

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         public override TLink Update(IList<TLink> restrictions, IList<TLink> substitution) =>
37             ↳ Links.Update(restrictions, Links.ResolveConstantAsSelfReference(Constants.Itself,
38                 ↳ restrictions, substitution));
39     }
40 }

```

./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],
        ↳ substitution[constants.TargetPart]);
20         return Links.Update(restrictions, substitution);
21     }
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.Doublets/Doublet.cs

```
1 using System;
2 using System.Collections.Generic;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Doublets
7 {
8     public struct Doublet<T> : IEquatable<Doublet<T>>
9     {
10         private static readonly EqualityComparer<T> _equalityComparer =
11             ↳ EqualityComparer<T>.Default;
12
13         public T Source { get; set; }
14         public T Target { get; set; }
15
16         public Doublet(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(Doublet<T> other) => _equalityComparer.Equals(Source, other.Source)
25             ↳ && _equalityComparer.Equals(Target, other.Target);
26
27         public override bool Equals(object obj) => obj is Doublet<T> doublet ?
28             ↳ base.Equals(doublet) : false;
29
30         public override int GetHashCode() => (Source, Target).GetHashCode();
31     }
32 }
```

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

```

```

498         return links.Update(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),
                ↪ nameof(index));
92             if (index == _constants.IndexPart)
93             {
94                 return Index;
95             }
96             if (index == _constants.SourcePart)
97             {
98                 return Source;
99             }
100             if (index == _constants.TargetPart)
101             {
102                 return Target;
103             }
104             throw new NotSupportedException(); // Impossible path due to
                ↪ Ensure.ArgumentInRange
105         }
106         set => throw new NotSupportedException();
107     }
108

```

```

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
11            ↳ Hybrid<TLink>(source).AbsoluteValue;
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>,
11        ↳ IConverter<TLink>
12    {
13        private static readonly EqualityComparer<TLink> _equalityComparer =
14            ↳ EqualityComparer<TLink>.Default;
15
16        private readonly IConverter<int, TLink> _powerOf2ToUnaryNumberConverter;
17
18        public AddressToUnaryNumberConverter(ILinks<TLink> links, IConverter<int, TLink>
19            ↳ powerOf2ToUnaryNumberConverter) : base(links) => _powerOf2ToUnaryNumberConverter =
20            ↳ powerOf2ToUnaryNumberConverter;
21
22        public TLink Convert(TLink number)
23        {
24            var nullConstant = Links.Constants.Null;
25            var one = Integer<TLink>.One;
26            var target = nullConstant;
27            for (int i = 0; !_equalityComparer.Equals(number, default) && i <
28                ↳ NumericType<TLink>.BitsLength; i++)
29            {
30                if (_equalityComparer.Equals(Bit.And(number, one), one))
31                {
32                    target = _equalityComparer.Equals(target, nullConstant)
33                        ? _powerOf2ToUnaryNumberConverter.Convert(i)
34                        : Links.GetOrCreate(_powerOf2ToUnaryNumberConverter.Convert(i), target);
35                }
36                number = Bit.ShiftRight(number, 1);
37            }
38            return target;
39        }
40    }

```

./Platform.Data.Doublets/Numbers/Unary/LinkToFrequencyNumberConverter.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     }
78 }
```

./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
14 #pragma warning disable 0649
15 #pragma warning disable 169
16 #pragma warning disable 618
17 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
18
19 // ReSharper disable StaticMemberInGenericType
20 // ReSharper disable BuiltInTypeReferenceStyle
21 // ReSharper disable MemberCanBePrivate.Local
22 // ReSharper disable UnusedMember.Local
23
24 namespace Platform.Data.Doublets.ResizableDirectMemory
25 {
26     public partial class ResizableDirectMemoryLinks<TLink> : DisposableBase, ILinks<TLink>
27     {
28         private static readonly EqualityComparer<TLink> _equalityComparer =
29             ↪ EqualityComparer<TLink>.Default;
30         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
31
32         /// <summary>Возвращает размер одной связи в байтах.</summary>
33         public static readonly int LinkSizeInBytes = Structure<Link>.Size;
34
35         public static readonly int LinkHeaderSizeInBytes = Structure<LinkHeader>.Size;
36
37         public static readonly long DefaultLinksSizeStep = LinkSizeInBytes * 1024 * 1024;
38
39         private struct Link
40         {
41             public static readonly int SourceOffset = Marshal.OffsetOf(typeof(Link),
42                 ↪ nameof(Source)).ToInt32();
43             public static readonly int TargetOffset = Marshal.OffsetOf(typeof(Link),
44                 ↪ nameof(Target)).ToInt32();
45             public static readonly int LeftAsSourceOffset = Marshal.OffsetOf(typeof(Link),
46                 ↪ nameof(LeftAsSource)).ToInt32();
47             public static readonly int RightAsSourceOffset = Marshal.OffsetOf(typeof(Link),
48                 ↪ nameof(RightAsSource)).ToInt32();
49             public static readonly int SizeAsSourceOffset = Marshal.OffsetOf(typeof(Link),
50                 ↪ nameof(SizeAsSource)).ToInt32();
51             public static readonly int LeftAsTargetOffset = Marshal.OffsetOf(typeof(Link),
52                 ↪ nameof(LeftAsTarget)).ToInt32();
53             public static readonly int RightAsTargetOffset = Marshal.OffsetOf(typeof(Link),
54                 ↪ nameof(RightAsTarget)).ToInt32();
55             public static readonly int SizeAsTargetOffset = Marshal.OffsetOf(typeof(Link),
56                 ↪ nameof(SizeAsTarget)).ToInt32();
57
58             public TLink Source;
59             public TLink Target;
60             public TLink LeftAsSource;
61             public TLink RightAsTarget;
62             public int Size;
63         }
64     }
65 }

```

```

52     public TLink RightAsSource;
53     public TLink SizeAsSource;
54     public TLink LeftAsTarget;
55     public TLink RightAsTarget;
56     public TLink SizeAsTarget;
57
58     [MethodImpl(MethodImplOptions.AggressiveInlining)]
59     public static TLink GetSource(IntPtr pointer) => (pointer +
60         ↪ SourceOffset).GetValue<TLink>();
61     [MethodImpl(MethodImplOptions.AggressiveInlining)]
62     public static TLink GetTarget(IntPtr pointer) => (pointer +
63         ↪ TargetOffset).GetValue<TLink>();
64     [MethodImpl(MethodImplOptions.AggressiveInlining)]
65     public static TLink GetLeftAsSource(IntPtr pointer) => (pointer +
66         ↪ LeftAsSourceOffset).GetValue<TLink>();
67     [MethodImpl(MethodImplOptions.AggressiveInlining)]
68     public static TLink GetRightAsSource(IntPtr pointer) => (pointer +
69         ↪ RightAsSourceOffset).GetValue<TLink>();
70     [MethodImpl(MethodImplOptions.AggressiveInlining)]
71     public static TLink GetSizeAsSource(IntPtr pointer) => (pointer +
72         ↪ SizeAsSourceOffset).GetValue<TLink>();
73     [MethodImpl(MethodImplOptions.AggressiveInlining)]
74     public static TLink GetLeftAsTarget(IntPtr pointer) => (pointer +
75         ↪ LeftAsTargetOffset).GetValue<TLink>();
76     [MethodImpl(MethodImplOptions.AggressiveInlining)]
77     public static TLink GetRightAsTarget(IntPtr pointer) => (pointer +
78         ↪ RightAsTargetOffset).GetValue<TLink>();
79     [MethodImpl(MethodImplOptions.AggressiveInlining)]
80     public static TLink GetSizeAsTarget(IntPtr pointer) => (pointer +
81         ↪ SizeAsTargetOffset).GetValue<TLink>();
82
83     [MethodImpl(MethodImplOptions.AggressiveInlining)]
84     public static void SetSource(IntPtr pointer, TLink value) => (pointer +
85         ↪ SourceOffset).SetValue(value);
86     [MethodImpl(MethodImplOptions.AggressiveInlining)]
87     public static void SetTarget(IntPtr pointer, TLink value) => (pointer +
88         ↪ TargetOffset).SetValue(value);
89     [MethodImpl(MethodImplOptions.AggressiveInlining)]
90     public static void SetLeftAsSource(IntPtr pointer, TLink value) => (pointer +
91         ↪ LeftAsSourceOffset).SetValue(value);
92     [MethodImpl(MethodImplOptions.AggressiveInlining)]
93     public static void SetRightAsSource(IntPtr pointer, TLink value) => (pointer +
94         ↪ RightAsSourceOffset).SetValue(value);
95     [MethodImpl(MethodImplOptions.AggressiveInlining)]
96     public static void SetSizeAsSource(IntPtr pointer, TLink value) => (pointer +
97         ↪ SizeAsSourceOffset).SetValue(value);
98     [MethodImpl(MethodImplOptions.AggressiveInlining)]
99     public static void SetLeftAsTarget(IntPtr pointer, TLink value) => (pointer +
100         ↪ LeftAsTargetOffset).SetValue(value);
101     [MethodImpl(MethodImplOptions.AggressiveInlining)]
102     public static void SetRightAsTarget(IntPtr pointer, TLink value) => (pointer +
103         ↪ RightAsTargetOffset).SetValue(value);
104     [MethodImpl(MethodImplOptions.AggressiveInlining)]
105     public static void SetSizeAsTarget(IntPtr pointer, TLink value) => (pointer +
106         ↪ SizeAsTargetOffset).SetValue(value);
107 }
108
109 private struct LinksHeader
110 {
111     public static readonly int AllocatedLinksOffset =
112         ↪ Marshal.OffsetOf<typeof>(LinksHeader, nameof(AllocatedLinks)).ToInt32();
113     public static readonly int ReservedLinksOffset =
114         ↪ Marshal.OffsetOf<typeof>(LinksHeader, nameof(ReservedLinks)).ToInt32();
115     public static readonly int FreeLinksOffset = Marshal.OffsetOf<typeof>(LinksHeader,
116         ↪ nameof(FreeLinks)).ToInt32();
117     public static readonly int FirstFreeLinkOffset =
118         ↪ Marshal.OffsetOf<typeof>(LinksHeader, nameof(FirstFreeLink)).ToInt32();
119     public static readonly int FirstAsSourceOffset =
120         ↪ Marshal.OffsetOf<typeof>(LinksHeader, nameof(FirstAsSource)).ToInt32();
121     public static readonly int FirstAsTargetOffset =
122         ↪ Marshal.OffsetOf<typeof>(LinksHeader, nameof(FirstAsTarget)).ToInt32();
123     public static readonly int LastFreeLinkOffset =
124         ↪ Marshal.OffsetOf<typeof>(LinksHeader, nameof(LastFreeLink)).ToInt32();
125
126     public TLink AllocatedLinks;
127     public TLink ReservedLinks;
128     public TLink FreeLinks;

```

```

106     public TLink FirstFreeLink;
107     public TLink FirstAsSource;
108     public TLink FirstAsTarget;
109     public TLink LastFreeLink;
110     public TLink Reserved8;
111
112     [MethodImpl(MethodImplOptions.AggressiveInlining)]
113     public static TLink GetAllocatedLinks(IntPtr pointer) => (pointer +
114         ↪ AllocatedLinksOffset).GetValue<TLink>();
115     [MethodImpl(MethodImplOptions.AggressiveInlining)]
116     public static TLink GetReservedLinks(IntPtr pointer) => (pointer +
117         ↪ ReservedLinksOffset).GetValue<TLink>();
118     [MethodImpl(MethodImplOptions.AggressiveInlining)]
119     public static TLink GetFreeLinks(IntPtr pointer) => (pointer +
120         ↪ FreeLinksOffset).GetValue<TLink>();
121     [MethodImpl(MethodImplOptions.AggressiveInlining)]
122     public static TLink GetFirstFreeLink(IntPtr pointer) => (pointer +
123         ↪ FirstFreeLinkOffset).GetValue<TLink>();
124     [MethodImpl(MethodImplOptions.AggressiveInlining)]
125     public static TLink GetFirstAsSource(IntPtr pointer) => (pointer +
126         ↪ FirstAsSourceOffset).GetValue<TLink>();
127     [MethodImpl(MethodImplOptions.AggressiveInlining)]
128     public static TLink GetFirstAsTarget(IntPtr pointer) => (pointer +
129         ↪ FirstAsTargetOffset).GetValue<TLink>();
130     [MethodImpl(MethodImplOptions.AggressiveInlining)]
131     public static TLink GetLastFreeLink(IntPtr pointer) => (pointer +
132         ↪ LastFreeLinkOffset).GetValue<TLink>();
133
134     [MethodImpl(MethodImplOptions.AggressiveInlining)]
135     public static IntPtr GetFirstAsSourcePointer(IntPtr pointer) => pointer +
136         ↪ FirstAsSourceOffset;
137     [MethodImpl(MethodImplOptions.AggressiveInlining)]
138     public static IntPtr GetFirstAsTargetPointer(IntPtr pointer) => pointer +
139         ↪ FirstAsTargetOffset;
140
141     [MethodImpl(MethodImplOptions.AggressiveInlining)]
142     public static void SetAllocatedLinks(IntPtr pointer, TLink value) => (pointer +
143         ↪ AllocatedLinksOffset).SetValue(value);
144     [MethodImpl(MethodImplOptions.AggressiveInlining)]
145     public static void SetReservedLinks(IntPtr pointer, TLink value) => (pointer +
146         ↪ ReservedLinksOffset).SetValue(value);
147     [MethodImpl(MethodImplOptions.AggressiveInlining)]
148     public static void SetFreeLinks(IntPtr pointer, TLink value) => (pointer +
149         ↪ FreeLinksOffset).SetValue(value);
150     [MethodImpl(MethodImplOptions.AggressiveInlining)]
151     public static void SetFirstFreeLink(IntPtr pointer, TLink value) => (pointer +
152         ↪ FirstFreeLinkOffset).SetValue(value);
153     [MethodImpl(MethodImplOptions.AggressiveInlining)]
154     public static void SetFirstAsSource(IntPtr pointer, TLink value) => (pointer +
155         ↪ FirstAsSourceOffset).SetValue(value);
156     [MethodImpl(MethodImplOptions.AggressiveInlining)]
157     public static void SetFirstAsTarget(IntPtr pointer, TLink value) => (pointer +
158         ↪ FirstAsTargetOffset).SetValue(value);
159     [MethodImpl(MethodImplOptions.AggressiveInlining)]
160     public static void SetLastFreeLink(IntPtr pointer, TLink value) => (pointer +
161         ↪ LastFreeLinkOffset).SetValue(value);
162 }
163
164 private readonly long _memoryReservationStep;
165
166 private readonly IResizableDirectMemory _memory;
167 private IntPtr _header;
168 private IntPtr _links;
169
170 private LinksTargetsTreeMethods _targetsTreeMethods;
171 private LinksSourcesTreeMethods _sourcesTreeMethods;
172
173 // TODO: Возможно чтобы гарантированно проверять на то, является ли связь удалённой,
174 ↪ нужно использовать не список а дерево, так как так можно быстрее проверить на
175 ↪ наличие связи внутри
176 private UnusedLinksListMethods _unusedLinksListMethods;
177
178 /// <summary>
179 /// Возвращает общее число связей находящихся в хранилище.
180 /// </summary>
181 private TLink Total => Subtract(LinksHeader.GetAllocatedLinks(_header),
182     ↪ LinksHeader.GetFreeLinks(_header));

```

```

165 public LinksConstants<TLink> Constants { get; }
166
167 public ResizableDirectMemoryLinks(string address)
168     : this(address, DefaultLinksSizeStep)
169 {
170 }
171
172 /// <summary>
173 /// Создаёт экземпляр базы данных Links в файле по указанному адресу, с указанным
174   ↳ минимальным шагом расширения базы данных.
175 /// </summary>
176 /// <param name="address">Полный путь к файлу базы данных.</param>
177 /// <param name="memoryReservationStep">Минимальный шаг расширения базы данных в
178   ↳ байтах.</param>
179 public ResizableDirectMemoryLinks(string address, long memoryReservationStep)
180     : this(new FileMappedResizableDirectMemory(address, memoryReservationStep),
181   ↳ memoryReservationStep)
182 {
183 }
184
185 public ResizableDirectMemoryLinks(IResizableDirectMemory memory)
186     : this(memory, DefaultLinksSizeStep)
187 {
188 }
189
190 public ResizableDirectMemoryLinks(IResizableDirectMemory memory, long
191   ↳ memoryReservationStep)
192 {
193     Constants = Default<LinksConstants<TLink>>.Instance;
194     _memory = memory;
195     _memoryReservationStep = memoryReservationStep;
196     if (memory.ReservedCapacity < memoryReservationStep)
197     {
198         memory.ReservedCapacity = memoryReservationStep;
199     }
200     SetPointers(_memory);
201     // Гарантия корректности _memory.UsedCapacity относительно _header->AllocatedLinks
202     _memory.UsedCapacity = ((long)(Integer<TLink>)LinksHeader.GetAllocatedLinks(_header)
203   ↳ * LinkSizeInBytes) + LinkHeaderSizeInBytes;
204     // Гарантия корректности _header->ReservedLinks относительно _memory.ReservedCapacity
205     LinksHeader.SetReservedLinks(_header, (Integer<TLink>)((_memory.ReservedCapacity -
206   ↳ LinkHeaderSizeInBytes) / LinkSizeInBytes));
207 }
208
209 [MethodImpl(MethodImplOptions.AggressiveInlining)]
210 public TLink Count(IList<TLink> restrictions)
211 {
212     // Если нет ограничений, тогда возвращаем общее число связей находящихся в хранилище.
213     if (restrictions.Count == 0)
214     {
215         return Total;
216     }
217     if (restrictions.Count == 1)
218     {
219         var index = restrictions[Constants.IndexPart];
220         if (_equalityComparer.Equals(index, Constants.Any))
221         {
222             return Total;
223         }
224         return Exists(index) ? Integer<TLink>.One : Integer<TLink>.Zero;
225     }
226     if (restrictions.Count == 2)
227     {
228         var index = restrictions[Constants.IndexPart];
229         var value = restrictions[1];
230         if (_equalityComparer.Equals(index, Constants.Any))
231         {
232             if (_equalityComparer.Equals(value, Constants.Any))
233             {
234                 return Total; // Any - как отсутствие ограничения
235             }
236             return Add(_sourcesTreeMethods.CountUsages(value),
237   ↳ _targetsTreeMethods.CountUsages(value));
238         }
239         else
240         {
241             if (!Exists(index))
242             {

```

```

236         return Integer<TLink>.Zero;
237     }
238     if (_equalityComparer.Equals(value, Constants.Any))
239     {
240         return Integer<TLink>.One;
241     }
242     var storedLinkValue = GetLinkUnsafe(index);
243     if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), value) ||
244         _equalityComparer.Equals(Link.GetTarget(storedLinkValue), value))
245     {
246         return Integer<TLink>.One;
247     }
248     return Integer<TLink>.Zero;
249 }
250 }
251 if (restrictions.Count == 3)
252 {
253     var index = restrictions[Constants.IndexPart];
254     var source = restrictions[Constants.SourcePart];
255     var target = restrictions[Constants.TargetPart];
256
257     if (_equalityComparer.Equals(index, Constants.Any))
258     {
259         if (_equalityComparer.Equals(source, Constants.Any) &&
260             ↪ _equalityComparer.Equals(target, Constants.Any))
261         {
262             return Total;
263         }
264         else if (_equalityComparer.Equals(source, Constants.Any))
265         {
266             return _targetsTreeMethods.CountUsages(target);
267         }
268         else if (_equalityComparer.Equals(target, Constants.Any))
269         {
270             return _sourcesTreeMethods.CountUsages(source);
271         }
272         else //if(source != Any && target != Any)
273         {
274             // Эквивалент Exists(source, target) => Count(Any, source, target) > 0
275             var link = _sourcesTreeMethods.Search(source, target);
276             return _equalityComparer.Equals(link, Constants.Null) ?
277                 ↪ Integer<TLink>.Zero : Integer<TLink>.One;
278         }
279     }
280     else
281     {
282         if (!Exists(index))
283         {
284             return Integer<TLink>.Zero;
285         }
286         if (_equalityComparer.Equals(source, Constants.Any) &&
287             ↪ _equalityComparer.Equals(target, Constants.Any))
288         {
289             return Integer<TLink>.One;
290         }
291         var storedLinkValue = GetLinkUnsafe(index);
292         if (!_equalityComparer.Equals(source, Constants.Any) &&
293             ↪ !_equalityComparer.Equals(target, Constants.Any))
294         {
295             if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), source) &&
296                 _equalityComparer.Equals(Link.GetTarget(storedLinkValue), target))
297             {
298                 return Integer<TLink>.One;
299             }
300             return Integer<TLink>.Zero;
301         }
302         var value = default(TLink);
303         if (_equalityComparer.Equals(source, Constants.Any))
304         {
305             value = target;
306         }
307         if (_equalityComparer.Equals(target, Constants.Any))
308         {
309             value = source;
310         }
311         if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), value) ||
312             _equalityComparer.Equals(Link.GetTarget(storedLinkValue), value))
313         {

```

```

310         return Integer<TLink>.One;
311     }
312     return Integer<TLink>.Zero;
313 }
314 }
315 throw new NotSupportedException("Другие размеры и способы ограничений не
    ↳ поддерживаются.");
316 }
317
318 [MethodImpl(MethodImplOptions.AggressiveInlining)]
319 public TLink Each(Func<IList<TLink>, TLink> handler, IList<TLink> restrictions)
320 {
321     if (restrictions.Count == 0)
322     {
323         for (TLink link = Integer<TLink>.One; _comparer.Compare(link,
    ↳ (Integer<TLink>)LinksHeader.GetAllocatedLinks(_header)) <= 0; link =
    ↳ Increment(link))
324         {
325             if (Exists(link) && _equalityComparer.Equals(handler(GetLinkStruct(link)),
    ↳ Constants.Break))
326             {
327                 return Constants.Break;
328             }
329         }
330         return Constants.Continue;
331     }
332     if (restrictions.Count == 1)
333     {
334         var index = restrictions[Constants.IndexPart];
335         if (_equalityComparer.Equals(index, Constants.Any))
336         {
337             return Each(handler, ArrayPool<TLink>.Empty);
338         }
339         if (!Exists(index))
340         {
341             return Constants.Continue;
342         }
343         return handler(GetLinkStruct(index));
344     }
345     if (restrictions.Count == 2)
346     {
347         var index = restrictions[Constants.IndexPart];
348         var value = restrictions[1];
349         if (_equalityComparer.Equals(index, Constants.Any))
350         {
351             if (_equalityComparer.Equals(value, Constants.Any))
352             {
353                 return Each(handler, ArrayPool<TLink>.Empty);
354             }
355             if (_equalityComparer.Equals(Each(handler, new[] { index, value,
    ↳ Constants.Any }), Constants.Break))
356             {
357                 return Constants.Break;
358             }
359             return Each(handler, new[] { index, Constants.Any, value });
360         }
361         else
362         {
363             if (!Exists(index))
364             {
365                 return Constants.Continue;
366             }
367             if (_equalityComparer.Equals(value, Constants.Any))
368             {
369                 return handler(GetLinkStruct(index));
370             }
371             var storedLinkValue = GetLinkUnsafe(index);
372             if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), value) ||
    ↳ _equalityComparer.Equals(Link.GetTarget(storedLinkValue), value))
373             {
374                 return handler(GetLinkStruct(index));
375             }
376             return Constants.Continue;
377         }
378     }
379     if (restrictions.Count == 3)
380     {
381
382

```

```

383 var index = restrictions[Constants.IndexPart];
384 var source = restrictions[Constants.SourcePart];
385 var target = restrictions[Constants.TargetPart];
386 if (_equalityComparer.Equals(index, Constants.Any))
387 {
388     if (_equalityComparer.Equals(source, Constants.Any) &&
389         ↪ _equalityComparer.Equals(target, Constants.Any))
390     {
391         return Each(handler, ArrayPool<TLink>.Empty);
392     }
393     else if (_equalityComparer.Equals(source, Constants.Any))
394     {
395         return _targetsTreeMethods.EachUsage(target, handler);
396     }
397     else if (_equalityComparer.Equals(target, Constants.Any))
398     {
399         return _sourcesTreeMethods.EachUsage(source, handler);
400     }
401     else //if(source != Any && target != Any)
402     {
403         var link = _sourcesTreeMethods.Search(source, target);
404         return _equalityComparer.Equals(link, Constants.Null) ?
405             ↪ Constants.Continue : handler(GetLinkStruct(link));
406     }
407 }
408 else
409 {
410     if (!Exists(index))
411     {
412         return Constants.Continue;
413     }
414     if (_equalityComparer.Equals(source, Constants.Any) &&
415         ↪ _equalityComparer.Equals(target, Constants.Any))
416     {
417         return handler(GetLinkStruct(index));
418     }
419     var storedLinkValue = GetLinkUnsafe(index);
420     if (!_equalityComparer.Equals(source, Constants.Any) &&
421         ↪ !_equalityComparer.Equals(target, Constants.Any))
422     {
423         if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), source) &&
424             ↪ _equalityComparer.Equals(Link.GetTarget(storedLinkValue), target))
425         {
426             return handler(GetLinkStruct(index));
427         }
428         return Constants.Continue;
429     }
430     var value = default(TLink);
431     if (_equalityComparer.Equals(source, Constants.Any))
432     {
433         value = target;
434     }
435     if (_equalityComparer.Equals(target, Constants.Any))
436     {
437         value = source;
438     }
439     if (_equalityComparer.Equals(Link.GetSource(storedLinkValue), value) ||
440         ↪ _equalityComparer.Equals(Link.GetTarget(storedLinkValue), value))
441     {
442         return handler(GetLinkStruct(index));
443     }
444     return Constants.Continue;
445 }
446 }
447 throw new NotSupportedException("Другие размеры и способы ограничений не
448     ↪ поддерживаются.");
449 }
450
451 /// <remarks>
452 /// TODO: Возможно можно перемещать значения, если указан индекс, но значение существует
453     ↪ в другом месте (но не в менеджере памяти, а в логике Links)
454 /// </remarks>
455 [MethodImpl(MethodImplOptions.AggressiveInlining)]
456 public TLink Update(IList<TLink> restrictions, IList<TLink> substitution)
457 {
458     var linkIndex = restrictions[Constants.IndexPart];
459     var link = GetLinkUnsafe(linkIndex);

```



```

454 // Будет корректно работать только в том случае, если пространство выделенной связи
455 ↪ предварительно заполнено нулями
456 if (!_equalityComparer.Equals(Link.GetSource(link), Constants.Null))
457 {
458     _sourcesTreeMethods.Detach(LinksHeader.GetFirstAsSourcePointer(_header),
459     ↪ linkIndex);
460 }
461 if (!_equalityComparer.Equals(Link.GetTarget(link), Constants.Null))
462 {
463     _targetsTreeMethods.Detach(LinksHeader.GetFirstAsTargetPointer(_header),
464     ↪ linkIndex);
465 }
466 Link.SetSource(link, substitution[Constants.SourcePart]);
467 Link.SetTarget(link, substitution[Constants.TargetPart]);
468 if (!_equalityComparer.Equals(Link.GetSource(link), Constants.Null))
469 {
470     _sourcesTreeMethods.Attach(LinksHeader.GetFirstAsSourcePointer(_header),
471     ↪ linkIndex);
472 }
473 if (!_equalityComparer.Equals(Link.GetTarget(link), Constants.Null))
474 {
475     _targetsTreeMethods.Attach(LinksHeader.GetFirstAsTargetPointer(_header),
476     ↪ linkIndex);
477 }
478 return linkIndex;
479 }
480
481 [MethodImpl(MethodImplOptions.AggressiveInlining)]
482 public Link<TLink> GetLinkStruct(TLink linkIndex)
483 {
484     var link = GetLinkUnsafe(linkIndex);
485     return new Link<TLink>(linkIndex, Link.GetSource(link), Link.GetTarget(link));
486 }
487
488 [MethodImpl(MethodImplOptions.AggressiveInlining)]
489 private IntPtr GetLinkUnsafe(TLink linkIndex) => _links.GetElement(LinkSizeInBytes,
490 ↪ linkIndex);
491
492 /// <remarks>
493 /// TODO: Возможно нужно будет заполнение нулями, если внешнее API ими не заполняет
494 ↪ пространство
495 /// </remarks>
496 public TLink Create(IList<TLink> restrictions)
497 {
498     var freeLink = LinksHeader.GetFirstFreeLink(_header);
499     if (!_equalityComparer.Equals(freeLink, Constants.Null))
500     {
501         _unusedLinksListMethods.Detach(freeLink);
502     }
503     else
504     {
505         var maximumPossibleInnerReference =
506         ↪ Constants.PossibleInnerReferencesRange.Maximum;
507         if (_comparer.Compare(LinksHeader.GetAllocatedLinks(_header),
508         ↪ maximumPossibleInnerReference) > 0)
509         {
510             throw new LinksLimitReachedException<TLink>(maximumPossibleInnerReference);
511         }
512         if (_comparer.Compare(LinksHeader.GetAllocatedLinks(_header),
513         ↪ Decrement(LinksHeader.GetReservedLinks(_header))) >= 0)
514         {
515             _memory.ReservedCapacity += _memory.ReservationStep;
516             SetPointers(_memory);
517             LinksHeader.SetReservedLinks(_header,
518             ↪ (Integer<TLink>)(_memory.ReservedCapacity / LinkSizeInBytes));
519         }
520         LinksHeader.SetAllocatedLinks(_header,
521         ↪ Increment(LinksHeader.GetAllocatedLinks(_header)));
522         _memory.UsedCapacity += LinkSizeInBytes;
523         freeLink = LinksHeader.GetAllocatedLinks(_header);
524     }
525     return freeLink;
526 }
527
528 public void Delete(IList<TLink> restrictions)
529 {
530     var link = restrictions[Constants.IndexPart];
531     if (_comparer.Compare(link, LinksHeader.GetAllocatedLinks(_header)) < 0)

```

```

520     {
521         _unusedLinksListMethods.AttachAsFirst(link);
522     }
523     else if (_equalityComparer.Equals(link, LinksHeader.GetAllocatedLinks(_header)))
524     {
525         LinksHeader.SetAllocatedLinks(_header,
526             ↪ Decrement(LinksHeader.GetAllocatedLinks(_header)));
527         _memory.UsedCapacity -= LinkSizeInBytes;
528         // Убираем все связи, находящиеся в списке свободных в конце файла, до тех пор,
529         ↪ пока не дойдём до первой существующей связи
530         // Позволяет оптимизировать количество выделенных связей (AllocatedLinks)
531         while ((_comparer.Compare(LinksHeader.GetAllocatedLinks(_header),
532             ↪ Integer<TLink>.Zero) > 0) &&
533             ↪ IsUnusedLink(LinksHeader.GetAllocatedLinks(_header)))
534         {
535             _unusedLinksListMethods.Detach(LinksHeader.GetAllocatedLinks(_header));
536             LinksHeader.SetAllocatedLinks(_header,
537                 ↪ Decrement(LinksHeader.GetAllocatedLinks(_header)));
538             _memory.UsedCapacity -= LinkSizeInBytes;
539         }
540     }
541 }
542
543 /// <remarks>
544 /// TODO: Возможно это должно быть событием, вызываемым из IMemory, в том случае, если
545 ↪ адрес реально поменялся
546 ///
547 /// Указатель this.links может быть в том же месте,
548 /// так как 0-я связь не используется и имеет такой же размер как Header,
549 /// поэтому header размещается в том же месте, что и 0-я связь
550 /// </remarks>
551 private void SetPointers(IDirectMemory memory)
552 {
553     if (memory == null)
554     {
555         _links = IntPtr.Zero;
556         _header = _links;
557         _unusedLinksListMethods = null;
558         _targetsTreeMethods = null;
559         _unusedLinksListMethods = null;
560     }
561     else
562     {
563         _links = memory.Pointer;
564         _header = _links;
565         _sourcesTreeMethods = new LinksSourcesTreeMethods(this);
566         _targetsTreeMethods = new LinksTargetsTreeMethods(this);
567         _unusedLinksListMethods = new UnusedLinksListMethods(_links, _header);
568     }
569 }
570
571 [MethodImpl(MethodImplOptions.AggressiveInlining)]
572 private bool Exists(TLink link)
573     => (_comparer.Compare(link, Constants.PossibleInnerReferencesRange.Minimum) >= 0)
574     && (_comparer.Compare(link, LinksHeader.GetAllocatedLinks(_header)) <= 0)
575     && !IsUnusedLink(link);
576
577 [MethodImpl(MethodImplOptions.AggressiveInlining)]
578 private bool IsUnusedLink(TLink link)
579     => _equalityComparer.Equals(LinksHeader.GetFirstFreeLink(_header), link)
580     || (_equalityComparer.Equals(Link.GetSizeAsSource(GetLinkUnsafe(link)),
581         ↪ Constants.Null)
582         && !_equalityComparer.Equals(Link.GetSource(GetLinkUnsafe(link)), Constants.Null));
583
584 #region DisposableBase
585
586 protected override bool AllowMultipleDisposeCalls => true;
587
588 protected override void Dispose(bool manual, bool wasDisposed)
589 {
590     if (!wasDisposed)
591     {
592         SetPointers(null);
593         _memory.DisposeIfPossible();
594     }
595 }
596
597 #endregion
598
599 }

```

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

```

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

```

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

```

1  using System;
2  using System.Text;
3  using System.Collections.Generic;
4  using System.Runtime.CompilerServices;
5  using Platform.Numbers;
6  using Platform.Unsafe;
7  using Platform.Collections.Methods.Trees;
8
9  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
10
11 namespace Platform.Data.Doublets.ResizableDirectMemory
12 {
13     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     }
22 }

```

```

18 private readonly ResizableDirectMemoryLinks<TLink> _memory;
19 private readonly LinksConstants<TLink> _constants;
20 protected readonly IntPtr Links;
21 protected readonly IntPtr Header;
22
23
24 protected LinksTreeMethodsBase(ResizableDirectMemoryLinks<TLink> memory)
25 {
26     Links = memory._links;
27     Header = memory._header;
28     _memory = memory;
29     _constants = memory.Constants;
30 }
31
32 [MethodImpl(MethodImplOptions.AggressiveInlining)]
33 protected abstract TLink GetTreeRoot();
34
35 [MethodImpl(MethodImplOptions.AggressiveInlining)]
36 protected abstract TLink GetBasePartValue(TLink link);
37
38 public TLink this[TLink index]
39 {
40     get
41     {
42         var root = GetTreeRoot();
43         if (GreaterOrEqualThan(index, GetSize(root)))
44         {
45             return GetZero();
46         }
47         while (!EqualToZero(root))
48         {
49             var left = GetLeftOrDefault(root);
50             var leftSize = GetSizeOrZero(left);
51             if (LessThan(index, leftSize))
52             {
53                 root = left;
54                 continue;
55             }
56             if (IsEquals(index, leftSize))
57             {
58                 return root;
59             }
60             root = GetRightOrDefault(root);
61             index = Subtract(index, Increment(leftSize));
62         }
63         return GetZero(); // TODO: Impossible situation exception (only if tree
64                             ↳ structure broken)
65     }
66 }
67
68 // TODO: Return indices range instead of references count
69 public TLink CountUsages(TLink link)
70 {
71     var root = GetTreeRoot();
72     var total = GetSize(root);
73     var totalRightIgnore = GetZero();
74     while (!EqualToZero(root))
75     {
76         var @base = GetBasePartValue(root);
77         if (LessOrEqualThan(@base, link))
78         {
79             root = GetRightOrDefault(root);
80         }
81         else
82         {
83             totalRightIgnore = Add(totalRightIgnore, Increment(GetRightSize(root)));
84             root = GetLeftOrDefault(root);
85         }
86     }
87     root = GetTreeRoot();
88     var totalLeftIgnore = GetZero();
89     while (!EqualToZero(root))
90     {
91         var @base = GetBasePartValue(root);
92         if (GreaterOrEqualThan(@base, link))
93         {
94             root = GetLeftOrDefault(root);
95         }
96         else

```

```

96         {
97             totalLeftIgnore = Add(totalLeftIgnore, Increment(GetLeftSize(root)));
98
99             root = GetRightOrDefault(root);
100         }
101     }
102     return Subtract(Subtract(total, totalRightIgnore), totalLeftIgnore);
103 }
104
105 public TLink EachUsage(TLink link, Func<IList<TLink>, TLink> handler)
106 {
107     var root = GetTreeRoot();
108     if (EqualToZero(root))
109     {
110         return _constants.Continue;
111     }
112     TLink first = GetZero(), current = root;
113     while (!EqualToZero(current))
114     {
115         var @base = GetBasePartValue(current);
116         if (GreaterOrEqualThan(@base, link))
117         {
118             if (IsEquals(@base, link))
119             {
120                 first = current;
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((Links.GetElement(LinkSizeInBytes, node) +
152         ↪ Link.SourceOffset).GetValue<TLink>());
153     sb.Append('-');
154     sb.Append('>');
155     sb.Append((Links.GetElement(LinkSizeInBytes, node) +
156         ↪ Link.TargetOffset).GetValue<TLink>());
157 }
158
159 private class LinksSourcesTreeMethods : LinksTreeMethodsBase
160 {
161     public LinksSourcesTreeMethods(ResizableDirectMemoryLinks<TLink> memory)
162         : base(memory)
163     {
164     }
165
166     protected override IntPtr GetLeftPointer(TLink node) =>
167         ↪ Links.GetElement(LinkSizeInBytes, node) + Link.LeftAsSourceOffset;
168
169     protected override IntPtr GetRightPointer(TLink node) =>
170         ↪ Links.GetElement(LinkSizeInBytes, node) + Link.RightAsSourceOffset;

```

```

169     protected override TLink GetLeftValue(TLink node) =>
170         ↳ (Links.GetElement(LinkSizeInBytes, node) +
171         ↳ Link.LeftAsSourceOffset).GetValue<TLink>();
172
173     protected override TLink GetRightValue(TLink node) =>
174         ↳ (Links.GetElement(LinkSizeInBytes, node) +
175         ↳ Link.RightAsSourceOffset).GetValue<TLink>();
176
177     protected override TLink GetSize(TLink node)
178     {
179         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
180         ↳ Link.SizeAsSourceOffset).GetValue<TLink>();
181         return Bit<TLink>.PartialRead(previousValue, 5, -5);
182     }
183
184     protected override void SetLeft(TLink node, TLink left) =>
185         ↳ (Links.GetElement(LinkSizeInBytes, node) +
186         ↳ Link.LeftAsSourceOffset).SetValue(left);
187
188     protected override void SetRight(TLink node, TLink right) =>
189         ↳ (Links.GetElement(LinkSizeInBytes, node) +
190         ↳ Link.RightAsSourceOffset).SetValue(right);
191
192     protected override void SetSize(TLink node, TLink size)
193     {
194         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
195         ↳ Link.SizeAsSourceOffset).GetValue<TLink>();
196         (Links.GetElement(LinkSizeInBytes, node) +
197         ↳ Link.SizeAsSourceOffset).SetValue(Bit<TLink>.PartialWrite(previousValue,
198         ↳ size, 5, -5));
199     }
200
201     protected override bool GetLeftIsChild(TLink node)
202     {
203         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
204         ↳ Link.SizeAsSourceOffset).GetValue<TLink>();
205         //return (Integer<TLink>)Bit<TLink>.PartialRead(previousValue, 4, 1);
206         return !_equalityComparer.Equals(Bit<TLink>.PartialRead(previousValue, 4, 1),
207         ↳ default);
208     }
209
210     protected override void SetLeftIsChild(TLink node, bool value)
211     {
212         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
213         ↳ Link.SizeAsSourceOffset).GetValue<TLink>();
214         var modified = Bit<TLink>.PartialWrite(previousValue, (Integer<TLink>)value, 4,
215         ↳ 1);
216         (Links.GetElement(LinkSizeInBytes, node) +
217         ↳ Link.SizeAsSourceOffset).SetValue(modified);
218     }
219
220     protected override bool GetRightIsChild(TLink node)
221     {
222         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
223         ↳ Link.SizeAsSourceOffset).GetValue<TLink>();
224         //return (Integer<TLink>)Bit<TLink>.PartialRead(previousValue, 3, 1);
225         return !_equalityComparer.Equals(Bit<TLink>.PartialRead(previousValue, 3, 1),
226         ↳ default);
227     }
228
229     protected override void SetRightIsChild(TLink node, bool value)
230     {
231         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
232         ↳ Link.SizeAsSourceOffset).GetValue<TLink>();
233         var modified = Bit<TLink>.PartialWrite(previousValue, (Integer<TLink>)value, 3,
234         ↳ 1);
235         (Links.GetElement(LinkSizeInBytes, node) +
236         ↳ Link.SizeAsSourceOffset).SetValue(modified);
237     }
238
239     protected override sbyte GetBalance(TLink node)
240     {
241         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
242         ↳ Link.SizeAsSourceOffset).GetValue<TLink>();
243         var value = (ulong)(Integer<TLink>)Bit<TLink>.PartialRead(previousValue, 0, 3);
244         var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) << 5) | value & 3 |
245         ↳ 124 : value & 3);
246     }

```

```

222     return unpackedValue;
223 }
224
225 protected override void SetBalance(TLink node, sbyte value)
226 {
227     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
228         ↪ Link.SizeAsSourceOffset).GetValue<TLink>();
229     var packagedValue = (TLink)(Integer<TLink>)((((byte)value >> 5) & 4) | value &
230         ↪ 3);
231     var modified = Bit<TLink>.PartialWrite(previousValue, packagedValue, 0, 3);
232     (Links.GetElement(LinkSizeInBytes, node) +
233         ↪ Link.SizeAsSourceOffset).SetValue(modified);
234 }
235
236 protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
237 {
238     var firstSource = (Links.GetElement(LinkSizeInBytes, first) +
239         ↪ Link.SourceOffset).GetValue<TLink>();
240     var secondSource = (Links.GetElement(LinkSizeInBytes, second) +
241         ↪ Link.SourceOffset).GetValue<TLink>();
242     return LessThan(firstSource, secondSource) ||
243         (IsEquals(firstSource, secondSource) &&
244         ↪ LessThan((Links.GetElement(LinkSizeInBytes, first) +
245             ↪ Link.TargetOffset).GetValue<TLink>(),
246             ↪ (Links.GetElement(LinkSizeInBytes, second) +
247             ↪ Link.TargetOffset).GetValue<TLink>()));
248 }
249
250 protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
251 {
252     var firstSource = (Links.GetElement(LinkSizeInBytes, first) +
253         ↪ Link.SourceOffset).GetValue<TLink>();
254     var secondSource = (Links.GetElement(LinkSizeInBytes, second) +
255         ↪ Link.SourceOffset).GetValue<TLink>();
256     return GreaterThan(firstSource, secondSource) ||
257         (IsEquals(firstSource, secondSource) &&
258         ↪ GreaterThan((Links.GetElement(LinkSizeInBytes, first) +
259             ↪ Link.TargetOffset).GetValue<TLink>(),
260             ↪ (Links.GetElement(LinkSizeInBytes, second) +
261             ↪ Link.TargetOffset).GetValue<TLink>()));
262 }
263
264 protected override TLink GetTreeRoot() => (Header +
265     ↪ LinksHeader.FirstAsSourceOffset).GetValue<TLink>();
266
267 protected override TLink GetBasePartValue(TLink link) =>
268     ↪ (Links.GetElement(LinkSizeInBytes, link) + Link.SourceOffset).GetValue<TLink>();
269
270 /// <summary>
271 /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
272     ↪ (концом)
273 /// по дереву (индексу) связей, отсортированному по Source, а затем по Target.
274 /// </summary>
275 /// <param name="source">Индекс связи, которая является началом на искомой
276     ↪ связи.</param>
277 /// <param name="target">Индекс связи, которая является концом на искомой
278     ↪ связи.</param>
279 /// <returns>Индекс искомой связи.</returns>
280 public TLink Search(TLink source, TLink target)
281 {
282     var root = GetTreeRoot();
283     while (!EqualToZero(root))
284     {
285         var rootSource = (Links.GetElement(LinkSizeInBytes, root) +
286             ↪ Link.SourceOffset).GetValue<TLink>();
287         var rootTarget = (Links.GetElement(LinkSizeInBytes, root) +
288             ↪ Link.TargetOffset).GetValue<TLink>();
289         if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
290             ↪ node.Key < root.Key
291         {
292             root = GetLeftOrDefault(root);
293         }
294         else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget))
295             ↪ // node.Key > root.Key
296         {
297             root = GetRightOrDefault(root);
298         }
299     }
300 }

```

```

274         }
275         else // node.Key == root.Key
276         {
277             return root;
278         }
279     }
280     return GetZero();
281 }
282
283 [MethodImpl(MethodImplOptions.AggressiveInlining)]
284 private bool FirstIsToLeftOfSecond(TLink firstSource, TLink firstTarget, TLink
    ↳ secondSource, TLink secondTarget) => LessThan(firstSource, secondSource) ||
    ↳ (IsEquals(firstSource, secondSource) && LessThan(firstTarget, secondTarget));
285
286 [MethodImpl(MethodImplOptions.AggressiveInlining)]
287 private bool FirstIsToTheRightOfSecond(TLink firstSource, TLink firstTarget, TLink
    ↳ secondSource, TLink secondTarget) => GreaterThan(firstSource, secondSource) ||
    ↳ (IsEquals(firstSource, secondSource) && GreaterThan(firstTarget, secondTarget));
288 }
289
290 private class LinksTargetsTreeMethods : LinksTreeMethodsBase
291 {
292     public LinksTargetsTreeMethods(ResizableDirectMemoryLinks<TLink> memory)
293         : base(memory)
294     {
295     }
296
297     protected override IntPtr GetLeftPointer(TLink node) =>
298         ↳ Links.GetElement(LinkSizeInBytes, node) + Link.LeftAsTargetOffset;
299
300     protected override IntPtr GetRightPointer(TLink node) =>
301         ↳ Links.GetElement(LinkSizeInBytes, node) + Link.RightAsTargetOffset;
302
303     protected override TLink GetLeftValue(TLink node) =>
304         ↳ (Links.GetElement(LinkSizeInBytes, node) +
305         ↳ Link.LeftAsTargetOffset).GetValue<TLink>();
306
307     protected override TLink GetRightValue(TLink node) =>
308         ↳ (Links.GetElement(LinkSizeInBytes, node) +
309         ↳ Link.RightAsTargetOffset).GetValue<TLink>();
310
311     protected override TLink GetSize(TLink node)
312     {
313         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
314         ↳ Link.SizeAsTargetOffset).GetValue<TLink>();
315         return Bit<TLink>.PartialRead(previousValue, 5, -5);
316     }
317
318     protected override void SetLeft(TLink node, TLink left) =>
319         ↳ (Links.GetElement(LinkSizeInBytes, node) +
320         ↳ Link.LeftAsTargetOffset).SetValue(left);
321
322     protected override void SetRight(TLink node, TLink right) =>
323         ↳ (Links.GetElement(LinkSizeInBytes, node) +
324         ↳ Link.RightAsTargetOffset).SetValue(right);
325
326     protected override void SetSize(TLink node, TLink size)
327     {
328         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
329         ↳ Link.SizeAsTargetOffset).GetValue<TLink>();
330         (Links.GetElement(LinkSizeInBytes, node) +
331         ↳ Link.SizeAsTargetOffset).SetValue(Bit<TLink>.PartialWrite(previousValue,
332         ↳ size, 5, -5));
333     }
334
335     protected override bool GetLeftIsChild(TLink node)
336     {
337         var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
338         ↳ Link.SizeAsTargetOffset).GetValue<TLink>();
339         //return (Integer<TLink>).Bit<TLink>.PartialRead(previousValue, 4, 1);
340         return !_equalityComparer.Equals(Bit<TLink>.PartialRead(previousValue, 4, 1),
341         ↳ default);
342     }
343
344     protected override void SetLeftIsChild(TLink node, bool value)
345     {

```



```

330     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
331         ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
332     var modified = Bit<TLink>.PartialWrite(previousValue, (Integer<TLink>)value, 4,
333         ↪ 1);
334     (Links.GetElement(LinkSizeInBytes, node) +
335         ↪ Link.SizeAsTargetOffset).SetValue(modified);
336 }
337
338 protected override bool GetRightIsChild(TLink node)
339 {
340     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
341         ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
342     //return (Integer<TLink>)Bit<TLink>.PartialRead(previousValue, 3, 1);
343     return !_equalityComparer.Equals(Bit<TLink>.PartialRead(previousValue, 3, 1),
344         ↪ default);
345 }
346
347 protected override void SetRightIsChild(TLink node, bool value)
348 {
349     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
350         ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
351     var modified = Bit<TLink>.PartialWrite(previousValue, (Integer<TLink>)value, 3,
352         ↪ 1);
353     (Links.GetElement(LinkSizeInBytes, node) +
354         ↪ Link.SizeAsTargetOffset).SetValue(modified);
355 }
356
357 protected override sbyte GetBalance(TLink node)
358 {
359     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
360         ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
361     var value = (ulong)(Integer<TLink>)Bit<TLink>.PartialRead(previousValue, 0, 3);
362     var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) << 5) | value & 3 |
363         ↪ 124 : value & 3);
364     return unpackedValue;
365 }
366
367 protected override void SetBalance(TLink node, sbyte value)
368 {
369     var previousValue = (Links.GetElement(LinkSizeInBytes, node) +
370         ↪ Link.SizeAsTargetOffset).GetValue<TLink>();
371     var packagedValue = (TLink)(Integer<TLink>)((((byte)value >> 5) & 4) | value &
372         ↪ 3);
373     var modified = Bit<TLink>.PartialWrite(previousValue, packagedValue, 0, 3);
374     (Links.GetElement(LinkSizeInBytes, node) +
375         ↪ Link.SizeAsTargetOffset).SetValue(modified);
376 }
377
378 protected override bool FirstIsToTheLeftOfSecond(TLink first, TLink second)
379 {
380     var firstTarget = (Links.GetElement(LinkSizeInBytes, first) +
381         ↪ Link.TargetOffset).GetValue<TLink>();
382     var secondTarget = (Links.GetElement(LinkSizeInBytes, second) +
383         ↪ Link.TargetOffset).GetValue<TLink>();
384     return LessThan(firstTarget, secondTarget) ||
385         (IsEquals(firstTarget, secondTarget) &&
386         ↪ LessThan((Links.GetElement(LinkSizeInBytes, first) +
387         ↪ Link.SourceOffset).GetValue<TLink>(),
388         ↪ (Links.GetElement(LinkSizeInBytes, second) +
389         ↪ Link.SourceOffset).GetValue<TLink>()));
390 }
391
392 protected override bool FirstIsToTheRightOfSecond(TLink first, TLink second)
393 {
394     var firstTarget = (Links.GetElement(LinkSizeInBytes, first) +
395         ↪ Link.TargetOffset).GetValue<TLink>();
396     var secondTarget = (Links.GetElement(LinkSizeInBytes, second) +
397         ↪ Link.TargetOffset).GetValue<TLink>();
398     return GreaterThan(firstTarget, secondTarget) ||
399         (IsEquals(firstTarget, secondTarget) &&
400         ↪ GreaterThan((Links.GetElement(LinkSizeInBytes, first) +
401         ↪ Link.SourceOffset).GetValue<TLink>(),
402         ↪ (Links.GetElement(LinkSizeInBytes, second) +
403         ↪ Link.SourceOffset).GetValue<TLink>()));
404 }

```

```

381         protected override TLink GetTreeRoot() => (Header +
            ↳ LinksHeader.FirstAsTargetOffset).GetValue<TLink>();
382
383         protected override TLink GetBasePartValue(TLink link) =>
            ↳ (Links.GetElement(LinkSizeInBytes, link) + Link.TargetOffset).GetValue<TLink>();
384     }
385 }
386 }

```

./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.

```

```

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

```

```

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

```

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

```

```

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

```

```

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

```

```

./Platform.Data.Doublets/ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.ListMethods.cs
1 using Platform.Collections.Methods.Lists;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Data.Doublets.ResizableDirectMemory
6 {
7     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 }

```

./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         root = left;
50         continue;
51     }
52     if (index == leftSize)
53     {
54         return root;
55     }
56     root = GetRightOrDefault(root);
57     index -= leftSize + 1;
58 }
59 return 0; // TODO: Impossible situation exception (only if tree structure
    ↳ broken)
60 }
61 }
62
63 // TODO: Return indices range instead of references count
64 public ulong CountUsages(ulong link)
65 {
66     var root = GetTreeRoot();
67     var total = GetSize(root);
68     var totalRightIgnore = OUL;
69     while (root != 0)
70     {
71         var @base = GetBasePartValue(root);
72         if (@base <= link)
73         {
74             root = GetRightOrDefault(root);
75         }
76         else
77         {
78             totalRightIgnore += GetRightSize(root) + 1;
79             root = GetLeftOrDefault(root);
80         }
81     }
82     root = GetTreeRoot();
83     var totalLeftIgnore = OUL;
84     while (root != 0)
85     {
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
153 private class LinksSourcesTreeMethods : LinksTreeMethodsBase
154 {
155     public LinksSourcesTreeMethods(UInt64ResizableDirectMemoryLinks memory)
156         : base(memory)
157     {
158     }
159
160     protected override IntPtr GetLeftPointer(ulong node) => new
161     ↪ IntPtr(&Links[node].LeftAsSource);
162
163     protected override IntPtr GetRightPointer(ulong node) => new
164     ↪ IntPtr(&Links[node].RightAsSource);
165
166     protected override ulong GetLeftValue(ulong node) => Links[node].LeftAsSource;
167     protected override ulong GetRightValue(ulong node) => Links[node].RightAsSource;
168     protected override ulong GetSize(ulong node)
169     {
170         var previousValue = Links[node].SizeAsSource;
171         //return Math.PartialRead(previousValue, 5, -5);
172         return (previousValue & 4294967264) >> 5;
173     }
174
175     protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsSource
176     ↪ = left;
177
178     protected override void SetRight(ulong node, ulong right) =>
179     ↪ Links[node].RightAsSource = right;
180
181     protected override void SetSize(ulong node, ulong size)
182     {
183         var previousValue = Links[node].SizeAsSource;
184         //var modified = Math.PartialWrite(previousValue, size, 5, -5);
185         var modified = (previousValue & 31) | ((size & 134217727) << 5);
186         Links[node].SizeAsSource = modified;
187     }
188
189     protected override bool GetLeftIsChild(ulong node)
190     {
191         var previousValue = Links[node].SizeAsSource;
192         //return (Integer)Math.PartialRead(previousValue, 4, 1);
193         return (previousValue & 16) >> 4 == 1UL;
194     }
195
196     protected override void SetLeftIsChild(ulong node, bool value)
197     {
198         var previousValue = Links[node].SizeAsSource;
199         //var modified = Math.PartialWrite(previousValue, (ulong)(Integer)value, 4, 1);
200         var modified = (previousValue & 4294967279) | ((value ? 1UL : 0UL) << 4);
201         Links[node].SizeAsSource = modified;

```

```

202 protected override bool GetRightIsChild(ulong node)
203 {
204     var previousValue = Links[node].SizeAsSource;
205     //return (Integer)Math.PartialRead(previousValue, 3, 1);
206     return (previousValue & 8) >> 3 == 1UL;
207 }
208
209 protected override void SetRightIsChild(ulong node, bool value)
210 {
211     var previousValue = Links[node].SizeAsSource;
212     //var modified = Math.PartialWrite(previousValue, (ulong)(Integer)value, 3, 1);
213     var modified = (previousValue & 4294967287) | ((value ? 1UL : 0UL) << 3);
214     Links[node].SizeAsSource = modified;
215 }
216
217 protected override sbyte GetBalance(ulong node)
218 {
219     var previousValue = Links[node].SizeAsSource;
220     //var value = Math.PartialRead(previousValue, 0, 3);
221     var value = previousValue & 7;
222     var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) << 5) | value & 3 |
    ↪ 124 : value & 3);
223     return unpackedValue;
224 }
225
226 protected override void SetBalance(ulong node, sbyte value)
227 {
228     var previousValue = Links[node].SizeAsSource;
229     var packagedValue = (ulong)((((byte)value >> 5) & 4) | value & 3);
230     //var modified = Math.PartialWrite(previousValue, packagedValue, 0, 3);
231     var modified = (previousValue & 4294967288) | (packagedValue & 7);
232     Links[node].SizeAsSource = modified;
233 }
234
235 protected override bool FirstIsToTheLeftOfSecond(ulong first, ulong second)
236     => Links[first].Source < Links[second].Source ||
237     (Links[first].Source == Links[second].Source && Links[first].Target <
    ↪ 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 >
    ↪ Links[second].Target);
242
243 protected override ulong GetTreeRoot() => Header->FirstAsSource;
244
245 protected override ulong GetBasePartValue(ulong link) => Links[link].Source;
246
247 /// <summary>
248 /// Выполняет поиск и возвращает индекс связи с указанными Source (началом) и Target
    ↪ (концом)
249 /// по дереву (индексу) связей, отсортированному по Source, а затем по Target.
250 /// </summary>
251 /// <param name="source">Индекс связи, которая является началом на искомой
    ↪ связи.</param>
252 /// <param name="target">Индекс связи, которая является концом на искомой
    ↪ связи.</param>
253 /// <returns>Индекс искомой связи.</returns>
254 public ulong Search(ulong source, ulong target)
255 {
256     var root = Header->FirstAsSource;
257     while (root != 0)
258     {
259         var rootSource = Links[root].Source;
260         var rootTarget = Links[root].Target;
261         if (FirstIsToTheLeftOfSecond(source, target, rootSource, rootTarget)) //
    ↪ node.Key < root.Key
262         {
263             root = GetLeftOrDefault(root);
264         }
265         else if (FirstIsToTheRightOfSecond(source, target, rootSource, rootTarget))
    ↪ // node.Key > root.Key
266         {
267             root = GetRightOrDefault(root);
268         }
269         else // node.Key == root.Key
270         {
271             return root;

```

```

    }
    }
    return 0;
}

[MethodImpl(MethodImplOptions.AggressiveInlining)]
private static bool FirstIsToTheLeftOfSecond(ulong firstSource, ulong firstTarget,
    ↪ ulong secondSource, ulong secondTarget)
    => firstSource < secondSource || (firstSource == secondSource && firstTarget <
    ↪ secondTarget);

[MethodImpl(MethodImplOptions.AggressiveInlining)]
private static bool FirstIsToTheRightOfSecond(ulong firstSource, ulong firstTarget,
    ↪ ulong secondSource, ulong secondTarget)
    => firstSource > secondSource || (firstSource == secondSource && firstTarget >
    ↪ secondTarget);

[MethodImpl(MethodImplOptions.AggressiveInlining)]
protected override void ClearNode(ulong node)
{
    Links[node].LeftAsSource = OUL;
    Links[node].RightAsSource = OUL;
    Links[node].SizeAsSource = OUL;
}

[MethodImpl(MethodImplOptions.AggressiveInlining)]
protected override ulong GetZero() => OUL;

[MethodImpl(MethodImplOptions.AggressiveInlining)]
protected override ulong GetOne() => 1UL;

[MethodImpl(MethodImplOptions.AggressiveInlining)]
protected override ulong GetTwo() => 2UL;

[MethodImpl(MethodImplOptions.AggressiveInlining)]
protected override bool ValueEqualToZero(IntPtr pointer) =>
    ↪ *(ulong*)pointer.ToPointer() == OUL;

[MethodImpl(MethodImplOptions.AggressiveInlining)]
protected override bool EqualToZero(ulong value) => value == OUL;

[MethodImpl(MethodImplOptions.AggressiveInlining)]
protected override bool IsEquals(ulong first, ulong second) => first == second;

[MethodImpl(MethodImplOptions.AggressiveInlining)]
protected override bool GreaterThanZero(ulong value) => value > OUL;

[MethodImpl(MethodImplOptions.AggressiveInlining)]
protected override bool GreaterThan(ulong first, ulong second) => first > second;

[MethodImpl(MethodImplOptions.AggressiveInlining)]
protected override bool GreaterOrEqualThan(ulong first, ulong second) => first >=
    ↪ second;

[MethodImpl(MethodImplOptions.AggressiveInlining)]
protected override bool GreaterOrEqualThanZero(ulong value) => true; // value >= 0
    ↪ is always true for ulong

[MethodImpl(MethodImplOptions.AggressiveInlining)]
protected override bool LessOrEqualThanZero(ulong value) => value == 0; // value is
    ↪ always >= 0 for ulong

[MethodImpl(MethodImplOptions.AggressiveInlining)]
protected override bool LessOrEqualThan(ulong first, ulong second) => first <=
    ↪ second;

[MethodImpl(MethodImplOptions.AggressiveInlining)]
protected override bool LessThanZero(ulong value) => false; // value < 0 is always
    ↪ false for ulong

[MethodImpl(MethodImplOptions.AggressiveInlining)]
protected override bool LessThan(ulong first, ulong second) => first < second;

[MethodImpl(MethodImplOptions.AggressiveInlining)]
protected override ulong Increment(ulong value) => ++value;

[MethodImpl(MethodImplOptions.AggressiveInlining)]
protected override ulong Decrement(ulong value) => --value;

```

```

341     [MethodImpl(MethodImplOptions.AggressiveInlining)]
342     protected override ulong Add(ulong first, ulong second) => first + second;
343
344     [MethodImpl(MethodImplOptions.AggressiveInlining)]
345     protected override ulong Subtract(ulong first, ulong second) => first - second;
346 }
347
348 private class LinksTargetsTreeMethods : LinksTreeMethodsBase
349 {
350     public LinksTargetsTreeMethods(UInt64ResizableDirectMemoryLinks memory)
351         : base(memory)
352     {
353     }
354
355     //protected override IntPtr GetLeft(ulong node) => new
356     ↪ IntPtr(&Links[node].LeftAsTarget);
357
358     //protected override IntPtr GetRight(ulong node) => new
359     ↪ IntPtr(&Links[node].RightAsTarget);
360
361     //protected override ulong GetSize(ulong node) => Links[node].SizeAsTarget;
362
363     //protected override void SetLeft(ulong node, ulong left) =>
364     ↪ Links[node].LeftAsTarget = left;
365
366     //protected override void SetRight(ulong node, ulong right) =>
367     ↪ Links[node].RightAsTarget = right;
368
369     //protected override void SetSize(ulong node, ulong size) =>
370     ↪ Links[node].SizeAsTarget = size;
371
372     protected override IntPtr GetLeftPointer(ulong node) => new
373     ↪ IntPtr(&Links[node].LeftAsTarget);
374
375     protected override IntPtr GetRightPointer(ulong node) => new
376     ↪ IntPtr(&Links[node].RightAsTarget);
377
378     protected override ulong GetLeftValue(ulong node) => Links[node].LeftAsTarget;
379
380     protected override ulong GetRightValue(ulong node) => Links[node].RightAsTarget;
381
382     protected override ulong GetSize(ulong node)
383     {
384         var previousValue = Links[node].SizeAsTarget;
385         //return Math.PartialRead(previousValue, 5, -5);
386         return (previousValue & 4294967264) >> 5;
387     }
388
389     protected override void SetLeft(ulong node, ulong left) => Links[node].LeftAsTarget
390     ↪ = left;
391
392     protected override void SetRight(ulong node, ulong right) =>
393     ↪ Links[node].RightAsTarget = right;
394
395     protected override void SetSize(ulong node, ulong size)
396     {
397         var previousValue = Links[node].SizeAsTarget;
398         //var modified = Math.PartialWrite(previousValue, size, 5, -5);
399         var modified = (previousValue & 31) | ((size & 134217727) << 5);
400         Links[node].SizeAsTarget = modified;
401     }
402
403     protected override bool GetLeftIsChild(ulong node)
404     {
405         var previousValue = Links[node].SizeAsTarget;
406         //return (Integer)Math.PartialRead(previousValue, 4, 1);
407         return (previousValue & 16) >> 4 == 1UL;
408         // TODO: Check if this is possible to use
409         //var nodeSize = GetSize(node);
410         //var left = GetLeftValue(node);
411         //var leftSize = GetSizeOrZero(left);
412         //return leftSize > 0 && nodeSize > leftSize;
413     }
414
415     protected override void SetLeftIsChild(ulong node, bool value)
416     {
417         var previousValue = Links[node].SizeAsTarget;
418         //var modified = Math.PartialWrite(previousValue, (ulong)(Integer)value, 4, 1);

```

```

410         var modified = (previousValue & 4294967279) | ((value ? 1UL : 0UL) << 4);
411         Links[node].SizeAsTarget = modified;
412     }
413
414     protected override bool GetRightIsChild(ulong node)
415     {
416         var previousValue = Links[node].SizeAsTarget;
417         //return (Integer)Math.PartialRead(previousValue, 3, 1);
418         return (previousValue & 8) >> 3 == 1UL;
419         // TODO: Check if this is possible to use
420         //var nodeSize = GetSize(node);
421         //var right = GetRightValue(node);
422         //var rightSize = GetSizeOrZero(right);
423         //return rightSize > 0 && nodeSize > rightSize;
424     }
425
426     protected override void SetRightIsChild(ulong node, bool value)
427     {
428         var previousValue = Links[node].SizeAsTarget;
429         //var modified = Math.PartialWrite(previousValue, (ulong)(Integer)value, 3, 1);
430         var modified = (previousValue & 4294967287) | ((value ? 1UL : 0UL) << 3);
431         Links[node].SizeAsTarget = modified;
432     }
433
434     protected override sbyte GetBalance(ulong node)
435     {
436         var previousValue = Links[node].SizeAsTarget;
437         //var value = Math.PartialRead(previousValue, 0, 3);
438         var value = previousValue & 7;
439         var unpackedValue = (sbyte)((value & 4) > 0 ? ((value & 4) << 5) | value & 3 |
440             ↪ 124 : value & 3);
441         return unpackedValue;
442     }
443
444     protected override void SetBalance(ulong node, sbyte value)
445     {
446         var previousValue = Links[node].SizeAsTarget;
447         var packagedValue = (ulong)((((byte)value >> 5) & 4) | value & 3);
448         //var modified = Math.PartialWrite(previousValue, packagedValue, 0, 3);
449         var modified = (previousValue & 4294967288) | (packagedValue & 7);
450         Links[node].SizeAsTarget = modified;
451     }
452
453     protected override bool FirstIsToTheLeftOfSecond(ulong first, ulong second)
454     => Links[first].Target < Links[second].Target ||
455         (Links[first].Target == Links[second].Target && Links[first].Source <
456             ↪ Links[second].Source);
457
458     protected override bool FirstIsToTheRightOfSecond(ulong first, ulong second)
459     => Links[first].Target > Links[second].Target ||
460         (Links[first].Target == Links[second].Target && Links[first].Source >
461             ↪ Links[second].Source);
462
463     protected override ulong GetTreeRoot() => Header->FirstAsTarget;
464
465     protected override ulong GetBasePartValue(ulong link) => Links[link].Target;
466
467     [MethodImpl(MethodImplOptions.AggressiveInlining)]
468     protected override void ClearNode(ulong node)
469     {
470         Links[node].LeftAsTarget = 0UL;
471         Links[node].RightAsTarget = 0UL;
472         Links[node].SizeAsTarget = 0UL;
473     }
474 }
475 }
476 }
477 }

```

./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         private void HalveSequence(IList<TLink> destination, IList<TLink> source, int length)
42         {
43             var loopedLength = length - (length % 2);
44             for (var i = 0; i < loopedLength; i += 2)
45             {
46                 destination[i / 2] = Links.GetOrCreate(source[i], source[i + 1]);
47             }
48             if (length > loopedLength)
49             {
50                 destination[length / 2] = source[length - 1];
51             }
52         }
53     }
54 }

```

./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      public CompressingConverter(ILinks<TLink> links, IConverter<IList<TLink>, TLink>
66      ↪ baseConverter, LinkFrequenciesCache<TLink> doubletFrequenciesCache, TLink
67      ↪ minFrequencyToCompress, bool doInitialFrequenciesIncrement)
68      : base(links)
69      {
70          _baseConverter = baseConverter;
71          _doubletFrequenciesCache = doubletFrequenciesCache;
72          if (_comparer.Compare(minFrequencyToCompress, Integer<TLink>.One) < 0)
73          {
74              minFrequencyToCompress = Integer<TLink>.One;
75          }
76          _minFrequencyToCompress = minFrequencyToCompress;
77          _doInitialFrequenciesIncrement = doInitialFrequenciesIncrement;
78          ResetMaxDoublet();
79      }
80
81      public override TLink Convert(IList<TLink> source) =>
82      ↪ _baseConverter.Convert(Compress(source));
83
84      /// <remarks>
85      /// Original algorithm idea: https://en.wikipedia.org/wiki/Byte\_pair\_encoding .
86      /// Faster version (doublets' frequencies dictionary is not recreated).
87      /// </remarks>
88      private IList<TLink> Compress(IList<TLink> sequence)
89      {
90          if (sequence.IsNullOrEmpty())
91          {
92              return null;
93          }
94          if (sequence.Count == 1)

```

```

{
    return sequence;
}
if (sequence.Count == 2)
{
    return new[] { Links.GetOrCreate(sequence[0], sequence[1]) };
}
// TODO: arraypool with min size (to improve cache locality) or stackalloc with Sigil
var copy = new HalfDoublet[sequence.Count];
Doublet<TLink> doublet = default;
for (var i = 1; i < sequence.Count; i++)
{
    doublet.Source = sequence[i - 1];
    doublet.Target = sequence[i];
    LinkFrequency<TLink> data;
    if (_doInitialFrequenciesIncrement)
    {
        data = _doubletFrequenciesCache.IncrementFrequency(ref doublet);
    }
    else
    {
        data = _doubletFrequenciesCache.GetFrequency(ref doublet);
        if (data == null)
        {
            throw new NotSupportedException("If you ask not to increment
            ↪ frequencies, it is expected that all frequencies for the sequence
            ↪ are prepared.");
        }
    }
    copy[i - 1].Element = sequence[i - 1];
    copy[i - 1].DoubletData = data;
    UpdateMaxDoublet(ref doublet, data);
}
copy[sequence.Count - 1].Element = sequence[sequence.Count - 1];
copy[sequence.Count - 1].DoubletData = new LinkFrequency<TLink>();
if (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
{
    var newLength = ReplaceDoublets(copy);
    sequence = new TLink[newLength];
    for (int i = 0; i < newLength; i++)
    {
        sequence[i] = copy[i].Element;
    }
}
return sequence;
}

/// <remarks>
/// Original algorithm idea: https://en.wikipedia.org/wiki/Byte\_pair\_encoding
/// </remarks>
private int ReplaceDoublets(HalfDoublet[] copy)
{
    var oldLength = copy.Length;
    var newLength = copy.Length;
    while (_comparer.Compare(_maxDoubletData.Frequency, default) > 0)
    {
        var maxDoubletSource = _maxDoublet.Source;
        var maxDoubletTarget = _maxDoublet.Target;
        if (_equalityComparer.Equals(_maxDoubletData.Link, _constants.Null))
        {
            _maxDoubletData.Link = Links.GetOrCreate(maxDoubletSource, maxDoubletTarget);
        }
        var maxDoubletReplacementLink = _maxDoubletData.Link;
        oldLength--;
        var oldLengthMinusTwo = oldLength - 1;
        // Substitute all usages
        int w = 0, r = 0; // (r == read, w == write)
        for (; r < oldLength; r++)
        {
            if (_equalityComparer.Equals(copy[r].Element, maxDoubletSource) &&
            ↪ _equalityComparer.Equals(copy[r + 1].Element, maxDoubletTarget))
            {
                if (r > 0)
                {
                    var previous = copy[w - 1].Element;
                    copy[w - 1].DoubletData.DecrementFrequency();
                    copy[w - 1].DoubletData =
                    ↪ _doubletFrequenciesCache.IncrementFrequency(previous,
                    ↪ maxDoubletReplacementLink);
                }
            }
        }
    }
}

```

```

157     }
158     if (r < oldLengthMinusTwo)
159     {
160         var next = copy[r + 2].Element;
161         copy[r + 1].DoubletData.DecrementFrequency();
162         copy[w].DoubletData = _doubletFrequenciesCache.IncrementFrequency(maxDoubletReplacementLink,
163             ↪ xDoubletReplacementLink,
164             ↪ next);
165     }
166     copy[w++].Element = maxDoubletReplacementLink;
167     r++;
168     newLength--;
169 }
170 else
171 {
172     copy[w++] = copy[r];
173 }
174 }
175 if (w < newLength)
176 {
177     copy[w] = copy[r];
178 }
179 oldLength = newLength;
180 ResetMaxDoublet();
181 UpdateMaxDoublet(copy, newLength);
182 }
183 return newLength;
184 }
185
186 [MethodImpl(MethodImplOptions.AggressiveInlining)]
187 private void ResetMaxDoublet()
188 {
189     _maxDoublet = new Doublet<TLink>();
190     _maxDoubletData = new LinkFrequency<TLink>();
191 }
192
193 [MethodImpl(MethodImplOptions.AggressiveInlining)]
194 private void UpdateMaxDoublet(HalfDoublet[] copy, int length)
195 {
196     Doublet<TLink> doublet = default;
197     for (var i = 1; i < length; i++)
198     {
199         doublet.Source = copy[i - 1].Element;
200         doublet.Target = copy[i].Element;
201         UpdateMaxDoublet(ref doublet, copy[i - 1].DoubletData);
202     }
203 }
204
205 [MethodImpl(MethodImplOptions.AggressiveInlining)]
206 private void UpdateMaxDoublet(ref Doublet<TLink> doublet, LinkFrequency<TLink> data)
207 {
208     var frequency = data.Frequency;
209     var maxFrequency = _maxDoubletData.Frequency;
210     //if (frequency > _minFrequencyToCompress && (maxFrequency < frequency ||
211     ↪ (maxFrequency == frequency && doublet.Source + doublet.Target < /* gives better
212     ↪ compression string data (and gives collisions quickly) */ _maxDoublet.Source +
213     ↪ _maxDoublet.Target)))
214     if (_comparer.Compare(frequency, _minFrequencyToCompress) > 0 &&
215         (_comparer.Compare(maxFrequency, frequency) < 0 ||
216         ↪ (_equalityComparer.Equals(maxFrequency, frequency) &&
217         ↪ _comparer.Compare(Arithmetic.Add(doublet.Source, doublet.Target),
218         ↪ Arithmetic.Add(_maxDoublet.Source, _maxDoublet.Target)) > 0))) /* gives
219         ↪ better stability and better compression on sequent data and even on random
220         ↪ numbers data (but gives collisions anyway) */
221     {
222         _maxDoublet = doublet;
223         _maxDoubletData = data;
224     }
225 }
226 }
227 }

```

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

```

```

65     }
66     }
67     else
68     {
69         currentLevel = levels[i];
70         levelRepeat = 1;
71         if (skipOnce)
72         {
73             skipOnce = false;
74         }
75         else
76         {
77             sequence[w] = sequence[i - 1];
78             levels[w] = levels[i - 1];
79             previousLevel = levels[w];
80             w++;
81         }
82         if (i == length - 1)
83         {
84             sequence[w] = sequence[i];
85             levels[w] = levels[i];
86             w++;
87         }
88     }
89     }
90     length = w;
91 }
92 return links.GetOrCreate(sequence[0], sequence[1]);
93 }
94
95 private static TLink GetGreatestNeighbourLowerThanCurrentOrCurrent(TLink previous, TLink
↪ current, TLink next)
96 {
97     return _comparer.Compare(previous, next) > 0
98         ? _comparer.Compare(previous, current) < 0 ? previous : current
99         : _comparer.Compare(next, current) < 0 ? next : current;
100 }
101
102 private static TLink GetNextLowerThanCurrentOrCurrent(TLink current, TLink next) =>
↪ _comparer.Compare(next, current) < 0 ? next : current;
103
104 private static TLink GetPreviousLowerThanCurrentOrCurrent(TLink previous, TLink current)
↪ => _comparer.Compare(previous, current) < 0 ? previous : current;
105 }
106 }

```

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

```

1 using System.Collections.Generic;
2 using Platform.Interfaces;
3
4 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
5
6 namespace Platform.Data.Doublets.Sequences.Converters
7 {
8     public class SequenceToItsLocalElementLevelsConverter<TLink> : LinksOperatorBase<TLink>,
↪ IConverter<IList<TLink>>
9     {
10         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
11
12         private readonly IConverter<Doublet<TLink>, TLink> _linkToItsFrequencyToNumberConveter;
13
14         public SequenceToItsLocalElementLevelsConverter(IList<TLink> links,
↪ IConverter<Doublet<TLink>, TLink> linkToItsFrequencyToNumberConveter) : base(links)
↪ => _linkToItsFrequencyToNumberConveter = linkToItsFrequencyToNumberConveter;
15
16         public IList<TLink> Convert(IList<TLink> sequence)
17         {
18             var levels = new TLink[sequence.Count];
19             levels[0] = GetFrequencyNumber(sequence[0], sequence[1]);
20             for (var i = 1; i < sequence.Count - 1; i++)
21             {
22                 var previous = GetFrequencyNumber(sequence[i - 1], sequence[i]);
23                 var next = GetFrequencyNumber(sequence[i], sequence[i + 1]);
24                 levels[i] = _comparer.Compare(previous, next) > 0 ? previous : next;
25             }
26             levels[levels.Length - 1] = GetFrequencyNumber(sequence[sequence.Count - 2],
↪ sequence[sequence.Count - 1]);
27             return levels;
28         }
29     }

```

```

29
30     public TLink GetFrequencyNumber(TLink source, TLink target) =>
        ↪ _linkToItsFrequencyToNumberConveter.Convert(new Doublet<TLink>(source, target));
31 }
32 }

```

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

```

1 using Platform.Interfaces;
2
3 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5 namespace Platform.Data.Doublets.Sequences.CreteriaMatchers
6 {
7     public class DefaultSequenceElementCriterionMatcher<TLink> : LinksOperatorBase<TLink>,
        ↪ ICriterionMatcher<TLink>
8     {
9         public DefaultSequenceElementCriterionMatcher(ILinks<TLink> links) : base(links) { }
10        public bool IsMatched(TLink argument) => Links.IsPartialPoint(argument);
11    }
12 }

```

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

```

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

```

```

27         while (!_equalityComparer.Equals(_heightProvider.Get(cursor), default))
28         {
29             var source = Links.GetSource(cursor);
30             var target = Links.GetTarget(cursor);
31             if (_equalityComparer.Equals(_heightProvider.Get(source),
32                 ↪ _heightProvider.Get(target)))
33             {
34                 break;
35             }
36             else
37             {
38                 _stack.Push(source);
39                 cursor = target;
40             }
41             var left = cursor;
42             var right = appendant;
43             while (!_equalityComparer.Equals(cursor = _stack.Pop(), Links.Constants.Null))
44             {
45                 right = Links.GetOrCreate(left, right);
46                 left = cursor;
47             }
48             return Links.GetOrCreate(left, right);
49         }
50     }
51 }

```

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

```

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

```

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

```

1  using System;
2  using System.Linq;
3  using System.Collections.Generic;
4  using Platform.Interfaces;
5  using Platform.Collections;
6  using Platform.Collections.Lists;
7  using Platform.Collections.Segments;
8  using Platform.Collections.Segments.Walkers;
9  using Platform.Singletons;
10 using Platform.Numbers;
11 using Platform.Data.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 ItemEqualityComparer : IEqualityComparer<KeyValuePair<IList<TLink>,
27         ↪ IList<TLink>>>
28         {
29             private readonly IListEqualityComparer<TLink> _listComparer;
30             public ItemEqualityComparer() => _listComparer =
31             ↪ Default<IListEqualityComparer<TLink>>.Instance;

```

```

28     public bool Equals(KeyValuePair<IList<TLink>, IList<TLink>> left,
    ↪     KeyValuePair<IList<TLink>, IList<TLink>> right) =>
    ↪     _listComparer.Equals(left.Key, right.Key) && _listComparer.Equals(left.Value,
    ↪     right.Value);
29     public int GetHashCode(KeyValuePair<IList<TLink>, IList<TLink>> pair) =>
    ↪     (_listComparer.GetHashCode(pair.Key),
    ↪     _listComparer.GetHashCode(pair.Value)).GetHashCode();
30 }
31
32 private class ItemComparer : IComparer<KeyValuePair<IList<TLink>, IList<TLink>>>
33 {
34     private readonly IListComparer<TLink> _listComparer;
35
36     public ItemComparer() => _listComparer = Default<IListComparer<TLink>>.Instance;
37
38     public int Compare(KeyValuePair<IList<TLink>, IList<TLink>> left,
    ↪     KeyValuePair<IList<TLink>, IList<TLink>> right)
39     {
40         var intermediateResult = _listComparer.Compare(left.Key, right.Key);
41         if (intermediateResult == 0)
42         {
43             intermediateResult = _listComparer.Compare(left.Value, right.Value);
44         }
45         return intermediateResult;
46     }
47 }
48
49 public DuplicateSegmentsProvider(ILinks<TLink> links, ILinks<TLink> sequences)
50     : base(minimumStringSegmentLength: 2)
51 {
52     _links = links;
53     _sequences = sequences;
54 }
55
56 public IList<KeyValuePair<IList<TLink>, IList<TLink>>> Get()
57 {
58     _groups = new HashSet<KeyValuePair<IList<TLink>,
    ↪     IList<TLink>>>(Default<ItemEquilityComparer>.Instance);
59     var count = _links.Count();
60     _visited = new BitString((long)(Integer<TLink>)count + 1);
61     _links.Each(link =>
62     {
63         var linkIndex = _links.GetIndex(link);
64         var linkBitIndex = (long)(Integer<TLink>)linkIndex;
65         if (!_visited.Get(linkBitIndex))
66         {
67             var sequenceElements = new List<TLink>();
68             var filler = new ListFiller<TLink, TLink>(sequenceElements,
    ↪             _sequences.Constants.Break);
69             _sequences.Each(filler.AddAllValuesAndReturnConstant, new
    ↪             LinkAddress<TLink>(linkIndex));
70             if (sequenceElements.Count > 2)
71             {
72                 WalkAll(sequenceElements);
73             }
74         }
75         return _links.Constants.Continue;
76     });
77     var resultList = _groups.ToList();
78     var comparer = Default<ItemComparer>.Instance;
79     resultList.Sort(comparer);
80     #if DEBUG
81     foreach (var item in resultList)
82     {
83         PrintDuplicates(item);
84     }
85     #endif
86     return resultList;
87 }
88
89 protected override Segment<TLink> CreateSegment(IList<TLink> elements, int offset, int
    ↪     length) => new Segment<TLink>(elements, offset, length);
90
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;
141     }
142     var duplicatesKey = duplicatesItem.Key;
143     var keyString = UnicodeMap.FromLinksToString((IList<ulong>)duplicatesKey);
144     Console.WriteLine($"> {keyString} ({string.Join(", ", duplicatesKey)})");
145     var duplicatesList = duplicatesItem.Value;
146     for (int i = 0; i < duplicatesList.Count; i++)
147     {
148         ulong sequenceIndex = (Integer<TLink>)duplicatesList[i];
149         var formattedSequenceStructure = ulongLinks.FormatStructure(sequenceIndex, x =>
150             ↪ Point<ulong>.IsPartialPoint(x), (sb, link) => _ =
151             ↪ UnicodeMap.IsCharLink(link.Index) ?
152             ↪ sb.Append(UnicodeMap.FromLinkToChar(link.Index)) : sb.Append(link.Index));
153         Console.WriteLine(formattedSequenceStructure);
154         var sequenceString = UnicodeMap.FromSequenceLinkToString(sequenceIndex,
155             ↪ ulongLinks);
156         Console.WriteLine(sequenceString);
157     }
158     Console.WriteLine();
159 }
160 }

```

./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             [MethodImpl(MethodImplOptions.AggressiveInlining)]
33             public LinkFrequency<TLink> GetFrequency(TLink source, TLink target)
34             {
35                 var doublet = new Doublet<TLink>(source, target);
36                 return GetFrequency(ref doublet);
37             }
38
39             [MethodImpl(MethodImplOptions.AggressiveInlining)]
40             public LinkFrequency<TLink> GetFrequency(ref Doublet<TLink> doublet)
41             {
42                 _doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data);
43                 return data;
44             }
45
46             public void IncrementFrequencies(IList<TLink> sequence)
47             {
48                 for (var i = 1; i < sequence.Count; i++)
49                 {
50                     IncrementFrequency(sequence[i - 1], sequence[i]);
51                 }
52             }
53
54             [MethodImpl(MethodImplOptions.AggressiveInlining)]
55             public LinkFrequency<TLink> IncrementFrequency(TLink source, TLink target)
56             {
57                 var doublet = new Doublet<TLink>(source, target);
58                 return IncrementFrequency(ref doublet);
59             }
60
61             public void PrintFrequencies(IList<TLink> sequence)
62             {
63                 for (var i = 1; i < sequence.Count; i++)
64                 {
65                     PrintFrequency(sequence[i - 1], sequence[i]);
66                 }
67             }
68
69             public void PrintFrequency(TLink source, TLink target)
70             {
71                 var number = GetFrequency(source, target).Frequency;
72                 Console.WriteLine("{0},{1}) - {2}", source, target, number);
73             }
74
75             [MethodImpl(MethodImplOptions.AggressiveInlining)]
76             public LinkFrequency<TLink> IncrementFrequency(ref Doublet<TLink> doublet)
77             {
78                 if (_doubletsCache.TryGetValue(doublet, out LinkFrequency<TLink> data))
79                 {
80                     data.IncrementFrequency();
81                 }
82                 else
83                 {
84                     var link = Links.SearchOrDefault(doublet.Source, doublet.Target);
85                     data = new LinkFrequency<TLink>(Integer<TLink>.One, link);
86                     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 public void ValidateFrequencies()
94 {
95     foreach (var entry in _doubletsCache)
96     {
97         var value = entry.Value;
98         var linkIndex = value.Link;
99         if (!_equalityComparer.Equals(linkIndex, default))
100         {
101             var frequency = value.Frequency;
102             var count = _frequencyCounter.Count(linkIndex);
103             // TODO: Why `frequency` always greater than `count` by 1?
104             if (((_comparer.Compare(frequency, count) > 0) &&
105                 ↪ (_comparer.Compare(Arithmetic.Subtract(frequency, count),
106                 ↪ Integer<TLink>.One) > 0))
107                 || ((_comparer.Compare(count, frequency) > 0) &&
108                 ↪ (_comparer.Compare(Arithmetic.Subtract(count, frequency),
109                 ↪ Integer<TLink>.One) > 0)))
110             {
111                 throw new InvalidOperationException("Frequencies validation failed.");
112             }
113             //else
114             //{
115             //    if (value.Frequency > 0)
116             //    {
117             //        var frequency = value.Frequency;
118             //        linkIndex = _createLink(entry.Key.Source, entry.Key.Target);
119             //        var count = _countLinkFrequency(linkIndex);
120             //        if ((frequency > count && frequency - count > 1) || (count > frequency
121             //            ↪ && count - frequency > 1))
122             //            throw new Exception("Frequencies validation failed.");
123             //    }
124             //}
125     }
126 }
127 }
128 }
129 }

```

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

```

1 using System.Runtime.CompilerServices;
2 using Platform.Numbers;
3
4 #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                 // ...
35             }
36         }
37     }
38 }
```

```

32         return _total;
33     }
34     StopableSequenceWalker.WalkRight(_sequenceLink, _links.GetSource, _links.GetTarget,
35         ↪ IsElement, VisitElement);
36     return _total;
37 }
38 private bool IsElement(TLink x) => _equalityComparer.Equals(x, _symbol) ||
39     ↪ _links.IsPartialPoint(x); // TODO: Use SequenceElementCriteriaMatcher instead of
40     ↪ IsPartialPoint
41
42 private bool VisitElement(TLink element)
43 {
44     if (_equalityComparer.Equals(element, _symbol))
45     {
46         _total = Arithmetic.Increment(_total);
47     }
48     return true;
49 }

```

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

```

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

```

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

```

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

```

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

```

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

```

```

4
5 namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
6 {
7     public class TotalSequenceSymbolFrequencyCounter<TLink> : ICounter<TLink, TLink>
8     {
9         private readonly ILinks<TLink> _links;
10        public TotalSequenceSymbolFrequencyCounter(ILinks<TLink> links) => _links = links;
11        public TLink Count(TLink symbol) => new
12            ↳ TotalSequenceSymbolFrequencyOneOffCounter<TLink>(_links, symbol).Count();
13    }
14 }
15
16 ./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyOneOffCounter.cs
17
18 using System.Collections.Generic;
19 using Platform.Interfaces;
20 using Platform.Numbers;
21
22 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
23
24 namespace Platform.Data.Doublets.Sequences.Frequencies.Counters
25 {
26     public class TotalSequenceSymbolFrequencyOneOffCounter<TLink> : ICounter<TLink>
27     {
28         private static readonly EqualityComparer<TLink> _equalityComparer =
29             ↳ EqualityComparer<TLink>.Default;
30         private static readonly Comparer<TLink> _comparer = Comparer<TLink>.Default;
31
32         protected readonly ILinks<TLink> _links;
33         protected readonly TLink _symbol;
34         protected readonly HashSet<TLink> _visits;
35         protected TLink _total;
36
37         public TotalSequenceSymbolFrequencyOneOffCounter(ILinks<TLink> links, TLink symbol)
38         {
39             _links = links;
40             _symbol = symbol;
41             _visits = new HashSet<TLink>();
42             _total = default;
43         }
44
45         public TLink Count()
46         {
47             if (_comparer.Compare(_total, default) > 0 || _visits.Count > 0)
48             {
49                 return _total;
50             }
51             CountCore(_symbol);
52             return _total;
53         }
54
55         private void CountCore(TLink link)
56         {
57             var any = _links.Constants.Any;
58             if (_equalityComparer.Equals(_links.Count(any, link), default))
59             {
60                 CountSequenceSymbolFrequency(link);
61             }
62             else
63             {
64                 _links.Each(EachElementHandler, any, link);
65             }
66         }
67
68         protected virtual void CountSequenceSymbolFrequency(TLink link)
69         {
70             var symbolFrequencyCounter = new SequenceSymbolFrequencyOneOffCounter<TLink>(_links,
71                 ↳ link, _symbol);
72             _total = Arithmetic.Add(_total, symbolFrequencyCounter.Count());
73         }
74
75         private TLink EachElementHandler(IList<TLink> doublet)
76         {
77             var constants = _links.Constants;
78             var doubletIndex = doublet[constants.IndexPart];
79             if (_visits.Add(doubletIndex))
80             {
81                 CountCore(doubletIndex);
82             }
83             return constants.Continue;
84         }
85     }
86 }

```

```

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);
45             }
46             else
47             {
48                 height = _unaryNumberToAddressConverter.Convert(heightValue);
49             }
50             return height;
51         }
52     }
53 }
54
55 ./Platform.Data.Doublets/Sequences/HeightProviders/DefaultSequenceRightHeightProvider.cs
56 1  using Platform.Interfaces;
57 2  using Platform.Numbers;
58 3
59 4  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
60 5
61 6  namespace Platform.Data.Doublets.Sequences.HeightProviders
62 7  {
63 8      public class DefaultSequenceRightHeightProvider<TLink> : LinksOperatorBase<TLink>,
64 9          ↳ ISequenceHeightProvider<TLink>
65 10     {
66 11         private readonly ICriterionMatcher<TLink> _elementMatcher;
67 12
68 13         public DefaultSequenceRightHeightProvider(ILinks<TLink> links, ICriterionMatcher<TLink>
69 14             ↳ elementMatcher) : base(links) => _elementMatcher = elementMatcher;
70 15
71 16         public TLink Get(TLink sequence)
72 17         {
73 18             var height = default(TLink);
74 19             var pairOrElement = sequence;
75             while (!_elementMatcher.IsMatched(pairOrElement))
76             {

```

```

20         pairOrElement = Links.GetTarget(pairOrElement);
21         height = Arithmetic.Increment(height);
22     }
23     return height;
24 }
25 }
26 }

```

./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(IList<TLink> sequence)
19         {
20             var indexed = true;
21             var i = sequence.Count;
22             while (--i >= 1 && (indexed = IsIndexedWithIncrement(sequence[i - 1], sequence[i])))
23                 ⇨ { }
24         }
25     }
26 }

```



```

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

```

./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(ICollection<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(ICollection<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 }

```

```

34     private bool IsIndexedWithIncrement(TLink source, TLink target)
35     {
36         var link = Links.SearchOrDefault(source, target);
37         var indexed = !_equalityComparer.Equals(link, default);
38         if (indexed)
39         {
40             Increment(link);
41         }
42         return indexed;
43     }
44
45     private void Increment(TLink link)
46     {
47         var previousFrequency = _frequencyPropertyOperator.Get(link);
48         var frequency = _frequencyIncrementer.Increment(previousFrequency);
49         _frequencyPropertyOperator.Set(link, frequency);
50     }
51 }
52 }

```

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

```

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

```

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

```

1  using System.Collections.Generic;
2
3  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
4
5  namespace Platform.Data.Doublets.Sequences.Indexes
6  {
7      public class SequenceIndex<TLink> : LinksOperatorBase<TLink>, ISequenceIndex<TLink>
8      {
9          private static readonly EqualityComparer<TLink> _equalityComparer =
10             ↳ EqualityComparer<TLink>.Default;
11
12         public SequenceIndex(ILinks<TLink> links) : base(links) { }
13
14         public virtual bool Add(IList<TLink> sequence)
15         {
16             var indexed = true;
17             var i = sequence.Count;
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;
46                 var i = sequence.Count;
47                 while (--i >= 1 && (indexed =
48                     ⇨ !_equalityComparer.Equals(links.SearchOrDefault(sequence[i - 1],
49                     ⇨ sequence[i]), default))) { }
50                 return indexed;
51             });
52         }
53     }
54 }
```

./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 }
```

```

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(ICollection<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(ICollection<TElement> collection)
47     {
48         _list.Add(collection[0]);
49         return _returnConstant;
50     }
51
52     [MethodImpl(MethodImplOptions.AggressiveInlining)]
53     public TReturnConstant AddAllValuesAndReturnConstant(ICollection<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 звездочкой.
39     /// URL, PURL - реестр используемых во вне ссылок на ресурсы,

```

```

36  /// так же проблема может быть решена при реализации дистанционных триггеров.
37  /// Нужны ли уникальные указатели вообще?
38  /// Что если обращение к информации будет происходить через содержимое всегда?
39  ///
40  /// Писать тесты.
41  ///
42  ///
43  /// Можно убрать зависимость от конкретной реализации Links,
44  /// на зависимость от абстрактного элемента, который может быть представлен несколькими
45  /// ↳ способами.
46  ///
47  /// Можно ли как-то сделать один общий интерфейс
48  ///
49  /// Блокчейн и/или гит для распределённой записи транзакций.
50  ///
51  /// </remarks>
52  public partial class Sequences : ILinks<LinkIndex> // IList<string>, IList<LinkIndex[]>
53  ↳ (после завершения реализации Sequences)
54  {
55      /// <summary>Возвращает значение LinkIndex, обозначающее любое количество
56      ↳ связей.</summary>
57      public const LinkIndex ZeroOrMany = LinkIndex.MaxValue;
58
59      public SequencesOptions<LinkIndex> Options { get; }
60      public SynchronizedLinks<LinkIndex> Links { get; }
61      private readonly ISynchronization _sync;
62
63      public LinksConstants<LinkIndex> Constants { get; }
64
65      public Sequences(SynchronizedLinks<LinkIndex> links, SequencesOptions<LinkIndex> options)
66      {
67          Links = links;
68          _sync = links.SyncRoot;
69          Options = options;
70          Options.ValidateOptions();
71          Options.InitOptions(Links);
72          Constants = Default<LinksConstants<LinkIndex>>.Instance;
73      }
74
75      public Sequences(SynchronizedLinks<LinkIndex> links)
76      : this(links, new SequencesOptions<LinkIndex>())
77      {
78      }
79
80      public bool IsSequence(LinkIndex sequence)
81      {
82          return _sync.ExecuteReadOperation(() =>
83          {
84              if (Options.UseSequenceMarker)
85              {
86                  return Options.MarkedSequenceMatcher.IsMatched(sequence);
87              }
88              return !Links.Unsync.IsPartialPoint(sequence);
89          });
90      }
91
92      [MethodImpl(MethodImplOptions.AggressiveInlining)]
93      private LinkIndex GetSequenceByElements(LinkIndex sequence)
94      {
95          if (Options.UseSequenceMarker)
96          {
97              return Links.SearchOrDefault(Options.SequenceMarkerLink, sequence);
98          }
99          return sequence;
100      }
101
102      private LinkIndex GetSequenceElements(LinkIndex sequence)
103      {
104          if (Options.UseSequenceMarker)
105          {
106              var linkContents = new UInt64Link(Links.GetLink(sequence));
107              if (linkContents.Source == Options.SequenceMarkerLink)
108              {
109                  return linkContents.Target;
110              }
111              if (linkContents.Target == Options.SequenceMarkerLink)
112              {
113                  return linkContents.Source;
114              }
115          }
116      }

```

```

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>
↳ restrictions)
233 {
234     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,
↳ Options.SequenceMarkerLink, any));
250                 }
251                 else
252                 {
253                     return Links.Unsync.Each(handler, new Link<LinkIndex>(any, any,
↳ any));
254                 }
255             }
256             var sequence =
↳ Options.Walker.Walk(link).ToArray().ConvertToRestrictionsValues();
257             sequence[0] = link;
258             return handler(sequence);
259         }
260         else if (restrictions.Count == 2)
261         {
262             throw new NotImplementedException();
263         }
264         else if (restrictions.Count == 3)

```

```

265     {
266         return Links.Unsync.Each(handler, restrictions);
267     }
268     else
269     {
270         var sequence = restrictions.ExtractValues();
271         if (Options.UseIndex && !Options.Index.MightContain(sequence))
272         {
273             return Constants.Break;
274         }
275         return EachCore(handler, sequence);
276     }
277     });
278 }
279
280 private LinkIndex EachCore(Func<IList<LinkIndex>, LinkIndex> handler, IList<LinkIndex>
    ↪ values)
281 {
282     var matcher = new Matcher(this, values, new HashSet<LinkIndex>(), handler);
283     // TODO: Find out why matcher.HandleFullMatched executed twice for the same sequence
    ↪ Id.
284     Func<IList<LinkIndex>, LinkIndex> innerHandler = Options.UseSequenceMarker ?
    ↪ (Func<IList<LinkIndex>, LinkIndex>)matcher.HandleFullMatchedSequence :
    ↪ matcher.HandleFullMatched;
285     //if (sequence.Length >= 2)
286     if (StepRight(innerHandler, values[0], values[1]) != Constants.Continue)
287     {
288         return Constants.Break;
289     }
290     var last = values.Count - 2;
291     for (var i = 1; i < last; i++)
292     {
293         if (PartialStepRight(innerHandler, values[i], values[i + 1]) !=
    ↪ Constants.Continue)
294         {
295             return Constants.Break;
296         }
297     }
298     if (values.Count >= 3)
299     {
300         if (StepLeft(innerHandler, values[values.Count - 2], values[values.Count - 1])
    ↪ != Constants.Continue)
301         {
302             return Constants.Break;
303         }
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 &&
↪ !sequence.EqualTo(newSequence))
395     {
396         bestVariant = CompactCore(newSequence);
397     }
398     else
399     {
400         bestVariant = CreateCore(newSequence);
401     }
402     // TODO: Check all options only ones before loop execution
403     // Возможно нужно две версии Each, возвращающий фактические последовательности и с
↪ маркером,

```

```

404 // или возможно даже возвращать и тот и тот вариант. С другой стороны все варианты
405 ↪ можно получить имея только фактические последовательности.
406 foreach (var variant in Each(sequence))
407 {
408     if (variant != bestVariant)
409     {
410         UpdateOneCore(variant, bestVariant);
411     }
412     return bestVariant;
413 }
414
415 private void UpdateOneCore(LinkIndex sequence, LinkIndex newSequence)
416 {
417     if (Options.UseGarbageCollection)
418     {
419         var sequenceElements = GetSequenceElements(sequence);
420         var sequenceElementsContents = new UInt64Link(Links.GetLink(sequenceElements));
421         var sequenceLink = GetSequenceByElements(sequenceElements);
422         var newSequenceElements = GetSequenceElements(newSequence);
423         var newSequenceLink = GetSequenceByElements(newSequenceElements);
424         if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
425         {
426             if (sequenceLink != Constants.Null)
427             {
428                 Links.Unsync.MergeUsages(sequenceLink, newSequenceLink);
429             }
430             Links.Unsync.MergeUsages(sequenceElements, newSequenceElements);
431         }
432         ClearGarbage(sequenceElementsContents.Source);
433         ClearGarbage(sequenceElementsContents.Target);
434     }
435     else
436     {
437         if (Options.UseSequenceMarker)
438         {
439             var sequenceElements = GetSequenceElements(sequence);
440             var sequenceLink = GetSequenceByElements(sequenceElements);
441             var newSequenceElements = GetSequenceElements(newSequence);
442             var newSequenceLink = GetSequenceByElements(newSequenceElements);
443             if (Options.UseCascadeUpdate || CountUsages(sequence) == 0)
444             {
445                 if (sequenceLink != Constants.Null)
446                 {
447                     Links.Unsync.MergeUsages(sequenceLink, newSequenceLink);
448                 }
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(IList<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
552 #region Garbage Collection
553
554 /// <remarks>
555 /// TODO: Добавить дополнительный обработчик / событие CanBeDeleted которое можно
556 /// ↪ определить извне или в унаследованном классе
557 /// </remarks>
558 [MethodImpl(MethodImplOptions.AggressiveInlining)]

```

```

557 private bool IsGarbage(LinkIndex link) => link != Options.SequenceMarkerLink &&
    ↳ !Links.Unsync.IsPartialPoint(link) && Links.Count(link) == 0;
558
559 private void ClearGarbage(LinkIndex link)
560 {
561     if (IsGarbage(link))
562     {
563         var contents = new UInt64Link(Links.GetLink(link));
564         Links.Unsync.Delete(link);
565         ClearGarbage(contents.Source);
566         ClearGarbage(contents.Target);
567     }
568 }
569
570 #endregion
571
572 #region Walkers
573
574 public bool EachPart(Func<LinkIndex, bool> handler, LinkIndex sequence)
575 {
576     return _sync.ExecuteReadOperation(() =>
577     {
578         var links = Links.Unsync;
579         foreach (var part in Options.Walker.Walk(sequence))
580         {
581             if (!handler(part))
582             {
583                 return false;
584             }
585         }
586         return true;
587     });
588 }
589
590 public class Matcher : RightSequenceWalker<LinkIndex>
591 {
592     private readonly Sequences _sequences;
593     private readonly IList<LinkIndex> _patternSequence;
594     private readonly HashSet<LinkIndex> _linksInSequence;
595     private readonly HashSet<LinkIndex> _results;
596     private readonly Func<IList<LinkIndex>, LinkIndex> _stopableHandler;
597     private readonly HashSet<LinkIndex> _readAsElements;
598     private int _filterPosition;
599
600     public Matcher(Sequences sequences, IList<LinkIndex> patternSequence,
    ↳ HashSet<LinkIndex> results, Func<IList<LinkIndex>, LinkIndex> stopableHandler,
    ↳ HashSet<LinkIndex> readAsElements = null)
    : base(sequences.Links.Unsync, new DefaultStack<LinkIndex>())
601     {
602         _sequences = sequences;
603         _patternSequence = patternSequence;
604         _linksInSequence = new HashSet<LinkIndex>(patternSequence.Where(x => x !=
    ↳ Links.Constants.Any && x != ZeroOrMany));
605         _results = results;
606         _stopableHandler = stopableHandler;
607         _readAsElements = readAsElements;
608     }
609
610     protected override bool IsElement(LinkIndex link) => base.IsElement(link) ||
    ↳ (_readAsElements != null && _readAsElements.Contains(link)) ||
    ↳ _linksInSequence.Contains(link);
611
612     public bool FullMatch(LinkIndex sequenceToMatch)
613     {
614         _filterPosition = 0;
615         foreach (var part in Walk(sequenceToMatch))
616         {
617             if (!FullMatchCore(part))
618             {
619                 break;
620             }
621         }
622         return _filterPosition == _patternSequence.Count;
623     }
624
625     private bool FullMatchCore(LinkIndex element)
626     {
627         if (_filterPosition == _patternSequence.Count)
628         {
629             _filterPosition = -2; // Длиннее чем нужно
630

```

```

631         return false;
632     }
633     if (_patternSequence[_filterPosition] != Links.Constants.Any
634         && element != _patternSequence[_filterPosition])
635     {
636         _filterPosition = -1;
637         return false; // Начинается/Продолжается иначе
638     }
639     _filterPosition++;
640     return true;
641 }
642
643 public void AddFullMatchedToResults(IList<LinkIndex> restrictions)
644 {
645     var sequenceToMatch = restrictions[Links.Constants.IndexPart];
646     if (FullMatch(sequenceToMatch))
647     {
648         _results.Add(sequenceToMatch);
649     }
650 }
651
652 public LinkIndex HandleFullMatched(IList<LinkIndex> restrictions)
653 {
654     var sequenceToMatch = restrictions[Links.Constants.IndexPart];
655     if (FullMatch(sequenceToMatch) && _results.Add(sequenceToMatch))
656     {
657         return _stopableHandler(new LinkAddress<LinkIndex>(sequenceToMatch));
658     }
659     return Links.Constants.Continue;
660 }
661
662 public LinkIndex HandleFullMatchedSequence(IList<LinkIndex> restrictions)
663 {
664     var sequenceToMatch = restrictions[Links.Constants.IndexPart];
665     var sequence = _sequences.GetSequenceByElements(sequenceToMatch);
666     if (sequence != Links.Constants.Null && FullMatch(sequenceToMatch) &&
667         ↪ _results.Add(sequenceToMatch))
668     {
669         return _stopableHandler(new LinkAddress<LinkIndex>(sequence));
670     }
671     return Links.Constants.Continue;
672 }
673
674 /// <remarks>
675 /// TODO: Add support for LinksConstants.Any
676 /// </remarks>
677 public bool PartialMatch(LinkIndex sequenceToMatch)
678 {
679     _filterPosition = -1;
680     foreach (var part in Walk(sequenceToMatch))
681     {
682         if (!PartialMatchCore(part))
683         {
684             break;
685         }
686     }
687     return _filterPosition == _patternSequence.Count - 1;
688 }
689
690 private bool PartialMatchCore(LinkIndex element)
691 {
692     if (_filterPosition == (_patternSequence.Count - 1))
693     {
694         return false; // Нашлось
695     }
696     if (_filterPosition >= 0)
697     {
698         if (element == _patternSequence[_filterPosition + 1])
699         {
700             _filterPosition++;
701         }
702         else
703         {
704             _filterPosition = -1;
705         }
706     }
707     if (_filterPosition < 0)
708     {
709         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 }
757
758 #endregion
759 }
760 }

```

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

```

1  using System;
2  using LinkIndex = System.UInt64;
3  using System.Collections.Generic;
4  using Stack = System.Collections.Generic.Stack<ulong>;
5  using System.Linq;
6  using System.Text;
7  using Platform.Collections;
8  using Platform.Data.Exceptions;
9  using Platform.Data.Sequences;
10 using Platform.Data.Doublets.Sequences.Frequencies.Counters;
11 using Platform.Data.Doublets.Sequences.Walkers;
12 using Platform.Collections.Stacks;
13
14 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
15
16 namespace Platform.Data.Doublets.Sequences
17 {
18     partial class Sequences
19     {
20         #region Create All Variants (Not Practical)
21
22         /// <remarks>
23         /// Number of links that is needed to generate all variants for
24         /// sequence of length N corresponds to https://oeis.org/A014143/list sequence.
25         /// </remarks>

```

```

26 public ulong[] CreateAllVariants2(ulong[] sequence)
27 {
28     return _sync.ExecuteWriteOperation(() =>
29     {
30         if (sequence.IsNullOrEmpty())
31         {
32             return new ulong[0];
33         }
34         Links.EnsureEachLinkExists(sequence);
35         if (sequence.Length == 1)
36         {
37             return sequence;
38         }
39         return CreateAllVariants2Core(sequence, 0, sequence.Length - 1);
40     });
41 }
42
43 private ulong[] CreateAllVariants2Core(ulong[] sequence, long startAt, long stopAt)
44 {
45     #if DEBUG
46         if ((stopAt - startAt) < 0)
47         {
48             throw new ArgumentOutOfRangeException(nameof(startAt), "startAt должен быть
49                 ↳ меньше или равен stopAt");
50         }
51     #endif
52     if ((stopAt - startAt) == 0)
53     {
54         return new[] { sequence[startAt] };
55     }
56     if ((stopAt - startAt) == 1)
57     {
58         return new[] { Links.Unsync.CreateAndUpdate(sequence[startAt], sequence[stopAt])
59             ↳ };
60     }
61     var variants = new ulong[(ulong)Platform.Numbers.Math.Catalan(stopAt - startAt)];
62     var last = 0;
63     for (var splitter = startAt; splitter < stopAt; splitter++)
64     {
65         var left = CreateAllVariants2Core(sequence, startAt, splitter);
66         var right = CreateAllVariants2Core(sequence, splitter + 1, stopAt);
67         for (var i = 0; i < left.Length; i++)
68         {
69             for (var j = 0; j < right.Length; j++)
70             {
71                 var variant = Links.Unsync.CreateAndUpdate(left[i], right[j]);
72                 if (variant == Constants.Null)
73                 {
74                     throw new NotImplementedException("Creation cancellation is not
75                         ↳ implemented.");
76                 }
77                 variants[last++] = variant;
78             }
79         }
80     }
81     return variants;
82 }
83
84 public List<ulong> CreateAllVariants1(params ulong[] sequence)
85 {
86     return _sync.ExecuteWriteOperation(() =>
87     {
88         if (sequence.IsNullOrEmpty())
89         {
90             return new List<ulong>();
91         }
92         Links.Unsync.EnsureEachLinkExists(sequence);
93         if (sequence.Length == 1)
94         {
95             return new List<ulong> { sequence[0] };
96         }
97         var results = new
98             ↳ List<ulong>((int)Platform.Numbers.Math.Catalan(sequence.Length));
99         return CreateAllVariants1Core(sequence, results);
100     });
101 }
102
103 private List<ulong> CreateAllVariants1Core(ulong[] sequence, List<ulong> results)

```

```

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         return true;
148     }, sequence);
149     return visitedLinks;
150 }
151
152 private void Each1(Func<ulong, bool> handler, params ulong[] sequence)
153 {
154     if (sequence.Length == 2)
155     {
156         Links.Unsync.Each(sequence[0], sequence[1], handler);
157     }
158     else
159     {
160         var innerSequenceLength = sequence.Length - 1;
161         for (var li = 0; li < innerSequenceLength; li++)
162         {
163             var left = sequence[li];
164             var right = sequence[li + 1];
165             if (left == 0 && right == 0)
166             {
167                 continue;
168             }
169             var linkIndex = li;
170             ulong[] innerSequence = null;
171             Links.Unsync.Each(doublet =>
172             {
173                 if (innerSequence == null)
174                 {
175                     innerSequence = new ulong[innerSequenceLength];
176                     for (var isi = 0; isi < linkIndex; isi++)
177                     {

```



```

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 {
224     if (sequence.IsNullOrEmpty())
225     {
226         return;
227     }
228     Links.EnsureEachLinkIsAnyOrExists(sequence);
229     if (sequence.Length == 1)
230     {
231         var link = sequence[0];
232         if (link > 0)
233         {
234             handler(new LinkAddress<LinkIndex>(link));
235         }
236         else
237         {
238             Links.Each(Constants.Any, Constants.Any, handler);
239         }
240     }
241     else if (sequence.Length == 2)
242     {
243         //_links.Each(sequence[0], sequence[1], handler);
244         //  o_|      x_o ...
245         // x_|      |__|
246         Links.Each(sequence[1], Constants.Any, doublet =>
247         {
248             var match = Links.SearchOrDefault(sequence[0], doublet);
249             if (match != Constants.Null)
250             {
251                 handler(new LinkAddress<LinkIndex>(match));
252             }
253             return true;
254         });
255     }
256 }

```

```

253     });
254     // |_x      ... x_o
255     // |_o      |___|
256     Links.Unsync.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;
301     var firstSource = Links.Unsync.GetTarget(upStep);
302     while (firstSource != right && firstSource != upStep)
303     {
304         upStep = firstSource;
305         firstSource = Links.Unsync.GetSource(upStep);
306     }
307     if (firstSource == right)
308     {
309         handler(new LinkAddress<LinkIndex>(stepFrom));
310     }
311 }
312
313 // TODO: Test
314 private void PartialStepLeft(Action<IList<LinkIndex>> handler, ulong left, ulong right)
315 {
316     Links.Unsync.Each(right, Constants.Any, doublet =>
317     {
318         StepLeft(handler, left, doublet);
319         if (right != doublet)
320         {
321             PartialStepLeft(handler, left, doublet);
322         }
323         return true;
324     });
325 }
326
327 private void StepLeft(Action<IList<LinkIndex>> handler, ulong left, ulong right)
328 {
329     Links.Unsync.Each(Constants.Any, right, leftStep =>
330     {
331         TryStepLeftUp(handler, left, leftStep);
332     });
333 }

```

```

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(IList<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                         }
411                     }
412                 );
413             }
414             handler(result);
415         }
416     });
417 }

```

```

408         return false;
409     }
410     if (x != sequence[filterPosition])
411     {
412         filterPosition = -1;
413         return false; // Начинается иначе
414     }
415     filterPosition++;
416     return true;
417     });
418     if (filterPosition == sequence.Length)
419     {
420         results.Add(resultIndex);
421     }
422 }
423 if (sequence.Length >= 2)
424 {
425     StepRight(handler, sequence[0], sequence[1]);
426 }
427 var last = sequence.Length - 2;
428 for (var i = 1; i < last; i++)
429 {
430     PartialStepRight(handler, sequence[i], sequence[i + 1]);
431 }
432 if (sequence.Length >= 3)
433 {
434     StepLeft(handler, sequence[sequence.Length - 2],
435         ↪ sequence[sequence.Length - 1]);
436 }
437 }
438 return results;
439 });
440 }
441
442 public HashSet<ulong> GetAllMatchingSequences1(params ulong[] sequence)
443 {
444     return _sync.ExecuteReadOperation(() =>
445     {
446         var results = new HashSet<ulong>();
447         if (sequence.Length > 0)
448         {
449             Links.EnsureEachLinkExists(sequence);
450             var firstElement = sequence[0];
451             if (sequence.Length == 1)
452             {
453                 results.Add(firstElement);
454                 return results;
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         }
482         return results;
483     });
484 }

```

```

483 public const int MaxSequenceFormatSize = 200;
484
485 public string FormatSequence(LinkIndex sequenceLink, params LinkIndex[] knownElements)
486     => FormatSequence(sequenceLink, (sb, x) => sb.Append(x), true, knownElements);
487
488 public string FormatSequence(LinkIndex sequenceLink, Action<StringBuilder, LinkIndex>
489     elementToString, bool insertComma, params LinkIndex[] knownElements) =>
490     Links.SyncRoot.ExecuteReadOperation(() => FormatSequence(Links.Unsync, sequenceLink,
491         elementToString, insertComma, knownElements));
492
493 private string FormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
494     Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
495     LinkIndex[] knownElements)
496 {
497     var linksInSequence = new HashSet<ulong>(knownElements);
498     //var entered = new HashSet<ulong>();
499     var sb = new StringBuilder();
500     sb.Append('{');
501     if (links.Exists(sequenceLink))
502     {
503         StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
504             x => linksInSequence.Contains(x) || links.IsPartialPoint(x), element => //
505             entered.AddAndReturnVoid, x => { }, entered.DoNotContains
506             {
507                 if (insertComma && sb.Length > 1)
508                 {
509                     sb.Append(',');
510                 }
511                 //if (entered.Contains(element))
512                 //{
513                 //    sb.Append('{');
514                 //    elementToString(sb, element);
515                 //    sb.Append('}');
516                 //}
517                 //else
518                 elementToString(sb, element);
519                 if (sb.Length < MaxSequenceFormatSize)
520                 {
521                     return true;
522                 }
523                 sb.Append(insertComma ? ", ..." : "...");
524                 return false;
525             });
526     }
527     sb.Append('}');
528     return sb.ToString();
529 }
530
531 public string SafeFormatSequence(LinkIndex sequenceLink, params LinkIndex[]
532     knownElements) => SafeFormatSequence(sequenceLink, (sb, x) => sb.Append(x), true,
533     knownElements);
534
535 public string SafeFormatSequence(LinkIndex sequenceLink, Action<StringBuilder,
536     LinkIndex> elementToString, bool insertComma, params LinkIndex[] knownElements) =>
537     Links.SyncRoot.ExecuteReadOperation(() => SafeFormatSequence(Links.Unsync,
538         sequenceLink, elementToString, insertComma, knownElements));
539
540 private string SafeFormatSequence(ILinks<LinkIndex> links, LinkIndex sequenceLink,
541     Action<StringBuilder, LinkIndex> elementToString, bool insertComma, params
542     LinkIndex[] knownElements)
543 {
544     var linksInSequence = new HashSet<ulong>(knownElements);
545     var entered = new HashSet<ulong>();
546     var sb = new StringBuilder();
547     sb.Append('{');
548     if (links.Exists(sequenceLink))
549     {
550         StopableSequenceWalker.WalkRight(sequenceLink, links.GetSource, links.GetTarget,
551             x => linksInSequence.Contains(x) || links.IsFullPoint(x),
552             entered.AddAndReturnVoid, x => { }, entered.DoNotContains, element =>
553             {
554                 if (insertComma && sb.Length > 1)
555                 {
556                     sb.Append(',');
557                 }
558                 if (entered.Contains(element))

```

```

545         {
546             sb.Append('{');
547             elementToString(sb, element);
548             sb.Append('}');
549         }
550         else
551         {
552             elementToString(sb, element);
553         }
554         if (sb.Length < MaxSequenceFormatSize)
555         {
556             return true;
557         }
558         sb.Append(insertComma ? ", ..." : "...");
559         return false;
560     });
561 }
562 sb.Append('}');
563 return sb.ToString();
564 }
565
566 public List<ulong> GetAllPartiallyMatchingSequences0(params ulong[] sequence)
567 {
568     return _sync.ExecuteReadOperation(() =>
569     {
570         if (sequence.Length > 0)
571         {
572             Links.EnsureEachLinkExists(sequence);
573             var results = new HashSet<ulong>();
574             for (var i = 0; i < sequence.Length; i++)
575             {
576                 AllUsagesCore(sequence[i], results);
577             }
578             var filteredResults = new List<ulong>();
579             var linksInSequence = new HashSet<ulong>(sequence);
580             foreach (var result in results)
581             {
582                 var filterPosition = -1;
583                 StopableSequenceWalker.WalkRight(result, Links.Unsync.GetSource,
584                     ↪ Links.Unsync.GetTarget,
585                     x => linksInSequence.Contains(x) || Links.Unsync.GetTarget(x) == x,
586                     ↪ x =>
587                     {
588                         if (filterPosition == (sequence.Length - 1))
589                         {
590                             return false;
591                         }
592                         if (filterPosition >= 0)
593                         {
594                             if (x == sequence[filterPosition + 1])
595                             {
596                                 filterPosition++;
597                             }
598                             else
599                             {
600                                 return false;
601                             }
602                         }
603                         if (filterPosition < 0)
604                         {
605                             if (x == sequence[0])
606                             {
607                                 filterPosition = 0;
608                             }
609                         }
610                         return true;
611                     });
612                 if (filterPosition == (sequence.Length - 1))
613                 {
614                     filteredResults.Add(result);
615                 }
616             }
617             return filteredResults;
618         }
619         return new List<ulong>();
620     });
621 }
622
623 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 //    {
671 //        if (sequence.Length > 0)
672 //        {
673 //            _links.EnsureEachLinkIsAnyOrExists(sequence);
674 //
675 //            var firstResults = new HashSet<ulong>();
676 //            var lastResults = new HashSet<ulong>();
677 //
678 //            var first = sequence.First(x => x != LinksConstants.Any);
679 //            var last = sequence.Last(x => x != LinksConstants.Any);
680 //
681 //            AllUsagesCore(first, firstResults);
682 //            AllUsagesCore(last, lastResults);
683 //
684 //            firstResults.IntersectWith(lastResults);
685 //
686 //            //for (var i = 0; i < sequence.Length; i++)
687 //            //    AllUsagesCore(sequence[i], results);
688 //
689 //            var filteredResults = new HashSet<ulong>();
690 //            var matcher = new Matcher(this, sequence, filteredResults, null);
691 //            matcher.AddAllPartialMatchedToResults(firstResults);
692 //            return filteredResults;
693 //        }
694 //
695 //        return new HashSet<ulong>();
696 //    });
697 //}
698
699 public HashSet<ulong> GetAllPartiallyMatchingSequences3(params ulong[] sequence)
700 {

```

```

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);
748             collector1.Collect(Links.Unsync.GetLink(sequence[0]));
749             var next = new HashSet<ulong>();
750             for (var i = 1; i < sequence.Count; i++)
751             {
752                 var collector = new AllUsagesCollector1(Links.Unsync, next);
753                 collector.Collect(Links.Unsync.GetLink(sequence[i]));
754
755                 results.IntersectWith(next);
756                 next.Clear();
757             }
758             var filteredResults = new HashSet<ulong>();
759             var matcher = new Matcher(this, sequence, filteredResults, null,
760                 ↪ readAsElements);
761             matcher.AddAllPartialMatchedToResultsAndReadAsElements(results.OrderBy(x =>
762                 ↪ x)); // OrderBy is a Hack
763             return filteredResults;
764         }
765         return new HashSet<ulong>();
766     });
767 }
768
769 // Does not work
770 //public HashSet<ulong> GetAllPartiallyMatchingSequences5(HashSet<ulong> readAsElements,
771     ↪ params ulong[] sequence)
772 //{
773 //    var visited = new HashSet<ulong>();
774 //    var results = new HashSet<ulong>();
775 //    var matcher = new Matcher(this, sequence, visited, x => { results.Add(x); return
776     ↪ true; }, readAsElements);

```



```

772 //     var last = sequence.Length - 1;
773 //     for (var i = 0; i < last; i++)
774 //     {
775 //         PartialStepRight(matcher.PartialMatch, sequence[i], sequence[i + 1]);
776 //     }
777 //     return results;
778 // }
779
780 public List<ulong> GetAllPartiallyMatchingSequences(params ulong[] sequence)
781 {
782     return _sync.ExecuteReadOperation(() =>
783     {
784         if (sequence.Length > 0)
785         {
786             Links.EnsureEachLinkExists(sequence);
787             //var firstElement = sequence[0];
788             //if (sequence.Length == 1)
789             //{
790             //    //results.Add(firstElement);
791             //    return results;
792             //}
793             //if (sequence.Length == 2)
794             //{
795             //    //var doublet = _links.SearchCore(firstElement, sequence[1]);
796             //    //if (doublet != Doublets.Links.Null)
797             //    //    results.Add(doublet);
798             //    return results;
799             //}
800             //var lastElement = sequence[sequence.Length - 1];
801             //Func<ulong, bool> handler = x =>
802             //{
803             //    if (StartsWith(x, firstElement) && EndsWith(x, lastElement))
804             //        results.Add(x);
805             //    return true;
806             //};
807             //if (sequence.Length >= 2)
808             //    StepRight(handler, sequence[0], sequence[1]);
809             //var last = sequence.Length - 2;
810             //for (var i = 1; i < last; i++)
811             //    PartialStepRight(handler, sequence[i], sequence[i + 1]);
812             //if (sequence.Length >= 3)
813             //    StepLeft(handler, sequence[sequence.Length - 2],
814             //        sequence[sequence.Length - 1]);
815             //if (sequence.Length == 1)
816             //    throw new NotImplementedException(); // all sequences, containing
817             //        this element?
818             //if (sequence.Length == 2)
819             //{
820             //    var results = new List<ulong>();
821             //    PartialStepRight(results.Add, sequence[0], sequence[1]);
822             //    return results;
823             //}
824             //var matches = new List<List<ulong>>();
825             //var last = sequence.Length - 1;
826             //for (var i = 0; i < last; i++)
827             //{
828             //    var results = new List<ulong>();
829             //    //StepRight(results.Add, sequence[i], sequence[i + 1]);
830             //    PartialStepRight(results.Add, sequence[i], sequence[i + 1]);
831             //    if (results.Count > 0)
832             //        matches.Add(results);
833             //    else
834             //        return results;
835             //    if (matches.Count == 2)
836             //    {
837             //        var merged = new List<ulong>();
838             //        for (var j = 0; j < matches[0].Count; j++)
839             //            for (var k = 0; k < matches[1].Count; k++)
840             //                CloseInnerConnections(merged.Add, matches[0][j],
841             //                    matches[1][k]);
842             //        if (merged.Count > 0)
843             //            matches = new List<List<ulong>> { merged };
844             //        else
845             //            return new List<ulong>();
846             //    }
847             //}
848         }
849     });
850 }

```

```

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 /// <remarks>
877 /// TODO: Может потребоваться ограничение на уровень глубины рекурсии
878 /// </remarks>
879 public HashSet<ulong> AllUsages(ulong link)
880 {
881     return _sync.ExecuteReadOperation(() =>
882     {
883         var usages = new HashSet<ulong>();
884         AllUsagesCore(link, usages);
885         return usages;
886     });
887 }
888
889 // При сборе всех использований (последовательностей) можно сохранять обратный путь к
890 // той связи с которой начинался поиск (STTTSSSTT),
891 // причём достаточно одного бита для хранения перехода влево или вправо
892 private void AllUsagesCore(ulong link, HashSet<ulong> usages)
893 {
894     bool handler(ulong doublet)
895     {
896         if (usages.Add(doublet))
897         {
898             AllUsagesCore(doublet, usages);
899         }
900         return true;
901     }
902     Links.Unsync.Each(link, Constants.Any, handler);
903     Links.Unsync.Each(Constants.Any, link, handler);
904 }
905
906 public HashSet<ulong> AllBottomUsages(ulong link)
907 {
908     return _sync.ExecuteReadOperation(() =>
909     {
910         var visits = new HashSet<ulong>();
911         var usages = new HashSet<ulong>();
912         AllBottomUsagesCore(link, visits, usages);
913         return usages;
914     });
915 }
916
917 private void AllBottomUsagesCore(ulong link, HashSet<ulong> visits, HashSet<ulong>
918     //    usages)
919 {
920     bool handler(ulong doublet)
921     {

```

```

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,
940             ↪ Options.MarkedSequenceMatcher, symbol);
941         return counter.Count();
942     }
943     else
944     {
945         var counter = new TotalSequenceSymbolFrequencyOneOffCounter<ulong>(Links,
946             ↪ symbol);
947         return counter.Count();
948     }
949 }
950
951 private bool AllUsagesCore1(ulong link, HashSet<ulong> usages, Func<IList<LinkIndex>,
952     ↪ LinkIndex> outerHandler)
953 {
954     bool handler(ulong doublet)
955     {
956         if (usages.Add(doublet))
957         {
958             if (outerHandler(new LinkAddress<LinkIndex>(doublet)) != Constants.Continue)
959             {
960                 return false;
961             }
962             if (!AllUsagesCore1(doublet, usages, outerHandler))
963             {
964                 return false;
965             }
966         }
967         return true;
968     }
969     return Links.Unsync.Each(link, Constants.Any, handler)
970         && Links.Unsync.Each(Constants.Any, link, handler);
971 }
972
973 public void CalculateAllUsages(ulong[] totals)
974 {
975     var calculator = new AllUsagesCalculator(Links, totals);
976     calculator.Calculate();
977 }
978
979 public void CalculateAllUsages2(ulong[] totals)
980 {
981     var calculator = new AllUsagesCalculator2(Links, totals);
982     calculator.Calculate();
983 }
984
985 private class AllUsagesCalculator
986 {
987     private readonly SynchronizedLinks<ulong> _links;
988     private readonly ulong[] _totals;
989
990     public AllUsagesCalculator(SynchronizedLinks<ulong> links, ulong[] totals)
991     {
992         _links = links;
993         _totals = totals;
994     }
995 }

```

```

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     private class AllUsagesCalculator2
1019     {
1020         private readonly SynchronizedLinks<ulong> _links;
1021         private readonly ulong[] _totals;
1022
1023         public AllUsagesCalculator2(SynchronizedLinks<ulong> links, ulong[] totals)
1024         {
1025             _links = links;
1026             _totals = totals;
1027         }
1028
1029         public void Calculate() => _links.Each(_links.Constants.Any, _links.Constants.Any,
1030             ↪ CalculateCore);
1031
1032         private bool IsElement(ulong link)
1033         {
1034             // _linksInSequence.Contains(link) ||
1035             return _links.Unsync.GetTarget(link) == link || _links.Unsync.GetSource(link) ==
1036                 ↪ link;
1037         }
1038
1039         private bool CalculateCore(ulong link)
1040         {
1041             // TODO: Проработать защиту от заикливания
1042             // Основано на SequenceWalker.WalkLeft
1043             Func<ulong, ulong> getSource = _links.Unsync.GetSource;
1044             Func<ulong, ulong> getTarget = _links.Unsync.GetTarget;
1045             Func<ulong, bool> isElement = IsElement;
1046             void visitLeaf(ulong parent)
1047             {
1048                 if (link != parent)
1049                 {
1050                     _totals[parent]++;
1051                 }
1052             }
1053             void visitNode(ulong parent)
1054             {
1055                 if (link != parent)
1056                 {
1057                     _totals[parent]++;
1058                 }
1059             }
1060             var stack = new Stack();
1061             var element = link;
1062             if (isElement(element))
1063             {
1064                 visitLeaf(element);
1065             }
1066             else
1067             {
1068                 while (true)
1069                 {
1070                     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             // 06пабортка 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 }
1096 _totals[link]++;
1097 return true;
1098 }
1099 }
1100
1101 private class AllUsagesCollector
1102 {
1103     private readonly ILinks<ulong> _links;
1104     private readonly HashSet<ulong> _usages;
1105
1106     public AllUsagesCollector(ILinks<ulong> links, HashSet<ulong> usages)
1107     {
1108         _links = links;
1109         _usages = usages;
1110     }
1111
1112     public bool Collect(ulong link)
1113     {
1114         if (_usages.Add(link))
1115         {
1116             _links.Each(link, _links.Constants.Any, Collect);
1117             _links.Each(_links.Constants.Any, link, Collect);
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(ICollection<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 private HashSet<ulong> GetAllPartiallyMatchingSequencesCore(ulong[] sequence,
1226     ↳ 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,
1243             ↳ secondLinkUsage);
1244             StepRight(filler.AddFirstAndReturnConstant, previousMatching,
1245             ↳ secondLinkUsage);
1246             TryStepRightUp(filler.AddFirstAndReturnConstant, secondLinkUsage,
1247             ↳ previousMatching);
1248             //PartialStepRight(matchings.AddAndReturnVoid, secondLinkUsage,
1249             ↳ sequence[startAt]); // почему-то эта ошибочная запись приводит к
1250             ↳ желаемым результатам.
1251             PartialStepRight(filler.AddFirstAndReturnConstant, previousMatching,
1252             ↳ secondLinkUsage);
1253         }
1254     }
1255     if (matchings.Count == 0)
1256     {
1257         return matchings;
1258     }
1259     return GetAllPartiallyMatchingSequencesCore(sequence, matchings, startAt + 1); // ??
1260 }
1261
1262 private static void EnsureEachLinkIsAnyOrZeroOrManyOrExists(SynchronizedLinks<ulong>
1263     ↳ links, params ulong[] sequence)
1264 {
1265     if (sequence == null)
1266     {
1267         return;
1268     }
1269     for (var i = 0; i < sequence.Length; i++)
1270     {
1271         if (sequence[i] != links.Constants.Any && sequence[i] != ZeroOrMany &&
1272             ↳ !links.Exists(sequence[i]))
1273         {
1274             throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
1275             ↳ $"patternSequence[{i}]");
1276         }
1277     }
1278 }
1279
1280 // Pattern Matching -> Key To Triggers
1281 public HashSet<ulong> MatchPattern(params ulong[] patternSequence)
1282 {
1283     return _sync.ExecuteReadOperation(() =>
1284     {
1285         patternSequence = Simplify(patternSequence);
1286         if (patternSequence.Length > 0)
1287         {
1288             EnsureEachLinkIsAnyOrZeroOrManyOrExists(Links, patternSequence);
1289             var uniqueSequenceElements = new HashSet<ulong>();
1290             for (var i = 0; i < patternSequence.Length; i++)
1291             {
1292                 if (patternSequence[i] != Constants.Any && patternSequence[i] !=
1293                     ↳ ZeroOrMany)
1294                 {
1295                     uniqueSequenceElements.Add(patternSequence[i]);
1296                 }
1297             }
1298             var results = new HashSet<ulong>();
1299             foreach (var uniqueSequenceElement in uniqueSequenceElements)
1300             {

```

```

1291         AllUsagesCore(uniqueSequenceElement, results);
1292     }
1293     var filteredResults = new HashSet<ulong>();
1294     var matcher = new PatternMatcher(this, patternSequence, filteredResults);
1295     matcher.AddAllPatternMatchedToResults(results);
1296     return filteredResults;
1297 }
1298 return new HashSet<ulong>();
1299 });
1300 }
1301
1302 // Найти все возможные связи между указанным списком связей.
1303 // Находит связи между всеми указанными связями в любом порядке.
1304 // TODO: решить что делать с повторами (когда одни и те же элементы встречаются
1305 //        несколько раз в последовательности)
1306 public HashSet<ulong> GetAllConnections(params ulong[] linksToConnect)
1307 {
1308     return _sync.ExecuteReadOperation(() =>
1309     {
1310         var results = new HashSet<ulong>();
1311         if (linksToConnect.Length > 0)
1312         {
1313             Links.EnsureEachLinkExists(linksToConnect);
1314             AllUsagesCore(linksToConnect[0], results);
1315             for (var i = 1; i < linksToConnect.Length; i++)
1316             {
1317                 var next = new HashSet<ulong>();
1318                 AllUsagesCore(linksToConnect[i], next);
1319                 results.IntersectWith(next);
1320             }
1321             return results;
1322         }
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             {
1339                 var collector = new AllUsagesCollector(Links.Unsync, next);
1340                 collector.Collect(linksToConnect[i]);
1341                 results.IntersectWith(next);
1342                 next.Clear();
1343             }
1344             return results;
1345         }
1346     });
1347 }
1348
1349 public HashSet<ulong> GetAllConnections2(params ulong[] linksToConnect)
1350 {
1351     return _sync.ExecuteReadOperation(() =>
1352     {
1353         var results = new HashSet<ulong>();
1354         if (linksToConnect.Length > 0)
1355         {
1356             Links.EnsureEachLinkExists(linksToConnect);
1357             var collector1 = new AllUsagesCollector(Links, results);
1358             collector1.Collect(linksToConnect[0]);
1359             //AllUsagesCore(linksToConnect[0], results);
1360             for (var i = 1; i < linksToConnect.Length; i++)
1361             {
1362                 var next = new HashSet<ulong>();
1363                 var collector = new AllUsagesIntersectingCollector(Links, results, next);
1364                 collector.Collect(linksToConnect[i]);
1365                 //AllUsagesCore(linksToConnect[i], next);
1366                 //results.IntersectWith(next);
1367                 results = next;
1368             }
1369         }
1370     });
1371 }

```



```

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             return results.GetSetUInt64Indices();
1393         }
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     // Строим новую последовательность
1420     zeroOrManyStepped = false;
1421     var newSequence = new ulong[newLength];
1422     long j = 0;
1423     for (var i = 0; i < sequence.Length; i++)
1424     {
1425         //var current = zeroOrManyStepped;
1426         //zeroOrManyStepped = patternSequence[i] == zeroOrMany;
1427         //if (current && zeroOrManyStepped)
1428         //    continue;
1429         //var newZeroOrManyStepped = patternSequence[i] == zeroOrMany;
1430         //if (zeroOrManyStepped && newZeroOrManyStepped)
1431         //    continue;
1432         //zeroOrManyStepped = newZeroOrManyStepped;
1433         if (sequence[i] == ZeroOrMany)
1434         {
1435             if (zeroOrManyStepped)
1436             {
1437                 continue;
1438             }
1439             zeroOrManyStepped = true;
1440         }
1441         else
1442         {
1443             //if (zeroOrManyStepped) Is it efficient?
1444             zeroOrManyStepped = false;
1445             newSequence[j++] = sequence[i];
1446         }
1447     }
1448 }

```

```

1445     }
1446     return newSequence;
1447 }
1448
1449 public static void TestSimplify()
1450 {
1451     var sequence = new ulong[] { ZeroOrMany, ZeroOrMany, 2, 3, 4, ZeroOrMany,
        ↪ ZeroOrMany, ZeroOrMany, 4, ZeroOrMany, ZeroOrMany, ZeroOrMany };
1452     var simplifiedSequence = Simplify(sequence);
1453 }
1454
1455 public List<ulong> GetSimilarSequences() => new List<ulong>();
1456
1457 public void Prediction()
1458 {
1459     //_links
1460     //_sequences
1461 }
1462
1463 #region From Triplets
1464
1465 //public static void DeleteSequence(Link sequence)
1466 //{
1467 //}
1468
1469 public List<ulong> CollectMatchingSequences(ulong[] links)
1470 {
1471     if (links.Length == 1)
1472     {
1473         throw new Exception("Подпоследовательности с одним элементом не
        ↪ поддерживаются.");
1474     }
1475     var leftBound = 0;
1476     var rightBound = links.Length - 1;
1477     var left = links[leftBound++];
1478     var right = links[rightBound--];
1479     var results = new List<ulong>();
1480     CollectMatchingSequences(left, leftBound, links, right, rightBound, ref results);
1481     return results;
1482 }
1483
1484 private void CollectMatchingSequences(ulong leftLink, int leftBound, ulong[]
        ↪ middleLinks, ulong rightLink, int rightBound, ref List<ulong> results)
1485 {
1486     var leftLinkTotalReferers = Links.Unsync.Count(leftLink);
1487     var rightLinkTotalReferers = Links.Unsync.Count(rightLink);
1488     if (leftLinkTotalReferers <= rightLinkTotalReferers)
1489     {
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,
        ↪ rightLink, rightBound, ref results);
1500                 }
1501             }
1502         }
1503         else
1504         {
1505             for (var i = elements.Length - 1; i >= 0; i--)
1506             {
1507                 var element = elements[i];
1508                 if (element != 0)
1509                 {
1510                     results.Add(element);
1511                 }
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         }
1592         return true;
1593     });
1594     return added > 0;
1595 }

```

```

1595 public ulong[] GetLeftElements(ulong startLink, ulong leftLink)
1596 {
1597     var result = new ulong[5];
1598     TryStepLeft(startLink, leftLink, result, 0);
1599     Links.Each(startLink, Constants.Any, couple =>
1600     {
1601         if (couple != startLink)
1602         {
1603             if (TryStepLeft(couple, leftLink, result, 2))
1604             {
1605                 return false;
1606             }
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         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 private readonly List<PatternBlock> _pattern;
1675 private int _patternPosition;
1676 private long _sequencePosition;
1677
1678 #endregion
1679
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             else if (_patternSequence[i] == ZeroOrMany)
1726             {
1727                 patternBlock.Type = PatternBlockType.Gap;
1728                 patternBlock.Start = 0;
1729                 patternBlock.Stop = long.MaxValue;
1730             }
1731             else
1732             {
1733                 patternBlock.Type = PatternBlockType.Elements;
1734                 patternBlock.Start = i;
1735                 patternBlock.Stop = i;
1736             }
1737         }
1738         else if (patternBlock.Type == PatternBlockType.Elements)
1739         {
1740             if (_patternSequence[i] == _sequences.Constants.Any)
1741             {
1742                 pattern.Add(patternBlock);
1743                 patternBlock = new PatternBlock
1744                 {
1745                     Type = PatternBlockType.Gap,
1746                     Start = 1,
1747                     Stop = 1
1748                 };
1749             }
1750             else if (_patternSequence[i] == ZeroOrMany)
1751             {
1752                 pattern.Add(patternBlock);
1753                 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 //private void GetNextPatternElement(out LinkIndex element, out long mininumGap, out
1829 ↪ long maximumGap)
1830 //{
1831 //    mininumGap = 0;
1832 //    maximumGap = 0;
1833 //    element = 0;
1834 //    for (; _patternPosition < _patternSequence.Length; _patternPosition++)
1835 //    {
1836 //        if (_patternSequence[_patternPosition] == Doublets.Links.Null)
1837 //            mininumGap++;
1838 //        else if (_patternSequence[_patternPosition] == ZeroOrMany)
1839 //            maximumGap = long.MaxValue;
1840 //        else
1841 //            break;
1842 //    }
1843 //    if (maximumGap < mininumGap)
1844 //        maximumGap = mininumGap;
1845 //}
1846
1847 private bool PatternMatchCore(LinkIndex element)
1848 {
1849     if (_patternPosition >= _pattern.Count)
1850     {
1851         _patternPosition = -2;
1852         return false;
1853     }
1854     var currentPatternBlock = _pattern[_patternPosition];
1855     if (currentPatternBlock.Type == PatternBlockType.Gap)
1856     {
1857         //var currentMatchingBlockLength = (_sequencePosition -
1858         ↪ _lastMatchedBlockPosition);
1859         if (_sequencePosition < currentPatternBlock.Start)
1860         {
1861             _sequencePosition++;
1862             return true; // Двигаемся дальше
1863         }
1864         // Это последний блок
1865         if (_pattern.Count == _patternPosition + 1)
1866         {
1867             _patternPosition++;
1868             _sequencePosition = 0;
1869             return false; // Полное соответствие
1870         }
1871         else
1872         {
1873             if (_sequencePosition > currentPatternBlock.Stop)
1874             {
1875                 return false; // Соответствие невозможно
1876             }
1877             var nextPatternBlock = _pattern[_patternPosition + 1];
1878             if (_patternSequence[nextPatternBlock.Start] == element)
1879             {
1880                 if (nextPatternBlock.Start < nextPatternBlock.Stop)
1881                 {
1882                     _patternPosition++;
1883                     _sequencePosition = 1;
1884                 }
1885                 else
1886                 {
1887                     _patternPosition += 2;
1888                     _sequencePosition = 0;
1889                 }
1890             }
1891         }
1892     }
1893     else // currentPatternBlock.Type == PatternBlockType.Elements
1894     {
1895         var patternElementPosition = currentPatternBlock.Start + _sequencePosition;
1896         if (_patternSequence[patternElementPosition] != element)
1897         {
1898             return false; // Соответствие невозможно
1899         }
1900         if (patternElementPosition == currentPatternBlock.Stop)
1901         {
1902             _patternPosition++;
1903             _sequencePosition = 0;
1904         }
1905         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 #endregion
1958
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
26             ↪ LinkAddress<TLink>(sequence));
27         return list;
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
17     ↪ ILinks<TLink> must contain GetConstants function.
18     {
19         private static readonly EqualityComparer<TLink> _equalityComparer =
20         ↪ EqualityComparer<TLink>.Default;
21
22         public TLink SequenceMarkerLink { get; set; }
23         public bool UseCascadeUpdate { get; set; }
24         public bool UseCascadeDelete { get; set; }
25         public bool UseIndex { get; set; } // TODO: Update Index on sequence update/delete.
26         public bool UseSequenceMarker { get; set; }
27         public bool UseCompression { get; set; }
28         public bool UseGarbageCollection { get; set; }
29         public bool EnforceSingleSequenceVersionOnWriteBasedOnExisting { get; set; }
30         public bool EnforceSingleSequenceVersionOnWriteBasedOnNew { get; set; }
31
32         public MarkedSequenceCriterionMatcher<TLink> MarkedSequenceMatcher { get; set; }
33         public IConverter<IList<TLink>, TLink> LinksToSequenceConverter { get; set; }
34         public ISequenceIndex<TLink> Index { get; set; }
35         public ISequenceWalker<TLink> Walker { get; set; }
36         public bool ReadFullSequence { get; set; }
37
38         // TODO: Реализовать компактификацию при чтении
39         //public bool EnforceSingleSequenceVersionOnRead { get; set; }
40         //public bool UseRequestMarker { get; set; }
41         //public bool StoreRequestResults { get; set; }
42
43         public void InitOptions(ISynchronizedLinks<TLink> links)
44         {
45             if (UseSequenceMarker)
46             {
47                 if (_equalityComparer.Equals(SequenceMarkerLink, links.Constants.Null))
48                 {
49                     SequenceMarkerLink = links.CreatePoint();
50                 }
51                 else
52                 {
53                     if (!links.Exists(SequenceMarkerLink))
54                     {
55                         var link = links.CreatePoint();
56                         if (!_equalityComparer.Equals(link, SequenceMarkerLink))
57                         {
58                             throw new InvalidOperationException("Cannot recreate sequence marker
59                             ↪ link.");
60                         }
61                     }
62                 }
63             }
64             if (MarkedSequenceMatcher == null)
65             {
66                 MarkedSequenceMatcher = new MarkedSequenceCriterionMatcher<TLink>(links,
67                     ↪ SequenceMarkerLink);
68             }
69         }
70     }
71 }

```

```

63     }
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 }
115 }
116 }

```

./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 }

```

```

33     }
34 }
35 }
36 }
37 }

```

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

```

1  using System;
2  using System.Collections.Generic;
3  using System.Runtime.CompilerServices;
4
5  #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7  //#define USEARRAYPOOL
8  #if USEARRAYPOOL
9  using Platform.Collections;
10 #endif
11
12 namespace Platform.Data.Doublets.Sequences.Walkers
13 {
14     public class LeveledSequenceWalker<TLink> : LinksOperatorBase<TLink>, ISequenceWalker<TLink>
15     {
16         private static readonly EqualityComparer<TLink> _equalityComparer =
17             ↳ EqualityComparer<TLink>.Default;
18
19         private readonly Func<TLink, bool> _isElement;
20
21         public LeveledSequenceWalker(ILinks<TLink> links, Func<TLink, bool> isElement) :
22             ↳ base(links) => _isElement = isElement;
23
24         public LeveledSequenceWalker(ILinks<TLink> links) : base(links) => _isElement =
25             ↳ Links.IsPartialPoint;
26
27         public IEnumerable<TLink> Walk(TLink sequence) => ToArray(sequence);
28
29         public TLink[] ToArray(TLink sequence)
30         {
31             var length = 1;
32             var array = new TLink[length];
33             array[0] = sequence;
34             if (_isElement(sequence))
35             {
36                 return array;
37             }
38             bool hasElements;
39             do
40             {
41                 length *= 2;
42 #if USEARRAYPOOL
43                 var nextArray = ArrayPool.Allocate<ulong>(length);
44 #else
45                 var nextArray = new TLink[length];
46 #endif
47                 hasElements = false;
48                 for (var i = 0; i < array.Length; i++)
49                 {
50                     var candidate = array[i];
51                     if (_equalityComparer.Equals(array[i], default))
52                     {
53                         continue;
54                     }
55                     var doubletOffset = i * 2;
56                     if (_isElement(candidate))
57                     {
58                         nextArray[doubletOffset] = candidate;
59                     }
60                     else
61                     {
62                         var link = Links.GetLink(candidate);
63                         var linkSource = Links.GetSource(link);
64                         var linkTarget = Links.GetTarget(link);
65                         nextArray[doubletOffset] = linkSource;
66                         nextArray[doubletOffset + 1] = linkTarget;
67                         if (!hasElements)
68                         {
69                             hasElements = !(_isElement(linkSource) && _isElement(linkTarget));
70                         }
71                     }
72                 }
73             } while (length < array.Length);
74             return array;
75         }
76     }
77 #if USEARRAYPOOL
78 }
79 #endif

```

```

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 }

```

```

22     [MethodImpl(MethodImplOptions.AggressiveInlining)]
23     protected override IEnumerable<TLink> WalkContents(TLink element)
24     {
25         var parts = Links.GetLink(element);
26         for (var i = Links.Constants.IndexPart + 1; i < parts.Count; i++)
27         {
28             var part = parts[i];
29             if (IsElement(part))
30             {
31                 yield return part;
32             }
33         }
34     }
35 }
36 }

```

./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                 yield return element;
35             }
36             else
37             {
38                 while (true)
39                 {
40                     if (IsElement(element))
41                     {
42                         if (_stack.IsEmpty)
43                         {
44                             break;
45                         }
46                         element = _stack.Pop();
47                         foreach (var output in WalkContents(element))
48                         {
49                             yield return output;
50                         }
51                         element = GetNextElementAfterPop(element);
52                     }
53                     else
54                     {
55                         _stack.Push(element);
56                         element = GetNextElementAfterPush(element);
57                     }
58                 }
59             }
60         }
61     }
62 }

```

[MethodImpl(MethodImplOptions.AggressiveInlining)]

```

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                 _links.Delete(top);
39             }
40             return element;
41         }
42
43         public void Push(TLink element) => _links.Update(_stack, GetStackMarker(),
44             ↪ _links.GetOrCreate(GetTop(), element));
45     }
46 }

```

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

```

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

```

./Platform.Data.Doublets/SynchronizedLinks.cs

```
1 using System;
2 using System.Collections.Generic;
3 using Platform.Data.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,
39         //↳ IList<T> substitution, Func<IList<T>, IList<T>, T> substitutedHandler)
40         //{
41         //    if (restriction != null && substitution != null &&
42         //        ↳ !substitution.EqualTo(restriction))
43         //        return SyncRoot.ExecuteWriteOperation(restriction, matchedHandler,
44         //        ↳ substitution, substitutedHandler, Unsync.Trigger);
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 =
    ↳ Default<LinksConstants<ulong>>.Instance;
19
20 private const int Length = 3;
21
22 public readonly ulong Index;
23 public readonly ulong Source;
24 public readonly ulong Target;
25
26 public static readonly UInt64Link Null = new UInt64Link();
27
28 public UInt64Link(params ulong[] values)
29 {
30     Index = values.Length > _constants.IndexPart ? values[_constants.IndexPart] :
    ↳ _constants.Null;
31     Source = values.Length > _constants.SourcePart ? values[_constants.SourcePart] :
    ↳ _constants.Null;
32     Target = values.Length > _constants.TargetPart ? values[_constants.TargetPart] :
    ↳ _constants.Null;
33 }
34
35 public UInt64Link(IList<ulong> values)
36 {
37     Index = values.Count > _constants.IndexPart ? values[_constants.IndexPart] :
    ↳ _constants.Null;
38     Source = values.Count > _constants.SourcePart ? values[_constants.SourcePart] :
    ↳ _constants.Null;
39     Target = values.Count > _constants.TargetPart ? values[_constants.TargetPart] :
    ↳ _constants.Null;
40 }
41
42 public UInt64Link(ulong index, ulong source, ulong target)
43 {
44     Index = index;
45     Source = source;
46     Target = target;
47 }
48
49 public UInt64Link(ulong source, ulong target)
50 : this(_constants.Null, source, target)
51 {
52     Source = source;
53     Target = target;
54 }
55
56 public static UInt64Link Create(ulong source, ulong target) => new UInt64Link(source,
    ↳ target);
57
58 public override int GetHashCode() => (Index, Source, Target).GetHashCode();
59
60 public bool IsNull() => Index == _constants.Null
    && Source == _constants.Null
    && Target == _constants.Null;
61
62
63
64 public override bool Equals(object other) => other is UInt64Link &&
    ↳ Equals((UInt64Link)other);
65
66 public bool Equals(UInt64Link other) => Index == other.Index
    && Source == other.Source
    && Target == other.Target;
67
68
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),
87             ↳ nameof(index));
88         if (index == _constants.IndexPart)
89         {
90             return Index;
91         }
92         if (index == _constants.SourcePart)
93         {
94             return Source;
95         }
96         if (index == _constants.TargetPart)
97         {
98             return Target;
99         }
100         throw new NotSupportedException(); // Impossible path due to
101         ↳ Ensure.ArgumentInRange
102     }
103     set => throw new NotSupportedException();
104 }
105
106 public int Count => Length;
107
108 public bool IsReadOnly => true;
109
110 IEnumerator IEnumerable.GetEnumerator() => GetEnumerator();
111
112 public IEnumerator<ulong> GetEnumerator()
113 {
114     yield return Index;
115     yield return Source;
116     yield return Target;
117 }
118
119 public void Add(ulong item) => throw new NotSupportedException();
120
121 public void Clear() => throw new NotSupportedException();
122
123 public bool Contains(ulong item) => IndexOf(item) >= 0;
124
125 public void CopyTo(ulong[] array, int arrayIndex)
126 {
127     Ensure.OnDebug.ArgumentNotNull(array, nameof(array));
128     Ensure.OnDebug.ArgumentInRange(arrayIndex, new Range<int>(0, array.Length - 1),
129         ↳ nameof(arrayIndex));
130     if (arrayIndex + Length > array.Length)
131     {
132         throw new ArgumentException();
133     }
134     array[arrayIndex++] = Index;
135     array[arrayIndex++] = Source;
136     array[arrayIndex] = Target;
137 }
138
139 public bool Remove(ulong item) => Throw.A.NotSupportedExceptionAndReturn<bool>();
140
141 public int IndexOf(ulong item)
142 {
143     if (Index == item)
144     {
145         return _constants.IndexPart;
146     }
147     if (Source == item)
148     {
149         return _constants.SourcePart;
150     }
151     if (Target == item)
152     {
153         return _constants.TargetPart;
154     }
155     return -1;
156 }
157
158 public void Insert(int index, ulong item) => throw new NotSupportedException();
159
160 public void RemoveAt(int index) => throw new NotSupportedException();
161
162 #endregion
163 }

```

./Platform.Data.Doublets/UInt64LinkExtensions.cs

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

./Platform.Data.Doublets/UInt64LinksExtensions.cs

```
1  using System;
2  using System.Text;
3  using System.Collections.Generic;
4  using Platform.Singletons;
5  using Platform.Data.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         public static void EnsureEachLinkIsAnyOrExists(this ILinks<ulong> links, IList<ulong>
35             ↳ sequence)
36         {
37             if (sequence == null)
38             {
39                 return;
40             }
41             for (var i = 0; i < sequence.Count; i++)
42             {
43                 if (sequence[i] != Constants.Any && !links.Exists(sequence[i]))
44                 {
45                     throw new ArgumentLinkDoesNotExistsException<ulong>(sequence[i],
46                         ↳ $"sequence[{i}]");
47                 }
48             }
49
50         public static bool AnyLinkIsAny(this ILinks<ulong> links, params ulong[] sequence)
51         {
52             if (sequence == null)
53             {
54                 return false;
55             }
56             var constants = links.Constants;
57             for (var i = 0; i < sequence.Length; i++)
58             {
59                 if (sequence[i] == constants.Any)
60                 {
61                     return true;
62                 }
63             }
64         }
65     }
66 }
```

```

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                 else
116                 {
117                     var source = new UInt64Link(links.GetLink(link.Source));
118                     if (isElement(source))
119                     {
120                         appendElement(sb, source);
121                     }
122                     else
123                     {
124                         links.AppendStructure(sb, visited, source.Index, isElement,
125                             ↪ appendElement, renderIndex);
126                     }
127                 }
128                 sb.Append(' ');
129                 if (link.Target == link.Index)
130                 {
131                     sb.Append(link.Index);
132                 }
133                 else
134                 {
135                     var target = new UInt64Link(links.GetLink(link.Target));
136                     if (isElement(target))
137                     {
138                         appendElement(sb, target);
139                     }
140                 }
141             }
142         }
143     }

```

```

130         else
131         {
132             links.AppendStructure(sb, visited, target.Index, isElement,
133                                     ↪ appendElement, renderIndex);
134         }
135         sb.Append(' ');
136     }
137     else
138     {
139         if (renderDebug)
140         {
141             sb.Append('*');
142         }
143         sb.Append(linkIndex);
144     }
145 }
146 else
147 {
148     if (renderDebug)
149     {
150         sb.Append('~');
151     }
152     sb.Append(linkIndex);
153 }
154 }
155 }
156 }

```

./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<ulong> //-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
95             ↪ transactionId, UInt64Link before, UInt64Link after)
96         {
97             TransactionId = transactionId;
98             Before = before;
99             After = after;
100             Timestamp = uniqueTimestampFactory.Create();
101         }
102
103         public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong
104             ↪ transactionId, UInt64Link before)
105             : this(uniqueTimestampFactory, transactionId, before, default)
106         {
107         }
108
109         public Transition(UniqueTimestampFactory uniqueTimestampFactory, ulong transactionId
110             : this(uniqueTimestampFactory, transactionId, default, default)
111         {
112         }
113
114         public override string ToString() => $"{Timestamp} {TransactionId}: {Before} =>
115             ↪ {After}";
116     }
117
118     /// <remarks>
119     /// Другие варианты реализации транзакций (атомарности):
120     /// 1. Разделение хранения значения связи ((Source Target) или (Source Linker
121     ///     ↪ Target)) и индексов.
122     /// 2. Хранение трансформаций/операций в отдельном хранилище Links, но дополнительно
123     ///     ↪ потребуется решить вопрос
124     ///     со ссылками на внешние идентификаторы, или как-то иначе решить вопрос с
125     ///     ↪ пересечениями идентификаторов.
126     ///
127     /// Где хранить промежуточный список транзакций?

```

```

122 ///
123 /// В оперативной памяти:
124 /// Минусы:
125 /// 1. Может усложнить систему, если она будет функционировать самостоятельно,
126 /// так как нужно отдельно выделять память под список трансформаций.
127 /// 2. Выделенной оперативной памяти может не хватить, в том случае,
128 /// если транзакция использует слишком много трансформаций.
129 /// -> Можно использовать жёсткий диск для слишком длинных транзакций.
130 /// -> Максимальный размер списка трансформаций можно ограничить / задать
    ↪ константой.
131 /// 3. При подтверждении транзакции (Commit) все трансформации записываются разом
    ↪ создавая задержку.
132 ///
133 /// На жёстком диске:
134 /// Минусы:
135 /// 1. Длительный отклик, на запись каждой трансформации.
136 /// 2. Лог транзакций дополнительно наполняется отменёнными транзакциями.
137 /// -> Это может решаться упаковкой/исключением дублирующих операций.
138 /// -> Также это может решаться тем, что короткие транзакции вообще
139 /// не будут записываться в случае отката.
140 /// 3. Перед тем как выполнять отмену операций транзакции нужно дождаться пока все
    ↪ операции (трансформации)
    ↪ будут записаны в лог.
141 ///
142 ///
143 </remarks>
144 public class Transaction : DisposableBase
145 {
146     private readonly Queue<Transition> _transitions;
147     private readonly UInt64LinksTransactionsLayer _layer;
148     public bool IsCommitted { get; private set; }
149     public bool IsReverted { get; private set; }
150
151     public Transaction(UInt64LinksTransactionsLayer layer)
152     {
153         _layer = layer;
154         if (_layer._currentTransactionId != 0)
155         {
156             throw new NotSupportedException("Nested transactions not supported.");
157         }
158         IsCommitted = false;
159         IsReverted = false;
160         _transitions = new Queue<Transition>();
161         SetCurrentTransaction(layer, this);
162     }
163
164     public void Commit()
165     {
166         EnsureTransactionAllowsWriteOperations(this);
167         while (_transitions.Count > 0)
168         {
169             var transition = _transitions.Dequeue();
170             _layer._transitions.Enqueue(transition);
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,
    ↪ Transaction transaction)
189     {
190         layer._currentTransactionId = layer._lastCommittedTransactionId + 1;
191         layer._currentTransactionTransitions = transaction._transitions;
192         layer._currentTransaction = transaction;
193     }
194
195     public static void EnsureTransactionAllowsWriteOperations(Transaction transaction)
196     {

```

```

197         if (transaction.IsReverted)
198         {
199             throw new InvalidOperationException("Transation is reverted.");
200         }
201         if (transaction.IsCommitted)
202         {
203             throw new InvalidOperationException("Transation is committed.");
204         }
205     }
206
207     protected override void Dispose(bool manual, bool wasDisposed)
208     {
209         if (!wasDisposed && _layer != null && !_layer.IsDisposed)
210         {
211             if (!IsCommitted && !IsReverted)
212             {
213                 Revert();
214             }
215             _layer.ResetCurrentTransation();
216         }
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 законченную 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));

```



```

274         return createdLinkIndex;
275     }
276
277     public override ulong Update(IList<ulong> restrictions, IList<ulong> substitution)
278     {
279         var linkIndex = restrictions[Constants.IndexPart];
280         var beforeLink = new UInt64Link(Links.GetLink(linkIndex));
281         linkIndex = Links.Update(restrictions, substitution);
282         var afterLink = new UInt64Link(Links.GetLink(linkIndex));
283         CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
284             ↪ beforeLink, afterLink));
285         return linkIndex;
286     }
287
288     public override void Delete(IList<ulong> restrictions)
289     {
290         var link = restrictions[Constants.IndexPart];
291         var deletedLink = new UInt64Link(Links.GetLink(link));
292         Links.Delete(link);
293         CommitTransition(new Transition(_uniqueTimestampFactory, _currentTransactionId,
294             ↪ deletedLink, default));
295     }
296
297     [MethodImpl(MethodImplOptions.AggressiveInlining)]
298     private Queue<Transition> GetCurrentTransitions() => _currentTransactionTransitions ??
299         ↪ _transitions;
300
301     private void CommitTransition(Transition transition)
302     {
303         if (_currentTransaction != null)
304         {
305             Transaction.EnsureTransactionAllowsWriteOperations(_currentTransaction);
306         }
307         var transitions = GetCurrentTransitions();
308         transitions.Enqueue(transition);
309     }
310
311     private void RevertTransition(Transition transition)
312     {
313         if (transition.After.IsNull()) // Revert Deletion with Creation
314         {
315             Links.Create();
316         }
317         else if (transition.Before.IsNull()) // Revert Creation with Deletion
318         {
319             Links.Delete(transition.After.Index);
320         }
321         else // Revert Update
322         {
323             Links.Update(new[] { transition.After.Index, transition.Before.Source,
324                 ↪ transition.Before.Target });
325         }
326     }
327
328     private void ResetCurrentTransation()
329     {
330         _currentTransactionId = 0;
331         _currentTransactionTransitions = null;
332         _currentTransaction = null;
333     }
334
335     private void PushTransitions()
336     {
337         if (_log == null || _transitions == null)
338         {
339             return;
340         }
341         for (var i = 0; i < _transitions.Count; i++)
342         {
343             var transition = _transitions.Dequeue();
344
345             _log.Write(transition);
346             _lastCommittedTransition = transition;
347         }
348     }
349
350     private void TransitionsPusher()
351     {
352         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, _lastCommittedTransition);
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>,
9          ⇨ IConverter<char, TLink>
10     {
11         private readonly IConverter<TLink> _addressToNumberConverter;
12         private readonly TLink _unicodeSymbolMarker;
13
14         public CharToUnicodeSymbolConverter(ILinks<TLink> links, IConverter<TLink>
15             ⇨ addressToNumberConverter, TLink unicodeSymbolMarker) : base(links)
16         {
17             _addressToNumberConverter = addressToNumberConverter;
18             _unicodeSymbolMarker = unicodeSymbolMarker;
19         }
20
21         public TLink Convert(char source)
22         {
23             var unaryNumber = _addressToNumberConverter.Convert((Integer<TLink>)source);
24             return Links.GetOrCreate(unaryNumber, _unicodeSymbolMarker);
25         }
26     }
27 }

```

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

```

1  using Platform.Data.Doublets.Sequences.Indexes;
2  using Platform.Interfaces;
3  using System.Collections.Generic;

```

```

4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Data.Doublets.Unicode
8 {
9     public class StringToUnicodeSequenceConverter<TLink> : LinksOperatorBase<TLink>,
10         ↪ IConverter<string, TLink>
11     {
12         private readonly IConverter<char, TLink> _charToUnicodeSymbolConverter;
13         private readonly ISequenceIndex<TLink> _index;
14         private readonly IConverter<IList<TLink>, TLink> _listToSequenceLinkConverter;
15         private readonly TLink _unicodeSequenceMarker;
16
17         public StringToUnicodeSequenceConverter(ILinks<TLink> links, IConverter<char, TLink>
18             ↪ charToUnicodeSymbolConverter, ISequenceIndex<TLink> index, IConverter<IList<TLink>,
19             ↪ TLink> listToSequenceLinkConverter, TLink unicodeSequenceMarker) : base(links)
20         {
21             _charToUnicodeSymbolConverter = charToUnicodeSymbolConverter;
22             _index = index;
23             _listToSequenceLinkConverter = listToSequenceLinkConverter;
24             _unicodeSequenceMarker = unicodeSequenceMarker;
25         }
26
27         public TLink Convert(string source)
28         {
29             var elements = new TLink[source.Length];
30             for (int i = 0; i < source.Length; i++)
31             {
32                 elements[i] = _charToUnicodeSymbolConverter.Convert(source[i]);
33             }
34             _index.Add(elements);
35             var sequence = _listToSequenceLinkConverter.Convert(elements);
36             return Links.GetOrCreate(sequence, _unicodeSequenceMarker);
37         }
38     }
39 }

```

./Platform.Data.Doublets/Unicode/UnicodeMap.cs

```

1 using System;
2 using System.Collections.Generic;
3 using System.Globalization;
4 using System.Runtime.CompilerServices;
5 using System.Text;
6 using Platform.Data.Sequences;
7
8 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
9
10 namespace Platform.Data.Doublets.Unicode
11 {
12     public class UnicodeMap
13     {
14         public static readonly ulong FirstCharLink = 1;
15         public static readonly ulong LastCharLink = FirstCharLink + char.MaxValue;
16         public static readonly ulong MapSize = 1 + char.MaxValue;
17
18         private readonly ILinks<ulong> _links;
19         private bool _initialized;
20
21         public UnicodeMap(ILinks<ulong> links) => _links = links;
22
23         public static UnicodeMap InitNew(ILinks<ulong> links)
24         {
25             var map = new UnicodeMap(links);
26             map.Init();
27             return map;
28         }
29
30         public void Init()
31         {
32             if (_initialized)
33             {
34                 return;
35             }
36             _initialized = true;
37             var firstLink = _links.CreatePoint();
38             if (firstLink != FirstCharLink)
39             {
40                 _links.Delete(firstLink);
41             }
42             else
43             {

```

```

44     for (var i = FirstCharLink + 1; i <= LastCharLink; i++)
45     {
46         // From NIL to It (NIL -> Character) transformation meaning, (or infinite
47         ↪ amount of NIL characters before actual Character)
48         var createdLink = _links.CreatePoint();
49         _links.Update(createdLink, firstLink, createdLink);
50         if (createdLink != i)
51         {
52             throw new InvalidOperationException("Unable to initialize UTF 16
53             ↪ table.");
54         }
55     }
56 }
57
58 // 0 - null link
59 // 1 - nil character (0 character)
60 // ...
61 // 65536 (0(1) + 65535 = 65536 possible values)
62
63 [MethodImpl(MethodImplOptions.AggressiveInlining)]
64 public static ulong FromCharToLink(char character) => (ulong)character + 1;
65
66 [MethodImpl(MethodImplOptions.AggressiveInlining)]
67 public static char FromLinkToChar(ulong link) => (char)(link - 1);
68
69 [MethodImpl(MethodImplOptions.AggressiveInlining)]
70 public static bool IsCharLink(ulong link) => link <= MapSize;
71
72 public static string FromLinksToString(IList<ulong> linksList)
73 {
74     var sb = new StringBuilder();
75     for (int i = 0; i < linksList.Count; i++)
76     {
77         sb.Append(FromLinkToChar(linksList[i]));
78     }
79     return sb.ToString();
80 }
81
82 public static string FromSequenceLinkToString(ulong link, ILinks<ulong> links)
83 {
84     var sb = new StringBuilder();
85     if (links.Exists(link))
86     {
87         StopableSequenceWalker.WalkRight(link, links.GetSource, links.GetTarget,
88         ↪ x => x <= MapSize || links.GetSource(x) == x || links.GetTarget(x) == x,
89         ↪ element =>
90         {
91             sb.Append(FromLinkToChar(element));
92             return true;
93         });
94     }
95     return sb.ToString();
96 }
97
98 public static ulong[] FromCharsToLinkArray(char[] chars) => FromCharsToLinkArray(chars,
99 ↪ chars.Length);
100
101 public static ulong[] FromCharsToLinkArray(char[] chars, int count)
102 {
103     // char array to ulong array
104     var linksSequence = new ulong[count];
105     for (var i = 0; i < count; i++)
106     {
107         linksSequence[i] = FromCharToLink(chars[i]);
108     }
109     return linksSequence;
110 }
111
112 public static ulong[] FromStringToLinkArray(string sequence)
113 {
114     // char array to ulong array
115     var linksSequence = new ulong[sequence.Length];
116     for (var i = 0; i < sequence.Length; i++)
117     {
118         linksSequence[i] = FromCharToLink(sequence[i]);
119     }
120     return linksSequence;
121 }

```

```

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                     CharUnicodeInfo.GetUnicodeCategory(array[absoluteLength])))
165                 {
166                     relativeLength++;
167                     absoluteLength++;
168                 }
169             }
170             else
171             {
172                 var absoluteLength = offset + relativeLength;
173                 while (absoluteLength < array.Length && array[absoluteLength] > LastCharLink)
174                 {
175                     relativeLength++;
176                     absoluteLength++;
177                 }
178             }
179             // copy array
180             var innerSequence = new ulong[relativeLength];
181             var maxLength = offset + relativeLength;
182             for (var i = offset; i < maxLength; i++)
183             {
184                 innerSequence[i - offset] = array[i];
185             }
186             result.Add(innerSequence);
187             offset += relativeLength;
188         }
189         return result;
190     }
191 }
192
193 }

```

./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);
34             var charArray = _sequenceWalker.Walk(sequence).Select(_unicodeSymbolToCharConverter.
35                 ↳ Convert).ToArray();
36             return new string(charArray);
37         }
38     }
39 }

```

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

```

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

```

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

```
1 using System;
2 using Platform.Interfaces;
3 using Platform.Numbers;
4
5 #pragma warning disable CS1591 // Missing XML comment for publicly visible type or member
6
7 namespace Platform.Data.Doublets.Unicode
8 {
9     public class UnicodeSymbolToCharConverter<TLink> : LinksOperatorBase<TLink>,
10     ↪ IConverter<TLink, char>
11     {
12         private readonly IConverter<TLink> _numberToAddressConverter;
13         private readonly ICriterionMatcher<TLink> _unicodeSymbolCriterionMatcher;
14
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         public char Convert(TLink source)
23         {
24             if (!_unicodeSymbolCriterionMatcher.IsMatched(source))
25             {
26                 throw new ArgumentOutOfRangeException(nameof(source), source, "Specified link is
27                 ↪ not a unicode symbol.");
28             }
29             return (char)(ushort)(Integer<TLink>)_numberToAddressConverter.Convert(Links.GetSour
30             ↪ ce(source));
31         }
32     }
33 }
```

./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 GreaterOrEqualPerfomanceTest()
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     }
46 }
```

```

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 EqualsPerfomanceTest()
25         {
26             const int N = 1000000;
27
28             ulong x = 10;
29             ulong y = 500;
30
31             bool result = false;
32
33             var ts1 = Performance.Measure(() =>
34             {
35                 for (int i = 0; i < N; i++)
36                 {
37                     result = Equals1(x, y);
38                 }
39             });
40
41             var ts2 = Performance.Measure(() =>
42             {
43                 for (int i = 0; i < N; i++)
44                 {
45                     result = Equals2(x, y);
46                 }
47             });
48
49             var ts3 = Performance.Measure(() =>
50             {
51                 for (int i = 0; i < N; i++)
52                 {
53                     result = Equals3(x, y);

```



```

54     }
55 });
56
57 var equalityComparer1 = EqualityComparer<ulong>.Default;
58
59 var ts4 = Performance.Measure(() =>
60 {
61     for (int i = 0; i < N; i++)
62     {
63         result = equalityComparer1.Equals(x, y);
64     }
65 });
66
67 var equalityComparer2 = new UInt64EqualityComparer();
68
69 var ts5 = Performance.Measure(() =>
70 {
71     for (int i = 0; i < N; i++)
72     {
73         result = equalityComparer2.Equals(x, y);
74     }
75 });
76
77 Func<ulong, ulong, bool> equalityComparer3 = equalityComparer2.Equals;
78
79 var ts6 = Performance.Measure(() =>
80 {
81     for (int i = 0; i < N; i++)
82     {
83         result = equalityComparer3(x, y);
84     }
85 });
86
87 var comparer = Comparer<ulong>.Default;
88
89 var ts7 = Performance.Measure(() =>
90 {
91     for (int i = 0; i < N; i++)
92     {
93         result = comparer.Compare(x, y) == 0;
94     }
95 });
96
97 Assert.True(ts2 < ts1);
98 Assert.True(ts3 < ts2);
99 Assert.True(ts5 < ts4);
100 Assert.True(ts5 < ts6);
101
102 Console.WriteLine($"{ts1} {ts2} {ts3} {ts4} {ts5} {ts6} {ts7} {result}");
103 }
104 }
105 }

```

./Platform.Data.Doublets.Tests/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
66 }
67
68 }

```

./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         }
41     }
42 }

```

```

39     var unaryNumberToAddressConverter = new
    ↪     UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);
40     var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links, unaryOne);
41     var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
    ↪     frequencyMarker, unaryOne, unaryNumberIncrementer);
42     var frequencyPropertyOperator = new PropertyOperator<ulong>(links,
    ↪     frequencyPropertyMarker, frequencyMarker);
43     var index = new FrequencyIncrementingSequenceIndex<ulong>(links,
    ↪     frequencyPropertyOperator, frequencyIncrementer);
44     var linkToItsFrequencyNumberConverter = new
    ↪     LinkToItsFrequencyNumberConverter<ulong>(links, frequencyPropertyOperator,
    ↪     unaryNumberToAddressConverter);
45     var sequenceToItsLocalElementLevelsConverter = new
    ↪     SequenceToItsLocalElementLevelsConverter<ulong>(links,
    ↪     linkToItsFrequencyNumberConverter);
46     var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
    ↪     sequenceToItsLocalElementLevelsConverter);

47
48     var sequences = new Sequences.Sequences(links, new SequencesOptions<ulong>() {
    ↪     Walker = new LeveledSequenceWalker<ulong>(links) });
49
50     ExecuteTest(sequences, sequence, sequenceToItsLocalElementLevelsConverter,
    ↪     index, optimalVariantConverter);
51 }
52 }
53
54 [Fact]
55 public static void DictionaryBasedFrequencyStoredOptimalVariantSequenceTest()
56 {
57     using (var scope = new TempLinksTestScope(useSequences: false))
58     {
59         var links = scope.Links;
60
61         links.UseUnicode();
62
63         var sequence = UnicodeMap.FromStringToLinkArray(SequenceExample);
64
65         var linksToFrequencies = new Dictionary<ulong, ulong>();
66
67         var totalSequenceSymbolFrequencyCounter = new
    ↪     TotalSequenceSymbolFrequencyCounter<ulong>(links);
68
69         var linkFrequenciesCache = new LinkFrequenciesCache<ulong>(links,
    ↪     totalSequenceSymbolFrequencyCounter);
70
71         var index = new
    ↪     CachedFrequencyIncrementingSequenceIndex<ulong>(linkFrequenciesCache);
72         var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFrequencyNumberConverter<ulong>(linkFrequenciesCache);
73
74         var sequenceToItsLocalElementLevelsConverter = new
    ↪     SequenceToItsLocalElementLevelsConverter<ulong>(links,
    ↪     linkToItsFrequencyNumberConverter);
75         var optimalVariantConverter = new OptimalVariantConverter<ulong>(links,
    ↪     sequenceToItsLocalElementLevelsConverter);
76
77         var sequences = new Sequences.Sequences(links, new SequencesOptions<ulong>() {
    ↪     Walker = new LeveledSequenceWalker<ulong>(links) });
78
79         ExecuteTest(sequences, sequence, sequenceToItsLocalElementLevelsConverter,
    ↪     index, optimalVariantConverter);
80     }
81 }
82
83 private static void ExecuteTest(Sequences.Sequences sequences, ulong[] sequence,
    ↪     SequenceToItsLocalElementLevelsConverter<ulong>
    ↪     sequenceToItsLocalElementLevelsConverter, ISequenceIndex<ulong> index,
    ↪     OptimalVariantConverter<ulong> optimalVariantConverter)
84 {
85     index.Add(sequence);
86
87     var optimalVariant = optimalVariantConverter.Convert(sequence);
88
89     var readSequence1 = sequences.ToList(optimalVariant);
90
91     Assert.True(sequence.SequenceEqual(readSequence1));
92 }
93 }

```

```
94 }
```

```
./Platform.Data.Doublets.Tests/ReadSequenceTests.cs
```

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

```
./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     }
```

```

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
25     [Fact]
26     public static void BasicHeapMemoryTest()
27     {
28         using (var memory = new
29             ↳ HeapResizableDirectMemory(UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
30         using (var memoryAdapter = new UInt64ResizableDirectMemoryLinks(memory,
31             ↳ UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
32         {
33             memoryAdapter.TestBasicMemoryOperations();
34         }
35     }
36
37     private static void TestBasicMemoryOperations(this ILinks<ulong> memoryAdapter)
38     {
39         var link = memoryAdapter.Create();
40         memoryAdapter.Delete(link);
41     }
42
43     [Fact]
44     public static void NonexistentReferencesHeapMemoryTest()
45     {
46         using (var memory = new
47             ↳ HeapResizableDirectMemory(UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
48         using (var memoryAdapter = new UInt64ResizableDirectMemoryLinks(memory,
49             ↳ UInt64ResizableDirectMemoryLinks.DefaultLinksSizeStep))
50         {
51             memoryAdapter.TestNonexistentReferences();
52         }
53     }
54
55     private static void TestNonexistentReferences(this ILinks<ulong> memoryAdapter)
56     {
57         var link = memoryAdapter.Create();
58         memoryAdapter.Update(link, ulong.MaxValue, ulong.MaxValue);
59         var resultLink = _constants.Null;
60         memoryAdapter.Each(foundLink =>
61         {
62             resultLink = foundLink[_constants.IndexPart];
63             return _constants.Break;
64         }, _constants.Any, ulong.MaxValue, ulong.MaxValue);
65         Assert.True(resultLink == link);
66         Assert.True(memoryAdapter.Count(ulong.MaxValue) == 0);
67         memoryAdapter.Delete(link);
68     }
69 }

```

./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 }

```

./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 perfomance 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];

```

```

39     for (var i = 0; i < sequenceLength; i++)
40     {
41         sequence[i] = links.Create();
42     }
43
44     var sw1 = Stopwatch.StartNew();
45     var results1 = sequences.CreateAllVariants1(sequence); sw1.Stop();
46
47     var sw2 = Stopwatch.StartNew();
48     var results2 = sequences.CreateAllVariants2(sequence); sw2.Stop();
49
50     Assert.True(results1.Count > results2.Length);
51     Assert.True(sw1.Elapsed > sw2.Elapsed);
52
53     for (var i = 0; i < sequenceLength; i++)
54     {
55         links.Delete(sequence[i]);
56     }
57
58     Assert.True(links.Count() == 0);
59 }
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 //         // TODO: Из числа в bool значения o.UseSequenceMarker = ((value & 1) != 0)
81 //         o.
82 //
83 //         var sequences = new Sequences(links);
84 //
85 //         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++)

```



```

195         {
196             links.Delete(sequence[i]);
197         }
198     }
199 }
200
201 [Fact]
202 public static void AllPartialVariantsSearchTest()
203 {
204     const long sequenceLength = 8;
205
206     using (var scope = new TempLinksTestScope(useSequences: true))
207     {
208         var links = scope.Links;
209         var sequences = scope.Sequences;
210
211         var sequence = new ulong[sequenceLength];
212         for (var i = 0; i < sequenceLength; i++)
213         {
214             sequence[i] = links.Create();
215         }
216
217         var createResults = sequences.CreateAllVariants2(sequence);
218
219         //var createResultsStrings = createResults.Select(x => x + ": " +
220         ↪ sequences.FormatSequence(x)).ToList();
221         //Global.Trash = createResultsStrings;
222
223         var partialSequence = new ulong[sequenceLength - 2];
224
225         Array.Copy(sequence, 1, partialSequence, 0, (int)sequenceLength - 2);
226
227         var sw1 = Stopwatch.StartNew();
228         var searchResults1 =
229         ↪ sequences.GetAllPartiallyMatchingSequences0(partialSequence); sw1.Stop();
230
231         var sw2 = Stopwatch.StartNew();
232         var searchResults2 =
233         ↪ sequences.GetAllPartiallyMatchingSequences1(partialSequence); sw2.Stop();
234
235         //var sw3 = Stopwatch.StartNew();
236         //var searchResults3 =
237         ↪ sequences.GetAllPartiallyMatchingSequences2(partialSequence); sw3.Stop();
238
239         var sw4 = Stopwatch.StartNew();
240         var searchResults4 =
241         ↪ sequences.GetAllPartiallyMatchingSequences3(partialSequence); sw4.Stop();
242
243         //Global.Trash = searchResults3;
244
245         //var searchResults1Strings = searchResults1.Select(x => x + ": " +
246         ↪ sequences.FormatSequence(x)).ToList();
247         //Global.Trash = searchResults1Strings;
248
249         var intersection1 = createResults.Intersect(searchResults1).ToList();
250         Assert.True(intersection1.Count == createResults.Length);
251
252         var intersection2 = createResults.Intersect(searchResults2).ToList();
253         Assert.True(intersection2.Count == createResults.Length);
254
255         var intersection4 = createResults.Intersect(searchResults4).ToList();
256         Assert.True(intersection4.Count == createResults.Length);
257
258         for (var i = 0; i < sequenceLength; i++)
259         {
260             links.Delete(sequence[i]);
261         }
262     }
263 }
264
265 [Fact]
266 public static void BalancedPartialVariantsSearchTest()
267 {
268     const long sequenceLength = 200;
269
270     using (var scope = new TempLinksTestScope(useSequences: true))
271     {
272         var links = scope.Links;
273         var sequences = scope.Sequences;

```

```

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 [Fact(Skip = "Correct implementation is pending")]
303 public static void PatternMatchTest()
304 {
305     var zeroOrMany = Sequences.Sequences.ZeroOrMany;
306
307     using (var scope = new TempLinksTestScope(useSequences: true))
308     {
309         var links = scope.Links;
310         var sequences = scope.Sequences;
311
312         var e1 = links.Create();
313         var e2 = links.Create();
314
315         var sequence = new[]
316         {
317             e1, e2, e1, e2 // mama / papa
318         };
319
320         var balancedVariantConverter = new BalancedVariantConverter<ulong>(links);
321
322         var balancedVariant = balancedVariantConverter.Convert(sequence);
323
324         // 1: [1]
325         // 2: [2]
326         // 3: [1,2]
327         // 4: [1,2,1,2]
328
329         var doublet = links.GetSource(balancedVariant);
330
331         var matchedSequences1 = sequences.MatchPattern(e2, e1, zeroOrMany);
332
333         Assert.True(matchedSequences1.Count == 0);
334
335         var matchedSequences2 = sequences.MatchPattern(zeroOrMany, e2, e1);
336
337         Assert.True(matchedSequences2.Count == 0);
338
339         var matchedSequences3 = sequences.MatchPattern(e1, zeroOrMany, e1);
340
341         Assert.True(matchedSequences3.Count == 0);
342
343         var matchedSequences4 = sequences.MatchPattern(e1, zeroOrMany, e2);
344
345         Assert.Contains(doublet, matchedSequences4);
346         Assert.Contains(balancedVariant, matchedSequences4);
347     }
348 }

```

```

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     /// <summary>Imported from https://raw.githubusercontent.com/Konard/LinksPlatform/%
379     ↪ 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
380     ↪ %D0%B8%D0%BD%D0%B0%D0%BB%D0%BE%D1%81%D1%8C.md</summary>
381     private static readonly string _exampleText =
382         @"([english
383         ↪ 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?

![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)

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?

![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)

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

![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)

А что если мы вообразим движение? Нужно ли время? Каким самым коротким будет путь? Что будет
 ↪ если этот путь зафиксировать? Запомнить след? Как две точки становятся линией? Чертой?
 ↪ Гранью? Разделителем? Единицей?

![две белые точки, чёрная вертикальная
 ↪ линия](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/4.png "две
 ↪ белые точки, чёрная вертикальная
 ↪ линия")](https://raw.githubusercontent.com/Konard/LinksPlatform/master/doc/Intro/4.png)

398 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-anima
      ↳ tion-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
466          Assert.True(links.GetSource(links.GetSource(compressedVariant)) == sequence[0]);
467          Assert.True(links.GetTarget(links.GetSource(compressedVariant)) == sequence[1]);
468          Assert.True(links.GetSource(links.GetTarget(compressedVariant)) == sequence[2]);
469          Assert.True(links.GetTarget(links.GetTarget(compressedVariant)) == sequence[3]);
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)
      ↳ == sequence[0]);
481          Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 1)
      ↳ == sequence[1]);
482          Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 2)
      ↳ == sequence[2]);
483          Assert.True(links.GetSquareMatrixSequenceElementByIndex(compressedVariant, 4, 3)
      ↳ == sequence[3]);
484      }
485  }
486
487  [Fact]
488  public static void CompressionEfficiencyTest()

```

```

489 {
490     var strings = _exampleLoremIpsumText.Split(new[] { '\n', '\r' },
        ↳ StringSplitOptions.RemoveEmptyEntries);
491     var arrays = strings.Select(UnicodeMap.FromStringToLinkArray).ToArray();
492     var totalCharacters = arrays.Select(x => x.Length).Sum();
493
494     using (var scope1 = new TempLinksTestScope(useSequences: true))
495     using (var scope2 = new TempLinksTestScope(useSequences: true))
496     using (var scope3 = new TempLinksTestScope(useSequences: true))
497     {
498         scope1.Links.Unsync.UseUnicode();
499         scope2.Links.Unsync.UseUnicode();
500         scope3.Links.Unsync.UseUnicode();
501
502         var balancedVariantConverter1 = new
            ↳ BalancedVariantConverter<ulong>(scope1.Links.Unsync);
503         var totalSequenceSymbolFrequencyCounter = new
            ↳ TotalSequenceSymbolFrequencyCounter<ulong>(scope1.Links.Unsync);
504         var linkFrequenciesCache1 = new LinkFrequenciesCache<ulong>(scope1.Links.Unsync,
            ↳ totalSequenceSymbolFrequencyCounter);
505         var compressor1 = new CompressingConverter<ulong>(scope1.Links.Unsync,
            ↳ balancedVariantConverter1, linkFrequenciesCache1,
            ↳ doInitialFrequenciesIncrement: false);
506
507         //var compressor2 = scope2.Sequences;
508         var compressor3 = scope3.Sequences;
509
510         var constants = Default<LinksConstants<ulong>>.Instance;
511
512         var sequences = compressor3;
513         //var meaningRoot = links.CreatePoint();
514         //var unaryOne = links.CreateAndUpdate(meaningRoot, constants.Itself);
515         //var frequencyMarker = links.CreateAndUpdate(meaningRoot, constants.Itself);
516         //var frequencyPropertyMarker = links.CreateAndUpdate(meaningRoot,
            ↳ constants.Itself);
517
518         //var unaryNumberToAddressConverter = new
            ↳ UnaryNumberToAddressAddOperationConverter<ulong>(links, unaryOne);
519         //var unaryNumberIncrementer = new UnaryNumberIncrementer<ulong>(links,
            ↳ unaryOne);
520         //var frequencyIncrementer = new FrequencyIncrementer<ulong>(links,
            ↳ frequencyMarker, unaryOne, unaryNumberIncrementer);
521         //var frequencyPropertyOperator = new FrequencyPropertyOperator<ulong>(links,
            ↳ frequencyPropertyMarker, frequencyMarker);
522         //var linkFrequencyIncrementer = new LinkFrequencyIncrementer<ulong>(links,
            ↳ frequencyPropertyOperator, frequencyIncrementer);
523         //var linkToItsFrequencyNumberConverter = new
            ↳ LinkToItsFrequencyNumberConveter<ulong>(links, frequencyPropertyOperator,
            ↳ unaryNumberToAddressConverter);
524
525         var linkFrequenciesCache3 = new LinkFrequenciesCache<ulong>(scope3.Links.Unsync,
            ↳ totalSequenceSymbolFrequencyCounter);
526
527         var linkToItsFrequencyNumberConverter = new FrequenciesCacheBasedLinkToItsFreque
            ↳ ncyNumberConverter<ulong>(linkFrequenciesCache3);
528
529         var sequenceToItsLocalElementLevelsConverter = new
            ↳ SequenceToItsLocalElementLevelsConverter<ulong>(scope3.Links.Unsync,
            ↳ linkToItsFrequencyNumberConverter);
530         var optimalVariantConverter = new
            ↳ OptimalVariantConverter<ulong>(scope3.Links.Unsync,
            ↳ sequenceToItsLocalElementLevelsConverter);
531
532         var compressed1 = new ulong[arrays.Length];
533         var compressed2 = new ulong[arrays.Length];
534         var compressed3 = new ulong[arrays.Length];
535
536         var START = 0;
537         var END = arrays.Length;
538
539         //for (int i = START; i < END; i++)
540         //    linkFrequenciesCache1.IncrementFrequencies(arrays[i]);
541
542         var initialCount1 = scope2.Links.Unsync.Count();
543
544         var sw1 = Stopwatch.StartNew();
545
546         for (int i = START; i < END; i++)

```

```

547     {
548         linkFrequenciesCache1.IncrementFrequencies(arrays[i]);
549         compressed1[i] = compressor1.Convert(arrays[i]);
550     }
551
552     var elapsed1 = sw1.Elapsed;
553
554     var balancedVariantConverter2 = new
555     ↪ BalancedVariantConverter<ulong>(scope2.Links.Unsync);
556
557     var initialCount2 = scope2.Links.Unsync.Count();
558
559     var sw2 = Stopwatch.StartNew();
560
561     for (int i = START; i < END; i++)
562     {
563         compressed2[i] = balancedVariantConverter2.Convert(arrays[i]);
564     }
565
566     var elapsed2 = sw2.Elapsed;
567
568     for (int i = START; i < END; i++)
569     {
570         linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
571     }
572
573     var initialCount3 = scope3.Links.Unsync.Count();
574
575     var sw3 = Stopwatch.StartNew();
576
577     for (int i = START; i < END; i++)
578     {
579         //linkFrequenciesCache3.IncrementFrequencies(arrays[i]);
580         compressed3[i] = optimalVariantConverter.Convert(arrays[i]);
581     }
582
583     var elapsed3 = sw3.Elapsed;
584
585     Console.WriteLine($"Compressor: {elapsed1}, Balanced variant: {elapsed2},
586     ↪ Optimal variant: {elapsed3}");
587
588     // Assert.True(elapsed1 > elapsed2);
589
590     // Checks
591     for (int i = START; i < END; i++)
592     {
593         var sequence1 = compressed1[i];
594         var sequence2 = compressed2[i];
595         var sequence3 = compressed3[i];
596
597         var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
598         ↪ scope1.Links.Unsync);
599
600         var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
601         ↪ scope2.Links.Unsync);
602
603         var decompress3 = UnicodeMap.FromSequenceLinkToString(sequence3,
604         ↪ scope3.Links.Unsync);
605
606         var structure1 = scope1.Links.Unsync.FormatStructure(sequence1, link =>
607         ↪ link.IsPartialPoint());
608         var structure2 = scope2.Links.Unsync.FormatStructure(sequence2, link =>
609         ↪ link.IsPartialPoint());
610         var structure3 = scope3.Links.Unsync.FormatStructure(sequence3, link =>
611         ↪ link.IsPartialPoint());
612
613         //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
614         ↪ arrays[i].Length > 3)
615         //    Assert.False(structure1 == structure2);
616         //if (sequence3 != Constants.Null && sequence2 != Constants.Null &&
617         ↪ arrays[i].Length > 3)
618         //    Assert.False(structure3 == structure2);
619
620         Assert.True(strings[i] == decompress1 && decompress1 == decompress2);
621         Assert.True(strings[i] == decompress3 && decompress3 == decompress2);
622     }
623
624     Assert.True((int)(scope1.Links.Unsync.Count() - initialCount1) <
625     ↪ totalCharacters);

```

```

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

```



```

682
683 var START = 0;
684 var END = arrays.Length;
685
686 // Collisions proved (cannot be solved by max doublet comparison, no stable rule)
687 // Stability issue starts at 10001 or 11000
688 //for (int i = START; i < END; i++)
689 //{
690 //    var first = compressor1.Compress(arrays[i]);
691 //    var second = compressor1.Compress(arrays[i]);
692
693 //    if (first == second)
694 //        compressed1[i] = first;
695 //    else
696 //    {
697 //        // TODO: Find a solution for this case
698 //    }
699 //}
700
701 for (int i = START; i < END; i++)
702 {
703     var first = compressor1.Create(arrays[i].ConvertToRestrictionsValues());
704     var second = compressor1.Create(arrays[i].ConvertToRestrictionsValues());
705
706     if (first == second)
707     {
708         compressed1[i] = first;
709     }
710     else
711     {
712         // TODO: Find a solution for this case
713     }
714 }
715
716 var elapsed1 = sw1.Elapsed;
717
718 var balancedVariantConverter = new BalancedVariantConverter<ulong>(scope2.Links);
719
720 var sw2 = Stopwatch.StartNew();
721
722 for (int i = START; i < END; i++)
723 {
724     var first = balancedVariantConverter.Convert(arrays[i]);
725     var second = balancedVariantConverter.Convert(arrays[i]);
726
727     if (first == second)
728     {
729         compressed2[i] = first;
730     }
731 }
732
733 var elapsed2 = sw2.Elapsed;
734
735 Debug.WriteLine($"Compressor: {elapsed1}, Balanced sequence creator:
736 ↪ {elapsed2}");
737
738 Assert.True(elapsed1 > elapsed2);
739
740 // Checks
741 for (int i = START; i < END; i++)
742 {
743     var sequence1 = compressed1[i];
744     var sequence2 = compressed2[i];
745
746     if (sequence1 != _constants.Null && sequence2 != _constants.Null)
747     {
748         var decompress1 = UnicodeMap.FromSequenceLinkToString(sequence1,
749 ↪ scope1.Links);
750
751         var decompress2 = UnicodeMap.FromSequenceLinkToString(sequence2,
752 ↪ scope2.Links);
753
754         //var structure1 = scope1.Links.FormatStructure(sequence1, link =>
755 ↪ link.IsPartialPoint());
756         //var structure2 = scope2.Links.FormatStructure(sequence2, link =>
757 ↪ link.IsPartialPoint());
758
759         //if (sequence1 != Constants.Null && sequence2 != Constants.Null &&
760 ↪ arrays[i].Length > 3)

```

```

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

```

```

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         var sequences = scope.Sequences;
881
882         var sequence = new ulong[sequenceLength];
883         for (var i = 0; i < sequenceLength; i++)
884         {
885             sequence[i] = links.Create();
886         }
887
888         var createResults = sequences.CreateAllVariants2(sequence);
889
890         Global.Trash = createResults;
891
892         for (var i = 0; i < sequenceLength; i++)
893         {
894             links.Delete(sequence[i]);
895         }
896     }
897 }
898
899 [Fact]
900 public static void AllPossibleConnectionsTest()
901 {
902     const long sequenceLength = 5;
903
904     using (var scope = new TempLinksTestScope(useSequences: true))
905     {
906         var links = scope.Links;
907         var sequences = scope.Sequences;

```

```

904
905     var sequence = new ulong[sequenceLength];
906     for (var i = 0; i < sequenceLength; i++)
907     {
908         sequence[i] = links.Create();
909     }
910
911     var createResults = sequences.CreateAllVariants2(sequence);
912     var reverseResults = sequences.CreateAllVariants2(sequence.Reverse().ToArray());
913
914     for (var i = 0; i < 1; i++)
915     {
916         var sw1 = Stopwatch.StartNew();
917         var searchResults1 = sequences.GetAllConnections(sequence); sw1.Stop();
918
919         var sw2 = Stopwatch.StartNew();
920         var searchResults2 = sequences.GetAllConnections1(sequence); sw2.Stop();
921
922         var sw3 = Stopwatch.StartNew();
923         var searchResults3 = sequences.GetAllConnections2(sequence); sw3.Stop();
924
925         var sw4 = Stopwatch.StartNew();
926         var searchResults4 = sequences.GetAllConnections3(sequence); sw4.Stop();
927
928         Global.Trash = searchResults3;
929         Global.Trash = searchResults4; //-V3008
930
931         var intersection1 = createResults.Intersect(searchResults1).ToList();
932         Assert.True(intersection1.Count == createResults.Length);
933
934         var intersection2 = reverseResults.Intersect(searchResults1).ToList();
935         Assert.True(intersection2.Count == reverseResults.Length);
936
937         var intersection0 = searchResults1.Intersect(searchResults2).ToList();
938         Assert.True(intersection0.Count == searchResults2.Count);
939
940         var intersection3 = searchResults2.Intersect(searchResults3).ToList();
941         Assert.True(intersection3.Count == searchResults3.Count);
942
943         var intersection4 = searchResults3.Intersect(searchResults4).ToList();
944         Assert.True(intersection4.Count == searchResults4.Count);
945     }
946
947     for (var i = 0; i < sequenceLength; i++)
948     {
949         links.Delete(sequence[i]);
950     }
951 }
952
953 [Fact(Skip = "Correct implementation is pending")]
954 public static void CalculateAllUsagesTest()
955 {
956     const long sequenceLength = 3;
957
958     using (var scope = new TempLinksTestScope(useSequences: true))
959     {
960         var links = scope.Links;
961         var sequences = scope.Sequences;
962
963         var sequence = new ulong[sequenceLength];
964         for (var i = 0; i < sequenceLength; i++)
965         {
966             sequence[i] = links.Create();
967         }
968
969         var createResults = sequences.CreateAllVariants2(sequence);
970
971         //var reverseResults =
972         ↪ sequences.CreateAllVariants2(sequence.Reverse().ToArray());
973
974         for (var i = 0; i < 1; i++)
975         {
976             var linksTotalUsages1 = new ulong[links.Count() + 1];
977
978             sequences.CalculateAllUsages(linksTotalUsages1);
979
980             var linksTotalUsages2 = new ulong[links.Count() + 1];
981
982             sequences.CalculateAllUsages2(linksTotalUsages2);

```

```

983
984         var intersection1 = linksTotalUsages1.Intersect(linksTotalUsages2).ToList();
985         Assert.True(intersection1.Count == linksTotalUsages2.Length);
986     }
987
988     for (var i = 0; i < sequenceLength; i++)
989     {
990         links.Delete(sequence[i]);
991     }
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
           useLog = false) : this(new SequencesOptions<ulong>(), deleteFiles, useSequences,
           useLog) { }
19
20         public TempLinksTestScope(SequencesOptions<ulong> sequencesOptions, bool deleteFiles =
           true, bool useSequences = false, bool useLog = false)
21         {
22             _deleteFiles = deleteFiles;
23             TempFilename = Path.GetTempFileName();
24             TempTransactionLogFilename = Path.GetTempFileName();
25             var coreMemoryAdapter = new UInt64ResizableDirectMemoryLinks(TempFilename);
26             MemoryAdapter = useLog ? (ILinks<ulong>)new
           UInt64LinksTransactionsLayer(coreMemoryAdapter, TempTransactionLogFilename) :
           coreMemoryAdapter;
27             Links = new SynchronizedLinks<ulong>(new UInt64Links(MemoryAdapter));
28             if (useSequences)
29             {
30                 Sequences = new Sequences.Sequences(Links, sequencesOptions);
31             }
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>
150     ↪ links, int maximumOperationsPerCycle)
151 {
152     var comparer = Comparer<TLink>.Default;
153     for (var N = 1; N < maximumOperationsPerCycle; N++)
154     {
155         var random = new System.Random(N);
156         var created = 0;
157         var deleted = 0;
158         for (var i = 0; i < N; i++)
159         {
160             long linksCount = (Integer<TLink>)links.Count();
161             var createPoint = random.NextBoolean();
162             if (linksCount > 2 && createPoint)
163             {
164                 var linksAddressRange = new Range<ulong>(1, (ulong)linksCount);
165                 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         else
173         {
174             links.Create();
175             created++;
176         }
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         l2 = links.Update(l2, l2, l1, l2);
51
52         links.CreateAndUpdate(l2, itself);
53         links.CreateAndUpdate(l2, itself);
54
55         l2 = links.Update(l2, l1);
56
57         links.Delete(l2);
58
59         Global.Trash = links.Count();
60
61         links.Unsync.DisposeIfPossible(); // Close links to access log
62
63         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(scope
64             ↪ e.TempTransactionLogFilename);
65     }
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
88 [Fact]
89 public static void TransactionAutoRevertedTest()
90 {
91     // Auto Reverted (Because no commit at transaction)
92     using (var scope = new TempLinksTestScope(useLog: true))
93     {
94         var links = scope.Links;
95         var transactionsLayer = (UInt64LinksTransactionsLayer)scope.MemoryAdapter;
96         using (var transaction = transactionsLayer.BeginTransaction())
97         {
98             var l1 = links.Create();
99             var l2 = links.Create();
100
101             links.Update(l2, l2, l1, l2);
102         }
103
104         Assert.Equal(0UL, links.Count());
105
106         links.Unsync.DisposeIfPossible();
107
108         var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(s
109             ↪ cope.TempTransactionLogFilename);
110         Assert.Single(transitions);
111     }
112 }
113
114 [Fact]
115 public static void TransactionUserCodeErrorNoDataSavedTest()
116 {
117     // User Code Error (Autoreverted), no data saved
118     var itself = _constants.Itself;
119
120     TempLinksTestScope lastScope = null;
121     try
122     {
123         using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
124             ↪ useLog: true))
125         {
126             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     Global.Trash = links.Count();
148 }
149 catch
150 {
151     Assert.False(lastScope == null);
152
153     var transitions = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(l_
154     ↪ astScope.TempTransactionLogFilename);
155
156     Assert.True(transitions.Length == 1 && transitions[0].Before.IsNull() &&
157     ↪ transitions[0].After.IsNull());
158
159     lastScope.DeleteFiles();
160 }
161
162 [Fact]
163 public static void TransactionUserCodeErrorSomeDataSavedTest()
164 {
165     // User Code Error (Autoreverted), some data saved
166     var itself = _constants.Itself;
167
168     TempLinksTestScope lastScope = null;
169     try
170     {
171         ulong l1;
172         ulong l2;
173
174         using (var scope = new TempLinksTestScope(useLog: true))
175         {
176             var links = scope.Links;
177             l1 = links.CreateAndUpdate(itself, itself);
178             l2 = links.CreateAndUpdate(itself, itself);
179
180             l2 = links.Update(l2, l2, l1, l2);
181
182             links.CreateAndUpdate(l2, itself);
183             links.CreateAndUpdate(l2, itself);
184
185             links.Unsync.DisposeIfPossible();
186
187             Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(
188             ↪ scope.TempTransactionLogFilename);
189         }
190
191         using (var scope = lastScope = new TempLinksTestScope(deleteFiles: false,
192         ↪ useLog: true))
193         {
194             var links = scope.Links;
195             var transactionsLayer = (UInt64LinksTransactionsLayer)links.Unsync;
196             using (var transaction = transactionsLayer.BeginTransaction())
197             {
198                 l2 = links.Update(l2, l1);
199             }
200         }
201     }
202 }

```

```

197         links.Delete(l2);
198
199         ExceptionThrower();
200
201         transaction.Commit();
202     }
203
204     Global.Trash = links.Count();
205 }
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
247 [Fact]
248 public static void TransactionDamage()
249 {
250     var itself = _constants.Itself;
251
252     var tempDatabaseFilename = Path.GetTempFileName();
253     var tempTransactionLogFilename = Path.GetTempFileName();
254
255     // Commit
256     using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
        ↪ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
        ↪ tempTransactionLogFilename))
257     using (var links = new UInt64Links(memoryAdapter))
258     {
259         using (var transaction = memoryAdapter.BeginTransaction())
260         {
261             var l1 = links.CreateAndUpdate(itself, itself);
262             var l2 = links.CreateAndUpdate(itself, itself);
263
264             Global.Trash = links.Update(l2, l2, l1, l2);
265
266             links.Delete(l1);
267
268             transaction.Commit();
269         }

```

```

270         Global.Trash = links.Count();
271     }
272
273     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran_
274         ↪ sactionLogFilename);
275
276     // Damage database
277
278     FileHelpers.WriteFirst(tempTransactionLogFilename, new
279         ↪ UInt64LinksTransactionsLayer.Transition(new UniqueTimestampFactory(), 555));
280
281     // Try load damaged database
282     try
283     {
284         // TODO: Fix
285         using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
286             ↪ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
287             ↪ tempTransactionLogFilename))
288         using (var links = new UInt64Links(memoryAdapter))
289         {
290             Global.Trash = links.Count();
291         }
292     }
293     catch (NotSupportedException ex)
294     {
295         Assert.True(ex.Message == "Database is damaged, autorecovery is not supported
296             ↪ yet.");
297     }
298
299     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempTran_
300         ↪ sactionLogFilename);
301
302     File.Delete(tempDatabaseFilename);
303     File.Delete(tempTransactionLogFilename);
304 }
305
306 [Fact]
307 public static void Bug1Test()
308 {
309     var tempDatabaseFilename = Path.GetTempFileName();
310     var tempTransactionLogFilename = Path.GetTempFileName();
311
312     var itself = _constants.Itself;
313
314     // User Code Error (Autoreverted), some data saved
315     try
316     {
317         ulong l1;
318         ulong l2;
319
320         using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
321             ↪ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
322             ↪ tempTransactionLogFilename))
323         using (var links = new UInt64Links(memoryAdapter))
324         {
325             l1 = links.CreateAndUpdate(itself, itself);
326             l2 = links.CreateAndUpdate(itself, itself);
327
328             l2 = links.Update(l2, l2, l1, l2);
329
330             links.CreateAndUpdate(l2, itself);
331             links.CreateAndUpdate(l2, itself);
332         }
333
334         Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(temp_
335             ↪ TransactionLogFilename);
336
337         using (var memoryAdapter = new UInt64LinksTransactionsLayer(new
338             ↪ UInt64ResizableDirectMemoryLinks(tempDatabaseFilename),
339             ↪ tempTransactionLogFilename))
340         using (var links = new UInt64Links(memoryAdapter))
341         {
342             using (var transaction = memoryAdapter.BeginTransaction())
343             {
344                 l2 = links.Update(l2, l1);
345
346                 links.Delete(l2);

```

```

337         ExceptionThrower();
338     }
339     transaction.Commit();
340 }
341
342     Global.Trash = links.Count();
343 }
344 }
345 }
346 catch
347 {
348     Global.Trash = FileHelpers.ReadAll<UInt64LinksTransactionsLayer.Transition>(tempDatabaseFilename,
349     ↪ TransactionLogFilename);
350 }
351
352 File.Delete(tempDatabaseFilename);
353 File.Delete(tempTransactionLogFilename);
354 }
355
356 private static void ExceptionThrower() => throw new InvalidOperationException();
357
358 [Fact]
359 public static void PathsTest()
360 {
361     var source = _constants.SourcePart;
362     var target = _constants.TargetPart;
363
364     using (var scope = new TempLinksTestScope())
365     {
366         var links = scope.Links;
367         var l1 = links.CreatePoint();
368         var l2 = links.CreatePoint();
369
370         var r1 = links.GetByKeys(l1, source, target, source);
371         var r2 = links.CheckPathExistence(l2, l2, l2, l2);
372     }
373 }
374
375 [Fact]
376 public static void RecursiveStringFormattingTest()
377 {
378     using (var scope = new TempLinksTestScope(useSequences: true))
379     {
380         var links = scope.Links;
381         var sequences = scope.Sequences; // TODO: Auto use sequences on Sequences getter.
382
383         var a = links.CreatePoint();
384         var b = links.CreatePoint();
385         var c = links.CreatePoint();
386
387         var ab = links.CreateAndUpdate(a, b);
388         var cb = links.CreateAndUpdate(c, b);
389         var ac = links.CreateAndUpdate(a, c);
390
391         a = links.Update(a, c, b);
392         b = links.Update(b, a, c);
393         c = links.Update(c, a, b);
394
395         Debug.WriteLine(links.FormatStructure(ab, link => link.IsFullPoint(), true));
396         Debug.WriteLine(links.FormatStructure(cb, link => link.IsFullPoint(), true));
397         Debug.WriteLine(links.FormatStructure(ac, link => link.IsFullPoint(), true));
398
399         Assert.True(links.FormatStructure(cb, link => link.IsFullPoint(), true) ==
400         ↪ "(5:(4:5 (6:5 4)) 6)");
401         Assert.True(links.FormatStructure(ac, link => link.IsFullPoint(), true) ==
402         ↪ "(6:(5:(4:5 6) 6) 4)");
403         Assert.True(links.FormatStructure(ab, link => link.IsFullPoint(), true) ==
404         ↪ "(4:(5:4 (6:5 4)) 6)");
405
406         // TODO: Think how to build balanced syntax tree while formatting structure (eg.
407         ↪ "(4:(5:4 6) (6:5 4))" instead of "(4:(5:4 (6:5 4)) 6)"
408
409         Assert.True(sequences.SafeFormatSequence(cb, DefaultFormatter, false) ==
410         ↪ "{{5}{5}{4}{6}}");
411         Assert.True(sequences.SafeFormatSequence(ac, DefaultFormatter, false) ==
412         ↪ "{{5}{6}{6}{4}}");
413         Assert.True(sequences.SafeFormatSequence(ab, DefaultFormatter, false) ==
414         ↪ "{{4}{5}{4}{6}}");
415     }
416 }

```

```

408     }
409
410     private static void DefaultFormatter(StringBuilder sb, ulong link)
411     {
412         sb.Append(link.ToString());
413     }
414
415     #endregion
416
417     #region Performance
418
419     /*
420     public static void RunAllPerformanceTests()
421     {
422         try
423         {
424             links.TestLinksInSteps();
425         }
426         catch (Exception ex)
427         {
428             ex.WriteToConsole();
429         }
430
431         return;
432
433         try
434         {
435             //ThreadPool.SetMaxThreads(2, 2);
436
437             // Запускаем все тесты дважды, чтобы первоначальная инициализация не повлияла на
438             // Также это дополнительно помогает в отладке
439             // Увеличивает вероятность попадания информации в кэши
440             for (var i = 0; i < 10; i++)
441             {
442                 //0 - 10 ГБ
443                 //Каждые 100 МБ срез цифр
444
445                 //links.TestGetSourceFunction();
446                 //links.TestGetSourceFunctionInParallel();
447                 //links.TestGetTargetFunction();
448                 //links.TestGetTargetFunctionInParallel();
449                 links.Create64BillionLinks();
450
451                 links.TestRandomSearchFixed();
452                 //links.Create64BillionLinksInParallel();
453                 links.TestEachFunction();
454                 //links.TestForeach();
455                 //links.TestParallelForeach();
456             }
457
458             links.TestDeletionOfAllLinks();
459
460         }
461         catch (Exception ex)
462         {
463             ex.WriteToConsole();
464         }
465     }*/
466
467     /*
468     public static void TestLinksInSteps()
469     {
470         const long gibibyte = 1024 * 1024 * 1024;
471         const long mebibyte = 1024 * 1024;
472
473         var totalLinksToCreate = gibibyte /
474         Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
475         var linksStep = 102 * mebibyte /
476         Platform.Links.Data.Core.Doublets.Links.LinkSizeInBytes;
477
478         var creationMeasurements = new List<TimeSpan>();
479         var searchMeasurements = new List<TimeSpan>();
480         var deletionMeasurements = new List<TimeSpan>();
481
482         GetBaseRandomLoopOverhead(linksStep);
483         GetBaseRandomLoopOverhead(linksStep);
484
485         var stepLoopOverhead = GetBaseRandomLoopOverhead(linksStep);

```

```

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.",
↪ Iterations);
556
557         ulong counter = 0;
558

```

```

559     //var firstLink = links.First();
560     // Создаём одну связь, из которой будет производить считывание
561     var firstLink = links.Create();
562
563     var sw = Stopwatch.StartNew();
564
565     // Тестируем саму функцию
566     for (ulong i = 0; i < Iterations; i++)
567     {
568         counter += links.GetSource(firstLink);
569     }
570
571     var elapsedTime = sw.Elapsed;
572
573     var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
574
575     // Удаляем связь, из которой производилось считывание
576     links.Delete(firstLink);
577
578     ConsoleHelpers.Debug(
579         "{0} Iterations of GetSource function done in {1} ({2} Iterations per
        ↪ second), counter result: {3}",
        Iterations, elapsedTime, (long)iterationsPerSecond, counter);
581     }
582 }
583
584 [Fact(Skip = "performance test")]
585 public static void GetSourceInParallel()
586 {
587     using (var scope = new TempLinksTestScope())
588     {
589         var links = scope.Links;
590         ConsoleHelpers.Debug("Testing GetSource function with {0} Iterations in
        ↪ parallel.", Iterations);
591
592         long counter = 0;
593
594         //var firstLink = links.First();
595         var firstLink = links.Create();
596
597         var sw = Stopwatch.StartNew();
598
599         // Тестируем саму функцию
600         Parallel.For(0, Iterations, x =>
601         {
602             Interlocked.Add(ref counter, (long)links.GetSource(firstLink));
603             //Interlocked.Increment(ref counter);
604         });
605
606         var elapsedTime = sw.Elapsed;
607
608         var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
609
610         links.Delete(firstLink);
611
612         ConsoleHelpers.Debug(
613             "{0} Iterations of GetSource function done in {1} ({2} Iterations per
            ↪ second), counter result: {3}",
            Iterations, elapsedTime, (long)iterationsPerSecond, counter);
614     }
615 }
616
617 [Fact(Skip = "performance test")]
618 public static void TestGetTarget()
619 {
620     using (var scope = new TempLinksTestScope())
621     {
622         var links = scope.Links;
623         ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations.",
        ↪ Iterations);
624
625         ulong counter = 0;
626
627         //var firstLink = links.First();
628         var firstLink = links.Create();
629
630         var sw = Stopwatch.StartNew();
631
632         for (ulong i = 0; i < Iterations; i++)

```



```

634     {
635         counter += links.GetTarget(firstLink);
636     }
637
638     var elapsedTime = sw.Elapsed;
639
640     var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
641
642     links.Delete(firstLink);
643
644     ConsoleHelpers.Debug(
645         "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
        ↳ second), counter result: {3}",
        Iterations, elapsedTime, (long)iterationsPerSecond, counter);
646     }
647 }
648
649 [Fact(Skip = "performance test")]
650 public static void TestGetTargetInParallel()
651 {
652     using (var scope = new TempLinksTestScope())
653     {
654         var links = scope.Links;
655         ConsoleHelpers.Debug("Testing GetTarget function with {0} Iterations in
        ↳ parallel.", Iterations);
656
657         long counter = 0;
658
659         //var firstLink = links.First();
660         var firstLink = links.Create();
661
662         var sw = Stopwatch.StartNew();
663
664         Parallel.For(0, Iterations, x =>
665         {
666             Interlocked.Add(ref counter, (long)links.GetTarget(firstLink));
667             //Interlocked.Increment(ref counter);
668         });
669
670         var elapsedTime = sw.Elapsed;
671
672         var iterationsPerSecond = Iterations / elapsedTime.TotalSeconds;
673
674         links.Delete(firstLink);
675
676         ConsoleHelpers.Debug(
677             "{0} Iterations of GetTarget function done in {1} ({2} Iterations per
        ↳ second), counter result: {3}",
        Iterations, elapsedTime, (long)iterationsPerSecond, counter);
678     }
679 }
680
681 // TODO: Заполнить базу данных перед тестом
682 /*
683 [Fact]
684 public void TestRandomSearchFixed()
685 {
686     var tempFilename = Path.GetTempFileName();
687
688     using (var links = new Platform.Links.Data.Core.Doublets.Links(tempFilename,
689 ↳ DefaultLinksSizeStep))
690     {
691         long iterations = 64 * 1024 * 1024 /
692 ↳ 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 =
704 ↳ RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);
705             var target =
706 ↳ RandomHelpers.DefaultFactory.NextUInt64(LinksConstants.MinPossibleIndex, maxLink);

```

```

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 [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
754 public static void TestEach()
755 {
756     using (var scope = new TempLinksTestScope())
757     {
758         var links = scope.Links;
759
760         var counter = new Counter<IList<ulong>, ulong>(links.Constants.Continue);
761
762         ConsoleHelpers.Debug("Testing Each function.");
763
764         var sw = Stopwatch.StartNew();
765
766         links.Each(counter.IncrementAndReturnTrue);
767
768         var elapsedTime = sw.Elapsed;
769
770         var linksPerSecond = counter.Count / elapsedTime.TotalSeconds;
771
772         ConsoleHelpers.Debug("{0} Iterations of Each's handler function done in {1} ({2}
↵ links per second)",
↵ counter, elapsedTime, (long)linksPerSecond);
773     }
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.Doublets.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.Doublets.Links(tempFilename,
↵ DefaultLinksSizeStep))
815         {
816
817             long counter = 0;
818
819             ConsoleHelpers.Debug("Testing parallel foreach through links.");
820
821             var sw = Stopwatch.StartNew();
822
823             //Parallel.ForEach((IEnumerable<ulong>)links, x =>
824             //{
825                 //    Interlocked.Increment(ref counter);
826             //});
827
828             var elapsedTime = sw.Elapsed;
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)",
865     ↪ linksCreated, elapsedTime,
866     (long)linksPerSecond);
867
868 }
869
870 [Fact(Skip = "performance test")]
871 public static void Create64BillionLinksInParallel()
872 {
873     using (var scope = new TempLinksTestScope())
874     {
875         var links = scope.Links;
876         var linksBeforeTest = links.Count();
877
878         var sw = Stopwatch.StartNew();
879
880         long linksToCreate = 64 * 1024 * 1024 /
881             ↪ UInt64ResizableDirectMemoryLinks.LinkSizeInBytes;
882
883         ConsoleHelpers.Debug("Creating {0} links in parallel.", linksToCreate);
884
885         Parallel.For(0, linksToCreate, x => links.Create());
886
887         var elapsedTime = sw.Elapsed;
888
889         var linksCreated = links.Count() - linksBeforeTest;
890         var linksPerSecond = linksCreated / elapsedTime.TotalSeconds;
891
892         ConsoleHelpers.Debug("{0} links created in {1} ({2} links per second)",
893             ↪ linksCreated, elapsedTime,
894             (long)linksPerSecond);
895     }
896 }
897
898 [Fact(Skip = "useless: 0(0), was dependent on creation tests")]
899 public static void TestDeletionOfAllLinks()
900 {
901     using (var scope = new TempLinksTestScope())
902     {
903         var links = scope.Links;
904         var linksBeforeTest = links.Count();
905
906         ConsoleHelpers.Debug("Deleting all links");
907
908         var elapsedTime = Performance.Measure(links.DeleteAll);
909
910         var linksDeleted = linksBeforeTest - links.Count();
911         var linksPerSecond = linksDeleted / elapsedTime.TotalSeconds;
912
913         ConsoleHelpers.Debug("{0} links deleted in {1} ({2} links per second)",
914             ↪ linksDeleted, elapsedTime,
915             (long)linksPerSecond);
916     }
917 }
918
919 #endregion
920
921 }
922
923 }

```

./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 }

```

./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         {

```

```

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

```

```

92     var unicodeSequenceLink =
93         ↳ stringToUnicodeSequenceConverter.Convert(originalString);
94
95     var unicodeSymbolCriterionMatcher = new
96         ↳ UnicodeSymbolCriterionMatcher<ulong>(links, unicodeSymbolMarker);
97     var unicodeSymbolToCharConverter = new
98         ↳ UnicodeSymbolToCharConverter<ulong>(links, unaryNumberToAddressConverter,
99         ↳ unicodeSymbolCriterionMatcher);
100
101     var unicodeSequenceCriterionMatcher = new
102         ↳ UnicodeSequenceCriterionMatcher<ulong>(links, unicodeSequenceMarker);
103
104     var sequenceWalker = new LeveledSequenceWalker<ulong>(links,
105         ↳ unicodeSymbolCriterionMatcher.IsMatched);
106
107     var unicodeSequenceToStringConverter = new
108         ↳ UnicodeSequenceToStringConverter<ulong>(links,
109         ↳ unicodeSequenceCriterionMatcher, sequenceWalker,
110         ↳ unicodeSymbolToCharConverter);
111
112     var resultingString =
113         ↳ unicodeSequenceToStringConverter.Convert(unicodeSequenceLink);
114
115     Assert.Equal(originalString, resultingString);
116 }
117 }
118 }
119 }
120 }

```

Index

./Platform.Data.Doublets.Tests/ComparisonTests.cs, 143
./Platform.Data.Doublets.Tests/EqualityTests.cs, 144
./Platform.Data.Doublets.Tests/GenericLinksTests.cs, 145
./Platform.Data.Doublets.Tests/OptimalVariantSequenceTests.cs, 146
./Platform.Data.Doublets.Tests/ReadSequenceTests.cs, 148
./Platform.Data.Doublets.Tests/ResizableDirectMemoryLinksTests.cs, 148
./Platform.Data.Doublets.Tests/ScopeTests.cs, 149
./Platform.Data.Doublets.Tests/SequencesTests.cs, 150
./Platform.Data.Doublets.Tests/TempLinksTestScope.cs, 165
./Platform.Data.Doublets.Tests/TestExtensions.cs, 165
./Platform.Data.Doublets.Tests/UInt64LinksTests.cs, 168
./Platform.Data.Doublets.Tests/UnaryNumberConvertersTests.cs, 180
./Platform.Data.Doublets.Tests/UnicodeConvertersTests.cs, 181
./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, 43
./Platform.Data.Doublets/ResizableDirectMemory/ResizableDirectMemoryLinks.TreeMethods.cs, 43
./Platform.Data.Doublets/ResizableDirectMemory/ResizableDirectMemoryLinks.cs, 34
./Platform.Data.Doublets/ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.ListMethods.cs, 56
./Platform.Data.Doublets/ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.TreeMethods.cs, 57
./Platform.Data.Doublets/ResizableDirectMemory/UInt64ResizableDirectMemoryLinks.cs, 50
./Platform.Data.Doublets/Sequences/ArrayExtensions.cs, 63
./Platform.Data.Doublets/Sequences/Converters/BalancedVariantConverter.cs, 64
./Platform.Data.Doublets/Sequences/Converters/CompressingConverter.cs, 64
./Platform.Data.Doublets/Sequences/Converters/LinksListToSequenceConverterBase.cs, 67
./Platform.Data.Doublets/Sequences/Converters/OptimalVariantConverter.cs, 68
./Platform.Data.Doublets/Sequences/Converters/SequenceToItsLocalElementLevelsConverter.cs, 69
./Platform.Data.Doublets/Sequences/CriteriaMatchers/DefaultSequenceElementCriterionMatcher.cs, 70
./Platform.Data.Doublets/Sequences/CriteriaMatchers/MarkedSequenceCriterionMatcher.cs, 70
./Platform.Data.Doublets/Sequences/DefaultSequenceAppender.cs, 70
./Platform.Data.Doublets/Sequences/DuplicateSegmentsCounter.cs, 71
./Platform.Data.Doublets/Sequences/DuplicateSegmentsProvider.cs, 71
./Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkFrequenciesCache.cs, 73
./Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkFrequency.cs, 75

./Platform.Data.Doublets/Sequences/Frequencies/Cache/LinkToItsFrequencyValueConverter.cs, 75
./Platform.Data.Doublets/Sequences/Frequencies/Counters/MarkedSequenceSymbolFrequencyOneOffCounter.cs, 76
./Platform.Data.Doublets/Sequences/Frequencies/Counters/SequenceSymbolFrequencyOneOffCounter.cs, 76
./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyCounter.cs, 77
./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalMarkedSequenceSymbolFrequencyOneOffCounter.cs, 77
./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyCounter.cs, 77
./Platform.Data.Doublets/Sequences/Frequencies/Counters/TotalSequenceSymbolFrequencyOneOffCounter.cs, 78
./Platform.Data.Doublets/Sequences/HeightProviders/CachedSequenceHeightProvider.cs, 79
./Platform.Data.Doublets/Sequences/HeightProviders/DefaultSequenceRightHeightProvider.cs, 79
./Platform.Data.Doublets/Sequences/HeightProviders/ISequenceHeightProvider.cs, 80
./Platform.Data.Doublets/Sequences/IListExtensions.cs, 80
./Platform.Data.Doublets/Sequences/Indexes/CachedFrequencyIncrementingSequenceIndex.cs, 80
./Platform.Data.Doublets/Sequences/Indexes/FrequencyIncrementingSequenceIndex.cs, 81
./Platform.Data.Doublets/Sequences/Indexes/ISequenceIndex.cs, 82
./Platform.Data.Doublets/Sequences/Indexes/SequenceIndex.cs, 82
./Platform.Data.Doublets/Sequences/Indexes/SynchronizedSequenceIndex.cs, 82
./Platform.Data.Doublets/Sequences/ListFiller.cs, 83
./Platform.Data.Doublets/Sequences/Sequences.Experiments.cs, 94
./Platform.Data.Doublets/Sequences/Sequences.cs, 84
./Platform.Data.Doublets/Sequences/SequencesExtensions.cs, 120
./Platform.Data.Doublets/Sequences/SequencesOptions.cs, 121
./Platform.Data.Doublets/Sequences/SetFiller.cs, 122
./Platform.Data.Doublets/Sequences/Walkers/ISequenceWalker.cs, 123
./Platform.Data.Doublets/Sequences/Walkers/LeftSequenceWalker.cs, 123
./Platform.Data.Doublets/Sequences/Walkers/LeveledSequenceWalker.cs, 124
./Platform.Data.Doublets/Sequences/Walkers/RightSequenceWalker.cs, 125
./Platform.Data.Doublets/Sequences/Walkers/SequenceWalkerBase.cs, 126
./Platform.Data.Doublets/Stacks/Stack.cs, 127
./Platform.Data.Doublets/Stacks/StackExtensions.cs, 127
./Platform.Data.Doublets/SynchronizedLinks.cs, 127
./Platform.Data.Doublets/UInt64Link.cs, 128
./Platform.Data.Doublets/UInt64LinkExtensions.cs, 131
./Platform.Data.Doublets/UInt64LinksExtensions.cs, 131
./Platform.Data.Doublets/UInt64LinksTransactionsLayer.cs, 133
./Platform.Data.Doublets/Unicode/CharToUnicodeSymbolConverter.cs, 138
./Platform.Data.Doublets/Unicode/StringToUnicodeSequenceConverter.cs, 138
./Platform.Data.Doublets/Unicode/UnicodeMap.cs, 139
./Platform.Data.Doublets/Unicode/UnicodeSequenceCriterionMatcher.cs, 141
./Platform.Data.Doublets/Unicode/UnicodeSequenceToStringConverter.cs, 142
./Platform.Data.Doublets/Unicode/UnicodeSymbolCriterionMatcher.cs, 142
./Platform.Data.Doublets/Unicode/UnicodeSymbolToCharConverter.cs, 142