

# LinksPlatform's Platform.Data.Doublets Class Library

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

```

1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  namespace Platform.Data.Doublets.Decorators
4  {
5      public class LinksCascadeUniquenessAndUsagesResolver<TLink> : LinksUniquenessResolver<TLink>
6      {
7          public LinksCascadeUniquenessAndUsagesResolver(ILinks<TLink> links) : base(links) { }
8
9          protected override TLink ResolveAddressChangeConflict(TLink oldLinkAddress, TLink
10             ↪ newLinkAddress)
11          {
12              Links.MergeUsages(oldLinkAddress, newLinkAddress);
13              return base.ResolveAddressChangeConflict(oldLinkAddress, newLinkAddress);
14          }
15      }

```

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

```

1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  namespace Platform.Data.Doublets.Decorators
4  {
5      /// <remarks>
6      /// <para>Must be used in conjunction with NonNullContentsLinkDeletionResolver.</para>
7      /// <para>Должен использоваться вместе с NonNullContentsLinkDeletionResolver.</para>
8      /// </remarks>
9      public class LinksCascadeUsagesResolver<TLink> : LinksDecoratorBase<TLink>
10     {
11         public LinksCascadeUsagesResolver(ILinks<TLink> links) : base(links) { }
12
13         public override void Delete(TLink linkIndex)
14         {
15             this.DeleteAllUsages(linkIndex);
16             Links.Delete(linkIndex);
17         }
18     }
19 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using Platform.Data.Constants;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Decorators
8  {
9      public abstract class LinksDecoratorBase<TLink> : LinksOperatorBase<TLink>, ILinks<TLink>
10     {
11         public LinksCombinedConstants<TLink, TLink, int> Constants { get; }
12         protected LinksDecoratorBase(ILinks<TLink> links) : base(links) => Constants =
13             ↪ links.Constants;
14         public virtual TLink Count(IList<TLink> restriction) => Links.Count(restriction);
15         public virtual TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
16             ↪ => Links.Each(handler, restrictions);
17         public virtual TLink Create() => Links.Create();
18         public virtual TLink Update(IList<TLink> restrictions) => Links.Update(restrictions);
19         public virtual void Delete(TLink link) => Links.Delete(link);
20     }

```

## ./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  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.Decorators
9  {
10     public abstract class LinksDisposableDecoratorBase<TLink> : DisposableBase, ILinks<TLink>
11     {
12         public LinksCombinedConstants<TLink, TLink, int> Constants { get; }
13
14         public ILinks<TLink> Links { get; }
15     }

```

```

16     protected LinksDisposableDecoratorBase(ILinks<TLink> links)
17     {
18         Links = links;
19         Constants = links.Constants;
20     }
21
22     public virtual TLink Count(IList<TLink> restriction) => Links.Count(restriction);
23
24     public virtual TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
25     ↪ => Links.Each(handler, restrictions);
26
27     public virtual TLink Create() => Links.Create();
28
29     public virtual TLink Update(IList<TLink> restrictions) => Links.Update(restrictions);
30
31     public virtual void Delete(TLink link) => Links.Delete(link);
32
33     protected override bool AllowMultipleDisposeCalls => true;
34
35     protected override void Dispose(bool manual, bool wasDisposed)
36     {
37         if (!wasDisposed)
38         {
39             Links.DisposeIfPossible();
40         }
41     }
42 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Decorators
7  {
8      // TODO: Make LinksExternalReferenceValidator. A layer that checks each link to exist or to
9      ↪ be external (hybrid link's raw number).
10     public class LinksInnerReferenceExistenceValidator<TLink> : LinksDecoratorBase<TLink>
11     {
12         public LinksInnerReferenceExistenceValidator(ILinks<TLink> links) : base(links) { }
13
14         public override TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
15         {
16             Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
17             return Links.Each(handler, restrictions);
18         }
19
20         public override TLink Update(IList<TLink> restrictions)
21         {
22             // TODO: Possible values: null, ExistentLink or NonExistentHybrid(ExternalReference)
23             Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
24             return Links.Update(restrictions);
25         }
26
27         public override void Delete(TLink link)
28         {
29             Links.EnsureLinkExists(link, nameof(link));
30             Links.Delete(link);
31         }
32     }
33 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Decorators
7  {
8     public class LinksItselfConstantToSelfReferenceResolver<TLink> : LinksDecoratorBase<TLink>
9     {
10         private static readonly EqualityComparer<TLink> _equalityComparer =
11         ↪ EqualityComparer<TLink>.Default;
12
13         public LinksItselfConstantToSelfReferenceResolver(ILinks<TLink> links) : base(links) { }
14
15         public override TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)

```

```

15     {
16         var constants = Constants;
17         var itselfConstant = constants.Itself;
18         var indexPartConstant = constants.IndexPart;
19         var sourcePartConstant = constants.SourcePart;
20         var targetPartConstant = constants.TargetPart;
21         var restrictionsCount = restrictions.Count;
22         if (!_equalityComparer.Equals(constants.Any, itselfConstant)
23             && (((restrictionsCount > indexPartConstant) &&
24                 ↪ _equalityComparer.Equals(restrictions[indexPartConstant], itselfConstant))
25                 || ((restrictionsCount > sourcePartConstant) &&
26                     ↪ _equalityComparer.Equals(restrictions[sourcePartConstant], itselfConstant))
27                 || ((restrictionsCount > targetPartConstant) &&
28                     ↪ _equalityComparer.Equals(restrictions[targetPartConstant], itselfConstant))))
29             {
30                 // Itself constant is not supported for Each method right now, skipping execution
31                 return constants.Continue;
32             }
33         return Links.Each(handler, restrictions);
34     }
35 }

```

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

```

1 using System.Collections.Generic;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Data.Doublets.Decorators
6 {
7     /// <remarks>
8     /// Not practical if newSource and newTarget are too big.
9     /// To be able to use practical version we should allow to create link at any specific
10     ↪ location inside ResizableDirectMemoryLinks.
11     /// This in turn will require to implement not a list of empty links, but a list of ranges
12     ↪ to store it more efficiently.
13     /// </remarks>
14     public class LinksNonExistentDependenciesCreator<TLink> : LinksDecoratorBase<TLink>
15     {
16         public LinksNonExistentDependenciesCreator(ILinks<TLink> links) : base(links) { }
17
18         public override TLink Update(IList<TLink> restrictions)
19         {
20             var constants = Constants;
21             Links.EnsureCreated(restrictions[constants.SourcePart],
22                 ↪ restrictions[constants.TargetPart]);
23             return Links.Update(restrictions);
24         }
25     }
26 }

```

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

```

1 using System.Collections.Generic;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Data.Doublets.Decorators
6 {
7     public class LinksNullConstantToSelfReferenceResolver<TLink> : LinksDecoratorBase<TLink>
8     {
9         public LinksNullConstantToSelfReferenceResolver(ILinks<TLink> links) : base(links) { }
10
11         public override TLink Create()
12         {
13             var link = Links.Create();
14             return Links.Update(link, link, link);
15         }
16
17         public override TLink Update(IList<TLink> restrictions) =>
18             ↪ Links.Update(Links.ResolveConstantAsSelfReference(Constants.Null, restrictions));
19     }
20 }

```

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

```
1 using System.Collections.Generic;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Data.Doublets.Decorators
6 {
7     public class LinksUniquenessResolver<TLink> : LinksDecoratorBase<TLink>
8     {
9         private static readonly EqualityComparer<TLink> _equalityComparer =
10             ↳ EqualityComparer<TLink>.Default;
11
12         public LinksUniquenessResolver(ILinks<TLink> links) : base(links) { }
13
14         public override TLink Update(IList<TLink> restrictions)
15         {
16             var newLinkAddress = Links.SearchOrDefault(restrictions[Constants.SourcePart],
17                 ↳ restrictions[Constants.TargetPart]);
18             if (_equalityComparer.Equals(newLinkAddress, default))
19             {
20                 return Links.Update(restrictions);
21             }
22             return ResolveAddressChangeConflict(restrictions[Constants.IndexPart],
23                 ↳ newLinkAddress);
24         }
25
26         protected virtual TLink ResolveAddressChangeConflict(TLink oldLinkAddress, TLink
27             ↳ newLinkAddress)
28         {
29             if (!_equalityComparer.Equals(oldLinkAddress, newLinkAddress) &&
30                 ↳ Links.Exists(oldLinkAddress))
31             {
32                 Delete(oldLinkAddress);
33             }
34             return newLinkAddress;
35         }
36     }
37 }
```

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

```
1 using System.Collections.Generic;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Data.Doublets.Decorators
6 {
7     public class LinksUniquenessValidator<TLink> : LinksDecoratorBase<TLink>
8     {
9         public LinksUniquenessValidator(ILinks<TLink> links) : base(links) { }
10
11         public override TLink Update(IList<TLink> restrictions)
12         {
13             Links.EnsureDoesNotExists(restrictions[Constants.SourcePart],
14                 ↳ restrictions[Constants.TargetPart]);
15             return Links.Update(restrictions);
16         }
17     }
18 }
```

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

```
1 using System.Collections.Generic;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Data.Doublets.Decorators
6 {
7     public class LinksUsagesValidator<TLink> : LinksDecoratorBase<TLink>
8     {
9         public LinksUsagesValidator(ILinks<TLink> links) : base(links) { }
10
11         public override TLink Update(IList<TLink> restrictions)
12         {
13             Links.EnsureNoUsages(restrictions[Constants.IndexPart]);
14             return Links.Update(restrictions);
15         }
16
17         public override void Delete(TLink link)
18         {
19             Links.EnsureNoUsages(link);
20         }
21     }
22 }
```

```

20         Links.Delete(link);
21     }
22 }
23 }

```

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

```

1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  namespace Platform.Data.Doublets.Decorators
4  {
5      public class NonNullContentsLinkDeletionResolver<TLink> : LinksDecoratorBase<TLink>
6      {
7          public NonNullContentsLinkDeletionResolver(ILinks<TLink> links) : base(links) { }
8
9          public override void Delete(TLink linkIndex)
10         {
11             Links.EnforceResetValues(linkIndex);
12             Links.Delete(linkIndex);
13         }
14     }
15 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using Platform.Collections;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Decorators
8  {
9      /// <summary>
10     /// Представляет объект для работы с базой данных (файлом) в формате Links (массива связей).
11     /// </summary>
12     /// <remarks>
13     /// Возможные оптимизации:
14     /// Объединение в одном поле Source и Target с уменьшением до 32 бит.
15     ///     + меньше объём БД
16     ///     - меньше производительность
17     ///     - больше ограничение на количество связей в БД)
18     /// Ленивое хранение размеров поддеревьев (расчитываемое по мере использования БД)
19     ///     + меньше объём БД
20     ///     - больше сложность
21     ///
22     /// Текущее теоретическое ограничение на индекс связи, из-за использования 5 бит в размере
23     ↪ поддеревьев для AVL баланса и флагов нитей: 2 в степени(64 минус 5 равно 59 ) равно 576
24     ↪ 460 752 303 423 488
25     /// Желательно реализовать поддержку переключения между деревьями и битовыми индексами
26     ↪ (битовыми строками) - вариант матрицы (выстраиваемой лениво).
27     ///
28     /// Решить отключать ли проверки при компиляции под Release. Т.е. исключения будут
29     ↪ выбрасываться только при #if DEBUG
30     /// </remarks>
31     public class UInt64Links : LinksDisposableDecoratorBase<ulong>
32     {
33         public UInt64Links(ILinks<ulong> links) : base(links) { }
34
35         public override ulong Each(Func<IList<ulong>, ulong> handler, IList<ulong> restrictions)
36         {
37             this.EnsureLinkIsAnyOrExists(restrictions);
38             return Links.Each(handler, restrictions);
39         }
40
41         public override ulong Create() => Links.CreatePoint();
42
43         public override ulong Update(IList<ulong> restrictions)
44         {
45             var constants = Constants;
46             var nullConstant = constants.Null;
47             if (restrictions.IsNullOrEmpty())
48             {
49                 return nullConstant;
50             }
51             // TODO: Looks like this is a common type of exceptions linked with restrictions
52             ↪ support
53             if (restrictions.Count != 3)
54             {
55                 throw new NotSupportedException();
56             }
57         }
58     }
59 }

```

```

52     var indexPartConstant = constants.IndexPart;
53     var updatedLink = restrictions[indexPartConstant];
54     this.EnsureLinkExists(updatedLink,
55         ↪ $"{nameof(restrictions)}[{nameof(indexPartConstant)}]");
56     var sourcePartConstant = constants.SourcePart;
57     var newSource = restrictions[sourcePartConstant];
58     this.EnsureLinkIsItselfOrExists(newSource,
59         ↪ $"{nameof(restrictions)}[{nameof(sourcePartConstant)}]");
60     var targetPartConstant = constants.TargetPart;
61     var newTarget = restrictions[targetPartConstant];
62     this.EnsureLinkIsItselfOrExists(newTarget,
63         ↪ $"{nameof(restrictions)}[{nameof(targetPartConstant)}]");
64     var existedLink = nullConstant;
65     var itselfConstant = constants.Itself;
66     if (newSource != itselfConstant && newTarget != itselfConstant)
67     {
68         existedLink = this.SearchOrDefault(newSource, newTarget);
69     }
70     if (existedLink == nullConstant)
71     {
72         var before = Links.GetLink(updatedLink);
73         if (before[sourcePartConstant] != newSource || before[targetPartConstant] !=
74             ↪ newTarget)
75         {
76             Links.Update(updatedLink, newSource == itselfConstant ? updatedLink :
77                 ↪ newSource,
78                                     newTarget == itselfConstant ? updatedLink :
79                 ↪ newTarget);
80         }
81         return updatedLink;
82     }
83     else
84     {
85         return this.MergeAndDelete(updatedLink, existedLink);
86     }
87 }
88
89 public override void Delete(ulong linkIndex)
90 {
91     Links.EnsureLinkExists(linkIndex);
92     Links.EnforceResetValues(linkIndex);
93     this.DeleteAllUsages(linkIndex);
94     Links.Delete(linkIndex);
95 }
96
97 }

```

./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  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11 namespace Platform.Data.Doublets.Decorators
12 {
13     /// <remarks>
14     /// What does empty pattern (for condition or substitution) mean? Nothing or Everything?
15     /// Now we go with nothing. And nothing is something one, but empty, and cannot be changed
16     ↪ by itself. But can cause creation (update from nothing) or deletion (update to nothing).
17     ///
18     /// TODO: Decide to change to IDoubletLinks or not to change. (Better to create
19     ↪ DefaultUniLinksBase, that contains logic itself and can be implemented using both
20     ↪ IDoubletLinks and ILinks.)
21     /// </remarks>
22     internal class UniLinks<TLink> : LinksDecoratorBase<TLink>, IUniLinks<TLink>
23     {
24         private static readonly EqualityComparer<TLink> _equalityComparer =
25             ↪ EqualityComparer<TLink>.Default;
26
27         public UniLinks(ILinks<TLink> links) : base(links) { }
28
29         private struct Transition
30         {
31             public IList<TLink> Before;

```

```

28     public IList<TLink> After;
29
30     public Transition(IList<TLink> before, IList<TLink> after)
31     {
32         Before = before;
33         After = after;
34     }
35 }
36
37 //public static readonly TLink NullConstant = Use<LinksCombinedConstants<TLink, TLink,
38     ↳ int>>.Single.Null;
39 //public static readonly IReadOnlyList<TLink> NullLink = new
40     ↳ ReadOnlyCollection<TLink>(new List<TLink> { NullConstant, NullConstant, NullConstant
41     ↳ });
42
43 // TODO: Подумать о том, как реализовать древовидный Restriction и Substitution
44     ↳ (Links-Expression)
45 public TLink Trigger(IList<TLink> restriction, Func<IList<TLink>, IList<TLink>, TLink>
46     ↳ matchedHandler, IList<TLink> substitution, Func<IList<TLink>, IList<TLink>, TLink>
47     ↳ substitutedHandler)
48 {
49     ///List<Transition> transitions = null;
50     ///if (!restriction.IsNullOrEmpty())
51     ///{
52     ///    // Есть причина делать проход (чтение)
53     ///    if (matchedHandler != null)
54     ///    {
55     ///        if (!substitution.IsNullOrEmpty())
56     ///        {
57     ///            // restriction => { 0, 0, 0 } | { 0 } // Create
58     ///            // substitution => { itself, 0, 0 } | { itself, itself, itself } //
59     ↳ Create / Update
60     ///            // substitution => { 0, 0, 0 } | { 0 } // Delete
61     ///            transitions = new List<Transition>();
62     ///            if (Equals(substitution[Constants.IndexPart], Constants.Null))
63     ///            {
64     ///                // If index is Null, that means we always ignore every other
65     ↳ value (they are also Null by definition)
66     ///                var matchDecision = matchedHandler(, NullLink);
67     ///                if (Equals(matchDecision, Constants.Break))
68     ///                    return false;
69     ///                if (!Equals(matchDecision, Constants.Skip))
70     ///                    transitions.Add(new Transition(matchedLink, newValue));
71     ///            }
72     ///            else
73     ///            {
74     ///                Func<T, bool> handler;
75     ///                handler = link =>
76     ///                {
77     ///                    var matchedLink = Memory.GetLinkValue(link);
78     ///                    var newValue = Memory.GetLinkValue(link);
79     ///                    newValue[Constants.IndexPart] = Constants.Itself;
80     ///                    newValue[Constants.SourcePart] =
81     ↳ Equals(substitution[Constants.SourcePart], Constants.Itself) ?
82     ↳ matchedLink[Constants.IndexPart] : substitution[Constants.SourcePart];
83     ///                    newValue[Constants.TargetPart] =
84     ↳ Equals(substitution[Constants.TargetPart], Constants.Itself) ?
85     ↳ matchedLink[Constants.IndexPart] : substitution[Constants.TargetPart];
86     ///                    var matchDecision = matchedHandler(matchedLink, newValue);
87     ///                    if (Equals(matchDecision, Constants.Break))
88     ///                        return false;
89     ///                    if (!Equals(matchDecision, Constants.Skip))
90     ///                        transitions.Add(new Transition(matchedLink, newValue));
91     ///                    return true;
92     ///                };
93     ///                if (!Memory.Each(handler, restriction))
94     ///                    return Constants.Break;
95     ///            }
96     ///        }
97     ///        else
98     ///        {
99     ///            Func<T, bool> handler = link =>
100     ///            {
101     ///                var matchedLink = Memory.GetLinkValue(link);
102     ///                var matchDecision = matchedHandler(matchedLink, matchedLink);
103     ///                return !Equals(matchDecision, Constants.Break);
104     ///            };

```

```

93         if (!Memory.Each(handler, restriction))
94             return Constants.Break;
95     }
96 }
97 else
98 {
99     if (substitution != null)
100     {
101         transitions = new List<IList<T>>();
102         Func<T, bool> handler = link =>
103         {
104             var matchedLink = Memory.GetLinkValue(link);
105             transitions.Add(matchedLink);
106             return true;
107         };
108         if (!Memory.Each(handler, restriction))
109             return Constants.Break;
110     }
111     else
112     {
113         return Constants.Continue;
114     }
115 }
116 }
117 if (substitution != null)
118 {
119     // Есть причина делать замену (запись)
120     if (substitutedHandler != null)
121     {
122     }
123     else
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 //        return Constants.Break;
146 //}
147 //else // Update
148 //{
149 //    //List<IList<T>> matchedLinks = null;
150 //    if (matchedHandler != null)
151 //    {
152 //        matchedLinks = new List<IList<T>>();
153 //        Func<T, bool> handler = link =>
154 //        {
155 //            var matchedLink = Memory.GetLinkValue(link);
156 //            var matchDecision = matchedHandler(matchedLink);
157 //            if (Equals(matchDecision, Constants.Break))
158 //                return false;
159 //            if (!Equals(matchDecision, Constants.Skip))
160 //                matchedLinks.Add(matchedLink);
161 //            return true;
162 //        };
163 //        if (!Memory.Each(handler, restriction))
164 //            return Constants.Break;
165 //    }
166 //    if (!matchedLinks.IsNullOrEmpty())
167 //    {
168 //        var totalMatchedLinks = matchedLinks.Count;
169 //        for (var i = 0; i < totalMatchedLinks; i++)
170 //        {

```



```

170         //         var matchedLink = matchedLinks[i];
171         //         if (substitutedHandler != null)
172         //         {
173         //             var newValue = new List<T>(); // TODO: Prepare value to update here
174         //             // TODO: Decide is it actually needed to use Before and After
175         //             substitution handling.
176         //             var substitutedDecision = substitutedHandler(matchedLink,
177         //             newValue);
178         //             if (Equals(substitutedDecision, Constants.Break))
179         //                 return Constants.Break;
180         //             if (Equals(substitutedDecision, Constants.Continue))
181         //             {
182         //                 // Actual update here
183         //                 Memory.SetLinkValue(newValue);
184         //             }
185         //             if (Equals(substitutedDecision, Constants.Skip))
186         //             {
187         //                 // Cancel the update. TODO: decide use separate Cancel
188         //                 constant or Skip is enough?
189         //             }
190         //         }
191         //     }
192     }
193     return Constants.Continue;
194 }

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

```

```

{
    var linkToDelete = patternOrCondition[0];
    var before = Links.GetLink(linkToDelete);
    if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
        ↪ Constants.Break))
    {
        return Constants.Break;
    }
    var after = ArrayPool<TLink>.Empty;
    Links.Update(linkToDelete, Constants.Null, Constants.Null);
    Links.Delete(linkToDelete);
    if (matchHandler != null)
    {
        return substitutionHandler(before, after);
    }
    return Constants.Continue;
}
else
{
    throw new NotSupportedException();
}
}
else // Replace / Update
{
    if (patternOrCondition.Count == 1) //-V3125
    {
        var linkToUpdate = patternOrCondition[0];
        var before = Links.GetLink(linkToUpdate);
        if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
            ↪ Constants.Break))
        {
            return Constants.Break;
        }
        var after = (IList<TLink>)substitution.ToArray(); //-V3125
        if (_equalityComparer.Equals(after[0], default))
        {
            after[0] = linkToUpdate;
        }
        if (substitution.Count == 1)
        {
            if (!_equalityComparer.Equals(substitution[0], linkToUpdate))
            {
                after = Links.GetLink(substitution[0]);
                Links.Update(linkToUpdate, Constants.Null, Constants.Null);
                Links.Delete(linkToUpdate);
            }
        }
        else if (substitution.Count == 3)
        {
            Links.Update(after);
        }
        else
        {
            throw new NotSupportedException();
        }
        if (matchHandler != null)
        {
            return substitutionHandler(before, after);
        }
        return Constants.Continue;
    }
    else
    {
        throw new NotSupportedException();
    }
}
}
}

/// <remarks>
/// IList[IList[IList[T]]]
/// | | | |
/// | | | ----- |
/// | | | link |
/// | | ----- |
/// | | change |
/// | ----- |
/// | changes
/// </remarks>

```

```

316     public IList<IList<IList<TLink>>> Trigger(IList<TLink> condition, IList<TLink>
        ↳ substitution)
317     {
318         var changes = new List<IList<IList<TLink>>>();
319         Trigger(condition, AlwaysContinue, substitution, (before, after) =>
320         {
321             var change = new[] { before, after };
322             changes.Add(change);
323             return Constants.Continue;
324         });
325         return changes;
326     }
327
328     private TLink AlwaysContinue(IList<TLink> linkToMatch) => Constants.Continue;
329 }
330 }

```

#### ./Platform.Data.Doublets/DoubletComparer.cs

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets
7  {
8      /// <remarks>
9      /// TODO: Может стоит попробовать ref во всех методах (IRefEqualityComparer)
10     /// 2x faster with comparer
11     /// </remarks>
12     public class DoubletComparer<T> : IEqualityComparer<Doublet<T>>
13     {
14         public static readonly DoubletComparer<T> Default = new DoubletComparer<T>();
15
16         [MethodImpl(MethodImplOptions.AggressiveInlining)]
17         public bool Equals(Doublet<T> x, Doublet<T> y) => x.Equals(y);
18
19         [MethodImpl(MethodImplOptions.AggressiveInlining)]
20         public int GetHashCode(Doublet<T> obj) => obj.GetHashCode();
21     }
22 }

```

#### ./Platform.Data.Doublets/Doublet.cs

```

1  using System;
2  using System.Collections.Generic;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets
7  {
8      public struct Doublet<T> : IEquatable<Doublet<T>>
9      {
10         private static readonly EqualityComparer<T> _equalityComparer =
            ↳ EqualityComparer<T>.Default;
11
12         public T Source { get; set; }
13         public T Target { get; set; }
14
15         public Doublet(T source, T target)
16         {
17             Source = source;
18             Target = target;
19         }
20
21         public override string ToString() => $"{Source}->{Target}";
22
23         public bool Equals(Doublet<T> other) => _equalityComparer.Equals(Source, other.Source)
            ↳ && _equalityComparer.Equals(Target, other.Target);
24
25         public override bool Equals(object obj) => obj is Doublet<T> doublet ?
            ↳ base.Equals(doublet) : false;
26
27         public override int GetHashCode() => (Source, Target).GetHashCode();
28     }
29 }

```

#### ./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 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9 namespace Platform.Data.Doublets
10 {
11     public class Hybrid<T>
12     {
13         public readonly T Value;
14         public bool IsNothing => Convert.ToInt64(To.Signed(Value)) == 0;
15         public bool IsInternal => Convert.ToInt64(To.Signed(Value)) > 0;
16         public bool IsExternal => Convert.ToInt64(To.Signed(Value)) < 0;
17         public long AbsoluteValue =>
18             ↪ Platform.Numbers.Math.Abs(Convert.ToInt64(To.Signed(Value)));
19
20         public Hybrid(T value)
21         {
22             Ensure.Always.IsUnsignedInteger<T>();
23             Value = value;
24         }
25
26         public Hybrid(object value) => Value = To.UnsignedAs<T>(Convert.ChangeType(value,
27             ↪ Type<T>.SignedVersion));
28
29         public Hybrid(object value, bool isExternal)
30         {
31             var signedType = Type<T>.SignedVersion;
32             var signedValue = Convert.ChangeType(value, signedType);
33             var abs = typeof(Platform.Numbers.Math).GetTypeInfo().GetMethod("Abs").MakeGenericMe
34             ↪ thod(signedType);
35             var negate = typeof(Platform.Numbers.Math).GetTypeInfo().GetMethod("Negate").MakeGen
36             ↪ ericMethod(signedType);
37             var absoluteValue = abs.Invoke(null, new[] { signedValue });
38             var resultValue = isExternal ? negate.Invoke(null, new[] { absoluteValue }) :
39             ↪ absoluteValue;
40             Value = To.UnsignedAs<T>(resultValue);
41         }
42
43         public static implicit operator Hybrid<T>(T integer) => new Hybrid<T>(integer);
44
45         public static explicit operator Hybrid<T>(ulong integer) => new Hybrid<T>(integer);
46
47         public static explicit operator Hybrid<T>(long integer) => new Hybrid<T>(integer);
48
49         public static explicit operator Hybrid<T>(uint integer) => new Hybrid<T>(integer);
50
51         public static explicit operator Hybrid<T>(int integer) => new Hybrid<T>(integer);
52
53         public static explicit operator Hybrid<T>(ushort integer) => new Hybrid<T>(integer);
54
55         public static explicit operator Hybrid<T>(short integer) => new Hybrid<T>(integer);
56
57         public static explicit operator Hybrid<T>(byte integer) => new Hybrid<T>(integer);
58
59         public static explicit operator Hybrid<T>(sbyte integer) => new Hybrid<T>(integer);
60
61         public static implicit operator T(Hybrid<T> hybrid) => hybrid.Value;
62
63         public static explicit operator ulong(Hybrid<T> hybrid) =>
64             ↪ Convert.ToUInt64(hybrid.Value);
65
66         public static explicit operator long(Hybrid<T> hybrid) => hybrid.AbsoluteValue;
67
68         public static explicit operator uint(Hybrid<T> hybrid) => Convert.ToUInt32(hybrid.Value);
69
70         public static explicit operator int(Hybrid<T> hybrid) =>
71             ↪ Convert.ToInt32(hybrid.AbsoluteValue);
72
73         public static explicit operator ushort(Hybrid<T> hybrid) =>
74             ↪ Convert.ToUInt16(hybrid.Value);
75
76         public static explicit operator short(Hybrid<T> hybrid) =>
77             ↪ Convert.ToInt16(hybrid.AbsoluteValue);
78
79         public static explicit operator byte(Hybrid<T> hybrid) => Convert.ToByte(hybrid.Value);
80
81         public static explicit operator sbyte(Hybrid<T> hybrid) =>
82             ↪ Convert.ToSByte(hybrid.AbsoluteValue);

```

```

73
74     public override string ToString() => IsNothing ? default(T) == null ? "Nothing" :
    ↳ default(T).ToString() : IsExternal ? $"{<AbsoluteValue>}" : Value.ToString();
75 }
76 }

./Platform.Data.Doublets/ILinks.cs
1  using Platform.Data.Constants;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets
6  {
7      public interface ILinks<TLink> : ILinks<TLink, LinksCombinedConstants<TLink, TLink, int>>
8      {
9      }
10 }

./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 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
14
15 namespace Platform.Data.Doublets
16 {
17     public static class ILinksExtensions
18     {
19         public static void RunRandomCreations<TLink>(this ILinks<TLink> links, long
    ↳ amountOfCreations)
20         {
21             for (long i = 0; i < amountOfCreations; i++)
22             {
23                 var linksAddressRange = new Range<ulong>(0, (Integer<TLink>)links.Count());
24                 Integer<TLink> source = RandomHelpers.Default.NextUInt64(linksAddressRange);
25                 Integer<TLink> target = RandomHelpers.Default.NextUInt64(linksAddressRange);
26                 links.CreateAndUpdate(source, target);
27             }
28         }
29
30         public static void RunRandomSearches<TLink>(this ILinks<TLink> links, long
    ↳ amountOfSearches)
31         {
32             for (long i = 0; i < amountOfSearches; i++)
33             {
34                 var linkAddressRange = new Range<ulong>(1, (Integer<TLink>)links.Count());
35                 Integer<TLink> source = RandomHelpers.Default.NextUInt64(linkAddressRange);
36                 Integer<TLink> target = RandomHelpers.Default.NextUInt64(linkAddressRange);
37                 links.SearchOrDefault(source, target);
38             }
39         }
40
41         public static void RunRandomDeletions<TLink>(this ILinks<TLink> links, long
    ↳ amountOfDeletions)
42         {
43             var min = (ulong)amountOfDeletions > (Integer<TLink>)links.Count() ? 1 :
    ↳ (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
56         /// <remarks>

```

```

57  /// TODO: Возможно есть очень простой способ это сделать.
58  /// (Например просто удалить файл, или изменить его размер таким образом,
59  /// чтобы удалился весь контент)
60  /// Например через _header->AllocatedLinks в ResizableDirectMemoryLinks
61  /// </remarks>
62  public static void DeleteAll<TLink>(this ILinks<TLink> links)
63  {
64      var equalityComparer = EqualityComparer<TLink>.Default;
65      var comparer = Comparer<TLink>.Default;
66      for (var i = links.Count(); comparer.Compare(i, default) > 0; i =
        ↪ 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
76  public static TLink First<TLink>(this ILinks<TLink> links)
77  {
78      TLink firstLink = default;
79      var equalityComparer = EqualityComparer<TLink>.Default;
80      if (equalityComparer.Equals(links.Count(), default))
81      {
82          throw new Exception("В хранилище нет связей.");
83      }
84      links.Each(links.Constants.Any, links.Constants.Any, link =>
85      {
86          firstLink = link[links.Constants.IndexPart];
87          return links.Constants.Break;
88      });
89      if (equalityComparer.Equals(firstLink, default))
90      {
91          throw new Exception("В процессе поиска по хранилищу не было найдено связей.");
92      }
93      return firstLink;
94  }
95
96  public static bool IsInnerReference<TLink>(this ILinks<TLink> links, TLink reference)
97  {
98      var constants = links.Constants;
99      var comparer = Comparer<TLink>.Default;
100     return comparer.Compare(constants.MinPossibleIndex, reference) >= 0 &&
        ↪ comparer.Compare(reference, constants.MaxPossibleIndex) <= 0;
101 }
102
103 #region Paths
104
105 /// <remarks>
106 /// TODO: Как так? Как то что ниже может быть корректно?
107 /// Скорее всего практически не применимо
108 /// Предполагалось, что можно было конвертировать формируемый в проходе через
        ↪ 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 CheckPathExistance<TLink>(this ILinks<TLink> links, params TLink[]
        ↪ path)
113 {
114     var current = path[0];
115     //EnsureLinkExists(current, "path");
116     if (!links.Exists(current))
117     {
118         return false;
119     }
120     var equalityComparer = EqualityComparer<TLink>.Default;
121     var constants = links.Constants;
122     for (var i = 1; i < path.Length; i++)
123     {
124         var next = path[i];
125         var values = links.GetLink(current);
126         var source = values[constants.SourcePart];
127         var target = values[constants.TargetPart];
128         if (equalityComparer.Equals(source, target) && equalityComparer.Equals(source,
            ↪ next))
129         {

```

```

130         //throw new Exception(string.Format("Невозможно выбрать путь, так как и
        ↪ Source и Target совпадают с элементом пути {0}.", next));
131         return false;
132     }
133     if (!equalityComparer.Equals(next, source) && !equalityComparer.Equals(next,
        ↪ target))
134     {
135         //throw new Exception(string.Format("Невозможно продолжить путь через
        ↪ элемент пути {0}", next));
136         return false;
137     }
138     current = next;
139 }
140 return true;
141 }
142
143 /// <remarks>
144 /// Может потребовать дополнительного стека для PathElement's при использовании
    ↪ SequenceWalker.
145 /// </remarks>
146 public static TLink GetByKeyes<TLink>(this ILinks<TLink> links, TLink root, params int[]
    ↪ path)
147 {
148     links.EnsureLinkExists(root, "root");
149     var currentLink = root;
150     for (var i = 0; i < path.Length; i++)
151     {
152         currentLink = links.GetLink(currentLink)[path[i]];
153     }
154     return currentLink;
155 }
156
157 public static TLink GetSquareMatrixSequenceElementByIndex<TLink>(this ILinks<TLink>
    ↪ links, TLink root, ulong size, ulong index)
158 {
159     var constants = links.Constants;
160     var source = constants.SourcePart;
161     var target = constants.TargetPart;
162     if (!Platform.Numbers.Math.IsPowerOfTwo(size))
163     {
164         throw new ArgumentOutOfRangeException(nameof(size), "Sequences with sizes other
        ↪ than powers of two are not supported.");
165     }
166     var path = new BitArray(BitConverter.GetBytes(index));
167     var length = Bit.GetLowestPosition(size);
168     links.EnsureLinkExists(root, "root");
169     var currentLink = root;
170     for (var i = length - 1; i >= 0; i--)
171     {
172         currentLink = links.GetLink(currentLink)[path[i] ? target : source];
173     }
174     return currentLink;
175 }
176
177 #endregion
178
179 /// <summary>
180 /// Возвращает индекс указанной связи.
181 /// </summary>
182 /// <param name="links">Хранилище связей.</param>
183 /// <param name="link">Связь представленная списком, состоящим из её адреса и
    ↪ содержимого.</param>
184 /// <returns>Индекс начальной связи для указанной связи.</returns>
185 [MethodImpl(MethodImplOptions.AggressiveInlining)]
186 public static TLink GetIndex<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
    ↪ 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) =>
    ↪ links.GetLink(link)[links.Constants.SourcePart];
196
197 /// <summary>

```

```

198 /// Возвращает индекс начальной (Source) связи для указанной связи.
199 /// </summary>
200 /// <param name="links">Хранилище связей.</param>
201 /// <param name="link">Связь представленная списком, состоящим из её адреса и
    ↳ содержимого.</param>
202 /// <returns>Индекс начальной связи для указанной связи.</returns>
203 [MethodImpl(MethodImplOptions.AggressiveInlining)]
204 public static TLink GetSource<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
    ↳ link[links.Constants.SourcePart];
205
206 /// <summary>
207 /// Возвращает индекс конечной (Target) связи для указанной связи.
208 /// </summary>
209 /// <param name="links">Хранилище связей.</param>
210 /// <param name="link">Индекс связи.</param>
211 /// <returns>Индекс конечной связи для указанной связи.</returns>
212 [MethodImpl(MethodImplOptions.AggressiveInlining)]
213 public static TLink GetTarget<TLink>(this ILinks<TLink> links, TLink link) =>
    ↳ links.GetLink(link)[links.Constants.TargetPart];
214
215 /// <summary>
216 /// Возвращает индекс конечной (Target) связи для указанной связи.
217 /// </summary>
218 /// <param name="links">Хранилище связей.</param>
219 /// <param name="link">Связь представленная списком, состоящим из её адреса и
    ↳ содержимого.</param>
220 /// <returns>Индекс конечной связи для указанной связи.</returns>
221 [MethodImpl(MethodImplOptions.AggressiveInlining)]
222 public static TLink GetTarget<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
    ↳ link[links.Constants.TargetPart];
223
224 /// <summary>
225 /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
    ↳ (handler) для каждой подходящей связи.
226 /// </summary>
227 /// <param name="links">Хранилище связей.</param>
228 /// <param name="handler">Обработчик каждой подходящей связи.</param>
229 /// <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>
230 /// <returns>True, в случае если проход по связям не был прерван и False в обратном
    ↳ случае.</returns>
231 [MethodImpl(MethodImplOptions.AggressiveInlining)]
232 public static bool Each<TLink>(this ILinks<TLink> links, Func<IList<TLink>, TLink>
    ↳ handler, params TLink[] restrictions)
233     => EqualityComparer<TLink>.Default.Equals(links.Each(handler, restrictions),
    ↳ links.Constants.Continue);
234
235 /// <summary>
236 /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
    ↳ (handler) для каждой подходящей связи.
237 /// </summary>
238 /// <param name="links">Хранилище связей.</param>
239 /// <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>
240 /// <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>
241 /// <param name="handler">Обработчик каждой подходящей связи.</param>
242 /// <returns>True, в случае если проход по связям не был прерван и False в обратном
    ↳ случае.</returns>
243 [MethodImpl(MethodImplOptions.AggressiveInlining)]
244 public static bool Each<TLink>(this ILinks<TLink> links, TLink source, TLink target,
    ↳ Func<TLink, bool> handler)
245 {
246     var constants = links.Constants;
247     return links.Each(link => handler(link[constants.IndexPart]) ? constants.Continue :
    ↳ constants.Break, constants.Any, source, target);
248 }
249
250 /// <summary>
251 /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
    ↳ (handler) для каждой подходящей связи.
252 /// </summary>
253 /// <param name="links">Хранилище связей.</param>

```



```

254 /// <param name="source">Значение, определяющее соответствующие шаблону связи.
255   → (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве начала,
256   → Constants.Any - любое начало, 1..∞ конкретное начало)</param>
257 /// <param name="target">Значение, определяющее соответствующие шаблону связи.
258   → (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве конца,
259   → Constants.Any - любой конец, 1..∞ конкретный конец)</param>
260 /// <param name="handler">Обработчик каждой подходящей связи.</param>
261 /// <returns>True, в случае если проход по связям не был прерван и False в обратном
262   → случае.</returns>
263 [MethodImpl(MethodImplOptions.AggressiveInlining)]
264 public static bool Each<TLink>(this ILinks<TLink> links, TLink source, TLink target,
265   → Func<IList<TLink>, TLink> handler)
266 {
267     var constants = links.Constants;
268     return links.Each(handler, constants.Any, source, target);
269 }
270
271 [MethodImpl(MethodImplOptions.AggressiveInlining)]
272 public static IList<IList<TLink>> All<TLink>(this ILinks<TLink> links, params TLink[]
273   → restrictions)
274 {
275     long arraySize = (Integer<TLink>)links.Count(restrictions);
276     var array = new IList<TLink>[arraySize];
277     if (arraySize > 0)
278     {
279         var filler = new ArrayFiller<IList<TLink>, TLink>(array,
280             → links.Constants.Continue);
281         links.Each(filler.AddAndReturnConstant, restrictions);
282     }
283     return array;
284 }
285
286 [MethodImpl(MethodImplOptions.AggressiveInlining)]
287 public static IList<TLink> AllIndices<TLink>(this ILinks<TLink> links, params TLink[]
288   → restrictions)
289 {
290     long arraySize = (Integer<TLink>)links.Count(restrictions);
291     var array = new TLink[arraySize];
292     if (arraySize > 0)
293     {
294         var filler = new ArrayFiller<TLink, TLink>(array, links.Constants.Continue);
295         links.Each(filler.AddFirstAndReturnConstant, restrictions);
296     }
297     return array;
298 }
299
300 /// <summary>
301 /// Возвращает значение, определяющее существует ли связь с указанными началом и концом
302   → в хранилище связей.
303 /// </summary>
304 /// <param name="links">Хранилище связей.</param>
305 /// <param name="source">Начало связи.</param>
306 /// <param name="target">Конец связи.</param>
307 /// <returns>Значение, определяющее существует ли связь.</returns>
308 [MethodImpl(MethodImplOptions.AggressiveInlining)]
309 public static bool Exists<TLink>(this ILinks<TLink> links, TLink source, TLink target)
310   → => Comparer<TLink>.Default.Compare(links.Count(links.Constants.Any, source, target),
311   → default) > 0;
312
313 #region Ensure
314 // TODO: May be move to EnsureExtensions or make it both there and here
315
316 [MethodImpl(MethodImplOptions.AggressiveInlining)]
317 public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links, TLink
318   → reference, string argumentName)
319 {
320     if (links.IsInnerReference(reference) && !links.Exists(reference))
321     {
322         throw new ArgumentLinkDoesNotExistsException<TLink>(reference, argumentName);
323     }
324 }
325
326 [MethodImpl(MethodImplOptions.AggressiveInlining)]
327 public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links,
328   → IList<TLink> restrictions, string argumentName)
329 {
330     for (int i = 0; i < restrictions.Count; i++)

```

```

317     {
318         links.EnsureInnerReferenceExists(restrictions[i], argumentName);
319     }
320 }
321
322 [MethodImpl(MethodImplOptions.AggressiveInlining)]
323 public static void EnsureLinkIsAnyOrExists<TLink>(this ILinks<TLink> links, IList<TLink>
    ↪ restrictions)
324 {
325     for (int i = 0; i < restrictions.Count; i++)
326     {
327         links.EnsureLinkIsAnyOrExists(restrictions[i], nameof(restrictions));
328     }
329 }
330
331 [MethodImpl(MethodImplOptions.AggressiveInlining)]
332 public static void EnsureLinkIsAnyOrExists<TLink>(this ILinks<TLink> links, TLink link,
    ↪ string argumentName)
333 {
334     var equalityComparer = EqualityComparer<TLink>.Default;
335     if (!equalityComparer.Equals(link, links.Constants.Any) && !links.Exists(link))
336     {
337         throw new ArgumentLinkDoesNotExistsException<TLink>(link, argumentName);
338     }
339 }
340
341 [MethodImpl(MethodImplOptions.AggressiveInlining)]
342 public static void EnsureLinkIsItselfOrExists<TLink>(this ILinks<TLink> links, TLink
    ↪ link, string argumentName)
343 {
344     var equalityComparer = EqualityComparer<TLink>.Default;
345     if (!equalityComparer.Equals(link, links.Constants.Itself) && !links.Exists(link))
346     {
347         throw new ArgumentLinkDoesNotExistsException<TLink>(link, argumentName);
348     }
349 }
350
351 /// <param name="links">Хранилище связей.</param>
352 [MethodImpl(MethodImplOptions.AggressiveInlining)]
353 public static void EnsureDoesNotExists<TLink>(this ILinks<TLink> links, TLink source,
    ↪ TLink target)
354 {
355     if (links.Exists(source, target))
356     {
357         throw new LinkWithSameValueAlreadyExistsException();
358     }
359 }
360
361 /// <param name="links">Хранилище связей.</param>
362 public static void EnsureNoUsages<TLink>(this ILinks<TLink> links, TLink link)
363 {
364     if (links.HasUsages(link))
365     {
366         throw new ArgumentLinkHasDependenciesException<TLink>(link);
367     }
368 }
369
370 /// <param name="links">Хранилище связей.</param>
371 public static void EnsureCreated<TLink>(this ILinks<TLink> links, params TLink[]
    ↪ addresses) => links.EnsureCreated(links.Create, addresses);
372
373 /// <param name="links">Хранилище связей.</param>
374 public static void EnsurePointsCreated<TLink>(this ILinks<TLink> links, params TLink[]
    ↪ addresses) => links.EnsureCreated(links.CreatePoint, addresses);
375
376 /// <param name="links">Хранилище связей.</param>
377 public static void EnsureCreated<TLink>(this ILinks<TLink> links, Func<TLink> creator,
    ↪ params TLink[] addresses)
378 {
379     var constants = links.Constants;
380     var nonExistentAddresses = new HashSet<ulong>(addresses.Where(x =>
    ↪ !links.Exists(x)).Select(x => (ulong)(Integer<TLink>)x));
381     if (nonExistentAddresses.Count > 0)
382     {
383         var max = nonExistentAddresses.Max();
384         // TODO: Эту верхнюю границу нужно разрешить переопределять (проверить
    ↪ применяется ли эта логика)
385         max = System.Math.Min(max, (Integer<TLink>)constants.MaxPossibleIndex);

```

```

386     var createdLinks = new List<TLink>();
387     var equalityComparer = EqualityComparer<TLink>.Default;
388     TLink createdLink = creator();
389     while (!equalityComparer.Equals(createdLink, (Integer<TLink>)max))
390     {
391         createdLinks.Add(createdLink);
392     }
393     for (var i = 0; i < createdLinks.Count; i++)
394     {
395         if (!nonExistentAddresses.Contains((Integer<TLink>)createdLinks[i]))
396         {
397             links.Delete(createdLinks[i]);
398         }
399     }
400 }
401
402 #endregion
403
404 /// <param name="links">Хранилище связей.</param>
405 public static ulong CountUsages<TLink>(this ILinks<TLink> links, TLink link)
406 {
407     var constants = links.Constants;
408     var values = links.GetLink(link);
409     ulong usagesAsSource = (Integer<TLink>)links.Count(new Link<TLink>(constants.Any,
410         ↪ link, constants.Any));
411     var equalityComparer = EqualityComparer<TLink>.Default;
412     if (equalityComparer.Equals(values[constants.SourcePart], link))
413     {
414         usagesAsSource--;
415     }
416     ulong usagesAsTarget = (Integer<TLink>)links.Count(new Link<TLink>(constants.Any,
417         ↪ constants.Any, link));
418     if (equalityComparer.Equals(values[constants.TargetPart], link))
419     {
420         usagesAsTarget--;
421     }
422     return usagesAsSource + usagesAsTarget;
423 }
424
425 /// <param name="links">Хранилище связей.</param>
426 [MethodImpl(MethodImplOptions.AggressiveInlining)]
427 public static bool HasUsages<TLink>(this ILinks<TLink> links, TLink link) =>
428     ↪ links.CountUsages(link) > 0;
429
430 /// <param name="links">Хранилище связей.</param>
431 [MethodImpl(MethodImplOptions.AggressiveInlining)]
432 public static bool Equals<TLink>(this ILinks<TLink> links, TLink link, TLink source,
433     ↪ TLink target)
434 {
435     var constants = links.Constants;
436     var values = links.GetLink(link);
437     var equalityComparer = EqualityComparer<TLink>.Default;
438     return equalityComparer.Equals(values[constants.SourcePart], source) &&
439         ↪ equalityComparer.Equals(values[constants.TargetPart], target);
440 }
441
442 /// <summary>
443 /// Выполняет поиск связи с указанными Source (началом) и Target (концом).
444 /// </summary>
445 /// <param name="links">Хранилище связей.</param>
446 /// <param name="source">Индекс связи, которая является началом для искомой
447     ↪ связи.</param>
448 /// <param name="target">Индекс связи, которая является концом для искомой связи.</param>
449 /// <returns>Индекс искомой связи с указанными Source (началом) и Target
450     ↪ (концом).</returns>
451 [MethodImpl(MethodImplOptions.AggressiveInlining)]
452 public static TLink SearchOrDefault<TLink>(this ILinks<TLink> links, TLink source, TLink
453     ↪ target)
454 {
455     var constants = links.Constants;
456     var setter = new Setter<TLink, TLink>(constants.Continue, constants.Break, default);
457     links.Each(setter.SetFirstAndReturnFalse, constants.Any, source, target);
458     return setter.Result;
459 }
460
461 /// <param name="links">Хранилище связей.</param>
462 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

456 public static TLink CreatePoint<TLink>(this ILinks<TLink> links)
457 {
458     var link = links.Create();
459     return links.Update(link, link, link);
460 }
461
462 /// <param name="links">Хранилище связей.</param>
463 [MethodImpl(MethodImplOptions.AggressiveInlining)]
464 public static TLink CreateAndUpdate<TLink>(this ILinks<TLink> links, TLink source, TLink
    ↪ target) => links.Update(links.Create(), source, target);
465
466 /// <summary>
467 /// Обновляет связь с указанными началом (Source) и концом (Target)
468 /// на связь с указанными началом (NewSource) и концом (NewTarget).
469 /// </summary>
470 /// <param name="links">Хранилище связей.</param>
471 /// <param name="link">Индекс обновляемой связи.</param>
472 /// <param name="newSource">Индекс связи, которая является началом связи, на которую
    ↪ выполняется обновление.</param>
473 /// <param name="newTarget">Индекс связи, которая является концом связи, на которую
    ↪ выполняется обновление.</param>
474 /// <returns>Индекс обновлённой связи.</returns>
475 [MethodImpl(MethodImplOptions.AggressiveInlining)]
476 public static TLink Update<TLink>(this ILinks<TLink> links, TLink link, TLink newSource,
    ↪ TLink newTarget) => links.Update(new Link<TLink>(link, newSource, newTarget));
477
478 /// <summary>
479 /// Обновляет связь с указанными началом (Source) и концом (Target)
480 /// на связь с указанными началом (NewSource) и концом (NewTarget).
481 /// </summary>
482 /// <param name="links">Хранилище связей.</param>
483 /// <param name="restrictions">Ограничения на содержимое связей. Каждое ограничение
    ↪ может иметь значения: Constants.Null - 0-я связь, обозначающая ссылку на пустоту,
    ↪ Itself - требование установить ссылку на себя, 1.. $\infty$  конкретный адрес другой
    ↪ связи.</param>
484 /// <returns>Индекс обновлённой связи.</returns>
485 [MethodImpl(MethodImplOptions.AggressiveInlining)]
486 public static TLink Update<TLink>(this ILinks<TLink> links, params TLink[] restrictions)
487 {
488     if (restrictions.Length == 2)
489     {
490         return links.MergeAndDelete(restrictions[0], restrictions[1]);
491     }
492     if (restrictions.Length == 4)
493     {
494         return links.UpdateOrCreateOrGet(restrictions[0], restrictions[1],
            ↪ restrictions[2], restrictions[3]);
495     }
496     else
497     {
498         return links.Update(restrictions);
499     }
500 }
501
502 [MethodImpl(MethodImplOptions.AggressiveInlining)]
503 public static IList<TLink> ResolveConstantAsSelfReference<TLink>(this ILinks<TLink>
    ↪ links, TLink constant, IList<TLink> restrictions)
504 {
505     var equalityComparer = EqualityComparer<TLink>.Default;
506     var constants = links.Constants;
507     var index = restrictions[constants.IndexPart];
508     var source = restrictions[constants.SourcePart];
509     var target = restrictions[constants.TargetPart];
510     source = equalityComparer.Equals(source, constant) ? index : source;
511     target = equalityComparer.Equals(target, constant) ? index : target;
512     return new Link<TLink>(index, source, target);
513 }
514
515 /// <summary>
516 /// Создаёт связь (если она не существовала), либо возвращает индекс существующей связи
    ↪ с указанными Source (началом) и Target (концом).
517 /// </summary>
518 /// <param name="links">Хранилище связей.</param>
519 /// <param name="source">Индекс связи, которая является началом на создаваемой
    ↪ связи.</param>
520 /// <param name="target">Индекс связи, которая является концом для создаваемой
    ↪ связи.</param>

```

```

521 /// <returns>Индекс связи, с указанным Source (началом) и Target (концом)</returns>
522 [MethodImpl(MethodImplOptions.AggressiveInlining)]
523 public static TLink GetOrCreate<TLink>(this ILinks<TLink> links, TLink source, TLink
    ↳ target)
524 {
525     var link = links.SearchOrDefault(source, target);
526     if (EqualityComparer<TLink>.Default.Equals(link, default))
527     {
528         link = links.CreateAndUpdate(source, target);
529     }
530     return link;
531 }
532
533 /// <summary>
534 /// Обновляет связь с указанными началом (Source) и концом (Target)
535 /// на связь с указанными началом (NewSource) и концом (NewTarget).
536 /// </summary>
537 /// <param name="links">Хранилище связей.</param>
538 /// <param name="source">Индекс связи, которая является началом обновляемой
    ↳ связи.</param>
539 /// <param name="target">Индекс связи, которая является концом обновляемой связи.</param>
540 /// <param name="newSource">Индекс связи, которая является началом связи, на которую
    ↳ выполняется обновление.</param>
541 /// <param name="newTarget">Индекс связи, которая является концом связи, на которую
    ↳ выполняется обновление.</param>
542 /// <returns>Индекс обновлённой связи.</returns>
543 [MethodImpl(MethodImplOptions.AggressiveInlining)]
544 public static TLink UpdateOrCreateOrGet<TLink>(this ILinks<TLink> links, TLink source,
    ↳ TLink target, TLink newSource, TLink newTarget)
545 {
546     var equalityComparer = EqualityComparer<TLink>.Default;
547     var link = links.SearchOrDefault(source, target);
548     if (equalityComparer.Equals(link, default))
549     {
550         return links.CreateAndUpdate(newSource, newTarget);
551     }
552     if (equalityComparer.Equals(newSource, source) && equalityComparer.Equals(newTarget,
    ↳ target))
553     {
554         return link;
555     }
556     return links.Update(link, newSource, newTarget);
557 }
558
559 /// <summary>Удаляет связь с указанными началом (Source) и концом (Target).</summary>
560 /// <param name="links">Хранилище связей.</param>
561 /// <param name="source">Индекс связи, которая является началом удаляемой связи.</param>
562 /// <param name="target">Индекс связи, которая является концом удаляемой связи.</param>
563 [MethodImpl(MethodImplOptions.AggressiveInlining)]
564 public static TLink DeleteIfExists<TLink>(this ILinks<TLink> links, TLink source, TLink
    ↳ target)
565 {
566     var link = links.SearchOrDefault(source, target);
567     if (!EqualityComparer<TLink>.Default.Equals(link, default))
568     {
569         links.Delete(link);
570         return link;
571     }
572     return default;
573 }
574
575 /// <summary>Удаляет несколько связей.</summary>
576 /// <param name="links">Хранилище связей.</param>
577 /// <param name="deletedLinks">Список адресов связей к удалению.</param>
578 [MethodImpl(MethodImplOptions.AggressiveInlining)]
579 public static void DeleteMany<TLink>(this ILinks<TLink> links, IList<TLink> deletedLinks)
580 {
581     for (int i = 0; i < deletedLinks.Count; i++)
582     {
583         links.Delete(deletedLinks[i]);
584     }
585 }
586
587 /// <remarks>Before execution of this method ensure that deleted link is detached (all
    ↳ values - source and target are reset to null) or it might enter into infinite
    ↳ recursion.</remarks>
588 public static void DeleteAllUsages<TLink>(this ILinks<TLink> links, TLink linkIndex)
589 {

```

```

590     var anyConstant = links.Constants.Any;
591     var usagesAsSourceQuery = new Link<TLink>(anyConstant, linkIndex, anyConstant);
592     links.DeleteByQuery(usagesAsSourceQuery);
593     var usagesAsTargetQuery = new Link<TLink>(anyConstant, linkIndex, anyConstant);
594     links.DeleteByQuery(usagesAsTargetQuery);
595 }
596
597 public static void DeleteByQuery<TLink>(this ILinks<TLink> links, Link<TLink> query)
598 {
599     var count = (Integer<TLink>)links.Count(query);
600     if (count > 0)
601     {
602         var queryResult = new TLink[count];
603         var queryResultFiller = new ArrayFiller<TLink, TLink>(queryResult,
604             ↪ links.Constants.Continue);
605         links.Each(queryResultFiller.AddFirstAndReturnConstant, query);
606         for (var i = (long)count - 1; i >= 0; i--)
607         {
608             links.Delete(queryResult[i]);
609         }
610     }
611 }
612
613 // TODO: Move to Platform.Data
614 public static bool AreValuesReset<TLink>(this ILinks<TLink> links, TLink linkIndex)
615 {
616     var nullConstant = links.Constants.Null;
617     var equalityComparer = EqualityComparer<TLink>.Default;
618     var link = links.GetLink(linkIndex);
619     for (int i = 1; i < link.Count; i++)
620     {
621         if (!equalityComparer.Equals(link[i], nullConstant))
622         {
623             return false;
624         }
625     }
626     return true;
627 }
628
629 // TODO: Create a universal version of this method in Platform.Data (with using of for
630 ↪ loop)
631 public static void ResetValues<TLink>(this ILinks<TLink> links, TLink linkIndex)
632 {
633     var nullConstant = links.Constants.Null;
634     var updateRequest = new Link<TLink>(linkIndex, nullConstant, nullConstant);
635     links.Update(updateRequest);
636 }
637
638 // TODO: Create a universal version of this method in Platform.Data (with using of for
639 ↪ loop)
640 public static void EnforceResetValues<TLink>(this ILinks<TLink> links, TLink linkIndex)
641 {
642     if (!links.AreValuesReset(linkIndex))
643     {
644         links.ResetValues(linkIndex);
645     }
646 }
647
648 /// <summary>
649 /// Merging two usages graphs, all children of old link moved to be children of new link
650 ↪ or deleted.
651 /// </summary>
652 public static TLink MergeUsages<TLink>(this ILinks<TLink> links, TLink oldLinkIndex,
653 ↪ TLink newLinkIndex)
654 {
655     var equalityComparer = EqualityComparer<TLink>.Default;
656     if (!equalityComparer.Equals(oldLinkIndex, newLinkIndex))
657     {
658         var constants = links.Constants;
659         var usagesAsSourceQuery = new Link<TLink>(constants.Any, oldLinkIndex,
660             ↪ constants.Any);
661         long usagesAsSourceCount = (Integer<TLink>)links.Count(usagesAsSourceQuery);
662         var usagesAsTargetQuery = new Link<TLink>(constants.Any, constants.Any,
663             ↪ oldLinkIndex);
664         long usagesAsTargetCount = (Integer<TLink>)links.Count(usagesAsTargetQuery);
665         var isStandalonePoint = Point<TLink>.IsFullPoint(links.GetLink(oldLinkIndex)) &&
666             ↪ usagesAsSourceCount == 1 && usagesAsTargetCount == 1;
667         if (!isStandalonePoint)

```

```

660     {
661         var totalUsages = usagesAsSourceCount + usagesAsTargetCount;
662         if (totalUsages > 0)
663         {
664             var usages = ArrayPool.Allocate<TLink>(totalUsages);
665             var usagesFiller = new ArrayFiller<TLink, TLink>(usages,
666                 ↪ links.Constants.Continue);
667             var i = 0L;
668             if (usagesAsSourceCount > 0)
669             {
670                 links.Each(usagesFiller.AddFirstAndReturnConstant,
671                     ↪ usagesAsSourceQuery);
672                 for (; i < usagesAsSourceCount; i++)
673                 {
674                     var usage = usages[i];
675                     if (!equalityComparer.Equals(usage, oldLinkIndex))
676                     {
677                         links.Update(usage, newLinkIndex, links.GetTarget(usage));
678                     }
679                 }
680             }
681             if (usagesAsTargetCount > 0)
682             {
683                 links.Each(usagesFiller.AddFirstAndReturnConstant,
684                     ↪ usagesAsTargetQuery);
685                 for (; i < usages.Length; i++)
686                 {
687                     var usage = usages[i];
688                     if (!equalityComparer.Equals(usage, oldLinkIndex))
689                     {
690                         links.Update(usage, links.GetSource(usage), newLinkIndex);
691                     }
692                 }
693             }
694             ArrayPool.Free(usages);
695         }
696     }
697     return newLinkIndex;
698 }
699
700 /// <summary>
701 /// Replace one link with another (replaced link is deleted, children are updated or
702 /// ↪ deleted).
703 /// </summary>
704 [MethodImpl(MethodImplOptions.AggressiveInlining)]
705 public static TLink MergeAndDelete<TLink>(this ILinks<TLink> links, TLink oldLinkIndex,
706     ↪ TLink newLinkIndex)
707 {
708     var equalityComparer = EqualityComparer<TLink>.Default;
709     if (!equalityComparer.Equals(oldLinkIndex, newLinkIndex))
710     {
711         links.MergeUsages(oldLinkIndex, newLinkIndex);
712         links.Delete(oldLinkIndex);
713     }
714     return newLinkIndex;
715 }
716 }
717 }

```

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

```

1  using System.Collections.Generic;
2  using Platform.Interfaces;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Incrementers
7  {
8      public class FrequencyIncrementer<TLink> : LinksOperatorBase<TLink>, IIncrementer<TLink>
9      {
10         private static readonly EqualityComparer<TLink> _equalityComparer =
11             ↪ EqualityComparer<TLink>.Default;
12
13         private readonly TLink _frequencyMarker;
14         private readonly TLink _unaryOne;
15         private readonly IIncrementer<TLink> _unaryNumberIncrementer;
16
17         public FrequencyIncrementer(ILinks<TLink> links, TLink frequencyMarker, TLink unaryOne,
18             ↪ 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  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Incrementers
7  {
8      public class UnaryNumberIncrementer<TLink> : LinksOperatorBase<TLink>, IIncrementer<TLink>
9      {
10         private static readonly EqualityComparer<TLink> _equalityComparer =
11             ⇨ EqualityComparer<TLink>.Default;
12
13         private readonly TLink _unaryOne;
14
15         public UnaryNumberIncrementer(ILinks<TLink> links, TLink unaryOne) : base(links) =>
16             ⇨ _unaryOne = unaryOne;
17
18         public TLink Increment(TLink unaryNumber)
19         {
20             if (_equalityComparer.Equals(unaryNumber, _unaryOne))
21             {
22                 return Links.GetOrCreate(_unaryOne, _unaryOne);
23             }
24             var source = Links.GetSource(unaryNumber);
25             var target = Links.GetTarget(unaryNumber);
26             if (_equalityComparer.Equals(source, target))
27             {
28                 return Links.GetOrCreate(unaryNumber, _unaryOne);
29             }
30             else
31             {
32                 return Links.GetOrCreate(source, Increment(target));
33             }
34         }
35     }
36 }

```

./Platform.Data.Doublets/ISynchronizedLinks.cs

```

1  using Platform.Data.Constants;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets
6  {
7      public interface ISynchronizedLinks<TLink> : ISynchronizedLinks<TLink, ILinks<TLink>,
8          ⇨ LinksCombinedConstants<TLink, TLink, int>>, ILinks<TLink>
9      {
10     }
11 }

```

./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 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
11
12 namespace Platform.Data.Doublets
13 {
14     /// <summary>
15     /// Структура описывающая уникальную связь.
16     /// </summary>
17     public struct Link<TLink> : IEquatable<Link<TLink>>, IReadOnlyList<TLink>, IList<TLink>
18     {
19         public static readonly Link<TLink> Null = new Link<TLink>();
20
21         private static readonly LinksCombinedConstants<bool, TLink, int> _constants =
22             ↪ Default<LinksCombinedConstants<bool, TLink, int>>.Instance;
23         private static readonly EqualityComparer<TLink> _equalityComparer =
24             ↪ EqualityComparer<TLink>.Default;
25
26         private const int Length = 3;
27
28         public readonly TLink Index;
29         public readonly TLink Source;
30         public readonly TLink Target;
31
32         public Link(params TLink[] values)
33         {
34             Index = values.Length > _constants.IndexPart ? values[_constants.IndexPart] :
35                 ↪ _constants.Null;
36             Source = values.Length > _constants.SourcePart ? values[_constants.SourcePart] :
37                 ↪ _constants.Null;
38             Target = values.Length > _constants.TargetPart ? values[_constants.TargetPart] :
39                 ↪ _constants.Null;
40         }
41
42         public Link(IList<TLink> values)
43         {
44             Index = values.Count > _constants.IndexPart ? values[_constants.IndexPart] :
45                 ↪ _constants.Null;
46             Source = values.Count > _constants.SourcePart ? values[_constants.SourcePart] :
47                 ↪ _constants.Null;
48             Target = values.Count > _constants.TargetPart ? values[_constants.TargetPart] :
49                 ↪ _constants.Null;
50         }
51
52         public Link(TLink index, TLink source, TLink target)
53         {
54             Index = index;
55             Source = source;
56             Target = target;
57         }
58
59         public Link(TLink source, TLink target)
60             : this(_constants.Null, source, target)
61         {
62             Source = source;
63             Target = target;
64         }
65
66         public static Link<TLink> Create(TLink source, TLink target) => new Link<TLink>(source,
67             ↪ target);
68
69         public override int GetHashCode() => (Index, Source, Target).GetHashCode();
70
71         public bool IsNull() => _equalityComparer.Equals(Index, _constants.Null)
72             && _equalityComparer.Equals(Source, _constants.Null)
73             && _equalityComparer.Equals(Target, _constants.Null);
74
75         public override bool Equals(object other) => other is Link<TLink> &&
76             ↪ Equals((Link<TLink>)other);
77
78         public bool Equals(Link<TLink> other) => _equalityComparer.Equals(Index, other.Index)
79             && _equalityComparer.Equals(Source, other.Source)
80             && _equalityComparer.Equals(Target, other.Target);
81
82         public static string ToString(TLink index, TLink source, TLink target) => $"({index}:
83             ↪ {source}->{target})";
84
85         public static string ToString(TLink source, TLink target) => $"({source}->{target})";
86
87         public static implicit operator TLink[] (Link<TLink> link) => link.ToArray();

```

```

77
78 public static implicit operator Link<TLink>(TLink[] linkArray) => new
    ↳ Link<TLink>(linkArray);
79
80 public override string ToString() => _equalityComparer.Equals(Index, _constants.Null) ?
    ↳ ToString(Source, Target) : ToString(Index, Source, Target);
81
82 #region IList
83
84 public int Count => Length;
85
86 public bool IsReadOnly => true;
87
88 public TLink this[int index]
89 {
90     get
91     {
92         Ensure.Always.ArgumentInRange(index, new Range<int>(0, Length - 1),
    ↳ nameof(index));
93         if (index == _constants.IndexPart)
94         {
95             return Index;
96         }
97         if (index == _constants.SourcePart)
98         {
99             return Source;
100         }
101         if (index == _constants.TargetPart)
102         {
103             return Target;
104         }
105         throw new NotSupportedException(); // Impossible path due to
    ↳ Ensure.ArgumentInRange
106     }
107     set => throw new NotSupportedException();
108 }
109
110 IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
111
112 public IEnumerator<TLink> GetEnumerator()
113 {
114     yield return Index;
115     yield return Source;
116     yield return Target;
117 }
118
119 public void Add(TLink item) => throw new NotSupportedException();
120
121 public void Clear() => throw new NotSupportedException();
122
123 public bool Contains(TLink item) => IndexOf(item) >= 0;
124
125 public void CopyTo(TLink[] array, int arrayIndex)
126 {
127     Ensure.Always.ArgumentNotNull(array, nameof(array));
128     Ensure.Always.ArgumentInRange(arrayIndex, new Range<int>(0, array.Length - 1),
    ↳ nameof(arrayIndex));
129     if (arrayIndex + Length > array.Length)
130     {
131         throw new InvalidOperationException();
132     }
133     array[arrayIndex++] = Index;
134     array[arrayIndex++] = Source;
135     array[arrayIndex] = Target;
136 }
137
138 public bool Remove(TLink item) => Throw.A.NotSupportedExceptionAndReturn<bool>();
139
140 public int IndexOf(TLink item)
141 {
142     if (_equalityComparer.Equals(Index, item))
143     {
144         return _constants.IndexPart;
145     }
146     if (_equalityComparer.Equals(Source, item))
147     {
148         return _constants.SourcePart;
149     }
150     if (_equalityComparer.Equals(Target, item))

```

```

151         {
152             return _constants.TargetPart;
153         }
154         return -1;
155     }
156
157     public void Insert(int index, TLink item) => throw new NotSupportedException();
158
159     public void RemoveAt(int index) => throw new NotSupportedException();
160
161     #endregion
162 }
163 }

```

./Platform.Data.Doublets/LinkExtensions.cs

```

1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  namespace Platform.Data.Doublets
4  {
5      public static class LinkExtensions
6      {
7          public static bool IsFullPoint<TLink>(this Link<TLink> link) =>
8              ⇨ Point<TLink>.IsFullPoint(link);
9          public static bool IsPartialPoint<TLink>(this Link<TLink> link) =>
10             ⇨ Point<TLink>.IsPartialPoint(link);
11     }
12 }

```

./Platform.Data.Doublets/LinksOperatorBase.cs

```

1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  namespace Platform.Data.Doublets
4  {
5      public abstract class LinksOperatorBase<TLink>
6      {
7          public ILinks<TLink> Links { get; }
8          protected LinksOperatorBase(ILinks<TLink> links) => Links = links;
9      }
10 }

```

./Platform.Data.Doublets/Numbers/Unary/AddressToUnaryNumberConverter.cs

```

1  using System.Collections.Generic;
2  using Platform.Interfaces;
3  using Platform.Reflection;
4  using Platform.Numbers;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.Numbers.Unary
9  {
10     public class AddressToUnaryNumberConverter<TLink> : LinksOperatorBase<TLink>,
11         ⇨ IConverter<TLink>
12     {
13         private static readonly EqualityComparer<TLink> _equalityComparer =
14             ⇨ EqualityComparer<TLink>.Default;
15
16         private readonly IConverter<int, TLink> _powerOf2ToUnaryNumberConverter;
17
18         public AddressToUnaryNumberConverter(ILinks<TLink> links, IConverter<int, TLink>
19             ⇨ powerOf2ToUnaryNumberConverter) : base(links) => _powerOf2ToUnaryNumberConverter =
20             ⇨ powerOf2ToUnaryNumberConverter;
21
22         public TLink Convert(TLink sourceAddress)
23         {
24             var number = sourceAddress;
25             var nullConstant = Links.Constants.Null;
26             var one = Integer<TLink>.One;
27             var target = nullConstant;
28             for (int i = 0; !_equalityComparer.Equals(number, default) && i <
29                 ⇨ Type<TLink>.BitsLength; i++)
30             {
31                 if (_equalityComparer.Equals(Arithmetic.And(number, one), one))
32                 {
33                     target = _equalityComparer.Equals(target, nullConstant)
34                         ? _powerOf2ToUnaryNumberConverter.Convert(i)
35                         : Links.GetOrCreate(_powerOf2ToUnaryNumberConverter.Convert(i), target);
36                 }
37                 number = (Integer<TLink>)((ulong)(Integer<TLink>)number >> 1); // Should be
38                 ⇨ Bit.ShiftRight(number, 1)
39             }
40         }
41     }
42 }

```

```

33     }
34     return target;
35 }
36 }
37 }

```

./Platform.Data.Doublets/Numbers/Unary/LinkToItsFrequencyNumberConveter.cs

```

1  using System;
2  using System.Collections.Generic;
3  using Platform.Interfaces;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Numbers.Unary
8  {
9      public class LinkToItsFrequencyNumberConveter<TLink> : LinksOperatorBase<TLink>,
10         ↳ IConverter<Doublet<TLink>, TLink>
11      {
12          private static readonly EqualityComparer<TLink> _equalityComparer =
13             ↳ EqualityComparer<TLink>.Default;
14
15          private readonly IPropertyOperator<TLink, TLink> _frequencyPropertyOperator;
16          private readonly IConverter<TLink> _unaryNumberToAddressConverter;
17
18          public LinkToItsFrequencyNumberConveter(
19              ILinks<TLink> links,
20              IPropertyOperator<TLink, TLink> frequencyPropertyOperator,
21              IConverter<TLink> unaryNumberToAddressConverter)
22              : base(links)
23          {
24              _frequencyPropertyOperator = frequencyPropertyOperator;
25              _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
26          }
27
28          public TLink Convert(Doublet<TLink> doublet)
29          {
30              var link = Links.SearchOrDefault(doublet.Source, doublet.Target);
31              if (_equalityComparer.Equals(link, default))
32              {
33                  throw new ArgumentException($"Link ({doublet}) not found.", nameof(doublet));
34              }
35              var frequency = _frequencyPropertyOperator.Get(link);
36              if (_equalityComparer.Equals(frequency, default))
37              {
38                  return default;
39              }
40              var frequencyNumber = Links.GetSource(frequency);
41              return _unaryNumberToAddressConverter.Convert(frequencyNumber);
42          }
43      }
44  }

```

./Platform.Data.Doublets/Numbers/Unary/PowerOf2ToUnaryNumberConverter.cs

```

1  using System.Collections.Generic;
2  using Platform.Exceptions;
3  using Platform.Interfaces;
4  using Platform.Ranges;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.Numbers.Unary
9  {
10     public class PowerOf2ToUnaryNumberConverter<TLink> : LinksOperatorBase<TLink>,
11        ↳ IConverter<int, TLink>
12     {
13         private static readonly EqualityComparer<TLink> _equalityComparer =
14            ↳ EqualityComparer<TLink>.Default;
15
16         private readonly TLink[] _unaryNumberPowersOf2;
17
18         public PowerOf2ToUnaryNumberConverter(ILinks<TLink> links, TLink one) : base(links)
19         {
20             _unaryNumberPowersOf2 = new TLink[64];
21             _unaryNumberPowersOf2[0] = one;
22         }
23
24         public TLink Convert(int power)
25         {
26             Ensure.Always.ArgumentInRange(power, new Range<int>(0, _unaryNumberPowersOf2.Length
27                 ↳ - 1), nameof(power));
28         }
29     }
30 }

```

```

25         if (!_equalityComparer.Equals(_unaryNumberPowersOf2[power], default))
26         {
27             return _unaryNumberPowersOf2[power];
28         }
29         var previousPowerOf2 = Convert(power - 1);
30         var powerOf2 = Links.GetOrCreate(previousPowerOf2, previousPowerOf2);
31         _unaryNumberPowersOf2[power] = powerOf2;
32         return powerOf2;
33     }
34 }
35 }

```

./Platform.Data.Doublets/Numbers/Unary/UnaryNumberToAddressAddOperationConverter.cs

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using Platform.Interfaces;
4  using Platform.Numbers;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.Numbers.Unary
9  {
10     public class UnaryNumberToAddressAddOperationConverter<TLink> : LinksOperatorBase<TLink>,
11         ⇨ IConverter<TLink>
12     {
13         private static readonly EqualityComparer<TLink> _equalityComparer =
14             ⇨ EqualityComparer<TLink>.Default;
15
16         private Dictionary<TLink, TLink> _unaryToUInt64;
17         private readonly TLink _unaryOne;
18
19         public UnaryNumberToAddressAddOperationConverter(ILinks<TLink> links, TLink unaryOne)
20             : base(links)
21         {
22             _unaryOne = unaryOne;
23             InitUnaryToUInt64();
24         }
25
26         private void InitUnaryToUInt64()
27         {
28             var one = Integer<TLink>.One;
29             _unaryToUInt64 = new Dictionary<TLink, TLink>
30             {
31                 { _unaryOne, one }
32             };
33             var unary = _unaryOne;
34             var number = one;
35             for (var i = 1; i < 64; i++)
36             {
37                 unary = Links.GetOrCreate(unary, unary);
38                 number = Double(number);
39                 _unaryToUInt64.Add(unary, number);
40             }
41         }
42
43         public TLink Convert(TLink unaryNumber)
44         {
45             if (_equalityComparer.Equals(unaryNumber, default))
46             {
47                 return default;
48             }
49             if (_equalityComparer.Equals(unaryNumber, _unaryOne))
50             {
51                 return Integer<TLink>.One;
52             }
53             var source = Links.GetSource(unaryNumber);
54             var target = Links.GetTarget(unaryNumber);
55             if (_equalityComparer.Equals(source, target))
56             {
57                 return _unaryToUInt64[unaryNumber];
58             }
59             else
60             {
61                 var result = _unaryToUInt64[source];
62                 TLink lastValue;
63                 while (!_unaryToUInt64.TryGetValue(target, out lastValue))
64                 {
65                     source = Links.GetSource(target);
66                     result = Arithmetic<TLink>.Add(result, _unaryToUInt64[source]);
67                     target = Links.GetTarget(target);
68                 }
69                 return result;
70             }
71         }
72     }
73 }

```

```

66         }
67         result = Arithmetic<TLink>.Add(result, lastValue);
68         return result;
69     }
70 }
71
72 [MethodImpl(MethodImplOptions.AggressiveInlining)]
73 private static TLink Double(TLink number) => (Integer<TLink>)((Integer<TLink>)number *
    ↪ 2UL);
74 }
75 }

```

./Platform.Data.Doublets/Numbers/Unary/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  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Data.Doublets.Numbers.Unary
10 {
11     public class UnaryNumberToAddressOrOperationConverter<TLink> : LinksOperatorBase<TLink>,
    ↪ IConverter<TLink>
12     {
13         private static readonly EqualityComparer<TLink> _equalityComparer =
    ↪ EqualityComparer<TLink>.Default;
14
15         private readonly IDictionary<TLink, int> _unaryNumberPowerOf2Indicies;
16
17         public UnaryNumberToAddressOrOperationConverter(ILinks<TLink> links, IConverter<int,
    ↪ TLink> powerOf2ToUnaryNumberConverter)
    : base(links)
18         {
19             _unaryNumberPowerOf2Indicies = new Dictionary<TLink, int>();
20             for (int i = 0; i < Type<TLink>.BitsLength; i++)
21             {
22                 _unaryNumberPowerOf2Indicies.Add(powerOf2ToUnaryNumberConverter.Convert(i), i);
23             }
24         }
25
26         public TLink Convert(TLink sourceNumber)
27         {
28             var nullConstant = Links.Constants.Null;
29             var source = sourceNumber;
30             var target = nullConstant;
31             if (!_equalityComparer.Equals(source, nullConstant))
32             {
33                 while (true)
34                 {
35                     if (_unaryNumberPowerOf2Indicies.TryGetValue(source, out int powerOf2Index))
36                     {
37                         SetBit(ref target, powerOf2Index);
38                         break;
39                     }
40                     else
41                     {
42                         powerOf2Index = _unaryNumberPowerOf2Indicies[Links.GetSource(source)];
43                         SetBit(ref target, powerOf2Index);
44                         source = Links.GetTarget(source);
45                     }
46                 }
47             }
48             return target;
49         }
50     }
51
52     [MethodImpl(MethodImplOptions.AggressiveInlining)]
53     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))
54 }
55 }

```

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

```

1  using System.Linq;
2  using System.Collections.Generic;
3  using Platform.Interfaces;
4

```

```

5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Data.Doublets.PropertyOperators
8 {
9     public class PropertiesOperator<TLink> : LinksOperatorBase<TLink>,
10     ↪ IPropertiesOperator<TLink, TLink, TLink>
11     {
12         private static readonly EqualityComparer<TLink> _equalityComparer =
13         ↪ EqualityComparer<TLink>.Default;
14
15         public PropertiesOperator(ILinks<TLink> links) : base(links) { }
16
17         public TLink GetValue(TLink @object, TLink property)
18         {
19             var objectProperty = Links.SearchOrDefault(@object, property);
20             if (_equalityComparer.Equals(objectProperty, default))
21             {
22                 return default;
23             }
24             var valueLink = Links.All(Links.Constants.Any, objectProperty).SingleOrDefault();
25             if (valueLink == null)
26             {
27                 return default;
28             }
29             return Links.GetTarget(valueLink[Links.Constants.IndexPart]);
30         }
31
32         public void SetValue(TLink @object, TLink property, TLink value)
33         {
34             var objectProperty = Links.GetOrCreate(@object, property);
35             Links.DeleteMany(Links.AllIndices(Links.Constants.Any, objectProperty));
36             Links.GetOrCreate(objectProperty, value);
37         }
38     }
39 }

```

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

```

1 using System.Collections.Generic;
2 using Platform.Interfaces;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Doublets.PropertyOperators
7 {
8     public class PropertyOperator<TLink> : LinksOperatorBase<TLink>, IPropertyOperator<TLink,
9     ↪ TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
12         ↪ EqualityComparer<TLink>.Default;
13
14         private readonly TLink _propertyMarker;
15         private readonly TLink _propertyValueMarker;
16
17         public PropertyOperator(ILinks<TLink> links, TLink propertyMarker, TLink
18         ↪ propertyValueMarker) : base(links)
19         {
20             _propertyMarker = propertyMarker;
21             _propertyValueMarker = propertyValueMarker;
22         }
23
24         public TLink Get(TLink link)
25         {
26             var property = Links.SearchOrDefault(link, _propertyMarker);
27             var container = GetContainer(property);
28             var value = GetValue(container);
29             return value;
30         }
31
32         private TLink GetContainer(TLink property)
33         {
34             var valueContainer = default(TLink);
35             if (_equalityComparer.Equals(property, default))
36             {
37                 return valueContainer;
38             }
39             var constants = Links.Constants;
40             var continueConstant = constants.Continue;
41             var breakConstant = constants.Break;
42             var anyConstant = constants.Any;
43             var query = new Link<TLink>(anyConstant, property, anyConstant);

```

```

41     Links.ForEach(candidate =>
42     {
43         var candidateTarget = Links.GetTarget(candidate);
44         var valueTarget = Links.GetTarget(candidateTarget);
45         if (_equalityComparer.Equals(valueTarget, _propertyValueMarker))
46         {
47             valueContainer = Links.GetIndex(candidate);
48             return breakConstant;
49         }
50         return countinueConstant;
51     }, query);
52     return valueContainer;
53 }
54
55 private TLink GetValue(TLink container) => _equalityComparer.Equals(container, default)
56     ↪ ? default : Links.GetTarget(container);
57
58 public void Set(TLink link, TLink value)
59 {
60     var property = Links.GetOrCreate(link, _propertyMarker);
61     var container = GetContainer(property);
62     if (_equalityComparer.Equals(container, default))
63     {
64         Links.GetOrCreate(property, value);
65     }
66     else
67     {
68         Links.Update(container, property, value);
69     }
70 }
71 }

```

./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 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
19
20 // ReSharper disable StaticMemberInGenericType
21 // ReSharper disable BuiltInTypeReferenceStyle
22 // ReSharper disable MemberCanBePrivate.Local
23 // ReSharper disable UnusedMember.Local
24
25 namespace Platform.Data.Doublets.ResizableDirectMemory
26 {
27     public partial class ResizableDirectMemoryLinks<TLink> : DisposableBase, ILinks<TLink>
28     {
29         private static readonly EqualityComparer<TLink> _equalityComparer =
30             ↪ EqualityComparer<TLink>.Default;
31         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
32
33         /// <summary>Возвращает размер одной связи в байтах.</summary>
34         public static readonly int LinkSizeInBytes = Structure<Link>.Size;
35
36         public static readonly int LinkHeaderSizeInBytes = Structure<LinksHeader>.Size;
37
38         public static readonly long DefaultLinksSizeStep = LinkSizeInBytes * 1024 * 1024;
39
40         private struct Link
41         {
42             public static readonly int SourceOffset = Marshal.OffsetOf(typeof(Link),
43                 ↪ nameof(Source)).ToInt32();
44             public static readonly int TargetOffset = Marshal.OffsetOf(typeof(Link),
45                 ↪ nameof(Target)).ToInt32();
46             public static readonly int LeftAsSourceOffset = Marshal.OffsetOf(typeof(Link),
47                 ↪ nameof(LeftAsSource)).ToInt32();

```



```

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

```

```

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

```

```

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

```

```

224     var value = restrictions[1];
225     if (_equalityComparer.Equals(index, Constants.Any))
226     {
227         if (_equalityComparer.Equals(value, Constants.Any))
228         {
229             return Total; // Any - как отсутствие ограничения
230         }
231         return Add(_sourcesTreeMethods.CountUsages(value),
232             ↪ _targetsTreeMethods.CountUsages(value));
233     }
234     else
235     {
236         if (!Exists(index))
237         {
238             return Integer<TLink>.Zero;
239         }
240         if (_equalityComparer.Equals(value, Constants.Any))
241         {
242             return Integer<TLink>.One;
243         }
244         var storedLinkValue = GetLinkUnsafe(index);
245         if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), value) ||
246             ↪ _equalityComparer.Equals(Link.GetTarget(storedLinkValue), value))
247         {
248             return Integer<TLink>.One;
249         }
250         return Integer<TLink>.Zero;
251     }
252     }
253     if (restrictions.Count == 3)
254     {
255         var index = restrictions[Constants.IndexPart];
256         var source = restrictions[Constants.SourcePart];
257         var target = restrictions[Constants.TargetPart];
258
259         if (_equalityComparer.Equals(index, Constants.Any))
260         {
261             if (_equalityComparer.Equals(source, Constants.Any) &&
262                 ↪ _equalityComparer.Equals(target, Constants.Any))
263             {
264                 return Total;
265             }
266             else if (_equalityComparer.Equals(source, Constants.Any))
267             {
268                 return _targetsTreeMethods.CountUsages(target);
269             }
270             else if (_equalityComparer.Equals(target, Constants.Any))
271             {
272                 return _sourcesTreeMethods.CountUsages(source);
273             }
274             else //if(source != Any && target != Any)
275             {
276                 // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
277                 var link = _sourcesTreeMethods.Search(source, target);
278                 return _equalityComparer.Equals(link, Constants.Null) ?
279                     ↪ Integer<TLink>.Zero : Integer<TLink>.One;
280             }
281         }
282         else
283         {
284             if (!Exists(index))
285             {
286                 return Integer<TLink>.Zero;
287             }
288             if (_equalityComparer.Equals(source, Constants.Any) &&
289                 ↪ _equalityComparer.Equals(target, Constants.Any))
290             {
291                 return Integer<TLink>.One;
292             }
293             var storedLinkValue = GetLinkUnsafe(index);
294             if (!_equalityComparer.Equals(source, Constants.Any) &&
295                 ↪ !_equalityComparer.Equals(target, Constants.Any))
296             {
297                 if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), source) &&
298                     ↪ _equalityComparer.Equals(Link.GetTarget(storedLinkValue), target))
299                 {
300                     return Integer<TLink>.One;
301                 }
302             }
303         }
304     }

```

```

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

```

```

370         {
371             return handler(GetLinkStruct(index));
372         }
373         var storedLinkValue = GetLinkUnsafe(index);
374         if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), value) ||
375             _equalityComparer.Equals(Link.GetTarget(storedLinkValue), value))
376         {
377             return handler(GetLinkStruct(index));
378         }
379         return Constants.Continue;
380     }
381 }
382 if (restrictions.Count == 3)
383 {
384     var index = restrictions[Constants.IndexPart];
385     var source = restrictions[Constants.SourcePart];
386     var target = restrictions[Constants.TargetPart];
387     if (_equalityComparer.Equals(index, Constants.Any))
388     {
389         if (_equalityComparer.Equals(source, Constants.Any) &&
390             ↪ _equalityComparer.Equals(target, Constants.Any))
391         {
392             return Each(handler, ArrayPool<TLink>.Empty);
393         }
394         else if (_equalityComparer.Equals(source, Constants.Any))
395         {
396             return _targetsTreeMethods.EachUsage(target, handler);
397         }
398         else if (_equalityComparer.Equals(target, Constants.Any))
399         {
400             return _sourcesTreeMethods.EachUsage(source, handler);
401         }
402         else //if(source != Any && target != Any)
403         {
404             var link = _sourcesTreeMethods.Search(source, target);
405             return _equalityComparer.Equals(link, Constants.Null) ?
406                 ↪ Constants.Continue : handler(GetLinkStruct(link));
407         }
408     }
409     else
410     {
411         if (!Exists(index))
412         {
413             return Constants.Continue;
414         }
415         if (_equalityComparer.Equals(source, Constants.Any) &&
416             ↪ _equalityComparer.Equals(target, Constants.Any))
417         {
418             return handler(GetLinkStruct(index));
419         }
420         var storedLinkValue = GetLinkUnsafe(index);
421         if (!_equalityComparer.Equals(source, Constants.Any) &&
422             ↪ !_equalityComparer.Equals(target, Constants.Any))
423         {
424             if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), source) &&
425                 _equalityComparer.Equals(Link.GetTarget(storedLinkValue), target))
426             {
427                 return handler(GetLinkStruct(index));
428             }
429             return Constants.Continue;
430         }
431         var value = default(TLink);
432         if (_equalityComparer.Equals(source, Constants.Any))
433         {
434             value = target;
435         }
436         if (_equalityComparer.Equals(target, Constants.Any))
437         {
438             value = source;
439         }
440         if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), value) ||
441             _equalityComparer.Equals(Link.GetTarget(storedLinkValue), value))
442         {
443             return handler(GetLinkStruct(index));
444         }
445         return Constants.Continue;
446     }
447 }

```

```

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

```

```

508     }
509     LinksHeader.SetAllocatedLinks(_header,
510         ↳ Increment(LinksHeader.GetAllocatedLinks(_header)));
511     _memory.UsedCapacity += LinkSizeInBytes;
512     freeLink = LinksHeader.GetAllocatedLinks(_header);
513 }
514 return freeLink;
515 }
516 public void Delete(TLink link)
517 {
518     if (_comparer.Compare(link, LinksHeader.GetAllocatedLinks(_header)) < 0)
519     {
520         _unusedLinksListMethods.AttachAsFirst(link);
521     }
522     else if (_equalityComparer.Equals(link, LinksHeader.GetAllocatedLinks(_header)))
523     {
524         LinksHeader.SetAllocatedLinks(_header,
525             ↳ Decrement(LinksHeader.GetAllocatedLinks(_header)));
526         _memory.UsedCapacity -= LinkSizeInBytes;
527         // Убираем все связи, находящиеся в списке свободных в конце файла, до тех пор,
528         // пока не дойдём до первой существующей связи
529         // Позволяет оптимизировать количество выделенных связей (AllocatedLinks)
530         while ((_comparer.Compare(LinksHeader.GetAllocatedLinks(_header),
531             ↳ Integer<TLink>.Zero) > 0) &&
532             ↳ IsUnusedLink(LinksHeader.GetAllocatedLinks(_header)))
533         {
534             _unusedLinksListMethods.Detach(LinksHeader.GetAllocatedLinks(_header));
535             LinksHeader.SetAllocatedLinks(_header,
536                 ↳ Decrement(LinksHeader.GetAllocatedLinks(_header)));
537             _memory.UsedCapacity -= LinkSizeInBytes;
538         }
539     }
540 }
541
542 /// <remarks>
543 /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том случае, если
544 /// ↳ адрес реально поменялся
545 ///
546 /// Указатель this.links может быть в том же месте,
547 /// так как 0-я связь не используется и имеет такой же размер как Header,
548 /// поэтому header размещается в том же месте, что и 0-я связь
549 /// </remarks>
550 private void SetPointers(IDirectMemory memory)
551 {
552     if (memory == null)
553     {
554         _links = IntPtr.Zero;
555         _header = _links;
556         _unusedLinksListMethods = null;
557         _targetsTreeMethods = null;
558         _unusedLinksListMethods = null;
559     }
560     else
561     {
562         _links = memory.Pointer;
563         _header = _links;
564         _sourcesTreeMethods = new LinksSourcesTreeMethods(this);
565         _targetsTreeMethods = new LinksTargetsTreeMethods(this);
566         _unusedLinksListMethods = new UnusedLinksListMethods(_links, _header);
567     }
568 }
569
570 [MethodImpl(MethodImplOptions.AggressiveInlining)]
571 private bool Exists(TLink link)
572 => (_comparer.Compare(link, Constants.MinPossibleIndex) >= 0)
573     && (_comparer.Compare(link, LinksHeader.GetAllocatedLinks(_header)) <= 0)
574     && !IsUnusedLink(link);
575
576 [MethodImpl(MethodImplOptions.AggressiveInlining)]
577 private bool IsUnusedLink(TLink link)
578 => _equalityComparer.Equals(LinksHeader.GetFirstFreeLink(_header), link)
579     || (_equalityComparer.Equals(Link.GetSizeAsSource(GetLinkUnsafe(link)),
580         ↳ Constants.Null)
581         && !_equalityComparer.Equals(Link.GetSource(GetLinkUnsafe(link)), Constants.Null));
582
583 #region DisposableBase
584
585 protected override bool AllowMultipleDisposeCalls => true;

```



```

579
580     protected override void Dispose(bool manual, bool wasDisposed)
581     {
582         if (!wasDisposed)
583         {
584             SetPointers(null);
585             _memory.DisposeIfPossible();
586         }
587     }
588
589     #endregion
590 }
591 }

```

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

```

1  using System;
2  using Platform.Unsafe;
3  using Platform.Collections.Methods.Lists;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.ResizableDirectMemory
8  {
9      partial class ResizableDirectMemoryLinks<TLink>
10     {
11         private class UnusedLinksListMethods : CircularDoublyLinkedListMethods<TLink>
12         {
13             private readonly IntPtr _links;
14             private readonly IntPtr _header;
15
16             public UnusedLinksListMethods(IntPtr links, IntPtr header)
17             {
18                 _links = links;
19                 _header = header;
20             }
21
22             protected override TLink GetFirst() => (_header +
23                 ↳ LinksHeader.FirstFreeLinkOffset).GetValue<TLink>();
24
25             protected override TLink GetLast() => (_header +
26                 ↳ LinksHeader.LastFreeLinkOffset).GetValue<TLink>();
27
28             protected override TLink GetPrevious(TLink element) =>
29                 ↳ (_links.GetElement(LinkSizeInBytes, element) +
30                 ↳ Link.SourceOffset).GetValue<TLink>();
31
32             protected override TLink GetNext(TLink element) =>
33                 ↳ (_links.GetElement(LinkSizeInBytes, element) +
34                 ↳ Link.TargetOffset).GetValue<TLink>();
35
36             protected override TLink GetSize() => (_header +
37                 ↳ LinksHeader.FreeLinksOffset).GetValue<TLink>();
38
39             protected override void SetFirst(TLink element) => (_header +
40                 ↳ LinksHeader.FirstFreeLinkOffset).SetValue(element);
41
42             protected override void SetLast(TLink element) => (_header +
43                 ↳ LinksHeader.LastFreeLinkOffset).SetValue(element);
44
45             protected override void SetPrevious(TLink element, TLink previous) =>
46                 ↳ (_links.GetElement(LinkSizeInBytes, element) +
47                 ↳ Link.SourceOffset).SetValue(previous);
48
49             protected override void SetNext(TLink element, TLink next) =>
50                 ↳ (_links.GetElement(LinkSizeInBytes, element) + Link.TargetOffset).SetValue(next);
51
52             protected override void SetSize(TLink size) => (_header +
53                 ↳ LinksHeader.FreeLinksOffset).SetValue(size);
54         }
55     }
56 }

```

./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 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
11
12 namespace Platform.Data.Doublets.ResizableDirectMemory
13 {
14     partial class ResizableDirectMemoryLinks<TLink>
15     {
16         private abstract class LinksTreeMethodsBase :
17             ↳ SizedAndThreadedAVLBalancedTreeMethods<TLink>
18         {
19             private readonly ResizableDirectMemoryLinks<TLink> _memory;
20             private readonly LinksCombinedConstants<TLink, TLink, int> _constants;
21             protected readonly IntPtr Links;
22             protected readonly IntPtr Header;
23
24             protected LinksTreeMethodsBase(ResizableDirectMemoryLinks<TLink> memory)
25             {
26                 Links = memory._links;
27                 Header = memory._header;
28                 _memory = memory;
29                 _constants = memory.Constants;
30             }
31
32             [MethodImpl(MethodImplOptions.AggressiveInlining)]
33             protected abstract TLink GetTreeRoot();
34
35             [MethodImpl(MethodImplOptions.AggressiveInlining)]
36             protected abstract TLink GetBasePartValue(TLink link);
37
38             public TLink this[TLink index]
39             {
40                 get
41                 {
42                     var root = GetTreeRoot();
43                     if (GreaterOrEqualThan(index, GetSize(root)))
44                     {
45                         return GetZero();
46                     }
47                     while (!EqualToZero(root))
48                     {
49                         var left = GetLeftOrDefault(root);
50                         var leftSize = GetSizeOrZero(left);
51                         if (LessThan(index, leftSize))
52                         {
53                             root = left;
54                             continue;
55                         }
56                         if (IsEquals(index, leftSize))
57                         {
58                             return root;
59                         }
60                         root = GetRightOrDefault(root);
61                         index = Subtract(index, Increment(leftSize));
62                     }
63                     return GetZero(); // TODO: Impossible situation exception (only if tree
64                                     ↳ structure broken)
65                 }
66             }
67
68             // TODO: Return indices range instead of references count
69             public TLink CountUsages(TLink link)
70             {
71                 var root = GetTreeRoot();
72                 var total = GetSize(root);
73                 var totalRightIgnore = GetZero();
74                 while (!EqualToZero(root))
75                 {
76                     var @base = GetBasePartValue(root);
77                     if (LessOrEqualThan(@base, link))
78                     {
79                         root = GetRightOrDefault(root);
80                     }
81                     else
82                     {
83                         totalRightIgnore = Add(totalRightIgnore, Increment(GetRightSize(root)));
84                         root = GetLeftOrDefault(root);
85                     }
86                 }
87             }
88         }
89     }
90 }

```

```

85     root = GetTreeRoot();
86     var totalLeftIgnore = GetZero();
87     while (!EqualToZero(root))
88     {
89         var @base = GetBasePartValue(root);
90         if (GreaterOrEqualThan(@base, link))
91         {
92             root = GetLeftOrDefault(root);
93         }
94         else
95         {
96             totalLeftIgnore = Add(totalLeftIgnore, Increment(GetLeftSize(root)));
97             root = GetRightOrDefault(root);
98         }
99     }
100     return Subtract(Subtract(total, totalRightIgnore), totalLeftIgnore);
101 }
102
103 public TLink EachUsage(TLink link, Func<IList<TLink>, TLink> handler)
104 {
105     var root = GetTreeRoot();
106     if (EqualToZero(root))
107     {
108         return _constants.Continue;
109     }
110     TLink first = GetZero(), current = root;
111     while (!EqualToZero(current))
112     {
113         var @base = GetBasePartValue(current);
114         if (GreaterOrEqualThan(@base, link))
115         {
116             if (IsEquals(@base, link))
117             {
118                 first = current;
119             }
120             current = GetLeftOrDefault(current);
121         }
122         else
123         {
124             current = GetRightOrDefault(current);
125         }
126     }
127     if (!EqualToZero(first))
128     {
129         current = first;
130         while (true)
131         {
132             if (IsEquals(handler(_memory.GetLinkStruct(current)), _constants.Break))
133             {
134                 return _constants.Break;
135             }
136             current = GetNext(current);
137             if (EqualToZero(current) || !IsEquals(GetBasePartValue(current), link))
138             {
139                 break;
140             }
141         }
142     }
143     return _constants.Continue;
144 }
145
146 protected override void PrintNodeValue(TLink node, StringBuilder sb)
147 {
148     sb.Append(' ');
149     sb.Append((Links.GetElement(LinkSizeInBytes, node) +
150         ↪ Link.SourceOffset).GetValue<TLink>());
151     sb.Append('-');
152     sb.Append('>');
153     sb.Append((Links.GetElement(LinkSizeInBytes, node) +
154         ↪ Link.TargetOffset).GetValue<TLink>());
155 }
156
157 private class LinksSourcesTreeMethods : LinksTreeMethodsBase
158 {
159     public LinksSourcesTreeMethods(ResizableDirectMemoryLinks<TLink> memory)
160         : base(memory)

```

```

161 {
162 }
163
164 protected override IntPtr GetLeftPointer(TLink node) =>
165     ↳ Links.GetElement(LinkSizeInBytes, node) + Link.LeftAsSourceOffset;
166
167 protected override IntPtr GetRightPointer(TLink node) =>
168     ↳ Links.GetElement(LinkSizeInBytes, node) + Link.RightAsSourceOffset;
169
170 protected override TLink GetLeftValue(TLink node) =>
171     ↳ (Links.GetElement(LinkSizeInBytes, node) +
172     ↳ Link.LeftAsSourceOffset).GetValue<TLink>();
173
174 protected override TLink GetRightValue(TLink node) =>
175     ↳ (Links.GetElement(LinkSizeInBytes, node) +
176     ↳ Link.RightAsSourceOffset).GetValue<TLink>();
177
178 protected override TLink GetSize(TLink node)
179 {
180     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
181     ↳ Link.SizeAsSourceOffset).GetValue<TLink>();
182     return Bit.PartialRead(previousValue, 5, -5);
183 }
184
185 protected override void SetLeft(TLink node, TLink left) =>
186     ↳ (Links.GetElement(LinkSizeInBytes, node) +
187     ↳ Link.LeftAsSourceOffset).SetValue(left);
188
189 protected override void SetRight(TLink node, TLink right) =>
190     ↳ (Links.GetElement(LinkSizeInBytes, node) +
191     ↳ Link.RightAsSourceOffset).SetValue(right);
192
193 protected override void SetSize(TLink node, TLink size)
194 {
195     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
196     ↳ Link.SizeAsSourceOffset).GetValue<TLink>();
197     (Links.GetElement(LinkSizeInBytes, node) +
198     ↳ Link.SizeAsSourceOffset).SetValue(Bit.PartialWrite(previousValue, size, 5,
199     ↳ -5));
200 }
201
202 protected override bool GetLeftIsChild(TLink node)
203 {
204     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
205     ↳ Link.SizeAsSourceOffset).GetValue<TLink>();
206     return (Integer<TLink>)Bit.PartialRead(previousValue, 4, 1);
207 }
208
209 protected override void SetLeftIsChild(TLink node, bool value)
210 {
211     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
212     ↳ Link.SizeAsSourceOffset).GetValue<TLink>();
213     var modified = Bit.PartialWrite(previousValue, (TLink)(Integer<TLink>)value, 4,
214     ↳ 1);
215     (Links.GetElement(LinkSizeInBytes, node) +
216     ↳ Link.SizeAsSourceOffset).SetValue(modified);
217 }
218
219 protected override bool GetRightIsChild(TLink node)
220 {
221     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
222     ↳ Link.SizeAsSourceOffset).GetValue<TLink>();
223     return (Integer<TLink>)Bit.PartialRead(previousValue, 3, 1);
224 }
225
226 protected override void SetRightIsChild(TLink node, bool value)
227 {
228     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
229     ↳ Link.SizeAsSourceOffset).GetValue<TLink>();
230     var modified = Bit.PartialWrite(previousValue, (TLink)(Integer<TLink>)value, 3,
231     ↳ 1);
232     (Links.GetElement(LinkSizeInBytes, node) +
233     ↳ Link.SizeAsSourceOffset).SetValue(modified);
234 }
235
236 protected override sbyte GetBalance(TLink node)
237 {

```

```

216     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
217         ↪ Link.SizeAsSourceOffset).GetValue<TLink>();
218     var value = (ulong)(Integer<TLink>)Bit.PartialRead(previousValue, 0, 3);
219     var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) << 5) | value & 3 |
220         ↪ 124 : value & 3);
221     return unpackedValue;
222 }
223
224 protected override void SetBalance(TLink node, sbyte value)
225 {
226     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
227         ↪ Link.SizeAsSourceOffset).GetValue<TLink>();
228     var packagedValue = (TLink)(Integer<TLink>)((((byte)value >> 5) & 4) | value &
229         ↪ 3);
230     var modified = Bit.PartialWrite(previousValue, packagedValue, 0, 3);
231     (Links.GetElement(LinkSizeInBytes, node) +
232         ↪ Link.SizeAsSourceOffset).SetValue(modified);
233 }
234
235 protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
236 {
237     var firstSource = (Links.GetElement(LinkSizeInBytes, first) +
238         ↪ Link.SourceOffset).GetValue<TLink>();
239     var secondSource = (Links.GetElement(LinkSizeInBytes, second) +
240         ↪ Link.SourceOffset).GetValue<TLink>();
241     return LessThan(firstSource, secondSource) ||
242         (IsEquals(firstSource, secondSource) &&
243         ↪ LessThan((Links.GetElement(LinkSizeInBytes, first) +
244             ↪ Link.TargetOffset).GetValue<TLink>(),
245             ↪ (Links.GetElement(LinkSizeInBytes, second) +
246             ↪ Link.TargetOffset).GetValue<TLink>()));
247 }
248
249 protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
250 {
251     var firstSource = (Links.GetElement(LinkSizeInBytes, first) +
252         ↪ Link.SourceOffset).GetValue<TLink>();
253     var secondSource = (Links.GetElement(LinkSizeInBytes, second) +
254         ↪ Link.SourceOffset).GetValue<TLink>();
255     return GreaterThan(firstSource, secondSource) ||
256         (IsEquals(firstSource, secondSource) &&
257         ↪ GreaterThan((Links.GetElement(LinkSizeInBytes, first) +
258             ↪ Link.TargetOffset).GetValue<TLink>(),
259             ↪ (Links.GetElement(LinkSizeInBytes, second) +
260             ↪ Link.TargetOffset).GetValue<TLink>()));
261 }
262
263 protected override TLink GetTreeRoot() => (Header +
264     ↪ LinksHeader.FirstAsSourceOffset).GetValue<TLink>();
265
266 protected override TLink GetBasePartValue(TLink link) =>
267     ↪ (Links.GetElement(LinkSizeInBytes, link) + Link.SourceOffset).GetValue<TLink>();
268
269 /// <summary>
270 /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
271     ↪ (концом)
272 /// по дереву (индексу) связей, отсортированному по Source, а затем по Target.
273 /// </summary>
274 /// <param name="source">Индекс связи, которая является началом на искомой
275     ↪ связи.</param>
276 /// <param name="target">Индекс связи, которая является концом на искомой
277     ↪ связи.</param>
278 /// <returns>Индекс искомой связи.</returns>
279 public TLink Search(TLink source, TLink target)
280 {
281     var root = GetTreeRoot();
282     while (!EqualToZero(root))
283     {
284         var rootSource = (Links.GetElement(LinkSizeInBytes, root) +
285             ↪ Link.SourceOffset).GetValue<TLink>();
286         var rootTarget = (Links.GetElement(LinkSizeInBytes, root) +
287             ↪ Link.TargetOffset).GetValue<TLink>();
288         if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
289             ↪ node.Key < root.Key
290         {
291             root = GetLeftOrDefault(root);
292         }
293     }
294 }

```

```

267     }
268     else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget))
269         ↪ // node.Key > root.Key
270     {
271         root = GetRightOrDefault(root);
272     }
273     else // node.Key == root.Key
274     {
275         return root;
276     }
277     return GetZero();
278 }
279
280 [MethodImpl(MethodImplOptions.AggressiveInlining)]
281 private bool FirstIsToLeftOfSecond(TLink firstSource, TLink firstTarget, TLink
282     ↪ secondSource, TLink secondTarget) => LessThan(firstSource, secondSource) ||
283     ↪ (IsEquals(firstSource, secondSource) && LessThan(firstTarget, secondTarget));
284
285 [MethodImpl(MethodImplOptions.AggressiveInlining)]
286 private bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget, TLink
287     ↪ secondSource, TLink secondTarget) => GreaterThan(firstSource, secondSource) ||
288     ↪ (IsEquals(firstSource, secondSource) && GreaterThan(firstTarget, secondTarget));
289 }
290
291 private class LinksTargetsTreeMethods : LinksTreeMethodsBase
292 {
293     public LinksTargetsTreeMethods(ResizableDirectMemoryLinks<TLink> memory)
294         : base(memory)
295     {
296     }
297
298     protected override IntPtr GetLeftPointer(TLink node) =>
299         ↪ Links.GetElement(LinkSizeInBytes, node) + Link.LeftAsTargetOffset;
300
301     protected override IntPtr GetRightPointer(TLink node) =>
302         ↪ Links.GetElement(LinkSizeInBytes, node) + Link.RightAsTargetOffset;
303
304     protected override TLink GetLeftValue(TLink node) =>
305         ↪ (Links.GetElement(LinkSizeInBytes, node) +
306         ↪ Link.LeftAsTargetOffset).GetValue<TLink>();
307
308     protected override TLink GetRightValue(TLink node) =>
309         ↪ (Links.GetElement(LinkSizeInBytes, node) +
310         ↪ Link.RightAsTargetOffset).GetValue<TLink>();
311
312     protected override TLink GetSize(TLink node)
313     {
314         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
315         ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
316         return Bit.PartialRead(previousValue, 5, -5);
317     }
318
319     protected override void SetLeft(TLink node, TLink left) =>
320         ↪ (Links.GetElement(LinkSizeInBytes, node) +
321         ↪ Link.LeftAsTargetOffset).SetValue(left);
322
323     protected override void SetRight(TLink node, TLink right) =>
324         ↪ (Links.GetElement(LinkSizeInBytes, node) +
325         ↪ Link.RightAsTargetOffset).SetValue(right);
326
327     protected override void SetSize(TLink node, TLink size)
328     {
329         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
330         ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
331         (Links.GetElement(LinkSizeInBytes, node) +
332         ↪ Link.SizeAsTargetOffset).SetValue(Bit.PartialWrite(previousValue, size, 5,
333         ↪ -5));
334     }
335
336     protected override bool GetLeftIsChild(TLink node)
337     {
338         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
339         ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
340         return (Integer<TLink>)Bit.PartialRead(previousValue, 4, 1);
341     }
342 }

```

```

324 protected override void SetLeftIsChild(TLink node, bool value)
325 {
326     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
        ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
327     var modified = Bit.PartialWrite(previousValue, (TLink)(Integer<TLink>)value, 4,
        ↪ 1);
328     (Links.GetElement(LinkSizeInBytes, node) +
        ↪ Link.SizeAsTargetOffset).SetValue(modified);
329 }
330
331 protected override bool GetRightIsChild(TLink node)
332 {
333     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
        ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
334     return (Integer<TLink>)Bit.PartialRead(previousValue, 3, 1);
335 }
336
337 protected override void SetRightIsChild(TLink node, bool value)
338 {
339     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
        ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
340     var modified = Bit.PartialWrite(previousValue, (TLink)(Integer<TLink>)value, 3,
        ↪ 1);
341     (Links.GetElement(LinkSizeInBytes, node) +
        ↪ Link.SizeAsTargetOffset).SetValue(modified);
342 }
343
344 protected override sbyte GetBalance(TLink node)
345 {
346     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
        ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
347     var value = (ulong)(Integer<TLink>)Bit.PartialRead(previousValue, 0, 3);
348     var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) << 5) | value & 3 |
        ↪ 124 : value & 3);
349     return unpackedValue;
350 }
351
352 protected override void SetBalance(TLink node, sbyte value)
353 {
354     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
        ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
355     var packagedValue = (TLink)(Integer<TLink>)(((byte)value >> 5) & 4) | value &
        ↪ 3);
356     var modified = Bit.PartialWrite(previousValue, packagedValue, 0, 3);
357     (Links.GetElement(LinkSizeInBytes, node) +
        ↪ Link.SizeAsTargetOffset).SetValue(modified);
358 }
359
360 protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
361 {
362     var firstTarget = (Links.GetElement(LinkSizeInBytes, first) +
        ↪ Link.TargetOffset).GetValue<TLink>();
363     var secondTarget = (Links.GetElement(LinkSizeInBytes, second) +
        ↪ Link.TargetOffset).GetValue<TLink>();
364     return LessThan(firstTarget, secondTarget) ||
365         (IsEquals(firstTarget, secondTarget) &&
        ↪ LessThan((Links.GetElement(LinkSizeInBytes, first) +
        ↪ Link.SourceOffset).GetValue<TLink>(),
        ↪ (Links.GetElement(LinkSizeInBytes, second) +
        ↪ Link.SourceOffset).GetValue<TLink>()));
366 }
367
368 protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
369 {
370     var firstTarget = (Links.GetElement(LinkSizeInBytes, first) +
        ↪ Link.TargetOffset).GetValue<TLink>();
371     var secondTarget = (Links.GetElement(LinkSizeInBytes, second) +
        ↪ Link.TargetOffset).GetValue<TLink>();
372     return GreaterThan(firstTarget, secondTarget) ||
373         (IsEquals(firstTarget, secondTarget) &&
        ↪ GreaterThan((Links.GetElement(LinkSizeInBytes, first) +
        ↪ Link.SourceOffset).GetValue<TLink>(),
        ↪ (Links.GetElement(LinkSizeInBytes, second) +
        ↪ Link.SourceOffset).GetValue<TLink>()));
374 }
375

```

```

376         protected override TLink GetTreeRoot() => (Header +
            ↳ LinksHeader.FirstAsTargetOffset).GetValue<TLink>();
377
378         protected override TLink GetBasePartValue(TLink link) =>
            ↳ (Links.GetElement(LinkSizeInBytes, link) + Link.TargetOffset).GetValue<TLink>();
379     }
380 }
381 }

```

./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  #pragma warning disable 0649
12  #pragma warning disable 169
13  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
14
15  // ReSharper disable BuiltInTypeReferenceStyle
16
17  //#define ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
18
19  namespace Platform.Data.Doublets.ResizableDirectMemory
20  {
21      using id = UInt64;
22
23      public unsafe partial class UInt64ResizableDirectMemoryLinks : DisposableBase, ILinks<id>
24      {
25          /// <summary>Возвращает размер одной связи в байтах.</summary>
26          /// <remarks>
27          /// Используется только во вне класса, не рекомендуется использовать внутри.
28          /// Так как во вне не обязательно будет доступен unsafe C#.
29          /// </remarks>
30          public static readonly int LinkSizeInBytes = sizeof(Link);
31
32          public static readonly long DefaultLinksSizeStep = LinkSizeInBytes * 1024 * 1024;
33
34          private struct Link
35          {
36              public id Source;
37              public id Target;
38              public id LeftAsSource;
39              public id RightAsSource;
40              public id SizeAsSource;
41              public id LeftAsTarget;
42              public id RightAsTarget;
43              public id SizeAsTarget;
44          }
45
46          private struct LinksHeader
47          {
48              public id AllocatedLinks;
49              public id ReservedLinks;
50              public id FreeLinks;
51              public id FirstFreeLink;
52              public id FirstAsSource;
53              public id FirstAsTarget;
54              public id LastFreeLink;
55              public id Reserved8;
56          }
57
58          private readonly long _memoryReservationStep;
59
60          private readonly IResizableDirectMemory _memory;
61          private LinksHeader* _header;
62          private Link* _links;
63
64          private LinksTargetsTreeMethods _targetsTreeMethods;
65          private LinksSourcesTreeMethods _sourcesTreeMethods;
66
67          // TODO: Возможно чтобы гарантированно проверять на то, является ли связь удалённой,
68          ↳ нужно использовать не список а дерево, так как так можно быстрее проверить на
69          ↳ наличие связи внутри
70          private UnusedLinksListMethods _unusedLinksListMethods;
71
72          /// <summary>

```



```

71     /// Возвращает общее число связей находящихся в хранилище.
72     /// </summary>
73     private id Total => _header->AllocatedLinks - _header->FreeLinks;
74
75     // TODO: Дать возможность переопределять в конструкторе
76     public LinksCombinedConstants<id, id, int> Constants { get; }
77
78     public UInt64ResizableDirectMemoryLinks(string address) : this(address,
79         ↪ DefaultLinksSizeStep) { }
80
81     /// <summary>
82     /// Создаёт экземпляр базы данных Links в файле по указанному адресу, с указанным
83     ↪ минимальным шагом расширения базы данных.
84     /// </summary>
85     /// <param name="address">Полный путь к файлу базы данных.</param>
86     /// <param name="memoryReservationStep">Минимальный шаг расширения базы данных в
87     ↪ байтах.</param>
88     public UInt64ResizableDirectMemoryLinks(string address, long memoryReservationStep) :
89     ↪ this(new FileMappedResizableDirectMemory(address, memoryReservationStep),
90     ↪ memoryReservationStep) { }
91
92     public UInt64ResizableDirectMemoryLinks(IResizableDirectMemory memory) : this(memory,
93     ↪ DefaultLinksSizeStep) { }
94
95     public UInt64ResizableDirectMemoryLinks(IResizableDirectMemory memory, long
96     ↪ memoryReservationStep)
97     {
98         Constants = Default<LinksCombinedConstants<id, id, int>>.Instance;
99         _memory = memory;
100         _memoryReservationStep = memoryReservationStep;
101         if (memory.ReservedCapacity < memoryReservationStep)
102         {
103             memory.ReservedCapacity = memoryReservationStep;
104         }
105         SetPointers(_memory);
106         // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
107         _memory.UsedCapacity = ((long)_header->AllocatedLinks * sizeof(Link)) +
108         ↪ sizeof(LinksHeader);
109         // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
110         _header->ReservedLinks = (id)((_memory.ReservedCapacity - sizeof(LinksHeader)) /
111         ↪ sizeof(Link));
112     }
113
114     [MethodImpl(MethodImplOptions.AggressiveInlining)]
115     public id Count(IList<id> restrictions)
116     {
117         // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
118         if (restrictions.Count == 0)
119         {
120             return Total;
121         }
122         if (restrictions.Count == 1)
123         {
124             var index = restrictions[Constants.IndexPart];
125             if (index == Constants.Any)
126             {
127                 return Total;
128             }
129             return Exists(index) ? 1UL : 0UL;
130         }
131         if (restrictions.Count == 2)
132         {
133             var index = restrictions[Constants.IndexPart];
134             var value = restrictions[1];
135             if (index == Constants.Any)
136             {
137                 if (value == Constants.Any)
138                 {
139                     return Total; // Any - как отсутствие ограничения
140                 }
141                 return _sourcesTreeMethods.CountUsages(value)
142                     + _targetsTreeMethods.CountUsages(value);
143             }
144             else
145             {
146                 if (!Exists(index))
147                 {
148                     return 0;
149                 }
150             }
151         }
152     }

```

```

140     }
141     if (value == Constants.Any)
142     {
143         return 1;
144     }
145     var storedLinkValue = GetLinkUnsafe(index);
146     if (storedLinkValue->Source == value ||
147         storedLinkValue->Target == value)
148     {
149         return 1;
150     }
151     return 0;
152 }
153 }
154 if (restrictions.Count == 3)
155 {
156     var index = restrictions[Constants.IndexPart];
157     var source = restrictions[Constants.SourcePart];
158     var target = restrictions[Constants.TargetPart];
159     if (index == Constants.Any)
160     {
161         if (source == Constants.Any && target == Constants.Any)
162         {
163             return Total;
164         }
165         else if (source == Constants.Any)
166         {
167             return _targetsTreeMethods.CountUsages(target);
168         }
169         else if (target == Constants.Any)
170         {
171             return _sourcesTreeMethods.CountUsages(source);
172         }
173         else //if(source != Any && target != Any)
174         {
175             // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
176             var link = _sourcesTreeMethods.Search(source, target);
177             return link == Constants.Null ? OUL : 1UL;
178         }
179     }
180     else
181     {
182         if (!Exists(index))
183         {
184             return 0;
185         }
186         if (source == Constants.Any && target == Constants.Any)
187         {
188             return 1;
189         }
190         var storedLinkValue = GetLinkUnsafe(index);
191         if (source != Constants.Any && target != Constants.Any)
192         {
193             if (storedLinkValue->Source == source &&
194                 storedLinkValue->Target == target)
195             {
196                 return 1;
197             }
198             return 0;
199         }
200         var value = default(id);
201         if (source == Constants.Any)
202         {
203             value = target;
204         }
205         if (target == Constants.Any)
206         {
207             value = source;
208         }
209         if (storedLinkValue->Source == value ||
210             storedLinkValue->Target == value)
211         {
212             return 1;
213         }
214         return 0;
215     }
216 }
217 throw new NotSupportedException("Другие размеры и способы ограничений не
↪ поддерживаются.");

```

```

218 }
219
220 [MethodImpl(MethodImplOptions.AggressiveInlining)]
221 public id Each(Func<IList<id>, id> handler, IList<id> restrictions)
222 {
223     if (restrictions.Count == 0)
224     {
225         for (id link = 1; link <= _header->AllocatedLinks; link++)
226         {
227             if (Exists(link))
228             {
229                 if (handler(GetLinkStruct(link)) == Constants.Break)
230                 {
231                     return Constants.Break;
232                 }
233             }
234         }
235         return Constants.Continue;
236     }
237     if (restrictions.Count == 1)
238     {
239         var index = restrictions[Constants.IndexPart];
240         if (index == Constants.Any)
241         {
242             return Each(handler, ArrayPool<ulong>.Empty);
243         }
244         if (!Exists(index))
245         {
246             return Constants.Continue;
247         }
248         return handler(GetLinkStruct(index));
249     }
250     if (restrictions.Count == 2)
251     {
252         var index = restrictions[Constants.IndexPart];
253         var value = restrictions[1];
254         if (index == Constants.Any)
255         {
256             if (value == Constants.Any)
257             {
258                 return Each(handler, ArrayPool<ulong>.Empty);
259             }
260             if (Each(handler, new[] { index, value, Constants.Any }) == Constants.Break)
261             {
262                 return Constants.Break;
263             }
264             return Each(handler, new[] { index, Constants.Any, value });
265         }
266         else
267         {
268             if (!Exists(index))
269             {
270                 return Constants.Continue;
271             }
272             if (value == Constants.Any)
273             {
274                 return handler(GetLinkStruct(index));
275             }
276             var storedLinkValue = GetLinkUnsafe(index);
277             if (storedLinkValue->Source == value ||
278                 storedLinkValue->Target == value)
279             {
280                 return handler(GetLinkStruct(index));
281             }
282             return Constants.Continue;
283         }
284     }
285     if (restrictions.Count == 3)
286     {
287         var index = restrictions[Constants.IndexPart];
288         var source = restrictions[Constants.SourcePart];
289         var target = restrictions[Constants.TargetPart];
290         if (index == Constants.Any)
291         {
292             if (source == Constants.Any && target == Constants.Any)
293             {
294                 return Each(handler, ArrayPool<ulong>.Empty);
295             }

```

```

296         else if (source == Constants.Any)
297         {
298             return _targetsTreeMethods.EachReference(target, handler);
299         }
300         else if (target == Constants.Any)
301         {
302             return _sourcesTreeMethods.EachReference(source, handler);
303         }
304         else //if(source != Any && target != Any)
305         {
306             var link = _sourcesTreeMethods.Search(source, target);
307             return link == Constants.Null ? Constants.Continue :
308                 ↪ handler(GetLinkStruct(link));
309         }
310     }
311     else
312     {
313         if (!Exists(index))
314         {
315             return Constants.Continue;
316         }
317         if (source == Constants.Any && target == Constants.Any)
318         {
319             return handler(GetLinkStruct(index));
320         }
321         var storedLinkValue = GetLinkUnsafe(index);
322         if (source != Constants.Any && target != Constants.Any)
323         {
324             if (storedLinkValue->Source == source &&
325                 storedLinkValue->Target == target)
326             {
327                 return handler(GetLinkStruct(index));
328             }
329             return Constants.Continue;
330         }
331         var value = default(id);
332         if (source == Constants.Any)
333         {
334             value = target;
335         }
336         if (target == Constants.Any)
337         {
338             value = source;
339         }
340         if (storedLinkValue->Source == value ||
341             storedLinkValue->Target == value)
342         {
343             return handler(GetLinkStruct(index));
344         }
345         return Constants.Continue;
346     }
347 }
348 throw new NotSupportedException("Другие размеры и способы ограничений не
349 ↪ поддерживаются.");
350 }
351
352 /// <remarks>
353 /// TODO: Возможно можно перемещать значения, если указан индекс, но значение существует
354 ↪ в другом месте (но не в менеджере памяти, а в логике Links)
355 /// </remarks>
356 [MethodImpl(MethodImplOptions.AggressiveInlining)]
357 public id Update(IList<id> values)
358 {
359     var linkIndex = values[Constants.IndexPart];
360     var link = GetLinkUnsafe(linkIndex);
361     // Будет корректно работать только в том случае, если пространство выделенной связи
362     ↪ предварительно заполнено нулями
363     if (link->Source != Constants.Null)
364     {
365         _sourcesTreeMethods.Detach(new IntPtr(&_header->FirstAsSource), linkIndex);
366     }
367     if (link->Target != Constants.Null)
368     {
369         _targetsTreeMethods.Detach(new IntPtr(&_header->FirstAsTarget), linkIndex);
370     }
371 }
372 #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
373     var leftTreeSize = _sourcesTreeMethods.GetSize(new IntPtr(&_header->FirstAsSource));
374     var rightTreeSize = _targetsTreeMethods.GetSize(new IntPtr(&_header->FirstAsTarget));
375 #endif

```

```

370         if (leftTreeSize != rightTreeSize)
371         {
372             throw new Exception("One of the trees is broken.");
373         }
374     #endif
375     link->Source = values[Constants.SourcePart];
376     link->Target = values[Constants.TargetPart];
377     if (link->Source != Constants.Null)
378     {
379         _sourcesTreeMethods.Attach(new IntPtr(&_header->FirstAsSource), linkIndex);
380     }
381     if (link->Target != Constants.Null)
382     {
383         _targetsTreeMethods.Attach(new IntPtr(&_header->FirstAsTarget), linkIndex);
384     }
385     #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
386     leftTreeSize = _sourcesTreeMethods.GetSize(new IntPtr(&_header->FirstAsSource));
387     rightTreeSize = _targetsTreeMethods.GetSize(new IntPtr(&_header->FirstAsTarget));
388     if (leftTreeSize != rightTreeSize)
389     {
390         throw new Exception("One of the trees is broken.");
391     }
392     #endif
393     return linkIndex;
394 }
395
396 [MethodImpl(MethodImplOptions.AggressiveInlining)]
397 private IList<id> GetLinkStruct(id linkIndex)
398 {
399     var link = GetLinkUnsafe(linkIndex);
400     return new UInt64Link(linkIndex, link->Source, link->Target);
401 }
402
403 [MethodImpl(MethodImplOptions.AggressiveInlining)]
404 private Link* GetLinkUnsafe(id linkIndex) => &_links[linkIndex];
405
406 /// <remarks>
407 /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
408 ///        ↪ пространство
409 /// </remarks>
410 public id Create()
411 {
412     var freeLink = _header->FirstFreeLink;
413     if (freeLink != Constants.Null)
414     {
415         _unusedLinksListMethods.Detach(freeLink);
416     }
417     else
418     {
419         if (_header->AllocatedLinks > Constants.MaxPossibleIndex)
420         {
421             throw new LinksLimitReachedException(Constants.MaxPossibleIndex);
422         }
423         if (_header->AllocatedLinks >= _header->ReservedLinks - 1)
424         {
425             _memory.ReservedCapacity += _memory.ReservationStep;
426             SetPointers(_memory);
427             _header->ReservedLinks = (id)(_memory.ReservedCapacity / sizeof(Link));
428         }
429         _header->AllocatedLinks++;
430         _memory.UsedCapacity += sizeof(Link);
431         freeLink = _header->AllocatedLinks;
432     }
433     return freeLink;
434 }
435
436 public void Delete(id link)
437 {
438     if (link < _header->AllocatedLinks)
439     {
440         _unusedLinksListMethods.AttachAsFirst(link);
441     }
442     else if (link == _header->AllocatedLinks)
443     {
444         _header->AllocatedLinks--;
445         _memory.UsedCapacity -= sizeof(Link);
446         // Убираем все связи, находящиеся в списке свободных в конце файла, до тех пор,
447         // пока не дойдём до первой существующей связи
448         // Позволяет оптимизировать количество выделенных связей (AllocatedLinks)

```

```

447         while (_header->AllocatedLinks > 0 && IsUnusedLink(_header->AllocatedLinks))
448         {
449             _unusedLinksListMethods.Detach(_header->AllocatedLinks);
450             _header->AllocatedLinks--;
451             _memory.UsedCapacity -= sizeof(Link);
452         }
453     }
454 }
455
456 /// <remarks>
457 /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том случае, если
458   ↪ адрес реально поменялся
459 ///
460 /// Указатель this.links может быть в том же месте,
461 /// так как 0-я связь не используется и имеет такой же размер как Header,
462 /// поэтому header размещается в том же месте, что и 0-я связь
463 /// </remarks>
464 private void SetPointers(IResizableDirectMemory memory)
465 {
466     if (memory == null)
467     {
468         _header = null;
469         _links = null;
470         _unusedLinksListMethods = null;
471         _targetsTreeMethods = null;
472         _unusedLinksListMethods = null;
473     }
474     else
475     {
476         _header = (LinksHeader*)(void*)memory.Pointer;
477         _links = (Link*)(void*)memory.Pointer;
478         _sourcesTreeMethods = new LinksSourcesTreeMethods(this);
479         _targetsTreeMethods = new LinksTargetsTreeMethods(this);
480         _unusedLinksListMethods = new UnusedLinksListMethods(_links, _header);
481     }
482 }
483
484 [MethodImpl(MethodImplOptions.AggressiveInlining)]
485 private bool Exists(id link) => link >= Constants.MinPossibleIndex && link <=
486   ↪ _header->AllocatedLinks && !IsUnusedLink(link);
487
488 [MethodImpl(MethodImplOptions.AggressiveInlining)]
489 private bool IsUnusedLink(id link) => _header->FirstFreeLink == link
490   || (_links[link].SizeAsSource == Constants.Null &&
491   ↪ _links[link].Source != Constants.Null);
492
493 #region Disposable
494
495 protected override bool AllowMultipleDisposeCalls => true;
496
497 protected override void Dispose(bool manual, bool wasDisposed)
498 {
499     if (!wasDisposed)
500     {
501         SetPointers(null);
502         _memory.DisposeIfPossible();
503     }
504 }
505
506 #endregion
507 }
508 }

```

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

```

1  using Platform.Collections.Methods.Lists;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets.ResizableDirectMemory
6  {
7      unsafe partial class UInt64ResizableDirectMemoryLinks
8      {
9          private class UnusedLinksListMethods : CircularDoublyLinkedListMethods<ulong>
10         {
11             private readonly Link* _links;
12             private readonly LinksHeader* _header;
13
14             public UnusedLinksListMethods(Link* links, LinksHeader* header)
15             {
16                 _links = links;

```

```

17         _header = header;
18     }
19
20     protected override ulong GetFirst() => _header->FirstFreeLink;
21
22     protected override ulong GetLast() => _header->LastFreeLink;
23
24     protected override ulong GetPrevious(ulong element) => _links[element].Source;
25
26     protected override ulong GetNext(ulong element) => _links[element].Target;
27
28     protected override ulong GetSize() => _header->FreeLinks;
29
30     protected override void SetFirst(ulong element) => _header->FirstFreeLink = element;
31
32     protected override void SetLast(ulong element) => _header->LastFreeLink = element;
33
34     protected override void SetPrevious(ulong element, ulong previous) =>
35         ↪ _links[element].Source = previous;
36
37     protected override void SetNext(ulong element, ulong next) => _links[element].Target
38         ↪ = next;
39
40     protected override void SetSize(ulong size) => _header->FreeLinks = size;
41 }

```

./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  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Data.Doublets.ResizableDirectMemory
11 {
12     unsafe partial class UInt64ResizableDirectMemoryLinks
13     {
14         private abstract class LinksTreeMethodsBase :
15             ↪ SizedAndThreadedAVLBalancedTreeMethods<ulong>
16         {
17             private readonly UInt64ResizableDirectMemoryLinks _memory;
18             private readonly LinksCombinedConstants<ulong, ulong, int> _constants;
19             protected readonly Link* Links;
20             protected readonly LinksHeader* Header;
21
22             protected LinksTreeMethodsBase(UInt64ResizableDirectMemoryLinks 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 ulong GetTreeRoot();
32
33             [MethodImpl(MethodImplOptions.AggressiveInlining)]
34             protected abstract ulong GetBasePartValue(ulong link);
35
36             public ulong this[ulong index]
37             {
38                 get
39                 {
40                     var root = GetTreeRoot();
41                     if (index >= GetSize(root))
42                     {
43                         return 0;
44                     }
45                     while (root != 0)
46                     {
47                         var left = GetLeftOrDefault(root);
48                         var leftSize = GetSizeOrZero(left);
49                         if (index < leftSize)
50                         {
51                             root = left;
52                             continue;

```

```

52         }
53         if (index == leftSize)
54         {
55             return root;
56         }
57         root = GetRightOrDefault(root);
58         index -= leftSize + 1;
59     }
60     return 0; // TODO: Impossible situation exception (only if tree structure
        ↳ broken)
61     }
62 }
63
64 // TODO: Return indices range instead of references count
65 public ulong CountUsages(ulong link)
66 {
67     var root = GetTreeRoot();
68     var total = GetSize(root);
69     var totalRightIgnore = 0UL;
70     while (root != 0)
71     {
72         var @base = GetBasePartValue(root);
73         if (@base <= link)
74         {
75             root = GetRightOrDefault(root);
76         }
77         else
78         {
79             totalRightIgnore += GetRightSize(root) + 1;
80             root = GetLeftOrDefault(root);
81         }
82     }
83     root = GetTreeRoot();
84     var totalLeftIgnore = 0UL;
85     while (root != 0)
86     {
87         var @base = GetBasePartValue(root);
88         if (@base >= link)
89         {
90             root = GetLeftOrDefault(root);
91         }
92         else
93         {
94             totalLeftIgnore += GetLeftSize(root) + 1;
95             root = GetRightOrDefault(root);
96         }
97     }
98     return total - totalRightIgnore - totalLeftIgnore;
99 }
100
101 public ulong EachReference(ulong link, Func<IList<ulong>, ulong> handler)
102 {
103     var root = GetTreeRoot();
104     if (root == 0)
105     {
106         return _constants.Continue;
107     }
108     ulong first = 0, current = root;
109     while (current != 0)
110     {
111         var @base = GetBasePartValue(current);
112         if (@base >= link)
113         {
114             if (@base == link)
115             {
116                 first = current;
117             }
118             current = GetLeftOrDefault(current);
119         }
120         else
121         {
122             current = GetRightOrDefault(current);
123         }
124     }
125     if (first != 0)
126     {
127         current = first;
128         while (true)
129         {

```



```

130         if (handler(_memory.GetLinkStruct(current)) == _constants.Break)
131         {
132             return _constants.Break;
133         }
134         current = GetNext(current);
135         if (current == 0 || GetBasePartValue(current) != link)
136         {
137             break;
138         }
139     }
140 }
141 return _constants.Continue;
142 }
143
144 protected override void PrintNodeValue(ulong node, StringBuilder sb)
145 {
146     sb.Append(' ');
147     sb.Append(Links[node].Source);
148     sb.Append('-');
149     sb.Append('>');
150     sb.Append(Links[node].Target);
151 }
152 }
153
154 private class LinksSourcesTreeMethods : LinksTreeMethodsBase
155 {
156     public LinksSourcesTreeMethods(UInt64ResizableDirectMemoryLinks memory)
157         : base(memory)
158     {
159     }
160
161     protected override IntPtr GetLeftPointer(ulong node) => new
162     ↪ IntPtr(&Links[node].LeftAsSource);
163
164     protected override IntPtr GetRightPointer(ulong node) => new
165     ↪ IntPtr(&Links[node].RightAsSource);
166
167     protected override ulong GetLeftValue(ulong node) => Links[node].LeftAsSource;
168     protected override ulong GetRightValue(ulong node) => Links[node].RightAsSource;
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     {

```

```

205     var previousValue = Links[node].SizeAsSource;
206     //return (Integer)Math.PartialRead(previousValue, 3, 1);
207     return (previousValue & 8) >> 3 == 1UL;
208 }
209
210 protected override void SetRightIsChild(ulong node, bool value)
211 {
212     var previousValue = Links[node].SizeAsSource;
213     //var modified = Math.PartialWrite(previousValue, (ulong)(Integer)value, 3, 1);
214     var modified = (previousValue & 4294967287) | ((value ? 1UL : 0UL) << 3);
215     Links[node].SizeAsSource = modified;
216 }
217
218 protected override sbyte GetBalance(ulong node)
219 {
220     var previousValue = Links[node].SizeAsSource;
221     //var value = Math.PartialRead(previousValue, 0, 3);
222     var value = previousValue & 7;
223     var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) << 5) | value & 3 |
224     ↪ 124 : value & 3);
225     return unpackedValue;
226 }
227
228 protected override void SetBalance(ulong node, sbyte value)
229 {
230     var previousValue = Links[node].SizeAsSource;
231     var packagedValue = (ulong)((((byte)value >> 5) & 4) | value & 3);
232     //var modified = Math.PartialWrite(previousValue, packagedValue, 0, 3);
233     var modified = (previousValue & 4294967288) | (packagedValue & 7);
234     Links[node].SizeAsSource = modified;
235 }
236
237 protected override bool FirstIsToTheLeftOfSecond(ulong first, ulong second)
238     => Links[first].Source < Links[second].Source ||
239     (Links[first].Source == Links[second].Source && Links[first].Target <
240     ↪ Links[second].Target);
241
242 protected override bool FirstIsToTheRightOfSecond(ulong first, ulong second)
243     => Links[first].Source > Links[second].Source ||
244     (Links[first].Source == Links[second].Source && Links[first].Target >
245     ↪ Links[second].Target);
246
247 protected override ulong GetTreeRoot() => Header->FirstAsSource;
248
249 protected override ulong GetBasePartValue(ulong link) => Links[link].Source;
250
251 /// <summary>
252 /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
253 ↪ (концом)
254 /// по дереву (индексу) связей, отсортированному по Source, а затем по Target.
255 /// </summary>
256 /// <param name="source">Индекс связи, которая является началом на искомой
257 ↪ связи.</param>
258 /// <param name="target">Индекс связи, которая является концом на искомой
259 ↪ связи.</param>
260 /// <returns>Индекс искомой связи.</returns>
261 public ulong Search(ulong source, ulong target)
262 {
263     var root = Header->FirstAsSource;
264     while (root != 0)
265     {
266         var rootSource = Links[root].Source;
267         var rootTarget = Links[root].Target;
268         if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
269             ↪ node.Key < root.Key
270         {
271             root = GetLeftOrDefault(root);
272         }
273         else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget))
274             ↪ // node.Key > root.Key
275         {
276             root = GetRightOrDefault(root);
277         }
278         else // node.Key == root.Key
279         {
280             return root;
281         }
282     }
283 }

```

```

275     return 0;
276 }
277
278 [MethodImpl(MethodImplOptions.AggressiveInlining)]
279 private static bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
280     ↳ ulong secondSource, ulong secondTarget)
281     => firstSource < secondSource || (firstSource == secondSource && firstTarget <
282     ↳ secondTarget);
283
284 [MethodImpl(MethodImplOptions.AggressiveInlining)]
285 private static bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
286     ↳ ulong secondSource, ulong secondTarget)
287     => firstSource > secondSource || (firstSource == secondSource && firstTarget >
288     ↳ secondTarget);
289
290 [MethodImpl(MethodImplOptions.AggressiveInlining)]
291 protected override void ClearNode(ulong node)
292 {
293     Links[node].LeftAsSource = OUL;
294     Links[node].RightAsSource = OUL;
295     Links[node].SizeAsSource = OUL;
296 }
297
298 [MethodImpl(MethodImplOptions.AggressiveInlining)]
299 protected override ulong GetZero() => OUL;
300
301 [MethodImpl(MethodImplOptions.AggressiveInlining)]
302 protected override ulong GetOne() => 1UL;
303
304 [MethodImpl(MethodImplOptions.AggressiveInlining)]
305 protected override ulong GetTwo() => 2UL;
306
307 [MethodImpl(MethodImplOptions.AggressiveInlining)]
308 protected override bool ValueEqualToZero(IntPtr pointer) =>
309     ↳ *(ulong*)pointer.ToPointer() == OUL;
310
311 [MethodImpl(MethodImplOptions.AggressiveInlining)]
312 protected override bool EqualToZero(ulong value) => value == OUL;
313
314 [MethodImpl(MethodImplOptions.AggressiveInlining)]
315 protected override bool IsEquals(ulong first, ulong second) => first == second;
316
317 [MethodImpl(MethodImplOptions.AggressiveInlining)]
318 protected override bool GreaterThanZero(ulong value) => value > OUL;
319
320 [MethodImpl(MethodImplOptions.AggressiveInlining)]
321 protected override bool GreaterThan(ulong first, ulong second) => first > second;
322
323 [MethodImpl(MethodImplOptions.AggressiveInlining)]
324 protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >=
325     ↳ second;
326
327 [MethodImpl(MethodImplOptions.AggressiveInlining)]
328 protected override bool GreaterOrEqualThanZero(ulong value) => true; // value >= 0
329     ↳ is always true for ulong
330
331 [MethodImpl(MethodImplOptions.AggressiveInlining)]
332 protected override bool LessOrEqualThanZero(ulong value) => value == 0; // value is
333     ↳ always >= 0 for ulong
334
335 [MethodImpl(MethodImplOptions.AggressiveInlining)]
336 protected override bool LessOrEqualThan(ulong first, ulong second) => first <=
337     ↳ second;
338
339 [MethodImpl(MethodImplOptions.AggressiveInlining)]
340 protected override bool LessThanZero(ulong value) => false; // value < 0 is always
341     ↳ false for ulong
342
343 [MethodImpl(MethodImplOptions.AggressiveInlining)]
344 protected override bool LessThan(ulong first, ulong second) => first < second;
345
346 [MethodImpl(MethodImplOptions.AggressiveInlining)]
347 protected override ulong Increment(ulong value) => ++value;
348
349 [MethodImpl(MethodImplOptions.AggressiveInlining)]
350 protected override ulong Decrement(ulong value) => --value;
351
352 [MethodImpl(MethodImplOptions.AggressiveInlining)]
353 protected override ulong Add(ulong first, ulong second) => first + second;

```

```

344     [MethodImpl(MethodImplOptions.AggressiveInlining)]
345     protected override ulong Subtract(ulong first, ulong second) => first - second;
346 }
347
348
349 private class LinksTargetsTreeMethods : LinksTreeMethodsBase
350 {
351     public LinksTargetsTreeMethods(UInt64ResizableDirectMemoryLinks memory)
352         : base(memory)
353     {
354     }
355
356     //protected override IntPtr GetLeft(ulong node) => new
357     ↪ IntPtr(&Links[node].LeftAsTarget);
358
359     //protected override IntPtr GetRight(ulong node) => new
360     ↪ IntPtr(&Links[node].RightAsTarget);
361
362     //protected override ulong GetSize(ulong node) => Links[node].SizeAsTarget;
363
364     //protected override void SetLeft(ulong node, ulong left) =>
365     ↪ Links[node].LeftAsTarget = left;
366
367     //protected override void SetRight(ulong node, ulong right) =>
368     ↪ Links[node].RightAsTarget = right;
369
370     //protected override void SetSize(ulong node, ulong size) =>
371     ↪ Links[node].SizeAsTarget = size;
372
373     protected override IntPtr GetLeftPointer(ulong node) => new
374     ↪ IntPtr(&Links[node].LeftAsTarget);
375
376     protected override IntPtr GetRightPointer(ulong node) => new
377     ↪ IntPtr(&Links[node].RightAsTarget);
378
379     protected override ulong GetLeftValue(ulong node) => Links[node].LeftAsTarget;
380
381     protected override ulong GetRightValue(ulong node) => Links[node].RightAsTarget;
382
383     protected override ulong GetSize(ulong node)
384     {
385         var previousValue = Links[node].SizeAsTarget;
386         //return Math.PartialRead(previousValue, 5, -5);
387         return (previousValue & 4294967264) >> 5;
388     }
389
390     protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsTarget
391     ↪ = left;
392
393     protected override void SetRight(ulong node, ulong right) =>
394     ↪ Links[node].RightAsTarget = right;
395
396     protected override void SetSize(ulong node, ulong size)
397     {
398         var previousValue = Links[node].SizeAsTarget;
399         //var modified = Math.PartialWrite(previousValue, size, 5, -5);
400         var modified = (previousValue & 31) | ((size & 134217727) << 5);
401         Links[node].SizeAsTarget = modified;
402     }
403
404     protected override bool GetLeftIsChild(ulong node)
405     {
406         var previousValue = Links[node].SizeAsTarget;
407         //return (Integer)Math.PartialRead(previousValue, 4, 1);
408         return (previousValue & 16) >> 4 == 1UL;
409         // TODO: Check if this is possible to use
410         //var nodeSize = GetSize(node);
411         //var left = GetLeftValue(node);
412         //var leftSize = GetSizeOrZero(left);
413         //return leftSize > 0 && nodeSize > leftSize;
414     }
415
416     protected override void SetLeftIsChild(ulong node, bool value)
417     {
418         var previousValue = Links[node].SizeAsTarget;
419         //var modified = Math.PartialWrite(previousValue, (ulong)(Integer)value, 4, 1);
420         var modified = (previousValue & 4294967279) | ((value ? 1UL : 0UL) << 4);
421         Links[node].SizeAsTarget = modified;
422     }

```

```

413     }
414
415     protected override bool GetRightIsChild(ulong node)
416     {
417         var previousValue = Links[node].SizeAsTarget;
418         //return (Integer)Math.PartialRead(previousValue, 3, 1);
419         return (previousValue & 8) >> 3 == 1UL;
420         // TODO: Check if this is possible to use
421         //var nodeSize = GetSize(node);
422         //var right = GetRightValue(node);
423         //var rightSize = GetSizeOrZero(right);
424         //return rightSize > 0 && nodeSize > rightSize;
425     }
426
427     protected override void SetRightIsChild(ulong node, bool value)
428     {
429         var previousValue = Links[node].SizeAsTarget;
430         //var modified = Math.PartialWrite(previousValue, (ulong)(Integer)value, 3, 1);
431         var modified = (previousValue & 4294967287) | ((value ? 1UL : 0UL) << 3);
432         Links[node].SizeAsTarget = modified;
433     }
434
435     protected override sbyte GetBalance(ulong node)
436     {
437         var previousValue = Links[node].SizeAsTarget;
438         //var value = Math.PartialRead(previousValue, 0, 3);
439         var value = previousValue & 7;
440         var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) << 5) | value & 3 |
441             ↪ 124 : value & 3);
442         return unpackedValue;
443     }
444
445     protected override void SetBalance(ulong node, sbyte value)
446     {
447         var previousValue = Links[node].SizeAsTarget;
448         var packagedValue = (ulong)((((byte)value >> 5) & 4) | value & 3);
449         //var modified = Math.PartialWrite(previousValue, packagedValue, 0, 3);
450         var modified = (previousValue & 4294967288) | (packagedValue & 7);
451         Links[node].SizeAsTarget = modified;
452     }
453
454     protected override bool FirstIsToTheLeftOfSecond(ulong first, ulong second)
455     => Links[first].Target < Links[second].Target ||
456         (Links[first].Target == Links[second].Target && Links[first].Source <
457             ↪ Links[second].Source);
458
459     protected override bool FirstIsToTheRightOfSecond(ulong first, ulong second)
460     => Links[first].Target > Links[second].Target ||
461         (Links[first].Target == Links[second].Target && Links[first].Source >
462             ↪ Links[second].Source);
463
464     protected override ulong GetTreeRoot() => Header->FirstAsTarget;
465
466     protected override ulong GetBasePartValue(ulong link) => Links[link].Target;
467
468     [MethodImpl(MethodImplOptions.AggressiveInlining)]
469     protected override void ClearNode(ulong node)
470     {
471         Links[node].LeftAsTarget = 0UL;
472         Links[node].RightAsTarget = 0UL;
473         Links[node].SizeAsTarget = 0UL;
474     }
475 }
476 }
477 }
478 }

```

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

```

1  using System.Collections.Generic;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets.Sequences.Converters
6  {
7      public class BalancedVariantConverter<TLink> : LinksListToSequenceConverterBase<TLink>
8      {
9          public BalancedVariantConverter(ILinks<TLink> links) : base(links) { }
10
11          public override TLink Convert(ICollection<TLink> sequence)
12          {

```

```

13     var length = sequence.Count;
14     if (length < 1)
15     {
16         return default;
17     }
18     if (length == 1)
19     {
20         return sequence[0];
21     }
22     // Make copy of next layer
23     if (length > 2)
24     {
25         // TODO: Try to use stackalloc (which at the moment is not working with
26         // ↳ generics) but will be possible with Sigil
27         var halvedSequence = new TLink[(length / 2) + (length % 2)];
28         HalveSequence(halvedSequence, sequence, length);
29         sequence = halvedSequence;
30         length = halvedSequence.Length;
31     }
32     // Keep creating layer after layer
33     while (length > 2)
34     {
35         HalveSequence(sequence, sequence, length);
36         length = (length / 2) + (length % 2);
37     }
38     return Links.GetOrCreate(sequence[0], sequence[1]);
39 }
40 private void HalveSequence(IList<TLink> destination, IList<TLink> source, int length)
41 {
42     var loopedLength = length - (length % 2);
43     for (var i = 0; i < loopedLength; i += 2)
44     {
45         destination[i / 2] = Links.GetOrCreate(source[i], source[i + 1]);
46     }
47     if (length > loopedLength)
48     {
49         destination[length / 2] = source[length - 1];
50     }
51 }
52 }
53 }

```

# ./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 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
12
13 namespace Platform.Data.Doublets.Sequences.Converters
14 {
15     /// <remarks>
16     /// TODO: Возможно будет лучше если алгоритм будет выполняться полностью изолированно от
17     /// ↳ Links на этапе сжатия.
18     /// А именно будет создаваться временный список пар необходимых для выполнения сжатия, в
19     /// ↳ таком случае тип значения элемента массива может быть любым, как char так и ulong.
20     /// Как только список/словарь пар был выявлен можно разом выполнить создание всех этих
21     /// ↳ пар, а так же разом выполнить замену.
22     /// </remarks>
23     public class CompressingConverter<TLink> : LinksListToSequenceConverterBase<TLink>
24     {
25         private static readonly LinksCombinedConstants<bool, TLink, long> _constants =
26             ↳ Default<LinksCombinedConstants<bool, TLink, long>>.Instance;
27         private static readonly EqualityComparer<TLink> _equalityComparer =
28             ↳ EqualityComparer<TLink>.Default;
29         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
30
31         private readonly IConverter<IList<TLink>, TLink> _baseConverter;
32         private readonly LinkFrequenciesCache<TLink> _doubletFrequenciesCache;
33         private readonly TLink _minFrequencyToCompress;
34         private readonly bool _doInitialFrequenciesIncrement;
35         private Doublet<TLink> _maxDoublet;
36         private LinkFrequency<TLink> _maxDoubletData;

```

```

32
33 private struct HalfDoublet
34 {
35     public TLink Element;
36     public LinkFrequency<TLink> DoubletData;
37
38     public HalfDoublet(TLink element, LinkFrequency<TLink> doubletData)
39     {
40         Element = element;
41         DoubletData = doubletData;
42     }
43
44     public override string ToString() => $"{Element}: ({DoubletData})";
45 }
46
47 public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
48     ↪ baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache)
49     : this(links, baseConverter, doubletFrequenciesCache, Integer<TLink>.One, true)
50 {
51 }
52
53 public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
54     ↪ baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache, bool
55     ↪ doInitialFrequenciesIncrement)
56     : this(links, baseConverter, doubletFrequenciesCache, Integer<TLink>.One,
57     ↪ doInitialFrequenciesIncrement)
58 {
59 }
60
61 public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
62     ↪ baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache, TLink
63     ↪ minFrequencyToCompress, bool doInitialFrequenciesIncrement)
64     : base(links)
65 {
66     _baseConverter = baseConverter;
67     _doubletFrequenciesCache = doubletFrequenciesCache;
68     if (_comparer.Compare(minFrequencyToCompress, Integer<TLink>.One) < 0)
69     {
70         minFrequencyToCompress = Integer<TLink>.One;
71     }
72     _minFrequencyToCompress = minFrequencyToCompress;
73     _doInitialFrequenciesIncrement = doInitialFrequenciesIncrement;
74     ResetMaxDoublet();
75 }
76
77 public override TLink Convert(IList<TLink> source) =>
78     ↪ _baseConverter.Convert(Compress(source));
79
80 /// <remarks>
81 /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte\_pair\_encoding .
82 /// Faster version (doublets' frequencies dictionary is not recreated).
83 /// </remarks>
84 private IList<TLink> Compress(IList<TLink> sequence)
85 {
86     if (sequence.IsNullOrEmpty())
87     {
88         return null;
89     }
90     if (sequence.Count == 1)
91     {
92         return sequence;
93     }
94     if (sequence.Count == 2)
95     {
96         return new[] { Links.GetOrCreate(sequence[0], sequence[1]) };
97     }
98     // TODO: arraypool with min size (to improve cache locality) or stackalloc with Sigil
99     var copy = new HalfDoublet[sequence.Count];
100     Doublet<TLink> doublet = default;
101     for (var i = 1; i < sequence.Count; i++)
102     {
103         doublet.Source = sequence[i - 1];
104         doublet.Target = sequence[i];
105         LinkFrequency<TLink> data;
106         if (_doInitialFrequenciesIncrement)
107         {
108             data = _doubletFrequenciesCache.IncrementFrequency(ref doublet);
109         }
110         else

```

```

104     {
105         data = _doubletFrequenciesCache.GetFrequency(ref doublet);
106         if (data == null)
107         {
108             throw new NotSupportedException("If you ask not to increment
109                 ↪ frequencies, it is expected that all frequencies for the sequence
110                 ↪ are prepared.");
111         }
112     }
113     copy[i - 1].Element = sequence[i - 1];
114     copy[i - 1].DoubletData = data;
115     UpdateMaxDoublet(ref doublet, data);
116 }
117 copy[sequence.Count - 1].Element = sequence[sequence.Count - 1];
118 copy[sequence.Count - 1].DoubletData = new LinkFrequency<TLink>();
119 if (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
120 {
121     var newLength = ReplaceDoublets(copy);
122     sequence = new TLink[newLength];
123     for (int i = 0; i < newLength; i++)
124     {
125         sequence[i] = copy[i].Element;
126     }
127 }
128 return sequence;
129 }
130
131 /// <remarks>
132 /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte\_pair\_encoding
133 /// </remarks>
134 private int ReplaceDoublets(HalfDoublet[] copy)
135 {
136     var oldLength = copy.Length;
137     var newLength = copy.Length;
138     while (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
139     {
140         var maxDoubletSource = _maxDoublet.Source;
141         var maxDoubletTarget = _maxDoublet.Target;
142         if (_equalityComparer.Equals(_maxDoubletData.Link, _constants.Null))
143         {
144             _maxDoubletData.Link = Links.GetOrCreate(maxDoubletSource, maxDoubletTarget);
145         }
146         var maxDoubletReplacementLink = _maxDoubletData.Link;
147         oldLength--;
148         var oldLengthMinusTwo = oldLength - 1;
149         // Substitute all usages
150         int w = 0, r = 0; // (r == read, w == write)
151         for (; r < oldLength; r++)
152         {
153             if (_equalityComparer.Equals(copy[r].Element, maxDoubletSource) &&
154                 ↪ _equalityComparer.Equals(copy[r + 1].Element, maxDoubletTarget))
155             {
156                 if (r > 0)
157                 {
158                     var previous = copy[w - 1].Element;
159                     copy[w - 1].DoubletData.DecrementFrequency();
160                     copy[w - 1].DoubletData =
161                         ↪ _doubletFrequenciesCache.IncrementFrequency(previous,
162                         ↪ maxDoubletReplacementLink);
163                 }
164                 if (r < oldLengthMinusTwo)
165                 {
166                     var next = copy[r + 2].Element;
167                     copy[r + 1].DoubletData.DecrementFrequency();
168                     copy[w].DoubletData = _doubletFrequenciesCache.IncrementFrequency(maxDoubletReplacementLink,
169                         ↪ next);
170                 }
171                 copy[w++].Element = maxDoubletReplacementLink;
172                 r++;
173                 newLength--;
174             }
175             else
176             {
177                 copy[w++] = copy[r];
178             }
179         }
180         if (w < newLength)

```



```

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

```

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

```

1 using System.Collections.Generic;
2 using Platform.Interfaces;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Doublets.Sequences.Converters
7 {
8     public abstract class LinksListToSequenceConverterBase<TLink> : IConverter<IList<TLink>,
9     ↪ TLink>
10     {
11         protected readonly ILinks<TLink> Links;
12         public LinksListToSequenceConverterBase(ILinks<TLink> links) => Links = links;
13         public abstract TLink Convert(IList<TLink> source);
14     }
15 }

```

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

```

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

```

```

9 public class OptimalVariantConverter<TLink> : LinksListToSequenceConverterBase<TLink>
10 {
11     private static readonly EqualityComparer<TLink> _equalityComparer =
12         ↳ EqualityComparer<TLink>.Default;
13     private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
14     private readonly IConverter<IList<TLink>> _sequenceToItsLocalElementLevelsConverter;
15
16     public OptimalVariantConverter(ILinks<TLink> links, IConverter<IList<TLink>>
17         ↳ sequenceToItsLocalElementLevelsConverter) : base(links)
18     => _sequenceToItsLocalElementLevelsConverter =
19         ↳ sequenceToItsLocalElementLevelsConverter;
20
21     public override TLink Convert(IList<TLink> sequence)
22     {
23         var length = sequence.Count;
24         if (length == 1)
25         {
26             return sequence[0];
27         }
28         var links = Links;
29         if (length == 2)
30         {
31             return links.GetOrCreate(sequence[0], sequence[1]);
32         }
33         sequence = sequence.ToArray();
34         var levels = _sequenceToItsLocalElementLevelsConverter.Convert(sequence);
35         while (length > 2)
36         {
37             var levelRepeat = 1;
38             var currentLevel = levels[0];
39             var previousLevel = levels[0];
40             var skipOnce = false;
41             var w = 0;
42             for (var i = 1; i < length; i++)
43             {
44                 if (_equalityComparer.Equals(currentLevel, levels[i]))
45                 {
46                     levelRepeat++;
47                     skipOnce = false;
48                     if (levelRepeat == 2)
49                     {
50                         sequence[w] = links.GetOrCreate(sequence[i - 1], sequence[i]);
51                         var newLevel = i >= length - 1 ?
52                             GetPreviousLowerThanCurrentOrCurrent(previousLevel,
53                                 ↳ currentLevel) :
54                             i < 2 ?
55                                 GetNextLowerThanCurrentOrCurrent(currentLevel, levels[i + 1]) :
56                                 GetGreatestNeighbourLowerThanCurrentOrCurrent(previousLevel,
57                                     ↳ currentLevel, levels[i + 1]);
58                         levels[w] = newLevel;
59                         previousLevel = currentLevel;
60                         w++;
61                         levelRepeat = 0;
62                         skipOnce = true;
63                     }
64                     else if (i == length - 1)
65                     {
66                         sequence[w] = sequence[i];
67                         levels[w] = levels[i];
68                         w++;
69                     }
70                 }
71                 else
72                 {
73                     currentLevel = levels[i];
74                     levelRepeat = 1;
75                     if (skipOnce)
76                     {
77                         skipOnce = false;
78                     }
79                     else
80                     {
81                         sequence[w] = sequence[i - 1];
82                         levels[w] = levels[i - 1];
83                         previousLevel = levels[w];
84                         w++;
85                     }
86                     if (i == length - 1)
87                     {
88

```

```

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
↪ current, TLink next)
96 {
97     return _comparer.Compare(previous, next) > 0
98         ? _comparer.Compare(previous, current) < 0 ? previous : current
99         : _comparer.Compare(next, current) < 0 ? next : current;
100 }
101
102 private static TLink GetNextLowerThanCurrentOrCurrent(TLink current, TLink next) =>
↪ _comparer.Compare(next, current) < 0 ? next : current;
103
104 private static TLink GetPreviousLowerThanCurrentOrCurrent(TLink previous, TLink current)
↪ => _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 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Doublets.Sequences.Converters
7 {
8     public class SequenceToItsLocalElementLevelsConverter<TLink> : LinksOperatorBase<TLink>,
↪ IConverter<IList<TLink>>
9     {
10         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
11
12         private readonly IConverter<Doublet<TLink>, TLink> _linkToItsFrequencyToNumberConveter;
13
14         public SequenceToItsLocalElementLevelsConverter(ILinks<TLink> links,
↪ IConverter<Doublet<TLink>, TLink> linkToItsFrequencyToNumberConveter) : base(links)
↪ => _linkToItsFrequencyToNumberConveter = linkToItsFrequencyToNumberConveter;
15
16         public IList<TLink> Convert(IList<TLink> sequence)
17         {
18             var levels = new TLink[sequence.Count];
19             levels[0] = GetFrequencyNumber(sequence[0], sequence[1]);
20             for (var i = 1; i < sequence.Count - 1; i++)
21             {
22                 var previous = GetFrequencyNumber(sequence[i - 1], sequence[i]);
23                 var next = GetFrequencyNumber(sequence[i], sequence[i + 1]);
24                 levels[i] = _comparer.Compare(previous, next) > 0 ? previous : next;
25             }
26             levels[levels.Length - 1] = GetFrequencyNumber(sequence[sequence.Count - 2],
↪ sequence[sequence.Count - 1]);
27             return levels;
28         }
29
30         public TLink GetFrequencyNumber(TLink source, TLink target) =>
↪ _linkToItsFrequencyToNumberConveter.Convert(new Doublet<TLink>(source, target));
31     }
32 }

```

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

```

1 using Platform.Interfaces;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Data.Doublets.Sequences.CreteriaMatchers
6 {
7     public class DefaultSequenceElementCriterionMatcher<TLink> : LinksOperatorBase<TLink>,
↪ ICriterionMatcher<TLink>
8     {
9         public DefaultSequenceElementCriterionMatcher(ILinks<TLink> links) : base(links) { }
10         public bool IsMatched(TLink argument) => Links.IsPartialPoint(argument);

```

```

11     }
12 }

```

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

```

1  using System.Collections.Generic;
2  using Platform.Interfaces;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Sequences.CreteriaMatchers
7  {
8      public class MarkedSequenceCriterionMatcher<TLink> : ICriterionMatcher<TLink>
9      {
10         private static readonly EqualityComparer<TLink> _equalityComparer =
11             ↪ EqualityComparer<TLink>.Default;
12
13         private readonly ILinks<TLink> _links;
14         private readonly TLink _sequenceMarkerLink;
15
16         public MarkedSequenceCriterionMatcher(ILinks<TLink> links, TLink sequenceMarkerLink)
17         {
18             _links = links;
19             _sequenceMarkerLink = sequenceMarkerLink;
20         }
21
22         public bool IsMatched(TLink sequenceCandidate)
23         => _equalityComparer.Equals(_links.GetSource(sequenceCandidate), _sequenceMarkerLink)
24         || !_equalityComparer.Equals(_links.SearchOrDefault(_sequenceMarkerLink,
25             ↪ sequenceCandidate), _links.Constants.Null);
26     }
27 }

```

./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.Sequences;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets.Sequences
9  {
10     public class DefaultSequenceAppender<TLink> : LinksOperatorBase<TLink>,
11         ↪ ISequenceAppender<TLink>
12     {
13         private static readonly EqualityComparer<TLink> _equalityComparer =
14             ↪ EqualityComparer<TLink>.Default;
15
16         private readonly IStack<TLink> _stack;
17         private readonly ISequenceHeightProvider<TLink> _heightProvider;
18
19         public DefaultSequenceAppender(ILinks<TLink> links, IStack<TLink> stack,
20             ↪ ISequenceHeightProvider<TLink> heightProvider)
21             : base(links)
22         {
23             _stack = stack;
24             _heightProvider = heightProvider;
25         }
26
27         public TLink Append(TLink sequence, TLink appendant)
28         {
29             var cursor = sequence;
30             while (!_equalityComparer.Equals(_heightProvider.Get(cursor), default))
31             {
32                 var source = Links.GetSource(cursor);
33                 var target = Links.GetTarget(cursor);
34                 if (_equalityComparer.Equals(_heightProvider.Get(source),
35                     ↪ _heightProvider.Get(target)))
36                 {
37                     break;
38                 }
39                 else
40                 {
41                     _stack.Push(source);
42                     cursor = target;
43                 }
44             }
45             var left = cursor;
46             var right = appendant;
47             while (!_equalityComparer.Equals(cursor = _stack.Pop(), Links.Constants.Null))

```

```

44         {
45             right = Links.GetOrCreate(left, right);
46             left = cursor;
47         }
48         return Links.GetOrCreate(left, right);
49     }
50 }
51 }

```

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

```

1  using System.Collections.Generic;
2  using System.Linq;
3  using Platform.Interfaces;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Sequences
8  {
9      public class DuplicateSegmentsCounter<TLink> : ICounter<int>
10     {
11         private readonly IProvider<IList<KeyValuePair<IList<TLink>, IList<TLink>>>>
12             ↪ _duplicateFragmentsProvider;
13         public DuplicateSegmentsCounter(IProvider<IList<KeyValuePair<IList<TLink>,
14             ↪ IList<TLink>>>> duplicateFragmentsProvider) => _duplicateFragmentsProvider =
15             ↪ duplicateFragmentsProvider;
16         public int Count() => _duplicateFragmentsProvider.Get().Sum(x => x.Value.Count);
17     }
18 }

```

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

```

1  using System;
2  using System.Linq;
3  using System.Collections.Generic;
4  using Platform.Interfaces;
5  using Platform.Collections;
6  using Platform.Collections.Lists;
7  using Platform.Collections.Segments;
8  using Platform.Collections.Segments.Walkers;
9  using Platform.Singletons;
10 using Platform.Numbers;
11 using Platform.Data.Sequences;
12 using Platform.Data.Doublets.Unicode;
13
14 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
15
16 namespace Platform.Data.Doublets.Sequences
17 {
18     public class DuplicateSegmentsProvider<TLink> :
19         ↪ DictionaryBasedDuplicateSegmentsWalkerBase<TLink>,
20         ↪ IProvider<IList<KeyValuePair<IList<TLink>, IList<TLink>>>>
21     {
22         private readonly ILinks<TLink> _links;
23         private readonly ISequences<TLink> _sequences;
24         private HashSet<KeyValuePair<IList<TLink>, IList<TLink>>> _groups;
25         private BitString _visited;
26
27         private class ItemEquilityComparer : IEqualityComparer<KeyValuePair<IList<TLink>,
28             ↪ IList<TLink>>>
29         {
30             private readonly IListEqualityComparer<TLink> _listComparer;
31             public ItemEquilityComparer() => _listComparer =
32                 ↪ Default<IListEqualityComparer<TLink>>.Instance;
33             public bool Equals(KeyValuePair<IList<TLink>, IList<TLink>> left,
34                 ↪ KeyValuePair<IList<TLink>, IList<TLink>> right) =>
35                 ↪ _listComparer.Equals(left.Key, right.Key) && _listComparer.Equals(left.Value,
36                 ↪ right.Value);
37             public int GetHashCode(KeyValuePair<IList<TLink>, IList<TLink>> pair) =>
38                 ↪ (_listComparer.GetHashCode(pair.Key),
39                 ↪ _listComparer.GetHashCode(pair.Value)).GetHashCode();
40         }
41
42         private class ItemComparer : IComparer<KeyValuePair<IList<TLink>, IList<TLink>>>
43         {
44             private readonly IListComparer<TLink> _listComparer;
45
46             public ItemComparer() => _listComparer = Default<IListComparer<TLink>>.Instance;
47
48             public int Compare(KeyValuePair<IList<TLink>, IList<TLink>> left,
49                 ↪ KeyValuePair<IList<TLink>, IList<TLink>> right)
50             {

```

```

41         var intermediateResult = _listComparer.Compare(left.Key, right.Key);
42         if (intermediateResult == 0)
43         {
44             intermediateResult = _listComparer.Compare(left.Value, right.Value);
45         }
46         return intermediateResult;
47     }
48 }
49
50 public DuplicateSegmentsProvider(ILinks<TLink> links, ISequences<TLink> sequences)
51     : base(minimumStringSegmentLength: 2)
52 {
53     _links = links;
54     _sequences = sequences;
55 }
56
57 public IList<KeyValuePair<IList<TLink>, IList<TLink>>> Get()
58 {
59     _groups = new HashSet<KeyValuePair<IList<TLink>,
60         ↪ IList<TLink>>>(Default<ItemEqualityComparer>.Instance);
61     var count = _links.Count();
62     _visited = new BitString((long)(Integer<TLink>)count + 1);
63     _links.Each(link =>
64     {
65         var linkIndex = _links.GetIndex(link);
66         var linkBitIndex = (long)(Integer<TLink>)linkIndex;
67         if (!_visited.Get(linkBitIndex))
68         {
69             var sequenceElements = new List<TLink>();
70             _sequences.EachPart(sequenceElements.AddAndReturnTrue, linkIndex);
71             if (sequenceElements.Count > 2)
72             {
73                 WalkAll(sequenceElements);
74             }
75             return _links.Constants.Continue;
76         });
77     var resultList = _groups.ToList();
78     var comparer = Default<ItemComparer>.Instance;
79     resultList.Sort(comparer);
80     #if DEBUG
81     foreach (var item in resultList)
82     {
83         PrintDuplicates(item);
84     }
85     #endif
86     return resultList;
87 }
88
89 protected override Segment<TLink> CreateSegment(IList<TLink> elements, int offset, int
90     ↪ length) => new Segment<TLink>(elements, offset, length);
91
92 protected override void OnDuplicateFound(Segment<TLink> segment)
93 {
94     var duplicates = CollectDuplicatesForSegment(segment);
95     if (duplicates.Count > 1)
96     {
97         _groups.Add(new KeyValuePair<IList<TLink>, IList<TLink>>(segment.ToArray(),
98             ↪ duplicates));
99     }
100 }
101
102 private List<TLink> CollectDuplicatesForSegment(Segment<TLink> segment)
103 {
104     var duplicates = new List<TLink>();
105     var readAsElement = new HashSet<TLink>();
106     _sequences.Each(sequence =>
107     {
108         duplicates.Add(sequence);
109         readAsElement.Add(sequence);
110         return true; // Continue
111     }, segment);
112     if (duplicates.Any(x => _visited.Get((Integer<TLink>)x)))
113     {
114         return new List<TLink>();
115     }
116     foreach (var duplicate in duplicates)
117     {

```

```

116         var duplicateBitIndex = (long)(Integer<TLink>)duplicate;
117         _visited.Set(duplicateBitIndex);
118     }
119     if (_sequences is Sequences sequencesExperiments)
120     {
121         var partiallyMatched = sequencesExperiments.GetAllPartiallyMatchingSequences4((H
            ↳ ashSet<ulong>)(object)readAsElement,
            ↳ (IList<ulong>)segment);
122         foreach (var partiallyMatchedSequence in partiallyMatched)
123         {
124             TLink sequenceIndex = (Integer<TLink>)partiallyMatchedSequence;
125             duplicates.Add(sequenceIndex);
126         }
127     }
128     duplicates.Sort();
129     return duplicates;
130 }
131
132 private void PrintDuplicates(KeyValuePair<IList<TLink>, IList<TLink>> duplicatesItem)
133 {
134     if (!(_links is ILinks<ulong> ulongLinks))
135     {
136         return;
137     }
138     var duplicatesKey = duplicatesItem.Key;
139     var keyString = UnicodeMap.FromLinksToString((IList<ulong>)duplicatesKey);
140     Console.WriteLine($"> {keyString} ({string.Join(", ", duplicatesKey)}");
141     var duplicatesList = duplicatesItem.Value;
142     for (int i = 0; i < duplicatesList.Count; i++)
143     {
144         ulong sequenceIndex = (Integer<TLink>)duplicatesList[i];
145         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));
146         Console.WriteLine(formattedSequenceStructure);
147         var sequenceString = UnicodeMap.FromSequenceLinkToString(sequenceIndex,
            ↳ ulongLinks);
148         Console.WriteLine(sequenceString);
149     }
150     Console.WriteLine();
151 }
152 }
153 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Interfaces;
5  using Platform.Numbers;
6
7  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
10 {
11     /// <remarks>
12     /// Can be used to operate with many CompressingConverters (to keep global frequencies data
13     ↳ between them).
14     /// TODO: Extract interface to implement frequencies storage inside Links storage
15     /// </remarks>
16     public class LinkFrequenciesCache<TLink> : LinksOperatorBase<TLink>
17     {
18         private static readonly EqualityComparer<TLink> _equalityComparer =
19             ↳ EqualityComparer<TLink>.Default;
20         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
21
22         private readonly Dictionary<Doublet<TLink>, LinkFrequency<TLink>> _doubletsCache;
23         private readonly ICounter<TLink, TLink> _frequencyCounter;
24
25         public LinkFrequenciesCache(ILinks<TLink> links, ICounter<TLink, TLink> frequencyCounter)
26             : base(links)
27         {
28             _doubletsCache = new Dictionary<Doublet<TLink>, LinkFrequency<TLink>>(4096,
29                 ↳ DoubletComparer<TLink>.Default);
30             _frequencyCounter = frequencyCounter;
31         }
32
33         [MethodImpl(MethodImplOptions.AggressiveInlining)]
34     }
35 }

```

```

public LinkFrequency<TLink> GetFrequency(TLink source, TLink target)
{
    var doublet = new Doublet<TLink>(source, target);
    return GetFrequency(ref doublet);
}

[MethodImpl(MethodImplOptions.AggressiveInlining)]
public LinkFrequency<TLink> GetFrequency(ref Doublet<TLink> doublet)
{
    _doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data);
    return data;
}

public void IncrementFrequencies(IList<TLink> sequence)
{
    for (var i = 1; i < sequence.Count; i++)
    {
        IncrementFrequency(sequence[i - 1], sequence[i]);
    }
}

[MethodImpl(MethodImplOptions.AggressiveInlining)]
public LinkFrequency<TLink> IncrementFrequency(TLink source, TLink target)
{
    var doublet = new Doublet<TLink>(source, target);
    return IncrementFrequency(ref doublet);
}

public void PrintFrequencies(IList<TLink> sequence)
{
    for (var i = 1; i < sequence.Count; i++)
    {
        PrintFrequency(sequence[i - 1], sequence[i]);
    }
}

public void PrintFrequency(TLink source, TLink target)
{
    var number = GetFrequency(source, target).Frequency;
    Console.WriteLine("{0},{1} - {2}", source, target, number);
}

[MethodImpl(MethodImplOptions.AggressiveInlining)]
public LinkFrequency<TLink> IncrementFrequency(ref Doublet<TLink> doublet)
{
    if (_doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data))
    {
        data.IncrementFrequency();
    }
    else
    {
        var link = Links.SearchOrDefault(doublet.Source, doublet.Target);
        data = new LinkFrequency<TLink>(Integer<TLink>.One, link);
        if (!_equalityComparer.Equals(link, default))
        {
            data.Frequency = Arithmetic.Add(data.Frequency,
                ↪ _frequencyCounter.Count(link));
        }
        _doubletsCache.Add(doublet, data);
    }
    return data;
}

public void ValidateFrequencies()
{
    foreach (var entry in _doubletsCache)
    {
        var value = entry.Value;
        var linkIndex = value.Link;
        if (!_equalityComparer.Equals(linkIndex, default))
        {
            var frequency = value.Frequency;
            var count = _frequencyCounter.Count(linkIndex);
            // TODO: Why `frequency` always greater than `count` by 1?
            if (((_comparer.Compare(frequency, count) > 0) &&
                ↪ (_comparer.Compare(Arithmetic.Subtract(frequency, count),
                ↪ Integer<TLink>.One) > 0))

```



```

105         || (_comparer.Compare(count, frequency) > 0) &&
           ↳ (_comparer.Compare(Arithmetic.Subtract(count, frequency),
           ↳ Integer<TLink>.One) > 0)))
106     {
107         throw new InvalidOperationException("Frequencies validation failed.");
108     }
109 }
110 //else
111 //{
112 //    if (value.Frequency > 0)
113 //    {
114 //        var frequency = value.Frequency;
115 //        linkIndex = _createLink(entry.Key.Source, entry.Key.Target);
116 //        var count = _countLinkFrequency(linkIndex);
117 //
118 //        if ((frequency > count && frequency - count > 1) || (count > frequency
           ↳ && count - frequency > 1))
119 //            throw new Exception("Frequencies validation failed.");
120 //    }
121 //}
122 }
123 }
124 }
125 }

```

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

```

1 using System.Runtime.CompilerServices;
2 using Platform.Numbers;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
7 {
8     public class LinkFrequency<TLink>
9     {
10         public TLink Frequency { get; set; }
11         public TLink Link { get; set; }
12
13         public LinkFrequency(TLink frequency, TLink link)
14         {
15             Frequency = frequency;
16             Link = link;
17         }
18
19         public LinkFrequency() { }
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public void IncrementFrequency() => Frequency = Arithmetic<TLink>.Increment(Frequency);
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         public void DecrementFrequency() => Frequency = Arithmetic<TLink>.Decrement(Frequency);
26
27         public override string ToString() => $"F: {Frequency}, L: {Link}";
28     }
29 }

```

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

```

1 using Platform.Interfaces;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Data.Doublets.Sequences.Frequencies.Cache
6 {
7     public class FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<TLink> :
           ↳ IConverter<Doublet<TLink>, TLink>
8     {
9         private readonly LinkFrequenciesCache<TLink> _cache;
10        public
           ↳ FrequenciesCacheBasedLinkToItsFrequencyNumberConverter(LinkFrequenciesCache<TLink>
           ↳ cache) => _cache = cache;
11        public TLink Convert(Doublet<TLink> source) => _cache.GetFrequency(ref source).Frequency;
12    }
13 }

```

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

```

1 using Platform.Interfaces;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member

```

```

4
5 namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
6 {
7     public class MarkedSequenceSymbolFrequencyOneOffCounter<TLink> :
8         ↳ SequenceSymbolFrequencyOneOffCounter<TLink>
9     {
10         private readonly ICriterionMatcher<TLink> _markedSequenceMatcher;
11
12         public MarkedSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links,
13             ↳ ICriterionMatcher<TLink> markedSequenceMatcher, TLink sequenceLink, TLink symbol)
14             : base(links, sequenceLink, symbol)
15             => _markedSequenceMatcher = markedSequenceMatcher;
16
17         public override TLink Count()
18         {
19             if (!_markedSequenceMatcher.IsMatched(_sequenceLink))
20             {
21                 return default;
22             }
23             return base.Count();
24         }
25     }
26 }

```

./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 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8 namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
9 {
10     public class SequenceSymbolFrequencyOneOffCounter<TLink> : ICounter<TLink>
11     {
12         private static readonly EqualityComparer<TLink> _equalityComparer =
13             ↳ EqualityComparer<TLink>.Default;
14         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
15
16         protected readonly ILinks<TLink> _links;
17         protected readonly TLink _sequenceLink;
18         protected readonly TLink _symbol;
19         protected TLink _total;
20
21         public SequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links, TLink sequenceLink,
22             ↳ TLink symbol)
23         {
24             _links = links;
25             _sequenceLink = sequenceLink;
26             _symbol = symbol;
27             _total = default;
28         }
29
30         public virtual TLink Count()
31         {
32             if (_comparer.Compare(_total, default) > 0)
33             {
34                 return _total;
35             }
36             StopableSequenceWalker.WalkRight(_sequenceLink, _links.GetSource, _links.GetTarget,
37                 ↳ IsElement, VisitElement);
38             return _total;
39         }
40
41         private bool IsElement(TLink x) => _equalityComparer.Equals(x, _symbol) ||
42             ↳ _links.IsPartialPoint(x); // TODO: Use SequenceElementCriteriaMatcher instead of
43             ↳ IsPartialPoint
44
45         private bool VisitElement(TLink element)
46         {
47             if (_equalityComparer.Equals(element, _symbol))
48             {
49                 _total = Arithmetic.Increment(_total);
50             }
51             return true;
52         }
53     }
54 }

```

```

./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyCounter.cs
1  using Platform.Interfaces;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
6  {
7      public class TotalMarkedSequenceSymbolFrequencyCounter<TLink> : ICounter<TLink, TLink>
8      {
9          private readonly ILinks<TLink> _links;
10         private readonly ICriterionMatcher<TLink> _markedSequenceMatcher;
11
12         public TotalMarkedSequenceSymbolFrequencyCounter(ILinks<TLink> links,
13             ↪ ICriterionMatcher<TLink> markedSequenceMatcher)
14         {
15             _links = links;
16             _markedSequenceMatcher = markedSequenceMatcher;
17         }
18
19         public TLink Count(TLink argument) => new
20             ↪ TotalMarkedSequenceSymbolFrequencyOneOffCounter<TLink>(_links,
21             ↪ _markedSequenceMatcher, argument).Count();
22     }
23 }

```

```

./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyOneOffCounter
1  using Platform.Interfaces;
2  using Platform.Numbers;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
7  {
8      public class TotalMarkedSequenceSymbolFrequencyOneOffCounter<TLink> :
9          ↪ TotalSequenceSymbolFrequencyOneOffCounter<TLink>
10      {
11         private readonly ICriterionMatcher<TLink> _markedSequenceMatcher;
12
13         public TotalMarkedSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links,
14             ↪ ICriterionMatcher<TLink> markedSequenceMatcher, TLink symbol)
15             : base(links, symbol)
16             => _markedSequenceMatcher = markedSequenceMatcher;
17
18         protected override void CountSequenceSymbolFrequency(TLink link)
19         {
20             var symbolFrequencyCounter = new
21                 ↪ MarkedSequenceSymbolFrequencyOneOffCounter<TLink>(_links,
22                 ↪ _markedSequenceMatcher, link, _symbol);
23             _total = Arithmetic.Add(_total, symbolFrequencyCounter.Count());
24         }
25     }
26 }

```

```

./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyCounter.cs
1  using Platform.Interfaces;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
6  {
7      public class TotalSequenceSymbolFrequencyCounter<TLink> : ICounter<TLink, TLink>
8      {
9          private readonly ILinks<TLink> _links;
10         public TotalSequenceSymbolFrequencyCounter(ILinks<TLink> links) => _links = links;
11         public TLink Count(TLink symbol) => new
12             ↪ TotalSequenceSymbolFrequencyOneOffCounter<TLink>(_links, symbol).Count();
13     }
14 }

```

```

./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyOneOffCounter.cs
1  using System.Collections.Generic;
2  using Platform.Interfaces;
3  using Platform.Numbers;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
8  {
9      public class TotalSequenceSymbolFrequencyOneOffCounter<TLink> : ICounter<TLink>

```

```

10 {
11     private static readonly EqualityComparer<TLink> _equalityComparer =
12         ↪ EqualityComparer<TLink>.Default;
13     private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
14
15     protected readonly ILinks<TLink> _links;
16     protected readonly TLink _symbol;
17     protected readonly HashSet<TLink> _visits;
18     protected TLink _total;
19
20     public TotalSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links, TLink symbol)
21     {
22         _links = links;
23         _symbol = symbol;
24         _visits = new HashSet<TLink>();
25         _total = default;
26     }
27
28     public TLink Count()
29     {
30         if (_comparer.Compare(_total, default) > 0 || _visits.Count > 0)
31         {
32             return _total;
33         }
34         CountCore(_symbol);
35         return _total;
36     }
37
38     private void CountCore(TLink link)
39     {
40         var any = _links.Constants.Any;
41         if (_equalityComparer.Equals(_links.Count(any, link), default))
42         {
43             CountSequenceSymbolFrequency(link);
44         }
45         else
46         {
47             _links.Each(EachElementHandler, any, link);
48         }
49     }
50
51     protected virtual void CountSequenceSymbolFrequency(TLink link)
52     {
53         var symbolFrequencyCounter = new SequenceSymbolFrequencyOneOffCounter<TLink>(_links,
54             ↪ link, _symbol);
55         _total = Arithmetic.Add(_total, symbolFrequencyCounter.Count());
56     }
57
58     private TLink EachElementHandler(IList<TLink> doublet)
59     {
60         var constants = _links.Constants;
61         var doubletIndex = doublet[constants.IndexPart];
62         if (_visits.Add(doubletIndex))
63         {
64             CountCore(doubletIndex);
65         }
66         return constants.Continue;
67     }
68 }

```

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

```

1 using System.Collections.Generic;
2 using Platform.Interfaces;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Doublets.Sequences.HeightProviders
7 {
8     public class CachedSequenceHeightProvider<TLink> : LinksOperatorBase<TLink>,
9         ↪ ISequenceHeightProvider<TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
12             ↪ EqualityComparer<TLink>.Default;
13
14         private readonly TLink _heightPropertyMarker;
15         private readonly ISequenceHeightProvider<TLink> _baseHeightProvider;
16         private readonly IConverter<TLink> _addressToUnaryNumberConverter;
17         private readonly IConverter<TLink> _unaryNumberToAddressConverter;
18         private readonly IPropertiesOperator<TLink, TLink, TLink> _propertyOperator;

```

```

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  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Sequences.HeightProviders
7  {
8      public class DefaultSequenceRightHeightProvider<TLink> : LinksOperatorBase<TLink>,
9          ↳ ISequenceHeightProvider<TLink>
10     {
11         private readonly ICriterionMatcher<TLink> _elementMatcher;
12
13         public DefaultSequenceRightHeightProvider(ILinks<TLink> links, ICriterionMatcher<TLink>
14             ↳ elementMatcher) : base(links) => _elementMatcher = elementMatcher;
15
16         public TLink Get(TLink sequence)
17         {
18             var height = default(TLink);
19             var pairOrElement = sequence;
20             while (!_elementMatcher.IsMatched(pairOrElement))
21             {
22                 pairOrElement = Links.GetTarget(pairOrElement);
23                 height = Arithmetic.Increment(height);
24             }
25             return height;
26         }
27     }
28 }

```

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

```

1  using Platform.Interfaces;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets.Sequences.HeightProviders
6  {
7      public interface ISequenceHeightProvider<TLink> : IProvider<TLink, TLink>
8      {
9      }
10 }

```

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

```

1  using System.Collections.Generic;
2  using Platform.Data.Doublets.Sequences.Frequencies.Cache;

```

```

3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Doublets.Sequences.Indexes
7 {
8     public class CachedFrequencyIncrementingSequenceIndex<TLink> : ISequenceIndex<TLink>
9     {
10         private static readonly EqualityComparer<TLink> _equalityComparer =
11             ↳ EqualityComparer<TLink>.Default;
12
13         private readonly LinkFrequenciesCache<TLink> _cache;
14
15         public CachedFrequencyIncrementingSequenceIndex(LinkFrequenciesCache<TLink> cache) =>
16             ↳ _cache = cache;
17
18         public bool Add(IList<TLink> sequence)
19         {
20             var indexed = true;
21             var i = sequence.Count;
22             while (--i >= 1 && (indexed = IsIndexedWithIncrement(sequence[i - 1], sequence[i])))
23                 ↳ { }
24             for (; i >= 1; i--)
25             {
26                 _cache.IncrementFrequency(sequence[i - 1], sequence[i]);
27             }
28             return indexed;
29         }
30
31         private bool IsIndexedWithIncrement(TLink source, TLink target)
32         {
33             var frequency = _cache.GetFrequency(source, target);
34             if (frequency == null)
35             {
36                 return false;
37             }
38             var indexed = !_equalityComparer.Equals(frequency.Frequency, default);
39             if (indexed)
40             {
41                 _cache.IncrementFrequency(source, target);
42             }
43             return indexed;
44         }
45
46         public bool MightContain(IList<TLink> sequence)
47         {
48             var indexed = true;
49             var i = sequence.Count;
50             while (--i >= 1 && (indexed = IsIndexed(sequence[i - 1], sequence[i]))) { }
51             return indexed;
52         }
53
54         private bool IsIndexed(TLink source, TLink target)
55         {
56             var frequency = _cache.GetFrequency(source, target);
57             if (frequency == null)
58             {
59                 return false;
60             }
61             return !_equalityComparer.Equals(frequency.Frequency, default);
62         }
63     }
64 }

```

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

```

1 using Platform.Interfaces;
2 using System.Collections.Generic;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Doublets.Sequences.Indexes
7 {
8     public class FrequencyIncrementingSequenceIndex<TLink> : SequenceIndex<TLink>,
9         ↳ ISequenceIndex<TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
12             ↳ EqualityComparer<TLink>.Default;
13
14         private readonly IPropertyOperator<TLink, TLink> _frequencyPropertyOperator;
15         private readonly IIncrementer<TLink> _frequencyIncrementer;
16     }
17 }

```

```

15     public FrequencyIncrementingSequenceIndex(ILinks<TLink> links, IPropertyOperator<TLink,
16         ↪ TLink> frequencyPropertyOperator, IIncrementer<TLink> frequencyIncrementer)
17         : base(links)
18     {
19         _frequencyPropertyOperator = frequencyPropertyOperator;
20         _frequencyIncrementer = frequencyIncrementer;
21     }
22     public override bool Add(IList<TLink> sequence)
23     {
24         var indexed = true;
25         var i = sequence.Count;
26         while (--i >= 1 && (indexed = IsIndexedWithIncrement(sequence[i - 1], sequence[i])))
27             ↪ { }
28         for (; i >= 1; i--)
29         {
30             Increment(Links.GetOrCreate(sequence[i - 1], sequence[i]));
31         }
32         return indexed;
33     }
34     private bool IsIndexedWithIncrement(TLink source, TLink target)
35     {
36         var link = Links.SearchOrCreate(source, target);
37         var indexed = !_equalityComparer.Equals(link, default);
38         if (indexed)
39         {
40             Increment(link);
41         }
42         return indexed;
43     }
44     private void Increment(TLink link)
45     {
46         var previousFrequency = _frequencyPropertyOperator.Get(link);
47         var frequency = _frequencyIncrementer.Increment(previousFrequency);
48         _frequencyPropertyOperator.Set(link, frequency);
49     }
50 }
51 }
52 }

```

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

```

1 using System.Collections.Generic;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Data.Doublets.Sequences.Indexes
6 {
7     public interface ISequenceIndex<TLink>
8     {
9         /// <summary>
10         /// Индексирует последовательность глобально, и возвращает значение,
11         /// определяющие была ли запрошенная последовательность проиндексирована ранее.
12         /// </summary>
13         /// <param name="sequence">Последовательность для индексации.</param>
14         bool Add(IList<TLink> sequence);
15
16         bool MightContain(IList<TLink> sequence);
17     }
18 }

```

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

```

1 using System.Collections.Generic;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Data.Doublets.Sequences.Indexes
6 {
7     public class SequenceIndex<TLink> : LinksOperatorBase<TLink>, ISequenceIndex<TLink>
8     {
9         private static readonly EqualityComparer<TLink> _equalityComparer =
10             ↪ EqualityComparer<TLink>.Default;
11
12         public SequenceIndex(ILinks<TLink> links) : base(links) { }
13
14         public virtual bool Add(IList<TLink> sequence)
15         {
16             var indexed = true;
17             var i = sequence.Count;

```

```

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

```

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

```

1 using System.Collections.Generic;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Data.Doublets.Sequences.Indexes
6 {
7     public class SynchronizedSequenceIndex<TLink> : ISequenceIndex<TLink>
8     {
9         private static readonly EqualityComparer<TLink> _equalityComparer =
10             ↪ EqualityComparer<TLink>.Default;
11
12         private readonly ISynchronizedLinks<TLink> _links;
13
14         public SynchronizedSequenceIndex(ISynchronizedLinks<TLink> links) => _links = links;
15
16         public bool Add(IList<TLink> sequence)
17         {
18             var indexed = true;
19             var i = sequence.Count;
20             var links = _links.Unsync;
21             _links.SyncRoot.ExecuteReadOperation(() =>
22             {
23                 while (--i >= 1 && (indexed =
24                     ↪ !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1],
25                     ↪ sequence[i]), default))) { }
26             });
27             if (!indexed)
28             {
29                 _links.SyncRoot.ExecuteWriteOperation(() =>
30                 {
31                     for (; i >= 1; i--)
32                     {
33                         links.GetOrCreate(sequence[i - 1], sequence[i]);
34                     }
35                 });
36             }
37             return indexed;
38         }
39
40         public bool MightContain(IList<TLink> sequence)
41         {
42             var links = _links.Unsync;
43             return _links.SyncRoot.ExecuteReadOperation(() =>
44             {
45                 var indexed = true;
46                 var i = sequence.Count;
47                 while (--i >= 1 && (indexed =
48                     ↪ !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1],
49                     ↪ sequence[i]), default))) { }
50                 return indexed;
51             });
52         }
53     }
54 }
55 }

```



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

```

73     Options = options;
74
75     Options.ValidateOptions();
76     Options.InitOptions(Links);
77 }
78
79 public bool IsSequence(ulong sequence)
80 {
81     return Sync.ExecuteReadOperation(() =>
82     {
83         if (Options.UseSequenceMarker)
84         {
85             return Options.MarkedSequenceMatcher.IsMatched(sequence);
86         }
87         return !Links.Unsync.IsPartialPoint(sequence);
88     });
89 }
90
91 [MethodImpl(MethodImplOptions.AggressiveInlining)]
92 private ulong GetSequenceByElements(ulong sequence)
93 {
94     if (Options.UseSequenceMarker)
95     {
96         return Links.SearchOrDefault(Options.SequenceMarkerLink, sequence);
97     }
98     return sequence;
99 }
100
101 private ulong GetSequenceElements(ulong sequence)
102 {
103     if (Options.UseSequenceMarker)
104     {
105         var linkContents = new UInt64Link(Links.GetLink(sequence));
106         if (linkContents.Source == Options.SequenceMarkerLink)
107         {
108             return linkContents.Target;
109         }
110         if (linkContents.Target == Options.SequenceMarkerLink)
111         {
112             return linkContents.Source;
113         }
114     }
115     return sequence;
116 }
117
118 #region Count
119
120 public ulong Count(params ulong[] sequence)
121 {
122     if (sequence.Length == 0)
123     {
124         return Links.Count(_constants.Any, Options.SequenceMarkerLink, _constants.Any);
125     }
126     if (sequence.Length == 1) // Первая связь это адрес
127     {
128         if (sequence[0] == _constants.Null)
129         {
130             return 0;
131         }
132         if (sequence[0] == _constants.Any)
133         {
134             return Count();
135         }
136         if (Options.UseSequenceMarker)
137         {
138             return Links.Count(_constants.Any, Options.SequenceMarkerLink, sequence[0]);
139         }
140         return Links.Exists(sequence[0]) ? 1UL : 0;
141     }
142     throw new NotImplementedException();
143 }
144
145 private ulong CountUsages(params ulong[] restrictions)
146 {
147     if (restrictions.Length == 0)
148     {
149         return 0;
150     }
151     if (restrictions.Length == 1) // Первая связь это адрес

```

```

152     {
153         if (restrictions[0] == _constants.Null)
154         {
155             return 0;
156         }
157         if (Options.UseSequenceMarker)
158         {
159             var elementsLink = GetSequenceElements(restrictions[0]);
160             var sequenceLink = GetSequenceByElements(elementsLink);
161             if (sequenceLink != _constants.Null)
162             {
163                 return Links.Count(sequenceLink) + Links.Count(elementsLink) - 1;
164             }
165             return Links.Count(elementsLink);
166         }
167         return Links.Count(restrictions[0]);
168     }
169     throw new NotImplementedException();
170 }
171
172 #endregion
173
174 #region Create
175
176 public ulong Create(params ulong[] sequence)
177 {
178     return Sync.ExecuteWriteOperation(() =>
179     {
180         if (sequence.IsNullOrEmpty())
181         {
182             return _constants.Null;
183         }
184         Links.EnsureEachLinkExists(sequence);
185         return CreateCore(sequence);
186     });
187 }
188
189 private ulong CreateCore(params ulong[] sequence)
190 {
191     if (Options.UseIndex)
192     {
193         Options.Index.Add(sequence);
194     }
195     var sequenceRoot = default(ulong);
196     if (Options.EnforceSingleSequenceVersionOnWriteBasedOnExisting)
197     {
198         var matches = Each(sequence);
199         if (matches.Count > 0)
200         {
201             sequenceRoot = matches[0];
202         }
203     }
204     else if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew)
205     {
206         return CompactCore(sequence);
207     }
208     if (sequenceRoot == default)
209     {
210         sequenceRoot = Options.LinksToSequenceConverter.Convert(sequence);
211     }
212     if (Options.UseSequenceMarker)
213     {
214         Links.Unsync.CreateAndUpdate(Options.SequenceMarkerLink, sequenceRoot);
215     }
216     return sequenceRoot; // Возвращаем корень последовательности (т.е. сами элементы)
217 }
218
219 #endregion
220
221 #region Each
222
223 public List<ulong> Each(params ulong[] sequence)
224 {
225     var results = new List<ulong>();
226     Each(results.AddAndReturnTrue, sequence);
227     return results;
228 }
229
230 public bool Each(Func<ulong, bool> handler, IList<ulong> sequence)

```

```

231 {
232     return Sync.ExecuteReadOperation(() =>
233     {
234         if (sequence.IsNullOrEmpty())
235         {
236             return true;
237         }
238         Links.EnsureEachLinkIsAnyOrExists(sequence);
239         if (sequence.Count == 1)
240         {
241             var link = sequence[0];
242             if (link == _constants.Any)
243             {
244                 return Links.Unsync.Each(_constants.Any, _constants.Any, handler);
245             }
246             return handler(link);
247         }
248         if (sequence.Count == 2)
249         {
250             return Links.Unsync.Each(sequence[0], sequence[1], handler);
251         }
252         if (Options.UseIndex && !Options.Index.MightContain(sequence))
253         {
254             return false;
255         }
256         return EachCore(handler, sequence);
257     });
258 }
259
260 private bool EachCore(Func<ulong, bool> handler, IList<ulong> sequence)
261 {
262     var matcher = new Matcher(this, sequence, new HashSet<LinkIndex>(), handler);
263     // TODO: Find out why matcher.HandleFullMatched executed twice for the same sequence
264     ↪ Id.
265     Func<ulong, bool> innerHandler = Options.UseSequenceMarker ? (Func<ulong,
266     ↪ bool>)matcher.HandleFullMatchedSequence : matcher.HandleFullMatched;
267     //if (sequence.Length >= 2)
268     if (!StepRight(innerHandler, sequence[0], sequence[1]))
269     {
270         return false;
271     }
272     var last = sequence.Count - 2;
273     for (var i = 1; i < last; i++)
274     {
275         if (!PartialStepRight(innerHandler, sequence[i], sequence[i + 1]))
276         {
277             return false;
278         }
279     }
280     if (sequence.Count >= 3)
281     {
282         if (!StepLeft(innerHandler, sequence[sequence.Count - 2],
283         ↪ sequence[sequence.Count - 1]))
284         {
285             return false;
286         }
287     }
288     return true;
289 }
290
291 private bool PartialStepRight(Func<ulong, bool> handler, ulong left, ulong right)
292 {
293     return Links.Unsync.Each(_constants.Any, left, doublet =>
294     {
295         if (!StepRight(handler, doublet, right))
296         {
297             return false;
298         }
299         if (left != doublet)
300         {
301             return PartialStepRight(handler, doublet, right);
302         }
303         return true;
304     });
305 }

```

```

304 private bool StepRight(Func<ulong, bool> handler, ulong left, ulong right) =>
305     ↳ Links.Unsync.Each(left, _constants.Any, rightStep => TryStepRightUp(handler, right,
306     ↳ rightStep));
307
308 private bool TryStepRightUp(Func<ulong, bool> handler, ulong right, ulong stepFrom)
309 {
310     var upStep = stepFrom;
311     var firstSource = Links.Unsync.GetTarget(upStep);
312     while (firstSource != right && firstSource != upStep)
313     {
314         upStep = firstSource;
315         firstSource = Links.Unsync.GetSource(upStep);
316     }
317     if (firstSource == right)
318     {
319         return handler(stepFrom);
320     }
321     return true;
322 }
323
324 private bool StepLeft(Func<ulong, bool> handler, ulong left, ulong right) =>
325     ↳ Links.Unsync.Each(_constants.Any, right, leftStep => TryStepLeftUp(handler, left,
326     ↳ leftStep));
327
328 private bool TryStepLeftUp(Func<ulong, bool> handler, ulong left, ulong stepFrom)
329 {
330     var upStep = stepFrom;
331     var firstTarget = Links.Unsync.GetSource(upStep);
332     while (firstTarget != left && firstTarget != upStep)
333     {
334         upStep = firstTarget;
335         firstTarget = Links.Unsync.GetTarget(upStep);
336     }
337     if (firstTarget == left)
338     {
339         return handler(stepFrom);
340     }
341     return true;
342 }
343
344 #endregion
345
346 #region Update
347
348 public ulong Update(ulong[] sequence, ulong[] newSequence)
349 {
350     if (sequence.IsNullOrEmpty() && newSequence.IsNullOrEmpty())
351     {
352         return _constants.Null;
353     }
354     if (sequence.IsNullOrEmpty())
355     {
356         return Create(newSequence);
357     }
358     if (newSequence.IsNullOrEmpty())
359     {
360         Delete(sequence);
361         return _constants.Null;
362     }
363     return Sync.ExecuteWriteOperation(() =>
364     {
365         Links.EnsureEachLinkIsAnyOrExists(sequence);
366         Links.EnsureEachLinkExists(newSequence);
367         return UpdateCore(sequence, newSequence);
368     });
369 }
370
371 private ulong UpdateCore(ulong[] sequence, ulong[] newSequence)
372 {
373     ulong bestVariant;
374     if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew &&
375         ↳ !sequence.EqualTo(newSequence))
376     {
377         bestVariant = CompactCore(newSequence);
378     }
379     else
380     {
381         bestVariant = CreateCore(newSequence);
382     }
383 }

```

```

378 // TODO: Check all options only ones before loop execution
379 // Возможно нужно две версии Each, возвращающий фактические последовательности и с
    ↳ маркером,
380 // или возможно даже возвращать и тот и тот вариант. С другой стороны все варианты
    ↳ можно получить имея только фактические последовательности.
381 foreach (var variant in Each(sequence))
382 {
383     if (variant != bestVariant)
384     {
385         UpdateOneCore(variant, bestVariant);
386     }
387 }
388 return bestVariant;
389 }
390
391 private void UpdateOneCore(ulong sequence, ulong newSequence)
392 {
393     if (Options.UseGarbageCollection)
394     {
395         var sequenceElements = GetSequenceElements(sequence);
396         var sequenceElementsContents = new UInt64Link(Links.GetLink(sequenceElements));
397         var sequenceLink = GetSequenceByElements(sequenceElements);
398         var newSequenceElements = GetSequenceElements(newSequence);
399         var newSequenceLink = GetSequenceByElements(newSequenceElements);
400         if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
401         {
402             if (sequenceLink != _constants.Null)
403             {
404                 Links.Unsync.MergeUsages(sequenceLink, newSequenceLink);
405             }
406             Links.Unsync.MergeUsages(sequenceElements, newSequenceElements);
407         }
408         ClearGarbage(sequenceElementsContents.Source);
409         ClearGarbage(sequenceElementsContents.Target);
410     }
411     else
412     {
413         if (Options.UseSequenceMarker)
414         {
415             var sequenceElements = GetSequenceElements(sequence);
416             var sequenceLink = GetSequenceByElements(sequenceElements);
417             var newSequenceElements = GetSequenceElements(newSequence);
418             var newSequenceLink = GetSequenceByElements(newSequenceElements);
419             if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
420             {
421                 if (sequenceLink != _constants.Null)
422                 {
423                     Links.Unsync.MergeUsages(sequenceLink, newSequenceLink);
424                 }
425                 Links.Unsync.MergeUsages(sequenceElements, newSequenceElements);
426             }
427         }
428         else
429         {
430             if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
431             {
432                 Links.Unsync.MergeUsages(sequence, newSequence);
433             }
434         }
435     }
436 }
437
438 #endregion
439
440 #region Delete
441
442 public void Delete(params ulong[] sequence)
443 {
444     Sync.ExecuteWriteOperation(() =>
445     {
446         // TODO: Check all options only ones before loop execution
447         foreach (var linkToDelete in Each(sequence))
448         {
449             DeleteOneCore(linkToDelete);
450         }
451     });
452 }
453

```

```

454 private void DeleteOneCore(ulong link)
455 {
456     if (Options.UseGarbageCollection)
457     {
458         var sequenceElements = GetSequenceElements(link);
459         var sequenceElementsContents = new UInt64Link(Links.GetLink(sequenceElements));
460         var sequenceLink = GetSequenceByElements(sequenceElements);
461         if (Options.UseCascadeDelete || CountUsages(link) == 0)
462         {
463             if (sequenceLink != _constants.Null)
464             {
465                 Links.Unsync.Delete(sequenceLink);
466             }
467             Links.Unsync.Delete(link);
468         }
469         ClearGarbage(sequenceElementsContents.Source);
470         ClearGarbage(sequenceElementsContents.Target);
471     }
472     else
473     {
474         if (Options.UseSequenceMarker)
475         {
476             var sequenceElements = GetSequenceElements(link);
477             var sequenceLink = GetSequenceByElements(sequenceElements);
478             if (Options.UseCascadeDelete || CountUsages(link) == 0)
479             {
480                 if (sequenceLink != _constants.Null)
481                 {
482                     Links.Unsync.Delete(sequenceLink);
483                 }
484                 Links.Unsync.Delete(link);
485             }
486         }
487         else
488         {
489             if (Options.UseCascadeDelete || CountUsages(link) == 0)
490             {
491                 Links.Unsync.Delete(link);
492             }
493         }
494     }
495 }
496
497 #endregion
498
499 #region Compactification
500
501 /// <remarks>
502 /// bestVariant можно выбирать по максимальному числу использований,
503 /// но сбалансированный позволяет гарантировать уникальность (если есть возможность,
504 /// гарантировать его использование в других местах).
505 ///
506 /// Получается этот метод должен игнорировать Options.EnforceSingleSequenceVersionOnWrite
507 /// </remarks>
508 public ulong Compact(params ulong[] sequence)
509 {
510     return Sync.ExecuteWriteOperation(() =>
511     {
512         if (sequence.IsNullOrEmpty())
513         {
514             return _constants.Null;
515         }
516         Links.EnsureEachLinkExists(sequence);
517         return CompactCore(sequence);
518     });
519 }
520
521 [MethodImpl(MethodImplOptions.AggressiveInlining)]
522 private ulong CompactCore(params ulong[] sequence) => UpdateCore(sequence, sequence);
523
524 #endregion
525
526 #region Garbage Collection
527
528 /// <remarks>
529 /// TODO: Добавить дополнительный обработчик / событие CanBeDeleted которое можно
530 ///     ↪ определить извне или в унаследованном классе
531 /// </remarks>
532 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

532 private bool IsGarbage(ulong link) => link != Options.SequenceMarkerLink &&
    ↳ !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,
    ↳ HashSet<LinkIndex> results, Func<LinkIndex, bool> stopableHandler,
    ↳ HashSet<LinkIndex> readAsElements = null)
    : base(sequences.Links.Unsync, new DefaultStack<ulong>())
576     {
577         _sequences = sequences;
578         _patternSequence = patternSequence;
579         _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
    ↳ _constants.Any && x != ZeroOrMany));
580         _results = results;
581         _stopableHandler = stopableHandler;
582         _readAsElements = readAsElements;
583     }
584
585     protected override bool IsElement(ulong link) => base.IsElement(link) ||
    ↳ (_readAsElements != null && _readAsElements.Contains(link)) ||
    ↳ _linksInSequence.Contains(link);
586
587     public bool FullMatch(LinkIndex sequenceToMatch)
588     {
589         _filterPosition = 0;
590         foreach (var part in Walk(sequenceToMatch))
591         {
592             if (!FullMatchCore(part))
593             {
594                 break;
595             }
596         }
597         return _filterPosition == _patternSequence.Count;
598     }
599
600     private bool FullMatchCore(LinkIndex element)
601     {
602         if (_filterPosition == _patternSequence.Count)
603         {
604             _filterPosition = -2; // Длиннее чем нужно
605         }
606     }

```



```

606         return false;
607     }
608     if (_patternSequence[_filterPosition] != _constants.Any
609         && element != _patternSequence[_filterPosition])
610     {
611         _filterPosition = -1;
612         return false; // Начинается/Продолжается иначе
613     }
614     _filterPosition++;
615     return true;
616 }
617
618 public void AddFullMatchedToResults(ulong sequenceToMatch)
619 {
620     if (FullMatch(sequenceToMatch))
621     {
622         _results.Add(sequenceToMatch);
623     }
624 }
625
626 public bool HandleFullMatched(ulong sequenceToMatch)
627 {
628     if (FullMatch(sequenceToMatch) && _results.Add(sequenceToMatch))
629     {
630         return _stopableHandler(sequenceToMatch);
631     }
632     return true;
633 }
634
635 public bool HandleFullMatchedSequence(ulong sequenceToMatch)
636 {
637     var sequence = _sequences.GetSequenceByElements(sequenceToMatch);
638     if (sequence != _constants.Null && FullMatch(sequenceToMatch) &&
639         ↪ _results.Add(sequenceToMatch))
640     {
641         return _stopableHandler(sequence);
642     }
643     return true;
644 }
645
646 /// <remarks>
647 /// TODO: Add support for LinksConstants.Any
648 /// </remarks>
649 public bool PartialMatch(LinkIndex sequenceToMatch)
650 {
651     _filterPosition = -1;
652     foreach (var part in Walk(sequenceToMatch))
653     {
654         if (!PartialMatchCore(part))
655         {
656             break;
657         }
658     }
659     return _filterPosition == _patternSequence.Count - 1;
660 }
661
662 private bool PartialMatchCore(LinkIndex element)
663 {
664     if (_filterPosition == (_patternSequence.Count - 1))
665     {
666         return false; // Нашлось
667     }
668     if (_filterPosition >= 0)
669     {
670         if (element == _patternSequence[_filterPosition + 1])
671         {
672             _filterPosition++;
673         }
674         else
675         {
676             _filterPosition = -1;
677         }
678     }
679     if (_filterPosition < 0)
680     {
681         if (element == _patternSequence[0])
682         {
683             _filterPosition = 0;
684         }
685     }
686 }

```

```

684     }
685     return true; // Ищем дальше
686 }
687
688 public void AddPartialMatchedToResults(ulong sequenceToMatch)
689 {
690     if (PartialMatch(sequenceToMatch))
691     {
692         _results.Add(sequenceToMatch);
693     }
694 }
695
696 public bool HandlePartialMatched(ulong sequenceToMatch)
697 {
698     if (PartialMatch(sequenceToMatch))
699     {
700         return _stopableHandler(sequenceToMatch);
701     }
702     return true;
703 }
704
705 public void AddAllPartialMatchedToResults(IEnumerable<ulong> sequencesToMatch)
706 {
707     foreach (var sequenceToMatch in sequencesToMatch)
708     {
709         if (PartialMatch(sequenceToMatch))
710         {
711             _results.Add(sequenceToMatch);
712         }
713     }
714 }
715
716 public void AddAllPartialMatchedToResultsAndReadAsElements(IEnumerable<ulong>
↵ sequencesToMatch)
717 {
718     foreach (var sequenceToMatch in sequencesToMatch)
719     {
720         if (PartialMatch(sequenceToMatch))
721         {
722             _readAsElements.Add(sequenceToMatch);
723             _results.Add(sequenceToMatch);
724         }
725     }
726 }
727 }
728
729 #endregion
730 }
731 }

```

./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 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
15
16 namespace Platform.Data.Doublets.Sequences
17 {
18     partial class Sequences
19     {
20         #region Create All Variants (Not Practical)
21
22         /// <remarks>
23         /// Number of links that is needed to generate all variants for
24         /// sequence of length N corresponds to https://oeis.org/A014143/list sequence.
25         /// </remarks>
26         public ulong[] CreateAllVariants2(ulong[] sequence)
27         {
28             return Sync.ExecuteWriteOperation(() =>
29             {

```

```

30         if (sequence.IsNullOrEmpty())
31         {
32             return new ulong[0];
33         }
34         Links.EnsureEachLinkExists(sequence);
35         if (sequence.Length == 1)
36         {
37             return sequence;
38         }
39         return CreateAllVariants2Core(sequence, 0, sequence.Length - 1);
40     });
41 }
42
43 private ulong[] CreateAllVariants2Core(ulong[] sequence, long startAt, long stopAt)
44 {
45     #if DEBUG
46         if ((stopAt - startAt) < 0)
47         {
48             throw new ArgumentOutOfRangeException(nameof(startAt), "startAt должен быть
49                 ↳ меньше или равен stopAt");
50         }
51         #endif
52         if ((stopAt - startAt) == 0)
53         {
54             return new[] { sequence[startAt] };
55         }
56         if ((stopAt - startAt) == 1)
57         {
58             return new[] { Links.Unsync.CreateAndUpdate(sequence[startAt], sequence[stopAt])
59                 ↳ };
60         }
61         var variants = new ulong[(ulong)Platform.Numbers.Math.Catalan(stopAt - startAt)];
62         var last = 0;
63         for (var splitter = startAt; splitter < stopAt; splitter++)
64         {
65             var left = CreateAllVariants2Core(sequence, startAt, splitter);
66             var right = CreateAllVariants2Core(sequence, splitter + 1, stopAt);
67             for (var i = 0; i < left.Length; i++)
68             {
69                 for (var j = 0; j < right.Length; j++)
70                 {
71                     var variant = Links.Unsync.CreateAndUpdate(left[i], right[j]);
72                     if (variant == _constants.Null)
73                     {
74                         throw new NotImplementedException("Creation cancellation is not
75                             ↳ implemented.");
76                     }
77                     variants[last++] = variant;
78                 }
79             }
80         }
81         return variants;
82     }
83
84     public List<ulong> CreateAllVariants1(params ulong[] sequence)
85     {
86         return Sync.ExecuteWriteOperation(() =>
87         {
88             if (sequence.IsNullOrEmpty())
89             {
90                 return new List<ulong>();
91             }
92             Links.Unsync.EnsureEachLinkExists(sequence);
93             if (sequence.Length == 1)
94             {
95                 return new List<ulong> { sequence[0] };
96             }
97             var results = new
98                 ↳ List<ulong>((int)Platform.Numbers.Math.Catalan(sequence.Length));
99             return CreateAllVariants1Core(sequence, results);
100         });
101     }
102
103     private List<ulong> CreateAllVariants1Core(ulong[] sequence, List<ulong> results)
104     {
105         if (sequence.Length == 2)
106         {
107             var link = Links.Unsync.CreateAndUpdate(sequence[0], sequence[1]);

```

```

104         if (link == _constants.Null)
105         {
106             throw new NotImplementedException("Creation cancellation is not
                ↳ implemented.");
107         }
108         results.Add(link);
109         return results;
110     }
111     var innerSequenceLength = sequence.Length - 1;
112     var innerSequence = new ulong[innerSequenceLength];
113     for (var li = 0; li < innerSequenceLength; li++)
114     {
115         var link = Links.Unsync.CreateAndUpdate(sequence[li], sequence[li + 1]);
116         if (link == _constants.Null)
117         {
118             throw new NotImplementedException("Creation cancellation is not
                ↳ 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                     for (var isi = 0; isi < linkIndex; isi++)
175                     {
176                         innerSequence[isi] = sequence[isi];
177                     }
178                     for (var isi = linkIndex + 1; isi < innerSequenceLength; isi++)
179                     {

```

```

180         innerSequence[isi] = sequence[isi + 1];
181     }
182 }
183     innerSequence[linkIndex] = doublet;
184     Each1(handler, innerSequence);
185     return _constants.Continue;
186 });
187 }
188 }
189 }
190
191 public HashSet<ulong> EachPart(params ulong[] sequence)
192 {
193     var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
194     EachPartCore(link =>
195     {
196         if (!visitedLinks.Contains(link))
197         {
198             visitedLinks.Add(link); // изучить почему случаются повторы
199         }
200         return true;
201     }, sequence);
202     return visitedLinks;
203 }
204
205 public void EachPart(Func<ulong, bool> handler, params ulong[] sequence)
206 {
207     var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
208     EachPartCore(link =>
209     {
210         if (!visitedLinks.Contains(link))
211         {
212             visitedLinks.Add(link); // изучить почему случаются повторы
213             return handler(link);
214         }
215         return true;
216     }, sequence);
217 }
218
219 private void EachPartCore(Func<ulong, bool> handler, params ulong[] sequence)
220 {
221     if (sequence.IsNullOrEmpty())
222     {
223         return;
224     }
225     Links.EnsureEachLinkIsAnyOrExists(sequence);
226     if (sequence.Length == 1)
227     {
228         var link = sequence[0];
229         if (link > 0)
230         {
231             handler(link);
232         }
233         else
234         {
235             Links.Each(_constants.Any, _constants.Any, handler);
236         }
237     }
238     else if (sequence.Length == 2)
239     {
240         // _links.Each(sequence[0], sequence[1], handler);
241         // o_|      x_o ...
242         // x_|      |___|
243         Links.Each(sequence[1], _constants.Any, doublet =>
244         {
245             var match = Links.SearchOrDefault(sequence[0], doublet);
246             if (match != _constants.Null)
247             {
248                 handler(match);
249             }
250             return true;
251         });
252         // |_x      ... x_o
253         // |_o      |___|
254         Links.Each(_constants.Any, sequence[0], doublet =>
255         {
256             var match = Links.SearchOrDefault(doublet, sequence[1]);
257             if (match != 0)

```

```

258         {
259             handler(match);
260         }
261         return true;
262     });
263     //          .-x o-.
264     //          |___|
265     PartialStepRight(x => handler(x), sequence[0], sequence[1]);
266 }
267 else
268 {
269     // TODO: Implement other variants
270     return;
271 }
272 }
273
274 private void PartialStepRight(Action<ulong> handler, ulong left, ulong right)
275 {
276     Links.Unsync.Each(_constants.Any, left, doublet =>
277     {
278         StepRight(handler, doublet, right);
279         if (left != doublet)
280         {
281             PartialStepRight(handler, doublet, right);
282         }
283         return true;
284     });
285 }
286
287 private void StepRight(Action<ulong> handler, ulong left, ulong right)
288 {
289     Links.Unsync.Each(left, _constants.Any, rightStep =>
290     {
291         TryStepRightUp(handler, right, rightStep);
292         return true;
293     });
294 }
295
296 private void TryStepRightUp(Action<ulong> handler, ulong right, ulong stepFrom)
297 {
298     var upStep = stepFrom;
299     var firstSource = Links.Unsync.GetTarget(upStep);
300     while (firstSource != right && firstSource != upStep)
301     {
302         upStep = firstSource;
303         firstSource = Links.Unsync.GetSource(upStep);
304     }
305     if (firstSource == right)
306     {
307         handler(stepFrom);
308     }
309 }
310
311 // TODO: Test
312 private void PartialStepLeft(Action<ulong> handler, ulong left, ulong right)
313 {
314     Links.Unsync.Each(right, _constants.Any, doublet =>
315     {
316         StepLeft(handler, left, doublet);
317         if (right != doublet)
318         {
319             PartialStepLeft(handler, left, doublet);
320         }
321         return true;
322     });
323 }
324
325 private void StepLeft(Action<ulong> handler, ulong left, ulong right)
326 {
327     Links.Unsync.Each(_constants.Any, right, leftStep =>
328     {
329         TryStepLeftUp(handler, left, leftStep);
330         return true;
331     });
332 }
333
334 private void TryStepLeftUp(Action<ulong> handler, ulong left, ulong stepFrom)
335 {
336     var upStep = stepFrom;

```

```

337     var firstTarget = Links.Unsync.GetSource(upStep);
338     while (firstTarget != left && firstTarget != upStep)
339     {
340         upStep = firstTarget;
341         firstTarget = Links.Unsync.GetTarget(upStep);
342     }
343     if (firstTarget == left)
344     {
345         handler(stepFrom);
346     }
347 }
348
349 private bool StartsWith(ulong sequence, ulong link)
350 {
351     var upStep = sequence;
352     var firstSource = Links.Unsync.GetSource(upStep);
353     while (firstSource != link && firstSource != upStep)
354     {
355         upStep = firstSource;
356         firstSource = Links.Unsync.GetSource(upStep);
357     }
358     return firstSource == link;
359 }
360
361 private bool EndsWith(ulong sequence, ulong link)
362 {
363     var upStep = sequence;
364     var lastTarget = Links.Unsync.GetTarget(upStep);
365     while (lastTarget != link && lastTarget != upStep)
366     {
367         upStep = lastTarget;
368         lastTarget = Links.Unsync.GetTarget(upStep);
369     }
370     return lastTarget == link;
371 }
372
373 public List<ulong> GetAllMatchingSequences0(params ulong[] sequence)
374 {
375     return Sync.ExecuteReadOperation(() =>
376     {
377         var results = new List<ulong>();
378         if (sequence.Length > 0)
379         {
380             Links.EnsureEachLinkExists(sequence);
381             var firstElement = sequence[0];
382             if (sequence.Length == 1)
383             {
384                 results.Add(firstElement);
385                 return results;
386             }
387             if (sequence.Length == 2)
388             {
389                 var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
390                 if (doublet != _constants.Null)
391                 {
392                     results.Add(doublet);
393                 }
394                 return results;
395             }
396             var linksInSequence = new HashSet<ulong>(sequence);
397             void handler(ulong result)
398             {
399                 var filterPosition = 0;
400                 StoppableSequenceWalker.WalkRight(result, Links.Unsync.GetSource,
401                 ↪ Links.Unsync.GetTarget,
402                 ↪ x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x,
403                 ↪ x =>
404                 {
405                     if (filterPosition == sequence.Length)
406                     {
407                         filterPosition = -2; // Длиннее чем нужно
408                         return false;
409                     }
410                     if (x != sequence[filterPosition])
411                     {
412                         filterPosition = -1;
413                         return false; // Начинается иначе
414                     }
415                     filterPosition++;
416                 }
417             }
418         }
419     });
420 }

```

```

414         return true;
415     });
416     if (filterPosition == sequence.Length)
417     {
418         results.Add(result);
419     }
420 }
421 if (sequence.Length >= 2)
422 {
423     StepRight(handler, sequence[0], sequence[1]);
424 }
425 var last = sequence.Length - 2;
426 for (var i = 1; i < last; i++)
427 {
428     PartialStepRight(handler, sequence[i], sequence[i + 1]);
429 }
430 if (sequence.Length >= 3)
431 {
432     StepLeft(handler, sequence[sequence.Length - 2],
433         ↪ sequence[sequence.Length - 1]);
434 }
435 }
436 return results;
437 });
438 }
439
440 public HashSet<ulong> GetAllMatchingSequences1(params ulong[] sequence)
441 {
442     return Sync.ExecuteReadOperation(() =>
443     {
444         var results = new HashSet<ulong>();
445         if (sequence.Length > 0)
446         {
447             Links.EnsureEachLinkExists(sequence);
448             var firstElement = sequence[0];
449             if (sequence.Length == 1)
450             {
451                 results.Add(firstElement);
452                 return results;
453             }
454             if (sequence.Length == 2)
455             {
456                 var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
457                 if (doublet != _constants.Null)
458                 {
459                     results.Add(doublet);
460                 }
461                 return results;
462             }
463             var matcher = new Matcher(this, sequence, results, null);
464             if (sequence.Length >= 2)
465             {
466                 StepRight(matcher.AddFullMatchedToResults, sequence[0], sequence[1]);
467             }
468             var last = sequence.Length - 2;
469             for (var i = 1; i < last; i++)
470             {
471                 PartialStepRight(matcher.AddFullMatchedToResults, sequence[i],
472                     ↪ sequence[i + 1]);
473             }
474             if (sequence.Length >= 3)
475             {
476                 StepLeft(matcher.AddFullMatchedToResults, sequence[sequence.Length - 2],
477                     ↪ sequence[sequence.Length - 1]);
478             }
479         }
480         return results;
481     });
482 }
483
484 public const int MaxSequenceFormatSize = 200;
485
486 public string FormatSequence(LinkIndex sequenceLink, params LinkIndex[] knownElements)
487     ↪ => FormatSequence(sequenceLink, (sb, x) => sb.Append(x), true, knownElements);

```



```

486 public string FormatSequence(LinkIndex sequenceLink, Action<StringBuilder, LinkIndex>
    ↳ elementToString, bool insertComma, params LinkIndex[] knownElements) =>
    ↳ Links.SyncRoot.ExecuteReadOperation(() => FormatSequence(Links.Unsync, sequenceLink,
    ↳ elementToString, insertComma, knownElements));
487
488 private string FormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
    ↳ Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
    ↳ LinkIndex[] knownElements)
489 {
490     var linksInSequence = new HashSet<ulong>(knownElements);
491     //var entered = new HashSet<ulong>();
492     var sb = new StringBuilder();
493     sb.Append('{');
494     if (links.Exists(sequenceLink))
495     {
496         StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
497             x => linksInSequence.Contains(x) || links.IsPartialPoint(x), element => //
    ↳ entered.AddAndReturnVoid, x => { }, entered.DoNotContains
498         {
499             if (insertComma && sb.Length > 1)
500             {
501                 sb.Append(',');
502             }
503             //if (entered.Contains(element))
504             //{
505             //    sb.Append('{');
506             //    elementToString(sb, element);
507             //    sb.Append('}');
508             //}
509             //else
510             elementToString(sb, element);
511             if (sb.Length < MaxSequenceFormatSize)
512             {
513                 return true;
514             }
515             sb.Append(insertComma ? ", ..." : "...");
516             return false;
517         });
518     }
519     sb.Append('}');
520     return sb.ToString();
521 }
522
523 public string SafeFormatSequence(LinkIndex sequenceLink, params LinkIndex[]
    ↳ knownElements) => SafeFormatSequence(sequenceLink, (sb, x) => sb.Append(x), true,
    ↳ knownElements);
524
525 public string SafeFormatSequence(LinkIndex sequenceLink, Action<StringBuilder,
    ↳ LinkIndex> elementToString, bool insertComma, params LinkIndex[] knownElements) =>
    ↳ Links.SyncRoot.ExecuteReadOperation(() => SafeFormatSequence(Links.Unsync,
    ↳ sequenceLink, elementToString, insertComma, knownElements));
526
527 private string SafeFormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
    ↳ Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
    ↳ LinkIndex[] knownElements)
528 {
529     var linksInSequence = new HashSet<ulong>(knownElements);
530     var entered = new HashSet<ulong>();
531     var sb = new StringBuilder();
532     sb.Append('{');
533     if (links.Exists(sequenceLink))
534     {
535         StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
536             x => linksInSequence.Contains(x) || links.IsFullPoint(x),
    ↳ entered.AddAndReturnVoid, x => { }, entered.DoNotContains, element =>
537         {
538             if (insertComma && sb.Length > 1)
539             {
540                 sb.Append(',');
541             }
542             if (entered.Contains(element))
543             {
544                 sb.Append('{');
545                 elementToString(sb, element);
546                 sb.Append('}');
547             }
548             else

```

```

549         {
550             elementToString(sb, element);
551         }
552         if (sb.Length < MaxSequenceFormatSize)
553         {
554             return true;
555         }
556         sb.Append(insertComma ? ", ..." : "...");
557         return false;
558     });
559 }
560 sb.Append('}');
561 return sb.ToString();
562 }
563
564 public List<ulong> GetAllPartiallyMatchingSequences0(params ulong[] sequence)
565 {
566     return Sync.ExecuteReadOperation(() =>
567     {
568         if (sequence.Length > 0)
569         {
570             Links.EnsureEachLinkExists(sequence);
571             var results = new HashSet<ulong>();
572             for (var i = 0; i < sequence.Length; i++)
573             {
574                 AllUsagesCore(sequence[i], results);
575             }
576             var filteredResults = new List<ulong>();
577             var linksInSequence = new HashSet<ulong>(sequence);
578             foreach (var result in results)
579             {
580                 var filterPosition = -1;
581                 StopableSequenceWalker.WalkRight(result, Links.Unsync.GetSource,
582                     ↪ Links.Unsync.GetTarget,
583                     ↪ x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x,
584                     {
585                         if (filterPosition == (sequence.Length - 1))
586                         {
587                             return false;
588                         }
589                         if (filterPosition >= 0)
590                         {
591                             if (x == sequence[filterPosition + 1])
592                             {
593                                 filterPosition++;
594                             }
595                             else
596                             {
597                                 return false;
598                             }
599                         }
600                         if (filterPosition < 0)
601                         {
602                             if (x == sequence[0])
603                             {
604                                 filterPosition = 0;
605                             }
606                         }
607                         return true;
608                     }
609                 ));
610                 if (filterPosition == (sequence.Length - 1))
611                 {
612                     filteredResults.Add(result);
613                 }
614             }
615             return filteredResults;
616         }
617         return new List<ulong>();
618     });
619 }
620
621 public HashSet<ulong> GetAllPartiallyMatchingSequences1(params ulong[] sequence)
622 {
623     return Sync.ExecuteReadOperation(() =>
624     {
625         if (sequence.Length > 0)
626         {
627             Links.EnsureEachLinkExists(sequence);

```

```

626         var results = new HashSet<ulong>();
627         for (var i = 0; i < sequence.Length; i++)
628         {
629             AllUsagesCore(sequence[i], results);
630         }
631         var filteredResults = new HashSet<ulong>();
632         var matcher = new Matcher(this, sequence, filteredResults, null);
633         matcher.AddAllPartialMatchedToResults(results);
634         return filteredResults;
635     }
636     return new HashSet<ulong>();
637 });
638 }
639
640 public bool GetAllPartiallyMatchingSequences2(Func<ulong, bool> handler, params ulong[]
    ↪ sequence)
641 {
642     return Sync.ExecuteReadOperation(() =>
643     {
644         if (sequence.Length > 0)
645         {
646             Links.EnsureEachLinkExists(sequence);
647
648             var results = new HashSet<ulong>();
649             var filteredResults = new HashSet<ulong>();
650             var matcher = new Matcher(this, sequence, filteredResults, handler);
651             for (var i = 0; i < sequence.Length; i++)
652             {
653                 if (!AllUsagesCore1(sequence[i], results, matcher.HandlePartialMatched))
654                 {
655                     return false;
656                 }
657             }
658             return true;
659         }
660         return true;
661     });
662 }
663
664 //public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
665 //{
666 //    return Sync.ExecuteReadOperation(() =>
667 //    {
668 //        if (sequence.Length > 0)
669 //        {
670 //            _links.EnsureEachLinkIsAnyOrExists(sequence);
671
672 //            var firstResults = new HashSet<ulong>();
673 //            var lastResults = new HashSet<ulong>();
674
675 //            var first = sequence.First(x => x != LinksConstants.Any);
676 //            var last = sequence.Last(x => x != LinksConstants.Any);
677
678 //            AllUsagesCore(first, firstResults);
679 //            AllUsagesCore(last, lastResults);
680
681 //            firstResults.IntersectWith(lastResults);
682
683 //            //for (var i = 0; i < sequence.Length; i++)
684 //            //    AllUsagesCore(sequence[i], results);
685
686 //            var filteredResults = new HashSet<ulong>();
687 //            var matcher = new Matcher(this, sequence, filteredResults, null);
688 //            matcher.AddAllPartialMatchedToResults(firstResults);
689 //            return filteredResults;
690 //        }
691
692 //        return new HashSet<ulong>();
693 //    });
694 //}
695
696 public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
697 {
698     return Sync.ExecuteReadOperation(() =>
699     {
700         if (sequence.Length > 0)
701         {
702             Links.EnsureEachLinkIsAnyOrExists(sequence);
703             var firstResults = new HashSet<ulong>();

```

```

704     var lastResults = new HashSet<ulong>();
705     var first = sequence.First(x => x != _constants.Any);
706     var last = sequence.Last(x => x != _constants.Any);
707     AllUsagesCore(first, firstResults);
708     AllUsagesCore(last, lastResults);
709     firstResults.IntersectWith(lastResults);
710     //for (var i = 0; i < sequence.Length; i++)
711     //    AllUsagesCore(sequence[i], results);
712     var filteredResults = new HashSet<ulong>();
713     var matcher = new Matcher(this, sequence, filteredResults, null);
714     matcher.AddAllPartialMatchedToResults(firstResults);
715     return filteredResults;
716 }
717 return new HashSet<ulong>();
718 });
719 }
720
721 public HashSet<ulong> GetAllPartiallyMatchingSequences4(HashSet<ulong> readAsElements,
722 ↪ IList<ulong> sequence)
723 {
724     return Sync.ExecuteReadOperation(() =>
725     {
726         if (sequence.Count > 0)
727         {
728             Links.EnsureEachLinkExists(sequence);
729             var results = new HashSet<LinkIndex>();
730             //var nextResults = new HashSet<ulong>();
731             //for (var i = 0; i < sequence.Length; i++)
732             //{
733             //    AllUsagesCore(sequence[i], nextResults);
734             //    if (results.IsNullOrEmpty())
735             //    {
736             //        results = nextResults;
737             //        nextResults = new HashSet<ulong>();
738             //    }
739             //    else
740             //    {
741             //        results.IntersectWith(nextResults);
742             //        nextResults.Clear();
743             //    }
744             //}
745             var collector1 = new AllUsagesCollector1(Links.Unsync, results);
746             collector1.Collect(Links.Unsync.GetLink(sequence[0]));
747             var next = new HashSet<ulong>();
748             for (var i = 1; i < sequence.Count; i++)
749             {
750                 var collector = new AllUsagesCollector1(Links.Unsync, next);
751                 collector.Collect(Links.Unsync.GetLink(sequence[i]));
752
753                 results.IntersectWith(next);
754                 next.Clear();
755             }
756             var filteredResults = new HashSet<ulong>();
757             var matcher = new Matcher(this, sequence, filteredResults, null,
758 ↪ readAsElements);
759             matcher.AddAllPartialMatchedToResultsAndReadAsElements(results.OrderBy(x =>
760 ↪ x)); // OrderBy is a Hack
761             return filteredResults;
762         }
763         return new HashSet<ulong>();
764     });
765 }
766
767 // Does not work
768 public HashSet<ulong> GetAllPartiallyMatchingSequences5(HashSet<ulong> readAsElements,
769 ↪ params ulong[] sequence)
770 {
771     var visited = new HashSet<ulong>();
772     var results = new HashSet<ulong>();
773     var matcher = new Matcher(this, sequence, visited, x => { results.Add(x); return
774 ↪ true; }, readAsElements);
775     var last = sequence.Length - 1;
776     for (var i = 0; i < last; i++)
777     {
778         PartialStepRight(matcher.PartialMatch, sequence[i], sequence[i + 1]);
779     }
780     return results;
781 }

```

```

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

```

```

850         }
851         //for (int i = 0; i < matches[0].Count; i++)
852         //    AllUsagesCore(matches[0][i], usages);
853         //usages.UnionWith(matches[0]);
854         return usages.ToList();
855     }
856     var firstLinkUsages = new HashSet<ulong>();
857     AllUsagesCore(sequence[0], firstLinkUsages);
858     firstLinkUsages.Add(sequence[0]);
859     //var previousMatchings = firstLinkUsages.ToList(); //new List<ulong>() {
860     //    sequence[0] }; // or all sequences, containing this element?
861     //return GetAllPartiallyMatchingSequencesCore(sequence, firstLinkUsages,
862     //    1).ToList();
863     var results = new HashSet<ulong>();
864     foreach (var match in GetAllPartiallyMatchingSequencesCore(sequence,
865     //    firstLinkUsages, 1))
866     {
867         AllUsagesCore(match, results);
868     }
869     return results.ToList();
870 }
871 return new List<ulong>();
872 });
873 }
874
875 /// <remarks>
876 /// TODO: Может потребоваться ограничение на уровень глубины рекурсии
877 /// </remarks>
878 public HashSet<ulong> AllUsages(ulong link)
879 {
880     return Sync.ExecuteReadOperation(() =>
881     {
882         var usages = new HashSet<ulong>();
883         AllUsagesCore(link, usages);
884         return usages;
885     });
886 }
887
888 // При сборе всех использований (последовательностей) можно сохранять обратный путь к
889 // той связи с которой начинался поиск (STTTSSSTT),
890 // причём достаточно одного бита для хранения перехода влево или вправо
891 private void AllUsagesCore(ulong link, HashSet<ulong> usages)
892 {
893     bool handler(ulong doublet)
894     {
895         if (usages.Add(doublet))
896         {
897             AllUsagesCore(doublet, usages);
898         }
899         return true;
900     }
901     Links.Unsync.Each(link, _constants.Any, handler);
902     Links.Unsync.Each(_constants.Any, link, handler);
903 }
904
905 public HashSet<ulong> AllBottomUsages(ulong link)
906 {
907     return Sync.ExecuteReadOperation(() =>
908     {
909         var visits = new HashSet<ulong>();
910         var usages = new HashSet<ulong>();
911         AllBottomUsagesCore(link, visits, usages);
912         return usages;
913     });
914 }
915
916 private void AllBottomUsagesCore(ulong link, HashSet<ulong> visits, HashSet<ulong>
917 //    usages)
918 {
919     bool handler(ulong doublet)
920     {
921         if (visits.Add(doublet))
922         {
923             AllBottomUsagesCore(doublet, visits, usages);
924         }
925         return true;
926     }
927     if (Links.Unsync.Count(_constants.Any, link) == 0)

```

```

923     {
924         usages.Add(link);
925     }
926     else
927     {
928         Links.Unsync.Each(link, _constants.Any, handler);
929         Links.Unsync.Each(_constants.Any, link, handler);
930     }
931 }
932
933 public ulong CalculateTotalSymbolFrequencyCore(ulong symbol)
934 {
935     if (Options.UseSequenceMarker)
936     {
937         var counter = new TotalMarkedSequenceSymbolFrequencyOneOffCounter<ulong>(Links,
938             ↪ Options.MarkedSequenceMatcher, symbol);
939         return counter.Count();
940     }
941     else
942     {
943         var counter = new TotalSequenceSymbolFrequencyOneOffCounter<ulong>(Links,
944             ↪ symbol);
945         return counter.Count();
946     }
947 }
948
949 private bool AllUsagesCore1(ulong link, HashSet<ulong> usages, Func<ulong, bool>
950     ↪ outerHandler)
951 {
952     bool handler(ulong doublet)
953     {
954         if (usages.Add(doublet))
955         {
956             if (!outerHandler(doublet))
957             {
958                 return false;
959             }
960             if (!AllUsagesCore1(doublet, usages, outerHandler))
961             {
962                 return false;
963             }
964         }
965         return true;
966     }
967     return Links.Unsync.Each(link, _constants.Any, handler)
968         && Links.Unsync.Each(_constants.Any, link, handler);
969 }
970
971 public void CalculateAllUsages(ulong[] totals)
972 {
973     var calculator = new AllUsagesCalculator(Links, totals);
974     calculator.Calculate();
975 }
976
977 public void CalculateAllUsages2(ulong[] totals)
978 {
979     var calculator = new AllUsagesCalculator2(Links, totals);
980     calculator.Calculate();
981 }
982
983 private class AllUsagesCalculator
984 {
985     private readonly SynchronizedLinks<ulong> _links;
986     private readonly ulong[] _totals;
987
988     public AllUsagesCalculator(SynchronizedLinks<ulong> links, ulong[] totals)
989     {
990         _links = links;
991         _totals = totals;
992     }
993
994     public void Calculate() => _links.Each(_constants.Any, _constants.Any,
995         ↪ CalculateCore);
996
997     private bool CalculateCore(ulong link)
998     {
999         if (_totals[link] == 0)
1000         {
1001             var total = 1UL;

```

```

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

```



```

1075         // Обработка элемента
1076         if (isElement(target))
1077         {
1078             visitLeaf(target);
1079         }
1080         if (isElement(source))
1081         {
1082             visitLeaf(source);
1083         }
1084         element = source;
1085     }
1086     else
1087     {
1088         stack.Push(element);
1089         visitNode(element);
1090         element = getTarget(element);
1091     }
1092 }
1093 }
1094 _totals[link]++;
1095 return true;
1096 }
1097 }
1098
1099 private class AllUsagesCollector
1100 {
1101     private readonly ILinks<ulong> _links;
1102     private readonly HashSet<ulong> _usages;
1103
1104     public AllUsagesCollector(ILinks<ulong> links, HashSet<ulong> usages)
1105     {
1106         _links = links;
1107         _usages = usages;
1108     }
1109
1110     public bool Collect(ulong link)
1111     {
1112         if (_usages.Add(link))
1113         {
1114             _links.Each(link, _constants.Any, Collect);
1115             _links.Each(_constants.Any, link, Collect);
1116         }
1117         return true;
1118     }
1119 }
1120
1121 private class AllUsagesCollector1
1122 {
1123     private readonly ILinks<ulong> _links;
1124     private readonly HashSet<ulong> _usages;
1125     private readonly ulong _continue;
1126
1127     public AllUsagesCollector1(ILinks<ulong> links, HashSet<ulong> usages)
1128     {
1129         _links = links;
1130         _usages = usages;
1131         _continue = _links.Constants.Continue;
1132     }
1133
1134     public ulong Collect(IList<ulong> link)
1135     {
1136         var linkIndex = _links.GetIndex(link);
1137         if (_usages.Add(linkIndex))
1138         {
1139             _links.Each(Collect, _constants.Any, linkIndex);
1140         }
1141         return _continue;
1142     }
1143 }
1144
1145 private class AllUsagesCollector2
1146 {
1147     private readonly ILinks<ulong> _links;
1148     private readonly BitString _usages;
1149
1150     public AllUsagesCollector2(ILinks<ulong> links, BitString usages)
1151     {
1152         _links = links;
1153         _usages = usages;
1154     }

```

```

1155     public bool Collect(ulong link)
1156     {
1157         if (_usages.Add((long)link))
1158         {
1159             _links.Each(link, _constants.Any, Collect);
1160             _links.Each(_constants.Any, link, Collect);
1161         }
1162         return true;
1163     }
1164 }
1165
1166 private class AllUsagesIntersectingCollector
1167 {
1168     private readonly SynchronizedLinks<ulong> _links;
1169     private readonly HashSet<ulong> _intersectWith;
1170     private readonly HashSet<ulong> _usages;
1171     private readonly HashSet<ulong> _enter;
1172
1173     public AllUsagesIntersectingCollector(SynchronizedLinks<ulong> links, HashSet<ulong>
1174     ↪ intersectWith, HashSet<ulong> usages)
1175     {
1176         _links = links;
1177         _intersectWith = intersectWith;
1178         _usages = usages;
1179         _enter = new HashSet<ulong>(); // защита от зацикливания
1180     }
1181
1182     public bool Collect(ulong link)
1183     {
1184         if (_enter.Add(link))
1185         {
1186             if (_intersectWith.Contains(link))
1187             {
1188                 _usages.Add(link);
1189             }
1190             _links.Unsync.Each(link, _constants.Any, Collect);
1191             _links.Unsync.Each(_constants.Any, link, Collect);
1192         }
1193         return true;
1194     }
1195 }
1196
1197 private void CloseInnerConnections(Action<ulong> handler, ulong left, ulong right)
1198 {
1199     TryStepLeftUp(handler, left, right);
1200     TryStepRightUp(handler, right, left);
1201 }
1202
1203 private void AllCloseConnections(Action<ulong> handler, ulong left, ulong right)
1204 {
1205     // Direct
1206     if (left == right)
1207     {
1208         handler(left);
1209     }
1210     var doublet = Links.Unsync.SearchOrDefault(left, right);
1211     if (doublet != _constants.Null)
1212     {
1213         handler(doublet);
1214     }
1215     // Inner
1216     CloseInnerConnections(handler, left, right);
1217     // Outer
1218     StepLeft(handler, left, right);
1219     StepRight(handler, left, right);
1220     PartialStepRight(handler, left, right);
1221     PartialStepLeft(handler, left, right);
1222 }
1223
1224 private HashSet<ulong> GetAllPartiallyMatchingSequencesCore(ulong[] sequence,
1225     ↪ HashSet<ulong> previousMatchings, long startAt)
1226 {
1227     if (startAt >= sequence.Length) // ?
1228     {
1229         return previousMatchings;
1230     }
1231     var secondLinkUsages = new HashSet<ulong>();
1232     AllUsagesCore(sequence[startAt], secondLinkUsages);

```

```

1232 secondLinkUsages.Add(sequence[startAt]);
1233 var matchings = new HashSet<ulong>();
1234 //for (var i = 0; i < previousMatchings.Count; i++)
1235 foreach (var secondLinkUsage in secondLinkUsages)
1236 {
1237     foreach (var previousMatching in previousMatchings)
1238     {
1239         //AllCloseConnections(matchings.AddAndReturnVoid, previousMatching,
1240         ↪ secondLinkUsage);
1241         StepRight(matchings.AddAndReturnVoid, previousMatching, secondLinkUsage);
1242         TryStepRightUp(matchings.AddAndReturnVoid, secondLinkUsage,
1243         ↪ previousMatching);
1244         //PartialStepRight(matchings.AddAndReturnVoid, secondLinkUsage,
1245         ↪ sequence[startAt]); // почему-то эта ошибочная запись приводит к
1246         ↪ желаемым результатам.
1247         PartialStepRight(matchings.AddAndReturnVoid, previousMatching,
1248         ↪ secondLinkUsage);
1249     }
1250 }
1251 if (matchings.Count == 0)
1252 {
1253     return matchings;
1254 }
1255 return GetAllPartiallyMatchingSequencesCore(sequence, matchings, startAt + 1); // ??
1256 }
1257
1258 private static void EnsureEachLinkIsAnyOrZeroOrManyOrExists(SynchronizedLinks<ulong>
1259 ↪ links, params ulong[] sequence)
1260 {
1261     if (sequence == null)
1262     {
1263         return;
1264     }
1265     for (var i = 0; i < sequence.Length; i++)
1266     {
1267         if (sequence[i] != _constants.Any && sequence[i] != ZeroOrMany &&
1268         ↪ !links.Exists(sequence[i]))
1269         {
1270             throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
1271             ↪ $"patternSequence[{i}]");
1272         }
1273     }
1274 }
1275
1276 // Pattern Matching -> Key To Triggers
1277 public HashSet<ulong> MatchPattern(params ulong[] patternSequence)
1278 {
1279     return Sync.ExecuteReadOperation(() =>
1280     {
1281         patternSequence = Simplify(patternSequence);
1282         if (patternSequence.Length > 0)
1283         {
1284             EnsureEachLinkIsAnyOrZeroOrManyOrExists(Links, patternSequence);
1285             var uniqueSequenceElements = new HashSet<ulong>();
1286             for (var i = 0; i < patternSequence.Length; i++)
1287             {
1288                 if (patternSequence[i] != _constants.Any && patternSequence[i] !=
1289                 ↪ ZeroOrMany)
1290                 {
1291                     uniqueSequenceElements.Add(patternSequence[i]);
1292                 }
1293             }
1294             var results = new HashSet<ulong>();
1295             foreach (var uniqueSequenceElement in uniqueSequenceElements)
1296             {
1297                 AllUsagesCore(uniqueSequenceElement, results);
1298             }
1299             var filteredResults = new HashSet<ulong>();
1300             var matcher = new PatternMatcher(this, patternSequence, filteredResults);
1301             matcher.AddAllPatternMatchedToResults(results);
1302             return filteredResults;
1303         }
1304         return new HashSet<ulong>();
1305     });
1306 }
1307
1308 // Найти все возможные связи между указанным списком связей.

```

```

1300 // Находит связи между всеми указанными связями в любом порядке.
1301 // TODO: решить что делать с повторами (когда одни и те же элементы встречаются
1302 //        → несколько раз в последовательности)
1303 public HashSet<ulong> GetAllConnections(params ulong[] linksToConnect)
1304 {
1305     return Sync.ExecuteReadOperation(() =>
1306     {
1307         var results = new HashSet<ulong>();
1308         if (linksToConnect.Length > 0)
1309         {
1310             Links.EnsureEachLinkExists(linksToConnect);
1311             AllUsagesCore(linksToConnect[0], results);
1312             for (var i = 1; i < linksToConnect.Length; i++)
1313             {
1314                 var next = new HashSet<ulong>();
1315                 AllUsagesCore(linksToConnect[i], next);
1316                 results.IntersectWith(next);
1317             }
1318             return results;
1319         });
1320 }
1321
1322 public HashSet<ulong> GetAllConnections1(params ulong[] linksToConnect)
1323 {
1324     return Sync.ExecuteReadOperation(() =>
1325     {
1326         var results = new HashSet<ulong>();
1327         if (linksToConnect.Length > 0)
1328         {
1329             Links.EnsureEachLinkExists(linksToConnect);
1330             var collector1 = new AllUsagesCollector(Links.Unsync, results);
1331             collector1.Collect(linksToConnect[0]);
1332             var next = new HashSet<ulong>();
1333             for (var i = 1; i < linksToConnect.Length; i++)
1334             {
1335                 var collector = new AllUsagesCollector(Links.Unsync, next);
1336                 collector.Collect(linksToConnect[i]);
1337                 results.IntersectWith(next);
1338                 next.Clear();
1339             }
1340             return results;
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 public List<ulong> GetAllConnections3(params ulong[] linksToConnect)
1369 {
1370     return Sync.ExecuteReadOperation(() =>
1371     {
1372         var results = new BitString((long)Links.Unsync.Count() + 1); // new
1373         //        → BitArray((int)_links.Total + 1);
1374         if (linksToConnect.Length > 0)
1375

```

```

1376     {
1377         Links.EnsureEachLinkExists(linksToConnect);
1378         var collector1 = new AllUsagesCollector2(Links.Unsync, results);
1379         collector1.Collect(linksToConnect[0]);
1380         for (var i = 1; i < linksToConnect.Length; i++)
1381         {
1382             var next = new BitString((long)Links.Unsync.Count() + 1); //new
1383             ↪ BitArray((int)_links.Total + 1);
1384             var collector = new AllUsagesCollector2(Links.Unsync, next);
1385             collector.Collect(linksToConnect[i]);
1386             results = results.And(next);
1387         }
1388         return results.GetSetUInt64Indices();
1389     });
1390 }
1391
1392 private static ulong[] Simplify(ulong[] sequence)
1393 {
1394     // Считаем новый размер последовательности
1395     long newLength = 0;
1396     var zeroOrManyStepped = false;
1397     for (var i = 0; i < sequence.Length; i++)
1398     {
1399         if (sequence[i] == ZeroOrMany)
1400         {
1401             if (zeroOrManyStepped)
1402             {
1403                 continue;
1404             }
1405             zeroOrManyStepped = true;
1406         }
1407         else
1408         {
1409             //if (zeroOrManyStepped) Is it efficient?
1410             zeroOrManyStepped = false;
1411         }
1412         newLength++;
1413     }
1414     // Строим новую последовательность
1415     zeroOrManyStepped = false;
1416     var newSequence = new ulong[newLength];
1417     long j = 0;
1418     for (var i = 0; i < sequence.Length; i++)
1419     {
1420         //var current = zeroOrManyStepped;
1421         //zeroOrManyStepped = patternSequence[i] == zeroOrMany;
1422         //if (current && zeroOrManyStepped)
1423         //    continue;
1424         //var newZeroOrManyStepped = patternSequence[i] == zeroOrMany;
1425         //if (zeroOrManyStepped && newZeroOrManyStepped)
1426         //    continue;
1427         //zeroOrManyStepped = newZeroOrManyStepped;
1428         if (sequence[i] == ZeroOrMany)
1429         {
1430             if (zeroOrManyStepped)
1431             {
1432                 continue;
1433             }
1434             zeroOrManyStepped = true;
1435         }
1436         else
1437         {
1438             //if (zeroOrManyStepped) Is it efficient?
1439             zeroOrManyStepped = false;
1440         }
1441         newSequence[j++] = sequence[i];
1442     }
1443     return newSequence;
1444 }
1445
1446 public static void TestSimplify()
1447 {
1448     var sequence = new ulong[] { ZeroOrMany, ZeroOrMany, 2, 3, 4, ZeroOrMany,
1449     ↪ ZeroOrMany, ZeroOrMany, 4, ZeroOrMany, ZeroOrMany, ZeroOrMany };
1450     var simplifiedSequence = Simplify(sequence);
1451 }
1452
1453 public List<ulong> GetSimilarSequences() => new List<ulong>();

```

```

1453 public void Prediction()
1454 {
1455     //_links
1456     //_sequences
1457 }
1458
1459 #region From Triplets
1460
1461 //public static void DeleteSequence(Link sequence)
1462 //{
1463 //}
1464
1465 public List<ulong> CollectMatchingSequences(ulong[] links)
1466 {
1467     if (links.Length == 1)
1468     {
1469         throw new Exception("Подпоследовательности с одним элементом не
1470             ↳ поддерживаются.");
1471     }
1472     var leftBound = 0;
1473     var rightBound = links.Length - 1;
1474     var left = links[leftBound++];
1475     var right = links[rightBound--];
1476     var results = new List<ulong>();
1477     CollectMatchingSequences(left, leftBound, links, right, rightBound, ref results);
1478     return results;
1479 }
1480
1481 private void CollectMatchingSequences(ulong leftLink, int leftBound, ulong[]
1482     ↳ middleLinks, ulong rightLink, int rightBound, ref List<ulong> results)
1483 {
1484     var leftLinkTotalReferers = Links.Unsync.Count(leftLink);
1485     var rightLinkTotalReferers = Links.Unsync.Count(rightLink);
1486     if (leftLinkTotalReferers <= rightLinkTotalReferers)
1487     {
1488         var nextLeftLink = middleLinks[leftBound];
1489         var elements = GetRightElements(leftLink, nextLeftLink);
1490         if (leftBound <= rightBound)
1491         {
1492             for (var i = elements.Length - 1; i >= 0; i--)
1493             {
1494                 var element = elements[i];
1495                 if (element != 0)
1496                 {
1497                     CollectMatchingSequences(element, leftBound + 1, middleLinks,
1498                         ↳ rightLink, rightBound, ref results);
1499                 }
1500             }
1501         }
1502         else
1503         {
1504             for (var i = elements.Length - 1; i >= 0; i--)
1505             {
1506                 var element = elements[i];
1507                 if (element != 0)
1508                 {
1509                     results.Add(element);
1510                 }
1511             }
1512         }
1513     }
1514     else
1515     {
1516         var nextRightLink = middleLinks[rightBound];
1517         var elements = GetLeftElements(rightLink, nextRightLink);
1518         if (leftBound <= rightBound)
1519         {
1520             for (var i = elements.Length - 1; i >= 0; i--)
1521             {
1522                 var element = elements[i];
1523                 if (element != 0)
1524                 {
1525                     CollectMatchingSequences(leftLink, leftBound, middleLinks,
1526                         ↳ elements[i], rightBound - 1, ref results);
1527                 }
1528             }
1529         }
1530     }
1531 }

```

```

1527     else
1528     {
1529         for (var i = elements.Length - 1; i >= 0; i--)
1530         {
1531             var element = elements[i];
1532             if (element != 0)
1533             {
1534                 results.Add(element);
1535             }
1536         }
1537     }
1538 }
1539
1540
1541 public ulong[] GetRightElements(ulong startLink, ulong rightLink)
1542 {
1543     var result = new ulong[5];
1544     TryStepRight(startLink, rightLink, result, 0);
1545     Links.Each(_constants.Any, startLink, couple =>
1546     {
1547         if (couple != startLink)
1548         {
1549             if (TryStepRight(couple, rightLink, result, 2))
1550             {
1551                 return false;
1552             }
1553         }
1554         return true;
1555     });
1556     if (Links.GetTarget(Links.GetTarget(startLink)) == rightLink)
1557     {
1558         result[4] = startLink;
1559     }
1560     return result;
1561 }
1562
1563 public bool TryStepRight(ulong startLink, ulong rightLink, ulong[] result, int offset)
1564 {
1565     var added = 0;
1566     Links.Each(startLink, _constants.Any, couple =>
1567     {
1568         if (couple != startLink)
1569         {
1570             var coupleTarget = Links.GetTarget(couple);
1571             if (coupleTarget == rightLink)
1572             {
1573                 result[offset] = couple;
1574                 if (++added == 2)
1575                 {
1576                     return false;
1577                 }
1578             }
1579             else if (Links.GetSource(coupleTarget) == rightLink) // coupleTarget.Linker
1580             ↪ == Net.And &&
1581             {
1582                 result[offset + 1] = couple;
1583                 if (++added == 2)
1584                 {
1585                     return false;
1586                 }
1587             }
1588         }
1589         return true;
1590     });
1591     return added > 0;
1592 }
1593
1594 public ulong[] GetLeftElements(ulong startLink, ulong leftLink)
1595 {
1596     var result = new ulong[5];
1597     TryStepLeft(startLink, leftLink, result, 0);
1598     Links.Each(startLink, _constants.Any, couple =>
1599     {
1600         if (couple != startLink)
1601         {
1602             if (TryStepLeft(couple, leftLink, result, 2))
1603             {
1604                 return false;
1605             }
1606         }
1607     });
1608     return result;
1609 }

```

```

1605     }
1606     return true;
1607 });
1608 if (Links.GetSource(Links.GetSource(leftLink)) == startLink)
1609 {
1610     result[4] = leftLink;
1611 }
1612 return result;
1613 }
1614
1615 public bool TryStepLeft(ulong startLink, ulong leftLink, ulong[] result, int offset)
1616 {
1617     var added = 0;
1618     Links.Each(_constants.Any, startLink, couple =>
1619     {
1620         if (couple != startLink)
1621         {
1622             var coupleSource = Links.GetSource(couple);
1623             if (coupleSource == leftLink)
1624             {
1625                 result[offset] = couple;
1626                 if (++added == 2)
1627                 {
1628                     return false;
1629                 }
1630             }
1631             else if (Links.GetTarget(coupleSource) == leftLink) // coupleSource.Linker
1632                 ↪ == Net.And &&
1633             {
1634                 result[offset + 1] = couple;
1635                 if (++added == 2)
1636                 {
1637                     return false;
1638                 }
1639             }
1640             return true;
1641         });
1642     return added > 0;
1643 }
1644
1645 #endregion
1646
1647 #region Walkers
1648
1649 public class PatternMatcher : RightSequenceWalker<ulong>
1650 {
1651     private readonly Sequences _sequences;
1652     private readonly ulong[] _patternSequence;
1653     private readonly HashSet<LinkIndex> _linksInSequence;
1654     private readonly HashSet<LinkIndex> _results;
1655
1656     #region Pattern Match
1657
1658     enum PatternBlockType
1659     {
1660         Undefined,
1661         Gap,
1662         Elements
1663     }
1664
1665     struct PatternBlock
1666     {
1667         public PatternBlockType Type;
1668         public long Start;
1669         public long Stop;
1670     }
1671
1672     private readonly List<PatternBlock> _pattern;
1673     private int _patternPosition;
1674     private long _sequencePosition;
1675
1676     #endregion
1677
1678     public PatternMatcher(Sequences sequences, LinkIndex[] patternSequence,
1679         ↪ HashSet<LinkIndex> results)
1680         : base(sequences.Links.Unsync, new DefaultStack<ulong>())
1681     {
1682         _sequences = sequences;
1683         _patternSequence = patternSequence;

```



```

1683         _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
1684             ↳ _constants.Any && x != ZeroOrMany));
1685         _results = results;
1686         _pattern = CreateDetailedPattern();
1687     }
1688
1689     protected override bool IsElement(ulong link) => _linksInSequence.Contains(link) ||
1690         ↳ base.IsElement(link);
1691
1692     public bool PatternMatch(LinkIndex sequenceToMatch)
1693     {
1694         _patternPosition = 0;
1695         _sequencePosition = 0;
1696         foreach (var part in Walk(sequenceToMatch))
1697         {
1698             if (!PatternMatchCore(part))
1699             {
1700                 break;
1701             }
1702         }
1703         return _patternPosition == _pattern.Count || (_patternPosition == _pattern.Count
1704             ↳ - 1 && _pattern[_patternPosition].Start == 0);
1705     }
1706
1707     private List<PatternBlock> CreateDetailedPattern()
1708     {
1709         var pattern = new List<PatternBlock>();
1710         var patternBlock = new PatternBlock();
1711         for (var i = 0; i < _patternSequence.Length; i++)
1712         {
1713             if (patternBlock.Type == PatternBlockType.Undefined)
1714             {
1715                 if (_patternSequence[i] == _constants.Any)
1716                 {
1717                     patternBlock.Type = PatternBlockType.Gap;
1718                     patternBlock.Start = 1;
1719                     patternBlock.Stop = 1;
1720                 }
1721                 else if (_patternSequence[i] == ZeroOrMany)
1722                 {
1723                     patternBlock.Type = PatternBlockType.Gap;
1724                     patternBlock.Start = 0;
1725                     patternBlock.Stop = long.MaxValue;
1726                 }
1727                 else
1728                 {
1729                     patternBlock.Type = PatternBlockType.Elements;
1730                     patternBlock.Start = i;
1731                     patternBlock.Stop = i;
1732                 }
1733             }
1734             else if (patternBlock.Type == PatternBlockType.Elements)
1735             {
1736                 if (_patternSequence[i] == _constants.Any)
1737                 {
1738                     pattern.Add(patternBlock);
1739                     patternBlock = new PatternBlock
1740                     {
1741                         Type = PatternBlockType.Gap,
1742                         Start = 1,
1743                         Stop = 1
1744                     };
1745                 }
1746                 else if (_patternSequence[i] == ZeroOrMany)
1747                 {
1748                     pattern.Add(patternBlock);
1749                     patternBlock = new PatternBlock
1750                     {
1751                         Type = PatternBlockType.Gap,
1752                         Start = 0,
1753                         Stop = long.MaxValue
1754                     };
1755                 }
1756                 else
1757                 {
1758                     patternBlock.Stop = i;
1759                 }
1760             }
1761             else // patternBlock.Type == PatternBlockType.Gap
1762             {

```

```

1760         if (_patternSequence[i] == _constants.Any)
1761         {
1762             patternBlock.Start++;
1763             if (patternBlock.Stop < patternBlock.Start)
1764             {
1765                 patternBlock.Stop = patternBlock.Start;
1766             }
1767         }
1768         else if (_patternSequence[i] == ZeroOrMany)
1769         {
1770             patternBlock.Stop = long.MaxValue;
1771         }
1772         else
1773         {
1774             pattern.Add(patternBlock);
1775             patternBlock = new PatternBlock
1776             {
1777                 Type = PatternBlockType.Elements,
1778                 Start = i,
1779                 Stop = i
1780             };
1781         }
1782     }
1783 }
1784 if (patternBlock.Type != PatternBlockType.Undefined)
1785 {
1786     pattern.Add(patternBlock);
1787 }
1788 return pattern;
1789 }
1790
1791 // match: search for regexp anywhere in text
1792 //int match(char* regexp, char* text)
1793 //{
1794 //    do
1795 //    {
1796 //    } while (*text++ != '\0');
1797 //    return 0;
1798 //}
1799
1800 // matchhere: search for regexp at beginning of text
1801 //int matchhere(char* regexp, char* text)
1802 //{
1803 //    if (regexp[0] == '\0')
1804 //        return 1;
1805 //    if (regexp[1] == '*')
1806 //        return matchstar(regexp[0], regexp + 2, text);
1807 //    if (regexp[0] == '$' && regexp[1] == '\0')
1808 //        return *text == '\0';
1809 //    if (*text != '\0' && (regexp[0] == '.' || regexp[0] == *text))
1810 //        return matchhere(regexp + 1, text + 1);
1811 //    return 0;
1812 //}
1813
1814 // matchstar: search for c*regexp at beginning of text
1815 //int matchstar(int c, char* regexp, char* text)
1816 //{
1817 //    do
1818 //    {
1819 //        /* a * matches zero or more instances */
1820 //        if (matchhere(regexp, text))
1821 //            return 1;
1822 //    } while (*text != '\0' && (*text++ == c || c == '.'));
1823 //    return 0;
1824 //}
1825
1826 //private void GetNextPatternElement(out LinkIndex element, out long mininumGap, out
1827 //    long maximumGap)
1828 //{
1829 //    mininumGap = 0;
1830 //    maximumGap = 0;
1831 //    element = 0;
1832 //    for (; _patternPosition < _patternSequence.Length; _patternPosition++)
1833 //    {
1834 //        if (_patternSequence[_patternPosition] == Doublets.Links.Null)
1835 //            mininumGap++;
1836 //        else if (_patternSequence[_patternPosition] == ZeroOrMany)
1837 //            maximumGap = long.MaxValue;
1838 //        else
1839 //            break;

```

```

1838 //    }
1839
1840 //    if (maximumGap < mininumGap)
1841 //        maximumGap = mininumGap;
1842 //}
1843
1844 private bool PatternMatchCore(LinkIndex element)
1845 {
1846     if (_patternPosition >= _pattern.Count)
1847     {
1848         _patternPosition = -2;
1849         return false;
1850     }
1851     var currentPatternBlock = _pattern[_patternPosition];
1852     if (currentPatternBlock.Type == PatternBlockType.Gap)
1853     {
1854         //var currentMatchingBlockLength = (_sequencePosition -
1855         ↪ _lastMatchedBlockPosition);
1856         if (_sequencePosition < currentPatternBlock.Start)
1857         {
1858             _sequencePosition++;
1859             return true; // Двигаемся дальше
1860         }
1861         // Это последний блок
1862         if (_pattern.Count == _patternPosition + 1)
1863         {
1864             _patternPosition++;
1865             _sequencePosition = 0;
1866             return false; // Полное соответствие
1867         }
1868         else
1869         {
1870             if (_sequencePosition > currentPatternBlock.Stop)
1871             {
1872                 return false; // Соответствие невозможно
1873             }
1874             var nextPatternBlock = _pattern[_patternPosition + 1];
1875             if (_patternSequence[nextPatternBlock.Start] == element)
1876             {
1877                 if (nextPatternBlock.Start < nextPatternBlock.Stop)
1878                 {
1879                     _patternPosition++;
1880                     _sequencePosition = 1;
1881                 }
1882                 else
1883                 {
1884                     _patternPosition += 2;
1885                     _sequencePosition = 0;
1886                 }
1887             }
1888         }
1889     }
1890     else // currentPatternBlock.Type == PatternBlockType.Elements
1891     {
1892         var patternElementPosition = currentPatternBlock.Start + _sequencePosition;
1893         if (_patternSequence[patternElementPosition] != element)
1894         {
1895             return false; // Соответствие невозможно
1896         }
1897         if (patternElementPosition == currentPatternBlock.Stop)
1898         {
1899             _patternPosition++;
1900             _sequencePosition = 0;
1901         }
1902         else
1903         {
1904             _sequencePosition++;
1905         }
1906     }
1907     return true;
1908     //if (_patternSequence[_patternPosition] != element)
1909     //    return false;
1910     //else
1911     //{
1912     //    _sequencePosition++;
1913     //    _patternPosition++;
1914     //    return true;
1915     //}
1916     //}

```

```

1916         //if (_filterPosition == _patternSequence.Length)
1917         //{
1918         //    _filterPosition = -2; // Длиннее чем нужно
1919         //    return false;
1920         //}
1921         //if (element != _patternSequence[_filterPosition])
1922         //{
1923         //    _filterPosition = -1;
1924         //    return false; // Начинается иначе
1925         //}
1926         //_filterPosition++;
1927         //if (_filterPosition == (_patternSequence.Length - 1))
1928         //    return false;
1929         //if (_filterPosition >= 0)
1930         //{
1931         //    if (element == _patternSequence[_filterPosition + 1])
1932         //        _filterPosition++;
1933         //    else
1934         //        return false;
1935         //}
1936         //if (_filterPosition < 0)
1937         //{
1938         //    if (element == _patternSequence[0])
1939         //        _filterPosition = 0;
1940         //}
1941     }
1942
1943     public void AddAllPatternMatchedToResults(IEnumerable<ulong> sequencesToMatch)
1944     {
1945         foreach (var sequenceToMatch in sequencesToMatch)
1946         {
1947             if (PatternMatch(sequenceToMatch))
1948             {
1949                 _results.Add(sequenceToMatch);
1950             }
1951         }
1952     }
1953 }
1954
1955 #endregion
1956 }
1957 }

```

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

```

1  using Platform.Collections.Lists;
2  using Platform.Data.Sequences;
3  using System.Collections.Generic;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Sequences
8  {
9      public static class SequencesExtensions
10     {
11         public static TLink Create<TLink>(this ISequences<TLink> sequences, IList<TLink[]>
12             ↳ groupedSequence)
13         {
14             var finalSequence = new TLink[groupedSequence.Count];
15             for (var i = 0; i < finalSequence.Length; i++)
16             {
17                 var part = groupedSequence[i];
18                 finalSequence[i] = part.Length == 1 ? part[0] : sequences.Create(part);
19             }
20             return sequences.Create(finalSequence);
21         }
22
23         public static IList<TLink> ToList<TLink>(this ISequences<TLink> sequences, TLink
24             ↳ sequence)
25         {
26             var list = new List<TLink>();
27             sequences.EachPart(list.AddAndReturnTrue, sequence);
28             return list;
29         }
30     }
31 }

```

./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 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
13
14 namespace Platform.Data.Doublets.Sequences
15 {
16     public class SequencesOptions<TLink> // TODO: To use type parameter <TLink> the
        ↪ ILinks<TLink> must contain GetConstants function.
17     {
18         private static readonly EqualityComparer<TLink> _equalityComparer =
        ↪ 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
                                ↪ link.");
56                         }
57                     }
58                 }
59                 if (MarkedSequenceMatcher == null)
60                 {
61                     MarkedSequenceMatcher = new MarkedSequenceCriterionMatcher<TLink>(links,
                                ↪ SequenceMarkerLink);
62                 }
63             }
64             var balancedVariantConverter = new BalancedVariantConverter<TLink>(links);
65             if (UseCompression)
66             {
67                 if (LinksToSequenceConverter == null)
68                 {
69                     ICounter<TLink, TLink> totalSequenceSymbolFrequencyCounter;
70                     if (UseSequenceMarker)
71                     {
72                         totalSequenceSymbolFrequencyCounter = new
                                ↪ TotalMarkedSequenceSymbolFrequencyCounter<TLink>(links,
                                ↪ MarkedSequenceMatcher);
73                     }
74                     else
75                     {

```

```

76         totalSequenceSymbolFrequencyCounter = new
77             ↪ TotalSequenceSymbolFrequencyCounter<TLink>(links);
78     }
79     var doubletFrequenciesCache = new LinkFrequenciesCache<TLink>(links,
80         ↪ totalSequenceSymbolFrequencyCounter);
81     var compressingConverter = new CompressingConverter<TLink>(links,
82         ↪ balancedVariantConverter, doubletFrequenciesCache);
83     LinksToSequenceConverter = compressingConverter;
84 }
85 else
86 {
87     if (LinksToSequenceConverter == null)
88     {
89         LinksToSequenceConverter = balancedVariantConverter;
90     }
91     if (UseIndex && Index == null)
92     {
93         Index = new SequenceIndex<TLink>(links);
94     }
95     if (Walker == null)
96     {
97         Walker = new RightSequenceWalker<TLink>(links, new DefaultStack<TLink>());
98     }
99 }
100 public void ValidateOptions()
101 {
102     if (UseGarbageCollection && !UseSequenceMarker)
103     {
104         throw new NotSupportedException("To use garbage collection UseSequenceMarker
105             ↪ option must be on.");
106     }
107 }
108 }

```

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

```

1 using System.Collections.Generic;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Data.Doublets.Sequences.Walkers
6 {
7     public interface ISequenceWalker<TLink>
8     {
9         IEnumerable<TLink> Walk(TLink sequence);
10    }
11 }

```

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

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3 using Platform.Collections.Stacks;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Data.Doublets.Sequences.Walkers
8 {
9     public class LeftSequenceWalker<TLink> : SequenceWalkerBase<TLink>
10    {
11        public LeftSequenceWalker(ILinks<TLink> links, IStack<TLink> stack) : base(links, stack)
12            ↪ { }
13
14        [MethodImpl(MethodImplOptions.AggressiveInlining)]
15        protected override TLink GetNextElementAfterPop(TLink element) =>
16            ↪ Links.GetSource(element);
17
18        [MethodImpl(MethodImplOptions.AggressiveInlining)]
19        protected override TLink GetNextElementAfterPush(TLink element) =>
20            ↪ Links.GetTarget(element);
21
22        [MethodImpl(MethodImplOptions.AggressiveInlining)]
23        protected override IEnumerable<TLink> WalkContents(TLink element)
24        {
25            var parts = Links.GetLink(element);
26            var start = Links.Constants.IndexPart + 1;

```

```

24         for (var i = parts.Count - 1; i >= start; i--)
25         {
26             var part = parts[i];
27             if (IsElement(part))
28             {
29                 yield return part;
30             }
31         }
32     }
33 }
34 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  //#define USEARRAYPOOL
8  #if USEARRAYPOOL
9      using Platform.Collections;
10 #endif
11
12 namespace Platform.Data.Doublets.Sequences.Walkers
13 {
14     public class LeveledSequenceWalker<TLink> : LinksOperatorBase<TLink>, ISequenceWalker<TLink>
15     {
16         private static readonly EqualityComparer<TLink> _equalityComparer =
17             ↳ EqualityComparer<TLink>.Default;
18
19         private readonly Func<TLink, bool> _isElement;
20
21         public LeveledSequenceWalker(ILinks<TLink> links, Func<TLink, bool> isElement) :
22             ↳ base(links) => _isElement = isElement;
23
24         public LeveledSequenceWalker(ILinks<TLink> links) : base(links) => _isElement =
25             ↳ Links.IsPartialPoint;
26
27         public IEnumerable<TLink> Walk(TLink sequence) => ToArray(sequence);
28
29         public TLink[] ToArray(TLink sequence)
30         {
31             var length = 1;
32             var array = new TLink[length];
33             array[0] = sequence;
34             if (_isElement(sequence))
35             {
36                 return array;
37             }
38             bool hasElements;
39             do
40             {
41                 length *= 2;
42 #if USEARRAYPOOL
43                 var nextArray = ArrayPool.Allocate<ulong>(length);
44 #else
45                 var nextArray = new TLink[length];
46 #endif
47                 hasElements = false;
48                 for (var i = 0; i < array.Length; i++)
49                 {
50                     var candidate = array[i];
51                     if (_equalityComparer.Equals(array[i], default))
52                     {
53                         continue;
54                     }
55                     var doubletOffset = i * 2;
56                     if (_isElement(candidate))
57                     {
58                         nextArray[doubletOffset] = candidate;
59                     }
60                     else
61                     {
62                         var link = Links.GetLink(candidate);
63                         var linkSource = Links.GetSource(link);
64                         var linkTarget = Links.GetTarget(link);
65                         nextArray[doubletOffset] = linkSource;
66                         nextArray[doubletOffset + 1] = linkTarget;
67                         if (!hasElements)

```

```

65         {
66             hasElements = !(_isElement(linkSource) && _isElement(linkTarget));
67         }
68     }
69 }
70 #if USEARRAYPOOL
71     if (array.Length > 1)
72     {
73         ArrayPool.Free(array);
74     }
75 #endif
76     array = nextArray;
77 }
78 while (hasElements);
79 var filledElementsCount = CountFilledElements(array);
80 if (filledElementsCount == array.Length)
81 {
82     return array;
83 }
84 else
85 {
86     return CopyFilledElements(array, filledElementsCount);
87 }
88 }
89
90 [MethodImpl(MethodImplOptions.AggressiveInlining)]
91 private static TLink[] CopyFilledElements(TLink[] array, int filledElementsCount)
92 {
93     var finalArray = new TLink[filledElementsCount];
94     for (int i = 0, j = 0; i < array.Length; i++)
95     {
96         if (!_equalityComparer.Equals(array[i], default))
97         {
98             finalArray[j] = array[i];
99             j++;
100         }
101     }
102     #if USEARRAYPOOL
103         ArrayPool.Free(array);
104     #endif
105     return finalArray;
106 }
107
108 [MethodImpl(MethodImplOptions.AggressiveInlining)]
109 private static int CountFilledElements(TLink[] array)
110 {
111     var count = 0;
112     for (var i = 0; i < array.Length; i++)
113     {
114         if (!_equalityComparer.Equals(array[i], default))
115         {
116             count++;
117         }
118     }
119     return count;
120 }
121 }
122 }

```

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

```

1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3 using Platform.Collections.Stacks;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Data.Doublets.Sequences.Walkers
8 {
9     public class RightSequenceWalker<TLink> : SequenceWalkerBase<TLink>
10     {
11         public RightSequenceWalker(ILinks<TLink> links, IStack<TLink> stack) : base(links,
12             ↪ stack) { }
13
14         [MethodImpl(MethodImplOptions.AggressiveInlining)]
15         protected override TLink GetNextElementAfterPop(TLink element) =>
16             ↪ Links.GetTarget(element);
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected override TLink GetNextElementAfterPush(TLink element) =>
20             ↪ Links.GetSource(element);
21     }
22 }

```



```

18     [MethodImpl(MethodImplOptions.AggressiveInlining)]
19     protected override IEnumerable<TLink> WalkContents(TLink element)
20     {
21         var parts = Links.GetLink(element);
22         for (var i = Links.Constants.IndexPart + 1; i < parts.Count; i++)
23         {
24             var part = parts[i];
25             if (IsElement(part))
26             {
27                 yield return part;
28             }
29         }
30     }
31 }
32 }
33 }

```

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

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3  using Platform.Collections.Stacks;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Sequences.Walkers
8  {
9      public abstract class SequenceWalkerBase<TLink> : LinksOperatorBase<TLink>,
10         ↳ ISequenceWalker<TLink>
11     {
12         private readonly IStack<TLink> _stack;
13
14         protected SequenceWalkerBase(ILinks<TLink> links, IStack<TLink> stack) : base(links) =>
15             ↳ _stack = stack;
16
17         public IEnumerable<TLink> Walk(TLink sequence)
18         {
19             _stack.Clear();
20             var element = sequence;
21             if (IsElement(element))
22             {
23                 yield return element;
24             }
25             else
26             {
27                 while (true)
28                 {
29                     if (IsElement(element))
30                     {
31                         if (_stack.IsEmpty)
32                         {
33                             break;
34                         }
35                         element = _stack.Pop();
36                         foreach (var output in WalkContents(element))
37                         {
38                             yield return output;
39                         }
40                         element = GetNextElementAfterPop(element);
41                     }
42                     else
43                     {
44                         _stack.Push(element);
45                         element = GetNextElementAfterPush(element);
46                     }
47                 }
48             }
49         }
50
51         [MethodImpl(MethodImplOptions.AggressiveInlining)]
52         protected virtual bool IsElement(TLink elementLink) => Links.IsPartialPoint(elementLink);
53
54         [MethodImpl(MethodImplOptions.AggressiveInlining)]
55         protected abstract TLink GetNextElementAfterPop(TLink element);
56
57         [MethodImpl(MethodImplOptions.AggressiveInlining)]
58         protected abstract TLink GetNextElementAfterPush(TLink element);
59
60         [MethodImpl(MethodImplOptions.AggressiveInlining)]
61         protected abstract IEnumerable<TLink> WalkContents(TLink element);

```

```

60     }
61 }

./Platform.Data.Doublets/Stacks/Stack.cs
1  using System.Collections.Generic;
2  using Platform.Collections.Stacks;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Stacks
7  {
8      public class Stack<TLink> : IStack<TLink>
9      {
10         private static readonly EqualityComparer<TLink> _equalityComparer =
11             ⇨ EqualityComparer<TLink>.Default;
12
13         private readonly ILinks<TLink> _links;
14         private readonly TLink _stack;
15
16         public bool IsEmpty => _equalityComparer.Equals(Peek(), _stack);
17
18         public Stack(ILinks<TLink> links, TLink stack)
19         {
20             _links = links;
21             _stack = stack;
22         }
23
24         private TLink GetStackMarker() => _links.GetSource(_stack);
25
26         private TLink GetTop() => _links.GetTarget(_stack);
27
28         public TLink Peek() => _links.GetTarget(GetTop());
29
30         public TLink Pop()
31         {
32             var element = Peek();
33             if (!_equalityComparer.Equals(element, _stack))
34             {
35                 var top = GetTop();
36                 var previousTop = _links.GetSource(top);
37                 _links.Update(_stack, GetStackMarker(), previousTop);
38                 _links.Delete(top);
39             }
40             return element;
41         }
42
43         public void Push(TLink element) => _links.Update(_stack, GetStackMarker(),
44             ⇨ _links.GetOrCreate(GetTop(), element));
45     }
46 }

```

```

./Platform.Data.Doublets/Stacks/StackExtensions.cs
1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  namespace Platform.Data.Doublets.Stacks
4  {
5      public static class StackExtensions
6      {
7          public static TLink CreateStack<TLink>(this ILinks<TLink> links, TLink stackMarker)
8          {
9              var stackPoint = links.CreatePoint();
10             var stack = links.Update(stackPoint, stackMarker, stackPoint);
11             return stack;
12         }
13     }
14 }

```

```

./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  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9  namespace Platform.Data.Doublets
10 {
11     /// <remarks>
12     /// TODO: Autogeneration of synchronized wrapper (decorator).

```

```

13  /// TODO: Try to unfold code of each method using IL generation for performance improvements.
14  /// TODO: Or even to unfold multiple layers of implementations.
15  /// </remarks>
16  public class SynchronizedLinks<T> : ISynchronizedLinks<T>
17  {
18      public LinksCombinedConstants<T, T, int> Constants { get; }
19      public ISynchronization SyncRoot { get; }
20      public ILinks<T> Sync { get; }
21      public ILinks<T> Unsync { get; }
22
23      public SynchronizedLinks(ILinks<T> links) : this(new ReaderWriterLockSynchronization(),
24          ↪ links) { }
25
26      public SynchronizedLinks(ISynchronization synchronization, ILinks<T> links)
27      {
28          SyncRoot = synchronization;
29          Sync = this;
30          Unsync = links;
31          Constants = links.Constants;
32      }
33
34      public T Count(IList<T> restriction) => SyncRoot.ExecuteReadOperation(restriction,
35          ↪ Unsync.Count);
36      public T Each(Func<IList<T>, T> handler, IList<T> restrictions) =>
37          ↪ SyncRoot.ExecuteReadOperation(handler, restrictions, (handler1, restrictions1) =>
38          ↪ Unsync.Each(handler1, restrictions1));
39      public T Create() => SyncRoot.ExecuteWriteOperation(Unsync.Create);
40      public T Update(IList<T> restrictions) => SyncRoot.ExecuteWriteOperation(restrictions,
41          ↪ Unsync.Update);
42      public void Delete(T link) => SyncRoot.ExecuteWriteOperation(link, Unsync.Delete);
43
44      //public T Trigger(IList<T> restriction, Func<IList<T>, IList<T>, T> matchedHandler,
45      ↪ IList<T> substitution, Func<IList<T>, IList<T>, T> substitutedHandler)
46      //{
47      //    if (restriction != null && substitution != null &&
48      ↪ !substitution.EqualTo(restriction))
49      //        return SyncRoot.ExecuteWriteOperation(restriction, matchedHandler,
50      ↪ substitution, substitutedHandler, Unsync.Trigger);
51      //    return SyncRoot.ExecuteReadOperation(restriction, matchedHandler, substitution,
52      ↪ substitutedHandler, Unsync.Trigger);
53      //}
54  }
55  }

```

#### ./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 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
11 namespace Platform.Data.Doublets
12 {
13     /// <summary>
14     /// Структура описывающая уникальную связь.
15     /// </summary>
16     public struct UInt64Link : IEquatable<UInt64Link>, IReadOnlyList<ulong>, IList<ulong>
17     {
18         private static readonly LinksCombinedConstants<bool, ulong, int> _constants =
19             ↪ Default<LinksCombinedConstants<bool, ulong, int>>.Instance;
20
21         private const int Length = 3;
22
23         public readonly ulong Index;
24         public readonly ulong Source;
25         public readonly ulong Target;
26
27         public static readonly UInt64Link Null = new UInt64Link();
28
29         public UInt64Link(params ulong[] values)
30         {
31             Index = values.Length > _constants.IndexPart ? values[_constants.IndexPart] :
32             ↪ _constants.Null;

```

```

32         Source = values.Length > _constants.SourcePart ? values[_constants.SourcePart] :
33         ↪ _constants.Null;
34         Target = values.Length > _constants.TargetPart ? values[_constants.TargetPart] :
35         ↪ _constants.Null;
36     }
37
38     public UInt64Link(IList<ulong> values)
39     {
40         Index = values.Count > _constants.IndexPart ? values[_constants.IndexPart] :
41         ↪ _constants.Null;
42         Source = values.Count > _constants.SourcePart ? values[_constants.SourcePart] :
43         ↪ _constants.Null;
44         Target = values.Count > _constants.TargetPart ? values[_constants.TargetPart] :
45         ↪ _constants.Null;
46     }
47
48     public UInt64Link(ulong index, ulong source, ulong target)
49     {
50         Index = index;
51         Source = source;
52         Target = target;
53     }
54
55     public UInt64Link(ulong source, ulong target)
56     : this(_constants.Null, source, target)
57     {
58         Source = source;
59         Target = target;
60     }
61
62     public static UInt64Link Create(ulong source, ulong target) => new UInt64Link(source,
63     ↪ target);
64
65     public override int GetHashCode() => (Index, Source, Target).GetHashCode();
66
67     public bool IsNull() => Index == _constants.Null
68         && Source == _constants.Null
69         && Target == _constants.Null;
70
71     public override bool Equals(object other) => other is UInt64Link &&
72     ↪ Equals((UInt64Link)other);
73
74     public bool Equals(UInt64Link other) => Index == other.Index
75         && Source == other.Source
76         && Target == other.Target;
77
78     public static string ToString(ulong index, ulong source, ulong target) => $"{index}:
79     ↪ {source}->{target}";
80
81     public static string ToString(ulong source, ulong target) => $"{source}->{target}";
82
83     public static implicit operator ulong[] (UInt64Link link) => link.ToArray();
84
85     public static implicit operator UInt64Link(ulong[] linkArray) => new
86     ↪ UInt64Link(linkArray);
87
88     public override string ToString() => Index == _constants.Null ? ToString(Source, Target)
89     ↪ : ToString(Index, Source, Target);
90
91     #region IList
92
93     public ulong this[int index]
94     {
95         get
96         {
97             Ensure.Always.ArgumentInRange(index, new Range<int>(0, Length - 1),
98             ↪ nameof(index));
99             if (index == _constants.IndexPart)
100             {
101                 return Index;
102             }
103             if (index == _constants.SourcePart)
104             {
105                 return Source;
106             }
107             if (index == _constants.TargetPart)
108             {
109                 return Target;
110             }
111         }
112     }

```

```

100         throw new NotSupportedException(); // Impossible path due to
101         ↪ Ensure.ArgumentInRange
102     }
103     set => throw new NotSupportedException();
104 }
105 public int Count => Length;
106 public bool IsReadOnly => true;
107
108 IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
109
110 public IEnumerator<ulong> GetEnumerator()
111 {
112     yield return Index;
113     yield return Source;
114     yield return Target;
115 }
116
117 public void Add(ulong item) => throw new NotSupportedException();
118
119 public void Clear() => throw new NotSupportedException();
120
121 public bool Contains(ulong item) => IndexOf(item) >= 0;
122
123 public void CopyTo(ulong[] 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 ArgumentException();
131     }
132     array[arrayIndex++] = Index;
133     array[arrayIndex++] = Source;
134     array[arrayIndex] = Target;
135 }
136
137 public bool Remove(ulong item) => Throw.A.NotSupportedExceptionAndReturn<bool>();
138
139 public int IndexOf(ulong item)
140 {
141     if (Index == item)
142     {
143         return _constants.IndexPart;
144     }
145     if (Source == item)
146     {
147         return _constants.SourcePart;
148     }
149     if (Target == item)
150     {
151         return _constants.TargetPart;
152     }
153     return -1;
154 }
155
156 public void Insert(int index, ulong item) => throw new NotSupportedException();
157
158 public void RemoveAt(int index) => throw new NotSupportedException();
159
160 #endregion
161 }
162 }
163 }

```

./Platform.Data.Doublets/UInt64LinkExtensions.cs

```

1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  namespace Platform.Data.Doublets
4  {
5      public static class UInt64LinkExtensions
6      {
7          public static bool IsFullPoint(this UInt64Link link) => Point<ulong>.IsFullPoint(link);
8          public static bool IsPartialPoint(this UInt64Link link) =>
9              ↪ Point<ulong>.IsPartialPoint(link);
10     }
11 }

```

./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 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11 namespace Platform.Data.Doublets
12 {
13     public static class UInt64LinksExtensions
14     {
15         public static readonly LinksCombinedConstants<bool, ulong, int> Constants =
16             ↪ Default<LinksCombinedConstants<bool, ulong, int>>.Instance;
17
18         public static void UseUnicode(this ILinks<ulong> links) => UnicodeMap.InitNew(links);
19
20         public static void EnsureEachLinkExists(this ILinks<ulong> links, IList<ulong> sequence)
21         {
22             if (sequence == null)
23             {
24                 return;
25             }
26             for (var i = 0; i < sequence.Count; i++)
27             {
28                 if (!links.Exists(sequence[i]))
29                 {
30                     throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
31                         ↪ $"sequence[{i}]");
32                 }
33             }
34         }
35
36         public static void EnsureEachLinkIsAnyOrExists(this ILinks<ulong> links, IList<ulong>
37             ↪ sequence)
38         {
39             if (sequence == null)
40             {
41                 return;
42             }
43             for (var i = 0; i < sequence.Count; i++)
44             {
45                 if (sequence[i] != Constants.Any && !links.Exists(sequence[i]))
46                 {
47                     throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
48                         ↪ $"sequence[{i}]");
49                 }
50             }
51         }
52
53         public static bool AnyLinkIsAny(this ILinks<ulong> links, params ulong[] sequence)
54         {
55             if (sequence == null)
56             {
57                 return false;
58             }
59             var constants = links.Constants;
60             for (var i = 0; i < sequence.Length; i++)
61             {
62                 if (sequence[i] == constants.Any)
63                 {
64                     return true;
65                 }
66             }
67             return false;
68         }
69
70         public static string FormatStructure(this ILinks<ulong> links, ulong linkIndex,
71             ↪ Func<UInt64Link, bool> isElement, bool renderIndex = false, bool renderDebug = false)
72         {
73             var sb = new StringBuilder();
74             var visited = new HashSet<ulong>();
75             links.AppendStructure(sb, visited, linkIndex, isElement, (innerSb, link) =>
76                 ↪ innerSb.Append(link.Index), renderIndex, renderDebug);
77             return sb.ToString();
78         }
79     }
80 }
```

```

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
84 public static void AppendStructure(this ILinks<ulong> links, StringBuilder sb,
85     ↳ HashSet<ulong> visited, ulong linkIndex, Func<UInt64Link, bool> isElement,
86     ↳ Action<StringBuilder, UInt64Link> appendElement, bool renderIndex = false, bool
87     ↳ renderDebug = false)
88 {
89     if (sb == null)
90     {
91         throw new ArgumentNullException(nameof(sb));
92     }
93     if (linkIndex == Constants.Null || linkIndex == Constants.Any || linkIndex ==
94     ↳ Constants.Itself)
95     {
96         return;
97     }
98     if (links.Exists(linkIndex))
99     {
100         if (visited.Add(linkIndex))
101         {
102             sb.Append('(');
103             var link = new UInt64Link(links.GetLink(linkIndex));
104             if (renderIndex)
105             {
106                 sb.Append(link.Index);
107                 sb.Append(':');
108             }
109             if (link.Source == link.Index)
110             {
111                 sb.Append(link.Index);
112             }
113             else
114             {
115                 var source = new UInt64Link(links.GetLink(link.Source));
116                 if (isElement(source))
117                 {
118                     appendElement(sb, source);
119                 }
120                 else
121                 {
122                     links.AppendStructure(sb, visited, source.Index, isElement,
123                     ↳ appendElement, renderIndex);
124                 }
125             }
126             sb.Append(' ');
127             if (link.Target == link.Index)
128             {
129                 sb.Append(link.Index);
130             }
131             else
132             {
133                 var target = new UInt64Link(links.GetLink(link.Target));
134                 if (isElement(target))
135                 {
136                     appendElement(sb, target);
137                 }
138                 else
139                 {
140                     links.AppendStructure(sb, visited, target.Index, isElement,
141                     ↳ appendElement, renderIndex);
142                 }
143             }
144             sb.Append(')');
145         }
146         else
147         {
148             if (renderDebug)
149             {
150

```

```

142         sb.Append('*');
143     }
144     sb.Append(linkIndex);
145 }
146 }
147 else
148 {
149     if (renderDebug)
150     {
151         sb.Append('~');
152     }
153     sb.Append(linkIndex);
154 }
155 }
156 }
157 }

```

./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 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
15
16 namespace Platform.Data.Doublets
17 {
18     public class UInt64LinksTransactionsLayer : LinksDisposableDecoratorBase

```



```

62     /// }
63     ///
64     /// public TransactionItemType Type
65     /// {
66     ///     get
67     ///     {
68     ///         // Использовать по одному биту из TransactionId и Timestamp,
69     ///         // для значения в 2 бита, которое представляет тип операции
70     ///         throw new NotImplementedException();
71     ///     }
72     /// }
73     /// }
74     ///
75     /// private struct Transition
76     /// {
77     ///     public TransitionHeader Header;
78     ///     public Link Source;
79     ///     public Link Linker;
80     ///     public Link Target;
81     /// }
82     ///
83     /// </remarks>
84     public struct Transition
85     {
86         public static readonly long Size = Structure<Transition>.Size;
87
88         public readonly ulong TransactionId;
89         public readonly UInt64Link Before;
90         public readonly UInt64Link After;
91         public readonly Timestamp Timestamp;
92
93         public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
94             ↪ transactionId, UInt64Link before, UInt64Link after)
95         {
96             TransactionId = transactionId;
97             Before = before;
98             After = after;
99             Timestamp = uniqueTimestampFactory.Create();
100         }
101
102         public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
103             ↪ transactionId, UInt64Link before)
104             : this(uniqueTimestampFactory, transactionId, before, default)
105         {
106         }
107
108         public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong transactionId
109             : this(uniqueTimestampFactory, transactionId, default, default)
110         {
111         }
112
113         public override string ToString() => $"{Timestamp} {TransactionId}: {Before} =>
114             ↪ {After}";
115     }
116
117     /// <remarks>
118     /// Другие варианты реализации транзакций (атомарности):
119     /// 1. Разделение хранения значения связи ((Source Target) или (Source Linker
120     ///     ↪ Target)) и индексов.
121     /// 2. Хранение трансформаций/операций в отдельном хранилище Links, но дополнительно
122     ///     ↪ потребуется решить вопрос
123     ///     со ссылками на внешние идентификаторы, или как-то иначе решить вопрос с
124     ///     ↪ пересечениями идентификаторов.
125     ///
126     /// Где хранить промежуточный список транзакций?
127     ///
128     /// В оперативной памяти:
129     /// Минусы:
130     /// 1. Может усложнить систему, если она будет функционировать самостоятельно,
131     ///     так как нужно отдельно выделять память под список трансформаций.
132     /// 2. Выделенной оперативной памяти может не хватить, в том случае,
133     ///     если транзакция использует слишком много трансформаций.
134     ///     -> Можно использовать жёсткий диск для слишком длинных транзакций.
135     ///     -> Максимальный размер списка трансформаций можно ограничить / задать
136     ///     ↪ константой.
137     /// 3. При подтверждении транзакции (Commit) все трансформации записываются разом
138     ///     ↪ создавая задержку.
139     ///

```

```

132 /// На жёстком диске:
133 /// Минусы:
134 /// 1. Длительный отклик, на запись каждой трансформации.
135 /// 2. Лог транзакций дополнительно наполняется отменёнными транзакциями.
136 /// -> Это может решаться упаковкой/исключением дублирующих операций.
137 /// -> Также это может решаться тем, что короткие транзакции вообще
138 /// не будут записываться в случае отката.
139 /// 3. Перед тем как выполнять отмену операций транзакции нужно дождаться пока все
    ↪ операции (трансформации)
140 /// будут записаны в лог.
141 ///
142 /// </remarks>
143 public class Transaction : DisposableBase
144 {
145     private readonly Queue<Transition> _transitions;
146     private readonly UInt64LinksTransactionsLayer _layer;
147     public bool IsCommitted { get; private set; }
148     public bool IsReverted { get; private set; }
149
150     public Transaction(UInt64LinksTransactionsLayer layer)
151     {
152         _layer = layer;
153         if (_layer._currentTransactionId != 0)
154         {
155             throw new NotSupportedException("Nested transactions not supported.");
156         }
157         IsCommitted = false;
158         IsReverted = false;
159         _transitions = new Queue<Transition>();
160         SetCurrentTransaction(layer, this);
161     }
162
163     public void Commit()
164     {
165         EnsureTransactionAllowsWriteOperations(this);
166         while (_transitions.Count > 0)
167         {
168             var transition = _transitions.Dequeue();
169             _layer._transitions.Enqueue(transition);
170         }
171         _layer._lastCommittedTransactionId = _layer._currentTransactionId;
172         IsCommitted = true;
173     }
174
175     private void Revert()
176     {
177         EnsureTransactionAllowsWriteOperations(this);
178         var transitionsToRevert = new Transition[_transitions.Count];
179         _transitions.CopyTo(transitionsToRevert, 0);
180         for (var i = transitionsToRevert.Length - 1; i >= 0; i--)
181         {
182             _layer.RevertTransition(transitionsToRevert[i]);
183         }
184         IsReverted = true;
185     }
186
187     public static void SetCurrentTransaction(UInt64LinksTransactionsLayer layer,
    ↪ Transaction transaction)
188     {
189         layer._currentTransactionId = layer._lastCommittedTransactionId + 1;
190         layer._currentTransactionTransitions = transaction._transitions;
191         layer._currentTransaction = transaction;
192     }
193
194     public static void EnsureTransactionAllowsWriteOperations(Transaction transaction)
195     {
196         if (transaction.IsReverted)
197         {
198             throw new InvalidOperationException("Transation is reverted.");
199         }
200         if (transaction.IsCommitted)
201         {
202             throw new InvalidOperationException("Transation is committed.");
203         }
204     }
205
206     protected override void Dispose(bool manual, bool wasDisposed)
207     {
208         if (!wasDisposed && _layer != null && !_layer.IsDisposed)

```

```

209         {
210             if (!IsCommitted && !IsReverted)
211             {
212                 Revert();
213             }
214             _layer.ResetCurrentTransation();
215         }
216     }
217 }
218
219 public static readonly TimeSpan DefaultPushDelay = TimeSpan.FromSeconds(0.1);
220
221 private readonly string _logAddress;
222 private readonly FileStream _log;
223 private readonly Queue<Transition> _transitions;
224 private readonly UniqueTimestampFactory _uniqueTimestampFactory;
225 private Task _transitionsPusher;
226 private Transition _lastCommittedTransition;
227 private ulong _currentTransactionId;
228 private Queue<Transition> _currentTransactionTransitions;
229 private Transaction _currentTransaction;
230 private ulong _lastCommittedTransactionId;
231
232 public UInt64LinksTransactionsLayer(ILinks<ulong> links, string logAddress)
233     : base(links)
234 {
235     if (string.IsNullOrEmpty(logAddress))
236     {
237         throw new ArgumentNullException(nameof(logAddress));
238     }
239     // В первой строке файла хранится последняя закоммиченную транзакцию.
240     // При запуске это используется для проверки удачного закрытия файла лога.
241     // In the first line of the file the last committed transaction is stored.
242     // On startup, this is used to check that the log file is successfully closed.
243     var lastCommittedTransition = FileHelpers.ReadFirstOrDefault<Transition>(logAddress);
244     var lastWrittenTransition = FileHelpers.ReadLastOrDefault<Transition>(logAddress);
245     if (!lastCommittedTransition.Equals(lastWrittenTransition))
246     {
247         Dispose();
248         throw new NotSupportedException("Database is damaged, autorecovery is not
249             ↳ supported yet.");
250     }
251     if (lastCommittedTransition.Equals(default(Transition)))
252     {
253         FileHelpers.WriteFirst(logAddress, lastCommittedTransition);
254     }
255     _lastCommittedTransition = lastCommittedTransition;
256     // TODO: Think about a better way to calculate or store this value
257     var allTransitions = FileHelpers.ReadAll<Transition>(logAddress);
258     _lastCommittedTransactionId = allTransitions.Max(x => x.TransactionId);
259     _uniqueTimestampFactory = new UniqueTimestampFactory();
260     _logAddress = logAddress;
261     _log = FileHelpers.Append(logAddress);
262     _transitions = new Queue<Transition>();
263     _transitionsPusher = new Task(TransitionsPusher);
264     _transitionsPusher.Start();
265 }
266
267 public IList<ulong> GetLinkValue(ulong link) => Links.GetLink(link);
268
269 public override ulong Create()
270 {
271     var createdLinkIndex = Links.Create();
272     var createdLink = new UInt64Link(Links.GetLink(createdLinkIndex));
273     CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
274         ↳ default, createdLink));
275     return createdLinkIndex;
276 }
277
278 public override ulong Update(IList<ulong> parts)
279 {
280     var linkIndex = parts[Constants.IndexPart];
281     var beforeLink = new UInt64Link(Links.GetLink(linkIndex));
282     linkIndex = Links.Update(parts);
283     var afterLink = new UInt64Link(Links.GetLink(linkIndex));
284     CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
285         ↳ beforeLink, afterLink));
286     return linkIndex;
287 }

```

```

285
286 public override void Delete(ulong link)
287 {
288     var deletedLink = new UInt64Link(Links.GetLink(link));
289     Links.Delete(link);
290     CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
291         ↪ deletedLink, default));
292 }
293
294 [MethodImpl(MethodImplOptions.AggressiveInlining)]
295 private Queue<Transition> GetCurrentTransitions() => _currentTransactionTransitions ??
296     ↪ _transitions;
297
298 private void CommitTransition(Transition transition)
299 {
300     if (_currentTransaction != null)
301     {
302         Transaction.EnsureTransactionAllowsWriteOperations(_currentTransaction);
303     }
304     var transitions = GetCurrentTransitions();
305     transitions.Enqueue(transition);
306 }
307
308 private void RevertTransition(Transition transition)
309 {
310     if (transition.After.IsNull()) // Revert Deletion with Creation
311     {
312         Links.Create();
313     }
314     else if (transition.Before.IsNull()) // Revert Creation with Deletion
315     {
316         Links.Delete(transition.After.Index);
317     }
318     else // Revert Update
319     {
320         Links.Update(new[] { transition.After.Index, transition.Before.Source,
321             ↪ transition.Before.Target });
322     }
323 }
324
325 private void ResetCurrentTransation()
326 {
327     _currentTransactionId = 0;
328     _currentTransactionTransitions = null;
329     _currentTransaction = null;
330 }
331
332 private void PushTransitions()
333 {
334     if (_log == null || _transitions == null)
335     {
336         return;
337     }
338     for (var i = 0; i < _transitions.Count; i++)
339     {
340         var transition = _transitions.Dequeue();
341         _log.Write(transition);
342         _lastCommittedTransition = transition;
343     }
344 }
345
346 private void TransitionsPusher()
347 {
348     while (!IsDisposed && _transitionsPusher != null)
349     {
350         Thread.Sleep(DefaultPushDelay);
351         PushTransitions();
352     }
353 }
354
355 public Transaction BeginTransaction() => new Transaction(this);
356
357 private void DisposeTransitions()
358 {
359     try
360     {
361         var pusher = _transitionsPusher;
362         if (pusher != null)

```

```

361         {
362             _transitionsPusher = null;
363             pusher.Wait();
364         }
365         if (_transitions != null)
366         {
367             PushTransitions();
368         }
369         _log.DisposeIfPossible();
370         FileHelpers.WriteFirst(_logAddress, _lastCommittedTransition);
371     }
372     catch
373     {
374     }
375 }
376
377 #region DisposalBase
378
379 protected override void Dispose(bool manual, bool wasDisposed)
380 {
381     if (!wasDisposed)
382     {
383         DisposeTransitions();
384     }
385     base.Dispose(manual, wasDisposed);
386 }
387
388 #endregion
389 }
390 }

```

#### ./Platform.Data.Doublets/Unicode/CharToUnicodeSymbolConverter.cs

```

1  using Platform.Interfaces;
2  using Platform.Numbers;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Unicode
7  {
8      public class CharToUnicodeSymbolConverter<TLink> : LinksOperatorBase<TLink>,
9          ⇨ IConverter<char, TLink>
10     {
11         private readonly IConverter<TLink> _addressToNumberConverter;
12         private readonly TLink _unicodeSymbolMarker;
13
14         public CharToUnicodeSymbolConverter(ILinks<TLink> links, IConverter<TLink>
15             ⇨ addressToNumberConverter, TLink unicodeSymbolMarker) : base(links)
16         {
17             _addressToNumberConverter = addressToNumberConverter;
18             _unicodeSymbolMarker = unicodeSymbolMarker;
19         }
20
21         public TLink Convert(char source)
22         {
23             var unaryNumber = _addressToNumberConverter.Convert((Integer<TLink>)source);
24             return Links.GetOrCreate(unaryNumber, _unicodeSymbolMarker);
25         }
26     }
27 }

```

#### ./Platform.Data.Doublets/Unicode/StringToUnicodeSequenceConverter.cs

```

1  using Platform.Data.Doublets.Sequences.Indexes;
2  using Platform.Interfaces;
3  using System.Collections.Generic;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  namespace Platform.Data.Doublets.Unicode
8  {
9      public class StringToUnicodeSequenceConverter<TLink> : LinksOperatorBase<TLink>,
10          ⇨ IConverter<string, TLink>
11     {
12         private readonly IConverter<char, TLink> _charToUnicodeSymbolConverter;
13         private readonly ISequenceIndex<TLink> _index;
14         private readonly IConverter<IList<TLink>, TLink> _listToSequenceLinkConverter;
15         private readonly TLink _unicodeSequenceMarker;
16
17         public StringToUnicodeSequenceConverter(ILinks<TLink> links, IConverter<char, TLink>
18             ⇨ charToUnicodeSymbolConverter, ISequenceIndex<TLink> index, IConverter<IList<TLink>,
19             ⇨ TLink> listToSequenceLinkConverter, TLink unicodeSequenceMarker) : base(links)

```

```

17     {
18         _charToUnicodeSymbolConverter = charToUnicodeSymbolConverter;
19         _index = index;
20         _listToSequenceLinkConverter = listToSequenceLinkConverter;
21         _unicodeSequenceMarker = unicodeSequenceMarker;
22     }
23
24     public TLink Convert(string source)
25     {
26         var elements = new List<TLink>();
27         for (int i = 0; i < source.Length; i++)
28         {
29             elements.Add(_charToUnicodeSymbolConverter.Convert(source[i]));
30         }
31         _index.Add(elements);
32         var sequence = _listToSequenceLinkConverter.Convert(elements);
33         return Links.GetOrCreate(sequence, _unicodeSequenceMarker);
34     }
35 }
36 }

```

./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  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Data.Doublets.Unicode
11 {
12     public class UnicodeMap
13     {
14         public static readonly ulong FirstCharLink = 1;
15         public static readonly ulong LastCharLink = FirstCharLink + char.MaxValue;
16         public static readonly ulong MapSize = 1 + char.MaxValue;
17
18         private readonly ILinks<ulong> _links;
19         private bool _initialized;
20
21         public UnicodeMap(ILinks<ulong> links) => _links = links;
22
23         public static UnicodeMap InitNew(ILinks<ulong> links)
24         {
25             var map = new UnicodeMap(links);
26             map.Init();
27             return map;
28         }
29
30         public void Init()
31         {
32             if (_initialized)
33             {
34                 return;
35             }
36             _initialized = true;
37             var firstLink = _links.CreatePoint();
38             if (firstLink != FirstCharLink)
39             {
40                 _links.Delete(firstLink);
41             }
42             else
43             {
44                 for (var i = FirstCharLink + 1; i <= LastCharLink; i++)
45                 {
46                     // From NIL to It (NIL -> Character) transformation meaning, (or infinite
47                     ↪ amount of NIL characters before actual Character)
48                     var createdLink = _links.CreatePoint();
49                     _links.Update(createdLink, firstLink, createdLink);
50                     if (createdLink != i)
51                     {
52                         throw new InvalidOperationException("Unable to initialize UTF 16
53                         ↪ table.");
54                     }
55                 }
56             }
57         }
58     }
59 }

```

```

57 // 0 - null link
58 // 1 - nil character (0 character)
59 // ...
60 // 65536 (0(1) + 65535 = 65536 possible values)
61
62 [MethodImpl(MethodImplOptions.AggressiveInlining)]
63 public static ulong FromCharToLink(char character) => (ulong)character + 1;
64
65 [MethodImpl(MethodImplOptions.AggressiveInlining)]
66 public static char FromLinkToChar(ulong link) => (char)(link - 1);
67
68 [MethodImpl(MethodImplOptions.AggressiveInlining)]
69 public static bool IsCharLink(ulong link) => link <= MapSize;
70
71 public static string FromLinksToString(IList<ulong> linksList)
72 {
73     var sb = new StringBuilder();
74     for (int i = 0; i < linksList.Count; i++)
75     {
76         sb.Append(FromLinkToChar(linksList[i]));
77     }
78     return sb.ToString();
79 }
80
81 public static string FromSequenceLinkToString(ulong link, ILinks<ulong> links)
82 {
83     var sb = new StringBuilder();
84     if (links.Exists(link))
85     {
86         StopableSequenceWalker.WalkRight(link, links.GetSource, links.GetTarget,
87             x => x <= MapSize || links.GetSource(x) == x || links.GetTarget(x) == x,
88             ↪ element =>
89             {
90                 sb.Append(FromLinkToChar(element));
91                 return true;
92             }
93         );
94     }
95     return sb.ToString();
96
97 public static ulong[] FromCharsToLinkArray(char[] chars) => FromCharsToLinkArray(chars,
98     ↪ chars.Length);
99
100 public static ulong[] FromCharsToLinkArray(char[] chars, int count)
101 {
102     // char array to ulong array
103     var linksSequence = new ulong[count];
104     for (var i = 0; i < count; i++)
105     {
106         linksSequence[i] = FromCharToLink(chars[i]);
107     }
108     return linksSequence;
109 }
110
111 public static ulong[] FromStringToLinkArray(string sequence)
112 {
113     // char array to ulong array
114     var linksSequence = new ulong[sequence.Length];
115     for (var i = 0; i < sequence.Length; i++)
116     {
117         linksSequence[i] = FromCharToLink(sequence[i]);
118     }
119     return linksSequence;
120 }
121
122 public static List<ulong[]> FromStringToLinkArrayGroups(string sequence)
123 {
124     var result = new List<ulong[]>();
125     var offset = 0;
126     while (offset < sequence.Length)
127     {
128         var currentCategory = CharUnicodeInfo.GetUnicodeCategory(sequence[offset]);
129         var relativeLength = 1;
130         var absoluteLength = offset + relativeLength;
131         while (absoluteLength < sequence.Length &&
132             currentCategory ==
133             ↪ CharUnicodeInfo.GetUnicodeCategory(sequence[absoluteLength]))
134         {
135             relativeLength++;
136         }
137     }
138 }

```

```

133         absoluteLength++;
134     }
135     // char array to ulong array
136     var innerSequence = new ulong[relativeLength];
137     var maxLength = offset + relativeLength;
138     for (var i = offset; i < maxLength; i++)
139     {
140         innerSequence[i - offset] = FromCharToLink(sequence[i]);
141     }
142     result.Add(innerSequence);
143     offset += relativeLength;
144 }
145 return result;
146 }
147
148 public static List<ulong[]> FromLinkArrayToLinkArrayGroups(ulong[] array)
149 {
150     var result = new List<ulong[]>();
151     var offset = 0;
152     while (offset < array.Length)
153     {
154         var relativeLength = 1;
155         if (array[offset] <= LastCharLink)
156         {
157             var currentCategory =
158                 CharUnicodeInfo.GetUnicodeCategory(FromLinkToChar(array[offset]));
159             var absoluteLength = offset + relativeLength;
160             while (absoluteLength < array.Length &&
161                 array[absoluteLength] <= LastCharLink &&
162                 currentCategory == CharUnicodeInfo.GetUnicodeCategory(FromLinkToChar(
163                     array[absoluteLength])))
164             {
165                 relativeLength++;
166                 absoluteLength++;
167             }
168         }
169         else
170         {
171             var absoluteLength = offset + relativeLength;
172             while (absoluteLength < array.Length && array[absoluteLength] > LastCharLink)
173             {
174                 relativeLength++;
175                 absoluteLength++;
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     }
188     return result;
189 }

```

./Platform.Data.Doublets/Unicode/UnicodeSequenceCriterionMatcher.cs

```

1 using Platform.Interfaces;
2 using System.Collections.Generic;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Doublets.Unicode
7 {
8     public class UnicodeSequenceCriterionMatcher<TLink> : LinksOperatorBase<TLink>,
9         ↳ ICriterionMatcher<TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
12             ↳ EqualityComparer<TLink>.Default;
13         private readonly TLink _unicodeSequenceMarker;
14         public UnicodeSequenceCriterionMatcher(ILinks<TLink> links, TLink unicodeSequenceMarker)
15             ↳ : base(links) => _unicodeSequenceMarker = unicodeSequenceMarker;
16         public bool IsMatched(TLink link) => _equalityComparer.Equals(links.GetTarget(link),
17             ↳ _unicodeSequenceMarker);
18     }
19 }

```



```
15 }
```

```
./Platform.Data.Doublets/Unicode/UnicodeSequenceToStringConverter.cs
```

```
1 using System;
2 using System.Linq;
3 using Platform.Data.Doublets.Sequences.Walkers;
4 using Platform.Interfaces;
5
6 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8 namespace Platform.Data.Doublets.Unicode
9 {
10     public class UnicodeSequenceToStringConverter<TLink> : LinksOperatorBase<TLink>,
11         ↪ IConverter<TLink, string>
12     {
13         private readonly ICriterionMatcher<TLink> _unicodeSequenceCriterionMatcher;
14         private readonly ISequenceWalker<TLink> _sequenceWalker;
15         private readonly IConverter<TLink, char> _unicodeSymbolToCharConverter;
16
17         public UnicodeSequenceToStringConverter(ILinks<TLink> links, ICriterionMatcher<TLink>
18             ↪ unicodeSequenceCriterionMatcher, ISequenceWalker<TLink> sequenceWalker,
19             ↪ IConverter<TLink, char> unicodeSymbolToCharConverter) : base(links)
20         {
21             _unicodeSequenceCriterionMatcher = unicodeSequenceCriterionMatcher;
22             _sequenceWalker = sequenceWalker;
23             _unicodeSymbolToCharConverter = unicodeSymbolToCharConverter;
24         }
25
26         public string Convert(TLink source)
27         {
28             if(!_unicodeSequenceCriterionMatcher.IsMatched(source))
29             {
30                 throw new ArgumentOutOfRangeException(nameof(source), source, "Specified link is
31                     ↪ not a unicode sequence.");
32             }
33             var sequence = Links.GetSource(source);
34             var charArray = _sequenceWalker.Walk(sequence).Select(_unicodeSymbolToCharConverter.
35                 ↪ Convert).ToArray();
36             return new string(charArray);
37         }
38     }
39 }
```

```
./Platform.Data.Doublets/Unicode/UnicodeSymbolCriterionMatcher.cs
```

```
1 using Platform.Interfaces;
2 using System.Collections.Generic;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Doublets.Unicode
7 {
8     public class UnicodeSymbolCriterionMatcher<TLink> : LinksOperatorBase<TLink>,
9         ↪ ICriterionMatcher<TLink>
10     {
11         private static readonly EqualityComparer<TLink> _equalityComparer =
12             ↪ EqualityComparer<TLink>.Default;
13         private readonly TLink _unicodeSymbolMarker;
14         public UnicodeSymbolCriterionMatcher(ILinks<TLink> links, TLink unicodeSymbolMarker) :
15             ↪ base(links) => _unicodeSymbolMarker = unicodeSymbolMarker;
16         public bool IsMatched(TLink link) => _equalityComparer.Equals(Links.GetTarget(link),
17             ↪ _unicodeSymbolMarker);
18     }
19 }
```

```
./Platform.Data.Doublets/Unicode/UnicodeSymbolToCharConverter.cs
```

```
1 using System;
2 using Platform.Interfaces;
3 using Platform.Numbers;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Data.Doublets.Unicode
8 {
9     public class UnicodeSymbolToCharConverter<TLink> : LinksOperatorBase<TLink>,
10         ↪ IConverter<TLink, char>
11     {
12         private readonly IConverter<TLink> _numberToAddressConverter;
13         private readonly ICriterionMatcher<TLink> _unicodeSymbolCriterionMatcher;
```

```

14     public UnicodeSymbolToCharConverter(ILinks<TLink> links, IConverter<TLink>
        ↳ numberToAddressConverter, ICriterionMatcher<TLink> unicodeSymbolCriterionMatcher) :
        ↳ base(links)
15     {
16         _numberToAddressConverter = numberToAddressConverter;
17         _unicodeSymbolCriterionMatcher = unicodeSymbolCriterionMatcher;
18     }
19
20     public char Convert(TLink source)
21     {
22         if (!_unicodeSymbolCriterionMatcher.IsMatched(source))
23         {
24             throw new ArgumentOutOfRangeException(nameof(source), source, "Specified link is
        ↳ not a unicode symbol.");
25         }
26         return (char)(ushort)(Integer<TLink>)_numberToAddressConverter.Convert(Links.GetSource(
        ↳ source));
27     }
28 }
29 }

```

./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                 scope.Use<ILinks<ulong>>().TestCRUDOperations();
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                 [Fact]
33                 public static void UInt16CRUDTest()
34                 {
35                     using (var scope = new Scope<Types<HeapResizableDirectMemory,
36                     ↳ ResizableDirectMemoryLinks<ushort>>>())
37                     {
38                         scope.Use<ILinks<ushort>>().TestCRUDOperations();
39                     }
40
41                     [Fact]
42                     public static void UInt8CRUDTest()
43                     {
44                         using (var scope = new Scope<Types<HeapResizableDirectMemory,
45                         ↳ ResizableDirectMemoryLinks<byte>>>())
46                         {
47                             scope.Use<ILinks<byte>>().TestCRUDOperations();
48                         }
49
50                     private static void TestCRUDOperations<T>(this ILinks<T> links)
51                     {
52                         var constants = links.Constants;
53
54                         var equalityComparer = EqualityComparer<T>.Default;
55
56                         // Create Link
57                         Assert.True(equalityComparer.Equals(links.Count(), Integer<T>.Zero));
58
59                         var setter = new Setter<T>(constants.Null);
60                         links.Each(constants.Any, constants.Any, setter.SetAndReturnTrue);
61
62                         Assert.True(equalityComparer.Equals(setter.Result, constants.Null));
63

```

```

64     var linkAddress = links.Create();
65
66     var link = new Link<T>(links.GetLink(linkAddress));
67
68     Assert.True(link.Count == 3);
69     Assert.True(equalityComparer.Equals(link.Index, linkAddress));
70     Assert.True(equalityComparer.Equals(link.Source, constants.Null));
71     Assert.True(equalityComparer.Equals(link.Target, constants.Null));
72
73     Assert.True(equalityComparer.Equals(links.Count(), Integer<T>.One));
74
75     // Get first link
76     setter = new Setter<T>(constants.Null);
77     links.Each(constants.Any, constants.Any, setter.SetAndReturnFalse);
78
79     Assert.True(equalityComparer.Equals(setter.Result, linkAddress));
80
81     // Update link to reference itself
82     links.Update(linkAddress, linkAddress, linkAddress);
83
84     link = new Link<T>(links.GetLink(linkAddress));
85
86     Assert.True(equalityComparer.Equals(link.Source, linkAddress));
87     Assert.True(equalityComparer.Equals(link.Target, linkAddress));
88
89     // Update link to reference null (prepare for delete)
90     var updated = links.Update(linkAddress, constants.Null, constants.Null);
91
92     Assert.True(equalityComparer.Equals(updated, linkAddress));
93
94     link = new Link<T>(links.GetLink(linkAddress));
95
96     Assert.True(equalityComparer.Equals(link.Source, constants.Null));
97     Assert.True(equalityComparer.Equals(link.Target, constants.Null));
98
99     // Delete link
100    links.Delete(linkAddress);
101
102    Assert.True(equalityComparer.Equals(links.Count(), Integer<T>.Zero));
103
104    setter = new Setter<T>(constants.Null);
105    links.Each(constants.Any, constants.Any, setter.SetAndReturnTrue);
106
107    Assert.True(equalityComparer.Equals(setter.Result, constants.Null));
108}
109
110[Fact]
111public static void UInt64RawNumbersCRUDTest()
112{
113    using (var scope = new Scope<Types<HeapResizableDirectMemory,
114        ↳ ResizableDirectMemoryLinks<ulong>>>())
115    {
116        scope.Use<ILinks<ulong>>().TestRawNumbersCRUDOperations();
117    }
118}
119
120[Fact]
121public static void UInt32RawNumbersCRUDTest()
122{
123    using (var scope = new Scope<Types<HeapResizableDirectMemory,
124        ↳ ResizableDirectMemoryLinks<uint>>>())
125    {
126        scope.Use<ILinks<uint>>().TestRawNumbersCRUDOperations();
127    }
128}
129
130[Fact]
131public static void UInt16RawNumbersCRUDTest()
132{
133    using (var scope = new Scope<Types<HeapResizableDirectMemory,
134        ↳ ResizableDirectMemoryLinks<ushort>>>())
135    {
136        scope.Use<ILinks<ushort>>().TestRawNumbersCRUDOperations();
137    }
138}
139
140[Fact]
141public static void UInt8RawNumbersCRUDTest()
142{

```

```

140     using (var scope = new Scope<Types<HeapResizableDirectMemory,
141         ↳ ResizableDirectMemoryLinks<byte>>>())
142     {
143         scope.Use<ILinks<byte>>().TestRawNumbersCRUDOperations();
144     }
145 }
146 private static void TestRawNumbersCRUDOperations<T>(this ILinks<T> links)
147 {
148     // Constants
149     var constants = links.Constants;
150     var equalityComparer = EqualityComparer<T>.Default;
151
152     var h106E = new Hybrid<T>(106L, isExternal: true);
153     var h107E = new Hybrid<T>(-char.ConvertFromUtf32(107)[0]);
154     var h108E = new Hybrid<T>(-108L);
155
156     Assert.Equal(106L, h106E.AbsoluteValue);
157     Assert.Equal(107L, h107E.AbsoluteValue);
158     Assert.Equal(108L, h108E.AbsoluteValue);
159
160     // Create Link (External -> External)
161     var linkAddress1 = links.Create();
162
163     links.Update(linkAddress1, h106E, h108E);
164
165     var link1 = new Link<T>(links.GetLink(linkAddress1));
166
167     Assert.True(equalityComparer.Equals(link1.Source, h106E));
168     Assert.True(equalityComparer.Equals(link1.Target, h108E));
169
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 EqualsPerfomanceTest()
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                     var created = 0;

```

```

45         var deleted = 0;
46
47         for (var i = 0; i < N; i++)
48         {
49             var linksCount = links.Count();
50
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 [Fact]
89 public static void CascadeUpdateTest()
90 {
91     var itself = _constants.Itself;
92
93     using (var scope = new TempLinksTestScope(useLog: true))
94     {
95         var links = scope.Links;
96
97         var l1 = links.Create();
98         var l2 = links.Create();
99
100         l2 = links.Update(l2, l2, l1, l2);
101
102         links.CreateAndUpdate(l2, itself);
103         links.CreateAndUpdate(l2, itself);
104
105         l2 = links.Update(l2, l1);
106
107         links.Delete(l2);
108
109         Global.Trash = links.Count();
110
111         links.Unsync.DisposeIfPossible(); // Close links to access log
112
113         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scope
114             ↪ 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(0UL, links.Count());
154
155         links.Unsync.DisposeIfPossible();
156
157         var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(s
        ↪ cope.TempTransactionLogFilename);
158         Assert.Single(transitions);
159     }
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,
        ↪ useLog: true))
172         {
173             var links = scope.Links;
174             var transactionsLayer = (UInt64LinksTransactionsLayer)((LinksDisposableDecor
        ↪ atorBase<ulong>)links.Unsync).Links;
175             using (var transaction = transactionsLayer.BeginTransaction())
176             {
177                 var l1 = links.CreateAndUpdate(itself, itself);
178                 var l2 = links.CreateAndUpdate(itself, itself);
179
180                 l2 = links.Update(l2, l2, l1, l2);
181
182                 links.CreateAndUpdate(l2, itself);
183                 links.CreateAndUpdate(l2, itself);
184
185                 //Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transi
        ↪ tion>(scope.TempTransactionLogFilename);
186
187                 l2 = links.Update(l2, l1);
188
189                 links.Delete(l2);
190
191                 ExceptionThrower();
192
193                 transaction.Commit();
194             }
195
196             Global.Trash = links.Count();
197         }

```

```

198     }
199     catch
200     {
201         Assert.False(lastScope == null);
202
203         var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(1,
204             ↳ astScope.TempTransactionLogFilename);
205
206         Assert.True(transitions.Length == 1 && transitions[0].Before.IsNull() &&
207             ↳ transitions[0].After.IsNull());
208
209         lastScope.DeleteFiles();
210     }
211 }
212
213 [Fact]
214 public static void TransactionUserCodeErrorSomeDataSavedTest()
215 {
216     // User Code Error (Autoreverted), some data saved
217     var itself = _constants.Itself;
218
219     TempLinksTestScope lastScope = null;
220     try
221     {
222         {
223             ulong l1;
224             ulong l2;
225
226             using (var scope = new TempLinksTestScope(useLog: true))
227             {
228                 var links = scope.Links;
229                 l1 = links.CreateAndUpdate(itself, itself);
230                 l2 = links.CreateAndUpdate(itself, itself);
231
232                 l2 = links.Update(l2, l2, l1, l2);
233
234                 links.CreateAndUpdate(l2, itself);
235                 links.CreateAndUpdate(l2, itself);
236
237                 links.Unsync.DisposeIfPossible();
238
239                 Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(1,
240                     ↳ scope.TempTransactionLogFilename);
241             }
242
243             using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
244                 ↳ useLog: true))
245             {
246                 var links = scope.Links;
247                 var transactionsLayer = (UInt64LinksTransactionsLayer)links.Unsync;
248                 using (var transaction = transactionsLayer.BeginTransaction())
249                 {
250                     l2 = links.Update(l2, l1);
251
252                     links.Delete(l2);
253
254                     ExceptionThrower();
255
256                     transaction.Commit();
257                 }
258
259                 Global.Trash = links.Count();
260             }
261         }
262     }
263     catch
264     {
265         Assert.False(lastScope == null);
266
267         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(last,
268             ↳ Scope.TempTransactionLogFilename);
269
270         lastScope.DeleteFiles();
271     }
272 }
273
274 [Fact]
275 public static void TransactionCommit()
276 {
277     var itself = _constants.Itself;
278
279     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
343         ↪ yet.");
344     }
345     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran_
346     ↪ sactionLogFilename);
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
366         ↪ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
367         ↪ tempTransactionLogFilename))
368         using (var links = new UInt64Links(memoryAdapter))
369         {
370             l1 = links.CreateAndUpdate(itself, itself);
371             l2 = links.CreateAndUpdate(itself, itself);
372
373             l2 = links.Update(l2, l2, l1, l2);
374
375             links.CreateAndUpdate(l2, itself);
376             links.CreateAndUpdate(l2, itself);
377         }
378
379         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(temp_
380         ↪ TransactionLogFilename);
381
382         using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
383         ↪ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
384         ↪ tempTransactionLogFilename))
385         using (var links = new UInt64Links(memoryAdapter))
386         {
387             using (var transaction = memoryAdapter.BeginTransaction())
388             {
389                 l2 = links.Update(l2, l1);
390
391                 links.Delete(l2);
392
393                 ExceptionThrower();
394
395                 transaction.Commit();
396             }
397
398             Global.Trash = links.Count();
399         }
400     }
401     catch
402     {
403         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(temp_
404         ↪ TransactionLogFilename);
405     }
406
407     File.Delete(tempDatabaseFilename);
408     File.Delete(tempTransactionLogFilename);
409 }
410
411 private static void ExceptionThrower()
412 {
413     throw new Exception();
414 }
415
416 [Fact]
417 public static void PathsTest()
418 {

```

```

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
470 private static void DefaultFormatter(StringBuilder sb, ulong link)
471 {
472     sb.Append(link.ToString());
473 }
474
475 #endregion
476
477 #region Performance
478
479 /*
480 public static void RunAllPerformanceTests()
481 {
482     try
483     {
484         links.TestLinksInSteps();
485     }
486     catch (Exception ex)
487     {
488         ex.WriteToConsole();
489     }
490 }
491
492 return;

```

```

485     try
486     {
487         //ThreadPool.SetMaxThreads(2, 2);
488
489         // Запускаем все тесты дважды, чтобы первоначальная инициализация не повлияла на
490         ↪ результат
491         // Также это дополнительно помогает в отладке
492         // Увеличивает вероятность попадания информации в кэши
493         for (var i = 0; i < 10; i++)
494         {
495             //0 - 10 ГБ
496             //Каждые 100 МБ срез цифр
497
498             //links.TestGetSourceFunction();
499             //links.TestGetSourceFunctionInParallel();
500             //links.TestGetTargetFunction();
501             //links.TestGetTargetFunctionInParallel();
502             links.Create64BillionLinks();
503
504             links.TestRandomSearchFixed();
505             //links.Create64BillionLinksInParallel();
506             links.TestEachFunction();
507             //links.TestForeach();
508             //links.TestParallelForeach();
509         }
510
511         links.TestDeletionOfAllLinks();
512
513     }
514     catch (Exception ex)
515     {
516         ex.WriteToConsole();
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 /
527     ↪ Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
528     var linksStep = 102 * mebibyte /
529     ↪ Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
530
531     var creationMeasurements = new List<TimeSpan>();
532     var searchMeasurements = new List<TimeSpan>();
533     var deletionMeasurements = new List<TimeSpan>();
534
535     GetBaseRandomLoopOverhead(linksStep);
536     GetBaseRandomLoopOverhead(linksStep);
537
538     var stepLoopOverhead = GetBaseRandomLoopOverhead(linksStep);
539
540     ConsoleHelpers.Debug("Step loop overhead: {0}.", stepLoopOverhead);
541
542     var loops = totalLinksToCreate / linksStep;
543
544     for (int i = 0; i < loops; i++)
545     {
546         creationMeasurements.Add(Measure(() => links.RunRandomCreations(linksStep)));
547         searchMeasurements.Add(Measure(() => links.RunRandomSearches(linksStep)));
548
549         Console.WriteLine("\rC + S {0}/{1}", i + 1, loops);
550     }
551
552     ConsoleHelpers.Debug();
553
554     for (int i = 0; i < loops; i++)
555     {
556         deletionMeasurements.Add(Measure(() => links.RunRandomDeletions(linksStep)));
557
558         Console.WriteLine("\rD {0}/{1}", i + 1, loops);
559     }
560
561     ConsoleHelpers.Debug();
562     ConsoleHelpers.Debug("C S D");

```

```

562         for (int i = 0; i < loops; i++)
563         {
564             ConsoleHelpers.Debug("{0} {1} {2}", creationMeasurements[i],
565             ↪ 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,
573             ↪ searchMeasurements[i] - stepLoopOverhead, deletionMeasurements[i] - stepLoopOverhead);
574         }
575
576         ConsoleHelpers.Debug("All tests done. Total links left in database: {0}.",
577         ↪ links.Total);
578     }
579
580     private static void CreatePoints(this Platform.Links.Data.Core.Doublets.Links links, long
581     ↪ amountToCreate)
582     {
583         for (long i = 0; i < amountToCreate; i++)
584             links.Create(0, 0);
585     }
586
587     private static TimeSpan GetBaseRandomLoopOverhead(long loops)
588     {
589         return Measure(() =>
590         {
591             ulong maxValue = RandomHelpers.DefaultFactory.NextUInt64();
592             ulong result = 0;
593             for (long i = 0; i < loops; i++)
594             {
595                 var source = RandomHelpers.DefaultFactory.NextUInt64(maxValue);
596                 var target = RandomHelpers.DefaultFactory.NextUInt64(maxValue);
597
598                 result += maxValue + source + target;
599             }
600             Global.Trash = result;
601         });
602     }
603     */
604
605     [Fact(Skip = "performance test")]
606     public static void GetSourceTest()
607     {
608         using (var scope = new TempLinksTestScope())
609         {
610             var links = scope.Links;
611             ConsoleHelpers.Debug("Testing GetSource function with {0} Iterations.",
612             ↪ Iterations);
613
614             ulong counter = 0;
615
616             //var firstLink = links.First();
617             // Создаём одну связь, из которой будет производить считывание
618             var firstLink = links.Create();
619
620             var sw = Stopwatch.StartNew();
621
622             // Тестируем саму функцию
623             for (ulong i = 0; i < Iterations; i++)
624             {
625                 counter += links.GetSource(firstLink);
626             }
627
628             var elapsedTime = sw.Elapsed;
629
630             var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
631
632             // Удаляем связь, из которой производилось считывание
633             links.Delete(firstLink);
634
635             ConsoleHelpers.Debug(
636                 "{0} Iterations of GetSource function done in {1} ({2} Iterations per
637                 ↪ second), counter result: {3}",
638                 Iterations, elapsedTime, (long)iterationsPerSecond, counter);
639         }
640     }

```

```

635 }
636
637 [Fact(Skip = "performance test")]
638 public static void GetSourceInParallel()
639 {
640     using (var scope = new TempLinksTestScope())
641     {
642         var links = scope.Links;
643         ConsoleHelpers.Debug("Testing GetSource function with {0} Iterations in
        ↪ parallel.", Iterations);
644
645         long counter = 0;
646
647         //var firstLink = links.First();
648         var firstLink = links.Create();
649
650         var sw = Stopwatch.StartNew();
651
652         // Тестируем саму функцию
653         Parallel.For(0, Iterations, x =>
654         {
655             Interlocked.Add(ref counter, (long)links.GetSource(firstLink));
656             //Interlocked.Increment(ref counter);
657         });
658
659         var elapsedTime = sw.Elapsed;
660
661         var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
662
663         links.Delete(firstLink);
664
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     long counter = 0;
711
712     //var firstLink = links.First();
713     var firstLink = links.Create();
714
715     var sw = Stopwatch.StartNew();
716
717     Parallel.For(0, Iterations, x =>
718     {
719         Interlocked.Add(ref counter, (long)links.GetTarget(firstLink));
720         //Interlocked.Increment(ref counter);
721     });
722
723     var elapsedTime = sw.Elapsed;
724
725     var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
726
727     links.Delete(firstLink);
728
729     ConsoleHelpers.Debug(
730         "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
731         ↳ 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,
744     ↳ DefaultLinksSizeStep))
745     {
746         long iterations = 64 * 1024 * 1024 /
747     ↳ Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
748
749         ulong counter = 0;
750         var maxLink = links.Total;
751
752         ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.", iterations);
753
754         var sw = Stopwatch.StartNew();
755
756         for (var i = iterations; i > 0; i--)
757         {
758             var source =
759     ↳ RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
760             var target =
761     ↳ RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
762
763             counter += links.Search(source, target);
764         }
765
766         var elapsedTime = sw.Elapsed;
767
768         var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;
769
770         ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
771     ↳ Iterations per second), c: {3}", iterations, elapsedTime, (long)iterationsPerSecond,
772     ↳ counter);
773     }
774
775     File.Delete(tempFilename);
776 }*/
777
778 [Fact(Skip = "useless: O(0), was dependent on creation tests")]
779 public static void TestRandomSearchAll()
780 {
781     using (var scope = new TempLinksTestScope())
782     {
783         var links = scope.Links;
784         ulong counter = 0;
785
786         var maxLink = links.Count();
787
788         var iterations = links.Count();

```

```

783     ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.",
784         ↪ 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,
791             ↪ maxLink);
792
793         var source = RandomHelpers.Default.NextUInt64(linksAddressRange);
794         var target = RandomHelpers.Default.NextUInt64(linksAddressRange);
795
796         counter += links.SearchOrDefault(source, target);
797     }
798
799     var elapsedTime = sw.Elapsed;
800     var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;
801
802     ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
803         ↪ Iterations per second), c: {3}",
804         iterations, elapsedTime, (long)iterationsPerSecond, counter);
805 }
806
807 [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
808 public static void TestEach()
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}
827             ↪ links per second)",
828             counter, elapsedTime, (long)linksPerSecond);
829     }
830
831     /*
832     [Fact]
833     public static void TestForeach()
834     {
835         var tempFilename = Path.GetTempFileName();
836
837         using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
838             ↪ DefaultLinksSizeStep))
839         {
840             ulong counter = 0;
841
842             ConsoleHelpers.Debug("Testing foreach through links.");
843
844             var sw = Stopwatch.StartNew();
845
846             //foreach (var link in links)
847             //{
848                 counter++;
849             //}
850
851             var elapsedTime = sw.Elapsed;
852
853             var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
854
855             ConsoleHelpers.Debug("{0} Iterations of Foreach's handler block done in {1} ({2}
856                 ↪ links per second)", counter, elapsedTime, (long)linksPerSecond);

```

```

857         File.Delete(tempFilename);
858     }
859     */
860
861     /*
862     [Fact]
863     public static void TestParallelForeach()
864     {
865         var tempFilename = Path.GetTempFileName();
866
867         using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
↵ DefaultLinksSizeStep))
868         {
869             long counter = 0;
870
871             ConsoleHelpers.Debug("Testing parallel foreach through links.");
872
873             var sw = Stopwatch.StartNew();
874
875             //Parallel.ForEach((IEnumerable<ulong>)links, x =>
876             //{
877             //    Interlocked.Increment(ref counter);
878             //});
879
880             var elapsedTime = sw.Elapsed;
881
882             var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
883
884             ConsoleHelpers.Debug("{0} Iterations of Parallel Foreach's handler block done in
↵ {1} ({2} links per second)", counter, elapsedTime, (long)linksPerSecond);
885         }
886
887         File.Delete(tempFilename);
888     }
889     */
890
891     [Fact(Skip = "performance test")]
892     public static void Create64BillionLinks()
893     {
894         using (var scope = new TempLinksTestScope())
895         {
896             var links = scope.Links;
897             var linksBeforeTest = links.Count();
898
899             long linksToCreate = 64 * 1024 * 1024 /
↵ UInt64ResizableDirectMemoryLinks.LinkSizeInBytes;
900
901             ConsoleHelpers.Debug("Creating {0} links.", linksToCreate);
902
903             var elapsedTime = Performance.Measure(() =>
904             {
905                 for (long i = 0; i < linksToCreate; i++)
906                 {
907                     links.Create();
908                 }
909             });
910
911             var linksCreated = links.Count() - linksBeforeTest;
912             var linksPerSecond = linksCreated / elapsedTime.TotalSeconds;
913
914             ConsoleHelpers.Debug("Current links count: {0}.", links.Count());
915
916             ConsoleHelpers.Debug("{0} links created in {1} ({2} links per second)",
↵ linksCreated, elapsedTime,
917                 (long)linksPerSecond);
918         }
919     }
920
921     [Fact(Skip = "performance test")]
922     public static void Create64BillionLinksInParallel()
923     {
924         using (var scope = new TempLinksTestScope())
925         {
926             var links = scope.Links;
927             var linksBeforeTest = links.Count();
928
929             var sw = Stopwatch.StartNew();
930
931

```

```

932     long linksToCreate = 64 * 1024 * 1024 /
933         ↳ UInt64ResizableDirectMemoryLinks.LinkSizeInBytes;
934
935     ConsoleHelpers.Debug("Creating {0} links in parallel.", linksToCreate);
936
937     Parallel.For(0, linksToCreate, x => links.Create());
938
939     var elapsedTime = sw.Elapsed;
940
941     var linksCreated = links.Count() - linksBeforeTest;
942     var linksPerSecond = linksCreated / elapsedTime.TotalSeconds;
943
944     ConsoleHelpers.Debug("{0} links created in {1} ({2} links per second)",
945         ↳ linksCreated, elapsedTime,
946         (long)linksPerSecond);
947 }
948
949 [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
950 public static void TestDeletionOfAllLinks()
951 {
952     using (var scope = new TempLinksTestScope())
953     {
954         var links = scope.Links;
955         var linksBeforeTest = links.Count();
956
957         ConsoleHelpers.Debug("Deleting all links");
958
959         var elapsedTime = Performance.Measure(links.DeleteAll);
960
961         var linksDeleted = linksBeforeTest - links.Count();
962         var linksPerSecond = linksDeleted / elapsedTime.TotalSeconds;
963
964         ConsoleHelpers.Debug("{0} links deleted in {1} ({2} links per second)",
965             ↳ linksDeleted, elapsedTime,
966             (long)linksPerSecond);
967     }
968 }
969 #endregion
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.Numbers.Unary;
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 unaryNumberToAddressConverter = 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         ↪ unaryNumberToAddressConverter);
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 [Fact]
66 public static void DictionaryBasedFrequencyStoredOptimalVariantSequenceTest()
67 {
68     using (var scope = new TempLinksTestScope(useSequences: false))
69     {
70         var links = scope.Links;
71
72         links.UseUnicode();
73
74         var sequence = UnicodeMap.FromStringToLinkArray(SequenceExample);
75
76         var linksToFrequencies = new Dictionary<ulong, ulong>();
77
78         var totalSequenceSymbolFrequencyCounter = new
79             ↪ TotalSequenceSymbolFrequencyCounter<ulong>(links);
80
81         var linkFrequenciesCache = new LinkFrequenciesCache<ulong>(links,
82             ↪ totalSequenceSymbolFrequencyCounter);
83
84         var index = new
85             ↪ CachedFrequencyIncrementingSequenceIndex<ulong>(linkFrequenciesCache);
86         var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFreque
87             ↪ ncyNumberConverter<ulong>(linkFrequenciesCache);
88
89         var sequenceToItsLocalElementLevelsConverter = new
90             ↪ SequenceToItsLocalElementLevelsConverter<ulong>(links,
91             ↪ linkToItsFrequencyNumberConverter);
92         var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
93             ↪ sequenceToItsLocalElementLevelsConverter);
94
95         var sequences = new Sequences.Sequences(links, new SequencesOptions<ulong>() {
96             ↪ Walker = new LeveledSequenceWalker<ulong>(links) });
97
98         ExecuteTest(sequences, sequence, sequenceToItsLocalElementLevelsConverter,
99             ↪ index, optimalVariantConverter);
100     }
101 }
102
103 private static void ExecuteTest(Sequences.Sequences sequences, ulong[] sequence,
104     ↪ SequenceToItsLocalElementLevelsConverter<ulong>
105     ↪ sequenceToItsLocalElementLevelsConverter, ISequenceIndex<ulong> index,
106     ↪ OptimalVariantConverter<ulong> optimalVariantConverter)
107 {
108     index.Add(sequence);
109
110     var optimalVariant = optimalVariantConverter.Convert(sequence);
111
112     var readSequence1 = sequences.ToList(optimalVariant);
113 }

```

```

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() {
24                     ↪ Walker = new LeveledSequenceWalker(links) });;;
25
26                 var sequence = new ulong[sequenceLength];
27                 for (var i = 0; i < sequenceLength; i++)
28                 {
29                     sequence[i] = links.Create();
30                 }
31
32                 var balancedVariantConverter = new BalancedVariantConverter(links);
33
34                 var sw1 = Stopwatch.StartNew();
35                 var balancedVariant = balancedVariantConverter.Convert(sequence); sw1.Stop();
36
37                 var sw2 = Stopwatch.StartNew();
38                 var readSequence1 = sequences.ToList(balancedVariant); sw2.Stop();
39
40                 var sw3 = Stopwatch.StartNew();
41                 var readSequence2 = new List();
42                 SequenceWalker.WalkRight(balancedVariant,
43                                         links.GetSource,
44                                         links.GetTarget,
45                                         links.IsPartialPoint,
46                                         readSequence2.Add);
47
48                 sw3.Stop();
49
50                 Assert.True(sequence.SequenceEqual(readSequence1));
51                 Assert.True(sequence.SequenceEqual(readSequence2));
52
53                 // Assert.True(sw2.Elapsed < sw3.Elapsed);
54
55                 Console.WriteLine($"Stack-based walker: {sw3.Elapsed}, Level-based reader:
56                     ↪ {sw2.Elapsed}");
57
58                 for (var i = 0; i < sequenceLength; i++)
59                 {
60                     links.Delete(sequence[i]);
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)
37                 {
38                     var link = memoryAdapter.Create();
39                     memoryAdapter.Delete(link);
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                     }
67                 }
68             }
69         }
70     }
71 }

```

./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);
48                 sw1.Stop();
49             }
50         }
51     }
52 }

```



```

47
48     var sw2 = Stopwatch.StartNew();
49     var results2 = sequences.CreateAllVariants2(sequence); sw2.Stop();
50
51     Assert.True(results1.Count > results2.Length);
52     Assert.True(sw1.Elapsed > sw2.Elapsed);
53
54     for (var i = 0; i < sequenceLength; i++)
55     {
56         links.Delete(sequence[i]);
57     }
58
59     Assert.True(links.Count() == 0);
60 }
61 }
62
63 [Fact]
64 //public void CUDTest()
65 //{
66 //    var tempFilename = Path.GetTempFileName();
67 //
68 //    const long sequenceLength = 8;
69 //
70 //    const ulong itself = LinksConstants.Itself;
71 //
72 //    using (var memoryAdapter = new ResizableDirectMemoryLinks(tempFilename,
73 //        ↪ DefaultLinksSizeStep))
74 //    using (var links = new Links(memoryAdapter))
75 //    {
76 //        var sequence = new ulong[sequenceLength];
77 //        for (var i = 0; i < sequenceLength; i++)
78 //            sequence[i] = links.Create(itself, itself);
79 //
80 //        SequencesOptions o = new SequencesOptions();
81 //
82 //        // TODO: Из числа в bool значения o.UseSequenceMarker = ((value & 1) != 0)
83 //        o.
84 //
85 //        var sequences = new Sequences(links);
86 //
87 //        var sw1 = Stopwatch.StartNew();
88 //        var results1 = sequences.CreateAllVariants1(sequence); sw1.Stop();
89 //
90 //        var sw2 = Stopwatch.StartNew();
91 //        var results2 = sequences.CreateAllVariants2(sequence); sw2.Stop();
92 //
93 //        Assert.True(results1.Count > results2.Length);
94 //        Assert.True(sw1.Elapsed > sw2.Elapsed);
95 //
96 //        for (var i = 0; i < sequenceLength; i++)
97 //            links.Delete(sequence[i]);
98 //    }
99 //
100 //    File.Delete(tempFilename);
101 //}
102
103 [Fact]
104 public static void AllVariantsSearchTest()
105 {
106     const long sequenceLength = 8;
107
108     using (var scope = new TempLinksTestScope(useSequences: true))
109     {
110         var links = scope.Links;
111         var sequences = scope.Sequences;
112
113         var sequence = new ulong[sequenceLength];
114         for (var i = 0; i < sequenceLength; i++)
115         {
116             sequence[i] = links.Create();
117         }
118
119         var createResults = sequences.CreateAllVariants2(sequence).Distinct().ToArray();
120
121         //for (int i = 0; i < createResults.Length; i++)
122         //    sequences.Create(createResults[i]);
123
124         var sw0 = Stopwatch.StartNew();
125

```

```

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();
237
238         //var sw3 = Stopwatch.StartNew();
239         //var searchResults3 =
240         ↪ sequences.GetAllPartiallyMatchingSequences2(partialSequence); sw3.Stop();
241
242         var sw4 = Stopwatch.StartNew();
243         var searchResults4 =
244         ↪ sequences.GetAllPartiallyMatchingSequences3(partialSequence); sw4.Stop();
245
246         //Global.Trash = searchResults3;
247
248         //var searchResults1Strings = searchResults1.Select(x => x + ": " +
249         ↪ sequences.FormatSequence(x)).ToList();
250         //Global.Trash = searchResults1Strings;
251
252         var intersection1 = createResults.Intersect(searchResults1).ToList();
253         Assert.True(intersection1.Count == createResults.Length);
254
255         var intersection2 = createResults.Intersect(searchResults2).ToList();
256         Assert.True(intersection2.Count == createResults.Length);
257
258         var intersection4 = createResults.Intersect(searchResults4).ToList();
259         Assert.True(intersection4.Count == createResults.Length);
260
261         for (var i = 0; i < sequenceLength; i++)
262         {
263             links.Delete(sequence[i]);
264         }
265     }
266 }
267
268 [Fact]
269 public static void BalancedPartialVariantsSearchTest()
270 {
271     const long sequenceLength = 200;
272
273     using (var scope = new TempLinksTestScope(useSequences: true))
274     {
275         var links = scope.Links;
276         var sequences = scope.Sequences;
277
278         var sequence = new ulong[sequenceLength];
279         for (var i = 0; i < sequenceLength; i++)
280         {
281             sequence[i] = links.Create();
282         }
283
284         var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);

```

```

279
280     var balancedVariant = balancedVariantConverter.Convert(sequence);
281
282     var partialSequence = new ulong[sequenceLength - 2];
283
284     Array.Copy(sequence, 1, partialSequence, 0, (int)sequenceLength - 2);
285
286     var sw1 = Stopwatch.StartNew();
287     var searchResults1 =
288         ↪ sequences.GetAllPartiallyMatchingSequences0(partialSequence); sw1.Stop();
289
290     var sw2 = Stopwatch.StartNew();
291     var searchResults2 =
292         ↪ sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();
293
294     Assert.True(searchResults1.Count == 1 && balancedVariant == searchResults1[0]);
295
296     Assert.True(searchResults2.Count == 1 && balancedVariant ==
297         ↪ searchResults2.First());
298
299     for (var i = 0; i < sequenceLength; i++)
300     {
301         links.Delete(sequence[i]);
302     }
303
304     [Fact(Skip = "Correct implementation is pending")]
305     public static void PatternMatchTest()
306     {
307         var zeroOrMany = Sequences.Sequences.ZeroOrMany;
308
309         using (var scope = new TempLinksTestScope(useSequences: true))
310         {
311             var links = scope.Links;
312             var sequences = scope.Sequences;
313
314             var e1 = links.Create();
315             var e2 = links.Create();
316
317             var sequence = new[]
318             {
319                 e1, e2, e1, e2 // mama / papa
320             };
321
322             var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
323
324             var balancedVariant = balancedVariantConverter.Convert(sequence);
325
326             // 1: [1]
327             // 2: [2]
328             // 3: [1,2]
329             // 4: [1,2,1,2]
330
331             var doublet = links.GetSource(balancedVariant);
332
333             var matchedSequences1 = sequences.MatchPattern(e2, e1, zeroOrMany);
334
335             Assert.True(matchedSequences1.Count == 0);
336
337             var matchedSequences2 = sequences.MatchPattern(zeroOrMany, e2, e1);
338
339             Assert.True(matchedSequences2.Count == 0);
340
341             var matchedSequences3 = sequences.MatchPattern(e1, zeroOrMany, e1);
342
343             Assert.True(matchedSequences3.Count == 0);
344
345             var matchedSequences4 = sequences.MatchPattern(e1, zeroOrMany, e2);
346
347             Assert.Contains(doublet, matchedSequences4);
348             Assert.Contains(balancedVariant, matchedSequences4);
349
350             for (var i = 0; i < sequence.Length; i++)
351             {
352                 links.Delete(sequence[i]);
353             }
354         }
355     }

```

```

356 [Fact]
357 public static void IndexTest()
358 {
359     using (var scope = new TempLinksTestScope(new SequencesOptions<ulong> { UseIndex =
        ↳ true }, useSequences: true))
360     {
361         var links = scope.Links;
362         var sequences = scope.Sequences;
363         var index = sequences.Options.Index;
364
365         var e1 = links.Create();
366         var e2 = links.Create();
367
368         var sequence = new[]
369         {
370             e1, e2, e1, e2 // mama / papa
371         };
372
373         Assert.False(index.MightContain(sequence));
374
375         index.Add(sequence);
376
377         Assert.True(index.MightContain(sequence));
378     }
379 }
380
381 /// <summary>Imported from https://raw.githubusercontent.com/Konard/LinksPlatform/%
    ↳ 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
    ↳ %D0%B8%D0%BD%D0%B0%D0%BB%D0%BE%D1%81%D1%8C.md</summary>
382 private static readonly string _exampleText =
383     @"([english
    ↳ 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/Links
    ↳ Platform/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-animation-500.gif
↳ "анимация")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro-animation-500.gif);
436
```

```

437     private static readonly string _exampleLoremIpsumText =
438         @"Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
439             ↪ incididunt ut labore et dolore magna aliqua.
440 Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo
441             ↪ consequat.";
442
443 [Fact]
444 public static void CompressionTest()
445 {
446     using (var scope = new TempLinksTestScope(useSequences: true))
447     {
448         var links = scope.Links;
449         var sequences = scope.Sequences;
450
451         var e1 = links.Create();
452         var e2 = links.Create();
453
454         var sequence = new[]
455         {
456             e1, e2, e1, e2 // mama / papa / template [(m/p), a] { [1] [2] [1] [2] }
457         };
458
459         var balancedVariantConverter = new BalancedVariantConverter<ulong>(links.Unsync);
460         var totalSequenceSymbolFrequencyCounter = new
461             ↪ TotalSequenceSymbolFrequencyCounter<ulong>(links.Unsync);
462         var doubletFrequenciesCache = new LinkFrequenciesCache<ulong>(links.Unsync,
463             ↪ totalSequenceSymbolFrequencyCounter);
464         var compressingConverter = new CompressingConverter<ulong>(links.Unsync,
465             ↪ balancedVariantConverter, doubletFrequenciesCache);
466
467         var compressedVariant = compressingConverter.Convert(sequence);
468
469         // 1: [1]          (1->1) point
470         // 2: [2]          (2->2) point
471         // 3: [1,2]        (1->2) doublet
472         // 4: [1,2,1,2]    (3->3) doublet
473
474         Assert.True(links.GetSource(links.GetSource(compressedVariant)) == sequence[0]);
475         Assert.True(links.GetTarget(links.GetSource(compressedVariant)) == sequence[1]);
476         Assert.True(links.GetSource(links.GetTarget(compressedVariant)) == sequence[2]);
477         Assert.True(links.GetTarget(links.GetTarget(compressedVariant)) == sequence[3]);
478
479         var source = _constants.SourcePart;
480         var target = _constants.TargetPart;
481
482         Assert.True(links.GetByKeys(compressedVariant, source, source) == sequence[0]);
483         Assert.True(links.GetByKeys(compressedVariant, source, target) == sequence[1]);
484         Assert.True(links.GetByKeys(compressedVariant, target, source) == sequence[2]);
485         Assert.True(links.GetByKeys(compressedVariant, target, target) == sequence[3]);
486
487         // 4 - length of sequence
488         Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 0)
489             ↪ == sequence[0]);
490         Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 1)
491             ↪ == sequence[1]);
492         Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 2)
493             ↪ == sequence[2]);
494         Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 3)
495             ↪ == sequence[3]);
496     }
497 }
498
499 [Fact]
500 public static void CompressionEfficiencyTest()
501 {
502     var strings = _exampleLoremIpsumText.Split(new[] { '\n', '\r' },
503         ↪ StringSplitOptions.RemoveEmptyEntries);
504     var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
505     var totalCharacters = arrays.Select(x => x.Length).Sum();
506
507     using (var scope1 = new TempLinksTestScope(useSequences: true))
508     using (var scope2 = new TempLinksTestScope(useSequences: true))
509     using (var scope3 = new TempLinksTestScope(useSequences: true))
510     {
511         scope1.Links.Unsync.UseUnicode();
512         scope2.Links.Unsync.UseUnicode();
513         scope3.Links.Unsync.UseUnicode();
514     }
515 }

```

```

506 var balancedVariantConverter1 = new
    ↳ BalancedVariantConverter<ulong>(scope1.Links.Unsync);
507 var totalSequenceSymbolFrequencyCounter = new
    ↳ TotalSequenceSymbolFrequencyCounter<ulong>(scope1.Links.Unsync);
508 var linkFrequenciesCache1 = new LinkFrequenciesCache<ulong>(scope1.Links.Unsync,
    ↳ totalSequenceSymbolFrequencyCounter);
509 var compressor1 = new CompressingConverter<ulong>(scope1.Links.Unsync,
    ↳ balancedVariantConverter1, linkFrequenciesCache1,
    ↳ doInitialFrequenciesIncrement: false);
510
511 var compressor2 = scope2.Sequences;
512 var compressor3 = scope3.Sequences;
513
514 var constants = Default<LinksCombinedConstants<bool, ulong, int>>.Instance;
515
516 var sequences = compressor3;
517 //var meaningRoot = links.CreatePoint();
518 //var unaryOne = links.CreateAndUpdate(meaningRoot, constants.Itself);
519 //var frequencyMarker = links.CreateAndUpdate(meaningRoot, constants.Itself);
520 //var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot,
    ↳ constants.Itself);
521
522 //var unaryNumberToAddressConverter = new
    ↳ UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);
523 //var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links,
    ↳ unaryOne);
524 //var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
    ↳ 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
    ↳ LinkToItsFrequencyNumberConverter<ulong>(links, frequencyPropertyOperator,
    ↳ unaryNumberToAddressConverter);
528
529 var linkFrequenciesCache3 = new LinkFrequenciesCache<ulong>(scope3.Links.Unsync,
    ↳ totalSequenceSymbolFrequencyCounter);
530
531 var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<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($"Compressor: {elapsed1}, Balanced variant: {elapsed2},
589         ↳ Optimal variant: {elapsed3}");
590
591     // Assert.True(elapsed1 > elapsed2);
592
593     // Checks
594     for (int i = START; i < END; i++)
595     {
596         var sequence1 = compressed1[i];
597         var sequence2 = compressed2[i];
598         var sequence3 = compressed3[i];
599
600         var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
601             ↳ scope1.Links.Unsync);
602
603         var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
604             ↳ scope2.Links.Unsync);
605
606         var decompress3 = UnicodeMap.FromSequenceLinkToString(sequence3,
607             ↳ scope3.Links.Unsync);
608
609         var structure1 = scope1.Links.Unsync.FormatStructure(sequence1, link =>
610             ↳ link.IsPartialPoint());
611         var structure2 = scope2.Links.Unsync.FormatStructure(sequence2, link =>
612             ↳ link.IsPartialPoint());
613         var structure3 = scope3.Links.Unsync.FormatStructure(sequence3, link =>
614             ↳ link.IsPartialPoint());
615
616         //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
617             ↳ arrays[i].Length > 3)
618         //    Assert.False(structure1 == structure2);
619         //if (sequence3 != Constants.Null && sequence2 != Constants.Null &&
620             ↳ arrays[i].Length > 3)
621         //    Assert.False(structure3 == structure2);
622
623         Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
624         Assert.True(strings[i] == decompress3 && decompress3 == decompress2);
625     }
626
627     Assert.True((int)(scope1.Links.Unsync.Count() - initialCount1) <
628         ↳ totalCharacters);
629     Assert.True((int)(scope2.Links.Unsync.Count() - initialCount2) <
630         ↳ totalCharacters);
631     Assert.True((int)(scope3.Links.Unsync.Count() - initialCount3) <
632         ↳ totalCharacters);
633
634     Console.WriteLine($"{{(double)(scope1.Links.Unsync.Count() - initialCount1) /
635         ↳ totalCharacters}} | {{(double)(scope2.Links.Unsync.Count() - initialCount2) /
636         ↳ totalCharacters}} | {{(double)(scope3.Links.Unsync.Count() - initialCount3) /
637         ↳ totalCharacters}}");
638
639     Assert.True(scope1.Links.Unsync.Count() - initialCount1 <
640         ↳ scope2.Links.Unsync.Count() - initialCount2);
641     Assert.True(scope3.Links.Unsync.Count() - initialCount3 <
642         ↳ scope2.Links.Unsync.Count() - initialCount2);

```

```

626     var duplicateProvider1 = new
627         ↳ DuplicateSegmentsProvider<ulong>(scope1.Links.Unsync, scope1.Sequences);
628     var duplicateProvider2 = new
629         ↳ DuplicateSegmentsProvider<ulong>(scope2.Links.Unsync, scope2.Sequences);
630     var duplicateProvider3 = new
631         ↳ DuplicateSegmentsProvider<ulong>(scope3.Links.Unsync, scope3.Sequences);
632
633     var duplicateCounter1 = new DuplicateSegmentsCounter<ulong>(duplicateProvider1);
634     var duplicateCounter2 = new DuplicateSegmentsCounter<ulong>(duplicateProvider2);
635     var duplicateCounter3 = new DuplicateSegmentsCounter<ulong>(duplicateProvider3);
636
637     var duplicates1 = duplicateCounter1.Count();
638
639     ConsoleHelpers.Debug("-----");
640
641     var duplicates2 = duplicateCounter2.Count();
642
643     ConsoleHelpers.Debug("-----");
644
645     var duplicates3 = duplicateCounter3.Count();
646
647     Console.WriteLine($"{duplicates1} | {duplicates2} | {duplicates3}");
648
649     linkFrequenciesCache1.ValidateFrequencies();
650     linkFrequenciesCache3.ValidateFrequencies();
651 }
652
653 [Fact]
654 public static void CompressionStabilityTest()
655 {
656     // TODO: Fix bug (do a separate test)
657     //const ulong minNumbers = 0;
658     //const ulong maxNumbers = 1000;
659
660     const ulong minNumbers = 10000;
661     const ulong maxNumbers = 12500;
662
663     var strings = new List<string>();
664
665     for (ulong i = minNumbers; i < maxNumbers; i++)
666     {
667         strings.Add(i.ToString());
668     }
669
670     var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
671     var totalCharacters = arrays.Select(x => x.Length).Sum();
672
673     using (var scope1 = new TempLinksTestScope(useSequences: true, sequencesOptions: new
674         ↳ SequencesOptions<ulong> { UseCompression = true,
675         ↳ EnforceSingleSequenceVersionOnWriteBasedOnExisting = true }))
676     using (var scope2 = new TempLinksTestScope(useSequences: true))
677     {
678         scope1.Links.UseUnicode();
679         scope2.Links.UseUnicode();
680
681         //var compressor1 = new Compressor(scope1.Links.Unsync, scope1.Sequences);
682         var compressor1 = scope1.Sequences;
683         var compressor2 = scope2.Sequences;
684
685         var compressed1 = new ulong[arrays.Length];
686         var compressed2 = new ulong[arrays.Length];
687
688         var sw1 = Stopwatch.StartNew();
689
690         var START = 0;
691         var END = arrays.Length;
692
693         // Collisions proved (cannot be solved by max doublet comparison, no stable rule)
694         // Stability issue starts at 10001 or 11000
695         //for (int i = START; i < END; i++)
696         //{
697             var first = compressor1.Compress(arrays[i]);
698             var second = compressor1.Compress(arrays[i]);
699
700             if (first == second)
701                 compressed1[i] = first;
702             else
703                 {

```

```

701 //      // TODO: Find a solution for this case
702 //  }
703 //}
704
705 for (int i = START; i < END; i++)
706 {
707     var first = compressor1.Create(arrays[i]);
708     var second = compressor1.Create(arrays[i]);
709
710     if (first == second)
711     {
712         compressed1[i] = first;
713     }
714     else
715     {
716         // TODO: Find a solution for this case
717     }
718 }
719
720 var elapsed1 = sw1.Elapsed;
721
722 var balancedVariantConverter = new BalancedVariantConverter<ulong>(scope2.Links);
723
724 var sw2 = Stopwatch.StartNew();
725
726 for (int i = START; i < END; i++)
727 {
728     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());

```

```

772         //compressor1.ValidateFrequencies();
773     }
774 }
775
776 [Fact]
777 public static void RandomNumbersCompressionQualityTest()
778 {
779     const ulong N = 500;
780
781     //const ulong minNumbers = 10000;
782     //const ulong maxNumbers = 20000;
783
784     //var strings = new List<string>();
785
786     //for (ulong i = 0; i < N; i++)
787     //    strings.Add(RandomHelpers.DefaultFactory.NextUInt64(minNumbers,
788         ↪ maxNumbers).ToString());
789
789     var strings = new List<string>();
790
791     for (ulong i = 0; i < N; i++)
792     {
793         strings.Add(RandomHelpers.Default.NextUInt64().ToString());
794     }
795
796     strings = strings.Distinct().ToList();
797
798     var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
799     var totalCharacters = arrays.Select(x => x.Length).Sum();
800
801     using (var scope1 = new TempLinksTestScope(useSequences: true, sequencesOptions: new
802         ↪ SequencesOptions<ulong> { UseCompression = true,
803         ↪ EnforceSingleSequenceVersionOnWriteBasedOnExisting = true }))
804     using (var scope2 = new TempLinksTestScope(useSequences: true))
805     {
806         scope1.Links.UseUnicode();
807         scope2.Links.UseUnicode();
808
809         var compressor1 = scope1.Sequences;
810         var compressor2 = scope2.Sequences;
811
812         var compressed1 = new ulong[arrays.Length];
813         var compressed2 = new ulong[arrays.Length];
814
815         var sw1 = Stopwatch.StartNew();
816
817         var START = 0;
818         var END = arrays.Length;
819
820         for (int i = START; i < END; i++)
821         {
822             compressed1[i] = compressor1.Create(arrays[i]);
823         }
824
825         var elapsed1 = sw1.Elapsed;
826
827         var balancedVariantConverter = new BalancedVariantConverter<ulong>(scope2.Links);
828         var sw2 = Stopwatch.StartNew();
829
830         for (int i = START; i < END; i++)
831         {
832             compressed2[i] = balancedVariantConverter.Convert(arrays[i]);
833         }
834
835         var elapsed2 = sw2.Elapsed;
836
837         Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator:
838             ↪ {elapsed2}");
839
840         Assert.True(elapsed1 > elapsed2);
841
842         // Checks
843         for (int i = START; i < END; i++)
844         {
845             var sequence1 = compressed1[i];
846             var sequence2 = compressed2[i];
847
848             if (sequence1 != _constants.Null && sequence2 != _constants.Null)
849             {

```

```

848         var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
849             ↪ scope1.Links);
850         var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
851             ↪ scope2.Links);
852         Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
853     }
854 }
855
856 Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
857 Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
858
859 Debug.WriteLine($"{{(double)(scope1.Links.Count() - UnicodeMap.MapSize) /
    ↪ totalCharacters}} | {{(double)(scope2.Links.Count() - UnicodeMap.MapSize) /
    ↪ totalCharacters}}");
860
861 // Can be worse than balanced variant
862 //Assert.True(scope1.Links.Count() <= scope2.Links.Count());
863
864 //compressor1.ValidateFrequencies();
865 }
866 }
867
868 [Fact]
869 public static void AllTreeBreakDownAtSequencesCreationBugTest()
870 {
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         Global.Trash = createResults;
890
891         for (var i = 0; i < sequenceLength; i++)
892         {
893             links.Delete(sequence[i]);
894         }
895     }
896 }
897
898 [Fact]
899 public static void AllPossibleConnectionsTest()
900 {
901     const long sequenceLength = 5;
902
903     using (var scope = new TempLinksTestScope(useSequences: true))
904     {
905         var links = scope.Links;
906         var sequences = scope.Sequences;
907
908         var sequence = new ulong[sequenceLength];
909         for (var i = 0; i < sequenceLength; i++)
910         {
911             sequence[i] = links.Create();
912         }
913
914         var createResults = sequences.CreateAllVariants2(sequence);
915         var reverseResults = sequences.CreateAllVariants2(sequence.Reverse().ToArray());
916
917         for (var i = 0; i < 1; i++)
918         {
919             var sw1 = Stopwatch.StartNew();
920             var searchResults1 = sequences.GetAllConnections(sequence); sw1.Stop();
921
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();
29            TempTransactionLogFilename = Path.GetTempFileName();
30
31            var coreMemoryAdapter = new UInt64ResizableDirectMemoryLinks(TempFilename);
32
33            MemoryAdapter = useLog ? (ILinks<ulong>)new
34                ↪ UInt64LinksTransactionsLayer(coreMemoryAdapter, TempTransactionLogFilename) :
35                ↪ coreMemoryAdapter;
36
37            Links = new SynchronizedLinks<ulong>(new UInt64Links(MemoryAdapter));
38            if (useSequences)
39            {
40                Sequences = new Sequences.Sequences(Links, sequencesOptions);
41            }
42        }
43
44        protected override void Dispose(bool manual, bool wasDisposed)
45        {
46            if (!wasDisposed)
47            {
48                Links.Unsync.DisposeIfPossible();
49                if (_deleteFiles)
50                {
51                    DeleteFiles();
52                }
53            }
54        }
55
56        public void DeleteFiles()
57        {
58            File.Delete(TempFilename);
59            File.Delete(TempTransactionLogFilename);
60        }
61    }
62 }
```

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

```
1 using Xunit;
2 using Platform.Random;
3 using Platform.Data.Doublets.Numbers.Unary;
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;
```





```

39     var resultingCharacter = unicodeSymbolToCharConverter.Convert(characterLink);
40
41     Assert.Equal(originalCharacter, resultingCharacter);
42 }
43 }
44
45 [Fact]
46 public static void StringAndUnicodeSequenceConvertersTest()
47 {
48     using (var scope = new TempLinksTestScope())
49     {
50         var links = scope.Links;
51
52         var itself = links.Constants.Itself;
53
54         var meaningRoot = links.CreatePoint();
55         var unaryOne = links.CreateAndUpdate(meaningRoot, itself);
56         var unicodeSymbolMarker = links.CreateAndUpdate(meaningRoot, itself);
57         var unicodeSequenceMarker = links.CreateAndUpdate(meaningRoot, itself);
58         var frequencyMarker = links.CreateAndUpdate(meaningRoot, itself);
59         var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot, itself);
60
61         var powerOf2ToUnaryNumberConverter = new
62             ↪ PowerOf2ToUnaryNumberConverter<ulong>(links, unaryOne);
63         var addressToUnaryNumberConverter = new
64             ↪ AddressToUnaryNumberConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
65         var charToUnicodeSymbolConverter = new
66             ↪ CharToUnicodeSymbolConverter<ulong>(links, addressToUnaryNumberConverter,
67             ↪ unicodeSymbolMarker);
68
69         var unaryNumberToAddressConverter = new
70             ↪ UnaryNumberToAddressOrOperationConverter<ulong>(links,
71             ↪ powerOf2ToUnaryNumberConverter);
72         var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links, unaryOne);
73         var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
74             ↪ frequencyMarker, unaryOne, unaryNumberIncrementer);
75         var frequencyPropertyOperator = new PropertyOperator<ulong>(links,
76             ↪ frequencyPropertyMarker, frequencyMarker);
77         var index = new FrequencyIncrementingSequenceIndex<ulong>(links,
78             ↪ frequencyPropertyOperator, frequencyIncrementer);
79         var linkToItsFrequencyNumberConverter = new
80             ↪ LinkToItsFrequencyNumberConverter<ulong>(links, frequencyPropertyOperator,
81             ↪ unaryNumberToAddressConverter);
82         var sequenceToItsLocalElementLevelsConverter = new
83             ↪ SequenceToItsLocalElementLevelsConverter<ulong>(links,
84             ↪ linkToItsFrequencyNumberConverter);
85         var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
86             ↪ sequenceToItsLocalElementLevelsConverter);
87
88         var stringToUnicodeSymbolConverter = new
89             ↪ StringToUnicodeSequenceConverter<ulong>(links, charToUnicodeSymbolConverter,
90             ↪ index, optimalVariantConverter, unicodeSequenceMarker);
91
92         var originalString = "Hello";
93
94         var unicodeSequenceLink = stringToUnicodeSymbolConverter.Convert(originalString);
95
96         var unicodeSymbolCriterionMatcher = new
97             ↪ UnicodeSymbolCriterionMatcher<ulong>(links, unicodeSymbolMarker);
98         var unicodeSymbolToCharConverter = new
99             ↪ UnicodeSymbolToCharConverter<ulong>(links, unaryNumberToAddressConverter,
100             ↪ unicodeSymbolCriterionMatcher);
101
102         var unicodeSequenceCriterionMatcher = new
103             ↪ UnicodeSequenceCriterionMatcher<ulong>(links, unicodeSequenceMarker);
104
105         var sequenceWalker = new LeveledSequenceWalker<ulong>(links,
106             ↪ unicodeSymbolCriterionMatcher.IsMatched);
107
108         var unicodeSequenceToStringConverter = new
109             ↪ UnicodeSequenceToStringConverter<ulong>(links,
110             ↪ unicodeSequenceCriterionMatcher, sequenceWalker,
111             ↪ unicodeSymbolToCharConverter);
112
113         var resultingString =
114             ↪ unicodeSequenceToStringConverter.Convert(unicodeSequenceLink);
115
116         Assert.Equal(originalString, resultingString);

```

92

93

94

95

}

}

}

}

## Index

./Platform.Data.Doublets.Tests/ComparisonTests.cs, 138  
./Platform.Data.Doublets.Tests/DoubletLinksTests.cs, 139  
./Platform.Data.Doublets.Tests/EqualityTests.cs, 142  
./Platform.Data.Doublets.Tests/LinksTests.cs, 143  
./Platform.Data.Doublets.Tests/OptimalVariantSequenceTests.cs, 156  
./Platform.Data.Doublets.Tests/ReadSequenceTests.cs, 158  
./Platform.Data.Doublets.Tests/ResizableDirectMemoryLinksTests.cs, 158  
./Platform.Data.Doublets.Tests/ScopeTests.cs, 159  
./Platform.Data.Doublets.Tests/SequencesTests.cs, 160  
./Platform.Data.Doublets.Tests/TempLinksTestScope.cs, 174  
./Platform.Data.Doublets.Tests/UnaryNumberConvertersTests.cs, 175  
./Platform.Data.Doublets.Tests/UnicodeConvertersTests.cs, 176  
./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, 5  
./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, 11  
./Platform.Data.Doublets/Hybrid.cs, 11  
./Platform.Data.Doublets/ILinks.cs, 13  
./Platform.Data.Doublets/ILinksExtensions.cs, 13  
./Platform.Data.Doublets/ISynchronizedLinks.cs, 24  
./Platform.Data.Doublets/Incrementers/FrequencyIncrementer.cs, 23  
./Platform.Data.Doublets/Incrementers/UnaryNumberIncrementer.cs, 24  
./Platform.Data.Doublets/Link.cs, 24  
./Platform.Data.Doublets/LinkExtensions.cs, 27  
./Platform.Data.Doublets/LinksOperatorBase.cs, 27  
./Platform.Data.Doublets/Numbers/Unary/AddressToUnaryNumberConverter.cs, 27  
./Platform.Data.Doublets/Numbers/Unary/LinkToltsFrequencyNumberConveter.cs, 28  
./Platform.Data.Doublets/Numbers/Unary/PowerOf2ToUnaryNumberConverter.cs, 28  
./Platform.Data.Doublets/Numbers/Unary/UnaryNumberToAddressAddOperationConverter.cs, 29  
./Platform.Data.Doublets/Numbers/Unary/UnaryNumberToAddressOrOperationConverter.cs, 30  
./Platform.Data.Doublets/PropertyOperators/PropertiesOperator.cs, 30  
./Platform.Data.Doublets/PropertyOperators/PropertyOperator.cs, 31  
./Platform.Data.Doublets/ResizableDirectMemory/ResizableDirectMemoryLinks.ListMethods.cs, 41  
./Platform.Data.Doublets/ResizableDirectMemory/ResizableDirectMemoryLinks.TreeMethods.cs, 41  
./Platform.Data.Doublets/ResizableDirectMemory/ResizableDirectMemoryLinks.cs, 32  
./Platform.Data.Doublets/ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.ListMethods.cs, 54  
./Platform.Data.Doublets/ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.TreeMethods.cs, 55  
./Platform.Data.Doublets/ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.cs, 48  
./Platform.Data.Doublets/Sequences/Converters/BalancedVariantConverter.cs, 61  
./Platform.Data.Doublets/Sequences/Converters/CompressingConverter.cs, 62  
./Platform.Data.Doublets/Sequences/Converters/LinksListToSequenceConverterBase.cs, 65  
./Platform.Data.Doublets/Sequences/Converters/OptimalVariantConverter.cs, 65  
./Platform.Data.Doublets/Sequences/Converters/SequenceToltsLocalElementLevelsConverter.cs, 67  
./Platform.Data.Doublets/Sequences/CreteriaMatchers/DefaultSequenceElementCriterionMatcher.cs, 67  
./Platform.Data.Doublets/Sequences/CreteriaMatchers/MarkedSequenceCriterionMatcher.cs, 68  
./Platform.Data.Doublets/Sequences/DefaultSequenceAppender.cs, 68  
./Platform.Data.Doublets/Sequences/DuplicateSegmentsCounter.cs, 69  
./Platform.Data.Doublets/Sequences/DuplicateSegmentsProvider.cs, 69  
./Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkFrequenciesCache.cs, 71  
./Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkFrequency.cs, 73  
./Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkToltsFrequencyValueConverter.cs, 73  
./Platform.Data.Doublets/Sequences/Frequencies/Counters/MarkedSequenceSymbolFrequencyOneOffCounter.cs, 73  
./Platform.Data.Doublets/Sequences/Frequencies/Counters/SequenceSymbolFrequencyOneOffCounter.cs, 74  
./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyCounter.cs, 74

./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyOneOffCounter.cs, 75  
./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyCounter.cs, 75  
./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyOneOffCounter.cs, 75  
./Platform.Data.Doublets/Sequences/HeightProviders/CachedSequenceHeightProvider.cs, 76  
./Platform.Data.Doublets/Sequences/HeightProviders/DefaultSequenceRightHeightProvider.cs, 77  
./Platform.Data.Doublets/Sequences/HeightProviders/ISequenceHeightProvider.cs, 77  
./Platform.Data.Doublets/Sequences/Indexes/CachedFrequencyIncrementingSequenceIndex.cs, 77  
./Platform.Data.Doublets/Sequences/Indexes/FrequencyIncrementingSequenceIndex.cs, 78  
./Platform.Data.Doublets/Sequences/Indexes/ISequenceIndex.cs, 79  
./Platform.Data.Doublets/Sequences/Indexes/SequenceIndex.cs, 79  
./Platform.Data.Doublets/Sequences/Indexes/SynchronizedSequenceIndex.cs, 80  
./Platform.Data.Doublets/Sequences/Sequences.Experiments.cs, 90  
./Platform.Data.Doublets/Sequences/Sequences.cs, 80  
./Platform.Data.Doublets/Sequences/SequencesExtensions.cs, 116  
./Platform.Data.Doublets/Sequences/SequencesOptions.cs, 116  
./Platform.Data.Doublets/Sequences/Walkers/ISequenceWalker.cs, 118  
./Platform.Data.Doublets/Sequences/Walkers/LeftSequenceWalker.cs, 118  
./Platform.Data.Doublets/Sequences/Walkers/LeveledSequenceWalker.cs, 119  
./Platform.Data.Doublets/Sequences/Walkers/RightSequenceWalker.cs, 120  
./Platform.Data.Doublets/Sequences/Walkers/SequenceWalkerBase.cs, 121  
./Platform.Data.Doublets/Stacks/Stack.cs, 122  
./Platform.Data.Doublets/Stacks/StackExtensions.cs, 122  
./Platform.Data.Doublets/SynchronizedLinks.cs, 122  
./Platform.Data.Doublets/UInt64Link.cs, 123  
./Platform.Data.Doublets/UInt64LinkExtensions.cs, 125  
./Platform.Data.Doublets/UInt64LinksExtensions.cs, 125  
./Platform.Data.Doublets/UInt64LinksTransactionsLayer.cs, 128  
./Platform.Data.Doublets/Unicode/CharToUnicodeSymbolConverter.cs, 133  
./Platform.Data.Doublets/Unicode/StringToUnicodeSequenceConverter.cs, 133  
./Platform.Data.Doublets/Unicode/UnicodeMap.cs, 134  
./Platform.Data.Doublets/Unicode/UnicodeSequenceCriterionMatcher.cs, 136  
./Platform.Data.Doublets/Unicode/UnicodeSequenceToStringConverter.cs, 137  
./Platform.Data.Doublets/Unicode/UnicodeSymbolCriterionMatcher.cs, 137  
./Platform.Data.Doublets/Unicode/UnicodeSymbolToCharConverter.cs, 137