new LinkWithSameValueAlreadyExistsException();
356     }
357 }
358
359 /// <param name="links">Хранилище связей.</param>
360 public static void EnsureNoUsages<TLink>(this ILinks<TLink> links, TLink link)
361 {
362     if (links.HasUsages(link))
363     {
364         throw new ArgumentLinkHasDependenciesException<TLink>(link);
365     }
366 }
367
368 /// <param name="links">Хранилище связей.</param>
369 public static void EnsureCreated<TLink>(this ILinks<TLink> links, params TLink[]
    ↪ addresses) => links.EnsureCreated(links.Create, addresses);
370
371 /// <param name="links">Хранилище связей.</param>
372 public static void EnsurePointsCreated<TLink>(this ILinks<TLink> links, params TLink[]
    ↪ addresses) => links.EnsureCreated(links.CreatePoint, addresses);
373
374 /// <param name="links">Хранилище связей.</param>
375 public static void EnsureCreated<TLink>(this ILinks<TLink> links, Func<TLink> creator,
    ↪ params TLink[] addresses)
376 {
377     var constants = links.Constants;
378     var nonExistentAddresses = new HashSet<ulong>(addresses.Where(x =>
    ↪ !links.Exists(x)).Select(x => (ulong)(Integer<TLink>)x));
379     if (nonExistentAddresses.Count > 0)
380     {
381         var max = nonExistentAddresses.Max();
382         // TODO: Эту верхнюю границу нужно разрешить переопределять (проверить
    ↪ применяется ли эта логика)
383         max = System.Math.Min(max, (Integer<TLink>)constants.MaxPossibleIndex);
384         var createdLinks = new List<TLink>();
385         var equalityComparer = EqualityComparer<TLink>.Default;
386         TLink createdLink = creator();
387         while (!equalityComparer.Equals(createdLink, (Integer<TLink>)max))
388         {
389             createdLinks.Add(createdLink);
390         }
391         for (var i = 0; i < createdLinks.Count; i++)
392         {
393             if (!nonExistentAddresses.Contains((Integer<TLink>)createdLinks[i]))
394             {
395                 links.Delete(createdLinks[i]);
396             }
397         }
398     }
399 }
400
401 #endregion
402
403 /// <param name="links">Хранилище связей.</param>
404 public static ulong CountUsages<TLink>(this ILinks<TLink> links, TLink link)
405 {
406     var constants = links.Constants;
407     var values = links.GetLink(link);
408     ulong usagesAsSource = (Integer<TLink>)links.Count(new Link<TLink>(constants.Any,
    ↪ link, constants.Any));
409     var equalityComparer = EqualityComparer<TLink>.Default;
410     if (equalityComparer.Equals(values[constants.SourcePart], link))
411     {
412         usagesAsSource--;
413     }
414     ulong usagesAsTarget = (Integer<TLink>)links.Count(new Link<TLink>(constants.Any,
    ↪ constants.Any, link));
415     if (equalityComparer.Equals(values[constants.TargetPart], link))

```

```

416     {
417         usagesAsTarget--;
418     }
419     return usagesAsSource + usagesAsTarget;
420 }
421
422 /// <param name="links">Хранилище связей.</param>
423 [MethodImpl(MethodImplOptions.AggressiveInlining)]
424 public static bool HasUsages<TLink>(this ILinks<TLink> links, TLink link) =>
425     ↪ links.CountUsages(link) > 0;
426
427 /// <param name="links">Хранилище связей.</param>
428 [MethodImpl(MethodImplOptions.AggressiveInlining)]
429 public static bool Equals<TLink>(this ILinks<TLink> links, TLink link, TLink source,
430     ↪ TLink target)
431 {
432     var constants = links.Constants;
433     var values = links.GetLink(link);
434     var equalityComparer = EqualityComparer<TLink>.Default;
435     return equalityComparer.Equals(values[constants.SourcePart], source) &&
436         ↪ equalityComparer.Equals(values[constants.TargetPart], target);
437 }
438
439 /// <summary>
440 /// Выполняет поиск связи с указанными Source (началом) и Target (концом).
441 /// </summary>
442 /// <param name="links">Хранилище связей.</param>
443 /// <param name="source">Индекс связи, которая является началом для искомой
444     ↪ связи.</param>
445 /// <param name="target">Индекс связи, которая является концом для искомой связи.</param>
446 /// <returns>Индекс искомой связи с указанными Source (началом) и Target
447     ↪ (концом).</returns>
448 [MethodImpl(MethodImplOptions.AggressiveInlining)]
449 public static TLink SearchOrDefault<TLink>(this ILinks<TLink> links, TLink source, TLink
450     ↪ target)
451 {
452     var constants = links.Constants;
453     var setter = new Setter<TLink, TLink>(constants.Continue, constants.Break, default);
454     links.Each(setter.SetFirstAndReturnFalse, constants.Any, source, target);
455     return setter.Result;
456 }
457
458 /// <param name="links">Хранилище связей.</param>
459 [MethodImpl(MethodImplOptions.AggressiveInlining)]
460 public static TLink CreatePoint<TLink>(this ILinks<TLink> links)
461 {
462     var link = links.Create();
463     return links.Update(link, link, link);
464 }
465
466 /// <param name="links">Хранилище связей.</param>
467 [MethodImpl(MethodImplOptions.AggressiveInlining)]
468 public static TLink CreateAndUpdate<TLink>(this ILinks<TLink> links, TLink source, TLink
469     ↪ target) => links.Update(links.Create(), source, target);
470
471 /// <summary>
472 /// Обновляет связь с указанными началом (Source) и концом (Target)
473 /// на связь с указанными началом (NewSource) и концом (NewTarget).
474 /// </summary>
475 /// <param name="links">Хранилище связей.</param>
476 /// <param name="link">Индекс обновляемой связи.</param>
477 /// <param name="newSource">Индекс связи, которая является началом связи, на которую
478     ↪ выполняется обновление.</param>
479 /// <param name="newTarget">Индекс связи, которая является концом связи, на которую
480     ↪ выполняется обновление.</param>
481 /// <returns>Индекс обновлённой связи.</returns>
482 [MethodImpl(MethodImplOptions.AggressiveInlining)]
483 public static TLink Update<TLink>(this ILinks<TLink> links, TLink link, TLink newSource,
484     ↪ TLink newTarget) => links.Update(new Link<TLink>(link, newSource, newTarget));
485
486 /// <summary>
487 /// Обновляет связь с указанными началом (Source) и концом (Target)
488 /// на связь с указанными началом (NewSource) и концом (NewTarget).
489 /// </summary>
490 /// <param name="links">Хранилище связей.</param>

```

```

481  /// <param name="restrictions">Ограничения на содержимое связей. Каждое ограничение
482  ↪ может иметь значения: Constants.Null - 0-я связь, обозначающая ссылку на пустоту,
483  ↪ Itself - требование установить ссылку на себя, 1..∞ конкретный адрес другой
484  ↪ связи.</param>
485  /// <returns>Индекс обновлённой связи.</returns>
486  [MethodImpl(MethodImplOptions.AggressiveInlining)]
487  public static TLink Update<TLink>(this ILinks<TLink> links, params TLink[] restrictions)
488  {
489      if (restrictions.Length == 2)
490      {
491          return links.MergeAndDelete(restrictions[0], restrictions[1]);
492      }
493      if (restrictions.Length == 4)
494      {
495          return links.UpdateOrCreateOrGet(restrictions[0], restrictions[1],
496          ↪ restrictions[2], restrictions[3]);
497      }
498      else
499      {
500          return links.Update(restrictions);
501      }
502  }
503
504  [MethodImpl(MethodImplOptions.AggressiveInlining)]
505  public static IList<TLink> ResolveConstantAsSelfReference<TLink>(this ILinks<TLink>
506  ↪ links, TLink constant, IList<TLink> restrictions)
507  {
508      var equalityComparer = EqualityComparer<TLink>.Default;
509      var constants = links.Constants;
510      var index = restrictions[constants.IndexPart];
511      var source = restrictions[constants.SourcePart];
512      var target = restrictions[constants.TargetPart];
513      source = equalityComparer.Equals(source, constant) ? index : source;
514      target = equalityComparer.Equals(target, constant) ? index : target;
515      return new Link<TLink>(index, source, target);
516  }
517
518  /// <summary>
519  /// Создаёт связь (если она не существовала), либо возвращает индекс существующей связи
520  ↪ с указанными Source (началом) и Target (концом).
521  /// </summary>
522  /// <param name="links">Хранилище связей.</param>
523  /// <param name="source">Индекс связи, которая является началом на создаваемой
524  ↪ связи.</param>
525  /// <param name="target">Индекс связи, которая является концом для создаваемой
526  ↪ связи.</param>
527  /// <returns>Индекс связи, с указанным Source (началом) и Target (концом)</returns>
528  [MethodImpl(MethodImplOptions.AggressiveInlining)]
529  public static TLink GetOrCreate<TLink>(this ILinks<TLink> links, TLink source, TLink
530  ↪ target)
531  {
532      var link = links.SearchOrDefault(source, target);
533      if (EqualityComparer<TLink>.Default.Equals(link, default))
534      {
535          link = links.CreateAndUpdate(source, target);
536      }
537      return link;
538  }
539
540  /// <summary>
541  /// Обновляет связь с указанными началом (Source) и концом (Target)
542  /// на связь с указанными началом (NewSource) и концом (NewTarget).
543  /// </summary>
544  /// <param name="links">Хранилище связей.</param>
545  /// <param name="source">Индекс связи, которая является началом обновляемой
546  ↪ связи.</param>
547  /// <param name="target">Индекс связи, которая является концом обновляемой связи.</param>
548  /// <param name="newSource">Индекс связи, которая является началом связи, на которую
549  ↪ выполняется обновление.</param>
550  /// <param name="newTarget">Индекс связи, которая является концом связи, на которую
551  ↪ выполняется обновление.</param>
552  /// <returns>Индекс обновлённой связи.</returns>
553  [MethodImpl(MethodImplOptions.AggressiveInlining)]
554  public static TLink UpdateOrCreateOrGet<TLink>(this ILinks<TLink> links, TLink source,
555  ↪ TLink target, TLink newSource, TLink newTarget)
556  {
557      var equalityComparer = EqualityComparer<TLink>.Default;

```

```

545     var link = links.SearchOrDefault(source, target);
546     if (equalityComparer.Equals(link, default))
547     {
548         return links.CreateAndUpdate(newSource, newTarget);
549     }
550     if (equalityComparer.Equals(newSource, source) && equalityComparer.Equals(newTarget,
551     ↪ target))
552     {
553         return link;
554     }
555     return links.Update(link, newSource, newTarget);
556 }
557 /// <summary>Удаляет связь с указанными началом (Source) и концом (Target).</summary>
558 /// <param name="links">Хранилище связей.</param>
559 /// <param name="source">Индекс связи, которая является началом удаляемой связи.</param>
560 /// <param name="target">Индекс связи, которая является концом удаляемой связи.</param>
561 [MethodImpl(MethodImplOptions.AggressiveInlining)]
562 public static TLink DeleteIfExists<TLink>(this ILinks<TLink> links, TLink source, TLink
563 ↪ target)
564 {
565     var link = links.SearchOrDefault(source, target);
566     if (!equalityComparer<TLink>.Default.Equals(link, default))
567     {
568         links.Delete(link);
569         return link;
570     }
571     return default;
572 }
573 /// <summary>Удаляет несколько связей.</summary>
574 /// <param name="links">Хранилище связей.</param>
575 /// <param name="deletedLinks">Список адресов связей к удалению.</param>
576 [MethodImpl(MethodImplOptions.AggressiveInlining)]
577 public static void DeleteMany<TLink>(this ILinks<TLink> links, IList<TLink> deletedLinks)
578 {
579     for (int i = 0; i < deletedLinks.Count; i++)
580     {
581         links.Delete(deletedLinks[i]);
582     }
583 }
584 /// <remarks>Before execution of this method ensure that deleted link is detached (all
585 ↪ values - source and target are reset to null) or it might enter into infinite
586 ↪ recursion.</remarks>
587 public static void DeleteAllUsages<TLink>(this ILinks<TLink> links, TLink linkIndex)
588 {
589     var anyConstant = links.Constants.Any;
590     var usagesAsSourceQuery = new Link<TLink>(anyConstant, linkIndex, anyConstant);
591     links.DeleteByQuery(usagesAsSourceQuery);
592     var usagesAsTargetQuery = new Link<TLink>(anyConstant, linkIndex, anyConstant);
593     links.DeleteByQuery(usagesAsTargetQuery);
594 }
595 public static void DeleteByQuery<TLink>(this ILinks<TLink> links, Link<TLink> query)
596 {
597     var count = (Integer<TLink>)links.Count(query);
598     if (count > 0)
599     {
600         var queryResult = new TLink[count];
601         var queryResultFiller = new ArrayFiller<TLink, TLink>(queryResult,
602         ↪ links.Constants.Continue);
603         links.Each(queryResultFiller.AddFirstAndReturnConstant, query);
604         for (var i = (long)count - 1; i >= 0; i--)
605         {
606             links.Delete(queryResult[i]);
607         }
608     }
609 }
610 // TODO: Move to Platform.Data
611 public static bool AreValuesReset<TLink>(this ILinks<TLink> links, TLink linkIndex)
612 {
613     var nullConstant = links.Constants.Null;
614     var equalityComparer = EqualityComparer<TLink>.Default;
615     var link = links.GetLink(linkIndex);
616     for (int i = 1; i < link.Count; i++)
617     {

```

```

618         if (!equalityComparer.Equals(link[i], nullConstant))
619         {
620             return false;
621         }
622     }
623     return true;
624 }
625
626 // TODO: Create a universal version of this method in Platform.Data (with using of for
627     ↪ loop)
628 public static void ResetValues<TLink>(this ILinks<TLink> links, TLink linkIndex)
629 {
630     var nullConstant = links.Constants.Null;
631     var updateRequest = new Link<TLink>(linkIndex, nullConstant, nullConstant);
632     links.Update(updateRequest);
633 }
634
635 // TODO: Create a universal version of this method in Platform.Data (with using of for
636     ↪ loop)
637 public static void EnforceResetValues<TLink>(this ILinks<TLink> links, TLink linkIndex)
638 {
639     if (!links.AreValuesReset(linkIndex))
640     {
641         links.ResetValues(linkIndex);
642     }
643 }
644
645 /// <summary>
646 /// Merging two usages graphs, all children of old link moved to be children of new link
647     ↪ or deleted.
648 /// </summary>
649 public static TLink MergeUsages<TLink>(this ILinks<TLink> links, TLink oldLinkIndex,
650     ↪ TLink newLinkIndex)
651 {
652     var equalityComparer = EqualityComparer<TLink>.Default;
653     if (!equalityComparer.Equals(oldLinkIndex, newLinkIndex))
654     {
655         var constants = links.Constants;
656         var usagesAsSourceQuery = new Link<TLink>(constants.Any, oldLinkIndex,
657             ↪ constants.Any);
658         long usagesAsSourceCount = (Integer<TLink>)links.Count(usagesAsSourceQuery);
659         var usagesAsTargetQuery = new Link<TLink>(constants.Any, constants.Any,
660             ↪ oldLinkIndex);
661         long usagesAsTargetCount = (Integer<TLink>)links.Count(usagesAsTargetQuery);
662         var isStandalonePoint = Point<TLink>.IsFullPoint(links.GetLink(oldLinkIndex)) &&
663             ↪ usagesAsSourceCount == 1 && usagesAsTargetCount == 1;
664         if (!isStandalonePoint)
665         {
666             var totalUsages = usagesAsSourceCount + usagesAsTargetCount;
667             if (totalUsages > 0)
668             {
669                 var usages = ArrayPool.Allocate<TLink>(totalUsages);
670                 var usagesFiller = new ArrayFiller<TLink, TLink>(usages,
671                     ↪ links.Constants.Continue);
672                 var i = 0L;
673                 if (usagesAsSourceCount > 0)
674                 {
675                     links.Each(usagesFiller.AddFirstAndReturnConstant,
676                         ↪ usagesAsSourceQuery);
677                     for (; i < usagesAsSourceCount; i++)
678                     {
679                         var usage = usages[i];
680                         if (!equalityComparer.Equals(usage, oldLinkIndex))
681                         {
682                             links.Update(usage, newLinkIndex, links.GetTarget(usage));
683                         }
684                     }
685                 }
686                 if (usagesAsTargetCount > 0)
687                 {
688                     links.Each(usagesFiller.AddFirstAndReturnConstant,
689                         ↪ usagesAsTargetQuery);
690                     for (; i < usages.Length; i++)
691                     {
692                         var usage = usages[i];
693                         if (!equalityComparer.Equals(usage, oldLinkIndex))
694                         {
695                             links.Update(usage, links.GetSource(usage), newLinkIndex);
696                         }
697                     }
698                 }
699             }
700         }
701     }
702 }

```

```

686         }
687     }
688 }
689     ArrayPool.Free(usages);
690 }
691 }
692 }
693     return newLinkIndex;
694 }
695
696 /// <summary>
697 /// Replace one link with another (replaced link is deleted, children are updated or
698   ↳ deleted).
699 /// </summary>
700 [MethodImpl(MethodImplOptions.AggressiveInlining)]
701 public static TLink MergeAndDelete<TLink>(this ILinks<TLink> links, TLink oldLinkIndex,
702   ↳ TLink newLinkIndex)
703 {
704     var equalityComparer = EqualityComparer<TLink>.Default;
705     if (!equalityComparer.Equals(oldLinkIndex, newLinkIndex))
706     {
707         links.MergeUsages(oldLinkIndex, newLinkIndex);
708         links.Delete(oldLinkIndex);
709     }
710     return newLinkIndex;
711 }

```

#### ./Platform.Data.Doublets/Incrementers/FrequencyIncrementer.cs

```

1  using System.Collections.Generic;
2  using Platform.Interfaces;
3
4  namespace Platform.Data.Doublets.Incrementers
5  {
6      public class FrequencyIncrementer<TLink> : LinksOperatorBase<TLink>, IIncrementer<TLink>
7      {
8          private static readonly EqualityComparer<TLink> _equalityComparer =
9              ↳ EqualityComparer<TLink>.Default;
10
11          private readonly TLink _frequencyMarker;
12          private readonly TLink _unaryOne;
13          private readonly IIncrementer<TLink> _unaryNumberIncrementer;
14
15          public FrequencyIncrementer(ILinks<TLink> links, TLink frequencyMarker, TLink unaryOne,
16              ↳ IIncrementer<TLink> unaryNumberIncrementer)
17              : base(links)
18          {
19              _frequencyMarker = frequencyMarker;
20              _unaryOne = unaryOne;
21              _unaryNumberIncrementer = unaryNumberIncrementer;
22          }
23
24          public TLink Increment(TLink frequency)
25          {
26              if (_equalityComparer.Equals(frequency, default))
27              {
28                  return Links.GetOrCreate(_unaryOne, _frequencyMarker);
29              }
30              var source = Links.GetSource(frequency);
31              var incrementedSource = _unaryNumberIncrementer.Increment(source);
32              return Links.GetOrCreate(incrementedSource, _frequencyMarker);
33          }
34      }
35  }

```

#### ./Platform.Data.Doublets/Incrementers/UnaryNumberIncrementer.cs

```

1  using System.Collections.Generic;
2  using Platform.Interfaces;
3
4  namespace Platform.Data.Doublets.Incrementers
5  {
6      public class UnaryNumberIncrementer<TLink> : LinksOperatorBase<TLink>, IIncrementer<TLink>
7      {
8          private static readonly EqualityComparer<TLink> _equalityComparer =
9              ↳ EqualityComparer<TLink>.Default;
10
11          private readonly TLink _unaryOne;

```

```

12     public UnaryNumberIncrementer(ILinks<TLink> links, TLink unaryOne) : base(links) =>
13         ↪ _unaryOne = unaryOne;
14
15     public TLink Increment(TLink unaryNumber)
16     {
17         if (_equalityComparer.Equals(unaryNumber, _unaryOne))
18         {
19             return Links.GetOrCreate(_unaryOne, _unaryOne);
20         }
21         var source = Links.GetSource(unaryNumber);
22         var target = Links.GetTarget(unaryNumber);
23         if (_equalityComparer.Equals(source, target))
24         {
25             return Links.GetOrCreate(unaryNumber, _unaryOne);
26         }
27         else
28         {
29             return Links.GetOrCreate(source, Increment(target));
30         }
31     }
32 }

```

./Platform.Data.Doublets/ISynchronizedLinks.cs

```

1 using Platform.Data.Constants;
2
3 namespace Platform.Data.Doublets
4 {
5     public interface ISynchronizedLinks<TLink> : ISynchronizedLinks<TLink, ILinks<TLink>,
6         ↪ LinksCombinedConstants<TLink, TLink, int>>, ILinks<TLink>
7     {
8     }
9 }

```

./Platform.Data.Doublets/Link.cs

```

1 using System;
2 using System.Collections;
3 using System.Collections.Generic;
4 using Platform.Exceptions;
5 using Platform.Ranges;
6 using Platform.Singletons;
7 using Platform.Collections.Lists;
8 using Platform.Data.Constants;
9
10 namespace Platform.Data.Doublets
11 {
12     /// <summary>
13     /// Структура описывающая уникальную связь.
14     /// </summary>
15     public struct Link<TLink> : IEquatable<Link<TLink>>, IReadOnlyList<TLink>, IList<TLink>
16     {
17         public static readonly Link<TLink> Null = new Link<TLink>();
18
19         private static readonly LinksCombinedConstants<bool, TLink, int> _constants =
20             ↪ Default<LinksCombinedConstants<bool, TLink, int>>.Instance;
21         private static readonly EqualityComparer<TLink> _equalityComparer =
22             ↪ EqualityComparer<TLink>.Default;
23
24         private const int Length = 3;
25
26         public readonly TLink Index;
27         public readonly TLink Source;
28         public readonly TLink Target;
29
30         public Link(params TLink[] values)
31         {
32             Index = values.Length > _constants.IndexPart ? values[_constants.IndexPart] :
33                 ↪ _constants.Null;
34             Source = values.Length > _constants.SourcePart ? values[_constants.SourcePart] :
35                 ↪ _constants.Null;
36             Target = values.Length > _constants.TargetPart ? values[_constants.TargetPart] :
37                 ↪ _constants.Null;
38         }
39
40         public Link(IList<TLink> values)
41         {
42             Index = values.Count > _constants.IndexPart ? values[_constants.IndexPart] :
43                 ↪ _constants.Null;
44             Source = values.Count > _constants.SourcePart ? values[_constants.SourcePart] :
45                 ↪ _constants.Null;

```



```

39         Target = values.Count > _constants.TargetPart ? values[_constants.TargetPart] :
        ↪ _constants.Null;
40     }
41
42     public Link(TLink index, TLink source, TLink target)
43     {
44         Index = index;
45         Source = source;
46         Target = target;
47     }
48
49     public Link(TLink source, TLink target)
50         : this(_constants.Null, source, target)
51     {
52         Source = source;
53         Target = target;
54     }
55
56     public static Link<TLink> Create(TLink source, TLink target) => new Link<TLink>(source,
        ↪ target);
57
58     public override int GetHashCode() => (Index, Source, Target).GetHashCode();
59
60     public bool IsNull() => _equalityComparer.Equals(Index, _constants.Null)
        && _equalityComparer.Equals(Source, _constants.Null)
        && _equalityComparer.Equals(Target, _constants.Null);
61
62
63     public override bool Equals(object other) => other is Link<TLink> &&
        ↪ Equals((Link<TLink>)other);
64
65     public bool Equals(Link<TLink> other) => _equalityComparer.Equals(Index, other.Index)
        && _equalityComparer.Equals(Source, other.Source)
        && _equalityComparer.Equals(Target, other.Target);
66
67
68     public static string ToString(TLink index, TLink source, TLink target) => $"{index}:
        ↪ {source}->{target}";
69
70
71     public static string ToString(TLink source, TLink target) => $"{source}->{target}";
72
73     public static implicit operator TLink[] (Link<TLink> link) => link.ToArray();
74
75     public static implicit operator Link<TLink>(TLink[] linkArray) => new
        ↪ Link<TLink>(linkArray);
76
77
78     public override string ToString() => _equalityComparer.Equals(Index, _constants.Null) ?
        ↪ ToString(Source, Target) : ToString(Index, Source, Target);
79
80     #region IList
81
82     public int Count => Length;
83
84     public bool IsReadOnly => true;
85
86     public TLink this[int index]
87     {
88         get
89         {
90             Ensure.Always.ArgumentInRange(index, new Range<int>(0, Length - 1),
                ↪ nameof(index));
91             if (index == _constants.IndexPart)
92             {
93                 return Index;
94             }
95             if (index == _constants.SourcePart)
96             {
97                 return Source;
98             }
99             if (index == _constants.TargetPart)
100             {
101                 return Target;
102             }
103             throw new NotSupportedException(); // Impossible path due to
                ↪ Ensure.ArgumentInRange
104         }
105         set => throw new NotSupportedException();
106     }
107
108     IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
109

```

```

110 public IEnumerator<TLink> GetEnumerator()
111 {
112     yield return Index;
113     yield return Source;
114     yield return Target;
115 }
116
117 public void Add(TLink item) => throw new NotSupportedException();
118
119 public void Clear() => throw new NotSupportedException();
120
121 public bool Contains(TLink item) => IndexOf(item) >= 0;
122
123 public void CopyTo(TLink[] array, int arrayIndex)
124 {
125     Ensure.Always.ArgumentNotNull(array, nameof(array));
126     Ensure.Always.ArgumentInRange(arrayIndex, new Range<int>(0, array.Length - 1),
127         ↪ nameof(arrayIndex));
128     if (arrayIndex + Length > array.Length)
129     {
130         throw new InvalidOperationException();
131     }
132     array[arrayIndex++] = Index;
133     array[arrayIndex++] = Source;
134     array[arrayIndex] = Target;
135 }
136
137 public bool Remove(TLink item) => Throw.A.NotSupportedExceptionAndReturn<bool>();
138
139 public int IndexOf(TLink item)
140 {
141     if (_equalityComparer.Equals(Index, item))
142     {
143         return _constants.IndexPart;
144     }
145     if (_equalityComparer.Equals(Source, item))
146     {
147         return _constants.SourcePart;
148     }
149     if (_equalityComparer.Equals(Target, item))
150     {
151         return _constants.TargetPart;
152     }
153     return -1;
154 }
155
156 public void Insert(int index, TLink item) => throw new NotSupportedException();
157
158 public void RemoveAt(int index) => throw new NotSupportedException();
159
160 #endregion
161 }

```

./Platform.Data.Doublets/LinkExtensions.cs

```

1 namespace Platform.Data.Doublets
2 {
3     public static class LinkExtensions
4     {
5         public static bool IsFullPoint<TLink>(this Link<TLink> link) =>
6             ↪ Point<TLink>.IsFullPoint(link);
7         public static bool IsPartialPoint<TLink>(this Link<TLink> link) =>
8             ↪ Point<TLink>.IsPartialPoint(link);
9     }
10 }

```

./Platform.Data.Doublets/LinksOperatorBase.cs

```

1 namespace Platform.Data.Doublets
2 {
3     public abstract class LinksOperatorBase<TLink>
4     {
5         public ILinks<TLink> Links { get; }
6         protected LinksOperatorBase(ILinks<TLink> links) => Links = links;
7     }
8 }

```

./Platform.Data.Doublets/PropertyOperators/PropertiesOperator.cs

```

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

```

```

3 using Platform.Interfaces;
4
5 namespace Platform.Data.Doublets.PropertyOperators
6 {
7     public class PropertiesOperator<TLink> : LinksOperatorBase<TLink>,
8     ↪ IPropertiesOperator<TLink, TLink, TLink>
9     {
10         private static readonly EqualityComparer<TLink> _equalityComparer =
11         ↪ EqualityComparer<TLink>.Default;
12
13         public PropertiesOperator(ILinks<TLink> links) : base(links) { }
14
15         public TLink GetValue(TLink @object, TLink property)
16         {
17             var objectProperty = Links.SearchOrDefault(@object, property);
18             if (_equalityComparer.Equals(objectProperty, default))
19             {
20                 return default;
21             }
22             var valueLink = Links.All(Links.Constants.Any, objectProperty).SingleOrDefault();
23             if (valueLink == null)
24             {
25                 return default;
26             }
27             return Links.GetTarget(valueLink[Links.Constants.IndexPart]);
28         }
29
30         public void SetValue(TLink @object, TLink property, TLink value)
31         {
32             var objectProperty = Links.GetOrCreate(@object, property);
33             Links.DeleteMany(Links.AllIndices(Links.Constants.Any, objectProperty));
34             Links.GetOrCreate(objectProperty, value);
35         }
36     }
37 }

```

./Platform.Data.Doublets/PropertyOperators/PropertyOperator.cs

```

1 using System.Collections.Generic;
2 using Platform.Interfaces;
3
4 namespace Platform.Data.Doublets.PropertyOperators
5 {
6     public class PropertyOperator<TLink> : LinksOperatorBase<TLink>, IPropertyOperator<TLink,
7     ↪ TLink>
8     {
9         private static readonly EqualityComparer<TLink> _equalityComparer =
10         ↪ EqualityComparer<TLink>.Default;
11
12         private readonly TLink _propertyMarker;
13         private readonly TLink _propertyValueMarker;
14
15         public PropertyOperator(ILinks<TLink> links, TLink propertyMarker, TLink
16         ↪ propertyValueMarker) : base(links)
17         {
18             _propertyMarker = propertyMarker;
19             _propertyValueMarker = propertyValueMarker;
20         }
21
22         public TLink Get(TLink link)
23         {
24             var property = Links.SearchOrDefault(link, _propertyMarker);
25             var container = GetContainer(property);
26             var value = GetValue(container);
27             return value;
28         }
29
30         private TLink GetContainer(TLink property)
31         {
32             var valueContainer = default(TLink);
33             if (_equalityComparer.Equals(property, default))
34             {
35                 return valueContainer;
36             }
37             var constants = Links.Constants;
38             var continueConstant = constants.Continue;
39             var breakConstant = constants.Break;
40             var anyConstant = constants.Any;
41             var query = new Link<TLink>(anyConstant, property, anyConstant);
42             Links.Each(candidate =>

```

```

41         var candidateTarget = Links.GetTarget(candidate);
42         var valueTarget = Links.GetTarget(candidateTarget);
43         if (_equalityComparer.Equals(valueTarget, _propertyValueMarker))
44         {
45             valueContainer = Links.GetIndex(candidate);
46             return breakConstant;
47         }
48         return countinueConstant;
49     }, query);
50     return valueContainer;
51 }
52
53 private TLink GetValue(TLink container) => _equalityComparer.Equals(container, default)
54     ↪ ? default : Links.GetTarget(container);
55
56 public void Set(TLink link, TLink value)
57 {
58     var property = Links.GetOrCreate(link, _propertyMarker);
59     var container = GetContainer(property);
60     if (_equalityComparer.Equals(container, default))
61     {
62         Links.GetOrCreate(property, value);
63     }
64     else
65     {
66         Links.Update(container, property, value);
67     }
68 }
69 }

```

./Platform.Data.Doublets/ResizableDirectMemory/ResizableDirectMemoryLinks.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using System.Runtime.InteropServices;
5  using Platform.Disposables;
6  using Platform.Singletons;
7  using Platform.Collections.Arrays;
8  using Platform.Numbers;
9  using Platform.Unsafe;
10 using Platform.Memory;
11 using Platform.Data.Exceptions;
12 using Platform.Data.Constants;
13 using static Platform.Numbers.Arithmetic;
14
15 #pragma warning disable 0649
16 #pragma warning disable 169
17 #pragma warning disable 618
18
19 // ReSharper disable StaticMemberInGenericType
20 // ReSharper disable BuiltInTypeReferenceStyle
21 // ReSharper disable MemberCanBePrivate.Local
22 // ReSharper disable UnusedMember.Local
23
24 namespace Platform.Data.Doublets.ResizableDirectMemory
25 {
26     public partial class ResizableDirectMemoryLinks<TLink> : DisposableBase, ILinks<TLink>
27     {
28         private static readonly EqualityComparer<TLink> _equalityComparer =
29             ↪ EqualityComparer<TLink>.Default;
30         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
31
32         /// <summary>Возвращает размер одной связи в байтах.</summary>
33         public static readonly int LinkSizeInBytes = Structure<Link>.Size;
34
35         public static readonly int LinkHeaderSizeInBytes = Structure<LinksHeader>.Size;
36
37         public static readonly long DefaultLinksSizeStep = LinkSizeInBytes * 1024 * 1024;
38
39         private struct Link
40         {
41             public static readonly int SourceOffset = Marshal.OffsetOf(typeof(Link),
42                 ↪ nameof(Source)).ToInt32();
43             public static readonly int TargetOffset = Marshal.OffsetOf(typeof(Link),
44                 ↪ nameof(Target)).ToInt32();
45             public static readonly int LeftAsSourceOffset = Marshal.OffsetOf(typeof(Link),
46                 ↪ nameof(LeftAsSource)).ToInt32();
47             public static readonly int RightAsSourceOffset = Marshal.OffsetOf(typeof(Link),
48                 ↪ nameof(RightAsSource)).ToInt32();

```

```

44     public static readonly int SizeAsSourceOffset = Marshal.OffsetOf(typeof(Link),
45         ↳ nameof(SizeAsSource)).ToInt32();
46     public static readonly int LeftAsTargetOffset = Marshal.OffsetOf(typeof(Link),
47         ↳ nameof(LeftAsTarget)).ToInt32();
48     public static readonly int RightAsTargetOffset = Marshal.OffsetOf(typeof(Link),
49         ↳ nameof(RightAsTarget)).ToInt32();
50     public static readonly int SizeAsTargetOffset = Marshal.OffsetOf(typeof(Link),
51         ↳ nameof(SizeAsTarget)).ToInt32();
52
53     public TLink Source;
54     public TLink Target;
55     public TLink LeftAsSource;
56     public TLink RightAsSource;
57     public TLink SizeAsSource;
58     public TLink LeftAsTarget;
59     public TLink RightAsTarget;
60     public TLink SizeAsTarget;
61
62     [MethodImpl(MethodImplOptions.AggressiveInlining)]
63     public static TLink GetSource(IntPtr pointer) => (pointer +
64         ↳ SourceOffset).GetValue<TLink>();
65     [MethodImpl(MethodImplOptions.AggressiveInlining)]
66     public static TLink GetTarget(IntPtr pointer) => (pointer +
67         ↳ TargetOffset).GetValue<TLink>();
68     [MethodImpl(MethodImplOptions.AggressiveInlining)]
69     public static TLink GetLeftAsSource(IntPtr pointer) => (pointer +
70         ↳ LeftAsSourceOffset).GetValue<TLink>();
71     [MethodImpl(MethodImplOptions.AggressiveInlining)]
72     public static TLink GetRightAsSource(IntPtr pointer) => (pointer +
73         ↳ RightAsSourceOffset).GetValue<TLink>();
74     [MethodImpl(MethodImplOptions.AggressiveInlining)]
75     public static TLink GetSizeAsSource(IntPtr pointer) => (pointer +
76         ↳ SizeAsSourceOffset).GetValue<TLink>();
77     [MethodImpl(MethodImplOptions.AggressiveInlining)]
78     public static TLink GetLeftAsTarget(IntPtr pointer) => (pointer +
79         ↳ LeftAsTargetOffset).GetValue<TLink>();
80     [MethodImpl(MethodImplOptions.AggressiveInlining)]
81     public static TLink GetRightAsTarget(IntPtr pointer) => (pointer +
82         ↳ RightAsTargetOffset).GetValue<TLink>();
83     [MethodImpl(MethodImplOptions.AggressiveInlining)]
84     public static TLink GetSizeAsTarget(IntPtr pointer) => (pointer +
85         ↳ SizeAsTargetOffset).GetValue<TLink>();
86
87     [MethodImpl(MethodImplOptions.AggressiveInlining)]
88     public static void SetSource(IntPtr pointer, TLink value) => (pointer +
89         ↳ SourceOffset).SetValue(value);
90     [MethodImpl(MethodImplOptions.AggressiveInlining)]
91     public static void SetTarget(IntPtr pointer, TLink value) => (pointer +
92         ↳ TargetOffset).SetValue(value);
93     [MethodImpl(MethodImplOptions.AggressiveInlining)]
94     public static void SetLeftAsSource(IntPtr pointer, TLink value) => (pointer +
95         ↳ LeftAsSourceOffset).SetValue(value);
96     [MethodImpl(MethodImplOptions.AggressiveInlining)]
97     public static void SetRightAsSource(IntPtr pointer, TLink value) => (pointer +
98         ↳ RightAsSourceOffset).SetValue(value);
99     [MethodImpl(MethodImplOptions.AggressiveInlining)]
100    public static void SetSizeAsSource(IntPtr pointer, TLink value) => (pointer +
101        ↳ SizeAsSourceOffset).SetValue(value);
102    [MethodImpl(MethodImplOptions.AggressiveInlining)]
103    public static void SetLeftAsTarget(IntPtr pointer, TLink value) => (pointer +
104        ↳ LeftAsTargetOffset).SetValue(value);
105    [MethodImpl(MethodImplOptions.AggressiveInlining)]
106    public static void SetRightAsTarget(IntPtr pointer, TLink value) => (pointer +
107        ↳ RightAsTargetOffset).SetValue(value);
108    [MethodImpl(MethodImplOptions.AggressiveInlining)]
109    public static void SetSizeAsTarget(IntPtr pointer, TLink value) => (pointer +
110        ↳ SizeAsTargetOffset).SetValue(value);
111
112 }
113
114 private struct LinksHeader
115 {
116     public static readonly int AllocatedLinksOffset =
117         ↳ Marshal.OffsetOf(typeof(LinksHeader), nameof(AllocatedLinks)).ToInt32();
118     public static readonly int ReservedLinksOffset =
119         ↳ Marshal.OffsetOf(typeof(LinksHeader), nameof(ReservedLinks)).ToInt32();
120     public static readonly int FreeLinksOffset = Marshal.OffsetOf(typeof(LinksHeader),
121         ↳ nameof(FreeLinks)).ToInt32();

```

```

98     public static readonly int FirstFreeLinkOffset =
99         ↳ Marshal.OffsetOf(typeof(LinksHeader), nameof(FirstFreeLink)).ToInt32();
100    public static readonly int FirstAsSourceOffset =
101        ↳ Marshal.OffsetOf(typeof(LinksHeader), nameof(FirstAsSource)).ToInt32();
102    public static readonly int FirstAsTargetOffset =
103        ↳ Marshal.OffsetOf(typeof(LinksHeader), nameof(FirstAsTarget)).ToInt32();
104    public static readonly int LastFreeLinkOffset =
105        ↳ Marshal.OffsetOf(typeof(LinksHeader), nameof(LastFreeLink)).ToInt32();
106
107    public TLink AllocatedLinks;
108    public TLink ReservedLinks;
109    public TLink FreeLinks;
110    public TLink FirstFreeLink;
111    public TLink FirstAsSource;
112    public TLink FirstAsTarget;
113    public TLink LastFreeLink;
114    public TLink Reserved8;
115
116    [MethodImpl(MethodImplOptions.AggressiveInlining)]
117    public static TLink GetAllocatedLinks(IntPtr pointer) => (pointer +
118        ↳ AllocatedLinksOffset).GetValue<TLink>();
119    [MethodImpl(MethodImplOptions.AggressiveInlining)]
120    public static TLink GetReservedLinks(IntPtr pointer) => (pointer +
121        ↳ ReservedLinksOffset).GetValue<TLink>();
122    [MethodImpl(MethodImplOptions.AggressiveInlining)]
123    public static TLink GetFreeLinks(IntPtr pointer) => (pointer +
124        ↳ FreeLinksOffset).GetValue<TLink>();
125    [MethodImpl(MethodImplOptions.AggressiveInlining)]
126    public static TLink GetFirstFreeLink(IntPtr pointer) => (pointer +
127        ↳ FirstFreeLinkOffset).GetValue<TLink>();
128    [MethodImpl(MethodImplOptions.AggressiveInlining)]
129    public static TLink GetFirstAsSource(IntPtr pointer) => (pointer +
130        ↳ FirstAsSourceOffset).GetValue<TLink>();
131    [MethodImpl(MethodImplOptions.AggressiveInlining)]
132    public static TLink GetFirstAsTarget(IntPtr pointer) => (pointer +
133        ↳ FirstAsTargetOffset).GetValue<TLink>();
134    [MethodImpl(MethodImplOptions.AggressiveInlining)]
135    public static TLink GetLastFreeLink(IntPtr pointer) => (pointer +
136        ↳ LastFreeLinkOffset).GetValue<TLink>();
137
138    [MethodImpl(MethodImplOptions.AggressiveInlining)]
139    public static IntPtr GetFirstAsSourcePointer(IntPtr pointer) => pointer +
140        ↳ FirstAsSourceOffset;
141    [MethodImpl(MethodImplOptions.AggressiveInlining)]
142    public static IntPtr GetFirstAsTargetPointer(IntPtr pointer) => pointer +
143        ↳ FirstAsTargetOffset;
144
145    [MethodImpl(MethodImplOptions.AggressiveInlining)]
146    public static void SetAllocatedLinks(IntPtr pointer, TLink value) => (pointer +
147        ↳ AllocatedLinksOffset).SetValue(value);
148    [MethodImpl(MethodImplOptions.AggressiveInlining)]
149    public static void SetReservedLinks(IntPtr pointer, TLink value) => (pointer +
150        ↳ ReservedLinksOffset).SetValue(value);
151    [MethodImpl(MethodImplOptions.AggressiveInlining)]
152    public static void SetFreeLinks(IntPtr pointer, TLink value) => (pointer +
153        ↳ FreeLinksOffset).SetValue(value);
154    [MethodImpl(MethodImplOptions.AggressiveInlining)]
155    public static void SetFirstFreeLink(IntPtr pointer, TLink value) => (pointer +
156        ↳ FirstFreeLinkOffset).SetValue(value);
157    [MethodImpl(MethodImplOptions.AggressiveInlining)]
158    public static void SetFirstAsSource(IntPtr pointer, TLink value) => (pointer +
159        ↳ FirstAsSourceOffset).SetValue(value);
160    [MethodImpl(MethodImplOptions.AggressiveInlining)]
161    public static void SetFirstAsTarget(IntPtr pointer, TLink value) => (pointer +
162        ↳ FirstAsTargetOffset).SetValue(value);
163    [MethodImpl(MethodImplOptions.AggressiveInlining)]
164    public static void SetLastFreeLink(IntPtr pointer, TLink value) => (pointer +
165        ↳ LastFreeLinkOffset).SetValue(value);
166
167    }
168
169    private readonly long _memoryReservationStep;
170
171    private readonly IResizableDirectMemory _memory;
172    private IntPtr _header;
173    private IntPtr _links;
174
175    private LinksTargetsTreeMethods _targetsTreeMethods;
176    private LinksSourcesTreeMethods _sourcesTreeMethods;

```

```

156 // TODO: Возможно чтобы гарантированно проверять на то, является ли связь удалённой,
157 ↪ нужно использовать не список а дерево, так как так можно быстрее проверить на
158 ↪ наличие связи внутри
159 private UnusedLinksListMethods _unusedLinksListMethods;
160
161 /// <summary>
162 /// Возвращает общее число связей находящихся в хранилище.
163 /// </summary>
164 private TLink Total => Subtract(LinksHeader.GetAllocatedLinks(_header),
165 ↪ LinksHeader.GetFreeLinks(_header));
166
167 public LinksCombinedConstants<TLink, TLink, int> Constants { get; }
168
169 public ResizableDirectMemoryLinks(string address)
170 : this(address, DefaultLinksSizeStep)
171 {
172 }
173
174 /// <summary>
175 /// Создаёт экземпляр базы данных Links в файле по указанному адресу, с указанным
176 ↪ минимальным шагом расширения базы данных.
177 /// </summary>
178 /// <param name="address">Полный путь к файлу базы данных.</param>
179 /// <param name="memoryReservationStep">Минимальный шаг расширения базы данных в
180 ↪ байтах.</param>
181 public ResizableDirectMemoryLinks(string address, long memoryReservationStep)
182 : this(new FileMappedResizableDirectMemory(address, memoryReservationStep),
183 ↪ memoryReservationStep)
184 {
185 }
186
187 public ResizableDirectMemoryLinks(IResizableDirectMemory memory)
188 : this(memory, DefaultLinksSizeStep)
189 {
190 }
191
192 public ResizableDirectMemoryLinks(IResizableDirectMemory memory, long
193 ↪ memoryReservationStep)
194 {
195     Constants = Default<LinksCombinedConstants<TLink, TLink, int>>.Instance;
196     _memory = memory;
197     _memoryReservationStep = memoryReservationStep;
198     if (memory.ReservedCapacity < memoryReservationStep)
199     {
200         memory.ReservedCapacity = memoryReservationStep;
201     }
202     SetPointers(_memory);
203     // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
204     _memory.UsedCapacity = ((long)(Integer<TLink>)LinksHeader.GetAllocatedLinks(_header)
205 ↪ * LinkSizeInBytes) + LinkHeaderSizeInBytes;
206     // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
207     LinksHeader.SetReservedLinks(_header, (Integer<TLink>)((_memory.ReservedCapacity -
208 ↪ LinkHeaderSizeInBytes) / LinkSizeInBytes));
209 }
210
211 [MethodImpl(MethodImplOptions.AggressiveInlining)]
212 public TLink Count(IList<TLink> restrictions)
213 {
214     // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
215     if (restrictions.Count == 0)
216     {
217         return Total;
218     }
219     if (restrictions.Count == 1)
220     {
221         var index = restrictions[Constants.IndexPart];
222         if (_equalityComparer.Equals(index, Constants.Any))
223         {
224             return Total;
225         }
226         return Exists(index) ? Integer<TLink>.One : Integer<TLink>.Zero;
227     }
228     if (restrictions.Count == 2)
229     {
230         var index = restrictions[Constants.IndexPart];
231         var value = restrictions[1];
232         if (_equalityComparer.Equals(index, Constants.Any))

```

```

225 {
226     if (_equalityComparer.Equals(value, Constants.Any))
227     {
228         return Total; // Any - как отсутствие ограничения
229     }
230     return Add(_sourcesTreeMethods.CountUsages(value),
231         ↪ _targetsTreeMethods.CountUsages(value));
232 }
233 else
234 {
235     if (!Exists(index))
236     {
237         return Integer<TLink>.Zero;
238     }
239     if (_equalityComparer.Equals(value, Constants.Any))
240     {
241         return Integer<TLink>.One;
242     }
243     var storedLinkValue = GetLinkUnsafe(index);
244     if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), value) ||
245         ↪ _equalityComparer.Equals(Link.GetTarget(storedLinkValue), value))
246     {
247         return Integer<TLink>.One;
248     }
249     return Integer<TLink>.Zero;
250 }
251 if (restrictions.Count == 3)
252 {
253     var index = restrictions[Constants.IndexPart];
254     var source = restrictions[Constants.SourcePart];
255     var target = restrictions[Constants.TargetPart];
256
257     if (_equalityComparer.Equals(index, Constants.Any))
258     {
259         if (_equalityComparer.Equals(source, Constants.Any) &&
260             ↪ _equalityComparer.Equals(target, Constants.Any))
261         {
262             return Total;
263         }
264         else if (_equalityComparer.Equals(source, Constants.Any))
265         {
266             return _targetsTreeMethods.CountUsages(target);
267         }
268         else if (_equalityComparer.Equals(target, Constants.Any))
269         {
270             return _sourcesTreeMethods.CountUsages(source);
271         }
272         else //if(source != Any && target != Any)
273         {
274             // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
275             var link = _sourcesTreeMethods.Search(source, target);
276             return _equalityComparer.Equals(link, Constants.Null) ?
277                 ↪ Integer<TLink>.Zero : Integer<TLink>.One;
278         }
279     }
280     else
281     {
282         if (!Exists(index))
283         {
284             return Integer<TLink>.Zero;
285         }
286         if (_equalityComparer.Equals(source, Constants.Any) &&
287             ↪ _equalityComparer.Equals(target, Constants.Any))
288         {
289             return Integer<TLink>.One;
290         }
291         var storedLinkValue = GetLinkUnsafe(index);
292         if (!_equalityComparer.Equals(source, Constants.Any) &&
293             ↪ !_equalityComparer.Equals(target, Constants.Any))
294         {
295             if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), source) &&
296                 ↪ _equalityComparer.Equals(Link.GetTarget(storedLinkValue), target))
297             {
298                 return Integer<TLink>.One;
299             }
300             return Integer<TLink>.Zero;
301         }
302     }
303 }

```



```

298     var value = default(TLink);
299     if (_equalityComparer.Equals(source, Constants.Any))
300     {
301         value = target;
302     }
303     if (_equalityComparer.Equals(target, Constants.Any))
304     {
305         value = source;
306     }
307     if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), value) ||
308         _equalityComparer.Equals(Link.GetTarget(storedLinkValue), value))
309     {
310         return Integer<TLink>.One;
311     }
312     return Integer<TLink>.Zero;
313 }
314 }
315 throw new NotSupportedException("Другие размеры и способы ограничений не
    ↳ поддерживаются.");
316 }
317
318 [MethodImpl(MethodImplOptions.AggressiveInlining)]
319 public TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
320 {
321     if (restrictions.Count == 0)
322     {
323         for (TLink link = Integer<TLink>.One; _comparer.Compare(link,
324             ↳ (Integer<TLink>)LinksHeader.GetAllocatedLinks(_header)) <= 0; link =
325             ↳ Increment(link))
326         {
327             if (Exists(link) && _equalityComparer.Equals(handler(GetLinkStruct(link)),
328                 ↳ Constants.Break))
329             {
330                 return Constants.Break;
331             }
332         }
333         return Constants.Continue;
334     }
335     if (restrictions.Count == 1)
336     {
337         var index = restrictions[Constants.IndexPart];
338         if (_equalityComparer.Equals(index, Constants.Any))
339         {
340             return Each(handler, ArrayPool<TLink>.Empty);
341         }
342         if (!Exists(index))
343         {
344             return Constants.Continue;
345         }
346         return handler(GetLinkStruct(index));
347     }
348     if (restrictions.Count == 2)
349     {
350         var index = restrictions[Constants.IndexPart];
351         var value = restrictions[1];
352         if (_equalityComparer.Equals(index, Constants.Any))
353         {
354             if (_equalityComparer.Equals(value, Constants.Any))
355             {
356                 return Each(handler, ArrayPool<TLink>.Empty);
357             }
358             if (_equalityComparer.Equals(Each(handler, new[] { index, value,
359                 ↳ Constants.Any }), Constants.Break))
360             {
361                 return Constants.Break;
362             }
363             return Each(handler, new[] { index, Constants.Any, value });
364         }
365         else
366         {
367             if (!Exists(index))
368             {
369                 return Constants.Continue;
370             }
371             if (_equalityComparer.Equals(value, Constants.Any))
372             {
373                 return handler(GetLinkStruct(index));
374             }
375         }
376     }
377 }

```

```

371     }
372     var storedLinkValue = GetLinkUnsafe(index);
373     if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), value) ||
374         _equalityComparer.Equals(Link.GetTarget(storedLinkValue), value))
375     {
376         return handler(GetLinkStruct(index));
377     }
378     return Constants.Continue;
379 }
380 }
381 if (restrictions.Count == 3)
382 {
383     var index = restrictions[Constants.IndexPart];
384     var source = restrictions[Constants.SourcePart];
385     var target = restrictions[Constants.TargetPart];
386     if (_equalityComparer.Equals(index, Constants.Any))
387     {
388         if (_equalityComparer.Equals(source, Constants.Any) &&
389             ↪ _equalityComparer.Equals(target, Constants.Any))
390         {
391             return Each(handler, ArrayPool<TLink>.Empty);
392         }
393         else if (_equalityComparer.Equals(source, Constants.Any))
394         {
395             return _targetsTreeMethods.EachUsage(target, handler);
396         }
397         else if (_equalityComparer.Equals(target, Constants.Any))
398         {
399             return _sourcesTreeMethods.EachUsage(source, handler);
400         }
401         else //if(source != Any && target != Any)
402         {
403             var link = _sourcesTreeMethods.Search(source, target);
404             return _equalityComparer.Equals(link, Constants.Null) ?
405                 ↪ Constants.Continue : handler(GetLinkStruct(link));
406         }
407     }
408     else
409     {
410         if (!Exists(index))
411         {
412             return Constants.Continue;
413         }
414         if (_equalityComparer.Equals(source, Constants.Any) &&
415             ↪ _equalityComparer.Equals(target, Constants.Any))
416         {
417             return handler(GetLinkStruct(index));
418         }
419         var storedLinkValue = GetLinkUnsafe(index);
420         if (!_equalityComparer.Equals(source, Constants.Any) &&
421             ↪ !_equalityComparer.Equals(target, Constants.Any))
422         {
423             if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), source) &&
424                 _equalityComparer.Equals(Link.GetTarget(storedLinkValue), target))
425             {
426                 return handler(GetLinkStruct(index));
427             }
428             return Constants.Continue;
429         }
430         var value = default(TLink);
431         if (_equalityComparer.Equals(source, Constants.Any))
432         {
433             value = target;
434         }
435         if (_equalityComparer.Equals(target, Constants.Any))
436         {
437             value = source;
438         }
439         if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), value) ||
440             _equalityComparer.Equals(Link.GetTarget(storedLinkValue), value))
441         {
442             return handler(GetLinkStruct(index));
443         }
444         return Constants.Continue;
445     }
446 }
447 throw new NotSupportedException("Другие размеры и способы ограничений не
448     ↪ поддерживаются.");

```

```

444 }
445
446 /// <remarks>
447 /// TODO: Возможно можно перемещать значения, если указан индекс, но значение существует
448   ↳ в другом месте (но не в менеджере памяти, а в логике Links)
449 /// </remarks>
450 [MethodImpl(MethodImplOptions.AggressiveInlining)]
451 public TLink Update(ICollection<TLink> values)
452 {
453     var linkIndex = values[Constants.IndexPart];
454     var link = GetLinkUnsafe(linkIndex);
455     // Будет корректно работать только в том случае, если пространство выделенной связи
456     ↳ предварительно заполнено нулями
457     if (!_equalityComparer.Equals(Link.GetSource(link), Constants.Null))
458     {
459         _sourcesTreeMethods.Detach(LinksHeader.GetFirstAsSourcePointer(_header),
460             ↳ linkIndex);
461     }
462     if (!_equalityComparer.Equals(Link.GetTarget(link), Constants.Null))
463     {
464         _targetsTreeMethods.Detach(LinksHeader.GetFirstAsTargetPointer(_header),
465             ↳ linkIndex);
466     }
467     Link.SetSource(link, values[Constants.SourcePart]);
468     Link.SetTarget(link, values[Constants.TargetPart]);
469     if (!_equalityComparer.Equals(Link.GetSource(link), Constants.Null))
470     {
471         _sourcesTreeMethods.Attach(LinksHeader.GetFirstAsSourcePointer(_header),
472             ↳ linkIndex);
473     }
474     if (!_equalityComparer.Equals(Link.GetTarget(link), Constants.Null))
475     {
476         _targetsTreeMethods.Attach(LinksHeader.GetFirstAsTargetPointer(_header),
477             ↳ linkIndex);
478     }
479     return linkIndex;
480 }
481
482 [MethodImpl(MethodImplOptions.AggressiveInlining)]
483 public Link<TLink> GetLinkStruct(TLink linkIndex)
484 {
485     var link = GetLinkUnsafe(linkIndex);
486     return new Link<TLink>(linkIndex, Link.GetSource(link), Link.GetTarget(link));
487 }
488
489 [MethodImpl(MethodImplOptions.AggressiveInlining)]
490 private IntPtr GetLinkUnsafe(TLink linkIndex) => _links.GetElement(LinkSizeInBytes,
491     ↳ linkIndex);
492
493 /// <remarks>
494 /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
495   ↳ пространство
496 /// </remarks>
497 public TLink Create()
498 {
499     var freeLink = LinksHeader.GetFirstFreeLink(_header);
500     if (!_equalityComparer.Equals(freeLink, Constants.Null))
501     {
502         _unusedLinksListMethods.Detach(freeLink);
503     }
504     else
505     {
506         if (_comparer.Compare(LinksHeader.GetAllocatedLinks(_header),
507             ↳ Constants.MaxPossibleIndex) > 0)
508         {
509             throw new
510                 ↳ LinksLimitReachedException((Integer<TLink>)Constants.MaxPossibleIndex);
511         }
512         if (_comparer.Compare(LinksHeader.GetAllocatedLinks(_header),
513             ↳ Decrement(LinksHeader.GetReservedLinks(_header))) >= 0)
514         {
515             _memory.ReservedCapacity += _memory.ReservationStep;
516             SetPointers(_memory);
517             LinksHeader.SetReservedLinks(_header,
518                 ↳ (Integer<TLink>)(_memory.ReservedCapacity / LinkSizeInBytes));
519         }
520         LinksHeader.SetAllocatedLinks(_header,
521             ↳ Increment(LinksHeader.GetAllocatedLinks(_header)));
522     }
523 }

```

```

509         _memory.UsedCapacity += LinkSizeInBytes;
510         freeLink = LinksHeader.GetAllocatedLinks(_header);
511     }
512     return freeLink;
513 }
514
515 public void Delete(TLink link)
516 {
517     if (_comparer.Compare(link, LinksHeader.GetAllocatedLinks(_header)) < 0)
518     {
519         _unusedLinksListMethods.AttachAsFirst(link);
520     }
521     else if (_equalityComparer.Equals(link, LinksHeader.GetAllocatedLinks(_header)))
522     {
523         LinksHeader.SetAllocatedLinks(_header,
524             ↪ Decrement(LinksHeader.GetAllocatedLinks(_header)));
525         _memory.UsedCapacity -= LinkSizeInBytes;
526         // Убираем все связи, находящиеся в списке свободных в конце файла, до тех пор,
527         // пока не дойдём до первой существующей связи
528         // Позволяет оптимизировать количество выделенных связей (AllocatedLinks)
529         while ((_comparer.Compare(LinksHeader.GetAllocatedLinks(_header),
530             ↪ Integer<TLink>.Zero) > 0) &&
531             ↪ IsUnusedLink(LinksHeader.GetAllocatedLinks(_header)))
532         {
533             _unusedLinksListMethods.Detach(LinksHeader.GetAllocatedLinks(_header));
534             LinksHeader.SetAllocatedLinks(_header,
535                 ↪ Decrement(LinksHeader.GetAllocatedLinks(_header)));
536             _memory.UsedCapacity -= LinkSizeInBytes;
537         }
538     }
539 }
540
541 /// <remarks>
542 /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том случае, если
543 /// ↪ адрес реально поменялся
544 ///
545 /// Указатель this.links может быть в том же месте,
546 /// так как 0-я связь не используется и имеет такой же размер как Header,
547 /// поэтому header размещается в том же месте, что и 0-я связь
548 /// </remarks>
549 private void SetPointers(IDirectMemory memory)
550 {
551     if (memory == null)
552     {
553         _links = IntPtr.Zero;
554         _header = _links;
555         _unusedLinksListMethods = null;
556         _targetsTreeMethods = null;
557         _unusedLinksListMethods = null;
558     }
559     else
560     {
561         _links = memory.Pointer;
562         _header = _links;
563         _sourcesTreeMethods = new LinksSourcesTreeMethods(this);
564         _targetsTreeMethods = new LinksTargetsTreeMethods(this);
565         _unusedLinksListMethods = new UnusedLinksListMethods(_links, _header);
566     }
567 }
568
569 [MethodImpl(MethodImplOptions.AggressiveInlining)]
570 private bool Exists(TLink link)
571 => (_comparer.Compare(link, Constants.MinPossibleIndex) >= 0)
572     && (_comparer.Compare(link, LinksHeader.GetAllocatedLinks(_header)) <= 0)
573     && !IsUnusedLink(link);
574
575 [MethodImpl(MethodImplOptions.AggressiveInlining)]
576 private bool IsUnusedLink(TLink link)
577 => _equalityComparer.Equals(LinksHeader.GetFirstFreeLink(_header), link)
578     || (_equalityComparer.Equals(Link.GetSizeAsSource(GetLinkUnsafe(link)),
579         ↪ Constants.Null)
580         && !_equalityComparer.Equals(Link.GetSource(GetLinkUnsafe(link)), Constants.Null));
581
582 #region DisposableBase
583
584 protected override bool AllowMultipleDisposeCalls => true;
585
586 protected override void Dispose(bool manual, bool wasDisposed)
587 {

```

```

581         if (!wasDisposed)
582         {
583             SetPointers(null);
584             _memory.DisposeIfPossible();
585         }
586     }
587
588     #endregion
589 }
590 }

```

./Platform.Data.Doublets/ResizableDirectMemory/ResizableDirectMemoryLinks.ListMethods.cs

```

1  using System;
2  using Platform.Unsafe;
3  using Platform.Collections.Methods.Lists;
4
5  namespace Platform.Data.Doublets.ResizableDirectMemory
6  {
7      partial class ResizableDirectMemoryLinks<TLink>
8      {
9          private class UnusedLinksListMethods : CircularDoublyLinkedListMethods<TLink>
10         {
11             private readonly IntPtr _links;
12             private readonly IntPtr _header;
13
14             public UnusedLinksListMethods(IntPtr links, IntPtr header)
15             {
16                 _links = links;
17                 _header = header;
18             }
19
20             protected override TLink GetFirst() => (_header +
21                 ↳ LinksHeader.FirstFreeLinkOffset).GetValue<TLink>();
22
23             protected override TLink GetLast() => (_header +
24                 ↳ LinksHeader.LastFreeLinkOffset).GetValue<TLink>();
25
26             protected override TLink GetPrevious(TLink element) =>
27                 ↳ (_links.GetElement(LinkSizeInBytes, element) +
28                 ↳ Link.SourceOffset).GetValue<TLink>();
29
30             protected override TLink GetNext(TLink element) =>
31                 ↳ (_links.GetElement(LinkSizeInBytes, element) +
32                 ↳ Link.TargetOffset).GetValue<TLink>();
33
34             protected override TLink GetSize() => (_header +
35                 ↳ LinksHeader.FreeLinksOffset).GetValue<TLink>();
36
37             protected override void SetFirst(TLink element) => (_header +
38                 ↳ LinksHeader.FirstFreeLinkOffset).SetValue(element);
39
40             protected override void SetLast(TLink element) => (_header +
41                 ↳ LinksHeader.LastFreeLinkOffset).SetValue(element);
42
43             protected override void SetPrevious(TLink element, TLink previous) =>
44                 ↳ (_links.GetElement(LinkSizeInBytes, element) +
45                 ↳ Link.SourceOffset).SetValue(previous);
46
47             protected override void SetNext(TLink element, TLink next) =>
48                 ↳ (_links.GetElement(LinkSizeInBytes, element) + Link.TargetOffset).SetValue(next);
49
50             protected override void SetSize(TLink size) => (_header +
51                 ↳ LinksHeader.FreeLinksOffset).SetValue(size);
52         }
53     }
54 }

```

./Platform.Data.Doublets/ResizableDirectMemory/ResizableDirectMemoryLinks.TreeMethods.cs

```

1  using System;
2  using System.Text;
3  using System.Collections.Generic;
4  using System.Runtime.CompilerServices;
5  using Platform.Numbers;
6  using Platform.Unsafe;
7  using Platform.Collections.Methods.Trees;
8  using Platform.Data.Constants;
9
10 namespace Platform.Data.Doublets.ResizableDirectMemory
11 {

```

```

12 partial class ResizableDirectMemoryLinks<TLink>
13 {
14     private abstract class LinksTreeMethodsBase :
15         ↳ SizedAndThreadedAVLBalancedTreeMethods<TLink>
16     {
17         private readonly ResizableDirectMemoryLinks<TLink> _memory;
18         private readonly LinksCombinedConstants<TLink, TLink, int> _constants;
19         protected readonly IntPtr Links;
20         protected readonly IntPtr Header;
21
22         protected LinksTreeMethodsBase(ResizableDirectMemoryLinks<TLink> memory)
23         {
24             Links = memory._links;
25             Header = memory._header;
26             _memory = memory;
27             _constants = memory.Constants;
28         }
29
30         [MethodImpl(MethodImplOptions.AggressiveInlining)]
31         protected abstract TLink GetTreeRoot();
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34         protected abstract TLink GetBasePartValue(TLink link);
35
36         public TLink this[TLink index]
37         {
38             get
39             {
40                 var root = GetTreeRoot();
41                 if (GreaterOrEqualThan(index, GetSize(root)))
42                 {
43                     return GetZero();
44                 }
45                 while (!EqualToZero(root))
46                 {
47                     var left = GetLeftOrDefault(root);
48                     var leftSize = GetSizeOrZero(left);
49                     if (LessThan(index, leftSize))
50                     {
51                         root = left;
52                         continue;
53                     }
54                     if (IsEquals(index, leftSize))
55                     {
56                         return root;
57                     }
58                     root = GetRightOrDefault(root);
59                     index = Subtract(index, Increment(leftSize));
60                 }
61                 return GetZero(); // TODO: Impossible situation exception (only if tree
62                                 ↳ structure broken)
63             }
64         }
65
66         // TODO: Return indices range instead of references count
67         public TLink CountUsages(TLink link)
68         {
69             var root = GetTreeRoot();
70             var total = GetSize(root);
71             var totalRightIgnore = GetZero();
72             while (!EqualToZero(root))
73             {
74                 var @base = GetBasePartValue(root);
75                 if (LessOrEqualThan(@base, link))
76                 {
77                     root = GetRightOrDefault(root);
78                 }
79                 else
80                 {
81                     totalRightIgnore = Add(totalRightIgnore, Increment(GetRightSize(root)));
82                     root = GetLeftOrDefault(root);
83                 }
84             }
85             root = GetTreeRoot();
86             var totalLeftIgnore = GetZero();
87             while (!EqualToZero(root))
88             {
89                 var @base = GetBasePartValue(root);
90                 if (GreaterOrEqualThan(@base, link))

```

```

89         {
90             root = GetLeftOrDefault(root);
91         }
92         else
93         {
94             totalLeftIgnore = Add(totalLeftIgnore, Increment(GetLeftSize(root)));
95
96             root = GetRightOrDefault(root);
97         }
98     }
99     return Subtract(Subtract(total, totalRightIgnore), totalLeftIgnore);
100 }
101
102 public TLink EachUsage(TLink link, Func<IList<TLink>, TLink> handler)
103 {
104     var root = GetTreeRoot();
105     if (EqualToZero(root))
106     {
107         return _constants.Continue;
108     }
109     TLink first = GetZero(), current = root;
110     while (!EqualToZero(current))
111     {
112         var @base = GetBasePartValue(current);
113         if (GreaterOrEqualThan(@base, link))
114         {
115             if (IsEquals(@base, link))
116             {
117                 first = current;
118             }
119             current = GetLeftOrDefault(current);
120         }
121         else
122         {
123             current = GetRightOrDefault(current);
124         }
125     }
126     if (!EqualToZero(first))
127     {
128         current = first;
129         while (true)
130         {
131             if (IsEquals(handler(_memory.GetLinkStruct(current)), _constants.Break))
132             {
133                 return _constants.Break;
134             }
135             current = GetNext(current);
136             if (EqualToZero(current) || !IsEquals(GetBasePartValue(current), link))
137             {
138                 break;
139             }
140         }
141     }
142     return _constants.Continue;
143 }
144
145 protected override void PrintNodeValue(TLink node, StringBuilder sb)
146 {
147     sb.Append(' ');
148     sb.Append((Links.GetElement(LinkSizeInBytes, node) +
149         ↪ Link.SourceOffset).GetValue<TLink>());
149     sb.Append('-');
150     sb.Append('>');
151     sb.Append((Links.GetElement(LinkSizeInBytes, node) +
152         ↪ Link.TargetOffset).GetValue<TLink>());
153 }
154
155 private class LinksSourcesTreeMethods : LinksTreeMethodsBase
156 {
157     public LinksSourcesTreeMethods(ResizableDirectMemoryLinks<TLink> memory)
158         : base(memory)
159     {
160     }
161
162     protected override IntPtr GetLeftPointer(TLink node) =>
163         ↪ Links.GetElement(LinkSizeInBytes, node) + Link.LeftAsSourceOffset;

```

```

164     protected override IntPtr GetRightPointer(TLink node) =>
165         ↳ Links.GetElement(LinkSizeInBytes, node) + Link.RightAsSourceOffset;
166
167     protected override TLink GetLeftValue(TLink node) =>
168         ↳ (Links.GetElement(LinkSizeInBytes, node) +
169         ↳ Link.LeftAsSourceOffset).GetValue<TLink>();
170
171     protected override TLink GetRightValue(TLink node) =>
172         ↳ (Links.GetElement(LinkSizeInBytes, node) +
173         ↳ Link.RightAsSourceOffset).GetValue<TLink>();
174
175     protected override TLink GetSize(TLink node)
176     {
177         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
178         ↳ Link.SizeAsSourceOffset).GetValue<TLink>();
179         return Bit.PartialRead(previousValue, 5, -5);
180     }
181
182     protected override void SetLeft(TLink node, TLink left) =>
183         ↳ (Links.GetElement(LinkSizeInBytes, node) +
184         ↳ Link.LeftAsSourceOffset).SetValue(left);
185
186     protected override void SetRight(TLink node, TLink right) =>
187         ↳ (Links.GetElement(LinkSizeInBytes, node) +
188         ↳ Link.RightAsSourceOffset).SetValue(right);
189
190     protected override void SetSize(TLink node, TLink size)
191     {
192         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
193         ↳ Link.SizeAsSourceOffset).GetValue<TLink>();
194         (Links.GetElement(LinkSizeInBytes, node) +
195         ↳ Link.SizeAsSourceOffset).SetValue(Bit.PartialWrite(previousValue, size, 5,
196         ↳ -5));
197     }
198
199     protected override bool GetLeftIsChild(TLink node)
200     {
201         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
202         ↳ Link.SizeAsSourceOffset).GetValue<TLink>();
203         return (Integer<TLink>)Bit.PartialRead(previousValue, 4, 1);
204     }
205
206     protected override void SetLeftIsChild(TLink node, bool value)
207     {
208         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
209         ↳ Link.SizeAsSourceOffset).GetValue<TLink>();
210         var modified = Bit.PartialWrite(previousValue, (TLink)(Integer<TLink>)value, 4,
211         ↳ 1);
212         (Links.GetElement(LinkSizeInBytes, node) +
213         ↳ Link.SizeAsSourceOffset).SetValue(modified);
214     }
215
216     protected override bool GetRightIsChild(TLink node)
217     {
218         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
219         ↳ Link.SizeAsSourceOffset).GetValue<TLink>();
220         return (Integer<TLink>)Bit.PartialRead(previousValue, 3, 1);
221     }
222
223     protected override void SetRightIsChild(TLink node, bool value)
224     {
225         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
226         ↳ Link.SizeAsSourceOffset).GetValue<TLink>();
227         var modified = Bit.PartialWrite(previousValue, (TLink)(Integer<TLink>)value, 3,
228         ↳ 1);
229         (Links.GetElement(LinkSizeInBytes, node) +
230         ↳ Link.SizeAsSourceOffset).SetValue(modified);
231     }
232
233     protected override sbyte GetBalance(TLink node)
234     {
235         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
236         ↳ Link.SizeAsSourceOffset).GetValue<TLink>();
237         var value = (ulong)(Integer<TLink>)Bit.PartialRead(previousValue, 0, 3);
238         var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) << 5) | value & 3 |
239         ↳ 124 : value & 3);
240         return unpackedValue;
241     }

```



```

218 }
219
220 protected override void SetBalance(TLink node, sbyte value)
221 {
222     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
223         ↪ Link.SizeAsSourceOffset).GetValue<TLink>();
224     var packagedValue = (TLink)(Integer<TLink>)((((byte)value >> 5) & 4) | value &
225         ↪ 3);
226     var modified = Bit.PartialWrite(previousValue, packagedValue, 0, 3);
227     (Links.GetElement(LinkSizeInBytes, node) +
228         ↪ Link.SizeAsSourceOffset).SetValue(modified);
229 }
230
231 protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
232 {
233     var firstSource = (Links.GetElement(LinkSizeInBytes, first) +
234         ↪ Link.SourceOffset).GetValue<TLink>();
235     var secondSource = (Links.GetElement(LinkSizeInBytes, second) +
236         ↪ Link.SourceOffset).GetValue<TLink>();
237     return LessThan(firstSource, secondSource) ||
238         (IsEquals(firstSource, secondSource) &&
239         ↪ LessThan((Links.GetElement(LinkSizeInBytes, first) +
240             ↪ Link.TargetOffset).GetValue<TLink>(),
241             ↪ (Links.GetElement(LinkSizeInBytes, second) +
242             ↪ Link.TargetOffset).GetValue<TLink>()));
243 }
244
245 protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
246 {
247     var firstSource = (Links.GetElement(LinkSizeInBytes, first) +
248         ↪ Link.SourceOffset).GetValue<TLink>();
249     var secondSource = (Links.GetElement(LinkSizeInBytes, second) +
250         ↪ Link.SourceOffset).GetValue<TLink>();
251     return GreaterThan(firstSource, secondSource) ||
252         (IsEquals(firstSource, secondSource) &&
253         ↪ GreaterThan((Links.GetElement(LinkSizeInBytes, first) +
254             ↪ Link.TargetOffset).GetValue<TLink>(),
255             ↪ (Links.GetElement(LinkSizeInBytes, second) +
256             ↪ Link.TargetOffset).GetValue<TLink>()));
257 }
258
259 protected override TLink GetTreeRoot() => (Header +
260     ↪ LinksHeader.FirstAsSourceOffset).GetValue<TLink>();
261
262 protected override TLink GetBasePartValue(TLink link) =>
263     ↪ (Links.GetElement(LinkSizeInBytes, link) + Link.SourceOffset).GetValue<TLink>();
264
265 /// <summary>
266 /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
267 ↪ (концом)
268 /// по дереву (индексу) связей, отсортированному по Source, а затем по Target.
269 /// </summary>
270 /// <param name="source">Индекс связи, которая является началом на искомой
271 ↪ связи.</param>
272 /// <param name="target">Индекс связи, которая является концом на искомой
273 ↪ связи.</param>
274 /// <returns>Индекс искомой связи.</returns>
275 public TLink Search(TLink source, TLink target)
276 {
277     var root = GetTreeRoot();
278     while (!EqualToZero(root))
279     {
280         var rootSource = (Links.GetElement(LinkSizeInBytes, root) +
281             ↪ Link.SourceOffset).GetValue<TLink>();
282         var rootTarget = (Links.GetElement(LinkSizeInBytes, root) +
283             ↪ Link.TargetOffset).GetValue<TLink>();
284         if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
285             ↪ node.Key < root.Key
286         {
287             root = GetLeftOrDefault(root);
288         }
289         else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget))
290             ↪ // node.Key > root.Key
291         {
292             root = GetRightOrDefault(root);
293         }
294     }
295 }

```

```

270         else // node.Key == root.Key
271         {
272             return root;
273         }
274     }
275     return GetZero();
276 }
277
278 [MethodImpl(MethodImplOptions.AggressiveInlining)]
279 private bool FirstIsToLeftOfSecond(TLink firstSource, TLink firstTarget, TLink
    ↳ secondSource, TLink secondTarget) => LessThan(firstSource, secondSource) ||
    ↳ (IsEquals(firstSource, secondSource) && LessThan(firstTarget, secondTarget));
280
281 [MethodImpl(MethodImplOptions.AggressiveInlining)]
282 private bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget, TLink
    ↳ secondSource, TLink secondTarget) => GreaterThan(firstSource, secondSource) ||
    ↳ (IsEquals(firstSource, secondSource) && GreaterThan(firstTarget, secondTarget));
283 }
284
285 private class LinksTargetsTreeMethods : LinksTreeMethodsBase
286 {
287     public LinksTargetsTreeMethods(ResizableDirectMemoryLinks<TLink> memory)
288         : base(memory)
289     {
290     }
291
292     protected override IntPtr GetLeftPointer(TLink node) =>
293         ↳ Links.GetElement(LinkSizeInBytes, node) + Link.LeftAsTargetOffset;
294
295     protected override IntPtr GetRightPointer(TLink node) =>
296         ↳ Links.GetElement(LinkSizeInBytes, node) + Link.RightAsTargetOffset;
297
298     protected override TLink GetLeftValue(TLink node) =>
299         ↳ (Links.GetElement(LinkSizeInBytes, node) +
300         ↳ Link.LeftAsTargetOffset).GetValue<TLink>();
301
302     protected override TLink GetRightValue(TLink node) =>
303         ↳ (Links.GetElement(LinkSizeInBytes, node) +
304         ↳ Link.RightAsTargetOffset).GetValue<TLink>();
305
306     protected override TLink GetSize(TLink node)
307     {
308         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
309         ↳ Link.SizeAsTargetOffset).GetValue<TLink>();
310         return Bit.PartialRead(previousValue, 5, -5);
311     }
312
313     protected override void SetLeft(TLink node, TLink left) =>
314         ↳ (Links.GetElement(LinkSizeInBytes, node) +
315         ↳ Link.LeftAsTargetOffset).SetValue(left);
316
317     protected override void SetRight(TLink node, TLink right) =>
318         ↳ (Links.GetElement(LinkSizeInBytes, node) +
319         ↳ Link.RightAsTargetOffset).SetValue(right);
320
321     protected override void SetSize(TLink node, TLink size)
322     {
323         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
324         ↳ Link.SizeAsTargetOffset).GetValue<TLink>();
325         (Links.GetElement(LinkSizeInBytes, node) +
326         ↳ Link.SizeAsTargetOffset).SetValue(Bit.PartialWrite(previousValue, size, 5,
327         ↳ -5));
328     }
329
330     protected override bool GetLeftIsChild(TLink node)
331     {
332         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
333         ↳ Link.SizeAsTargetOffset).GetValue<TLink>();
334         return (Integer<TLink>)Bit.PartialRead(previousValue, 4, 1);
335     }
336
337     protected override void SetLeftIsChild(TLink node, bool value)
338     {
339         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
340         ↳ Link.SizeAsTargetOffset).GetValue<TLink>();
341         var modified = Bit.PartialWrite(previousValue, (TLink)(Integer<TLink>)value, 4,
342         ↳ 1);

```

```

326         (Links.GetElement(LinkSizeInBytes, node) +
327         ↪ Link.SizeAsTargetOffset).SetValue(modified);
328     }
329     protected override bool GetRightIsChild(TLink node)
330     {
331         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
332         ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
333         return (Integer<TLink>)Bit.PartialRead(previousValue, 3, 1);
334     }
335     protected override void SetRightIsChild(TLink node, bool value)
336     {
337         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
338         ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
339         var modified = Bit.PartialWrite(previousValue, (TLink)(Integer<TLink>)value, 3,
340         ↪ 1);
341         (Links.GetElement(LinkSizeInBytes, node) +
342         ↪ Link.SizeAsTargetOffset).SetValue(modified);
343     }
344     protected override sbyte GetBalance(TLink node)
345     {
346         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
347         ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
348         var value = (ulong)(Integer<TLink>)Bit.PartialRead(previousValue, 0, 3);
349         var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) << 5) | value & 3 |
350         ↪ 124 : value & 3);
351         return unpackedValue;
352     }
353     protected override void SetBalance(TLink node, sbyte value)
354     {
355         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
356         ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
357         var packagedValue = (TLink)(Integer<TLink>)(((byte)value >> 5) & 4) | value &
358         ↪ 3);
359         var modified = Bit.PartialWrite(previousValue, packagedValue, 0, 3);
360         (Links.GetElement(LinkSizeInBytes, node) +
361         ↪ Link.SizeAsTargetOffset).SetValue(modified);
362     }
363     protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
364     {
365         var firstTarget = (Links.GetElement(LinkSizeInBytes, first) +
366         ↪ Link.TargetOffset).GetValue<TLink>();
367         var secondTarget = (Links.GetElement(LinkSizeInBytes, second) +
368         ↪ Link.TargetOffset).GetValue<TLink>();
369         return LessThan(firstTarget, secondTarget) ||
370         ↪ (IsEquals(firstTarget, secondTarget) &&
371         ↪ LessThan((Links.GetElement(LinkSizeInBytes, first) +
372         ↪ Link.SourceOffset).GetValue<TLink>(),
373         ↪ (Links.GetElement(LinkSizeInBytes, second) +
374         ↪ Link.SourceOffset).GetValue<TLink>()));
375     }
376     protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
377     {
378         var firstTarget = (Links.GetElement(LinkSizeInBytes, first) +
379         ↪ Link.TargetOffset).GetValue<TLink>();
380         var secondTarget = (Links.GetElement(LinkSizeInBytes, second) +
381         ↪ Link.TargetOffset).GetValue<TLink>();
382         return GreaterThan(firstTarget, secondTarget) ||
383         ↪ (IsEquals(firstTarget, secondTarget) &&
384         ↪ GreaterThan((Links.GetElement(LinkSizeInBytes, first) +
385         ↪ Link.SourceOffset).GetValue<TLink>(),
386         ↪ (Links.GetElement(LinkSizeInBytes, second) +
387         ↪ Link.SourceOffset).GetValue<TLink>()));
388     }
389     protected override TLink GetTreeRoot() => (Header +
390     ↪ LinksHeader.FirstAsTargetOffset).GetValue<TLink>();
391     protected override TLink GetBasePartValue(TLink link) =>
392     ↪ (Links.GetElement(LinkSizeInBytes, link) + Link.TargetOffset).GetValue<TLink>();
393 }

```

./Platform.Data.Doublets/ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Disposables;
5  using Platform.Collections.Arrays;
6  using Platform.Singletons;
7  using Platform.Memory;
8  using Platform.Data.Exceptions;
9  using Platform.Data.Constants;
10
11  // #define ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
12
13  #pragma warning disable 0649
14  #pragma warning disable 169
15
16  // ReSharper disable BuiltInTypeReferenceStyle
17
18  namespace Platform.Data.Doublets.ResizableDirectMemory
19  {
20      using id = UInt64;
21
22      public unsafe partial class UInt64ResizableDirectMemoryLinks : DisposableBase, ILinks<id>
23      {
24          /// <summary>Возвращает размер одной связи в байтах.</summary>
25          /// <remarks>
26          ///     Используется только во вне класса, не рекомендуется использовать внутри.
27          ///     Так как во вне не обязательно будет доступен unsafe C#.
28          /// </remarks>
29          public static readonly int LinkSizeInBytes = sizeof(Link);
30
31          public static readonly long DefaultLinksSizeStep = LinkSizeInBytes * 1024 * 1024;
32
33          private struct Link
34          {
35              public id Source;
36              public id Target;
37              public id LeftAsSource;
38              public id RightAsSource;
39              public id SizeAsSource;
40              public id LeftAsTarget;
41              public id RightAsTarget;
42              public id SizeAsTarget;
43          }
44
45          private struct LinksHeader
46          {
47              public id AllocatedLinks;
48              public id ReservedLinks;
49              public id FreeLinks;
50              public id FirstFreeLink;
51              public id FirstAsSource;
52              public id FirstAsTarget;
53              public id LastFreeLink;
54              public id Reserved8;
55          }
56
57          private readonly long _memoryReservationStep;
58
59          private readonly IResizableDirectMemory _memory;
60          private LinksHeader* _header;
61          private Link* _links;
62
63          private LinksTargetsTreeMethods _targetsTreeMethods;
64          private LinksSourcesTreeMethods _sourcesTreeMethods;
65
66          // TODO: Возможно чтобы гарантированно проверять на то, является ли связь удалённой,
67          //      ↪ нужно использовать не список а дерево, так как так можно быстрее проверить на
68          //      ↪ наличие связи внутри
69          private UnusedLinksListMethods _unusedLinksListMethods;
70
71          /// <summary>
72          ///     Возвращает общее число связей находящихся в хранилище.
73          /// </summary>
74          private id Total => _header->AllocatedLinks - _header->FreeLinks;
75
76          // TODO: Дать возможность переопределять в конструкторе
77          public LinksCombinedConstants<id, id, int> Constants { get; }
78
79          public UInt64ResizableDirectMemoryLinks(string address) : this(address,
80              ↪ DefaultLinksSizeStep) { }

```

```

78
79 /// <summary>
80 /// Создаёт экземпляры базы данных Links в файле по указанному адресу, с указанным
81   ↳ минимальным шагом расширения базы данных.
82 /// </summary>
83 /// <param name="address">Полный путь к файлу базы данных.</param>
84 /// <param name="memoryReservationStep">Минимальный шаг расширения базы данных в
85   ↳ байтах.</param>
86 public UInt64ResizableDirectMemoryLinks(string address, long memoryReservationStep) :
87   ↳ this(new FileMappedResizableDirectMemory(address, memoryReservationStep),
88   ↳ memoryReservationStep) { }
89
90 public UInt64ResizableDirectMemoryLinks(IResizableDirectMemory memory) : this(memory,
91   ↳ DefaultLinksSizeStep) { }
92
93 public UInt64ResizableDirectMemoryLinks(IResizableDirectMemory memory, long
94   ↳ memoryReservationStep)
95 {
96     Constants = Default<LinksCombinedConstants<id, id, int>>.Instance;
97     _memory = memory;
98     _memoryReservationStep = memoryReservationStep;
99     if (memory.ReservedCapacity < memoryReservationStep)
100     {
101         memory.ReservedCapacity = memoryReservationStep;
102     }
103     SetPointers(_memory);
104     // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
105     _memory.UsedCapacity = ((long)_header->AllocatedLinks * sizeof(Link)) +
106   ↳ sizeof(LinksHeader);
107     // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
108     _header->ReservedLinks = (id)((_memory.ReservedCapacity - sizeof(LinksHeader)) /
109   ↳ sizeof(Link));
110 }
111
112 [MethodImpl(MethodImplOptions.AggressiveInlining)]
113 public id Count(IList<id> restrictions)
114 {
115     // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
116     if (restrictions.Count == 0)
117     {
118         return Total;
119     }
120     if (restrictions.Count == 1)
121     {
122         var index = restrictions[Constants.IndexPart];
123         if (index == Constants.Any)
124         {
125             return Total;
126         }
127         return Exists(index) ? 1UL : 0UL;
128     }
129     if (restrictions.Count == 2)
130     {
131         var index = restrictions[Constants.IndexPart];
132         var value = restrictions[1];
133         if (index == Constants.Any)
134         {
135             if (value == Constants.Any)
136             {
137                 return Total; // Any - как отсутствие ограничения
138             }
139             return _sourcesTreeMethods.CountUsages(value)
140                 + _targetsTreeMethods.CountUsages(value);
141         }
142         else
143         {
144             if (!Exists(index))
145             {
146                 return 0;
147             }
148             if (value == Constants.Any)
149             {
150                 return 1;
151             }
152             var storedLinkValue = GetLinkUnsafe(index);
153             if (storedLinkValue->Source == value ||
154                 storedLinkValue->Target == value)
155             {

```

```

148         return 1;
149     }
150     return 0;
151 }
152 }
153 if (restrictions.Count == 3)
154 {
155     var index = restrictions[Constants.IndexPart];
156     var source = restrictions[Constants.SourcePart];
157     var target = restrictions[Constants.TargetPart];
158     if (index == Constants.Any)
159     {
160         if (source == Constants.Any && target == Constants.Any)
161         {
162             return Total;
163         }
164         else if (source == Constants.Any)
165         {
166             return _targetsTreeMethods.CountUsages(target);
167         }
168         else if (target == Constants.Any)
169         {
170             return _sourcesTreeMethods.CountUsages(source);
171         }
172         else //if(source != Any && target != Any)
173         {
174             // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
175             var link = _sourcesTreeMethods.Search(source, target);
176             return link == Constants.Null ? OUL : 1UL;
177         }
178     }
179     else
180     {
181         if (!Exists(index))
182         {
183             return 0;
184         }
185         if (source == Constants.Any && target == Constants.Any)
186         {
187             return 1;
188         }
189         var storedLinkValue = GetLinkUnsafe(index);
190         if (source != Constants.Any && target != Constants.Any)
191         {
192             if (storedLinkValue->Source == source &&
193                 storedLinkValue->Target == target)
194             {
195                 return 1;
196             }
197             return 0;
198         }
199         var value = default(id);
200         if (source == Constants.Any)
201         {
202             value = target;
203         }
204         if (target == Constants.Any)
205         {
206             value = source;
207         }
208         if (storedLinkValue->Source == value ||
209             storedLinkValue->Target == value)
210         {
211             return 1;
212         }
213         return 0;
214     }
215 }
216 throw new NotSupportedException("Другие размеры и способы ограничений не
    ↳ поддерживаются.");
217 }
218
219 [MethodImpl(MethodImplOptions.AggressiveInlining)]
220 public id Each(Func<IList<id>, id> handler, IList<id> restrictions)
221 {
222     if (restrictions.Count == 0)
223     {
224         for (id link = 1; link <= _header->AllocatedLinks; link++)
225         {

```

```

226         if (Exists(link))
227         {
228             if (handler(GetLinkStruct(link)) == Constants.Break)
229             {
230                 return Constants.Break;
231             }
232         }
233     }
234     return Constants.Continue;
235 }
236 if (restrictions.Count == 1)
237 {
238     var index = restrictions[Constants.IndexPart];
239     if (index == Constants.Any)
240     {
241         return Each(handler, ArrayPool<ulong>.Empty);
242     }
243     if (!Exists(index))
244     {
245         return Constants.Continue;
246     }
247     return handler(GetLinkStruct(index));
248 }
249 if (restrictions.Count == 2)
250 {
251     var index = restrictions[Constants.IndexPart];
252     var value = restrictions[1];
253     if (index == Constants.Any)
254     {
255         if (value == Constants.Any)
256         {
257             return Each(handler, ArrayPool<ulong>.Empty);
258         }
259         if (Each(handler, new[] { index, value, Constants.Any }) == Constants.Break)
260         {
261             return Constants.Break;
262         }
263         return Each(handler, new[] { index, Constants.Any, value });
264     }
265     else
266     {
267         if (!Exists(index))
268         {
269             return Constants.Continue;
270         }
271         if (value == Constants.Any)
272         {
273             return handler(GetLinkStruct(index));
274         }
275         var storedLinkValue = GetLinkUnsafe(index);
276         if (storedLinkValue->Source == value ||
277             storedLinkValue->Target == value)
278         {
279             return handler(GetLinkStruct(index));
280         }
281         return Constants.Continue;
282     }
283 }
284 if (restrictions.Count == 3)
285 {
286     var index = restrictions[Constants.IndexPart];
287     var source = restrictions[Constants.SourcePart];
288     var target = restrictions[Constants.TargetPart];
289     if (index == Constants.Any)
290     {
291         if (source == Constants.Any && target == Constants.Any)
292         {
293             return Each(handler, ArrayPool<ulong>.Empty);
294         }
295         else if (source == Constants.Any)
296         {
297             return _targetsTreeMethods.EachReference(target, handler);
298         }
299         else if (target == Constants.Any)
300         {
301             return _sourcesTreeMethods.EachReference(source, handler);
302         }
303         else //if(source != Any && target != Any)

```

```

304         {
305             var link = _sourcesTreeMethods.Search(source, target);
306             return link == Constants.Null ? Constants.Continue :
                 ↪ handler(GetLinkStruct(link));
307         }
308     }
309     else
310     {
311         if (!Exists(index))
312         {
313             return Constants.Continue;
314         }
315         if (source == Constants.Any && target == Constants.Any)
316         {
317             return handler(GetLinkStruct(index));
318         }
319         var storedLinkValue = GetLinkUnsafe(index);
320         if (source != Constants.Any && target != Constants.Any)
321         {
322             if (storedLinkValue->Source == source &&
323                 storedLinkValue->Target == target)
324             {
325                 return handler(GetLinkStruct(index));
326             }
327             return Constants.Continue;
328         }
329         var value = default(id);
330         if (source == Constants.Any)
331         {
332             value = target;
333         }
334         if (target == Constants.Any)
335         {
336             value = source;
337         }
338         if (storedLinkValue->Source == value ||
339             storedLinkValue->Target == value)
340         {
341             return handler(GetLinkStruct(index));
342         }
343         return Constants.Continue;
344     }
345 }
346 throw new NotSupportedException("Другие размеры и способы ограничений не
    ↪ поддерживаются.");
347 }
348
349 /// <remarks>
350 /// TODO: Возможно можно перемещать значения, если указан индекс, но значение существует
    ↪ в другом месте (но не в менеджере памяти, а в логике Links)
351 /// </remarks>
352 [MethodImpl(MethodImplOptions.AggressiveInlining)]
353 public id Update(IList<id> values)
354 {
355     var linkIndex = values[Constants.IndexPart];
356     var link = GetLinkUnsafe(linkIndex);
357     // Будет корректно работать только в том случае, если пространство выделенной связи
    ↪ предварительно заполнено нулями
358     if (link->Source != Constants.Null)
359     {
360         _sourcesTreeMethods.Detach(new IntPtr(&_header->FirstAsSource), linkIndex);
361     }
362     if (link->Target != Constants.Null)
363     {
364         _targetsTreeMethods.Detach(new IntPtr(&_header->FirstAsTarget), linkIndex);
365     }
366 #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
367     var leftTreeSize = _sourcesTreeMethods.GetSize(new IntPtr(&_header->FirstAsSource));
368     var rightTreeSize = _targetsTreeMethods.GetSize(new IntPtr(&_header->FirstAsTarget));
369     if (leftTreeSize != rightTreeSize)
370     {
371         throw new Exception("One of the trees is broken.");
372     }
373 #endif
374     link->Source = values[Constants.SourcePart];
375     link->Target = values[Constants.TargetPart];
376     if (link->Source != Constants.Null)
377     {

```



```

378         _sourcesTreeMethods.Attach(new IntPtr(&_header->FirstAsSource), linkIndex);
379     }
380     if (link->Target != Constants.Null)
381     {
382         _targetsTreeMethods.Attach(new IntPtr(&_header->FirstAsTarget), linkIndex);
383     }
384     #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
385     leftTreeSize = _sourcesTreeMethods.GetSize(new IntPtr(&_header->FirstAsSource));
386     rightTreeSize = _targetsTreeMethods.GetSize(new IntPtr(&_header->FirstAsTarget));
387     if (leftTreeSize != rightTreeSize)
388     {
389         throw new Exception("One of the trees is broken.");
390     }
391     #endif
392     return linkIndex;
393 }
394
395 [MethodImpl(MethodImplOptions.AggressiveInlining)]
396 private IList<id> GetLinkStruct(id linkIndex)
397 {
398     var link = GetLinkUnsafe(linkIndex);
399     return new UInt64Link(linkIndex, link->Source, link->Target);
400 }
401
402 [MethodImpl(MethodImplOptions.AggressiveInlining)]
403 private Link* GetLinkUnsafe(id linkIndex) => &_amp;links[linkIndex];
404
405 /// <remarks>
406 /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
407   ↳ пространство
408 /// </remarks>
409 public id Create()
410 {
411     var freeLink = _header->FirstFreeLink;
412     if (freeLink != Constants.Null)
413     {
414         _unusedLinksListMethods.Detach(freeLink);
415     }
416     else
417     {
418         if (_header->AllocatedLinks > Constants.MaxPossibleIndex)
419         {
420             throw new LinksLimitReachedException(Constants.MaxPossibleIndex);
421         }
422         if (_header->AllocatedLinks >= _header->ReservedLinks - 1)
423         {
424             _memory.ReservedCapacity += _memory.ReservationStep;
425             SetPointers(_memory);
426             _header->ReservedLinks = (id)(_memory.ReservedCapacity / sizeof(Link));
427         }
428         _header->AllocatedLinks++;
429         _memory.UsedCapacity += sizeof(Link);
430         freeLink = _header->AllocatedLinks;
431     }
432     return freeLink;
433 }
434
435 public void Delete(id link)
436 {
437     if (link < _header->AllocatedLinks)
438     {
439         _unusedLinksListMethods.AttachAsFirst(link);
440     }
441     else if (link == _header->AllocatedLinks)
442     {
443         _header->AllocatedLinks--;
444         _memory.UsedCapacity -= sizeof(Link);
445         // Убираем все связи, находящиеся в списке свободных в конце файла, до тех пор,
446         // пока не дойдём до первой существующей связи
447         // Позволяет оптимизировать количество выделенных связей (AllocatedLinks)
448         while (_header->AllocatedLinks > 0 && IsUnusedLink(_header->AllocatedLinks))
449         {
450             _unusedLinksListMethods.Detach(_header->AllocatedLinks);
451             _header->AllocatedLinks--;
452             _memory.UsedCapacity -= sizeof(Link);
453         }
454     }
455 }

```

```

455 /// <remarks>
456 /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том случае, если
    ↪ адрес реально поменялся
457 ///
458 /// Указатель this.links может быть в том же месте,
459 /// так как 0-я связь не используется и имеет такой же размер как Header,
460 /// поэтому header размещается в том же месте, что и 0-я связь
461 /// </remarks>
462 private void SetPointers(IResizableDirectMemory memory)
463 {
464     if (memory == null)
465     {
466         _header = null;
467         _links = null;
468         _unusedLinksListMethods = null;
469         _targetsTreeMethods = null;
470         _unusedLinksListMethods = null;
471     }
472     else
473     {
474         _header = (LinksHeader*)(void*)memory.Pointer;
475         _links = (Link*)(void*)memory.Pointer;
476         _sourcesTreeMethods = new LinksSourcesTreeMethods(this);
477         _targetsTreeMethods = new LinksTargetsTreeMethods(this);
478         _unusedLinksListMethods = new UnusedLinksListMethods(_links, _header);
479     }
480 }
481
482 [MethodImpl(MethodImplOptions.AggressiveInlining)]
483 private bool Exists(id link) => link >= Constants.MinPossibleIndex && link <=
    ↪ _header->AllocatedLinks && !IsUnusedLink(link);
484
485 [MethodImpl(MethodImplOptions.AggressiveInlining)]
486 private bool IsUnusedLink(id link) => _header->FirstFreeLink == link
    || (_links[link].SizeAsSource == Constants.Null &&
    ↪ _links[link].Source != Constants.Null);
487
488 #region Disposable
489
490 protected override bool AllowMultipleDisposeCalls => true;
491
492 protected override void Dispose(bool manual, bool wasDisposed)
493 {
494     if (!wasDisposed)
495     {
496         SetPointers(null);
497         _memory.DisposeIfPossible();
498     }
499 }
500
501 #endregion
502
503 }
504

```

./Platform.Data.Doublets/ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.ListMethods.cs

```

1 using Platform.Collections.Methods.Lists;
2
3 namespace Platform.Data.Doublets.ResizableDirectMemory
4 {
5     unsafe partial class UInt64ResizableDirectMemoryLinks
6     {
7         private class UnusedLinksListMethods : CircularDoublyLinkedListMethods<ulong>
8         {
9             private readonly Link* _links;
10             private readonly LinksHeader* _header;
11
12             public UnusedLinksListMethods(Link* links, LinksHeader* header)
13             {
14                 _links = links;
15                 _header = header;
16             }
17
18             protected override ulong GetFirst() => _header->FirstFreeLink;
19
20             protected override ulong GetLast() => _header->LastFreeLink;
21
22             protected override ulong GetPrevious(ulong element) => _links[element].Source;
23
24             protected override ulong GetNext(ulong element) => _links[element].Target;
25

```

```

26         protected override ulong GetSize() => _header->FreeLinks;
27
28         protected override void SetFirst(ulong element) => _header->FirstFreeLink = element;
29
30         protected override void SetLast(ulong element) => _header->LastFreeLink = element;
31
32         protected override void SetPrevious(ulong element, ulong previous) =>
33             ↪ _links[element].Source = previous;
34
35         protected override void SetNext(ulong element, ulong next) => _links[element].Target
36             ↪ = next;
37
38         protected override void SetSize(ulong size) => _header->FreeLinks = size;
39     }

```

./Platform.Data.Doublets/ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.TreeMethods.cs

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using System.Text;
5  using Platform.Collections.Methods.Trees;
6  using Platform.Data.Constants;
7
8  namespace Platform.Data.Doublets.ResizableDirectMemory
9  {
10     unsafe partial class UInt64ResizableDirectMemoryLinks
11     {
12         private abstract class LinksTreeMethodsBase :
13             ↪ SizedAndThreadedAVLBalancedTreeMethods<ulong>
14         {
15             private readonly UInt64ResizableDirectMemoryLinks _memory;
16             private readonly LinksCombinedConstants<ulong, ulong, int> _constants;
17             protected readonly Link* Links;
18             protected readonly LinksHeader* Header;
19
20             protected LinksTreeMethodsBase(UInt64ResizableDirectMemoryLinks memory)
21             {
22                 Links = memory._links;
23                 Header = memory._header;
24                 _memory = memory;
25                 _constants = memory.Constants;
26             }
27
28             [MethodImpl(MethodImplOptions.AggressiveInlining)]
29             protected abstract ulong GetTreeRoot();
30
31             [MethodImpl(MethodImplOptions.AggressiveInlining)]
32             protected abstract ulong GetBasePartValue(ulong link);
33
34             public ulong this[ulong index]
35             {
36                 get
37                 {
38                     var root = GetTreeRoot();
39                     if (index >= GetSize(root))
40                     {
41                         return 0;
42                     }
43                     while (root != 0)
44                     {
45                         var left = GetLeftOrDefault(root);
46                         var leftSize = GetSizeOrZero(left);
47                         if (index < leftSize)
48                         {
49                             root = left;
50                             continue;
51                         }
52                         if (index == leftSize)
53                         {
54                             return root;
55                         }
56                         root = GetRightOrDefault(root);
57                         index -= leftSize + 1;
58                     }
59                     return 0; // TODO: Impossible situation exception (only if tree structure
60                     ↪ broken)
61                 }
62             }
63     }

```

```

61 // TODO: Return indices range instead of references count
62 public ulong CountUsages(ulong link)
63 {
64     var root = GetTreeRoot();
65     var total = GetSize(root);
66     var totalRightIgnore = OUL;
67     while (root != 0)
68     {
69         var @base = GetBasePartValue(root);
70         if (@base <= link)
71         {
72             root = GetRightOrDefault(root);
73         }
74         else
75         {
76             totalRightIgnore += GetRightSize(root) + 1;
77             root = GetLeftOrDefault(root);
78         }
79     }
80     root = GetTreeRoot();
81     var totalLeftIgnore = OUL;
82     while (root != 0)
83     {
84         var @base = GetBasePartValue(root);
85         if (@base >= link)
86         {
87             root = GetLeftOrDefault(root);
88         }
89         else
90         {
91             totalLeftIgnore += GetLeftSize(root) + 1;
92             root = GetRightOrDefault(root);
93         }
94     }
95     return total - totalRightIgnore - totalLeftIgnore;
96 }
97
98 public ulong EachReference(ulong link, Func<IList<ulong>, ulong> handler)
99 {
100     var root = GetTreeRoot();
101     if (root == 0)
102     {
103         return _constants.Continue;
104     }
105     ulong first = 0, current = root;
106     while (current != 0)
107     {
108         var @base = GetBasePartValue(current);
109         if (@base >= link)
110         {
111             if (@base == link)
112             {
113                 first = current;
114             }
115             current = GetLeftOrDefault(current);
116         }
117         else
118         {
119             current = GetRightOrDefault(current);
120         }
121     }
122     if (first != 0)
123     {
124         current = first;
125         while (true)
126         {
127             if (handler(_memory.GetLinkStruct(current)) == _constants.Break)
128             {
129                 return _constants.Break;
130             }
131             current = GetNext(current);
132             if (current == 0 || GetBasePartValue(current) != link)
133             {
134                 break;
135             }
136         }
137     }
138     return _constants.Continue;
139 }

```

```

140     }
141
142     protected override void PrintNodeValue(ulong node, StringBuilder sb)
143     {
144         sb.Append(' ');
145         sb.Append(Links[node].Source);
146         sb.Append('-');
147         sb.Append('>');
148         sb.Append(Links[node].Target);
149     }
150 }
151
152 private class LinksSourcesTreeMethods : LinksTreeMethodsBase
153 {
154     public LinksSourcesTreeMethods(UInt64ResizableDirectMemoryLinks memory)
155         : base(memory)
156     {
157     }
158
159     protected override IntPtr GetLeftPointer(ulong node) => new
160         ↳ IntPtr(&Links[node].LeftAsSource);
161
162     protected override IntPtr GetRightPointer(ulong node) => new
163         ↳ IntPtr(&Links[node].RightAsSource);
164
165     protected override ulong GetLeftValue(ulong node) => Links[node].LeftAsSource;
166
167     protected override ulong GetRightValue(ulong node) => Links[node].RightAsSource;
168
169     protected override ulong GetSize(ulong node)
170     {
171         var previousValue = Links[node].SizeAsSource;
172         //return Math.PartialRead(previousValue, 5, -5);
173         return (previousValue & 4294967264) >> 5;
174     }
175
176     protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsSource
177         ↳ = left;
178
179     protected override void SetRight(ulong node, ulong right) =>
180         ↳ Links[node].RightAsSource = right;
181
182     protected override void SetSize(ulong node, ulong size)
183     {
184         var previousValue = Links[node].SizeAsSource;
185         //var modified = Math.PartialWrite(previousValue, size, 5, -5);
186         var modified = (previousValue & 31) | ((size & 134217727) << 5);
187         Links[node].SizeAsSource = modified;
188     }
189
190     protected override bool GetLeftIsChild(ulong node)
191     {
192         var previousValue = Links[node].SizeAsSource;
193         //return (Integer)Math.PartialRead(previousValue, 4, 1);
194         return (previousValue & 16) >> 4 == 1UL;
195     }
196
197     protected override void SetLeftIsChild(ulong node, bool value)
198     {
199         var previousValue = Links[node].SizeAsSource;
200         //var modified = Math.PartialWrite(previousValue, (ulong)(Integer)value, 4, 1);
201         var modified = (previousValue & 4294967279) | ((value ? 1UL : 0UL) << 4);
202         Links[node].SizeAsSource = modified;
203     }
204
205     protected override bool GetRightIsChild(ulong node)
206     {
207         var previousValue = Links[node].SizeAsSource;
208         //return (Integer)Math.PartialRead(previousValue, 3, 1);
209         return (previousValue & 8) >> 3 == 1UL;
210     }
211
212     protected override void SetRightIsChild(ulong node, bool value)
213     {
214         var previousValue = Links[node].SizeAsSource;
215         //var modified = Math.PartialWrite(previousValue, (ulong)(Integer)value, 3, 1);
216         var modified = (previousValue & 4294967287) | ((value ? 1UL : 0UL) << 3);
217         Links[node].SizeAsSource = modified;
218     }
219 }

```

```

215
216 protected override sbyte GetBalance(ulong node)
217 {
218     var previousValue = Links[node].SizeAsSource;
219     //var value = Math.PartialRead(previousValue, 0, 3);
220     var value = previousValue & 7;
221     var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) << 5) | value & 3 |
222     ↪ 124 : value & 3);
223     return unpackedValue;
224 }
225
226 protected override void SetBalance(ulong node, sbyte value)
227 {
228     var previousValue = Links[node].SizeAsSource;
229     var packagedValue = (ulong)((byte)value >> 5) & 4 | value & 3;
230     //var modified = Math.PartialWrite(previousValue, packagedValue, 0, 3);
231     var modified = (previousValue & 4294967288) | (packagedValue & 7);
232     Links[node].SizeAsSource = modified;
233 }
234
235 protected override bool FirstIsToLeftOfSecond(ulong first, ulong second)
236     => Links[first].Source < Links[second].Source ||
237     (Links[first].Source == Links[second].Source && Links[first].Target <
238     ↪ Links[second].Target);
239
240 protected override bool FirstIsToTheRightOfSecond(ulong first, ulong second)
241     => Links[first].Source > Links[second].Source ||
242     (Links[first].Source == Links[second].Source && Links[first].Target >
243     ↪ Links[second].Target);
244
245 protected override ulong GetTreeRoot() => Header->FirstAsSource;
246
247 protected override ulong GetBasePartValue(ulong link) => Links[link].Source;
248
249 /// <summary>
250 /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
251 ↪ (концом)
252 /// по дереву (индексу) связей, отсортированному по Source, а затем по Target.
253 /// </summary>
254 /// <param name="source">Индекс связи, которая является началом на искомой
255 ↪ связи.</param>
256 /// <param name="target">Индекс связи, которая является концом на искомой
257 ↪ связи.</param>
258 /// <returns>Индекс искомой связи.</returns>
259 public ulong Search(ulong source, ulong target)
260 {
261     var root = Header->FirstAsSource;
262     while (root != 0)
263     {
264         var rootSource = Links[root].Source;
265         var rootTarget = Links[root].Target;
266         if (FirstIsToLeftOfSecond(source, target, rootSource, rootTarget)) //
267             ↪ node.Key < root.Key
268         {
269             root = GetLeftOrDefault(root);
270         }
271         else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget))
272             ↪ // node.Key > root.Key
273         {
274             root = GetRightOrDefault(root);
275         }
276         else // node.Key == root.Key
277         {
278             return root;
279         }
280     }
281     return 0;
282 }
283
284 [MethodImpl(MethodImplOptions.AggressiveInlining)]
285 private static bool FirstIsToLeftOfSecond(ulong firstSource, ulong firstTarget,
286     ↪ ulong secondSource, ulong secondTarget)
287     => firstSource < secondSource || (firstSource == secondSource && firstTarget <
288     ↪ secondTarget);
289
290 [MethodImpl(MethodImplOptions.AggressiveInlining)]
291 private static bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
292     ↪ ulong secondSource, ulong secondTarget)

```

```

282         => firstSource > secondSource || (firstSource == secondSource && firstTarget >
283             ↪ secondTarget);
284
285 [MethodImpl(MethodImplOptions.AggressiveInlining)]
286 protected override void ClearNode(ulong node)
287 {
288     Links[node].LeftAsSource = OUL;
289     Links[node].RightAsSource = OUL;
290     Links[node].SizeAsSource = OUL;
291 }
292
293 [MethodImpl(MethodImplOptions.AggressiveInlining)]
294 protected override ulong GetZero() => OUL;
295
296 [MethodImpl(MethodImplOptions.AggressiveInlining)]
297 protected override ulong GetOne() => 1UL;
298
299 [MethodImpl(MethodImplOptions.AggressiveInlining)]
300 protected override ulong GetTwo() => 2UL;
301
302 [MethodImpl(MethodImplOptions.AggressiveInlining)]
303 protected override bool ValueEqualToZero(IntPtr pointer) =>
304     ↪ *(ulong*)pointer.ToPointer() == OUL;
305
306 [MethodImpl(MethodImplOptions.AggressiveInlining)]
307 protected override bool EqualToZero(ulong value) => value == OUL;
308
309 [MethodImpl(MethodImplOptions.AggressiveInlining)]
310 protected override bool IsEquals(ulong first, ulong second) => first == second;
311
312 [MethodImpl(MethodImplOptions.AggressiveInlining)]
313 protected override bool GreaterThanZero(ulong value) => value > OUL;
314
315 [MethodImpl(MethodImplOptions.AggressiveInlining)]
316 protected override bool GreaterThan(ulong first, ulong second) => first > second;
317
318 [MethodImpl(MethodImplOptions.AggressiveInlining)]
319 protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >=
320     ↪ second;
321
322 [MethodImpl(MethodImplOptions.AggressiveInlining)]
323 protected override bool GreaterOrEqualThanZero(ulong value) => true; // value >= 0
324     ↪ is always true for ulong
325
326 [MethodImpl(MethodImplOptions.AggressiveInlining)]
327 protected override bool LessOrEqualThanZero(ulong value) => value == 0; // value is
328     ↪ always >= 0 for ulong
329
330 [MethodImpl(MethodImplOptions.AggressiveInlining)]
331 protected override bool LessOrEqualThan(ulong first, ulong second) => first <=
332     ↪ second;
333
334 [MethodImpl(MethodImplOptions.AggressiveInlining)]
335 protected override bool LessThanZero(ulong value) => false; // value < 0 is always
336     ↪ false for ulong
337
338 [MethodImpl(MethodImplOptions.AggressiveInlining)]
339 protected override bool LessThan(ulong first, ulong second) => first < second;
340
341 [MethodImpl(MethodImplOptions.AggressiveInlining)]
342 protected override ulong Increment(ulong value) => ++value;
343
344 [MethodImpl(MethodImplOptions.AggressiveInlining)]
345 protected override ulong Decrement(ulong value) => --value;
346
347 [MethodImpl(MethodImplOptions.AggressiveInlining)]
348 protected override ulong Add(ulong first, ulong second) => first + second;
349
350 [MethodImpl(MethodImplOptions.AggressiveInlining)]
351 protected override ulong Subtract(ulong first, ulong second) => first - second;
352 }
353
354 private class LinksTargetsTreeMethods : LinksTreeMethodsBase
355 {
356     public LinksTargetsTreeMethods(UInt64ResizableDirectMemoryLinks memory)
357         : base(memory)
358     {
359     }
360 }

```

```

354 //protected override IntPtr GetLeft(ulong node) => new
355     ↳ IntPtr(&Links[node].LeftAsTarget);
356
357 //protected override IntPtr GetRight(ulong node) => new
358     ↳ IntPtr(&Links[node].RightAsTarget);
359
360 //protected override ulong GetSize(ulong node) => Links[node].SizeAsTarget;
361
362 //protected override void SetLeft(ulong node, ulong left) =>
363     ↳ Links[node].LeftAsTarget = left;
364
365 //protected override void SetRight(ulong node, ulong right) =>
366     ↳ Links[node].RightAsTarget = right;
367
368 //protected override void SetSize(ulong node, ulong size) =>
369     ↳ Links[node].SizeAsTarget = size;
370
371 protected override IntPtr GetLeftPointer(ulong node) => new
372     ↳ IntPtr(&Links[node].LeftAsTarget);
373
374 protected override IntPtr GetRightPointer(ulong node) => new
375     ↳ IntPtr(&Links[node].RightAsTarget);
376
377 protected override ulong GetLeftValue(ulong node) => Links[node].LeftAsTarget;
378
379 protected override ulong GetRightValue(ulong node) => Links[node].RightAsTarget;
380
381 protected override ulong GetSize(ulong node)
382 {
383     var previousValue = Links[node].SizeAsTarget;
384     //return Math.PartialRead(previousValue, 5, -5);
385     return (previousValue & 4294967264) >> 5;
386 }
387
388 protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsTarget
389     ↳ = left;
390
391 protected override void SetRight(ulong node, ulong right) =>
392     ↳ Links[node].RightAsTarget = right;
393
394 protected override void SetSize(ulong node, ulong size)
395 {
396     var previousValue = Links[node].SizeAsTarget;
397     //var modified = Math.PartialWrite(previousValue, size, 5, -5);
398     var modified = (previousValue & 31) | ((size & 134217727) << 5);
399     Links[node].SizeAsTarget = modified;
400 }
401
402 protected override bool GetLeftIsChild(ulong node)
403 {
404     var previousValue = Links[node].SizeAsTarget;
405     //return (Integer)Math.PartialRead(previousValue, 4, 1);
406     return (previousValue & 16) >> 4 == 1UL;
407     // TODO: Check if this is possible to use
408     //var nodeSize = GetSize(node);
409     //var left = GetLeftValue(node);
410     //var leftSize = GetSizeOrZero(left);
411     //return leftSize > 0 && nodeSize > leftSize;
412 }
413
414 protected override void SetLeftIsChild(ulong node, bool value)
415 {
416     var previousValue = Links[node].SizeAsTarget;
417     //var modified = Math.PartialWrite(previousValue, (ulong)(Integer)value, 4, 1);
418     var modified = (previousValue & 4294967279) | ((value ? 1UL : 0UL) << 4);
419     Links[node].SizeAsTarget = modified;
420 }
421
422 protected override bool GetRightIsChild(ulong node)
423 {
424     var previousValue = Links[node].SizeAsTarget;
425     //return (Integer)Math.PartialRead(previousValue, 3, 1);
426     return (previousValue & 8) >> 3 == 1UL;
427     // TODO: Check if this is possible to use
428     //var nodeSize = GetSize(node);
429     //var right = GetRightValue(node);
430     //var rightSize = GetSizeOrZero(right);
431     //return rightSize > 0 && nodeSize > rightSize;

```



```

423     }
424
425     protected override void SetRightIsChild(ulong node, bool value)
426     {
427         var previousValue = Links[node].SizeAsTarget;
428         //var modified = Math.PartialWrite(previousValue, (ulong)(Integer)value, 3, 1);
429         var modified = (previousValue & 4294967287) | ((value ? 1UL : 0UL) << 3);
430         Links[node].SizeAsTarget = modified;
431     }
432
433     protected override sbyte GetBalance(ulong node)
434     {
435         var previousValue = Links[node].SizeAsTarget;
436         //var value = Math.PartialRead(previousValue, 0, 3);
437         var value = previousValue & 7;
438         var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) << 5) | value & 3 |
439             ↪ 124 : value & 3);
440         return unpackedValue;
441     }
442
443     protected override void SetBalance(ulong node, sbyte value)
444     {
445         var previousValue = Links[node].SizeAsTarget;
446         var packagedValue = (ulong)((((byte)value >> 5) & 4) | value & 3);
447         //var modified = Math.PartialWrite(previousValue, packagedValue, 0, 3);
448         var modified = (previousValue & 4294967288) | (packagedValue & 7);
449         Links[node].SizeAsTarget = modified;
450     }
451
452     protected override bool FirstIsToTheLeftOfSecond(ulong first, ulong second)
453     => Links[first].Target < Links[second].Target ||
454         (Links[first].Target == Links[second].Target && Links[first].Source <
455             ↪ Links[second].Source);
456
457     protected override bool FirstIsToTheRightOfSecond(ulong first, ulong second)
458     => Links[first].Target > Links[second].Target ||
459         (Links[first].Target == Links[second].Target && Links[first].Source >
460             ↪ Links[second].Source);
461
462     protected override ulong GetTreeRoot() => Header->FirstAsTarget;
463
464     protected override ulong GetBasePartValue(ulong link) => Links[link].Target;
465
466     [MethodImpl(MethodImplOptions.AggressiveInlining)]
467     protected override void ClearNode(ulong node)
468     {
469         Links[node].LeftAsTarget = 0UL;
470         Links[node].RightAsTarget = 0UL;
471         Links[node].SizeAsTarget = 0UL;
472     }
473 }
474 }

```

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

```

1  using System.Collections.Generic;
2
3  namespace Platform.Data.Doublets.Sequences.Converters
4  {
5      public class BalancedVariantConverter<TLink> : LinksListToSequenceConverterBase<TLink>
6      {
7          public BalancedVariantConverter(ILinks<TLink> links) : base(links) { }
8
9          public override TLink Convert(ICollection<TLink> sequence)
10         {
11             var length = sequence.Count;
12             if (length < 1)
13             {
14                 return default;
15             }
16             if (length == 1)
17             {
18                 return sequence[0];
19             }
20             // Make copy of next layer
21             if (length > 2)
22             {
23                 // TODO: Try to use stackalloc (which at the moment is not working with
24                 ↪ generics) but will be possible with Sigil

```

```

24         var halvedSequence = new TLink[(length / 2) + (length % 2)];
25         HalveSequence(halvedSequence, sequence, length);
26         sequence = halvedSequence;
27         length = halvedSequence.Length;
28     }
29     // Keep creating layer after layer
30     while (length > 2)
31     {
32         HalveSequence(sequence, sequence, length);
33         length = (length / 2) + (length % 2);
34     }
35     return Links.GetOrCreate(sequence[0], sequence[1]);
36 }
37
38 private void HalveSequence(IList<TLink> destination, IList<TLink> source, int length)
39 {
40     var loopedLength = length - (length % 2);
41     for (var i = 0; i < loopedLength; i += 2)
42     {
43         destination[i / 2] = Links.GetOrCreate(source[i], source[i + 1]);
44     }
45     if (length > loopedLength)
46     {
47         destination[length / 2] = source[length - 1];
48     }
49 }
50 }
51 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Interfaces;
5  using Platform.Collections;
6  using Platform.Singletons;
7  using Platform.Numbers;
8  using Platform.Data.Constants;
9  using Platform.Data.Doublets.Sequences.Frequencies.Cache;
10
11 namespace Platform.Data.Doublets.Sequences.Converters
12 {
13     /// <remarks>
14     /// TODO: Возможно будет лучше если алгоритм будет выполняться полностью изолированно от
15     /// ↳ Links на этапе сжатия.
16     /// А именно будет создаваться временный список пар необходимых для выполнения сжатия, в
17     /// ↳ таком случае тип значения элемента массива может быть любым, как char так и ulong.
18     /// Как только список/словарь пар был выявлен можно разом выполнить создание всех этих
19     /// ↳ пар, а так же разом выполнить замену.
20     /// </remarks>
21     public class CompressingConverter<TLink> : LinksListToSequenceConverterBase<TLink>
22     {
23         private static readonly LinksCombinedConstants<bool, TLink, long> _constants =
24             ↳ Default<LinksCombinedConstants<bool, TLink, long>>.Instance;
25         private static readonly EqualityComparer<TLink> _equalityComparer =
26             ↳ EqualityComparer<TLink>.Default;
27         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
28
29         private readonly IConverter<IList<TLink>, TLink> _baseConverter;
30         private readonly LinkFrequenciesCache<TLink> _doubletFrequenciesCache;
31         private readonly TLink _minFrequencyToCompress;
32         private readonly bool _doInitialFrequenciesIncrement;
33         private Doublet<TLink> _maxDoublet;
34         private LinkFrequency<TLink> _maxDoubletData;
35
36         private struct HalfDoublet
37         {
38             public TLink Element;
39             public LinkFrequency<TLink> DoubletData;
40
41             public HalfDoublet(TLink element, LinkFrequency<TLink> doubletData)
42             {
43                 Element = element;
44                 DoubletData = doubletData;
45             }
46
47             public override string ToString() => $"{Element}: ({DoubletData})";
48         }
49
50         public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
51             ↳ baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache)

```

```

46         : this(links, baseConverter, doubletFrequenciesCache, Integer<TLink>.One, true)
47     {
48     }
49
50     public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
    ↪ baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache, bool
    ↪ doInitialFrequenciesIncrement)
51         : this(links, baseConverter, doubletFrequenciesCache, Integer<TLink>.One,
    ↪ doInitialFrequenciesIncrement)
52     {
53     }
54
55     public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
    ↪ baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache, TLink
    ↪ minFrequencyToCompress, bool doInitialFrequenciesIncrement)
56         : base(links)
57     {
58         _baseConverter = baseConverter;
59         _doubletFrequenciesCache = doubletFrequenciesCache;
60         if (_comparer.Compare(minFrequencyToCompress, Integer<TLink>.One) < 0)
61         {
62             minFrequencyToCompress = Integer<TLink>.One;
63         }
64         _minFrequencyToCompress = minFrequencyToCompress;
65         _doInitialFrequenciesIncrement = doInitialFrequenciesIncrement;
66         ResetMaxDoublet();
67     }
68
69     public override TLink Convert(IList<TLink> source) =>
    ↪ _baseConverter.Convert(Compress(source));
70
71     /// <remarks>
72     /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte\_pair\_encoding .
73     /// Faster version (doublets' frequencies dictionary is not recreated).
74     /// </remarks>
75     private IList<TLink> Compress(IList<TLink> sequence)
76     {
77         if (sequence.IsNullOrEmpty())
78         {
79             return null;
80         }
81         if (sequence.Count == 1)
82         {
83             return sequence;
84         }
85         if (sequence.Count == 2)
86         {
87             return new[] { Links.GetOrCreate(sequence[0], sequence[1]) };
88         }
89         // TODO: arraypool with min size (to improve cache locality) or stackalloc with Sigil
90         var copy = new HalfDoublet[sequence.Count];
91         Doublet<TLink> doublet = default;
92         for (var i = 1; i < sequence.Count; i++)
93         {
94             doublet.Source = sequence[i - 1];
95             doublet.Target = sequence[i];
96             LinkFrequency<TLink> data;
97             if (_doInitialFrequenciesIncrement)
98             {
99                 data = _doubletFrequenciesCache.IncrementFrequency(ref doublet);
100             }
101             else
102             {
103                 data = _doubletFrequenciesCache.GetFrequency(ref doublet);
104                 if (data == null)
105                 {
106                     throw new NotSupportedException("If you ask not to increment
    ↪ frequencies, it is expected that all frequencies for the sequence
    ↪ are prepared.");
107                 }
108             }
109             copy[i - 1].Element = sequence[i - 1];
110             copy[i - 1].DoubletData = data;
111             UpdateMaxDoublet(ref doublet, data);
112         }
113         copy[sequence.Count - 1].Element = sequence[sequence.Count - 1];
114         copy[sequence.Count - 1].DoubletData = new LinkFrequency<TLink>();
115         if (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)

```

```

116     {
117         var newLength = ReplaceDoublets(copy);
118         sequence = new TLink[newLength];
119         for (int i = 0; i < newLength; i++)
120         {
121             sequence[i] = copy[i].Element;
122         }
123     }
124     return sequence;
125 }
126
127 /// <remarks>
128 /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte\_pair\_encoding
129 /// </remarks>
130 private int ReplaceDoublets(HalfDoublet[] copy)
131 {
132     var oldLength = copy.Length;
133     var newLength = copy.Length;
134     while (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
135     {
136         var maxDoubletSource = _maxDoublet.Source;
137         var maxDoubletTarget = _maxDoublet.Target;
138         if (_equalityComparer.Equals(_maxDoubletData.Link, _constants.Null))
139         {
140             _maxDoubletData.Link = Links.GetOrCreate(maxDoubletSource, maxDoubletTarget);
141         }
142         var maxDoubletReplacementLink = _maxDoubletData.Link;
143         oldLength--;
144         var oldLengthMinusTwo = oldLength - 1;
145         // Substitute all usages
146         int w = 0, r = 0; // (r == read, w == write)
147         for (; r < oldLength; r++)
148         {
149             if (_equalityComparer.Equals(copy[r].Element, maxDoubletSource) &&
150                 ↪ _equalityComparer.Equals(copy[r + 1].Element, maxDoubletTarget))
151             {
152                 if (r > 0)
153                 {
154                     var previous = copy[w - 1].Element;
155                     copy[w - 1].DoubletData.DecrementFrequency();
156                     copy[w - 1].DoubletData =
157                         ↪ _doubletFrequenciesCache.IncrementFrequency(previous,
158                             ↪ maxDoubletReplacementLink);
159                 }
160                 if (r < oldLengthMinusTwo)
161                 {
162                     var next = copy[r + 2].Element;
163                     copy[r + 1].DoubletData.DecrementFrequency();
164                     copy[w].DoubletData = _doubletFrequenciesCache.IncrementFrequency(maxDoubletReplacementLink,
165                         ↪ next);
166                 }
167                 copy[w++].Element = maxDoubletReplacementLink;
168                 r++;
169                 newLength--;
170             }
171             else
172             {
173                 copy[w++] = copy[r];
174             }
175         }
176         if (w < newLength)
177         {
178             copy[w] = copy[r];
179         }
180         oldLength = newLength;
181         ResetMaxDoublet();
182         UpdateMaxDoublet(copy, newLength);
183     }
184     return newLength;
185 }
186
187 [MethodImpl(MethodImplOptions.AggressiveInlining)]
188 private void ResetMaxDoublet()
189 {
190     _maxDoublet = new Doublet<TLink>();
191     _maxDoubletData = new LinkFrequency<TLink>();
192 }

```

```

190 [MethodImpl(MethodImplOptions.AggressiveInlining)]
191 private void UpdateMaxDoublet(HalfDoublet[] copy, int length)
192 {
193     Doublet<TLink> doublet = default;
194     for (var i = 1; i < length; i++)
195     {
196         doublet.Source = copy[i - 1].Element;
197         doublet.Target = copy[i].Element;
198         UpdateMaxDoublet(ref doublet, copy[i - 1].DoubletData);
199     }
200 }
201
202 [MethodImpl(MethodImplOptions.AggressiveInlining)]
203 private void UpdateMaxDoublet(ref Doublet<TLink> doublet, LinkFrequency<TLink> data)
204 {
205     var frequency = data.Frequency;
206     var maxFrequency = _maxDoubletData.Frequency;
207     //if (frequency > _minFrequencyToCompress && (maxFrequency < frequency ||
208     ↪ (maxFrequency == frequency && doublet.Source + doublet.Target < /* gives better
209     ↪ compression string data (and gives collisions quickly) */ _maxDoublet.Source +
210     ↪ _maxDoublet.Target)))
211     if (_comparer.Compare(frequency, _minFrequencyToCompress) > 0 &&
212     ↪ (_comparer.Compare(maxFrequency, frequency) < 0 ||
213     ↪ (_equalityComparer.Equals(maxFrequency, frequency) &&
214     ↪ _comparer.Compare(Arithmetic.Add(doublet.Source, doublet.Target),
215     ↪ Arithmetic.Add(_maxDoublet.Source, _maxDoublet.Target)) > 0))) /* gives
216     ↪ better stability and better compression on sequent data and even on random
217     ↪ numbers data (but gives collisions anyway) */
218     {
219         _maxDoublet = doublet;
220         _maxDoubletData = data;
221     }
222 }
223 }
224 }
225 }
226 }

```

#### ./Platform.Data.Doublets/Sequences/Converters/LinksListToSequenceConverterBase.cs

```

1 using System.Collections.Generic;
2 using Platform.Interfaces;
3
4 namespace Platform.Data.Doublets.Sequences.Converters
5 {
6     public abstract class LinksListToSequenceConverterBase<TLink> : IConverter<IList<TLink>,
7     ↪ TLink>
8     {
9         protected readonly ILinks<TLink> Links;
10        public LinksListToSequenceConverterBase(ILinks<TLink> links) => Links = links;
11        public abstract TLink Convert(IList<TLink> source);
12    }
13 }

```

#### ./Platform.Data.Doublets/Sequences/Converters/OptimalVariantConverter.cs

```

1 using System.Collections.Generic;
2 using System.Linq;
3 using Platform.Interfaces;
4
5 namespace Platform.Data.Doublets.Sequences.Converters
6 {
7     public class OptimalVariantConverter<TLink> : LinksListToSequenceConverterBase<TLink>
8     {
9         private static readonly EqualityComparer<TLink> _equalityComparer =
10        ↪ EqualityComparer<TLink>.Default;
11        private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
12
13        private readonly IConverter<IList<TLink>> _sequenceToItsLocalElementLevelsConverter;
14
15        public OptimalVariantConverter(ILinks<TLink> links, IConverter<IList<TLink>>
16        ↪ sequenceToItsLocalElementLevelsConverter) : base(links)
17        => _sequenceToItsLocalElementLevelsConverter =
18        ↪ sequenceToItsLocalElementLevelsConverter;
19
20        public override TLink Convert(IList<TLink> sequence)
21        {
22            var length = sequence.Count;
23            if (length == 1)
24            {
25                return sequence[0];
26            }
27            var links = Links;
28        }
29    }
30 }

```

```

25     if (length == 2)
26     {
27         return links.GetOrCreate(sequence[0], sequence[1]);
28     }
29     sequence = sequence.ToArray();
30     var levels = _sequenceToItsLocalElementLevelsConverter.Convert(sequence);
31     while (length > 2)
32     {
33         var levelRepeat = 1;
34         var currentLevel = levels[0];
35         var previousLevel = levels[0];
36         var skipOnce = false;
37         var w = 0;
38         for (var i = 1; i < length; i++)
39         {
40             if (_equalityComparer.Equals(currentLevel, levels[i]))
41             {
42                 levelRepeat++;
43                 skipOnce = false;
44                 if (levelRepeat == 2)
45                 {
46                     sequence[w] = links.GetOrCreate(sequence[i - 1], sequence[i]);
47                     var newLevel = i >= length - 1 ?
48                         GetPreviousLowerThanCurrentOrCurrent(previousLevel,
49                             ↪ currentLevel) :
50                         i < 2 ?
51                         GetNextLowerThanCurrentOrCurrent(currentLevel, levels[i + 1]) :
52                         GetGreatestNeighbourLowerThanCurrentOrCurrent(previousLevel,
53                             ↪ currentLevel, levels[i + 1]);
54                     levels[w] = newLevel;
55                     previousLevel = currentLevel;
56                     w++;
57                     levelRepeat = 0;
58                     skipOnce = true;
59                 }
60                 else if (i == length - 1)
61                 {
62                     sequence[w] = sequence[i];
63                     levels[w] = levels[i];
64                     w++;
65                 }
66             }
67             else
68             {
69                 currentLevel = levels[i];
70                 levelRepeat = 1;
71                 if (skipOnce)
72                 {
73                     skipOnce = false;
74                 }
75                 else
76                 {
77                     sequence[w] = sequence[i - 1];
78                     levels[w] = levels[i - 1];
79                     previousLevel = levels[w];
80                     w++;
81                 }
82                 if (i == length - 1)
83                 {
84                     sequence[w] = sequence[i];
85                     levels[w] = levels[i];
86                     w++;
87                 }
88             }
89         }
90         length = w;
91     }
92     return links.GetOrCreate(sequence[0], sequence[1]);
93 }
94
95 private static TLink GetGreatestNeighbourLowerThanCurrentOrCurrent(TLink previous, TLink
96 ↪ current, TLink next)
97 {
98     return _comparer.Compare(previous, next) > 0
99         ? _comparer.Compare(previous, current) < 0 ? previous : current
100         : _comparer.Compare(next, current) < 0 ? next : current;
101 }

```

```

100     private static TLink GetNextLowerThanCurrentOrCurrent(TLink current, TLink next) =>
101         ↪ _comparer.Compare(next, current) < 0 ? next : current;
102
103     private static TLink GetPreviousLowerThanCurrentOrCurrent(TLink previous, TLink current)
104         ↪ => _comparer.Compare(previous, current) < 0 ? previous : current;
105 }
106 }

```

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

```

1  using System.Collections.Generic;
2  using Platform.Interfaces;
3
4  namespace Platform.Data.Doublets.Sequences.Converters
5  {
6      public class SequenceToItsLocalElementLevelsConverter<TLink> : LinksOperatorBase<TLink>,
7          ↪ IConverter<IList<TLink>>
8      {
9          private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
10
11         private readonly IConverter<Doublet<TLink>, TLink> _linkToItsFrequencyToNumberConveter;
12
13         public SequenceToItsLocalElementLevelsConverter(ILinks<TLink> links,
14             ↪ IConverter<Doublet<TLink>, TLink> linkToItsFrequencyToNumberConveter) : base(links)
15             ↪ => _linkToItsFrequencyToNumberConveter = linkToItsFrequencyToNumberConveter;
16
17         public IList<TLink> Convert(IList<TLink> sequence)
18         {
19             var levels = new TLink[sequence.Count];
20             levels[0] = GetFrequencyNumber(sequence[0], sequence[1]);
21             for (var i = 1; i < sequence.Count - 1; i++)
22             {
23                 var previous = GetFrequencyNumber(sequence[i - 1], sequence[i]);
24                 var next = GetFrequencyNumber(sequence[i], sequence[i + 1]);
25                 levels[i] = _comparer.Compare(previous, next) > 0 ? previous : next;
26             }
27             levels[levels.Length - 1] = GetFrequencyNumber(sequence[sequence.Count - 2],
28                 ↪ sequence[sequence.Count - 1]);
29             return levels;
30         }
31
32         public TLink GetFrequencyNumber(TLink source, TLink target) =>
33             ↪ _linkToItsFrequencyToNumberConveter.Convert(new Doublet<TLink>(source, target));
34     }
35 }

```

./Platform.Data.Doublets/Sequences/CreteriaMatchers/DefaultSequenceElementCriterionMatcher.cs

```

1  using Platform.Interfaces;
2
3  namespace Platform.Data.Doublets.Sequences.CreteriaMatchers
4  {
5      public class DefaultSequenceElementCriterionMatcher<TLink> : LinksOperatorBase<TLink>,
6          ↪ ICriterionMatcher<TLink>
7      {
8          public DefaultSequenceElementCriterionMatcher(ILinks<TLink> links) : base(links) { }
9          public bool IsMatched(TLink argument) => Links.IsPartialPoint(argument);
10     }

```

./Platform.Data.Doublets/Sequences/CreteriaMatchers/MarkedSequenceCriterionMatcher.cs

```

1  using System.Collections.Generic;
2  using Platform.Interfaces;
3
4  namespace Platform.Data.Doublets.Sequences.CreteriaMatchers
5  {
6      public class MarkedSequenceCriterionMatcher<TLink> : ICriterionMatcher<TLink>
7      {
8          private static readonly EqualityComparer<TLink> _equalityComparer =
9             ↪ EqualityComparer<TLink>.Default;
10
11         private readonly ILinks<TLink> _links;
12         private readonly TLink _sequenceMarkerLink;
13
14         public MarkedSequenceCriterionMatcher(ILinks<TLink> links, TLink sequenceMarkerLink)
15         {
16             _links = links;
17             _sequenceMarkerLink = sequenceMarkerLink;
18         }
19
20         public bool IsMatched(TLink sequenceCandidate)

```

```

20         => _equalityComparer.Equals(_links.GetSource(sequenceCandidate), _sequenceMarkerLink)
21         || !_equalityComparer.Equals(_links.SearchOrDefault(_sequenceMarkerLink,
22             ↪ sequenceCandidate), _links.Constants.Null);
23     }

```

# ./Platform.Data.Doublets/Sequences/DefaultSequenceAppender.cs

```

1  using System.Collections.Generic;
2  using Platform.Collections.Stacks;
3  using Platform.Data.Doublets.Sequences.HeightProviders;
4  using Platform.Data.Sequencing;
5
6  namespace Platform.Data.Doublets.Sequences
7  {
8      public class DefaultSequenceAppender<TLink> : LinksOperatorBase<TLink>,
9          ↪ ISequenceAppender<TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
12             ↪ EqualityComparer<TLink>.Default;
13
14         private readonly IStack<TLink> _stack;
15         private readonly ISequenceHeightProvider<TLink> _heightProvider;
16
17         public DefaultSequenceAppender(ILinks<TLink> links, IStack<TLink> stack,
18             ↪ ISequenceHeightProvider<TLink> heightProvider)
19             : base(links)
20         {
21             _stack = stack;
22             _heightProvider = heightProvider;
23         }
24
25         public TLink Append(TLink sequence, TLink appendant)
26         {
27             var cursor = sequence;
28             while (!_equalityComparer.Equals(_heightProvider.Get(cursor), default))
29             {
30                 var source = Links.GetSource(cursor);
31                 var target = Links.GetTarget(cursor);
32                 if (_equalityComparer.Equals(_heightProvider.Get(source),
33                     ↪ _heightProvider.Get(target)))
34                 {
35                     break;
36                 }
37                 else
38                 {
39                     _stack.Push(source);
40                     cursor = target;
41                 }
42             }
43             var left = cursor;
44             var right = appendant;
45             while (!_equalityComparer.Equals(cursor = _stack.Pop(), Links.Constants.Null))
46             {
47                 right = Links.GetOrCreate(left, right);
48                 left = cursor;
49             }
50             return Links.GetOrCreate(left, right);
51         }
52     }
53 }

```

# ./Platform.Data.Doublets/Sequences/DuplicateSegmentsCounter.cs

```

1  using System.Collections.Generic;
2  using System.Linq;
3  using Platform.Interfaces;
4
5  namespace Platform.Data.Doublets.Sequences
6  {
7      public class DuplicateSegmentsCounter<TLink> : ICounter<int>
8      {
9          private readonly IProvider<IList<KeyValuePair<IList<TLink>, IList<TLink>>>>
10             ↪ _duplicateFragmentsProvider;
11         public DuplicateSegmentsCounter(IProvider<IList<KeyValuePair<IList<TLink>,
12             ↪ IList<TLink>>>> duplicateFragmentsProvider) => _duplicateFragmentsProvider =
13             ↪ duplicateFragmentsProvider;
14         public int Count() => _duplicateFragmentsProvider.Get().Sum(x => x.Value.Count);
15     }
16 }

```





```

69         {
70             WalkAll(sequenceElements);
71         }
72     }
73     return _links.Constants.Continue;
74 });
75 var resultList = _groups.ToList();
76 var comparer = Default<ItemComparer>.Instance;
77 resultList.Sort(comparer);
78 #if DEBUG
79     foreach (var item in resultList)
80     {
81         PrintDuplicates(item);
82     }
83 #endif
84     return resultList;
85 }
86
87 protected override Segment<TLink> CreateSegment(IList<TLink> elements, int offset, int
    ↪ length) => new Segment<TLink>(elements, offset, length);
88
89 protected override void OnDuplicateFound(Segment<TLink> segment)
90 {
91     var duplicates = CollectDuplicatesForSegment(segment);
92     if (duplicates.Count > 1)
93     {
94         _groups.Add(new KeyValuePair<IList<TLink>, IList<TLink>>(segment.ToArray(),
    ↪ duplicates));
95     }
96 }
97
98 private List<TLink> CollectDuplicatesForSegment(Segment<TLink> segment)
99 {
100     var duplicates = new List<TLink>();
101     var readAsElement = new HashSet<TLink>();
102     _sequences.Each(sequence =>
103     {
104         duplicates.Add(sequence);
105         readAsElement.Add(sequence);
106         return true; // Continue
107     }, segment);
108     if (duplicates.Any(x => _visited.Get((Integer<TLink>)x)))
109     {
110         return new List<TLink>();
111     }
112     foreach (var duplicate in duplicates)
113     {
114         var duplicateBitIndex = (long)(Integer<TLink>)duplicate;
115         _visited.Set(duplicateBitIndex);
116     }
117     if (_sequences is Sequences sequencesExperiments)
118     {
119         var partiallyMatched = sequencesExperiments.GetAllPartiallyMatchingSequences4((H_
    ↪ ashSet<ulong>)(object)readAsElement,
    ↪ (IList<ulong>)segment);
120         foreach (var partiallyMatchedSequence in partiallyMatched)
121         {
122             TLink sequenceIndex = (Integer<TLink>)partiallyMatchedSequence;
123             duplicates.Add(sequenceIndex);
124         }
125     }
126     duplicates.Sort();
127     return duplicates;
128 }
129
130 private void PrintDuplicates(KeyValuePair<IList<TLink>, IList<TLink>> duplicatesItem)
131 {
132     if (!(_links is ILinks<ulong> ulongLinks))
133     {
134         return;
135     }
136     var duplicatesKey = duplicatesItem.Key;
137     var keyString = UnicodeMap.FromLinksToString((IList<ulong>)duplicatesKey);
138     Console.WriteLine($"> {keyString} ({string.Join(", ", duplicatesKey)}");
139     var duplicatesList = duplicatesItem.Value;
140     for (int i = 0; i < duplicatesList.Count; i++)
141     {
142         ulong sequenceIndex = (Integer<TLink>)duplicatesList[i];

```

```

143         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));
144         Console.WriteLine(formattedSequenceStructure);
145         var sequenceString = UnicodeMap.FromSequenceLinkToString(sequenceIndex,
            ↳ ulongLinks);
146         Console.WriteLine(sequenceString);
147     }
148     Console.WriteLine();
149 }
150 }
151 }

```

./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  namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
8  {
9      /// <remarks>
10     /// Can be used to operate with many CompressingConverters (to keep global frequencies data
11     ↳ between them).
12     /// TODO: Extract interface to implement frequencies storage inside Links storage
13     /// </remarks>
14     public class LinkFrequenciesCache<TLink> : LinksOperatorBase<TLink>
15     {
16         private static readonly EqualityComparer<TLink> _equalityComparer =
17         ↳ EqualityComparer<TLink>.Default;
18         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
19
20         private readonly Dictionary<Doublet<TLink>, LinkFrequency<TLink>> _doubletsCache;
21         private readonly ICounter<TLink, TLink> _frequencyCounter;
22
23         public LinkFrequenciesCache(ILinks<TLink> links, ICounter<TLink, TLink> frequencyCounter)
24         : base(links)
25         {
26             _doubletsCache = new Dictionary<Doublet<TLink>, LinkFrequency<TLink>>(4096,
27             ↳ DoubletComparer<TLink>.Default);
28             _frequencyCounter = frequencyCounter;
29         }
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         public LinkFrequency<TLink> GetFrequency(TLink source, TLink target)
33         {
34             var doublet = new Doublet<TLink>(source, target);
35             return GetFrequency(ref doublet);
36         }
37
38         [MethodImpl(MethodImplOptions.AggressiveInlining)]
39         public LinkFrequency<TLink> GetFrequency(ref Doublet<TLink> doublet)
40         {
41             _doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data);
42             return data;
43         }
44
45         public void IncrementFrequencies(IList<TLink> sequence)
46         {
47             for (var i = 1; i < sequence.Count; i++)
48             {
49                 IncrementFrequency(sequence[i - 1], sequence[i]);
50             }
51         }
52
53         [MethodImpl(MethodImplOptions.AggressiveInlining)]
54         public LinkFrequency<TLink> IncrementFrequency(TLink source, TLink target)
55         {
56             var doublet = new Doublet<TLink>(source, target);
57             return IncrementFrequency(ref doublet);
58         }
59
60         public void PrintFrequencies(IList<TLink> sequence)
61         {
62             for (var i = 1; i < sequence.Count; i++)
63             {
64                 PrintFrequency(sequence[i - 1], sequence[i]);
65             }
66         }
67     }
68 }

```

```

62     }
63 }
64
65 public void PrintFrequency(TLink source, TLink target)
66 {
67     var number = GetFrequency(source, target).Frequency;
68     Console.WriteLine("{0},{1} - {2}", source, target, number);
69 }
70
71 [MethodImpl(MethodImplOptions.AggressiveInlining)]
72 public LinkFrequency<TLink> IncrementFrequency(ref Doublet<TLink> doublet)
73 {
74     if (_doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data))
75     {
76         data.IncrementFrequency();
77     }
78     else
79     {
80         var link = Links.SearchOrDefault(doublet.Source, doublet.Target);
81         data = new LinkFrequency<TLink>(Integer<TLink>.One, link);
82         if (!_equalityComparer.Equals(link, default))
83         {
84             data.Frequency = Arithmetic.Add(data.Frequency,
85                 ↪ _frequencyCounter.Count(link));
86         }
87         _doubletsCache.Add(doublet, data);
88     }
89     return data;
90 }
91
92 public void ValidateFrequencies()
93 {
94     foreach (var entry in _doubletsCache)
95     {
96         var value = entry.Value;
97         var linkIndex = value.Link;
98         if (!_equalityComparer.Equals(linkIndex, default))
99         {
100             var frequency = value.Frequency;
101             var count = _frequencyCounter.Count(linkIndex);
102             // TODO: Why `frequency` always greater than `count` by 1?
103             if (((_comparer.Compare(frequency, count) > 0) &&
104                 ↪ (_comparer.Compare(Arithmetic.Subtract(frequency, count),
105                 ↪ Integer<TLink>.One) > 0))
106                 || ((_comparer.Compare(count, frequency) > 0) &&
107                 ↪ (_comparer.Compare(Arithmetic.Subtract(count, frequency),
108                 ↪ Integer<TLink>.One) > 0)))
109             {
110                 throw new InvalidOperationException("Frequencies validation failed.");
111             }
112             //else
113             //{
114             //    if (value.Frequency > 0)
115             //    {
116             //        var frequency = value.Frequency;
117             //        linkIndex = _createLink(entry.Key.Source, entry.Key.Target);
118             //        var count = _countLinkFrequency(linkIndex);
119             //        if ((frequency > count && frequency - count > 1) || (count > frequency
120             //            ↪ && count - frequency > 1))
121             //            throw new Exception("Frequencies validation failed.");
122             //    }
123             //}
124     }
125 }
126
127 }
128
129 }

```

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

```

1 using System.Runtime.CompilerServices;
2 using Platform.Numbers;
3
4 namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
5 {
6     public class LinkFrequency<TLink>
7     {
8         public TLink Frequency { get; set; }

```

```

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

```

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

```

1 using Platform.Interfaces;
2
3 namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
4 {
5     public class FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<TLink> :
6         ↳ IConverter<Doublet<TLink>, TLink>
7     {
8         private readonly LinkFrequenciesCache<TLink> _cache;
9         public
10             ↳ FrequenciesCacheBasedLinkToItsFrequencyNumberConverter(LinkFrequenciesCache<TLink>
11                 ↳ cache) => _cache = cache;
12         public TLink Convert(Doublet<TLink> source) => _cache.GetFrequency(ref source).Frequency;
13     }
14 }

```

./Platform.Data.Doublets/Sequences/Frequencies/Counters/MarkedSequenceSymbolFrequencyOneOffCounter.cs

```

1 using Platform.Interfaces;
2
3 namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
4 {
5     public class MarkedSequenceSymbolFrequencyOneOffCounter<TLink> :
6         ↳ SequenceSymbolFrequencyOneOffCounter<TLink>
7     {
8         private readonly ICriterionMatcher<TLink> _markedSequenceMatcher;
9
10         public MarkedSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links,
11             ↳ ICriterionMatcher<TLink> markedSequenceMatcher, TLink sequenceLink, TLink symbol)
12             : base(links, sequenceLink, symbol)
13             => _markedSequenceMatcher = markedSequenceMatcher;
14
15         public override TLink Count()
16         {
17             if (!_markedSequenceMatcher.IsMatched(_sequenceLink))
18             {
19                 return default;
20             }
21             return base.Count();
22         }
23     }
24 }

```

./Platform.Data.Doublets/Sequences/Frequencies/Counters/SequenceSymbolFrequencyOneOffCounter.cs

```

1 using System.Collections.Generic;
2 using Platform.Interfaces;
3 using Platform.Numbers;
4 using Platform.Data.Sequences;
5
6 namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
7 {
8     public class SequenceSymbolFrequencyOneOffCounter<TLink> : ICounter<TLink>
9     {
10         private static readonly EqualityComparer<TLink> _equalityComparer =
11             ↳ EqualityComparer<TLink>.Default;
12         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
13
14         protected readonly ILinks<TLink> _links;
15         protected readonly TLink _sequenceLink;
16         protected readonly TLink _symbol;
17         protected TLink _total;
18     }
19 }

```

```

17
18     public SequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links, TLink sequenceLink,
19         ↪ TLink symbol)
20     {
21         _links = links;
22         _sequenceLink = sequenceLink;
23         _symbol = symbol;
24         _total = default;
25     }
26
27     public virtual TLink Count()
28     {
29         if (_comparer.Compare(_total, default) > 0)
30         {
31             return _total;
32         }
33         StopableSequenceWalker.WalkRight(_sequenceLink, _links.GetSource, _links.GetTarget,
34         ↪ IsElement, VisitElement);
35         return _total;
36     }
37
38     private bool IsElement(TLink x) => _equalityComparer.Equals(x, _symbol) ||
39     ↪ _links.IsPartialPoint(x); // TODO: Use SequenceElementCriteriaMatcher instead of
40     ↪ IsPartialPoint
41
42     private bool VisitElement(TLink element)
43     {
44         if (_equalityComparer.Equals(element, _symbol))
45         {
46             _total = Arithmetic.Increment(_total);
47         }
48         return true;
49     }
50 }
51

```

./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyCounter.cs

```

1  using Platform.Interfaces;
2
3  namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
4  {
5      public class TotalMarkedSequenceSymbolFrequencyCounter<TLink> : ICounter<TLink, TLink>
6      {
7          private readonly ILinks<TLink> _links;
8          private readonly ICriterionMatcher<TLink> _markedSequenceMatcher;
9
10         public TotalMarkedSequenceSymbolFrequencyCounter(ILinks<TLink> links,
11         ↪ ICriterionMatcher<TLink> markedSequenceMatcher)
12         {
13             _links = links;
14             _markedSequenceMatcher = markedSequenceMatcher;
15         }
16
17         public TLink Count(TLink argument) => new
18         ↪ TotalMarkedSequenceSymbolFrequencyOneOffCounter<TLink>(_links,
19         ↪ _markedSequenceMatcher, argument).Count();
20     }
21 }
22

```

./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyOneOffCounter

```

1  using Platform.Interfaces;
2  using Platform.Numbers;
3
4  namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
5  {
6      public class TotalMarkedSequenceSymbolFrequencyOneOffCounter<TLink> :
7      ↪ TotalSequenceSymbolFrequencyOneOffCounter<TLink>
8      {
9          private readonly ICriterionMatcher<TLink> _markedSequenceMatcher;
10
11         public TotalMarkedSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links,
12         ↪ ICriterionMatcher<TLink> markedSequenceMatcher, TLink symbol)
13         : base(links, symbol)
14         => _markedSequenceMatcher = markedSequenceMatcher;
15
16         protected override void CountSequenceSymbolFrequency(TLink link)
17         {
18             var symbolFrequencyCounter = new
19             ↪ MarkedSequenceSymbolFrequencyOneOffCounter<TLink>(_links,
20             ↪ _markedSequenceMatcher, link, _symbol);
21         }
22     }
23 }
24

```

```

17         _total = Arithmetic.Add(_total, symbolFrequencyCounter.Count());
18     }
19 }
20 }

```

./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyCounter.cs

```

1 using Platform.Interfaces;
2
3 namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
4 {
5     public class TotalSequenceSymbolFrequencyCounter<TLink> : ICounter<TLink, TLink>
6     {
7         private readonly ILinks<TLink> _links;
8         public TotalSequenceSymbolFrequencyCounter(ILinks<TLink> links) => _links = links;
9         public TLink Count(TLink symbol) => new
10             ↳ TotalSequenceSymbolFrequencyOneOffCounter<TLink>(_links, symbol).Count();
11     }
12 }

```

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

```

1 using System.Collections.Generic;
2 using Platform.Interfaces;
3 using Platform.Numbers;
4
5 namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
6 {
7     public class TotalSequenceSymbolFrequencyOneOffCounter<TLink> : ICounter<TLink>
8     {
9         private static readonly EqualityComparer<TLink> _equalityComparer =
10             ↳ EqualityComparer<TLink>.Default;
11         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
12
13         protected readonly ILinks<TLink> _links;
14         protected readonly TLink _symbol;
15         protected readonly HashSet<TLink> _visits;
16         protected TLink _total;
17
18         public TotalSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links, TLink symbol)
19         {
20             _links = links;
21             _symbol = symbol;
22             _visits = new HashSet<TLink>();
23             _total = default;
24         }
25
26         public TLink Count()
27         {
28             if (_comparer.Compare(_total, default) > 0 || _visits.Count > 0)
29             {
30                 return _total;
31             }
32             CountCore(_symbol);
33             return _total;
34         }
35
36         private void CountCore(TLink link)
37         {
38             var any = _links.Constants.Any;
39             if (_equalityComparer.Equals(_links.Count(any, link), default))
40             {
41                 CountSequenceSymbolFrequency(link);
42             }
43             else
44             {
45                 _links.Each(EachElementHandler, any, link);
46             }
47         }
48
49         protected virtual void CountSequenceSymbolFrequency(TLink link)
50         {
51             var symbolFrequencyCounter = new SequenceSymbolFrequencyOneOffCounter<TLink>(_links,
52                 ↳ link, _symbol);
53             _total = Arithmetic.Add(_total, symbolFrequencyCounter.Count());
54         }
55
56         private TLink EachElementHandler(IList<TLink> doublet)
57         {
58             var constants = _links.Constants;
59             var doubletIndex = doublet[constants.IndexPart];
60             if (_visits.Add(doubletIndex))

```

```

59         {
60             CountCore(doupletIndex);
61         }
62         return constants.Continue;
63     }
64 }
65 }

```

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

```

1  using System.Collections.Generic;
2  using Platform.Interfaces;
3
4  namespace Platform.Data.Doublets.Sequences.HeightProviders
5  {
6      public class CachedSequenceHeightProvider<TLink> : LinksOperatorBase<TLink>,
7          ↳ ISequenceHeightProvider<TLink>
8      {
9          private static readonly EqualityComparer<TLink> _equalityComparer =
10             ↳ EqualityComparer<TLink>.Default;
11
12         private readonly TLink _heightPropertyMarker;
13         private readonly ISequenceHeightProvider<TLink> _baseHeightProvider;
14         private readonly IConverter<TLink> _addressToUnaryNumberConverter;
15         private readonly IConverter<TLink> _unaryNumberToAddressConverter;
16         private readonly IPropertiesOperator<TLink, TLink, TLink> _propertyOperator;
17
18         public CachedSequenceHeightProvider(
19             ILinks<TLink> links,
20             ISequenceHeightProvider<TLink> baseHeightProvider,
21             IConverter<TLink> addressToUnaryNumberConverter,
22             IConverter<TLink> unaryNumberToAddressConverter,
23             TLink heightPropertyMarker,
24             IPropertiesOperator<TLink, TLink, TLink> propertyOperator)
25             : base(links)
26         {
27             _heightPropertyMarker = heightPropertyMarker;
28             _baseHeightProvider = baseHeightProvider;
29             _addressToUnaryNumberConverter = addressToUnaryNumberConverter;
30             _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
31             _propertyOperator = propertyOperator;
32         }
33
34         public TLink Get(TLink sequence)
35         {
36             TLink height;
37             var heightValue = _propertyOperator.GetValue(sequence, _heightPropertyMarker);
38             if (_equalityComparer.Equals(heightValue, default))
39             {
40                 height = _baseHeightProvider.Get(sequence);
41                 heightValue = _addressToUnaryNumberConverter.Convert(height);
42                 _propertyOperator.SetValue(sequence, _heightPropertyMarker, heightValue);
43             }
44             else
45             {
46                 height = _unaryNumberToAddressConverter.Convert(heightValue);
47             }
48             return height;
49         }
50     }
51 }

```

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

```

1  using Platform.Interfaces;
2  using Platform.Numbers;
3
4  namespace Platform.Data.Doublets.Sequences.HeightProviders
5  {
6      public class DefaultSequenceRightHeightProvider<TLink> : LinksOperatorBase<TLink>,
7          ↳ ISequenceHeightProvider<TLink>
8      {
9          private readonly ICriterionMatcher<TLink> _elementMatcher;
10
11         public DefaultSequenceRightHeightProvider(ILinks<TLink> links, ICriterionMatcher<TLink>
12             ↳ elementMatcher) : base(links) => _elementMatcher = elementMatcher;
13
14         public TLink Get(TLink sequence)
15         {
16             var height = default(TLink);
17             var pairOrElement = sequence;
18             while (!_elementMatcher.IsMatched(pairOrElement))

```



```

17         {
18             pairOrElement = Links.GetTarget(pairOrElement);
19             height = Arithmetic.Increment(height);
20         }
21         return height;
22     }
23 }
24 }

```

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

```

1 using Platform.Interfaces;
2
3 namespace Platform.Data.Doublets.Sequences.HeightProviders
4 {
5     public interface ISequenceHeightProvider<TLink> : IProvider<TLink, TLink>
6     {
7     }
8 }

```

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

```

1 using System.Collections.Generic;
2 using Platform.Data.Doublets.Sequences.Frequencies.Cache;
3
4 namespace Platform.Data.Doublets.Sequences.Indexes
5 {
6     public class CachedFrequencyIncrementingSequenceIndex<TLink> : ISequenceIndex<TLink>
7     {
8         private static readonly EqualityComparer<TLink> _equalityComparer =
9             ↳ EqualityComparer<TLink>.Default;
10
11         private readonly LinkFrequenciesCache<TLink> _cache;
12
13         public CachedFrequencyIncrementingSequenceIndex(LinkFrequenciesCache<TLink> cache) =>
14             ↳ _cache = cache;
15
16         public bool Add(IList<TLink> sequence)
17         {
18             var indexed = true;
19             var i = sequence.Count;
20             while (--i >= 1 && (indexed = IsIndexedWithIncrement(sequence[i - 1], sequence[i])))
21                 ↳ { }
22             for (; i >= 1; i--)
23             {
24                 _cache.IncrementFrequency(sequence[i - 1], sequence[i]);
25             }
26             return indexed;
27         }
28
29         private bool IsIndexedWithIncrement(TLink source, TLink target)
30         {
31             var frequency = _cache.GetFrequency(source, target);
32             if (frequency == null)
33             {
34                 return false;
35             }
36             var indexed = !_equalityComparer.Equals(frequency.Frequency, default);
37             if (indexed)
38             {
39                 _cache.IncrementFrequency(source, target);
40             }
41             return indexed;
42         }
43
44         public bool MightContain(IList<TLink> sequence)
45         {
46             var indexed = true;
47             var i = sequence.Count;
48             while (--i >= 1 && (indexed = IsIndexed(sequence[i - 1], sequence[i]))) { }
49             return indexed;
50         }
51
52         private bool IsIndexed(TLink source, TLink target)
53         {
54             var frequency = _cache.GetFrequency(source, target);
55             if (frequency == null)
56             {
57                 return false;
58             }
59             return !_equalityComparer.Equals(frequency.Frequency, default);
60         }
61     }
62 }

```

```

58     }
59 }

./Platform.Data.Doublets/Sequences/Indexes/FrequencyIncrementingSequenceIndex.cs
1  using Platform.Interfaces;
2  using System.Collections.Generic;
3
4  namespace Platform.Data.Doublets.Sequences.Indexes
5  {
6      public class FrequencyIncrementingSequenceIndex<TLink> : SequenceIndex<TLink>,
        ↳ ISequenceIndex<TLink>
7      {
8          private static readonly EqualityComparer<TLink> _equalityComparer =
        ↳ EqualityComparer<TLink>.Default;
9
10         private readonly IPropertyOperator<TLink, TLink> _frequencyPropertyOperator;
11         private readonly IIncrementer<TLink> _frequencyIncrementer;
12
13         public FrequencyIncrementingSequenceIndex(ILinks<TLink> links, IPropertyOperator<TLink,
        ↳ TLink> frequencyPropertyOperator, IIncrementer<TLink> frequencyIncrementer)
        : base(links)
14         {
15             _frequencyPropertyOperator = frequencyPropertyOperator;
16             _frequencyIncrementer = frequencyIncrementer;
17         }
18
19         public override bool Add(IList<TLink> sequence)
20         {
21             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                 Increment(Links.GetOrCreate(sequence[i - 1], sequence[i]));
28             }
29             return indexed;
30         }
31
32         private bool IsIndexedWithIncrement(TLink source, TLink target)
33         {
34             var link = Links.SearchOrCreate(source, target);
35             var indexed = !_equalityComparer.Equals(link, default);
36             if (indexed)
37             {
38                 Increment(link);
39             }
40             return indexed;
41         }
42
43         private void Increment(TLink link)
44         {
45             var previousFrequency = _frequencyPropertyOperator.Get(link);
46             var frequency = _frequencyIncrementer.Increment(previousFrequency);
47             _frequencyPropertyOperator.Set(link, frequency);
48         }
49     }
50 }

```

```

./Platform.Data.Doublets/Sequences/Indexes/ISequenceIndex.cs
1  using System.Collections.Generic;
2
3  namespace Platform.Data.Doublets.Sequences.Indexes
4  {
5      public interface ISequenceIndex<TLink>
6      {
7          /// <summary>
8          /// Индексирует последовательность глобально, и возвращает значение,
9          /// определяющие была ли запрошенная последовательность проиндексирована ранее.
10         /// </summary>
11         /// <param name="sequence">Последовательность для индексации.</param>
12         bool Add(IList<TLink> sequence);
13
14         bool MightContain(IList<TLink> sequence);
15     }
16 }

```

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

```
1 using System.Collections.Generic;
2
3 namespace Platform.Data.Doublets.Sequences.Indexes
4 {
5     public class SequenceIndex<TLink> : LinksOperatorBase<TLink>, ISequenceIndex<TLink>
6     {
7         private static readonly EqualityComparer<TLink> _equalityComparer =
8             ↳ EqualityComparer<TLink>.Default;
9
10        public SequenceIndex(ILinks<TLink> links) : base(links) { }
11
12        public virtual bool Add(IList<TLink> sequence)
13        {
14            var indexed = true;
15            var i = sequence.Count;
16            while (--i >= 1 && (indexed =
17                ↳ !_equalityComparer.Equals(Links.SearchOrDefault(sequence[i - 1], sequence[i]),
18                ↳ default))) { }
19            for (; i >= 1; i--)
20            {
21                Links.GetOrCreate(sequence[i - 1], sequence[i]);
22            }
23            return indexed;
24        }
25
26        public virtual bool MightContain(IList<TLink> sequence)
27        {
28            var indexed = true;
29            var i = sequence.Count;
30            while (--i >= 1 && (indexed =
31                ↳ !_equalityComparer.Equals(Links.SearchOrDefault(sequence[i - 1], sequence[i]),
32                ↳ default))) { }
33            return indexed;
34        }
35    }
36 }
```

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

```
1 using System.Collections.Generic;
2
3 namespace Platform.Data.Doublets.Sequences.Indexes
4 {
5     public class SynchronizedSequenceIndex<TLink> : ISequenceIndex<TLink>
6     {
7         private static readonly EqualityComparer<TLink> _equalityComparer =
8             ↳ EqualityComparer<TLink>.Default;
9
10        private readonly ISynchronizedLinks<TLink> _links;
11
12        public SynchronizedSequenceIndex(ISynchronizedLinks<TLink> links) => _links = links;
13
14        public bool Add(IList<TLink> sequence)
15        {
16            var indexed = true;
17            var i = sequence.Count;
18            var links = _links.Unsync;
19            _links.SyncRoot.ExecuteReadOperation(() =>
20            {
21                while (--i >= 1 && (indexed =
22                    ↳ !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1],
23                    ↳ sequence[i]), default))) { }
24            });
25            if (!indexed)
26            {
27                _links.SyncRoot.ExecuteWriteOperation(() =>
28                {
29                    for (; i >= 1; i--)
30                    {
31                        links.GetOrCreate(sequence[i - 1], sequence[i]);
32                    }
33                });
34            }
35            return indexed;
36        }
37
38        public bool MightContain(IList<TLink> sequence)
39        {
40            var links = _links.Unsync;
41            return _links.SyncRoot.ExecuteReadOperation(() =>
```

```

39         {
40             var indexed = true;
41             var i = sequence.Count;
42             while (--i >= 1 && (indexed =
                ↳ !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1],
                ↳ sequence[i]), default))) { }
43             return indexed;
44         });
45     }
46 }
47 }

```

./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.Threading.Synchronization;
8  using Platform.Singletons;
9  using LinkIndex = System.UInt64;
10 using Platform.Data.Constants;
11 using Platform.Data.Sequences;
12 using Platform.Data.Doublets.Sequences.Walkers;
13 using Platform.Collections.Stacks;
14
15 namespace Platform.Data.Doublets.Sequences
16 {
17     /// <summary>
18     /// Представляет коллекцию последовательностей связей.
19     /// </summary>
20     /// <remarks>
21     /// Обязательно реализовать атомарность каждого публичного метода.
22     ///
23     /// TODO:
24     ///
25     /// !!! Повышение вероятности повторного использования групп (подпоследовательностей),
26     /// через естественную группировку по unicode типам, все whitespace вместе, все символы
27     /// ↳ вместе, все числа вместе и т.п.
28     /// + использовать ровно сбалансированный вариант, чтобы уменьшать вложенность (глубину
29     /// ↳ графа)
30     ///
31     /// х*у - найти все связи между, в последовательностях любой формы, если не стоит
32     /// ↳ ограничитель на то, что является последовательностью, а что нет,
33     /// то находятся любые структуры связей, которые содержат эти элементы именно в таком
34     /// ↳ порядке.
35     ///
36     /// Рост последовательности слева и справа.
37     /// Поиск со звёздочкой.
38     /// URL, PURL - реестр используемых во вне ссылок на ресурсы,
39     /// так же проблема может быть решена при реализации дистанционных триггеров.
40     /// Нужны ли уникальные указатели вообще?
41     /// Что если обращение к информации будет происходить через содержимое всегда?
42     ///
43     /// Писать тесты.
44     ///
45     /// 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     /// Блокчейн и/или гит для распределённой записи транзакций.
52     ///
53     /// </remarks>
54     public partial class Sequences : ISequences<ulong> // IList<string>, IList<ulong[]> (после
55     ↳ завершения реализации Sequences)
56     {
57         private static readonly LinksCombinedConstants<bool, ulong, long> _constants =
58         ↳ Default<LinksCombinedConstants<bool, ulong, long>>.Instance;
59
60         /// <summary>Возвращает значение ulong, обозначающее любое количество связей.</summary>
61         public const ulong ZeroOrMany = ulong.MaxValue;
62
63         public SequencesOptions<ulong> Options;
64         public readonly SynchronizedLinks<ulong> Links;
65         public readonly ISynchronization Sync;

```

```

61
62 public Sequences(SynchronizedLinks<ulong> links)
63     : this(links, new SequencesOptions<ulong>())
64 {
65 }
66
67 public Sequences(SynchronizedLinks<ulong> links, SequencesOptions<ulong> options)
68 {
69     Links = links;
70     Sync = links.SyncRoot;
71     Options = options;
72
73     Options.ValidateOptions();
74     Options.InitOptions(Links);
75 }
76
77 public bool IsSequence(ulong sequence)
78 {
79     return Sync.ExecuteReadOperation(() =>
80     {
81         if (Options.UseSequenceMarker)
82         {
83             return Options.MarkedSequenceMatcher.IsMatched(sequence);
84         }
85         return !Links.Unsync.IsPartialPoint(sequence);
86     });
87 }
88
89 [MethodImpl(MethodImplOptions.AggressiveInlining)]
90 private ulong GetSequenceByElements(ulong sequence)
91 {
92     if (Options.UseSequenceMarker)
93     {
94         return Links.SearchOrDefault(Options.SequenceMarkerLink, sequence);
95     }
96     return sequence;
97 }
98
99 private ulong GetSequenceElements(ulong sequence)
100 {
101     if (Options.UseSequenceMarker)
102     {
103         var linkContents = new UInt64Link(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 public ulong Count(params ulong[] sequence)
119 {
120     if (sequence.Length == 0)
121     {
122         return Links.Count(_constants.Any, Options.SequenceMarkerLink, _constants.Any);
123     }
124     if (sequence.Length == 1) // Первая связь это адрес
125     {
126         if (sequence[0] == _constants.Null)
127         {
128             return 0;
129         }
130         if (sequence[0] == _constants.Any)
131         {
132             return Count();
133         }
134         if (Options.UseSequenceMarker)
135         {
136             return Links.Count(_constants.Any, Options.SequenceMarkerLink, sequence[0]);
137         }
138         return Links.Exists(sequence[0]) ? 1UL : 0;
139     }

```

```

140         throw new NotImplementedException();
141     }
142
143     private ulong CountUsages(params ulong[] restrictions)
144     {
145         if (restrictions.Length == 0)
146         {
147             return 0;
148         }
149         if (restrictions.Length == 1) // Первая связь это адрес
150         {
151             if (restrictions[0] == _constants.Null)
152             {
153                 return 0;
154             }
155             if (Options.UseSequenceMarker)
156             {
157                 var elementsLink = GetSequenceElements(restrictions[0]);
158                 var sequenceLink = GetSequenceByElements(elementsLink);
159                 if (sequenceLink != _constants.Null)
160                 {
161                     return Links.Count(sequenceLink) + Links.Count(elementsLink) - 1;
162                 }
163                 return Links.Count(elementsLink);
164             }
165             return Links.Count(restrictions[0]);
166         }
167         throw new NotImplementedException();
168     }
169
170 #endregion
171
172 #region Create
173
174     public ulong Create(params ulong[] sequence)
175     {
176         return Sync.ExecuteWriteOperation(() =>
177         {
178             if (sequence.IsNullOrEmpty())
179             {
180                 return _constants.Null;
181             }
182             Links.EnsureEachLinkExists(sequence);
183             return CreateCore(sequence);
184         });
185     }
186
187     private ulong CreateCore(params ulong[] sequence)
188     {
189         if (Options.UseIndex)
190         {
191             Options.Index.Add(sequence);
192         }
193         var sequenceRoot = default(ulong);
194         if (Options.EnforceSingleSequenceVersionOnWriteBasedOnExisting)
195         {
196             var matches = Each(sequence);
197             if (matches.Count > 0)
198             {
199                 sequenceRoot = matches[0];
200             }
201         }
202         else if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew)
203         {
204             return CompactCore(sequence);
205         }
206         if (sequenceRoot == default)
207         {
208             sequenceRoot = Options.LinksToSequenceConverter.Convert(sequence);
209         }
210         if (Options.UseSequenceMarker)
211         {
212             Links.Unsync.CreateAndUpdate(Options.SequenceMarkerLink, sequenceRoot);
213         }
214         return sequenceRoot; // Возвращаем корень последовательности (т.е. сами элементы)
215     }
216
217 #endregion
218

```

```

219 #region Each
220
221 public List<ulong> Each(params ulong[] sequence)
222 {
223     var results = new List<ulong>();
224     Each(results.AddAndReturnTrue, sequence);
225     return results;
226 }
227
228 public bool Each(Func<ulong, bool> handler, IList<ulong> sequence)
229 {
230     return Sync.ExecuteReadOperation(() =>
231     {
232         if (sequence.IsNullOrEmpty())
233         {
234             return true;
235         }
236         Links.EnsureEachLinkIsAnyOrExists(sequence);
237         if (sequence.Count == 1)
238         {
239             var link = sequence[0];
240             if (link == _constants.Any)
241             {
242                 return Links.Unsync.Each(_constants.Any, _constants.Any, handler);
243             }
244             return handler(link);
245         }
246         if (sequence.Count == 2)
247         {
248             return Links.Unsync.Each(sequence[0], sequence[1], handler);
249         }
250         if (Options.UseIndex && !Options.Index.MightContain(sequence))
251         {
252             return false;
253         }
254         return EachCore(handler, sequence);
255     });
256 }
257
258 private bool EachCore(Func<ulong, bool> handler, IList<ulong> sequence)
259 {
260     var matcher = new Matcher(this, sequence, new HashSet<LinkIndex>(), handler);
261     // TODO: Find out why matcher.HandleFullMatched executed twice for the same sequence
262     ↪ Id.
263     Func<ulong, bool> innerHandler = Options.UseSequenceMarker ? (Func<ulong,
264     ↪ bool>)matcher.HandleFullMatchedSequence : matcher.HandleFullMatched;
265     //if (sequence.Length >= 2)
266     if (!StepRight(innerHandler, sequence[0], sequence[1]))
267     {
268         return false;
269     }
270     var last = sequence.Count - 2;
271     for (var i = 1; i < last; i++)
272     {
273         if (!PartialStepRight(innerHandler, sequence[i], sequence[i + 1]))
274         {
275             return false;
276         }
277     }
278     if (sequence.Count >= 3)
279     {
280         if (!StepLeft(innerHandler, sequence[sequence.Count - 2],
281         ↪ sequence[sequence.Count - 1]))
282         {
283             return false;
284         }
285     }
286     return true;
287 }
288
289 private bool PartialStepRight(Func<ulong, bool> handler, ulong left, ulong right)
290 {
291     return Links.Unsync.Each(_constants.Any, left, doublet =>
292     {
293         if (!StepRight(handler, doublet, right))
294         {
295             return false;
296         }
297     })
298 }

```

```

294         if (left != doublet)
295         {
296             return PartialStepRight(handler, doublet, right);
297         }
298         return true;
299     });
300 }
301
302 private bool StepRight(Func<ulong, bool> handler, ulong left, ulong right) =>
    ↳ Links.Unsync.Each(left, _constants.Any, rightStep => TryStepRightUp(handler, right,
    ↳ rightStep));
303
304 private bool TryStepRightUp(Func<ulong, bool> handler, ulong right, ulong stepFrom)
305 {
306     var upStep = stepFrom;
307     var firstSource = Links.Unsync.GetTarget(upStep);
308     while (firstSource != right && firstSource != upStep)
309     {
310         upStep = firstSource;
311         firstSource = Links.Unsync.GetSource(upStep);
312     }
313     if (firstSource == right)
314     {
315         return handler(stepFrom);
316     }
317     return true;
318 }
319
320 private bool StepLeft(Func<ulong, bool> handler, ulong left, ulong right) =>
    ↳ Links.Unsync.Each(_constants.Any, right, leftStep => TryStepLeftUp(handler, left,
    ↳ leftStep));
321
322 private bool TryStepLeftUp(Func<ulong, bool> handler, ulong left, ulong stepFrom)
323 {
324     var upStep = stepFrom;
325     var firstTarget = Links.Unsync.GetSource(upStep);
326     while (firstTarget != left && firstTarget != upStep)
327     {
328         upStep = firstTarget;
329         firstTarget = Links.Unsync.GetTarget(upStep);
330     }
331     if (firstTarget == left)
332     {
333         return handler(stepFrom);
334     }
335     return true;
336 }
337
338 #endregion
339
340 #region Update
341
342 public ulong Update(ulong[] sequence, ulong[] newSequence)
343 {
344     if (sequence.IsNullOrEmpty() && newSequence.IsNullOrEmpty())
345     {
346         return _constants.Null;
347     }
348     if (sequence.IsNullOrEmpty())
349     {
350         return Create(newSequence);
351     }
352     if (newSequence.IsNullOrEmpty())
353     {
354         Delete(sequence);
355         return _constants.Null;
356     }
357     return Sync.ExecuteWriteOperation(() =>
358     {
359         Links.EnsureEachLinkIsAnyOrExists(sequence);
360         Links.EnsureEachLinkExists(newSequence);
361         return UpdateCore(sequence, newSequence);
362     });
363 }
364
365 private ulong UpdateCore(ulong[] sequence, ulong[] newSequence)
366 {
367     ulong bestVariant;

```



```

368     if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew &&
369         ↪ !sequence.EqualTo(newSequence))
370     {
371         bestVariant = CompactCore(newSequence);
372     }
373     else
374     {
375         bestVariant = CreateCore(newSequence);
376     }
377     // TODO: Check all options only ones before loop execution
378     // Возможно нужно две версии Each, возвращающий фактические последовательности и с
379     ↪ маркером,
380     // или возможно даже возвращать и тот и тот вариант. С другой стороны все варианты
381     ↪ можно получить имея только фактические последовательности.
382     foreach (var variant in Each(sequence))
383     {
384         if (variant != bestVariant)
385         {
386             UpdateOneCore(variant, bestVariant);
387         }
388     }
389     return bestVariant;
390 }
391
392 private void UpdateOneCore(ulong sequence, ulong newSequence)
393 {
394     if (Options.UseGarbageCollection)
395     {
396         var sequenceElements = GetSequenceElements(sequence);
397         var sequenceElementsContents = new UInt64Link(Links.GetLink(sequenceElements));
398         var sequenceLink = GetSequenceByElements(sequenceElements);
399         var newSequenceElements = GetSequenceElements(newSequence);
400         var newSequenceLink = GetSequenceByElements(newSequenceElements);
401         if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
402         {
403             if (sequenceLink != _constants.Null)
404             {
405                 Links.Unsync.MergeUsages(sequenceLink, newSequenceLink);
406             }
407             Links.Unsync.MergeUsages(sequenceElements, newSequenceElements);
408         }
409         ClearGarbage(sequenceElementsContents.Source);
410         ClearGarbage(sequenceElementsContents.Target);
411     }
412     else
413     {
414         if (Options.UseSequenceMarker)
415         {
416             var sequenceElements = GetSequenceElements(sequence);
417             var sequenceLink = GetSequenceByElements(sequenceElements);
418             var newSequenceElements = GetSequenceElements(newSequence);
419             var newSequenceLink = GetSequenceByElements(newSequenceElements);
420             if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
421             {
422                 if (sequenceLink != _constants.Null)
423                 {
424                     Links.Unsync.MergeUsages(sequenceLink, newSequenceLink);
425                 }
426                 Links.Unsync.MergeUsages(sequenceElements, newSequenceElements);
427             }
428         }
429         else
430         {
431             if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
432             {
433                 Links.Unsync.MergeUsages(sequence, newSequence);
434             }
435         }
436     }
437 }
438
439 #endregion
440
441 #region Delete
442
443 public void Delete(params ulong[] sequence)
444 {
445     Sync.ExecuteWriteOperation(() =>

```

```

443     {
444         // TODO: Check all options only ones before loop execution
445         foreach (var linkToDelete in Each(sequence))
446         {
447             DeleteOneCore(linkToDelete);
448         }
449     });
450 }
451
452 private void DeleteOneCore(ulong link)
453 {
454     if (Options.UseGarbageCollection)
455     {
456         var sequenceElements = GetSequenceElements(link);
457         var sequenceElementsContents = new UInt64Link(Links.GetLink(sequenceElements));
458         var sequenceLink = GetSequenceByElements(sequenceElements);
459         if (Options.UseCascadeDelete || CountUsages(link) == 0)
460         {
461             if (sequenceLink != _constants.Null)
462             {
463                 Links.Unsync.Delete(sequenceLink);
464             }
465             Links.Unsync.Delete(link);
466         }
467         ClearGarbage(sequenceElementsContents.Source);
468         ClearGarbage(sequenceElementsContents.Target);
469     }
470     else
471     {
472         if (Options.UseSequenceMarker)
473         {
474             var sequenceElements = GetSequenceElements(link);
475             var sequenceLink = GetSequenceByElements(sequenceElements);
476             if (Options.UseCascadeDelete || CountUsages(link) == 0)
477             {
478                 if (sequenceLink != _constants.Null)
479                 {
480                     Links.Unsync.Delete(sequenceLink);
481                 }
482                 Links.Unsync.Delete(link);
483             }
484         }
485         else
486         {
487             if (Options.UseCascadeDelete || CountUsages(link) == 0)
488             {
489                 Links.Unsync.Delete(link);
490             }
491         }
492     }
493 }
494
495 #endregion
496
497 #region Compactification
498
499 /// <remarks>
500 /// bestVariant можно выбирать по максимальному числу использований,
501 /// но балансированный позволяет гарантировать уникальность (если есть возможность,
502 /// гарантировать его использование в других местах).
503 ///
504 /// Получается этот метод должен игнорировать Options.EnforceSingleSequenceVersionOnWrite
505 /// </remarks>
506 public ulong Compact(params ulong[] sequence)
507 {
508     return Sync.ExecuteWriteOperation(() =>
509     {
510         if (sequence.IsNullOrEmpty())
511         {
512             return _constants.Null;
513         }
514         Links.EnsureEachLinkExists(sequence);
515         return CompactCore(sequence);
516     });
517 }
518
519 [MethodImpl(MethodImplOptions.AggressiveInlining)]
520 private ulong CompactCore(params ulong[] sequence) => UpdateCore(sequence, sequence);
521

```

```

522 #endregion
523
524 #region Garbage Collection
525
526 /// <remarks>
527 /// TODO: Добавить дополнительный обработчик / событие CanBeDeleted которое можно
528 /// ↳ определить извне или в унаследованном классе
529 /// </remarks>
530 [MethodImpl(MethodImplOptions.AggressiveInlining)]
531 private bool IsGarbage(ulong link) => link != Options.SequenceMarkerLink &&
532 ↳ !Links.Unsync.IsPartialPoint(link) && Links.Count(link) == 0;
533
534 private void ClearGarbage(ulong link)
535 {
536     if (IsGarbage(link))
537     {
538         var contents = new UInt64Link(Links.GetLink(link));
539         Links.Unsync.Delete(link);
540         ClearGarbage(contents.Source);
541         ClearGarbage(contents.Target);
542     }
543 }
544
545 #endregion
546
547 #region Walkers
548
549 public bool EachPart(Func<ulong, bool> handler, ulong sequence)
550 {
551     return Sync.ExecuteReadOperation(() =>
552     {
553         var links = Links.Unsync;
554         foreach (var part in Options.Walker.Walk(sequence))
555         {
556             if (!handler(part))
557             {
558                 return false;
559             }
560         }
561         return true;
562     });
563 }
564
565 public class Matcher : RightSequenceWalker<ulong>
566 {
567     private readonly Sequences _sequences;
568     private readonly IList<LinkIndex> _patternSequence;
569     private readonly HashSet<LinkIndex> _linksInSequence;
570     private readonly HashSet<LinkIndex> _results;
571     private readonly Func<ulong, bool> _stopableHandler;
572     private readonly HashSet<ulong> _readAsElements;
573     private int _filterPosition;
574
575     public Matcher(Sequences sequences, IList<LinkIndex> patternSequence,
576 ↳ HashSet<LinkIndex> results, Func<LinkIndex, bool> stopableHandler,
577 ↳ HashSet<LinkIndex> readAsElements = null)
578     : base(sequences.Links.Unsync, new DefaultStack<ulong>())
579     {
580         _sequences = sequences;
581         _patternSequence = patternSequence;
582         _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
583 ↳ _constants.Any && x != ZeroOrMany));
584         _results = results;
585         _stopableHandler = stopableHandler;
586         _readAsElements = readAsElements;
587     }
588
589     protected override bool IsElement(ulong link) => base.IsElement(link) ||
590 ↳ (_readAsElements != null && _readAsElements.Contains(link)) ||
591 ↳ _linksInSequence.Contains(link);
592
593     public bool FullMatch(LinkIndex sequenceToMatch)
594     {
595         _filterPosition = 0;
596         foreach (var part in Walk(sequenceToMatch))
597         {
598             if (!FullMatchCore(part))
599             {
600                 break;
601             }
602         }
603     }

```

```

595     }
596     return _filterPosition == _patternSequence.Count;
597 }
598
599 private bool FullMatchCore(LinkIndex element)
600 {
601     if (_filterPosition == _patternSequence.Count)
602     {
603         _filterPosition = -2; // Длиннее чем нужно
604         return false;
605     }
606     if (_patternSequence[_filterPosition] != _constants.Any
607         && element != _patternSequence[_filterPosition])
608     {
609         _filterPosition = -1;
610         return false; // Начинается/Продолжается иначе
611     }
612     _filterPosition++;
613     return true;
614 }
615
616 public void AddFullMatchedToResults(ulong sequenceToMatch)
617 {
618     if (FullMatch(sequenceToMatch))
619     {
620         _results.Add(sequenceToMatch);
621     }
622 }
623
624 public bool HandleFullMatched(ulong sequenceToMatch)
625 {
626     if (FullMatch(sequenceToMatch) && _results.Add(sequenceToMatch))
627     {
628         return _stopableHandler(sequenceToMatch);
629     }
630     return true;
631 }
632
633 public bool HandleFullMatchedSequence(ulong sequenceToMatch)
634 {
635     var sequence = _sequences.GetSequenceByElements(sequenceToMatch);
636     if (sequence != _constants.Null && FullMatch(sequenceToMatch) &&
637         ↪ _results.Add(sequenceToMatch))
638     {
639         return _stopableHandler(sequence);
640     }
641     return true;
642 }
643
644 /// <remarks>
645 /// TODO: Add support for LinksConstants.Any
646 /// </remarks>
647 public bool PartialMatch(LinkIndex sequenceToMatch)
648 {
649     _filterPosition = -1;
650     foreach (var part in Walk(sequenceToMatch))
651     {
652         if (!PartialMatchCore(part))
653         {
654             break;
655         }
656     }
657     return _filterPosition == _patternSequence.Count - 1;
658 }
659
660 private bool PartialMatchCore(LinkIndex element)
661 {
662     if (_filterPosition == (_patternSequence.Count - 1))
663     {
664         return false; // Нашлось
665     }
666     if (_filterPosition >= 0)
667     {
668         if (element == _patternSequence[_filterPosition + 1])
669         {
670             _filterPosition++;
671         }
672         else
673         {

```

```

673         _filterPosition = -1;
674     }
675 }
676 if (_filterPosition < 0)
677 {
678     if (element == _patternSequence[0])
679     {
680         _filterPosition = 0;
681     }
682 }
683 return true; // Ищем дальше
684 }
685
686 public void AddPartialMatchedToResults(ulong sequenceToMatch)
687 {
688     if (PartialMatch(sequenceToMatch))
689     {
690         _results.Add(sequenceToMatch);
691     }
692 }
693
694 public bool HandlePartialMatched(ulong sequenceToMatch)
695 {
696     if (PartialMatch(sequenceToMatch))
697     {
698         return _stopableHandler(sequenceToMatch);
699     }
700     return true;
701 }
702
703 public void AddAllPartialMatchedToResults(IEnumerable<ulong> sequencesToMatch)
704 {
705     foreach (var sequenceToMatch in sequencesToMatch)
706     {
707         if (PartialMatch(sequenceToMatch))
708         {
709             _results.Add(sequenceToMatch);
710         }
711     }
712 }
713
714 public void AddAllPartialMatchedToResultsAndReadAsElements(IEnumerable<ulong>
↵ sequencesToMatch)
715 {
716     foreach (var sequenceToMatch in sequencesToMatch)
717     {
718         if (PartialMatch(sequenceToMatch))
719         {
720             _readAsElements.Add(sequenceToMatch);
721             _results.Add(sequenceToMatch);
722         }
723     }
724 }
725 }
726 #endregion
727 }
728 }
729 }

```

# ./Platform.Data.Doublets/Sequences/Sequences.Experiments.cs

```

1  using System;
2  using LinkIndex = System.UInt64;
3  using System.Collections.Generic;
4  using Stack = System.Collections.Generic.Stack<ulong>;
5  using System.Linq;
6  using System.Text;
7  using Platform.Collections;
8  using Platform.Data.Exceptions;
9  using Platform.Data.Sequences;
10 using Platform.Data.Doublets.Sequences.Frequencies.Counters;
11 using Platform.Data.Doublets.Sequences.Walkers;
12 using Platform.Collections.Stacks;
13
14 namespace Platform.Data.Doublets.Sequences
15 {
16     partial class Sequences
17     {
18         #region Create All Variants (Not Practical)
19
20         /// <remarks>

```

```

21  /// Number of links that is needed to generate all variants for
22  /// sequence of length N corresponds to https://oeis.org/A014143/list sequence.
23  /// </remarks>
24  public ulong[] CreateAllVariants2(ulong[] sequence)
25  {
26      return Sync.ExecuteWriteOperation(() =>
27      {
28          if (sequence.IsNullOrEmpty())
29          {
30              return new ulong[0];
31          }
32          Links.EnsureEachLinkExists(sequence);
33          if (sequence.Length == 1)
34          {
35              return sequence;
36          }
37          return CreateAllVariants2Core(sequence, 0, sequence.Length - 1);
38      });
39  }
40
41  private ulong[] CreateAllVariants2Core(ulong[] sequence, long startAt, long stopAt)
42  {
43      #if DEBUG
44          if ((stopAt - startAt) < 0)
45          {
46              throw new ArgumentOutOfRangeException(nameof(startAt), "startAt должен быть
47                  ↪ меньше или равен stopAt");
48          }
49          #endif
50          if ((stopAt - startAt) == 0)
51          {
52              return new[] { sequence[startAt] };
53          }
54          if ((stopAt - startAt) == 1)
55          {
56              return new[] { Links.Unsync.CreateAndUpdate(sequence[startAt], sequence[stopAt])
57                  ↪ };
58          }
59          var variants = new ulong[(ulong)Numbers.Math.Catalan(stopAt - startAt)];
60          var last = 0;
61          for (var splitter = startAt; splitter < stopAt; splitter++)
62          {
63              var left = CreateAllVariants2Core(sequence, startAt, splitter);
64              var right = CreateAllVariants2Core(sequence, splitter + 1, stopAt);
65              for (var i = 0; i < left.Length; i++)
66              {
67                  for (var j = 0; j < right.Length; j++)
68                  {
69                      var variant = Links.Unsync.CreateAndUpdate(left[i], right[j]);
70                      if (variant == _constants.Null)
71                      {
72                          throw new NotImplementedException("Creation cancellation is not
73                              ↪ implemented.");
74                      }
75                      variants[last++] = variant;
76                  }
77              }
78          }
79          return variants;
80      }
81
82  public List<ulong> CreateAllVariants1(params ulong[] sequence)
83  {
84      return Sync.ExecuteWriteOperation(() =>
85      {
86          if (sequence.IsNullOrEmpty())
87          {
88              return new List<ulong>();
89          }
90          Links.Unsync.EnsureEachLinkExists(sequence);
91          if (sequence.Length == 1)
92          {
93              return new List<ulong> { sequence[0] };
94          }
95          var results = new List<ulong>((int)Numbers.Math.Catalan(sequence.Length));
96          return CreateAllVariants1Core(sequence, results);
97      });
98  }
99

```

```

96
97 private List<ulong> CreateAllVariants1Core(ulong[] sequence, List<ulong> results)
98 {
99     if (sequence.Length == 2)
100     {
101         var link = Links.Unsync.CreateAndUpdate(sequence[0], sequence[1]);
102         if (link == _constants.Null)
103         {
104             throw new NotImplementedException("Creation cancellation is not
105                 ↳ implemented.");
106         }
107         results.Add(link);
108         return results;
109     }
110     var innerSequenceLength = sequence.Length - 1;
111     var innerSequence = new ulong[innerSequenceLength];
112     for (var li = 0; li < innerSequenceLength; li++)
113     {
114         var link = Links.Unsync.CreateAndUpdate(sequence[li], sequence[li + 1]);
115         if (link == _constants.Null)
116         {
117             throw new NotImplementedException("Creation cancellation is not
118                 ↳ implemented.");
119         }
120         for (var isi = 0; isi < li; isi++)
121         {
122             innerSequence[isi] = sequence[isi];
123         }
124         innerSequence[li] = link;
125         for (var isi = li + 1; isi < innerSequenceLength; isi++)
126         {
127             innerSequence[isi] = sequence[isi + 1];
128         }
129         CreateAllVariants1Core(innerSequence, results);
130     }
131     return results;
132 }
133
134 #endregion
135
136 public HashSet<ulong> Each1(params ulong[] sequence)
137 {
138     var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
139     Each1(link =>
140     {
141         if (!visitedLinks.Contains(link))
142         {
143             visitedLinks.Add(link); // изучить почему случаются повторы
144         }
145         return true;
146     }, sequence);
147     return visitedLinks;
148 }
149
150 private void Each1(Func<ulong, bool> handler, params ulong[] sequence)
151 {
152     if (sequence.Length == 2)
153     {
154         Links.Unsync.Each(sequence[0], sequence[1], handler);
155     }
156     else
157     {
158         var innerSequenceLength = sequence.Length - 1;
159         for (var li = 0; li < innerSequenceLength; li++)
160         {
161             var left = sequence[li];
162             var right = sequence[li + 1];
163             if (left == 0 && right == 0)
164             {
165                 continue;
166             }
167             var linkIndex = li;
168             ulong[] innerSequence = null;
169             Links.Unsync.Each(left, right, doublet =>
170             {
171                 if (innerSequence == null)
172                 {
173                     innerSequence = new ulong[innerSequenceLength];
174                 }
175             });
176         }
177     }
178 }

```

```

172         for (var isi = 0; isi < linkIndex; isi++)
173         {
174             innerSequence[isi] = sequence[isi];
175         }
176         for (var isi = linkIndex + 1; isi < innerSequenceLength; isi++)
177         {
178             innerSequence[isi] = sequence[isi + 1];
179         }
180     }
181     innerSequence[linkIndex] = doublet;
182     Each1(handler, innerSequence);
183     return _constants.Continue;
184 });
185 }
186 }
187 }
188
189 public HashSet<ulong> EachPart(params ulong[] sequence)
190 {
191     var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
192     EachPartCore(link =>
193     {
194         if (!visitedLinks.Contains(link))
195         {
196             visitedLinks.Add(link); // изучить почему случаются повторы
197         }
198         return true;
199     }, sequence);
200     return visitedLinks;
201 }
202
203 public void EachPart(Func<ulong, bool> handler, params ulong[] sequence)
204 {
205     var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
206     EachPartCore(link =>
207     {
208         if (!visitedLinks.Contains(link))
209         {
210             visitedLinks.Add(link); // изучить почему случаются повторы
211             return handler(link);
212         }
213         return true;
214     }, sequence);
215 }
216
217 private void EachPartCore(Func<ulong, bool> handler, params ulong[] sequence)
218 {
219     if (sequence.IsNullOrEmpty())
220     {
221         return;
222     }
223     Links.EnsureEachLinkIsAnyOrExists(sequence);
224     if (sequence.Length == 1)
225     {
226         var link = sequence[0];
227         if (link > 0)
228         {
229             handler(link);
230         }
231         else
232         {
233             Links.Each(_constants.Any, _constants.Any, handler);
234         }
235     }
236     else if (sequence.Length == 2)
237     {
238         // _links.Each(sequence[0], sequence[1], handler);
239         //   o_ |      x_o ...
240         // x_ |      |__|
241         Links.Each(sequence[1], _constants.Any, doublet =>
242         {
243             var match = Links.SearchOrDefault(sequence[0], doublet);
244             if (match != _constants.Null)
245             {
246                 handler(match);
247             }
248             return true;
249         });

```



```

250 // |_x      ... x_o
251 // |_o      |__|
252 Links.Each(_constants.Any, sequence[0], doublet =>
253 {
254     var match = Links.SearchOrDefault(doublet, sequence[1]);
255     if (match != 0)
256     {
257         handler(match);
258     }
259     return true;
260 });
261 //      . _x o _ .
262 //      |__|
263 PartialStepRight(x => handler(x), sequence[0], sequence[1]);
264 }
265 else
266 {
267     // TODO: Implement other variants
268     return;
269 }
270 }
271
272 private void PartialStepRight(Action<ulong> handler, ulong left, ulong right)
273 {
274     Links.Unsync.Each(_constants.Any, left, doublet =>
275     {
276         StepRight(handler, doublet, right);
277         if (left != doublet)
278         {
279             PartialStepRight(handler, doublet, right);
280         }
281         return true;
282     });
283 }
284
285 private void StepRight(Action<ulong> handler, ulong left, ulong right)
286 {
287     Links.Unsync.Each(left, _constants.Any, rightStep =>
288     {
289         TryStepRightUp(handler, right, rightStep);
290         return true;
291     });
292 }
293
294 private void TryStepRightUp(Action<ulong> handler, ulong right, ulong stepFrom)
295 {
296     var upStep = stepFrom;
297     var firstSource = Links.Unsync.GetTarget(upStep);
298     while (firstSource != right && firstSource != upStep)
299     {
300         upStep = firstSource;
301         firstSource = Links.Unsync.GetSource(upStep);
302     }
303     if (firstSource == right)
304     {
305         handler(stepFrom);
306     }
307 }
308
309 // TODO: Test
310 private void PartialStepLeft(Action<ulong> handler, ulong left, ulong right)
311 {
312     Links.Unsync.Each(right, _constants.Any, doublet =>
313     {
314         StepLeft(handler, left, doublet);
315         if (right != doublet)
316         {
317             PartialStepLeft(handler, left, doublet);
318         }
319         return true;
320     });
321 }
322
323 private void StepLeft(Action<ulong> handler, ulong left, ulong right)
324 {
325     Links.Unsync.Each(_constants.Any, right, leftStep =>
326     {
327         TryStepLeftUp(handler, left, leftStep);
328         return true;

```

```

329     });
330 }
331
332 private void TryStepLeftUp(Action<ulong> handler, ulong left, ulong stepFrom)
333 {
334     var upStep = stepFrom;
335     var firstTarget = Links.Unsync.GetSource(upStep);
336     while (firstTarget != left && firstTarget != upStep)
337     {
338         upStep = firstTarget;
339         firstTarget = Links.Unsync.GetTarget(upStep);
340     }
341     if (firstTarget == left)
342     {
343         handler(stepFrom);
344     }
345 }
346
347 private bool StartsWith(ulong sequence, ulong link)
348 {
349     var upStep = sequence;
350     var firstSource = Links.Unsync.GetSource(upStep);
351     while (firstSource != link && firstSource != upStep)
352     {
353         upStep = firstSource;
354         firstSource = Links.Unsync.GetSource(upStep);
355     }
356     return firstSource == link;
357 }
358
359 private bool EndsWith(ulong sequence, ulong link)
360 {
361     var upStep = sequence;
362     var lastTarget = Links.Unsync.GetTarget(upStep);
363     while (lastTarget != link && lastTarget != upStep)
364     {
365         upStep = lastTarget;
366         lastTarget = Links.Unsync.GetTarget(upStep);
367     }
368     return lastTarget == link;
369 }
370
371 public List<ulong> GetAllMatchingSequences0(params ulong[] sequence)
372 {
373     return Sync.ExecuteReadOperation(() =>
374     {
375         var results = new List<ulong>();
376         if (sequence.Length > 0)
377         {
378             Links.EnsureEachLinkExists(sequence);
379             var firstElement = sequence[0];
380             if (sequence.Length == 1)
381             {
382                 results.Add(firstElement);
383                 return results;
384             }
385             if (sequence.Length == 2)
386             {
387                 var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
388                 if (doublet != _constants.Null)
389                 {
390                     results.Add(doublet);
391                 }
392                 return results;
393             }
394             var linksInSequence = new HashSet<ulong>(sequence);
395             void handler(ulong result)
396             {
397                 var filterPosition = 0;
398                 StoppableSequenceWalker.WalkRight(result, Links.Unsync.GetSource,
399                     ↪ Links.Unsync.GetTarget,
400                     x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x,
401                     ↪ x =>
402                     {
403                         if (filterPosition == sequence.Length)
404                         {
405                             filterPosition = -2; // Длиннее чем нужно
406                             return false;
407                         }
408                     }
409                 )
410             }
411         }
412     });
413 }

```

```

406         if (x != sequence[filterPosition])
407         {
408             filterPosition = -1;
409             return false; // Начинается иначе
410         }
411         filterPosition++;
412         return true;
413     });
414     if (filterPosition == sequence.Length)
415     {
416         results.Add(result);
417     }
418 }
419 if (sequence.Length >= 2)
420 {
421     StepRight(handler, sequence[0], sequence[1]);
422 }
423 var last = sequence.Length - 2;
424 for (var i = 1; i < last; i++)
425 {
426     PartialStepRight(handler, sequence[i], sequence[i + 1]);
427 }
428 if (sequence.Length >= 3)
429 {
430     StepLeft(handler, sequence[sequence.Length - 2],
431         ↪ sequence[sequence.Length - 1]);
432 }
433 }
434 return results;
435 });
436 }
437
438 public HashSet<ulong> GetAllMatchingSequences1(params ulong[] sequence)
439 {
440     return Sync.ExecuteReadOperation(() =>
441     {
442         var results = new HashSet<ulong>();
443         if (sequence.Length > 0)
444         {
445             Links.EnsureEachLinkExists(sequence);
446             var firstElement = sequence[0];
447             if (sequence.Length == 1)
448             {
449                 results.Add(firstElement);
450                 return results;
451             }
452             if (sequence.Length == 2)
453             {
454                 var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
455                 if (doublet != _constants.Null)
456                 {
457                     results.Add(doublet);
458                 }
459                 return results;
460             }
461             var matcher = new Matcher(this, sequence, results, null);
462             if (sequence.Length >= 2)
463             {
464                 StepRight(matcher.AddFullMatchedToResults, sequence[0], sequence[1]);
465             }
466             var last = sequence.Length - 2;
467             for (var i = 1; i < last; i++)
468             {
469                 PartialStepRight(matcher.AddFullMatchedToResults, sequence[i],
470                     ↪ sequence[i + 1]);
471             }
472             if (sequence.Length >= 3)
473             {
474                 StepLeft(matcher.AddFullMatchedToResults, sequence[sequence.Length - 2],
475                     ↪ sequence[sequence.Length - 1]);
476             }
477         }
478         return results;
479     });
480 }
481
482 public const int MaxSequenceFormatSize = 200;

```

```

481 public string FormatSequence(LinkIndex sequenceLink, params LinkIndex[] knownElements)
482     => FormatSequence(sequenceLink, (sb, x) => sb.Append(x), true, knownElements);
483
484 public string FormatSequence(LinkIndex sequenceLink, Action<StringBuilder, LinkIndex>
    elementToString, bool insertComma, params LinkIndex[] knownElements) =>
    Links.SyncRoot.ExecuteReadOperation(() => FormatSequence(Links.Unsync, sequenceLink,
    elementToString, insertComma, knownElements));
485
486 private string FormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
    Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
    LinkIndex[] knownElements)
487 {
488     var linksInSequence = new HashSet<ulong>(knownElements);
489     //var entered = new HashSet<ulong>();
490     var sb = new StringBuilder();
491     sb.Append('{');
492     if (links.Exists(sequenceLink))
493     {
494         StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
495             x => linksInSequence.Contains(x) || links.IsPartialPoint(x), element => //
            entered.AddAndReturnVoid, x => { }, entered.DoNotContains
            {
496                 if (insertComma && sb.Length > 1)
497                 {
498                     sb.Append(',');
499                 }
500                 //if (entered.Contains(element))
501                 //{
502                 //    sb.Append('{');
503                 //    elementToString(sb, element);
504                 //    sb.Append('}');
505                 //}
506                 //else
507                 elementToString(sb, element);
508                 if (sb.Length < MaxSequenceFormatSize)
509                 {
510                     return true;
511                 }
512                 sb.Append(insertComma ? ", ..." : "...");
513                 return false;
514             });
515     }
516     sb.Append('}');
517     return sb.ToString();
518 }
519
520 public string SafeFormatSequence(LinkIndex sequenceLink, params LinkIndex[]
    knownElements) => SafeFormatSequence(sequenceLink, (sb, x) => sb.Append(x), true,
    knownElements);
521
522 public string SafeFormatSequence(LinkIndex sequenceLink, Action<StringBuilder,
    LinkIndex> elementToString, bool insertComma, params LinkIndex[] knownElements) =>
    Links.SyncRoot.ExecuteReadOperation(() => SafeFormatSequence(Links.Unsync,
    sequenceLink, elementToString, insertComma, knownElements));
523
524 private string SafeFormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
    Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
    LinkIndex[] knownElements)
525 {
526     var linksInSequence = new HashSet<ulong>(knownElements);
527     var entered = new HashSet<ulong>();
528     var sb = new StringBuilder();
529     sb.Append('{');
530     if (links.Exists(sequenceLink))
531     {
532         StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
533             x => linksInSequence.Contains(x) || links.IsFullPoint(x),
534             entered.AddAndReturnVoid, x => { }, entered.DoNotContains, element =>
            {
535                 if (insertComma && sb.Length > 1)
536                 {
537                     sb.Append(',');
538                 }
539                 if (entered.Contains(element))
540                 {
541                     sb.Append('{');
542

```

```

543         elementToString(sb, element);
544         sb.Append('}');
545     }
546     else
547     {
548         elementToString(sb, element);
549     }
550     if (sb.Length < MaxSequenceFormatSize)
551     {
552         return true;
553     }
554     sb.Append(insertComma ? ", ..." : "...");
555     return false;
556 });
557 }
558 sb.Append('}');
559 return sb.ToString();
560 }
561
562 public List<ulong> GetAllPartiallyMatchingSequences0(params ulong[] sequence)
563 {
564     return Sync.ExecuteReadOperation(() =>
565     {
566         if (sequence.Length > 0)
567         {
568             Links.EnsureEachLinkExists(sequence);
569             var results = new HashSet<ulong>();
570             for (var i = 0; i < sequence.Length; i++)
571             {
572                 AllUsagesCore(sequence[i], results);
573             }
574             var filteredResults = new List<ulong>();
575             var linksInSequence = new HashSet<ulong>(sequence);
576             foreach (var result in results)
577             {
578                 var filterPosition = -1;
579                 StopableSequenceWalker.WalkRight(result, Links.Unsync.GetSource,
580                     ↪ Links.Unsync.GetTarget,
581                     ↪ x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x,
582                     ↪ x =>
583                     {
584                         if (filterPosition == (sequence.Length - 1))
585                         {
586                             return false;
587                         }
588                         if (filterPosition >= 0)
589                         {
590                             if (x == sequence[filterPosition + 1])
591                             {
592                                 filterPosition++;
593                             }
594                             else
595                             {
596                                 return false;
597                             }
598                         }
599                         if (filterPosition < 0)
600                         {
601                             if (x == sequence[0])
602                             {
603                                 filterPosition = 0;
604                             }
605                         }
606                         return true;
607                     }
608                 );
609                 if (filterPosition == (sequence.Length - 1))
610                 {
611                     filteredResults.Add(result);
612                 }
613             }
614             return filteredResults;
615         }
616         return new List<ulong>();
617     });
618 }
619
620 public HashSet<ulong> GetAllPartiallyMatchingSequences1(params ulong[] sequence)
621 {
622     return Sync.ExecuteReadOperation(() =>

```

```

620     {
621         if (sequence.Length > 0)
622         {
623             Links.EnsureEachLinkExists(sequence);
624             var results = new HashSet<ulong>();
625             for (var i = 0; i < sequence.Length; i++)
626             {
627                 AllUsagesCore(sequence[i], results);
628             }
629             var filteredResults = new HashSet<ulong>();
630             var matcher = new Matcher(this, sequence, filteredResults, null);
631             matcher.AddAllPartialMatchedToResults(results);
632             return filteredResults;
633         }
634         return new HashSet<ulong>();
635     });
636 }
637
638 public bool GetAllPartiallyMatchingSequences2(Func<ulong, bool> handler, params ulong[]
639     ↪ sequence)
640 {
641     return Sync.ExecuteReadOperation(() =>
642     {
643         if (sequence.Length > 0)
644         {
645             Links.EnsureEachLinkExists(sequence);
646
647             var results = new HashSet<ulong>();
648             var filteredResults = new HashSet<ulong>();
649             var matcher = new Matcher(this, sequence, filteredResults, handler);
650             for (var i = 0; i < sequence.Length; i++)
651             {
652                 if (!AllUsagesCore1(sequence[i], results, matcher.HandlePartialMatched))
653                 {
654                     return false;
655                 }
656             }
657             return true;
658         }
659         return true;
660     });
661 }
662
663 //public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
664 //{
665 //    return Sync.ExecuteReadOperation(() =>
666 //    {
667 //        if (sequence.Length > 0)
668 //        {
669 //            _links.EnsureEachLinkIsAnyOrExists(sequence);
670
671 //            var firstResults = new HashSet<ulong>();
672 //            var lastResults = new HashSet<ulong>();
673
674 //            var first = sequence.First(x => x != LinksConstants.Any);
675 //            var last = sequence.Last(x => x != LinksConstants.Any);
676
677 //            AllUsagesCore(first, firstResults);
678 //            AllUsagesCore(last, lastResults);
679
680 //            firstResults.IntersectWith(lastResults);
681
682 //            //for (var i = 0; i < sequence.Length; i++)
683 //            //    AllUsagesCore(sequence[i], results);
684
685 //            var filteredResults = new HashSet<ulong>();
686 //            var matcher = new Matcher(this, sequence, filteredResults, null);
687 //            matcher.AddAllPartialMatchedToResults(firstResults);
688 //            return filteredResults;
689 //        }
690
691 //        return new HashSet<ulong>();
692 //    });
693 //}
694
695 public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
696 {
697     return Sync.ExecuteReadOperation(() =>
698     {

```

```

698     if (sequence.Length > 0)
699     {
700         Links.EnsureEachLinkIsAnyOrExists(sequence);
701         var firstResults = new HashSet<ulong>();
702         var lastResults = new HashSet<ulong>();
703         var first = sequence.First(x => x != _constants.Any);
704         var last = sequence.Last(x => x != _constants.Any);
705         AllUsagesCore(first, firstResults);
706         AllUsagesCore(last, lastResults);
707         firstResults.IntersectWith(lastResults);
708         //for (var i = 0; i < sequence.Length; i++)
709         //    AllUsagesCore(sequence[i], results);
710         var filteredResults = new HashSet<ulong>();
711         var matcher = new Matcher(this, sequence, filteredResults, null);
712         matcher.AddAllPartialMatchedToResults(firstResults);
713         return filteredResults;
714     }
715     return new HashSet<ulong>();
716 });
717 }
718
719 public HashSet<ulong> GetAllPartiallyMatchingSequences4(HashSet<ulong> readAsElements,
720     ↳ IList<ulong> sequence)
721 {
722     return Sync.ExecuteReadOperation(() =>
723     {
724         if (sequence.Count > 0)
725         {
726             Links.EnsureEachLinkExists(sequence);
727             var results = new HashSet<LinkIndex>();
728             //var nextResults = new HashSet<ulong>();
729             //for (var i = 0; i < sequence.Length; i++)
730             //{
731             //    AllUsagesCore(sequence[i], nextResults);
732             //    if (results.IsNullOrEmpty())
733             //    {
734             //        results = nextResults;
735             //        nextResults = new HashSet<ulong>();
736             //    }
737             //    else
738             //    {
739             //        results.IntersectWith(nextResults);
740             //        nextResults.Clear();
741             //    }
742             //}
743             var collector1 = new AllUsagesCollector1(Links.Unsync, results);
744             collector1.Collect(Links.Unsync.GetLink(sequence[0]));
745             var next = new HashSet<ulong>();
746             for (var i = 1; i < sequence.Count; i++)
747             {
748                 var collector = new AllUsagesCollector1(Links.Unsync, next);
749                 collector.Collect(Links.Unsync.GetLink(sequence[i]));
750
751                 results.IntersectWith(next);
752                 next.Clear();
753             }
754             var filteredResults = new HashSet<ulong>();
755             var matcher = new Matcher(this, sequence, filteredResults, null,
756                 ↳ readAsElements);
757             matcher.AddAllPartialMatchedToResultsAndReadAsElements(results.OrderBy(x =>
758                 ↳ x)); // OrderBy is a Hack
759             return filteredResults;
760         }
761         return new HashSet<ulong>();
762     });
763 }
764
765 // Does not work
766 public HashSet<ulong> GetAllPartiallyMatchingSequences5(HashSet<ulong> readAsElements,
767     ↳ params ulong[] sequence)
768 {
769     var visited = new HashSet<ulong>();
770     var results = new HashSet<ulong>();
771     var matcher = new Matcher(this, sequence, visited, x => { results.Add(x); return
772         ↳ true; }, readAsElements);
773     var last = sequence.Length - 1;
774     for (var i = 0; i < last; i++)
775     {

```

```

771         PartialStepRight(matcher.PartialMatch, sequence[i], sequence[i + 1]);
772     }
773     return results;
774 }
775
776 public List<ulong> GetAllPartiallyMatchingSequences(params ulong[] sequence)
777 {
778     return Sync.ExecuteReadOperation(() =>
779     {
780         if (sequence.Length > 0)
781         {
782             Links.EnsureEachLinkExists(sequence);
783             //var firstElement = sequence[0];
784             //if (sequence.Length == 1)
785             //{
786                 //    //results.Add(firstElement);
787                 //    return results;
788             //}
789             //if (sequence.Length == 2)
790             //{
791                 //    //var doublet = _links.SearchCore(firstElement, sequence[1]);
792                 //    //if (doublet != Doublets.Links.Null)
793                 //    //    results.Add(doublet);
794                 //    return results;
795             //}
796             //var lastElement = sequence[sequence.Length - 1];
797             //Func<ulong, bool> handler = x =>
798             //{
799                 //    if (StartsWith(x, firstElement) && EndsWith(x, lastElement))
800                     //    results.Add(x);
801                     //    return true;
802             //};
803             //if (sequence.Length >= 2)
804                 //    StepRight(handler, sequence[0], sequence[1]);
805             //var last = sequence.Length - 2;
806             //for (var i = 1; i < last; i++)
807                 //    PartialStepRight(handler, sequence[i], sequence[i + 1]);
808             //if (sequence.Length >= 3)
809                 //    StepLeft(handler, sequence[sequence.Length - 2],
810                     //    sequence[sequence.Length - 1]);
811             //if (sequence.Length == 1)
812             //{
813                 //    throw new NotImplementedException(); // all sequences, containing
814                     //    this element?
815             //}
816             //if (sequence.Length == 2)
817             //{
818                 //    var results = new List<ulong>();
819                 //    PartialStepRight(results.Add, sequence[0], sequence[1]);
820                 //    return results;
821             //}
822             //var matches = new List<List<ulong>>();
823             //var last = sequence.Length - 1;
824             //for (var i = 0; i < last; i++)
825             //{
826                 //    var results = new List<ulong>();
827                 //    //StepRight(results.Add, sequence[i], sequence[i + 1]);
828                 //    PartialStepRight(results.Add, sequence[i], sequence[i + 1]);
829                 //    if (results.Count > 0)
830                     //    matches.Add(results);
831                 //    else
832                     //    return results;
833                 //    if (matches.Count == 2)
834                 //    {
835                     //        var merged = new List<ulong>();
836                     //        for (var j = 0; j < matches[0].Count; j++)
837                         //        for (var k = 0; k < matches[1].Count; k++)
838                             //        CloseInnerConnections(merged.Add, matches[0][j],
839                                 //        matches[1][k]);
840                     //        if (merged.Count > 0)
841                         //        matches = new List<List<ulong>> { merged };
842                     //        else
843                         //        return new List<ulong>();
844                 //    }
845             //}
846             //if (matches.Count > 0)
847             //{

```



```

844         //var usages = new HashSet<ulong>();
845         //for (int i = 0; i < sequence.Length; i++)
846         //{
847             AllUsagesCore(sequence[i], usages);
848         //}
849         //for (int i = 0; i < matches[0].Count; i++)
850         //    AllUsagesCore(matches[0][i], usages);
851         //usages.UnionWith(matches[0]);
852         return usages.ToList();
853     //}
854     var firstLinkUsages = new HashSet<ulong>();
855     AllUsagesCore(sequence[0], firstLinkUsages);
856     firstLinkUsages.Add(sequence[0]);
857     //var previousMatchings = firstLinkUsages.ToList(); //new List<ulong>() {
858     //    sequence[0] }; // or all sequences, containing this element?
859     //return GetAllPartiallyMatchingSequencesCore(sequence, firstLinkUsages,
860     //    1).ToList();
861     var results = new HashSet<ulong>();
862     foreach (var match in GetAllPartiallyMatchingSequencesCore(sequence,
863         firstLinkUsages, 1))
864     {
865         AllUsagesCore(match, results);
866     }
867     return results.ToList();
868 }
869
870 /// <remarks>
871 /// TODO: Может потребоваться ограничение на уровень глубины рекурсии
872 /// </remarks>
873 public HashSet<ulong> AllUsages(ulong link)
874 {
875     return Sync.ExecuteReadOperation(() =>
876     {
877         var usages = new HashSet<ulong>();
878         AllUsagesCore(link, usages);
879         return usages;
880     });
881 }
882
883 // При сборе всех использований (последовательностей) можно сохранять обратный путь к
884 // той связи с которой начинался поиск (STTTSSSTT),
885 // причём достаточно одного бита для хранения перехода влево или вправо
886 private void AllUsagesCore(ulong link, HashSet<ulong> usages)
887 {
888     bool handler(ulong doublet)
889     {
890         if (usages.Add(doublet))
891         {
892             AllUsagesCore(doublet, usages);
893         }
894         return true;
895     }
896     Links.Unsync.Each(link, _constants.Any, handler);
897     Links.Unsync.Each(_constants.Any, link, handler);
898 }
899
900 public HashSet<ulong> AllBottomUsages(ulong link)
901 {
902     return Sync.ExecuteReadOperation(() =>
903     {
904         var visits = new HashSet<ulong>();
905         var usages = new HashSet<ulong>();
906         AllBottomUsagesCore(link, visits, usages);
907         return usages;
908     });
909 }
910
911 private void AllBottomUsagesCore(ulong link, HashSet<ulong> visits, HashSet<ulong>
912     usages)
913 {
914     bool handler(ulong doublet)
915     {
916         if (visits.Add(doublet))
917         {
918             AllBottomUsagesCore(doublet, visits, usages);

```

```

917     }
918     return true;
919 }
920 if (Links.Unsync.Count(_constants.Any, link) == 0)
921 {
922     usages.Add(link);
923 }
924 else
925 {
926     Links.Unsync.Each(link, _constants.Any, handler);
927     Links.Unsync.Each(_constants.Any, link, handler);
928 }
929 }
930
931 public ulong CalculateTotalSymbolFrequencyCore(ulong symbol)
932 {
933     if (Options.UseSequenceMarker)
934     {
935         var counter = new TotalMarkedSequenceSymbolFrequencyOneOffCounter<ulong>(Links,
936             ↪ Options.MarkedSequenceMatcher, symbol);
937         return counter.Count();
938     }
939     else
940     {
941         var counter = new TotalSequenceSymbolFrequencyOneOffCounter<ulong>(Links,
942             ↪ symbol);
943         return counter.Count();
944     }
945 }
946
947 private bool AllUsagesCore1(ulong link, HashSet<ulong> usages, Func<ulong, bool>
948     ↪ outerHandler)
949 {
950     bool handler(ulong doublet)
951     {
952         if (usages.Add(doublet))
953         {
954             if (!outerHandler(doublet))
955             {
956                 return false;
957             }
958             if (!AllUsagesCore1(doublet, usages, outerHandler))
959             {
960                 return false;
961             }
962         }
963         return true;
964     }
965     return Links.Unsync.Each(link, _constants.Any, handler)
966         && Links.Unsync.Each(_constants.Any, link, handler);
967 }
968
969 public void CalculateAllUsages(ulong[] totals)
970 {
971     var calculator = new AllUsagesCalculator(Links, totals);
972     calculator.Calculate();
973 }
974
975 public void CalculateAllUsages2(ulong[] totals)
976 {
977     var calculator = new AllUsagesCalculator2(Links, totals);
978     calculator.Calculate();
979 }
980
981 private class AllUsagesCalculator
982 {
983     private readonly SynchronizedLinks<ulong> _links;
984     private readonly ulong[] _totals;
985
986     public AllUsagesCalculator(SynchronizedLinks<ulong> links, ulong[] totals)
987     {
988         _links = links;
989         _totals = totals;
990     }
991
992     public void Calculate() => _links.Each(_constants.Any, _constants.Any,
993         ↪ CalculateCore);
994
995     private bool CalculateCore(ulong link)

```

```

992     {
993         if (_totals[link] == 0)
994         {
995             var total = 1UL;
996             _totals[link] = total;
997             var visitedChildren = new HashSet<ulong>();
998             bool linkCalculator(ulong child)
999             {
1000                 if (link != child && visitedChildren.Add(child))
1001                 {
1002                     total += _totals[child] == 0 ? 1 : _totals[child];
1003                 }
1004                 return true;
1005             }
1006             _links.Unsync.Each(link, _constants.Any, linkCalculator);
1007             _links.Unsync.Each(_constants.Any, link, linkCalculator);
1008             _totals[link] = total;
1009         }
1010         return true;
1011     }
1012 }
1013
1014 private class AllUsagesCalculator2
1015 {
1016     private readonly SynchronizedLinks<ulong> _links;
1017     private readonly ulong[] _totals;
1018
1019     public AllUsagesCalculator2(SynchronizedLinks<ulong> links, ulong[] totals)
1020     {
1021         _links = links;
1022         _totals = totals;
1023     }
1024
1025     public void Calculate() => _links.Each(_constants.Any, _constants.Any,
1026         ↪ CalculateCore);
1027
1028     private bool IsElement(ulong link)
1029     {
1030         // _linksInSequence.Contains(link) ||
1031         return _links.Unsync.GetTarget(link) == link || _links.Unsync.GetSource(link) ==
1032             ↪ link;
1033     }
1034
1035     private bool CalculateCore(ulong link)
1036     {
1037         // TODO: Проработать защиту от заикливания
1038         // Основано на SequenceWalker.WalkLeft
1039         Func<ulong, ulong> getSource = _links.Unsync.GetSource;
1040         Func<ulong, ulong> getTarget = _links.Unsync.GetTarget;
1041         Func<ulong, bool> isElement = IsElement;
1042         void visitLeaf(ulong parent)
1043         {
1044             if (link != parent)
1045             {
1046                 _totals[parent]++;
1047             }
1048         }
1049         void visitNode(ulong parent)
1050         {
1051             if (link != parent)
1052             {
1053                 _totals[parent]++;
1054             }
1055         }
1056         var stack = new Stack();
1057         var element = link;
1058         if (isElement(element))
1059         {
1060             visitLeaf(element);
1061         }
1062         else
1063         {
1064             while (true)
1065             {
1066                 if (isElement(element))
1067                 {
1068                     if (stack.Count == 0)
1069                     {
1070                         break;
1071                     }
1072                 }
1073             }
1074         }
1075     }
1076 }

```

```

1069         }
1070         element = stack.Pop();
1071         var source = getSource(element);
1072         var target = getTarget(element);
1073         // 06пабoтка элемеHта
1074         if (isElement(target))
1075         {
1076             visitLeaf(target);
1077         }
1078         if (isElement(source))
1079         {
1080             visitLeaf(source);
1081         }
1082         element = source;
1083     }
1084     else
1085     {
1086         stack.Push(element);
1087         visitNode(element);
1088         element = getTarget(element);
1089     }
1090 }
1091 }
1092 _totals[link]++;
1093 return true;
1094 }
1095 }
1096
1097 private class AllUsagesCollector
1098 {
1099     private readonly ILinks<ulong> _links;
1100     private readonly HashSet<ulong> _usages;
1101
1102     public AllUsagesCollector(ILinks<ulong> links, HashSet<ulong> usages)
1103     {
1104         _links = links;
1105         _usages = usages;
1106     }
1107
1108     public bool Collect(ulong link)
1109     {
1110         if (_usages.Add(link))
1111         {
1112             _links.Each(link, _constants.Any, Collect);
1113             _links.Each(_constants.Any, link, Collect);
1114         }
1115         return true;
1116     }
1117 }
1118
1119 private class AllUsagesCollector1
1120 {
1121     private readonly ILinks<ulong> _links;
1122     private readonly HashSet<ulong> _usages;
1123     private readonly ulong _continue;
1124
1125     public AllUsagesCollector1(ILinks<ulong> links, HashSet<ulong> usages)
1126     {
1127         _links = links;
1128         _usages = usages;
1129         _continue = _links.Constants.Continue;
1130     }
1131
1132     public ulong Collect(ICollection<ulong> link)
1133     {
1134         var linkIndex = _links.GetIndex(link);
1135         if (_usages.Add(linkIndex))
1136         {
1137             _links.Each(Collect, _constants.Any, linkIndex);
1138         }
1139         return _continue;
1140     }
1141 }
1142
1143 private class AllUsagesCollector2
1144 {
1145     private readonly ILinks<ulong> _links;
1146     private readonly BitString _usages;
1147
1148     public AllUsagesCollector2(ILinks<ulong> links, BitString usages)

```

```

1149     {
1150         _links = links;
1151         _usages = usages;
1152     }
1153
1154     public bool Collect(ulong link)
1155     {
1156         if (_usages.Add((long)link))
1157         {
1158             _links.Each(link, _constants.Any, Collect);
1159             _links.Each(_constants.Any, link, Collect);
1160         }
1161         return true;
1162     }
1163 }
1164
1165 private class AllUsagesIntersectingCollector
1166 {
1167     private readonly SynchronizedLinks<ulong> _links;
1168     private readonly HashSet<ulong> _intersectWith;
1169     private readonly HashSet<ulong> _usages;
1170     private readonly HashSet<ulong> _enter;
1171
1172     public AllUsagesIntersectingCollector(SynchronizedLinks<ulong> links, HashSet<ulong>
↪ intersectWith, HashSet<ulong> usages)
1173     {
1174         _links = links;
1175         _intersectWith = intersectWith;
1176         _usages = usages;
1177         _enter = new HashSet<ulong>(); // защита от зацикливания
1178     }
1179
1180     public bool Collect(ulong link)
1181     {
1182         if (_enter.Add(link))
1183         {
1184             if (_intersectWith.Contains(link))
1185             {
1186                 _usages.Add(link);
1187             }
1188             _links.Unsync.Each(link, _constants.Any, Collect);
1189             _links.Unsync.Each(_constants.Any, link, Collect);
1190         }
1191         return true;
1192     }
1193 }
1194
1195 private void CloseInnerConnections(Action<ulong> handler, ulong left, ulong right)
1196 {
1197     TryStepLeftUp(handler, left, right);
1198     TryStepRightUp(handler, right, left);
1199 }
1200
1201 private void AllCloseConnections(Action<ulong> handler, ulong left, ulong right)
1202 {
1203     // Direct
1204     if (left == right)
1205     {
1206         handler(left);
1207     }
1208     var doublet = Links.Unsync.SearchOrDefault(left, right);
1209     if (doublet != _constants.Null)
1210     {
1211         handler(doublet);
1212     }
1213     // Inner
1214     CloseInnerConnections(handler, left, right);
1215     // Outer
1216     StepLeft(handler, left, right);
1217     StepRight(handler, left, right);
1218     PartialStepRight(handler, left, right);
1219     PartialStepLeft(handler, left, right);
1220 }
1221
1222 private HashSet<ulong> GetAllPartiallyMatchingSequencesCore(ulong[] sequence,
↪ HashSet<ulong> previousMatchings, long startAt)
1223 {
1224     if (startAt >= sequence.Length) // ?
1225     {

```

```

1226         return previousMatchings;
1227     }
1228     var secondLinkUsages = new HashSet<ulong>();
1229     AllUsagesCore(sequence[startAt], secondLinkUsages);
1230     secondLinkUsages.Add(sequence[startAt]);
1231     var matchings = new HashSet<ulong>();
1232     //for (var i = 0; i < previousMatchings.Count; i++)
1233     foreach (var secondLinkUsage in secondLinkUsages)
1234     {
1235         foreach (var previousMatching in previousMatchings)
1236         {
1237             //AllCloseConnections(matchings.AddAndReturnVoid, previousMatching,
1238             ↪ secondLinkUsage);
1239             StepRight(matchings.AddAndReturnVoid, previousMatching, secondLinkUsage);
1240             TryStepRightUp(matchings.AddAndReturnVoid, secondLinkUsage,
1241             ↪ previousMatching);
1242             //PartialStepRight(matchings.AddAndReturnVoid, secondLinkUsage,
1243             ↪ sequence[startAt]); // почему-то эта ошибочная запись приводит к
1244             ↪ желаемым результатам.
1245             PartialStepRight(matchings.AddAndReturnVoid, previousMatching,
1246             ↪ secondLinkUsage);
1247         }
1248     }
1249     if (matchings.Count == 0)
1250     {
1251         return matchings;
1252     }
1253     return GetAllPartiallyMatchingSequencesCore(sequence, matchings, startAt + 1); // ??
1254 }
1255
1256 private static void EnsureEachLinkIsAnyOrZeroOrManyOrExists(SynchronizedLinks<ulong>
1257 ↪ links, params ulong[] sequence)
1258 {
1259     if (sequence == null)
1260     {
1261         return;
1262     }
1263     for (var i = 0; i < sequence.Length; i++)
1264     {
1265         if (sequence[i] != _constants.Any && sequence[i] != ZeroOrMany &&
1266         ↪ !links.Exists(sequence[i]))
1267         {
1268             throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
1269             ↪ $"patternSequence[{i}]");
1270         }
1271     }
1272 }
1273
1274 // Pattern Matching -> Key To Triggers
1275 public HashSet<ulong> MatchPattern(params ulong[] patternSequence)
1276 {
1277     return Sync.ExecuteReadOperation(() =>
1278     {
1279         patternSequence = Simplify(patternSequence);
1280         if (patternSequence.Length > 0)
1281         {
1282             EnsureEachLinkIsAnyOrZeroOrManyOrExists(Links, patternSequence);
1283             var uniqueSequenceElements = new HashSet<ulong>();
1284             for (var i = 0; i < patternSequence.Length; i++)
1285             {
1286                 if (patternSequence[i] != _constants.Any && patternSequence[i] !=
1287                 ↪ ZeroOrMany)
1288                 {
1289                     uniqueSequenceElements.Add(patternSequence[i]);
1290                 }
1291             }
1292             var results = new HashSet<ulong>();
1293             foreach (var uniqueSequenceElement in uniqueSequenceElements)
1294             {
1295                 AllUsagesCore(uniqueSequenceElement, results);
1296             }
1297             var filteredResults = new HashSet<ulong>();
1298             var matcher = new PatternMatcher(this, patternSequence, filteredResults);
1299             matcher.AddAllPatternMatchedToResults(results);
1300             return filteredResults;
1301         }
1302     }
1303     return new HashSet<ulong>();
1304 }

```

```

1294     });
1295 }
1296
1297 // Найти все возможные связи между указанным списком связей.
1298 // Находит связи между всеми указанными связями в любом порядке.
1299 // TODO: решить что делать с повторами (когда одни и те же элементы встречаются
1300 //        несколько раз в последовательности)
1301 public HashSet<ulong> GetAllConnections(params ulong[] linksToConnect)
1302 {
1303     return Sync.ExecuteReadOperation(() =>
1304     {
1305         var results = new HashSet<ulong>();
1306         if (linksToConnect.Length > 0)
1307         {
1308             Links.EnsureEachLinkExists(linksToConnect);
1309             AllUsagesCore(linksToConnect[0], results);
1310             for (var i = 1; i < linksToConnect.Length; i++)
1311             {
1312                 var next = new HashSet<ulong>();
1313                 AllUsagesCore(linksToConnect[i], next);
1314                 results.IntersectWith(next);
1315             }
1316             return results;
1317         }
1318     });
1319 }
1320
1321 public HashSet<ulong> GetAllConnections1(params ulong[] linksToConnect)
1322 {
1323     return Sync.ExecuteReadOperation(() =>
1324     {
1325         var results = new HashSet<ulong>();
1326         if (linksToConnect.Length > 0)
1327         {
1328             Links.EnsureEachLinkExists(linksToConnect);
1329             var collector1 = new AllUsagesCollector(Links.Unsync, results);
1330             collector1.Collect(linksToConnect[0]);
1331             var next = new HashSet<ulong>();
1332             for (var i = 1; i < linksToConnect.Length; i++)
1333             {
1334                 var collector = new AllUsagesCollector(Links.Unsync, next);
1335                 collector.Collect(linksToConnect[i]);
1336                 results.IntersectWith(next);
1337                 next.Clear();
1338             }
1339             return results;
1340         }
1341     });
1342 }
1343
1344 public HashSet<ulong> GetAllConnections2(params ulong[] linksToConnect)
1345 {
1346     return Sync.ExecuteReadOperation(() =>
1347     {
1348         var results = new HashSet<ulong>();
1349         if (linksToConnect.Length > 0)
1350         {
1351             Links.EnsureEachLinkExists(linksToConnect);
1352             var collector1 = new AllUsagesCollector(Links, results);
1353             collector1.Collect(linksToConnect[0]);
1354             //AllUsagesCore(linksToConnect[0], results);
1355             for (var i = 1; i < linksToConnect.Length; i++)
1356             {
1357                 var next = new HashSet<ulong>();
1358                 var collector = new AllUsagesIntersectingCollector(Links, results, next);
1359                 collector.Collect(linksToConnect[i]);
1360                 //AllUsagesCore(linksToConnect[i], next);
1361                 //results.IntersectWith(next);
1362                 results = next;
1363             }
1364             return results;
1365         }
1366     });
1367 }
1368
1369 public List<ulong> GetAllConnections3(params ulong[] linksToConnect)
1370 {
1371     return Sync.ExecuteReadOperation(() =>

```

```

1371 {
1372     var results = new BitString((long)Links.Unsync.Count() + 1); // new
        ↳ BitArray((int)_links.Total + 1);
1373     if (linksToConnect.Length > 0)
1374     {
1375         Links.EnsureEachLinkExists(linksToConnect);
1376         var collector1 = new AllUsagesCollector2(Links.Unsync, results);
1377         collector1.Collect(linksToConnect[0]);
1378         for (var i = 1; i < linksToConnect.Length; i++)
1379         {
1380             var next = new BitString((long)Links.Unsync.Count() + 1); //new
                ↳ BitArray((int)_links.Total + 1);
1381             var collector = new AllUsagesCollector2(Links.Unsync, next);
1382             collector.Collect(linksToConnect[i]);
1383             results = results.And(next);
1384         }
1385     }
1386     return results.GetSetUInt64Indices();
1387 });
1388 }
1389
1390 private static ulong[] Simplify(ulong[] sequence)
1391 {
1392     // Считаем новый размер последовательности
1393     long newLength = 0;
1394     var zeroOrManyStepped = false;
1395     for (var i = 0; i < sequence.Length; i++)
1396     {
1397         if (sequence[i] == ZeroOrMany)
1398         {
1399             if (zeroOrManyStepped)
1400             {
1401                 continue;
1402             }
1403             zeroOrManyStepped = true;
1404         }
1405         else
1406         {
1407             //if (zeroOrManyStepped) Is it efficient?
1408             zeroOrManyStepped = false;
1409         }
1410         newLength++;
1411     }
1412     // Строим новую последовательность
1413     zeroOrManyStepped = false;
1414     var newSequence = new ulong[newLength];
1415     long j = 0;
1416     for (var i = 0; i < sequence.Length; i++)
1417     {
1418         //var current = zeroOrManyStepped;
1419         //zeroOrManyStepped = patternSequence[i] == zeroOrMany;
1420         //if (current && zeroOrManyStepped)
1421         //    continue;
1422         //var newZeroOrManyStepped = patternSequence[i] == zeroOrMany;
1423         //if (zeroOrManyStepped && newZeroOrManyStepped)
1424         //    continue;
1425         //zeroOrManyStepped = newZeroOrManyStepped;
1426         if (sequence[i] == ZeroOrMany)
1427         {
1428             if (zeroOrManyStepped)
1429             {
1430                 continue;
1431             }
1432             zeroOrManyStepped = true;
1433         }
1434         else
1435         {
1436             //if (zeroOrManyStepped) Is it efficient?
1437             zeroOrManyStepped = false;
1438         }
1439         newSequence[j++] = sequence[i];
1440     }
1441     return newSequence;
1442 }
1443
1444 public static void TestSimplify()
1445 {
1446     var sequence = new ulong[] { ZeroOrMany, ZeroOrMany, 2, 3, 4, ZeroOrMany,
        ↳ ZeroOrMany, ZeroOrMany, 4, ZeroOrMany, ZeroOrMany, ZeroOrMany };

```



```

1447     var simplifiedSequence = Simplify(sequence);
1448 }
1449
1450 public List<ulong> GetSimilarSequences() => new List<ulong>();
1451
1452 public void Prediction()
1453 {
1454     //_links
1455     //_sequences
1456 }
1457
1458 #region From Triplets
1459
1460 //public static void DeleteSequence(Link sequence)
1461 //{
1462 //}
1463
1464 public List<ulong> CollectMatchingSequences(ulong[] links)
1465 {
1466     if (links.Length == 1)
1467     {
1468         throw new Exception("Подпоследовательности с одним элементом не
1469             ↳ поддерживаются.");
1470     }
1471     var leftBound = 0;
1472     var rightBound = links.Length - 1;
1473     var left = links[leftBound++];
1474     var right = links[rightBound--];
1475     var results = new List<ulong>();
1476     CollectMatchingSequences(left, leftBound, links, right, rightBound, ref results);
1477     return results;
1478 }
1479
1480 private void CollectMatchingSequences(ulong leftLink, int leftBound, ulong[]
1481     ↳ middleLinks, ulong rightLink, int rightBound, ref List<ulong> results)
1482 {
1483     var leftLinkTotalReferers = Links.Unsync.Count(leftLink);
1484     var rightLinkTotalReferers = Links.Unsync.Count(rightLink);
1485     if (leftLinkTotalReferers <= rightLinkTotalReferers)
1486     {
1487         var nextLeftLink = middleLinks[leftBound];
1488         var elements = GetRightElements(leftLink, nextLeftLink);
1489         if (leftBound <= rightBound)
1490         {
1491             for (var i = elements.Length - 1; i >= 0; i--)
1492             {
1493                 var element = elements[i];
1494                 if (element != 0)
1495                 {
1496                     CollectMatchingSequences(element, leftBound + 1, middleLinks,
1497                         ↳ rightLink, rightBound, ref results);
1498                 }
1499             }
1500         }
1501         else
1502         {
1503             for (var i = elements.Length - 1; i >= 0; i--)
1504             {
1505                 var element = elements[i];
1506                 if (element != 0)
1507                 {
1508                     results.Add(element);
1509                 }
1510             }
1511         }
1512     }
1513     else
1514     {
1515         var nextRightLink = middleLinks[rightBound];
1516         var elements = GetLeftElements(rightLink, nextRightLink);
1517         if (leftBound <= rightBound)
1518         {
1519             for (var i = elements.Length - 1; i >= 0; i--)
1520             {
1521                 var element = elements[i];
1522                 if (element != 0)
1523                 {

```

```

1521         CollectMatchingSequences(leftLink, leftBound, middleLinks,
1522             ↪ elements[i], rightBound - 1, ref results);
1523     }
1524 }
1525 else
1526 {
1527     for (var i = elements.Length - 1; i >= 0; i--)
1528     {
1529         var element = elements[i];
1530         if (element != 0)
1531         {
1532             results.Add(element);
1533         }
1534     }
1535 }
1536 }
1537 }
1538
1539 public ulong[] GetRightElements(ulong startLink, ulong rightLink)
1540 {
1541     var result = new ulong[5];
1542     TryStepRight(startLink, rightLink, result, 0);
1543     Links.Each(_constants.Any, startLink, couple =>
1544     {
1545         if (couple != startLink)
1546         {
1547             if (TryStepRight(couple, rightLink, result, 2))
1548             {
1549                 return false;
1550             }
1551         }
1552         return true;
1553     });
1554     if (Links.GetTarget(Links.GetTarget(startLink)) == rightLink)
1555     {
1556         result[4] = startLink;
1557     }
1558     return result;
1559 }
1560
1561 public bool TryStepRight(ulong startLink, ulong rightLink, ulong[] result, int offset)
1562 {
1563     var added = 0;
1564     Links.Each(startLink, _constants.Any, couple =>
1565     {
1566         if (couple != startLink)
1567         {
1568             var coupleTarget = Links.GetTarget(couple);
1569             if (coupleTarget == rightLink)
1570             {
1571                 result[offset] = couple;
1572                 if (++added == 2)
1573                 {
1574                     return false;
1575                 }
1576             }
1577             else if (Links.GetSource(coupleTarget) == rightLink) // coupleTarget.Linker
1578                 ↪ == Net.And &&
1579             {
1580                 result[offset + 1] = couple;
1581                 if (++added == 2)
1582                 {
1583                     return false;
1584                 }
1585             }
1586         }
1587         return true;
1588     });
1589     return added > 0;
1590 }
1591
1592 public ulong[] GetLeftElements(ulong startLink, ulong leftLink)
1593 {
1594     var result = new ulong[5];
1595     TryStepLeft(startLink, leftLink, result, 0);
1596     Links.Each(startLink, _constants.Any, couple =>

```

```

1597         if (couple != startLink)
1598         {
1599             if (TryStepLeft(couple, leftLink, result, 2))
1600             {
1601                 return false;
1602             }
1603         }
1604         return true;
1605     });
1606     if (Links.GetSource(Links.GetSource(leftLink)) == startLink)
1607     {
1608         result[4] = leftLink;
1609     }
1610     return result;
1611 }
1612
1613 public bool TryStepLeft(ulong startLink, ulong leftLink, ulong[] result, int offset)
1614 {
1615     var added = 0;
1616     Links.Each(_constants.Any, startLink, couple =>
1617     {
1618         if (couple != startLink)
1619         {
1620             var coupleSource = Links.GetSource(couple);
1621             if (coupleSource == leftLink)
1622             {
1623                 result[offset] = couple;
1624                 if (++added == 2)
1625                 {
1626                     return false;
1627                 }
1628             }
1629             else if (Links.GetTarget(coupleSource) == leftLink) // coupleSource.Linker
1630             ↪ == Net.And &&
1631             {
1632                 result[offset + 1] = couple;
1633                 if (++added == 2)
1634                 {
1635                     return false;
1636                 }
1637             }
1638         }
1639         return true;
1640     });
1641     return added > 0;
1642 }
1643
1644 #endregion
1645
1646 #region Walkers
1647
1648 public class PatternMatcher : RightSequenceWalker<ulong>
1649 {
1650     private readonly Sequences _sequences;
1651     private readonly ulong[] _patternSequence;
1652     private readonly HashSet<LinkIndex> _linksInSequence;
1653     private readonly HashSet<LinkIndex> _results;
1654
1655     #region Pattern Match
1656
1657     enum PatternBlockType
1658     {
1659         Undefined,
1660         Gap,
1661         Elements
1662     }
1663
1664     struct PatternBlock
1665     {
1666         public PatternBlockType Type;
1667         public long Start;
1668         public long Stop;
1669     }
1670
1671     private readonly List<PatternBlock> _pattern;
1672     private int _patternPosition;
1673     private long _sequencePosition;
1674
1675     #endregion

```

```

1676 public PatternMatcher(Sequences sequences, LinkIndex[] patternSequence,
1677 ↪ HashSet<LinkIndex> results)
1678 : base(sequences.Links.Unsync, new DefaultStack<ulong>())
1679 {
1680     _sequences = sequences;
1681     _patternSequence = patternSequence;
1682     _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
1683 ↪ _constants.Any && x != ZeroOrMany));
1684     _results = results;
1685     _pattern = CreateDetailedPattern();
1686 }
1687
1688 protected override bool IsElement(ulong link) => _linksInSequence.Contains(link) ||
1689 ↪ base.IsElement(link);
1690
1691 public bool PatternMatch(LinkIndex sequenceToMatch)
1692 {
1693     _patternPosition = 0;
1694     _sequencePosition = 0;
1695     foreach (var part in Walk(sequenceToMatch))
1696     {
1697         if (!PatternMatchCore(part))
1698         {
1699             break;
1700         }
1701     }
1702     return _patternPosition == _pattern.Count || (_patternPosition == _pattern.Count
1703 ↪ - 1 && _pattern[_patternPosition].Start == 0);
1704 }
1705
1706 private List<PatternBlock> CreateDetailedPattern()
1707 {
1708     var pattern = new List<PatternBlock>();
1709     var patternBlock = new PatternBlock();
1710     for (var i = 0; i < _patternSequence.Length; i++)
1711     {
1712         if (patternBlock.Type == PatternBlockType.Undefined)
1713         {
1714             if (_patternSequence[i] == _constants.Any)
1715             {
1716                 patternBlock.Type = PatternBlockType.Gap;
1717                 patternBlock.Start = 1;
1718                 patternBlock.Stop = 1;
1719             }
1720             else if (_patternSequence[i] == ZeroOrMany)
1721             {
1722                 patternBlock.Type = PatternBlockType.Gap;
1723                 patternBlock.Start = 0;
1724                 patternBlock.Stop = long.MaxValue;
1725             }
1726             else
1727             {
1728                 patternBlock.Type = PatternBlockType.Elements;
1729                 patternBlock.Start = i;
1730                 patternBlock.Stop = i;
1731             }
1732         }
1733         else if (patternBlock.Type == PatternBlockType.Elements)
1734         {
1735             if (_patternSequence[i] == _constants.Any)
1736             {
1737                 pattern.Add(patternBlock);
1738                 patternBlock = new PatternBlock
1739                 {
1740                     Type = PatternBlockType.Gap,
1741                     Start = 1,
1742                     Stop = 1
1743                 };
1744             }
1745             else if (_patternSequence[i] == ZeroOrMany)
1746             {
1747                 pattern.Add(patternBlock);
1748                 patternBlock = new PatternBlock
1749                 {
1750                     Type = PatternBlockType.Gap,
1751                     Start = 0,
1752                     Stop = long.MaxValue
1753                 };
1754             }
1755             else

```

```

1752         {
1753             patternBlock.Stop = i;
1754         }
1755     }
1756     else // patternBlock.Type == PatternBlockType.Gap
1757     {
1758         if (_patternSequence[i] == _constants.Any)
1759         {
1760             patternBlock.Start++;
1761             if (patternBlock.Stop < patternBlock.Start)
1762             {
1763                 patternBlock.Stop = patternBlock.Start;
1764             }
1765         }
1766         else if (_patternSequence[i] == ZeroOrMany)
1767         {
1768             patternBlock.Stop = long.MaxValue;
1769         }
1770         else
1771         {
1772             pattern.Add(patternBlock);
1773             patternBlock = new PatternBlock
1774             {
1775                 Type = PatternBlockType.Elements,
1776                 Start = i,
1777                 Stop = i
1778             };
1779         }
1780     }
1781 }
1782 if (patternBlock.Type != PatternBlockType.Undefined)
1783 {
1784     pattern.Add(patternBlock);
1785 }
1786 return pattern;
1787 }
1788
1789 /** match: search for regexp anywhere in text */
1790 //int match(char* regexp, char* text)
1791 //{
1792 //    do
1793 //    {
1794 //        } while (*text++ != '\0');
1795 //    return 0;
1796 //}
1797
1798 /** matchhere: search for regexp at beginning of text */
1799 //int matchhere(char* regexp, char* text)
1800 //{
1801 //    if (regexp[0] == '\0')
1802 //        return 1;
1803 //    if (regexp[1] == '*')
1804 //        return matchstar(regexp[0], regexp + 2, text);
1805 //    if (regexp[0] == '$' && regexp[1] == '\0')
1806 //        return *text == '\0';
1807 //    if (*text != '\0' && (regexp[0] == '.' || regexp[0] == *text))
1808 //        return matchhere(regexp + 1, text + 1);
1809 //    return 0;
1810 //}
1811
1812 /** matchstar: search for c*regexp at beginning of text */
1813 //int matchstar(int c, char* regexp, char* text)
1814 //{
1815 //    do
1816 //    {
1817 //        /* a * matches zero or more instances */
1818 //        if (matchhere(regexp, text))
1819 //            return 1;
1820 //    } while (*text != '\0' && (*text++ == c || c == '.'));
1821 //    return 0;
1822 //}
1823
1824 //private void GetNextPatternElement(out LinkIndex element, out long mininumGap, out
1825 //    ↪ long maximumGap)
1826 //{
1827 //    mininumGap = 0;
1828 //    maximumGap = 0;
1829 //    element = 0;
1830 //    for (; _patternPosition < _patternSequence.Length; _patternPosition++)
1831 //    {

```

```

1830         //         if (_patternSequence[_patternPosition] == Doublets.Links.Null)
1831             //             mininumGap++;
1832         //         else if (_patternSequence[_patternPosition] == ZeroOrMany)
1833             //             maximumGap = long.MaxValue;
1834         //         else
1835             //             break;
1836         //     }
1837
1838         //     if (maximumGap < mininumGap)
1839             //         maximumGap = mininumGap;
1840         // }
1841
1842 private bool PatternMatchCore(LinkIndex element)
1843 {
1844     if (_patternPosition >= _pattern.Count)
1845     {
1846         _patternPosition = -2;
1847         return false;
1848     }
1849     var currentPatternBlock = _pattern[_patternPosition];
1850     if (currentPatternBlock.Type == PatternBlockType.Gap)
1851     {
1852         //var currentMatchingBlockLength = (_sequencePosition -
1853         ↪ _lastMatchedBlockPosition);
1854         if (_sequencePosition < currentPatternBlock.Start)
1855         {
1856             _sequencePosition++;
1857             return true; // Двигаемся дальше
1858         }
1859         // Это последний блок
1860         if (_pattern.Count == _patternPosition + 1)
1861         {
1862             _patternPosition++;
1863             _sequencePosition = 0;
1864             return false; // Полное соответствие
1865         }
1866         else
1867         {
1868             if (_sequencePosition > currentPatternBlock.Stop)
1869             {
1870                 return false; // Соответствие невозможно
1871             }
1872             var nextPatternBlock = _pattern[_patternPosition + 1];
1873             if (_patternSequence[nextPatternBlock.Start] == element)
1874             {
1875                 if (nextPatternBlock.Start < nextPatternBlock.Stop)
1876                 {
1877                     _patternPosition++;
1878                     _sequencePosition = 1;
1879                 }
1880                 else
1881                 {
1882                     _patternPosition += 2;
1883                     _sequencePosition = 0;
1884                 }
1885             }
1886         }
1887     }
1888     else // currentPatternBlock.Type == PatternBlockType.Elements
1889     {
1890         var patternElementPosition = currentPatternBlock.Start + _sequencePosition;
1891         if (_patternSequence[patternElementPosition] != element)
1892         {
1893             return false; // Соответствие невозможно
1894         }
1895         if (patternElementPosition == currentPatternBlock.Stop)
1896         {
1897             _patternPosition++;
1898             _sequencePosition = 0;
1899         }
1900         else
1901         {
1902             _sequencePosition++;
1903         }
1904     }
1905     return true;
1906     //if (_patternSequence[_patternPosition] != element)
1907     //    return false;
1908     //else

```

```

1908         //{
1909         //     _sequencePosition++;
1910         //     _patternPosition++;
1911         //     return true;
1912         //}
1913         //////////////////////////////////////////////////
1914         //if (_filterPosition == _patternSequence.Length)
1915         //{
1916         //     _filterPosition = -2; // Длиннее чем нужно
1917         //     return false;
1918         //}
1919         //if (element != _patternSequence[_filterPosition])
1920         //{
1921         //     _filterPosition = -1;
1922         //     return false; // Начинается иначе
1923         //}
1924         //_filterPosition++;
1925         //if (_filterPosition == (_patternSequence.Length - 1))
1926         //    return false;
1927         //if (_filterPosition >= 0)
1928         //{
1929         //    if (element == _patternSequence[_filterPosition + 1])
1930         //        _filterPosition++;
1931         //    else
1932         //        return false;
1933         //}
1934         //if (_filterPosition < 0)
1935         //{
1936         //    if (element == _patternSequence[0])
1937         //        _filterPosition = 0;
1938         //}
1939     }
1940
1941     public void AddAllPatternMatchedToResults(IEnumerable<ulong> sequencesToMatch)
1942     {
1943         foreach (var sequenceToMatch in sequencesToMatch)
1944         {
1945             if (PatternMatch(sequenceToMatch))
1946             {
1947                 _results.Add(sequenceToMatch);
1948             }
1949         }
1950     }
1951 }
1952
1953 #endregion
1954 }
1955 }

```

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

```

1  using Platform.Collections.Lists;
2  using Platform.Data.Sequences;
3  using System.Collections.Generic;
4
5  namespace Platform.Data.Doublets.Sequences
6  {
7      public static class SequencesExtensions
8      {
9          public static TLink Create<TLink>(this ISequences<TLink> sequences, IList<TLink[]>
10             ↳ groupedSequence)
11          {
12              var finalSequence = new TLink[groupedSequence.Count];
13              for (var i = 0; i < finalSequence.Length; i++)
14              {
15                  var part = groupedSequence[i];
16                  finalSequence[i] = part.Length == 1 ? part[0] : sequences.Create(part);
17              }
18              return sequences.Create(finalSequence);
19          }
20
21          public static IList<TLink> ToList<TLink>(this ISequences<TLink> sequences, TLink
22             ↳ sequence)
23          {
24              var list = new List<TLink>();
25              sequences.EachPart(list.AddAndReturnTrue, sequence);
26              return list;
27          }
28      }
29  }

```

27 }

./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.Data.Doublets.Sequences.Frequencies.Cache;
6 using Platform.Data.Doublets.Sequences.Frequencies.Counters;
7 using Platform.Data.Doublets.Sequences.Converters;
8 using Platform.Data.Doublets.Sequences.CriteriaMatchers;
9 using Platform.Data.Doublets.Sequences.Walkers;
10 using Platform.Data.Doublets.Sequences.Indexes;
11
12 namespace Platform.Data.Doublets.Sequences
13 {
14     public class SequencesOptions<TLink> // TODO: To use type parameter <TLink> the
15     ↪ ILinks<TLink> must contain GetConstants function.
16     {
17         private static readonly EqualityComparer<TLink> _equalityComparer =
18         ↪ EqualityComparer<TLink>.Default;
19
20         public TLink SequenceMarkerLink { get; set; }
21         public bool UseCascadeUpdate { get; set; }
22         public bool UseCascadeDelete { get; set; }
23         public bool UseIndex { get; set; } // TODO: Update Index on sequence update/delete.
24         public bool UseSequenceMarker { get; set; }
25         public bool UseCompression { get; set; }
26         public bool UseGarbageCollection { get; set; }
27         public bool EnforceSingleSequenceVersionOnWriteBasedOnExisting { get; set; }
28         public bool EnforceSingleSequenceVersionOnWriteBasedOnNew { get; set; }
29
30         public MarkedSequenceCriterionMatcher<TLink> MarkedSequenceMatcher { get; set; }
31         public IConverter<IList<TLink>, TLink> LinksToSequenceConverter { get; set; }
32         public ISequenceIndex<TLink> Index { get; set; }
33         public ISequenceWalker<TLink> Walker { get; set; }
34
35         // TODO: Реализовать компактификацию при чтении
36         //public bool EnforceSingleSequenceVersionOnRead { get; set; }
37         //public bool UseRequestMarker { get; set; }
38         //public bool StoreRequestResults { get; set; }
39
40         public void InitOptions(ISynchronizedLinks<TLink> links)
41         {
42             if (UseSequenceMarker)
43             {
44                 if (_equalityComparer.Equals(SequenceMarkerLink, links.Constants.Null))
45                 {
46                     SequenceMarkerLink = links.CreatePoint();
47                 }
48                 else
49                 {
50                     if (!links.Exists(SequenceMarkerLink))
51                     {
52                         var link = links.CreatePoint();
53                         if (!_equalityComparer.Equals(link, SequenceMarkerLink))
54                         {
55                             throw new InvalidOperationException("Cannot recreate sequence marker
56                             ↪ link.");
57                         }
58                     }
59                 }
60             }
61             if (MarkedSequenceMatcher == null)
62             {
63                 MarkedSequenceMatcher = new MarkedSequenceCriterionMatcher<TLink>(links,
64                 ↪ SequenceMarkerLink);
65             }
66             var balancedVariantConverter = new BalancedVariantConverter<TLink>(links);
67             if (UseCompression)
68             {
69                 if (LinksToSequenceConverter == null)
70                 {
71                     ICounter<TLink, TLink> totalSequenceSymbolFrequencyCounter;
72                     if (UseSequenceMarker)
73                     {
74                         totalSequenceSymbolFrequencyCounter = new
75                         ↪ TotalMarkedSequenceSymbolFrequencyCounter<TLink>(links,
76                         ↪ MarkedSequenceMatcher);
77                     }
78                     else
79                     {
80                         totalSequenceSymbolFrequencyCounter = new
81                         ↪ TotalSequenceSymbolFrequencyCounter<TLink>(links);
82                     }
83                 }
84             }
85         }
86     }
87 }

```



```

71     }
72     else
73     {
74         totalSequenceSymbolFrequencyCounter = new
            ↳ TotalSequenceSymbolFrequencyCounter<TLink>(links);
75     }
76     var doubletFrequenciesCache = new LinkFrequenciesCache<TLink>(links,
            ↳ totalSequenceSymbolFrequencyCounter);
77     var compressingConverter = new CompressingConverter<TLink>(links,
            ↳ balancedVariantConverter, doubletFrequenciesCache);
78     LinksToSequenceConverter = compressingConverter;
79 }
80 }
81 else
82 {
83     if (LinksToSequenceConverter == null)
84     {
85         LinksToSequenceConverter = balancedVariantConverter;
86     }
87 }
88 if (UseIndex && Index == null)
89 {
90     Index = new SequenceIndex<TLink>(links);
91 }
92 if (Walker == null)
93 {
94     Walker = new RightSequenceWalker<TLink>(links, new DefaultStack<TLink>());
95 }
96 }
97
98 public void ValidateOptions()
99 {
100     if (UseGarbageCollection && !UseSequenceMarker)
101     {
102         throw new NotSupportedException("To use garbage collection UseSequenceMarker
            ↳ option must be on.");
103     }
104 }
105 }
106 }

```

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

```

1 using System.Collections.Generic;
2
3 namespace Platform.Data.Doublets.Sequences.Walkers
4 {
5     public interface ISequenceWalker<TLink>
6     {
7         IEnumerable<TLink> Walk(TLink sequence);
8     }
9 }

```

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

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3 using Platform.Collections.Stacks;
4
5 namespace Platform.Data.Doublets.Sequences.Walkers
6 {
7     public class LeftSequenceWalker<TLink> : SequenceWalkerBase<TLink>
8     {
9         public LeftSequenceWalker(ILinks<TLink> links, IStack<TLink> stack) : base(links, stack)
            ↳ { }
10
11         [MethodImpl(MethodImplOptions.AggressiveInlining)]
12         protected override TLink GetNextElementAfterPop(TLink element) =>
            ↳ Links.GetSource(element);
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         protected override TLink GetNextElementAfterPush(TLink element) =>
            ↳ Links.GetTarget(element);
16
17         [MethodImpl(MethodImplOptions.AggressiveInlining)]
18         protected override IEnumerable<TLink> WalkContents(TLink element)
19         {
20             var parts = Links.GetLink(element);
21             var start = Links.Constants.IndexPart + 1;
22             for (var i = parts.Count - 1; i >= start; i--)

```

```

23     {
24         var part = parts[i];
25         if (IsElement(part))
26         {
27             yield return part;
28         }
29     }
30 }
31 }
32 }

```

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

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  ///#define USEARRAYPOOL
5  #if USEARRAYPOOL
6  using Platform.Collections;
7  #endif
8
9  namespace Platform.Data.Doublets.Sequences.Walkers
10 {
11     public class LeveledSequenceWalker<TLink> : LinksOperatorBase<TLink>, ISequenceWalker<TLink>
12     {
13         private static readonly EqualityComparer<TLink> _equalityComparer =
14             ⇨ EqualityComparer<TLink>.Default;
15
16         public LeveledSequenceWalker(ILinks<TLink> links) : base(links) { }
17
18         public IEnumerable<TLink> Walk(TLink sequence) => ToArray(sequence);
19
20         [MethodImpl(MethodImplOptions.AggressiveInlining)]
21         protected virtual bool IsElement(TLink elementLink) => Links.IsPartialPoint(elementLink);
22
23         public TLink[] ToArray(TLink sequence)
24         {
25             var length = 1;
26             var array = new TLink[length];
27             array[0] = sequence;
28             if (IsElement(sequence))
29             {
30                 return array;
31             }
32             bool hasElements;
33             do
34             {
35                 length *= 2;
36                 #if USEARRAYPOOL
37                     var nextArray = ArrayPool.Allocate<ulong>(length);
38                 #else
39                     var nextArray = new TLink[length];
40                 #endif
41                 hasElements = false;
42                 for (var i = 0; i < array.Length; i++)
43                 {
44                     var candidate = array[i];
45                     if (_equalityComparer.Equals(array[i], default))
46                     {
47                         continue;
48                     }
49                     var doubletOffset = i * 2;
50                     if (IsElement(candidate))
51                     {
52                         nextArray[doubletOffset] = candidate;
53                     }
54                     else
55                     {
56                         var link = Links.GetLink(candidate);
57                         var linkSource = Links.GetSource(link);
58                         var linkTarget = Links.GetTarget(link);
59                         nextArray[doubletOffset] = linkSource;
60                         nextArray[doubletOffset + 1] = linkTarget;
61                         if (!hasElements)
62                         {
63                             hasElements = !(IsElement(linkSource) && IsElement(linkTarget));
64                         }
65                     }
66                 }
67             } while (array.Length > 1)
68         }
69     }
70 }

```

```

68         {
69             ArrayPool.Free(array);
70         }
71 #endif
72         array = nextArray;
73     }
74     while (hasElements);
75     var filledElementsCount = CountFilledElements(array);
76     if (filledElementsCount == array.Length)
77     {
78         return array;
79     }
80     else
81     {
82         return CopyFilledElements(array, filledElementsCount);
83     }
84 }
85
86 [MethodImpl(MethodImplOptions.AggressiveInlining)]
87 private static TLink[] CopyFilledElements(TLink[] array, int filledElementsCount)
88 {
89     var finalArray = new TLink[filledElementsCount];
90     for (int i = 0, j = 0; i < array.Length; i++)
91     {
92         if (!_equalityComparer.Equals(array[i], default))
93         {
94             finalArray[j] = array[i];
95             j++;
96         }
97     }
98 #if USEARRAYPOOL
99     ArrayPool.Free(array);
100 #endif
101     return finalArray;
102 }
103
104 [MethodImpl(MethodImplOptions.AggressiveInlining)]
105 private static int CountFilledElements(TLink[] array)
106 {
107     var count = 0;
108     for (var i = 0; i < array.Length; i++)
109     {
110         if (!_equalityComparer.Equals(array[i], default))
111         {
112             count++;
113         }
114     }
115     return count;
116 }
117 }
118 }

```

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

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using Platform.Collections.Stacks;
4
5  namespace Platform.Data.Doublets.Sequences.Walkers
6  {
7      public class RightSequenceWalker<TLink> : SequenceWalkerBase<TLink>
8      {
9          public RightSequenceWalker(ILinks<TLink> links, IStack<TLink> stack) : base(links,
10              ↪ stack) { }
11
12          [MethodImpl(MethodImplOptions.AggressiveInlining)]
13          protected override TLink GetNextElementAfterPop(TLink element) =>
14              ↪ Links.GetTarget(element);
15
16          [MethodImpl(MethodImplOptions.AggressiveInlining)]
17          protected override TLink GetNextElementAfterPush(TLink element) =>
18              ↪ Links.GetSource(element);
19
20          [MethodImpl(MethodImplOptions.AggressiveInlining)]
21          protected override IEnumerable<TLink> WalkContents(TLink element)
22          {
23              var parts = Links.GetLink(element);
24              for (var i = Links.Constants.IndexPart + 1; i < parts.Count; i++)
25              {
26                  var part = parts[i];

```

```

24         if (IsElement(part))
25         {
26             yield return part;
27         }
28     }
29 }
30 }
31 }

```

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

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using Platform.Collections.Stacks;
4
5  namespace Platform.Data.Doublets.Sequences.Walkers
6  {
7      public abstract class SequenceWalkerBase<TLink> : LinksOperatorBase<TLink>,
8          ↳ ISequenceWalker<TLink>
9      {
10         private readonly IStack<TLink> _stack;
11
12         protected SequenceWalkerBase(ILinks<TLink> links, IStack<TLink> stack) : base(links) =>
13             ↳ _stack = stack;
14
15         public IEnumerable<TLink> Walk(TLink sequence)
16         {
17             _stack.Clear();
18             var element = sequence;
19             if (IsElement(element))
20             {
21                 yield return element;
22             }
23             else
24             {
25                 while (true)
26                 {
27                     if (IsElement(element))
28                     {
29                         if (_stack.IsEmpty)
30                         {
31                             break;
32                         }
33                         element = _stack.Pop();
34                         foreach (var output in WalkContents(element))
35                         {
36                             yield return output;
37                         }
38                         element = GetNextElementAfterPop(element);
39                     }
40                     else
41                     {
42                         _stack.Push(element);
43                         element = GetNextElementAfterPush(element);
44                     }
45                 }
46             }
47
48             [MethodImpl(MethodImplOptions.AggressiveInlining)]
49             protected virtual bool IsElement(TLink elementLink) => Links.IsPartialPoint(elementLink);
50
51             [MethodImpl(MethodImplOptions.AggressiveInlining)]
52             protected abstract TLink GetNextElementAfterPop(TLink element);
53
54             [MethodImpl(MethodImplOptions.AggressiveInlining)]
55             protected abstract TLink GetNextElementAfterPush(TLink element);
56
57             [MethodImpl(MethodImplOptions.AggressiveInlining)]
58             protected abstract IEnumerable<TLink> WalkContents(TLink element);
59 }

```

./Platform.Data.Doublets/Stacks/Stack.cs

```

1  using System.Collections.Generic;
2  using Platform.Collections.Stacks;
3
4  namespace Platform.Data.Doublets.Stacks
5  {
6      public class Stack<TLink> : IStack<TLink>

```

```

7 {
8     private static readonly EqualityComparer<TLink> _equalityComparer =
9         ↪ EqualityComparer<TLink>.Default;
10
11     private readonly ILinks<TLink> _links;
12     private readonly TLink _stack;
13
14     public bool IsEmpty => _equalityComparer.Equals(Peek(), _stack);
15
16     public Stack(ILinks<TLink> links, TLink stack)
17     {
18         _links = links;
19         _stack = stack;
20     }
21
22     private TLink GetStackMarker() => _links.GetSource(_stack);
23
24     private TLink GetTop() => _links.GetTarget(_stack);
25
26     public TLink Peek() => _links.GetTarget(GetTop());
27
28     public TLink Pop()
29     {
30         var element = Peek();
31         if (!_equalityComparer.Equals(element, _stack))
32         {
33             var top = GetTop();
34             var previousTop = _links.GetSource(top);
35             _links.Update(_stack, GetStackMarker(), previousTop);
36             _links.Delete(top);
37         }
38         return element;
39     }
40
41     public void Push(TLink element) => _links.Update(_stack, GetStackMarker(),
42         ↪ _links.GetOrCreate(GetTop(), element));
43 }

```

./Platform.Data.Doublets/Stacks/StackExtensions.cs

```

1 namespace Platform.Data.Doublets.Stacks
2 {
3     public static class StackExtensions
4     {
5         public static TLink CreateStack<TLink>(this ILinks<TLink> links, TLink stackMarker)
6         {
7             var stackPoint = links.CreatePoint();
8             var stack = links.Update(stackPoint, stackMarker, stackPoint);
9             return stack;
10         }
11     }
12 }

```

./Platform.Data.Doublets/SynchronizedLinks.cs

```

1 using System;
2 using System.Collections.Generic;
3 using Platform.Data.Constants;
4 using Platform.Data.Doublets;
5 using Platform.Threading.Synchronization;
6
7 namespace Platform.Data.Doublets
8 {
9     /// <remarks>
10     /// TODO: Autogeneration of synchronized wrapper (decorator).
11     /// TODO: Try to unfold code of each method using IL generation for performance improvements.
12     /// TODO: Or even to unfold multiple layers of implementations.
13     /// </remarks>
14     public class SynchronizedLinks<T> : ISynchronizedLinks<T>
15     {
16         public LinksCombinedConstants<T, T, int> Constants { get; }
17         public ISynchronization SyncRoot { get; }
18         public ILinks<T> Sync { get; }
19         public ILinks<T> Unsync { get; }
20
21         public SynchronizedLinks(ILinks<T> links) : this(new ReaderWriterLockSynchronization(),
22             ↪ links) { }
23
24         public SynchronizedLinks(ISynchronization synchronization, ILinks<T> links)
25         {
26
27         }
28     }
29 }

```

```

25     SyncRoot = synchronization;
26     Sync = this;
27     Unsync = links;
28     Constants = links.Constants;
29 }
30
31 public T Count(IList<T> restriction) => SyncRoot.ExecuteReadOperation(restriction,
    ↳ Unsync.Count);
32 public T Each(Func<IList<T>, T> handler, IList<T> restrictions) =>
    ↳ SyncRoot.ExecuteReadOperation(handler, restrictions, (handler1, restrictions1) =>
    ↳ Unsync.Each(handler1, restrictions1));
33 public T Create() => SyncRoot.ExecuteWriteOperation(Unsync.Create);
34 public T Update(IList<T> restrictions) => SyncRoot.ExecuteWriteOperation(restrictions,
    ↳ Unsync.Update);
35 public void Delete(T link) => SyncRoot.ExecuteWriteOperation(link, Unsync.Delete);
36
37 //public T Trigger(IList<T> restriction, Func<IList<T>, IList<T>, T> matchedHandler,
    ↳ IList<T> substitution, Func<IList<T>, IList<T>, T> substitutedHandler)
38 //{
39 //    if (restriction != null && substitution != null &&
    ↳ !substitution.EqualTo(restriction))
40 //        return SyncRoot.ExecuteWriteOperation(restriction, matchedHandler,
    ↳ substitution, substitutedHandler, Unsync.Trigger);
41 //
42 //    return SyncRoot.ExecuteReadOperation(restriction, matchedHandler, substitution,
    ↳ substitutedHandler, Unsync.Trigger);
43 //}
44 }
45 }

```

#### ./Platform.Data.Doublets/UInt64Link.cs

```

1  using System;
2  using System.Collections;
3  using System.Collections.Generic;
4  using Platform.Exceptions;
5  using Platform.Ranges;
6  using Platform.Singletons;
7  using Platform.Collections.Lists;
8  using Platform.Data.Constants;
9
10 namespace Platform.Data.Doublets
11 {
12     /// <summary>
13     /// Структура описывающая уникальную связь.
14     /// </summary>
15     public struct UInt64Link : IEquatable<UInt64Link>, IReadOnlyList<ulong>, IList<ulong>
16     {
17         private static readonly LinksCombinedConstants<bool, ulong, int> _constants =
            ↳ Default<LinksCombinedConstants<bool, ulong, int>>.Instance;
18
19         private const int Length = 3;
20
21         public readonly ulong Index;
22         public readonly ulong Source;
23         public readonly ulong Target;
24
25         public static readonly UInt64Link Null = new UInt64Link();
26
27         public UInt64Link(params ulong[] values)
28         {
29             Index = values.Length > _constants.IndexPart ? values[_constants.IndexPart] :
                ↳ _constants.Null;
30             Source = values.Length > _constants.SourcePart ? values[_constants.SourcePart] :
                ↳ _constants.Null;
31             Target = values.Length > _constants.TargetPart ? values[_constants.TargetPart] :
                ↳ _constants.Null;
32         }
33
34         public UInt64Link(IList<ulong> values)
35         {
36             Index = values.Count > _constants.IndexPart ? values[_constants.IndexPart] :
                ↳ _constants.Null;
37             Source = values.Count > _constants.SourcePart ? values[_constants.SourcePart] :
                ↳ _constants.Null;
38             Target = values.Count > _constants.TargetPart ? values[_constants.TargetPart] :
                ↳ _constants.Null;
39         }
40
41         public UInt64Link(ulong index, ulong source, ulong target)

```

```

42     {
43         Index = index;
44         Source = source;
45         Target = target;
46     }
47
48     public UInt64Link(ulong source, ulong target)
49         : this(_constants.Null, source, target)
50     {
51         Source = source;
52         Target = target;
53     }
54
55     public static UInt64Link Create(ulong source, ulong target) => new UInt64Link(source,
56         ↪ target);
57
58     public override int GetHashCode() => (Index, Source, Target).GetHashCode();
59
60     public bool IsNull() => Index == _constants.Null
61         && Source == _constants.Null
62         && Target == _constants.Null;
63
64     public override bool Equals(object other) => other is UInt64Link &&
65         ↪ Equals((UInt64Link)other);
66
67     public bool Equals(UInt64Link other) => Index == other.Index
68         && Source == other.Source
69         && Target == other.Target;
70
71     public static string ToString(ulong index, ulong source, ulong target) => $"{index}:
72         ↪ {source}->{target}";
73
74     public static string ToString(ulong source, ulong target) => $"{source}->{target}";
75
76     public static implicit operator ulong[] (UInt64Link link) => link.ToArray();
77
78     public static implicit operator UInt64Link(ulong[] linkArray) => new
79         ↪ UInt64Link(linkArray);
80
81     public override string ToString() => Index == _constants.Null ? ToString(Source, Target)
82         ↪ : ToString(Index, Source, Target);
83
84     #region IList
85
86     public ulong this[int index]
87     {
88         get
89         {
90             Ensure.Always.ArgumentInRange(index, new Range<int>(0, Length - 1),
91                 ↪ nameof(index));
92             if (index == _constants.IndexPart)
93             {
94                 return Index;
95             }
96             if (index == _constants.SourcePart)
97             {
98                 return Source;
99             }
100             if (index == _constants.TargetPart)
101             {
102                 return Target;
103             }
104             throw new NotSupportedException(); // Impossible path due to
105                 ↪ Ensure.ArgumentInRange
106         }
107         set => throw new NotSupportedException();
108     }
109
110     public int Count => Length;
111
112     public bool IsReadOnly => true;
113
114     IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
115
116     public IEnumerator<ulong> GetEnumerator()
117     {
118         yield return Index;
119         yield return Source;
120         yield return Target;
121     }
122

```

```

115     public void Add(ulong item) => throw new NotSupportedException();
116
117     public void Clear() => throw new NotSupportedException();
118
119     public bool Contains(ulong item) => IndexOf(item) >= 0;
120
121     public void CopyTo(ulong[] array, int arrayIndex)
122     {
123         Ensure.Always.ArgumentNotNull(array, nameof(array));
124         Ensure.Always.ArgumentInRange(arrayIndex, new Range<int>(0, array.Length - 1),
125             ↳ nameof(arrayIndex));
126         if (arrayIndex + Length > array.Length)
127         {
128             throw new ArgumentException();
129         }
130         array[arrayIndex++] = Index;
131         array[arrayIndex++] = Source;
132         array[arrayIndex] = Target;
133     }
134
135     public bool Remove(ulong item) => Throw.A.NotSupportedExceptionAndReturn<bool>();
136
137     public int IndexOf(ulong item)
138     {
139         if (Index == item)
140         {
141             return _constants.IndexPart;
142         }
143         if (Source == item)
144         {
145             return _constants.SourcePart;
146         }
147         if (Target == item)
148         {
149             return _constants.TargetPart;
150         }
151         return -1;
152     }
153
154     public void Insert(int index, ulong item) => throw new NotSupportedException();
155
156     public void RemoveAt(int index) => throw new NotSupportedException();
157
158     #endregion
159 }
160
161 }

```

./Platform.Data.Doublets/UInt64LinkExtensions.cs

```

1 namespace Platform.Data.Doublets
2 {
3     public static class UInt64LinkExtensions
4     {
5         public static bool IsFullPoint(this UInt64Link link) => Point<ulong>.IsFullPoint(link);
6         public static bool IsPartialPoint(this UInt64Link link) =>
7             ↳ Point<ulong>.IsPartialPoint(link);
8     }
9 }

```

./Platform.Data.Doublets/UInt64LinksExtensions.cs

```

1 using System;
2 using System.Text;
3 using System.Collections.Generic;
4 using Platform.Singletons;
5 using Platform.Data.Constants;
6 using Platform.Data.Exceptions;
7 using Platform.Data.Doublets.Unicode;
8
9 namespace Platform.Data.Doublets
10 {
11     public static class UInt64LinksExtensions
12     {
13         public static readonly LinksCombinedConstants<bool, ulong, int> Constants =
14             ↳ Default<LinksCombinedConstants<bool, ulong, int>>.Instance;
15
16         public static void UseUnicode(this ILinks<ulong> links) => UnicodeMap.InitNew(links);
17
18         public static void EnsureEachLinkExists(this ILinks<ulong> links, IList<ulong> sequence)
19         {

```



```

19     if (sequence == null)
20     {
21         return;
22     }
23     for (var i = 0; i < sequence.Count; i++)
24     {
25         if (!links.Exists(sequence[i]))
26         {
27             throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
28                 ↪ $"sequence[{i}]");
29         }
30     }
31 }
32 public static void EnsureEachLinkIsAnyOrExists(this ILinks<ulong> links, IList<ulong>
33     ↪ sequence)
34 {
35     if (sequence == null)
36     {
37         return;
38     }
39     for (var i = 0; i < sequence.Count; i++)
40     {
41         if (sequence[i] != Constants.Any && !links.Exists(sequence[i]))
42         {
43             throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
44                 ↪ $"sequence[{i}]");
45         }
46     }
47 }
48 public static bool AnyLinkIsAny(this ILinks<ulong> links, params ulong[] sequence)
49 {
50     if (sequence == null)
51     {
52         return false;
53     }
54     var constants = links.Constants;
55     for (var i = 0; i < sequence.Length; i++)
56     {
57         if (sequence[i] == constants.Any)
58         {
59             return true;
60         }
61     }
62     return false;
63 }
64 public static string FormatStructure(this ILinks<ulong> links, ulong linkIndex,
65     ↪ Func<UInt64Link, bool> isElement, bool renderIndex = false, bool renderDebug = false)
66 {
67     var sb = new StringBuilder();
68     var visited = new HashSet<ulong>();
69     links.AppendStructure(sb, visited, linkIndex, isElement, (innerSb, link) =>
70         ↪ innerSb.Append(link.Index), renderIndex, renderDebug);
71     return sb.ToString();
72 }
73 public static string FormatStructure(this ILinks<ulong> links, ulong linkIndex,
74     ↪ Func<UInt64Link, bool> isElement, Action<StringBuilder, UInt64Link> appendElement,
75     ↪ bool renderIndex = false, bool renderDebug = false)
76 {
77     var sb = new StringBuilder();
78     var visited = new HashSet<ulong>();
79     links.AppendStructure(sb, visited, linkIndex, isElement, appendElement, renderIndex,
80         ↪ renderDebug);
81     return sb.ToString();
82 }
83 public static void AppendStructure(this ILinks<ulong> links, StringBuilder sb,
84     ↪ HashSet<ulong> visited, ulong linkIndex, Func<UInt64Link, bool> isElement,
85     ↪ Action<StringBuilder, UInt64Link> appendElement, bool renderIndex = false, bool
86     ↪ renderDebug = false)
87 {
88     if (sb == null)
89     {
90         throw new ArgumentNullException(nameof(sb));
91     }
92 }

```

```

86         if (linkIndex == Constants.Null || linkIndex == Constants.Any || linkIndex ==
            ↪ Constants.Itself)
87         {
88             return;
89         }
90         if (links.Exists(linkIndex))
91         {
92             if (visited.Add(linkIndex))
93             {
94                 sb.Append('(');
95                 var link = new UInt64Link(links.GetLink(linkIndex));
96                 if (renderIndex)
97                 {
98                     sb.Append(link.Index);
99                     sb.Append(':');
100                 }
101                 if (link.Source == link.Index)
102                 {
103                     sb.Append(link.Index);
104                 }
105                 else
106                 {
107                     var source = new UInt64Link(links.GetLink(link.Source));
108                     if (isElement(source))
109                     {
110                         appendElement(sb, source);
111                     }
112                     else
113                     {
114                         links.AppendStructure(sb, visited, source.Index, isElement,
                            ↪ appendElement, renderIndex);
115                     }
116                 }
117                 sb.Append(' ');
118                 if (link.Target == link.Index)
119                 {
120                     sb.Append(link.Index);
121                 }
122                 else
123                 {
124                     var target = new UInt64Link(links.GetLink(link.Target));
125                     if (isElement(target))
126                     {
127                         appendElement(sb, target);
128                     }
129                     else
130                     {
131                         links.AppendStructure(sb, visited, target.Index, isElement,
                            ↪ appendElement, renderIndex);
132                     }
133                 }
134                 sb.Append(')');
135             }
136             else
137             {
138                 if (renderDebug)
139                 {
140                     sb.Append('*');
141                 }
142                 sb.Append(linkIndex);
143             }
144         }
145         else
146         {
147             if (renderDebug)
148             {
149                 sb.Append('~');
150             }
151             sb.Append(linkIndex);
152         }
153     }
154 }
155 }

```

./Platform.Data.Doublets/UInt64LinksTransactionsLayer.cs

```

1 using System;
2 using System.Linq;
3 using System.Collections.Generic;

```

```

4 using System.IO;
5 using System.Runtime.CompilerServices;
6 using System.Threading;
7 using System.Threading.Tasks;
8 using Platform.Disposables;
9 using Platform.Timestamps;
10 using Platform.Unsafe;
11 using Platform.IO;
12 using Platform.Data.Doublets.Decorators;
13
14 namespace Platform.Data.Doublets
15 {
16     public class UInt64LinksTransactionsLayer : LinksDisposableDecoratorBase<ulong> //-V3073
17     {
18         /// <remarks>
19         /// Альтернативные варианты хранения трансформации (элемента транзакции):
20         ///
21         /// private enum TransitionType
22         /// {
23         ///     Creation,
24         ///     UpdateOf,
25         ///     UpdateTo,
26         ///     Deletion
27         /// }
28         ///
29         /// private struct Transition
30         /// {
31         ///     public ulong TransactionId;
32         ///     public UniqueTimestamp Timestamp;
33         ///     public TransactionItemType Type;
34         ///     public Link Source;
35         ///     public Link Linker;
36         ///     public Link Target;
37         /// }
38         ///
39         /// Или
40         ///
41         /// public struct TransitionHeader
42         /// {
43         ///     public ulong TransactionIdCombined;
44         ///     public ulong TimestampCombined;
45         ///
46         ///     public ulong TransactionId
47         ///     {
48         ///         get
49         ///         {
50         ///             return (ulong) mask & TransactionIdCombined;
51         ///         }
52         ///     }
53         ///
54         ///     public UniqueTimestamp Timestamp
55         ///     {
56         ///         get
57         ///         {
58         ///             return (UniqueTimestamp)mask & TransactionIdCombined;
59         ///         }
60         ///     }
61         ///
62         ///     public TransactionItemType Type
63         ///     {
64         ///         get
65         ///         {
66         ///             // Использовать по одному биту из TransactionId и Timestamp,
67         ///             // для значения в 2 бита, которое представляет тип операции
68         ///             throw new NotImplementedException();
69         ///         }
70         ///     }
71         /// }
72         ///
73         /// private struct Transition
74         /// {
75         ///     public TransitionHeader Header;
76         ///     public Link Source;
77         ///     public Link Linker;
78         ///     public Link Target;
79         /// }
80         ///
81         /// </remarks>

```

```

82 public struct Transition
83 {
84     public static readonly long Size = Structure<Transition>.Size;
85
86     public readonly ulong TransactionId;
87     public readonly UInt64Link Before;
88     public readonly UInt64Link After;
89     public readonly Timestamp Timestamp;
90
91     public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
92     ↪ transactionId, UInt64Link before, UInt64Link after)
93     {
94         TransactionId = transactionId;
95         Before = before;
96         After = after;
97         Timestamp = uniqueTimestampFactory.Create();
98     }
99
100     public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
101     ↪ transactionId, UInt64Link before)
102     : this(uniqueTimestampFactory, transactionId, before, default)
103     {
104     }
105
106     public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong transactionId
107     ↪ : this(uniqueTimestampFactory, transactionId, default, default)
108     {
109     }
110
111     public override string ToString() => $"{Timestamp} {TransactionId}: {Before} =>
112     ↪ {After}";
113 }
114
115 /// <remarks>
116 /// Другие варианты реализации транзакций (атомарности):
117 /// 1. Разделение хранения значения связи ((Source Target) или (Source Linker
118 ↪ Target)) и индексов.
119 /// 2. Хранение трансформаций/операций в отдельном хранилище Links, но дополнительно
120 ↪ потребуется решить вопрос
121 /// со ссылками на внешние идентификаторы, или как-то иначе решить вопрос с
122 ↪ пересечениями идентификаторов.
123 ///
124 /// Где хранить промежуточный список транзакций?
125 ///
126 /// В оперативной памяти:
127 /// Минусы:
128 /// 1. Может усложнить систему, если она будет функционировать самостоятельно,
129 /// так как нужно отдельно выделять память под список трансформаций.
130 /// 2. Выделенной оперативной памяти может не хватить, в том случае,
131 /// если транзакция использует слишком много трансформаций.
132 /// -> Можно использовать жёсткий диск для слишком длинных транзакций.
133 /// -> Максимальный размер списка трансформаций можно ограничить / задать
134 ↪ константой.
135 /// 3. При подтверждении транзакции (Commit) все трансформации записываются разом
136 ↪ создавая задержку.
137 ///
138 /// На жёстком диске:
139 /// Минусы:
140 /// 1. Длительный отклик, на запись каждой трансформации.
141 /// 2. Лог транзакций дополнительно наполняется отменёнными транзакциями.
142 /// -> Это может решаться упаковкой/исключением дублирующих операций.
143 /// -> Также это может решаться тем, что короткие транзакции вообще
144 ↪ не будут записываться в случае отката.
145 /// 3. Перед тем как выполнять отмену операций транзакции нужно дождаться пока все
146 ↪ операции (трансформации)
147 ↪ будут записаны в лог.
148 ///
149 /// </remarks>
150 public class Transaction : DisposableBase
151 {
152     private readonly Queue<Transition> _transitions;
153     private readonly UInt64LinksTransactionsLayer _layer;
154     public bool IsCommitted { get; private set; }
155     public bool IsReverted { get; private set; }
156
157     public Transaction(UInt64LinksTransactionsLayer layer)
158     {
159         _layer = layer;
160         if (_layer._currentTransactionId != 0)

```

```

152     {
153         throw new NotSupportedException("Nested transactions not supported.");
154     }
155     IsCommitted = false;
156     IsReverted = false;
157     _transitions = new Queue<Transition>();
158     SetCurrentTransaction(layer, this);
159 }
160
161 public void Commit()
162 {
163     EnsureTransactionAllowsWriteOperations(this);
164     while (_transitions.Count > 0)
165     {
166         var transition = _transitions.Dequeue();
167         _layer._transitions.Enqueue(transition);
168     }
169     _layer._lastCommittedTransactionId = _layer._currentTransactionId;
170     IsCommitted = true;
171 }
172
173 private void Revert()
174 {
175     EnsureTransactionAllowsWriteOperations(this);
176     var transitionsToRevert = new Transition[_transitions.Count];
177     _transitions.CopyTo(transitionsToRevert, 0);
178     for (var i = transitionsToRevert.Length - 1; i >= 0; i--)
179     {
180         _layer.RevertTransition(transitionsToRevert[i]);
181     }
182     IsReverted = true;
183 }
184
185 public static void SetCurrentTransaction(UInt64LinksTransactionsLayer layer,
    ↪ Transaction transaction)
186 {
187     layer._currentTransactionId = layer._lastCommittedTransactionId + 1;
188     layer._currentTransactionTransitions = transaction._transitions;
189     layer._currentTransaction = transaction;
190 }
191
192 public static void EnsureTransactionAllowsWriteOperations(Transaction transaction)
193 {
194     if (transaction.IsReverted)
195     {
196         throw new InvalidOperationException("Transation is reverted.");
197     }
198     if (transaction.IsCommitted)
199     {
200         throw new InvalidOperationException("Transation is committed.");
201     }
202 }
203
204 protected override void Dispose(bool manual, bool wasDisposed)
205 {
206     if (!wasDisposed && _layer != null && !_layer.IsDisposed)
207     {
208         if (!IsCommitted && !IsReverted)
209         {
210             Revert();
211         }
212         _layer.ResetCurrentTransation();
213     }
214 }
215
216
217 public static readonly TimeSpan DefaultPushDelay = TimeSpan.FromSeconds(0.1);
218
219 private readonly string _logAddress;
220 private readonly FileStream _log;
221 private readonly Queue<Transition> _transitions;
222 private readonly UniqueTimestampFactory _uniqueTimestampFactory;
223 private Task _transitionsPusher;
224 private Transition _lastCommittedTransition;
225 private ulong _currentTransactionId;
226 private Queue<Transition> _currentTransactionTransitions;
227 private Transaction _currentTransaction;
228 private ulong _lastCommittedTransactionId;
229
230 public UInt64LinksTransactionsLayer(ILinks<ulong> links, string logAddress)

```

```

231         : base(links)
232     {
233         if (string.IsNullOrEmpty(logAddress))
234         {
235             throw new ArgumentNullException(nameof(logAddress));
236         }
237         // В первой строке файла хранится последняя закоммиченную транзакцию.
238         // При запуске это используется для проверки удачного закрытия файла лога.
239         // In the first line of the file the last committed transaction is stored.
240         // On startup, this is used to check that the log file is successfully closed.
241         var lastCommittedTransition = FileHelpers.ReadFirstOrDefault<Transition>(logAddress);
242         var lastWrittenTransition = FileHelpers.ReadLastOrDefault<Transition>(logAddress);
243         if (!lastCommittedTransition.Equals(lastWrittenTransition))
244         {
245             Dispose();
246             throw new NotSupportedException("Database is damaged, autorecovery is not
                ↳ supported yet.");
247         }
248         if (lastCommittedTransition.Equals(default(Transition)))
249         {
250             FileHelpers.WriteFirst(logAddress, lastCommittedTransition);
251         }
252         _lastCommittedTransition = lastCommittedTransition;
253         // TODO: Think about a better way to calculate or store this value
254         var allTransitions = FileHelpers.ReadAll<Transition>(logAddress);
255         _lastCommittedTransactionId = allTransitions.Max(x => x.TransactionId);
256         _uniqueTimestampFactory = new UniqueTimestampFactory();
257         _logAddress = logAddress;
258         _log = FileHelpers.Append(logAddress);
259         _transitions = new Queue<Transition>();
260         _transitionsPusher = new Task(TransitionsPusher);
261         _transitionsPusher.Start();
262     }
263
264     public IList<ulong> GetLinkValue(ulong link) => Links.GetLink(link);
265
266     public override ulong Create()
267     {
268         var createdLinkIndex = Links.Create();
269         var createdLink = new UInt64Link(Links.GetLink(createdLinkIndex));
270         CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
                ↳ default, createdLink));
271         return createdLinkIndex;
272     }
273
274     public override ulong Update(IList<ulong> parts)
275     {
276         var linkIndex = parts[Constants.IndexPart];
277         var beforeLink = new UInt64Link(Links.GetLink(linkIndex));
278         linkIndex = Links.Update(parts);
279         var afterLink = new UInt64Link(Links.GetLink(linkIndex));
280         CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
                ↳ beforeLink, afterLink));
281         return linkIndex;
282     }
283
284     public override void Delete(ulong link)
285     {
286         var deletedLink = new UInt64Link(Links.GetLink(link));
287         Links.Delete(link);
288         CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
                ↳ deletedLink, default));
289     }
290
291     [MethodImpl(MethodImplOptions.AggressiveInlining)]
292     private Queue<Transition> GetCurrentTransitions() => _currentTransactionTransitions ??
        ↳ _transitions;
293
294     private void CommitTransition(Transition transition)
295     {
296         if (_currentTransaction != null)
297         {
298             Transaction.EnsureTransactionAllowsWriteOperations(_currentTransaction);
299         }
300         var transitions = GetCurrentTransitions();
301         transitions.Enqueue(transition);
302     }
303

```

```

304 private void RevertTransition(Transition transition)
305 {
306     if (transition.After.IsNull()) // Revert Deletion with Creation
307     {
308         Links.Create();
309     }
310     else if (transition.Before.IsNull()) // Revert Creation with Deletion
311     {
312         Links.Delete(transition.After.Index);
313     }
314     else // Revert Update
315     {
316         Links.Update(new[] { transition.After.Index, transition.Before.Source,
317                               ↪ transition.Before.Target });
318     }
319 }
320 private void ResetCurrentTransation()
321 {
322     _currentTransactionId = 0;
323     _currentTransactionTransitions = null;
324     _currentTransaction = null;
325 }
326 private void PushTransitions()
327 {
328     if (_log == null || _transitions == null)
329     {
330         return;
331     }
332     for (var i = 0; i < _transitions.Count; i++)
333     {
334         var transition = _transitions.Dequeue();
335
336         _log.Write(transition);
337         _lastCommittedTransition = transition;
338     }
339 }
340 private void TransitionsPusher()
341 {
342     while (!IsDisposed && _transitionsPusher != null)
343     {
344         Thread.Sleep(DefaultPushDelay);
345         PushTransitions();
346     }
347 }
348 public Transaction BeginTransaction() => new Transaction(this);
349 private void DisposeTransitions()
350 {
351     try
352     {
353         var pusher = _transitionsPusher;
354         if (pusher != null)
355         {
356             _transitionsPusher = null;
357             pusher.Wait();
358         }
359         if (_transitions != null)
360         {
361             PushTransitions();
362         }
363         _log.DisposeIfPossible();
364         FileHelpers.WriteFirst(_logAddress, _lastCommittedTransition);
365     }
366     catch
367     {
368     }
369 }
370 #region DisposalBase
371 protected override void Dispose(bool manual, bool wasDisposed)
372 {
373     if (!wasDisposed)
374     {
375         DisposeTransitions();
376     }
377 }

```

```

382     }
383     base.Dispose(manual, wasDisposed);
384 }
385
386 #endregion
387 }
388 }

```

./Platform.Data.Doublets/UnaryNumbers/AddressToUnaryNumberConverter.cs

```

1  using System.Collections.Generic;
2  using Platform.Interfaces;
3  using Platform.Reflection;
4  using Platform.Numbers;
5
6  namespace Platform.Data.Doublets.UnaryNumbers
7  {
8      public class AddressToUnaryNumberConverter<TLink> : LinksOperatorBase<TLink>,
9          ⇨ IConverter<TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
12             ⇨ EqualityComparer<TLink>.Default;
13
14         private readonly IConverter<int, TLink> _powerOf2ToUnaryNumberConverter;
15
16         public AddressToUnaryNumberConverter(ILinks<TLink> links, IConverter<int, TLink>
17             ⇨ powerOf2ToUnaryNumberConverter) : base(links) => _powerOf2ToUnaryNumberConverter =
18             ⇨ powerOf2ToUnaryNumberConverter;
19
20         public TLink Convert(TLink sourceAddress)
21         {
22             var number = sourceAddress;
23             var nullConstant = Links.Constants.Null;
24             var one = Integer<TLink>.One;
25             var target = nullConstant;
26             for (int i = 0; !_equalityComparer.Equals(number, default) && i <
27                 ⇨ Type<TLink>.BitsLength; i++)
28             {
29                 if (_equalityComparer.Equals(Arithmetic.And(number, one), one))
30                 {
31                     target = _equalityComparer.Equals(target, nullConstant)
32                         ? _powerOf2ToUnaryNumberConverter.Convert(i)
33                         : Links.GetOrCreate(_powerOf2ToUnaryNumberConverter.Convert(i), target);
34                 }
35                 number = (Integer<TLink>)((ulong)(Integer<TLink>)number >> 1); // Should be
36                 ⇨ Bit.ShiftRight(number, 1)
37             }
38             return target;
39         }
40     }
41 }

```

./Platform.Data.Doublets/UnaryNumbers/LinkToItsFrequencyNumberConveter.cs

```

1  using System;
2  using System.Collections.Generic;
3  using Platform.Interfaces;
4
5  namespace Platform.Data.Doublets.UnaryNumbers
6  {
7      public class LinkToItsFrequencyNumberConveter<TLink> : LinksOperatorBase<TLink>,
8          ⇨ IConverter<Doublet<TLink>, TLink>
9      {
10         private static readonly EqualityComparer<TLink> _equalityComparer =
11             ⇨ EqualityComparer<TLink>.Default;
12
13         private readonly IPropertyOperator<TLink, TLink> _frequencyPropertyOperator;
14         private readonly IConverter<TLink> _unaryNumberToAddressConverter;
15
16         public LinkToItsFrequencyNumberConveter(
17             ILinks<TLink> links,
18             IPropertyOperator<TLink, TLink> frequencyPropertyOperator,
19             IConverter<TLink> unaryNumberToAddressConverter)
20             : base(links)
21         {
22             _frequencyPropertyOperator = frequencyPropertyOperator;
23             _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
24         }
25
26         public TLink Convert(Doublet<TLink> doublet)
27         {
28             var link = Links.SearchOrDefault(doublet.Source, doublet.Target);

```



```

27         if (_equalityComparer.Equals(link, Links.Constants.Null))
28         {
29             throw new ArgumentException($"Link ({doublet}) not found.", nameof(doublet));
30         }
31         var frequency = _frequencyPropertyOperator.Get(link);
32         if (_equalityComparer.Equals(frequency, default))
33         {
34             return default;
35         }
36         var frequencyNumber = Links.GetSource(frequency);
37         return _unaryNumberToAddressConverter.Convert(frequencyNumber);
38     }
39 }
40 }

```

#### ./Platform.Data.Doublets/UnaryNumbers/PowerOf2ToUnaryNumberConverter.cs

```

1  using System.Collections.Generic;
2  using Platform.Exceptions;
3  using Platform.Interfaces;
4  using Platform.Ranges;
5
6  namespace Platform.Data.Doublets.UnaryNumbers
7  {
8      public class PowerOf2ToUnaryNumberConverter<TLink> : LinksOperatorBase<TLink>,
9          ⇨ IConverter<int, TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
12             ⇨ EqualityComparer<TLink>.Default;
13
14         private readonly TLink[] _unaryNumberPowersOf2;
15
16         public PowerOf2ToUnaryNumberConverter(ILinks<TLink> links, TLink one) : base(links)
17         {
18             _unaryNumberPowersOf2 = new TLink[64];
19             _unaryNumberPowersOf2[0] = one;
20         }
21
22         public TLink Convert(int power)
23         {
24             Ensure.Always.ArgumentInRange(power, new Range<int>(0, _unaryNumberPowersOf2.Length
25                 ⇨ - 1), nameof(power));
26             if (!_equalityComparer.Equals(_unaryNumberPowersOf2[power], default))
27             {
28                 return _unaryNumberPowersOf2[power];
29             }
30             var previousPowerOf2 = Convert(power - 1);
31             var powerOf2 = Links.GetOrCreate(previousPowerOf2, previousPowerOf2);
32             _unaryNumberPowersOf2[power] = powerOf2;
33             return powerOf2;
34         }
35     }
36 }

```

#### ./Platform.Data.Doublets/UnaryNumbers/UnaryNumberToAddressAddOperationConverter.cs

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using Platform.Interfaces;
4  using Platform.Numbers;
5
6  namespace Platform.Data.Doublets.UnaryNumbers
7  {
8      public class UnaryNumberToAddressAddOperationConverter<TLink> : LinksOperatorBase<TLink>,
9          ⇨ IConverter<TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
12             ⇨ EqualityComparer<TLink>.Default;
13
14         private Dictionary<TLink, TLink> _unaryToUInt64;
15         private readonly TLink _unaryOne;
16
17         public UnaryNumberToAddressAddOperationConverter(ILinks<TLink> links, TLink unaryOne)
18             : base(links)
19         {
20             _unaryOne = unaryOne;
21             InitUnaryToUInt64();
22         }
23
24         private void InitUnaryToUInt64()
25         {
26
27         }
28     }
29 }

```

```

24     var one = Integer<TLink>.One;
25     _unaryToUInt64 = new Dictionary<TLink, TLink>
26     {
27         { _unaryOne, one }
28     };
29     var unary = _unaryOne;
30     var number = one;
31     for (var i = 1; i < 64; i++)
32     {
33         unary = Links.GetOrCreate(unary, unary);
34         number = Double(number);
35         _unaryToUInt64.Add(unary, number);
36     }
37 }
38
39 public TLink Convert(TLink unaryNumber)
40 {
41     if (_equalityComparer.Equals(unaryNumber, default))
42     {
43         return default;
44     }
45     if (_equalityComparer.Equals(unaryNumber, _unaryOne))
46     {
47         return Integer<TLink>.One;
48     }
49     var source = Links.GetSource(unaryNumber);
50     var target = Links.GetTarget(unaryNumber);
51     if (_equalityComparer.Equals(source, target))
52     {
53         return _unaryToUInt64[unaryNumber];
54     }
55     else
56     {
57         var result = _unaryToUInt64[source];
58         TLink lastValue;
59         while (!_unaryToUInt64.TryGetValue(target, out lastValue))
60         {
61             source = Links.GetSource(target);
62             result = Arithmetic<TLink>.Add(result, _unaryToUInt64[source]);
63             target = Links.GetTarget(target);
64         }
65         result = Arithmetic<TLink>.Add(result, lastValue);
66         return result;
67     }
68 }
69
70 [MethodImpl(MethodImplOptions.AggressiveInlining)]
71 private static TLink Double(TLink number) => (Integer<TLink>)((Integer<TLink>)number *
    ↪ 2UL);
72 }
73 }

```

./Platform.Data.Doublets/UnaryNumbers/UnaryNumberToAddressOrOperationConverter.cs

```

1  using System.Collections.Generic;
2  using Platform.Interfaces;
3  using Platform.Reflection;
4  using Platform.Numbers;
5  using System.Runtime.CompilerServices;
6
7  namespace Platform.Data.Doublets.UnaryNumbers
8  {
9      public class UnaryNumberToAddressOrOperationConverter<TLink> : LinksOperatorBase<TLink>,
    ↪ IConverter<TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
    ↪ EqualityComparer<TLink>.Default;
12
13         private readonly IDictionary<TLink, int> _unaryNumberPowerOf2Indicies;
14
15         public UnaryNumberToAddressOrOperationConverter(ILinks<TLink> links, IConverter<int,
    ↪ TLink> powerOf2ToUnaryNumberConverter)
    : base(links)
16         {
17             _unaryNumberPowerOf2Indicies = new Dictionary<TLink, int>();
18             for (int i = 0; i < Type<TLink>.BitsLength; i++)
19             {
20                 _unaryNumberPowerOf2Indicies.Add(powerOf2ToUnaryNumberConverter.Convert(i), i);
21             }
22         }
23     }

```

```

24
25 public TLink Convert(TLink sourceNumber)
26 {
27     var nullConstant = Links.Constants.Null;
28     var source = sourceNumber;
29     var target = nullConstant;
30     if (!_equalityComparer.Equals(source, nullConstant))
31     {
32         while (true)
33         {
34             if (_unaryNumberPowerOf2Indicies.TryGetValue(source, out int powerOf2Index))
35             {
36                 SetBit(ref target, powerOf2Index);
37                 break;
38             }
39             else
40             {
41                 powerOf2Index = _unaryNumberPowerOf2Indicies[Links.GetSource(source)];
42                 SetBit(ref target, powerOf2Index);
43                 source = Links.GetTarget(source);
44             }
45         }
46     }
47     return target;
48 }
49
50 [MethodImpl(MethodImplOptions.AggressiveInlining)]
51 private static void SetBit(ref TLink target, int powerOf2Index) => target =
    ↪ (Integer<TLink>)((Integer<TLink>)target | 1UL << powerOf2Index); // Should be
    ↪ Math.Or(target, Math.ShiftLeft(One, powerOf2Index))
52 }
53 }

```

./Platform.Data.Doublets/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 namespace Platform.Data.Doublets.Unicode
9 {
10     public class UnicodeMap
11     {
12         public static readonly ulong FirstCharLink = 1;
13         public static readonly ulong LastCharLink = FirstCharLink + char.MaxValue;
14         public static readonly ulong MapSize = 1 + char.MaxValue;
15
16         private readonly ILinks<ulong> _links;
17         private bool _initialized;
18
19         public UnicodeMap(ILinks<ulong> links) => _links = links;
20
21         public static UnicodeMap InitNew(ILinks<ulong> links)
22         {
23             var map = new UnicodeMap(links);
24             map.Init();
25             return map;
26         }
27
28         public void Init()
29         {
30             if (_initialized)
31             {
32                 return;
33             }
34             _initialized = true;
35             var firstLink = _links.CreatePoint();
36             if (firstLink != FirstCharLink)
37             {
38                 _links.Delete(firstLink);
39             }
40             else
41             {
42                 for (var i = FirstCharLink + 1; i <= LastCharLink; i++)
43                 {
44                     // From NIL to It (NIL -> Character) transformation meaning, (or infinite
45                     ↪ amount of NIL characters before actual Character)
46                     var createdLink = _links.CreatePoint();

```

```

46         _links.Update(createdLink, firstLink, createdLink);
47         if (createdLink != i)
48         {
49             throw new InvalidOperationException("Unable to initialize UTF 16
↳ table.");
50         }
51     }
52 }
53 }
54
55 // 0 - null link
56 // 1 - nil character (0 character)
57 // ...
58 // 65536 (0(1) + 65535 = 65536 possible values)
59
60 [MethodImpl(MethodImplOptions.AggressiveInlining)]
61 public static ulong FromCharToLink(char character) => (ulong)character + 1;
62
63 [MethodImpl(MethodImplOptions.AggressiveInlining)]
64 public static char FromLinkToChar(ulong link) => (char)(link - 1);
65
66 [MethodImpl(MethodImplOptions.AggressiveInlining)]
67 public static bool IsCharLink(ulong link) => link <= MapSize;
68
69 public static string FromLinksToString(IList<ulong> linksList)
70 {
71     var sb = new StringBuilder();
72     for (int i = 0; i < linksList.Count; i++)
73     {
74         sb.Append(FromLinkToChar(linksList[i]));
75     }
76     return sb.ToString();
77 }
78
79 public static string FromSequenceLinkToString(ulong link, ILinks<ulong> links)
80 {
81     var sb = new StringBuilder();
82     if (links.Exists(link))
83     {
84         StopableSequenceWalker.WalkRight(link, links.GetSource, links.GetTarget,
85             x => x <= MapSize || links.GetSource(x) == x || links.GetTarget(x) == x,
86             ↳ element =>
87             {
88                 sb.Append(FromLinkToChar(element));
89                 return true;
90             }
91         return sb.ToString();
92     }
93 }
94
95 public static ulong[] FromCharsToLinkArray(char[] chars) => FromCharsToLinkArray(chars,
↳ chars.Length);
96
97 public static ulong[] FromCharsToLinkArray(char[] chars, int count)
98 {
99     // char array to ulong array
100     var linksSequence = new ulong[count];
101     for (var i = 0; i < count; i++)
102     {
103         linksSequence[i] = FromCharToLink(chars[i]);
104     }
105     return linksSequence;
106 }
107
108 public static ulong[] FromStringToLinkArray(string sequence)
109 {
110     // char array to ulong array
111     var linksSequence = new ulong[sequence.Length];
112     for (var i = 0; i < sequence.Length; i++)
113     {
114         linksSequence[i] = FromCharToLink(sequence[i]);
115     }
116     return linksSequence;
117 }
118
119 public static List<ulong[]> FromStringToLinkArrayGroups(string sequence)
120 {
121     var result = new List<ulong[]>();
122     var offset = 0;

```

```

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

```

./Platform.Data.Doublets.Tests/ComparisonTests.cs

```

1 using System;
2 using System.Collections.Generic;
3 using Xunit;
4 using Platform.Diagnostics;
5
6 namespace Platform.Data.Doublets.Tests
7 {
8     public static class ComparisonTests
9     {

```

```

10     protected class UInt64Comparer : IComparer<ulong>
11     {
12         public int Compare(ulong x, ulong y) => x.CompareTo(y);
13     }
14
15     private static int Compare(ulong x, ulong y) => x.CompareTo(y);
16
17     [Fact]
18     public static void GreaterOrEqualPerformanceTest()
19     {
20         const int N = 1000000;
21
22         ulong x = 10;
23         ulong y = 500;
24
25         bool result = false;
26
27         var ts1 = Performance.Measure(() =>
28         {
29             for (int i = 0; i < N; i++)
30             {
31                 result = Compare(x, y) >= 0;
32             }
33         });
34
35         var comparer1 = Comparer<ulong>.Default;
36
37         var ts2 = Performance.Measure(() =>
38         {
39             for (int i = 0; i < N; i++)
40             {
41                 result = comparer1.Compare(x, y) >= 0;
42             }
43         });
44
45         Func<ulong, ulong, int> compareReference = comparer1.Compare;
46
47         var ts3 = Performance.Measure(() =>
48         {
49             for (int i = 0; i < N; i++)
50             {
51                 result = compareReference(x, y) >= 0;
52             }
53         });
54
55         var comparer2 = new UInt64Comparer();
56
57         var ts4 = Performance.Measure(() =>
58         {
59             for (int i = 0; i < N; i++)
60             {
61                 result = comparer2.Compare(x, y) >= 0;
62             }
63         });
64
65         Console.WriteLine($"{ts1} {ts2} {ts3} {ts4} {result}");
66     }
67 }
68

```

./Platform.Data.Doublets.Tests/DoubletLinksTests.cs

```

1  using System.Collections.Generic;
2  using Xunit;
3  using Platform.Reflection;
4  using Platform.Numbers;
5  using Platform.Memory;
6  using Platform.Scopes;
7  using Platform.Setters;
8  using Platform.Data.Doublets.ResizableDirectMemory;
9
10 namespace Platform.Data.Doublets.Tests
11 {
12     public static class DoubletLinksTests
13     {
14         [Fact]
15         public static void UInt64CRUDTest()
16         {
17             using (var scope = new Scope<Types<HeapResizableDirectMemory,
18                 ↳ ResizableDirectMemoryLinks<ulong>>>())
19             {
20

```

```

19         scope.Use<ILinks<ulong>>>().TestCRUDOperations();
20     }
21 }
22
23 [Fact]
24 public static void UInt32CRUDTest()
25 {
26     using (var scope = new Scope<Types<HeapResizableDirectMemory,
27         ↳ ResizableDirectMemoryLinks<uint>>>())
28     {
29         scope.Use<ILinks<uint>>>().TestCRUDOperations();
30     }
31 }
32
33 [Fact]
34 public static void UInt16CRUDTest()
35 {
36     using (var scope = new Scope<Types<HeapResizableDirectMemory,
37         ↳ ResizableDirectMemoryLinks<ushort>>>())
38     {
39         scope.Use<ILinks<ushort>>>().TestCRUDOperations();
40     }
41 }
42
43 [Fact]
44 public static void UInt8CRUDTest()
45 {
46     using (var scope = new Scope<Types<HeapResizableDirectMemory,
47         ↳ ResizableDirectMemoryLinks<byte>>>())
48     {
49         scope.Use<ILinks<byte>>>().TestCRUDOperations();
50     }
51 }
52
53 private static void TestCRUDOperations<T>(this ILinks<T> links)
54 {
55     var constants = links.Constants;
56
57     var equalityComparer = EqualityComparer<T>.Default;
58
59     // Create Link
60     Assert.True(equalityComparer.Equals(links.Count(), Integer<T>.Zero));
61
62     var setter = new Setter<T>(constants.Null);
63     links.Each(constants.Any, constants.Any, setter.SetAndReturnTrue);
64
65     Assert.True(equalityComparer.Equals(setter.Result, constants.Null));
66
67     var linkAddress = links.Create();
68
69     var link = new Link<T>(links.GetLink(linkAddress));
70
71     Assert.True(link.Count == 3);
72     Assert.True(equalityComparer.Equals(link.Index, linkAddress));
73     Assert.True(equalityComparer.Equals(link.Source, constants.Null));
74     Assert.True(equalityComparer.Equals(link.Target, constants.Null));
75
76     Assert.True(equalityComparer.Equals(links.Count(), Integer<T>.One));
77
78     // Get first link
79     setter = new Setter<T>(constants.Null);
80     links.Each(constants.Any, constants.Any, setter.SetAndReturnFalse);
81
82     Assert.True(equalityComparer.Equals(setter.Result, linkAddress));
83
84     // Update link to reference itself
85     links.Update(linkAddress, linkAddress, linkAddress);
86
87     link = new Link<T>(links.GetLink(linkAddress));
88
89     Assert.True(equalityComparer.Equals(link.Source, linkAddress));
90     Assert.True(equalityComparer.Equals(link.Target, linkAddress));
91
92     // Update link to reference null (prepare for delete)
93     var updated = links.Update(linkAddress, constants.Null, constants.Null);
94
95     Assert.True(equalityComparer.Equals(updated, linkAddress));
96
97     link = new Link<T>(links.GetLink(linkAddress));

```

```

95     Assert.True(equalityComparer.Equals(link.Source, constants.Null));
96     Assert.True(equalityComparer.Equals(link.Target, constants.Null));
97
98     // Delete link
99     links.Delete(linkAddress);
100
101     Assert.True(equalityComparer.Equals(links.Count(), Integer<T>.Zero));
102
103     setter = new Setter<T>(constants.Null);
104     links.Each(constants.Any, constants.Any, setter.SetAndReturnTrue);
105
106     Assert.True(equalityComparer.Equals(setter.Result, constants.Null));
107 }
108
109 [Fact]
110 public static void UInt64RawNumbersCRUDTest()
111 {
112     using (var scope = new Scope<Types<HeapResizableDirectMemory,
113         ↳ ResizableDirectMemoryLinks<ulong>>>())
114     {
115         scope.Use<ILinks<ulong>>().TestRawNumbersCRUDOperations();
116     }
117 }
118
119 [Fact]
120 public static void UInt32RawNumbersCRUDTest()
121 {
122     using (var scope = new Scope<Types<HeapResizableDirectMemory,
123         ↳ ResizableDirectMemoryLinks<uint>>>())
124     {
125         scope.Use<ILinks<uint>>().TestRawNumbersCRUDOperations();
126     }
127 }
128
129 [Fact]
130 public static void UInt16RawNumbersCRUDTest()
131 {
132     using (var scope = new Scope<Types<HeapResizableDirectMemory,
133         ↳ ResizableDirectMemoryLinks<ushort>>>())
134     {
135         scope.Use<ILinks<ushort>>().TestRawNumbersCRUDOperations();
136     }
137 }
138
139 [Fact]
140 public static void UInt8RawNumbersCRUDTest()
141 {
142     using (var scope = new Scope<Types<HeapResizableDirectMemory,
143         ↳ ResizableDirectMemoryLinks<byte>>>())
144     {
145         scope.Use<ILinks<byte>>().TestRawNumbersCRUDOperations();
146     }
147 }
148
149 private static void TestRawNumbersCRUDOperations<T>(this ILinks<T> links)
150 {
151     // Constants
152     var constants = links.Constants;
153     var equalityComparer = EqualityComparer<T>.Default;
154
155     var h106E = new Hybrid<T>(106L, isExternal: true);
156     var h107E = new Hybrid<T>(-char.ConvertFromUtf32(107)[0]);
157     var h108E = new Hybrid<T>(-108L);
158
159     Assert.Equal(106L, h106E.AbsoluteValue);
160     Assert.Equal(107L, h107E.AbsoluteValue);
161     Assert.Equal(108L, h108E.AbsoluteValue);
162
163     // Create Link (External -> External)
164     var linkAddress1 = links.Create();
165
166     links.Update(linkAddress1, h106E, h108E);
167
168     var link1 = new Link<T>(links.GetLink(linkAddress1));
169
170     Assert.True(equalityComparer.Equals(link1.Source, h106E));
171     Assert.True(equalityComparer.Equals(link1.Target, h108E));

```



```

170 // Create Link (Internal -> External)
171 var linkAddress2 = links.Create();
172
173 links.Update(linkAddress2, linkAddress1, h108E);
174
175 var link2 = new Link<T>(links.GetLink(linkAddress2));
176
177 Assert.True(equalityComparer.Equals(link2.Source, linkAddress1));
178 Assert.True(equalityComparer.Equals(link2.Target, h108E));
179
180 // Create Link (Internal -> Internal)
181 var linkAddress3 = links.Create();
182
183 links.Update(linkAddress3, linkAddress1, linkAddress2);
184
185 var link3 = new Link<T>(links.GetLink(linkAddress3));
186
187 Assert.True(equalityComparer.Equals(link3.Source, linkAddress1));
188 Assert.True(equalityComparer.Equals(link3.Target, linkAddress2));
189
190 // Search for created link
191 var setter1 = new Setter<T>(constants.Null);
192 links.Each(h106E, h108E, setter1.SetAndReturnFalse);
193
194 Assert.True(equalityComparer.Equals(setter1.Result, linkAddress1));
195
196 // Search for nonexistent link
197 var setter2 = new Setter<T>(constants.Null);
198 links.Each(h106E, h107E, setter2.SetAndReturnFalse);
199
200 Assert.True(equalityComparer.Equals(setter2.Result, constants.Null));
201
202 // Update link to reference null (prepare for delete)
203 var updated = links.Update(linkAddress3, constants.Null, constants.Null);
204
205 Assert.True(equalityComparer.Equals(updated, linkAddress3));
206
207 link3 = new Link<T>(links.GetLink(linkAddress3));
208
209 Assert.True(equalityComparer.Equals(link3.Source, constants.Null));
210 Assert.True(equalityComparer.Equals(link3.Target, constants.Null));
211
212 // Delete link
213 links.Delete(linkAddress3);
214
215 Assert.True(equalityComparer.Equals(links.Count(), Integer<T>.Two));
216
217 var setter3 = new Setter<T>(constants.Null);
218 links.Each(constants.Any, constants.Any, setter3.SetAndReturnTrue);
219
220 Assert.True(equalityComparer.Equals(setter3.Result, linkAddress2));
221 }
222
223 // TODO: Test layers
224 }
225 }

```

#### ./Platform.Data.Doublets.Tests/EqualityTests.cs

```

1 using System;
2 using System.Collections.Generic;
3 using Xunit;
4 using Platform.Diagnostics;
5
6 namespace Platform.Data.Doublets.Tests
7 {
8     public static class EqualityTests
9     {
10         protected class UInt64EqualityComparer : IEqualityComparer<ulong>
11         {
12             public bool Equals(ulong x, ulong y) => x == y;
13
14             public int GetHashCode(ulong obj) => obj.GetHashCode();
15         }
16
17         private static bool Equals1<T>(T x, T y) => Equals(x, y);
18
19         private static bool Equals2<T>(T x, T y) => x.Equals(y);
20
21         private static bool Equals3(ulong x, ulong y) => x == y;
22
23         [Fact]

```

```

24 public static void EqualsPerformanceTest()
25 {
26     const int N = 1000000;
27
28     ulong x = 10;
29     ulong y = 500;
30
31     bool result = false;
32
33     var ts1 = Performance.Measure(() =>
34     {
35         for (int i = 0; i < N; i++)
36         {
37             result = Equals1(x, y);
38         }
39     });
40
41     var ts2 = Performance.Measure(() =>
42     {
43         for (int i = 0; i < N; i++)
44         {
45             result = Equals2(x, y);
46         }
47     });
48
49     var ts3 = Performance.Measure(() =>
50     {
51         for (int i = 0; i < N; i++)
52         {
53             result = Equals3(x, y);
54         }
55     });
56
57     var equalityComparer1 = EqualityComparer<ulong>.Default;
58
59     var ts4 = Performance.Measure(() =>
60     {
61         for (int i = 0; i < N; i++)
62         {
63             result = equalityComparer1.Equals(x, y);
64         }
65     });
66
67     var equalityComparer2 = new UInt64EqualityComparer();
68
69     var ts5 = Performance.Measure(() =>
70     {
71         for (int i = 0; i < N; i++)
72         {
73             result = equalityComparer2.Equals(x, y);
74         }
75     });
76
77     Func<ulong, ulong, bool> equalityComparer3 = equalityComparer2.Equals;
78
79     var ts6 = Performance.Measure(() =>
80     {
81         for (int i = 0; i < N; i++)
82         {
83             result = equalityComparer3(x, y);
84         }
85     });
86
87     var comparer = Comparer<ulong>.Default;
88
89     var ts7 = Performance.Measure(() =>
90     {
91         for (int i = 0; i < N; i++)
92         {
93             result = comparer.Compare(x, y) == 0;
94         }
95     });
96
97     Assert.True(ts2 < ts1);
98     Assert.True(ts3 < ts2);
99     Assert.True(ts5 < ts4);
100    Assert.True(ts5 < ts6);
101
102    Console.WriteLine($"{ts1} {ts2} {ts3} {ts4} {ts5} {ts6} {ts7} {result}");

```

```

103     }
104 }
105 }

```

./Platform.Data.Doublets.Tests/LinksTests.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.IO;
11 using Platform.Ranges;
12 using Platform.Random;
13 using Platform.Timestamps;
14 using Platform.Singletons;
15 using Platform.Counters;
16 using Platform.Diagnostics;
17 using Platform.Data.Constants;
18 using Platform.Data.Doublets.ResizableDirectMemory;
19 using Platform.Data.Doublets.Decorators;
20
21 namespace Platform.Data.Doublets.Tests
22 {
23     public static class LinksTests
24     {
25         private static readonly LinksCombinedConstants<bool, ulong, int> _constants =
26             ↪ Default<LinksCombinedConstants<bool, ulong, int>>.Instance;
27
28         private const long Iterations = 10 * 1024;
29
30         #region Concept
31
32         [Fact]
33         public static void MultipleCreateAndDeleteTest()
34         {
35             //const int N = 21;
36
37             using (var scope = new TempLinksTestScope())
38             {
39                 var links = scope.Links;
40
41                 for (var N = 0; N < 100; N++)
42                 {
43                     var random = new System.Random(N);
44
45                     var created = 0;
46                     var deleted = 0;
47
48                     for (var i = 0; i < N; i++)
49                     {
50                         var linksCount = links.Count();
51                         var createPoint = random.NextBoolean();
52
53                         if (linksCount > 2 && createPoint)
54                         {
55                             var linksAddressRange = new Range<ulong>(1, linksCount);
56                             var source = random.NextUInt64(linksAddressRange);
57                             var target = random.NextUInt64(linksAddressRange); //-V3086
58
59                             var resultLink = links.CreateAndUpdate(source, target);
60                             if (resultLink > linksCount)
61                             {
62                                 created++;
63                             }
64                         }
65                         else
66                         {
67                             links.Create();
68                             created++;
69                         }
70                     }
71
72                     Assert.True(created == (int)links.Count());
73
74                     for (var i = 0; i < N; i++)
75                     {

```

```

76         var link = (ulong)i + 1;
77         if (links.Exists(link))
78         {
79             links.Delete(link);
80             deleted++;
81         }
82     }
83
84     Assert.True(links.Count() == 0);
85 }
86
87 }
88
89 [Fact]
90 public static void CascadeUpdateTest()
91 {
92     var itself = _constants.Itself;
93
94     using (var scope = new TempLinksTestScope(useLog: true))
95     {
96         var links = scope.Links;
97
98         var l1 = links.Create();
99         var l2 = links.Create();
100
101         l2 = links.Update(l2, l2, l1, l2);
102
103         links.CreateAndUpdate(l2, itself);
104         links.CreateAndUpdate(l2, itself);
105
106         l2 = links.Update(l2, l1);
107
108         links.Delete(l2);
109
110         Global.Trash = links.Count();
111
112         links.Unsync.DisposeIfPossible(); // Close links to access log
113
114         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scope
            ↪ e.TempTransactionLogFilename);
115     }
116 }
117
118 [Fact]
119 public static void BasicTransactionLogTest()
120 {
121     using (var scope = new TempLinksTestScope(useLog: true))
122     {
123         var links = scope.Links;
124         var l1 = links.Create();
125         var l2 = links.Create();
126
127         Global.Trash = links.Update(l2, l2, l1, l2);
128
129         links.Delete(l1);
130
131         links.Unsync.DisposeIfPossible(); // Close links to access log
132
133         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scope
            ↪ e.TempTransactionLogFilename);
134     }
135 }
136
137 [Fact]
138 public static void TransactionAutoRevertedTest()
139 {
140     // Auto Reverted (Because no commit at transaction)
141     using (var scope = new TempLinksTestScope(useLog: true))
142     {
143         var links = scope.Links;
144         var transactionsLayer = (UInt64LinksTransactionsLayer)scope.MemoryAdapter;
145         using (var transaction = transactionsLayer.BeginTransaction())
146         {
147             var l1 = links.Create();
148             var l2 = links.Create();
149
150             links.Update(l2, l2, l1, l2);
151         }
152     }

```

```

153         Assert.Equal(OUL, links.Count());
154
155         links.Unsync.DisposeIfPossible();
156
157         var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(s_
158             ↪ cope.TempTransactionLogFilename);
159         Assert.Single(transitions);
160     }
161 }
162 [Fact]
163 public static void TransactionUserCodeErrorNoDataSavedTest()
164 {
165     // User Code Error (Autoreverted), no data saved
166     var itself = _constants.Itself;
167
168     TempLinksTestScope lastScope = null;
169     try
170     {
171         using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
172             ↪ useLog: true))
173         {
174             var links = scope.Links;
175             var transactionsLayer = (UInt64LinksTransactionsLayer)((LinksDisposableDecor_
176                 ↪ atorBase<ulong>)links.Unsync).Links;
177             using (var transaction = transactionsLayer.BeginTransaction())
178             {
179                 var l1 = links.CreateAndUpdate(itself, itself);
180                 var l2 = links.CreateAndUpdate(itself, itself);
181
182                 l2 = links.Update(l2, l2, l1, l2);
183
184                 links.CreateAndUpdate(l2, itself);
185                 links.CreateAndUpdate(l2, itself);
186
187                 //Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transi_
188                 ↪ tion>(scope.TempTransactionLogFilename);
189
190                 l2 = links.Update(l2, l1);
191
192                 links.Delete(l2);
193
194                 ExceptionThrower();
195
196                 transaction.Commit();
197             }
198             Global.Trash = links.Count();
199         }
200     }
201     catch
202     {
203         Assert.False(lastScope == null);
204
205         var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(l_
206             ↪ astScope.TempTransactionLogFilename);
207
208         Assert.True(transitions.Length == 1 && transitions[0].Before.IsNull() &&
209             ↪ transitions[0].After.IsNull());
210
211         lastScope.DeleteFiles();
212     }
213 }
214 [Fact]
215 public static void TransactionUserCodeErrorSomeDataSavedTest()
216 {
217     // User Code Error (Autoreverted), some data saved
218     var itself = _constants.Itself;
219
220     TempLinksTestScope lastScope = null;
221     try
222     {
223         using (var scope = new TempLinksTestScope(useLog: true))
224         {
225             var links = scope.Links;
226             l1 = links.CreateAndUpdate(itself, itself);

```

```

227         l2 = links.CreateAndUpdate(itself, itself);
228
229         l2 = links.Update(l2, l2, l1, l2);
230
231         links.CreateAndUpdate(l2, itself);
232         links.CreateAndUpdate(l2, itself);
233
234         links.Unsync.DisposeIfPossible();
235
236         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(
            ↪ scope.TempTransactionLogFilename);
237     }
238
239     using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
        ↪ useLog: true))
240     {
241         var links = scope.Links;
242         var transactionsLayer = (UInt64LinksTransactionsLayer)links.Unsync;
243         using (var transaction = transactionsLayer.BeginTransaction())
244         {
245             l2 = links.Update(l2, l1);
246
247             links.Delete(l2);
248
249             ExceptionThrower();
250
251             transaction.Commit();
252         }
253
254         Global.Trash = links.Count();
255     }
256 }
257 catch
258 {
259     Assert.False(lastScope == null);
260
261     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(last
        ↪ Scope.TempTransactionLogFilename);
262
263     lastScope.DeleteFiles();
264 }
265 }
266
267 [Fact]
268 public static void TransactionCommit()
269 {
270     var itself = _constants.Itself;
271
272     var tempDatabaseFilename = Path.GetTempFileName();
273     var tempTransactionLogFilename = Path.GetTempFileName();
274
275     // Commit
276     using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
        ↪ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
        ↪ tempTransactionLogFilename))
277     using (var links = new UInt64Links(memoryAdapter))
278     {
279         using (var transaction = memoryAdapter.BeginTransaction())
280         {
281             var l1 = links.CreateAndUpdate(itself, itself);
282             var l2 = links.CreateAndUpdate(itself, itself);
283
284             Global.Trash = links.Update(l2, l2, l1, l2);
285
286             links.Delete(l1);
287
288             transaction.Commit();
289         }
290
291         Global.Trash = links.Count();
292     }
293
294     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran
        ↪ sactionLogFilename);
295 }
296
297 [Fact]
298 public static void TransactionDamage()
299 {

```

```

300     var itself = _constants.Itself;
301
302     var tempDatabaseFilename = Path.GetTempFileName();
303     var tempTransactionLogFilename = Path.GetTempFileName();
304
305     // Commit
306     using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
        ↳ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
        ↳ tempTransactionLogFilename))
307     using (var links = new UInt64Links(memoryAdapter))
308     {
309         using (var transaction = memoryAdapter.BeginTransaction())
310         {
311             var l1 = links.CreateAndUpdate(itself, itself);
312             var l2 = links.CreateAndUpdate(itself, itself);
313
314             Global.Trash = links.Update(l2, l2, l1, l2);
315
316             links.Delete(l1);
317
318             transaction.Commit();
319         }
320
321         Global.Trash = links.Count();
322     }
323
324     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran
        ↳ sactionLogFilename);
325
326     // Damage database
327
328     FileHelpers.WriteFirst(tempTransactionLogFilename, new
        ↳ UInt64LinksTransactionsLayer.Transition(new UniqueTimestampFactory(), 555));
329
330     // Try load damaged database
331     try
332     {
333         // TODO: Fix
334         using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
            ↳ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
            ↳ tempTransactionLogFilename))
335         using (var links = new UInt64Links(memoryAdapter))
336         {
337             Global.Trash = links.Count();
338         }
339     }
340     catch (NotSupportedException ex)
341     {
342         Assert.True(ex.Message == "Database is damaged, autorecovery is not supported
            ↳ yet.");
343     }
344
345     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran
        ↳ sactionLogFilename);
346
347     File.Delete(tempDatabaseFilename);
348     File.Delete(tempTransactionLogFilename);
349 }
350
351 [Fact]
352 public static void Bug1Test()
353 {
354     var tempDatabaseFilename = Path.GetTempFileName();
355     var tempTransactionLogFilename = Path.GetTempFileName();
356
357     var itself = _constants.Itself;
358
359     // User Code Error (Autoreverted), some data saved
360     try
361     {
362         ulong l1;
363         ulong l2;
364
365         using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
            ↳ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
            ↳ tempTransactionLogFilename))
366         using (var links = new UInt64Links(memoryAdapter))
367         {
368             l1 = links.CreateAndUpdate(itself, itself);

```

```

369         l2 = links.CreateAndUpdate(itself, itself);
370
371         l2 = links.Update(l2, l2, l1, l2);
372
373         links.CreateAndUpdate(l2, itself);
374         links.CreateAndUpdate(l2, itself);
375     }
376
377     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(temp_
    ↪ TransactionLogFilename);
378
379     using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
    ↪ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
    ↪ tempTransactionLogFilename))
380     using (var links = new UInt64Links(memoryAdapter))
381     {
382         using (var transaction = memoryAdapter.BeginTransaction())
383         {
384             l2 = links.Update(l2, l1);
385
386             links.Delete(l2);
387
388             ExceptionThrower();
389
390             transaction.Commit();
391         }
392
393         Global.Trash = links.Count();
394     }
395 }
396 catch
397 {
398     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(temp_
    ↪ TransactionLogFilename);
399 }
400
401 File.Delete(tempDatabaseFilename);
402 File.Delete(tempTransactionLogFilename);
403 }
404
405 private static void ExceptionThrower()
406 {
407     throw new Exception();
408 }
409
410 [Fact]
411 public static void PathsTest()
412 {
413     var source = _constants.SourcePart;
414     var target = _constants.TargetPart;
415
416     using (var scope = new TempLinksTestScope())
417     {
418         var links = scope.Links;
419         var l1 = links.CreatePoint();
420         var l2 = links.CreatePoint();
421
422         var r1 = links.GetByKeys(l1, source, target, source);
423         var r2 = links.CheckPathExistence(l2, l2, l2, l2);
424     }
425 }
426
427 [Fact]
428 public static void RecursiveStringFormattingTest()
429 {
430     using (var scope = new TempLinksTestScope(useSequences: true))
431     {
432         var links = scope.Links;
433         var sequences = scope.Sequences; // TODO: Auto use sequences on Sequences getter.
434
435         var a = links.CreatePoint();
436         var b = links.CreatePoint();
437         var c = links.CreatePoint();
438
439         var ab = links.CreateAndUpdate(a, b);
440         var cb = links.CreateAndUpdate(c, b);
441         var ac = links.CreateAndUpdate(a, c);
442
443         a = links.Update(a, c, b);

```



```

444     b = links.Update(b, a, c);
445     c = links.Update(c, a, b);
446
447     Debug.WriteLine(links.FormatStructure(ab, link => link.IsFullPoint(), true));
448     Debug.WriteLine(links.FormatStructure(cb, link => link.IsFullPoint(), true));
449     Debug.WriteLine(links.FormatStructure(ac, link => link.IsFullPoint(), true));
450
451     Assert.True(links.FormatStructure(cb, link => link.IsFullPoint(), true) ==
452         ↪ "(5:(4:5 (6:5 4)) 6)");
453     Assert.True(links.FormatStructure(ac, link => link.IsFullPoint(), true) ==
454         ↪ "(6:(5:(4:5 6) 6) 4)");
455     Assert.True(links.FormatStructure(ab, link => link.IsFullPoint(), true) ==
456         ↪ "(4:(5:4 (6:5 4)) 6)");
457
458     // TODO: Think how to build balanced syntax tree while formatting structure (eg.
459     ↪ "(4:(5:4 6) (6:5 4))" instead of "(4:(5:4 (6:5 4)) 6)"
460
461     Assert.True(sequences.SafeFormatSequence(cb, DefaultFormatter, false) ==
462         ↪ "{{5}{5}{4}{6}}");
463     Assert.True(sequences.SafeFormatSequence(ac, DefaultFormatter, false) ==
464         ↪ "{{5}{6}{6}{4}}");
465     Assert.True(sequences.SafeFormatSequence(ab, DefaultFormatter, false) ==
466         ↪ "{{4}{5}{4}{6}}");
467 }
468
469 private static void DefaultFormatter(StringBuilder sb, ulong link)
470 {
471     sb.Append(link.ToString());
472 }
473
474 #endregion
475
476 #region Performance
477
478 /*
479 public static void RunAllPerformanceTests()
480 {
481     try
482     {
483         links.TestLinksInSteps();
484     }
485     catch (Exception ex)
486     {
487         ex.WriteToConsole();
488     }
489
490     return;
491
492     try
493     {
494         //ThreadPool.SetMaxThreads(2, 2);
495
496         // Запускаем все тесты дважды, чтобы первоначальная инициализация не повлияла на
497         ↪ результат
498         // Также это дополнительно помогает в отладке
499         // Увеличивает вероятность попадания информации в кэши
500         for (var i = 0; i < 10; i++)
501         {
502             //0 - 10 ГБ
503             //Каждые 100 МБ срез цифр
504
505             //links.TestGetSourceFunction();
506             //links.TestGetSourceFunctionInParallel();
507             //links.TestGetTargetFunction();
508             //links.TestGetTargetFunctionInParallel();
509             links.Create64BillionLinks();
510
511             links.TestRandomSearchFixed();
512             //links.Create64BillionLinksInParallel();
513             links.TestEachFunction();
514             //links.TestForeach();
515             //links.TestParallelForeach();
516         }
517
518         links.TestDeletionOfAllLinks();
519     }
520     catch (Exception ex)

```

```

515         {
516             ex.WriteLineToConsole();
517         }
518     }*/
519
520     /*
521     public static void TestLinksInSteps()
522     {
523         const long gibibyte = 1024 * 1024 * 1024;
524         const long mebibyte = 1024 * 1024;
525
526         var totalLinksToCreate = gibibyte /
↪ Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
527         var linksStep = 102 * mebibyte /
↪ Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
528
529         var creationMeasurements = new List<TimeSpan>();
530         var searchMeasurements = new List<TimeSpan>();
531         var deletionMeasurements = new List<TimeSpan>();
532
533         GetBaseRandomLoopOverhead(linksStep);
534         GetBaseRandomLoopOverhead(linksStep);
535
536         var stepLoopOverhead = GetBaseRandomLoopOverhead(linksStep);
537
538         ConsoleHelpers.Debug("Step loop overhead: {0}.", stepLoopOverhead);
539
540         var loops = totalLinksToCreate / linksStep;
541
542         for (int i = 0; i < loops; i++)
543         {
544             creationMeasurements.Add(Measure(() => links.RunRandomCreations(linksStep)));
545             searchMeasurements.Add(Measure(() => links.RunRandomSearches(linksStep)));
546
547             Console.WriteLine("\rC + S {0}/{1}", i + 1, loops);
548         }
549
550         ConsoleHelpers.Debug();
551
552         for (int i = 0; i < loops; i++)
553         {
554             deletionMeasurements.Add(Measure(() => links.RunRandomDeletions(linksStep)));
555
556             Console.WriteLine("\rD {0}/{1}", i + 1, loops);
557         }
558
559         ConsoleHelpers.Debug();
560
561         ConsoleHelpers.Debug("C S D");
562
563         for (int i = 0; i < loops; i++)
564         {
565             ConsoleHelpers.Debug("{0} {1} {2}", creationMeasurements[i],
↪ searchMeasurements[i], deletionMeasurements[i]);
566         }
567
568         ConsoleHelpers.Debug("C S D (no overhead)");
569
570         for (int i = 0; i < loops; i++)
571         {
572             ConsoleHelpers.Debug("{0} {1} {2}", creationMeasurements[i] - stepLoopOverhead,
↪ searchMeasurements[i] - stepLoopOverhead, deletionMeasurements[i] - stepLoopOverhead);
573         }
574
575         ConsoleHelpers.Debug("All tests done. Total links left in database: {0}.",
↪ links.Total);
576     }
577
578     private static void CreatePoints(this Platform.Links.Data.Core.Doublets.Links links, long
↪ amountToCreate)
579     {
580         for (long i = 0; i < amountToCreate; i++)
581             links.Create(0, 0);
582     }
583
584     private static TimeSpan GetBaseRandomLoopOverhead(long loops)
585     {
586         return Measure(() =>
587         {

```

```

588         ulong maxValue = RandomHelpers.DefaultFactory.NextUInt64();
589         ulong result = 0;
590         for (long i = 0; i < loops; i++)
591         {
592             var source = RandomHelpers.DefaultFactory.NextUInt64(maxValue);
593             var target = RandomHelpers.DefaultFactory.NextUInt64(maxValue);
594
595             result += maxValue + source + target;
596         }
597         Global.Trash = result;
598     });
599 }
600 */
601
602 [Fact(Skip = "performance test")]
603 public static void GetSourceTest()
604 {
605     using (var scope = new TempLinksTestScope())
606     {
607         var links = scope.Links;
608         ConsoleHelpers.Debug("Testing GetSource function with {0} Iterations.",
609             ↪ Iterations);
610
611         ulong counter = 0;
612
613         //var firstLink = links.First();
614         // Создаём одну связь, из которой будет производить считывание
615         var firstLink = links.Create();
616
617         var sw = Stopwatch.StartNew();
618
619         // Тестируем саму функцию
620         for (ulong i = 0; i < Iterations; i++)
621         {
622             counter += links.GetSource(firstLink);
623         }
624
625         var elapsedTime = sw.Elapsed;
626
627         var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
628
629         // Удаляем связь, из которой производилось считывание
630         links.Delete(firstLink);
631
632         ConsoleHelpers.Debug(
633             "{0} Iterations of GetSource function done in {1} ({2} Iterations per
634             ↪ second), counter result: {3}",
635             Iterations, elapsedTime, (long)iterationsPerSecond, counter);
636     }
637 }
638
639 [Fact(Skip = "performance test")]
640 public static void GetSourceInParallel()
641 {
642     using (var scope = new TempLinksTestScope())
643     {
644         var links = scope.Links;
645         ConsoleHelpers.Debug("Testing GetSource function with {0} Iterations in
646             ↪ parallel.", Iterations);
647
648         long counter = 0;
649
650         //var firstLink = links.First();
651         var firstLink = links.Create();
652
653         var sw = Stopwatch.StartNew();
654
655         // Тестируем саму функцию
656         Parallel.For(0, Iterations, x =>
657         {
658             Interlocked.Add(ref counter, (long)links.GetSource(firstLink));
659             //Interlocked.Increment(ref counter);
660         });
661
662         var elapsedTime = sw.Elapsed;
663
664         var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
665
666         links.Delete(firstLink);
667     }
668 }

```

```

665         ConsoleHelpers.Debug(
666             "{0} Iterations of GetSource function done in {1} ({2} Iterations per
        ↳ second), counter result: {3}",
667             Iterations, elapsedTime, (long)iterationsPerSecond, counter);
668     }
669 }
670
671 [Fact(Skip = "performance test")]
672 public static void TestGetTarget()
673 {
674     using (var scope = new TempLinksTestScope())
675     {
676         var links = scope.Links;
677         ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations.",
        ↳ Iterations);
678
679         ulong counter = 0;
680
681         //var firstLink = links.First();
682         var firstLink = links.Create();
683
684         var sw = Stopwatch.StartNew();
685
686         for (ulong i = 0; i < Iterations; i++)
687         {
688             counter += links.GetTarget(firstLink);
689         }
690
691         var elapsedTime = sw.Elapsed;
692
693         var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
694
695         links.Delete(firstLink);
696
697         ConsoleHelpers.Debug(
698             "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
        ↳ second), counter result: {3}",
699             Iterations, elapsedTime, (long)iterationsPerSecond, counter);
700     }
701 }
702
703 [Fact(Skip = "performance test")]
704 public static void TestGetTargetInParallel()
705 {
706     using (var scope = new TempLinksTestScope())
707     {
708         var links = scope.Links;
709         ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations in
        ↳ parallel.", Iterations);
710
711         long counter = 0;
712
713         //var firstLink = links.First();
714         var firstLink = links.Create();
715
716         var sw = Stopwatch.StartNew();
717
718         Parallel.For(0, Iterations, x =>
719         {
720             Interlocked.Add(ref counter, (long)links.GetTarget(firstLink));
721             //Interlocked.Increment(ref counter);
722         });
723
724         var elapsedTime = sw.Elapsed;
725
726         var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
727
728         links.Delete(firstLink);
729
730         ConsoleHelpers.Debug(
731             "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
        ↳ second), counter result: {3}",
732             Iterations, elapsedTime, (long)iterationsPerSecond, counter);
733     }
734 }
735
736 // TODO: Заполнить базу данных перед тестом
737 /*
738 [Fact]

```

```

739     public void TestRandomSearchFixed()
740     {
741         var tempFilename = Path.GetTempFileName();
742
743         using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
↪ DefaultLinksSizeStep))
744         {
745             long iterations = 64 * 1024 * 1024 /
↪ Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
746
747             ulong counter = 0;
748             var maxLink = links.Total;
749
750             ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.", iterations);
751
752             var sw = Stopwatch.StartNew();
753
754             for (var i = iterations; i > 0; i--)
755             {
756                 var source =
↪ RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
757                 var target =
↪ RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
758
759                 counter += links.Search(source, target);
760             }
761
762             var elapsedTime = sw.Elapsed;
763
764             var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;
765
766             ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
↪ Iterations per second), c: {3}", iterations, elapsedTime, (long)iterationsPerSecond,
↪ counter);
767         }
768
769         File.Delete(tempFilename);
770     }*/
771
772     [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
773     public static void TestRandomSearchAll()
774     {
775         using (var scope = new TempLinksTestScope())
776         {
777             var links = scope.Links;
778             ulong counter = 0;
779
780             var maxLink = links.Count();
781
782             var iterations = links.Count();
783
784             ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.",
↪ links.Count());
785
786             var sw = Stopwatch.StartNew();
787
788             for (var i = iterations; i > 0; i--)
789             {
790                 var linksAddressRange = new Range<ulong>(_constants.MinPossibleIndex,
↪ maxLink);
791
792                 var source = RandomHelpers.Default.NextUInt64(linksAddressRange);
793                 var target = RandomHelpers.Default.NextUInt64(linksAddressRange);
794
795                 counter += links.SearchOrDefault(source, target);
796             }
797
798             var elapsedTime = sw.Elapsed;
799
800             var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;
801
802             ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
↪ Iterations per second), c: {3}",
            iterations, elapsedTime, (long)iterationsPerSecond, counter);
803         }
804     }
805
806     [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
807     public static void TestEach()
808

```

```

809 {
810     using (var scope = new TempLinksTestScope())
811     {
812         var links = scope.Links;
813
814         var counter = new Counter<IList<ulong>, ulong>(links.Constants.Continue);
815
816         ConsoleHelpers.Debug("Testing Each function.");
817
818         var sw = Stopwatch.StartNew();
819
820         links.Each(counter.IncrementAndReturnTrue);
821
822         var elapsedTime = sw.Elapsed;
823
824         var linksPerSecond = counter.Count / elapsedTime.TotalSeconds;
825
826         ConsoleHelpers.Debug("{0} Iterations of Each's handler function done in {1} ({2}
↪         ↪ links per second)",
            counter, elapsedTime, (long)linksPerSecond);
827     }
828 }
829
830 /*
831 [Fact]
832 public static void TestForeach()
833 {
834     var tempFilename = Path.GetTempFileName();
835
836     using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
↪     DefaultLinksSizeStep))
837     {
838         ulong counter = 0;
839
840         ConsoleHelpers.Debug("Testing foreach through links.");
841
842         var sw = Stopwatch.StartNew();
843
844         //foreach (var link in links)
845         //{
846             //    counter++;
847         //}
848
849         var elapsedTime = sw.Elapsed;
850
851         var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
852
853         ConsoleHelpers.Debug("{0} Iterations of Foreach's handler block done in {1} ({2}
↪ links per second)", counter, elapsedTime, (long)linksPerSecond);
854     }
855
856     File.Delete(tempFilename);
857 }
858 */
859
860 /*
861 [Fact]
862 public static void TestParallelForeach()
863 {
864     var tempFilename = Path.GetTempFileName();
865
866     using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
↪     DefaultLinksSizeStep))
867     {
868         long counter = 0;
869
870         ConsoleHelpers.Debug("Testing parallel foreach through links.");
871
872         var sw = Stopwatch.StartNew();
873
874         //Parallel.ForEach((IEnumerable<ulong>)links, x =>
875         //{
876             //    Interlocked.Increment(ref counter);
877         //});
878
879         var elapsedTime = sw.Elapsed;
880
881         var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
882
883
884

```

```

885         ConsoleHelpers.Debug("{0} Iterations of Parallel Foreach's handler block done in
↪ {1} ({2} links per second)", counter, elapsedTime, (long)linksPerSecond);
886     }
887
888     File.Delete(tempFilename);
889 }
890 */
891
892 [Fact(Skip = "performance test")]
893 public static void Create64BillionLinks()
894 {
895     using (var scope = new TempLinksTestScope())
896     {
897         var links = scope.Links;
898         var linksBeforeTest = links.Count();
899
900         long linksToCreate = 64 * 1024 * 1024 /
↪ UInt64ResizableDirectMemoryLinks.LinkSizeInBytes;
901
902         ConsoleHelpers.Debug("Creating {0} links.", linksToCreate);
903
904         var elapsedTime = Performance.Measure(() =>
905         {
906             for (long i = 0; i < linksToCreate; i++)
907             {
908                 links.Create();
909             }
910         });
911
912         var linksCreated = links.Count() - linksBeforeTest;
913         var linksPerSecond = linksCreated / elapsedTime.TotalSeconds;
914
915         ConsoleHelpers.Debug("Current links count: {0}.", links.Count());
916
917         ConsoleHelpers.Debug("{0} links created in {1} ({2} links per second)",
↪ linksCreated, elapsedTime,
918             (long)linksPerSecond);
919     }
920 }
921
922 [Fact(Skip = "performance test")]
923 public static void Create64BillionLinksInParallel()
924 {
925     using (var scope = new TempLinksTestScope())
926     {
927         var links = scope.Links;
928         var linksBeforeTest = links.Count();
929
930         var sw = Stopwatch.StartNew();
931
932         long linksToCreate = 64 * 1024 * 1024 /
↪ UInt64ResizableDirectMemoryLinks.LinkSizeInBytes;
933
934         ConsoleHelpers.Debug("Creating {0} links in parallel.", linksToCreate);
935
936         Parallel.For(0, linksToCreate, x => links.Create());
937
938         var elapsedTime = sw.Elapsed;
939
940         var linksCreated = links.Count() - linksBeforeTest;
941         var linksPerSecond = linksCreated / elapsedTime.TotalSeconds;
942
943         ConsoleHelpers.Debug("{0} links created in {1} ({2} links per second)",
↪ linksCreated, elapsedTime,
944             (long)linksPerSecond);
945     }
946 }
947
948 [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
949 public static void TestDeletionOfAllLinks()
950 {
951     using (var scope = new TempLinksTestScope())
952     {
953         var links = scope.Links;
954         var linksBeforeTest = links.Count();
955
956         ConsoleHelpers.Debug("Deleting all links");
957
958         var elapsedTime = Performance.Measure(links.DeleteAll);
959

```

```

960         var linksDeleted = linksBeforeTest - links.Count();
961         var linksPerSecond = linksDeleted / elapsedTime.TotalSeconds;
962
963         ConsoleHelpers.Debug("{0} links deleted in {1} ({2} links per second)",
964             ↳ linksDeleted, elapsedTime,
965             (long)linksPerSecond);
966     }
967 }
968 #endregion
969 }
970 }

```

./Platform.Data.Doublets.Tests/OptimalVariantSequenceTests.cs

```

1  using System;
2  using System.Linq;
3  using System.Collections.Generic;
4  using Xunit;
5  using Platform.Data.Doublets.Sequences;
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.PropertyOperators;
10 using Platform.Data.Doublets.Incrementers;
11 using Platform.Data.Doublets.Sequences.Walkers;
12 using Platform.Data.Doublets.Sequences.Indexes;
13 using Platform.Data.Doublets.Unicode;
14 using Platform.Data.Doublets.UnaryNumbers;
15
16 namespace Platform.Data.Doublets.Tests
17 {
18     public static class OptimalVariantSequenceTests
19     {
20         private const string SequenceExample = "зеленела зелёная зелень";
21
22         [Fact]
23         public static void LinksBasedFrequencyStoredOptimalVariantSequenceTest()
24         {
25             using (var scope = new TempLinksTestScope(useSequences: false))
26             {
27                 var links = scope.Links;
28                 var constants = links.Constants;
29
30                 links.UseUnicode();
31
32                 var sequence = UnicodeMap.FromStringToLinkArray(SequenceExample);
33
34                 var meaningRoot = links.CreatePoint();
35                 var unaryOne = links.CreateAndUpdate(meaningRoot, constants.Itself);
36                 var frequencyMarker = links.CreateAndUpdate(meaningRoot, constants.Itself);
37                 var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot,
38                     ↳ constants.Itself);
39
40                 var unaryNumberToAddressConveter = new
41                     ↳ UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);
42                 var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links, unaryOne);
43                 var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
44                     ↳ frequencyMarker, unaryOne, unaryNumberIncrementer);
45                 var frequencyPropertyOperator = new PropertyOperator<ulong>(links,
46                     ↳ frequencyPropertyMarker, frequencyMarker);
47                 var index = new FrequencyIncrementingSequenceIndex<ulong>(links,
48                     ↳ frequencyPropertyOperator, frequencyIncrementer);
49                 var linkToItsFrequencyNumberConverter = new
50                     ↳ LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
51                     ↳ unaryNumberToAddressConveter);
52                 var sequenceToItsLocalElementLevelsConverter = new
53                     ↳ SequenceToItsLocalElementLevelsConverter<ulong>(links,
54                     ↳ linkToItsFrequencyNumberConverter);
55                 var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
56                     ↳ sequenceToItsLocalElementLevelsConverter);
57
58                 var sequences = new Sequences.Sequences(links, new SequencesOptions<ulong>() {
59                     ↳ Walker = new LeveledSequenceWalker<ulong>(links) });
60
61                 ExecuteTest(sequences, sequence, sequenceToItsLocalElementLevelsConverter,
62                     ↳ index, optimalVariantConverter);
63             }
64         }
65     }
66 }

```



```

55 public static void DictionaryBasedFrequencyStoredOptimalVariantSequenceTest()
56 {
57     using (var scope = new TempLinksTestScope(useSequences: false))
58     {
59         var links = scope.Links;
60
61         links.UseUnicode();
62
63         var sequence = UnicodeMap.FromStringToLinkArray(SequenceExample);
64
65         var linksToFrequencies = new Dictionary<ulong, ulong>();
66
67         var totalSequenceSymbolFrequencyCounter = new
        ↪ TotalSequenceSymbolFrequencyCounter<ulong>(links);
68
69         var linkFrequenciesCache = new LinkFrequenciesCache<ulong>(links,
        ↪ totalSequenceSymbolFrequencyCounter);
70
71         var index = new
        ↪ CachedFrequencyIncrementingSequenceIndex<ulong>(linkFrequenciesCache);
72         var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFreque
        ↪ ncyNumberConverter<ulong>(linkFrequenciesCache);
73
74         var sequenceToItsLocalElementLevelsConverter = new
        ↪ SequenceToItsLocalElementLevelsConverter<ulong>(links,
        ↪ linkToItsFrequencyNumberConverter);
75         var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
        ↪ sequenceToItsLocalElementLevelsConverter);
76
77         var sequences = new Sequences.Sequences(links, new SequencesOptions<ulong>() {
        ↪ Walker = new LeveledSequenceWalker<ulong>(links) });
78
79         ExecuteTest(sequences, sequence, sequenceToItsLocalElementLevelsConverter,
        ↪ index, optimalVariantConverter);
80     }
81 }
82
83 private static void ExecuteTest(Sequences.Sequences sequences, ulong[] sequence,
    ↪ SequenceToItsLocalElementLevelsConverter<ulong>
    ↪ sequenceToItsLocalElementLevelsConverter, ISequenceIndex<ulong> index,
    ↪ OptimalVariantConverter<ulong> optimalVariantConverter)
84 {
85     index.Add(sequence);
86
87     var optimalVariant = optimalVariantConverter.Convert(sequence);
88
89     var readSequence1 = sequences.ToList(optimalVariant);
90
91     Assert.True(sequence.SequenceEqual(readSequence1));
92 }
93 }
94 }

```

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

```

./Platform.Data.Doublets.Tests/ResizableDirectMemoryLinksTests.cs

```

1  using System.IO;
2  using Xunit;
3  using Platform.Singletons;
4  using Platform.Memory;
5  using Platform.Data.Constants;
6  using Platform.Data.Doublets.ResizableDirectMemory;
7
8  namespace Platform.Data.Doublets.Tests
9  {
10     public static class ResizableDirectMemoryLinksTests
11     {
12         private static readonly LinksCombinedConstants<ulong, ulong, int> _constants =
13             ↪ Default<LinksCombinedConstants<ulong, ulong, int>>.Instance;
14
15         [Fact]
16         public static void BasicFileMappedMemoryTest()
17         {
18             var tempFilename = Path.GetTempFileName();
19             using (var memoryAdapter = new UInt64ResizableDirectMemoryLinks(tempFilename))
20             {
21                 memoryAdapter.TestBasicMemoryOperations();
22             }
23             File.Delete(tempFilename);
24
25             [Fact]
26             public static void BasicHeapMemoryTest()
27             {
28                 using (var memory = new
29                     ↪ HeapResizableDirectMemory(UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
30                 using (var memoryAdapter = new UInt64ResizableDirectMemoryLinks(memory,
31                     ↪ UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
32                 {
33                     memoryAdapter.TestBasicMemoryOperations();
34                 }
35
36                 private static void TestBasicMemoryOperations(this ILinks<ulong> memoryAdapter)

```

```

36     {
37         var link = memoryAdapter.Create();
38         memoryAdapter.Delete(link);
39     }
40
41     [Fact]
42     public static void NonexistentReferencesHeapMemoryTest()
43     {
44         using (var memory = new
45             ↪ HeapResizableDirectMemory(UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
46         using (var memoryAdapter = new UInt64ResizableDirectMemoryLinks(memory,
47             ↪ UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
48         {
49             memoryAdapter.TestNonexistentReferences();
50         }
51
52         private static void TestNonexistentReferences(this ILinks<ulong> memoryAdapter)
53         {
54             var link = memoryAdapter.Create();
55             memoryAdapter.Update(link, ulong.MaxValue, ulong.MaxValue);
56             var resultLink = _constants.Null;
57             memoryAdapter.Each(foundLink =>
58             {
59                 resultLink = foundLink[_constants.IndexPart];
60                 return _constants.Break;
61             }, _constants.Any, ulong.MaxValue, ulong.MaxValue);
62             Assert.True(resultLink == link);
63             Assert.True(memoryAdapter.Count(ulong.MaxValue) == 0);
64             memoryAdapter.Delete(link);
65         }
66     }

```

#### ./Platform.Data.Doublets.Tests/ScopeTests.cs

```

1  using Xunit;
2  using Platform.Scopes;
3  using Platform.Memory;
4  using Platform.Data.Doublets.ResizableDirectMemory;
5  using Platform.Data.Doublets.Decorators;
6
7  namespace Platform.Data.Doublets.Tests
8  {
9      public static class ScopeTests
10     {
11         [Fact]
12         public static void SingleDependencyTest()
13         {
14             using (var scope = new Scope())
15             {
16                 scope.IncludeAssemblyOf<IMemory>();
17                 var instance = scope.Use<IDirectMemory>();
18                 Assert.IsType<HeapResizableDirectMemory>(instance);
19             }
20         }
21
22         [Fact]
23         public static void CascadeDependencyTest()
24         {
25             using (var scope = new Scope())
26             {
27                 scope.Include<TemporaryFileMappedResizableDirectMemory>();
28                 scope.Include<UInt64ResizableDirectMemoryLinks>();
29                 var instance = scope.Use<ILinks<ulong>>();
30                 Assert.IsType<UInt64ResizableDirectMemoryLinks>(instance);
31             }
32         }
33
34         [Fact]
35         public static void FullAutoResolutionTest()
36         {
37             using (var scope = new Scope(autoInclude: true, autoExplore: true))
38             {
39                 var instance = scope.Use<UInt64Links>();
40                 Assert.IsType<UInt64Links>(instance);
41             }
42         }
43     }
44 }

```

./Platform.Data.Doublets.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.Random;
8  using Platform.IO;
9  using Platform.Singletons;
10 using Platform.Data.Constants;
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.Tests
18 {
19     public static class SequencesTests
20     {
21         private static readonly LinksCombinedConstants<bool, ulong, int> _constants =
22             ↪ Default<LinksCombinedConstants<bool, ulong, int>>.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))
36             {
37                 var links = scope.Links;
38                 var sequences = scope.Sequences;
39
40                 var sequence = new ulong[sequenceLength];
41                 for (var i = 0; i < sequenceLength; i++)
42                 {
43                     sequence[i] = links.Create();
44                 }
45
46                 var sw1 = Stopwatch.StartNew();
47                 var results1 = sequences.CreateAllVariants1(sequence); sw1.Stop();
48
49                 var sw2 = Stopwatch.StartNew();
50                 var results2 = sequences.CreateAllVariants2(sequence); sw2.Stop();
51
52                 Assert.True(results1.Count > results2.Length);
53                 Assert.True(sw1.Elapsed > sw2.Elapsed);
54
55                 for (var i = 0; i < sequenceLength; i++)
56                 {
57                     links.Delete(sequence[i]);
58                 }
59
60                 Assert.True(links.Count() == 0);
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             //}
```

```

79
80 // SequencesOptions o = new SequencesOptions();
81
82 // TODO: Из числа в bool значения o.UseSequenceMarker = ((value & 1) != 0)
83 // o.
84
85
86 // var sequences = new Sequences(links);
87
88 // var sw1 = Stopwatch.StartNew();
89 // var results1 = sequences.CreateAllVariants1(sequence); sw1.Stop();
90
91 // var sw2 = Stopwatch.StartNew();
92 // var results2 = sequences.CreateAllVariants2(sequence); sw2.Stop();
93
94 // Assert.True(results1.Count > results2.Length);
95 // Assert.True(sw1.Elapsed > sw2.Elapsed);
96
97 // for (var i = 0; i < sequenceLength; i++)
98 //     links.Delete(sequence[i]);
99 // }
100
101 // File.Delete(tempFilename);
102 //}
103
104 [Fact]
105 public static void AllVariantsSearchTest()
106 {
107     const long sequenceLength = 8;
108
109     using (var scope = new TempLinksTestScope(useSequences: true))
110     {
111         var links = scope.Links;
112         var sequences = scope.Sequences;
113
114         var sequence = new ulong[sequenceLength];
115         for (var i = 0; i < sequenceLength; i++)
116         {
117             sequence[i] = links.Create();
118         }
119
120         var createResults = sequences.CreateAllVariants2(sequence).Distinct().ToArray();
121
122         //for (int i = 0; i < createResults.Length; i++)
123         //    sequences.Create(createResults[i]);
124
125         var sw0 = Stopwatch.StartNew();
126         var searchResults0 = sequences.GetAllMatchingSequences0(sequence); sw0.Stop();
127
128         var sw1 = Stopwatch.StartNew();
129         var searchResults1 = sequences.GetAllMatchingSequences1(sequence); sw1.Stop();
130
131         var sw2 = Stopwatch.StartNew();
132         var searchResults2 = sequences.Each1(sequence); sw2.Stop();
133
134         var sw3 = Stopwatch.StartNew();
135         var searchResults3 = sequences.Each(sequence); sw3.Stop();
136
137         var intersection0 = createResults.Intersect(searchResults0).ToList();
138         Assert.True(intersection0.Count == searchResults0.Count);
139         Assert.True(intersection0.Count == createResults.Length);
140
141         var intersection1 = createResults.Intersect(searchResults1).ToList();
142         Assert.True(intersection1.Count == searchResults1.Count);
143         Assert.True(intersection1.Count == createResults.Length);
144
145         var intersection2 = createResults.Intersect(searchResults2).ToList();
146         Assert.True(intersection2.Count == searchResults2.Count);
147         Assert.True(intersection2.Count == createResults.Length);
148
149         var intersection3 = createResults.Intersect(searchResults3).ToList();
150         Assert.True(intersection3.Count == searchResults3.Count);
151         Assert.True(intersection3.Count == createResults.Length);
152
153         for (var i = 0; i < sequenceLength; i++)
154         {
155             links.Delete(sequence[i]);
156         }
157     }
158 }

```

```

159 [Fact]
160 public static void BalancedVariantSearchTest()
161 {
162     const long sequenceLength = 200;
163
164     using (var scope = new TempLinksTestScope(useSequences: true))
165     {
166         var links = scope.Links;
167         var sequences = scope.Sequences;
168
169         var sequence = new ulong[sequenceLength];
170         for (var i = 0; i < sequenceLength; i++)
171         {
172             sequence[i] = links.Create();
173         }
174
175         var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
176
177         var sw1 = Stopwatch.StartNew();
178         var balancedVariant = balancedVariantConverter.Convert(sequence); sw1.Stop();
179
180         var sw2 = Stopwatch.StartNew();
181         var searchResults2 = sequences.GetAllMatchingSequences0(sequence); sw2.Stop();
182
183         var sw3 = Stopwatch.StartNew();
184         var searchResults3 = sequences.GetAllMatchingSequences1(sequence); sw3.Stop();
185
186         // На количестве в 200 элементов это будет занимать вечность
187         //var sw4 = Stopwatch.StartNew();
188         //var searchResults4 = sequences.Each(sequence); sw4.Stop();
189
190         Assert.True(searchResults2.Count == 1 && balancedVariant == searchResults2[0]);
191
192         Assert.True(searchResults3.Count == 1 && balancedVariant ==
193             ↪ searchResults3.First());
194
195         //Assert.True(sw1.Elapsed < sw2.Elapsed);
196
197         for (var i = 0; i < sequenceLength; i++)
198         {
199             links.Delete(sequence[i]);
200         }
201     }
202 }
203
204 [Fact]
205 public static void AllPartialVariantsSearchTest()
206 {
207     const long sequenceLength = 8;
208
209     using (var scope = new TempLinksTestScope(useSequences: true))
210     {
211         var links = scope.Links;
212         var sequences = scope.Sequences;
213
214         var sequence = new ulong[sequenceLength];
215         for (var i = 0; i < sequenceLength; i++)
216         {
217             sequence[i] = links.Create();
218         }
219
220         var createResults = sequences.CreateAllVariants2(sequence);
221
222         //var createResultsStrings = createResults.Select(x => x + ": " +
223             ↪ sequences.FormatSequence(x)).ToList();
224         //Global.Trash = createResultsStrings;
225
226         var partialSequence = new ulong[sequenceLength - 2];
227
228         Array.Copy(sequence, 1, partialSequence, 0, (int)sequenceLength - 2);
229
230         var sw1 = Stopwatch.StartNew();
231         var searchResults1 =
232             ↪ sequences.GetAllPartiallyMatchingSequences0(partialSequence); sw1.Stop();
233
234         var sw2 = Stopwatch.StartNew();
235         var searchResults2 =
236             ↪ sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();

```

```

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))
271     {
272         var links = scope.Links;
273         var sequences = scope.Sequences;
274
275         var sequence = new ulong[sequenceLength];
276         for (var i = 0; i < sequenceLength; i++)
277         {
278             sequence[i] = links.Create();
279         }
280
281         var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
282         var balancedVariant = balancedVariantConverter.Convert(sequence);
283
284         var partialSequence = new ulong[sequenceLength - 2];
285
286         Array.Copy(sequence, 1, partialSequence, 0, (int)sequenceLength - 2);
287
288         var sw1 = Stopwatch.StartNew();
289         var searchResults1 =
290             ↪ sequences.GetAllPartiallyMatchingSequences0(partialSequence); sw1.Stop();
291
292         var sw2 = Stopwatch.StartNew();
293         var searchResults2 =
294             ↪ sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();
295
296         Assert.True(searchResults1.Count == 1 && balancedVariant == searchResults1[0]);
297
298         Assert.True(searchResults2.Count == 1 && balancedVariant ==
299             ↪ searchResults2.First());
300
301         for (var i = 0; i < sequenceLength; i++)
302         {
303             links.Delete(sequence[i]);
304         }
305     }
306 }
307
308 [Fact(Skip = "Correct implementation is pending")]
309 public static void PatternMatchTest()
310 {
311     var zeroOrMany = Sequences.Sequences.ZeroOrMany;
312 }

```

```

308 using (var scope = new TempLinksTestScope(useSequences: true))
309 {
310     var links = scope.Links;
311     var sequences = scope.Sequences;
312
313     var e1 = links.Create();
314     var e2 = links.Create();
315
316     var sequence = new[]
317     {
318         e1, e2, e1, e2 // mama / papa
319     };
320
321     var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
322     var balancedVariant = balancedVariantConverter.Convert(sequence);
323
324     // 1: [1]
325     // 2: [2]
326     // 3: [1,2]
327     // 4: [1,2,1,2]
328
329     var doublet = links.GetSource(balancedVariant);
330
331     var matchedSequences1 = sequences.MatchPattern(e2, e1, zeroOrMany);
332
333     Assert.True(matchedSequences1.Count == 0);
334
335     var matchedSequences2 = sequences.MatchPattern(zeroOrMany, e2, e1);
336
337     Assert.True(matchedSequences2.Count == 0);
338
339     var matchedSequences3 = sequences.MatchPattern(e1, zeroOrMany, e1);
340
341     Assert.True(matchedSequences3.Count == 0);
342
343     var matchedSequences4 = sequences.MatchPattern(e1, zeroOrMany, e2);
344
345     Assert.Contains(doublet, matchedSequences4);
346     Assert.Contains(balancedVariant, matchedSequences4);
347
348     for (var i = 0; i < sequence.Length; i++)
349     {
350         links.Delete(sequence[i]);
351     }
352 }
353 }
354 }
355
356 [Fact]
357 public static void IndexTest()
358 {
359     using (var scope = new TempLinksTestScope(new SequencesOptions<ulong> { UseIndex =
360         ↪ true }, useSequences: true))
361     {
362         var links = scope.Links;
363         var sequences = scope.Sequences;
364         var index = sequences.Options.Index;
365
366         var e1 = links.Create();
367         var e2 = links.Create();
368
369         var sequence = new[]
370         {
371             e1, e2, e1, e2 // mama / papa
372         };
373
374         Assert.False(index.MightContain(sequence));
375
376         index.Add(sequence);
377
378         Assert.True(index.MightContain(sequence));
379     }
380 }
381
382 /// <summary>Imported from https://raw.githubusercontent.com/wiki/Konard/LinksPlatform/%
383 ↪ D0%9E-%D1%82%D0%BE%D0%BC%2C-%D0%BA%D0%B0%D0%BA-%D0%B2%D1%81%D1%91-%D0%BD%D0%B0%D1%87
384 ↪ %D0%B8%D0%BD%D0%B0%D0%BB%D0%BE%D1%81%D1%8C.md</summary>
385 private static readonly string _exampleText =
386     @"([english
387     ↪ version] (https://github.com/Konard/LinksPlatform/wiki/About-the-beginning))

```



384  
385 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?

386  
387 `[[[чёрное пространство, белое`  
→ `пространство](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/1.png`  
→ `"[чёрное пространство, белое пространство"]](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/1.png)`

388  
389 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?

390  
391 `[[[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)`

392  
393 А что если повторить? Сделать копию? Создать дубликат? Из одного сделать два? Может это быть  
→ так? Инверсия? Отражение? Сумма?

394  
395 `[[[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)`

396  
397 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?

398  
399 `[[[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)`

400  
401 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?

402  
403 `[[[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)`

404  
405 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?

406  
407 `[[[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)`

408  
409 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?

410  
411 `[[[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)`

412  
413 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?

414  
415 `[[[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)`

416

```

417 Допустим у нас есть смысл ""связать"" и смысл ""направления"", много ли это нам даёт? Много ли
    ↳ вариантов интерпретации? А что если уточнить, каким именно образом выполнена связь? Что если
    ↳ можно задать ей чёткий, конкретный смысл? Что это будет? Тип? Глагол? Связка? Действие?
    ↳ Трансформация? Переход из состояния в состояние? Или всё это и есть объект, суть которого в
    ↳ его конечном состоянии, если конечно конец определён направлением?
418
419 [![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)
420
421 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?
422
423 [![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)
424
425 На один уровень внутрь (вниз)? Или на один уровень во вне (вверх)? Или это можно назвать шагом
    ↳ рекурсии или фрактала?
426
427 [![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)
428
429 Последовательность? Массив? Список? Множество? Объект? Таблица? Элементы? Цвета? Символы? Буквы?
    ↳ Слово? Цифры? Число? Алфавит? Дерево? Сеть? Граф? Гиперграф?
430
431 [![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)
432
433 ...
434
435 [![анимация](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro-anim
    ↳ ation-500.gif
    ↳ ""анимация"")] (https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro
    ↳ -animation-500.gif)";
436
437
438     private static readonly string _exampleLoremIpsumText =
439         @"Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
    ↳ incididunt ut labore et dolore magna aliqua.
440 Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo
    ↳ consequat.";
441
442     [Fact]
443     public static void CompressionTest()
444     {
445         using (var scope = new TempLinksTestScope(useSequences: true))
446         {
447             var links = scope.Links;
448             var sequences = scope.Sequences;
449
450             var e1 = links.Create();
451             var e2 = links.Create();
452
453             var sequence = new[]
454             {
455                 e1, e2, e1, e2 // mama / papa / template [(m/p), a] { [1] [2] [1] [2] }
456             ];
457
458             var balancedVariantConverter = new BalancedVariantConverter(links.Unsync);
459             var totalSequenceSymbolFrequencyCounter = new
    ↳ TotalSequenceSymbolFrequencyCounter(links.Unsync);
460             var doubletFrequenciesCache = new LinkFrequenciesCache(links.Unsync,
    ↳ totalSequenceSymbolFrequencyCounter);
461             var compressingConverter = new CompressingConverter(links.Unsync,
    ↳ balancedVariantConverter, doubletFrequenciesCache);

```

```

462     var compressedVariant = compressingConverter.Convert(sequence);
463
464     // 1: [1]      (1->1) point
465     // 2: [2]      (2->2) point
466     // 3: [1,2]    (1->2) doublet
467     // 4: [1,2,1,2] (3->3) doublet
468
469     Assert.True(links.GetSource(links.GetSource(compressedVariant)) == sequence[0]);
470     Assert.True(links.GetTarget(links.GetSource(compressedVariant)) == sequence[1]);
471     Assert.True(links.GetSource(links.GetTarget(compressedVariant)) == sequence[2]);
472     Assert.True(links.GetTarget(links.GetTarget(compressedVariant)) == sequence[3]);
473
474     var source = _constants.SourcePart;
475     var target = _constants.TargetPart;
476
477     Assert.True(links.GetByKeys(compressedVariant, source, source) == sequence[0]);
478     Assert.True(links.GetByKeys(compressedVariant, source, target) == sequence[1]);
479     Assert.True(links.GetByKeys(compressedVariant, target, source) == sequence[2]);
480     Assert.True(links.GetByKeys(compressedVariant, target, target) == sequence[3]);
481
482     // 4 - length of sequence
483     Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 0)
484         ↪ == sequence[0]);
485     Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 1)
486         ↪ == sequence[1]);
487     Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 2)
488         ↪ == sequence[2]);
489     Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 3)
490         ↪ == sequence[3]);
491 }
492 }
493
494 [Fact]
495 public static void CompressionEfficiencyTest()
496 {
497     var strings = _exampleLoremIpsumText.Split(new[] { '\n', '\r' },
498         ↪ StringSplitOptions.RemoveEmptyEntries);
499     var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
500     var totalCharacters = arrays.Select(x => x.Length).Sum();
501
502     using (var scope1 = new TempLinksTestScope(useSequences: true))
503     using (var scope2 = new TempLinksTestScope(useSequences: true))
504     using (var scope3 = new TempLinksTestScope(useSequences: true))
505     {
506         scope1.Links.Unsync.UseUnicode();
507         scope2.Links.Unsync.UseUnicode();
508         scope3.Links.Unsync.UseUnicode();
509
510         var balancedVariantConverter1 = new
511             ↪ BalancedVariantConverter<ulong>(scope1.Links.Unsync);
512         var totalSequenceSymbolFrequencyCounter = new
513             ↪ TotalSequenceSymbolFrequencyCounter<ulong>(scope1.Links.Unsync);
514         var linkFrequenciesCache1 = new LinkFrequenciesCache<ulong>(scope1.Links.Unsync,
515             ↪ totalSequenceSymbolFrequencyCounter);
516         var compressor1 = new CompressingConverter<ulong>(scope1.Links.Unsync,
517             ↪ balancedVariantConverter1, linkFrequenciesCache1,
518             ↪ doInitialFrequenciesIncrement: false);
519
520         var compressor2 = scope2.Sequences;
521         var compressor3 = scope3.Sequences;
522
523         var constants = Default<LinksCombinedConstants<bool, ulong, int>>.Instance;
524
525         var sequences = compressor3;
526         //var meaningRoot = links.CreatePoint();
527         //var unaryOne = links.CreateAndUpdate(meaningRoot, constants.Itself);
528         //var frequencyMarker = links.CreateAndUpdate(meaningRoot, constants.Itself);
529         //var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot,
530             ↪ constants.Itself);
531
532         //var unaryNumberToAddressConveter = new
533             ↪ UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);
534         //var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links,
535             ↪ unaryOne);
536         //var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
537             ↪ frequencyMarker, unaryOne, unaryNumberIncrementer);

```

```

525 //var frequencyPropertyOperator = new FrequencyPropertyOperator<ulong>(links,
    ↳ frequencyPropertyMarker, frequencyMarker);
526 //var linkFrequencyIncrementer = new LinkFrequencyIncrementer<ulong>(links,
    ↳ frequencyPropertyOperator, frequencyIncrementer);
527 //var linkToItsFrequencyNumberConverter = new
    ↳ LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
    ↳ unaryNumberToAddressConveter);
528
529 var linkFrequenciesCache3 = new LinkFrequenciesCache<ulong>(scope3.Links.Unsync,
    ↳ totalSequenceSymbolFrequencyCounter);
530
531 var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFreque_
    ↳ ncyNumberConverter<ulong>(linkFrequenciesCache3);
532
533 var sequenceToItsLocalElementLevelsConverter = new
    ↳ SequenceToItsLocalElementLevelsConverter<ulong>(scope3.Links.Unsync,
    ↳ linkToItsFrequencyNumberConverter);
534 var optimalVariantConverter = new
    ↳ OptimalVariantConverter<ulong>(scope3.Links.Unsync,
    ↳ sequenceToItsLocalElementLevelsConverter);
535
536 var compressed1 = new ulong[arrays.Length];
537 var compressed2 = new ulong[arrays.Length];
538 var compressed3 = new ulong[arrays.Length];
539
540 var START = 0;
541 var END = arrays.Length;
542
543 //for (int i = START; i < END; i++)
544 //    linkFrequenciesCache1.IncrementFrequencies(arrays[i]);
545
546 var initialCount1 = scope2.Links.Unsync.Count();
547
548 var sw1 = Stopwatch.StartNew();
549
550 for (int i = START; i < END; i++)
551 {
552     linkFrequenciesCache1.IncrementFrequencies(arrays[i]);
553     compressed1[i] = compressor1.Convert(arrays[i]);
554 }
555
556 var elapsed1 = sw1.Elapsed;
557
558 var balancedVariantConverter2 = new
    ↳ BalancedVariantConverter<ulong>(scope2.Links.Unsync);
559
560 var initialCount2 = scope2.Links.Unsync.Count();
561
562 var sw2 = Stopwatch.StartNew();
563
564 for (int i = START; i < END; i++)
565 {
566     compressed2[i] = balancedVariantConverter2.Convert(arrays[i]);
567 }
568
569 var elapsed2 = sw2.Elapsed;
570
571 for (int i = START; i < END; i++)
572 {
573     linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
574 }
575
576 var initialCount3 = scope3.Links.Unsync.Count();
577
578 var sw3 = Stopwatch.StartNew();
579
580 for (int i = START; i < END; i++)
581 {
582     //linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
583     compressed3[i] = optimalVariantConverter.Convert(arrays[i]);
584 }
585
586 var elapsed3 = sw3.Elapsed;
587
588 Console.WriteLine($"Compressor: {elapsed1}, Balanced variant: {elapsed2},
    ↳ Optimal variant: {elapsed3}");
589
590 // Assert.True(elapsed1 > elapsed2);
591

```

```

592 // Checks
593 for (int i = START; i < END; i++)
594 {
595     var sequence1 = compressed1[i];
596     var sequence2 = compressed2[i];
597     var sequence3 = compressed3[i];
598
599     var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
600         ↪ scope1.Links.Unsync);
601
602     var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
603         ↪ scope2.Links.Unsync);
604
605     var decompress3 = UnicodeMap.FromSequenceLinkToString(sequence3,
606         ↪ scope3.Links.Unsync);
607
608     var structure1 = scope1.Links.Unsync.FormatStructure(sequence1, link =>
609         ↪ link.IsPartialPoint());
610     var structure2 = scope2.Links.Unsync.FormatStructure(sequence2, link =>
611         ↪ link.IsPartialPoint());
612     var structure3 = scope3.Links.Unsync.FormatStructure(sequence3, link =>
613         ↪ link.IsPartialPoint());
614
615     //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
616     ↪ arrays[i].Length > 3)
617     //    Assert.False(structure1 == structure2);
618     //if (sequence3 != Constants.Null && sequence2 != Constants.Null &&
619     ↪ arrays[i].Length > 3)
620     //    Assert.False(structure3 == structure2);
621
622     Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
623     Assert.True(strings[i] == decompress3 && decompress3 == decompress2);
624 }
625
626 Assert.True((int)(scope1.Links.Unsync.Count() - initialCount1) <
627     ↪ totalCharacters);
628 Assert.True((int)(scope2.Links.Unsync.Count() - initialCount2) <
629     ↪ totalCharacters);
630 Assert.True((int)(scope3.Links.Unsync.Count() - initialCount3) <
631     ↪ totalCharacters);
632
633 Console.WriteLine($"{(double)(scope1.Links.Unsync.Count() - initialCount1) /
634     ↪ totalCharacters} | {(double)(scope2.Links.Unsync.Count() - initialCount2) /
635     ↪ totalCharacters} | {(double)(scope3.Links.Unsync.Count() - initialCount3) /
636     ↪ totalCharacters}");
637
638 Assert.True(scope1.Links.Unsync.Count() - initialCount1 <
639     ↪ scope2.Links.Unsync.Count() - initialCount2);
640 Assert.True(scope3.Links.Unsync.Count() - initialCount3 <
641     ↪ scope2.Links.Unsync.Count() - initialCount2);
642
643 var duplicateProvider1 = new
644     ↪ DuplicateSegmentsProvider<ulong>(scope1.Links.Unsync, scope1.Sequences);
645 var duplicateProvider2 = new
646     ↪ DuplicateSegmentsProvider<ulong>(scope2.Links.Unsync, scope2.Sequences);
647 var duplicateProvider3 = new
648     ↪ DuplicateSegmentsProvider<ulong>(scope3.Links.Unsync, scope3.Sequences);
649
650 var duplicateCounter1 = new DuplicateSegmentsCounter<ulong>(duplicateProvider1);
651 var duplicateCounter2 = new DuplicateSegmentsCounter<ulong>(duplicateProvider2);
652 var duplicateCounter3 = new DuplicateSegmentsCounter<ulong>(duplicateProvider3);
653
654 var duplicates1 = duplicateCounter1.Count();
655
656 ConsoleHelpers.Debug("-----");
657
658 var duplicates2 = duplicateCounter2.Count();
659
660 ConsoleHelpers.Debug("-----");
661
662 var duplicates3 = duplicateCounter3.Count();
663
664 Console.WriteLine($"{duplicates1} | {duplicates2} | {duplicates3}");
665
666 linkFrequenciesCache1.ValidateFrequencies();
667 linkFrequenciesCache3.ValidateFrequencies();
668
669 }
670

```

```

651 [Fact]
652 public static void CompressionStabilityTest()
653 {
654     // TODO: Fix bug (do a separate test)
655     //const ulong minNumbers = 0;
656     //const ulong maxNumbers = 1000;
657
658     const ulong minNumbers = 10000;
659     const ulong maxNumbers = 12500;
660
661     var strings = new List<string>();
662
663     for (ulong i = minNumbers; i < maxNumbers; i++)
664     {
665         strings.Add(i.ToString());
666     }
667
668     var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
669     var totalCharacters = arrays.Select(x => x.Length).Sum();
670
671     using (var scope1 = new TempLinksTestScope(useSequences: true, sequencesOptions: new
672         ↪ SequencesOptions<ulong> { UseCompression = true,
673         ↪ EnforceSingleSequenceVersionOnWriteBasedOnExisting = true }))
674     using (var scope2 = new TempLinksTestScope(useSequences: true))
675     {
676         scope1.Links.UseUnicode();
677         scope2.Links.UseUnicode();
678
679         //var compressor1 = new Compressor(scope1.Links.Unsync, scope1.Sequences);
680         var compressor1 = scope1.Sequences;
681         var compressor2 = scope2.Sequences;
682
683         var compressed1 = new ulong[arrays.Length];
684         var compressed2 = new ulong[arrays.Length];
685
686         var sw1 = Stopwatch.StartNew();
687
688         var START = 0;
689         var END = arrays.Length;
690
691         // Collisions proved (cannot be solved by max doublet comparison, no stable rule)
692         // Stability issue starts at 10001 or 11000
693         //for (int i = START; i < END; i++)
694         //{
695         //    var first = compressor1.Compress(arrays[i]);
696         //    var second = compressor1.Compress(arrays[i]);
697
698         //    if (first == second)
699         //        compressed1[i] = first;
700         //    else
701         //    {
702         //        // TODO: Find a solution for this case
703         //    }
704         //}
705
706         for (int i = START; i < END; i++)
707         {
708             var first = compressor1.Create(arrays[i]);
709             var second = compressor1.Create(arrays[i]);
710
711             if (first == second)
712             {
713                 compressed1[i] = first;
714             }
715             else
716             {
717                 // TODO: Find a solution for this case
718             }
719         }
720
721         var elapsed1 = sw1.Elapsed;
722
723         var balancedVariantConverter = new BalancedVariantConverter<ulong>(scope2.Links);
724
725         var sw2 = Stopwatch.StartNew();
726
727         for (int i = START; i < END; i++)
728         {
729             var first = balancedVariantConverter.Convert(arrays[i]);

```

```

729         var second = balancedVariantConverter.Convert(arrays[i]);
730
731         if (first == second)
732         {
733             compressed2[i] = first;
734         }
735     }
736
737     var elapsed2 = sw2.Elapsed;
738
739     Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator:
740     ↪ {elapsed2}");
741
742     Assert.True(elapsed1 > elapsed2);
743
744     // Checks
745     for (int i = START; i < END; i++)
746     {
747         var sequence1 = compressed1[i];
748         var sequence2 = compressed2[i];
749
750         if (sequence1 != _constants.Null && sequence2 != _constants.Null)
751         {
752             var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
753             ↪ scope1.Links);
754
755             var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
756             ↪ scope2.Links);
757
758             //var structure1 = scope1.Links.FormatStructure(sequence1, link =>
759             ↪ link.IsPartialPoint());
760             //var structure2 = scope2.Links.FormatStructure(sequence2, link =>
761             ↪ link.IsPartialPoint());
762
763             //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
764             ↪ arrays[i].Length > 3)
765             //    Assert.False(structure1 == structure2);
766
767             Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
768         }
769     }
770
771     Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
772     Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
773
774     Debug.WriteLine($"{{(double)(scope1.Links.Count() - UnicodeMap.MapSize) /
775     ↪ totalCharacters}} | {{(double)(scope2.Links.Count() - UnicodeMap.MapSize) /
776     ↪ totalCharacters}}");
777
778     Assert.True(scope1.Links.Count() <= scope2.Links.Count());
779
780     //compressor1.ValidateFrequencies();
781 }
782
783 [Fact]
784 public static void RandomNumbersCompressionQualityTest()
785 {
786     const ulong N = 500;
787
788     //const ulong minNumbers = 10000;
789     //const ulong maxNumbers = 20000;
790
791     //var strings = new List<string>();
792
793     //for (ulong i = 0; i < N; i++)
794     //    strings.Add(RandomHelpers.DefaultFactory.NextUInt64(minNumbers,
795     ↪ maxNumbers).ToString());
796
797     var strings = new List<string>();
798
799     for (ulong i = 0; i < N; i++)
800     {
801         strings.Add(RandomHelpers.Default.NextUInt64().ToString());
802     }
803
804     strings = strings.Distinct().ToList();
805
806     var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();

```

```

var totalCharacters = arrays.Select(x => x.Length).Sum();

using (var scope1 = new TempLinksTestScope(useSequences: true, sequencesOptions: new
    ↳ SequencesOptions<ulong> { UseCompression = true,
    ↳ EnforceSingleSequenceVersionOnWriteBasedOnExisting = true }))
using (var scope2 = new TempLinksTestScope(useSequences: true))
{
    scope1.Links.UseUnicode();
    scope2.Links.UseUnicode();

    var compressor1 = scope1.Sequences;
    var compressor2 = scope2.Sequences;

    var compressed1 = new ulong[arrays.Length];
    var compressed2 = new ulong[arrays.Length];

    var sw1 = Stopwatch.StartNew();

    var START = 0;
    var END = arrays.Length;

    for (int i = START; i < END; i++)
    {
        compressed1[i] = compressor1.Create(arrays[i]);
    }

    var elapsed1 = sw1.Elapsed;

    var balancedVariantConverter = new BalancedVariantConverter<ulong>(scope2.Links);
    var sw2 = Stopwatch.StartNew();

    for (int i = START; i < END; i++)
    {
        compressed2[i] = balancedVariantConverter.Convert(arrays[i]);
    }

    var elapsed2 = sw2.Elapsed;

    Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator:
        ↳ {elapsed2}");

    Assert.True(elapsed1 > elapsed2);

    // Checks
    for (int i = START; i < END; i++)
    {
        var sequence1 = compressed1[i];
        var sequence2 = compressed2[i];

        if (sequence1 != _constants.Null && sequence2 != _constants.Null)
        {
            var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
                ↳ scope1.Links);

            var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
                ↳ scope2.Links);

            Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
        }
    }

    Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
    Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) < totalCharacters);

    Debug.WriteLine($"{{(double)(scope1.Links.Count() - UnicodeMap.MapSize) /
        ↳ totalCharacters}} | {{(double)(scope2.Links.Count() - UnicodeMap.MapSize) /
        ↳ totalCharacters}");

    // Can be worse than balanced variant
    //Assert.True(scope1.Links.Count() <= scope2.Links.Count());

    //compressor1.ValidateFrequencies();
}

}

[Fact]
public static void AllTreeBreakDownAtSequencesCreationBugTest()
{

```



```

871 // Made out of AllPossibleConnectionsTest test.
872
873 //const long sequenceLength = 5; //100% bug
874 const long sequenceLength = 4; //100% bug
875 //const long sequenceLength = 3; //100% _no_bug (ok)
876
877 using (var scope = new TempLinksTestScope(useSequences: true))
878 {
879     var links = scope.Links;
880     var sequences = scope.Sequences;
881
882     var sequence = new ulong[sequenceLength];
883     for (var i = 0; i < sequenceLength; i++)
884     {
885         sequence[i] = links.Create();
886     }
887
888     var createResults = sequences.CreateAllVariants2(sequence);
889
890     Global.Trash = createResults;
891
892     for (var i = 0; i < sequenceLength; i++)
893     {
894         links.Delete(sequence[i]);
895     }
896 }
897
898
899 [Fact]
900 public static void AllPossibleConnectionsTest()
901 {
902     const long sequenceLength = 5;
903
904     using (var scope = new TempLinksTestScope(useSequences: true))
905     {
906         var links = scope.Links;
907         var sequences = scope.Sequences;
908
909         var sequence = new ulong[sequenceLength];
910         for (var i = 0; i < sequenceLength; i++)
911         {
912             sequence[i] = links.Create();
913         }
914
915         var createResults = sequences.CreateAllVariants2(sequence);
916         var reverseResults = sequences.CreateAllVariants2(sequence.Reverse().ToArray());
917
918         for (var i = 0; i < 1; i++)
919         {
920             var sw1 = Stopwatch.StartNew();
921             var searchResults1 = sequences.GetAllConnections(sequence); sw1.Stop();
922
923             var sw2 = Stopwatch.StartNew();
924             var searchResults2 = sequences.GetAllConnections1(sequence); sw2.Stop();
925
926             var sw3 = Stopwatch.StartNew();
927             var searchResults3 = sequences.GetAllConnections2(sequence); sw3.Stop();
928
929             var sw4 = Stopwatch.StartNew();
930             var searchResults4 = sequences.GetAllConnections3(sequence); sw4.Stop();
931
932             Global.Trash = searchResults3;
933             Global.Trash = searchResults4; //-V3008
934
935             var intersection1 = createResults.Intersect(searchResults1).ToList();
936             Assert.True(intersection1.Count == createResults.Length);
937
938             var intersection2 = reverseResults.Intersect(searchResults1).ToList();
939             Assert.True(intersection2.Count == reverseResults.Length);
940
941             var intersection0 = searchResults1.Intersect(searchResults2).ToList();
942             Assert.True(intersection0.Count == searchResults2.Count);
943
944             var intersection3 = searchResults2.Intersect(searchResults3).ToList();
945             Assert.True(intersection3.Count == searchResults3.Count);
946
947             var intersection4 = searchResults3.Intersect(searchResults4).ToList();
948             Assert.True(intersection4.Count == searchResults4.Count);
949         }
950     }

```

```

951         for (var i = 0; i < sequenceLength; i++)
952         {
953             links.Delete(sequence[i]);
954         }
955     }
956 }
957
958 [Fact(Skip = "Correct implementation is pending")]
959 public static void CalculateAllUsagesTest()
960 {
961     const long sequenceLength = 3;
962
963     using (var scope = new TempLinksTestScope(useSequences: true))
964     {
965         var links = scope.Links;
966         var sequences = scope.Sequences;
967
968         var sequence = new ulong[sequenceLength];
969         for (var i = 0; i < sequenceLength; i++)
970         {
971             sequence[i] = links.Create();
972         }
973
974         var createResults = sequences.CreateAllVariants2(sequence);
975
976         //var reverseResults =
977         ↪ sequences.CreateAllVariants2(sequence.Reverse().ToArray());
978
979         for (var i = 0; i < 1; i++)
980         {
981             var linksTotalUsages1 = new ulong[links.Count() + 1];
982             sequences.CalculateAllUsages(linksTotalUsages1);
983
984             var linksTotalUsages2 = new ulong[links.Count() + 1];
985             sequences.CalculateAllUsages2(linksTotalUsages2);
986
987             var intersection1 = linksTotalUsages1.Intersect(linksTotalUsages2).ToList();
988             Assert.True(intersection1.Count == linksTotalUsages2.Length);
989         }
990
991         for (var i = 0; i < sequenceLength; i++)
992         {
993             links.Delete(sequence[i]);
994         }
995     }
996 }
997 }
998 }
999 }

```

./Platform.Data.Doublets.Tests/TempLinksTestScope.cs

```

1  using System.IO;
2  using Platform.Disposables;
3  using Platform.Data.Doublets.ResizableDirectMemory;
4  using Platform.Data.Doublets.Sequences;
5  using Platform.Data.Doublets.Decorators;
6
7  namespace Platform.Data.Doublets.Tests
8  {
9      public class TempLinksTestScope : DisposableBase
10     {
11         public readonly ILinks<ulong> MemoryAdapter;
12         public readonly SynchronizedLinks<ulong> Links;
13         public readonly Sequences.Sequences Sequences;
14         public readonly string TempFilename;
15         public readonly string TempTransactionLogFilename;
16         private readonly bool _deleteFiles;
17
18         public TempLinksTestScope(bool deleteFiles = true, bool useSequences = false, bool
19             ↪ useLog = false)
20             : this(new SequencesOptions<ulong>(), deleteFiles, useSequences, useLog)
21         {
22         }
23
24         public TempLinksTestScope(SequencesOptions<ulong> sequencesOptions, bool deleteFiles =
25             ↪ true, bool useSequences = false, bool useLog = false)
26         {
27             _deleteFiles = deleteFiles;
28             TempFilename = Path.GetTempFileName();

```

```

27     TempTransactionLogFilename = Path.GetTempFileName();
28
29     var coreMemoryAdapter = new UInt64ResizableDirectMemoryLinks(TempFilename);
30
31     MemoryAdapter = useLog ? (ILinks<ulong>)new
        ↳ UInt64LinksTransactionsLayer(coreMemoryAdapter, TempTransactionLogFilename) :
        ↳ coreMemoryAdapter;
32
33     Links = new SynchronizedLinks<ulong>(new UInt64Links(MemoryAdapter));
34     if (useSequences)
35     {
36         Sequences = new Sequences.Sequences(Links, sequencesOptions);
37     }
38 }
39
40 protected override void Dispose(bool manual, bool wasDisposed)
41 {
42     if (!wasDisposed)
43     {
44         Links.Unsync.DisposeIfPossible();
45         if (_deleteFiles)
46         {
47             DeleteFiles();
48         }
49     }
50 }
51
52 public void DeleteFiles()
53 {
54     File.Delete(TempFilename);
55     File.Delete(TempTransactionLogFilename);
56 }
57 }
58 }

```

./Platform.Data.Doublets.Tests/UnaryNumberConvertersTests.cs

```

1  using Xunit;
2  using Platform.Random;
3  using Platform.Data.Doublets.UnaryNumbers;
4
5  namespace Platform.Data.Doublets.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
                    ↳ PowerOf2ToUnaryNumberConverter<ulong>(links, one);
19                 var toUnaryNumberConverter = new AddressToUnaryNumberConverter<ulong>(links,
                    ↳ powerOf2ToUnaryNumberConverter);
20                 var random = new System.Random(0);
21                 ulong[] numbers = new ulong[N];
22                 ulong[] unaryNumbers = new ulong[N];
23                 for (int i = 0; i < N; i++)
24                 {
25                     numbers[i] = random.NextUInt64();
26                     unaryNumbers[i] = toUnaryNumberConverter.Convert(numbers[i]);
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 }

```

36 }  
37 }  
38 }

## Index

./Platform.Data.Doublets.Tests/ComparisonTests.cs, 133  
./Platform.Data.Doublets.Tests/DoubletLinksTests.cs, 134  
./Platform.Data.Doublets.Tests/EqualityTests.cs, 137  
./Platform.Data.Doublets.Tests/LinksTests.cs, 139  
./Platform.Data.Doublets.Tests/OptimalVariantSequenceTests.cs, 152  
./Platform.Data.Doublets.Tests/ReadSequenceTests.cs, 153  
./Platform.Data.Doublets.Tests/ResizableDirectMemoryLinksTests.cs, 154  
./Platform.Data.Doublets.Tests/ScopeTests.cs, 155  
./Platform.Data.Doublets.Tests/SequencesTests.cs, 156  
./Platform.Data.Doublets.Tests/TempLinksTestScope.cs, 170  
./Platform.Data.Doublets.Tests/UnaryNumberConvertersTests.cs, 171  
./Platform.Data.Doublets/Decorators/LinksCascadeUniquenessAndUsagesResolver.cs, 1  
./Platform.Data.Doublets/Decorators/LinksCascadeUsagesResolver.cs, 1  
./Platform.Data.Doublets/Decorators/LinksDecoratorBase.cs, 1  
./Platform.Data.Doublets/Decorators/LinksDisposableDecoratorBase.cs, 1  
./Platform.Data.Doublets/Decorators/LinksInnerReferenceExistenceValidator.cs, 2  
./Platform.Data.Doublets/Decorators/LinksItselfConstantToSelfReferenceResolver.cs, 2  
./Platform.Data.Doublets/Decorators/LinksNonExistentDependenciesCreator.cs, 3  
./Platform.Data.Doublets/Decorators/LinksNullConstantToSelfReferenceResolver.cs, 3  
./Platform.Data.Doublets/Decorators/LinksUniquenessResolver.cs, 3  
./Platform.Data.Doublets/Decorators/LinksUniquenessValidator.cs, 4  
./Platform.Data.Doublets/Decorators/LinksUsagesValidator.cs, 4  
./Platform.Data.Doublets/Decorators/NonNullContentsLinkDeletionResolver.cs, 4  
./Platform.Data.Doublets/Decorators/UInt64Links.cs, 5  
./Platform.Data.Doublets/Decorators/UniLinks.cs, 6  
./Platform.Data.Doublets/Doublet.cs, 11  
./Platform.Data.Doublets/DoubletComparer.cs, 10  
./Platform.Data.Doublets/Hybrid.cs, 11  
./Platform.Data.Doublets/ILinks.cs, 12  
./Platform.Data.Doublets/ILinksExtensions.cs, 12  
./Platform.Data.Doublets/ISynchronizedLinks.cs, 24  
./Platform.Data.Doublets/Incrementers/FrequencyIncrementer.cs, 23  
./Platform.Data.Doublets/Incrementers/UnaryNumberIncrementer.cs, 23  
./Platform.Data.Doublets/Link.cs, 24  
./Platform.Data.Doublets/LinkExtensions.cs, 26  
./Platform.Data.Doublets/LinksOperatorBase.cs, 26  
./Platform.Data.Doublets/PropertyOperators/PropertiesOperator.cs, 26  
./Platform.Data.Doublets/PropertyOperators/PropertyOperator.cs, 27  
./Platform.Data.Doublets/ResizableDirectMemory/ResizableDirectMemoryLinks.ListMethods.cs, 37  
./Platform.Data.Doublets/ResizableDirectMemory/ResizableDirectMemoryLinks.TreeMethods.cs, 37  
./Platform.Data.Doublets/ResizableDirectMemory/ResizableDirectMemoryLinks.cs, 28  
./Platform.Data.Doublets/ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.ListMethods.cs, 50  
./Platform.Data.Doublets/ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.TreeMethods.cs, 51  
./Platform.Data.Doublets/ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.cs, 44  
./Platform.Data.Doublets/Sequences/Converters/BalancedVariantConverter.cs, 57  
./Platform.Data.Doublets/Sequences/Converters/CompressingConverter.cs, 58  
./Platform.Data.Doublets/Sequences/Converters/LinksListToSequenceConverterBase.cs, 61  
./Platform.Data.Doublets/Sequences/Converters/OptimalVariantConverter.cs, 61  
./Platform.Data.Doublets/Sequences/Converters/SequenceToltsLocalElementLevelsConverter.cs, 63  
./Platform.Data.Doublets/Sequences/CriteriaMatchers/DefaultSequenceElementCriterionMatcher.cs, 63  
./Platform.Data.Doublets/Sequences/CriteriaMatchers/MarkedSequenceCriterionMatcher.cs, 63  
./Platform.Data.Doublets/Sequences/DefaultSequenceAppender.cs, 64  
./Platform.Data.Doublets/Sequences/DuplicateSegmentsCounter.cs, 64  
./Platform.Data.Doublets/Sequences/DuplicateSegmentsProvider.cs, 64  
./Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkFrequenciesCache.cs, 67  
./Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkFrequency.cs, 68  
./Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkToltsFrequencyValueConverter.cs, 69  
./Platform.Data.Doublets/Sequences/Frequencies/Counters/MarkedSequenceSymbolFrequencyOneOffCounter.cs, 69  
./Platform.Data.Doublets/Sequences/Frequencies/Counters/SequenceSymbolFrequencyOneOffCounter.cs, 69  
./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyCounter.cs, 70  
./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyOneOffCounter.cs, 70  
./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyCounter.cs, 71  
./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyOneOffCounter.cs, 71  
./Platform.Data.Doublets/Sequences/HeightProviders/CachedSequenceHeightProvider.cs, 72  
./Platform.Data.Doublets/Sequences/HeightProviders/DefaultSequenceRightHeightProvider.cs, 72  
./Platform.Data.Doublets/Sequences/HeightProviders/ISequenceHeightProvider.cs, 73

./Platform.Data.Doublets/Sequences/Indexes/CachedFrequencyIncrementingSequenceIndex.cs, 73  
./Platform.Data.Doublets/Sequences/Indexes/FrequencyIncrementingSequenceIndex.cs, 74  
./Platform.Data.Doublets/Sequences/Indexes/ISequenceIndex.cs, 74  
./Platform.Data.Doublets/Sequences/Indexes/SequenceIndex.cs, 74  
./Platform.Data.Doublets/Sequences/Indexes/SynchronizedSequenceIndex.cs, 75  
./Platform.Data.Doublets/Sequences/Sequences.Experiments.cs, 85  
./Platform.Data.Doublets/Sequences/Sequences.cs, 76  
./Platform.Data.Doublets/Sequences/SequencesExtensions.cs, 111  
./Platform.Data.Doublets/Sequences/SequencesOptions.cs, 112  
./Platform.Data.Doublets/Sequences/Walkers/ISequenceWalker.cs, 113  
./Platform.Data.Doublets/Sequences/Walkers/LeftSequenceWalker.cs, 113  
./Platform.Data.Doublets/Sequences/Walkers/LeveledSequenceWalker.cs, 114  
./Platform.Data.Doublets/Sequences/Walkers/RightSequenceWalker.cs, 115  
./Platform.Data.Doublets/Sequences/Walkers/SequenceWalkerBase.cs, 116  
./Platform.Data.Doublets/Stacks/Stack.cs, 116  
./Platform.Data.Doublets/Stacks/StackExtensions.cs, 117  
./Platform.Data.Doublets/SynchronizedLinks.cs, 117  
./Platform.Data.Doublets/UInt64Link.cs, 118  
./Platform.Data.Doublets/UInt64LinkExtensions.cs, 120  
./Platform.Data.Doublets/UInt64LinksExtensions.cs, 120  
./Platform.Data.Doublets/UInt64LinksTransactionsLayer.cs, 122  
./Platform.Data.Doublets/UnaryNumbers/AddressToUnaryNumberConverter.cs, 128  
./Platform.Data.Doublets/UnaryNumbers/LinkToItsFrequencyNumberConveter.cs, 128  
./Platform.Data.Doublets/UnaryNumbers/PowerOf2ToUnaryNumberConverter.cs, 129  
./Platform.Data.Doublets/UnaryNumbers/UnaryNumberToAddressAddOperationConverter.cs, 129  
./Platform.Data.Doublets/UnaryNumbers/UnaryNumberToAddressOrOperationConverter.cs, 130  
./Platform.Data.Doublets/Unicode/UnicodeMap.cs, 131