

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              // Use Facade (the last decorator) to ensure recursion working correctly
13              Facade.MergeUsages(oldLinkAddress, newLinkAddress);
14              return base.ResolveAddressChangeConflict(oldLinkAddress, newLinkAddress);
15          }
16      }

```

./Platform.Data.Doublets/Decorators/LinksCascadeUsagesResolver.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      /// <para>Must be used in conjunction with NonNullContentsLinkDeletionResolver.</para>
9      /// <para>Должен использоваться вместе с NonNullContentsLinkDeletionResolver.</para>
10     /// </remarks>
11     public class LinksCascadeUsagesResolver<TLink> : LinksDecoratorBase<TLink>
12     {
13         public LinksCascadeUsagesResolver(ILinks<TLink> links) : base(links) { }
14
15         public override void Delete(IList<TLink> restrictions)
16         {
17             var linkIndex = restrictions[Constants.IndexPart];
18             // Use Facade (the last decorator) to ensure recursion working correctly
19             Facade.DeleteAllUsages(linkIndex);
20             Links.Delete(linkIndex);
21         }
22     }
23 }

```

./Platform.Data.Doublets/Decorators/LinksDecoratorBase.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 abstract class LinksDecoratorBase<TLink> : LinksOperatorBase<TLink>, ILinks<TLink>
9      {
10         public LinksConstants<TLink> Constants { get; }
11
12         private ILinks<TLink> _facade;
13
14         public ILinks<TLink> Facade
15         {
16             get => _facade;
17             private set
18             {
19                 _facade = value;
20                 if (Links is LinksDecoratorBase<TLink> decorator)
21                 {
22                     decorator.Facade = value;
23                 }
24             }
25         }
26
27         protected LinksDecoratorBase(ILinks<TLink> links) : base(links)
28         {
29             Constants = links.Constants;
30             Facade = this;
31         }
32
33         public virtual TLink Count(IList<TLink> restrictions) => Links.Count(restrictions);

```

```

34
35     public virtual TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
        ↪ => Links.Each(handler, restrictions);
36
37     public virtual TLink Create(IList<TLink> restrictions) => Links.Create(restrictions);
38
39     public virtual TLink Update(IList<TLink> restrictions, IList<TLink> substitution) =>
        ↪ Links.Update(restrictions, substitution);
40
41     public virtual void Delete(IList<TLink> restrictions) => Links.Delete(restrictions);
42 }
43 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using Platform.Disposables;
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 LinksDisposableDecoratorBase<TLink> : DisposableBase, ILinks<TLink>
10     {
11         public LinksConstants<TLink> Constants { get; }
12
13         public ILinks<TLink> Links { get; }
14
15         protected LinksDisposableDecoratorBase(ILinks<TLink> links)
16         {
17             Links = links;
18             Constants = links.Constants;
19         }
20
21         public virtual TLink Count(IList<TLink> restrictions) => Links.Count(restrictions);
22
23         public virtual TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
            ↪ => Links.Each(handler, restrictions);
24
25         public virtual TLink Create(IList<TLink> restrictions) => Links.Create(restrictions);
26
27         public virtual TLink Update(IList<TLink> restrictions, IList<TLink> substitution) =>
            ↪ Links.Update(restrictions, substitution);
28
29         public virtual void Delete(IList<TLink> restrictions) => Links.Delete(restrictions);
30
31         protected override bool AllowMultipleDisposeCalls => true;
32
33         protected override void Dispose(bool manual, bool wasDisposed)
34         {
35             if (!wasDisposed)
36             {
37                 Links.DisposeIfPossible();
38             }
39         }
40     }
41 }

```

./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
8      ↪ be external (hybrid link's raw number).
9      public class LinksInnerReferenceExistenceValidator<TLink> : LinksDecoratorBase<TLink>
10     {
11         public LinksInnerReferenceExistenceValidator(ILinks<TLink> links) : base(links) { }
12
13         public override TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
14         {
15             Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
16             return Links.Each(handler, restrictions);
17         }
18
19         public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution)

```

```

20     {
21         // TODO: Possible values: null, ExistentLink or NonExistentHybrid(ExternalReference)
22         Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
23         Links.EnsureInnerReferenceExists(substitution, nameof(substitution));
24         return Links.Update(restrictions, substitution);
25     }
26
27     public override void Delete(IList<TLink> restrictions)
28     {
29         var link = restrictions[Constants.IndexPart];
30         Links.EnsureLinkExists(link, nameof(link));
31         Links.Delete(link);
32     }
33 }
34 }

```

./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)
16         {
17             var constants = Constants;
18             var itselfConstant = constants.Itself;
19             var indexPartConstant = constants.IndexPart;
20             var sourcePartConstant = constants.SourcePart;
21             var targetPartConstant = constants.TargetPart;
22             var restrictionsCount = restrictions.Count;
23             if (!_equalityComparer.Equals(constants.Any, itselfConstant)
24                 && ((restrictionsCount > indexPartConstant) &&
25                     ↳ _equalityComparer.Equals(restrictions[indexPartConstant], itselfConstant))
26                 || ((restrictionsCount > sourcePartConstant) &&
27                     ↳ _equalityComparer.Equals(restrictions[sourcePartConstant], itselfConstant))
28                 || ((restrictionsCount > targetPartConstant) &&
29                     ↳ _equalityComparer.Equals(restrictions[targetPartConstant], itselfConstant))))
30             {
31                 // Itself constant is not supported for Each method right now, skipping execution
32                 return constants.Continue;
33             }
34             return Links.Each(handler, restrictions);
35         }
36
37         public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution) =>
38             ↳ Links.Update(restrictions, Links.ResolveConstantAsSelfReference(Constants.Itself,
39                 ↳ restrictions, substitution));
40     }
41 }

```

./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, IList<TLink> substitution)
19         {

```

```

18         var constants = Constants;
19         Links.EnsureCreated(substitution[constants.SourcePart],
20             ↳ substitution[constants.TargetPart]);
21         return Links.Update(restrictions, substitution);
22     }
23 }

```

./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(IList<TLink> restrictions)
12         {
13             var link = Links.Create();
14             return Links.Update(link, link, link);
15         }
16
17         public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution) =>
18             ↳ Links.Update(restrictions, Links.ResolveConstantAsSelfReference(Constants.Null,
19             ↳ restrictions, substitution));
20     }
21 }

```

./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, IList<TLink> substitution)
15         {
16             var newLinkAddress = Links.SearchOrDefault(substitution[Constants.SourcePart],
17             ↳ substitution[Constants.TargetPart]);
18             if (_equalityComparer.Equals(newLinkAddress, default))
19             {
20                 return Links.Update(restrictions, substitution);
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                 Facade.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, IList<TLink> substitution)
12        {
13            Links.EnsureDoesNotExists(substitution[Constants.SourcePart],
14                ↳ substitution[Constants.TargetPart]);
15            return Links.Update(restrictions, substitution);
16        }
17    }

```

./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, IList<TLink> substitution)
12         {
13             Links.EnsureNoUsages(restrictions[Constants.IndexPart]);
14             return Links.Update(restrictions, substitution);
15         }
16
17         public override void Delete(IList<TLink> restrictions)
18         {
19             var link = restrictions[Constants.IndexPart];
20             Links.EnsureNoUsages(link);
21             Links.Delete(link);
22         }
23     }
24 }

```

./Platform.Data.Doublets/Decorators/NonNullContentsLinkDeletionResolver.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 NonNullContentsLinkDeletionResolver<TLink> : LinksDecoratorBase<TLink>
8      {
9          public NonNullContentsLinkDeletionResolver(ILinks<TLink> links) : base(links) { }
10
11         public override void Delete(IList<TLink> restrictions)
12         {
13             var linkIndex = restrictions[Constants.IndexPart];
14             Links.EnforceResetValues(linkIndex);
15             Links.Delete(linkIndex);
16         }
17     }
18 }

```

./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
    ↳ поддеревьев для AVL баланса и флагов нитей: 2 в степени(64 минус 5 равно 59 ) равно 576
    ↳ 460 752 303 423 488
23 /// Желательно реализовать поддержку переключения между деревьями и битовыми индексами
    ↳ (битовыми строками) - вариант матрицы (выстраиваемой лениво).
24 ///
25 /// Решить отключать ли проверки при компиляции под Release. Т.е. исключения будут
    ↳ выбрасываться только при #if DEBUG
26 /// </remarks>
27 public class UInt64Links : LinksDisposableDecoratorBase<ulong>
28 {
29     public UInt64Links(ILinks<ulong> links) : base(links) { }
30
31     public override ulong Each(Func<IList<ulong>, ulong> handler, IList<ulong> restrictions)
32     {
33         this.EnsureLinkIsAnyOrExists(restrictions);
34         return Links.Each(handler, restrictions);
35     }
36
37     public override ulong Create(IList<ulong> restrictions) => Links.CreatePoint();
38
39     public override ulong Update(IList<ulong> restrictions, IList<ulong> substitution)
40     {
41         var constants = Constants;
42         var nullConstant = constants.Null;
43         if (restrictions.IsNullOrEmpty())
44         {
45             return nullConstant;
46         }
47         // TODO: Looks like this is a common type of exceptions linked with restrictions
48         ↳ support
49         if (substitution.Count != 3)
50         {
51             throw new NotSupportedException();
52         }
53         var indexPartConstant = constants.IndexPart;
54         var updatedLink = restrictions[indexPartConstant];
55         this.EnsureLinkExists(updatedLink,
56             ↳ $"{nameof(restrictions)}[{nameof(indexPartConstant)}]");
57         var sourcePartConstant = constants.SourcePart;
58         var newSource = substitution[sourcePartConstant];
59         this.EnsureLinkIsItselfOrExists(newSource,
60             ↳ $"{nameof(substitution)}[{nameof(sourcePartConstant)}]");
61         var targetPartConstant = constants.TargetPart;
62         var newTarget = substitution[targetPartConstant];
63         this.EnsureLinkIsItselfOrExists(newTarget,
64             ↳ $"{nameof(substitution)}[{nameof(targetPartConstant)}]");
65         var existedLink = nullConstant;
66         var itselfConstant = constants.Itself;
67         if (newSource != itselfConstant && newTarget != itselfConstant)
68         {
69             existedLink = this.SearchOrDefault(newSource, newTarget);
70         }
71         if (existedLink == nullConstant)
72         {
73             var before = Links.GetLink(updatedLink);
74             if (before[sourcePartConstant] != newSource || before[targetPartConstant] !=
75                 ↳ newTarget)
76             {
77                 Links.Update(updatedLink, newSource == itselfConstant ? updatedLink :
78                     ↳ newSource,
79                             newTarget == itselfConstant ? updatedLink :
80                                 ↳ newTarget);
81             }
82             return updatedLink;
83         }
84         else
85         {
86             return this.MergeAndDelete(updatedLink, existedLink);
87         }
88     }
89
90     public override void Delete(IList<ulong> restrictions)
91     {
92         var linkIndex = restrictions[Constants.IndexPart];
93         Links.EnsureLinkExists(linkIndex);
94         Links.EnforceResetValues(linkIndex);
95     }
96 }

```

```

88         this.DeleteAllUsages(linkIndex);
89         Links.Delete(linkIndex);
90     }
91 }
92 }

```

./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;
32             public IList<TLink> After;
33
34             public Transition(IList<TLink> before, IList<TLink> after)
35             {
36                 Before = before;
37                 After = after;
38             }
39         }
40
41         //public static readonly TLink NullConstant = Use<LinksConstants<TLink>>.Single.Null;
42         //public static readonly IReadOnlyList<TLink> NullLink = new
43         ↪ ReadOnlyCollection<TLink>(new List<TLink> { NullConstant, NullConstant, NullConstant
44         ↪ });
45
46         // TODO: Подумать о том, как реализовать древовидный Restriction и Substitution
47         ↪ (Links-Expression)
48         public TLink Trigger(IList<TLink> restriction, Func<IList<TLink>, IList<TLink>, TLink>
49         ↪ matchedHandler, IList<TLink> substitution, Func<IList<TLink>, IList<TLink>, TLink>
50         ↪ substitutedHandler)
51         {
52             ////List<Transition> transitions = null;
53             ////if (!restriction.IsNullOrEmpty())
54             ////{
55             ////    // Есть причина делать проход (чтение)
56             ////    if (matchedHandler != null)
57             ////    {
58             ////        if (!substitution.IsNullOrEmpty())
59             ////        {
60             ////            // restriction => { 0, 0, 0 } | { 0 } // Create
61             ////            // substitution => { itself, 0, 0 } | { itself, itself, itself } //
62             ↪ Create / Update
63             ////            // substitution => { 0, 0, 0 } | { 0 } // Delete
64             ////            transitions = new List<Transition>();
65             ////            if (Equals(substitution[Constants.IndexPart], Constants.Null))
66             ////            {
67             ////                // If index is Null, that means we always ignore every other
68             ↪ value (they are also Null by definition)
69             ////            var matchDecision = matchedHandler(, NullLink);
70             ////            if (Equals(matchDecision, Constants.Break))
71             ////                return false;
72             ////            if (!Equals(matchDecision, Constants.Skip))

```

```

62         transitions.Add(new Transition(matchedLink, newValue));
63     }
64     else
65     {
66         Func<T, bool> handler;
67         handler = link =>
68         {
69             var matchedLink = Memory.GetLinkValue(link);
70             var newValue = Memory.GetLinkValue(link);
71             newValue[Constants.IndexPart] = Constants.Itself;
72             newValue[Constants.SourcePart] =
↳ Equals(substitution[Constants.SourcePart], Constants.Itself) ?
↳ matchedLink[Constants.IndexPart] : substitution[Constants.SourcePart];
73             newValue[Constants.TargetPart] =
↳ Equals(substitution[Constants.TargetPart], Constants.Itself) ?
↳ matchedLink[Constants.IndexPart] : substitution[Constants.TargetPart];
74             var matchDecision = matchedHandler(matchedLink, newValue);
75             if (Equals(matchDecision, Constants.Break))
76                 return false;
77             if (!Equals(matchDecision, Constants.Skip))
78                 transitions.Add(new Transition(matchedLink, newValue));
79             return true;
80         };
81         if (!Memory.Each(handler, restriction))
82             return Constants.Break;
83     }
84 }
85 else
86 {
87     Func<T, bool> handler = link =>
88     {
89         var matchedLink = Memory.GetLinkValue(link);
90         var matchDecision = matchedHandler(matchedLink, matchedLink);
91         return !Equals(matchDecision, Constants.Break);
92     };
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<ILink<T>> matchedLinks = null;
150     //     if (matchedHandler != null)
151     //     {
152     //         matchedLinks = new List<ILink<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     //         {
171     //             var matchedLink = matchedLinks[i];
172     //             if (substitutedHandler != null)
173     //             {
174     //                 var newValue = new List<T>(); // TODO: Prepare value to update here
175     //                 // TODO: Decide is it actually needed to use Before and After
176     ↪ substitution handling.
177     //                 var substitutedDecision = substitutedHandler(matchedLink,
178     ↪ newValue);
179     //                 if (Equals(substitutedDecision, Constants.Break))
180     //                     return Constants.Break;
181     //                 if (Equals(substitutedDecision, Constants.Continue))
182     //                 {
183     //                     // Actual update here
184     //                     Memory.SetLinkValue(newValue);
185     //                 }
186     //                 if (Equals(substitutedDecision, Constants.Skip))
187     //                 {
188     //                     // Cancel the update. TODO: decide use separate Cancel
189     ↪ constant or Skip is enough?
190     //                 }
191     //             }
192     //         }
193     //     }
194     // }
195     return Constants.Continue;
196 }
197
198 public TLink Trigger(ILink<TLink> patternOrCondition, Func<ILink<TLink>, TLink>
199 ↪ matchHandler, ILink<TLink> substitution, Func<ILink<TLink>, ILink<TLink>, TLink>
200 ↪ substitutionHandler)
201 {
202     if (patternOrCondition.IsNullOrEmpty() && substitution.IsNullOrEmpty())
203     {
204         return Constants.Continue;
205     }
206     else if (patternOrCondition.EqualTo(substitution)) // Should be Each here TODO:
207     ↪ Check if it is a correct condition
208     {
209         // Or it only applies to trigger without matchHandler.
210         throw new NotImplementedException();
211     }
212     else if (!substitution.IsNullOrEmpty()) // Creation

```

```

206 {
207     var before = ArrayPool<TLink>.Empty;
208     // Что должно означать False здесь? Остановиться (перестать идти) или пропустить
209     → (пройти мимо) или пустить (взять)?
210     if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
211     → Constants.Break))
212     {
213         return Constants.Break;
214     }
215     var after = (IList<TLink>)substitution.ToArray();
216     if (_equalityComparer.Equals(after[0], default))
217     {
218         var newLink = Links.Create();
219         after[0] = newLink;
220     }
221     if (substitution.Count == 1)
222     {
223         after = Links.GetLink(substitution[0]);
224     }
225     else if (substitution.Count == 3)
226     {
227         //Links.Create(after);
228     }
229     else
230     {
231         throw new NotSupportedException();
232     }
233     if (matchHandler != null)
234     {
235         return substitutionHandler(before, after);
236     }
237     return Constants.Continue;
238 }
239 else if (!patternOrCondition.IsNullOrEmpty()) // Deletion
240 {
241     if (patternOrCondition.Count == 1)
242     {
243         var linkToDelete = patternOrCondition[0];
244         var before = Links.GetLink(linkToDelete);
245         if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
246         → Constants.Break))
247         {
248             return Constants.Break;
249         }
250         var after = ArrayPool<TLink>.Empty;
251         Links.Update(linkToDelete, Constants.Null, Constants.Null);
252         Links.Delete(linkToDelete);
253         if (matchHandler != null)
254         {
255             return substitutionHandler(before, after);
256         }
257         return Constants.Continue;
258     }
259     else
260     {
261         throw new NotSupportedException();
262     }
263 }
264 else // Replace / Update
265 {
266     if (patternOrCondition.Count == 1) //-V3125
267     {
268         var linkToUpdate = patternOrCondition[0];
269         var before = Links.GetLink(linkToUpdate);
270         if (matchHandler != null && _equalityComparer.Equals(matchHandler(before),
271         → Constants.Break))
272         {
273             return Constants.Break;
274         }
275         var after = (IList<TLink>)substitution.ToArray(); //-V3125
276         if (_equalityComparer.Equals(after[0], default))
277         {
278             after[0] = linkToUpdate;
279         }
280         if (substitution.Count == 1)
281         {
282             if (!_equalityComparer.Equals(substitution[0], linkToUpdate))
283             {

```


./Platform.Data.Doubles/Douplet.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.Doubles
7 {
8     public struct Douplet<T> : IEquatable<Douplet<T>>
9     {
10         private static readonly EqualityComparer<T> _equalityComparer =
11             ↳ EqualityComparer<T>.Default;
12
13         public T Source { get; set; }
14         public T Target { get; set; }
15
16         public Douplet(T source, T target)
17         {
18             Source = source;
19             Target = target;
20         }
21
22         public override string ToString() => $"{Source}->{Target}";
23
24         public bool Equals(Douplet<T> other) => _equalityComparer.Equals(Source, other.Source)
25             ↳ && _equalityComparer.Equals(Target, other.Target);
26
27         public override bool Equals(object obj) => obj is Douplet<T> doublet ?
28             ↳ base.Equals(doublet) : false;
29
30         public override int GetHashCode() => (Source, Target).GetHashCode();
31     }
32 }
```

./Platform.Data.Doubles/Hybrid.cs

```
1 using System;
2 using System.Reflection;
3 using System.Reflection.Emit;
4 using Platform.Reflection;
5 using Platform.Converters;
6 using Platform.Exceptions;
7
8 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Data.Doubles
11 {
12     public class Hybrid<T>
13     {
14         private static readonly Func<object, T> _absAndConvert;
15         private static readonly Func<object, T> _absAndNegateAndConvert;
16
17         static Hybrid()
18         {
19             _absAndConvert = DelegateHelpers.Compile<Func<object, T>>(emitter =>
20             {
21                 Ensure.Always.IsUnsignedInteger<T>();
22                 emitter.LoadArgument(0);
23                 var signedVersion = NumericType<T>.SignedVersion;
24                 var signedVersionField =
25                     ↳ typeof(NumericType<T>).GetTypeInfo().GetField("SignedVersion",
26                     ↳ BindingFlags.Static | BindingFlags.Public);
27                 //emitter.LoadField(signedVersionField);
28                 emitter.Emit(OpCodes.Ldsfld, signedVersionField);
29                 var changeTypeMethod = typeof(Convert).GetTypeInfo().GetMethod("ChangeType",
30                     ↳ Types<object, Type>.Array);
31                 emitter.Call(changeTypeMethod);
32                 emitter.UnboxValue(signedVersion);
33                 var absMethod = typeof(Math).GetTypeInfo().GetMethod("Abs", new[] {
34                     ↳ signedVersion });
35                 emitter.Call(absMethod);
36                 var unsignedMethod = typeof(To).GetTypeInfo().GetMethod("Unsigned", new[] {
37                     ↳ signedVersion });
38                 emitter.Call(unsignedMethod);
39                 emitter.Return();
40             });
41             _absAndNegateAndConvert = DelegateHelpers.Compile<Func<object, T>>(emitter =>
42             {
43                 Ensure.Always.IsUnsignedInteger<T>();
44                 emitter.LoadArgument(0);
45             });
46         }
47     }
48 }
```

```

var signedVersion = NumericType<T>.SignedVersion;
var signedVersionField =
    ↳ typeof(NumericType<T>).GetTypeInfo().GetField("SignedVersion",
    ↳ BindingFlags.Static | BindingFlags.Public);
//emitter.LoadField(signedVersionField);
emitter.Emit(OpCodes.Ldsfld, signedVersionField);
var changeTypeMethod = typeof(Convert).GetTypeInfo().GetMethod("ChangeType",
    ↳ Types<object, Type>.Array);
emitter.Call(changeTypeMethod);
emitter.UnboxValue(signedVersion);
var absMethod = typeof(Math).GetTypeInfo().GetMethod("Abs", new[] {
    ↳ signedVersion });
emitter.Call(absMethod);
var negateMethod = typeof(Platform.Numbers.Math).GetTypeInfo().GetMethod("Negate",
    ↳ ").MakeGenericMethod(signedVersion);
emitter.Call(negateMethod);
var unsignedMethod = typeof(To).GetTypeInfo().GetMethod("Unsigned", new[] {
    ↳ signedVersion });
emitter.Call(unsignedMethod);
emitter.Return();
});
}

public readonly T Value;
public bool IsNothing => Convert.ToInt64(To.Signed(Value)) == 0;
public bool IsInternal => Convert.ToInt64(To.Signed(Value)) > 0;
public bool IsExternal => Convert.ToInt64(To.Signed(Value)) < 0;
public long AbsoluteValue =>
    ↳ Platform.Numbers.Math.Abs(Convert.ToInt64(To.Signed(Value)));

public Hybrid(T value)
{
    Ensure.OnDebug.IsUnsignedInteger<T>();
    Value = value;
}

public Hybrid(object value) => Value = To.UnsignedAs<T>(Convert.ChangeType(value,
    ↳ NumericType<T>.SignedVersion));

public Hybrid(object value, bool isExternal)
{
    //var signedType = Type<T>.SignedVersion;
    //var signedValue = Convert.ChangeType(value, signedType);
    //var abs = typeof(Platform.Numbers.Math).GetTypeInfo().GetMethod("Abs").MakeGeneric
    ↳ Method(signedType);
    //var negate = typeof(Platform.Numbers.Math).GetTypeInfo().GetMethod("Negate").MakeG
    ↳ enericMethod(signedType);
    //var absoluteValue = abs.Invoke(null, new[] { signedValue });
    //var resultValue = isExternal ? negate.Invoke(null, new[] { absoluteValue }) :
    ↳ absoluteValue;
    //Value = To.UnsignedAs<T>(resultValue);
    if (isExternal)
    {
        Value = _absAndNegateAndConvert(value);
    }
    else
    {
        Value = _absAndConvert(value);
    }
}

public static implicit operator Hybrid<T>(T integer) => new Hybrid<T>(integer);
public static explicit operator Hybrid<T>(ulong integer) => new Hybrid<T>(integer);
public static explicit operator Hybrid<T>(long integer) => new Hybrid<T>(integer);
public static explicit operator Hybrid<T>(uint integer) => new Hybrid<T>(integer);
public static explicit operator Hybrid<T>(int integer) => new Hybrid<T>(integer);
public static explicit operator Hybrid<T>(ushort integer) => new Hybrid<T>(integer);
public static explicit operator Hybrid<T>(short integer) => new Hybrid<T>(integer);
public static explicit operator Hybrid<T>(byte integer) => new Hybrid<T>(integer);
public static explicit operator Hybrid<T>(sbyte integer) => new Hybrid<T>(integer);

```

```

107     public static implicit operator T(Hybrid<T> hybrid) => hybrid.Value;
108
109     public static explicit operator ulong(Hybrid<T> hybrid) =>
110         ↪ Convert.ToUInt64(hybrid.Value);
111
112     public static explicit operator long(Hybrid<T> hybrid) => hybrid.AbsoluteValue;
113
114     public static explicit operator uint(Hybrid<T> hybrid) => Convert.ToUInt32(hybrid.Value);
115
116     public static explicit operator int(Hybrid<T> hybrid) =>
117         ↪ Convert.ToInt32(hybrid.AbsoluteValue);
118
119     public static explicit operator ushort(Hybrid<T> hybrid) =>
120         ↪ Convert.ToUInt16(hybrid.Value);
121
122     public static explicit operator short(Hybrid<T> hybrid) =>
123         ↪ Convert.ToInt16(hybrid.AbsoluteValue);
124
125     public static explicit operator byte(Hybrid<T> hybrid) => Convert.ToByte(hybrid.Value);
126
127     public static explicit operator sbyte(Hybrid<T> hybrid) =>
128         ↪ Convert.ToSByte(hybrid.AbsoluteValue);
129
130     public override string ToString() => IsNothing ? default(T) == null ? "Nothing" :
131         ↪ default(T).ToString() : IsExternal ? $"{<AbsoluteValue>}" : Value.ToString();
132 }

```

./Platform.Data.Doublets/ILinks.cs

```

1  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
2
3  using System.Collections.Generic;
4
5  namespace Platform.Data.Doublets
6  {
7      public interface ILinks<TLink> : ILinks<TLink, LinksConstants<TLink>>
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 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 static class ILinksExtensions
19     {
20         public static void RunRandomCreations<TLink>(this ILinks<TLink> links, long
21             ↪ amountOfCreations)
22         {
23             for (long i = 0; i < amountOfCreations; i++)
24             {
25                 var linksAddressRange = new Range<ulong>(0, (Integer<TLink>)links.Count());
26                 Integer<TLink> source = RandomHelpers.Default.NextUInt64(linksAddressRange);
27                 Integer<TLink> target = RandomHelpers.Default.NextUInt64(linksAddressRange);
28                 links.CreateAndUpdate(source, target);
29             }
30
31             public static void RunRandomSearches<TLink>(this ILinks<TLink> links, long
32                 ↪ amountOfSearches)
33             {
34                 for (long i = 0; i < amountOfSearches; i++)
35                 {
36                     var linkAddressRange = new Range<ulong>(1, (Integer<TLink>)links.Count());

```

```

36         Integer<TLink> source = RandomHelpers.Default.NextUInt64(linkAddressRange);
37         Integer<TLink> target = RandomHelpers.Default.NextUInt64(linkAddressRange);
38         links.SearchOrDefault(source, target);
39     }
40 }
41
42 public static void RunRandomDeletions<TLink>(this ILinks<TLink> links, long
↳ amountOfDeletions)
43 {
44     var min = (ulong)amountOfDeletions > (Integer<TLink>)links.Count() ? 1 :
↳ (Integer<TLink>)links.Count() - (ulong)amountOfDeletions;
45     for (long i = 0; i < amountOfDeletions; i++)
46     {
47         var linksAddressRange = new Range<ulong>(min, (Integer<TLink>)links.Count());
48         Integer<TLink> link = RandomHelpers.Default.NextUInt64(linksAddressRange);
49         links.Delete(link);
50         if ((Integer<TLink>)links.Count() < min)
51         {
52             break;
53         }
54     }
55 }
56
57 public static void Delete<TLink>(this ILinks<TLink> links, TLink linkToDelete) =>
↳ links.Delete(new LinkAddress<TLink>(linkToDelete));
58
59 /// <remarks>
60 /// TODO: Возможно есть очень простой способ это сделать.
61 /// (Например просто удалить файл, или изменить его размер таким образом,
62 /// чтобы удалился весь контент)
63 /// Например через _header->AllocatedLinks в ResizableDirectMemoryLinks
64 /// </remarks>
65 public static void DeleteAll<TLink>(this ILinks<TLink> links)
66 {
67     var equalityComparer = EqualityComparer<TLink>.Default;
68     var comparer = Comparer<TLink>.Default;
69     for (var i = links.Count(); comparer.Compare(i, default) > 0; i =
↳ Arithmetic.Decrement(i))
70     {
71         links.Delete(i);
72         if (!equalityComparer.Equals(links.Count(), Arithmetic.Decrement(i)))
73         {
74             i = links.Count();
75         }
76     }
77 }
78
79 public static TLink First<TLink>(this ILinks<TLink> links)
80 {
81     TLink firstLink = default;
82     var equalityComparer = EqualityComparer<TLink>.Default;
83     if (equalityComparer.Equals(links.Count(), default))
84     {
85         throw new InvalidOperationException("В хранилище нет связей.");
86     }
87     links.Each(links.Constants.Any, links.Constants.Any, link =>
88     {
89         firstLink = link[links.Constants.IndexPart];
90         return links.Constants.Break;
91     });
92     if (equalityComparer.Equals(firstLink, default))
93     {
94         throw new InvalidOperationException("В процессе поиска по хранилищу не было
↳ найдено связей.");
95     }
96     return firstLink;
97 }
98
99 #region Paths
100
101 /// <remarks>
102 /// TODO: Как так? Как то что ниже может быть корректно?
103 /// Скорее всего практически не применимо
104 /// Предполагалось, что можно было конвертировать формируемый в проходе через
↳ SequenceWalker
105 /// Stack в конкретный путь из Source, Target до связи, но это не всегда так.
106 /// TODO: Возможно нужен метод, который именно выбрасывает исключения (EnsurePathExists)
107 /// </remarks>

```

```

108 public static bool CheckPathExistance<TLink>(this ILinks<TLink> links, params TLink[]
109     ↪ path)
110 {
111     var current = path[0];
112     //EnsureLinkExists(current, "path");
113     if (!links.Exists(current))
114     {
115         return false;
116     }
117     var equalityComparer = EqualityComparer<TLink>.Default;
118     var constants = links.Constants;
119     for (var i = 1; i < path.Length; i++)
120     {
121         var next = path[i];
122         var values = links.GetLink(current);
123         var source = values[constants.SourcePart];
124         var target = values[constants.TargetPart];
125         if (equalityComparer.Equals(source, target) && equalityComparer.Equals(source,
126             ↪ next))
127         {
128             //throw new InvalidOperationException(string.Format("Невозможно выбрать
129             ↪ путь, так как и Source и Target совпадают с элементом пути {0}.", next));
130             return false;
131         }
132         if (!equalityComparer.Equals(next, source) && !equalityComparer.Equals(next,
133             ↪ target))
134         {
135             //throw new InvalidOperationException(string.Format("Невозможно продолжить
136             ↪ путь через элемент пути {0}", next));
137             return false;
138         }
139         current = next;
140     }
141     return true;
142 }
143
144 /// <remarks>
145 /// Может потребовать дополнительного стека для PathElement's при использовании
146 ↪ SequenceWalker.
147 /// </remarks>
148 public static TLink GetByKeyes<TLink>(this ILinks<TLink> links, TLink root, params int[]
149     ↪ path)
150 {
151     links.EnsureLinkExists(root, "root");
152     var currentLink = root;
153     for (var i = 0; i < path.Length; i++)
154     {
155         currentLink = links.GetLink(currentLink)[path[i]];
156     }
157     return currentLink;
158 }
159
160 public static TLink GetSquareMatrixSequenceElementByIndex<TLink>(this ILinks<TLink>
161     ↪ links, TLink root, ulong size, ulong index)
162 {
163     var constants = links.Constants;
164     var source = constants.SourcePart;
165     var target = constants.TargetPart;
166     if (!Platform.Numbers.Math.IsPowerOfTwo(size))
167     {
168         throw new ArgumentOutOfRangeException(nameof(size), "Sequences with sizes other
169         ↪ than powers of two are not supported.");
170     }
171     var path = new BitArray(BitConverter.GetBytes(index));
172     var length = Bit.GetLowestPosition(size);
173     links.EnsureLinkExists(root, "root");
174     var currentLink = root;
175     for (var i = length - 1; i >= 0; i--)
176     {
177         currentLink = links.GetLink(currentLink)[path[i] ? target : source];
178     }
179     return currentLink;
180 }
181
182 #endregion
183
184 /// <summary>
185 /// Возвращает индекс указанной связи.

```



```

177     /// </summary>
178     /// <param name="links">Хранилище связей.</param>
179     /// <param name="link">Связь представленная списком, состоящим из её адреса и
    ↳ содержимого.</param>
180     /// <returns>Индекс начальной связи для указанной связи.</returns>
181     [MethodImpl(MethodImplOptions.AggressiveInlining)]
182     public static TLink GetIndex<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
    ↳ link[links.Constants.IndexPart];
183
184     /// <summary>
185     /// Возвращает индекс начальной (Source) связи для указанной связи.
186     /// </summary>
187     /// <param name="links">Хранилище связей.</param>
188     /// <param name="link">Индекс связи.</param>
189     /// <returns>Индекс начальной связи для указанной связи.</returns>
190     [MethodImpl(MethodImplOptions.AggressiveInlining)]
191     public static TLink GetSource<TLink>(this ILinks<TLink> links, TLink link) =>
    ↳ links.GetLink(link)[links.Constants.SourcePart];
192
193     /// <summary>
194     /// Возвращает индекс начальной (Source) связи для указанной связи.
195     /// </summary>
196     /// <param name="links">Хранилище связей.</param>
197     /// <param name="link">Связь представленная списком, состоящим из её адреса и
    ↳ содержимого.</param>
198     /// <returns>Индекс начальной связи для указанной связи.</returns>
199     [MethodImpl(MethodImplOptions.AggressiveInlining)]
200     public static TLink GetSource<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
    ↳ link[links.Constants.SourcePart];
201
202     /// <summary>
203     /// Возвращает индекс конечной (Target) связи для указанной связи.
204     /// </summary>
205     /// <param name="links">Хранилище связей.</param>
206     /// <param name="link">Индекс связи.</param>
207     /// <returns>Индекс конечной связи для указанной связи.</returns>
208     [MethodImpl(MethodImplOptions.AggressiveInlining)]
209     public static TLink GetTarget<TLink>(this ILinks<TLink> links, TLink link) =>
    ↳ links.GetLink(link)[links.Constants.TargetPart];
210
211     /// <summary>
212     /// Возвращает индекс конечной (Target) связи для указанной связи.
213     /// </summary>
214     /// <param name="links">Хранилище связей.</param>
215     /// <param name="link">Связь представленная списком, состоящим из её адреса и
    ↳ содержимого.</param>
216     /// <returns>Индекс конечной связи для указанной связи.</returns>
217     [MethodImpl(MethodImplOptions.AggressiveInlining)]
218     public static TLink GetTarget<TLink>(this ILinks<TLink> links, IList<TLink> link) =>
    ↳ link[links.Constants.TargetPart];
219
220     /// <summary>
221     /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
    ↳ (handler) для каждой подходящей связи.
222     /// </summary>
223     /// <param name="links">Хранилище связей.</param>
224     /// <param name="handler">Обработчик каждой подходящей связи.</param>
225     /// <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>
226     /// <returns>True, в случае если проход по связям не был прерван и False в обратном
    ↳ случае.</returns>
227     [MethodImpl(MethodImplOptions.AggressiveInlining)]
228     public static bool Each<TLink>(this ILinks<TLink> links, Func<IList<TLink>, TLink>
    ↳ handler, params TLink[] restrictions)
229     => EqualityComparer<TLink>.Default.Equals(links.Each(handler, restrictions),
    ↳ links.Constants.Continue);
230
231     /// <summary>
232     /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
    ↳ (handler) для каждой подходящей связи.
233     /// </summary>
234     /// <param name="links">Хранилище связей.</param>
235     /// <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>

```

```

236 /// <param name="target">Значение, определяющее соответствующие шаблону связи.
237   ↳ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве конца,
238   ↳ Constants.Any - любой конец, 1..∞ конкретный конец)</param>
239 /// <param name="handler">Обработчик каждой подходящей связи.</param>
240 /// <returns>True, в случае если проход по связям не был прерван и False в обратном
241   ↳ случае.</returns>
242 [MethodImpl(MethodImplOptions.AggressiveInlining)]
243 public static bool Each<TLink>(this ILinks<TLink> links, TLink source, TLink target,
244   ↳ Func<TLink, bool> handler)
245 {
246     var constants = links.Constants;
247     return links.Each(link => handler(link[constants.IndexPart]) ? constants.Continue :
248       ↳ constants.Break, constants.Any, source, target);
249 }
250
251 /// <summary>
252 /// Выполняет проход по всем связям, соответствующим шаблону, вызывая обработчик
253   ↳ (handler) для каждой подходящей связи.
254 /// </summary>
255 /// <param name="links">Хранилище связей.</param>
256 /// <param name="source">Значение, определяющее соответствующие шаблону связи.
257   ↳ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве начала,
258   ↳ Constants.Any - любое начало, 1..∞ конкретное начало)</param>
259 /// <param name="target">Значение, определяющее соответствующие шаблону связи.
260   ↳ (Constants.Null - 0-я связь, обозначающая ссылку на пустоту в качестве конца,
261   ↳ Constants.Any - любой конец, 1..∞ конкретный конец)</param>
262 /// <param name="handler">Обработчик каждой подходящей связи.</param>
263 /// <returns>True, в случае если проход по связям не был прерван и False в обратном
264   ↳ случае.</returns>
265 [MethodImpl(MethodImplOptions.AggressiveInlining)]
266 public static bool Each<TLink>(this ILinks<TLink> links, TLink source, TLink target,
267   ↳ Func<IList<TLink>, TLink> handler)
268 {
269     var constants = links.Constants;
270     return links.Each(handler, constants.Any, source, target);
271 }
272
273 [MethodImpl(MethodImplOptions.AggressiveInlining)]
274 public static IList<IList<TLink>> All<TLink>(this ILinks<TLink> links, params TLink[]
275   ↳ restrictions)
276 {
277     long arraySize = (Integer<TLink>)links.Count(restrictions);
278     var array = new IList<TLink>[arraySize];
279     if (arraySize > 0)
280     {
281         var filler = new ArrayFiller<IList<TLink>, TLink>(array,
282           ↳ links.Constants.Continue);
283         links.Each(filler.AddAndReturnConstant, restrictions);
284     }
285     return array;
286 }
287
288 [MethodImpl(MethodImplOptions.AggressiveInlining)]
289 public static IList<TLink> AllIndices<TLink>(this ILinks<TLink> links, params TLink[]
290   ↳ restrictions)
291 {
292     long arraySize = (Integer<TLink>)links.Count(restrictions);
293     var array = new TLink[arraySize];
294     if (arraySize > 0)
295     {
296         var filler = new ArrayFiller<TLink, TLink>(array, links.Constants.Continue);
297         links.Each(filler.AddFirstAndReturnConstant, restrictions);
298     }
299     return array;
300 }
301
302 /// <summary>
303 /// Возвращает значение, определяющее существует ли связь с указанными началом и концом
304   ↳ в хранилище связей.
305 /// </summary>
306 /// <param name="links">Хранилище связей.</param>
307 /// <param name="source">Начало связи.</param>
308 /// <param name="target">Конец связи.</param>
309 /// <returns>Значение, определяющее существует ли связь.</returns>
310 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

295 public static bool Exists<TLink>(this ILinks<TLink> links, TLink source, TLink target)
296     => Comparer<TLink>.Default.Compare(links.Count(links.Constants.Any, source, target),
297     => default) > 0;
298
299 #region Ensure
300 // TODO: May be move to EnsureExtensions or make it both there and here
301 [MethodImpl(MethodImplOptions.AggressiveInlining)]
302 public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links, TLink
303     => reference, string argumentName)
304 {
305     if (links.Constants.IsInnerReference(reference) && !links.Exists(reference))
306     {
307         throw new ArgumentLinkDoesNotExistsException<TLink>(reference, argumentName);
308     }
309 }
310 [MethodImpl(MethodImplOptions.AggressiveInlining)]
311 public static void EnsureInnerReferenceExists<TLink>(this ILinks<TLink> links,
312     => IList<TLink> restrictions, string argumentName)
313 {
314     for (int i = 0; i < restrictions.Count; i++)
315     {
316         links.EnsureInnerReferenceExists(restrictions[i], argumentName);
317     }
318 }
319 [MethodImpl(MethodImplOptions.AggressiveInlining)]
320 public static void EnsureLinkIsAnyOrExists<TLink>(this ILinks<TLink> links, IList<TLink>
321     => restrictions)
322 {
323     for (int i = 0; i < restrictions.Count; i++)
324     {
325         links.EnsureLinkIsAnyOrExists(restrictions[i], nameof(restrictions));
326     }
327 }
328 [MethodImpl(MethodImplOptions.AggressiveInlining)]
329 public static void EnsureLinkIsAnyOrExists<TLink>(this ILinks<TLink> links, TLink link,
330     => string argumentName)
331 {
332     var equalityComparer = EqualityComparer<TLink>.Default;
333     if (!equalityComparer.Equals(link, links.Constants.Any) && !links.Exists(link))
334     {
335         throw new ArgumentLinkDoesNotExistsException<TLink>(link, argumentName);
336     }
337 }
338 [MethodImpl(MethodImplOptions.AggressiveInlining)]
339 public static void EnsureLinkIsItselfOrExists<TLink>(this ILinks<TLink> links, TLink
340     => link, string argumentName)
341 {
342     var equalityComparer = EqualityComparer<TLink>.Default;
343     if (!equalityComparer.Equals(link, links.Constants.Itself) && !links.Exists(link))
344     {
345         throw new ArgumentLinkDoesNotExistsException<TLink>(link, argumentName);
346     }
347 }
348
349 /// <param name="links">Хранилище связей.</param>
350 [MethodImpl(MethodImplOptions.AggressiveInlining)]
351 public static void EnsureDoesNotExists<TLink>(this ILinks<TLink> links, TLink source,
352     => TLink target)
353 {
354     if (links.Exists(source, target))
355     {
356         throw new LinkWithSameValueAlreadyExistsException();
357     }
358 }
359
360 /// <param name="links">Хранилище связей.</param>
361 public static void EnsureNoUsages<TLink>(this ILinks<TLink> links, TLink link)
362 {
363     if (links.HasUsages(link))
364     {
365         throw new ArgumentLinkHasDependenciesException<TLink>(link);
366     }
367 }

```

```

365
366 /// <param name="links">Хранилище связей.</param>
367 public static void EnsureCreated<TLink>(this ILinks<TLink> links, params TLink[]
    ↳ addresses) => links.EnsureCreated(links.Create, addresses);
368
369 /// <param name="links">Хранилище связей.</param>
370 public static void EnsurePointsCreated<TLink>(this ILinks<TLink> links, params TLink[]
    ↳ addresses) => links.EnsureCreated(links.CreatePoint, addresses);
371
372 /// <param name="links">Хранилище связей.</param>
373 public static void EnsureCreated<TLink>(this ILinks<TLink> links, Func<TLink> creator,
    ↳ params TLink[] addresses)
374 {
375     var constants = links.Constants;
376
377     var nonExistentAddresses = new HashSet<TLink>(addresses.Where(x =>
    ↳ !links.Exists(x)));
378     if (nonExistentAddresses.Count > 0)
379     {
380         var max = nonExistentAddresses.Max();
381         max = (Integer<TLink>)System.Math.Min((ulong)(Integer<TLink>)max,
    ↳ (ulong)(Integer<TLink>)constants.PossibleInnerReferencesRange.Maximum);
382         var createdLinks = new List<TLink>();
383         var equalityComparer = EqualityComparer<TLink>.Default;
384         TLink createdLink = creator();
385         while (!equalityComparer.Equals(createdLink, max))
386         {
387             createdLinks.Add(createdLink);
388         }
389         for (var i = 0; i < createdLinks.Count; i++)
390         {
391             if (!nonExistentAddresses.Contains(createdLinks[i]))
392             {
393                 links.Delete(createdLinks[i]);
394             }
395         }
396     }
397 }
398
399 #endregion
400
401 /// <param name="links">Хранилище связей.</param>
402 public static TLink CountUsages<TLink>(this ILinks<TLink> links, TLink link)
403 {
404     var constants = links.Constants;
405     var values = links.GetLink(link);
406     TLink usagesAsSource = links.Count(new Link<TLink>(constants.Any, link,
    ↳ constants.Any));
407     var equalityComparer = EqualityComparer<TLink>.Default;
408     if (equalityComparer.Equals(values[constants.SourcePart], link))
409     {
410         usagesAsSource = Arithmetic<TLink>.Decrement(usagesAsSource);
411     }
412     TLink usagesAsTarget = links.Count(new Link<TLink>(constants.Any, constants.Any,
    ↳ link));
413     if (equalityComparer.Equals(values[constants.TargetPart], link))
414     {
415         usagesAsTarget = Arithmetic<TLink>.Decrement(usagesAsTarget);
416     }
417     return Arithmetic<TLink>.Add(usagesAsSource, usagesAsTarget);
418 }
419
420 /// <param name="links">Хранилище связей.</param>
421 [MethodImpl(MethodImplOptions.AggressiveInlining)]
422 public static bool HasUsages<TLink>(this ILinks<TLink> links, TLink link) =>
    ↳ Comparer<TLink>.Default.Compare(links.CountUsages(link), Integer<TLink>.Zero) > 0;
423
424 /// <param name="links">Хранилище связей.</param>
425 [MethodImpl(MethodImplOptions.AggressiveInlining)]
426 public static bool Equals<TLink>(this ILinks<TLink> links, TLink link, TLink source,
    ↳ TLink target)
427 {
428     var constants = links.Constants;
429     var values = links.GetLink(link);
430     var equalityComparer = EqualityComparer<TLink>.Default;
431     return equalityComparer.Equals(values[constants.SourcePart], source) &&
    ↳ equalityComparer.Equals(values[constants.TargetPart], target);
432 }

```

```

433 /// <summary>
434 /// Выполняет поиск связи с указанными Source (началом) и Target (концом).
435 /// </summary>
436 /// <param name="links">Хранилище связей.</param>
437 /// <param name="source">Индекс связи, которая является началом для искомой
438   ↳ связи.</param>
439 /// <param name="target">Индекс связи, которая является концом для искомой связи.</param>
440 /// <returns>Индекс искомой связи с указанными Source (началом) и Target
441   ↳ (концом).</returns>
442 [MethodImpl(MethodImplOptions.AggressiveInlining)]
443 public static TLink SearchOrDefault<TLink>(this ILinks<TLink> links, TLink source, TLink
444   ↳ target)
445 {
446     var constants = links.Constants;
447     var setter = new Setter<TLink, TLink>(constants.Continue, constants.Break, default);
448     links.Each(setter.SetFirstAndReturnFalse, constants.Any, source, target);
449     return setter.Result;
450 }
451
452 /// <param name="links">Хранилище связей.</param>
453 [MethodImpl(MethodImplOptions.AggressiveInlining)]
454 public static TLink Create<TLink>(this ILinks<TLink> links) => links.Create(null);
455
456 /// <param name="links">Хранилище связей.</param>
457 [MethodImpl(MethodImplOptions.AggressiveInlining)]
458 public static TLink CreatePoint<TLink>(this ILinks<TLink> links)
459 {
460     var link = links.Create();
461     return links.Update(link, link, link);
462 }
463
464 /// <param name="links">Хранилище связей.</param>
465 [MethodImpl(MethodImplOptions.AggressiveInlining)]
466 public static TLink CreateAndUpdate<TLink>(this ILinks<TLink> links, TLink source, TLink
467   ↳ target) => links.Update(links.Create(), source, target);
468
469 /// <summary>
470 /// Обновляет связь с указанными началом (Source) и концом (Target)
471 /// на связь с указанными началом (NewSource) и концом (NewTarget).
472 /// </summary>
473 /// <param name="links">Хранилище связей.</param>
474 /// <param name="link">Индекс обновляемой связи.</param>
475 /// <param name="newSource">Индекс связи, которая является началом связи, на которую
476   ↳ выполняется обновление.</param>
477 /// <param name="newTarget">Индекс связи, которая является концом связи, на которую
478   ↳ выполняется обновление.</param>
479 /// <returns>Индекс обновлённой связи.</returns>
480 [MethodImpl(MethodImplOptions.AggressiveInlining)]
481 public static TLink Update<TLink>(this ILinks<TLink> links, TLink link, TLink newSource,
482   ↳ TLink newTarget) => links.Update(new LinkAddress<TLink>(link), new Link<TLink>(link,
483   ↳ newSource, newTarget));
484
485 /// <summary>
486 /// Обновляет связь с указанными началом (Source) и концом (Target)
487 /// на связь с указанными началом (NewSource) и концом (NewTarget).
488 /// </summary>
489 /// <param name="links">Хранилище связей.</param>
490 /// <param name="restrictions">Ограничения на содержимое связей. Каждое ограничение
491   ↳ может иметь значения: Constants.Null - 0-я связь, обозначающая ссылку на пустоту,
492   ↳ Itself - требование установить ссылку на себя, 1.. $\infty$  конкретный адрес другой
493   ↳ связи.</param>
494 /// <returns>Индекс обновлённой связи.</returns>
495 [MethodImpl(MethodImplOptions.AggressiveInlining)]
496 public static TLink Update<TLink>(this ILinks<TLink> links, params TLink[] restrictions)
497 {
498     if (restrictions.Length == 2)
499     {
500         return links.MergeAndDelete(restrictions[0], restrictions[1]);
501     }
502     if (restrictions.Length == 4)
503     {
504         return links.UpdateOrCreateOrGet(restrictions[0], restrictions[1],
505           ↳ restrictions[2], restrictions[3]);
506     }
507     else
508     {
509

```

```

498         return links.Update(new LinkAddress<TLink>(restrictions[0]), restrictions);
499     }
500 }
501
502 [MethodImpl(MethodImplOptions.AggressiveInlining)]
503 public static IList<TLink> ResolveConstantAsSelfReference<TLink>(this ILinks<TLink>
    ↳ links, TLink constant, IList<TLink> restrictions, IList<TLink> substitution)
504 {
505     var equalityComparer = EqualityComparer<TLink>.Default;
506     var constants = links.Constants;
507     var restrictionsIndex = restrictions[constants.IndexPart];
508     var substitutionIndex = substitution[constants.IndexPart];
509     if (equalityComparer.Equals(substitutionIndex, default))
510     {
511         substitutionIndex = restrictionsIndex;
512     }
513     var source = substitution[constants.SourcePart];
514     var target = substitution[constants.TargetPart];
515     source = equalityComparer.Equals(source, constant) ? substitutionIndex : source;
516     target = equalityComparer.Equals(target, constant) ? substitutionIndex : target;
517     return new Link<TLink>(substitutionIndex, source, target);
518 }
519
520 /// <summary>
521 /// Создаёт связь (если она не существовала), либо возвращает индекс существующей связи
    ↳ с указанными Source (началом) и Target (концом).
522 /// </summary>
523 /// <param name="links">Хранилище связей.</param>
524 /// <param name="source">Индекс связи, которая является началом на создаваемой
    ↳ связи.</param>
525 /// <param name="target">Индекс связи, которая является концом для создаваемой
    ↳ связи.</param>
526 /// <returns>Индекс связи, с указанным Source (началом) и Target (концом)</returns>
527 [MethodImpl(MethodImplOptions.AggressiveInlining)]
528 public static TLink GetOrCreate<TLink>(this ILinks<TLink> links, TLink source, TLink
    ↳ target)
529 {
530     var link = links.SearchOrDefault(source, target);
531     if (EqualityComparer<TLink>.Default.Equals(link, default))
532     {
533         link = links.CreateAndUpdate(source, target);
534     }
535     return link;
536 }
537
538 /// <summary>
539 /// Обновляет связь с указанными началом (Source) и концом (Target)
    ↳ на связь с указанными началом (NewSource) и концом (NewTarget).
540 /// </summary>
541 /// <param name="links">Хранилище связей.</param>
542 /// <param name="source">Индекс связи, которая является началом обновляемой
    ↳ связи.</param>
543 /// <param name="target">Индекс связи, которая является концом обновляемой связи.</param>
544 /// <param name="newSource">Индекс связи, которая является началом связи, на которую
    ↳ выполняется обновление.</param>
545 /// <param name="newTarget">Индекс связи, которая является концом связи, на которую
    ↳ выполняется обновление.</param>
546 /// <returns>Индекс обновлённой связи.</returns>
547 [MethodImpl(MethodImplOptions.AggressiveInlining)]
548 public static TLink UpdateOrCreateOrGet<TLink>(this ILinks<TLink> links, TLink source,
    ↳ TLink target, TLink newSource, TLink newTarget)
549 {
550     var equalityComparer = EqualityComparer<TLink>.Default;
551     var link = links.SearchOrDefault(source, target);
552     if (equalityComparer.Equals(link, default))
553     {
554         return links.CreateAndUpdate(newSource, newTarget);
555     }
556     if (equalityComparer.Equals(newSource, source) && equalityComparer.Equals(newTarget,
    ↳ target))
557     {
558         return link;
559     }
560     return links.Update(link, newSource, newTarget);
561 }
562
563 /// <summary>Удаляет связь с указанными началом (Source) и концом (Target).</summary>
564

```

```

565 /// <param name="links">Хранилище связей.</param>
566 /// <param name="source">Индекс связи, которая является началом удаляемой связи.</param>
567 /// <param name="target">Индекс связи, которая является концом удаляемой связи.</param>
568 [MethodImpl(MethodImplOptions.AggressiveInlining)]
569 public static TLink DeleteIfExists<TLink>(this ILinks<TLink> links, TLink source, TLink
    ↪ target)
570 {
571     var link = links.SearchOrDefault(source, target);
572     if (!EqualityComparer<TLink>.Default.Equals(link, default))
573     {
574         links.Delete(link);
575         return link;
576     }
577     return default;
578 }
579
580 /// <summary>Удаляет несколько связей.</summary>
581 /// <param name="links">Хранилище связей.</param>
582 /// <param name="deletedLinks">Список адресов связей к удалению.</param>
583 [MethodImpl(MethodImplOptions.AggressiveInlining)]
584 public static void DeleteMany<TLink>(this ILinks<TLink> links, IList<TLink> deletedLinks)
585 {
586     for (int i = 0; i < deletedLinks.Count; i++)
587     {
588         links.Delete(deletedLinks[i]);
589     }
590 }
591
592 /// <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>
593 public static void DeleteAllUsages<TLink>(this ILinks<TLink> links, TLink linkIndex)
594 {
595     var anyConstant = links.Constants.Any;
596     var usagesAsSourceQuery = new Link<TLink>(anyConstant, linkIndex, anyConstant);
597     links.DeleteByQuery(usagesAsSourceQuery);
598     var usagesAsTargetQuery = new Link<TLink>(anyConstant, linkIndex, anyConstant);
599     links.DeleteByQuery(usagesAsTargetQuery);
600 }
601
602 public static void DeleteByQuery<TLink>(this ILinks<TLink> links, Link<TLink> query)
603 {
604     var count = (Integer<TLink>)links.Count(query);
605     if (count > 0)
606     {
607         var queryResult = new TLink[count];
608         var queryResultFiller = new ArrayFiller<TLink, TLink>(queryResult,
            ↪ links.Constants.Continue);
609         links.Each(queryResultFiller.AddFirstAndReturnConstant, query);
610         for (var i = (long)count - 1; i >= 0; i--)
611         {
612             links.Delete(queryResult[i]);
613         }
614     }
615 }
616
617 // TODO: Move to Platform.Data
618 public static bool AreValuesReset<TLink>(this ILinks<TLink> links, TLink linkIndex)
619 {
620     var nullConstant = links.Constants.Null;
621     var equalityComparer = EqualityComparer<TLink>.Default;
622     var link = links.GetLink(linkIndex);
623     for (int i = 1; i < link.Count; i++)
624     {
625         if (!equalityComparer.Equals(link[i], nullConstant))
626         {
627             return false;
628         }
629     }
630     return true;
631 }
632
633 // TODO: Create a universal version of this method in Platform.Data (with using of for
    ↪ loop)
634 public static void ResetValues<TLink>(this ILinks<TLink> links, TLink linkIndex)
635 {
636     var nullConstant = links.Constants.Null;
637     var updateRequest = new Link<TLink>(linkIndex, nullConstant, nullConstant);

```

```

638     links.Update(updateRequest);
639 }
640
641 // TODO: Create a universal version of this method in Platform.Data (with using of for
642 → loop)
643 public static void EnforceResetValues<TLink>(this ILinks<TLink> links, TLink linkIndex)
644 {
645     if (!links.AreValuesReset(linkIndex))
646     {
647         links.ResetValues(linkIndex);
648     }
649 }
650
651 /// <summary>
652 /// Merging two usages graphs, all children of old link moved to be children of new link
653 → or deleted.
654 /// </summary>
655 public static TLink MergeUsages<TLink>(this ILinks<TLink> links, TLink oldLinkIndex,
656 → TLink newLinkIndex)
657 {
658     var equalityComparer = EqualityComparer<TLink>.Default;
659     if (!equalityComparer.Equals(oldLinkIndex, newLinkIndex))
660     {
661         var constants = links.Constants;
662         var usagesAsSourceQuery = new Link<TLink>(constants.Any, oldLinkIndex,
663 → constants.Any);
664         long usagesAsSourceCount = (Integer<TLink>)links.Count(usagesAsSourceQuery);
665         var usagesAsTargetQuery = new Link<TLink>(constants.Any, constants.Any,
666 → oldLinkIndex);
667         long usagesAsTargetCount = (Integer<TLink>)links.Count(usagesAsTargetQuery);
668         var isStandalonePoint = Point<TLink>.IsFullPoint(links.GetLink(oldLinkIndex)) &&
669 → usagesAsSourceCount == 1 && usagesAsTargetCount == 1;
670         if (!isStandalonePoint)
671         {
672             var totalUsages = usagesAsSourceCount + usagesAsTargetCount;
673             if (totalUsages > 0)
674             {
675                 var usages = ArrayPool.Allocate<TLink>(totalUsages);
676                 var usagesFiller = new ArrayFiller<TLink, TLink>(usages,
677 → links.Constants.Continue);
678                 var i = 0L;
679                 if (usagesAsSourceCount > 0)
680                 {
681                     links.Each(usagesFiller.AddFirstAndReturnConstant,
682 → usagesAsSourceQuery);
683                     for (; i < usagesAsSourceCount; i++)
684                     {
685                         var usage = usages[i];
686                         if (!equalityComparer.Equals(usage, oldLinkIndex))
687                         {
688                             links.Update(usage, newLinkIndex, links.GetTarget(usage));
689                         }
690                     }
691                 }
692                 if (usagesAsTargetCount > 0)
693                 {
694                     links.Each(usagesFiller.AddFirstAndReturnConstant,
695 → usagesAsTargetQuery);
696                     for (; i < usages.Length; i++)
697                     {
698                         var usage = usages[i];
699                         if (!equalityComparer.Equals(usage, oldLinkIndex))
700                         {
701                             links.Update(usage, links.GetSource(usage), newLinkIndex);
702                         }
703                     }
704                 }
705                 ArrayPool.Free(usages);
706             }
707         }
708     }
709     return newLinkIndex;
710 }
711
712 /// <summary>
713 /// Replace one link with another (replaced link is deleted, children are updated or
714 → deleted).

```



```

705     /// </summary>
706     [MethodImpl(MethodImplOptions.AggressiveInlining)]
707     public static TLink MergeAndDelete<TLink>(this ILinks<TLink> links, TLink oldLinkIndex,
708     ↪ TLink newLinkIndex)
709     {
710         var equalityComparer = EqualityComparer<TLink>.Default;
711         if (!equalityComparer.Equals(oldLinkIndex, newLinkIndex))
712         {
713             links.MergeUsages(oldLinkIndex, newLinkIndex);
714             links.Delete(oldLinkIndex);
715         }
716         return newLinkIndex;
717     }
718     public static ILinks<TLink>
719     ↪ DecorateWithAutomaticUniquenessAndUsagesResolution<TLink>(this ILinks<TLink> links)
720     {
721         links = new LinksCascadeUsagesResolver<TLink>(links);
722         links = new NonNullContentsLinkDeletionResolver<TLink>(links);
723         links = new LinksCascadeUniquenessAndUsagesResolver<TLink>(links);
724         return links;
725     }
726 }

```

./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)
19         : base(links)
20         {
21             _frequencyMarker = frequencyMarker;
22             _unaryOne = unaryOne;
23             _unaryNumberIncrementer = unaryNumberIncrementer;
24         }
25
26         public TLink Increment(TLink frequency)
27         {
28             if (_equalityComparer.Equals(frequency, default))
29             {
30                 return Links.GetOrCreate(_unaryOne, _frequencyMarker);
31             }
32             var source = Links.GetSource(frequency);
33             var incrementedSource = _unaryNumberIncrementer.Increment(source);
34             return Links.GetOrCreate(incrementedSource, _frequencyMarker);
35         }
36     }
37 }

```

./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     public UnaryNumberIncrementer(ILinks<TLink> links, TLink unaryOne) : base(links) =>
15         ↪ _unaryOne = unaryOne;
16
17     public TLink Increment(TLink unaryNumber)
18     {
19         if (_equalityComparer.Equals(unaryNumber, _unaryOne))
20         {
21             return Links.GetOrCreate(_unaryOne, _unaryOne);
22         }
23         var source = Links.GetSource(unaryNumber);
24         var target = Links.GetTarget(unaryNumber);
25         if (_equalityComparer.Equals(source, target))
26         {
27             return Links.GetOrCreate(unaryNumber, _unaryOne);
28         }
29         else
30         {
31             return Links.GetOrCreate(source, Increment(target));
32         }
33     }
34 }

```

./Platform.Data.Doublets/ISynchronizedLinks.cs

```

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

```

./Platform.Data.Doublets/Link.cs

```

1  using System;
2  using System.Collections;
3  using System.Collections.Generic;
4  using Platform.Exceptions;
5  using Platform.Ranges;
6  using Platform.Singletons;
7  using Platform.Collections.Lists;
8
9  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11 namespace Platform.Data.Doublets
12 {
13     /// <summary>
14     /// Структура описывающая уникальную связь.
15     /// </summary>
16     public struct Link<TLink> : IEquatable<Link<TLink>>, IReadOnlyList<TLink>, IList<TLink>
17     {
18         public static readonly Link<TLink> Null = new Link<TLink>();
19
20         private static readonly LinksConstants<TLink> _constants =
21             ↪ Default<LinksConstants<TLink>>.Instance;
22         private static readonly EqualityComparer<TLink> _equalityComparer =
23             ↪ EqualityComparer<TLink>.Default;
24
25         private const int Length = 3;
26
27         public readonly TLink Index;
28         public readonly TLink Source;
29         public readonly TLink Target;
30
31         public Link(params TLink[] values)
32         {
33             Index = values.Length > _constants.IndexPart ? values[_constants.IndexPart] :
34                 ↪ _constants.Null;
35             Source = values.Length > _constants.SourcePart ? values[_constants.SourcePart] :
36                 ↪ _constants.Null;
37             Target = values.Length > _constants.TargetPart ? values[_constants.TargetPart] :
38                 ↪ _constants.Null;
39         }
40
41         public Link(IList<TLink> values)
42         {
43             Index = values.Count > _constants.IndexPart ? values[_constants.IndexPart] :
44                 ↪ _constants.Null;

```

```

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

```

```

109     IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
110
111     public IEnumerator<TLink> GetEnumerator()
112     {
113         yield return Index;
114         yield return Source;
115         yield return Target;
116     }
117
118     public void Add(TLink item) => throw new NotSupportedException();
119
120     public void Clear() => throw new NotSupportedException();
121
122     public bool Contains(TLink item) => IndexOf(item) >= 0;
123
124     public void CopyTo(TLink[] array, int arrayIndex)
125     {
126         Ensure.OnDebug.ArgumentNotNull(array, nameof(array));
127         Ensure.OnDebug.ArgumentInRange(arrayIndex, new Range<int>(0, array.Length - 1),
128             ↪ 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 }

```

./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 }

```

```

9     }
10 }

```

./Platform.Data.Doublets/Numbers/Raw/AddressToRawNumberConverter.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.Numbers.Raw
6 {
7     public class AddressToRawNumberConverter<TLink> : IConverter<TLink>
8     {
9         public TLink Convert(TLink source) => new Hybrid<TLink>(source, isExternal: true);
10    }
11 }

```

./Platform.Data.Doublets/Numbers/Raw/RawNumberToAddressConverter.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.Numbers.Raw
7 {
8     public class RawNumberToAddressConverter<TLink> : IConverter<TLink>
9     {
10        public TLink Convert(TLink source) => (Integer<TLink>)new
            ↳ Hybrid<TLink>(source).AbsoluteValue;
11    }
12 }

```

./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>,
        ↳ IConverter<TLink>
11    {
12        private static readonly EqualityComparer<TLink> _equalityComparer =
            ↳ EqualityComparer<TLink>.Default;
13
14        private readonly IConverter<int, TLink> _powerOf2ToUnaryNumberConverter;
15
16        public AddressToUnaryNumberConverter(ILinks<TLink> links, IConverter<int, TLink>
            ↳ powerOf2ToUnaryNumberConverter) : base(links) => _powerOf2ToUnaryNumberConverter =
            ↳ powerOf2ToUnaryNumberConverter;
17
18        public TLink Convert(TLink number)
19        {
20            var nullConstant = Links.Constants.Null;
21            var one = Integer<TLink>.One;
22            var target = nullConstant;
23            for (int i = 0; !_equalityComparer.Equals(number, default) && i <
                ↳ NumericType<TLink>.BitsLength; i++)
24            {
25                if (_equalityComparer.Equals(Bit.And(number, one), one))
26                {
27                    target = _equalityComparer.Equals(target, nullConstant)
28                        ? _powerOf2ToUnaryNumberConverter.Convert(i)
29                        : Links.GetOrCreate(_powerOf2ToUnaryNumberConverter.Convert(i), target);
30                }
31                number = Bit.ShiftRight(number, 1);
32            }
33            return target;
34        }
35    }
36 }

```

./Platform.Data.Doublets/Numbers/Unary/LinkToToltsFrequencyNumberConveter.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             if (!_equalityComparer.Equals(_unaryNumberPowersOf2[power], default))
29             {
30                 return _unaryNumberPowersOf2[power];
31             }
32             var previousPowerOf2 = Convert(power - 1);
33             var powerOf2 = Links.GetOrCreate(previousPowerOf2, previousPowerOf2);
34             _unaryNumberPowersOf2[power] = powerOf2;
35             return powerOf2;
36         }
37     }
38 }

```

./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                 result = Arithmetic<TLink>.Add(result, lastValue);
70                 return result;
71             }
72         }
73
74         [MethodImpl(MethodImplOptions.AggressiveInlining)]
75         private static TLink Double(TLink number) => (Integer<TLink>)((Integer<TLink>)number *
76             ⇨ 2UL);
77     }
```

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

```
1 using System.Collections.Generic;
2 using System.Runtime.CompilerServices;
3 using Platform.Interfaces;
4 using Platform.Reflection;
5 using Platform.Numbers;
6
7 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
8
9 namespace Platform.Data.Doublets.Numbers.Unary
10 {
11     public class UnaryNumberToAddressOrOperationConverter<TLink> : LinksOperatorBase<TLink>,
12         ⇨ IConverter<TLink>
13     {
14         private static readonly EqualityComparer<TLink> _equalityComparer =
15             ⇨ EqualityComparer<TLink>.Default;
16
17         private readonly IDictionary<TLink, int> _unaryNumberPowerOf2Indicies;
18
19         public UnaryNumberToAddressOrOperationConverter(ILinks<TLink> links, IConverter<int>,
20             ⇨ TLink> powerOf2ToUnaryNumberConverter)
21             : base(links)
22         {
23             _unaryNumberPowerOf2Indicies = new Dictionary<TLink, int>();
24             for (int i = 0; i < NumericType<TLink>.BitsLength; i++)
25             {
26                 _unaryNumberPowerOf2Indicies.Add(powerOf2ToUnaryNumberConverter.Convert(i), i);
27             }
28
29             public TLink Convert(TLink sourceNumber)
30             {
31                 var nullConstant = Links.Constants.Null;
32                 var source = sourceNumber;
33                 var target = nullConstant;
34                 if (!_equalityComparer.Equals(source, nullConstant))
35                 {
36                     while (true)
37                     {
38                         if (_unaryNumberPowerOf2Indicies.TryGetValue(source, out int powerOf2Index))
39                         {
40                             SetBit(ref target, powerOf2Index);
41                             break;
42                         }
43                         else
44                         {
45                             powerOf2Index = _unaryNumberPowerOf2Indicies[Links.GetSource(source)];
46                             SetBit(ref target, powerOf2Index);
47                             source = Links.GetTarget(source);
48                         }
49                     }
50                 }
51                 return target;
52             }
53
54             [MethodImpl(MethodImplOptions.AggressiveInlining)]
55             private static void SetBit(ref TLink target, int powerOf2Index) => target =
56                 ⇨ Bit.Or(target, Bit.ShiftLeft(Integer<TLink>.One, powerOf2Index));
57         }
58     }
59 }
```

./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         }
```



```

17     var objectProperty = Links.SearchOrDefault(@object, property);
18     if (_equalityComparer.Equals(objectProperty, default))
19     {
20         return default;
21     }
22     var valueLink = Links.All(Links.Constants.Any, objectProperty).SingleOrDefault();
23     if (valueLink == null)
24     {
25         return default;
26     }
27     return Links.GetTarget(valueLink[Links.Constants.IndexPart]);
28 }
29
30 public void SetValue(TLink @object, TLink property, TLink value)
31 {
32     var objectProperty = Links.GetOrCreate(@object, property);
33     Links.DeleteMany(Links.AllIndices(Links.Constants.Any, objectProperty));
34     Links.GetOrCreate(objectProperty, value);
35 }
36 }
37 }

```

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

```

1  using System.Collections.Generic;
2  using Platform.Interfaces;
3
4  #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 countinueConstant = constants.Continue;
41             var breakConstant = constants.Break;
42             var anyConstant = constants.Any;
43             var query = new Link<TLink>(anyConstant, property, anyConstant);
44             Links.Each(candidate =>
45             {
46                 var candidateTarget = Links.GetTarget(candidate);
47                 var valueTarget = Links.GetTarget(candidateTarget);
48                 if (_equalityComparer.Equals(valueTarget, _propertyValueMarker))
49                 {
50                     valueContainer = Links.GetIndex(candidate);
51                     return breakConstant;
52                 }
53                 return countinueConstant;
54             }, query);
55             return valueContainer;
56         }
57     }
58 }

```

```

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 static Platform.Numbers.Arithmetic;
13 using static System.Runtime.CompilerServices.Unsafe;
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 unsafe 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 long LinkSizeInBytes = Structure<Link>.Size;
35
36         public static readonly long LinkHeaderSizeInBytes = Structure<LinksHeader>.Size;
37
38         public static readonly long DefaultLinksSizeStep = LinkSizeInBytes * 1024 * 1024;
39
40         private struct Link
41         {
42             public static readonly long SourceOffset = Marshal.OffsetOf(typeof(Link),
43                 ↪ nameof(Source)).ToInt32();
44             public static readonly long TargetOffset = Marshal.OffsetOf(typeof(Link),
45                 ↪ nameof(Target)).ToInt32();
46             public static readonly long LeftAsSourceOffset = Marshal.OffsetOf(typeof(Link),
47                 ↪ nameof(LeftAsSource)).ToInt32();
48             public static readonly long RightAsSourceOffset = Marshal.OffsetOf(typeof(Link),
49                 ↪ nameof(RightAsSource)).ToInt32();
50             public static readonly long SizeAsSourceOffset = Marshal.OffsetOf(typeof(Link),
51                 ↪ nameof(SizeAsSource)).ToInt32();
52             public static readonly long LeftAsTargetOffset = Marshal.OffsetOf(typeof(Link),
53                 ↪ nameof(LeftAsTarget)).ToInt32();
54             public static readonly long RightAsTargetOffset = Marshal.OffsetOf(typeof(Link),
55                 ↪ nameof(RightAsTarget)).ToInt32();
56             public static readonly long SizeAsTargetOffset = Marshal.OffsetOf(typeof(Link),
57                 ↪ nameof(SizeAsTarget)).ToInt32();
58
59             public TLink Source;
60             public TLink Target;
61         }
62     }
63 }

```

```

52     public TLink LeftAsSource;
53     public TLink RightAsSource;
54     public TLink SizeAsSource;
55     public TLink LeftAsTarget;
56     public TLink RightAsTarget;
57     public TLink SizeAsTarget;
58 }
59
60 private struct LinksHeader
61 {
62     public static readonly int AllocatedLinksOffset =
63         ↪ Marshal.OffsetOf(typeof(LinksHeader), nameof(AllocatedLinks)).ToInt32();
64     public static readonly int ReservedLinksOffset =
65         ↪ Marshal.OffsetOf(typeof(LinksHeader), nameof(ReservedLinks)).ToInt32();
66     public static readonly int FreeLinksOffset = Marshal.OffsetOf(typeof(LinksHeader),
67         ↪ nameof(FreeLinks)).ToInt32();
68     public static readonly int FirstFreeLinkOffset =
69         ↪ Marshal.OffsetOf(typeof(LinksHeader), nameof(FirstFreeLink)).ToInt32();
70     public static readonly int FirstAsSourceOffset =
71         ↪ Marshal.OffsetOf(typeof(LinksHeader), nameof(FirstAsSource)).ToInt32();
72     public static readonly int FirstAsTargetOffset =
73         ↪ Marshal.OffsetOf(typeof(LinksHeader), nameof(FirstAsTarget)).ToInt32();
74     public static readonly int LastFreeLinkOffset =
75         ↪ Marshal.OffsetOf(typeof(LinksHeader), nameof(LastFreeLink)).ToInt32();
76
77     public TLink AllocatedLinks;
78     public TLink ReservedLinks;
79     public TLink FreeLinks;
80     public TLink FirstFreeLink;
81     public TLink FirstAsSource;
82     public TLink FirstAsTarget;
83     public TLink LastFreeLink;
84     public TLink Reserved8;
85 }
86
87 private readonly long _memoryReservationStep;
88
89 private readonly IResizableDirectMemory _memory;
90 private byte* _header;
91 private byte* _links;
92
93 private LinksTargetsTreeMethods _targetsTreeMethods;
94 private LinksSourcesTreeMethods _sourcesTreeMethods;
95
96 // TODO: Возможно чтобы гарантированно проверять на то, является ли связь удалённой,
97 ↪ нужно использовать не список а дерево, так как так можно быстрее проверить на
98 ↪ наличие связи внутри
99 private UnusedLinksListMethods _unusedLinksListMethods;
100
101 /// <summary>
102 /// Возвращает общее число связей находящихся в хранилище.
103 /// </summary>
104 private TLink Total => Subtract(AsRef<LinksHeader>(_header).AllocatedLinks,
105     ↪ AsRef<LinksHeader>(_header).FreeLinks);
106
107 public LinksConstants<TLink> Constants { get; }
108
109 public ResizableDirectMemoryLinks(string address)
110     : this(address, DefaultLinksSizeStep)
111 {
112 }
113
114 /// <summary>
115 /// Создаёт экземпляр базы данных Links в файле по указанному адресу, с указанным
116 ↪ минимальным шагом расширения базы данных.
117 /// </summary>
118 /// <param name="address">Полный путь к файлу базы данных.</param>
119 /// <param name="memoryReservationStep">Минимальный шаг расширения базы данных в
120 ↪ байтах.</param>
121 public ResizableDirectMemoryLinks(string address, long memoryReservationStep)
122     : this(new FileMappedResizableDirectMemory(address, memoryReservationStep),
123         ↪ memoryReservationStep)
124 {
125 }
126
127 public ResizableDirectMemoryLinks(IResizableDirectMemory memory)
128     : this(memory, DefaultLinksSizeStep)
129 {
130 }

```

```

119 public ResizableDirectMemoryLinks(IResizableDirectMemory memory, long
    ↳ memoryReservationStep)
120 {
121     Constants = Default<LinksConstants<TLink>>.Instance;
122     _memory = memory;
123     memoryReservationStep = memoryReservationStep;
124     if (memory.ReservedCapacity < memoryReservationStep)
125     {
126         memory.ReservedCapacity = memoryReservationStep;
127     }
128     SetPointers(_memory);
129     ref var header = ref AsRef<LinksHeader>(_header);
130     // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
131     _memory.UsedCapacity = ((Integer<TLink>)header.AllocatedLinks * LinkSizeInBytes) +
    ↳ LinkHeaderSizeInBytes;
132     // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
133     header.ReservedLinks = (Integer<TLink>)((_memory.ReservedCapacity -
    ↳ LinkHeaderSizeInBytes) / LinkSizeInBytes);
134 }
135
136 [MethodImpl(MethodImplOptions.AggressiveInlining)]
137 public TLink Count(IList<TLink> restrictions)
138 {
139     // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
140     if (restrictions.Count == 0)
141     {
142         return Total;
143     }
144     if (restrictions.Count == 1)
145     {
146         var index = restrictions[Constants.IndexPart];
147         if (_equalityComparer.Equals(index, Constants.Any))
148         {
149             return Total;
150         }
151         return Exists(index) ? Integer<TLink>.One : Integer<TLink>.Zero;
152     }
153     if (restrictions.Count == 2)
154     {
155         var index = restrictions[Constants.IndexPart];
156         var value = restrictions[1];
157         if (_equalityComparer.Equals(index, Constants.Any))
158         {
159             if (_equalityComparer.Equals(value, Constants.Any))
160             {
161                 return Total; // Any - как отсутствие ограничения
162             }
163             return Add(_sourcesTreeMethods.CountUsages(value),
    ↳ _targetsTreeMethods.CountUsages(value));
164         }
165         else
166         {
167             if (!Exists(index))
168             {
169                 return Integer<TLink>.Zero;
170             }
171             if (_equalityComparer.Equals(value, Constants.Any))
172             {
173                 return Integer<TLink>.One;
174             }
175             ref var storedLinkValue = ref GetLinkUnsafe(index);
176             if (_equalityComparer.Equals(storedLinkValue.Source, value) ||
177                 _equalityComparer.Equals(storedLinkValue.Target, value))
178             {
179                 return Integer<TLink>.One;
180             }
181             return Integer<TLink>.Zero;
182         }
183     }
184     if (restrictions.Count == 3)
185     {
186         var index = restrictions[Constants.IndexPart];
187         var source = restrictions[Constants.SourcePart];
188         var target = restrictions[Constants.TargetPart];
189
190         if (_equalityComparer.Equals(index, Constants.Any))
191         {
192             if (_equalityComparer.Equals(source, Constants.Any) &&
    ↳ _equalityComparer.Equals(target, Constants.Any))

```

```

193     {
194         return Total;
195     }
196     else if (_equalityComparer.Equals(source, Constants.Any))
197     {
198         return _targetsTreeMethods.CountUsages(target);
199     }
200     else if (_equalityComparer.Equals(target, Constants.Any))
201     {
202         return _sourcesTreeMethods.CountUsages(source);
203     }
204     else //if(source != Any && target != Any)
205     {
206         // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
207         var link = _sourcesTreeMethods.Search(source, target);
208         return _equalityComparer.Equals(link, Constants.Null) ?
209             ⇨ Integer<TLink>.Zero : Integer<TLink>.One;
210     }
211     else
212     {
213         if (!Exists(index))
214         {
215             return Integer<TLink>.Zero;
216         }
217         if (_equalityComparer.Equals(source, Constants.Any) &&
218             ⇨ _equalityComparer.Equals(target, Constants.Any))
219         {
220             return Integer<TLink>.One;
221         }
222         ref var storedLinkValue = ref GetLinkUnsafe(index);
223         if (!_equalityComparer.Equals(source, Constants.Any) &&
224             ⇨ !_equalityComparer.Equals(target, Constants.Any))
225         {
226             if (_equalityComparer.Equals(storedLinkValue.Source, source) &&
227                 _equalityComparer.Equals(storedLinkValue.Target, target))
228             {
229                 return Integer<TLink>.One;
230             }
231             return Integer<TLink>.Zero;
232         }
233         var value = default(TLink);
234         if (_equalityComparer.Equals(source, Constants.Any))
235         {
236             value = target;
237         }
238         if (_equalityComparer.Equals(target, Constants.Any))
239         {
240             value = source;
241         }
242         if (_equalityComparer.Equals(storedLinkValue.Source, value) ||
243             _equalityComparer.Equals(storedLinkValue.Target, value))
244         {
245             return Integer<TLink>.One;
246         }
247         return Integer<TLink>.Zero;
248     }
249 }
250
251 throw new NotSupportedException("Другие размеры и способы ограничений не
252 ⇨ поддерживаются.");
253
254 [MethodImpl(MethodImplOptions.AggressiveInlining)]
255 public TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
256 {
257     if (restrictions.Count == 0)
258     {
259         for (TLink link = Integer<TLink>.One; _comparer.Compare(link,
260             ⇨ (Integer<TLink>)AsRef<LinksHeader>(_header).AllocatedLinks) <= 0; link =
261             ⇨ Increment(link))
262         {
263             if (Exists(link) && _equalityComparer.Equals(handler(GetLinkStruct(link)),
264                 ⇨ Constants.Break))
265             {
266                 return Constants.Break;
267             }
268         }
269     }
270     return Constants.Continue;
271 }

```

```

264 }
265 if (restrictions.Count == 1)
266 {
267     var index = restrictions[Constants.IndexPart];
268     if (_equalityComparer.Equals(index, Constants.Any))
269     {
270         return Each(handler, ArrayPool<TLink>.Empty);
271     }
272     if (!Exists(index))
273     {
274         return Constants.Continue;
275     }
276     return handler(GetLinkStruct(index));
277 }
278 if (restrictions.Count == 2)
279 {
280     var index = restrictions[Constants.IndexPart];
281     var value = restrictions[1];
282     if (_equalityComparer.Equals(index, Constants.Any))
283     {
284         if (_equalityComparer.Equals(value, Constants.Any))
285         {
286             return Each(handler, ArrayPool<TLink>.Empty);
287         }
288         if (_equalityComparer.Equals(Each(handler, new[] { index, value,
289             ↪ Constants.Any }), Constants.Break))
290         {
291             return Constants.Break;
292         }
293         return Each(handler, new[] { index, Constants.Any, value });
294     }
295     else
296     {
297         if (!Exists(index))
298         {
299             return Constants.Continue;
300         }
301         if (_equalityComparer.Equals(value, Constants.Any))
302         {
303             return handler(GetLinkStruct(index));
304         }
305         ref var storedLinkValue = ref GetLinkUnsafe(index);
306         if (_equalityComparer.Equals(storedLinkValue.Source, value) ||
307             ↪ _equalityComparer.Equals(storedLinkValue.Target, value))
308         {
309             return handler(GetLinkStruct(index));
310         }
311         return Constants.Continue;
312     }
313 }
314 if (restrictions.Count == 3)
315 {
316     var index = restrictions[Constants.IndexPart];
317     var source = restrictions[Constants.SourcePart];
318     var target = restrictions[Constants.TargetPart];
319     if (_equalityComparer.Equals(index, Constants.Any))
320     {
321         if (_equalityComparer.Equals(source, Constants.Any) &&
322             ↪ _equalityComparer.Equals(target, Constants.Any))
323         {
324             return Each(handler, ArrayPool<TLink>.Empty);
325         }
326         else if (_equalityComparer.Equals(source, Constants.Any))
327         {
328             return _targetsTreeMethods.EachUsage(target, handler);
329         }
330         else if (_equalityComparer.Equals(target, Constants.Any))
331         {
332             return _sourcesTreeMethods.EachUsage(source, handler);
333         }
334         else //if(source != Any && target != Any)
335         {
336             var link = _sourcesTreeMethods.Search(source, target);
337             return _equalityComparer.Equals(link, Constants.Null) ?
338                 ↪ Constants.Continue : handler(GetLinkStruct(link));
339         }
340     }
341     else

```

```

339 {
340     if (!Exists(index))
341     {
342         return Constants.Continue;
343     }
344     if (_equalityComparer.Equals(source, Constants.Any) &&
        ↪ _equalityComparer.Equals(target, Constants.Any))
345     {
346         return handler(GetLinkStruct(index));
347     }
348     ref var storedLinkValue = ref GetLinkUnsafe(index);
349     if (!_equalityComparer.Equals(source, Constants.Any) &&
        ↪ !_equalityComparer.Equals(target, Constants.Any))
350     {
351         if (_equalityComparer.Equals(storedLinkValue.Source, source) &&
352             _equalityComparer.Equals(storedLinkValue.Target, target))
353         {
354             return handler(GetLinkStruct(index));
355         }
356         return Constants.Continue;
357     }
358     var value = default(TLink);
359     if (_equalityComparer.Equals(source, Constants.Any))
360     {
361         value = target;
362     }
363     if (_equalityComparer.Equals(target, Constants.Any))
364     {
365         value = source;
366     }
367     if (_equalityComparer.Equals(storedLinkValue.Source, value) ||
368         _equalityComparer.Equals(storedLinkValue.Target, value))
369     {
370         return handler(GetLinkStruct(index));
371     }
372     return Constants.Continue;
373 }
374 }
375 throw new NotSupportedException("Другие размеры и способы ограничений не
        ↪ поддерживаются.");
376 }
377
378 /// <remarks>
379 /// TODO: Возможно можно перемещать значения, если указан индекс, но значение существует
        ↪ в другом месте (но не в менеджере памяти, а в логике Links)
380 /// </remarks>
381 [MethodImpl(MethodImplOptions.AggressiveInlining)]
382 public TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
383 {
384     var linkIndex = restrictions[Constants.IndexPart];
385     ref var link = ref GetLinkUnsafe(linkIndex);
386     ref var firstAsSource = ref AsRef<LinksHeader>(_header).FirstAsSource;
387     ref var firstAsTarget = ref AsRef<LinksHeader>(_header).FirstAsTarget;
388     // Будет корректно работать только в том случае, если пространство выделенной связи
        ↪ предварительно заполнено нулями
389     if (!_equalityComparer.Equals(link.Source, Constants.Null))
390     {
391         _sourcesTreeMethods.Detach(ref firstAsSource, linkIndex);
392     }
393     if (!_equalityComparer.Equals(link.Target, Constants.Null))
394     {
395         _targetsTreeMethods.Detach(ref firstAsTarget, linkIndex);
396     }
397     link.Source = substitution[Constants.SourcePart];
398     link.Target = substitution[Constants.TargetPart];
399     if (!_equalityComparer.Equals(link.Source, Constants.Null))
400     {
401         _sourcesTreeMethods.Attach(ref firstAsSource, linkIndex);
402     }
403     if (!_equalityComparer.Equals(link.Target, Constants.Null))
404     {
405         _targetsTreeMethods.Attach(ref firstAsTarget, linkIndex);
406     }
407     return linkIndex;
408 }
409
410 [MethodImpl(MethodImplOptions.AggressiveInlining)]
411 public Link<TLink> GetLinkStruct(TLink linkIndex)

```

```

412 {
413     ref var link = ref GetLinkUnsafe(linkIndex);
414     return new Link<TLink>(linkIndex, link.Source, link.Target);
415 }
416
417 [MethodImpl(MethodImplOptions.AggressiveInlining)]
418 private ref Link GetLinkUnsafe(TLink linkIndex) => ref AsRef<Link>(_links +
419     ↳ LinkSizeInBytes * (Integer<TLink>)linkIndex);
420
421 /// <remarks>
422 /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
423     ↳ пространство
424 /// </remarks>
425 public TLink Create(ICollection<TLink> restrictions)
426 {
427     ref var header = ref AsRef<LinksHeader>(_header);
428     var freeLink = header.FirstFreeLink;
429     if (!_equalityComparer.Equals(freeLink, Constants.Null))
430     {
431         _unusedLinksListMethods.Detach(freeLink);
432     }
433     else
434     {
435         var maximumPossibleInnerReference =
436             ↳ Constants.PossibleInnerReferencesRange.Maximum;
437         if (_comparer.Compare(header.AllocatedLinks, maximumPossibleInnerReference) > 0)
438         {
439             throw new LinksLimitReachedException<TLink>(maximumPossibleInnerReference);
440         }
441         if (_comparer.Compare(header.AllocatedLinks, Decrement(header.ReservedLinks)) >=
442             ↳ 0)
443         {
444             _memory.ReservedCapacity += _memory.ReservationStep;
445             SetPointers(_memory);
446             header.ReservedLinks = (Integer<TLink>)(_memory.ReservedCapacity /
447                 ↳ LinkSizeInBytes);
448         }
449         header.AllocatedLinks = Increment(header.AllocatedLinks);
450         _memory.UsedCapacity += LinkSizeInBytes;
451         freeLink = header.AllocatedLinks;
452     }
453     return freeLink;
454 }
455
456 public void Delete(ICollection<TLink> restrictions)
457 {
458     ref var header = ref AsRef<LinksHeader>(_header);
459     var link = restrictions[Constants.IndexPart];
460     if (_comparer.Compare(link, header.AllocatedLinks) < 0)
461     {
462         _unusedLinksListMethods.AttachAsFirst(link);
463     }
464     else if (_equalityComparer.Equals(link, header.AllocatedLinks))
465     {
466         header.AllocatedLinks = Decrement(header.AllocatedLinks);
467         _memory.UsedCapacity -= LinkSizeInBytes;
468         // Убираем все связи, находящиеся в списке свободных в конце файла, до тех пор,
469         // ↳ пока не дойдём до первой существующей связи
470         // Позволяет оптимизировать количество выделенных связей (AllocatedLinks)
471         while ((_comparer.Compare(header.AllocatedLinks, Integer<TLink>.Zero) > 0) &&
472             ↳ IsUnusedLink(header.AllocatedLinks))
473         {
474             _unusedLinksListMethods.Detach(header.AllocatedLinks);
475             header.AllocatedLinks = Decrement(header.AllocatedLinks);
476             _memory.UsedCapacity -= LinkSizeInBytes;
477         }
478     }
479 }
480
481 /// <remarks>
482 /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том случае, если
483     ↳ адрес реально поменялся
484 ///
485 /// Указатель this.links может быть в том же месте,
486 /// так как 0-я связь не используется и имеет такой же размер как Header,
487 /// поэтому header размещается в том же месте, что и 0-я связь
488 /// </remarks>
489 private void SetPointers(IDirectMemory memory)

```



```

482 {
483     if (memory == null)
484     {
485         _links = null;
486         _header = _links;
487         _unusedLinksListMethods = null;
488         _targetsTreeMethods = null;
489         _unusedLinksListMethods = null;
490     }
491     else
492     {
493         _links = (byte*)(void*)memory.Pointer;
494         _header = _links;
495         _sourcesTreeMethods = new LinksSourcesTreeMethods(this);
496         _targetsTreeMethods = new LinksTargetsTreeMethods(this);
497         _unusedLinksListMethods = new UnusedLinksListMethods(_links, _header);
498     }
499 }
500
501 [MethodImpl(MethodImplOptions.AggressiveInlining)]
502 private bool Exists(TLink link)
503     => (_comparer.Compare(link, Constants.PossibleInnerReferencesRange.Minimum) >= 0)
504         && (_comparer.Compare(link, AsRef<LinksHeader>(_header).AllocatedLinks) <= 0)
505         && !IsUnusedLink(link);
506
507 [MethodImpl(MethodImplOptions.AggressiveInlining)]
508 private bool IsUnusedLink(TLink link)
509     => _equalityComparer.Equals(AsRef<LinksHeader>(_header).FirstFreeLink, link)
510         || (_equalityComparer.Equals(GetLinkUnsafe(link).SizeAsSource, Constants.Null)
511             && !_equalityComparer.Equals(GetLinkUnsafe(link).Source, Constants.Null));
512
513 #region DisposableBase
514
515 protected override bool AllowMultipleDisposeCalls => true;
516
517 protected override void Dispose(bool manual, bool wasDisposed)
518 {
519     if (!wasDisposed)
520     {
521         SetPointers(null);
522         _memory.DisposeIfPossible();
523     }
524 }
525
526 #endregion
527 }
528 }

```

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

```

1 using Platform.Collections.Methods.Lists;
2 using Platform.Numbers;
3 using System.Runtime.CompilerServices;
4 using static System.Runtime.CompilerServices.Unsafe;
5
6 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8 namespace Platform.Data.Doublets.ResizableDirectMemory
9 {
10     partial class ResizableDirectMemoryLinks<TLink>
11     {
12         private unsafe class UnusedLinksListMethods : CircularDoublyLinkedListMethods<TLink>
13         {
14             private readonly byte* _links;
15             private readonly byte* _header;
16
17             public UnusedLinksListMethods(byte* links, byte* header)
18             {
19                 _links = links;
20                 _header = header;
21             }
22
23             [MethodImpl(MethodImplOptions.AggressiveInlining)]
24             protected override TLink GetFirst() => Read<TLink>(_header +
25                 ↳ LinksHeader.FirstFreeLinkOffset);
26
27             [MethodImpl(MethodImplOptions.AggressiveInlining)]
28             protected override TLink GetLast() => Read<TLink>(_header +
29                 ↳ LinksHeader.LastFreeLinkOffset);
30
31             [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

30     protected override TLink GetPrevious(TLink element) => Read<TLink>(_links +
    ↪ LinkSizeInBytes * (Integer<TLink>)element + Link.SourceOffset);
31
32     [MethodImpl(MethodImplOptions.AggressiveInlining)]
33     protected override TLink GetNext(TLink element) => Read<TLink>(_links +
    ↪ LinkSizeInBytes * (Integer<TLink>)element + Link.TargetOffset);
34
35     [MethodImpl(MethodImplOptions.AggressiveInlining)]
36     protected override TLink GetSize() => Read<TLink>(_header +
    ↪ LinksHeader.FreeLinksOffset);
37
38     [MethodImpl(MethodImplOptions.AggressiveInlining)]
39     protected override void SetFirst(TLink element) => Write(_header +
    ↪ LinksHeader.FirstFreeLinkOffset, element);
40
41     [MethodImpl(MethodImplOptions.AggressiveInlining)]
42     protected override void SetLast(TLink element) => Write(_header +
    ↪ LinksHeader.LastFreeLinkOffset, element);
43
44     [MethodImpl(MethodImplOptions.AggressiveInlining)]
45     protected override void SetPrevious(TLink element, TLink previous) => Write(_links +
    ↪ LinkSizeInBytes * (Integer<TLink>)element + Link.SourceOffset, previous);
46
47     [MethodImpl(MethodImplOptions.AggressiveInlining)]
48     protected override void SetNext(TLink element, TLink next) => Write(_links +
    ↪ LinkSizeInBytes * (Integer<TLink>)element + Link.TargetOffset, next);
49
50     [MethodImpl(MethodImplOptions.AggressiveInlining)]
51     protected override void SetSize(TLink size) => Write(_header +
    ↪ LinksHeader.FreeLinksOffset, size);
52 }
53 }
54 }

```

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

```

1  using System;
2  using System.Text;
3  using System.Collections.Generic;
4  using System.Runtime.CompilerServices;
5  using Platform.Numbers;
6  using Platform.Collections.Methods.Trees;
7  using static System.Runtime.CompilerServices.Unsafe;
8
9  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11 namespace Platform.Data.Doublets.ResizableDirectMemory
12 {
13     unsafe partial class ResizableDirectMemoryLinks<TLink>
14     {
15         private abstract class LinksTreeMethodsBase :
16             ↪ SizedAndThreadedAVLBalancedTreeMethods<TLink>
17         {
18             //private static readonly EqualityComparer<TLink> _equalityComparer =
19             ↪ EqualityComparer<TLink>.Default;
20
21             private readonly ResizableDirectMemoryLinks<TLink> _memory;
22             private readonly LinksConstants<TLink> _constants;
23             protected readonly byte* Links;
24             protected readonly byte* Header;
25
26             protected LinksTreeMethodsBase(ResizableDirectMemoryLinks<TLink> memory)
27             {
28                 Links = memory._links;
29                 Header = memory._header;
30                 _memory = memory;
31                 _constants = memory.Constants;
32             }
33
34             [MethodImpl(MethodImplOptions.AggressiveInlining)]
35             protected abstract TLink GetTreeRoot();
36
37             [MethodImpl(MethodImplOptions.AggressiveInlining)]
38             protected abstract TLink GetBasePartValue(TLink link);
39
40             public TLink this[TLink index]
41             {
42                 get
43                 {
44                     var root = GetTreeRoot();
45                     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
        ↳ structure broken)
64 }
65 }
66
67 // TODO: Return indices range instead of references count
68 public TLink CountUsages(TLink link)
69 {
70     var root = GetTreeRoot();
71     var total = GetSize(root);
72     var totalRightIgnore = GetZero();
73     while (!EqualToZero(root))
74     {
75         var @base = GetBasePartValue(root);
76         if (LessOrEqualThan(@base, link))
77         {
78             root = GetRightOrDefault(root);
79         }
80         else
81         {
82             totalRightIgnore = Add(totalRightIgnore, Increment(GetRightSize(root)));
83             root = GetLeftOrDefault(root);
84         }
85     }
86     root = GetTreeRoot();
87     var totalLeftIgnore = GetZero();
88     while (!EqualToZero(root))
89     {
90         var @base = GetBasePartValue(root);
91         if (GreaterOrEqualThan(@base, link))
92         {
93             root = GetLeftOrDefault(root);
94         }
95         else
96         {
97             totalLeftIgnore = Add(totalLeftIgnore, Increment(GetLeftSize(root)));
98             root = GetRightOrDefault(root);
99         }
100     }
101     return Subtract(Subtract(total, totalRightIgnore), totalLeftIgnore);
102 }
103
104 public TLink EachUsage(TLink link, Func<IList<TLink>, TLink> handler)
105 {
106     var root = GetTreeRoot();
107     if (EqualToZero(root))
108     {
109         return _constants.Continue;
110     }
111     TLink first = GetZero(), current = root;
112     while (!EqualToZero(current))
113     {
114         var @base = GetBasePartValue(current);
115         if (GreaterOrEqualThan(@base, link))
116         {
117             if (IsEquals(@base, link))
118             {
119                 first = current;
120             }

```

```

121         }
122         current = GetLeftOrDefault(current);
123     }
124     else
125     {
126         current = GetRightOrDefault(current);
127     }
128 }
129 if (!EqualToZero(first))
130 {
131     current = first;
132     while (true)
133     {
134         if (IsEquals(handler(_memory.GetLinkStruct(current)), _constants.Break))
135         {
136             return _constants.Break;
137         }
138         current = GetNext(current);
139         if (EqualToZero(current) || !IsEquals(GetBasePartValue(current), link))
140         {
141             break;
142         }
143     }
144 }
145 return _constants.Continue;
146 }
147
148 protected override void PrintNodeValue(TLink node, StringBuilder sb)
149 {
150     sb.Append(' ');
151     sb.Append(Read<TLink>(Links + LinkSizeInBytes * (Integer<TLink>)node +
152         ↳ Link.SourceOffset));
153     sb.Append('-');
154     sb.Append('>');
155     sb.Append(Read<TLink>(Links + LinkSizeInBytes * (Integer<TLink>)node +
156         ↳ Link.TargetOffset));
157 }
158
159 private class LinksSourcesTreeMethods : LinksTreeMethodsBase
160 {
161     public LinksSourcesTreeMethods(ResizableDirectMemoryLinks<TLink> memory)
162         : base(memory)
163     {
164     }
165
166     protected unsafe override ref TLink GetLeftReference(TLink node) => ref
167         ↳ AsRef<TLink>((void*)(Links + LinkSizeInBytes * (Integer<TLink>)node +
168         ↳ Link.LeftAsSourceOffset));
169
170     protected unsafe override ref TLink GetRightReference(TLink node) => ref
171         ↳ AsRef<TLink>((void*)(Links + LinkSizeInBytes * (Integer<TLink>)node +
172         ↳ Link.RightAsSourceOffset));
173
174     protected override TLink GetLeft(TLink node) => Read<TLink>(Links + LinkSizeInBytes
175         ↳ * (Integer<TLink>)node + Link.LeftAsSourceOffset);
176
177     protected override TLink GetRight(TLink node) => Read<TLink>(Links + LinkSizeInBytes
178         ↳ * (Integer<TLink>)node + Link.RightAsSourceOffset);
179
180     protected override TLink GetSize(TLink node)
181     {
182         var previousValue = Read<TLink>(Links + LinkSizeInBytes * (Integer<TLink>)node +
183             ↳ Link.SizeAsSourceOffset);
184         return Bit<TLink>.PartialRead(previousValue, 5, -5);
185     }
186
187     protected override void SetLeft(TLink node, TLink left) => Write(Links +
188         ↳ LinkSizeInBytes * (Integer<TLink>)node + Link.LeftAsSourceOffset, left);
189
190     protected override void SetRight(TLink node, TLink right) => Write(Links +
191         ↳ LinkSizeInBytes * (Integer<TLink>)node + Link.RightAsSourceOffset, right);
192
193     protected override void SetSize(TLink node, TLink size)
194     {
195         var previousValue = Read<TLink>(Links + LinkSizeInBytes * (Integer<TLink>)node +
196             ↳ Link.SizeAsSourceOffset);

```

```

186         Write(Links + LinkSizeInBytes * (Integer<TLink>)node + Link.SizeAsSourceOffset,
187             ↪ Bit<TLink>.PartialWrite(previousValue, size, 5, -5));
188     }
189     protected override bool GetLeftIsChild(TLink node)
190     {
191         var previousValue = Read<TLink>(Links + LinkSizeInBytes * (Integer<TLink>)node +
192             ↪ Link.SizeAsSourceOffset);
193         //return (Integer<TLink>)Bit<TLink>.PartialRead(previousValue, 4, 1);
194         return !_equalityComparer.Equals(Bit<TLink>.PartialRead(previousValue, 4, 1),
195             ↪ default);
196     }
197     protected override void SetLeftIsChild(TLink node, bool value)
198     {
199         var previousValue = Read<TLink>(Links + LinkSizeInBytes * (Integer<TLink>)node +
200             ↪ Link.SizeAsSourceOffset);
201         var modified = Bit<TLink>.PartialWrite(previousValue, (Integer<TLink>)value, 4,
202             ↪ 1);
203         Write(Links + LinkSizeInBytes * (Integer<TLink>)node + Link.SizeAsSourceOffset,
204             ↪ modified);
205     }
206     protected override bool GetRightIsChild(TLink node)
207     {
208         var previousValue = Read<TLink>(Links + LinkSizeInBytes * (Integer<TLink>)node +
209             ↪ Link.SizeAsSourceOffset);
210         //return (Integer<TLink>)Bit<TLink>.PartialRead(previousValue, 3, 1);
211         return !_equalityComparer.Equals(Bit<TLink>.PartialRead(previousValue, 3, 1),
212             ↪ default);
213     }
214     protected override void SetRightIsChild(TLink node, bool value)
215     {
216         var previousValue = Read<TLink>(Links + LinkSizeInBytes * (Integer<TLink>)node +
217             ↪ Link.SizeAsSourceOffset);
218         var modified = Bit<TLink>.PartialWrite(previousValue, (Integer<TLink>)value, 3,
219             ↪ 1);
220         Write(Links + LinkSizeInBytes * (Integer<TLink>)node + Link.SizeAsSourceOffset,
221             ↪ modified);
222     }
223     protected override sbyte GetBalance(TLink node)
224     {
225         var previousValue = Read<TLink>(Links + LinkSizeInBytes * (Integer<TLink>)node +
226             ↪ Link.SizeAsSourceOffset);
227         var value = (ulong)(Integer<TLink>)Bit<TLink>.PartialRead(previousValue, 0, 3);
228         var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) << 5) | value & 3 |
229             ↪ 124 : value & 3);
230         return unpackedValue;
231     }
232     protected override void SetBalance(TLink node, sbyte value)
233     {
234         var previousValue = Read<TLink>(Links + LinkSizeInBytes * (Integer<TLink>)node +
235             ↪ Link.SizeAsSourceOffset);
236         var packagedValue = (TLink)(Integer<TLink>)((((byte)value >> 5) & 4) | value &
237             ↪ 3);
238         var modified = Bit<TLink>.PartialWrite(previousValue, packagedValue, 0, 3);
239         Write(Links + LinkSizeInBytes * (Integer<TLink>)node + Link.SizeAsSourceOffset,
240             ↪ modified);
241     }
242     protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
243     {
244         var firstSource = Read<TLink>(Links + LinkSizeInBytes * (Integer<TLink>)first +
245             ↪ Link.SourceOffset);
246         var secondSource = Read<TLink>(Links + LinkSizeInBytes * (Integer<TLink>)second
247             ↪ + Link.SourceOffset);
248         return LessThan(firstSource, secondSource) ||
249             (IsEquals(firstSource, secondSource) && LessThan(Read<TLink>(Links +
250                 ↪ LinkSizeInBytes * (Integer<TLink>)first + Link.TargetOffset),
251                 ↪ Read<TLink>(Links + LinkSizeInBytes * (Integer<TLink>)second +
252                 ↪ Link.TargetOffset)));
253     }

```

```

241     protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
242     {
243         var firstSource = Read<TLink>(Links + LinkSizeInBytes * (Integer<TLink>)first +
244             ↪ Link.SourceOffset);
245         var secondSource = Read<TLink>(Links + LinkSizeInBytes * (Integer<TLink>)second
246             ↪ + Link.SourceOffset);
247         return GreaterThan(firstSource, secondSource) ||
248             (IsEquals(firstSource, secondSource) && GreaterThan(Read<TLink>(Links +
249                 ↪ LinkSizeInBytes * (Integer<TLink>)first + Link.TargetOffset),
250                 ↪ Read<TLink>(Links + LinkSizeInBytes * (Integer<TLink>)second +
251                 ↪ Link.TargetOffset)));
252     }
253
254     protected override TLink GetTreeRoot() => Read<TLink>(Header +
255         ↪ LinksHeader.FirstAsSourceOffset);
256
257     protected override TLink GetBasePartValue(TLink link) => Read<TLink>(Links +
258         ↪ LinkSizeInBytes * (Integer<TLink>)link + Link.SourceOffset);
259
260     /// <summary>
261     /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
262     ↪ (концом)
263     /// по дереву (индексу) связей, отсортированному по Source, а затем по Target.
264     /// </summary>
265     /// <param name="source">Индекс связи, которая является началом на искомой
266     ↪ связи.</param>
267     /// <param name="target">Индекс связи, которая является концом на искомой
268     ↪ связи.</param>
269     /// <returns>Индекс искомой связи.</returns>
270     public TLink Search(TLink source, TLink target)
271     {
272         var root = GetTreeRoot();
273         while (!EqualToZero(root))
274         {
275             var rootSource = Read<TLink>(Links + LinkSizeInBytes * (Integer<TLink>)root
276                 ↪ + Link.SourceOffset);
277             var rootTarget = Read<TLink>(Links + LinkSizeInBytes * (Integer<TLink>)root
278                 ↪ + Link.TargetOffset);
279             if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
280                 ↪ node.Key < root.Key
281             {
282                 root = GetLeftOrDefault(root);
283             }
284             else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget))
285                 ↪ // node.Key > root.Key
286             {
287                 root = GetRightOrDefault(root);
288             }
289             else // node.Key == root.Key
290             {
291                 return root;
292             }
293         }
294         return GetZero();
295     }
296
297     [MethodImpl(MethodImplOptions.AggressiveInlining)]
298     private bool FirstIsToTheLeftOfSecond(TLink firstSource, TLink firstTarget, TLink
299         ↪ secondSource, TLink secondTarget) => LessThan(firstSource, secondSource) ||
300         ↪ (IsEquals(firstSource, secondSource) && LessThan(firstTarget, secondTarget));
301
302     [MethodImpl(MethodImplOptions.AggressiveInlining)]
303     private bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget, TLink
304         ↪ secondSource, TLink secondTarget) => GreaterThan(firstSource, secondSource) ||
305         ↪ (IsEquals(firstSource, secondSource) && GreaterThan(firstTarget, secondTarget));
306 }
307
308 private class LinksTargetsTreeMethods : LinksTreeMethodsBase
309 {
310     public LinksTargetsTreeMethods(ResizableDirectMemoryLinks<TLink> memory)
311         : base(memory)
312     {
313     }
314
315     protected unsafe override ref TLink GetLeftReference(TLink node) => ref
316         ↪ AsRef<TLink>((void*)(Links + LinkSizeInBytes * (Integer<TLink>)node +
317         ↪ Link.LeftAsTargetOffset));

```

```

298 protected unsafe override ref TLink GetRightReference(TLink node) => ref
299     ↳ AsRef<TLink>((void*)(Links + LinkSizeInBytes * (Integer<TLink>)node +
    ↳ Link.RightAsTargetOffset));

300
301 protected override TLink GetLeft(TLink node) => Read<TLink>(Links + LinkSizeInBytes
    ↳ * (Integer<TLink>)node + Link.LeftAsTargetOffset);

302
303 protected override TLink GetRight(TLink node) => Read<TLink>(Links + LinkSizeInBytes
    ↳ * (Integer<TLink>)node + Link.RightAsTargetOffset);

304
305 protected override TLink GetSize(TLink node)
306 {
307     var previousValue = Read<TLink>(Links + LinkSizeInBytes * (Integer<TLink>)node +
    ↳ Link.SizeAsTargetOffset);
308     return Bit<TLink>.PartialRead(previousValue, 5, -5);
309 }
310
311 protected override void SetLeft(TLink node, TLink left) => Write(Links +
    ↳ LinkSizeInBytes * (Integer<TLink>)node + Link.LeftAsTargetOffset, left);

312
313 protected override void SetRight(TLink node, TLink right) => Write(Links +
    ↳ LinkSizeInBytes * (Integer<TLink>)node + Link.RightAsTargetOffset, right);

314
315 protected override void SetSize(TLink node, TLink size)
316 {
317     var previousValue = Read<TLink>(Links + LinkSizeInBytes * (Integer<TLink>)node +
    ↳ Link.SizeAsTargetOffset);
318     Write(Links + LinkSizeInBytes * (Integer<TLink>)node + Link.SizeAsTargetOffset,
    ↳ Bit<TLink>.PartialWrite(previousValue, size, 5, -5));
319 }
320
321 protected override bool GetLeftIsChild(TLink node)
322 {
323     var previousValue = Read<TLink>(Links + LinkSizeInBytes * (Integer<TLink>)node +
    ↳ Link.SizeAsTargetOffset);
324     //return (Integer<TLink>)Bit<TLink>.PartialRead(previousValue, 4, 1);
325     return !_equalityComparer.Equals(Bit<TLink>.PartialRead(previousValue, 4, 1),
    ↳ default);
326 }
327
328 protected override void SetLeftIsChild(TLink node, bool value)
329 {
330     var previousValue = Read<TLink>(Links + LinkSizeInBytes * (Integer<TLink>)node +
    ↳ Link.SizeAsTargetOffset);
331     var modified = Bit<TLink>.PartialWrite(previousValue, (Integer<TLink>)value, 4,
    ↳ 1);
332     Write(Links + LinkSizeInBytes * (Integer<TLink>)node + Link.SizeAsTargetOffset,
    ↳ modified);
333 }
334
335 protected override bool GetRightIsChild(TLink node)
336 {
337     var previousValue = Read<TLink>(Links + LinkSizeInBytes * (Integer<TLink>)node +
    ↳ Link.SizeAsTargetOffset);
338     //return (Integer<TLink>)Bit<TLink>.PartialRead(previousValue, 3, 1);
339     return !_equalityComparer.Equals(Bit<TLink>.PartialRead(previousValue, 3, 1),
    ↳ default);
340 }
341
342 protected override void SetRightIsChild(TLink node, bool value)
343 {
344     var previousValue = Read<TLink>(Links + LinkSizeInBytes * (Integer<TLink>)node +
    ↳ Link.SizeAsTargetOffset);
345     var modified = Bit<TLink>.PartialWrite(previousValue, (Integer<TLink>)value, 3,
    ↳ 1);
346     Write(Links + LinkSizeInBytes * (Integer<TLink>)node + Link.SizeAsTargetOffset,
    ↳ modified);
347 }
348
349 protected override sbyte GetBalance(TLink node)
350 {
351     var previousValue = Read<TLink>(Links + LinkSizeInBytes * (Integer<TLink>)node +
    ↳ Link.SizeAsTargetOffset);
352     var value = (ulong)(Integer<TLink>)Bit<TLink>.PartialRead(previousValue, 0, 3);
353     var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) << 5) | value & 3 |
    ↳ 124 : value & 3);

```

```

354         return unpackedValue;
355     }
356
357     protected override void SetBalance(TLink node, sbyte value)
358     {
359         var previousValue = Read<TLink>(Links + LinkSizeInBytes * (Integer<TLink>)node +
360             ↪ Link.SizeAsTargetOffset);
361         var packagedValue = (TLink)(Integer<TLink>)((((byte)value >> 5) & 4) | value &
362             ↪ 3);
363         var modified = Bit<TLink>.PartialWrite(previousValue, packagedValue, 0, 3);
364         Write(Links + LinkSizeInBytes * (Integer<TLink>)node + Link.SizeAsTargetOffset,
365             ↪ modified);
366     }
367
368     protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
369     {
370         var firstTarget = Read<TLink>(Links + LinkSizeInBytes * (Integer<TLink>)first +
371             ↪ Link.TargetOffset);
372         var secondTarget = Read<TLink>(Links + LinkSizeInBytes * (Integer<TLink>)second
373             ↪ + Link.TargetOffset);
374         return LessThan(firstTarget, secondTarget) ||
375             (IsEquals(firstTarget, secondTarget) && LessThan(Read<TLink>(Links +
376                 ↪ LinkSizeInBytes * (Integer<TLink>)first + Link.SourceOffset),
377                 ↪ Read<TLink>(Links + LinkSizeInBytes * (Integer<TLink>)second +
378                 ↪ Link.SourceOffset)));
379     }
380
381     protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
382     {
383         var firstTarget = Read<TLink>(Links + LinkSizeInBytes * (Integer<TLink>)first +
384             ↪ Link.TargetOffset);
385         var secondTarget = Read<TLink>(Links + LinkSizeInBytes * (Integer<TLink>)second
386             ↪ + Link.TargetOffset);
387         return GreaterThan(firstTarget, secondTarget) ||
388             (IsEquals(firstTarget, secondTarget) && GreaterThan(Read<TLink>(Links +
389                 ↪ LinkSizeInBytes * (Integer<TLink>)first + Link.SourceOffset),
390                 ↪ Read<TLink>(Links + LinkSizeInBytes * (Integer<TLink>)second +
391                 ↪ Link.SourceOffset)));
392     }
393
394     protected override TLink GetTreeRoot() => Read<TLink>(Header +
395         ↪ LinksHeader.FirstAsTargetOffset);
396
397     protected override TLink GetBasePartValue(TLink link) => Read<TLink>(Links +
398         ↪ LinkSizeInBytes * (Integer<TLink>)link + Link.TargetOffset);
399 }
400 }

```

./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
10 #pragma warning disable 0649
11 #pragma warning disable 169
12 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
13
14 // ReSharper disable BuiltInTypeReferenceStyle
15
16 // #define ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
17
18 namespace Platform.Data.Doublets.ResizableDirectMemory
19 {
20     using id = UInt64;
21
22     public unsafe partial class UInt64ResizableDirectMemoryLinks : DisposableBase, ILinks<id>
23     {
24         /// <summary>Возвращает размер одной связи в байтах.</summary>
25         /// <remarks>
26         /// Используется только во вне класса, не рекомендуется использовать внутри.
27         /// Так как во вне не обязательно будет доступен unsafe C#.
28         /// </remarks>

```



```

29 public static readonly int LinkSizeInBytes = sizeof(Link);
30
31 public static readonly long DefaultLinksSizeStep = LinkSizeInBytes * 1024 * 1024;
32
33 private struct Link
34 {
35     public id Source;
36     public id Target;
37     public id LeftAsSource;
38     public id RightAsSource;
39     public id SizeAsSource;
40     public id LeftAsTarget;
41     public id RightAsTarget;
42     public id SizeAsTarget;
43 }
44
45 private struct LinksHeader
46 {
47     public id AllocatedLinks;
48     public id ReservedLinks;
49     public id FreeLinks;
50     public id FirstFreeLink;
51     public id FirstAsSource;
52     public id FirstAsTarget;
53     public id LastFreeLink;
54     public id Reserved8;
55 }
56
57 private readonly long _memoryReservationStep;
58
59 private readonly IResizableDirectMemory _memory;
60 private LinksHeader* _header;
61 private Link* _links;
62
63 private LinksTargetsTreeMethods _targetsTreeMethods;
64 private LinksSourcesTreeMethods _sourcesTreeMethods;
65
66 // TODO: Возможно чтобы гарантированно проверять на то, является ли связь удалённой,
67 ↪ нужно использовать не список а дерево, так как так можно быстрее проверить на
68 ↪ наличие связи внутри
69 private UnusedLinksListMethods _unusedLinksListMethods;
70
71 /// <summary>
72 /// Возвращает общее число связей находящихся в хранилище.
73 /// </summary>
74 private id Total => _header->AllocatedLinks - _header->FreeLinks;
75
76 // TODO: Дать возможность переопределять в конструкторе
77 public LinksConstants<id> Constants { get; }
78
79 public UInt64ResizableDirectMemoryLinks(string address) : this(address,
80 ↪ DefaultLinksSizeStep) { }
81
82 /// <summary>
83 /// Создаёт экземпляр базы данных Links в файле по указанному адресу, с указанным
84 ↪ минимальным шагом расширения базы данных.
85 /// </summary>
86 /// <param name="address">Полный путь к файлу базы данных.</param>
87 /// <param name="memoryReservationStep">Минимальный шаг расширения базы данных в
88 ↪ байтах.</param>
89 public UInt64ResizableDirectMemoryLinks(string address, long memoryReservationStep) :
90 ↪ this(new FileMappedResizableDirectMemory(address, memoryReservationStep),
91 ↪ memoryReservationStep) { }
92
93 public UInt64ResizableDirectMemoryLinks(IResizableDirectMemory memory) : this(memory,
94 ↪ DefaultLinksSizeStep) { }
95
96 public UInt64ResizableDirectMemoryLinks(IResizableDirectMemory memory, long
97 ↪ memoryReservationStep)
98 {
99     Constants = Default<LinksConstants<id>>.Instance;
100     _memory = memory;
101     _memoryReservationStep = memoryReservationStep;
102     if (memory.ReservedCapacity < memoryReservationStep)
103     {
104         memory.ReservedCapacity = memoryReservationStep;
105     }
106     SetPointers(_memory);
107     // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
108     _memory.UsedCapacity = ((long)_header->AllocatedLinks * sizeof(Link)) +
109     ↪ sizeof(LinksHeader);

```

```

100 // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
101 _header->ReservedLinks = (id)((_memory.ReservedCapacity - sizeof(LinksHeader)) /
    ↳ sizeof(Link));
102 }
103
104 [MethodImpl(MethodImplOptions.AggressiveInlining)]
105 public id Count(IList<id> restrictions)
106 {
107     // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
108     if (restrictions.Count == 0)
109     {
110         return Total;
111     }
112     if (restrictions.Count == 1)
113     {
114         var index = restrictions[Constants.IndexPart];
115         if (index == Constants.Any)
116         {
117             return Total;
118         }
119         return Exists(index) ? 1UL : 0UL;
120     }
121     if (restrictions.Count == 2)
122     {
123         var index = restrictions[Constants.IndexPart];
124         var value = restrictions[1];
125         if (index == Constants.Any)
126         {
127             if (value == Constants.Any)
128             {
129                 return Total; // Any - как отсутствие ограничения
130             }
131             return _sourcesTreeMethods.CountUsages(value)
132                 + _targetsTreeMethods.CountUsages(value);
133         }
134         else
135         {
136             if (!Exists(index))
137             {
138                 return 0;
139             }
140             if (value == Constants.Any)
141             {
142                 return 1;
143             }
144             var storedLinkValue = GetLinkUnsafe(index);
145             if (storedLinkValue->Source == value ||
146                 storedLinkValue->Target == value)
147             {
148                 return 1;
149             }
150             return 0;
151         }
152     }
153     if (restrictions.Count == 3)
154     {
155         var index = restrictions[Constants.IndexPart];
156         var source = restrictions[Constants.SourcePart];
157         var target = restrictions[Constants.TargetPart];
158         if (index == Constants.Any)
159         {
160             if (source == Constants.Any && target == Constants.Any)
161             {
162                 return Total;
163             }
164             else if (source == Constants.Any)
165             {
166                 return _targetsTreeMethods.CountUsages(target);
167             }
168             else if (target == Constants.Any)
169             {
170                 return _sourcesTreeMethods.CountUsages(source);
171             }
172             else //if(source != Any && target != Any)
173             {
174                 // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
175                 var link = _sourcesTreeMethods.Search(source, target);
176                 return link == Constants.Null ? 0UL : 1UL;

```

```

177     }
178 }
179 else
180 {
181     if (!Exists(index))
182     {
183         return 0;
184     }
185     if (source == Constants.Any && target == Constants.Any)
186     {
187         return 1;
188     }
189     var storedLinkValue = GetLinkUnsafe(index);
190     if (source != Constants.Any && target != Constants.Any)
191     {
192         if (storedLinkValue->Source == source &&
193             storedLinkValue->Target == target)
194         {
195             return 1;
196         }
197         return 0;
198     }
199     var value = default(id);
200     if (source == Constants.Any)
201     {
202         value = target;
203     }
204     if (target == Constants.Any)
205     {
206         value = source;
207     }
208     if (storedLinkValue->Source == value ||
209         storedLinkValue->Target == value)
210     {
211         return 1;
212     }
213     return 0;
214 }
215 }
216 throw new NotSupportedException("Другие размеры и способы ограничений не
    ↳ поддерживаются.");
217 }
218
219 [MethodImpl(MethodImplOptions.AggressiveInlining)]
220 public id Each(Func<IList<id>, id> handler, IList<id> restrictions)
221 {
222     if (restrictions.Count == 0)
223     {
224         for (id link = 1; link <= _header->AllocatedLinks; link++)
225         {
226             if (Exists(link))
227             {
228                 if (handler(GetLinkStruct(link)) == Constants.Break)
229                 {
230                     return Constants.Break;
231                 }
232             }
233         }
234         return Constants.Continue;
235     }
236     if (restrictions.Count == 1)
237     {
238         var index = restrictions[Constants.IndexPart];
239         if (index == Constants.Any)
240         {
241             return Each(handler, ArrayPool<ulong>.Empty);
242         }
243         if (!Exists(index))
244         {
245             return Constants.Continue;
246         }
247         return handler(GetLinkStruct(index));
248     }
249     if (restrictions.Count == 2)
250     {
251         var index = restrictions[Constants.IndexPart];
252         var value = restrictions[1];
253         if (index == Constants.Any)
254         {

```

```

255     if (value == Constants.Any)
256     {
257         return Each(handler, ArrayPool<ulong>.Empty);
258     }
259     if (Each(handler, new[] { index, value, Constants.Any }) == Constants.Break)
260     {
261         return Constants.Break;
262     }
263     return Each(handler, new[] { index, Constants.Any, value });
264 }
265 else
266 {
267     if (!Exists(index))
268     {
269         return Constants.Continue;
270     }
271     if (value == Constants.Any)
272     {
273         return handler(GetLinkStruct(index));
274     }
275     var storedLinkValue = GetLinkUnsafe(index);
276     if (storedLinkValue->Source == value ||
277         storedLinkValue->Target == value)
278     {
279         return handler(GetLinkStruct(index));
280     }
281     return Constants.Continue;
282 }
283 }
284 if (restrictions.Count == 3)
285 {
286     var index = restrictions[Constants.IndexPart];
287     var source = restrictions[Constants.SourcePart];
288     var target = restrictions[Constants.TargetPart];
289     if (index == Constants.Any)
290     {
291         if (source == Constants.Any && target == Constants.Any)
292         {
293             return Each(handler, ArrayPool<ulong>.Empty);
294         }
295         else if (source == Constants.Any)
296         {
297             return _targetsTreeMethods.EachReference(target, handler);
298         }
299         else if (target == Constants.Any)
300         {
301             return _sourcesTreeMethods.EachReference(source, handler);
302         }
303         else //if(source != Any && target != Any)
304         {
305             var link = _sourcesTreeMethods.Search(source, target);
306             return link == Constants.Null ? Constants.Continue :
307                 ↪ handler(GetLinkStruct(link));
308         }
309     }
310     else
311     {
312         if (!Exists(index))
313         {
314             return Constants.Continue;
315         }
316         if (source == Constants.Any && target == Constants.Any)
317         {
318             return handler(GetLinkStruct(index));
319         }
320         var storedLinkValue = GetLinkUnsafe(index);
321         if (source != Constants.Any && target != Constants.Any)
322         {
323             if (storedLinkValue->Source == source &&
324                 storedLinkValue->Target == target)
325             {
326                 return handler(GetLinkStruct(index));
327             }
328             return Constants.Continue;
329         }
330         var value = default(id);
331         if (source == Constants.Any)

```

```

332         value = target;
333     }
334     if (target == Constants.Any)
335     {
336         value = source;
337     }
338     if (storedLinkValue->Source == value ||
339         storedLinkValue->Target == value)
340     {
341         return handler(GetLinkStruct(index));
342     }
343     return Constants.Continue;
344 }
345 }
346 throw new NotSupportedException("Другие размеры и способы ограничений не
    ↳ поддерживаются.");
347 }
348
349 /// <remarks>
350 /// TODO: Возможно можно перемещать значения, если указан индекс, но значение существует
351 ↳ в другом месте (но не в менеджере памяти, а в логике Links)
352 /// </remarks>
353 [MethodImpl(MethodImplOptions.AggressiveInlining)]
354 public id Update(IList<id> restrictions, IList<id> substitution)
355 {
356     var linkIndex = restrictions[Constants.IndexPart];
357     var link = GetLinkUnsafe(linkIndex);
358     // Будет корректно работать только в том случае, если пространство выделенной связи
359     ↳ предварительно заполнено нулями
360     if (link->Source != Constants.Null)
361     {
362         _sourcesTreeMethods.Detach(ref _header->FirstAsSource, linkIndex);
363     }
364     if (link->Target != Constants.Null)
365     {
366         _targetsTreeMethods.Detach(ref _header->FirstAsTarget, linkIndex);
367     }
368 #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
369     var leftTreeSize = _sourcesTreeMethods.GetSize(new IntPtr(&_header->FirstAsSource));
370     var rightTreeSize = _targetsTreeMethods.GetSize(new IntPtr(&_header->FirstAsTarget));
371     if (leftTreeSize != rightTreeSize)
372     {
373         throw new Exception("One of the trees is broken.");
374     }
375 #endif
376     link->Source = substitution[Constants.SourcePart];
377     link->Target = substitution[Constants.TargetPart];
378     if (link->Source != Constants.Null)
379     {
380         _sourcesTreeMethods.Attach(ref _header->FirstAsSource, linkIndex);
381     }
382     if (link->Target != Constants.Null)
383     {
384         _targetsTreeMethods.Attach(ref _header->FirstAsTarget, linkIndex);
385     }
386 #if ENABLE_TREE_AUTO_DEBUG_AND_VALIDATION
387     leftTreeSize = _sourcesTreeMethods.GetSize(new IntPtr(&_header->FirstAsSource));
388     rightTreeSize = _targetsTreeMethods.GetSize(new IntPtr(&_header->FirstAsTarget));
389     if (leftTreeSize != rightTreeSize)
390     {
391         throw new Exception("One of the trees is broken.");
392     }
393 #endif
394     return linkIndex;
395 }
396
397 [MethodImpl(MethodImplOptions.AggressiveInlining)]
398 private IList<id> GetLinkStruct(id linkIndex)
399 {
400     var link = GetLinkUnsafe(linkIndex);
401     return new UInt64Link(linkIndex, link->Source, link->Target);
402 }
403
404 [MethodImpl(MethodImplOptions.AggressiveInlining)]
405 private Link* GetLinkUnsafe(id linkIndex) => &_amp;links[linkIndex];
406
407 /// <remarks>
408 /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
409 ↳ пространство

```

```

407 /// </remarks>
408 public id Create(IList<id> restrictions)
409 {
410     var freeLink = _header->FirstFreeLink;
411     if (freeLink != Constants.Null)
412     {
413         _unusedLinksListMethods.Detach(freeLink);
414     }
415     else
416     {
417         var maximumPossibleInnerReference =
418             ↪ Constants.PossibleInnerReferencesRange.Maximum;
419         if (_header->AllocatedLinks > maximumPossibleInnerReference)
420         {
421             throw new LinksLimitReachedException<id>(maximumPossibleInnerReference);
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(IList<id> restrictions)
437 {
438     var link = restrictions[Constants.IndexPart];
439     if (link < _header->AllocatedLinks)
440     {
441         _unusedLinksListMethods.AttachAsFirst(link);
442     }
443     else if (link == _header->AllocatedLinks)
444     {
445         _header->AllocatedLinks--;
446         _memory.UsedCapacity -= sizeof(Link);
447         // Убираем все связи, находящиеся в списке свободных в конце файла, до тех пор,
448         // пока не дойдём до первой существующей связи
449         // Позволяет оптимизировать количество выделенных связей (AllocatedLinks)
450         while (_header->AllocatedLinks > 0 && IsUnusedLink(_header->AllocatedLinks))
451         {
452             _unusedLinksListMethods.Detach(_header->AllocatedLinks);
453             _header->AllocatedLinks--;
454             _memory.UsedCapacity -= sizeof(Link);
455         }
456     }
457 }
458
459 /// <remarks>
460 /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том случае, если
461 /// ↪ адрес реально поменялся
462 ///
463 /// Указатель this.links может быть в том же месте,
464 /// так как 0-я связь не используется и имеет такой же размер как Header,
465 /// поэтому header размещается в том же месте, что и 0-я связь
466 /// </remarks>
467 private void SetPointers(IResizableDirectMemory memory)
468 {
469     if (memory == null)
470     {
471         _header = null;
472         _links = null;
473         _unusedLinksListMethods = null;
474         _targetsTreeMethods = null;
475         _unusedLinksListMethods = null;
476     }
477     else
478     {
479         _header = (LinksHeader*)(void*)memory.Pointer;
480         _links = (Link*)(void*)memory.Pointer;
481         _sourcesTreeMethods = new LinksSourcesTreeMethods(this);
482         _targetsTreeMethods = new LinksTargetsTreeMethods(this);
483         _unusedLinksListMethods = new UnusedLinksListMethods(_links, _header);
484     }
485 }

```

```

483     [MethodImpl(MethodImplOptions.AggressiveInlining)]
484     private bool Exists(id link) => link >= Constants.PossibleInnerReferencesRange.Minimum
485     ↪ && link <= _header->AllocatedLinks && !IsUnusedLink(link);
486
487     [MethodImpl(MethodImplOptions.AggressiveInlining)]
488     private bool IsUnusedLink(id link) => _header->FirstFreeLink == link
489     || (_links[link].SizeAsSource == Constants.Null &&
490     ↪ _links[link].Source != Constants.Null);
491
492     #region Disposable
493
494     protected override bool AllowMultipleDisposeCalls => true;
495
496     protected override void Dispose(bool manual, bool wasDisposed)
497     {
498         if (!wasDisposed)
499         {
500             SetPointers(null);
501             _memory.DisposeIfPossible();
502         }
503     }
504     #endregion
505 }
506 }

```

```

./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      public 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         }
42     }
43 }

```

```

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

```

```

8
9 namespace Platform.Data.Doublets.ResizableDirectMemory
10 {
11     public unsafe partial class UInt64ResizableDirectMemoryLinks
12     {
13         private abstract class LinksTreeMethodsBase :
14             ↳ SizedAndThreadedAVLBalancedTreeMethods<ulong>
15         {
16             private readonly UInt64ResizableDirectMemoryLinks _memory;
17             private readonly LinksConstants<ulong> _constants;
18             protected readonly Link* Links;
19             protected readonly LinksHeader* Header;
20
21             protected LinksTreeMethodsBase(UInt64ResizableDirectMemoryLinks memory)
22             {
23                 Links = memory._links;
24                 Header = memory._header;
25                 _memory = memory;
26                 _constants = memory.Constants;
27             }
28
29             [MethodImpl(MethodImplOptions.AggressiveInlining)]
30             protected abstract ulong GetTreeRoot();
31
32             [MethodImpl(MethodImplOptions.AggressiveInlining)]
33             protected abstract ulong GetBasePartValue(ulong link);
34
35             public ulong this[ulong index]
36             {
37                 get
38                 {
39                     var root = GetTreeRoot();
40                     if (index >= GetSize(root))
41                     {
42                         return 0;
43                     }
44                     while (root != 0)
45                     {
46                         var left = GetLeftOrDefault(root);
47                         var leftSize = GetSizeOrZero(left);
48                         if (index < leftSize)
49                         {
50                             root = left;
51                             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
61                     ↳ broken)
62                 }
63             }
64
65             // TODO: Return indices range instead of references count
66             public ulong CountUsages(ulong link)
67             {
68                 var root = GetTreeRoot();
69                 var total = GetSize(root);
70                 var totalRightIgnore = OUL;
71                 while (root != 0)
72                 {
73                     var @base = GetBasePartValue(root);
74                     if (@base <= link)
75                     {
76                         root = GetRightOrDefault(root);
77                     }
78                     else
79                     {
80                         totalRightIgnore += GetRightSize(root) + 1;
81                         root = GetLeftOrDefault(root);
82                     }
83                 }
84                 root = GetTreeRoot();
85                 var totalLeftIgnore = OUL;
86                 while (root != 0)
87                 {

```



```

86         var @base = GetBasePartValue(root);
87         if (@base >= link)
88         {
89             root = GetLeftOrDefault(root);
90         }
91         else
92         {
93             totalLeftIgnore += GetLeftSize(root) + 1;
94             root = GetRightOrDefault(root);
95         }
96     }
97     return total - totalRightIgnore - totalLeftIgnore;
98 }
99
100 public ulong EachReference(ulong link, Func<IList<ulong>, ulong> handler)
101 {
102     var root = GetTreeRoot();
103     if (root == 0)
104     {
105         return _constants.Continue;
106     }
107     ulong first = 0, current = root;
108     while (current != 0)
109     {
110         var @base = GetBasePartValue(current);
111         if (@base >= link)
112         {
113             if (@base == link)
114             {
115                 first = current;
116             }
117             current = GetLeftOrDefault(current);
118         }
119         else
120         {
121             current = GetRightOrDefault(current);
122         }
123     }
124     if (first != 0)
125     {
126         current = first;
127         while (true)
128         {
129             if (handler(_memory.GetLinkStruct(current)) == _constants.Break)
130             {
131                 return _constants.Break;
132             }
133             current = GetNext(current);
134             if (current == 0 || GetBasePartValue(current) != link)
135             {
136                 break;
137             }
138         }
139     }
140     return _constants.Continue;
141 }
142
143 protected override void PrintNodeValue(ulong node, StringBuilder sb)
144 {
145     sb.Append(' ');
146     sb.Append(Links[node].Source);
147     sb.Append('-');
148     sb.Append('>');
149     sb.Append(Links[node].Target);
150 }
151
152 private class LinksSourcesTreeMethods : LinksTreeMethodsBase
153 {
154     public LinksSourcesTreeMethods(UInt64ResizableDirectMemoryLinks memory)
155         : base(memory)
156     {
157     }
158
159     protected override ref ulong GetLeftReference(ulong node) => ref
160         ↪ Links[node].LeftAsSource;
161
162     protected override ref ulong GetRightReference(ulong node) => ref
163         ↪ Links[node].RightAsSource;

```

```

163     protected override ulong GetLeft(ulong node) => Links[node].LeftAsSource;
164
165     protected override ulong GetRight(ulong node) => Links[node].RightAsSource;
166
167     protected override ulong GetSize(ulong node)
168     {
169         var previousValue = Links[node].SizeAsSource;
170         //return Math.PartialRead(previousValue, 5, -5);
171         return (previousValue & 4294967264) >> 5;
172     }
173
174     protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsSource
175     ↪ = left;
176
177     protected override void SetRight(ulong node, ulong right) =>
178     ↪ Links[node].RightAsSource = right;
179
180     protected override void SetSize(ulong node, ulong size)
181     {
182         var previousValue = Links[node].SizeAsSource;
183         //var modified = Math.PartialWrite(previousValue, size, 5, -5);
184         var modified = (previousValue & 31) | ((size & 134217727) << 5);
185         Links[node].SizeAsSource = modified;
186     }
187
188     protected override bool GetLeftIsChild(ulong node)
189     {
190         var previousValue = Links[node].SizeAsSource;
191         //return (Integer)Math.PartialRead(previousValue, 4, 1);
192         return (previousValue & 16) >> 4 == 1UL;
193     }
194
195     protected override void SetLeftIsChild(ulong node, bool value)
196     {
197         var previousValue = Links[node].SizeAsSource;
198         //var modified = Math.PartialWrite(previousValue, (ulong)(Integer)value, 4, 1);
199         var modified = (previousValue & 4294967279) | ((value ? 1UL : 0UL) << 4);
200         Links[node].SizeAsSource = modified;
201     }
202
203     protected override bool GetRightIsChild(ulong node)
204     {
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);

```

```

238
239     protected override bool FirstIsToTheRightOfSecond(ulong first, ulong second)
240     => Links[first].Source > Links[second].Source ||
241         (Links[first].Source == Links[second].Source && Links[first].Target >
242             Links[second].Target);
243
244     protected override ulong GetTreeRoot() => Header->FirstAsSource;
245
246     protected override ulong GetBasePartValue(ulong link) => Links[link].Source;
247
248     /// <summary>
249     /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
250     /// (концом)
251     /// по дереву (индексу) связей, отсортированному по Source, а затем по Target.
252     /// </summary>
253     /// <param name="source">Индекс связи, которая является началом на искомой
254     /// связи.</param>
255     /// <param name="target">Индекс связи, которая является концом на искомой
256     /// связи.</param>
257     /// <returns>Индекс искомой связи.</returns>
258     public ulong Search(ulong source, ulong target)
259     {
260         var root = Header->FirstAsSource;
261         while (root != 0)
262         {
263             var rootSource = Links[root].Source;
264             var rootTarget = Links[root].Target;
265             if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
266                 ↳ node.Key < root.Key
267             {
268                 root = GetLeftOrDefault(root);
269             }
270             else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget))
271                 ↳ // node.Key > root.Key
272             {
273                 root = GetRightOrDefault(root);
274             }
275             else // node.Key == root.Key
276             {
277                 return root;
278             }
279         }
280         return 0;
281     }
282
283     [MethodImpl(MethodImplOptions.AggressiveInlining)]
284     private static bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
285         ↳ ulong secondSource, ulong secondTarget)
286     => firstSource < secondSource || (firstSource == secondSource && firstTarget <
287         ↳ secondTarget);
288
289     [MethodImpl(MethodImplOptions.AggressiveInlining)]
290     private static bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
291         ↳ ulong secondSource, ulong secondTarget)
292     => firstSource > secondSource || (firstSource == secondSource && firstTarget >
293         ↳ secondTarget);
294
295     [MethodImpl(MethodImplOptions.AggressiveInlining)]
296     protected override void ClearNode(ulong node)
297     {
298         Links[node].LeftAsSource = OUL;
299         Links[node].RightAsSource = OUL;
300         Links[node].SizeAsSource = OUL;
301     }
302
303     [MethodImpl(MethodImplOptions.AggressiveInlining)]
304     protected override ulong GetZero() => OUL;
305
306     [MethodImpl(MethodImplOptions.AggressiveInlining)]
307     protected override bool EqualToZero(ulong value) => value == OUL;
308
309     [MethodImpl(MethodImplOptions.AggressiveInlining)]
310     protected override bool IsEquals(ulong first, ulong second) => first == second;
311
312     [MethodImpl(MethodImplOptions.AggressiveInlining)]
313     protected override bool GreaterThanZero(ulong value) => value > OUL;
314
315     [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

306     protected override bool GreaterThan(ulong first, ulong second) => first > second;
307
308     [MethodImpl(MethodImplOptions.AggressiveInlining)]
309     protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >=
        ↳ second;
310
311     [MethodImpl(MethodImplOptions.AggressiveInlining)]
312     protected override bool GreaterOrEqualThanZero(ulong value) => true; // value >= 0
        ↳ is always true for ulong
313
314     [MethodImpl(MethodImplOptions.AggressiveInlining)]
315     protected override bool LessOrEqualThanZero(ulong value) => value == 0; // value is
        ↳ always >= 0 for ulong
316
317     [MethodImpl(MethodImplOptions.AggressiveInlining)]
318     protected override bool LessOrEqualThan(ulong first, ulong second) => first <=
        ↳ second;
319
320     [MethodImpl(MethodImplOptions.AggressiveInlining)]
321     protected override bool LessThanZero(ulong value) => false; // value < 0 is always
        ↳ false for ulong
322
323     [MethodImpl(MethodImplOptions.AggressiveInlining)]
324     protected override bool LessThan(ulong first, ulong second) => first < second;
325
326     [MethodImpl(MethodImplOptions.AggressiveInlining)]
327     protected override ulong Increment(ulong value) => ++value;
328
329     [MethodImpl(MethodImplOptions.AggressiveInlining)]
330     protected override ulong Decrement(ulong value) => --value;
331
332     [MethodImpl(MethodImplOptions.AggressiveInlining)]
333     protected override ulong Add(ulong first, ulong second) => first + second;
334
335     [MethodImpl(MethodImplOptions.AggressiveInlining)]
336     protected override ulong Subtract(ulong first, ulong second) => first - second;
337 }
338
339 private class LinksTargetsTreeMethods : LinksTreeMethodsBase
340 {
341     public LinksTargetsTreeMethods(UInt64ResizableDirectMemoryLinks memory)
342         : base(memory)
343     {
344     }
345
346     //protected override IntPtr GetLeft(ulong node) => new
        ↳ IntPtr(&Links[node].LeftAsTarget);
347
348     //protected override IntPtr GetRight(ulong node) => new
        ↳ IntPtr(&Links[node].RightAsTarget);
349
350     //protected override ulong GetSize(ulong node) => Links[node].SizeAsTarget;
351
352     //protected override void SetLeft(ulong node, ulong left) =>
        ↳ Links[node].LeftAsTarget = left;
353
354     //protected override void SetRight(ulong node, ulong right) =>
        ↳ Links[node].RightAsTarget = right;
355
356     //protected override void SetSize(ulong node, ulong size) =>
        ↳ Links[node].SizeAsTarget = size;
357
358     protected override ref ulong GetLeftReference(ulong node) => ref
        ↳ Links[node].LeftAsTarget;
359
360     protected override ref ulong GetRightReference(ulong node) => ref
        ↳ Links[node].RightAsTarget;
361
362     protected override ulong GetLeft(ulong node) => Links[node].LeftAsTarget;
363
364     protected override ulong GetRight(ulong node) => Links[node].RightAsTarget;
365
366     protected override ulong GetSize(ulong node)
367     {
368         var previousValue = Links[node].SizeAsTarget;
369         //return Math.PartialRead(previousValue, 5, -5);
370         return (previousValue & 4294967264) >> 5;
371     }
372

```

```

373     protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsTarget
374         ↳ = left;
375
376     protected override void SetRight(ulong node, ulong right) =>
377         ↳ Links[node].RightAsTarget = right;
378
379     protected override void SetSize(ulong node, ulong size)
380     {
381         var previousValue = Links[node].SizeAsTarget;
382         //var modified = Math.PartialWrite(previousValue, size, 5, -5);
383         var modified = (previousValue & 31) | ((size & 134217727) << 5);
384         Links[node].SizeAsTarget = modified;
385     }
386
387     protected override bool GetLeftIsChild(ulong node)
388     {
389         var previousValue = Links[node].SizeAsTarget;
390         //return (Integer)Math.PartialRead(previousValue, 4, 1);
391         return (previousValue & 16) >> 4 == 1UL;
392         // TODO: Check if this is possible to use
393         //var nodeSize = GetSize(node);
394         //var left = GetLeft(node);
395         //var leftSize = GetSizeOrZero(left);
396         //return leftSize > 0 && nodeSize > leftSize;
397     }
398
399     protected override void SetLeftIsChild(ulong node, bool value)
400     {
401         var previousValue = Links[node].SizeAsTarget;
402         //var modified = Math.PartialWrite(previousValue, (ulong)(Integer)value, 4, 1);
403         var modified = (previousValue & 4294967279) | ((value ? 1UL : 0UL) << 4);
404         Links[node].SizeAsTarget = modified;
405     }
406
407     protected override bool GetRightIsChild(ulong node)
408     {
409         var previousValue = Links[node].SizeAsTarget;
410         //return (Integer)Math.PartialRead(previousValue, 3, 1);
411         return (previousValue & 8) >> 3 == 1UL;
412         // TODO: Check if this is possible to use
413         //var nodeSize = GetSize(node);
414         //var right = GetRight(node);
415         //var rightSize = GetSizeOrZero(right);
416         //return rightSize > 0 && nodeSize > rightSize;
417     }
418
419     protected override void SetRightIsChild(ulong node, bool value)
420     {
421         var previousValue = Links[node].SizeAsTarget;
422         //var modified = Math.PartialWrite(previousValue, (ulong)(Integer)value, 3, 1);
423         var modified = (previousValue & 4294967287) | ((value ? 1UL : 0UL) << 3);
424         Links[node].SizeAsTarget = modified;
425     }
426
427     protected override sbyte GetBalance(ulong node)
428     {
429         var previousValue = Links[node].SizeAsTarget;
430         //var value = Math.PartialRead(previousValue, 0, 3);
431         var value = previousValue & 7;
432         var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) << 5) | value & 3 |
433             ↳ 124 : value & 3);
434         return unpackedValue;
435     }
436
437     protected override void SetBalance(ulong node, sbyte value)
438     {
439         var previousValue = Links[node].SizeAsTarget;
440         var packagedValue = (ulong)((((byte)value >> 5) & 4) | value & 3);
441         //var modified = Math.PartialWrite(previousValue, packagedValue, 0, 3);
442         var modified = (previousValue & 4294967288) | (packagedValue & 7);
443         Links[node].SizeAsTarget = modified;
444     }
445
446     protected override bool FirstIsToTheLeftOfSecond(ulong first, ulong second)
447     => Links[first].Target < Links[second].Target ||
448         (Links[first].Target == Links[second].Target && Links[first].Source <
449             ↳ Links[second].Source);

```

```

447         protected override bool FirstIsToTheRightOfSecond(ulong first, ulong second)
448             => Links[first].Target > Links[second].Target ||
449             (Links[first].Target == Links[second].Target && Links[first].Source >
450              ↳ Links[second].Source);
451
452         protected override ulong GetTreeRoot() => Header->FirstAsTarget;
453
454         protected override ulong GetBasePartValue(ulong link) => Links[link].Target;
455
456         [MethodImpl(MethodImplOptions.AggressiveInlining)]
457         protected override void ClearNode(ulong node)
458         {
459             Links[node].LeftAsTarget = OUL;
460             Links[node].RightAsTarget = OUL;
461             Links[node].SizeAsTarget = OUL;
462         }
463     }
464 }

```

./Platform.Data.Doublets/Sequences/ArrayExtensions.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.Sequences
7  {
8      public static class ArrayExtensions
9      {
10         public static IList<TLink> ConvertToRestrictionsValues<TLink>(this TLink[] array)
11         {
12             var restrictions = new TLink[array.Length + 1];
13             Array.Copy(array, 0, restrictions, 1, array.Length);
14             return restrictions;
15         }
16     }
17 }

```

./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(IList<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     }
41 }

```

```

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.Doublets.Sequences.Frequencies.Cache;
9
10 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
11
12 namespace Platform.Data.Doublets.Sequences.Converters
13 {
14     /// <remarks>
15     /// TODO: Возможно будет лучше если алгоритм будет выполняться полностью изолированно от
16     ///     Links на этапе сжатия.
17     ///     А именно будет создаваться временный список пар необходимых для выполнения сжатия, в
18     ///     таком случае тип значения элемента массива может быть любым, как char так и ulong.
19     ///     Как только список/словарь пар был выявлен можно разом выполнить создание всех этих
20     ///     пар, а так же разом выполнить замену.
21     /// </remarks>
22     public class CompressingConverter<TLink> : LinksListToSequenceConverterBase<TLink>
23     {
24         private static readonly LinksConstants<TLink> _constants =
25             ↳ Default<LinksConstants<TLink>>.Instance;
26         private static readonly EqualityComparer<TLink> _equalityComparer =
27             ↳ EqualityComparer<TLink>.Default;
28         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
29
30         private readonly IConverter<IList<TLink>, TLink> _baseConverter;
31         private readonly LinkFrequenciesCache<TLink> _doubletFrequenciesCache;
32         private readonly TLink _minFrequencyToCompress;
33         private readonly bool _doInitialFrequenciesIncrement;
34         private Doublet<TLink> _maxDoublet;
35         private LinkFrequency<TLink> _maxDoubletData;
36
37         private struct HalfDoublet
38         {
39             public TLink Element;
40             public LinkFrequency<TLink> DoubletData;
41
42             public HalfDoublet(TLink element, LinkFrequency<TLink> doubletData)
43             {
44                 Element = element;
45                 DoubletData = doubletData;
46             }
47
48             public override string ToString() => $"{Element}: ({DoubletData})";
49         }
50
51         public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
52             ↳ baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache)
53             : this(links, baseConverter, doubletFrequenciesCache, Integer<TLink>.One, true)
54         {
55         }
56
57         public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
58             ↳ baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache, bool
59             ↳ doInitialFrequenciesIncrement)
60             : this(links, baseConverter, doubletFrequenciesCache, Integer<TLink>.One,
61                 ↳ doInitialFrequenciesIncrement)
62         {
63         }
64     }
65 }

```

```

55 public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
56     ↳ baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache, TLink
57     ↳ minFrequencyToCompress, bool doInitialFrequenciesIncrement)
58     : base(links)
59 {
60     _baseConverter = baseConverter;
61     _doubletFrequenciesCache = doubletFrequenciesCache;
62     if (_comparer.Compare(minFrequencyToCompress, Integer<TLink>.One) < 0)
63     {
64         minFrequencyToCompress = Integer<TLink>.One;
65     }
66     _minFrequencyToCompress = minFrequencyToCompress;
67     _doInitialFrequenciesIncrement = doInitialFrequenciesIncrement;
68     ResetMaxDoublet();
69 }
70
71 public override TLink Convert(IList<TLink> source) =>
72     ↳ _baseConverter.Convert(Compress(source));
73
74 /// <remarks>
75 /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte\_pair\_encoding .
76 /// Faster version (doublets' frequencies dictionary is not recreated).
77 /// </remarks>
78 private IList<TLink> Compress(IList<TLink> sequence)
79 {
80     if (sequence.IsNullOrEmpty())
81     {
82         return null;
83     }
84     if (sequence.Count == 1)
85     {
86         return sequence;
87     }
88     if (sequence.Count == 2)
89     {
90         return new[] { Links.GetOrCreate(sequence[0], sequence[1]) };
91     }
92     // TODO: arraypool with min size (to improve cache locality) or stackalloc with Sigil
93     var copy = new HalfDoublet[sequence.Count];
94     Doublet<TLink> doublet = default;
95     for (var i = 1; i < sequence.Count; i++)
96     {
97         doublet.Source = sequence[i - 1];
98         doublet.Target = sequence[i];
99         LinkFrequency<TLink> data;
100         if (_doInitialFrequenciesIncrement)
101         {
102             data = _doubletFrequenciesCache.IncrementFrequency(ref doublet);
103         }
104         else
105         {
106             data = _doubletFrequenciesCache.GetFrequency(ref doublet);
107             if (data == null)
108             {
109                 throw new NotSupportedException("If you ask not to increment
110                 ↳ frequencies, it is expected that all frequencies for the sequence
111                 ↳ are prepared.");
112             }
113         }
114         copy[i - 1].Element = sequence[i - 1];
115         copy[i - 1].DoubletData = data;
116         UpdateMaxDoublet(ref doublet, data);
117     }
118     copy[sequence.Count - 1].Element = sequence[sequence.Count - 1];
119     copy[sequence.Count - 1].DoubletData = new LinkFrequency<TLink>();
120     if (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
121     {
122         var newLength = ReplaceDoublets(copy);
123         sequence = new TLink[newLength];
124         for (int i = 0; i < newLength; i++)
125         {
126             sequence[i] = copy[i].Element;
127         }
128     }
129     return sequence;
130 }
131
132 /// <remarks>

```



```

129 /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte\_pair\_encoding
130 /// </remarks>
131 private int ReplaceDoublets(HalfDoublet[] copy)
132 {
133     var oldLength = copy.Length;
134     var newLength = copy.Length;
135     while (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
136     {
137         var maxDoubletSource = _maxDoublet.Source;
138         var maxDoubletTarget = _maxDoublet.Target;
139         if (_equalityComparer.Equals(_maxDoubletData.Link, _constants.Null))
140         {
141             _maxDoubletData.Link = Links.GetOrCreate(maxDoubletSource, maxDoubletTarget);
142         }
143         var maxDoubletReplacementLink = _maxDoubletData.Link;
144         oldLength--;
145         var oldLengthMinusTwo = oldLength - 1;
146         // Substitute all usages
147         int w = 0, r = 0; // (r == read, w == write)
148         for (; r < oldLength; r++)
149         {
150             if (_equalityComparer.Equals(copy[r].Element, maxDoubletSource) &&
151                 ↪ _equalityComparer.Equals(copy[r + 1].Element, maxDoubletTarget))
152             {
153                 if (r > 0)
154                 {
155                     var previous = copy[w - 1].Element;
156                     copy[w - 1].DoubletData.DecrementFrequency();
157                     copy[w - 1].DoubletData =
158                         ↪ _doubletFrequenciesCache.IncrementFrequency(previous,
159                         ↪ maxDoubletReplacementLink);
160                 }
161                 if (r < oldLengthMinusTwo)
162                 {
163                     var next = copy[r + 2].Element;
164                     copy[r + 1].DoubletData.DecrementFrequency();
165                     copy[w].DoubletData = _doubletFrequenciesCache.IncrementFrequency(maxDoubletReplacementLink,
166                         ↪ next);
167                 }
168                 copy[w++].Element = maxDoubletReplacementLink;
169                 r++;
170                 newLength--;
171             }
172             else
173             {
174                 copy[w++] = copy[r];
175             }
176         }
177         if (w < newLength)
178         {
179             copy[w] = copy[r];
180         }
181         oldLength = newLength;
182         ResetMaxDoublet();
183         UpdateMaxDoublet(copy, newLength);
184     }
185     return newLength;
186 }
187
188 [MethodImpl(MethodImplOptions.AggressiveInlining)]
189 private void ResetMaxDoublet()
190 {
191     _maxDoublet = new Doublet<TLink>();
192     _maxDoubletData = new LinkFrequency<TLink>();
193 }
194
195 [MethodImpl(MethodImplOptions.AggressiveInlining)]
196 private void UpdateMaxDoublet(HalfDoublet[] copy, int length)
197 {
198     Doublet<TLink> doublet = default;
199     for (var i = 1; i < length; i++)
200     {
201         doublet.Source = copy[i - 1].Element;
202         doublet.Target = copy[i].Element;
203         UpdateMaxDoublet(ref doublet, copy[i - 1].DoubletData);
204     }
205 }

```

```

203 [MethodImpl(MethodImplOptions.AggressiveInlining)]
204 private void UpdateMaxDoublet(ref Doublet<TLink> doublet, LinkFrequency<TLink> data)
205 {
206     var frequency = data.Frequency;
207     var maxFrequency = _maxDoubletData.Frequency;
208     //if (frequency > _minFrequencyToCompress && (maxFrequency < frequency ||
    ↪ (maxFrequency == frequency && doublet.Source + doublet.Target < /* gives better
    ↪ compression string data (and gives collisions quickly) */ _maxDoublet.Source +
    ↪ _maxDoublet.Target)))
209 if (_comparer.Compare(frequency, _minFrequencyToCompress) > 0 &&
210 (_comparer.Compare(maxFrequency, frequency) < 0 ||
    ↪ (_equalityComparer.Equals(maxFrequency, frequency) &&
    ↪ _comparer.Compare(Arithmetic.Add(doublet.Source, doublet.Target),
    ↪ Arithmetic.Add(_maxDoublet.Source, _maxDoublet.Target)) > 0))) /* gives
    ↪ better stability and better compression on sequent data and even on runderm
    ↪ numbers data (but gives collisions anyway) */
211 {
212     _maxDoublet = doublet;
213     _maxDoubletData = data;
214 }
215 }
216 }
217 }

```

./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>,
    ↪ TLink>
9     {
10         protected readonly ILinks<TLink> Links;
11         public LinksListToSequenceConverterBase(ILinks<TLink> links) => Links = links;
12         public abstract TLink Convert(IList<TLink> source);
13     }
14 }

```

./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 =
    ↪ EqualityComparer<TLink>.Default;
12         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
13
14         private readonly IConverter<IList<TLink>> _sequenceToItsLocalElementLevelsConverter;
15
16         public OptimalVariantConverter(ILinks<TLink> links, IConverter<IList<TLink>>
    ↪ sequenceToItsLocalElementLevelsConverter) : base(links)
17         => _sequenceToItsLocalElementLevelsConverter =
    ↪ sequenceToItsLocalElementLevelsConverter;
18
19         public override TLink Convert(IList<TLink> sequence)
20         {
21             var length = sequence.Count;
22             if (length == 1)
23             {
24                 return sequence[0];
25             }
26             var links = Links;
27             if (length == 2)
28             {
29                 return links.GetOrCreate(sequence[0], sequence[1]);
30             }
31             sequence = sequence.ToArray();
32             var levels = _sequenceToItsLocalElementLevelsConverter.Convert(sequence);
33             while (length > 2)
34             {

```

```

35     var levelRepeat = 1;
36     var currentLevel = levels[0];
37     var previousLevel = levels[0];
38     var skipOnce = false;
39     var w = 0;
40     for (var i = 1; i < length; i++)
41     {
42         if (_equalityComparer.Equals(currentLevel, levels[i]))
43         {
44             levelRepeat++;
45             skipOnce = false;
46             if (levelRepeat == 2)
47             {
48                 sequence[w] = links.GetOrCreate(sequence[i - 1], sequence[i]);
49                 var newLevel = i >= length - 1 ?
50                     GetPreviousLowerThanCurrentOrCurrent(previousLevel,
51                     ↪ currentLevel) :
52                     i < 2 ?
53                     GetNextLowerThanCurrentOrCurrent(currentLevel, levels[i + 1]) :
54                     GetGreatestNeighbourLowerThanCurrentOrCurrent(previousLevel,
55                     ↪ currentLevel, levels[i + 1]);
56                 levels[w] = newLevel;
57                 previousLevel = currentLevel;
58                 w++;
59                 levelRepeat = 0;
60                 skipOnce = true;
61             }
62             else if (i == length - 1)
63             {
64                 sequence[w] = sequence[i];
65                 levels[w] = levels[i];
66                 w++;
67             }
68         }
69         else
70         {
71             currentLevel = levels[i];
72             levelRepeat = 1;
73             if (skipOnce)
74             {
75                 skipOnce = false;
76             }
77             else
78             {
79                 sequence[w] = sequence[i - 1];
80                 levels[w] = levels[i - 1];
81                 previousLevel = levels[w];
82                 w++;
83             }
84             if (i == length - 1)
85             {
86                 sequence[w] = sequence[i];
87                 levels[w] = levels[i];
88                 w++;
89             }
90         }
91     }
92     length = w;
93     return links.GetOrCreate(sequence[0], sequence[1]);
94 }
95 private static TLink GetGreatestNeighbourLowerThanCurrentOrCurrent(TLink previous, TLink
96 ↪ current, TLink next)
97 {
98     return _comparer.Compare(previous, next) > 0
99         ? _comparer.Compare(previous, current) < 0 ? previous : current
100         : _comparer.Compare(next, current) < 0 ? next : current;
101 }
102 private static TLink GetNextLowerThanCurrentOrCurrent(TLink current, TLink next) =>
103     ↪ _comparer.Compare(next, current) < 0 ? next : current;
104 private static TLink GetPreviousLowerThanCurrentOrCurrent(TLink previous, TLink current)
105     ↪ => _comparer.Compare(previous, current) < 0 ? previous : current;
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>,
9         ⇨ IConverter<IList<TLink>>
10     {
11         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
12
13         private readonly IConverter<Doublet<TLink>, TLink> _linkToItsFrequencyToNumberConveter;
14
15         public SequenceToItsLocalElementLevelsConverter(ILinks<TLink> links,
16             ⇨ IConverter<Doublet<TLink>, TLink> linkToItsFrequencyToNumberConveter) : base(links)
17             ⇨ => _linkToItsFrequencyToNumberConveter = linkToItsFrequencyToNumberConveter;
18
19         public IList<TLink> Convert(IList<TLink> sequence)
20         {
21             var levels = new TLink[sequence.Count];
22             levels[0] = GetFrequencyNumber(sequence[0], sequence[1]);
23             for (var i = 1; i < sequence.Count - 1; i++)
24             {
25                 var previous = GetFrequencyNumber(sequence[i - 1], sequence[i]);
26                 var next = GetFrequencyNumber(sequence[i], sequence[i + 1]);
27                 levels[i] = _comparer.Compare(previous, next) > 0 ? previous : next;
28             }
29             levels[levels.Length - 1] = GetFrequencyNumber(sequence[sequence.Count - 2],
30                 ⇨ sequence[sequence.Count - 1]);
31             return levels;
32         }
33
34         public TLink GetFrequencyNumber(TLink source, TLink target) =>
35             ⇨ _linkToItsFrequencyToNumberConveter.Convert(new Doublet<TLink>(source, target));
36     }
37 }
```

./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>,
8         ⇨ ICriterionMatcher<TLink>
9     {
10         public DefaultSequenceElementCriterionMatcher(ILinks<TLink> links) : base(links) { }
11         public bool IsMatched(TLink argument) => Links.IsPartialPoint(argument);
12     }
13 }
```

./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 }
```

```

24     }
25 }

./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>,
        ↳ ISequenceAppender<TLink>
11     {
12         private static readonly EqualityComparer<TLink> _equalityComparer =
        ↳ EqualityComparer<TLink>.Default;
13
14         private readonly IStack<TLink> _stack;
15         private readonly ISequenceHeightProvider<TLink> _heightProvider;
16
17         public DefaultSequenceAppender(ILinks<TLink> links, IStack<TLink> stack,
        ↳ ISequenceHeightProvider<TLink> heightProvider)
        : base(links)
18         {
19             _stack = stack;
20             _heightProvider = heightProvider;
21         }
22
23         public TLink Append(TLink sequence, TLink appendant)
24         {
25             var cursor = sequence;
26             while (!_equalityComparer.Equals(_heightProvider.Get(cursor), default))
27             {
28                 var source = Links.GetSource(cursor);
29                 var target = Links.GetTarget(cursor);
30                 if (_equalityComparer.Equals(_heightProvider.Get(source),
        ↳ _heightProvider.Get(target)))
31                 {
32                     break;
33                 }
34                 else
35                 {
36                     _stack.Push(source);
37                     cursor = target;
38                 }
39             }
40             var left = cursor;
41             var right = appendant;
42             while (!_equalityComparer.Equals(cursor = _stack.Pop(), Links.Constants.Null))
43             {
44                 right = Links.GetOrCreate(left, right);
45                 left = cursor;
46             }
47             return Links.GetOrCreate(left, right);
48         }
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>>>>
        ↳ _duplicateFragmentsProvider;
12         public DuplicateSegmentsCounter(IProvider<IList<KeyValuePair<IList<TLink>,
        ↳ IList<TLink>>>> duplicateFragmentsProvider) => _duplicateFragmentsProvider =
        ↳ duplicateFragmentsProvider;
13         public int Count() => _duplicateFragmentsProvider.Get().Sum(x => x.Value.Count);
14     }
15 }

```

./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.Doublets.Unicode;
12
13 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
14
15 namespace Platform.Data.Doublets.Sequences
16 {
17     public class DuplicateSegmentsProvider<TLink> :
18         ↳ DictionaryBasedDuplicateSegmentsWalkerBase<TLink>,
19         ↳ IProvider<IList<KeyValuePair<IList<TLink>, IList<TLink>>>>
20     {
21         private readonly ILinks<TLink> _links;
22         private readonly ILinks<TLink> _sequences;
23         private HashSet<KeyValuePair<IList<TLink>, IList<TLink>>> _groups;
24         private BitString _visited;
25
26         private class ItemEquilityComparer : IEqualityComparer<KeyValuePair<IList<TLink>,
27             ↳ IList<TLink>>>
28         {
29             private readonly IListEqualityComparer<TLink> _listComparer;
30             public ItemEquilityComparer() => _listComparer =
31                 ↳ Default<IListEqualityComparer<TLink>>.Instance;
32             public bool Equals(KeyValuePair<IList<TLink>, IList<TLink>> left,
33                 ↳ KeyValuePair<IList<TLink>, IList<TLink>> right) =>
34                 ↳ _listComparer.Equals(left.Key, right.Key) && _listComparer.Equals(left.Value,
35                 ↳ right.Value);
36             public int GetHashCode(KeyValuePair<IList<TLink>, IList<TLink>> pair) =>
37                 ↳ (_listComparer.GetHashCode(pair.Key),
38                 ↳ _listComparer.GetHashCode(pair.Value)).GetHashCode();
39         }
40
41         private class ItemComparer : IComparer<KeyValuePair<IList<TLink>, IList<TLink>>>
42         {
43             private readonly IListComparer<TLink> _listComparer;
44
45             public ItemComparer() => _listComparer = Default<IListComparer<TLink>>.Instance;
46
47             public int Compare(KeyValuePair<IList<TLink>, IList<TLink>> left,
48                 ↳ KeyValuePair<IList<TLink>, IList<TLink>> right)
49             {
50                 var intermediateResult = _listComparer.Compare(left.Key, right.Key);
51                 if (intermediateResult == 0)
52                 {
53                     intermediateResult = _listComparer.Compare(left.Value, right.Value);
54                 }
55                 return intermediateResult;
56             }
57         }
58
59         public DuplicateSegmentsProvider(ILinks<TLink> links, ILinks<TLink> sequences)
60             : base(minimumStringSegmentLength: 2)
61         {
62             _links = links;
63             _sequences = sequences;
64         }
65
66         public IList<KeyValuePair<IList<TLink>, IList<TLink>>> Get()
67         {
68             _groups = new HashSet<KeyValuePair<IList<TLink>,
69                 ↳ IList<TLink>>>(Default<ItemEquilityComparer>.Instance);
70             var count = _links.Count();
71             _visited = new BitString((long)(Integer<TLink>)count + 1);
72             _links.Each(link =>
73             {
74                 var linkIndex = _links.GetIndex(link);
75                 var linkBitIndex = (long)(Integer<TLink>)linkIndex;
76                 if (!_visited.Get(linkBitIndex))
77                 {
78                     var sequenceElements = new List<TLink>();
79                 }
80             });
81         }
82     }
83 }
```

```

68         var filler = new ListFiller<TLink, TLink>(sequenceElements,
69             ↪ _sequences.Constants.Break);
70         _sequences.Each(filler.AddAllValuesAndReturnConstant, new
71             ↪ LinkAddress<TLink>(linkIndex));
72         if (sequenceElements.Count > 2)
73         {
74             WalkAll(sequenceElements);
75         }
76         return _links.Constants.Continue;
77     });
78     var resultList = _groups.ToList();
79     var comparer = Default<ItemComparer>.Instance;
80     resultList.Sort(comparer);
81     #if DEBUG
82     foreach (var item in resultList)
83     {
84         PrintDuplicates(item);
85     }
86     #endif
87     return resultList;
88 }
89 protected override Segment<TLink> CreateSegment(IList<TLink> elements, int offset, int
90     ↪ length) => new Segment<TLink>(elements, offset, length);
91 protected override void OnDuplicateFound(Segment<TLink> segment)
92 {
93     var duplicates = CollectDuplicatesForSegment(segment);
94     if (duplicates.Count > 1)
95     {
96         _groups.Add(new KeyValuePair<IList<TLink>, IList<TLink>>(segment.ToArray(),
97             ↪ duplicates));
98     }
99 }
100 private List<TLink> CollectDuplicatesForSegment(Segment<TLink> segment)
101 {
102     var duplicates = new List<TLink>();
103     var readAsElement = new HashSet<TLink>();
104     var restrictions = segment.ConvertToRestrictionsValues();
105     restrictions[0] = _sequences.Constants.Any;
106     _sequences.Each(sequence =>
107     {
108         var sequenceIndex = sequence[_sequences.Constants.IndexPart];
109         duplicates.Add(sequenceIndex);
110         readAsElement.Add(sequenceIndex);
111         return _sequences.Constants.Continue;
112     }, restrictions);
113     if (duplicates.Any(x => _visited.Get((Integer<TLink>)x)))
114     {
115         return new List<TLink>();
116     }
117     foreach (var duplicate in duplicates)
118     {
119         var duplicateBitIndex = (long)(Integer<TLink>)duplicate;
120         _visited.Set(duplicateBitIndex);
121     }
122     if (_sequences is Sequences sequencesExperiments)
123     {
124         var partiallyMatched = sequencesExperiments.GetAllPartiallyMatchingSequences4((H_
125             ↪ ashSet<ulong>)(object)readAsElement,
126             ↪ (IList<ulong>)segment);
127         foreach (var partiallyMatchedSequence in partiallyMatched)
128         {
129             TLink sequenceIndex = (Integer<TLink>)partiallyMatchedSequence;
130             duplicates.Add(sequenceIndex);
131         }
132     }
133     duplicates.Sort();
134     return duplicates;
135 }
136 private void PrintDuplicates(KeyValuePair<IList<TLink>, IList<TLink>> duplicatesItem)
137 {
138     if (!(_links is ILinks<ulong> ulongLinks))
139     {
140         return;

```

```

140     }
141     var duplicatesKey = duplicatesItem.Key;
142     var keyString = UnicodeMap.FromLinksToString((IList<ulong>)duplicatesKey);
143     Console.WriteLine($"{keyString} ({string.Join(", ", duplicatesKey)})");
144     var duplicatesList = duplicatesItem.Value;
145     for (int i = 0; i < duplicatesList.Count; i++)
146     {
147         ulong sequenceIndex = (Integer<TLink>)duplicatesList[i];
148         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));
149         Console.WriteLine(formattedSequenceStructure);
150         var sequenceString = UnicodeMap.FromSequenceLinkToString(sequenceIndex,
            ↳ ulongLinks);
151         Console.WriteLine(sequenceString);
152     }
153     Console.WriteLine();
154 }
155 }
156 }

```

./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         public LinkFrequency<TLink> GetFrequency(TLink source, TLink target)
35         {
36             var doublet = new Doublet<TLink>(source, target);
37             return GetFrequency(ref doublet);
38         }
39
40         [MethodImpl(MethodImplOptions.AggressiveInlining)]
41         public LinkFrequency<TLink> GetFrequency(ref Doublet<TLink> doublet)
42         {
43             _doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data);
44             return data;
45         }
46
47         public void IncrementFrequencies(IList<TLink> sequence)
48         {
49             for (var i = 1; i < sequence.Count; i++)
50             {
51                 IncrementFrequency(sequence[i - 1], sequence[i]);
52             }
53         }
54
55         [MethodImpl(MethodImplOptions.AggressiveInlining)]
56         public LinkFrequency<TLink> IncrementFrequency(TLink source, TLink target)

```



```

54     {
55         var doublet = new Doublet<TLink>(source, target);
56         return IncrementFrequency(ref doublet);
57     }
58
59     public void PrintFrequencies(IList<TLink> sequence)
60     {
61         for (var i = 1; i < sequence.Count; i++)
62         {
63             PrintFrequency(sequence[i - 1], sequence[i]);
64         }
65     }
66
67     public void PrintFrequency(TLink source, TLink target)
68     {
69         var number = GetFrequency(source, target).Frequency;
70         Console.WriteLine("{0},{1} - {2}", source, target, number);
71     }
72
73     [MethodImpl(MethodImplOptions.AggressiveInlining)]
74     public LinkFrequency<TLink> IncrementFrequency(ref Doublet<TLink> doublet)
75     {
76         if (_doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data))
77         {
78             data.IncrementFrequency();
79         }
80         else
81         {
82             var link = Links.SearchOrDefault(doublet.Source, doublet.Target);
83             data = new LinkFrequency<TLink>(Integer<TLink>.One, link);
84             if (!_equalityComparer.Equals(link, default))
85             {
86                 data.Frequency = Arithmetic.Add(data.Frequency,
87                     ↪ _frequencyCounter.Count(link));
88             }
89             _doubletsCache.Add(doublet, data);
90         }
91         return data;
92     }
93
94     public void ValidateFrequencies()
95     {
96         foreach (var entry in _doubletsCache)
97         {
98             var value = entry.Value;
99             var linkIndex = value.Link;
100             if (!_equalityComparer.Equals(linkIndex, default))
101             {
102                 var frequency = value.Frequency;
103                 var count = _frequencyCounter.Count(linkIndex);
104                 // TODO: Why `frequency` always greater than `count` by 1?
105                 if (((_comparer.Compare(frequency, count) > 0) &&
106                     ↪ (_comparer.Compare(Arithmetic.Subtract(frequency, count),
107                     ↪ Integer<TLink>.One) > 0))
108                     || ((_comparer.Compare(count, frequency) > 0) &&
109                     ↪ (_comparer.Compare(Arithmetic.Subtract(count, frequency),
110                     ↪ Integer<TLink>.One) > 0)))
111                 {
112                     throw new InvalidOperationException("Frequencies validation failed.");
113                 }
114             }
115             //else
116             //{
117                 if (value.Frequency > 0)
118                 {
119                     var frequency = value.Frequency;
120                     linkIndex = _createLink(entry.Key.Source, entry.Key.Target);
121                     var count = _countLinkFrequency(linkIndex);
122
123                     if ((frequency > count && frequency - count > 1) || (count > frequency
124                     ↪ && count - frequency > 1))
125                         throw new Exception("Frequencies validation failed.");
126                 }
127             }
128         }
129     }
130 }

```

./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> :
8         ⇨ IConverter<Doublet<TLink>, TLink>
9     {
10         private readonly LinkFrequenciesCache<TLink> _cache;
11         public
12         ⇨ FrequenciesCacheBasedLinkToItsFrequencyNumberConverter(LinkFrequenciesCache<TLink>
13         ⇨ cache) => _cache = cache;
14         public TLink Convert(Doublet<TLink> source) => _cache.GetFrequency(ref source).Frequency;
15     }
16 }
```

./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.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.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 symbol)
39         {
40             if (_visits.Contains(symbol))
41                 return;
42             _visits.Add(symbol);
43             TLink count = _equalityComparer.Equals(symbol, _symbol) ? _total : default;
44             count = Arithmetic.Add(count, 1);
45             _total = _comparer.Compare(count, _total) > 0 ? count : _total;
46         }
47     }
48 }

```

```

35     }
36
37     private void CountCore(TLink link)
38     {
39         var any = _links.Constants.Any;
40         if (_equalityComparer.Equals(_links.Count(any, link), default))
41         {
42             CountSequenceSymbolFrequency(link);
43         }
44         else
45         {
46             _links.Each(EachElementHandler, any, link);
47         }
48     }
49
50     protected virtual void CountSequenceSymbolFrequency(TLink link)
51     {
52         var symbolFrequencyCounter = new SequenceSymbolFrequencyOneOffCounter<TLink>(_links,
53             ↪ link, _symbol);
54         _total = Arithmetic.Add(_total, symbolFrequencyCounter.Count());
55     }
56
57     private TLink EachElementHandler(IList<TLink> doublet)
58     {
59         var constants = _links.Constants;
60         var doubletIndex = doublet[constants.IndexPart];
61         if (_visits.Add(doubletIndex))
62         {
63             CountCore(doubletIndex);
64         }
65         return constants.Continue;
66     }
67 }

```

./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;
19
20         public CachedSequenceHeightProvider(
21             ILinks<TLink> links,
22             ISequenceHeightProvider<TLink> baseHeightProvider,
23             IConverter<TLink> addressToUnaryNumberConverter,
24             IConverter<TLink> unaryNumberToAddressConverter,
25             TLink heightPropertyMarker,
26             IPropertiesOperator<TLink, TLink, TLink> propertyOperator)
27             : base(links)
28         {
29             _heightPropertyMarker = heightPropertyMarker;
30             _baseHeightProvider = baseHeightProvider;
31             _addressToUnaryNumberConverter = addressToUnaryNumberConverter;
32             _unaryNumberToAddressConverter = unaryNumberToAddressConverter;
33             _propertyOperator = propertyOperator;
34         }
35
36         public TLink Get(TLink sequence)
37         {
38             TLink height;
39             var heightValue = _propertyOperator.GetValue(sequence, _heightPropertyMarker);
40             if (_equalityComparer.Equals(heightValue, default))
41             {
42                 height = _baseHeightProvider.Get(sequence);
43                 heightValue = _addressToUnaryNumberConverter.Convert(height);
44                 _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/IListExtensions.cs

```

1  using Platform.Collections;
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
7  {
8      public static class IListExtensions
9      {
10         public static TLink[] ExtractValues<TLink>(this IList<TLink> restrictions)
11         {
12             if(restrictions.IsNullOrEmpty() || restrictions.Count == 1)
13             {
14                 return new TLink[0];
15             }
16             var values = new TLink[restrictions.Count - 1];
17             for (int i = 1, j = 0; i < restrictions.Count; i++, j++)
18             {
19                 values[j] = restrictions[i];
20             }
21             return values;
22         }
23
24         public static IList<TLink> ConvertToRestrictionsValues<TLink>(this IList<TLink> list)
25         {
26             var restrictions = new TLink[list.Count + 1];

```

```

27         for (int i = 0, j = 1; i < list.Count; i++, j++)
28         {
29             restrictions[j] = list[i];
30         }
31         return restrictions;
32     }
33 }
34 }

```

./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(ICollection<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(ICollection<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        public FrequencyIncrementingSequenceIndex(ILinks<TLink> links, IPropertyOperator<TLink,
18            ↳ TLink> frequencyPropertyOperator, IIncrementer<TLink> frequencyIncrementer)
19            : base(links)
20        {
21            _frequencyPropertyOperator = frequencyPropertyOperator;
22            _frequencyIncrementer = frequencyIncrementer;
23        }
24
25        public override bool Add(IList<TLink> sequence)
26        {
27            var indexed = true;
28            var i = sequence.Count;
29            while (--i >= 1 && (indexed = IsIndexedWithIncrement(sequence[i - 1], sequence[i])))
30                ↳ { }
31            for (; i >= 1; i--)
32            {
33                Increment(Links.GetOrCreate(sequence[i - 1], sequence[i]));
34            }
35            return indexed;
36        }
37
38        private bool IsIndexedWithIncrement(TLink source, TLink target)
39        {
40            var link = Links.SearchOrCreate(source, target);
41            var indexed = !_equalityComparer.Equals(link, default);
42            if (indexed)
43            {
44                Increment(link);
45            }
46            return indexed;
47        }
48
49        private void Increment(TLink link)
50        {
51            var previousFrequency = _frequencyPropertyOperator.Get(link);
52            var frequency = _frequencyIncrementer.Increment(previousFrequency);
53            _frequencyPropertyOperator.Set(link, frequency);
54        }
55    }
56 }

```

./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;
18             while (--i >= 1 && (indexed =
19                 ↳ !_equalityComparer.Equals(Links.SearchOrDefault(sequence[i - 1], sequence[i]),
20                 ↳ default))) { }
21             for (; i >= 1; i--)
22             {
23                 Links.GetOrCreate(sequence[i - 1], sequence[i]);
24             }
25             return indexed;
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;

```

```

43         var i = sequence.Count;
44         while (--i >= 1 && (indexed =
            ↳ !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1],
            ↳ sequence[i]), default))) { }
45         return indexed;
46     });
47 }
48 }
49 }

```

./Platform.Data.Doublets/Sequences/ListFiller.cs

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Sequences
7  {
8      public class ListFiller<TElement, TReturnConstant>
9      {
10         protected readonly List<TElement> _list;
11         protected readonly TReturnConstant _returnConstant;
12
13         public ListFiller(List<TElement> list, TReturnConstant returnConstant)
14         {
15             _list = list;
16             _returnConstant = returnConstant;
17         }
18
19         public ListFiller(List<TElement> list) : this(list, default) { }
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public void Add(TElement element) => _list.Add(element);
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         public bool AddAndReturnTrue(TElement element)
26         {
27             _list.Add(element);
28             return true;
29         }
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         public bool AddFirstAndReturnTrue(IList<TElement> collection)
33         {
34             _list.Add(collection[0]);
35             return true;
36         }
37
38         [MethodImpl(MethodImplOptions.AggressiveInlining)]
39         public TReturnConstant AddAndReturnConstant(TElement element)
40         {
41             _list.Add(element);
42             return _returnConstant;
43         }
44
45         [MethodImpl(MethodImplOptions.AggressiveInlining)]
46         public TReturnConstant AddFirstAndReturnConstant(IList<TElement> collection)
47         {
48             _list.Add(collection[0]);
49             return _returnConstant;
50         }
51
52         [MethodImpl(MethodImplOptions.AggressiveInlining)]
53         public TReturnConstant AddAllValuesAndReturnConstant(IList<TElement> collection)
54         {
55             for (int i = 1; i < collection.Count; i++)
56             {
57                 _list.Add(collection[i]);
58             }
59             return _returnConstant;
60         }
61     }
62 }

```

./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.Doublets.Sequences.Walkers;
11 using Platform.Collections.Stacks;
12 using Platform.Collections.Arrays;
13
14 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
15
16 namespace Platform.Data.Doublets.Sequences
17 {
18     /// <summary>
19     /// Представляет коллекцию последовательностей связей.
20     /// </summary>
21     /// <remarks>
22     /// Обязательно реализовать атомарность каждого публичного метода.
23     ///
24     /// TODO:
25     ///
26     /// !!! Повышение вероятности повторного использования групп (подпоследовательностей),
27     /// через естественную группировку по unicode типам, все whitespace вместе, все символы
28     /// → вместе, все числа вместе и т.п.
29     /// + использовать ровно сбалансированный вариант, чтобы уменьшать вложенность (глубину
30     /// → графа)
31     ///
32     /// х*у - найти все связи между, в последовательностях любой формы, если не стоит
33     /// → ограничитель на то, что является последовательностью, а что нет,
34     /// то находятся любые структуры связей, которые содержат эти элементы именно в таком
35     /// → порядке.
36     ///
37     /// Рост последовательности слева и справа.
38     /// Поиск со звёздочкой.
39     /// URL, PURL - реестр используемых во вне ссылок на ресурсы,
40     /// так же проблема может быть решена при реализации дистанционных триггеров.
41     /// Нужны ли уникальные указатели вообще?
42     /// Что если обращение к информации будет происходить через содержимое всегда?
43     ///
44     /// Писать тесты.
45     ///
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     ///
54     /// Блокчейн и/или гит для распределённой записи транзакций.
55     ///
56     /// </remarks>
57     public partial class Sequences : ILinks<LinkIndex> // IList<string>, IList<LinkIndex[]>
58     {
59         (после завершения реализации Sequences)
60
61         {
62             /// <summary>Возвращает значение LinkIndex, обозначающее любое количество
63             /// → связей.</summary>
64             public const LinkIndex ZeroOrMany = LinkIndex.MaxValue;
65
66             public SequencesOptions<LinkIndex> Options { get; }
67             public SynchronizedLinks<LinkIndex> Links { get; }
68             private readonly ISynchronization _sync;
69
70             public LinksConstants<LinkIndex> Constants { get; }
71
72             public Sequences(SynchronizedLinks<LinkIndex> links, SequencesOptions<LinkIndex> options)
73             {
74                 Links = links;
75                 _sync = links.SyncRoot;
76                 Options = options;
77                 Options.ValidateOptions();
78                 Options.InitOptions(Links);
79                 Constants = Default<LinksConstants<LinkIndex>>.Instance;
80             }
81
82             public Sequences(SynchronizedLinks<LinkIndex> links)
83             : this(links, new SequencesOptions<LinkIndex>())
84             {
85             }
86         }
87     }
88 }

```

```

78 public bool IsSequence(LinkIndex sequence)
79 {
80     return _sync.ExecuteReadOperation(() =>
81     {
82         if (Options.UseSequenceMarker)
83         {
84             return Options.MarkedSequenceMatcher.IsMatched(sequence);
85         }
86         return !Links.Unsync.IsPartialPoint(sequence);
87     });
88 }
89
90 [MethodImpl(MethodImplOptions.AggressiveInlining)]
91 private LinkIndex GetSequenceByElements(LinkIndex sequence)
92 {
93     if (Options.UseSequenceMarker)
94     {
95         return Links.SearchOrDefault(Options.SequenceMarkerLink, sequence);
96     }
97     return sequence;
98 }
99
100 private LinkIndex GetSequenceElements(LinkIndex sequence)
101 {
102     if (Options.UseSequenceMarker)
103     {
104         var linkContents = new UInt64Link(Links.GetLink(sequence));
105         if (linkContents.Source == Options.SequenceMarkerLink)
106         {
107             return linkContents.Target;
108         }
109         if (linkContents.Target == Options.SequenceMarkerLink)
110         {
111             return linkContents.Source;
112         }
113     }
114     return sequence;
115 }
116
117 #region Count
118
119 public LinkIndex Count(IList<LinkIndex> restrictions)
120 {
121     if (restrictions.IsNullOrEmpty())
122     {
123         return Links.Count(Constants.Any, Options.SequenceMarkerLink, Constants.Any);
124     }
125     if (restrictions.Count == 1) // Первая связь это адрес
126     {
127         var sequenceIndex = restrictions[0];
128         if (sequenceIndex == Constants.Null)
129         {
130             return 0;
131         }
132         if (sequenceIndex == Constants.Any)
133         {
134             return Count(null);
135         }
136         if (Options.UseSequenceMarker)
137         {
138             return Links.Count(Constants.Any, Options.SequenceMarkerLink, sequenceIndex);
139         }
140         return Links.Exists(sequenceIndex) ? 1UL : 0;
141     }
142     throw new NotImplementedException();
143 }
144
145 private LinkIndex CountUsages(params LinkIndex[] 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 LinkIndex Create(IList<LinkIndex> restrictions)
177 {
178     return _sync.ExecuteWriteOperation(() =>
179     {
180         if (restrictions.IsNullOrEmpty())
181         {
182             return Constants.Null;
183         }
184         Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
185         return CreateCore(restrictions);
186     });
187 }
188
189 private LinkIndex CreateCore(IList<LinkIndex> restrictions)
190 {
191     LinkIndex[] sequence = restrictions.ExtractValues();
192     if (Options.UseIndex)
193     {
194         Options.Index.Add(sequence);
195     }
196     var sequenceRoot = default(LinkIndex);
197     if (Options.EnforceSingleSequenceVersionOnWriteBasedOnExisting)
198     {
199         var matches = Each(restrictions);
200         if (matches.Count > 0)
201         {
202             sequenceRoot = matches[0];
203         }
204     }
205     else if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew)
206     {
207         return CompactCore(sequence);
208     }
209     if (sequenceRoot == default)
210     {
211         sequenceRoot = Options.LinksToSequenceConverter.Convert(sequence);
212     }
213     if (Options.UseSequenceMarker)
214     {
215         Links.Unsync.CreateAndUpdate(Options.SequenceMarkerLink, sequenceRoot);
216     }
217     return sequenceRoot; // Возвращаем корень последовательности (т.е. сами элементы)
218 }
219
220 #endregion
221
222 #region Each
223
224 public List<LinkIndex> Each(IList<LinkIndex> sequence)
225 {
226     var results = new List<LinkIndex>();
227     var filler = new ListFiller<LinkIndex, LinkIndex>(results, Constants.Continue);
228     Each(filler.AddFirstAndReturnConstant, sequence);
229     return results;
230 }
231
232 public LinkIndex Each(Func<IList<LinkIndex>, LinkIndex> handler, IList<LinkIndex>
233     ↪ restrictions)
234 {
235     return _sync.ExecuteReadOperation(() =>

```

```

235 {
236     if (restrictions.IsNullOrEmpty())
237     {
238         return Constants.Continue;
239     }
240     Links.EnsureInnerReferenceExists(restrictions, nameof(restrictions));
241     if (restrictions.Count == 1)
242     {
243         var link = restrictions[0];
244         var any = Constants.Any;
245         if (link == any)
246         {
247             if (Options.UseSequenceMarker)
248             {
249                 return Links.Unsync.Each(handler, new Link<LinkIndex>(any,
250                     ↪ Options.SequenceMarkerLink, any));
251             }
252             else
253             {
254                 return Links.Unsync.Each(handler, new Link<LinkIndex>(any, any,
255                     ↪ any));
256             }
257             var sequence =
258                 ↪ Options.Walker.Walk(link).ToArray().ConvertToRestrictionsValues();
259             sequence[0] = link;
260             return handler(sequence);
261         }
262         else if (restrictions.Count == 2)
263         {
264             throw new NotImplementedException();
265         }
266         else if (restrictions.Count == 3)
267         {
268             return Links.Unsync.Each(handler, restrictions);
269         }
270         else
271         {
272             var sequence = restrictions.ExtractValues();
273             if (Options.UseIndex && !Options.Index.MightContain(sequence))
274             {
275                 return Constants.Break;
276             }
277             return EachCore(handler, sequence);
278         }
279     }
280 }
281
282 private LinkIndex EachCore(Func<IList<LinkIndex>, LinkIndex> handler, IList<LinkIndex>
283     ↪ values)
284 {
285     var matcher = new Matcher(this, values, new HashSet<LinkIndex>(), handler);
286     // TODO: Find out why matcher.HandleFullMatched executed twice for the same sequence
287     ↪ Id.
288     Func<IList<LinkIndex>, LinkIndex> innerHandler = Options.UseSequenceMarker ?
289     ↪ (Func<IList<LinkIndex>, LinkIndex>)matcher.HandleFullMatchedSequence :
290     ↪ matcher.HandleFullMatched;
291     //if (sequence.Length >= 2)
292     if (StepRight(innerHandler, values[0], values[1]) != Constants.Continue)
293     {
294         return Constants.Break;
295     }
296     var last = values.Count - 2;
297     for (var i = 1; i < last; i++)
298     {
299         if (PartialStepRight(innerHandler, values[i], values[i + 1]) !=
300             ↪ Constants.Continue)
301         {
302             return Constants.Break;
303         }
304     }
305     if (values.Count >= 3)
306     {
307         if (StepLeft(innerHandler, values[values.Count - 2], values[values.Count - 1])
308             ↪ != Constants.Continue)
309         {
310             return Constants.Break;
311         }
312     }
313 }

```

```

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

```

```

374     if (sequence.IsNullOrEmpty())
375     {
376         return Create(substitution);
377     }
378     if (newSequence.IsNullOrEmpty())
379     {
380         Delete(restrictions);
381         return Constants.Null;
382     }
383     return _sync.ExecuteWriteOperation(() =>
384     {
385         Links.EnsureEachLinkIsAnyOrExists(sequence);
386         Links.EnsureEachLinkExists(newSequence);
387         return UpdateCore(sequence, newSequence);
388     });
389 }
390
391 private LinkIndex UpdateCore(LinkIndex[] sequence, LinkIndex[] newSequence)
392 {
393     LinkIndex bestVariant;
394     if (Options.EnforceSingleSequenceVersionOnWriteBasedOnNew &&
395         ↪ !sequence.EqualTo(newSequence))
396     {
397         bestVariant = CompactCore(newSequence);
398     }
399     else
400     {
401         bestVariant = CreateCore(newSequence);
402     }
403     // TODO: Check all options only ones before loop execution
404     // Возможно нужно две версии Each, возвращающий фактические последовательности и с
405     ↪ маркером,
406     // или возможно даже возвращать и тот и тот вариант. С другой стороны все варианты
407     ↪ можно получить имея только фактические последовательности.
408     foreach (var variant in Each(sequence))
409     {
410         if (variant != bestVariant)
411         {
412             UpdateOneCore(variant, bestVariant);
413         }
414     }
415     return bestVariant;
416 }
417
418 private void UpdateOneCore(LinkIndex sequence, LinkIndex newSequence)
419 {
420     if (Options.UseGarbageCollection)
421     {
422         var sequenceElements = GetSequenceElements(sequence);
423         var sequenceElementsContents = new UInt64Link(Links.GetLink(sequenceElements));
424         var sequenceLink = GetSequenceByElements(sequenceElements);
425         var newSequenceElements = GetSequenceElements(newSequence);
426         var newSequenceLink = GetSequenceByElements(newSequenceElements);
427         if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
428         {
429             if (sequenceLink != Constants.Null)
430             {
431                 Links.Unsync.MergeUsages(sequenceLink, newSequenceLink);
432             }
433             Links.Unsync.MergeUsages(sequenceElements, newSequenceElements);
434         }
435         ClearGarbage(sequenceElementsContents.Source);
436         ClearGarbage(sequenceElementsContents.Target);
437     }
438     else
439     {
440         if (Options.UseSequenceMarker)
441         {
442             var sequenceElements = GetSequenceElements(sequence);
443             var sequenceLink = GetSequenceByElements(sequenceElements);
444             var newSequenceElements = GetSequenceElements(newSequence);
445             var newSequenceLink = GetSequenceByElements(newSequenceElements);
446             if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
447             {
448                 if (sequenceLink != Constants.Null)
449                 {
450                     Links.Unsync.MergeUsages(sequenceLink, newSequenceLink);
451                 }
452             }
453         }
454     }
455 }

```



```

449         Links.Unsync.MergeUsages(sequenceElements, newSequenceElements);
450     }
451 }
452 else
453 {
454     if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
455     {
456         Links.Unsync.MergeUsages(sequence, newSequence);
457     }
458 }
459 }
460 }
461
462 #endregion
463
464 #region Delete
465
466 public void Delete(ICollection<LinkIndex> restrictions)
467 {
468     _sync.ExecuteWriteOperation(() =>
469     {
470         var sequence = restrictions.ExtractValues();
471         // TODO: Check all options only ones before loop execution
472         foreach (var linkToDelete in Each(sequence))
473         {
474             DeleteOneCore(linkToDelete);
475         }
476     });
477 }
478
479 private void DeleteOneCore(LinkIndex link)
480 {
481     if (Options.UseGarbageCollection)
482     {
483         var sequenceElements = GetSequenceElements(link);
484         var sequenceElementsContents = new UInt64Link(Links.GetLink(sequenceElements));
485         var sequenceLink = GetSequenceByElements(sequenceElements);
486         if (Options.UseCascadeDelete || CountUsages(link) == 0)
487         {
488             if (sequenceLink != Constants.Null)
489             {
490                 Links.Unsync.Delete(sequenceLink);
491             }
492             Links.Unsync.Delete(link);
493         }
494         ClearGarbage(sequenceElementsContents.Source);
495         ClearGarbage(sequenceElementsContents.Target);
496     }
497     else
498     {
499         if (Options.UseSequenceMarker)
500         {
501             var sequenceElements = GetSequenceElements(link);
502             var sequenceLink = GetSequenceByElements(sequenceElements);
503             if (Options.UseCascadeDelete || CountUsages(link) == 0)
504             {
505                 if (sequenceLink != Constants.Null)
506                 {
507                     Links.Unsync.Delete(sequenceLink);
508                 }
509                 Links.Unsync.Delete(link);
510             }
511         }
512         else
513         {
514             if (Options.UseCascadeDelete || CountUsages(link) == 0)
515             {
516                 Links.Unsync.Delete(link);
517             }
518         }
519     }
520 }
521
522 #endregion
523
524 #region Compactification
525
526 /// <remarks>
527 /// bestVariant можно выбирать по максимальному числу использований,

```

```

528 /// но балансированный позволяет гарантировать уникальность (если есть возможность,
529 /// гарантировать его использование в других местах).
530 ///
531 /// Получается этот метод должен игнорировать Options.EnforceSingleSequenceVersionOnWrite
532 /// </remarks>
533 public LinkIndex Compact(params LinkIndex[] sequence)
534 {
535     return _sync.ExecuteWriteOperation(() =>
536     {
537         if (sequence.IsNullOrEmpty())
538         {
539             return Constants.Null;
540         }
541         Links.EnsureEachLinkExists(sequence);
542         return CompactCore(sequence);
543     });
544 }
545
546 [MethodImpl(MethodImplOptions.AggressiveInlining)]
547 private LinkIndex CompactCore(params LinkIndex[] sequence) => UpdateCore(sequence,
548     ↪ sequence);
549
550 #endregion
551 #region Garbage Collection
552
553 /// <remarks>
554 /// TODO: Добавить дополнительный обработчик / событие CanBeDeleted которое можно
555     ↪ определить извне или в унаследованном классе
556 /// </remarks>
557 [MethodImpl(MethodImplOptions.AggressiveInlining)]
558 private bool IsGarbage(LinkIndex link) => link != Options.SequenceMarkerLink &&
559     ↪ !Links.Unsync.IsPartialPoint(link) && Links.Count(link) == 0;
560
561 private void ClearGarbage(LinkIndex link)
562 {
563     if (IsGarbage(link))
564     {
565         var contents = new UInt64Link(Links.GetLink(link));
566         Links.Unsync.Delete(link);
567         ClearGarbage(contents.Source);
568         ClearGarbage(contents.Target);
569     }
570 }
571
572 #endregion
573 #region Walkers
574
575 public bool EachPart(Func<LinkIndex, bool> handler, LinkIndex sequence)
576 {
577     return _sync.ExecuteReadOperation(() =>
578     {
579         var links = Links.Unsync;
580         foreach (var part in Options.Walker.Walk(sequence))
581         {
582             if (!handler(part))
583             {
584                 return false;
585             }
586         }
587         return true;
588     });
589 }
590
591 public class Matcher : RightSequenceWalker<LinkIndex>
592 {
593     private readonly Sequences _sequences;
594     private readonly IList<LinkIndex> _patternSequence;
595     private readonly HashSet<LinkIndex> _linksInSequence;
596     private readonly HashSet<LinkIndex> _results;
597     private readonly Func<IList<LinkIndex>, LinkIndex> _stopableHandler;
598     private readonly HashSet<LinkIndex> _readAsElements;
599     private int _filterPosition;
600
601     public Matcher(Sequences sequences, IList<LinkIndex> patternSequence,
602         ↪ HashSet<LinkIndex> results, Func<IList<LinkIndex>, LinkIndex> stopableHandler,
603         ↪ HashSet<LinkIndex> readAsElements = null)
604         : base(sequences.Links.Unsync, new DefaultStack<LinkIndex>())
605     {
606     }

```

```

603     _sequences = sequences;
604     _patternSequence = patternSequence;
605     _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
        ↳ Links.Constants.Any && x != ZeroOrMany));
606     _results = results;
607     _stopableHandler = stopableHandler;
608     _readAsElements = readAsElements;
609 }
610
611 protected override bool IsElement(LinkIndex link) => base.IsElement(link) ||
        ↳ (_readAsElements != null && _readAsElements.Contains(link)) ||
        ↳ _linksInSequence.Contains(link);
612
613 public bool FullMatch(LinkIndex sequenceToMatch)
614 {
615     _filterPosition = 0;
616     foreach (var part in Walk(sequenceToMatch))
617     {
618         if (!FullMatchCore(part))
619         {
620             break;
621         }
622     }
623     return _filterPosition == _patternSequence.Count;
624 }
625
626 private bool FullMatchCore(LinkIndex element)
627 {
628     if (_filterPosition == _patternSequence.Count)
629     {
630         _filterPosition = -2; // Длиннее чем нужно
631         return false;
632     }
633     if (_patternSequence[_filterPosition] != Links.Constants.Any
        && element != _patternSequence[_filterPosition])
634     {
635         _filterPosition = -1;
636         return false; // Начинается/Продолжается иначе
637     }
638     _filterPosition++;
639     return true;
640 }
641
642 public void AddFullMatchedToResults(IList<LinkIndex> restrictions)
643 {
644     var sequenceToMatch = restrictions[Links.Constants.IndexPart];
645     if (FullMatch(sequenceToMatch))
646     {
647         _results.Add(sequenceToMatch);
648     }
649 }
650
651 public LinkIndex HandleFullMatched(IList<LinkIndex> restrictions)
652 {
653     var sequenceToMatch = restrictions[Links.Constants.IndexPart];
654     if (FullMatch(sequenceToMatch) && _results.Add(sequenceToMatch))
655     {
656         return _stopableHandler(new LinkAddress<LinkIndex>(sequenceToMatch));
657     }
658     return Links.Constants.Continue;
659 }
660
661 public LinkIndex HandleFullMatchedSequence(IList<LinkIndex> restrictions)
662 {
663     var sequenceToMatch = restrictions[Links.Constants.IndexPart];
664     var sequence = _sequences.GetSequenceByElements(sequenceToMatch);
665     if (sequence != Links.Constants.Null && FullMatch(sequenceToMatch) &&
        ↳ _results.Add(sequenceToMatch))
666     {
667         return _stopableHandler(new LinkAddress<LinkIndex>(sequence));
668     }
669     return Links.Constants.Continue;
670 }
671
672 /// <remarks>
673 /// TODO: Add support for LinksConstants.Any
674 /// </remarks>
675 public bool PartialMatch(LinkIndex sequenceToMatch)
676 {
677

```

```

678     _filterPosition = -1;
679     foreach (var part in Walk(sequenceToMatch))
680     {
681         if (!PartialMatchCore(part))
682         {
683             break;
684         }
685     }
686     return _filterPosition == _patternSequence.Count - 1;
687 }
688
689 private bool PartialMatchCore(LinkIndex element)
690 {
691     if (_filterPosition == (_patternSequence.Count - 1))
692     {
693         return false; // Нашлось
694     }
695     if (_filterPosition >= 0)
696     {
697         if (element == _patternSequence[_filterPosition + 1])
698         {
699             _filterPosition++;
700         }
701         else
702         {
703             _filterPosition = -1;
704         }
705     }
706     if (_filterPosition < 0)
707     {
708         if (element == _patternSequence[0])
709         {
710             _filterPosition = 0;
711         }
712     }
713     return true; // Ищем дальше
714 }
715
716 public void AddPartialMatchedToResults(LinkIndex sequenceToMatch)
717 {
718     if (PartialMatch(sequenceToMatch))
719     {
720         _results.Add(sequenceToMatch);
721     }
722 }
723
724 public LinkIndex HandlePartialMatched(ICollection<LinkIndex> restrictions)
725 {
726     var sequenceToMatch = restrictions[Links.Constants.IndexPart];
727     if (PartialMatch(sequenceToMatch))
728     {
729         return _stopableHandler(new LinkAddress<LinkIndex>(sequenceToMatch));
730     }
731     return Links.Constants.Continue;
732 }
733
734 public void AddAllPartialMatchedToResults(IEnumerable<LinkIndex> sequencesToMatch)
735 {
736     foreach (var sequenceToMatch in sequencesToMatch)
737     {
738         if (PartialMatch(sequenceToMatch))
739         {
740             _results.Add(sequenceToMatch);
741         }
742     }
743 }
744
745 public void AddAllPartialMatchedToResultsAndReadAsElements(IEnumerable<LinkIndex>
746 ↪ sequencesToMatch)
747 {
748     foreach (var sequenceToMatch in sequencesToMatch)
749     {
750         if (PartialMatch(sequenceToMatch))
751         {
752             _readAsElements.Add(sequenceToMatch);
753             _results.Add(sequenceToMatch);
754         }
755     }
756 }

```



```

71         {
72             throw new NotImplementedException("Creation cancellation is not
73                 ↳ implemented.");
74         }
75         variants[last++] = variant;
76     }
77 }
78 return variants;
79 }
80
81 public List<ulong> CreateAllVariants1(params ulong[] sequence)
82 {
83     return _sync.ExecuteWriteOperation(() =>
84     {
85         if (sequence.IsNullOrEmpty())
86         {
87             return new List<ulong>();
88         }
89         Links.Unsync.EnsureEachLinkExists(sequence);
90         if (sequence.Length == 1)
91         {
92             return new List<ulong> { sequence[0] };
93         }
94         var results = new
95             ↳ List<ulong>((int)Platform.Numbers.Math.Catalan(sequence.Length));
96         return CreateAllVariants1Core(sequence, results);
97     });
98 }
99 private List<ulong> CreateAllVariants1Core(ulong[] sequence, List<ulong> results)
100 {
101     if (sequence.Length == 2)
102     {
103         var link = Links.Unsync.CreateAndUpdate(sequence[0], sequence[1]);
104         if (link == Constants.Null)
105         {
106             throw new NotImplementedException("Creation cancellation is not
107                 ↳ implemented.");
108         }
109         results.Add(link);
110         return results;
111     }
112     var innerSequenceLength = sequence.Length - 1;
113     var innerSequence = new ulong[innerSequenceLength];
114     for (var li = 0; li < innerSequenceLength; li++)
115     {
116         var link = Links.Unsync.CreateAndUpdate(sequence[li], sequence[li + 1]);
117         if (link == Constants.Null)
118         {
119             throw new NotImplementedException("Creation cancellation is not
120                 ↳ implemented.");
121         }
122         for (var isi = 0; isi < li; isi++)
123         {
124             innerSequence[isi] = sequence[isi];
125         }
126         innerSequence[li] = link;
127         for (var isi = li + 1; isi < innerSequenceLength; isi++)
128         {
129             innerSequence[isi] = sequence[isi + 1];
130         }
131         CreateAllVariants1Core(innerSequence, results);
132     }
133     return results;
134 }
135
136 #endregion
137
138 public HashSet<ulong> Each1(params ulong[] sequence)
139 {
140     var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
141     Each1(link =>
142     {
143         if (!visitedLinks.Contains(link))
144         {
145             visitedLinks.Add(link); // изучить почему случаются повторы
146         }
147     });
148 }

```

```

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(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[Constants.IndexPart];
184                 Each1(handler, innerSequence);
185                 return Constants.Continue;
186             }, Constants.Any, left, right);
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         var linkIndex = link[Constants.IndexPart];
197         if (!visitedLinks.Contains(linkIndex))
198         {
199             visitedLinks.Add(linkIndex); // изучить почему случаются повторы
200         }
201         return Constants.Continue;
202     }, sequence);
203     return visitedLinks;
204 }
205
206 public void EachPart(Func<IList<LinkIndex>, LinkIndex> handler, params ulong[] sequence)
207 {
208     var visitedLinks = new HashSet<ulong>(); // Заменить на bitstring
209     EachPartCore(link =>
210     {
211         var linkIndex = link[Constants.IndexPart];
212         if (!visitedLinks.Contains(linkIndex))
213         {
214             visitedLinks.Add(linkIndex); // изучить почему случаются повторы
215             return handler(new LinkAddress<LinkIndex>(linkIndex));
216         }
217         return Constants.Continue;
218     }, sequence);
219 }
220
221 private void EachPartCore(Func<IList<LinkIndex>, LinkIndex> handler, params ulong[]
222     ↪ sequence)
223 {

```

```

223     if (sequence.IsNullOrEmpty())
224     {
225         return;
226     }
227     Links.EnsureEachLinkIsAnyOrExists(sequence);
228     if (sequence.Length == 1)
229     {
230         var link = sequence[0];
231         if (link > 0)
232         {
233             handler(new LinkAddress<LinkIndex>(link));
234         }
235         else
236         {
237             Links.Each(Constants.Any, Constants.Any, handler);
238         }
239     }
240     else if (sequence.Length == 2)
241     {
242         //_links.Each(sequence[0], sequence[1], handler);
243         //  o_|      x_o ...
244         // x_|      |___|
245         Links.Each(sequence[1], Constants.Any, doublet =>
246         {
247             var match = Links.SearchOrDefault(sequence[0], doublet);
248             if (match != Constants.Null)
249             {
250                 handler(new LinkAddress<LinkIndex>(match));
251             }
252             return true;
253         });
254         // |_x      ... x_o
255         // |_o      |___|
256         Links.Each(Constants.Any, sequence[0], doublet =>
257         {
258             var match = Links.SearchOrDefault(doublet, sequence[1]);
259             if (match != 0)
260             {
261                 handler(new LinkAddress<LinkIndex>(match));
262             }
263             return true;
264         });
265         //      ..x o_.
266         //      |___|
267         PartialStepRight(x => handler(x), sequence[0], sequence[1]);
268     }
269     else
270     {
271         throw new NotImplementedException();
272     }
273 }
274
275 private void PartialStepRight(Action<IList<LinkIndex>> handler, ulong left, ulong right)
276 {
277     Links.Unsync.Each(Constants.Any, left, doublet =>
278     {
279         StepRight(handler, doublet, right);
280         if (left != doublet)
281         {
282             PartialStepRight(handler, doublet, right);
283         }
284         return true;
285     });
286 }
287
288 private void StepRight(Action<IList<LinkIndex>> handler, ulong left, ulong right)
289 {
290     Links.Unsync.Each(left, Constants.Any, rightStep =>
291     {
292         TryStepRightUp(handler, right, rightStep);
293         return true;
294     });
295 }
296
297 private void TryStepRightUp(Action<IList<LinkIndex>> handler, ulong right, ulong
298 ↪ stepFrom)
299 {
300     var upStep = stepFrom;

```



```

300     var firstSource = Links.Unsync.GetTarget(upStep);
301     while (firstSource != right && firstSource != upStep)
302     {
303         upStep = firstSource;
304         firstSource = Links.Unsync.GetSource(upStep);
305     }
306     if (firstSource == right)
307     {
308         handler(new LinkAddress<LinkIndex>(stepFrom));
309     }
310 }
311
312 // TODO: Test
313 private void PartialStepLeft(Action<IList<LinkIndex>> handler, ulong left, ulong right)
314 {
315     Links.Unsync.Each(right, Constants.Any, doublet =>
316     {
317         StepLeft(handler, left, doublet);
318         if (right != doublet)
319         {
320             PartialStepLeft(handler, left, doublet);
321         }
322         return true;
323     });
324 }
325
326 private void StepLeft(Action<IList<LinkIndex>> handler, ulong left, ulong right)
327 {
328     Links.Unsync.Each(Constants.Any, right, leftStep =>
329     {
330         TryStepLeftUp(handler, left, leftStep);
331         return true;
332     });
333 }
334
335 private void TryStepLeftUp(Action<IList<LinkIndex>> handler, ulong left, ulong stepFrom)
336 {
337     var upStep = stepFrom;
338     var firstTarget = Links.Unsync.GetSource(upStep);
339     while (firstTarget != left && firstTarget != upStep)
340     {
341         upStep = firstTarget;
342         firstTarget = Links.Unsync.GetTarget(upStep);
343     }
344     if (firstTarget == left)
345     {
346         handler(new LinkAddress<LinkIndex>(stepFrom));
347     }
348 }
349
350 private bool StartsWith(ulong sequence, ulong link)
351 {
352     var upStep = sequence;
353     var firstSource = Links.Unsync.GetSource(upStep);
354     while (firstSource != link && firstSource != upStep)
355     {
356         upStep = firstSource;
357         firstSource = Links.Unsync.GetSource(upStep);
358     }
359     return firstSource == link;
360 }
361
362 private bool EndsWith(ulong sequence, ulong link)
363 {
364     var upStep = sequence;
365     var lastTarget = Links.Unsync.GetTarget(upStep);
366     while (lastTarget != link && lastTarget != upStep)
367     {
368         upStep = lastTarget;
369         lastTarget = Links.Unsync.GetTarget(upStep);
370     }
371     return lastTarget == link;
372 }
373
374 public List<ulong> GetAllMatchingSequences0(params ulong[] sequence)
375 {
376     return _sync.ExecuteReadOperation(() =>
377     {
378         var results = new List<ulong>();

```

```

379     if (sequence.Length > 0)
380     {
381         Links.EnsureEachLinkExists(sequence);
382         var firstElement = sequence[0];
383         if (sequence.Length == 1)
384         {
385             results.Add(firstElement);
386             return results;
387         }
388         if (sequence.Length == 2)
389         {
390             var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
391             if (doublet != Constants.Null)
392             {
393                 results.Add(doublet);
394             }
395             return results;
396         }
397         var linksInSequence = new HashSet<ulong>(sequence);
398         void handler(ICollection<LinkIndex> result)
399         {
400             var resultIndex = result[Links.Constants.IndexPart];
401             var filterPosition = 0;
402             StopableSequenceWalker.WalkRight(resultIndex, Links.Unsync.GetSource,
403                 ↪ Links.Unsync.GetTarget,
404                 ↪ x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x,
405                 ↪ x =>
406                 {
407                     if (filterPosition == sequence.Length)
408                     {
409                         filterPosition = -2; // Длиннее чем нужно
410                         return false;
411                     }
412                     if (x != sequence[filterPosition])
413                     {
414                         filterPosition = -1;
415                         return false; // Начинается иначе
416                     }
417                     filterPosition++;
418                     return true;
419                 });
420             if (filterPosition == sequence.Length)
421             {
422                 results.Add(resultIndex);
423             }
424         }
425         if (sequence.Length >= 2)
426         {
427             StepRight(handler, sequence[0], sequence[1]);
428         }
429         var last = sequence.Length - 2;
430         for (var i = 1; i < last; i++)
431         {
432             PartialStepRight(handler, sequence[i], sequence[i + 1]);
433         }
434         if (sequence.Length >= 3)
435         {
436             StepLeft(handler, sequence[sequence.Length - 2],
437                 ↪ sequence[sequence.Length - 1]);
438         }
439     }
440     return results;
441 });
442
443 public HashSet<ulong> GetAllMatchingSequences1(params ulong[] sequence)
444 {
445     return _sync.ExecuteReadOperation(() =>
446     {
447         var results = new HashSet<ulong>();
448         if (sequence.Length > 0)
449         {
450             Links.EnsureEachLinkExists(sequence);
451             var firstElement = sequence[0];
452             if (sequence.Length == 1)
453             {
454                 results.Add(firstElement);
455                 return results;
456             }

```

```

455     }
456     if (sequence.Length == 2)
457     {
458         var doublet = Links.SearchOrDefault(firstElement, sequence[1]);
459         if (doublet != Constants.Null)
460         {
461             results.Add(doublet);
462         }
463         return results;
464     }
465     var matcher = new Matcher(this, sequence, results, null);
466     if (sequence.Length >= 2)
467     {
468         StepRight(matcher.AddFullMatchedToResults, sequence[0], sequence[1]);
469     }
470     var last = sequence.Length - 2;
471     for (var i = 1; i < last; i++)
472     {
473         PartialStepRight(matcher.AddFullMatchedToResults, sequence[i],
474             ↪ sequence[i + 1]);
475     }
476     if (sequence.Length >= 3)
477     {
478         StepLeft(matcher.AddFullMatchedToResults, sequence[sequence.Length - 2],
479             ↪ sequence[sequence.Length - 1]);
480     }
481     return results;
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);
488
489 public string FormatSequence(LinkIndex sequenceLink, Action<StringBuilder, LinkIndex>
490     ↪ elementToString, bool insertComma, params LinkIndex[] knownElements) =>
491     ↪ Links.SyncRoot.ExecuteReadOperation(() => FormatSequence(Links.Unsync, sequenceLink,
492     ↪ elementToString, insertComma, knownElements));
493
494 private string FormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
495     ↪ Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
496     ↪ LinkIndex[] knownElements)
497 {
498     var linksInSequence = new HashSet<ulong>(knownElements);
499     //var entered = new HashSet<ulong>();
500     var sb = new StringBuilder();
501     sb.Append('{');
502     if (links.Exists(sequenceLink))
503     {
504         StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
505             x => linksInSequence.Contains(x) || links.IsPartialPoint(x), element => //
506             ↪ entered.AddAndReturnVoid, x => { }, entered.DoNotContains
507         {
508             if (insertComma && sb.Length > 1)
509             {
510                 sb.Append(',');
511             }
512             //if (entered.Contains(element))
513             //{
514             //    sb.Append('{');
515             //    elementToString(sb, element);
516             //    sb.Append('}');
517             //}
518             //else
519             elementToString(sb, element);
520             if (sb.Length < MaxSequenceFormatSize)
521             {
522                 return true;
523             }
524             sb.Append(insertComma ? ", ..." : "...");
525             return false;
526         }
527     }
528     sb.Append('}');
529     return sb.ToString();
530 }

```

```

524 public string SafeFormatSequence(LinkIndex sequenceLink, params LinkIndex[]
525     ↳ knownElements) => SafeFormatSequence(sequenceLink, (sb, x) => sb.Append(x), true,
526     ↳ knownElements);
527
528 public string SafeFormatSequence(LinkIndex sequenceLink, Action<StringBuilder,
529     ↳ LinkIndex> elementToString, bool insertComma, params LinkIndex[] knownElements) =>
530     ↳ Links.SyncRoot.ExecuteReadOperation(() => SafeFormatSequence(Links.Unsync,
531     ↳ sequenceLink, elementToString, insertComma, knownElements));
532
533 private string SafeFormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
534     ↳ Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
535     ↳ LinkIndex[] knownElements)
536 {
537     var linksInSequence = new HashSet<ulong>(knownElements);
538     var entered = new HashSet<ulong>();
539     var sb = new StringBuilder();
540     sb.Append('{');
541     if (links.Exists(sequenceLink))
542     {
543         StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
544             x => linksInSequence.Contains(x) || links.IsFullPoint(x),
545             ↳ entered.AddAndReturnVoid, x => { }, entered.DoNotContains, element =>
546             {
547                 if (insertComma && sb.Length > 1)
548                 {
549                     sb.Append(',');
550                 }
551                 if (entered.Contains(element))
552                 {
553                     sb.Append('{');
554                     elementToString(sb, element);
555                     sb.Append('}');
556                 }
557                 else
558                 {
559                     elementToString(sb, element);
560                 }
561                 if (sb.Length < MaxSequenceFormatSize)
562                 {
563                     return true;
564                 }
565                 sb.Append(insertComma ? ", ..." : "...");
566                 return false;
567             });
568     }
569     sb.Append('}');
570     return sb.ToString();
571 }
572
573 public List<ulong> GetAllPartiallyMatchingSequences0(params ulong[] sequence)
574 {
575     return _sync.ExecuteReadOperation(() =>
576     {
577         if (sequence.Length > 0)
578         {
579             Links.EnsureEachLinkExists(sequence);
580             var results = new HashSet<ulong>();
581             for (var i = 0; i < sequence.Length; i++)
582             {
583                 AllUsagesCore(sequence[i], results);
584             }
585             var filteredResults = new List<ulong>();
586             var linksInSequence = new HashSet<ulong>(sequence);
587             foreach (var result in results)
588             {
589                 var filterPosition = -1;
590                 StopableSequenceWalker.WalkRight(result, Links.Unsync.GetSource,
591                     ↳ Links.Unsync.GetTarget,
592                     x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x,
593                     ↳ x =>
594                     {
595                         if (filterPosition == (sequence.Length - 1))
596                         {
597                             return false;
598                         }
599                     }
600                     if (filterPosition >= 0)
601                     {

```

```

592         if (x == sequence[filterPosition + 1])
593         {
594             filterPosition++;
595         }
596         else
597         {
598             return false;
599         }
600     }
601     if (filterPosition < 0)
602     {
603         if (x == sequence[0])
604         {
605             filterPosition = 0;
606         }
607     }
608     return true;
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);
628             var results = new HashSet<ulong>();
629             for (var i = 0; i < sequence.Length; i++)
630             {
631                 AllUsagesCore(sequence[i], results);
632             }
633             var filteredResults = new HashSet<ulong>();
634             var matcher = new Matcher(this, sequence, filteredResults, null);
635             matcher.AddAllPartialMatchedToResults(results);
636             return filteredResults;
637         }
638         return new HashSet<ulong>();
639     });
640 }
641
642 public bool GetAllPartiallyMatchingSequences2(Func<IList<LinkIndex>, LinkIndex> handler,
643 ↪ params ulong[] sequence)
644 {
645     return _sync.ExecuteReadOperation(() =>
646     {
647         if (sequence.Length > 0)
648         {
649             Links.EnsureEachLinkExists(sequence);
650
651             var results = new HashSet<ulong>();
652             var filteredResults = new HashSet<ulong>();
653             var matcher = new Matcher(this, sequence, filteredResults, handler);
654             for (var i = 0; i < sequence.Length; i++)
655             {
656                 if (!AllUsagesCore1(sequence[i], results, matcher.HandlePartialMatched))
657                 {
658                     return false;
659                 }
660             }
661             return true;
662         }
663         return true;
664     });
665 }
666
667 //public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
668 //{
669 //    return Sync.ExecuteReadOperation(() =>
670 //    {

```

```

670 //         if (sequence.Length > 0)
671 //         {
672 //             _links.EnsureEachLinkIsAnyOrExists(sequence);
673
674 //             var firstResults = new HashSet<ulong>();
675 //             var lastResults = new HashSet<ulong>();
676
677 //             var first = sequence.First(x => x != LinksConstants.Any);
678 //             var last = sequence.Last(x => x != LinksConstants.Any);
679
680 //             AllUsagesCore(first, firstResults);
681 //             AllUsagesCore(last, lastResults);
682
683 //             firstResults.IntersectWith(lastResults);
684
685 //             //for (var i = 0; i < sequence.Length; i++)
686 //             //    AllUsagesCore(sequence[i], results);
687
688 //             var filteredResults = new HashSet<ulong>();
689 //             var matcher = new Matcher(this, sequence, filteredResults, null);
690 //             matcher.AddAllPartialMatchedToResults(firstResults);
691 //             return filteredResults;
692 //         }
693
694 //         return new HashSet<ulong>();
695 //     });
696 // }
697
698 public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
699 {
700     return _sync.ExecuteReadOperation(() =>
701     {
702         if (sequence.Length > 0)
703         {
704             Links.EnsureEachLinkIsAnyOrExists(sequence);
705             var firstResults = new HashSet<ulong>();
706             var lastResults = new HashSet<ulong>();
707             var first = sequence.First(x => x != Constants.Any);
708             var last = sequence.Last(x => x != Constants.Any);
709             AllUsagesCore(first, firstResults);
710             AllUsagesCore(last, lastResults);
711             firstResults.IntersectWith(lastResults);
712             //for (var i = 0; i < sequence.Length; i++)
713             //    AllUsagesCore(sequence[i], results);
714             var filteredResults = new HashSet<ulong>();
715             var matcher = new Matcher(this, sequence, filteredResults, null);
716             matcher.AddAllPartialMatchedToResults(firstResults);
717             return filteredResults;
718         }
719         return new HashSet<ulong>();
720     });
721 }
722
723 public HashSet<ulong> GetAllPartiallyMatchingSequences4(HashSet<ulong> readAsElements,
724     ↪ IList<ulong> sequence)
725 {
726     return _sync.ExecuteReadOperation(() =>
727     {
728         if (sequence.Count > 0)
729         {
730             Links.EnsureEachLinkExists(sequence);
731             var results = new HashSet<LinkIndex>();
732             //var nextResults = new HashSet<ulong>();
733             //for (var i = 0; i < sequence.Length; i++)
734             //{
735             //    AllUsagesCore(sequence[i], nextResults);
736             //    if (results.IsNullOrEmpty())
737             //    {
738             //        results = nextResults;
739             //        nextResults = new HashSet<ulong>();
740             //    }
741             //    else
742             //    {
743             //        results.IntersectWith(nextResults);
744             //        nextResults.Clear();
745             //    }
746             //}
747             var collector1 = new AllUsagesCollector1(Links.Unsync, results);

```

```

747     collector1.Collect(Links.Unsync.GetLink(sequence[0]));
748     var next = new HashSet<ulong>();
749     for (var i = 1; i < sequence.Count; i++)
750     {
751         var collector = new AllUsagesCollector1(Links.Unsync, next);
752         collector.Collect(Links.Unsync.GetLink(sequence[i]));
753
754         results.IntersectWith(next);
755         next.Clear();
756     }
757     var filteredResults = new HashSet<ulong>();
758     var matcher = new Matcher(this, sequence, filteredResults, null,
759         ↪ readAsElements);
760     matcher.AddAllPartialMatchedToResultsAndReadAsElements(results.OrderBy(x =>
761         ↪ x)); // OrderBy is a Hack
762     return filteredResults;
763 }
764 return new HashSet<ulong>();
765 });
766 }
767
768 // Does not work
769 //public HashSet<ulong> GetAllPartiallyMatchingSequences5(HashSet<ulong> readAsElements,
770 ↪ params ulong[] sequence)
771 //{
772 //    var visited = new HashSet<ulong>();
773 //    var results = new HashSet<ulong>();
774 //    var matcher = new Matcher(this, sequence, visited, x => { results.Add(x); return
775 ↪ true; }, readAsElements);
776 //    var last = sequence.Length - 1;
777 //    for (var i = 0; i < last; i++)
778 //    {
779 //        PartialStepRight(matcher.PartialMatch, sequence[i], sequence[i + 1]);
780 //    }
781 //    return results;
782 //}
783
784 public List<ulong> GetAllPartiallyMatchingSequences(params ulong[] sequence)
785 {
786     return _sync.ExecuteReadOperation(() =>
787     {
788         if (sequence.Length > 0)
789         {
790             Links.EnsureEachLinkExists(sequence);
791             //var firstElement = sequence[0];
792             //if (sequence.Length == 1)
793             //{
794             //    //results.Add(firstElement);
795             //    return results;
796             //}
797             //if (sequence.Length == 2)
798             //{
799             //    //var doublet = _links.SearchCore(firstElement, sequence[1]);
800             //    //if (doublet != Doublets.Links.Null)
801             //    //    results.Add(doublet);
802             //    return results;
803             //}
804             //var lastElement = sequence[sequence.Length - 1];
805             //Func<ulong, bool> handler = x =>
806             //{
807             //    if (StartsWith(x, firstElement) && EndsWith(x, lastElement))
808             //        ↪ results.Add(x);
809             //    return true;
810             //};
811             //if (sequence.Length >= 2)
812             //    StepRight(handler, sequence[0], sequence[1]);
813             //var last = sequence.Length - 2;
814             //for (var i = 1; i < last; i++)
815             //    PartialStepRight(handler, sequence[i], sequence[i + 1]);
816             //if (sequence.Length >= 3)
817             //    StepLeft(handler, sequence[sequence.Length - 2],
818             //        ↪ sequence[sequence.Length - 1]);
819             //if (sequence.Length == 1)
820             //    throw new NotImplementedException(); // all sequences, containing
821             //    ↪ this element?
822             //}
823         }
824     });
825 }

```

```

817         //if (sequence.Length == 2)
818         //{
819             //var results = new List<ulong>();
820             //PartialStepRight(results.Add, sequence[0], sequence[1]);
821             //return results;
822         //}
823         //var matches = new List<List<ulong>>();
824         //var last = sequence.Length - 1;
825         //for (var i = 0; i < last; i++)
826         //{
827             //var results = new List<ulong>();
828             //StepRight(results.Add, sequence[i], sequence[i + 1]);
829             //PartialStepRight(results.Add, sequence[i], sequence[i + 1]);
830             //if (results.Count > 0)
831                 //matches.Add(results);
832             //else
833                 //return results;
834             //if (matches.Count == 2)
835             //{
836                 //var merged = new List<ulong>();
837                 //for (var j = 0; j < matches[0].Count; j++)
838                     //for (var k = 0; k < matches[1].Count; k++)
839                         //CloseInnerConnections(merged.Add, matches[0][j],
840                         //    matches[1][k]);
841                 //if (merged.Count > 0)
842                     //matches = new List<List<ulong>> { merged };
843                 //else
844                     //return new List<ulong>();
845             //}
846         //if (matches.Count > 0)
847         //{
848             //var usages = new HashSet<ulong>();
849             //for (int i = 0; i < sequence.Length; i++)
850             //{
851                 //AllUsagesCore(sequence[i], usages);
852             //}
853             //for (int i = 0; i < matches[0].Count; i++)
854                 //AllUsagesCore(matches[0][i], usages);
855             //usages.UnionWith(matches[0]);
856             //return usages.ToList();
857         //}
858         var firstLinkUsages = new HashSet<ulong>();
859         AllUsagesCore(sequence[0], firstLinkUsages);
860         firstLinkUsages.Add(sequence[0]);
861         //var previousMatchings = firstLinkUsages.ToList(); //new List<ulong>() {
862         //    sequence[0] }; // or all sequences, containing this element?
863         //return GetAllPartiallyMatchingSequencesCore(sequence, firstLinkUsages,
864         //    1).ToList();
865         var results = new HashSet<ulong>();
866         foreach (var match in GetAllPartiallyMatchingSequencesCore(sequence,
867             firstLinkUsages, 1))
868         {
869             AllUsagesCore(match, results);
870         }
871         return results.ToList();
872     }
873     return new List<ulong>();
874 }
875 }
876
877 /// <remarks>
878 /// TODO: Может потребоваться ограничение на уровень глубины рекурсии
879 /// </remarks>
880 public HashSet<ulong> AllUsages(ulong link)
881 {
882     return _sync.ExecuteReadOperation(() =>
883     {
884         var usages = new HashSet<ulong>();
885         AllUsagesCore(link, usages);
886         return usages;
887     });
888 }
889
890 // При сборе всех использований (последовательностей) можно сохранять обратный путь к
891 // той связи с которой начинался поиск (STTTSSSTT),
892 // причём достаточно одного бита для хранения перехода влево или вправо

```



```

889 private void AllUsagesCore(ulong link, HashSet<ulong> usages)
890 {
891     bool handler(ulong doublet)
892     {
893         if (usages.Add(doublet))
894         {
895             AllUsagesCore(doublet, usages);
896         }
897         return true;
898     }
899     Links.Unsync.Each(link, Constants.Any, handler);
900     Links.Unsync.Each(Constants.Any, link, handler);
901 }
902
903 public HashSet<ulong> AllBottomUsages(ulong link)
904 {
905     return _sync.ExecuteReadOperation(() =>
906     {
907         var visits = new HashSet<ulong>();
908         var usages = new HashSet<ulong>();
909         AllBottomUsagesCore(link, visits, usages);
910         return usages;
911     });
912 }
913
914 private void AllBottomUsagesCore(ulong link, HashSet<ulong> visits, HashSet<ulong>
↪ usages)
915 {
916     bool handler(ulong doublet)
917     {
918         if (visits.Add(doublet))
919         {
920             AllBottomUsagesCore(doublet, visits, usages);
921         }
922         return true;
923     }
924     if (Links.Unsync.Count(Constants.Any, link) == 0)
925     {
926         usages.Add(link);
927     }
928     else
929     {
930         Links.Unsync.Each(link, Constants.Any, handler);
931         Links.Unsync.Each(Constants.Any, link, handler);
932     }
933 }
934
935 public ulong CalculateTotalSymbolFrequencyCore(ulong symbol)
936 {
937     if (Options.UseSequenceMarker)
938     {
939         var counter = new TotalMarkedSequenceSymbolFrequencyOneOffCounter<ulong>(Links,
↪ Options.MarkedSequenceMatcher, symbol);
940         return counter.Count();
941     }
942     else
943     {
944         var counter = new TotalSequenceSymbolFrequencyOneOffCounter<ulong>(Links,
↪ symbol);
945         return counter.Count();
946     }
947 }
948
949 private bool AllUsagesCore1(ulong link, HashSet<ulong> usages, Func<IList<LinkIndex>,
↪ LinkIndex> outerHandler)
950 {
951     bool handler(ulong doublet)
952     {
953         if (usages.Add(doublet))
954         {
955             if (outerHandler(new LinkAddress<LinkIndex>(doublet)) != Constants.Continue)
956             {
957                 return false;
958             }
959             if (!AllUsagesCore1(doublet, usages, outerHandler))
960             {
961                 return false;
962             }

```

```

963     }
964     return true;
965 }
966 return Links.Unsync.Each(link, Constants.Any, handler)
967     && Links.Unsync.Each(Constants.Any, link, handler);
968 }
969
970 public void CalculateAllUsages(ulong[] totals)
971 {
972     var calculator = new AllUsagesCalculator(Links, totals);
973     calculator.Calculate();
974 }
975
976 public void CalculateAllUsages2(ulong[] totals)
977 {
978     var calculator = new AllUsagesCalculator2(Links, totals);
979     calculator.Calculate();
980 }
981
982 private class AllUsagesCalculator
983 {
984     private readonly SynchronizedLinks<ulong> _links;
985     private readonly ulong[] _totals;
986
987     public AllUsagesCalculator(SynchronizedLinks<ulong> links, ulong[] totals)
988     {
989         _links = links;
990         _totals = totals;
991     }
992
993     public void Calculate() => _links.Each(_links.Constants.Any, _links.Constants.Any,
994         ↪ CalculateCore);
995
996     private bool CalculateCore(ulong link)
997     {
998         if (_totals[link] == 0)
999         {
1000             var total = 1UL;
1001             _totals[link] = total;
1002             var visitedChildren = new HashSet<ulong>();
1003             bool linkCalculator(ulong child)
1004             {
1005                 if (link != child && visitedChildren.Add(child))
1006                 {
1007                     total += _totals[child] == 0 ? 1 : _totals[child];
1008                 }
1009                 return true;
1010             }
1011             _links.Unsync.Each(link, _links.Constants.Any, linkCalculator);
1012             _links.Unsync.Each(_links.Constants.Any, link, linkCalculator);
1013             _totals[link] = total;
1014         }
1015         return true;
1016     }
1017 }
1018
1019 private class AllUsagesCalculator2
1020 {
1021     private readonly SynchronizedLinks<ulong> _links;
1022     private readonly ulong[] _totals;
1023
1024     public AllUsagesCalculator2(SynchronizedLinks<ulong> links, ulong[] totals)
1025     {
1026         _links = links;
1027         _totals = totals;
1028     }
1029
1030     public void Calculate() => _links.Each(_links.Constants.Any, _links.Constants.Any,
1031         ↪ CalculateCore);
1032
1033     private bool IsElement(ulong link)
1034     {
1035         // _linksInSequence.Contains(link) ||
1036         return _links.Unsync.GetTarget(link) == link || _links.Unsync.GetSource(link) ==
1037             ↪ link;
1038     }
1039
1040     private bool CalculateCore(ulong link)
1041     {

```

```

1039 // TODO: Проработать защиту от заикливания
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);
1077             // Обработка элемента
1078             if (isElement(target))
1079             {
1080                 visitLeaf(target);
1081             }
1082             if (isElement(source))
1083             {
1084                 visitLeaf(source);
1085             }
1086             element = source;
1087         }
1088         else
1089         {
1090             stack.Push(element);
1091             visitNode(element);
1092             element = getTarget(element);
1093         }
1094     }
1095     _totals[link]++;
1096     return true;
1097 }
1098 }
1099
1100 private class AllUsagesCollector
1101 {
1102     private readonly ILinks<ulong> _links;
1103     private readonly HashSet<ulong> _usages;
1104
1105     public AllUsagesCollector(ILinks<ulong> links, HashSet<ulong> usages)
1106     {
1107         _links = links;
1108         _usages = usages;
1109     }
1110
1111     public bool Collect(ulong link)
1112     {
1113         if (_usages.Add(link))
1114         {
1115             _links.Each(link, _links.Constants.Any, Collect);
1116             _links.Each(_links.Constants.Any, link, Collect);
1117         }
1118     }
1119 }

```

```

1118     }
1119     return true;
1120 }
1121 }
1122
1123 private class AllUsagesCollector1
1124 {
1125     private readonly ILinks<ulong> _links;
1126     private readonly HashSet<ulong> _usages;
1127     private readonly ulong _continue;
1128
1129     public AllUsagesCollector1(ILinks<ulong> links, HashSet<ulong> usages)
1130     {
1131         _links = links;
1132         _usages = usages;
1133         _continue = _links.Constants.Continue;
1134     }
1135
1136     public ulong Collect(IList<ulong> link)
1137     {
1138         var linkIndex = _links.GetIndex(link);
1139         if (_usages.Add(linkIndex))
1140         {
1141             _links.Each(Collect, _links.Constants.Any, linkIndex);
1142         }
1143         return _continue;
1144     }
1145 }
1146
1147 private class AllUsagesCollector2
1148 {
1149     private readonly ILinks<ulong> _links;
1150     private readonly BitString _usages;
1151
1152     public AllUsagesCollector2(ILinks<ulong> links, BitString usages)
1153     {
1154         _links = links;
1155         _usages = usages;
1156     }
1157
1158     public bool Collect(ulong link)
1159     {
1160         if (_usages.Add((long)link))
1161         {
1162             _links.Each(link, _links.Constants.Any, Collect);
1163             _links.Each(_links.Constants.Any, link, Collect);
1164         }
1165         return true;
1166     }
1167 }
1168
1169 private class AllUsagesIntersectingCollector
1170 {
1171     private readonly SynchronizedLinks<ulong> _links;
1172     private readonly HashSet<ulong> _intersectWith;
1173     private readonly HashSet<ulong> _usages;
1174     private readonly HashSet<ulong> _enter;
1175
1176     public AllUsagesIntersectingCollector(SynchronizedLinks<ulong> links, HashSet<ulong>
↵ intersectWith, HashSet<ulong> usages)
1177     {
1178         _links = links;
1179         _intersectWith = intersectWith;
1180         _usages = usages;
1181         _enter = new HashSet<ulong>(); // защита от зацикливания
1182     }
1183
1184     public bool Collect(ulong link)
1185     {
1186         if (_enter.Add(link))
1187         {
1188             if (_intersectWith.Contains(link))
1189             {
1190                 _usages.Add(link);
1191             }
1192             _links.Unsync.Each(link, _links.Constants.Any, Collect);
1193             _links.Unsync.Each(_links.Constants.Any, link, Collect);
1194         }
1195         return true;
1196     }

```

```

1197     }
1198
1199     private void CloseInnerConnections(Action<IList<LinkIndex>> handler, ulong left, ulong
    ↪ right)
1200     {
1201         TryStepLeftUp(handler, left, right);
1202         TryStepRightUp(handler, right, left);
1203     }
1204
1205     private void AllCloseConnections(Action<IList<LinkIndex>> handler, ulong left, ulong
    ↪ right)
1206     {
1207         // Direct
1208         if (left == right)
1209         {
1210             handler(new LinkAddress<LinkIndex>(left));
1211         }
1212         var doublet = Links.Unsync.SearchOrDefault(left, right);
1213         if (doublet != Constants.Null)
1214         {
1215             handler(new LinkAddress<LinkIndex>(doublet));
1216         }
1217         // Inner
1218         CloseInnerConnections(handler, left, right);
1219         // Outer
1220         StepLeft(handler, left, right);
1221         StepRight(handler, left, right);
1222         PartialStepRight(handler, left, right);
1223         PartialStepLeft(handler, left, right);
1224     }
1225
1226     private HashSet<ulong> GetAllPartiallyMatchingSequencesCore(ulong[] sequence,
    ↪ HashSet<ulong> previousMatchings, long startAt)
1227     {
1228         if (startAt >= sequence.Length) // ?
1229         {
1230             return previousMatchings;
1231         }
1232         var secondLinkUsages = new HashSet<ulong>();
1233         AllUsagesCore(sequence[startAt], secondLinkUsages);
1234         secondLinkUsages.Add(sequence[startAt]);
1235         var matchings = new HashSet<ulong>();
1236         var filler = new SetFiller<LinkIndex, LinkIndex>(matchings, Constants.Continue);
1237         //for (var i = 0; i < previousMatchings.Count; i++)
1238         foreach (var secondLinkUsage in secondLinkUsages)
1239         {
1240             foreach (var previousMatching in previousMatchings)
1241             {
1242                 //AllCloseConnections(matchings.AddAndReturnVoid, previousMatching,
    ↪ secondLinkUsage);
1243                 StepRight(filler.AddFirstAndReturnConstant, previousMatching,
    ↪ secondLinkUsage);
1244                 TryStepRightUp(filler.AddFirstAndReturnConstant, secondLinkUsage,
    ↪ previousMatching);
1245                 //PartialStepRight(matchings.AddAndReturnVoid, secondLinkUsage,
    ↪ sequence[startAt]); // почему-то эта ошибочная запись приводит к
    ↪ желаемым результатам.
1246                 PartialStepRight(filler.AddFirstAndReturnConstant, previousMatching,
    ↪ secondLinkUsage);
1247             }
1248         }
1249         if (matchings.Count == 0)
1250         {
1251             return matchings;
1252         }
1253         return GetAllPartiallyMatchingSequencesCore(sequence, matchings, startAt + 1); // ??
1254     }
1255
1256     private static void EnsureEachLinkIsAnyOrZeroOrManyOrExists(SynchronizedLinks<ulong>
    ↪ links, params ulong[] sequence)
1257     {
1258         if (sequence == null)
1259         {
1260             return;
1261         }
1262         for (var i = 0; i < sequence.Length; i++)
1263         {

```

```

1264         if (sequence[i] != links.Constants.Any && sequence[i] != ZeroOrMany &&
1265             ↪ !links.Exists(sequence[i]))
1266         {
1267             throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
1268                 ↪ $"patternSequence[{i}]");
1269         }
1270     }
1271 }
1272 // Pattern Matching -> Key To Triggers
1273 public HashSet<ulong> MatchPattern(params ulong[] patternSequence)
1274 {
1275     return _sync.ExecuteReadOperation(() =>
1276     {
1277         patternSequence = Simplify(patternSequence);
1278         if (patternSequence.Length > 0)
1279         {
1280             EnsureEachLinkIsAnyOrZeroOrManyOrExists(Links, patternSequence);
1281             var uniqueSequenceElements = new HashSet<ulong>();
1282             for (var i = 0; i < patternSequence.Length; i++)
1283             {
1284                 if (patternSequence[i] != Constants.Any && patternSequence[i] !=
1285                     ↪ ZeroOrMany)
1286                 {
1287                     uniqueSequenceElements.Add(patternSequence[i]);
1288                 }
1289             }
1290             var results = new HashSet<ulong>();
1291             foreach (var uniqueSequenceElement in uniqueSequenceElements)
1292             {
1293                 AllUsagesCore(uniqueSequenceElement, results);
1294             }
1295             var filteredResults = new HashSet<ulong>();
1296             var matcher = new PatternMatcher(this, patternSequence, filteredResults);
1297             matcher.AddAllPatternMatchedToResults(results);
1298             return filteredResults;
1299         }
1300         return new HashSet<ulong>();
1301     });
1302 }
1303 // Найти все возможные связи между указанным списком связей.
1304 // Находит связи между всеми указанными связями в любом порядке.
1305 // TODO: решить что делать с повторами (когда одни и те же элементы встречаются
1306 ↪ несколько раз в последовательности)
1307 public HashSet<ulong> GetAllConnections(params ulong[] linksToConnect)
1308 {
1309     return _sync.ExecuteReadOperation(() =>
1310     {
1311         var results = new HashSet<ulong>();
1312         if (linksToConnect.Length > 0)
1313         {
1314             Links.EnsureEachLinkExists(linksToConnect);
1315             AllUsagesCore(linksToConnect[0], results);
1316             for (var i = 1; i < linksToConnect.Length; i++)
1317             {
1318                 var next = new HashSet<ulong>();
1319                 AllUsagesCore(linksToConnect[i], next);
1320                 results.IntersectWith(next);
1321             }
1322             return results;
1323         }
1324     });
1325 }
1326 public HashSet<ulong> GetAllConnections1(params ulong[] linksToConnect)
1327 {
1328     return _sync.ExecuteReadOperation(() =>
1329     {
1330         var results = new HashSet<ulong>();
1331         if (linksToConnect.Length > 0)
1332         {
1333             Links.EnsureEachLinkExists(linksToConnect);
1334             var collector1 = new AllUsagesCollector(Links.Unsync, results);
1335             collector1.Collect(linksToConnect[0]);
1336             var next = new HashSet<ulong>();
1337             for (var i = 1; i < linksToConnect.Length; i++)
1338             {

```

```

1338         var collector = new AllUsagesCollector(Links.Unsync, next);
1339         collector.Collect(linksToConnect[i]);
1340         results.IntersectWith(next);
1341         next.Clear();
1342     }
1343 }
1344     return results;
1345 });
1346 }
1347
1348 public HashSet<ulong> GetAllConnections2(params ulong[] linksToConnect)
1349 {
1350     return _sync.ExecuteReadOperation(() =>
1351     {
1352         var results = new HashSet<ulong>();
1353         if (linksToConnect.Length > 0)
1354         {
1355             Links.EnsureEachLinkExists(linksToConnect);
1356             var collector1 = new AllUsagesCollector(Links, results);
1357             collector1.Collect(linksToConnect[0]);
1358             //AllUsagesCore(linksToConnect[0], results);
1359             for (var i = 1; i < linksToConnect.Length; i++)
1360             {
1361                 var next = new HashSet<ulong>();
1362                 var collector = new AllUsagesIntersectingCollector(Links, results, next);
1363                 collector.Collect(linksToConnect[i]);
1364                 //AllUsagesCore(linksToConnect[i], next);
1365                 //results.IntersectWith(next);
1366                 results = next;
1367             }
1368         }
1369         return results;
1370     });
1371 }
1372
1373 public List<ulong> GetAllConnections3(params ulong[] linksToConnect)
1374 {
1375     return _sync.ExecuteReadOperation(() =>
1376     {
1377         var results = new BitString((long)Links.Unsync.Count() + 1); // new
1378         ↪ BitArray((int)_links.Total + 1);
1379         if (linksToConnect.Length > 0)
1380         {
1381             Links.EnsureEachLinkExists(linksToConnect);
1382             var collector1 = new AllUsagesCollector2(Links.Unsync, results);
1383             collector1.Collect(linksToConnect[0]);
1384             for (var i = 1; i < linksToConnect.Length; i++)
1385             {
1386                 var next = new BitString((long)Links.Unsync.Count() + 1); //new
1387                 ↪ BitArray((int)_links.Total + 1);
1388                 var collector = new AllUsagesCollector2(Links.Unsync, next);
1389                 collector.Collect(linksToConnect[i]);
1390                 results = results.And(next);
1391             }
1392         }
1393         return results.GetSetUInt64Indices();
1394     });
1395 }
1396
1397 private static ulong[] Simplify(ulong[] sequence)
1398 {
1399     // Считаем новый размер последовательности
1400     long newLength = 0;
1401     var zeroOrManyStepped = false;
1402     for (var i = 0; i < sequence.Length; i++)
1403     {
1404         if (sequence[i] == ZeroOrMany)
1405         {
1406             if (zeroOrManyStepped)
1407             {
1408                 continue;
1409             }
1410             zeroOrManyStepped = true;
1411         }
1412         else
1413         {
1414             //if (zeroOrManyStepped) Is it efficient?
1415             zeroOrManyStepped = false;
1416             newLength++;
1417         }
1418     }
1419     return sequence.Take(newLength).ToArray();
1420 }

```

```

1414     }
1415     newLength++;
1416 }
1417 // Строим новую последовательность
1418 zeroOrManyStepped = false;
1419 var newSequence = new ulong[newLength];
1420 long j = 0;
1421 for (var i = 0; i < sequence.Length; i++)
1422 {
1423     //var current = zeroOrManyStepped;
1424     //zeroOrManyStepped = patternSequence[i] == zeroOrMany;
1425     //if (current && zeroOrManyStepped)
1426     //    continue;
1427     //var newZeroOrManyStepped = patternSequence[i] == zeroOrMany;
1428     //if (zeroOrManyStepped && newZeroOrManyStepped)
1429     //    continue;
1430     //zeroOrManyStepped = newZeroOrManyStepped;
1431     if (sequence[i] == ZeroOrMany)
1432     {
1433         if (zeroOrManyStepped)
1434         {
1435             continue;
1436         }
1437         zeroOrManyStepped = true;
1438     }
1439     else
1440     {
1441         //if (zeroOrManyStepped) Is it efficient?
1442         zeroOrManyStepped = false;
1443     }
1444     newSequence[j++] = sequence[i];
1445 }
1446 return newSequence;
1447 }
1448
1449 public static void TestSimplify()
1450 {
1451     var sequence = new ulong[] { ZeroOrMany, ZeroOrMany, 2, 3, 4, ZeroOrMany,
1452     ↪ ZeroOrMany, ZeroOrMany, 4, ZeroOrMany, ZeroOrMany, ZeroOrMany };
1453     var simplifiedSequence = Simplify(sequence);
1454 }
1455
1456 public List<ulong> GetSimilarSequences() => new List<ulong>();
1457
1458 public void Prediction()
1459 {
1460     //_links
1461     //_sequences
1462 }
1463
1464 #region From Triplets
1465
1466 //public static void DeleteSequence(Link sequence)
1467 //{
1468 //}
1469
1470 public List<ulong> CollectMatchingSequences(ulong[] links)
1471 {
1472     if (links.Length == 1)
1473     {
1474         throw new Exception("Подпоследовательности с одним элементом не
1475         ↪ поддерживаются.");
1476     }
1477     var leftBound = 0;
1478     var rightBound = links.Length - 1;
1479     var left = links[leftBound++];
1480     var right = links[rightBound--];
1481     var results = new List<ulong>();
1482     CollectMatchingSequences(left, leftBound, links, right, rightBound, ref results);
1483     return results;
1484 }
1485
1486 private void CollectMatchingSequences(ulong leftLink, int leftBound, ulong[]
1487     ↪ middleLinks, ulong rightLink, int rightBound, ref List<ulong> results)
1488 {
1489     var leftLinkTotalReferers = Links.Unsync.Count(leftLink);
1490     var rightLinkTotalReferers = Links.Unsync.Count(rightLink);
1491     if (leftLinkTotalReferers <= rightLinkTotalReferers)
1492     {

```



```

1490     var nextLeftLink = middleLinks[leftBound];
1491     var elements = GetRightElements(leftLink, nextLeftLink);
1492     if (leftBound <= rightBound)
1493     {
1494         for (var i = elements.Length - 1; i >= 0; i--)
1495         {
1496             var element = elements[i];
1497             if (element != 0)
1498             {
1499                 CollectMatchingSequences(element, leftBound + 1, middleLinks,
1500                     ↪ rightLink, rightBound, ref results);
1501             }
1502         }
1503     }
1504     else
1505     {
1506         for (var i = elements.Length - 1; i >= 0; i--)
1507         {
1508             var element = elements[i];
1509             if (element != 0)
1510             {
1511                 results.Add(element);
1512             }
1513         }
1514     }
1515     else
1516     {
1517         var nextRightLink = middleLinks[rightBound];
1518         var elements = GetLeftElements(rightLink, nextRightLink);
1519         if (leftBound <= rightBound)
1520         {
1521             for (var i = elements.Length - 1; i >= 0; i--)
1522             {
1523                 var element = elements[i];
1524                 if (element != 0)
1525                 {
1526                     CollectMatchingSequences(leftLink, leftBound, middleLinks,
1527                         ↪ elements[i], rightBound - 1, ref results);
1528                 }
1529             }
1530         }
1531         else
1532         {
1533             for (var i = elements.Length - 1; i >= 0; i--)
1534             {
1535                 var element = elements[i];
1536                 if (element != 0)
1537                 {
1538                     results.Add(element);
1539                 }
1540             }
1541         }
1542     }
1543 }
1544 public ulong[] GetRightElements(ulong startLink, ulong rightLink)
1545 {
1546     var result = new ulong[5];
1547     TryStepRight(startLink, rightLink, result, 0);
1548     Links.Each(Constants.Any, startLink, couple =>
1549     {
1550         if (couple != startLink)
1551         {
1552             if (TryStepRight(couple, rightLink, result, 2))
1553             {
1554                 return false;
1555             }
1556         }
1557         return true;
1558     });
1559     if (Links.GetTarget(Links.GetTarget(startLink)) == rightLink)
1560     {
1561         result[4] = startLink;
1562     }
1563     return result;
1564 }
1565

```

```

1566 public bool TryStepRight(ulong startLink, ulong rightLink, ulong[] result, int offset)
1567 {
1568     var added = 0;
1569     Links.Each(startLink, Constants.Any, couple =>
1570     {
1571         if (couple != startLink)
1572         {
1573             var coupleTarget = Links.GetTarget(couple);
1574             if (coupleTarget == rightLink)
1575             {
1576                 result[offset] = couple;
1577                 if (++added == 2)
1578                 {
1579                     return false;
1580                 }
1581             }
1582             else if (Links.GetSource(coupleTarget) == rightLink) // coupleTarget.Linker
1583                 == Net.And &&
1584             {
1585                 result[offset + 1] = couple;
1586                 if (++added == 2)
1587                 {
1588                     return false;
1589                 }
1590             }
1591             return true;
1592         });
1593     return added > 0;
1594 }
1595
1596 public ulong[] GetLeftElements(ulong startLink, ulong leftLink)
1597 {
1598     var result = new ulong[5];
1599     TryStepLeft(startLink, leftLink, result, 0);
1600     Links.Each(startLink, Constants.Any, couple =>
1601     {
1602         if (couple != startLink)
1603         {
1604             if (TryStepLeft(couple, leftLink, result, 2))
1605             {
1606                 return false;
1607             }
1608             return true;
1609         });
1610     if (Links.GetSource(Links.GetSource(leftLink)) == startLink)
1611     {
1612         result[4] = leftLink;
1613     }
1614     return result;
1615 }
1616
1617 public bool TryStepLeft(ulong startLink, ulong leftLink, ulong[] result, int offset)
1618 {
1619     var added = 0;
1620     Links.Each(Constants.Any, startLink, couple =>
1621     {
1622         if (couple != startLink)
1623         {
1624             var coupleSource = Links.GetSource(couple);
1625             if (coupleSource == leftLink)
1626             {
1627                 result[offset] = couple;
1628                 if (++added == 2)
1629                 {
1630                     return false;
1631                 }
1632             }
1633             else if (Links.GetTarget(coupleSource) == leftLink) // coupleSource.Linker
1634                 == Net.And &&
1635             {
1636                 result[offset + 1] = couple;
1637                 if (++added == 2)
1638                 {
1639                     return false;
1640                 }
1641             }
1642         }
1643     });
1644     return added > 0;
1645 }

```

```

1643         return true;
1644     });
1645     return added > 0;
1646 }
1647
1648 #endregion
1649
1650 #region Walkers
1651
1652 public class PatternMatcher : RightSequenceWalker<ulong>
1653 {
1654     private readonly Sequences _sequences;
1655     private readonly ulong[] _patternSequence;
1656     private readonly HashSet<LinkIndex> _linksInSequence;
1657     private readonly HashSet<LinkIndex> _results;
1658
1659     #region Pattern Match
1660
1661     enum PatternBlockType
1662     {
1663         Undefined,
1664         Gap,
1665         Elements
1666     }
1667
1668     struct PatternBlock
1669     {
1670         public PatternBlockType Type;
1671         public long Start;
1672         public long Stop;
1673     }
1674
1675     private readonly List<PatternBlock> _pattern;
1676     private int _patternPosition;
1677     private long _sequencePosition;
1678
1679     #endregion
1680
1681     public PatternMatcher(Sequences sequences, LinkIndex[] patternSequence,
1682         ↳ HashSet<LinkIndex> results)
1683         : base(sequences.Links.Unsync, new DefaultStack<ulong>())
1684     {
1685         _sequences = sequences;
1686         _patternSequence = patternSequence;
1687         _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
1688         ↳ _sequences.Constants.Any && x != ZeroOrMany));
1689         _results = results;
1690         _pattern = CreateDetailedPattern();
1691     }
1692
1693     protected override bool IsElement(ulong link) => _linksInSequence.Contains(link) ||
1694     ↳ base.IsElement(link);
1695
1696     public bool PatternMatch(LinkIndex sequenceToMatch)
1697     {
1698         _patternPosition = 0;
1699         _sequencePosition = 0;
1700         foreach (var part in Walk(sequenceToMatch))
1701         {
1702             if (!PatternMatchCore(part))
1703             {
1704                 break;
1705             }
1706         }
1707         return _patternPosition == _pattern.Count || (_patternPosition == _pattern.Count
1708         ↳ - 1 && _pattern[_patternPosition].Start == 0);
1709     }
1710
1711     private List<PatternBlock> CreateDetailedPattern()
1712     {
1713         var pattern = new List<PatternBlock>();
1714         var patternBlock = new PatternBlock();
1715         for (var i = 0; i < _patternSequence.Length; i++)
1716         {
1717             if (patternBlock.Type == PatternBlockType.Undefined)
1718             {
1719                 if (_patternSequence[i] == _sequences.Constants.Any)
1720                 {
1721                     patternBlock.Type = PatternBlockType.Gap;
1722                     patternBlock.Start = 1;
1723                     patternBlock.Stop = 1;
1724                 }
1725             }
1726         }
1727     }

```

```

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] == _sequences.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 {
1763     if (_patternSequence[i] == _sequences.Constants.Any)
1764     {
1765         patternBlock.Start++;
1766         if (patternBlock.Stop < patternBlock.Start)
1767         {
1768             patternBlock.Stop = patternBlock.Start;
1769         }
1770     }
1771     else if (_patternSequence[i] == ZeroOrMany)
1772     {
1773         patternBlock.Stop = long.MaxValue;
1774     }
1775     else
1776     {
1777         pattern.Add(patternBlock);
1778         patternBlock = new PatternBlock
1779         {
1780             Type = PatternBlockType.Elements,
1781             Start = i,
1782             Stop = i
1783         };
1784     }
1785 }
1786 }
1787 if (patternBlock.Type != PatternBlockType.Undefined)
1788 {
1789     pattern.Add(patternBlock);
1790 }
1791 return pattern;
1792 }
1793
1794 // match: search for regexp anywhere in text
1795 //int match(char* regexp, char* text)
1796 //{
1797 //    do
1798 //    {
1799 //        } while (*text++ != '\0');

```

```

1800 // return 0;
1801 //}
1802
1803 // matchhere: search for regexp at beginning of text
1804 //int matchhere(char* regexp, char* text)
1805 //{
1806 //    if (regexp[0] == '\0')
1807 //        return 1;
1808 //    if (regexp[1] == '*')
1809 //        return matchstar(regexp[0], regexp + 2, text);
1810 //    if (regexp[0] == '$' && regexp[1] == '\0')
1811 //        return *text == '\0';
1812 //    if (*text != '\0' && (regexp[0] == '.' || regexp[0] == *text))
1813 //        return matchhere(regexp + 1, text + 1);
1814 //    return 0;
1815 //}
1816
1817 // matchstar: search for c*regexp at beginning of text
1818 //int matchstar(int c, char* regexp, char* text)
1819 //{
1820 //    do
1821 //    {
1822 //        /* a * matches zero or more instances */
1823 //        if (matchhere(regexp, text))
1824 //            return 1;
1825 //    } while (*text != '\0' && (*text++ == c || c == '.'));
1826 //    return 0;
1827 //}
1828
1829 //private void GetNextPatternElement(out LinkIndex element, out long mininumGap, out
1830 //    long maximumGap)
1831 //{
1832 //    mininumGap = 0;
1833 //    maximumGap = 0;
1834 //    element = 0;
1835 //    for (; _patternPosition < _patternSequence.Length; _patternPosition++)
1836 //    {
1837 //        if (_patternSequence[_patternPosition] == Doublets.Links.Null)
1838 //            mininumGap++;
1839 //        else if (_patternSequence[_patternPosition] == ZeroOrMany)
1840 //            maximumGap = long.MaxValue;
1841 //        else
1842 //            break;
1843 //    }
1844 //    if (maximumGap < mininumGap)
1845 //        maximumGap = mininumGap;
1846 //}
1847
1848 private bool PatternMatchCore(LinkIndex element)
1849 {
1850     if (_patternPosition >= _pattern.Count)
1851     {
1852         _patternPosition = -2;
1853         return false;
1854     }
1855     var currentPatternBlock = _pattern[_patternPosition];
1856     if (currentPatternBlock.Type == PatternBlockType.Gap)
1857     {
1858         //var currentMatchingBlockLength = (_sequencePosition -
1859 //    ↪ _lastMatchedBlockPosition);
1860         if (_sequencePosition < currentPatternBlock.Start)
1861         {
1862             _sequencePosition++;
1863             return true; // Двигаемся дальше
1864         }
1865         // Это последний блок
1866         if (_pattern.Count == _patternPosition + 1)
1867         {
1868             _patternPosition++;
1869             _sequencePosition = 0;
1870             return false; // Полное соответствие
1871         }
1872         else
1873         {
1874             if (_sequencePosition > currentPatternBlock.Stop)
1875             {
1876                 return false; // Соответствие невозможно
1877             }
1878         }
1879     }

```

```

1876         var nextPatternBlock = _pattern[_patternPosition + 1];
1877         if (_patternSequence[nextPatternBlock.Start] == element)
1878         {
1879             if (nextPatternBlock.Start < nextPatternBlock.Stop)
1880             {
1881                 _patternPosition++;
1882                 _sequencePosition = 1;
1883             }
1884             else
1885             {
1886                 _patternPosition += 2;
1887                 _sequencePosition = 0;
1888             }
1889         }
1890     }
1891 }
1892 else // currentPatternBlock.Type == PatternBlockType.Elements
1893 {
1894     var patternElementPosition = currentPatternBlock.Start + _sequencePosition;
1895     if (_patternSequence[patternElementPosition] != element)
1896     {
1897         return false; // Соответствие невозможно
1898     }
1899     if (patternElementPosition == currentPatternBlock.Stop)
1900     {
1901         _patternPosition++;
1902         _sequencePosition = 0;
1903     }
1904     else
1905     {
1906         _sequencePosition++;
1907     }
1908 }
1909 return true;
1910 //if (_patternSequence[_patternPosition] != element)
1911 //    return false;
1912 //else
1913 //{
1914 //    _sequencePosition++;
1915 //    _patternPosition++;
1916 //    return true;
1917 //}
1918 ///////
1919 //if (_filterPosition == _patternSequence.Length)
1920 //{
1921 //    _filterPosition = -2; // Длиннее чем нужно
1922 //    return false;
1923 //}
1924 //if (element != _patternSequence[_filterPosition])
1925 //{
1926 //    _filterPosition = -1;
1927 //    return false; // Начинается иначе
1928 //}
1929 //_filterPosition++;
1930 //if (_filterPosition == (_patternSequence.Length - 1))
1931 //    return false;
1932 //if (_filterPosition >= 0)
1933 //{
1934 //    if (element == _patternSequence[_filterPosition + 1])
1935 //        _filterPosition++;
1936 //    else
1937 //        return false;
1938 //}
1939 //if (_filterPosition < 0)
1940 //{
1941 //    if (element == _patternSequence[0])
1942 //        _filterPosition = 0;
1943 //}
1944 }
1945
1946 public void AddAllPatternMatchedToResults(IEnumerable<ulong> sequencesToMatch)
1947 {
1948     foreach (var sequenceToMatch in sequencesToMatch)
1949     {
1950         if (PatternMatch(sequenceToMatch))
1951         {
1952             _results.Add(sequenceToMatch);
1953         }
1954     }

```

```

1955     }
1956 }
1957
1958 #endregion
1959 }
1960 }

```

./Platform.Data.Doublets/Sequences/SequencesExtensions.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.Sequences
7  {
8      public static class SequencesExtensions
9      {
10         public static TLink Create<TLink>(this ILinks<TLink> sequences, IList<TLink[]>
            ↳ groupedSequence)
11         {
12             var finalSequence = new TLink[groupedSequence.Count];
13             for (var i = 0; i < finalSequence.Length; i++)
14             {
15                 var part = groupedSequence[i];
16                 finalSequence[i] = part.Length == 1 ? part[0] :
                    ↳ sequences.Create(part.ConvertToRestrictionsValues());
17             }
18             return sequences.Create(finalSequence.ConvertToRestrictionsValues());
19         }
20
21         public static IList<TLink> ToList<TLink>(this ILinks<TLink> sequences, TLink sequence)
22         {
23             var list = new List<TLink>();
24             var filler = new ListFiller<TLink, TLink>(list, sequences.Constants.Break);
25             sequences.Each(filler.AddAllValuesAndReturnConstant, new
                ↳ LinkAddress<TLink>(sequence));
26             return list;
27         }
28     }
29 }

```

./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         public bool ReadFullSequence { get; set; }
35     }

```

```

36 // TODO: Реализовать компактификацию при чтении
37 //public bool EnforceSingleSequenceVersionOnRead { get; set; }
38 //public bool UseRequestMarker { get; set; }
39 //public bool StoreRequestResults { get; set; }
40
41 public void InitOptions(ISynchronizedLinks<TLink> links)
42 {
43     if (UseSequenceMarker)
44     {
45         if (_equalityComparer.Equals(SequenceMarkerLink, links.Constants.Null))
46         {
47             SequenceMarkerLink = links.CreatePoint();
48         }
49         else
50         {
51             if (!links.Exists(SequenceMarkerLink))
52             {
53                 var link = links.CreatePoint();
54                 if (!_equalityComparer.Equals(link, SequenceMarkerLink))
55                 {
56                     throw new InvalidOperationException("Cannot recreate sequence marker
57                         ↪ link.");
58                 }
59             }
60             if (MarkedSequenceMatcher == null)
61             {
62                 MarkedSequenceMatcher = new MarkedSequenceCriterionMatcher<TLink>(links,
63                     ↪ SequenceMarkerLink);
64             }
65             var balancedVariantConverter = new BalancedVariantConverter<TLink>(links);
66             if (UseCompression)
67             {
68                 if (LinksToSequenceConverter == null)
69                 {
70                     ICounter<TLink, TLink> totalSequenceSymbolFrequencyCounter;
71                     if (UseSequenceMarker)
72                     {
73                         totalSequenceSymbolFrequencyCounter = new
74                             ↪ TotalMarkedSequenceSymbolFrequencyCounter<TLink>(links,
75                             ↪ MarkedSequenceMatcher);
76                     }
77                     else
78                     {
79                         totalSequenceSymbolFrequencyCounter = new
80                             ↪ TotalSequenceSymbolFrequencyCounter<TLink>(links);
81                     }
82                     var doubletFrequenciesCache = new LinkFrequenciesCache<TLink>(links,
83                         ↪ totalSequenceSymbolFrequencyCounter);
84                     var compressingConverter = new CompressingConverter<TLink>(links,
85                         ↪ balancedVariantConverter, doubletFrequenciesCache);
86                     LinksToSequenceConverter = compressingConverter;
87                 }
88             }
89             else
90             {
91                 if (LinksToSequenceConverter == null)
92                 {
93                     LinksToSequenceConverter = balancedVariantConverter;
94                 }
95             }
96             if (UseIndex && Index == null)
97             {
98                 Index = new SequenceIndex<TLink>(links);
99             }
100             if (Walker == null)
101             {
102                 Walker = new RightSequenceWalker<TLink>(links, new DefaultStack<TLink>());
103             }
104         }
105     }
106
107 public void ValidateOptions()
108 {
109     if (UseGarbageCollection && !UseSequenceMarker)
110     {
111         throw new NotSupportedException("To use garbage collection UseSequenceMarker
112             ↪ option must be on.");
113     }
114 }

```



```

106     }
107 }
108 }
109 }

```

./Platform.Data.Doublets/Sequences/SetFiller.cs

```

1  using System.Collections.Generic;
2  using System.Runtime.CompilerServices;
3
4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6  namespace Platform.Data.Doublets.Sequences
7  {
8      public class SetFiller<TElement, TReturnConstant>
9      {
10         protected readonly ISet<TElement> _set;
11         protected readonly TReturnConstant _returnConstant;
12
13         public SetFiller(ISet<TElement> set, TReturnConstant returnConstant)
14         {
15             _set = set;
16             _returnConstant = returnConstant;
17         }
18
19         public SetFiller(ISet<TElement> set) : this(set, default) { }
20
21         [MethodImpl(MethodImplOptions.AggressiveInlining)]
22         public void Add(TElement element) => _set.Add(element);
23
24         [MethodImpl(MethodImplOptions.AggressiveInlining)]
25         public bool AddAndReturnTrue(TElement element)
26         {
27             _set.Add(element);
28             return true;
29         }
30
31         [MethodImpl(MethodImplOptions.AggressiveInlining)]
32         public bool AddFirstAndReturnTrue(ICollection<TElement> collection)
33         {
34             _set.Add(collection[0]);
35             return true;
36         }
37
38         [MethodImpl(MethodImplOptions.AggressiveInlining)]
39         public TReturnConstant AddAndReturnConstant(TElement element)
40         {
41             _set.Add(element);
42             return _returnConstant;
43         }
44
45         [MethodImpl(MethodImplOptions.AggressiveInlining)]
46         public TReturnConstant AddFirstAndReturnConstant(ICollection<TElement> collection)
47         {
48             _set.Add(collection[0]);
49             return _returnConstant;
50         }
51     }
52 }

```

./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;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4  using Platform.Collections.Stacks;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7

```

```

8 namespace Platform.Data.Doublets.Sequences.Walkers
9 {
10     public class LeftSequenceWalker<TLink> : SequenceWalkerBase<TLink>
11     {
12         public LeftSequenceWalker(ILinks<TLink> links, IStack<TLink> stack, Func<TLink, bool>
13             ↪ isElement) : base(links, stack, isElement) { }
14
15         public LeftSequenceWalker(ILinks<TLink> links, IStack<TLink> stack) : base(links, stack,
16             ↪ links.IsPartialPoint) { }
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected override TLink GetNextElementAfterPop(TLink element) =>
20             ↪ Links.GetSource(element);
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         protected override TLink GetNextElementAfterPush(TLink element) =>
24             ↪ Links.GetTarget(element);
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         protected override IEnumerable<TLink> WalkContents(TLink element)
28         {
29             var parts = Links.GetLink(element);
30             var start = Links.Constants.IndexPart + 1;
31             for (var i = parts.Count - 1; i >= start; i--)
32             {
33                 var part = parts[i];
34                 if (IsElement(part))
35                 {
36                     yield return part;
37                 }
38             }
39         }
40     }
41 }

```

./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                 var nextArray = ArrayPool.Allocate<ulong>(length);

```

```

42         var nextArray = new TLink[length];
43     #endif
44     hasElements = false;
45     for (var i = 0; i < array.Length; i++)
46     {
47         var candidate = array[i];
48         if (!_equalityComparer.Equals(array[i], default))
49         {
50             continue;
51         }
52         var doubletOffset = i * 2;
53         if (!_isElement(candidate))
54         {
55             nextArray[doubletOffset] = candidate;
56         }
57         else
58         {
59             var link = Links.GetLink(candidate);
60             var linkSource = Links.GetSource(link);
61             var linkTarget = Links.GetTarget(link);
62             nextArray[doubletOffset] = linkSource;
63             nextArray[doubletOffset + 1] = linkTarget;
64             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;
2 using System.Collections.Generic;
3 using System.Runtime.CompilerServices;
4 using Platform.Collections.Stacks;
5
6 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8 namespace Platform.Data.Doublets.Sequences.Walkers
9 {
10     public class RightSequenceWalker<TLink> : SequenceWalkerBase<TLink>
11     {
12         public RightSequenceWalker(ILinks<TLink> links, IStack<TLink> stack, Func<TLink, bool>
13             ↪ isElement) : base(links, stack, isElement) { }
14
15         public RightSequenceWalker(ILinks<TLink> links, IStack<TLink> stack) : base(links,
16             ↪ stack, links.IsPartialPoint) { }
17
18         [MethodImpl(MethodImplOptions.AggressiveInlining)]
19         protected override TLink GetNextElementAfterPop(TLink element) =>
20             ↪ Links.GetTarget(element);
21
22         [MethodImpl(MethodImplOptions.AggressiveInlining)]
23         protected override TLink GetNextElementAfterPush(TLink element) =>
24             ↪ Links.GetSource(element);
25
26         [MethodImpl(MethodImplOptions.AggressiveInlining)]
27         protected override IEnumerable<TLink> WalkContents(TLink element)
28         {
29             var parts = Links.GetLink(element);
30             for (var i = Links.Constants.IndexPart + 1; i < parts.Count; i++)
31             {
32                 var part = parts[i];
33                 if (IsElement(part))
34                 {
35                     yield return part;
36                 }
37             }
38         }
39     }
40 }
```

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

```
1 using System;
2 using System.Collections.Generic;
3 using System.Runtime.CompilerServices;
4 using Platform.Collections.Stacks;
5
6 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8 namespace Platform.Data.Doublets.Sequences.Walkers
9 {
10     public abstract class SequenceWalkerBase<TLink> : LinksOperatorBase<TLink>,
11         ↪ ISequenceWalker<TLink>
12     {
13         private readonly IStack<TLink> _stack;
14         private readonly Func<TLink, bool> _isElement;
15
16         protected SequenceWalkerBase(ILinks<TLink> links, IStack<TLink> stack, Func<TLink, bool>
17             ↪ isElement) : base(links)
18         {
19             _stack = stack;
20             _isElement = isElement;
21         }
22
23         protected SequenceWalkerBase(ILinks<TLink> links, IStack<TLink> stack) : this(links,
24             ↪ stack, links.IsPartialPoint)
25         {
26         }
27
28         public IEnumerable<TLink> Walk(TLink sequence)
29         {
30             _stack.Clear();
31             var element = sequence;
32             if (IsElement(element))
33             {
34             }
35         }
36     }
37 }
```

```

31         yield return element;
32     }
33     else
34     {
35         while (true)
36         {
37             if (IsElement(element))
38             {
39                 if (_stack.IsEmpty)
40                 {
41                     break;
42                 }
43                 element = _stack.Pop();
44                 foreach (var output in WalkContents(element))
45                 {
46                     yield return output;
47                 }
48                 element = GetNextElementAfterPop(element);
49             }
50             else
51             {
52                 _stack.Push(element);
53                 element = GetNextElementAfterPush(element);
54             }
55         }
56     }
57 }
58
59 [MethodImpl(MethodImplOptions.AggressiveInlining)]
60 protected virtual bool IsElement(TLink elementLink) => _isElement(elementLink);
61
62 [MethodImpl(MethodImplOptions.AggressiveInlining)]
63 protected abstract TLink GetNextElementAfterPop(TLink element);
64
65 [MethodImpl(MethodImplOptions.AggressiveInlining)]
66 protected abstract TLink GetNextElementAfterPush(TLink element);
67
68 [MethodImpl(MethodImplOptions.AggressiveInlining)]
69 protected abstract IEnumerable<TLink> WalkContents(TLink element);
70
71 }

```

./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             }
39         }
40     }
41 }

```

```

37         _links.Delete(top);
38     }
39     return element;
40 }
41
42 public void Push(TLink element) => _links.Update(_stack, GetStackMarker(),
    ↪ _links.GetOrCreate(GetTop(), element));
43 }
44 }

```

./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.Doublets;
4  using Platform.Threading.Synchronization;
5
6  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
7
8  namespace Platform.Data.Doublets
9  {
10     /// <remarks>
11     /// TODO: Autogeneration of synchronized wrapper (decorator).
12     /// TODO: Try to unfold code of each method using IL generation for performance improvements.
13     /// TODO: Or even to unfold multiple layers of implementations.
14     /// </remarks>
15     public class SynchronizedLinks<TLinkAddress> : ISynchronizedLinks<TLinkAddress>
16     {
17         public LinksConstants<TLinkAddress> Constants { get; }
18         public ISynchronization SyncRoot { get; }
19         public ILinks<TLinkAddress> Sync { get; }
20         public ILinks<TLinkAddress> Unsync { get; }
21
22         public SynchronizedLinks(ILinks<TLinkAddress> links) : this(new
            ↪ ReaderWriterLockSynchronization(), links) { }
23
24         public SynchronizedLinks(ISynchronization synchronization, ILinks<TLinkAddress> links)
25         {
26             SyncRoot = synchronization;
27             Sync = this;
28             Unsync = links;
29             Constants = links.Constants;
30         }
31
32         public TLinkAddress Count(IList<TLinkAddress> restriction) =>
            ↪ SyncRoot.ExecuteReadOperation(restriction, Unsync.Count);
33         public TLinkAddress Each(Func<IList<TLinkAddress>, TLinkAddress> handler,
            ↪ IList<TLinkAddress> restrictions) => SyncRoot.ExecuteReadOperation(handler,
            ↪ restrictions, (handler1, restrictions1) => Unsync.Each(handler1, restrictions1));
34         public TLinkAddress Create(IList<TLinkAddress> restrictions) =>
            ↪ SyncRoot.ExecuteWriteOperation(restrictions, Unsync.Create);
35         public TLinkAddress Update(IList<TLinkAddress> restrictions, IList<TLinkAddress>
            ↪ substitution) => SyncRoot.ExecuteWriteOperation(restrictions, substitution,
            ↪ Unsync.Update);
36         public void Delete(IList<TLinkAddress> restrictions) =>
            ↪ SyncRoot.ExecuteWriteOperation(restrictions, Unsync.Delete);
37
38         //public T Trigger(IList<T> restriction, Func<IList<T>, IList<T>, T> matchedHandler,
            ↪ IList<T> substitution, Func<IList<T>, IList<T>, T> substitutedHandler)
39         //{
40         //    if (restriction != null && substitution != null &&
            ↪ !substitution.EqualTo(restriction))
41         //        return SyncRoot.ExecuteWriteOperation(restriction, matchedHandler,
            ↪ substitution, substitutedHandler, Unsync.Trigger);

```

```

42
43     // return SyncRoot.ExecuteReadOperation(restriction, matchedHandler, substitution,
    ↪ substitutedHandler, Unsync.Trigger);
44     //}
45 }
46 }

```

./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
9  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
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 LinksConstants<ulong> _constants =
19             ↪ Default<LinksConstants<ulong>>.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;
33             Source = values.Length > _constants.SourcePart ? values[_constants.SourcePart] :
34                 ↪ _constants.Null;
35             Target = values.Length > _constants.TargetPart ? values[_constants.TargetPart] :
36                 ↪ _constants.Null;
37         }
38
39         public UInt64Link(IList<ulong> values)
40         {
41             Index = values.Count > _constants.IndexPart ? values[_constants.IndexPart] :
42                 ↪ _constants.Null;
43             Source = values.Count > _constants.SourcePart ? values[_constants.SourcePart] :
44                 ↪ _constants.Null;
45             Target = values.Count > _constants.TargetPart ? values[_constants.TargetPart] :
46                 ↪ _constants.Null;
47         }
48
49         public UInt64Link(ulong index, ulong source, ulong target)
50         {
51             Index = index;
52             Source = source;
53             Target = target;
54         }
55
56         public UInt64Link(ulong source, ulong target)
57             : this(_constants.Null, source, target)
58         {
59             Source = source;
60             Target = target;
61         }
62
63         public static UInt64Link Create(ulong source, ulong target) => new UInt64Link(source,
64             ↪ target);
65
66         public override int GetHashCode() => (Index, Source, Target).GetHashCode();
67
68         public bool IsNull() => Index == _constants.Null
69             && Source == _constants.Null
70             && Target == _constants.Null;
71
72         public override bool Equals(object other) => other is UInt64Link &&
73             ↪ Equals((UInt64Link)other);
74     }
75 }

```

```

65
66 public bool Equals(UInt64Link other) => Index == other.Index
67                                     && Source == other.Source
68                                     && Target == other.Target;
69
70 public static string ToString(ulong index, ulong source, ulong target) => $"{index}:
    ↳ {source}->{target}";
71
72 public static string ToString(ulong source, ulong target) => $"{source}->{target}";
73
74 public static implicit operator ulong[] (UInt64Link link) => link.ToArray();
75
76 public static implicit operator UInt64Link(ulong[] linkArray) => new
    ↳ UInt64Link(linkArray);
77
78 public override string ToString() => Index == _constants.Null ? ToString(Source, Target)
    ↳ : ToString(Index, Source, Target);
79
80 #region IList
81
82 public ulong this[int index]
83 {
84     get
85     {
86         Ensure.OnDebug.ArgumentInRange(index, new Range<int>(0, Length - 1),
            ↳ nameof(index));
87         if (index == _constants.IndexPart)
88         {
89             return Index;
90         }
91         if (index == _constants.SourcePart)
92         {
93             return Source;
94         }
95         if (index == _constants.TargetPart)
96         {
97             return Target;
98         }
99         throw new NotSupportedException(); // Impossible path due to
            ↳ Ensure.ArgumentInRange
100     }
101     set => throw new NotSupportedException();
102 }
103
104 public int Count => Length;
105
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.OnDebug.ArgumentNotNull(array, nameof(array));
126     Ensure.OnDebug.ArgumentInRange(arrayIndex, new Range<int>(0, array.Length - 1),
        ↳ nameof(arrayIndex));
127     if (arrayIndex + Length > array.Length)
128     {
129         throw new ArgumentException();
130     }
131     array[arrayIndex++] = Index;
132     array[arrayIndex++] = Source;
133     array[arrayIndex] = Target;
134 }
135
136 public bool Remove(ulong item) => Throw.A.NotSupportedExceptionAndReturn<bool>();
137

```



```

138     public int IndexOf(ulong item)
139     {
140         if (Index == item)
141         {
142             return _constants.IndexPart;
143         }
144         if (Source == item)
145         {
146             return _constants.SourcePart;
147         }
148         if (Target == item)
149         {
150             return _constants.TargetPart;
151         }
152         return -1;
153     }
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 }

```

./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.Exceptions;
6  using Platform.Data.Doublets.Unicode;
7
8  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Data.Doublets
11 {
12     public static class UInt64LinksExtensions
13     {
14         public static readonly LinksConstants<ulong> Constants =
15             ↪ Default<LinksConstants<ulong>>.Instance;
16
17         public static void UseUnicode(this ILinks<ulong> links) => UnicodeMap.InitNew(links);
18
19         public static void EnsureEachLinkExists(this ILinks<ulong> links, IList<ulong> sequence)
20         {
21             if (sequence == null)
22             {
23                 return;
24             }
25             for (var i = 0; i < sequence.Count; i++)
26             {
27                 if (!links.Exists(sequence[i]))
28                 {
29                     throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
30                         ↪ $"sequence[{i}]");
31                 }
32             }
33         }
34
35         public static void EnsureEachLinkIsAnyOrExists(this ILinks<ulong> links, IList<ulong>
36             ↪ sequence)
37         {
38             if (sequence == null)
39             {

```

```

37         return;
38     }
39     for (var i = 0; i < sequence.Count; i++)
40     {
41         if (sequence[i] != Constants.Any && !links.Exists(sequence[i]))
42         {
43             throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
44                 ↪ $"sequence[{i}]");
45         }
46     }
47 }
48 public static bool AnyLinkIsAny(this ILinks<ulong> links, params ulong[] sequence)
49 {
50     if (sequence == null)
51     {
52         return false;
53     }
54     var constants = links.Constants;
55     for (var i = 0; i < sequence.Length; i++)
56     {
57         if (sequence[i] == constants.Any)
58         {
59             return true;
60         }
61     }
62     return false;
63 }
64
65 public static string FormatStructure(this ILinks<ulong> links, ulong linkIndex,
66     ↪ Func<UInt64Link, bool> isElement, bool renderIndex = false, bool renderDebug = false)
67 {
68     var sb = new StringBuilder();
69     var visited = new HashSet<ulong>();
70     links.AppendStructure(sb, visited, linkIndex, isElement, (innerSb, link) =>
71         ↪ innerSb.Append(link.Index), renderIndex, renderDebug);
72     return sb.ToString();
73 }
74
75 public static string FormatStructure(this ILinks<ulong> links, ulong linkIndex,
76     ↪ Func<UInt64Link, bool> isElement, Action<StringBuilder, UInt64Link> appendElement,
77     ↪ bool renderIndex = false, bool renderDebug = false)
78 {
79     var sb = new StringBuilder();
80     var visited = new HashSet<ulong>();
81     links.AppendStructure(sb, visited, linkIndex, isElement, appendElement, renderIndex,
82         ↪ renderDebug);
83     return sb.ToString();
84 }
85
86 public static void AppendStructure(this ILinks<ulong> links, StringBuilder sb,
87     ↪ HashSet<ulong> visited, ulong linkIndex, Func<UInt64Link, bool> isElement,
88     ↪ Action<StringBuilder, UInt64Link> appendElement, bool renderIndex = false, bool
89     ↪ renderDebug = false)
90 {
91     if (sb == null)
92     {
93         throw new ArgumentNullException(nameof(sb));
94     }
95     if (linkIndex == Constants.Null || linkIndex == Constants.Any || linkIndex ==
96         ↪ Constants.Itself)
97     {
98         return;
99     }
100     if (links.Exists(linkIndex))
101     {
102         if (visited.Add(linkIndex))
103         {
104             sb.Append('(');
105             var link = new UInt64Link(links.GetLink(linkIndex));
106             if (renderIndex)
107             {
108                 sb.Append(link.Index);
109                 sb.Append(':');
110             }
111             if (link.Source == link.Index)
112             {
113                 sb.Append(link.Index);
114             }
115         }
116     }
117 }

```

```

105     }
106     else
107     {
108         var source = new UInt64Link(links.GetLink(link.Source));
109         if (isElement(source))
110         {
111             appendElement(sb, source);
112         }
113         else
114         {
115             links.AppendStructure(sb, visited, source.Index, isElement,
116                 ↪ appendElement, renderIndex);
117         }
118     }
119     sb.Append(' ');
120     if (link.Target == link.Index)
121     {
122         sb.Append(link.Index);
123     }
124     else
125     {
126         var target = new UInt64Link(links.GetLink(link.Target));
127         if (isElement(target))
128         {
129             appendElement(sb, target);
130         }
131         else
132         {
133             links.AppendStructure(sb, visited, target.Index, isElement,
134                 ↪ appendElement, renderIndex);
135         }
136     }
137     sb.Append(')');
138 }
139 else
140 {
141     if (renderDebug)
142     {
143         sb.Append('*');
144     }
145     sb.Append(linkIndex);
146 }
147 }
148 }
149 else
150 {
151     if (renderDebug)
152     {
153         sb.Append('~');
154     }
155     sb.Append(linkIndex);
156 }
157 }
158 }
159 }
160 }

```

./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 using Platform.Exceptions;
14
15 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
16
17 namespace Platform.Data.Doublets
18 {
19     public class UInt64LinksTransactionsLayer : LinksDisposableDecoratorBase //-V3073
20     {
21         /// <remarks>
22         /// Альтернативные варианты хранения трансформации (элемента транзакции):
23         ///

```

```

24     /// private enum TransitionType
25     /// {
26     ///     Creation,
27     ///     UpdateOf,
28     ///     UpdateTo,
29     ///     Deletion
30     /// }
31     ///
32     /// private struct Transition
33     /// {
34     ///     public ulong TransactionId;
35     ///     public UniqueTimestamp Timestamp;
36     ///     public TransactionItemType Type;
37     ///     public Link Source;
38     ///     public Link Linker;
39     ///     public Link Target;
40     /// }
41     ///
42     /// Или
43     ///
44     /// public struct TransitionHeader
45     /// {
46     ///     public ulong TransactionIdCombined;
47     ///     public ulong TimestampCombined;
48     ///
49     ///     public ulong TransactionId
50     ///     {
51     ///         get
52     ///         {
53     ///             return (ulong) mask & TransactionIdCombined;
54     ///         }
55     ///     }
56     ///
57     ///     public UniqueTimestamp Timestamp
58     ///     {
59     ///         get
60     ///         {
61     ///             return (UniqueTimestamp)mask & TransactionIdCombined;
62     ///         }
63     ///     }
64     ///
65     ///     public TransactionItemType Type
66     ///     {
67     ///         get
68     ///         {
69     ///             // Использовать по одному биту из TransactionId и Timestamp,
70     ///             // для значения в 2 бита, которое представляет тип операции
71     ///             throw new NotImplementedException();
72     ///         }
73     ///     }
74     /// }
75     ///
76     /// private struct Transition
77     /// {
78     ///     public TransitionHeader Header;
79     ///     public Link Source;
80     ///     public Link Linker;
81     ///     public Link Target;
82     /// }
83     ///
84     </remarks>
85 public struct Transition
86 {
87     public static readonly long Size = Structure<Transition>.Size;
88
89     public readonly ulong TransactionId;
90     public readonly UInt64Link Before;
91     public readonly UInt64Link After;
92     public readonly Timestamp Timestamp;
93
94     public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
↵ transactionId, UInt64Link before, UInt64Link after)
95     {
96         TransactionId = transactionId;
97         Before = before;
98         After = after;
99         Timestamp = uniqueTimestampFactory.Create();
100     }

```

```

101 public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
102     ↳ transactionId, UInt64Link before)
103     : this(uniqueTimestampFactory, transactionId, before, default)
104 {
105 }
106
107 public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong transactionId)
108     : this(uniqueTimestampFactory, transactionId, default, default)
109 {
110 }
111
112 public override string ToString() => $"{Timestamp} {TransactionId}: {Before} =>
113     ↳ {After}";
114 }
115
116 /// <remarks>
117 /// Другие варианты реализации транзакций (атомарности):
118 /// 1. Разделение хранения значения связи ((Source Target) или (Source Linker
119     ↳ Target)) и индексов.
120 /// 2. Хранение трансформаций/операций в отдельном хранилище Links, но дополнительно
121     ↳ потребуется решить вопрос
122     ↳ со ссылками на внешние идентификаторы, или как-то иначе решить вопрос с
123     ↳ пересечениями идентификаторов.
124 ///
125 /// Где хранить промежуточный список транзакций?
126 ///
127 /// В оперативной памяти:
128 /// Минусы:
129 /// 1. Может усложнить систему, если она будет функционировать самостоятельно,
130     ↳ так как нужно отдельно выделять память под список трансформаций.
131 /// 2. Выделенной оперативной памяти может не хватить, в том случае,
132     ↳ если транзакция использует слишком много трансформаций.
133     ↳ -> Можно использовать жёсткий диск для слишком длинных транзакций.
134     ↳ -> Максимальный размер списка трансформаций можно ограничить / задать
135     ↳ константой.
136 /// 3. При подтверждении транзакции (Commit) все трансформации записываются разом
137     ↳ создавая задержку.
138 ///
139 /// На жёстком диске:
140 /// Минусы:
141 /// 1. Длительный отклик, на запись каждой трансформации.
142 /// 2. Лог транзакций дополнительно наполняется отменёнными транзакциями.
143     ↳ -> Это может решаться упаковкой/исключением дублирующих операций.
144     ↳ -> Также это может решаться тем, что короткие транзакции вообще
145     ↳ не будут записываться в случае отката.
146 /// 3. Перед тем как выполнять отмену операций транзакции нужно дождаться пока все
147     ↳ операции (трансформации)
148     ↳ будут записаны в лог.
149 /// </remarks>
150 public class Transaction : DisposableBase
151 {
152     private readonly Queue<Transition> _transitions;
153     private readonly UInt64LinksTransactionsLayer _layer;
154     public bool IsCommitted { get; private set; }
155     public bool IsReverted { get; private set; }
156
157     public Transaction(UInt64LinksTransactionsLayer layer)
158     {
159         _layer = layer;
160         if (_layer._currentTransactionId != 0)
161         {
162             throw new NotSupportedException("Nested transactions not supported.");
163         }
164         IsCommitted = false;
165         IsReverted = false;
166         _transitions = new Queue<Transition>();
167         SetCurrentTransaction(layer, this);
168     }
169
170     public void Commit()
171     {
172         EnsureTransactionAllowsWriteOperations(this);
173         while (_transitions.Count > 0)
174         {
175             var transition = _transitions.Dequeue();
176             _layer._transitions.Enqueue(transition);
177         }
178     }
179 }

```

```

171     }
172     _layer._lastCommittedTransactionId = _layer._currentTransactionId;
173     IsCommitted = true;
174 }
175
176 private void Revert()
177 {
178     EnsureTransactionAllowsWriteOperations(this);
179     var transitionsToRevert = new Transition[_transitions.Count];
180     _transitions.CopyTo(transitionsToRevert, 0);
181     for (var i = transitionsToRevert.Length - 1; i >= 0; i--)
182     {
183         _layer.RevertTransition(transitionsToRevert[i]);
184     }
185     IsReverted = true;
186 }
187
188 public static void SetCurrentTransaction(UInt64LinksTransactionsLayer layer,
189 ↪ Transaction transaction)
190 {
191     layer._currentTransactionId = layer._lastCommittedTransactionId + 1;
192     layer._currentTransactionTransitions = transaction._transitions;
193     layer._currentTransaction = transaction;
194 }
195
196 public static void EnsureTransactionAllowsWriteOperations(Transaction transaction)
197 {
198     if (transaction.IsReverted)
199     {
200         throw new InvalidOperationException("Transation is reverted.");
201     }
202     if (transaction.IsCommitted)
203     {
204         throw new InvalidOperationException("Transation is committed.");
205     }
206 }
207
208 protected override void Dispose(bool manual, bool wasDisposed)
209 {
210     if (!wasDisposed && _layer != null && !_layer.IsDisposed)
211     {
212         if (!IsCommitted && !IsReverted)
213         {
214             Revert();
215         }
216         _layer.ResetCurrentTransation();
217     }
218 }
219
220 public static readonly TimeSpan DefaultPushDelay = TimeSpan.FromSeconds(0.1);
221
222 private readonly string _logAddress;
223 private readonly FileStream _log;
224 private readonly Queue<Transition> _transitions;
225 private readonly UniqueTimestampFactory _uniqueTimestampFactory;
226 private Task _transitionsPusher;
227 private Transition _lastCommittedTransition;
228 private ulong _currentTransactionId;
229 private Queue<Transition> _currentTransactionTransitions;
230 private Transaction _currentTransaction;
231 private ulong _lastCommittedTransactionId;
232
233 public UInt64LinksTransactionsLayer(ILinks<ulong> links, string logAddress)
234 : base(links)
235 {
236     if (string.IsNullOrEmpty(logAddress))
237     {
238         throw new ArgumentNullException(nameof(logAddress));
239     }
240     // В первой строке файла хранится последняя закоммиченную транзакцию.
241     // При запуске это используется для проверки удачного закрытия файла лога.
242     // In the first line of the file the last committed transaction is stored.
243     // On startup, this is used to check that the log file is successfully closed.
244     var lastCommittedTransition = FileHelpers.ReadFirstOrDefault<Transition>(logAddress);
245     var lastWrittenTransition = FileHelpers.ReadLastOrDefault<Transition>(logAddress);
246     if (!lastCommittedTransition.Equals(lastWrittenTransition))
247     {
248         Dispose();

```

```

249         throw new NotSupportedException("Database is damaged, autorecovery is not
250             ↳ supported yet.");
251     }
252     if (lastCommittedTransition.Equals(default(Transition)))
253     {
254         FileHelpers.WriteFirst(logAddress, lastCommittedTransition);
255     }
256     _lastCommittedTransition = lastCommittedTransition;
257     // TODO: Think about a better way to calculate or store this value
258     var allTransitions = FileHelpers.ReadAll<Transition>(logAddress);
259     _lastCommittedTransactionId = allTransitions.Max(x => x.TransactionId);
260     _uniqueTimestampFactory = new UniqueTimestampFactory();
261     _logAddress = logAddress;
262     _log = FileHelpers.Append(logAddress);
263     _transitions = new Queue<Transition>();
264     _transitionsPusher = new Task(TransitionsPusher);
265     _transitionsPusher.Start();
266 }
267
268 public IList<ulong> GetLinkValue(ulong link) => Links.GetLink(link);
269
270 public override ulong Create(IList<ulong> restrictions)
271 {
272     var createdLinkIndex = Links.Create();
273     var createdLink = new UInt64Link(Links.GetLink(createdLinkIndex));
274     CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
275         ↳ default, createdLink));
276     return createdLinkIndex;
277 }
278
279 public override ulong Update(IList<ulong> restrictions, IList<ulong> substitution)
280 {
281     var linkIndex = restrictions[Constants.IndexPart];
282     var beforeLink = new UInt64Link(Links.GetLink(linkIndex));
283     linkIndex = Links.Update(restrictions, substitution);
284     var afterLink = new UInt64Link(Links.GetLink(linkIndex));
285     CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
286         ↳ beforeLink, afterLink));
287     return linkIndex;
288 }
289
290 public override void Delete(IList<ulong> restrictions)
291 {
292     var link = restrictions[Constants.IndexPart];
293     var deletedLink = new UInt64Link(Links.GetLink(link));
294     Links.Delete(link);
295     CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
296         ↳ deletedLink, default));
297 }
298
299 [MethodImpl(MethodImplOptions.AggressiveInlining)]
300 private Queue<Transition> GetCurrentTransitions() => _currentTransactionTransitions ??
301     ↳ _transitions;
302
303 private void CommitTransition(Transition transition)
304 {
305     if (_currentTransaction != null)
306     {
307         Transaction.EnsureTransactionAllowsWriteOperations(_currentTransaction);
308     }
309     var transitions = GetCurrentTransitions();
310     transitions.Enqueue(transition);
311 }
312
313 private void RevertTransition(Transition transition)
314 {
315     if (transition.After.IsNull()) // Revert Deletion with Creation
316     {
317         Links.Create();
318     }
319     else if (transition.Before.IsNull()) // Revert Creation with Deletion
320     {
321         Links.Delete(transition.After.Index);
322     }
323     else // Revert Update
324     {
325         Links.Update(new[] { transition.After.Index, transition.Before.Source,
326             ↳ transition.Before.Target });
327     }
328 }

```

```

321     }
322 }
323
324 private void ResetCurrentTransation()
325 {
326     _currentTransactionId = 0;
327     _currentTransactionTransitions = null;
328     _currentTransaction = null;
329 }
330
331 private void PushTransitions()
332 {
333     if (_log == null || _transitions == null)
334     {
335         return;
336     }
337     for (var i = 0; i < _transitions.Count; i++)
338     {
339         var transition = _transitions.Dequeue();
340
341         _log.Write(transition);
342         _lastCommitedTransition = 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)
363         {
364             _transitionsPusher = null;
365             pusher.Wait();
366         }
367         if (_transitions != null)
368         {
369             PushTransitions();
370         }
371         _log.DisposeIfPossible();
372         FileHelpers.WriteFirst(_logAddress, _lastCommitedTransition);
373     }
374     catch (Exception ex)
375     {
376         ex.Ignore();
377     }
378 }
379
380 #region DisposalBase
381
382 protected override void Dispose(bool manual, bool wasDisposed)
383 {
384     if (!wasDisposed)
385     {
386         DisposeTransitions();
387     }
388     base.Dispose(manual, wasDisposed);
389 }
390
391 #endregion
392 }
393

```

./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>,
        ↪ IConverter<char, TLink>
9     {
10         private readonly IConverter<TLink> _addressToNumberConverter;
11         private readonly TLink _unicodeSymbolMarker;
12
13         public CharToUnicodeSymbolConverter(ILinks<TLink> links, IConverter<TLink>
        ↪ addressToNumberConverter, TLink unicodeSymbolMarker) : base(links)
14         {
15             _addressToNumberConverter = addressToNumberConverter;
16             _unicodeSymbolMarker = unicodeSymbolMarker;
17         }
18
19         public TLink Convert(char source)
20         {
21             var unaryNumber = _addressToNumberConverter.Convert((Integer<TLink>)source);
22             return Links.GetOrCreate(unaryNumber, _unicodeSymbolMarker);
23         }
24     }
25 }

```

./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>,
        ↪ IConverter<string, TLink>
10    {
11        private readonly IConverter<char, TLink> _charToUnicodeSymbolConverter;
12        private readonly ISequenceIndex<TLink> _index;
13        private readonly IConverter<IList<TLink>, TLink> _listToSequenceLinkConverter;
14        private readonly TLink _unicodeSequenceMarker;
15
16        public StringToUnicodeSequenceConverter(ILinks<TLink> links, IConverter<char, TLink>
        ↪ charToUnicodeSymbolConverter, ISequenceIndex<TLink> index, IConverter<IList<TLink>,
        ↪ 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 TLink[source.Length];
27            for (int i = 0; i < source.Length; i++)
28            {
29                elements[i] = _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     }
17 }

```

```

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 // 0 - null link
60 // 1 - nil character (0 character)
61 // ...
62 // 65536 (0(1) + 65535 = 65536 possible values)
63
64 [MethodImpl(MethodImplOptions.AggressiveInlining)]
65 public static ulong FromCharToLink(char character) => (ulong)character + 1;
66
67 [MethodImpl(MethodImplOptions.AggressiveInlining)]
68 public static char FromLinkToChar(ulong link) => (char)(link - 1);
69
70 [MethodImpl(MethodImplOptions.AggressiveInlining)]
71 public static bool IsCharLink(ulong link) => link <= MapSize;
72
73 public static string FromLinksToString(IList<ulong> linksList)
74 {
75     var sb = new StringBuilder();
76     for (int i = 0; i < linksList.Count; i++)
77     {
78         sb.Append(FromLinkToChar(linksList[i]));
79     }
80     return sb.ToString();
81 }
82
83 public static string FromSequenceLinkToString(ulong link, ILinks<ulong> links)
84 {
85     var sb = new StringBuilder();
86     if (links.Exists(link))
87     {
88         StopableSequenceWalker.WalkRight(link, links.GetSource, links.GetTarget,
89             x => x <= MapSize || links.GetSource(x) == x || links.GetTarget(x) == x,
90             ↪ element =>
91             {
92                 sb.Append(FromLinkToChar(element));
93                 return true;
94             }
95     }
96 }

```

```

92     }
93     return sb.ToString();
94 }
95
96 public static ulong[] FromCharsToLinkArray(char[] chars) => FromCharsToLinkArray(chars,
    ↪ chars.Length);
97
98 public static ulong[] FromCharsToLinkArray(char[] chars, int count)
99 {
100     // char array to ulong array
101     var linksSequence = new ulong[count];
102     for (var i = 0; i < count; i++)
103     {
104         linksSequence[i] = FromCharToLink(chars[i]);
105     }
106     return linksSequence;
107 }
108
109 public static ulong[] FromStringToLinkArray(string sequence)
110 {
111     // char array to ulong array
112     var linksSequence = new ulong[sequence.Length];
113     for (var i = 0; i < sequence.Length; i++)
114     {
115         linksSequence[i] = FromCharToLink(sequence[i]);
116     }
117     return linksSequence;
118 }
119
120 public static List<ulong[]> FromStringToLinkArrayGroups(string sequence)
121 {
122     var result = new List<ulong[]>();
123     var offset = 0;
124     while (offset < sequence.Length)
125     {
126         var currentCategory = CharUnicodeInfo.GetUnicodeCategory(sequence[offset]);
127         var relativeLength = 1;
128         var absoluteLength = offset + relativeLength;
129         while (absoluteLength < sequence.Length &&
130             currentCategory ==
131             ↪ CharUnicodeInfo.GetUnicodeCategory(sequence[absoluteLength]))
132         {
133             relativeLength++;
134             absoluteLength++;
135         }
136         // char array to ulong array
137         var innerSequence = new ulong[relativeLength];
138         var maxLength = offset + relativeLength;
139         for (var i = offset; i < maxLength; i++)
140         {
141             innerSequence[i - offset] = FromCharToLink(sequence[i]);
142         }
143         result.Add(innerSequence);
144         offset += relativeLength;
145     }
146     return result;
147 }
148
149 public static List<ulong[]> FromLinkArrayToLinkArrayGroups(ulong[] array)
150 {
151     var result = new List<ulong[]>();
152     var offset = 0;
153     while (offset < array.Length)
154     {
155         var relativeLength = 1;
156         if (array[offset] <= LastCharLink)
157         {
158             var currentCategory =
159             ↪ CharUnicodeInfo.GetUnicodeCategory(FromLinkToChar(array[offset]));
160             var absoluteLength = offset + relativeLength;
161             while (absoluteLength < array.Length &&
162                 array[absoluteLength] <= LastCharLink &&
163                 currentCategory == CharUnicodeInfo.GetUnicodeCategory(FromLinkToChar(
164                 ↪ array[absoluteLength])))
165             {
166                 relativeLength++;
167                 absoluteLength++;
168             }
169         }
170     }
171     return result;
172 }

```

```

167         else
168         {
169             var absoluteLength = offset + relativeLength;
170             while (absoluteLength < array.Length && array[absoluteLength] > LastCharLink)
171             {
172                 relativeLength++;
173                 absoluteLength++;
174             }
175         }
176         // copy array
177         var innerSequence = new ulong[relativeLength];
178         var maxLength = offset + relativeLength;
179         for (var i = offset; i < maxLength; i++)
180         {
181             innerSequence[i - offset] = array[i];
182         }
183         result.Add(innerSequence);
184         offset += relativeLength;
185     }
186     return result;
187 }
188 }
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 }

```

./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);

```

```

30         var charArray = _sequenceWalker.Walk(sequence).Select(_unicodeSymbolToCharConverter.
31             ↪ Convert).ToArray();
32         return new string(charArray);
33     }
34 }

```

./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
15        public UnicodeSymbolToCharConverter(ILinks<TLink> links, IConverter<TLink>
16            ↪ numberToAddressConverter, ICriterionMatcher<TLink> unicodeSymbolCriterionMatcher) :
17            ↪ base(links)
18        {
19            _numberToAddressConverter = numberToAddressConverter;
20            _unicodeSymbolCriterionMatcher = unicodeSymbolCriterionMatcher;
21        }
22
23        public char Convert(TLink source)
24        {
25            if (!_unicodeSymbolCriterionMatcher.IsMatched(source))
26            {
27                throw new ArgumentOutOfRangeException(nameof(source), source, "Specified link is
28                    ↪ not a unicode symbol.");
29            }
30            return (char)(ushort)(Integer<TLink>)_numberToAddressConverter.Convert(Links.GetSour
31                ↪ ce(source));
32        }
33    }
34 }

```

./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         private 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/EqualityTests.cs

```

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

```

```

25 {
26     const int N = 1000000;
27
28     ulong x = 10;
29     ulong y = 500;
30
31     bool result = false;
32
33     var ts1 = Performance.Measure(() =>
34     {
35         for (int i = 0; i < N; i++)
36         {
37             result = Equals1(x, y);
38         }
39     });
40
41     var ts2 = Performance.Measure(() =>
42     {
43         for (int i = 0; i < N; i++)
44         {
45             result = Equals2(x, y);
46         }
47     });
48
49     var ts3 = Performance.Measure(() =>
50     {
51         for (int i = 0; i < N; i++)
52         {
53             result = Equals3(x, y);
54         }
55     });
56
57     var equalityComparer1 = EqualityComparer<ulong>.Default;
58
59     var ts4 = Performance.Measure(() =>
60     {
61         for (int i = 0; i < N; i++)
62         {
63             result = equalityComparer1.Equals(x, y);
64         }
65     });
66
67     var equalityComparer2 = new UInt64EqualityComparer();
68
69     var ts5 = Performance.Measure(() =>
70     {
71         for (int i = 0; i < N; i++)
72         {
73             result = equalityComparer2.Equals(x, y);
74         }
75     });
76
77     Func<ulong, ulong, bool> equalityComparer3 = equalityComparer2.Equals;
78
79     var ts6 = Performance.Measure(() =>
80     {
81         for (int i = 0; i < N; i++)
82         {
83             result = equalityComparer3(x, y);
84         }
85     });
86
87     var comparer = Comparer<ulong>.Default;
88
89     var ts7 = Performance.Measure(() =>
90     {
91         for (int i = 0; i < N; i++)
92         {
93             result = comparer.Compare(x, y) == 0;
94         }
95     });
96
97     Assert.True(ts2 < ts1);
98     Assert.True(ts3 < ts2);
99     Assert.True(ts5 < ts4);
100    Assert.True(ts5 < ts6);
101
102    Console.WriteLine($"{{ts1}} {{ts2}} {{ts3}} {{ts4}} {{ts5}} {{ts6}} {{ts7}} {{result}}");
103 }

```

```
104     }
105 }
```

./Platform.Data.Doublets.Tests/GenericLinksTests.cs

```
1  using System;
2  using Xunit;
3  using Platform.Reflection;
4  using Platform.Memory;
5  using Platform.Scopes;
6  using Platform.Data.Doublets.ResizableDirectMemory;
7
8  namespace Platform.Data.Doublets.Tests
9  {
10     public unsafe static class GenericLinksTests
11     {
12         [Fact]
13         public static void CRUDTest()
14         {
15             Using<byte>(links => links.TestCRUDOperations());
16             Using<ushort>(links => links.TestCRUDOperations());
17             Using<uint>(links => links.TestCRUDOperations());
18             Using<ulong>(links => links.TestCRUDOperations());
19         }
20
21         [Fact]
22         public static void RawNumbersCRUDTest()
23         {
24             Using<byte>(links => links.TestRawNumbersCRUDOperations());
25             Using<ushort>(links => links.TestRawNumbersCRUDOperations());
26             Using<uint>(links => links.TestRawNumbersCRUDOperations());
27             Using<ulong>(links => links.TestRawNumbersCRUDOperations());
28         }
29
30         [Fact]
31         public static void MultipleRandomCreationsAndDeletionsTest()
32         {
33             //if (!RuntimeInformation.IsOSPlatform(OSPlatform.Linux))
34             //{
35                 // Using<byte>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution(
36                 //     ↪ ).TestMultipleRandomCreationsAndDeletions(16)); // Cannot use more because
37                 //     ↪ current implementation of tree cuts out 5 bits from the address space.
38                 // Using<ushort>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolutio
39                 //     ↪ n().TestMultipleRandomCreationsAndDeletions(100));
40                 // Using<uint>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution(
41                 //     ↪ ).TestMultipleRandomCreationsAndDeletions(100));
42                 //}
43             Using<ulong>(links => links.DecorateWithAutomaticUniquenessAndUsagesResolution().Tes
44             ↪ tMultipleRandomCreationsAndDeletions(100));
45         }
46
47         private static void Using<TLink>(Action<ILinks<TLink>> action)
48         {
49             //using (var scope = new Scope<Types<HeapResizableDirectMemory,
50             //     ↪ ResizableDirectMemoryLinks<TLink>>>())
51             //{
52                 // action(scope.Use<ILinks<TLink>>());
53             //}
54             using (var memory = new HeapResizableDirectMemory())
55             {
56                 Unsafe.MemoryBlock.Zero((void*)memory.Pointer, memory.ReservedCapacity); // Bug
57                 ↪ workaround
58                 using (var links = new ResizableDirectMemoryLinks<TLink>(memory))
59                 {
60                     action(links);
61                 }
62             }
63         }
64     }
65 }
```

./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
66         [Fact]
67         public static void DictionaryBasedFrequencyStoredOptimalVariantSequenceTest()
68         {
69             using (var scope = new TempLinksTestScope(useSequences: false))
70             {
71                 var links = scope.Links;
72
73                 links.UseUnicode();
74
75                 var sequence = UnicodeMap.FromStringToLinkArray(SequenceExample);
76
77                 var linksToFrequencies = new Dictionary<ulong, ulong>();
78
79                 var totalSequenceSymbolFrequencyCounter = new
80                     ↪ TotalSequenceSymbolFrequencyCounter<ulong>(links);
81
82                 var linkFrequenciesCache = new LinkFrequenciesCache<ulong>(links,
83                     ↪ totalSequenceSymbolFrequencyCounter);
84
85                 var index = new
86                     ↪ CachedFrequencyIncrementingSequenceIndex<ulong>(linkFrequenciesCache);

```

```

72     var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFrequencyNumberConverter(
73         ↪ ncyNumberConverter<ulong>(linkFrequenciesCache);
74
75     var sequenceToItsLocalElementLevelsConverter = new
76         ↪ SequenceToItsLocalElementLevelsConverter<ulong>(links,
77         ↪ linkToItsFrequencyNumberConverter);
78
79     var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
80         ↪ sequenceToItsLocalElementLevelsConverter);
81
82     var sequences = new Sequences.Sequences(links, new SequencesOptions<ulong>() {
83         ↪ Walker = new LeveledSequenceWalker<ulong>(links) });
84
85     ExecuteTest(sequences, sequence, sequenceToItsLocalElementLevelsConverter,
86         ↪ index, optimalVariantConverter);
87 }
88
89 private static void ExecuteTest(Sequences.Sequences sequences, ulong[] sequence,
90     ↪ SequenceToItsLocalElementLevelsConverter<ulong>
91     ↪ sequenceToItsLocalElementLevelsConverter, ISequenceIndex<ulong> index,
92     ↪ OptimalVariantConverter<ulong> optimalVariantConverter)
93 {
94     index.Add(sequence);
95
96     var optimalVariant = optimalVariantConverter.Convert(sequence);
97
98     var readSequence1 = sequences.ToList(optimalVariant);
99
100    Assert.True(sequence.SequenceEqual(readSequence1));
101 }
102
103 }
104
105 }

```

./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         }
49     }
50 }

```

```

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

```

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

```

1  using System.IO;
2  using Xunit;
3  using Platform.Singletons;
4  using Platform.Memory;
5  using Platform.Data.Doublets.ResizableDirectMemory;
6
7  namespace Platform.Data.Doublets.Tests
8  {
9      public static class ResizableDirectMemoryLinksTests
10     {
11         private static readonly LinksConstants<ulong> _constants =
12             ↳ Default<LinksConstants<ulong>>.Instance;
13
14         [Fact]
15         public static void BasicFileMappedMemoryTest()
16         {
17             var tempFilename = Path.GetTempFileName();
18             using (var memoryAdapter = new UInt64ResizableDirectMemoryLinks(tempFilename))
19             {
20                 memoryAdapter.TestBasicMemoryOperations();
21             }
22             File.Delete(tempFilename);
23
24             [Fact]
25             public static void BasicHeapMemoryTest()
26             {
27                 using (var memory = new
28                     ↳ HeapResizableDirectMemory(UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
29                 using (var memoryAdapter = new UInt64ResizableDirectMemoryLinks(memory,
30                     ↳ UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
31                 {
32                     memoryAdapter.TestBasicMemoryOperations();
33                 }
34
35                 private static void TestBasicMemoryOperations(this ILinks<ulong> memoryAdapter)
36                 {
37                     var link = memoryAdapter.Create();
38                     memoryAdapter.Delete(link);
39                 }
40
41                 [Fact]
42                 public static void NonexistentReferencesHeapMemoryTest()
43                 {
44                     using (var memory = new
45                         ↳ HeapResizableDirectMemory(UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
46                     using (var memoryAdapter = new UInt64ResizableDirectMemoryLinks(memory,
47                         ↳ UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
48                     {
49                         memoryAdapter.TestNonexistentReferences();
50                     }
51
52                     private static void TestNonexistentReferences(this ILinks<ulong> memoryAdapter)
53                     {
54                         var link = memoryAdapter.Create();
55                         memoryAdapter.Update(link, ulong.MaxValue, ulong.MaxValue);

```

```

54         var resultLink = _constants.Null;
55         memoryAdapter.Each(foundLink =>
56         {
57             resultLink = foundLink[_constants.IndexPart];
58             return _constants.Break;
59         }, _constants.Any, ulong.MaxValue, ulong.MaxValue);
60         Assert.True(resultLink == link);
61         Assert.True(memoryAdapter.Count(ulong.MaxValue) == 0);
62         memoryAdapter.Delete(link);
63     }
64 }
65 }

```

./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  using Platform.Reflection;
7
8  namespace Platform.Data.Doublets.Tests
9  {
10     public static class ScopeTests
11     {
12         [Fact]
13         public static void SingleDependencyTest()
14         {
15             using (var scope = new Scope())
16             {
17                 scope.IncludeAssemblyOf<IMemory>();
18                 var instance = scope.Use<IDirectMemory>();
19                 Assert.IsType<HeapResizableDirectMemory>(instance);
20             }
21         }
22
23         [Fact]
24         public static void CascadeDependencyTest()
25         {
26             using (var scope = new Scope())
27             {
28                 scope.Include<TemporaryFileMappedResizableDirectMemory>();
29                 scope.Include<UInt64ResizableDirectMemoryLinks>();
30                 var instance = scope.Use<ILinks<ulong>>();
31                 Assert.IsType<UInt64ResizableDirectMemoryLinks>(instance);
32             }
33         }
34
35         [Fact]
36         public static void FullAutoResolutionTest()
37         {
38             using (var scope = new Scope(autoInclude: true, autoExplore: true))
39             {
40                 var instance = scope.Use<UInt64Links>();
41                 Assert.IsType<UInt64Links>(instance);
42             }
43         }
44
45         [Fact]
46         public static void TypeParametersTest()
47         {
48             using (var scope = new Scope<Types<HeapResizableDirectMemory,
49 ↪ ResizableDirectMemoryLinks<ulong>>>())
50             {
51                 var links = scope.Use<ILinks<ulong>>();
52                 Assert.IsType<ResizableDirectMemoryLinks<ulong>>(links);
53             }
54         }
55     }
56 }

```

./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.Doublets.Sequences;
11 using Platform.Data.Doublets.Sequences.Frequencies.Cache;
12 using Platform.Data.Doublets.Sequences.Frequencies.Counters;
13 using Platform.Data.Doublets.Sequences.Converters;
14 using Platform.Data.Doublets.Unicode;
15
16 namespace Platform.Data.Doublets.Tests
17 {
18     public static class SequencesTests
19     {
20         private static readonly LinksConstants<ulong> _constants =
21             ↪ Default<LinksConstants<ulong>>.Instance;
22
23         static SequencesTests()
24         {
25             // Trigger static constructor to not mess with performance measurements
26             _ = BitString.GetBitMaskFromIndex(1);
27         }
28
29         [Fact]
30         public static void CreateAllVariantsTest()
31         {
32             const long sequenceLength = 8;
33
34             using (var scope = new TempLinksTestScope(useSequences: true))
35             {
36                 var links = scope.Links;
37                 var sequences = scope.Sequences;
38
39                 var sequence = new ulong[sequenceLength];
40                 for (var i = 0; i < sequenceLength; i++)
41                 {
42                     sequence[i] = links.Create();
43                 }
44
45                 var sw1 = Stopwatch.StartNew();
46                 var results1 = sequences.CreateAllVariants1(sequence); sw1.Stop();
47
48                 var sw2 = Stopwatch.StartNew();
49                 var results2 = sequences.CreateAllVariants2(sequence); sw2.Stop();
50
51                 Assert.True(results1.Count > results2.Length);
52                 Assert.True(sw1.Elapsed > sw2.Elapsed);
53
54                 for (var i = 0; i < sequenceLength; i++)
55                 {
56                     links.Delete(sequence[i]);
57                 }
58
59                 Assert.True(links.Count() == 0);
60             }
61
62             //[Fact]
63             //public void CUDTest()
64             //{
65             //    var tempFilename = Path.GetTempFileName();
66
67             //    const long sequenceLength = 8;
68
69             //    const ulong itself = LinksConstants.Itself;
70
71             //    using (var memoryAdapter = new ResizableDirectMemoryLinks(tempFilename,
72             //        ↪ DefaultLinksSizeStep))
73             //    using (var links = new Links(memoryAdapter))
74             //    {
75             //        var sequence = new ulong[sequenceLength];
76             //        for (var i = 0; i < sequenceLength; i++)
77             //            sequence[i] = links.Create(itself, itself);
78
79             //        SequencesOptions o = new SequencesOptions();
80
81             //        TODO: Из числа в bool значения o.UseSequenceMarker = ((value & 1) != 0)
82             //        o.
83
84             //        var sequences = new Sequences(links);
85
86             //        var sw1 = Stopwatch.StartNew();

```

```

86 //         var results1 = sequences.CreateAllVariants1(sequence); sw1.Stop();
87
88 //         var sw2 = Stopwatch.StartNew();
89 //         var results2 = sequences.CreateAllVariants2(sequence); sw2.Stop();
90
91 //         Assert.True(results1.Count > results2.Length);
92 //         Assert.True(sw1.Elapsed > sw2.Elapsed);
93
94 //         for (var i = 0; i < sequenceLength; i++)
95 //             links.Delete(sequence[i]);
96 //     }
97
98 //     File.Delete(tempFilename);
99 // }
100
101 [Fact]
102 public static void AllVariantsSearchTest()
103 {
104     const long sequenceLength = 8;
105
106     using (var scope = new TempLinksTestScope(useSequences: true))
107     {
108         var links = scope.Links;
109         var sequences = scope.Sequences;
110
111         var sequence = new ulong[sequenceLength];
112         for (var i = 0; i < sequenceLength; i++)
113         {
114             sequence[i] = links.Create();
115         }
116
117         var createResults = sequences.CreateAllVariants2(sequence).Distinct().ToArray();
118
119         //for (int i = 0; i < createResults.Length; i++)
120         //    sequences.Create(createResults[i]);
121
122         var sw0 = Stopwatch.StartNew();
123         var searchResults0 = sequences.GetAllMatchingSequences0(sequence); sw0.Stop();
124
125         var sw1 = Stopwatch.StartNew();
126         var searchResults1 = sequences.GetAllMatchingSequences1(sequence); sw1.Stop();
127
128         var sw2 = Stopwatch.StartNew();
129         var searchResults2 = sequences.Each1(sequence); sw2.Stop();
130
131         var sw3 = Stopwatch.StartNew();
132         var searchResults3 = sequences.Each(sequence.ConvertToRestrictionsValues());
133         ↪ sw3.Stop();
134
135         var intersection0 = createResults.Intersect(searchResults0).ToList();
136         Assert.True(intersection0.Count == searchResults0.Count);
137         Assert.True(intersection0.Count == createResults.Length);
138
139         var intersection1 = createResults.Intersect(searchResults1).ToList();
140         Assert.True(intersection1.Count == searchResults1.Count);
141         Assert.True(intersection1.Count == createResults.Length);
142
143         var intersection2 = createResults.Intersect(searchResults2).ToList();
144         Assert.True(intersection2.Count == searchResults2.Count);
145         Assert.True(intersection2.Count == createResults.Length);
146
147         var intersection3 = createResults.Intersect(searchResults3).ToList();
148         Assert.True(intersection3.Count == searchResults3.Count);
149         Assert.True(intersection3.Count == createResults.Length);
150
151         for (var i = 0; i < sequenceLength; i++)
152         {
153             links.Delete(sequence[i]);
154         }
155     }
156
157 [Fact]
158 public static void BalancedVariantSearchTest()
159 {
160     const long sequenceLength = 200;
161
162     using (var scope = new TempLinksTestScope(useSequences: true))
163     {
164         var links = scope.Links;

```

```

165     var sequences = scope.Sequences;
166
167     var sequence = new ulong[sequenceLength];
168     for (var i = 0; i < sequenceLength; i++)
169     {
170         sequence[i] = links.Create();
171     }
172
173     var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
174
175     var sw1 = Stopwatch.StartNew();
176     var balancedVariant = balancedVariantConverter.Convert(sequence); sw1.Stop();
177
178     var sw2 = Stopwatch.StartNew();
179     var searchResults2 = sequences.GetAllMatchingSequences0(sequence); sw2.Stop();
180
181     var sw3 = Stopwatch.StartNew();
182     var searchResults3 = sequences.GetAllMatchingSequences1(sequence); sw3.Stop();
183
184     // На количестве в 200 элементов это будет занимать вечность
185     //var sw4 = Stopwatch.StartNew();
186     //var searchResults4 = sequences.Each(sequence); sw4.Stop();
187
188     Assert.True(searchResults2.Count == 1 && balancedVariant == searchResults2[0]);
189
190     Assert.True(searchResults3.Count == 1 && balancedVariant ==
191         ↪ searchResults3.First());
192
193     //Assert.True(sw1.Elapsed < sw2.Elapsed);
194
195     for (var i = 0; i < sequenceLength; i++)
196     {
197         links.Delete(sequence[i]);
198     }
199 }
200
201 [Fact]
202 public static void AllPartialVariantsSearchTest()
203 {
204     const long sequenceLength = 8;
205
206     using (var scope = new TempLinksTestScope(useSequences: true))
207     {
208         var links = scope.Links;
209         var sequences = scope.Sequences;
210
211         var sequence = new ulong[sequenceLength];
212         for (var i = 0; i < sequenceLength; i++)
213         {
214             sequence[i] = links.Create();
215         }
216
217         var createResults = sequences.CreateAllVariants2(sequence);
218
219         //var createResultsStrings = createResults.Select(x => x + ": " +
220             ↪ sequences.FormatSequence(x)).ToList();
221         //Global.Trash = createResultsStrings;
222
223         var partialSequence = new ulong[sequenceLength - 2];
224
225         Array.Copy(sequence, 1, partialSequence, 0, (int)sequenceLength - 2);
226
227         var sw1 = Stopwatch.StartNew();
228         var searchResults1 =
229             ↪ sequences.GetAllPartiallyMatchingSequences0(partialSequence); sw1.Stop();
230
231         var sw2 = Stopwatch.StartNew();
232         var searchResults2 =
233             ↪ sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();
234
235         //var sw3 = Stopwatch.StartNew();
236         //var searchResults3 =
237             ↪ sequences.GetAllPartiallyMatchingSequences2(partialSequence); sw3.Stop();
238
239         var sw4 = Stopwatch.StartNew();
240         var searchResults4 =
241             ↪ sequences.GetAllPartiallyMatchingSequences3(partialSequence); sw4.Stop();
242
243     }
244 }

```

```

238 //Global.Trash = searchResults3;
239
240 //var searchResults1Strings = searchResults1.Select(x => x + ": " +
    ↳ sequences.FormatSequence(x)).ToList();
241 //Global.Trash = searchResults1Strings;
242
243 var intersection1 = createResults.Intersect(searchResults1).ToList();
244 Assert.True(intersection1.Count == createResults.Length);
245
246 var intersection2 = createResults.Intersect(searchResults2).ToList();
247 Assert.True(intersection2.Count == createResults.Length);
248
249 var intersection4 = createResults.Intersect(searchResults4).ToList();
250 Assert.True(intersection4.Count == createResults.Length);
251
252 for (var i = 0; i < sequenceLength; i++)
253 {
254     links.Delete(sequence[i]);
255 }
256 }
257 }
258
259 [Fact]
260 public static void BalancedPartialVariantsSearchTest()
261 {
262     const long sequenceLength = 200;
263
264     using (var scope = new TempLinksTestScope(useSequences: true))
265     {
266         var links = scope.Links;
267         var sequences = scope.Sequences;
268
269         var sequence = new ulong[sequenceLength];
270         for (var i = 0; i < sequenceLength; i++)
271         {
272             sequence[i] = links.Create();
273         }
274
275         var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
276
277         var balancedVariant = balancedVariantConverter.Convert(sequence);
278
279         var partialSequence = new ulong[sequenceLength - 2];
280
281         Array.Copy(sequence, 1, partialSequence, 0, (int)sequenceLength - 2);
282
283         var sw1 = Stopwatch.StartNew();
284         var searchResults1 =
285             ↳ sequences.GetAllPartiallyMatchingSequences0(partialSequence); sw1.Stop();
286
287         var sw2 = Stopwatch.StartNew();
288         var searchResults2 =
289             ↳ sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();
290
291         Assert.True(searchResults1.Count == 1 && balancedVariant == searchResults1[0]);
292
293         Assert.True(searchResults2.Count == 1 && balancedVariant ==
294             ↳ searchResults2.First());
295
296         for (var i = 0; i < sequenceLength; i++)
297         {
298             links.Delete(sequence[i]);
299         }
300     }
301 }
302
303 [Fact(Skip = "Correct implementation is pending")]
304 public static void PatternMatchTest()
305 {
306     var zeroOrMany = Sequences.Sequences.ZeroOrMany;
307
308     using (var scope = new TempLinksTestScope(useSequences: true))
309     {
310         var links = scope.Links;
311         var sequences = scope.Sequences;
312
313         var e1 = links.Create();
314         var e2 = links.Create();
315
316         var sequence = new[]

```



```

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

```

Обозначение пустоты, какое оно? Темнота ли это? Там где отсутствие света, отсутствие фотонов
 ↪ (носителей света)? Или это то, что полностью отражает свет? Пустой белый лист бумаги? Там
 ↪ где есть место для нового начала? Разве пустота это не характеристика пространства?
 ↪ Пространство это то, что можно чем-то наполнить?

384 [![чёрное пространство, белое
↳ пространство](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/1.png
↳ "чёрное пространство, белое пространство")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/1.png)

385

386 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?

387

388 [![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)

389

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

391

392 [![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)

393

394 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?
↳ Грань? Разделителем? Единицей?

395

396 [![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)

397

398 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?

399

400 [![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)

401

402 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?

403

404 [![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)

405

406 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?

407

408 [![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)

409

410 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?

411

412 [![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)

413

414 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?

415

```

416  [![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)
417
418  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?
419
420  [![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)
421
422  На один уровень внутрь (вниз)? Или на один уровень во вне (вверх)? Или это можно назвать шагом
    ↳  рекурсии или фрактала?
423
424  [![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)
425
426  Последовательность? Массив? Список? Множество? Объект? Таблица? Элементы? Цвета? Символы? Буквы?
    ↳  Слово? Цифры? Число? Алфавит? Дерево? Сеть? Граф? Гиперграф?
427
428  [![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)
429
430  ...
431
432  [![анимация](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro-animat
    ↳  ion-500.gif
    ↳  "анимация")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/intro
    ↳  -animation-500.gif)";
433
434      private static readonly string _exampleLoremIpsumText =
435          @"Lorem ipsum dolor sit amet, consectetur adipiscing elit, sed do eiusmod tempor
    ↳  incididunt ut labore et dolore magna aliqua.
436  Ut enim ad minim veniam, quis nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo
    ↳  consequat.";
437
438      [Fact]
439      public static void CompressionTest()
440      {
441          using (var scope = new TempLinksTestScope(useSequences: true))
442          {
443              var links = scope.Links;
444              var sequences = scope.Sequences;
445
446              var e1 = links.Create();
447              var e2 = links.Create();
448
449              var sequence = new[]
450              {
451                  e1, e2, e1, e2 // mama / papa / template [(m/p), a] { [1] [2] [1] [2] }
452              };
453
454              var balancedVariantConverter = new BalancedVariantConverter<ulong>(links.Unsync);
455              var totalSequenceSymbolFrequencyCounter = new
    ↳  TotalSequenceSymbolFrequencyCounter<ulong>(links.Unsync);
456              var doubletFrequenciesCache = new LinkFrequenciesCache<ulong>(links.Unsync,
    ↳  totalSequenceSymbolFrequencyCounter);
457              var compressingConverter = new CompressingConverter<ulong>(links.Unsync,
    ↳  balancedVariantConverter, doubletFrequenciesCache);
458
459              var compressedVariant = compressingConverter.Convert(sequence);
460
461              // 1: [1]          (1->1) point
462              // 2: [2]          (2->2) point
463              // 3: [1,2]        (1->2) doublet
464              // 4: [1,2,1,2]    (3->3) doublet

```

```

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

```

```

525     var linkFrequenciesCache3 = new LinkFrequenciesCache<ulong>(scope3.Links.Unsync,
526         ↪ totalSequenceSymbolFrequencyCounter);
527
528     var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<ulong>(linkFrequenciesCache3);
529
530     var sequenceToItsLocalElementLevelsConverter = new
531         ↪ SequenceToItsLocalElementLevelsConverter<ulong>(scope3.Links.Unsync,
532         ↪ linkToItsFrequencyNumberConverter);
533     var optimalVariantConverter = new
534         ↪ OptimalVariantConverter<ulong>(scope3.Links.Unsync,
535         ↪ sequenceToItsLocalElementLevelsConverter);
536
537     var compressed1 = new ulong[arrays.Length];
538     var compressed2 = new ulong[arrays.Length];
539     var compressed3 = new ulong[arrays.Length];
540
541     var START = 0;
542     var END = arrays.Length;
543
544     //for (int i = START; i < END; i++)
545     //    linkFrequenciesCache1.IncrementFrequencies(arrays[i]);
546
547     var initialCount1 = scope2.Links.Unsync.Count();
548
549     var sw1 = Stopwatch.StartNew();
550
551     for (int i = START; i < END; i++)
552     {
553         linkFrequenciesCache1.IncrementFrequencies(arrays[i]);
554         compressed1[i] = compressor1.Convert(arrays[i]);
555     }
556
557     var elapsed1 = sw1.Elapsed;
558
559     var balancedVariantConverter2 = new
560         ↪ BalancedVariantConverter<ulong>(scope2.Links.Unsync);
561
562     var initialCount2 = scope2.Links.Unsync.Count();
563
564     var sw2 = Stopwatch.StartNew();
565
566     for (int i = START; i < END; i++)
567     {
568         compressed2[i] = balancedVariantConverter2.Convert(arrays[i]);
569     }
570
571     var elapsed2 = sw2.Elapsed;
572
573     for (int i = START; i < END; i++)
574     {
575         linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
576     }
577
578     var initialCount3 = scope3.Links.Unsync.Count();
579
580     var sw3 = Stopwatch.StartNew();
581
582     for (int i = START; i < END; i++)
583     {
584         //linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
585         compressed3[i] = optimalVariantConverter.Convert(arrays[i]);
586     }
587
588     var elapsed3 = sw3.Elapsed;
589
590     Console.WriteLine($"Compressor: {elapsed1}, Balanced variant: {elapsed2},
591         ↪ Optimal variant: {elapsed3}");
592
593     // Assert.True(elapsed1 > elapsed2);
594
595     // Checks
596     for (int i = START; i < END; i++)
597     {
598         var sequence1 = compressed1[i];
599         var sequence2 = compressed2[i];
600         var sequence3 = compressed3[i];
601
602         var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
603             ↪ scope1.Links.Unsync);

```

```

596         var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
597             ↪ scope2.Links.Unsync);
598
599         var decompress3 = UnicodeMap.FromSequenceLinkToString(sequence3,
600             ↪ scope3.Links.Unsync);
601
602         var structure1 = scope1.Links.Unsync.FormatStructure(sequence1, link =>
603             ↪ link.IsPartialPoint());
604         var structure2 = scope2.Links.Unsync.FormatStructure(sequence2, link =>
605             ↪ link.IsPartialPoint());
606         var structure3 = scope3.Links.Unsync.FormatStructure(sequence3, link =>
607             ↪ link.IsPartialPoint());
608
609         //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
610             ↪ arrays[i].Length > 3)
611             // Assert.False(structure1 == structure2);
612         //if (sequence3 != Constants.Null && sequence2 != Constants.Null &&
613             ↪ arrays[i].Length > 3)
614             // Assert.False(structure3 == structure2);
615
616         Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
617         Assert.True(strings[i] == decompress3 && decompress3 == decompress2);
618     }
619
620     Assert.True((int)(scope1.Links.Unsync.Count() - initialCount1) <
621         ↪ totalCharacters);
622     Assert.True((int)(scope2.Links.Unsync.Count() - initialCount2) <
623         ↪ totalCharacters);
624     Assert.True((int)(scope3.Links.Unsync.Count() - initialCount3) <
625         ↪ totalCharacters);
626
627     Console.WriteLine($"{(double)(scope1.Links.Unsync.Count() - initialCount1) /
628         ↪ totalCharacters} | {(double)(scope2.Links.Unsync.Count() - initialCount2) /
629         ↪ totalCharacters} | {(double)(scope3.Links.Unsync.Count() - initialCount3) /
630         ↪ totalCharacters}");
631
632     Assert.True(scope1.Links.Unsync.Count() - initialCount1 <
633         ↪ scope2.Links.Unsync.Count() - initialCount2);
634     Assert.True(scope3.Links.Unsync.Count() - initialCount3 <
635         ↪ scope2.Links.Unsync.Count() - initialCount2);
636
637     var duplicateProvider1 = new
638         ↪ DuplicateSegmentsProvider<ulong>(scope1.Links.Unsync, scope1.Sequences);
639     var duplicateProvider2 = new
640         ↪ DuplicateSegmentsProvider<ulong>(scope2.Links.Unsync, scope2.Sequences);
641     var duplicateProvider3 = new
642         ↪ DuplicateSegmentsProvider<ulong>(scope3.Links.Unsync, scope3.Sequences);
643
644     var duplicateCounter1 = new DuplicateSegmentsCounter<ulong>(duplicateProvider1);
645     var duplicateCounter2 = new DuplicateSegmentsCounter<ulong>(duplicateProvider2);
646     var duplicateCounter3 = new DuplicateSegmentsCounter<ulong>(duplicateProvider3);
647
648     var duplicates1 = duplicateCounter1.Count();
649
650     ConsoleHelpers.Debug("-----");
651
652     var duplicates2 = duplicateCounter2.Count();
653
654     ConsoleHelpers.Debug("-----");
655
656     var duplicates3 = duplicateCounter3.Count();
657
658     Console.WriteLine($"{duplicates1} | {duplicates2} | {duplicates3}");
659
660     linkFrequenciesCache1.ValidateFrequencies();
661     linkFrequenciesCache3.ValidateFrequencies();
662 }
663
664 [Fact]
665 public static void CompressionStabilityTest()
666 {
667     // TODO: Fix bug (do a separate test)
668     //const ulong minNumbers = 0;
669     //const ulong maxNumbers = 1000;
670
671     const ulong minNumbers = 10000;

```

```

656     const ulong maxNumbers = 12500;
657
658     var strings = new List<string>();
659
660     for (ulong i = minNumbers; i < maxNumbers; i++)
661     {
662         strings.Add(i.ToString());
663     }
664
665     var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
666     var totalCharacters = arrays.Select(x => x.Length).Sum();
667
668     using (var scope1 = new TempLinksTestScope(useSequences: true, sequencesOptions: new
        ↳ SequencesOptions<ulong> { UseCompression = true,
        ↳ EnforceSingleSequenceVersionOnWriteBasedOnExisting = true }))
        using (var scope2 = new TempLinksTestScope(useSequences: true))
        {
            scope1.Links.UseUnicode();
            scope2.Links.UseUnicode();
        }
        //var compressor1 = new Compressor(scope1.Links.Unsync, scope1.Sequences);
        var compressor1 = scope1.Sequences;
        var compressor2 = scope2.Sequences;
        var compressed1 = new ulong[arrays.Length];
        var compressed2 = new ulong[arrays.Length];
        var sw1 = Stopwatch.StartNew();
        var START = 0;
        var END = arrays.Length;
        // Collisions proved (cannot be solved by max doublet comparison, no stable rule)
        // Stability issue starts at 10001 or 11000
        //for (int i = START; i < END; i++)
        //{
        //    var first = compressor1.Compress(arrays[i]);
        //    var second = compressor1.Compress(arrays[i]);
        //    if (first == second)
        //        compressed1[i] = first;
        //    else
        //    {
        //        // TODO: Find a solution for this case
        //    }
        //}
        for (int i = START; i < END; i++)
        {
            var first = compressor1.Create(arrays[i].ConvertToRestrictionsValues());
            var second = compressor1.Create(arrays[i].ConvertToRestrictionsValues());
            if (first == second)
            {
                compressed1[i] = first;
            }
            else
            {
                // TODO: Find a solution for this case
            }
        }
        var elapsed1 = sw1.Elapsed;
        var balancedVariantConverter = new BalancedVariantConverter<ulong>(scope2.Links);
        var sw2 = Stopwatch.StartNew();
        for (int i = START; i < END; i++)
        {
            var first = balancedVariantConverter.Convert(arrays[i]);
            var second = balancedVariantConverter.Convert(arrays[i]);
            if (first == second)
            {
                compressed2[i] = first;
            }
        }
        var elapsed2 = sw2.Elapsed;

```

```

734 Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator:
735 ↪ {elapsed2}");
736
737 Assert.True(elapsed1 > elapsed2);
738
739 // Checks
740 for (int i = START; i < END; i++)
741 {
742     var sequence1 = compressed1[i];
743     var sequence2 = compressed2[i];
744
745     if (sequence1 != _constants.Null && sequence2 != _constants.Null)
746     {
747         var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
748 ↪ scope1.Links);
749
750         var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
751 ↪ scope2.Links);
752
753         //var structure1 = scope1.Links.FormatStructure(sequence1, link =>
754 ↪ link.IsPartialPoint());
755         //var structure2 = scope2.Links.FormatStructure(sequence2, link =>
756 ↪ link.IsPartialPoint());
757
758         //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
759 ↪ arrays[i].Length > 3)
760         //    Assert.False(structure1 == structure2);
761
762         Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
763     }
764 }
765
766 Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
767 Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
768
769 Debug.WriteLine($"{{(double)(scope1.Links.Count() - UnicodeMap.MapSize) /
770 ↪ totalCharacters}} | {{(double)(scope2.Links.Count() - UnicodeMap.MapSize) /
771 ↪ totalCharacters}}");
772
773 Assert.True(scope1.Links.Count() <= scope2.Links.Count());
774
775 //compressor1.ValidateFrequencies();
776 }
777
778 [Fact]
779 public static void RandomNumbersCompressionQualityTest()
780 {
781     const ulong N = 500;
782
783     //const ulong minNumbers = 10000;
784     //const ulong maxNumbers = 20000;
785
786     //var strings = new List<string>();
787
788     //for (ulong i = 0; i < N; i++)
789     //    strings.Add(RandomHelpers.DefaultFactory.NextUInt64(minNumbers,
790 ↪ maxNumbers).ToString());
791
792     var strings = new List<string>();
793
794     for (ulong i = 0; i < N; i++)
795     {
796         strings.Add(RandomHelpers.Default.NextUInt64().ToString());
797     }
798
799     strings = strings.Distinct().ToList();
800
801     var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
802     var totalCharacters = arrays.Select(x => x.Length).Sum();
803
804     using (var scope1 = new TempLinksTestScope(useSequences: true, sequencesOptions: new
805 ↪ SequencesOptions<ulong> { UseCompression = true,
806 ↪ EnforceSingleSequenceVersionOnWriteBasedOnExisting = true }))
807     using (var scope2 = new TempLinksTestScope(useSequences: true))
808     {
809         scope1.Links.UseUnicode();

```



```

801     scope2.Links.UseUnicode();
802
803     var compressor1 = scope1.Sequences;
804     var compressor2 = scope2.Sequences;
805
806     var compressed1 = new ulong[arrays.Length];
807     var compressed2 = new ulong[arrays.Length];
808
809     var sw1 = Stopwatch.StartNew();
810
811     var START = 0;
812     var END = arrays.Length;
813
814     for (int i = START; i < END; i++)
815     {
816         compressed1[i] = compressor1.Create(arrays[i].ConvertToRestrictionsValues());
817     }
818
819     var elapsed1 = sw1.Elapsed;
820
821     var balancedVariantConverter = new BalancedVariantConverter<ulong>(scope2.Links);
822
823     var sw2 = Stopwatch.StartNew();
824
825     for (int i = START; i < END; i++)
826     {
827         compressed2[i] = balancedVariantConverter.Convert(arrays[i]);
828     }
829
830     var elapsed2 = sw2.Elapsed;
831
832     Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator:
833         ↳ {elapsed2}");
834
835     Assert.True(elapsed1 > elapsed2);
836
837     // Checks
838     for (int i = START; i < END; i++)
839     {
840         var sequence1 = compressed1[i];
841         var sequence2 = compressed2[i];
842
843         if (sequence1 != _constants.Null && sequence2 != _constants.Null)
844         {
845             var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
846                 ↳ scope1.Links);
847
848             var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
849                 ↳ scope2.Links);
850
851             Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
852         }
853     }
854
855     Assert.True((int)(scope1.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
856     Assert.True((int)(scope2.Links.Count() - UnicodeMap.MapSize) < totalCharacters);
857
858     Debug.WriteLine($"{{(double)(scope1.Links.Count() - UnicodeMap.MapSize) /
859         ↳ totalCharacters}} | {{(double)(scope2.Links.Count() - UnicodeMap.MapSize) /
860         ↳ totalCharacters}}");
861
862     // Can be worse than balanced variant
863     //Assert.True(scope1.Links.Count() <= scope2.Links.Count());
864
865     //compressor1.ValidateFrequencies();
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     }
881 }

```

```

876     var sequences = scope.Sequences;
877
878     var sequence = new ulong[sequenceLength];
879     for (var i = 0; i < sequenceLength; i++)
880     {
881         sequence[i] = links.Create();
882     }
883
884     var createResults = sequences.CreateAllVariants2(sequence);
885
886     Global.Trash = createResults;
887
888     for (var i = 0; i < sequenceLength; i++)
889     {
890         links.Delete(sequence[i]);
891     }
892 }
893
894 [Fact]
895 public static void AllPossibleConnectionsTest()
896 {
897     const long sequenceLength = 5;
898
899     using (var scope = new TempLinksTestScope(useSequences: true))
900     {
901         var links = scope.Links;
902         var sequences = scope.Sequences;
903
904         var sequence = new ulong[sequenceLength];
905         for (var i = 0; i < sequenceLength; i++)
906         {
907             sequence[i] = links.Create();
908         }
909
910         var createResults = sequences.CreateAllVariants2(sequence);
911         var reverseResults = sequences.CreateAllVariants2(sequence.Reverse().ToArray());
912
913         for (var i = 0; i < 1; i++)
914         {
915             var sw1 = Stopwatch.StartNew();
916             var searchResults1 = sequences.GetAllConnections(sequence); sw1.Stop();
917
918             var sw2 = Stopwatch.StartNew();
919             var searchResults2 = sequences.GetAllConnections1(sequence); sw2.Stop();
920
921             var sw3 = Stopwatch.StartNew();
922             var searchResults3 = sequences.GetAllConnections2(sequence); sw3.Stop();
923
924             var sw4 = Stopwatch.StartNew();
925             var searchResults4 = sequences.GetAllConnections3(sequence); sw4.Stop();
926
927             Global.Trash = searchResults3;
928             Global.Trash = searchResults4; //-V3008
929
930             var intersection1 = createResults.Intersect(searchResults1).ToList();
931             Assert.True(intersection1.Count == createResults.Length);
932
933             var intersection2 = reverseResults.Intersect(searchResults1).ToList();
934             Assert.True(intersection2.Count == reverseResults.Length);
935
936             var intersection0 = searchResults1.Intersect(searchResults2).ToList();
937             Assert.True(intersection0.Count == searchResults2.Count);
938
939             var intersection3 = searchResults2.Intersect(searchResults3).ToList();
940             Assert.True(intersection3.Count == searchResults3.Count);
941
942             var intersection4 = searchResults3.Intersect(searchResults4).ToList();
943             Assert.True(intersection4.Count == searchResults4.Count);
944         }
945
946         for (var i = 0; i < sequenceLength; i++)
947         {
948             links.Delete(sequence[i]);
949         }
950     }
951 }
952
953 [Fact(Skip = "Correct implementation is pending")]
954 public static void CalculateAllUsagesTest()
955

```

```

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

```

./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 ILinks<ulong> MemoryAdapter { get; }
12         public SynchronizedLinks<ulong> Links { get; }
13         public Sequences.Sequences Sequences { get; }
14         public string TempFilename { get; }
15         public string TempTransactionLogFilename { get; }
16         private readonly bool _deleteFiles;
17
18         public TempLinksTestScope(bool deleteFiles = true, bool useSequences = false, bool
19         ↪ useLog = false) : this(new SequencesOptions<ulong>(), deleteFiles, useSequences,
20         ↪ useLog) { }
21
22         public TempLinksTestScope(SequencesOptions<ulong> sequencesOptions, bool deleteFiles =
23         ↪ true, bool useSequences = false, bool useLog = false)
24         {
25             _deleteFiles = deleteFiles;
26             TempFilename = Path.GetTempFileName();
27             TempTransactionLogFilename = Path.GetTempFileName();
28             var coreMemoryAdapter = new UInt64ResizableDirectMemoryLinks(TempFilename);
29             MemoryAdapter = useLog ? (ILinks<ulong>)new
30             ↪ UInt64LinksTransactionsLayer(coreMemoryAdapter, TempTransactionLogFilename) :
31             ↪ coreMemoryAdapter;
32             Links = new SynchronizedLinks<ulong>(new UInt64Links(MemoryAdapter));
33             if (useSequences)
34             {
35                 Sequences = new Sequences.Sequences(Links, sequencesOptions);
36             }
37         }
38     }
39 }

```

```

32     }
33
34     protected override void Dispose(bool manual, bool wasDisposed)
35     {
36         if (!wasDisposed)
37         {
38             Links.Unsync.DisposeIfPossible();
39             if (_deleteFiles)
40             {
41                 DeleteFiles();
42             }
43         }
44     }
45
46     public void DeleteFiles()
47     {
48         File.Delete(TempFilename);
49         File.Delete(TempTransactionLogFilename);
50     }
51 }
52 }

```

./Platform.Data.Doublets.Tests/TestExtensions.cs

```

1  using System.Collections.Generic;
2  using Xunit;
3  using Platform.Ranges;
4  using Platform.Numbers;
5  using Platform.Random;
6  using Platform.Setters;
7
8  namespace Platform.Data.Doublets.Tests
9  {
10     public static class TestExtensions
11     {
12         public static void TestCRUDOperations<T>(this ILinks<T> links)
13         {
14             var constants = links.Constants;
15
16             var equalityComparer = EqualityComparer<T>.Default;
17
18             // Create Link
19             Assert.True(equalityComparer.Equals(links.Count(), Integer<T>.Zero));
20
21             var setter = new Setter<T>(constants.Null);
22             links.Each(constants.Any, constants.Any, setter.SetAndReturnTrue);
23
24             Assert.True(equalityComparer.Equals(setter.Result, constants.Null));
25
26             var linkAddress = links.Create();
27
28             var link = new Link<T>(links.GetLink(linkAddress));
29
30             Assert.True(link.Count == 3);
31             Assert.True(equalityComparer.Equals(link.Index, linkAddress));
32             Assert.True(equalityComparer.Equals(link.Source, constants.Null));
33             Assert.True(equalityComparer.Equals(link.Target, constants.Null));
34
35             Assert.True(equalityComparer.Equals(links.Count(), Integer<T>.One));
36
37             // Get first link
38             setter = new Setter<T>(constants.Null);
39             links.Each(constants.Any, constants.Any, setter.SetAndReturnFalse);
40
41             Assert.True(equalityComparer.Equals(setter.Result, linkAddress));
42
43             // Update link to reference itself
44             links.Update(linkAddress, linkAddress, linkAddress);
45
46             link = new Link<T>(links.GetLink(linkAddress));
47
48             Assert.True(equalityComparer.Equals(link.Source, linkAddress));
49             Assert.True(equalityComparer.Equals(link.Target, linkAddress));
50
51             // Update link to reference null (prepare for delete)
52             var updated = links.Update(linkAddress, constants.Null, constants.Null);
53
54             Assert.True(equalityComparer.Equals(updated, linkAddress));
55
56             link = new Link<T>(links.GetLink(linkAddress));
57

```

```

58     Assert.True(equalityComparer.Equals(link.Source, constants.Null));
59     Assert.True(equalityComparer.Equals(link.Target, constants.Null));
60
61     // Delete link
62     links.Delete(linkAddress);
63
64     Assert.True(equalityComparer.Equals(links.Count(), Integer<T>.Zero));
65
66     setter = new Setter<T>(constants.Null);
67     links.Each(constants.Any, constants.Any, setter.SetAndReturnTrue);
68
69     Assert.True(equalityComparer.Equals(setter.Result, constants.Null));
70 }
71
72 public static void TestRawNumbersCRUDOperations<T>(this ILinks<T> links)
73 {
74     // Constants
75     var constants = links.Constants;
76     var equalityComparer = EqualityComparer<T>.Default;
77
78     var h106E = new Hybrid<T>(106L, isExternal: true);
79     var h107E = new Hybrid<T>(-char.ConvertFromUtf32(107)[0]);
80     var h108E = new Hybrid<T>(-108L);
81
82     Assert.Equal(106L, h106E.AbsoluteValue);
83     Assert.Equal(107L, h107E.AbsoluteValue);
84     Assert.Equal(108L, h108E.AbsoluteValue);
85
86     // Create Link (External -> External)
87     var linkAddress1 = links.Create();
88
89     links.Update(linkAddress1, h106E, h108E);
90
91     var link1 = new Link<T>(links.GetLink(linkAddress1));
92
93     Assert.True(equalityComparer.Equals(link1.Source, h106E));
94     Assert.True(equalityComparer.Equals(link1.Target, h108E));
95
96     // Create Link (Internal -> External)
97     var linkAddress2 = links.Create();
98
99     links.Update(linkAddress2, linkAddress1, h108E);
100
101     var link2 = new Link<T>(links.GetLink(linkAddress2));
102
103     Assert.True(equalityComparer.Equals(link2.Source, linkAddress1));
104     Assert.True(equalityComparer.Equals(link2.Target, h108E));
105
106     // Create Link (Internal -> Internal)
107     var linkAddress3 = links.Create();
108
109     links.Update(linkAddress3, linkAddress1, linkAddress2);
110
111     var link3 = new Link<T>(links.GetLink(linkAddress3));
112
113     Assert.True(equalityComparer.Equals(link3.Source, linkAddress1));
114     Assert.True(equalityComparer.Equals(link3.Target, linkAddress2));
115
116     // Search for created link
117     var setter1 = new Setter<T>(constants.Null);
118     links.Each(h106E, h108E, setter1.SetAndReturnFalse);
119
120     Assert.True(equalityComparer.Equals(setter1.Result, linkAddress1));
121
122     // Search for nonexistent link
123     var setter2 = new Setter<T>(constants.Null);
124     links.Each(h106E, h107E, setter2.SetAndReturnFalse);
125
126     Assert.True(equalityComparer.Equals(setter2.Result, constants.Null));
127
128     // Update link to reference null (prepare for delete)
129     var updated = links.Update(linkAddress3, constants.Null, constants.Null);
130
131     Assert.True(equalityComparer.Equals(updated, linkAddress3));
132
133     link3 = new Link<T>(links.GetLink(linkAddress3));
134
135     Assert.True(equalityComparer.Equals(link3.Source, constants.Null));
136     Assert.True(equalityComparer.Equals(link3.Target, constants.Null));
137

```

```

138 // Delete link
139 links.Delete(linkAddress3);
140
141 Assert.True(equalityComparer.Equals(links.Count(), Integer<T>.Two));
142
143 var setter3 = new Setter<T>(constants.Null);
144 links.Each(constants.Any, constants.Any, setter3.SetAndReturnTrue);
145
146 Assert.True(equalityComparer.Equals(setter3.Result, linkAddress2));
147 }
148
149 public static void TestMultipleRandomCreationsAndDeletions<TLink>(this ILinks<TLink>
→ links, int maximumOperationsPerCycle)
150 {
151     var comparer = Comparer<TLink>.Default;
152     for (var N = 1; N < maximumOperationsPerCycle; N++)
153     {
154         var random = new System.Random(N);
155         var created = 0;
156         var deleted = 0;
157         for (var i = 0; i < N; i++)
158         {
159             long linksCount = (Integer<TLink>)links.Count();
160             var createPoint = random.NextBoolean();
161             if (linksCount > 2 && createPoint)
162             {
163                 var linksAddressRange = new Range<ulong>(1, (ulong)linksCount);
164                 TLink source = (Integer<TLink>)random.NextUInt64(linksAddressRange);
165                 TLink target = (Integer<TLink>)random.NextUInt64(linksAddressRange);
166                 → //-V3086
167                 var resultLink = links.CreateAndUpdate(source, target);
168                 if (comparer.Compare(resultLink, (Integer<TLink>)linksCount) > 0)
169                 {
170                     created++;
171                 }
172             }
173             else
174             {
175                 links.Create();
176                 created++;
177             }
178             Assert.True(created == (Integer<TLink>)links.Count());
179             for (var i = 0; i < N; i++)
180             {
181                 TLink link = (Integer<TLink>)(i + 1);
182                 if (links.Exists(link))
183                 {
184                     links.Delete(link);
185                     deleted++;
186                 }
187             }
188             Assert.True((Integer<TLink>)links.Count() == 0);
189         }
190     }
191 }
192 }

```

./Platform.Data.Doublets.Tests/UInt64LinksTests.cs

```

1 using System;
2 using System.Collections.Generic;
3 using System.Diagnostics;
4 using System.IO;
5 using System.Text;
6 using System.Threading;
7 using System.Threading.Tasks;
8 using Xunit;
9 using Platform.Disposables;
10 using Platform.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.Doublets.ResizableDirectMemory;
18 using Platform.Data.Doublets.Decorators;
19
20 namespace Platform.Data.Doublets.Tests
21 {

```

```

22 public static class UInt64LinksTests
23 {
24     private static readonly LinksConstants<ulong> _constants =
25         ↪ Default<LinksConstants<ulong>>.Instance;
26
27     private const long Iterations = 10 * 1024;
28
29     #region Concept
30
31     [Fact]
32     public static void MultipleCreateAndDeleteTest()
33     {
34         using (var scope = new TempLinksTestScope())
35         {
36             scope.Links.TestMultipleRandomCreationsAndDeletions(100);
37         }
38
39     [Fact]
40     public static void CascadeUpdateTest()
41     {
42         var itself = _constants.Itself;
43
44         using (var scope = new TempLinksTestScope(useLog: true))
45         {
46             var links = scope.Links;
47
48             var l1 = links.Create();
49             var l2 = links.Create();
50
51             l2 = links.Update(l2, l2, l1, l2);
52
53             links.CreateAndUpdate(l2, itself);
54             links.CreateAndUpdate(l2, itself);
55
56             l2 = links.Update(l2, l1);
57
58             links.Delete(l2);
59
60             Global.Trash = links.Count();
61
62             links.Unsync.DisposeIfPossible(); // Close links to access log
63
64             Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scope
65                 ↪ e.TempTransactionLogFilename);
66         }
67
68     [Fact]
69     public static void BasicTransactionLogTest()
70     {
71         using (var scope = new TempLinksTestScope(useLog: true))
72         {
73             var links = scope.Links;
74             var l1 = links.Create();
75             var l2 = links.Create();
76
77             Global.Trash = links.Update(l2, l2, l1, l2);
78
79             links.Delete(l1);
80
81             links.Unsync.DisposeIfPossible(); // Close links to access log
82
83             Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scope
84                 ↪ e.TempTransactionLogFilename);
85         }
86
87     [Fact]
88     public static void TransactionAutoRevertedTest()
89     {
90         // Auto Reverted (Because no commit at transaction)
91         using (var scope = new TempLinksTestScope(useLog: true))
92         {
93             var links = scope.Links;
94             var transactionsLayer = (UInt64LinksTransactionsLayer)scope.MemoryAdapter;
95             using (var transaction = transactionsLayer.BeginTransaction())
96             {
97                 var l1 = links.Create();
98                 var l2 = links.Create();

```

```

99         links.Update(l2, l2, l1, l2);
100     }
101
102     Assert.Equal(OUL, links.Count());
103
104     links.Unsync.DisposeIfPossible();
105
106     var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scope
107     ↪ cope.TempTransactionLogFilename);
108     Assert.Single(transitions);
109 }
110
111 [Fact]
112 public static void TransactionUserCodeErrorNoDataSavedTest()
113 {
114     // User Code Error (Autoreverted), no data saved
115     var itself = _constants.Itself;
116
117     TempLinksTestScope lastScope = null;
118     try
119     {
120         using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
121         ↪ useLog: true))
122         {
123             var links = scope.Links;
124             var transactionsLayer = (UInt64LinksTransactionsLayer)((LinksDisposableDecor
125             ↪ atorBase<ulong>)links.Unsync).Links;
126             using (var transaction = transactionsLayer.BeginTransaction())
127             {
128                 var l1 = links.CreateAndUpdate(itself, itself);
129                 var l2 = links.CreateAndUpdate(itself, itself);
130
131                 l2 = links.Update(l2, l2, l1, l2);
132
133                 links.CreateAndUpdate(l2, itself);
134                 links.CreateAndUpdate(l2, itself);
135
136                 //Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transi
137                 ↪ tion>(scope.TempTransactionLogFilename);
138
139                 l2 = links.Update(l2, l1);
140
141                 links.Delete(l2);
142
143                 ExceptionThrower();
144
145                 transaction.Commit();
146             }
147
148             Global.Trash = links.Count();
149         }
150     }
151     catch
152     {
153         Assert.False(lastScope == null);
154
155         var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(l
156         ↪ astScope.TempTransactionLogFilename);
157
158         Assert.True(transitions.Length == 1 && transitions[0].Before.IsNull() &&
159         ↪ transitions[0].After.IsNull());
160
161         lastScope.DeleteFiles();
162     }
163 }
164
165 [Fact]
166 public static void TransactionUserCodeErrorSomeDataSavedTest()
167 {
168     // User Code Error (Autoreverted), some data saved
169     var itself = _constants.Itself;
170
171     TempLinksTestScope lastScope = null;
172     try
173     {
174         ulong l1;
175         ulong l2;

```



```

173     using (var scope = new TempLinksTestScope(useLog: true))
174     {
175         var links = scope.Links;
176         l1 = links.CreateAndUpdate(itself, itself);
177         l2 = links.CreateAndUpdate(itself, itself);
178
179         l2 = links.Update(l2, l2, l1, l2);
180
181         links.CreateAndUpdate(l2, itself);
182         links.CreateAndUpdate(l2, itself);
183
184         links.Unsync.DisposeIfPossible();
185
186         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(
            ↪ scope.TempTransactionLogFilename);
187     }
188
189     using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
190     ↪ useLog: true))
191     {
192         var links = scope.Links;
193         var transactionsLayer = (UInt64LinksTransactionsLayer)links.Unsync;
194         using (var transaction = transactionsLayer.BeginTransaction())
195         {
196             l2 = links.Update(l2, l1);
197
198             links.Delete(l2);
199
200             ExceptionThrower();
201
202             transaction.Commit();
203         }
204
205         Global.Trash = links.Count();
206     }
207     catch
208     {
209         Assert.False(lastScope == null);
210
211         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(last
            ↪ Scope.TempTransactionLogFilename);
212
213         lastScope.DeleteFiles();
214     }
215 }
216
217 [Fact]
218 public static void TransactionCommit()
219 {
220     var itself = _constants.Itself;
221
222     var tempDatabaseFilename = Path.GetTempFileName();
223     var tempTransactionLogFilename = Path.GetTempFileName();
224
225     // Commit
226     using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
            ↪ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
            ↪ tempTransactionLogFilename))
227     using (var links = new UInt64Links(memoryAdapter))
228     {
229         using (var transaction = memoryAdapter.BeginTransaction())
230         {
231             var l1 = links.CreateAndUpdate(itself, itself);
232             var l2 = links.CreateAndUpdate(itself, itself);
233
234             Global.Trash = links.Update(l2, l2, l1, l2);
235
236             links.Delete(l1);
237
238             transaction.Commit();
239         }
240
241         Global.Trash = links.Count();
242     }
243
244     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran
            ↪ sactionLogFilename);
245 }

```

```

246 [Fact]
247 public static void TransactionDamage()
248 {
249     var itself = _constants.Itself;
250
251     var tempDatabaseFilename = Path.GetTempFileName();
252     var tempTransactionLogFilename = Path.GetTempFileName();
253
254     // Commit
255     using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
256         ↪ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
257         ↪ tempTransactionLogFilename))
258     using (var links = new UInt64Links(memoryAdapter))
259     {
260         using (var transaction = memoryAdapter.BeginTransaction())
261         {
262             var l1 = links.CreateAndUpdate(itself, itself);
263             var l2 = links.CreateAndUpdate(itself, itself);
264
265             Global.Trash = links.Update(l2, l2, l1, l2);
266
267             links.Delete(l1);
268
269             transaction.Commit();
270         }
271
272         Global.Trash = links.Count();
273     }
274
275     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran_
276         ↪ sactionLogFilename);
277
278     // Damage database
279     FileHelpers.WriteFirst(tempTransactionLogFilename, new
280         ↪ UInt64LinksTransactionsLayer.Transition(new UniqueTimestampFactory(), 555));
281
282     // Try load damaged database
283     try
284     {
285         // TODO: Fix
286         using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
287             ↪ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
288             ↪ tempTransactionLogFilename))
289         using (var links = new UInt64Links(memoryAdapter))
290         {
291             Global.Trash = links.Count();
292         }
293     }
294     catch (NotSupportedException ex)
295     {
296         Assert.True(ex.Message == "Database is damaged, autorecovery is not supported
297             ↪ yet.");
298     }
299
300     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran_
301         ↪ sactionLogFilename);
302
303     File.Delete(tempDatabaseFilename);
304     File.Delete(tempTransactionLogFilename);
305 }
306
307 [Fact]
308 public static void Bug1Test()
309 {
310     var tempDatabaseFilename = Path.GetTempFileName();
311     var tempTransactionLogFilename = Path.GetTempFileName();
312
313     var itself = _constants.Itself;
314
315     // User Code Error (Autoreverted), some data saved
316     try
317     {
318         ulong l1;
319         ulong l2;

```

```

315     using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
        ↳ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
        ↳ tempTransactionLogFilename))
316     using (var links = new UInt64Links(memoryAdapter))
317     {
318         l1 = links.CreateAndUpdate(itself, itself);
319         l2 = links.CreateAndUpdate(itself, itself);
320
321         l2 = links.Update(l2, l2, l1, l2);
322
323         links.CreateAndUpdate(l2, itself);
324         links.CreateAndUpdate(l2, itself);
325     }
326
327     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(temp
        ↳ TransactionLogFilename);
328
329     using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
        ↳ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
        ↳ tempTransactionLogFilename))
330     using (var links = new UInt64Links(memoryAdapter))
331     {
332         using (var transaction = memoryAdapter.BeginTransaction())
333         {
334             l2 = links.Update(l2, l1);
335
336             links.Delete(l2);
337
338             ExceptionThrower();
339
340             transaction.Commit();
341         }
342
343         Global.Trash = links.Count();
344     }
345 }
346 catch
347 {
348     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(temp
        ↳ TransactionLogFilename);
349 }
350
351 File.Delete(tempDatabaseFilename);
352 File.Delete(tempTransactionLogFilename);
353 }
354
355 private static void ExceptionThrower() => throw new InvalidOperationException();
356
357 [Fact]
358 public static void PathsTest()
359 {
360     var source = _constants.SourcePart;
361     var target = _constants.TargetPart;
362
363     using (var scope = new TempLinksTestScope())
364     {
365         var links = scope.Links;
366         var l1 = links.CreatePoint();
367         var l2 = links.CreatePoint();
368
369         var r1 = links.GetByKeys(l1, source, target, source);
370         var r2 = links.CheckPathExistence(l2, l2, l2, l2);
371     }
372 }
373
374 [Fact]
375 public static void RecursiveStringFormattingTest()
376 {
377     using (var scope = new TempLinksTestScope(useSequences: true))
378     {
379         var links = scope.Links;
380         var sequences = scope.Sequences; // TODO: Auto use sequences on Sequences getter.
381
382         var a = links.CreatePoint();
383         var b = links.CreatePoint();
384         var c = links.CreatePoint();
385
386         var ab = links.CreateAndUpdate(a, b);
387         var cb = links.CreateAndUpdate(c, b);

```

```

388     var ac = links.CreateAndUpdate(a, c);
389
390     a = links.Update(a, c, b);
391     b = links.Update(b, a, c);
392     c = links.Update(c, a, b);
393
394     Debug.WriteLine(links.FormatStructure(ab, link => link.IsFullPoint(), true));
395     Debug.WriteLine(links.FormatStructure(cb, link => link.IsFullPoint(), true));
396     Debug.WriteLine(links.FormatStructure(ac, link => link.IsFullPoint(), true));
397
398     Assert.True(links.FormatStructure(cb, link => link.IsFullPoint(), true) ==
399         ↪ "(5:(4:5 (6:5 4)) 6)");
400     Assert.True(links.FormatStructure(ac, link => link.IsFullPoint(), true) ==
401         ↪ "(6:(5:(4:5 6) 6) 4)");
402     Assert.True(links.FormatStructure(ab, link => link.IsFullPoint(), true) ==
403         ↪ "(4:(5:4 (6:5 4)) 6)");
404
405     // TODO: Think how to build balanced syntax tree while formatting structure (eg.
406     ↪ "(4:(5:4 6) (6:5 4))" instead of "(4:(5:4 (6:5 4)) 6)"
407
408     Assert.True(sequences.SafeFormatSequence(cb, DefaultFormatter, false) ==
409         ↪ "{5}{5}{4}{6}");
410     Assert.True(sequences.SafeFormatSequence(ac, DefaultFormatter, false) ==
411         ↪ "{5}{6}{6}{4}");
412     Assert.True(sequences.SafeFormatSequence(ab, DefaultFormatter, false) ==
413         ↪ "{4}{5}{4}{6}");
414 }
415
416 private static void DefaultFormatter(StringBuilder sb, ulong link)
417 {
418     sb.Append(link.ToString());
419 }
420
421 #endregion
422
423 #region Performance
424
425 /*
426 public static void RunAllPerformanceTests()
427 {
428     try
429     {
430         links.TestLinksInSteps();
431     }
432     catch (Exception ex)
433     {
434         ex.WriteToConsole();
435     }
436
437     return;
438
439     try
440     {
441         //ThreadPool.SetMaxThreads(2, 2);
442
443         // Запускаем все тесты дважды, чтобы первоначальная инициализация не повлияла на
444         ↪ результат
445         // Также это дополнительно помогает в отладке
446         // Увеличивает вероятность попадания информации в кэши
447         for (var i = 0; i < 10; i++)
448         {
449             //0 - 10 ГБ
450             //Каждые 100 МБ срез цифр
451
452             //links.TestGetSourceFunction();
453             //links.TestGetSourceFunctionInParallel();
454             //links.TestGetTargetFunction();
455             //links.TestGetTargetFunctionInParallel();
456             links.Create64BillionLinks();
457
458             links.TestRandomSearchFixed();
459             //links.Create64BillionLinksInParallel();
460             links.TestEachFunction();
461             //links.TestForeach();
462             //links.TestParallelForeach();
463         }
464
465         links.TestDeletionOfAllLinks();
466     }
467 }
468

```

```

459     }
460     catch (Exception ex)
461     {
462         ex.WriteToConsole();
463     }
464 }*/
465
466 /*
467 public static void TestLinksInSteps()
468 {
469     const long gibibyte = 1024 * 1024 * 1024;
470     const long mebibyte = 1024 * 1024;
471
472     var totalLinksToCreate = gibibyte /
↪ Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
474     var linksStep = 102 * mebibyte /
↪ Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
475
476     var creationMeasurements = new List<TimeSpan>();
477     var searchMeasurements = new List<TimeSpan>();
478     var deletionMeasurements = new List<TimeSpan>();
479
480     GetBaseRandomLoopOverhead(linksStep);
481     GetBaseRandomLoopOverhead(linksStep);
482
483     var stepLoopOverhead = GetBaseRandomLoopOverhead(linksStep);
484
485     ConsoleHelpers.Debug("Step loop overhead: {0}.", stepLoopOverhead);
486
487     var loops = totalLinksToCreate / linksStep;
488
489     for (int i = 0; i < loops; i++)
490     {
491         creationMeasurements.Add(Measure(() => links.RunRandomCreations(linksStep)));
492         searchMeasurements.Add(Measure(() => links.RunRandomSearches(linksStep)));
493
494         Console.WriteLine("\rC + S {0}/{1}", i + 1, loops);
495     }
496
497     ConsoleHelpers.Debug();
498
499     for (int i = 0; i < loops; i++)
500     {
501         deletionMeasurements.Add(Measure(() => links.RunRandomDeletions(linksStep)));
502
503         Console.WriteLine("\rD {0}/{1}", i + 1, loops);
504     }
505
506     ConsoleHelpers.Debug();
507
508     ConsoleHelpers.Debug("C S D");
509
510     for (int i = 0; i < loops; i++)
511     {
512         ConsoleHelpers.Debug("{0} {1} {2}", creationMeasurements[i],
↪ searchMeasurements[i], deletionMeasurements[i]);
513     }
514
515     ConsoleHelpers.Debug("C S D (no overhead)");
516
517     for (int i = 0; i < loops; i++)
518     {
519         ConsoleHelpers.Debug("{0} {1} {2}", creationMeasurements[i] - stepLoopOverhead,
↪ searchMeasurements[i] - stepLoopOverhead, deletionMeasurements[i] - stepLoopOverhead);
520     }
521
522     ConsoleHelpers.Debug("All tests done. Total links left in database: {0}.",
↪ links.Total);
523 }
524
525 private static void CreatePoints(this Platform.Links.Data.Core.Doublets.Links links, long
↪ amountToCreate)
526 {
527     for (long i = 0; i < amountToCreate; i++)
528         links.Create(0, 0);
529 }
530
531 private static TimeSpan GetBaseRandomLoopOverhead(long loops)

```

```

532 {
533     return Measure(() =>
534     {
535         ulong maxValue = RandomHelpers.DefaultFactory.NextUInt64();
536         ulong result = 0;
537         for (long i = 0; i < loops; i++)
538         {
539             var source = RandomHelpers.DefaultFactory.NextUInt64(maxValue);
540             var target = RandomHelpers.DefaultFactory.NextUInt64(maxValue);
541
542             result += maxValue + source + target;
543         }
544         Global.Trash = result;
545     });
546 }
547 */
548
549 [Fact(Skip = "performance test")]
550 public static void GetSourceTest()
551 {
552     using (var scope = new TempLinksTestScope())
553     {
554         var links = scope.Links;
555         ConsoleHelpers.Debug("Testing GetSource function with {0} Iterations.",
556             ↪ Iterations);
557
558         ulong counter = 0;
559
560         //var firstLink = links.First();
561         // Создаём одну связь, из которой будет производить считывание
562         var firstLink = links.Create();
563
564         var sw = Stopwatch.StartNew();
565
566         // Тестируем саму функцию
567         for (ulong i = 0; i < Iterations; i++)
568         {
569             counter += links.GetSource(firstLink);
570         }
571
572         var elapsedTime = sw.Elapsed;
573
574         var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
575
576         // Удаляем связь, из которой производилось считывание
577         links.Delete(firstLink);
578
579         ConsoleHelpers.Debug(
580             "({0} Iterations of GetSource function done in {1} ({2} Iterations per
581             ↪ second), counter result: {3}",
582             Iterations, elapsedTime, (long)iterationsPerSecond, counter);
583     }
584 }
585
586 [Fact(Skip = "performance test")]
587 public static void GetSourceInParallel()
588 {
589     using (var scope = new TempLinksTestScope())
590     {
591         var links = scope.Links;
592         ConsoleHelpers.Debug("Testing GetSource function with {0} Iterations in
593             ↪ parallel.", Iterations);
594
595         long counter = 0;
596
597         //var firstLink = links.First();
598         var firstLink = links.Create();
599
600         var sw = Stopwatch.StartNew();
601
602         // Тестируем саму функцию
603         Parallel.For(0, Iterations, x =>
604         {
605             Interlocked.Add(ref counter, (long)links.GetSource(firstLink));
606             //Interlocked.Increment(ref counter);
607         });
608
609         var elapsedTime = sw.Elapsed;
610
611         var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;

```

```

609         links.Delete(firstLink);
610
611     ConsoleHelpers.Debug(
612         "{0} Iterations of GetSource function done in {1} ({2} Iterations per
613         ↳ second), counter result: {3}",
614         Iterations, elapsedTime, (long)iterationsPerSecond, counter);
615     }
616 }
617
618 [Fact(Skip = "performance test")]
619 public static void TestGetTarget()
620 {
621     using (var scope = new TempLinksTestScope())
622     {
623         var links = scope.Links;
624         ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations.",
625         ↳ Iterations);
626
627         ulong counter = 0;
628
629         //var firstLink = links.First();
630         var firstLink = links.Create();
631
632         var sw = Stopwatch.StartNew();
633
634         for (ulong i = 0; i < Iterations; i++)
635         {
636             counter += links.GetTarget(firstLink);
637         }
638
639         var elapsedTime = sw.Elapsed;
640
641         var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
642
643         links.Delete(firstLink);
644
645         ConsoleHelpers.Debug(
646             "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
647             ↳ second), counter result: {3}",
648             Iterations, elapsedTime, (long)iterationsPerSecond, counter);
649     }
650 }
651
652 [Fact(Skip = "performance test")]
653 public static void TestGetTargetInParallel()
654 {
655     using (var scope = new TempLinksTestScope())
656     {
657         var links = scope.Links;
658         ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations in
659         ↳ parallel.", Iterations);
660
661         long counter = 0;
662
663         //var firstLink = links.First();
664         var firstLink = links.Create();
665
666         var sw = Stopwatch.StartNew();
667
668         Parallel.For(0, Iterations, x =>
669         {
670             Interlocked.Add(ref counter, (long)links.GetTarget(firstLink));
671             //Interlocked.Increment(ref counter);
672         });
673
674         var elapsedTime = sw.Elapsed;
675
676         var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
677
678         links.Delete(firstLink);
679
680         ConsoleHelpers.Debug(
681             "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
682             ↳ second), counter result: {3}",
683             Iterations, elapsedTime, (long)iterationsPerSecond, counter);
684     }
685 }

```

```

683 // TODO: Заполнить базу данных перед тестом
684 /*
685 [Fact]
686 public void TestRandomSearchFixed()
687 {
688     var tempFilename = Path.GetTempFileName();
689
690     using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
↵ DefaultLinksSizeStep))
691     {
692         long iterations = 64 * 1024 * 1024 /
↵ Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
693
694         ulong counter = 0;
695         var maxLink = links.Total;
696
697         ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.", iterations);
698
699         var sw = Stopwatch.StartNew();
700
701         for (var i = iterations; i > 0; i--)
702         {
703             var source =
↵ RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
704             var target =
↵ RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
705
706             counter += links.Search(source, target);
707         }
708
709         var elapsedTime = sw.Elapsed;
710
711         var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;
712
713         ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
↵ Iterations per second), c: {3}", iterations, elapsedTime, (long)iterationsPerSecond,
↵ counter);
714     }
715
716     File.Delete(tempFilename);
717 }*/
718
719 [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
720 public static void TestRandomSearchAll()
721 {
722     using (var scope = new TempLinksTestScope())
723     {
724         var links = scope.Links;
725         ulong counter = 0;
726
727         var maxLink = links.Count();
728
729         var iterations = links.Count();
730
731         ConsoleHelpers.Debug("Testing Random Search with {0} Iterations.",
↵ links.Count());
732
733         var sw = Stopwatch.StartNew();
734
735         for (var i = iterations; i > 0; i--)
736         {
737             var linksAddressRange = new
↵ Range<ulong>(_constants.PossibleInnerReferencesRange.Minimum, maxLink);
738
739             var source = RandomHelpers.Default.NextUInt64(linksAddressRange);
740             var target = RandomHelpers.Default.NextUInt64(linksAddressRange);
741
742             counter += links.SearchOrDefault(source, target);
743         }
744
745         var elapsedTime = sw.Elapsed;
746
747         var iterationsPerSecond = iterations / elapsedTime.TotalSeconds;
748
749         ConsoleHelpers.Debug("{0} Iterations of Random Search done in {1} ({2}
↵ Iterations per second), c: {3}",
↵ iterations, elapsedTime, (long)iterationsPerSecond, counter);
750     }
751 }
752 }
753

```



```

754 [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
755 public static void TestEach()
756 {
757     using (var scope = new TempLinksTestScope())
758     {
759         var links = scope.Links;
760
761         var counter = new Counter<IList<ulong>, ulong>(links.Constants.Continue);
762
763         ConsoleHelpers.Debug("Testing Each function.");
764
765         var sw = Stopwatch.StartNew();
766
767         links.Each(counter.IncrementAndReturnTrue);
768
769         var elapsedTime = sw.Elapsed;
770
771         var linksPerSecond = counter.Count / elapsedTime.TotalSeconds;
772
773         ConsoleHelpers.Debug("{0} Iterations of Each's handler function done in {1} ({2}
↪ links per second)",
            counter, elapsedTime, (long)linksPerSecond);
774     }
775 }
776
777
778 /*
779 [Fact]
780 public static void TestForeach()
781 {
782     var tempFilename = Path.GetTempFileName();
783
784     using (var links = new Platform.Links.Data.Core.Doublents.Links(tempFilename,
↪ DefaultLinksSizeStep))
785     {
786         ulong counter = 0;
787
788         ConsoleHelpers.Debug("Testing foreach through links.");
789
790         var sw = Stopwatch.StartNew();
791
792         //foreach (var link in links)
793         //{
794             counter++;
795         //}
796
797         var elapsedTime = sw.Elapsed;
798
799         var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
800
801         ConsoleHelpers.Debug("{0} Iterations of Foreach's handler block done in {1} ({2}
↪ links per second)", counter, elapsedTime, (long)linksPerSecond);
802     }
803
804     File.Delete(tempFilename);
805 }
806 */
807
808 /*
809 [Fact]
810 public static void TestParallelForeach()
811 {
812     var tempFilename = Path.GetTempFileName();
813
814     using (var links = new Platform.Links.Data.Core.Doublents.Links(tempFilename,
↪ DefaultLinksSizeStep))
815     {
816         long counter = 0;
817
818         ConsoleHelpers.Debug("Testing parallel foreach through links.");
819
820         var sw = Stopwatch.StartNew();
821
822         //Parallel.ForEach((IEnumerable<ulong>)links, x =>
823         //{
824             Interlocked.Increment(ref counter);
825         //});
826
827         var elapsedTime = sw.Elapsed;
828
829

```

```

830         var linksPerSecond = (double)counter / elapsedTime.TotalSeconds;
831
832         ConsoleHelpers.Debug("{0} Iterations of Parallel Foreach's handler block done in
↪ {1} ({2} links per second)", counter, elapsedTime, (long)linksPerSecond);
833     }
834
835     File.Delete(tempFilename);
836 }
837 */
838
839 [Fact(Skip = "performance test")]
840 public static void Create64BillionLinks()
841 {
842     using (var scope = new TempLinksTestScope())
843     {
844         var links = scope.Links;
845         var linksBeforeTest = links.Count();
846
847         long linksToCreate = 64 * 1024 * 1024 /
↪ UInt64ResizableDirectMemoryLinks.LinkSizeInBytes;
848
849         ConsoleHelpers.Debug("Creating {0} links.", linksToCreate);
850
851         var elapsedTime = Performance.Measure(() =>
852         {
853             for (long i = 0; i < linksToCreate; i++)
854             {
855                 links.Create();
856             }
857         });
858
859         var linksCreated = links.Count() - linksBeforeTest;
860         var linksPerSecond = linksCreated / elapsedTime.TotalSeconds;
861
862         ConsoleHelpers.Debug("Current links count: {0}.", links.Count());
863
864         ConsoleHelpers.Debug("{0} links created in {1} ({2} links per second)",
↪ linksCreated, elapsedTime,
865             (long)linksPerSecond);
866     }
867 }
868
869 [Fact(Skip = "performance test")]
870 public static void Create64BillionLinksInParallel()
871 {
872     using (var scope = new TempLinksTestScope())
873     {
874         var links = scope.Links;
875         var linksBeforeTest = links.Count();
876
877         var sw = Stopwatch.StartNew();
878
879         long linksToCreate = 64 * 1024 * 1024 /
↪ UInt64ResizableDirectMemoryLinks.LinkSizeInBytes;
880
881         ConsoleHelpers.Debug("Creating {0} links in parallel.", linksToCreate);
882
883         Parallel.For(0, linksToCreate, x => links.Create());
884
885         var elapsedTime = sw.Elapsed;
886
887         var linksCreated = links.Count() - linksBeforeTest;
888         var linksPerSecond = linksCreated / elapsedTime.TotalSeconds;
889
890         ConsoleHelpers.Debug("{0} links created in {1} ({2} links per second)",
↪ linksCreated, elapsedTime,
891             (long)linksPerSecond);
892     }
893 }
894
895 [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
896 public static void TestDeletionOfAllLinks()
897 {
898     using (var scope = new TempLinksTestScope())
899     {
900         var links = scope.Links;
901         var linksBeforeTest = links.Count();
902
903         ConsoleHelpers.Debug("Deleting all links");
904

```

```

905         var elapsedTime = Performance.Measure(links.DeleteAll);
906
907         var linksDeleted = linksBeforeTest - links.Count();
908         var linksPerSecond = linksDeleted / elapsedTime.TotalSeconds;
909
910         ConsoleHelpers.Debug("{0} links deleted in {1} ({2} links per second)",
911             ↪ linksDeleted, elapsedTime,
912             ↪ (long)linksPerSecond);
913     }
914 }
915 #endregion
916 }
917 }

```

./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;
16                 var meaningRoot = links.CreatePoint();
17                 var one = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
18                 var powerOf2ToUnaryNumberConverter = new
19                     ↪ PowerOf2ToUnaryNumberConverter<ulong>(links, one);
20                 var toUnaryNumberConverter = new AddressToUnaryNumberConverter<ulong>(links,
21                     ↪ powerOf2ToUnaryNumberConverter);
22                 var random = new System.Random(0);
23                 ulong[] numbers = new ulong[N];
24                 ulong[] unaryNumbers = new ulong[N];
25                 for (int i = 0; i < N; i++)
26                 {
27                     numbers[i] = random.NextUInt64();
28                     unaryNumbers[i] = toUnaryNumberConverter.Convert(numbers[i]);
29                 }
30                 var fromUnaryNumberConverterUsingOrOperation = new
31                     ↪ UnaryNumberToAddressOrOperationConverter<ulong>(links,
32                     ↪ powerOf2ToUnaryNumberConverter);
33                 var fromUnaryNumberConverterUsingAddOperation = new
34                     ↪ UnaryNumberToAddressAddOperationConverter<ulong>(links, one);
35                 for (int i = 0; i < N; i++)
36                 {
37                     Assert.Equal(numbers[i],
38                         ↪ fromUnaryNumberConverterUsingOrOperation.Convert(unaryNumbers[i]));
39                     Assert.Equal(numbers[i],
40                         ↪ fromUnaryNumberConverterUsingAddOperation.Convert(unaryNumbers[i]));
41                 }
42             }
43         }
44     }
45 }

```

./Platform.Data.Doublets.Tests/UnicodeConvertersTests.cs

```

1  using Xunit;
2  using Platform.Interfaces;
3  using Platform.Memory;
4  using Platform.Reflection;
5  using Platform.Scopes;
6  using Platform.Data.Doublets.Incrementers;
7  using Platform.Data.Doublets.Numbers.Raw;
8  using Platform.Data.Doublets.Numbers.Unary;
9  using Platform.Data.Doublets.PropertyOperators;
10 using Platform.Data.Doublets.ResizableDirectMemory;
11 using Platform.Data.Doublets.Sequences.Converters;
12 using Platform.Data.Doublets.Sequences.Indexes;
13 using Platform.Data.Doublets.Sequences.Walkers;
14 using Platform.Data.Doublets.Unicode;
15
16 namespace Platform.Data.Doublets.Tests

```

```

17 {
18     public static class UnicodeConvertersTests
19     {
20         [Fact]
21         public static void CharAndUnaryNumberUnicodeSymbolConvertersTest()
22         {
23             using (var scope = new TempLinksTestScope())
24             {
25                 var links = scope.Links;
26                 var meaningRoot = links.CreatePoint();
27                 var one = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
28                 var powerOf2ToUnaryNumberConverter = new
29                     ↪ PowerOf2ToUnaryNumberConverter<ulong>(links, one);
30                 var addressToUnaryNumberConverter = new
31                     ↪ AddressToUnaryNumberConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
32                 var unaryNumberToAddressConverter = new
33                     ↪ UnaryNumberToAddressOrOperationConverter<ulong>(links,
34                     ↪ powerOf2ToUnaryNumberConverter);
35                 TestCharAndUnicodeSymbolConverters(links, meaningRoot,
36                     ↪ addressToUnaryNumberConverter, unaryNumberToAddressConverter);
37             }
38         }
39
40         [Fact]
41         public static void CharAndRawNumberUnicodeSymbolConvertersTest()
42         {
43             using (var scope = new Scope<Types<HeapResizableDirectMemory,
44                 ↪ ResizableDirectMemoryLinks<ulong>>>())
45             {
46                 var links = scope.Use<ILinks<ulong>>>();
47                 var meaningRoot = links.CreatePoint();
48                 var addressToRawNumberConverter = new AddressToRawNumberConverter<ulong>();
49                 var rawNumberToAddressConverter = new RawNumberToAddressConverter<ulong>();
50                 TestCharAndUnicodeSymbolConverters(links, meaningRoot,
51                     ↪ addressToRawNumberConverter, rawNumberToAddressConverter);
52             }
53         }
54
55         private static void TestCharAndUnicodeSymbolConverters(ILinks<ulong> links, ulong
56             ↪ meaningRoot, IConverter<ulong> addressToNumberConverter, IConverter<ulong>
57             ↪ numberToAddressConverter)
58         {
59             var unicodeSymbolMarker = links.CreateAndUpdate(meaningRoot, links.Constants.Itself);
60             var charToUnicodeSymbolConverter = new CharToUnicodeSymbolConverter<ulong>(links,
61                 ↪ addressToNumberConverter, unicodeSymbolMarker);
62             var originalCharacter = 'H';
63             var characterLink = charToUnicodeSymbolConverter.Convert(originalCharacter);
64             var unicodeSymbolCriterionMatcher = new UnicodeSymbolCriterionMatcher<ulong>(links,
65                 ↪ unicodeSymbolMarker);
66             var unicodeSymbolToCharConverter = new UnicodeSymbolToCharConverter<ulong>(links,
67                 ↪ numberToAddressConverter, unicodeSymbolCriterionMatcher);
68             var resultingCharacter = unicodeSymbolToCharConverter.Convert(characterLink);
69             Assert.Equal(originalCharacter, resultingCharacter);
70         }
71
72         [Fact]
73         public static void StringAndUnicodeSequenceConvertersTest()
74         {
75             using (var scope = new TempLinksTestScope())
76             {
77                 var links = scope.Links;
78                 var itself = links.Constants.Itself;
79
80                 var meaningRoot = links.CreatePoint();
81                 var unaryOne = links.CreateAndUpdate(meaningRoot, itself);
82                 var unicodeSymbolMarker = links.CreateAndUpdate(meaningRoot, itself);
83                 var unicodeSequenceMarker = links.CreateAndUpdate(meaningRoot, itself);
84                 var frequencyMarker = links.CreateAndUpdate(meaningRoot, itself);
85                 var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot, itself);
86
87                 var powerOf2ToUnaryNumberConverter = new
88                     ↪ PowerOf2ToUnaryNumberConverter<ulong>(links, unaryOne);
89                 var addressToUnaryNumberConverter = new
90                     ↪ AddressToUnaryNumberConverter<ulong>(links, powerOf2ToUnaryNumberConverter);
91                 var charToUnicodeSymbolConverter = new
92                     ↪ CharToUnicodeSymbolConverter<ulong>(links, addressToUnaryNumberConverter,
93                     ↪ unicodeSymbolMarker);

```

```

79
80     var unaryNumberToAddressConverter = new
      ↳ UnaryNumberToAddressOrOperationConverter<ulong>(links,
      ↳ powerOf2ToUnaryNumberConverter);
81     var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links, unaryOne);
82     var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
      ↳ frequencyMarker, unaryOne, unaryNumberIncrementer);
83     var frequencyPropertyOperator = new PropertyOperator<ulong>(links,
      ↳ frequencyPropertyMarker, frequencyMarker);
84     var index = new FrequencyIncrementingSequenceIndex<ulong>(links,
      ↳ frequencyPropertyOperator, frequencyIncrementer);
85     var linkToItsFrequencyNumberConverter = new
      ↳ LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
      ↳ unaryNumberToAddressConverter);
86     var sequenceToItsLocalElementLevelsConverter = new
      ↳ SequenceToItsLocalElementLevelsConverter<ulong>(links,
      ↳ linkToItsFrequencyNumberConverter);
87     var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
      ↳ sequenceToItsLocalElementLevelsConverter);
88
89     var stringToUnicodeSequenceConverter = new
      ↳ StringToUnicodeSequenceConverter<ulong>(links, charToUnicodeSymbolConverter,
      ↳ index, optimalVariantConverter, unicodeSequenceMarker);
90
91     var originalString = "Hello";
92
93     var unicodeSequenceLink =
      ↳ stringToUnicodeSequenceConverter.Convert(originalString);
94
95     var unicodeSymbolCriterionMatcher = new
      ↳ UnicodeSymbolCriterionMatcher<ulong>(links, unicodeSymbolMarker);
96     var unicodeSymbolToCharConverter = new
      ↳ UnicodeSymbolToCharConverter<ulong>(links, unaryNumberToAddressConverter,
      ↳ unicodeSymbolCriterionMatcher);
97
98     var unicodeSequenceCriterionMatcher = new
      ↳ UnicodeSequenceCriterionMatcher<ulong>(links, unicodeSequenceMarker);
99
100    var sequenceWalker = new LeveledSequenceWalker<ulong>(links,
      ↳ unicodeSymbolCriterionMatcher.IsMatched);
101
102    var unicodeSequenceToStringConverter = new
      ↳ UnicodeSequenceToStringConverter<ulong>(links,
      ↳ unicodeSequenceCriterionMatcher, sequenceWalker,
      ↳ unicodeSymbolToCharConverter);
103
104    var resultingString =
      ↳ unicodeSequenceToStringConverter.Convert(unicodeSequenceLink);
105
106    Assert.Equal(originalString, resultingString);
107  }
108 }
109 }
110 }

```

Index

./Platform.Data.Doublets.Tests/ComparisonTests.cs, 141
./Platform.Data.Doublets.Tests/EqualityTests.cs, 142
./Platform.Data.Doublets.Tests/GenericLinksTests.cs, 144
./Platform.Data.Doublets.Tests/OptimalVariantSequenceTests.cs, 144
./Platform.Data.Doublets.Tests/ReadSequenceTests.cs, 146
./Platform.Data.Doublets.Tests/ResizableDirectMemoryLinksTests.cs, 147
./Platform.Data.Doublets.Tests/ScopeTests.cs, 148
./Platform.Data.Doublets.Tests/SequencesTests.cs, 148
./Platform.Data.Doublets.Tests/TempLinksTestScope.cs, 163
./Platform.Data.Doublets.Tests/TestExtensions.cs, 164
./Platform.Data.Doublets.Tests/UInt64LinksTests.cs, 166
./Platform.Data.Doublets.Tests/UnaryNumberConvertersTests.cs, 179
./Platform.Data.Doublets.Tests/UnicodeConvertersTests.cs, 179
./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, 2
./Platform.Data.Doublets/Decorators/LinksInnerReferenceExistenceValidator.cs, 2
./Platform.Data.Doublets/Decorators/LinksItselfConstantToSelfReferenceResolver.cs, 3
./Platform.Data.Doublets/Decorators/LinksNonExistentDependenciesCreator.cs, 3
./Platform.Data.Doublets/Decorators/LinksNullConstantToSelfReferenceResolver.cs, 4
./Platform.Data.Doublets/Decorators/LinksUniquenessResolver.cs, 4
./Platform.Data.Doublets/Decorators/LinksUniquenessValidator.cs, 4
./Platform.Data.Doublets/Decorators/LinksUsagesValidator.cs, 5
./Platform.Data.Doublets/Decorators/NonNullContentsLinkDeletionResolver.cs, 5
./Platform.Data.Doublets/Decorators/UInt64Links.cs, 5
./Platform.Data.Doublets/Decorators/UniLinks.cs, 7
./Platform.Data.Doublets/Doublet.cs, 11
./Platform.Data.Doublets/DoubletComparer.cs, 11
./Platform.Data.Doublets/Hybrid.cs, 12
./Platform.Data.Doublets/ILinks.cs, 14
./Platform.Data.Doublets/ILinksExtensions.cs, 14
./Platform.Data.Doublets/ISynchronizedLinks.cs, 26
./Platform.Data.Doublets/Incrementers/FrequencyIncrementer.cs, 25
./Platform.Data.Doublets/Incrementers/UnaryNumberIncrementer.cs, 25
./Platform.Data.Doublets/Link.cs, 26
./Platform.Data.Doublets/LinkExtensions.cs, 28
./Platform.Data.Doublets/LinksOperatorBase.cs, 28
./Platform.Data.Doublets/Numbers/Raw/AddressToRawNumberConverter.cs, 29
./Platform.Data.Doublets/Numbers/Raw/RawNumberToAddressConverter.cs, 29
./Platform.Data.Doublets/Numbers/Unary/AddressToUnaryNumberConverter.cs, 29
./Platform.Data.Doublets/Numbers/Unary/LinkToItsFrequencyNumberConverter.cs, 29
./Platform.Data.Doublets/Numbers/Unary/PowerOf2ToUnaryNumberConverter.cs, 30
./Platform.Data.Doublets/Numbers/Unary/UnaryNumberToAddressAddOperationConverter.cs, 31
./Platform.Data.Doublets/Numbers/Unary/UnaryNumberToAddressOrOperationConverter.cs, 31
./Platform.Data.Doublets/PropertyOperators/PropertiesOperator.cs, 32
./Platform.Data.Doublets/PropertyOperators/PropertyOperator.cs, 33
./Platform.Data.Doublets/ResizableDirectMemory/ResizableDirectMemoryLinks.ListMethods.cs, 41
./Platform.Data.Doublets/ResizableDirectMemory/ResizableDirectMemoryLinks.TreeMethods.cs, 42
./Platform.Data.Doublets/ResizableDirectMemory/ResizableDirectMemoryLinks.cs, 34
./Platform.Data.Doublets/ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.ListMethods.cs, 55
./Platform.Data.Doublets/ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.TreeMethods.cs, 55
./Platform.Data.Doublets/ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.cs, 48
./Platform.Data.Doublets/Sequences/ArrayExtensions.cs, 62
./Platform.Data.Doublets/Sequences/Converters/BalancedVariantConverter.cs, 62
./Platform.Data.Doublets/Sequences/Converters/CompressingConverter.cs, 63
./Platform.Data.Doublets/Sequences/Converters/LinksListToSequenceConverterBase.cs, 66
./Platform.Data.Doublets/Sequences/Converters/OptimalVariantConverter.cs, 66
./Platform.Data.Doublets/Sequences/Converters/SequenceToItsLocalElementLevelsConverter.cs, 67
./Platform.Data.Doublets/Sequences/CriteriaMatchers/DefaultSequenceElementCriterionMatcher.cs, 68
./Platform.Data.Doublets/Sequences/CriteriaMatchers/MarkedSequenceCriterionMatcher.cs, 68
./Platform.Data.Doublets/Sequences/DefaultSequenceAppender.cs, 69
./Platform.Data.Doublets/Sequences/DuplicateSegmentsCounter.cs, 69
./Platform.Data.Doublets/Sequences/DuplicateSegmentsProvider.cs, 69
./Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkFrequenciesCache.cs, 72
./Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkFrequency.cs, 74

./Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkToItsFrequencyValueConverter.cs, 74
./Platform.Data.Doublets/Sequences/Frequencies/Counters/MarkedSequenceSymbolFrequencyOneOffCounter.cs, 74
./Platform.Data.Doublets/Sequences/Frequencies/Counters/SequenceSymbolFrequencyOneOffCounter.cs, 74
./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyCounter.cs, 75
./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyOneOffCounter.cs, 75
./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyCounter.cs, 76
./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyOneOffCounter.cs, 76
./Platform.Data.Doublets/Sequences/HeightProviders/CachedSequenceHeightProvider.cs, 77
./Platform.Data.Doublets/Sequences/HeightProviders/DefaultSequenceRightHeightProvider.cs, 78
./Platform.Data.Doublets/Sequences/HeightProviders/ISequenceHeightProvider.cs, 78
./Platform.Data.Doublets/Sequences/IListExtensions.cs, 78
./Platform.Data.Doublets/Sequences/Indexes/CachedFrequencyIncrementingSequenceIndex.cs, 79
./Platform.Data.Doublets/Sequences/Indexes/FrequencyIncrementingSequenceIndex.cs, 79
./Platform.Data.Doublets/Sequences/Indexes/ISequenceIndex.cs, 80
./Platform.Data.Doublets/Sequences/Indexes/SequenceIndex.cs, 80
./Platform.Data.Doublets/Sequences/Indexes/SynchronizedSequenceIndex.cs, 81
./Platform.Data.Doublets/Sequences/ListFiller.cs, 82
./Platform.Data.Doublets/Sequences/Sequences.Experiments.cs, 93
./Platform.Data.Doublets/Sequences/Sequences.cs, 82
./Platform.Data.Doublets/Sequences/SequencesExtensions.cs, 119
./Platform.Data.Doublets/Sequences/SequencesOptions.cs, 119
./Platform.Data.Doublets/Sequences/SetFiller.cs, 121
./Platform.Data.Doublets/Sequences/Walkers/ISequenceWalker.cs, 121
./Platform.Data.Doublets/Sequences/Walkers/LeftSequenceWalker.cs, 121
./Platform.Data.Doublets/Sequences/Walkers/LeveledSequenceWalker.cs, 122
./Platform.Data.Doublets/Sequences/Walkers/RightSequenceWalker.cs, 124
./Platform.Data.Doublets/Sequences/Walkers/SequenceWalkerBase.cs, 124
./Platform.Data.Doublets/Stacks/Stack.cs, 125
./Platform.Data.Doublets/Stacks/StackExtensions.cs, 126
./Platform.Data.Doublets/SynchronizedLinks.cs, 126
./Platform.Data.Doublets/UInt64Link.cs, 127
./Platform.Data.Doublets/UInt64LinkExtensions.cs, 129
./Platform.Data.Doublets/UInt64LinksExtensions.cs, 129
./Platform.Data.Doublets/UInt64LinksTransactionsLayer.cs, 131
./Platform.Data.Doublets/Unicode/CharToUnicodeSymbolConverter.cs, 136
./Platform.Data.Doublets/Unicode/StringToUnicodeSequenceConverter.cs, 137
./Platform.Data.Doublets/Unicode/UnicodeMap.cs, 137
./Platform.Data.Doublets/Unicode/UnicodeSequenceCriterionMatcher.cs, 140
./Platform.Data.Doublets/Unicode/UnicodeSequenceToStringConverter.cs, 140
./Platform.Data.Doublets/Unicode/UnicodeSymbolCriterionMatcher.cs, 141
./Platform.Data.Doublets/Unicode/UnicodeSymbolToCharConverter.cs, 141